arXiv:2311.18295v1 [cs.DS] 30 Nov 2023

Almost-Linear Time Algorithms for Incremental Graphs: Cycle Detection, SCCs, s-t Shortest Path, and Minimum-Cost Flow

Li Chen*
Carnegie Mellon University
lichenntu@gmail.com

Rasmus Kyng†
ETH Zurich
kyng@inf.ethz.ch

Yang P. Liu‡
Institute for Advanced Study
yangpliu@ias.edu

Simon Meierhans†
ETH Zurich
mesimon@inf.ethz.ch

Maximilian Probst Gutenberg†
ETH Zurich
maximilian.probst@inf.ethz.ch

Abstract

We give the first almost-linear time algorithms for several problems in incremental graphs including cycle detection, strongly connected component maintenance, s-t shortest path, maximum flow, and minimum-cost flow. To solve these problems, we give a deterministic data structure that returns a $m^{o(1)}$-approximate minimum-ratio cycle in fully dynamic graphs in amortized $m^{o(1)}$ time per update. Combining this with the interior point method framework of Brand-Liu-Sidford (STOC 2023) gives the first almost-linear time algorithm for deciding the first update in an incremental graph after which the cost of the minimum-cost flow attains value at most some given threshold F. By rather direct reductions to minimum-cost flow, we are then able to solve the problems in incremental graphs mentioned above.

Our new data structure also leads to a modular and deterministic almost-linear time algorithm for minimum-cost flow by removing the need for complicated modeling of a restricted adversary, in contrast to recent randomized and deterministic algorithms for minimum-cost flow in Chen-Kyng-Liu-Peng-Probst Gutenberg-Sachdeva (FOCS 2022) & Brand-Chen-Kyng-Liu-Peng-Probst Gutenberg-Sachdeva-Sidford (FOCS 2023).

At a high level, our algorithm dynamizes the ℓ_1 oblivious routing of Rozhoň-Grunau-Haeupler-Zuzic-Li (STOC 2022), and develops a method to extract an approximate minimum ratio cycle from the structure of the oblivious routing. To maintain the oblivious routing, we use tools from concurrent work of Kyng-Meierhans-Probst Gutenberg which designed vertex sparsifiers for shortest paths, in order to maintain a sparse neighborhood cover in fully dynamic graphs.

To find a cycle, we first show that an approximate minimum ratio cycle can be represented as a fundamental cycle on a small set of trees resulting from the oblivious routing. Then, we find a cycle whose quality is comparable to the best tree cycle. This final cycle query step involves vertex and edge sparsification procedures reminiscent of the techniques introduced in Chen-Kyng-Liu-Peng-Probst Gutenberg-Sachdeva (FOCS 2022), but crucially requires a more powerful dynamic spanner, which can handle far more edge insertions than prior work. We build such a spanner via a construction that hearkens back to the classic greedy spanner algorithm of Althöfer-Das-Dobkin-Joseph-Soares (Discrete & Computational Geometry 1993).

*Li Chen was supported by NSF Grant CCF-2330255.

†The research leading to these results has received funding from grant no. 200021 204787 of the Swiss National Science Foundation.

‡This material is based upon work supported by the National Science Foundation under Grant No. DMS-1926686.

Contents

1 Introduction

The goal of *dynamic graph algorithms*[1] is to solve problems on a changing input graph, while minimizing the time required to compute and update solutions as the graph changes. In this paper, we focus on the setting of *incremental graphs*, where the input is a directed graph that undergoes edge insertions over time. We essentially settle the complexity of several fundamental and long-studied problems on incremental directed graphs, by giving algorithms that process up to m edge insertions while using $m^{1+o(1)}$ total update time. In this setting, we obtain almost-linear total update time algorithms for cycle detection [MNR96, KB06, LC07, AFM08, AF10, BFG09, HKM+12, BFGT16, CFKR13, BC18, BK20], maintaining strongly-connected components [BFGT16, BDP21], and approximate or thresholded versions of s-t shortest paths [Ber13, PVW20, CZ21, KMP22, DGWN22], weighted bipartite matching [Gup14, BKS23, BK23], and maximum and min-cost flow [GH22, BLS23, BCK+23b]. All our results are essentially achieved by reduction to one central problem, known as *incremental thresholded minimum-cost flow* [BLS23], defined as follows.

Definition 1.1 (Incremental thresholded minimum-cost flow)**.** *The thresholded min-cost flow problem is defined on a directed graph $G = (V, E)$ with capacities $\boldsymbol{u} \in \mathbb{R}^E$ and costs $\boldsymbol{c} \in \mathbb{R}^E$, undergoing edge insertions, along with vertex demands $\boldsymbol{d} \in \mathbb{R}^V$ and a threshold F. A dynamic algorithm solves the problem if, after every update, the algorithm outputs whether the graph can support a feasible flow $\boldsymbol{f} \in \mathbb{R}^E$ routing demand $\boldsymbol{d}$ with cost $\boldsymbol{c}^\top \boldsymbol{f}$ at most F, or answers that no such flow exists yet.*

Note that the optimal cost is non-increasing, as we can give 0 flow to the newly inserted edge, so there is a unique first time when such a flow is feasible, and it stays feasible afterwards. Our main result is an almost linear time algorithm for incremental thresholded min-cost flow.

Informal Theorem 1.2. *There exists a deterministic algorithm that solves the thresholded min-cost flow problem with m insertions in $m^{1+o(1)}$ total time, provided capacities and costs are polynomially bounded in m.*

From this result, we derive our other results on incremental graphs, except in the case of strongly-connected component maintenance, which requires a slightly different framework, but uses the same techniques and data structures. All our results are deterministic.

Our result builds on and strengthens many recent results on minimum-cost flow. Almost-linear time algorithms for minimum-cost flow were, after at least nine decades of research, obtained in [CKL+22]. A key element of their algorithm was the introduction of the dynamic min-ratio cycle problem, which is defined as follows.

Informal Definition 1.3 (α-Approximate dynamic min-ratio cycle problem)**.** *The dynamic min-ratio cycle problem is defined on a directed graph $G = (V, E, \boldsymbol{l}, \boldsymbol{g})$ with (undirected) lengths $\boldsymbol{l} \in \mathbb{R}^E_{\geq 0}$ and gradient $\boldsymbol{g} \in \mathbb{R}^E$. At each time step, the gradient and length of a single edge may be updated.*

A dynamic algorithm solves the problem if, after the i-th update, it can identify a cycle $\boldsymbol{c}_i \in \mathbb{R}^E$, such that $\boldsymbol{B}^T \boldsymbol{c}_i = \boldsymbol{0}$, and

$$\frac{\langle \boldsymbol{g}, \boldsymbol{c}_i \rangle}{\|\boldsymbol{L}\boldsymbol{c}_i\|_1} \leq \frac{1}{\alpha} \cdot \min_{\boldsymbol{B}^\top \boldsymbol{\Delta} = 0} \frac{\langle \boldsymbol{g}, \boldsymbol{\Delta} \rangle}{\|\boldsymbol{L}\boldsymbol{\Delta}\|_1}.$$

[1] We use the terms *dynamic algorithm* and *data structure* interchangeably.

[CKL+22] introduced an ℓ_1-interior point method (IPM), which they showed reduces min-cost flow on a graph with m edges to $m^{1+o(1)}$ update steps of solving the dynamic min-ratio problem defined above with approximation quality $\alpha = m^{o(1)}$. However, [CKL+22] did *not* manage to solve this problem against a general adversary. Instead, they developed a randomized data structure that solves the problem against an *oblivious* adversary,[2] and then they showed the ℓ_1-IPM creates a sequence of problems that behaves almost like an oblivious adversary, and managed to adapt their data structure to this particular sequence. This left an important open question: can approximate dynamic min-ratio cycle be solved against any adversary? Solving this question with a deterministic data structure would immediately yield a deterministic algorithm for min-cost flow.

[BLS23] raised the stakes of this question dramatically: they showed, by slightly modifying the [CKL+22] ℓ_1-IPM, that solving approximate dynamic min-ratio cycle across $m^{1+o(1)}$ update steps directly implies solving the incremental thresholded min-cost flow problem, and hence a host of important incremental graph problems. In this setting, techniques from [CKL+22] for solving dynamic min-ratio cycle against a restricted adversary fail. Instead, [BLS23] developed a randomized algorithm for solving this problem against a general adversary with total update time $m^{1+o(1)}\sqrt{n}$.

[BCK+23a] gave a deterministic algorithm for min-cost flow with almost-linear running time, but, remarkably, they still did not solve the approximate dynamic min-ratio cycle problem. Instead, they tailored a deterministic data structure to the specific update sequence generated by the ℓ_1-IPM, similar to the randomized approach in [CKL+22]. Very recently, [BCK+23b] gave the first algorithm to solve $(1+\epsilon)$-approximate maximum flow in *undirected graphs* in time $m^{1+o(1)}\epsilon^{-3}$. However, instead of attacking the dynamic min-ratio cycle problem head on, they showed that for undirected, approximate maximum flow, it can be circumvented: Rather than solving the problem with an ℓ_1-IPM, they could rely on a multiplicative weight update method [AKPS19], reminiscent of first-order methods for approximate undirected maximum flow [She13, KLOS14]. This method leads to a monotone version of dynamic min-ratio cycle (where, for long periods, lengths are only increasing and the gradient is fixed), with a very well-behaved update sequence, and in this more tractable setting, the data structure of [CKL+22] works, even for the incremental problem.

In this paper, we finally solve the approximate dynamic min-ratio cycle problem with $m^{1+o(1)}$ total update time deterministically, across $m^{1+o(1)}$ update steps. Thus, by the ℓ_1-IPM of [CKL+22], we immediately get a much more modular, and conceptually simpler, deterministic almost-linear time algorithm for min-cost flow than [CKL+22, BCK+23a]. And, more importantly, we solve thresholded min-cost flow, and hence the classic incremental graph problems of cycle detection, strongly-connected components, s-t shortest paths, and maximum and min-cost flow.

1.1 Paper Organization

The remainder of the paper is organized as follows. In Section 1.2, we give applications of our deterministic min-ratio cycle data structure to incremental min-cost flow and its corollaries. We give an overview of our algorithm in Section 1.3.

We give the preliminaries in Section 2. In Section 3, we state our data structures which use an ℓ_1-oblivious routing to reduce min-ratio cycle to min-ratio tree cycle, and a data structure to reduce min-ratio tree cycle to min-ratio cycle on a smaller graph. These are combined to give our overall min-ratio cycle data structure. Then we discuss how to use an interior point method to leverage this data structure to solve incremental min-cost flow. In Section 4 we build the ℓ_1 oblivious routing,

[2] An oblivious adversary is an adversary that does not depend on random choices made by the data structure

and in Section 5 we give the dynamic algorithm that reduces min-ratio tree cycle to min-ratio cycle on a smaller graph. In Section 6 we present our fully dynamic sparse neighborhood cover used in Section 4, and in Section 7 we present our new spanner algorithm used in Section 5. Finally, in Section 8 we recall the IPM framework and present the adaption necessary for incremental strongly connected components.

1.2 Applications

Application #1: More modular static and deterministic min-cost flow. [CKL+22] developed an ℓ_1-interior point method that reduced minimum-cost flow to approximate dynamic min-ratio cycle (Informal Definition 1.3) with $m^{1+o(1)}$ updates. This reduction is deterministic. But, [CKL+22] gave a data structure for min-ratio cycle that in fact only succeeds against *oblivious adversaries*. To adapt their data structure to the problem, they heavily used properties of the update sequence produced by the IPM. One of these modifications included an (arguably unintuitive) rebuilding game where layers of the data structure were rebuilt when it failed. Later, [BCK+23a] gave a deterministic algorithm for min-cost flow. However, the data structure they designed was based on that of [CKL+22], and still critically used properties of the IPM update sequence and a rebuilding game. By using our deterministic min-ratio cycle data structure, we achieve a deterministic min-cost flow algorithm that is more modular, in that the data structure and IPM can be completely separated, and avoid studying a rebuilding game.

Application #2: Incremental thresholded min-cost flow. [BLS23] showed that the IPM which reduces min-cost flow to dynamic min-ratio cycle naturally extends to the setting of *incremental thresholded min-cost flow* (Definition 1.1) with a few modifications. For this problem, the IPM still only needs to solve dynamic min-ratio cycle (Informal Definition 1.3) across $m^{1+o(1)}$ updates. However, due to difficulties in reasoning about the rebuilding game and the update sequence, [BLS23] was not able to show that the data structures of [CKL+22, BCK+23a] suffice for the incremental setting, and only achieved a $m^{1+o(1)}\sqrt{n}$ runtime. Our deterministic min-ratio cycle data structure avoids these issues. We prove the following.

Theorem 1.4 (Incremental thresholded min-cost flow)**.** *There is an algorithm* MinCostFlow(G, F) *that given an incremental directed graph* $G = (V, E, \boldsymbol{u}, \boldsymbol{c})$ *with capacities* $\boldsymbol{u}$ *in* $[1, U]$*, costs* $\boldsymbol{c}$ *in* $[-C, C]$ *such that* $U, C \leq m^{O(1)}$ *where* m *is an upper bound on the total number of edges in the graph, a demand* $\boldsymbol{d} \in \mathbb{R}^V$*, and a parameter* $F \in \mathbb{R}_{\geq 0}$ *reports a flow of cost at most* F *the moment such a flow becomes feasible. The algorithm is deterministic and runs in time* $m \cdot e^{O(\log^{167/168} m \log\log m)}$*.*

Remark 1.5. *The theorem can be extended to work with fixed point arithmetic with polylogarithmic bit precision (see [CKL+22]). Furthermore, the algorithm does not need to know the final edge count* m*, as we can restart after the edge count has doubled.*

A simple corollary of Theorem 1.4 is an algorithm for approximate incremental min-cost flow.

Theorem 1.6 (Approximate incremental min-cost flow)**.** *There is an algorithm that given an incremental directed graph* $G = (V, E, \boldsymbol{u}, \boldsymbol{c})$ *with capacities* $\boldsymbol{u}$ *in* $[1, U]$*, costs* $\boldsymbol{c}$ *in* $[1, C]$ *such that* $U, C \leq m^{O(1)}$ *where* m *is an upper bound on the number of edges in* G*, and demand* $\boldsymbol{d} \in \mathbb{R}^V$ *maintains a flow of cost at most* $(1+\epsilon)\mathrm{OPT}$ *throughout where* OPT *denotes the cost of the current min-cost flow. The algorithm is deterministic and runs in time* $m\epsilon^{-1} \cdot e^{O(\log^{167/168} m \log\log m)}$*.*

Proof. We run a thresholded min-cost flow algorithm (Theorem 1.4) for each threshold $(1+\epsilon)^i$, and notice that OPT is monotonically decreasing because the graph is incremental. Since the initial cost is polynomially upper bounded by mCU, the number of thresholded min-cost flow algorithms we run is $O(\epsilon^{-1}\log(mCU))$. The result follows directly from Theorem 1.4. □

This also implies approximate incremental maximum flow (as a special case) and approximate incremental weighted bipartite matching (by standard reductions). We note that the dependence on ϵ is optimal under the online matrix-vector (OMv) conjecture [HKNS15]. Indeed, even the dynamic incremental bipartite matching problem requires $\Omega(mn^{1-o(1)})$ time under OMv. This should be compared with several previous algorithms for dynamic incremental matching which incur worse ϵ dependencies [Gup14, BKS23, BK23] (at least ϵ^{-2}). Another notable prior work is [GH22] which gave the first sublinear amortized time algorithm for $(1-\epsilon)$-approximate maximum flow in incremental directed unit capacity graphs and achieved $m^{1.5+o(1)}/\epsilon^{1/2}$ time.

Application #3: Incremental cycle detection and strongly connected components. The incremental cycle detection problem asks us to find, in a directed graph undergoing edge insertions, the first update during which the graph has a directed cycle. This problem has been extensively studied over the last two decades [MNR96, KB06, LC07, AFM08, AF10, BFG09, HKM+12, BFGT16, CFKR13, BC18, BK20], with the current best runtimes being the minimum of $\widetilde{O}(m^{4/3})$ [BK20] and $O(n^2\log n)$ [BFGT16]. Incremental cycle detection can trivially be cast as a minimum cost cycle problem by giving every edge capacity 1, cost -1, and setting the threshold $F=-1$. Thus, we achieve a deterministic almost linear time algorithm for incremental cycle detection.

Theorem 1.7 (Incremental cycle detection)**.** *There is an algorithm that given a directed graph G undergoing edge insertions, reports the first update during which G has a directed cycle. The algorithm is deterministic and runs in time $m\cdot e^{O(\log^{167/168} m\log\log m)}$.*

A generalization of the incremental cycle detection problem is to maintain the strongly connected components (SCCs) in an incremental graph. For this problem, the best known runtimes are the same as those for incremental cycle detection, due to work of [BDP21] achieving runtime $\widetilde{O}(m^{4/3})$ and [BFGT16], which achieves $O(n^2\log n)$ time. Somewhat surprisingly, the dynamic IPM framework gives a way to maintain SCCs. Intuitively, this is because the IPM maintains some circulation on the graph, such that when some edge in the circulation has nontrivial amounts of flow, it is guaranteed to be in an SCC, and thus can be contracted. Contracting an edge preserves that the remaining flow is still a circulation, so we can continue running the IPM on the contracted graph.

Theorem 1.8 (Incremental SCC)**.** *There is an algorithm that given a directed graph G undergoing edge insertions, explicitly maintains the strongly connected components of G. The algorithm is deterministic and runs in time $m\cdot e^{O(\log^{167/168} m\log\log m)}$.*

We should mention that nearly all the previous works discussed above also give an algorithm for maintaining a topological sort in an incremental graph, where the best runtimes are again a combination of $\widetilde{O}(m^{4/3})$ [BK20] and $O(n^2\log n)$ [BFGT16]. We do not know how to do this with our current techniques.

Application #4: Incremental s-t shortest path. For fixed vertices s and t, the s-t shortest path, even in a graph with negative edge lengths, can be cast as a min-cost flow problem. Thus,

we obtain a deterministic almost linear time algorithm for a thresholded version of s-t shortest path even in graphs with negative edge lengths. For graphs with positive edge lengths, we obtain a $(1+\epsilon)$-approximation algorithm with total runtime $m\epsilon^{-1}e^{O(\log^{167/168} m \log\log m)}$ by Theorem 1.6.

Previously, the best runtimes to maintain $(1+\epsilon)$-approximate incremental s-t shortest path were $\widetilde{O}(m^{4/3}/\epsilon^2)$ [KMP22] and $\widetilde{O}(n^2/\epsilon^{2.5})$ [PVW20]. However, all previous algorithms [PVW20, CZ21, KMP22] consider the more general problem of maintaining $(1+\epsilon)$-approximate single-source shortest paths (SSSP) in directed graphs. [Ber13] further gives a randomized $\widetilde{O}(mn/\epsilon)$ time algorithm for the incremental $(1+\epsilon)$-approximate all-pairs shortest paths (APSP) problem. This is the best possible runtime given the approximation factor of $(1+\epsilon)$ [DHZ00, WW10]. The currently best deterministic such algorithm obtains total runtime $\widetilde{O}(mn^{4/3}/\epsilon)$ [KL19]. In planar incremental graphs, the APSP problem can be solved with $\widetilde{O}(\sqrt{n})$ worst-case time per update/query [DGWN22].

It is thus an interesting question whether the techniques in this work can be used to give a (deterministic) algorithm for $(1+\epsilon)$-approximate incremental single-source shortest path with almost linear total update time as this would result in near-optimal algorithms for both the $(1+\epsilon)$-approximate incremental SSSP and APSP problems. Finally, we point out that in undirected graphs, both of these questions are settled [HKN16, GWN20, BGS22, KMP23].

1.3 Overview

(Incremental) min-cost flow via dynamic min-ratio cycle problems. Building on the almost linear time min-cost flow algorithm of [CKL+22], [BLS23] used an interior point method (IPM) to show that incremental thresholded min-cost flow can be solved in $m^{1+o(1)}$ calls to a data structure which finds a min-ratio cycle in a graph undergoing at most $m^{1+o(1)}$ dynamic updates, and augments along this cycle. Our main technical contribution is an algorithm that essentially maintains an ℓ_1-oblivious routing for a dynamic graph that allows us to extract approximate min-ratio cycles.

Thus, we focus on designing such a data structure in the remainder of the overview. We give a description of the IPM in Section 8 both for completeness and because our SCC algorithm (Theorem 1.8) requires white-boxing the IPM. We present the SCC algorithm in Section 8.3.

Finding approximate min-ratio cycles via an ℓ_1-oblivious routing. Before we delve into a more technical discussion, let us first define ℓ_1-oblivious routings and show how to use these objects statically to extract an approximate min-ratio cycle.

Formally, an ℓ_1-oblivious routing for a graph G is a linear mapping $\boldsymbol{A} \in \mathbb{R}^{E\times V}$ that takes any demand vector $\boldsymbol{d} \in \mathbb{R}^V, \boldsymbol{d}\perp\mathbf{1}$ and maps it to a flow $\boldsymbol{f} = \boldsymbol{A}\boldsymbol{d}$ that routes the demand $\boldsymbol{d}$. The quality of an oblivious routing matrix $\boldsymbol{A}$ is the worst-case ℓ_1 cost of the flow $\boldsymbol{f}$ compared to the cheapest flow routing demand $\boldsymbol{d}$. It is not hard to establish that the quality of $\boldsymbol{A}$ in a unit-length graph is given by the maximum average flow path length to route a flow between the endpoints of an edge in E which is given by the quantity $\|\boldsymbol{A}\|_{1\to 1}$. The proof extends to lengths where the quality of $\boldsymbol{A}$ is then equal to $\|\boldsymbol{L}\boldsymbol{A}\boldsymbol{L}^{-1}\|_{1\to 1}$.

A very simple oblivious routing scheme (though of very poor quality) is to fix an orientation of the edge set E, take a spanning tree T in G, root it at an arbitrary vertex $r \in V(T)$, and let $\boldsymbol{A}$ be the matrix defined in the entry of edge $e = (x, y)$ and vertex v by

$$\boldsymbol{A}_{e,v} = \begin{cases} 1 & \text{if } e \in T[v,r] \text{ and } x \text{ appears on } T[v,r] \text{ before } y \\ -1 & \text{if } e \in T[v,r] \text{ and } x \text{ appears on } T[v,r] \text{ after } y \\ 0 & \text{otherwise} \end{cases}$$

Consider now the demand $\boldsymbol{d} = \mathbf{1}_t - \mathbf{1}_s$ that sends one unit of flow from source s to sink t. Let x be the lowest common ancestor (LCA) of s and t in T. Then note that $\boldsymbol{A}\boldsymbol{d} = \boldsymbol{A}(\mathbf{1}_t - \mathbf{1}_s)$ adds flow on the edges $T[s,x]$ and $T[x,t]$ but cancels the flow on $T[x,r]$. While using a single tree T to obtain an ℓ_1-oblivious routing cannot guarantee good quality, a distribution over roughly m carefully chosen trees (called low-stretch spanning trees) yields an ℓ_1-oblivious routing of quality $O(\log m)$ [Räc08].

Let us now consider the min-ratio cycle problem for G where $\boldsymbol{l}$ are the lengths and $\boldsymbol{g}$ denotes the gradients. Let us fix an optimal circulation $\boldsymbol{\Delta}^*$, e.g. $\boldsymbol{B}^\top \boldsymbol{\Delta}^* = 0$, and let us assume for convenience that the orientation of all edges coincides with the direction of the flow on $\boldsymbol{\Delta}^*$. Let us define for an edge $e = (u,v)$ the vector $\boldsymbol{b}_e = \mathbf{1}_v - \mathbf{1}_u$, i.e. the unit demand sending one unit of flow from the tail of e to its head. Then, note that for any oblivious routing $\boldsymbol{A}$, we have $\sum_{e \in E} \boldsymbol{A}\, \boldsymbol{\Delta}^*_e \boldsymbol{b}_e = \mathbf{0}$ because since $\boldsymbol{\Delta}^*$ is a circulation, the total demand at every vertex from vectors $\boldsymbol{b}_e$ weighted by $\boldsymbol{\Delta}^*_e$ is 0.

But this implies that there is an edge $e \in E$ such that the circulation obtained from routing one unit of flow along an edge e and the routing the demand $-\boldsymbol{b}_e$ via the ℓ_1-oblivious routing is competitive to the min-cycle ratio cycle. This follows from the calculations below, we have

$$\begin{aligned}
\min_{e \in E} \frac{\boldsymbol{g}^\top \boldsymbol{\Delta}^*_e(\mathbf{1}_e - \boldsymbol{A}\boldsymbol{b}_e)}{\|\boldsymbol{L}\,|\boldsymbol{\Delta}^*_e(\mathbf{1}_e - \boldsymbol{A}\boldsymbol{b}_e)|\|_1} &\leq \frac{\sum_{e \in E} \boldsymbol{g}^\top \boldsymbol{\Delta}^*_e(\mathbf{1}_e - \boldsymbol{A}\boldsymbol{b}_e)}{\sum_{e \in E} \|\boldsymbol{L}\,|\boldsymbol{\Delta}^*_e(\mathbf{1}_e - \boldsymbol{A}\boldsymbol{b}_e)|\|_1} = \frac{\boldsymbol{g}^\top \boldsymbol{\Delta}^*}{\sum_{e \in E} \|\boldsymbol{L}\,|\boldsymbol{\Delta}^*_e(\mathbf{1}_e - \boldsymbol{A}\boldsymbol{b}_e)|\|_1} \\
&\leq \frac{\boldsymbol{g}^\top \boldsymbol{\Delta}^*}{\sum_{e \in E} \boldsymbol{l}_e|\boldsymbol{\Delta}^*_e| + \|\boldsymbol{L}\boldsymbol{A}\boldsymbol{L}^{-1}\|_{1\to 1}\boldsymbol{l}_e|\boldsymbol{\Delta}^*_e|} \\
&= (1 + \|\boldsymbol{L}\boldsymbol{A}\boldsymbol{L}\|_{1\to 1})^{-1} \frac{\boldsymbol{g}^\top \boldsymbol{\Delta}^*}{\|\boldsymbol{L}\boldsymbol{\Delta}^*\|_1},
\end{aligned}$$

where the first inequality is an averaging argument, and the second uses the triangle inequality and the quality of the oblivious routing $\boldsymbol{A}$ with respect to $\boldsymbol{L}$. Thus, some circulation $\mathbf{1}_e - \boldsymbol{A}\boldsymbol{b}_e$ has quality within a $O(\|\boldsymbol{L}\boldsymbol{A}\boldsymbol{L}^{-1}\|_{1\to 1})$ factor of the best ratio.

Note that if $\boldsymbol{A}$ was again obtained from a single tree, then $\mathbf{1}_e - \boldsymbol{A}\boldsymbol{b}_e$ would form a simple cycle. However, in general, $\mathbf{1}_e - \boldsymbol{A}\boldsymbol{b}_e$ is simply a circulation. In our approach, we show that the particular structure of $\boldsymbol{A}$ then allows us to further decompose this circulation into few cycles such that one of these preserves an approximate min-ratio cycle.

Dynamic min-ratio cycles in previous work. Before we describe our new approach, let us briefly reflect on previous work with the perspective from above. [CKL+22] used probabilistic low-stretch spanning trees (LSSTs) to find the min-ratio cycle. Interpreting their framework with our perspective from above, they essentially sample $m^{o(1)}$ LSSTs from the distribution over trees that forms the ℓ_1-oblivious routing given in [Räc08] and argue that one of them has a fundamental tree cycle that is approximately a min-ratio cycle (the argument follows our discussion above as each LSSTs has the same expected guarantees as its underlying ℓ_1-oblivious routing). But since the IPM requires us to solve a dynamic min-ratio cycle problem, [CKL+22] then has to argue that updates to the graph do not interfere much with the randomness used when picking the LSSTs as they do not have time to sample a new tree after each update. Instead, they resample some edges if the data structure fails to produce a good cycle. While [BCK+23a] was able to derandomize the above construction, this was done using the structure of the update sequence produced by the IPM and does not solve the min-ratio cycle problem against an adaptive adversary.

High-level strategy. In this work, we give an algorithm that essentially maintains an ℓ_1-oblivious routing $\boldsymbol{A}$ of graph G with quality $m^{o(1)}$ and $m^{o(1)}$ update time. Moreover, we show that the

particular structure of the ℓ_1-oblivious routing $\boldsymbol{A}$ that we maintain allows us to decompose each circulation $\mathbf{1}_e - \boldsymbol{A}\boldsymbol{b}_e$ into only $m^{o(1)}$ cycles, one of which being approximately of quality comparable to the quality of the circulation $\mathbf{1}_e - \boldsymbol{A}\boldsymbol{b}_e$. This then allows us to efficiently extract from the ℓ_1-oblivious routing $\boldsymbol{A}$ an approximate min-ratio circulation at any time. This is in stark contrast with previous approaches that could only maintain a small subsample of the ℓ_1-oblivious routing and our stronger invariant presents major challenges both in terms of maintaining the ℓ_1-oblivious routing itself *and* for extracting a min-ratio circulation, as the object we maintain essentially covers all cycles approximately (instead of only maintaining each fixed cycle with high probability).

We emphasize another important point in our approach: while [CKL+22, BCK+23a] argues about preservation of lengths and gradients by the LSSTs simultaneously, our framework separates these concerns: we maintain the ℓ_1-oblivious routing with an algorithm that is oblivious to the gradients. Only thereafter, when the routing is given, do we introduce gradients again and show how to extract a min-ratio cycle. Thus, a crucial point is that we first deal with maintaining distances, where we offload some work to the toolbox designed in [KMP23], and then develop and use another set of tools that allow us to handle gradients and flow routing. We note that there are major obstacles to directly extending the the distance preservation techniques of [KMP23] to simultaneously handle gradients.

Our algorithmic framework. To obtain our algorithm, we build on the framework in [RGH+22] to maintain an ℓ_1-oblivious routing. [RGH+22] takes as an initial point a rather simple building block, a so-called sparse neighborhood cover (SNC). Given graph G and a distance parameter D, an SNC is a collection $\mathcal{C}$ of clusters $C \in \mathcal{C}$ such that

1. for every vertex $v \in V$, the ball of radius D around v, which we denote as $B_G(v, D)$, is contained in some cluster C, and
2. each vertex $v \in V$ appears in at most $\widetilde{O}(1)$ clusters, and
3. each cluster C has diameter at most $m^{o(1)} \cdot D$.

[RGH+22] maintains SNCs $\mathcal{C}_i$ for increasing distance parameters D_i. We choose $D_i = \gamma^i$ for $\gamma = m^{o(1)}$. Then, each cluster $C \in \mathcal{C}_i$ is assigned an arbitrary cluster center r_C and selects flow paths to go from centers of clusters $C \in \mathcal{C}_i$ to centers of clusters $C' \in \mathcal{C}_{i+1}$ with $C \subseteq C'$. Thus, note that flow out of C only goes to $\widetilde{O}(1)$ different clusters. Finally, [RGH+22] carefully chooses an extremely clever but simple weighting of these flow paths (we however will not maintain such weights).

In order to dynamize the algorithm from [RGH+22], we first design a new algorithm to maintain SNCs in a fully dynamic graph. While this in itself already presents a considerable challenge, we also need to maintain low-diameter trees spanning each cluster $C \in \mathcal{C}_i$ to keep track of the paths between centers of clusters $C \in \mathcal{C}_i$ and $C' \in \mathcal{C}_{i+1}$. In fact, later, when we route flow along min-ratio cycles, we have to use these low-diameter trees to efficiently maintain the flow routed. To maintain the low-diameter trees, we build on concurrent work of [KMP23].

Given the dynamic SNC algorithm, we then maintain a relaxed version of the oblivious routing of [RGH+22], which we call an *abstracted hierarchical routing graph* (abstracted HRG) $\widetilde{H}$. This graph $\widetilde{H}$ consists of $\kappa := O(\log_\gamma m)$ layers where each layer $0 \leq i \leq \kappa$ consists of a copy V_i of the vertex set $V(G)$. We draw an edge out of $v_i \in V_i$ as follows: let $v_i \in C_i$, and $C_i \subseteq C_{i+1}$. Then draw an edge from v_i to $r_{C_{i+1}}$. At a high level, this graph captures all possible paths that flow can go on in the oblivious routing. We direct edges in $\widetilde{H}$ from lower layers to higher layers, and thus $\widetilde{H}$ is

a directed acyclic graph (DAG). Again, the maximum out-degree of any vertex in $\widetilde{H}$ is $\widetilde{O}(1)$. We let the HRG be the graph H obtained from $\widetilde{H}$ by replacing each edge in $\widetilde{H}$ by a path between the same endpoints from the low-diameter tree on the corresponding cluster. Thus, paths in the HRG exactly are paths in G that the oblivious routing pushes flow onto. See Figure 1, and Definition 4.2 for a more formal definition.

We then show that picking random out-edges in $\widetilde{H}$ gives with probability $m^{-o(1)}$ a tree T that yields a fundamental cycle that approximately preserves the min-ratio cycle. We show that this approach can be derandomized and maintain $m^{o(1)}$ trees $T_1, T_2, \ldots, T_\lambda$ of $\widetilde{H}$ such that one of them has a fundamental cycle that approximately preserves the min-ratio cycle.

Finally, we are left with extracting such a fundamental tree cycle. At a high level, our approach is similar to how [CKL+22] extracts such cycles from its LSSTs, we use vertex and edge-reduction. To this end, we partitioning the trees $T_1, T_2, \ldots, T_\lambda$ further into forests $F_1, F_2, \ldots, F_\lambda$ each with $\Omega(m/k)$ connected components by deleting edges that ensure that each component is incident to $\widetilde{O}(k)$ volume of the underlying graph G. We then show for each $1 \leq i \leq \lambda$, how to extract an approximate min-ratio cycle: either if it was preserved on F_i, we use that each component of F_i is small to extract it, or if the approximate min-ratio cycle preserved was preserved by T_i but not by F_i, we extract it by recursing on the graphs obtained from contracting in G vertices in the same component of a forest F_i obtained from partitioning T_i. For each forest F_i, this contraction yields a graph P_i with only $O(m/k)$ vertices. We then use a spanner (i.e. a distance-preserving edge sparsifier) on this graph P_i to reduce the edge count down to $m^{1+o(1)}/k$ and recurse on the resulting smaller graph S_i. Applying edge sparsification to P_i unfortunately means we cannot guarantee that there exists a good tree cycle w.r.t. the contraction of tree T_i in S_i. Hence, our recursion on H_i needs to solve the min-ratio cycle problem, not the simpler min-ratio tree cycle problem. Our edge sparsification procedure needs significantly stronger properties than the spanner in [CKL+22]. We build a more powerful spanner to solve this issue, as we describe later in the overview.

In the next sections, we describe the components of the algorithm in more detail.

From hierarchical routing graphs (HRGs) to min-ratio cycle-preserving trees. We now describe how to use the HRG to maintain $m^{o(1)}$ trees, such that in some tree, the fundamental cycle arising from the tree combined with an off-tree edge is an approximate min-ratio circulation. To start, as we showed earlier, there is an edge $e \in E(G)$ such that obliviously routing e in the HRG gives an approximate min-ratio circulation. Algebraically, this circulation is $\boldsymbol{A}\boldsymbol{B}^\top \mathbf{1}_e - \mathbf{1}_e$, where $\mathbf{1}_e$ is the unit flow on edge e (note that $\boldsymbol{b}_e = \boldsymbol{B}^\top \mathbf{1}_e$). However, the flow $\boldsymbol{A}\boldsymbol{B}^\top \mathbf{1}_e$ may consist of multiple paths within the HRG, because vertices are in multiple clusters. To reduce to considering circulations which are a simple cycle, we will build a collection of trees that "covers" all flow paths in the HRG. To this end, we show that for $e = (u, v)$, we can decompose the flow $\boldsymbol{A}\boldsymbol{B}^\top \mathbf{1}_e$ into paths each consisting of a path aligning with the HRG from u to one of the nodes in a higher layer, and the reverse path from such a higher layer node down to vertex v. Then, to cover these flow paths, consider picking a random edge out of each vertex $v_i \in V_i$ for all i, and repeating $m^{o(1)}$ times, to find trees $T_1, T_2, \ldots, T_\lambda$. Indeed, note that all flow paths use at most $O(\kappa)$ hops in the abstracted HRG (this is the number of layers in the HRG where we defined $\kappa := O(\log_\gamma m)$ which is sublogarithmic), and each vertex has out-degree $\widetilde{O}(1)$. So the probability that a flow path is contained in a random tree T_j is $\widetilde{O}(1)^{-O(\kappa)} \geq m^{-o(1)}$ w.h.p.

A subtle issue arises: we claimed above that $\boldsymbol{A}\boldsymbol{B}^\top \mathbf{1}_e$ can be decomposed into flow paths that consist of a path segment starting in u that is only going up and a path segment ending in v that

is only going down in the HRG. We call these monotone cycles.

Unfortunately, this montone cycle decomposition must sometimes be lossy as shown by simple examples. However, we are still able to show that every circulation on the HRG can be decomposed into monotone cycles such that the sum of the cycle lengths is only a little larger than the length of the circulation which suffices to preserve a cycle of good quality.

Finally, we derandomize this construction by reducing the amount of randomness via coloring techniques and enumeration.

Completing our recursion: Querying min-ratio tree cycles with portal routing. Having constructed trees whose fundamental cycles yield approximate min-ratio cycles, we now need to solve the problem of finding a min-ratio cycle that consists of an off-tree edge and a simple path on a single tree in this collection derived from the HRG. We follow the high-level approach from [CKL+22]: we apply vertex and edge reduction tools to reduce the problem again to min-ratio cycle, but on a smaller graph with $O(m/k)$ edges, for some size reduction factor k. Then we recursively build an HRG on this graph to reduce to tree cycles again, all the way down to graphs of size $O(k)$. Combining these pieces gives our overall data structure.

Recall that, as described earlier, to solve the min-ratio tree cycle problem on a tree T, we first partition the tree into a forest F with m/k components of size $\widetilde{O}(k)$. We then check tree cycles that are internal to components, before constructing a smaller graph P with components of F contracted, so that P contains $O(m/k)$ vertices. P still has roughly m edges, and is hence rather dense. To construct P, we use a contraction scheme (called "portal routing" [ST04, KPSW19]), that differs from [CKL+22] as we have to ensure that no fundamental cycle is lengthened by contraction. This scheme is less stable than the scheme from [CKL+22] (called "core graphs"): Updates to G cause vertex splits in P where the edges incident to a split vertex may all decrease drastically in length (we model these decreases by re-inserting the edges with new lengths). This makes it challenging to apply edge sparsification to P, because a single split in P might cause up to $\widetilde{O}(k)$ edge insertions (the maximum degree of a vertex in P is $\widetilde{O}(k)$ by our decomposition of a tree T_i into a forest F_i). Thus, m/k updates to G may cause P to undergo roughly m edge insertions. But, we show that nonetheless, we can maintain a spanner S of P which only undergoes roughly m/k updates.

Fully dynamic sparse neighborhood cover. Dynamically maintaining the HRG and the hierarchical routing trees can be reduced to giving a deterministic data structure that maintains a sparse neighborhood cover of a fully dynamic graph, and a representation of a low-diameter tree over each cluster $C \in \mathcal{C}_i$. We discuss our fully dynamic SNC algorithm in Section 6. Although there are previous works on dynamic SNC [Chu21, BGS22, CZ23] it is unclear whether these algorithms can maintain a low-diameter tree representation of the SNC that allows for efficient routing.[3]

We maintain a fully dynamic SNC by leveraging several vertex sparsification tools from the work [KMP23], which developed a set of tools based on fully dynamic vertex sparsifiers that approximately preserve distances and have subpolynomial amortized update time. In particular, we require data structures which 1) maintain a SNC under edge *deletions* only (Theorem 6.1), 2) maintain a vertex sparsifier onto a dynamic terminal set A (Theorem 6.3), 3) maintain a low-diameter tree over a graph (Theorem 6.2).

[3]It is not hard to see that these constructions internally maintain low-diameter trees that embed with very high edge-congestion into G which does not suffice for our purposes.

We briefly describe how to use these pieces. Let the dynamic graph be G, and let $\widehat{G}$ be the dynamic graph if we ignore all insertions. We start by maintaining a decremental SNC on G with diameter parameter D. At the same time, we grow a terminal set A consisting of endpoints of all inserted edges, on which we maintain a vertex sparsifier called H. We recurse on H (with slightly larger diameter parameter), and rebuild G whenever $|H| > |G|/\gamma$, for a size reduction factor $\gamma = m^{o(1)}$.[4]

Let us describe why this construction works. We show that for every vertex v there is a maintained cluster that contains the ball of radius D/γ_{SNC} around it. There are two cases. If a low-diameter cluster $\widehat{C}$ in $\widehat{G}$ has no touching edge insertions, then any ball $B_{\widehat{G}}(v, D/\gamma_{\text{SNC}}) \subseteq \widehat{C}$ also satisfies $B_G(v, D/\gamma_{\text{SNC}}) \subseteq \widehat{C}$. Thus, $\widehat{C}$ still covers many balls around vertices v in G. Otherwise, there is an insertion touching $\widehat{C}$, say at vertex c. We know $c \in A$ (the terminal set), because we add all endpoints of insertions to A. Now, recall that we recurse on H, which is a vertex sparsifier for terminal set A. Thus, we recursively maintain some sparse neighborhood cover on H, in particular a low-diameter set containing $c \in C$ (let's call it C_H). Thus, we can consider the low-diameter set $C \cup C_H$, which we show contains balls around v.

To get a low-diameter forest/tree representation out of this construction, we maintain a low-diameter tree on each cluster in $\widehat{G}$ (and recursively on vertex sparsifiers), and glue these together at terminals (when forming the sets $C \cup C_H$) to form the overall forest. Our formal construction is more complicated than this, partially because we must duplicate vertices and forests multiple times due to vertex congestion in the vertex sparsifiers.

Low-recourse dynamic spanner under many insertions. When solving the problem of min-ratio tree cycle, we apply vertex and edge reduction techniques. We require a stronger spanner than [CKL+22] to implement our edge reduction step, and we develop this in Section 7.

A spanner H of a graph G with stretch γ is a subgraph of G with the same vertex set such that for every edge (u, v) in G there is a path between the u and v that is at most a factor γ longer than the edge e. By standard reductions, it suffices to consider the unit weight case.

In [CKL+22, BCK+23a] it was sufficient for the spanner to handle roughly n edge insertions and deletions, and vertex splits. In this case, the inserted edges can all be added to H without increasing its density by more than an additive n. Therefore, a dynamic spanner handling edge deletions and vertex splits suffices. We present a spanner that can handle a large amount of insertions.

Informal Theorem 1.9. *Given a dynamic graph G undergoing a sequence of up to Δn edge insertions and up to n edge deletions and vertex splits such that the maximum degree in G never exceeds Δ even when playing the update sequence without vertex splits. Then, there is an algorithm that maintains a spanner containing at most at most γn edges with stretch γ and total recourse γn for some $\gamma = m^{o(1)}$ independent of Δ. The total runtime is $\gamma n \Delta$.*

We first describe a simple low recourse version of this result, and then sketch how we obtain the efficient version. Recall the classic spanner construction by [ADD+93]. Given $G = (V, E)$, initialize the spanner $H = (V, \emptyset)$. Then, consider all edges $(u, v) \in E$ in some arbitrary order. Whenever there is no path of length $2 \log n$ between u and v in H, add (u, v) to H. This construction ensures that H has girth $> 2 \log n$, which implies that H only contains $O(n)$ edges. Notice that this already describes an incremental algorithm.

[4] Our approach is somewhat similar to the technique developed in [FGNS23] for maintaining approximate APSP in a fully dynamic graph by reducing to maintaining approximate APSP on decremental graphs.

After deletions and vertex splits, we observe that these operations only increase the girth of H. Therefore, after each edge deletion/vertex split we apply H we go through all edges (u, v) currently in G and check if there is a path of length $2 \log n$ between u and v in H. If not, we add (u, v) to H. Since the girth of H remains $> 2 \log n$, the number of edges in H remains bounded by $O(n)$. But an edge only leaves H when it is deleted since no edges leave H when a vertex is split. This bounds the amortized recourse caused by insertions and deletions. A simple potential function argument can be used to show that the recourse caused by simulating vertex splits is also at most $\widetilde{O}(n)$.

To achieve an efficient implementation, we first observe that, using fully dynamic APSP data structures [CZ23, FGNS23, KMP23], we can essentially implement the classic greedy incremental algorithm in almost-linear time. Furthermore, we can maintain explicit, short embedding paths for each edge $e \in E \setminus E_H$ into E_H. To manage edge deletions and vertex splits, we combine the incremental spanner idea with the decremental batching scheme of [CKL+22]. To make this idea work, we transform the incremental spanner to produce an embedding from G to H with low vertex congestion. It is worth mentioning that the work [BSS22] realized that the greedy algorithm can maintain a spanner with optimal $O(1)$ amortized recourse in the decremental setting, albeit with large polynomial running time. However, they did not observe that in fact the spanner can be made fully dynamic with extremely low recourse under edge insertions, and instead appealed to standard recursions to achieve $O(1)$ recourse per insertion (which is insufficient for our purposes).

Tree representations. A point we have glossed over is how we precisely route the min-ratio cycle that our algorithm returns. In [CKL+22], the algorithm dynamically maintain a spanning tree of the graph, and the approximate min-ratio cycle was always represented by $m^{o(1)}$ off-tree edges and $m^{o(1)}$ paths on this tree. Due to the complexity of our data structures, we actually maintain a generalization of a spanning tree, which we call a forest F that has a *flat embedding* into G. Formally, this means that there is a map $\Pi : V(F) \to V(G)$ such that for every edge $e = (x, y) \in V(F)$, $(\Pi(x), \Pi(y))$ is also an edge in G. This way, paths and cycles in F exactly correspond to paths and cycles in G. Overall, we will represent our min-ratio cycles as $m^{o(1)}$ off-tree edges and paths on F. The tree-representation of the SNC, and the tree in the portal routing construction both have flat embeddings into G. Crucially, this embedding is of low edge-congestion $m^{o(1)}$. This allows us to maintain the absolute change of flow for a single edge in G up to a $m^{o(1)}$ approximation which is crucial for the IPM to maintain (approximate) gradients and lengths efficiently.

2 Preliminaries

General notation. We denote vectors as bold lowercase letter $\boldsymbol{v}$, and matrices as bold uppercase letter $\boldsymbol{A}$. We let $\boldsymbol{v}(j)$ denote the j-th element in vector $\boldsymbol{v}$ and $\boldsymbol{A}(i, j)$ the element at position (i, j) accordingly. For a vector $\boldsymbol{v}$, we let $|\boldsymbol{v}|$ denote the coordinate-wise absolute value. Given two scalars α and β, we let $\alpha \approx_\kappa \beta$ if $\alpha/\kappa \leq \beta \leq \kappa\alpha$.

Graphs. In this article, we typically work with undirected graphs unless explicitly specified. That is while the min-cost flow problem is solved on a directed graph, we reduce via the IPM to sequence of subproblems on undirected graphs. On such graphs, we still use orientations of edges as we use gradients and flows over these graphs.

We let $G = (V, E, \boldsymbol{l}, \boldsymbol{g})$ denote a graph on the vertex set V with edge set E, $\boldsymbol{l}(\cdot) : E \mapsto \mathbb{R}_{\geq 0}$ is the vector storing the lengths of the edges, and $\boldsymbol{g}(\cdot) : E \mapsto \mathbb{R}$ is the vector for edge gradients. We

let $\boldsymbol{B}_G \in \mathbb{R}^{E \times V}$, or $\boldsymbol{B}$ if G is clear from the context, denote the edge vertex incidence matrix where we attribute arbitrary directions to edges. We usually let $n = |V|$ and $m = |E|$. We further denote $\boldsymbol{L} = \mathrm{diag}(\boldsymbol{l})$.

We call a flow vector $\boldsymbol{c} \in \mathbb{R}^E$ a circulation if $\boldsymbol{B}^\top \boldsymbol{c} = \boldsymbol{0}$. Given a graph G, we sometimes denote its vertex set by V_G and its edge set by E_G.

Distances. For a graph $G = (V, E, \boldsymbol{l})$ with edge lengths $\boldsymbol{l} \in \mathbb{R}^E_{\geq 0}$, we let $\mathrm{dist}_G(u, v)$ be the length of the shortest weighted path between u and v. We let $\mathrm{diam}(G)$ denote the maximum distance between two vertices in G, i.e. $\mathrm{diam}(G) = \max_{u,v} \mathrm{dist}_G(u, v)$. We denote the *ball* of radius D around v as $B_G(v, D) := \{u \in V(G) : \mathrm{dist}_G(u, v) \leq D\}$.

Embeddings. Given two graphs $G = (V_G, E_G)$ and $H = (V_H, E_H)$ with $V_G \subseteq V_H$ we call a map of edges in G to paths between their respective endpoints in H a (edge) embedding and often denote such an embedding by $\Pi_{G \mapsto H}$. Similarly, we often denote by $\Pi_{V_G \mapsto V_H}$ a map from vertices in V_G to vertices in V_H and we call it either a vertex embedding or a (vertex) map. We sometimes drop the subscript if it is clear from the context.

Definition 2.1 (Vertex and Edge congestion)**.** *Given a graph $G = (V, E)$, a (sub-)graph $H = (V_H, E_H)$ with $V \subseteq V_H$ and an edge embedding $\Pi_{G \mapsto H}$ that maps edges $e = (u, v)$ to a uv-path P in H we let:*

- *The edge congestion* $\mathrm{econg}(\Pi_{G \mapsto H}, e')$ *of edge $e' \in E_H$ be the total number of paths that use edge e'. Further, we let* $\mathrm{econg}(\Pi_{G \mapsto H})$ *be the maximum edge congestion observed by an edge in E_H.*
- *The vertex congestion* $\mathrm{vcong}(\Pi_{G \mapsto H}, v)$ *of $v \in V_H$ be the total number of paths in Π that contain v. Similarly, we let* $\mathrm{vcong}(\Pi_{G \mapsto H})$ *be the maximum vertex congestion observed by a vertex in V.*

We will sometimes consider broken embeddings, i.e. embeddings $\Pi_{G \mapsto H}$ where edges on an embedding path got removed or vertices got split. We therefore don't require the paths P to correspond to paths in H, but they may simply be an ordered collection of edges in H.

Definition 2.2 (Vertex map congestion)**.** *For a vertex map $\Pi_{V_G \mapsto V_H}$, we let* $\mathrm{vcong}(\Pi_{V_G \mapsto V_H})$ *be the maximum number of vertices in V_G that map to the same vertex in V_H.*

Definition 2.3 (Flat embedding)**.** *We say for graphs $G = (V, E)$ and $H = (V_H, E_H)$ that a map $\Pi_{V_H \mapsto V}$ is a* flat embedding *of H into G if for every edge $e = (x, y) \in E_H$, for $x' = \Pi_{V_H \mapsto V}(x)$ and $y' = \Pi_{V_H \mapsto V}(y)$, either (x', y') is an edge in G, or $x' = y'$.*

Oblivious routings. Our data-structure is based on dynamically maintaining ℓ_1-oblivious routings for graphs $G = (V, E, \boldsymbol{l})$, which are linear mappings $\boldsymbol{A} \in \mathbb{R}^{E \times V}$ of demands $\boldsymbol{d} \in \mathbb{R}^V$ to flows $\boldsymbol{f} \in \mathbb{R}^E$ which route that demand. In other words, for $\boldsymbol{d} \perp \boldsymbol{1}$ the flow $\boldsymbol{f} = \boldsymbol{A}\boldsymbol{d}$ satisfies $\boldsymbol{B}^T \boldsymbol{f} = \boldsymbol{d}$, i.e. it routes the demands $\boldsymbol{d}$. For every edge $e = (u, v)$ in G, we let $\boldsymbol{1}_e$ denote the flow that routes one unit across edge e, and $\boldsymbol{b}_e$ for the demand that it routes.

Definition 2.4 (Oblivious routing)**.** *We call a matrix $\boldsymbol{A} \in \mathbb{R}^{E \times V}$ a γ-approximate ℓ_1-oblivious routing for graph $G = (V, E, \boldsymbol{l})$ if*

1. *for all* $\boldsymbol{d}$ *so that* $\boldsymbol{d}^T\mathbf{1} = 0$*:* $\boldsymbol{B}^T\boldsymbol{A}\boldsymbol{d} = \boldsymbol{d}$
2. $\left\|\boldsymbol{L}\boldsymbol{A}\boldsymbol{B}^T\boldsymbol{L}^{-1}\right\|_{1\to 1} \leq \gamma$

Remark 2.5. *Notice that this coincides with the combinatorial definition of an* ℓ_1*-oblivious routing over competitive ratios, i.e. consider an arbitrary flow* $\boldsymbol{f}$ *and observe*

$$\frac{\left\|\boldsymbol{L}\boldsymbol{A}\boldsymbol{B}^T\boldsymbol{f}\right\|_1}{\|\boldsymbol{L}\boldsymbol{f}\|_1} = \frac{\|\boldsymbol{L}\boldsymbol{A}\boldsymbol{B}^T\boldsymbol{L}^{-1}\widetilde{\boldsymbol{f}}\|_1}{\|\widetilde{\boldsymbol{f}}\|_1} \leq \gamma$$

which follows by the definition of $\|\cdot\|_{1\to 1}$ *and* $\boldsymbol{f}$ *and thus* $\widetilde{\boldsymbol{f}}$ *are arbitrary.*

For an oblivious routing $\boldsymbol{A}$, we let $\boldsymbol{P} = \boldsymbol{I} - \boldsymbol{A}\boldsymbol{B}^\top$ denote the corresponding *cycle projection matrix*, as it is a projection onto the space of circulations on G.

An averaging argument shows that there is a circulation $\boldsymbol{P}\mathbf{1}_e$ whose quality is within a multiplicative factor of the quality of the best min-ratio circulation.

Lemma 2.6 (Min-ratio circulation)**.** *Given a graph* $G = (V, E, \boldsymbol{l}, \boldsymbol{g})$*, a cycle projection matrix* $\boldsymbol{P} \in \mathbb{R}^{E\times E}$ *for* G *and an arbitrary matrix* $\boldsymbol{M} \in \mathbb{R}^{k\times E}$ *we have*

$$\min_{\substack{e\in E\\ \beta\in\{-1,1\}}} \beta \cdot \boldsymbol{g}^T\boldsymbol{P}\mathbf{1}_e / \|\boldsymbol{M}\mathbf{1}_e\|_1 \leq \frac{1}{\left\|\boldsymbol{M}\boldsymbol{L}^{-1}\right\|_{1\to 1}} \min_{\boldsymbol{\Delta}:\boldsymbol{B}^\top\boldsymbol{\Delta}=\mathbf{0}} \boldsymbol{g}^T\boldsymbol{\Delta} / \|\boldsymbol{L}\boldsymbol{\Delta}\|_1 \tag{1}$$

Proof. Let $\boldsymbol{\Delta}^*$ denote the minimizer of the RHS. Because $\boldsymbol{\Delta}^*$ is a circulation, we get

$$\boldsymbol{g}^\top\boldsymbol{\Delta}^* = \boldsymbol{g}^\top\boldsymbol{P}\boldsymbol{\Delta}^* = \sum_{e\in E} \boldsymbol{\Delta}^*_e \cdot \boldsymbol{g}^\top\boldsymbol{P}\mathbf{1}_e.$$

For the denominator we also get that

$$\|\boldsymbol{L}\boldsymbol{\Delta}^*\|_1 = \sum_{e\in E} |\boldsymbol{\Delta}^*_e| \|\boldsymbol{L}\mathbf{1}_e\|_1 \geq \frac{1}{\|\boldsymbol{M}\boldsymbol{L}^{-1}\|_{1\to 1}} \sum_{e\in E} |\boldsymbol{\Delta}^*_e| \|\boldsymbol{M}\mathbf{1}_e\|_1.$$

Now, the desired claim follows by a standard averaging argument. □

Trees. For a tree $T = (V, E)$ we denote the path from u to v as $T[u, v]$. Given an extra edge e, we denote the cycle formed by the tree and edge e as $T_{\circlearrowleft}[e]$. Additionally, we denote the vector that sends a unit amount of flow along the path and cycle as $\mathbf{1}_{T[u,v]}$ and $\mathbf{1}_{T_{\circlearrowleft}[e]}$ respectively (the latter vector sends a unit of flow along e and then from v to u).

Dynamic Trees. We frequently use dynamic trees to efficiently maintain our data structures.

Lemma 2.7 (Dynamic trees, Lemma 3.3 in [CKL+22], derived from [ST83])**.** *There is a deterministic data structure* $\mathcal{D}$ *that maintains a dynamic tree* $T \subseteq G = (V, E)$ *under insertion/deletion of edges with gradients* $\boldsymbol{g}$ *and lengths* $\boldsymbol{l}$*, and supports the following operations:*

1. *Insert/delete edges* e *to* T*, under the condition that* T *is always a tree, or update the gradient* $\boldsymbol{g}(e)$ *or lengths* $\boldsymbol{l}(e)$*. The amortized time is* $\widetilde{O}(1)$ *per change.*

2. *For a path vector* $\boldsymbol{\Delta} = \mathbf{1}_{T[u,v]}$ *for some* $u, v \in V$*, return* $\langle \boldsymbol{g}, \boldsymbol{\Delta} \rangle$ *or* $\langle \boldsymbol{\ell}, |\boldsymbol{\Delta}| \rangle$ *in time* $\widetilde{O}(1)$*.*

3. *Maintain a flow* $\boldsymbol{f} \in \mathbb{R}^E$ *under operations* $\boldsymbol{f} \leftarrow \boldsymbol{f} + \eta\boldsymbol{\Delta}$ *for* $\eta \in \mathbb{R}$ *and path vector* $\boldsymbol{\Delta} = \mathbf{1}_{T[u,v]}$*, or query the value* $\boldsymbol{f}(e)$ *in amortized time* $\widetilde{O}(1)$*.*

4. *Maintain a positive flow* $\boldsymbol{f} \in \mathbb{R}^E_{>0}$ *under operations* $\boldsymbol{f} \leftarrow \boldsymbol{f} + \eta|\boldsymbol{\Delta}|$ *for* $\eta \in \mathbb{R}_{\geq 0}$ *and path vector* $\boldsymbol{\Delta} = \mathbf{1}_{T[u,v]}$*, or or query the value* $\boldsymbol{f}(e)$ *in amortized time* $\widetilde{O}(1)$*.*

5. DETECT(). *For a fixed parameter* ϵ*, and under positive flow updates (item 4), where* $\boldsymbol{\Delta}^{(t)}$ *is the update vector at time* t*, returns*

$$S^{(t)} \stackrel{\text{def}}{=} \left\{ e \in E : \boldsymbol{l}(e) \sum_{t' \in [\mathsf{last}_e^{(t)}+1, t]} |\boldsymbol{\Delta}^{(t')}(e)| \geq \epsilon \right\} \tag{2}$$

where $\mathsf{last}_e^{(t)}$ *is the last time before* t *that* e *was returned by* DETECT()*. Runs in time* $\widetilde{O}(|S^{(t)}|)$*.*

Degree reduction. Throughout, it is often convenient to assume the graphs we work with have bounded degrees.

Fact 2.8 (Standard BST dynamic degree reduction)**.** *Consider a dynamic graph* $G = (V, E)$ *undergoing edge insertions and deletions and insertions/deletions of isolated vertices. There exists a data structure* LOWDEG() *that maintains a graph* H *and a forest* F *such that*

1. F *contains exactly one connected component for each vertex* $v \in V$*, denoted* T_v*.*

2. T_v *is a balanced binary search tree with* $\deg_G(v)$ *leaves, each corresponding uniquely to edges incident to* v*.*

3. *The graph* H *consists of the forest* F *and for each edge* $(u, v) \in E_G$ *the graph* H *contains an edge between the leaves associated with* (u, v) *in* T_u *and* T_v*.*

Using this construction, we have that

1. *The maximum degree of* H *at any time is 3.*

2. *The number updates to* H *under updates to* G *is bounded by* $O(\log m)$ *worst case per update to* G*.*

3. *When* G *contains* m *edges,* H *contains at most* $4m$ *edges.*

3 Dynamic Min-Ratio Cycle

In this section we build a deterministic data structure that allows us to toggle along approximate min-ratio cycles in a dynamic graph. It is based on the data structures developed in Section 4 and Section 5.

Flat embeddings. Because min-ratio cycles in graphs may contain up to n edges, previous works [CKL+22] represented these cycles implicitly as a few paths on a dynamically changing spanning tree of the underlying graph G. The min-ratio cycle data structures we build in this work are not as naturally "tree-based" as in previous works, and therefore we cannot maintain a spanning tree of the original graph. We instead maintain a forest satisfying the following.

Definition 2.3 (Flat embedding)**.** *We say for graphs $G = (V, E)$ and $H = (V_H, E_H)$ that a map $\Pi_{V_H \mapsto V}$ is a* flat embedding *of H into G if for every edge $e = (x, y) \in E_H$, for $x' = \Pi_{V_H \mapsto V}(x)$ and $y' = \Pi_{V_H \mapsto V}(y)$, either (x', y') is an edge in G, or $x' = y'$.*

Sometimes, we simply say that H is *flat* for brevity where we implicitly also mean that there is a vertex map $\Pi_{V(H) \mapsto V}$. An intuitive way to understand Definition 2.3 is that H flatly embeds into G if paths in H naturally correspond to (not necessarily simple) paths of the same gradient in G, and length upper bounds. This implies that for a circulation in H, the image of the circulation on G has the same or better ratio. Ultimately, our algorithm will toggle cycles on a forest F which flatly embeds into G with low vertex and edge congestion. We then use a link-cut data structure to maintain the flow on H which will be sufficient to approximately maintain the (absolute) flow added to each edge in G given that the edge congestion of the flat embedding is low, and whenever the increase is significant, we update the flow on G by explicitly summing over the flow on edges in H that embed into the same edge. We defer details of this flow maintenance procedure to Section 8.

High-level strategy. Let us now describe our data structure to find min-ratio cycles. Our data structure is built recursively. First, we show that maintaining the min-ratio cycle on a dynamic graph can be reduced to maintaining the min-ratio fundamental tree cycles for a collection of $m^{o(1)}$ dynamic trees. Then, we show that maintaining min-ratio tree cycles can be reduced to min-ratio cycles on a graph H that is roughly smaller by a factor k than the input graph G. By carefully controlling the recourse of graph H to be bounded near-linearly in the number of updates to G, we can then recurse on H.

To formalize the reduction, we first define dynamic min-ratio cycle data structures.

Definition 3.1 (Dynamic min-ratio cycle data structure)**.** *Given a dynamic graph $G = (V, E, \boldsymbol{l}, \boldsymbol{g})$, an α-approximate* dynamic min-ratio cycle *data structure $\mathcal{D}$ supports the following operation:*

- *InsertEdge(e)/DeleteEdge(e): adds/removes edge e to/from G.*
- *InsertVertex(u): adds a new isolated vertex u to V.*

Under these updates, $\mathcal{D}$ maintains a flat forest F and $\Pi_{V(F) \mapsto V}$ of G. After each update, $\mathcal{D}$ outputs a cycle $\boldsymbol{c}$ represented as some tree paths on F, specified by endpoints, and some off-tree edges such that

$$\frac{\langle \boldsymbol{g}, \boldsymbol{c} \rangle}{\|\boldsymbol{L}\boldsymbol{c}\|_1} \leq \frac{1}{\alpha} \cdot \min_{\boldsymbol{B}^\top \boldsymbol{\Delta} = 0} \frac{\langle \boldsymbol{g}, \boldsymbol{\Delta} \rangle}{\|\boldsymbol{L}\boldsymbol{\Delta}\|_1}$$

We also need to formalize dynamic min-ratio tree cycle data structures.

Definition 3.2 (Dynamic min-ratio tree cycle data structure)**.** *Given a dynamic forest $T = (V, E_T)$ with a dynamic set of off-tree edges E_{off} that do not cross between components of T (we use a dynamic graph G to denote the union of T and E_{off}) and edge lengths $\boldsymbol{l}$ and gradients $\boldsymbol{g}$, an α-approximate* dynamic min-ratio tree cycle *data structure $\mathcal{D}$ supports the following operation:*

- *InsertTreeEdge*(e)/*DeleteTreeEdge*(e)*: adds/removes edge* e *to/from* T*.*
- *InsertOffTreeEdge*(e)/*DeleteOffTreeEdge*(e)*: adds/removes edge* e *to/from* E_{off}*.*
- *InsertVertex*(u)*: adds a new isolated vertex* u *to* G*.*

Under these updates, $\mathcal{D}$ *maintains a flat forest* F *and* $\Pi_{V(F)\mapsto V}$ *of the graph* $(V, E_T \cup E_{\text{off}})$*. After each update,* $\mathcal{D}$ *outputs a cycle* $\boldsymbol{c}$ *represented as some tree paths on* F*, specified by endpoints, and some off-tree edges such that*

$$\frac{\langle \boldsymbol{g}, \boldsymbol{c} \rangle}{\|\boldsymbol{L}\boldsymbol{c}\|_1} \leq \frac{1}{\alpha} \cdot \min_{\boldsymbol{\Delta} \in \{\pm \mathbf{1}_{T_{\circlearrowleft}[e]} : e \in E_{\text{off}}\}} \frac{\langle \boldsymbol{g}, \boldsymbol{\Delta} \rangle}{\|\boldsymbol{L}\boldsymbol{\Delta}\|_1}$$

In Section 4, we show the reduction from dynamic min-ratio cycles to tree cycles. The reduction is based on dynamically maintaining the ℓ_1-oblivious routing from [RGH+22]. The oblivious routing scheme is built using *sparse neighborhood covers*, which we show how to efficiently maintain in a fully dynamic setting (Section 6). The reduction is formalized as the following theorem.

Theorem 3.3 (Reducing dynamic min-ratio cycle to dynamic min-ratio tree cycle)**.** *Consider an* α*-approximate dynamic min-ratio tree cycle data structure* $\mathcal{D}^{TC}$ *(Definition 3.2) that, on any dynamic graph* H *with at most* m *tree and non-tree edges in total, satisfies*

- *it takes* $T_{\text{init}}(m)$*-time to initialize.*
- *each update takes* $T_{\text{upd}}(m)$*-amortized time.*
- *the flat forest* F^H *it maintains has vertex congestion* γ_{vcong}*.*
- *the output cycles* $\boldsymbol{c}$ *are represented by at most* γ_{cycle} *tree paths on* F^H *as well as* γ_{cycle} *off-tree edges.*

Then, there is an $(\alpha \cdot \kappa_{HRG} \cdot \gamma_{\text{route}})$*-approximate dynamic min-ratio cycle data structure* $\mathcal{D}^{\text{MRC}}$ *(Definition 3.2), that, on any dynamic graph* G *with at most* m *edges, achieves*

- $m \cdot \gamma_{\text{tree}} + \gamma_{\text{tree}} \cdot T_{\text{init}}(m \cdot \gamma_{\text{tree}})$ *initialization time.*
- $\gamma_{\text{tree}} \cdot T_{\text{upd}}(m \cdot \gamma_{\text{tree}})$ *amortized update time.*
- *the flat forest* F^G *it maintains has vertex congestion at most* $\gamma_{\text{tree}}\gamma_{\text{vcong}}$*.*
- *the output cycles are represented by at most* γ_{cycle} *tree paths on* F^G *and non-tree edges.*

where $\kappa_{HRG} = O(\log^{1/84} m), \gamma_{\text{route}} = e^{O(\log^{83/84} m)}$ *and* $\gamma_{\text{tree}} = e^{O(\log^{83/84} m \log\log m)}$ *are the parameters in Lemma 4.32.*

Later in Section 5, we reduce the maintenance of min-ratio tree cycles to min-ratio cycles on a smaller graph. This is done via dynamically maintaining the portal routing of off-tree edges, i.e., moving off-tree edges onto a small subset of carefully chosen vertices along the tree [ST04, KPSW19, CGH+20]. After moving off-tree edges to portals, we further reduce the number of these edges using spanners. To control the number of updates propagating to the next level of recursion, we design a new dynamic spanner that has a small recourse under vertex updates (Section 7). The reduction is formalized as the following theorem.

Theorem 3.4 (Reducing dynamic min-ratio tree cycle to smaller min-ratio cycle)**.** *Consider an α-approximate dynamic min-ratio cycle data structure $\mathcal{D}^{\mathrm{MRC}}$ (Definition 3.1) that, on any dynamic graph H with at most m edges, satisfies:*

- *It takes $T_{\mathrm{init}}(m)$-time to initialize.*
- *Each update takes $T_{\mathrm{upd}}(m)$-amortized time.*
- *The flat forest F^H it maintains has vertex congestion γ_{vcong}.*
- *The output cycles $\boldsymbol{c}$ are represented by γ_{cycle} tree paths on F^H and off-tree edges.*

Then, for any size reduction parameter k, there is a $(\alpha \cdot \gamma_{\mathrm{spanner}})$-approximate dynamic min-ratio tree cycle data structure $\mathcal{D}^{\mathrm{TC}}$ (Definition 3.2), where $\gamma_{\mathrm{spanner}} = e^{O(\log^{20/21} m \log\log m)}$ is the parameter given in Theorem 5.13, that, on any dynamic graph G with at most m total tree and non-tree edges, achieves

- *$m\gamma_{\mathrm{spanner}} + T_{\mathrm{init}}(\gamma_{\mathrm{spanner}} \cdot m/k)$ initialization time.*
- *$\gamma_{\mathrm{spanner}} \cdot \left(k^2\gamma_{\mathrm{vcong}} + \frac{T_{\mathrm{init}}(\gamma_{\mathrm{spanner}} \cdot m/k)}{m/k} + T_{\mathrm{upd}}(\gamma_{\mathrm{spanner}} \cdot m/k)\right)$ amortized update time.*
- *the flat embedding forest F^G it maintains has vertex congestion at most $2\gamma_{\mathrm{vcong}}$.*
- *the output cycles are represented by at most $\max\{\gamma_{\mathrm{cycle}}, \gamma_{\mathrm{spanner}}\}$ tree paths on F^G and non-tree edges.*

Our main result in this section is a deterministic dynamic min-ratio cycle data structure with $m^{o(1)}$-amortized update time. The data structure is based on the two reductions stated previously, Theorem 3.3 and Theorem 3.4. Ultimately, alternating the use of them allows us to recurse on a new dynamic graph problem with input size reduced by a factor of roughly k. Setting k properly, which is $e^{O(\log^{167/168} m \log\log m)}$ in our case, gives the desired result.

Theorem 3.5. *For some $\gamma_{\mathrm{MRC}} = e^{O(\log^{167/168} m \log\log m)}$, there is a γ_{MRC}-approximate dynamic min-ratio cycle data structure $\mathcal{D}$ that, on any dynamic graph G with at most m edges, satisfies*

- *initialization takes $m \cdot \gamma_{\mathrm{MRC}}$-time.*
- *each update takes amortized γ_{MRC}-time.*
- *the flat forest F^G it maintains has vertex and edge congestion at most γ_{MRC}.*
- *after each update, $\mathcal{D}$ outputs a γ_{MRC}-approximate min-raito cycle represented as γ_{MRC} tree paths on F^G specified by endpoints and γ_{MRC} off-tree edges.*

Proof. We build the data structure recursively using Theorem 3.3 and Theorem 3.4 with a reduction parameter k set to be $e^{O(\log^{167/168} m \log\log m)}$. We stop the recursion until the input graph has at most k edges, where the problem can be solved to a 2-approximation in $\widetilde{O}(k^2)$ time.

Let T_{init} and T_{upd} be our data structure's initialization and amortized update time. Theorem 3.3 and Theorem 3.4 yield

$$\begin{aligned}
T_{\text{init}}(m) &\leq m\gamma_{\text{tree}} + \gamma_{\text{tree}} T_{\text{init}}\left(\frac{\gamma_{\text{tree}}\gamma_{\text{spanner}} m}{k}\right) \\
&\leq m e^{O(\log^{83/84} m \log\log m)} + e^{O(\log^{83/84} m \log\log m)} \cdot T_{\text{init}}\left(\frac{e^{O(\log^{83/84} m \log\log m)} \cdot m}{k}\right) \\
&\leq m e^{O(\log^{83/84} m \log\log m)} + e^{O(\log^{83/84} m \log\log m)} \cdot T_{\text{init}}\left(\frac{m}{e^{O(\log^{167/168} m \log\log m)}}\right) \\
&\leq m e^{O(\log^{83/84} m \log\log m)}
\end{aligned}$$

by our choice of k. This also shows that the number of recursion levels is $d = O(\log^{1/168} m)$.

To bound the amortized update time using Theorem 3.4, we need to know the vertex congestion of the flat forest maintained by our recursive data structure. We notice that after a level of Theorem 3.3 and Theorem 3.4, the vertex congestion increased by a factor of $2\gamma_{\text{tree}} = e^{O(\log^{83/84} m \log\log m)}$. Therefore, the final vertex congestion after d levels is at most

$$\gamma_{\text{vcong}} \leq (2\gamma_{\text{tree}})^d = e^{O(\log^{167/168} m \log\log m)}$$

We can also bound the amortized update time as follows:

$$\begin{aligned}
T_{\text{upd}}(m) &\leq \gamma_{\text{tree}}\gamma_{\text{spanner}}\left(k^2\gamma_{\text{vcong}} + \frac{k}{m} T_{\text{init}}\left(\frac{\gamma_{\text{tree}}\gamma_{\text{spanner}} m}{k}\right) + T_{\text{upd}}\left(\frac{\gamma_{\text{tree}}\gamma_{\text{spanner}} m}{k}\right)\right) \\
&\leq k^2\gamma_{\text{vcong}} + (\gamma_{\text{tree}}\gamma_{\text{spanner}})^2 k + \gamma_{\text{tree}}\gamma_{\text{spanner}} T_{\text{upd}}\left(\frac{\gamma_{\text{tree}}\gamma_{\text{spanner}} m}{k}\right) \\
&\leq e^{O(\log^{167/168} m \log\log m)} + e^{O(\log^{83/84} m \log\log m)} \cdot T_{\text{upd}}\left(\frac{m}{e^{O(\log^{167/168} m \log\log m)}}\right) \\
&\leq e^{O(\log^{167/168} m \log\log m)}
\end{aligned}$$

Now, we bound the approximation ratio of the output. Each level of recursion, which includes one reduction of Theorem 3.3 and Theorem 3.4, increases the ratio by a factor of $\kappa_{\text{HRG}}\gamma_{\text{route}}\gamma_{\text{spanner}} \leq e^{O(\log^{83/84} m)}$. After $d = O(\log^{1/168} m)$ levels of recursion, the approximation ratio becomes

$$e^{O(d \cdot \log^{83/84} m)} = e^{O(\log^{167/168} m)}$$

The cycle output by the data structure can be represented as γ_{spanner} tree paths on the final flat forest F^G and γ_{spanner} off-tree paths. This concludes the guarantees of our dynamic min-ratio cycle data structure $\mathcal{D}$ with our choice of $\gamma_{MRC} = e^{O(\log^{167/168} m \log\log m)}$. □

To use the min-ratio data structure in the IPM framework, we need additional data structures that maintain a current (flow) solution, augment it with the output cycle, and detect large changes after augmentations. Fortunately, since the cycle is compactly supported on a flat forest maintained by the min-ratio cycle data structure, we can leverage powerful dynamic tree data structures such as link-cut trees (Lemma 2.7).

Theorem 3.6 (Flow maintenance)**.** *There is a data structure that given a dynamic graph $G = (V, E, \boldsymbol{l}, \boldsymbol{g}, \boldsymbol{c})$ and forest F^G with a flat embedding from F^G to G with edges congestion γ_{MRC} and some parameter $\epsilon > 0$ stores some implicit flow vector $\boldsymbol{f}^{\text{im}}$ and an explicit flow vector $\boldsymbol{f}^{\text{ex}}$ and receives updates of the form*

- *ApplyCycle(E_C, μ): Receives a cycle C represented by the off-tree edges E_C and tree paths in F^G. Sends μ flow along the formed tree paths in F^G by updating $\boldsymbol{f}^{\text{im}}$, and returns $(\boldsymbol{f}^{\text{im}} + \boldsymbol{f}^{\text{ex}})^\top \mathbf{1}_C$ as well as a set of edges $E' \subseteq E$ for which $\boldsymbol{f}^{\text{ex}}(E') \leftarrow \boldsymbol{f}^{\text{ex}}(E') + \boldsymbol{f}^{\text{im}}(E')$ and $\boldsymbol{f}^{\text{im}}(E') \leftarrow \mathbf{0}$.*
- *Whenever an edge e in F^G is updated, it identifies the edge e' in G it maps to and sets $\boldsymbol{f}^{\text{ex}}(e') \leftarrow \boldsymbol{f}^{\text{ex}}(e') + \boldsymbol{f}^{\text{im}}(e')$ and $\boldsymbol{f}^{\text{im}}(e') \leftarrow \mathbf{0}$.*

It maintains that $|\boldsymbol{f}^{\text{im}}(e)\boldsymbol{l}(e)| \leq \epsilon |\boldsymbol{f}^{\text{ex}}(e)|$ and that the total number of edges returned after t updates to F_G and total flow weight sent W is $W/\epsilon + t$ where the weight of a flow is the sum over the weight of its edges which is obtained by multiplying the amount of flow with the length of the edge. The runtime after t operations is at most $\widetilde{O}(t\gamma_{\text{MRC}} + r\gamma_{\text{MRC}})$ where r is the total number of returned edges.

Proof. Follows from maintaining the implicit flows on link-cut trees as in Lemma 2.7. □

Next, we combine the data structures presented in this section. We obtain a solver data structure, which allows us to solve (incremental) min-cost flow. We first define the data structure, and then state our result.

Definition 3.7 (Solver)**.** *We call a data structure $\mathcal{D} = \textsc{Solver}(G, \boldsymbol{l}, \boldsymbol{g}, \boldsymbol{c}, \boldsymbol{f}, q, \Gamma, \epsilon)$ that is initialized with*

- *a graph $G = (V, E)$ and*
- *lengths $\boldsymbol{l} \in \mathbb{R}^E_{\geq 0}$, gradients $\boldsymbol{g} \in \mathbb{R}^E$, costs $\boldsymbol{c} \in \mathbb{R}^E$, a flow $\boldsymbol{f} \in \mathbb{R}^E$ routing demand $\boldsymbol{d}$, and*
- *a quality parameter $q > 0$, a step-size parameter $\Gamma > 0$ and a accuracy parameter $\epsilon > 0$.*

a γ_{approx} min-ratio cycle solver if it (implicitly) maintains a flow vector $\boldsymbol{f}$ such that $\boldsymbol{f}$ routes demand $\boldsymbol{d}$ throughout and supports the following operations.

- *ApplyCycle(): One of the following happens.*
  - *Either the data structure finds a circulation $\boldsymbol{\Delta} \in \mathbb{R}^E$ such that $\boldsymbol{g}^\top \boldsymbol{\Delta} / \|\boldsymbol{L}\boldsymbol{\Delta}\|_1 \leq -q$ and $|\boldsymbol{g}^\top \boldsymbol{\Delta}| = \Gamma$. In that case it updates $\boldsymbol{f} \leftarrow \boldsymbol{f} + \boldsymbol{\Delta}$ and it returns a set of edges $E' \subseteq E$ alongside the maintained flow values $\boldsymbol{f}(e')$ for $e' \in E'$.*

    For every edge e, between the times that it is in E' during calls to ApplyCycle(), the value of $\boldsymbol{l}(e)\boldsymbol{f}(e)$ does not change by more than ϵ.
  - *Or it certifies that there is no circulation $\boldsymbol{\Delta} \in \mathbb{R}^E$ such that $\boldsymbol{g}^\top \boldsymbol{\Delta} / \|\boldsymbol{L}\boldsymbol{\Delta}\|_1 \leq -q/\gamma_{\text{approx}}$ for some parameter γ_{approx}.*
- *UpdateEdge(e, l, g): Updates the length and gradient of an edge e that was returned by the last call to ApplyCyle().*

- *InsertEdge*(e, l, g, c)*: Adds edge* e *to* G *with length* l*, gradient* g*, cost* c*. The flow* $\boldsymbol{f}(e)$ *is intialized to* 0*.*
- *ReturnCost*()*: Returns the flow cost* $\boldsymbol{c}^\top \boldsymbol{f}$*.*
- *ReturnFlow*()*: Explicitly returns the currently maintained flow* $\boldsymbol{f}$*.*

The sum of the sizes $|E'|$ *of the returned sets by* t *calls to ApplyCycle*() *is at most* $t\Gamma/\epsilon q$*.*

Theorem 3.8. *There is a data structure* $\mathcal{D} = \textsc{Solver}(G, \boldsymbol{l}, \boldsymbol{g}, \boldsymbol{c}, \boldsymbol{f}, q, \Gamma, \epsilon)$ *as in Definition 3.7 for* $\gamma_{\text{approx}} = e^{O(\log^{167/168} m)}$*. It has the following time complexity for* $\gamma_{\text{time}} = e^{O(\log^{167/168} m \log\log m)}$ *where* m *is an upper bound on the total number of edges in* G*:*

1. *The initialization takes time* $|E|\gamma_{\text{time}}$*.*
2. *The operation ApplyCycle*() *takes amortized time* $(|E'| + 1) \cdot \gamma_{\text{time}}$ *when* $|E'|$ *edges are returned.*
3. *The operations UpdateEdge*()*/InsertEdge*()*/ReturnCost*() *take amortized time* γ_{time}*.*
4. *The operation ReturnFlow*() *takes amortized time* $|E| \cdot \gamma_{\text{time}}$*.*

Proof. Follows from Theorem 3.5 and Theorem 3.6 since the total flow weight is at most $t\Gamma/q$ after t updates. □

We finally formally define the min-cost flow problem.

Definition 3.9 (Min-cost flow)**.** *Given a directed graph* $G = (V, E)$ *with edge capacities* $\boldsymbol{u}$ *and costs* $\boldsymbol{c}$ *and a demand vector* $\boldsymbol{d} \perp \mathbf{1}$*, the min-cost flow problem seeks to find*

$$\boldsymbol{f}^\star = \operatorname*{arg\,min}_{\boldsymbol{f}:\boldsymbol{B}^\top \boldsymbol{f} = \boldsymbol{d}} \boldsymbol{c}^\top \boldsymbol{f}.$$

We then describe how the solver data structure is used to solve min-cost flow.

Theorem 3.10 (Min-cost flow interior point method)**.** *There is an algorithm that given a directed graph* $G = (V, E)$ *with polynomially bounded edge capacities* $\boldsymbol{u}$ *and costs* $\boldsymbol{c}$*, a demand vector* $\boldsymbol{d} \perp \mathbf{1}$*, and access to a* γ_{approx} *min-ratio cycle solver data structure (as defined in Definition 3.7) exactly solves the min-cost flow problem where:*

- *The number of calls to* $\mathcal{D}$*.ApplyCycle*()*/UpdateEdge*()*/InsertEdge*()*/ReturnCost*() *is bounded by* $\widetilde{O}(m\gamma_{\text{approx}}^{O(1)})$
- *The number of calls to* $\mathcal{D}$*.ReturnFlow*() *is bounded by* $\widetilde{O}(\gamma_{\text{approx}}^{O(1)})$*.*

This result follows from previous work [CKL+22] as discussed in Section 8.

Recall, we defined incremental threshold min-cost flow in Definition 1.1. Next, we formalize how the solver data structure can be used to solve the incremental threshold min-cost flow problem.

Theorem 3.11 (Incremental threshold min-cost flow interior point method)**.** *There is an algorithm that given a directed graph* $G = (V, E)$ *with polynomially bounded edge capacities* $\boldsymbol{u}$ *and costs* $\boldsymbol{c}$*, a demand vector* $\boldsymbol{d} \perp \mathbf{1}$*, and access to a* γ_{approx} *min-ratio cycle solver data structure* $\mathcal{D}$ *(as defined in Definition 3.7), exactly solves the incremental min-cost flow problem where:*

- *The number of calls to $\mathcal{D}$.ApplyCycle()/UpdateEdge()/InsertEdge()/ReturnCost() is bounded by $\widetilde{O}(m\gamma_{\text{approx}}^{O(1)})$*
- *The number of calls to $\mathcal{D}$.ReturnFlow() is bounded by $\widetilde{O}(\gamma_{\text{approx}}^{O(1)})$.*

This result also follows from previous work [BLS23] and is discussed in Section 8.

Theorem 3.11 and Theorem 3.8 together imply our main result Theorem 1.4, an almost-linear time algorithm for threshold min-cost flow.

4 Hierarchical Routing Graphs, Trees and Min-Ratio Cycles

In this section, we reduce the dynamic min-ratio cycle problem (Definition 3.1) to $m^{o(1)}$ dynamic min-ratio tree cycle problems (Definition 3.2). That is, we show that one can maintain $m^{o(1)}$ trees such that the min-ratio tree cycle on these trees is a $m^{o(1)}$-approximate min-ratio cycle on the given graph. We achieve the following result.

Theorem 3.3 (Reducing dynamic min-ratio cycle to dynamic min-ratio tree cycle)**.** *Consider an α-approximate dynamic min-ratio tree cycle data structure $\mathcal{D}^{TC}$ (Definition 3.2) that, on any dynamic graph H with at most m tree and non-tree edges in total, satisfies*

- *it takes $T_{\text{init}}(m)$-time to initialize.*
- *each update takes $T_{\text{upd}}(m)$-amortized time.*
- *the flat forest F^H it maintains has vertex congestion γ_{vcong}.*
- *the output cycles $\boldsymbol{c}$ are represented by at most γ_{cycle} tree paths on F^H as well as γ_{cycle} off-tree edges.*

Then, there is an $(\alpha \cdot \kappa_{HRG} \cdot \gamma_{\text{route}})$-approximate dynamic min-ratio cycle data structure $\mathcal{D}^{\text{MRC}}$ (Definition 3.2), that, on any dynamic graph G with at most m edges, achieves

- *$m \cdot \gamma_{\text{tree}} + \gamma_{\text{tree}} \cdot T_{\text{init}}(m \cdot \gamma_{\text{tree}})$ initialization time.*
- *$\gamma_{\text{tree}} \cdot T_{\text{upd}}(m \cdot \gamma_{\text{tree}})$ amortized update time.*
- *the flat forest F^G it maintains has vertex congestion at most $\gamma_{\text{tree}}\gamma_{\text{vcong}}$.*
- *the output cycles are represented by at most γ_{cycle} tree paths on F^G and non-tree edges.*

where $\kappa_{HRG} = O(\log^{1/84} m), \gamma_{\text{route}} = e^{O(\log^{83/84} m)}$ and $\gamma_{\text{tree}} = e^{O(\log^{83/84} m \log\log m)}$ are the parameters in Lemma 4.32.

4.1 Sparse Neighborhood Covers and Hierarchical Routing Graphs

The first step towards constructing the flat forest over G is based on the ℓ_1 oblivious routing of [RGH+22], which gives a way of extracting an oblivious routing from a collection of sparse neighborhood covers, one at each distance scale. Thus, the first piece that our data structure requires is an algorithm for maintaining a sparse neighborhood cover in a fully dynamic graph, which we show in Section 6.

Theorem 4.1 (Fully-dynamic sparse neighborhood cover)**.** *Given an m-edge constant-degree input graph $G = (V, E, l)$ with polynomially-bounded lengths in $[1, L]$ and a diameter parameter $D \geq 1$ there is a data structure that supports a polynomially bounded number of updates of the following type:*

- *InsertEdge(e)/DeleteEdge(e): inserts/deletes edge e to/from G where insertions preserve that G has constant degree.*

Under these updates, the data structure maintains a forest F, map $\Pi_{V(F)\mapsto V}$, and a subset $S \subseteq V(F)$ that satisfy the following properties, for some $\gamma_{SNC} = e^{O(\log^{41/42} m \log\log m)}$:

1. *F with map $\Pi_{V(F)\mapsto V}$ is a flat embedding of G.*
2. *For any vertex $v \in V$ there is a tree $T \in F$ such that*
$$B_G(v, D/\gamma_{SNC}) \subseteq \Pi_{V(F)\mapsto V}(S \cap V(T)).$$
3. *For any tree $T \in F$ we have that $\mathrm{diam}_F(S \cap V(T)) \leq \gamma_{SNC} \cdot D$.*
4. *The congestion satisfies $\mathrm{vcong}(\Pi_{V(F)\mapsto V}) \leq \gamma_{SNC}$.*

The forest F, map $\Pi_{V(F)\mapsto V}$, and subset $S \subseteq V(F)$ are all maintained explicitly, and each undergoes at most γ_{SNC} amortized changes per update. The algorithm is deterministic, initializes in time $m \cdot \gamma_{SNC}$, and has amortized update time γ_{SNC}.

The reason for the set S is that in our construction, the forest F and trees $T \in F$ contain several extraneous vertices where we cannot control the diameter well. S is the set of "real vertices" which correspond to images of $v \in V(G)$ where we can control the diameter. This way, the sets $\Pi_{V(F)\mapsto V}(S \cap V(T))$ correspond to the clusters in a sparse neighborhood cover, and the forest F represents an approximate low-diameter tree connecting the vertices. As a subtle point, our data structure is only guaranteed to maintain S explicitly, and *not* each vertex in $S \cap V(T)$. This is because the algorithm for Theorem 4.1 requires detaching (potentially large) subtrees of F and reattaching them to other subtrees.

Given this data structure, we are able to maintain an object, which we call a *hierarchical routing graph* (HRG), that supports the oblivious routing of [RGH+22].

Definition 4.2 (Hierarchical routing graph)**.** *Given a graph $G = (V, E, \boldsymbol{l}, \boldsymbol{g})$ with polynomially bounded lengths $\boldsymbol{l}$ in $[1, L = m^{O(1)}]$, a hierarchical routing graph (HRG) H of G depends on parameters γ_{HRG}, γ_{diam}, $\kappa_{HRG} \geq 3$ and $\gamma_{HRG} \leq n$, so that $\gamma_{diam}^{\kappa_{HRG}} \geq n^2 L$. Consider the following objects:*

- *a collection of vertex sets $\mathcal{V} = \{V_1, \ldots V_{\kappa_{HRG}}\}$ with bijective mapping between each V_i and V denoted by $\Pi_{V_i\mapsto V}$ for $\kappa_{HRG} = O(\log^{1/84} m)$,*
- *a collection of dynamic forests $\mathcal{F} = (F_1, \ldots, F_{\kappa_{HRG}-1})$, called* routing forests*, maps $\Pi_{V(F_i)\mapsto V}$, and subsets $S_i \subseteq V(F_i)$,*
- *a collection of edge sets $\mathcal{E} = \mathcal{E}^{\text{out}} \cup \mathcal{E}^{\text{in}}$ for $\mathcal{E}^{\text{out}} := E_1^{out} \cup \cdots \cup E_{\kappa_{HRG}-1}^{out}$ and $\mathcal{E}^{\text{in}} := E_2^{in} \cup \cdots \cup E_{\kappa_{HRG}}^{in}$ called linking edges and*

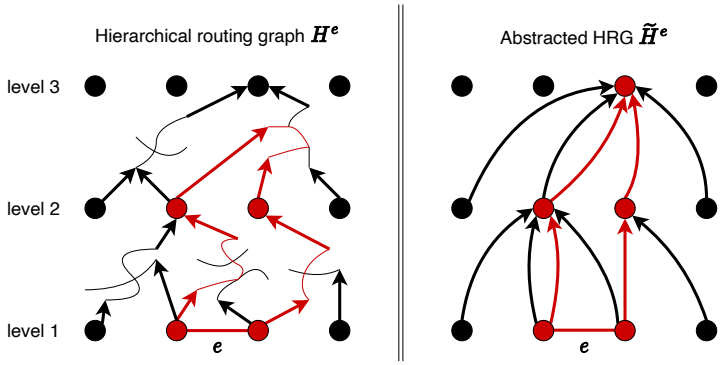

Figure 1: A hierarchical routing graph H^e (Definition 4.2) and its abstracted HRG $\widetilde{H}^e$ (Definition 4.3) alongside a monotone cycle circulation (Definition 4.21) on $\widetilde{H}^e$ and its mapping on H^e. Notice that every tree is only attached to a single vertex on the next level.

- *lengths and gradients* $\boldsymbol{l}_H$ *and* $\boldsymbol{g}_H$ *of all edges in* $\mathcal{E}$

These objects satisfy the following properties:

1. *Congestion: The vertex congestion* $\mathrm{vcong}(\Pi_{V(F_i)\mapsto V}) \le \gamma_{HRG}$ *for all* $i \in [\kappa_{HRG} - 1]$,
2. *Routing trees are flat: For all* $i \in [\kappa_{HRG} - 1]$, F_i *and* $\Pi_{V(F_i)\mapsto V}$ *are a flat embedding of* G.
3. *Routing tree diameter: For all* $T \in F_i$, *we have* $\mathrm{diam}(S_i \cap T) \le \gamma_{HRG} \cdot \gamma^i_{diam}$,
4. *Linking edge consistency: Every out-edge* $e \in \mathcal{E}^{\mathrm{out}}$ *is between a vertex* $v \in V_i$ *and* $u \in S_i$, *such that* $\Pi_{V_i\mapsto V}(v) = \Pi_{V(F_i)\mapsto V}(u)$. *Every in-edge* $e \in \mathcal{E}^{\mathrm{in}}$ *is between a vertex* $u \in S_{i-1}$ *and* $v \in V_i$ *with* $\Pi_{V(F_{i-1})\mapsto V}(u) = \Pi_{V_i\mapsto V}(v)$. *The length of edges* $e \in E_i^{\mathrm{out}}$ *for* $i = 1, \ldots, \kappa_{HRG} - 1$ *is* $\boldsymbol{l}_H(e) := \gamma_{HRG} \cdot \gamma^i_{diam}$, *and gradient is* $\boldsymbol{g}_H(e) = 0$. *The length of edges* $e \in E_i^{\mathrm{in}}$ *for* $i = 2, \ldots, \kappa_{HRG}$ *is* $\boldsymbol{l}_H(e) := \gamma_{HRG} \cdot \gamma^{i-1}_{diam}$, *and gradient is* $\boldsymbol{g}_H(e) = 0$.
5. *Outdegree: Every vertex* $v \in V_i$ *is adjacent to at most* γ_{HRG} *vertices in* F_i.
6. *Covering: For all* $v \in V$, *we have* $B_G(v, \gamma^i_{diam}/\gamma_{HRG}) \subseteq \Pi_{V(F_i)\mapsto V}(S_i \cap T)$ *for some* $T \in F_i$.

The vertex set of H *consists of all* V_i *and all* $V(F_i)$. *The edge set consists of all edges in all the* F_i, *and the linking edges. Given an additional edge* $e = (u, v) \in E$ *with length* $\boldsymbol{l}(e)$ *and gradient* $\boldsymbol{g}(e)$ *between two vertices* $u, v \in V$, *we let* H^e *be the hierarchical routing graph* H *with extra edge* e' *between the two unique vertices* $u', v' \in V_1$ *that map to* u *and* v *respectively, and we set* $\boldsymbol{l}_H(e') = \boldsymbol{l}(e)$ *and* $\boldsymbol{g}_H(e') = \boldsymbol{l}(e)$. *For convenience, we refer to* e' *as* e *when the context is clear.*

Critically, note that H flatly embeds into G, by the map $\Pi_{V(H)\mapsto V}$ which combines the maps $\Pi_{V(F_i)\mapsto V}$ and $\Pi_{V_i\mapsto V}$. Note that the linking edges $e \in \mathcal{E}$ have endpoints with the same image, and hence still satisfy Definition 2.3.

Next, we define an abstraction of the hierarchical routing graph H that we never construct, but plays a crucial role in our analysis. It represents paths from $v \in V_i$ to $v \in V_{i+1}$ in H as edges of length $\gamma_{\mathrm{HRG}}\gamma^i_{\mathrm{diam}}$, which is (up to constants) the length of the underlying path in H by properties 3 and 4 in Definition 4.2.

Definition 4.3 (Abstracted HRG)**.** *Given a hierarchical routing graph H of a graph $G = (V, E, \boldsymbol{l}, \boldsymbol{g})$, we define the abstracted HRG of H as $\widetilde{H} = (\mathcal{V}, \widetilde{\mathcal{E}}, \boldsymbol{l}_{\widetilde{H}}, \boldsymbol{g}_{\widetilde{H}})$, where the vertex set $\mathcal{V} = V_1 \cup V_2 \cup \cdots \cup V_{\kappa_{HRG}}$, where V_i are defined as in Definition 4.2. The edge set $\widetilde{\mathcal{E}}$ and the lengths and gradients are defined as follows.*

- *Edges: For every pair of edges (u, x) and (y, v) in $E(H)$ so that $u \in V_i$, $v \in V_{i+1}$ and x, y are in the same tree $T \in F_i$ the edge set $\widetilde{\mathcal{E}}$ contains an edge $e = (u, v)$. We let $\Pi_{\widetilde{H} \mapsto H}(e)$ map the edge e to the simple path between u and v given by concatenating the edge (u, x) the path $T[x, y]$ and the edge (y, v)*
- *Lengths: Edge e as in the first bullet point receives length $\boldsymbol{l}_{\widetilde{H}} = \gamma_{HRG} \cdot \gamma_{diam}^{i}$.*
- *Gradients: Edge e as in the first bullet point receives gradient $\boldsymbol{g}_{\widetilde{H}} = \boldsymbol{g}_H(\Pi_{\widetilde{H} \mapsto H}(e))$.*

Given an additional edge $e = (u, v) \in E$ between two vertices $u, v \in V$ of length $\boldsymbol{l}(e)$ and gradient $\boldsymbol{g}(e)$ we let $\widetilde{H}^e$ be the abstracted HRG $\widetilde{H}^e$ with an extra edge e' between the two unique vertices $u', v' \in V_1$ that map to u and v respectively, and we set $\boldsymbol{l}_{\widetilde{H}}(e') = \boldsymbol{l}(e)$ and $\boldsymbol{g}_{\widetilde{H}}(e') = \boldsymbol{g}(e)$. For convenience, we refer to e' as e when the context is clear. Similarly, we denote $\widetilde{H}$ after the addition of multiple extra edges $\overline{E} \subseteq E$ as $\widetilde{H}^{\overline{E}}$ with respective lengths and gradients.

Remark 4.4. *See Figure 1 for an example of a hierarchical routing graph H and its abstracted HRG $\widetilde{H}$. The example illustrates that multi-edges may arise in $\widetilde{H}$ even if H is a simple graph.*

Later, we will show that there is an oblivious routing (based on [RGH+22]) which has the following form: for every vertex v, demand from v is sent from its preimage in V_1 upwards on paths in any HRG that we construct.

4.2 Dynamic Hierarchical Routing Graphs

In this section, we show how we dynamically maintain a hierarchical routing graph H using sparse neighborhood covers from Theorem 4.1 of increasing radius.

Dynamic hierarchical routing graph. We state the main result of this section.

Theorem 4.5 (Dynamic HRG)**.** *Given an m-edge constant-degree input graph $G = (V, E, \boldsymbol{l}, \boldsymbol{g})$ with polynomially-bounded lengths in $[1, L]$, there is a data structure that supports a polynomially bounded number of updates of the following type:*

- *InsertEdge(e)/DeleteEdge(e): inserts/deletes edge e to/from G where insertions preserve that G has constant degree.*

Under these updates, the data structure DynamicHRG maintains a hierarchical routing graph of G with parameters $\gamma_{HRG} = e^{O(\log^{41/42} m \log\log m)}$, $\gamma_{diam} = e^{O(\log^{83/84} m \log\log m)}$, and $\kappa_{HRG} = \log^{1/84} m$. The algorithm is deterministic, can be initialized in time $O(m\gamma_{HRG})$, and processes each update in amortized time γ_{HRG}.

Algorithm. We first describe the data structure DynamicHRG for maintaining a hierarchical routing graph. The algorithm follows the construction of the static ℓ_1-oblivious routing of [RGH+22], but omits the computation of the distribution over the routing paths. The conditions of Definition 4.2 will nearly be satisfied trivially given the construction.

Given a constant-degree dynamic graph $G = (V, E, \boldsymbol{l}, \boldsymbol{g})$, we describe how our data structure maintains all the individual pieces of a hierarchical routing graph.

1. Vertex sets $\mathcal{V}$: Copies $V_1, \dots V_{\kappa_{\text{HRG}}}$ of the vertex set V alongside bijective mappings $\Pi_{V_i \mapsto V}$ for all $i = 1, \dots, \kappa_{\text{HRG}}$.

2. Forests $\mathcal{F}$: Dynamic SNC data structures $\mathcal{N}_1, \dots, \mathcal{N}_{\kappa_{\text{HRG}}-1}$ (Theorem 4.1) operating on G, where $\mathcal{N}_i$ has diameter parameter $D_i := \gamma^i_{\text{diam}}$. F_i and S_i in Definition 4.2 is the forest and set maintained by $\mathcal{N}_i$.

3. Linking edges $\mathcal{E}$: We first describe the sets E_i^{out} for $i = 1, \dots, \kappa_{\text{HRG}} - 1$, and then the sets E_i^{in} for $i = 2, \dots, \kappa_{\text{HRG}}$.

   - For every vertex pair $v \in V_i$ and $v' \in S_i$ that map to the same vertex in V, i.e., $\Pi_{V_i \mapsto V}(v) = \Pi_{V(F_i) \mapsto V}(v')$ we have an edge (v, v') in E_i^{out}.
   - For every $T \in F_{i-1}$, we add an edge to E_i^{in} between an arbitrary vertex $v \in S_{i-1} \cap T$ and its copy in V_i, i.e, the unique vertex $v' \in V_i$ so that $\Pi_{V_i \mapsto V}(v') = \Pi_{V(F_{i-1}) \mapsto V}(v)$.

4. Lengths and gradients: We define the lengths $\boldsymbol{l}_H$ and gradients $\boldsymbol{g}_H$ of H as in Definition 4.2. In particular, edges $e \in E(F_i)$ have the same gradient and longer lengths as their preimage in G, because F_i flatly embeds into G. Lengths of linking edges $e \in \mathcal{E}$ are described in Definition 4.2.

The algorithm then reacts to changes in G by passing them to all the data structures $\mathcal{N}_i$. Since the forests F_i and sets S_i are maintained explicitly, the in-edges are maintained directly. To maintain the out-edges, we use a link-cut tree to detect some vertex in S in each connected component of F_i to add an out-edge from (see for example [AHLT05] on how to find a vertex from a specified set in a tree of a dynamic forest).

Correctness. Next, we show that the various parameters in the hierarchical routing graph are bounded as claimed in Definition 4.2.

Lemma 4.6. *Our algorithm maintains a hierarchical routing graph H as in Definition 4.2.*

Proof. We check the properties of Definition 4.2 one at a time.

- Property 1 (congestion) follows from property 4 of Theorem 4.1 and the fact that each F_i is maintained by $\mathcal{N}_i$.
- Property 2 (flat routing trees) follows from property 1 of Theorem 4.1.
- Property 3 (routing tree diameter) follows from property 3 of Theorem 4.1.
- Property 4 (linking edges) follows from our construction of linking edges.

- Property 5 (out-degree) follows from property 4 of Theorem 4.1 and the way we construct E_i^{out}.
- Property 6 (covering) follows from property 2 of Theorem 4.1.

□

Runtime analysis. We conclude with the runtime analysis of our dynamic HRG algorithm.

Lemma 4.7. *Initializing a dynamic hierarchical routing graph H on a graph with m edges takes time $\widetilde{O}(\kappa_{HRG} \cdot \gamma_{SNC} \cdot m) = e^{O(\log^{41/42} m \log\log m)} \cdot m$. Afterwards, each update can be processed in amortized time $\widetilde{O}(\kappa_{HRG} \cdot \gamma_{SNC}^2) = e^{O(\log^{41/42} m \log\log m)}$.*

Proof. The run-time is dominated by the cost of maintaining the forests F_i and sets S_i since each change to F_i or S_i causes only a constant number of easily computable changes in $\mathcal{E}$. Therefore, the claimed runtime bound follows from Theorem 4.1, and an overhead of $O(\log n)$ for using link-cut trees to detect adding out-edges. □

4.3 Routings and Min-Ratio Circulations

In this section, we formally show that the edges in an HRG support an oblivious routing with competitive ratio at most $m^{o(1)}$. To do so, we walk through the construction of the ℓ_1 oblivious routing of [RGH+22] and analyze its competitive ratio in our setting. We will primarily work with the abstracted hierarchical routing graph $\widetilde{H}$ (Definition 4.3) in this section. We first define monotone paths on the $\widetilde{H}$, which are the (compressed) paths upwards or downwards in the HRG.

Definition 4.8 (Monotone path)**.** *We call a path P on $\widetilde{H}$ as in Definition 4.3 monotonically increasing (decreasing) if the layer index i of the vertices it visits is strictly increasing (decreasing).*

We next state a theorem about a suitable notion of ℓ_1 oblivious routings on G represented by $\widetilde{H}$. These are restricted to route along monotone paths.

Theorem 4.9. *Given graph $G = (V, E, \boldsymbol{l}, \boldsymbol{g})$ with edge-vertex incidence matrix $\boldsymbol{B}$ and a hierarchical routing graph H with abstraction $\widetilde{H}$ of G the following hold:*

1. *There is a set of monotonically increasing paths $\mathcal{P}$ in $\widetilde{H}$ that start at vertices in V_1 and end in $V_{\kappa_{HRG}-1}$, and a matrix $\boldsymbol{\Pi}_{V \mapsto \mathcal{P}} \in \mathbb{R}^{|\mathcal{P}| \times |V|}$ mapping each vertex in $v \in V$ to a distribution of paths in $\mathcal{P}$ starting from its copy in V_1 such that all paths with nonzero weight starting from a vertex v in connected component A of the graph G end at the same vertex r_A.*
2. *There is a matrix $\boldsymbol{\Pi}_{P \mapsto \widetilde{\mathcal{E}}} \in \mathbb{R}^{|\widetilde{\mathcal{E}}| \times |\mathcal{P}|}$ mapping paths to their edges.*
3. *For $\gamma_{\text{route}} := O(\kappa_{HRG} \gamma_{HRG}^6 \gamma_{diam})$, we have $\left\| \widetilde{\boldsymbol{L}} \widetilde{\boldsymbol{P}} \boldsymbol{L}^{-1} \right\|_{1 \to 1} \leq \gamma_{\text{route}}$ for*

$$\widetilde{\boldsymbol{P}} := \boldsymbol{\Pi}_{E \mapsto (V_1, V_1)} - \boldsymbol{\Pi}_{\mathcal{P} \mapsto \widetilde{\mathcal{E}}} \boldsymbol{\Pi}_{V \mapsto \mathcal{P}} \boldsymbol{B}^\top$$

and $\widetilde{\boldsymbol{L}} = \operatorname{diag}(\boldsymbol{l}_{\widetilde{H}^E})$ where $\boldsymbol{\Pi}_{E \mapsto (V_1, V_1)}$ maps edges $(u, v) \in E$ to edges between (u', v') where $u = \Pi_{V_1 \to V}(u')$ and $v = \Pi_{V_1 \to V}(v')$ (recall that $\Pi_{V_1 \to V}$ is a bijection).

We finally define the cycle projection matrix $\boldsymbol{P} := \boldsymbol{I} - \boldsymbol{\Pi}_{\widetilde{H}\mapsto H}\boldsymbol{\Pi}_{\mathcal{P}_{\widetilde{\mathcal{E}}}}\boldsymbol{\Pi}_{V\mapsto\mathcal{P}}\boldsymbol{B}^\top$*, where* $\boldsymbol{\Pi}_{\widetilde{H}\mapsto H}$ *is the matrix corresponding to the map* $\Pi_{\widetilde{H}\mapsto H}$ *which sends edges in* $\widetilde{H}$ *to paths in* H*.*

Before we prove Theorem 4.9, we state a corollary that makes the relation to min-ratio circulations explicit.

Corollary 4.10 (HRG circulation)**.** *Given a graph* $G = (V, E, \boldsymbol{l}, \boldsymbol{g})$ *with edge vertex incidence matrix* $\boldsymbol{B}$ *and a HRG* H *with abstraction* $\widetilde{H}$ *of* G *we have*

$$\min_{e\in|E|} \boldsymbol{g}_H^\top \boldsymbol{P}\mathbf{1}_e / \left\|\widetilde{\boldsymbol{L}}\widetilde{\boldsymbol{P}}\mathbf{1}_e\right\|_1 \leq \frac{1}{\gamma_{\text{route}}} \min_{\boldsymbol{\Delta}:\boldsymbol{B}^\top\boldsymbol{\Delta}=\mathbf{0}} \boldsymbol{g}^\top\boldsymbol{\Delta} / \|\boldsymbol{L}\boldsymbol{\Delta}\|_1 .$$

Proof. Setting $\boldsymbol{M} = \widetilde{\boldsymbol{L}}\widetilde{\boldsymbol{P}}$ and instantiating $\boldsymbol{P}$ as in Theorem 4.9 we obtain

$$\min_{e\in|E|} \boldsymbol{g}_H^\top \boldsymbol{P}\mathbf{1}_e / \left\|\widetilde{\boldsymbol{L}}\widetilde{\boldsymbol{P}}\mathbf{1}_e\right\|_1 \leq \frac{1}{\gamma_{\text{route}}} \min_{\boldsymbol{\Delta}:\boldsymbol{B}^\top\boldsymbol{\Delta}=\mathbf{0}} \boldsymbol{g}^\top\boldsymbol{\Delta} / \|\boldsymbol{L}\boldsymbol{\Delta}\|_1$$

directly by Lemma 2.6 since $\left\|\widetilde{\boldsymbol{L}}\widetilde{\boldsymbol{P}}\boldsymbol{L}^{-1}\right\|_{1\to 1} \leq \gamma_{\text{route}}$ by Theorem 4.9. □

Proof of Theorem 4.9. We finally show that the abstracted routing graph $\widetilde{H}$ of H supports a γ_{route}-approximate ℓ_1-oblivious routing to prove Theorem 4.9. The proof closely follows section 4 of [RGH+22] with the exception that we merely use their construction to prove the existence of a supported routing instead of extracting it. We note that the achieved approximation ratio in [RGH+22] is polylogarithmic while ours is a large subpolynomial factor. This is caused by the additional overhead for maintaining sparse neighborhood covers dynamically, and that we force our HRG to have fewer layers so that we can extract $m^{o(1)}$ trees out of it later.

We first fix a hierarchical routing graph H of G. To enable our analysis we define similar notation as [RGH+22] to better interface with their results.

Definition 4.11. *Our analysis requires the definition of the following objects.*

- *Clusters: We let* $\mathcal{C}_i := \{\Pi_{V(F_i)\mapsto V}(S_i \cap T) | T \in F_i\}$ *denote the set* $S_i \cap T$ *over trees* T *after mapping them to* G*. Recall that by property 6 of Definition 4.2 every ball of radius* $\gamma^i_{diam}/\gamma_{HRG}$ *in* G *is contained in a cluster* $C \in \mathcal{C}_i$*.*
- *Roots: Recall that for every tree* $T \in F_i$ *(corresponding to a cluster* $C \in \mathcal{C}_i$*) there is an unique edge* $(u, v) \in E_i^{\text{out}}$ *with* $u \in T$*. We call* $v \in V_{i+1}$ *the* root *of cluster* C*, and denote it as* $r_C := v$*.*
- *Edges and flows: To simplify our notation, we assume an extra clustering* $\mathcal{C}_0$ *such that each vertex* $v \in V_1$ *is the root of its own singleton cluster in* $\mathcal{C}_0$*. The diameter of these clusters is* $D_0 = \gamma^0_{diam} = 1$*. Then, we let* $e_{C,C'}$ *be the edge from* $r_C \in V_i$ *to* $r_{C'} \in V_{i+1}$ *in* $\widetilde{H}$*.*[5]

In the following, we define the set of monotone paths $\mathcal{P}_v$ for every $v \in V_1$ on the abstracted routing graph $\widetilde{H}$ alongside a probability weighting p_P for each $P \in \mathcal{P}_v$. We first define $p^i_C(v)$, which is proportional to the (weighted) amount of paths in $\mathcal{P}_v$ that end at r_C for some cluster $C \in \mathcal{C}_i$.

[5] To disambiguate multi-edges, we take the one using the tree giving rise to cluster C'.

Definition 4.12. *For $C \in \mathcal{C}_i$ where $i = 1, \ldots, \kappa_{HRG} - 2$ we let*

$$p_C^{(i)}(v) := \max\left\{0, \frac{\mathrm{dist}_G(v, V \setminus C)}{D_i} - \frac{1}{4\gamma_{HRG}}\right\}$$
$$w_i(v) := \sum_{C \in \mathcal{N}_i} p_C^{(i)}(v)$$

On the last level $\kappa_{HRG} - 1$ we choose a cluster $C_A \in \mathcal{C}_{\kappa_{HRG}-1}$ per connected component A of G that contains all vertices in A. Such a cluster always exists since we choose κ_{HRG} such that $\gamma_{diam}^{\kappa_{HRG}-1} \geq \gamma_{HRG} \cdot n \cdot L$ in Definition 4.2. We give vertices $v \in A$ weight $p_{C_A}^{\kappa_{HRG}-1}(v) = 1$ and set $w_{\kappa_{HRG}-1}(v) = 1$.

Then, every vertex $v_i \in V_i$ at level i that has flow originating from vertex $v \in V$ sends a $p_C^i(v)/w_i(v)$ fraction of said flow to r_C. We show that all such vertices are contained in the cluster C. Concretely, this yields the following path set.

Definition 4.13 (Paths and weights)**.** *Each flow path $P \in \mathcal{P}_v$ is given by $v = r_{C_0}$, r_{C_1}, …, $r_{C_{\kappa_{\mathrm{HRG}}-1}} = r$ where $C_i \in \mathcal{C}_i$. The weight of P is given by*

$$p_P := \prod_{i=1}^{\kappa_{HRG}-1} \frac{p_{C_i}^{(i)}(v)}{w_i(v)}.$$

Remark 4.14. *Notice that the choice of $p_{C_A}^{(\kappa_{HRG}-1)}$ in Definition 4.12 ensures that all paths with nonzero weight end at the same vertex $r = r_{C_A}$ for all vertices in the same connected component A.*

Thus we have described the routing projection matrix $\widetilde{\boldsymbol{P}} = (\boldsymbol{\Pi}_{E \mapsto (V_1, V_1)} - \boldsymbol{\Pi}_{\mathcal{P} \mapsto \widetilde{\mathcal{E}}} \boldsymbol{\Pi}_{V \mapsto \mathcal{P}}) \boldsymbol{B}^\top$ since we gave a mapping from vertices (in V_1 which is a copy of V) to distributions of paths and from paths to edges in $\widetilde{H}$. We first define the flow routed by a unit demand following [RGH+22].

Definition 4.15. *We let $f_{C,C'}(v) := \frac{p_C^{(i)}(v)}{w_i(v)} \cdot \frac{p_{C'}^{(i+1)}(v)}{w_{i+1}(v)}$ be the amount of flow sent along edge $e_{C,C'}$ for a unit demand on v.*

We first show that all the edges $e_{C,C'}$ with nonzero flow $f_{C,C'}(v)$ are in $\widetilde{H}$. To do so, we show that $C \subseteq C'$ in that case.

Lemma 4.16. *If $f_{C,C'}(v) \neq 0$ for some $C \in \mathcal{C}_i, C' \in \mathcal{C}_{i+1}$ and $v \in V$ we have $C \subseteq C'$ given $\gamma_{diam} \geq 8\gamma_{HRG}$.*

Proof. For $i + 1 = \kappa_{\mathrm{HRG}} - 1$ the claim follows directly. For other i, because $f_{C,C'}(v) \neq 0$ we have

$$\mathrm{dist}_G(v, V \setminus C')/D_{i+1} - 1/(4\gamma_{\mathrm{HRG}}) > 0$$

we get that

$$\mathrm{dist}_G(v, V \setminus C') \geq D_{i+1}/(4\gamma_{\mathrm{HRG}}) \geq 2D_i.$$

where the last inequality follows from $\gamma_{\mathrm{diam}} \geq 8\gamma_{\mathrm{HRG}}$ (see Theorem 4.5). By definition, if $p_C^{(i)}(v) \neq 0$ then C is a cluster of diameter D_i that contains v and is thus contained in C'. □

We then state the main lemma of the analysis of [RGH+22].

Lemma 4.17 (See Lemma 4.5 in [RGH+22]). *For every $C \in \mathcal{C}_{i-1}$ and $C' \in \mathcal{C}_i$ for $i = 1, \ldots, \kappa_{HRG}-1$ we have*

$$|f_{C,C'}(u) - f_{C,C'}(v)| \leq O(\gamma_{SNC}^2 \mathrm{dist}_G(u,v)/D_{i-1})$$

Given Lemma 4.17 a decomposition of the flow into shortest paths yields the following lemma.

Lemma 4.18 (See Corollary 4.5 in [RGH+22]). *Given $\gamma_{diam} \geq 8\gamma_{HRG}$ and any demand $\boldsymbol{d} \in \mathbb{R}^{|V|}$, we have that*

$$\sum_{i \in [\kappa_{HRG}-1]} \sum_{C \in \mathcal{C}_{i-1}, C' \in \mathcal{C}_i} \sum_{v \in V_1} |f_{C,C'}(v)\boldsymbol{d}(v)| \cdot D_i$$

is at most $O(\kappa_{HRG}\gamma_{HRG}^5\gamma_{diam})$ times the optimal routing length $\mathrm{OPT}(\boldsymbol{d})$ on G routing demands $\boldsymbol{d}$.

Given Lemma 4.18, we conclude with a proof of Theorem 4.9.

Proof of Theorem 4.9. We show the properties in *Theorem* 4.9 one by one.

1. The first property holds by inspecting Definition 4.12 and Definition 4.13.
2. The matrix mapping paths to their edges can be extracted from Definition 4.13 in a straightforward fashion.
3. We finally show an upper bound on $\left\|\widetilde{\boldsymbol{L}}\widetilde{\boldsymbol{P}}\boldsymbol{L}^{-1}\right\|_{1\to 1}$. We have

$$\begin{aligned}
\left\|\widetilde{\boldsymbol{L}}\widetilde{\boldsymbol{P}}\boldsymbol{L}^{-1}\right\|_{1\to 1} &\overset{(a)}{=} \max_{\boldsymbol{f}\in\mathbb{R}^{|E|}} \frac{\left\|\widetilde{\boldsymbol{L}}\widetilde{\boldsymbol{P}}\boldsymbol{L}^{-1}\boldsymbol{f}\right\|_1}{\|\boldsymbol{f}\|_1} \\
&\overset{(b)}{=} \max_{\widetilde{\boldsymbol{f}}\in\mathbb{R}^{|E|}} \frac{\left\|\widetilde{\boldsymbol{L}}\widetilde{\boldsymbol{P}}\widetilde{\boldsymbol{f}}\right\|_1}{\left\|\boldsymbol{L}\widetilde{\boldsymbol{f}}\right\|_1} \\
&\overset{(c)}{=} \max_{\widetilde{\boldsymbol{f}}\in\mathbb{R}^{|E|}} \frac{\left\|\widetilde{\boldsymbol{L}}(\boldsymbol{\Pi}_{E\mapsto(V_1,V_1)} - \boldsymbol{\Pi}_{\mathcal{P}\mapsto\widetilde{\mathcal{E}}}\boldsymbol{\Pi}_{V\mapsto\mathcal{P}}\boldsymbol{B}^\top)\widetilde{\boldsymbol{f}}\right\|_1}{\left\|\boldsymbol{L}\widetilde{\boldsymbol{f}}\right\|_1} \\
&\overset{(d)}{\leq} 1 + \max_{\widetilde{\boldsymbol{f}}\in\mathbb{R}^{|E|}} \frac{\left\|\widetilde{\boldsymbol{L}}\boldsymbol{\Pi}_{\mathcal{P}\mapsto\widetilde{\mathcal{E}}}\boldsymbol{\Pi}_{V\mapsto\mathcal{P}}\boldsymbol{B}^\top\widetilde{\boldsymbol{f}}\right\|_1}{\left\|\boldsymbol{L}\widetilde{\boldsymbol{f}}\right\|_1} \\
&\overset{(e)}{=} 1 + \max_{\boldsymbol{d}:\boldsymbol{d}^\top\mathbf{1}=0} \frac{\left\|\widetilde{\boldsymbol{L}}\boldsymbol{\Pi}_{\mathcal{P}\mapsto\widetilde{\mathcal{E}}}\boldsymbol{\Pi}_{V\mapsto\mathcal{P}}\boldsymbol{d}\right\|_1}{\mathrm{OPT}(\boldsymbol{d})} \\
&\overset{(f)}{\leq} 1 + \max_{\boldsymbol{d}:\boldsymbol{d}^\top\mathbf{1}=0} \frac{\sum_{i\in[\kappa_{\mathrm{HRG}}-1]}\sum_{C\in\mathcal{C}_{i-1},C'\in\mathcal{C}_i}\sum_{v\in V_1}|f_{C,C'}(v)\boldsymbol{d}(v)|\cdot O(\gamma_{\mathrm{SNC}}\cdot D_i)}{\mathrm{OPT}(\boldsymbol{d})} \\
&\overset{(g)}{\leq} O(\kappa_{\mathrm{HRG}}\gamma_{\mathrm{HRG}}^6\gamma_{\mathrm{diam}}).
\end{aligned}$$

Equation (a) follows from the definition of the norm, and equation (b) follows by substituting $\boldsymbol{f}$ with $\boldsymbol{L}\boldsymbol{f} = \widetilde{\boldsymbol{f}}$. Then (c) follows from the definition of $\widetilde{\boldsymbol{P}}$, (d) follows by the triangle inequality and that for $e \in E$ corresponding to $e' \in (V_1, V_1)$, we have $\boldsymbol{l}(e) = \boldsymbol{l}_{\widetilde{H}^E}(e')$, (e) follows by

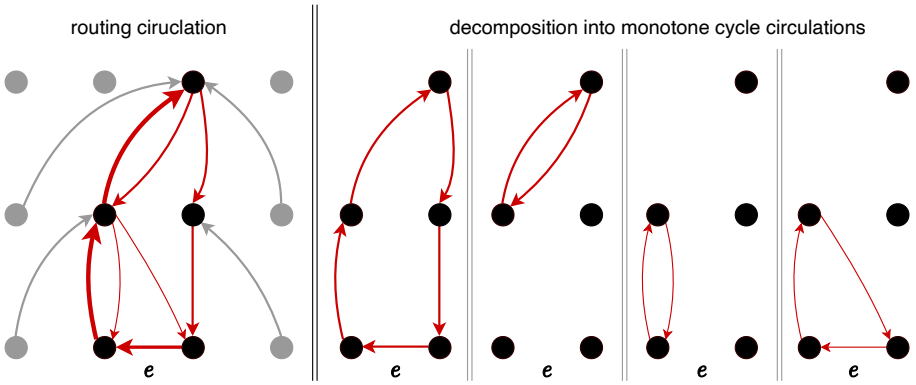

Figure 2: A routing circulation (Definition 4.19) depicted in on the left hand side in red. The thickness of the arrows corresponds to the amount of flow. On the right, we show a lossless decomposition of this flow into monotone cycles (Definition 4.21). Notice that we incur some loss in the decomposition in general.

substituting $\boldsymbol{d}$ for $\boldsymbol{B}^\top \widetilde{\boldsymbol{f}}$ and realizing that if there was a flow $\widetilde{\boldsymbol{f}}'$ with $\left\|\boldsymbol{L}\widetilde{\boldsymbol{f}}'\right\| \leq \left\|\boldsymbol{L}\widetilde{\boldsymbol{f}}\right\|$ routing demands $\boldsymbol{d}$ then this one has higher objective value. Inequality (f) follows from the definition of $f_{C,C'}(v)$ and $\widetilde{\boldsymbol{L}}$. Finally, inequality (g) follows from Lemma 4.18.

□

4.4 Flow Decomposition

Corollary 4.10 ensures that there is an edge $e \in E$ such that some circulation in H^e which consists of $e' \in (V_1, V_1)$ and a distribution over monotone paths has a good competitive ratio. We refer to such circulations as *routing circulations* (Definition 4.19). We then describe a way of decomposing these routing circulations into simple cycles that consist of a monotone increasing path, decreasing path, and possible edge e. These are more suitable for building a fast query data structure.

Next, we formally define routing circulations on $\widetilde{H}^e$.

Definition 4.19 (Routing circulation)**.** *Given an abstracted HRG $\widetilde{H}^e = (\mathcal{V}, \widetilde{\mathcal{E}}, \boldsymbol{l}_{\widetilde{H}^e})$ where $e = (u, v)$ we call the flow $\boldsymbol{f}$ a routing circulation on $\widetilde{H}^e$ if*

1. *the flow $\boldsymbol{f}$ is a circulation, i.e. $\boldsymbol{B}_{\widetilde{H}^e}^\top \boldsymbol{f} = \boldsymbol{0}$ and*
2. *the flow $\boldsymbol{f}$ is composed of flow on the edge e, and a sum of path flows such that each path P starts at either u or v and is monotonically increasing. Each path P originating at v carries some positive amount of flow, and each path originating at u carries some negative amount of flow (or vice versa). Notice that a positive amount of flow sends flow up the hierarchy, and a negative amount sends flow down the hierarchy.*

Remark 4.20. *Notice that $\widetilde{\boldsymbol{P}}\boldsymbol{1}_e$ as in Section 4.3 is a routing circulation on $\widetilde{H}^e$.*

Next we define monotone cycles, which are a crucial object in our algorithm. They are simple cycles in the abstracted HRG that first go monotonically up, and then down the hierarchy (or vice versa). They also may contain an extra edge $e \in (V_1, V_1)$.

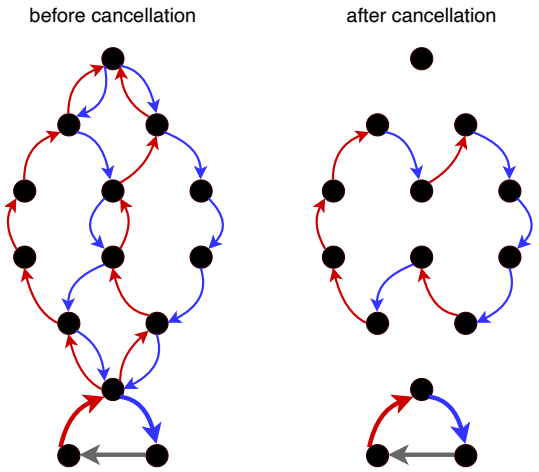

Figure 3: An example of a circulation on monotone paths that yields a non-monotone cycle after cancellation. Blue edges carry flow down, and red edges carry flow up. It is not possible to decompose this circulation into monotone cycles without increasing the flow weight. To achieve a good decomposition, we crucially exploit that the lengths of edges at level i are $\gamma_{\text{HRG}}\gamma_{\text{diam}}^{i}$, and thus edges between layers with high indices are much longer.

Definition 4.21 (Monotone cycles)**.** *Given a abstracted HRG $\widetilde{H}^e = (\mathcal{V}, \widetilde{\mathcal{E}}, \boldsymbol{l}_{\widetilde{H}^e})$ with extra edge $e = (u, v)$ we define a monotone cycle to be a simple cycle composed of a monotonically increasing path, monotonically decreasing path, and optionally the edge e.*

Next, we show that any routing circulation (Definition 4.19) can be decomposed into cycle flows on monotone cycles (Definition 4.21) without increasing the weight of the flow by more than a sub-logarithmic factor. See Figure 2 for an illustration of such a decomposition. The reason that a lossless decomposition does not exist is because in the oblivious routing, flow that has merged into a vertex may split further down the line. However, because the lengths in the abstracted HRG are geometrically increasing upwards, we can intuitively charge a flow that merges and splits to the highest level where it splits.

Lemma 4.22 (Flow decomposition on abstracted HRG)**.** *Given a routing circulation $\boldsymbol{f}$ on an abstracted HRG $\widetilde{H}^e$ with extra edge $e = (u, v)$ there exists a decomposition $\boldsymbol{f} = \sum_{i=1}^{k} \boldsymbol{c}_i$ for some k such that:*

1. *each $\boldsymbol{c}_i$ is a circulation supported on a single monotone cycle of $\widetilde{H}^e$, and*
2. $\sum_{i=1}^{k} \left\| \widetilde{\boldsymbol{L}} \boldsymbol{c}_i \right\|_1 \leq 4\kappa_{HRG} \left\| \widetilde{\boldsymbol{L}} \boldsymbol{f} \right\|_1$.

where $\widetilde{\boldsymbol{L}} = \text{diag}(\boldsymbol{l}_{\widetilde{H}^e})$.

Proof. We iteratively decompose a routing circulation $\boldsymbol{f}$ on $\widetilde{H}^e$. Throughout, we will let the "mass" of a circulation or path flow mean the amount on a single edge of the flow. Assume without loss of generality that there is one unit of flow on edge e from v to u. Because $\boldsymbol{f}$ is a routing circulation (Definition 4.19), the part of $\boldsymbol{f}$ restricted to H, i.e., without the edge e, can be written as the sum of flows representing a distribution over monotonically increasing paths from u (which we denote as

$\mathcal{P}_u$), and monotone decreasing paths into v (which we denote as $\mathcal{P}_v$). Now we iteratively decompose the flow into circulations. For each $i = \kappa_{\text{HRG}}, \ldots, 1$ we will build a collection of monotone cycle circulations $\mathcal{M}_i$ and circulation $\boldsymbol{f}_i$ satisfying:

- $\boldsymbol{f}_i + \sum_{j=i}^{\kappa_{\text{HRG}}} \sum_{\boldsymbol{c} \in \mathcal{M}_j} \boldsymbol{c} = \boldsymbol{f}$.
- Every edge e' adjacent to some vertex $v' \in V_j$ for $j \geq i+1$ has $\boldsymbol{f}_i(e') = 0$.
- $\boldsymbol{f}_i$ is a routing circulation (Definition 4.19). Let $\mathcal{P}_u^i$ be the monotone increasing paths from u, and $\mathcal{P}_v^i$ be the monotone decreasing flow paths into v, which only contain vertices in $V_1, \ldots, V_i$.
- $\|\widetilde{\boldsymbol{L}}\boldsymbol{f}_i\|_1 \leq \|\widetilde{\boldsymbol{L}}\boldsymbol{f}\|_1$.
- $\sum_{\boldsymbol{c} \in \mathcal{M}_i} \|\widetilde{\boldsymbol{L}}\boldsymbol{c}\|_1 \leq 4\|\widetilde{\boldsymbol{L}}\boldsymbol{f}\|_1$.

These imply the lemma by the second and the final bullet. The original flow $\boldsymbol{f}$ satisfies these properties for $i = \kappa_{\text{HRG}}$. Now, given a flow $\boldsymbol{f}_{i+1}$ satisfying these properties, we will construct a flow $\boldsymbol{f}_i$ and monotone cycle circulations $\mathcal{M}_i$. We first define $\widetilde{\mathcal{E}}_i := \{(u', v') \in \widetilde{\mathcal{E}} | u' \in V_i, v' \in V_{i+1}\}$ to be the set of edges in $\widetilde{H}^e$ between V_i and V_{i+1}. Let F_i be the total absolute value of the flow $\boldsymbol{f}_{i+1}$ on $\widetilde{\mathcal{E}}_i$, i.e.,

$$F_i = \sum_{e' \in \widetilde{\mathcal{E}}_i} |\boldsymbol{f}_{i+1}(e')|.$$

While there is some nonzero $\boldsymbol{f}_{i+1}(e')$ for $e' \in \widetilde{\mathcal{E}}_i$, do the following. Let e_1 have minimal nonzero $|\boldsymbol{f}_{i+1}(e_1)|$. Without loss of generality, assume that it has positive flow from $v_1 \in V_i$ to y in V_{i+1}. Because $\boldsymbol{f}_{i+1}$ is a circulation with no flow on edges from V_{i+1} to V_{i+2} by induction, there must be positive flow on some e_2 from y to $v_2 \in V_i$. Because $\boldsymbol{f}_{i+1}$ is a routing circulation by induction (see bullet 3 above), we know that there is a path from u to v_1 to y, which we call $P_u \in \mathcal{P}_u^{i+1}$, and a path from y to v_2 to v, which we call $P_v \in \mathcal{P}_v^{i+1}$. Let μ the smallest among $|\boldsymbol{f}_{i+1}(e_1)|$, and the mass of P_u or P_v. We will peel off a circulation $\boldsymbol{c}$ depending on two cases:

- If P_u, P_v intersect at $x \neq y$, then set $\boldsymbol{c}$ to be the circulation of mass μ from $x \to v_1 \to y \to v_2 \to x$. Remove mass μ from path P_u, and add mass μ of path $P_u[u, x]$ to $\boldsymbol{f}_i$. Remove mass μ from P_v, and add mass μ of path $P_v[x, v]$ to $\boldsymbol{f}_i$.
- If P_u, P_v do not intersect other than at y, let $\boldsymbol{c}$ denote μ mass of the cycle from $v \to u \to v_1 \to y \to v_2 \to v$. Remove μ flow from v to u, and μ mass from both paths P_u and P_v.

The process terminates because during each iteration we either set some $\boldsymbol{f}_{i+1}(e') = 0$, or shorten some path in $\mathcal{P}_u^{i+1}$ or $\mathcal{P}_v^{i+1}$ to under level i.

Let us check the five conditions of the iterative procedure. The first and second condition follow by construction. For the third condition, define $\mathcal{P}_u^i$ as the union of: (1) paths $P_u[u, x]$ when paths P_u, P_v intersected at $x \neq y$, (2) paths remaining in $\mathcal{P}_u^{i+1}$ shortcut to level i. Define $\mathcal{P}_v^i$ analogously. These paths show that $\boldsymbol{f}_i$ is a routing circulation by construction.

To check the fourth, note that by setting all flows on $\widetilde{\mathcal{E}}_i$ to 0 in $\boldsymbol{f}_{i+1}$, we decrease ℓ_1-cost $\boldsymbol{f}_{i+1}$ (i.e., $\|\boldsymbol{L}\boldsymbol{f}_{i+1}\|_1$) by $F_i \gamma_{\text{HRG}} \gamma_{\text{diam}}^i$. For edges in lower layers, the total weight increase is $F_i \sum_{j<i} \gamma_{\text{HRG}} \gamma_{\text{diam}}^j \leq 2F_i \gamma_{\text{HRG}} \gamma_{\text{diam}}^{i-1}$. Also, $|\boldsymbol{f}_i(e)| \leq |\boldsymbol{f}_{i+1}(e)|$. So $\|\widetilde{\boldsymbol{L}}\boldsymbol{f}_i\|_1 \leq \|\widetilde{\boldsymbol{L}}\boldsymbol{f}_{i+1}\|_1 \leq \|\widetilde{\boldsymbol{L}}\boldsymbol{f}\|_1$. The total weight of edges in the cycles $\boldsymbol{c} \in \mathcal{M}_i$ for edges in $\widetilde{H}$ is at most $2F_i \sum_{j \leq i} \gamma_{\text{HRG}} \gamma_{\text{diam}}^j \leq 4F_i \gamma_{\text{HRG}} \gamma_{\text{diam}}^i$. The total contribution of edge e is also at most its original flow in $\boldsymbol{f}_{i+1}$. So $\sum_{\boldsymbol{c} \in \mathcal{M}_i} \|\widetilde{\boldsymbol{L}}\boldsymbol{c}\|_1 \leq 4\|\widetilde{\boldsymbol{L}}\boldsymbol{f}_{i+1}\|_1 \leq 4\|\widetilde{\boldsymbol{L}}\boldsymbol{f}\|_1$. □

Remark 4.23. *It is not possible to decompose the flow in a lossless manner and exploiting that the lengths are geometrically increasing upwards is crucial. This is illustrated in Figure 3.*

The flow decomposition in Lemma 4.22 directly translates to a decomposition in the HRG H.

Definition 4.24 (Flow mapping and monotone cycles on H)**.** *Given a routing circulation $\boldsymbol{f}$ on the abstracted HRG $\widetilde{H}^e$ we let $\boldsymbol{f}_{H^e}$ denote the flow mapped via $\Pi_{\widetilde{H}\mapsto H}$ i.e. for $e' \neq e$ we have*

$$\boldsymbol{f}_{H^e}(e') = \sum_{e'' \text{s.t.} e' \in \Pi_{\widetilde{H}\mapsto H}(e'')} \boldsymbol{f}(e'')$$

and $\boldsymbol{f}_{H^e}(e) = \boldsymbol{f}(e)$. We define the monotone cycle circulations on H^e as the set of $\boldsymbol{f}_{H^e}$ for monotone cycle circulations $\boldsymbol{f}$ on $\widetilde{H}^e$.

We conclude that we can decompose a flow into monotone cycles on a HRG H^e given an upper bound on its weight on $\widetilde{H}^e$. This upper bound translates to the decomposition on H^e since we chose edge lengths in the abstracted HRG $\widetilde{H}$ as upper bounds of corresponding path lengths in H.

Claim 4.25. *For an edge $(u, v) \in \widetilde{H}$ with $u \in V_i, v \in V_{i+1}$, the corresponding path in H has length at most $3\gamma_{HRG}\gamma_{diam}^i$.*

Proof. Let the path in H be composed of edges (u, x), a path from x to y in the a tree $T \in F_i$, and (y, v). By construction of the linking edges $\mathcal{E}$, both (u, x) an (y, v) have length $\gamma_{\text{HRG}}\gamma_{\text{diam}}^i$. We know that $x, y \in S_i \cap T$ by property Item 4 of Definition 4.2, so $\text{dist}_T(x, y) \leq \text{diam}(S_i \cap T) \leq \gamma_{\text{HRG}}\gamma_{\text{diam}}^i$. Combining these gives the desired bound. □

Together with the results from the previous sections, this suffices to show that a monotone cycle of some H^e is an approximate min-ratio circulation.

Corollary 4.26. *Let H be a hierarchical routing graph of $G = (V, E, \boldsymbol{l}, \boldsymbol{g})$. Then, there exists some e and monotone cycle circulation $\boldsymbol{c}$ on H^e such that*

$$\boldsymbol{g}_H^\top \boldsymbol{c} / \left\|\boldsymbol{L}_H \boldsymbol{c}\right\|_1 \leq \frac{1}{O(\kappa_{HRG}\gamma_{\text{route}})} \min_{\boldsymbol{\Delta}:\boldsymbol{B}^\top\boldsymbol{\Delta}=\boldsymbol{0}} \boldsymbol{g}^\top \boldsymbol{\Delta} / \left\|\boldsymbol{L}\boldsymbol{\Delta}\right\|_1 .$$

Proof. By Corollary 4.10 there exists some routing circulation $\boldsymbol{f}$ on some $\widetilde{H}^e$ such that

$$\boldsymbol{g}_H^\top \boldsymbol{\Pi}_{\widetilde{H}\mapsto H}\boldsymbol{f} / \left\|\widetilde{\boldsymbol{L}}\boldsymbol{f}\right\|_1 \leq \frac{1}{\gamma_{\text{route}}} \min_{\boldsymbol{\Delta}:\boldsymbol{B}^\top\boldsymbol{\Delta}=\boldsymbol{0}} \boldsymbol{g}^\top \boldsymbol{\Delta} / \left\|\boldsymbol{L}\boldsymbol{\Delta}\right\|_1$$

Notice that the quantities on both sides of the inequality are negative without loss of generality. Then, we have

$$\frac{\sum_{i=1}^k \boldsymbol{g}_H^\top \boldsymbol{\Pi}_{\widetilde{H}\mapsto H}\boldsymbol{c}_i}{\sum_{i=1}^k \left\|\widetilde{\boldsymbol{L}}\boldsymbol{c}_i\right\|_1} \leq \frac{\boldsymbol{g}_H^\top \boldsymbol{f}}{4\kappa_{\text{HRG}} \left\|\widetilde{\boldsymbol{L}}\boldsymbol{f}\right\|_1}$$

by Lemma 4.22. We then use the standard averaging inequality

$$\min_{i\in[k]} \frac{\boldsymbol{g}_H^\top \boldsymbol{\Pi}_{\widetilde{H}\mapsto H}\boldsymbol{c}_i}{\left\|\boldsymbol{L}_H \boldsymbol{c}_i\right\|_1} \leq \frac{\sum_{i=1}^k \boldsymbol{g}_H^\top \boldsymbol{c}_i}{\sum_{i=1}^k \left\|\boldsymbol{L}_H \boldsymbol{c}_i\right\|_1}.$$

Finally, we let j be a minimizer of the left-hand side of the previous inequality, and define $\boldsymbol{c}' = \boldsymbol{\Pi}_{\widetilde{H}\mapsto H}\boldsymbol{c}_j$. We then have

$$\frac{\boldsymbol{g}_H^\top \boldsymbol{c}'}{\|\boldsymbol{L}_H \boldsymbol{c}'\|_1} \leq \frac{1}{3}\min_{i\in[k]} \frac{\boldsymbol{g}_H^\top \Pi_{\widetilde{H}\mapsto H}(\boldsymbol{c}_i)}{\|\boldsymbol{L}_H \boldsymbol{c}_i\|_1}$$

by Definition 4.2 and Definition 4.3 since the lengths on $\widetilde{H}$ are upper bounds for the lengths of the embedded paths in H up to a factor of 3 by Claim 4.25. This concludes the proof of this corollary since $\boldsymbol{c}'$ is a monotone cycle circulation on H^e by Definition 4.24. □

Recall that the HRG H flatly embeds into G. Thus, the monotone cycle circulation $\boldsymbol{c}$ on H is also a circulation in G via the edge map $\Pi_{V(H)\to V(G)}$.

4.5 Tree Decomposition and Maintenance

In this section, we show that a hierarchical routing graph H of a graph $G = (V, E, \boldsymbol{l}, \boldsymbol{g})$ can be decomposed into a collection of $m^{o(1)}$ many subtrees $\mathcal{T}$ of H and an easy-to-maintain set of off-tree edges such that every monotone cycle on a graph H^e for every edge $e \in E$ is represented by a tree cycle for some off-tree edge e' in some tree $T \in \mathcal{T}$. Together with the previous section, this reduces the query to solving the problem on tree cycles.

Definition 4.27 (Hierarchical routing tree)**.** *We call graph T a* hierarchical routing tree *(HRT) if it is a hierarchical routing graph (Definition 4.2) and additionally every vertex in V_i is adjacent to a single vertex in F_i, i.e., the out-degree (property 5 in Definition 4.2) is upper bounded by* 1 *instead of γ_{HRG}.*

We first define the sets of off-tree edges.

Definition 4.28. *Given a hierarchical routing graph H of a graph $G = (V, E, \boldsymbol{l}, \boldsymbol{g})$ we define the sets of off-tree edges E_{graph} and E_{pair} as follows:*

- *E_{graph} contains the images of edges $e \in G$ into (V_1, V_1), i.e.,*

  $$E_{\text{graph}} = \{(\Pi_{V\to V_1}(u), \Pi_{V\to V_1}(v)) : (u, v) = e \in E(G)\}.$$

  The gradients and lengths are the same as in G.

- *For all $v \in V_i$, and all pairs $(v, u_1), (v, u_2)$ of out-edges of v with $u_1, u_2 \in F_i$, add the edge $e = (u_1, u_2)$ to E_{pair}. Define $\boldsymbol{l}(e) = 2\gamma_{HRG}\gamma^i_{diam}$ and $\boldsymbol{g}(e) = 0$.*

Recall that there are two types of monotone cycles in $\widetilde{H}^e$: those containing e, and those consisting only of edges in $\widetilde{H}$. The former are naturally tree cycles in some tree induced by edges in E_{graph}. Similarly, the edges in E_{pair} are used to turn cycles consisting only of edges in $\widetilde{H}$ into tree cycles. Note that even after adding these edges to a HRG H, the result is still flat over G. This is because the E_{graph} clearly respect flatness, and by out-edge consistency (property 4 of Definition 4.2), we know that for a paired edge $(u_1, u_2) \in E_{\text{pair}}$ coming from v, that $\Pi_{V(H)\to V(G)}(u_1) = \Pi_{V(H)\to V(G)}(u_2) = \Pi_{V(H)\to V(G)}(v)$.

We observe that the total number of edges in E_{graph} and E_{pair} is almost linear in the number of edges of G.

Lemma 4.29. *The number of edges $|E_{\text{graph}}| + |E_{\text{pair}}| \leq |E| + \kappa_{HRG}\gamma^2_{HRG}|V|$.*

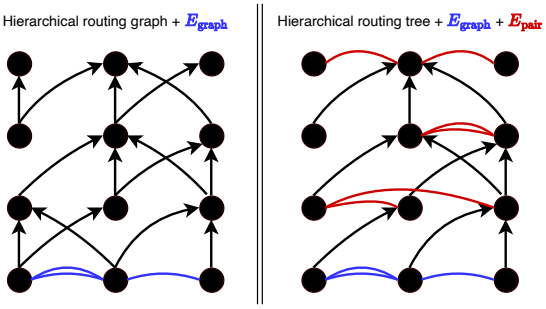

Figure 4: This figure displays a abstracted HRG $\widetilde{H}$ with extra edges E_{graph}, and one of its sub-trees. The red edges E_{pair} are generated by out-pairs in $\widetilde{H}$. We generate a set of trees such that every monotone cycle is a tree cycle formed by some tree and an off-tree edge in E_{pair} or E_{graph}.

Proof. We have $|E_{\text{graph}}| = |E|$. A vertex $v \in V_i$ has outdegree at most γ_{HRG} by property 5 of Definition 4.2, so it contributes at most γ_{HRG}^2 paired edges. The bound follows because the total number of vertices in $V_1, \ldots, V_{\kappa_{\text{HRG}}}$ is $n\kappa_{\text{HRG}}$. □

In the next lemma, we show that we can construct a collection of at most $m^{o(1)}$ hierarchical routing trees that are subgraphs of a hierarchical routing graph H so that every monotone cycle corresponds to a tree cycle formed by some tree and off-tree edge from either E_{graph} or E_{pair}.

We first describe a randomized procedure for generating such trees. Our deterministic construction will derandomize this approach in a standard way. Given a hierarchical routing graph H, consider sampling a random edge between $v \in V_i$ and $u \in F_i$ for every vertex $v \in V_i$ and every i. This yields a hierarchical routing tree T formed by dropping all edges from H that were not sampled. For simplicity, consider a monotone cycle that includes an edge $e \in E$. Then, for every vertex $v \in V_1$ that is part of the cycle, we have a chance of at least $1/\gamma_{\text{HRG}}$ to choose the correct edge to F_i. Since the cycle contains at most $2\kappa_{\text{HRG}}$ such vertices, the probability of preserving the min-circulation in the sampled tree T is at least $1/\gamma_{\text{HRG}}^{2\kappa_{\text{HRG}}}$. Therefore, we expect to find a tree containing the cycle after sampling roughly $\gamma_{\text{HRG}}^{2\kappa_{\text{HRG}}}$ trees. To make this process deterministic, we exploit that the choices of the vertices at level V_i can be highly correlated and then enumerate over all choices.

We define the collection of routing trees. To do so we associate an identifier consisting of $\log_2 |V_i| = \log_2 |V|$ bits to each vertex $v \in V_i$.

Definition 4.30. *Given a hierarchical routing graph H of $G = (V, E, \boldsymbol{l}, \boldsymbol{g})$, we define the tree collection $\mathcal{T}$ to consist of a hierarchical routing tree $T_{\boldsymbol{p},\boldsymbol{a},\boldsymbol{b}}$ for every triple $(\boldsymbol{p}, \boldsymbol{a}, \boldsymbol{b}) \in [\log_2 |V|]^{\kappa_{HRG}} \times [\gamma_{HRG}]^{\kappa_{HRG}} \times [\gamma_{HRG}]^{\kappa_{HRG}}$. We now describe the construction of $T_{\boldsymbol{p},\boldsymbol{a},\boldsymbol{b}}$. The vertex $v \in V_i$ chooses its edge to F_i as follows. If the $\boldsymbol{p}(i)$-th bit of the identifier of v is 1, then it chooses the $\boldsymbol{a}(i)$-th edge between v and F_i for some arbitrary but consistent ordering of the edges. If on the other hand the $\boldsymbol{p}(i)$-th bit is 0, it chooses the $\boldsymbol{b}(i)$-th edge between v and F_i. Whenever the $\boldsymbol{b}(i)$-th or $\boldsymbol{a}(i)$-th edge does not exist, substitute the missing one with an arbitrary edge to maintain connectivity. This concludes the description of the trees $T_{\boldsymbol{p},\boldsymbol{a},\boldsymbol{b}}$ and thus the set $\mathcal{T}$.*

We show that a hierarchical routing graph can be decomposed into a small set of trees such that each routing cycle is a tree cycle for one of the trees.

Lemma 4.31 (HRG decomposition)**.** *Given a hierarchical routing graph H of $G = (V, E, \boldsymbol{l}, \boldsymbol{g})$ its off-tree edges E_{graph} and E_{pair} (see Definition 4.28) and the set $\mathcal{T}$ of tree sub-graphs of H (see Definition 4.30), for every monotone cycle circulation $\boldsymbol{c}$ on H^e for some edge $e \in E$ there exists an edge in $e' \in E_{\text{graph}} \cup E_{\text{pair}}$ so that $\boldsymbol{c} = \beta \cdot \mathbf{1}_{T_{\circlearrowleft}[e']}$ for some $T \in \mathcal{T}$ and $\beta \in \mathbb{R}$.*

Proof. Let $\boldsymbol{c}$ be a monotone cycle circulation in the abstracted HRG, with corresponding monotone cycle circulation $\boldsymbol{c}_{H^e}$ in H^e. Note that a circulation on a cycle has the same amount of flow on each edge. We distinguish two similar but distinct cases.

1. Case 1: The cycle contains an off-HRG edge $e = (u, v)$. Decompose $\boldsymbol{c}$ into two paths of some length k, plus edge e. Say that it contains vertices $u = u_1, u_2, \ldots, u_k$ and $v = v_1, v_2, \ldots, v_k$ where $u_i, v_i \in V_i$ and $u_k = v_k$. Thus, $\boldsymbol{c}_{H^e}$ is a tree cycle of a tree $T \subseteq H$ (as in Definition 4.30) if the out-edge of u_i in T goes to a connected component in F_i with an in-edge to u_{i+1} (and the analogous statement for v_i, v_{i+1}), for all $i = 1, \ldots, k-1$. We construct a tuple $(\boldsymbol{p}, \boldsymbol{a}, \boldsymbol{b})$ so that $T_{\boldsymbol{p},\boldsymbol{a},\boldsymbol{b}}$ satisfies this. Since $u_i \neq v_i$, let $\boldsymbol{p}(i)$ be so that the $\boldsymbol{p}(i)$-th bit of u_i and v_i differ, for $i = 1, \ldots, k-1$. Say the $\boldsymbol{p}(i)$-th bit of u_i is 1 and v_i is 0. Then, let $\boldsymbol{a}(i)$ be the identifier of the out-edge of u_i to the connected component with an in-edge to u_{i+1}, and $\boldsymbol{b}(i)$ be the identifier of the out-edge of v_i to the connected component with an in-edge to v_{i+1} for $i = 1, \ldots, k-1$. If the $\boldsymbol{p}(i)$-th bit of u_i is 0 and v_i is 1, reverse this. Evidently, this gives a tree $T_{\boldsymbol{p},\boldsymbol{a},\boldsymbol{b}}$ (with off-tree edge e) containing $\boldsymbol{c}_{H^e}$. This concludes the first case.

2. Case 2: the cycle is internal to H. If the cycle is instead an internal monotone cycle of H we consider the unique vertex $v \in V_i$ at the lowest level i on the cycle. This vertex has two adjacent edges (u_1, v) and (v, u_2) on the cycle. There is an off-tree edge between u_1 and u_2 in E_{pair}. Then the rest of the argument is analogous to the first case.

□

Next, we show that given our algorithm that maintains a hierarchical routing graph H, there is an efficient algorithm that maintains the sets $\mathcal{T}$.

Lemma 4.32 (Tree maintenance)**.** *Given a dynamic HRG H of a dynamic graph $G = (V, E, \boldsymbol{l}, \boldsymbol{g})$ with polynomially-bounded lengths in $[1, L]$ and off HRG edges E_{graph}, there is a data structure that supports a polynomially bounded number of updates of the following type:*

- *InsertEdge$_H(e)$/DeleteEdge$_H(e)$: inserts/deletes edge e to/from H after each update to G such that H remains a hierarchical routing of the dynamic graph G.*
- *InsertOffEdge(e)/DeleteOffEdge(e): adds/removes edge e to/from E_{graph}*

Under these updates, the data structure maintains a collection of rooted trees $\mathcal{T}$ and a set of auxiliary edges $E_{\text{off}} = E_{\text{graph}} \cup E_{\text{pair}}$ as in Definition 4.28. Each tree $T \in \mathcal{T}$ is flat over G, explicitly represented by map $\Pi_{V(T) \to V(G)}$. Furthermore

$$\min_{T \in \mathcal{T}} \min_{\boldsymbol{c} \in \{\mathbf{1}_{T_{\circlearrowleft}[e]}, -\mathbf{1}_{T_{\circlearrowleft}[e]}\}} \frac{\boldsymbol{g}_T^\top \boldsymbol{c}}{\|\boldsymbol{L}_T \boldsymbol{c}\|_1} \leq \frac{1}{O(\kappa_{HRG} \gamma_{\text{route}})} \min_{\boldsymbol{\Delta}: \boldsymbol{B}^\top \boldsymbol{\Delta} = \mathbf{0}} \boldsymbol{g}^\top \boldsymbol{\Delta} / \|\boldsymbol{L}\boldsymbol{\Delta}\|_1 , \tag{3}$$

for $\gamma_{\text{route}} = e^{O(\log^{83/84} m)}$ and $\kappa_{HRG} = \log^{1/84} m$. The algorithm is deterministic, can be initialized in time $O(m\gamma_{\text{tree}})$ for $\gamma_{\text{tree}} = e^{O(\log^{83/84} m \log\log m)}$ and processes updates in amortized time γ_{tree}.

Proof. We will set $\gamma_{\text{HRG}} = e^{O(\log^{41/42} m \log\log m)}$, $\gamma_{\text{diam}} = e^{O(\log^{83/84} m)}$, $\kappa_{\text{HRG}} = \log^{1/84} m$, and first invoke the data structure of Theorem 4.5. With this setup, we first show that the sets E_{graph} and E_{pair} can be maintained efficiently. Indeed, every update to G leads to exactly one update of E_{graph}, and every update to H causes updates to E_{pair} if it affects an edge in E_i^{out}. In that case, it causes γ_{HRG} changes. Therefore, the amortized update time for maintaining the sets E_{graph} and E_{pair} is $\widetilde{O}(\gamma_{\text{HRG}} \cdot \kappa_{\text{HRG}}) = e^{O(\log^{41/42} m \log\log m)}$.

Then we consider the set of trees $\mathcal{T}$, which are defined by Definition 4.30. Recall that T is a subgraph of H, so we can directly decide which updates in H to propagate to T. This also implies that T is flat over G. (3) follows from the fact that $\mathcal{T}$ captures all monotone cycles (Lemma 4.31), and the quality of the best monotone cycle (Corollary 4.26).

The runtime is given by $O(|\mathcal{T}|\gamma_{\text{HRG}} \cdot \kappa_{\text{HRG}}) \leq \gamma_{\text{tree}}$, because $|\mathcal{T}| = O(\gamma_{\text{HRG}}^2 \log m)^{\kappa_{\text{HRG}}}$. The lemma follows. □

Proof of Theorem 3.3. The theorem follows from Theorem 4.5, Lemma 4.32 and Corollary 4.10. We use Theorem 4.5 to maintain a HRG of G and Lemma 4.32 to maintain a collection of γ_{tree} flat rooted trees $\mathcal{T}$. For each tree in the collection $T \in \mathcal{T}$ and its set of off-tree edges E_{off}, we use the given min-ratio tree cycle data structure (definition 3.2) to maintain a flat forest F and approximate tree cycle represented as paths on F. Because each T and E_{off} are flat in G, F is also flat on G. Therefore, our min-ratio cycle data structure maintains the flat forest F^G as a disjoint union (on disjoint vertex sets) of flat forests given by each of the γ_{tree} tree cycle data structures. □

5 Portal Routed Graphs and Min-Ratio Tree Cycles

In this section, we build dynamic min-ratio tree cycle data structures by reducing to a min-ratio cycle problem on a substantially smaller graph.

Theorem 3.4 (Reducing dynamic min-ratio tree cycle to smaller min-ratio cycle)**.** *Consider an α-approximate dynamic min-ratio cycle data structure $\mathcal{D}^{\text{MRC}}$ (Definition 3.1) that, on any dynamic graph H with at most m edges, satisfies:*

- *It takes $T_{\text{init}}(m)$-time to initialize.*
- *Each update takes $T_{\text{upd}}(m)$-amortized time.*
- *The flat forest F^H it maintains has vertex congestion γ_{vcong}.*
- *The output cycles $\boldsymbol{c}$ are represented by γ_{cycle} tree paths on F^H and off-tree edges.*

Then, for any size reduction parameter k, there is a $(\alpha \cdot \gamma_{\text{spanner}})$-approximate dynamic min-ratio tree cycle data structure $\mathcal{D}^{\text{TC}}$ (Definition 3.2), where $\gamma_{\text{spanner}} = e^{O(\log^{20/21} m \log\log m)}$ is the parameter given in Theorem 5.13, that, on any dynamic graph G with at most m total tree and non-tree edges, achieves

- *$m\gamma_{\text{spanner}} + T_{\text{init}}(\gamma_{\text{spanner}} \cdot m/k)$ initialization time.*
- *$\gamma_{\text{spanner}} \cdot \left(k^2\gamma_{\text{vcong}} + \frac{T_{\text{init}}(\gamma_{\text{spanner}} \cdot m/k)}{m/k} + T_{\text{upd}}(\gamma_{\text{spanner}} \cdot m/k)\right)$ amortized update time.*
- *the flat embedding forest F^G it maintains has vertex congestion at most $2\gamma_{\text{vcong}}$.*

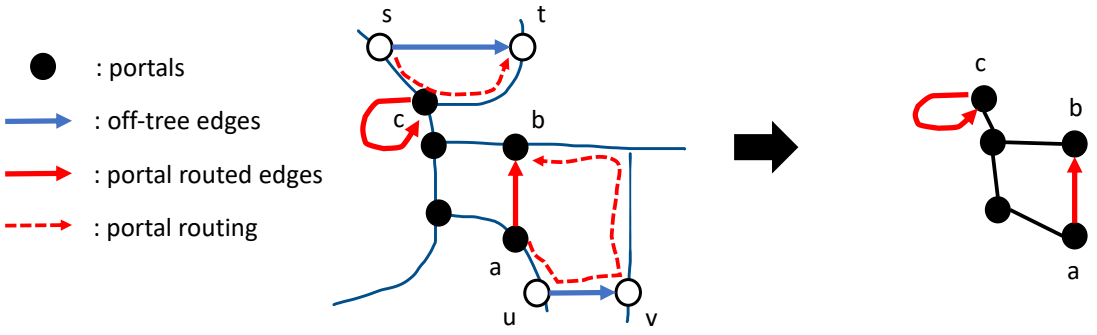

Figure 5: This example contains a tree, two off-tree edges (u, v) and (s, t), and a branch-free set of five portals. The right-hand side shows the portal routed graph. The edge (u, v) is moved to (a, b) and the edge (s, t) is moved to a self-loop (c, c).

- *the output cycles are represented by at most* $\max\{\gamma_{\text{cycle}}, \gamma_{\text{spanner}}\}$ *tree paths on* F^G *and non-tree edges.*

The data structure for Theorem 3.4 is presented in Algorithm 2. On a high-level, we construct and maintain a vertex sparsifier for the approximate min-ratio tree cycle problem. In particular, given any size reduction parameter k, we maintain a smaller graph $\mathcal{P}$ on roughly m/k vertices and reduce the problem of finding an approximate min-ratio cycle on $\mathcal{P}$. In the dynamic setting, we show that $\mathcal{P}$ can be maintained dynamically with roughly k^2 update time, but much lower recourse. Because $\mathcal{P}$ could still contain m edges, we maintain an edge sparsifier with respect to distances, i.e. a spanner, $\widehat{G}$ of $\mathcal{P}$ containing roughly m/k edges. Then, we recursively use $\mathcal{D}^{\text{MRC}}$ to maintain an approximate min-ratio cycle on $\widehat{G}$, which is transformed into an approximate min-ratio tree cycle on G.

The construction of the vertex sparsifier first identifies a set of m/k vertices, called *portals*, and moves every off-tree edge onto the portals. This results in a graph G' containing a tree and off-tree edges between m/k portals. The min-ratio tree cycle is preserved in G'. Then, one can replace every maximal path containing degree-2 vertices with an edge and repeatedly eliminate degree-1 vertices. The process ends with $\mathcal{P}$, a graph on the set of portals and some additional Steiner nodes, which preserves every cycle of G', as well as the min-ratio tree cycle of G.

5.1 Portal Routing and Portal Routed Graphs

For a cleaner presentation, we force the set of portals to contain all the Steiner nodes, i.e., the set of portals is *branch-free* on the tree.

Definition 5.1 (Branch-free set)**.** *Given a tree/forest* T*, a set* $R \subseteq V(T)$ *is* Branch-free *if for any* $u \notin R$*, the number of vertices* $v \in R$ *such that* $T[u, v]$ *containing no other vertex in* R *is at most* 2*.*

To define how we move the off-tree edges, we first define the portal routing of each off-tree edge e by short-cutting the tree cycle $T_{\circlearrowleft}[e]$ at the given set of portals. See Figure 5 for an illustration of portal routing.

Definition 5.2 (Portal routing)**.** *Given a tree/forest $T = (V, E_T)$, a branch-free set of* portals *$P \subseteq V$, and a set of off-tree edges E_{off}, we define the* portal routing *$\mathcal{P}(e)$ for each off-tree edge $e = (u, v)$ as follows:*

$$\mathcal{P}(e) = \begin{cases} T_{\circlearrowleft}[e] & \text{if there are less than 2 portals on the path } T[e] \\ T[a, u] \oplus e \oplus T[v, b] & \text{otherwise} \end{cases}$$

where a and b are the first and the last portal on the tree path $T[u, v]$. We also define $e^{\mathcal{P}} \stackrel{\text{def}}{=} (a, b)$ if there are at least two portals on the path $T[e]$.

Now, we are ready to define the *portal routed graph* $\mathcal{P}$, our notion of a vertex sparsifier for the min-ratio cycle problem. We move each off-tree edge e to the closest portals that short-cut its tree cycle $T_{\circlearrowleft}[e]$. Then, we replace each tree path between two portals $T[p_1, p_2]$ with an edge (p_1, p_2). This naturally defines an embedding from $\mathcal{P}$ into G. The edge lengths and gradients on $\mathcal{P}$ are defined according to the embedding so that cycles in G are preserved in $\mathcal{P}$. See Figure 5 for an illustration of portal routed graphs.

Definition 5.3 (Portal routed graph and embeddings)**.** *Given a graph $G = (V, E)$ which contains a tree/forest $T = (V, E_T)$ and a set of off-tree edges E_{off}, a branch-free set of* portals *$P \subseteq V$, and edge lengths $\boldsymbol{l}$ and gradients $\boldsymbol{g}$, we define the* Portal routed graph *$\mathcal{P}(G, T, P)$ as a graph with vertex set P and an embedding Π into G. It contains the following two types of edges:*

- *Tree-path edges: For each pair of portals $p_1, p_2 \in P$ such that $T[p_1, p_2]$ contains no other portals, we add an edge $e^{\mathcal{P}} = (p_1, p_2)$ to $\mathcal{P}(G, T, P)$ with length $\boldsymbol{l}^{\mathcal{P}}(e^{\mathcal{P}}) \stackrel{\text{def}}{=} \boldsymbol{l}(T[p_1, p_2])$ and gradient $\boldsymbol{g}^{\mathcal{P}}(e^{\mathcal{P}}) \stackrel{\text{def}}{=} 0$. We embed the edge into G using the tree path, i.e., $\Pi(e^{\mathcal{P}}) \stackrel{\text{def}}{=} T[p_1, p_2]$.*
- *Portal routed edges: For each off-tree edge $e \in E_{off}$ such that $e^{\mathcal{P}}$ is well-defined, we add an edge $e^{\mathcal{P}}$ to $\mathcal{P}(G, T, P)$ with length $\boldsymbol{l}^{\mathcal{P}}(e^{\mathcal{P}}) \stackrel{\text{def}}{=} \boldsymbol{l}(\mathcal{P}(e))$ and gradient $\boldsymbol{g}^{\mathcal{P}}(e^{\mathcal{P}}) \stackrel{\text{def}}{=} \langle \boldsymbol{g}, \mathbf{1}_{T_{\circlearrowleft}[e]} \rangle$. We embed the edge into G using the portal routing, i.e., $\Pi(e^{\mathcal{P}}) \stackrel{\text{def}}{=} \mathcal{P}(e)$.*

When the graph G, tree/forest T, and the portal set P are clear from the context, we denote $\mathcal{P}(G, T, P)$ by $\mathcal{P}$ for a clean presentation.

Remark 5.4. *In our usage, we maintain P as an incremental set and never change the gradient of any existing edge in $\mathcal{P}$, even if the tree T is changed. All tree-path edges have gradient zero. For a portal routed edge, its gradient $\boldsymbol{g}^{\mathcal{P}}(e^{\mathcal{P}})$ is defined w.r.t. the initial tree T^{init}.*

The following lemma argues that all tree cycles containing at least one portal are preserved in $\mathcal{P}$. Later in our min-ratio tree cycle data structure, we handle the tree cycles touching no portals separately.

Lemma 5.5 ($\mathcal{P}$ preserves min-ratio tree cycles)**.** *Under the setting of Definition 5.3, for any off-tree edge $e \in E_{off}$ such that $e^{\mathcal{P}}$ is in $\mathcal{P}$ and is not a self-loop and any sign $s \in \{\pm 1\}$, there is a cycle $\boldsymbol{c}^{\mathcal{P}}$ in $\mathcal{P}$ s.t.*

$$\frac{\langle \boldsymbol{g}^{\mathcal{P}}, \boldsymbol{c}^{\mathcal{P}} \rangle}{\left\| \boldsymbol{L}^{\mathcal{P}} \boldsymbol{c}^{\mathcal{P}} \right\|_1} = \frac{\langle \boldsymbol{g}, \Pi_{\mathcal{P} \mapsto G}(\boldsymbol{c}^{\mathcal{P}}) \rangle}{\| \boldsymbol{L} \Pi_{\mathcal{P} \mapsto G}(\boldsymbol{c}^{\mathcal{P}}) \|_1} = s \cdot \frac{\langle \boldsymbol{g}, \mathbf{1}_{T_{\circlearrowleft}[e]} \rangle}{\boldsymbol{l}(T_{\circlearrowleft}[e])}$$

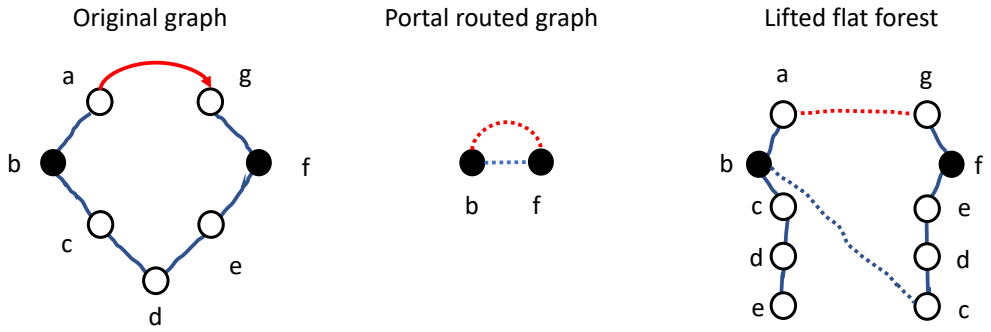

Figure 6: The original graph has two portals b and f and one off-tree edge (a, g). If the forest in the portal routed graph contains a single portal routed edge, it is lifted back to G with the edge (a, g) added. If it contains a single tree-path edge, it is lifted to G with the edge (b, c) added.

Proof. Fix the off-tree edge $e = (u, v) \in E_{\text{off}}$ and consider the tree cycle $T_{\circlearrowleft}[e]$. Let $p_1, p_2, \ldots, p_k$ be the portals along the tree path $T[u, v]$. One can decompose $T_{\circlearrowleft}[e]$ as follows:

$$T_{\circlearrowleft}[e] = T[p_1, p_2] \oplus \cdots \oplus T[p_{k-1}, p_k] \oplus (T[p_k, v] \oplus \mathsf{rev}(e) \oplus T[u, p_1])$$

Notice that each $T[p_i, p_{i+1}]$ corresponds to a tree-path edge (p_i, p_{i+1}) in $\mathcal{P}$ and the part in the parenthesis corresponds to the portal-routed edge $e^{\mathcal{P}} = (p_1, p_k)$ in reverse direction. Therefore, we can define the cycle $\boldsymbol{c}^{\mathcal{P}}$ to be

$$\boldsymbol{c}^{\mathcal{P}} \stackrel{\text{def}}{=} (p_1, p_2) \oplus \cdots \oplus (p_{k-1}, p_k) \oplus \mathsf{rev}(e^{\mathcal{P}})$$

We know that the cycle embeds into G as $T_{\circlearrowleft}[e]$, i.e., $\Pi_{\mathcal{P} \mapsto G}(\boldsymbol{c}^{\mathcal{P}}) = T_{\circlearrowleft}[e]$ and the lemma follows from the definition of edge gradients and lengths in the portal routed graph (Definition 5.3). □

In our data structure for proving Theorem 3.4, we use the given min-ratio cycle data structure $\mathcal{D}^{\text{MRC}}$ on $\mathcal{P}$ in a black-box manner. $\mathcal{D}^{\text{MRC}}$ additionally maintains a forest F that flatly embeds into $\mathcal{P}$ and outputs cycles as several off-tree edges and paths on F. In order to bring the cycle back to G, we first need to transform F into a forest F^G that flatly embeds into G. Then, we can bring the cycle output by $\mathcal{D}^{\text{MRC}}$ back to G as some off-tree edges and paths on F^G. See Figure 6 for an illustration.

Definition 5.6 (Lift a forest from $\mathcal{P}$)**.** *Under the setting of Definition 5.3, given a forest F and $\Pi_{V(F) \mapsto P}$ that are a flat embedding of $\mathcal{P}$, we define its lift in G, denoted by F^G, as follows:*

- *Dangling subtrees: For every vertex $u \in V(F)$ that maps to a portal p, we add u in F^G as well as a copy of T_p dangling down u. T_p is the maximal sub-tree in T containing only one portal which is p. We denote the copy by T_u.*
- *Lifting tree-path edges: For every edge $e^F = (u, v)$ in F that maps to a tree-path edge (p_1, p_2) in $\mathcal{P}$, we add an edge (u, a) to F^G where a is in T_v, the subtree dangling under v, that corresponds to the vertex incident to p_1 in the tree-path $T[p_1, p_2]$.*

- *Lifting portal routed edges: For every edge $e^F = (u, v)$ in F that maps to a portal routed edge $e^{\mathcal{P}} = (p_1, p_2)$, we add to F^G an edge (a, b) where $a \in T_u$ and $b \in T_v$ correspond to the endpoints of the pre-image of $e^{\mathcal{P}}$ respectively.*

The mapping $\Pi_{V(F^G)\mapsto V}$ is defined accordingly.

We show that bringing F to F^G increases the vertex congestion by a factor of 2. The constant factor blow-up is acceptable as the number of portals is a much smaller portion of $V(G)$ if we set the reduction parameter $k = \omega(1)$.

Lemma 5.7 (Vertex congestion of F^G)**.** *Under the setting of Definition 5.6, F^G is a forest and a flat embedding of G w.r.t. $\Pi_{V(F^G)\mapsto V}$. If every portal in $\mathcal{P}$ had vertex congestion γ from F to G,the mapping from F^G to G has vertex congestion 2γ.*

Proof. To see that F^G is a forest, one can contract each dangling subtree of F^G onto its portal vertex. The result is exactly F, which is a forest. Therefore, F^G must be a forest.

Next, we argue about the vertex congestion of F^G. For any non-portal vertex v and portal vertex p, v appears in T_p if $T[v, p]$ contains no other portal vertices. For v, this can only hold for at most two portals p_1, p_2 due to the branch-free-ness of the portal set. Because each portal appears at most γ times in F and F^G, u appears in F^G at most 2γ times, γ for each portals. □

After bringing the forest F maintained by $\mathcal{D}^{\mathrm{MRC}}$ back to G as F^G, we can bring back the cycle using F^G.

Lemma 5.8 (Lift a cycle from $\mathcal{P}$)**.** *Consider a graph $G = (V, E)$ with edge lengths $\boldsymbol{l}$ and gradients $\boldsymbol{g}$, a tree/forest T, a branch-free set of portals $P \subseteq V$, and a portal routed graph $\mathcal{P} \stackrel{\text{def}}{=} \mathcal{P}(G, T, P)$. Given a forest F and $\Pi_{V(F)\mapsto P}$ which are a lift of $\mathcal{P}$, and a cycle $\boldsymbol{c}$ on $\mathcal{P}$ which can be represented as γ tree paths on F and γ off-tree edges, the cycle $\Pi_{\mathcal{P}\mapsto G}(\boldsymbol{c})$ can be represented as γ tree paths on F^G and γ off-tree edges.*

In addition, we have $\langle \boldsymbol{g}^{\mathcal{P}}, \boldsymbol{c}\rangle = \langle \boldsymbol{g}, \Pi_{\mathcal{P}\mapsto G}(\boldsymbol{c})\rangle$ and $\|\boldsymbol{L}\Pi_{\mathcal{P}\mapsto G}(\boldsymbol{c})\| \le \|\boldsymbol{L}^{\mathcal{P}}\boldsymbol{c}\|_1$.

Proof. We can write $\boldsymbol{c}$ as

$$\boldsymbol{c} = F[p_1, p_2] \oplus (p_2, p_3) \oplus F[p_3, p_4] \oplus \ldots \oplus F[p_{2\gamma-1}, p_{2\gamma}] \oplus (p_{2\gamma}, p_1)$$

By Definition 5.6, $\Pi_{\mathcal{P}\mapsto G}$ maps each $F[p_i, p_{i+1}]$ part to $F^G[p_i, p_{i+1}]$. $\Pi_{\mathcal{P}\mapsto G}$ maps each (p_i, p_{i+1}) to $T[p_i, u] \oplus e \oplus T[v, p_{i+1}]$ where (p_i, p_{i+1}) is the portal routed edge of $e = (u, v) \in G$.

$\langle \boldsymbol{g}^{\mathcal{P}}, \boldsymbol{c}\rangle = \langle \boldsymbol{g}, \Pi_{\mathcal{P}\mapsto G}(\boldsymbol{c})\rangle$ comes from how we define $\boldsymbol{g}^{\mathcal{P}}$ and the fact that $\Pi_{\mathcal{P}\mapsto G}(\boldsymbol{c})$ remains a circulation. $\|\boldsymbol{L}\Pi_{\mathcal{P}\mapsto G}(\boldsymbol{c})\| \le \|\boldsymbol{L}^{\mathcal{P}}\boldsymbol{c}\|_1$ comes from triangle inequality and how we define $\boldsymbol{l}^{\mathcal{P}}$. The lemma follows. □

5.2 Dynamic Portal Routed Graphs

Using these definitions and properties, we are ready to state the lemma for the dynamic portal routed graph data structure.

Lemma 5.9 (Dynamic portal routed graphs)**.** *Given a size reduction parameter k, a dynamic tree/forest $T = (V, E_T)$ with a set of off-tree edges E_{off}, edge lengths $\boldsymbol{l}$ and gradients $\boldsymbol{g}$, there is a data structure that supports up to $O(m/k)$ number of updates of the following type:*

- *InsertTreeEdge*(e)/*DeleteTreeEdge*(e)*: adds/removes edge* e *to/from* T*.*
- *InsertOffTreeEdge*(e)/*DeleteOffTreeEdge*(e)*: adds/removes edge* e *to/from* E_{off}*.*
- *InsertVertex*(u)*: adds a new isolated vertex* u *to* V*.*
- *AddPortal*(u)*: adds* u *as a new portal vertex.*

In the beginning, the data structure initializes a set of portals P *of size* $O(m/k)$ *and a portal routed graph* $\mathcal{P}(G = (V, E_T \cup E_{\text{off}}), T, P)$*. After each update, the data structure adds at most* 4 *new portals and updates the portal routed graph* $\mathcal{P}$ *with* $O(1)$ *updates of the following type:*

- *InsertEdge*(e)/*DeleteEdge*(e)*: adds/removes edge* e *to/from* $\mathcal{P}$*.*
- *InsertVertex*(v, E_v)*: adds to* $\mathcal{P}$ *a set of edges* E_v *incident to* v *and adds* v *if it is new to* $\mathcal{P}$*.*
- *SplitAndMerge*(u, v, E_{move})*:* E_{move} *is a subset of edges incident to either* u *or* v*. This update adds a new vertex* w *to* $\mathcal{P}$ *and moves* E_{move} *to* w*, i.e., re-mapping endpoints of edges in* E_{move} *from either* u *or* v *to* w*. When* $u = v$*, this update simply splits the vertex* u*.*

Furthermore, suppose there is an algorithm $\mathcal{A}$ *that explicitly maintains a forest* F *and* $\Pi_{V(F)\mapsto P}$ *that are flat in* $\mathcal{P}$ *with maximum vertex congestion* γ_{vcong}*, i.e.,* $\mathcal{A}$ *outputs changes to* F *and* $\Pi_{V(F)\mapsto P}$ *after each update to* $\mathcal{P}$*. The data structure maintains* F^G *and* $\Pi_{V(F^G)\mapsto V}$ *(Definition 5.6). In particular, each edge update to* F *can be handled in* $m^{o(1)}$*-time and causes one edge update to* F^G*. Each vertex insertion/deletion to* F *can be handled in* $m^{o(1)}$*-time as well.*

The data structure takes $m^{1+o(1)}$*-time to initialize and* $k \cdot \gamma_{\text{vcong}} \cdot m^{o(1)}$*-time to handle each update.*

The data structure implementing Lemma 5.9 is presented as Algorithm 1. The key idea is that edge updates between portals can be directly passed to $\mathcal{P}$. We therefore first add u and v to P whenever edge e gets updated, and it suffices to implement the AddPortal operation efficiently. At initialization, the data structure first computes the set of portals P that decomposes the tree into roughly m/k pieces, each having k incident off-tree edges.

Adding a new portal only affects the portal routing of at most k off-tree edges and performs a split-and-merge operation on two vertices in $\mathcal{P}$. We also adds γ_{vcong} copies of subtrees dangling under the new portals to F^G, the flat forest in G. Therefore, explicitly writing down the updates to $\mathcal{P}$ takes only $O(k)$-time.

In this data structure, we use the tree decomposition tool from [ST03, ST04], which decomposes the tree into roughly m/k edge-disjoint pieces and each piece is adjacent to roughly k off-tree edges. This is a simple depth-first-search procedure over the tree and we formalized it as follows:

Lemma 5.10 (Tree decomposition, [ST04, Theorem 10.3])**.** *There is a deterministic linear-time algorithm that on a graph* $G = (V, E)$*, a rooted spanning tree* T*, and a reduction parameter* k*, outputs a decomposition* $\mathcal{W}$ *of* T *into edge-disjoint sub-trees such that:*

1. $|\mathcal{W}| = O(m/k)$.
2. $R \stackrel{\text{def}}{=} \partial\mathcal{W} \subseteq V$*, defined as the subset of vertices appear in multiple components, is branch-free.*

Algorithm 1: Dynamically maintain a portal routed graph $\mathcal{P}(G, T, P)$.

global variables
  T^{init}: The forest given initially.
  P: the set of portals.
  $\mathcal{P}$: the portal routed graph
  $\mathcal{A}$: the forest maintenance algorithm in $\mathcal{P}$.
  F: the flat forest in $\mathcal{P}$ maintained by a given algorithm $\mathcal{A}$.
  F^G: the flat forest in G
procedure *Initialize*$(T, E_{off}, \boldsymbol{l}, \boldsymbol{g}, k, \gamma_{\text{vcong}})$
  $T^{init} \leftarrow T$
  $\mathcal{W} \leftarrow$ TreeDecompose(G, T, k)
  $P \leftarrow \partial\mathcal{W}$
  $\mathcal{P} \leftarrow \mathcal{P}(G, T, P)$
  for $p \in P$ **do**
    Let T_p be the dangling subtree of p w.r.t. T^{init}.
    Add γ_{vcong} copies of dangling sub-trees $\{T_{p,1}, \ldots, T_{p,\gamma_{\text{vcong}}}\}$ to F^G.
  Initialize F using $\mathcal{A}$ and add the corresponding edges to F^G.
procedure *AddPortal*(u)
  if *adding u as a new portal makes P not branch-free w.r.t. T^{init}* **then**
    Let u^+ be the vertex such that $P \cup \{u, u^+\}$ is branch-free.
    AddPortal(u^+)
  Let E_u be the set of off-tree edges whose portal routing passes through u.
  if *u lies between two portals p_1 and p_2 in T^{init}* **then**
    Update all γ_{vcong} copies of dangling subtrees of T_{p_1} and T_{p_2}
    $\mathcal{P}$.SplitAndMerge$(p_1, p_2, E_u \cap (N_{\mathcal{P}}(p_1) \cup N_{\mathcal{P}}(p_2)))$
  else
    Let p be the portal closest to u.
    Update all γ_{vcong} copies of dangling subtrees of T_p
    $\mathcal{P}$.SplitAndMerge$(p, p, E_u \cap N_{\mathcal{P}}(p))$
  Update lengths and gradients for edges in E_u.
  $P \leftarrow P \cup \{u\}$
  $\mathcal{P}$.InsertVertex(u, E_u)
  Add γ_{vcong} copies of dangling sub-trees $\{T_{u,1}, \ldots, T_{u,\gamma_{\text{vcong}}}\}$ to F^G.
procedure *Update*(U)
  for $u \in V(U)$ **do**
    AddPortal(u)
  Pass the update to $\mathcal{P}$.
procedure *ForestUpdateEdge*$(e = (u, v))$
  Let i and j be the indices of the copies of u and v where the forest edge update happens in F.
  Update the corresponding edge between $T_{u,i}$ and $T_{v,j}$ in F^G

3. For every component $C \subseteq V$ of $\mathcal{W}$, the number of edges adjacent to non-boundary vertices of

C is at most $40k$.

In our implementation, we use the following claims regarding maintaining branch-free sets on a tree. The first claim shows that after adding a vertex to a branch-free set, we can maintain its branch-freeness by adding at most one additional vertex, which can be found efficiently using dynamic tree data structures.

Claim 5.11. *Given a tree/forest T and a branch-free set of vertices $P \subseteq V(T)$, for every vertex $u \in V(T)$, one can make $P \cup \{u\}$ branch-free by adding at most one additional vertex $u^+ \in V(T)$ to P.*

Proof. Imagine we root T at u. Then, we choose u^+ as the vertex farthest from u that has two children, each having a descendent in P. There can only be at most one such vertex u^+, as otherwise, P was not branch-free in the first place. □

The second claim shows that, if the portal set remains branch-free after adding a new portal, the portal routed graph undergoes a SplitAndMerge update as described in Lemma 5.9.

Claim 5.12. *Under the setting of Definition 5.3, given some vertex u such that $P \cup \{u\}$ remains branch-free, $\mathcal{P}(G, T, P \cup \{u\})$ can be obtained from $\mathcal{P}(G, T, P)$ with one SplitAndMerge operation followed by decreasing lengths of some edges incident to the u in $\mathcal{P}(G, T, P \cup \{u\})$ and at most two edge insertions and one deletion. The total number of affected edges whose length we decrease is $O(k)$.*

Proof. According to the definition of portal routed graphs (Definition 5.3), there are two types of (possibly affected) edges in $\mathcal{P}$.

- Affected tree-path edges: If u appears in between two portals p_1, p_2 in T such that $T[p_1, p_2]$ contains no other portals, we remove (p_1, p_2) and add two tree-path edges (p_1, u) and (u, p_2) to $\mathcal{P}$. This causes one edge deletions and two edge insertions.
- Affected portal routed edges: Let C be the component of $\mathcal{W}$, the decomposition of T induced by P, that contains u. One can observe every off-tree edge e whose portal routing $\mathcal{P}(e)$ touches u is incident to some vertex in C. The number of such edges is $O(k)$ by the description of the initialization of the portal set P and Lemma 5.10. Because P is branch-free, C has at most two boundary portals. Every portal routed edge incident to u in $\mathcal{P}(G, T, P \cup \{u\})$ was either incident to one of ∂C or not well-defined. Therefore, one can obtain the new portal routed graph with a SplitAndMerge operation from ∂C followed by inserting new edges/decreasing edge lengths around u.

□

Proof of Lemma 5.9. The correctness follows from Claim 5.11 and Claim 5.12.

We next analyze the runtime. Each update is handled in $k\gamma_{\text{vcong}} m^{o(1)}$-time because, by Claim 5.12, the number of affected tree-path and portal routed edges are $O(1)$ and $O(k)$ respectively. In addition, we add γ_{vcong} dangling subtrees for each new portal. Each dangling subtree has size $O(k)$. The bound on the update time follows.

The maintenance of F^G follows from Definition 5.6. In particular, whenever inserting/deleting an edge into/from F, we insert/delete the corresponding edge that lives in F^G. The vertex congestion bound follows from Lemma 5.7. □

5.3 Sparsified Portal Routed Graphs via Dynamic Low-Recourse Spanners

The other ingredient for the data structure of Theorem 3.4 is a dynamic spanner with low-recourse. We use the dynamic spanner to maintain a sparsifier $\widehat{G}$ of the portal routed graph $\mathcal{G}$ so that we run the given min-ratio cycle data structure $\mathcal{D}^{\mathrm{MRC}}$ on an instance of size roughly $1/k$ smaller. In addition, we want the sparsifier to handle updates to $\mathcal{P}$ such as SplitAndMerge and InsertVertex while keeping the recourse small. Naively treating these two operations as edge updates results in roughly k changes to $\widehat{G}$ and an update cost $\Omega(kT_{upd}(m/k))$ where T_{upd} is the amortized update time of $\mathcal{D}^{\mathrm{MRC}}$. Here we present a new dynamic spanner that handles both operations with low recourse and maintains an explicit embedding back to $\mathcal{P}$. See Section 7 for more discussion and technical details.

Theorem 5.13. *Given an m-edge n-vertex input graph $G = (V, E, l)$ with lengths in $[1, L]$, a degree threshold Δ such that G initially has maximum degree at most Δ, there is a data structure DynamicSpanner that supports a polynomially-bounded number of updates of the following type:*

- *InsertEdge(e): adds edge e to G, with the guarantee that $\deg^{\text{master}}(v) \leq \Delta$ in the graph G where $\deg_{\text{master}}$ is defined as the degree in G ignoring the deletions/splits.*
- *DeleteEdge(e): removes edge e from G*
- *SplitVertex$(v, E_{move}, E_{crossing})$: we assume that E_{move} is a set of edges incident to vertex v, and $E_{crossing}$ is a set of self-loops incident to v such that $E_{move} \cap E_{crossing} = \emptyset$.*

  The operation splits the vertex $v \in V$ into vertex v and a new vertex v'. It then moves all edges in E_{move} to v' by re-mapping all their endpoints from v to v'. Finally, it re-maps all edges in $E_{crossing}$ such that thereafter each such self-loop at v is mapped to an edge of the same length between v and v'. Returns a pointer to the new vertex v'.
- *InsertVertex(v, E_{inc}): Adds a vertex v to the graph G along with E_{inc} a set of edges that is incident on v.*

For $\gamma_{\text{spanner}} = e^{O(\log^{20/21} m \log\log m)}$, the algorithm maintains a subgraph $H \subseteq G$ and a graph embedding $\Pi_{G \mapsto H}$ such that

1. *at any time, for every $u, v \in V$, $\mathrm{dist}_G(u, v) \leq \mathrm{dist}_H(u, v) \leq \gamma_{\text{spanner}} \cdot \mathrm{dist}_G(u, v)$, and*
2. *at any time, H consists of at most $\widetilde{O}(n' \log L)$ edges where n' is the number of vertices in the final graph G, and*
3. *the total recourse of H, i.e., the total number of insertions/deletions and isolated vertex insertions to H, over a sequence of $\widehat{q}$ invocations of operations DeleteEdge, SplitVertex, InsertVertex and an arbitrary number of (legal) invocations of InsertEdge is at most $\gamma_{\text{spanner}}(n' + \widehat{q}) \log L$,*
4. *every edge $e = (u, v) \in E(G)$ is mapped to a uv-path $\Pi_{G \mapsto H}(e)$ consisting of at most γ_{spanner} many edges and ensures that at any time $\mathrm{econg}(\Pi_{G \mapsto H}) \leq \gamma_{\text{spanner}} \cdot \Delta \log L$. The number of embedding paths changed over a sequence of q updates to G is at most $\gamma_{\text{spanner}} \cdot \Delta \cdot q \log L$.*

The algorithm maintains H and $\Pi_{G \mapsto H}$ explicitly, is initialized in time $O(m)$ and thereafter processes each update in amortized time $\Delta \cdot \gamma_{\text{spanner}} \log L$.

When using Theorem 5.13 to maintain a sparsifier $\widehat{G}$ of the portal routed graph $\mathcal{P}$, the min-ratio cycle in $\mathcal{P}$ could disappear from $\widehat{G}$. In this case, we show that one of the *spanner cycles*, the cycle consists of an edge $e \in \mathcal{P}$ and its embedding path $\Pi_{\mathcal{P}\mapsto\widehat{G}}(e)$, has a comparable ratio.

Lemma 5.14 (Min-ratio cycle given spanner)**.** *Given a graph $G = (V, E)$ with edge lengths $\boldsymbol{l}$ and gradients $\boldsymbol{g}$, consider the spanner $H \subseteq G$ and $\Pi_{G\mapsto H}$ maintained by Theorem 5.13. Let $\boldsymbol{c}^H$ be an α-approximate min-ratio cycle in H and $\boldsymbol{c}^\Pi$ be the min-ratio cycle of the form $e \oplus \mathsf{rev}(\Pi_{G\mapsto H}(e)), e \in G$ (also denoted as $H[e]$, the* spanner cycle of e*). We have*

$$\min\left\{\frac{\langle \boldsymbol{g}, \boldsymbol{c}^H\rangle}{\|\boldsymbol{L}\boldsymbol{c}^H\|_1}, \frac{\langle \boldsymbol{g}, \boldsymbol{c}^\Pi\rangle}{\|\boldsymbol{L}\boldsymbol{c}^\Pi\|_1}\right\} \le \frac{1}{3\alpha\cdot\gamma_{spanner}} \min_{\boldsymbol{B}^\top\boldsymbol{\Delta}=0} \frac{\langle \boldsymbol{g}, \boldsymbol{\Delta}\rangle}{\|\boldsymbol{L}\boldsymbol{\Delta}\|_1}$$

That is, either $\boldsymbol{c}^H$ or $\boldsymbol{c}^\Pi$ is a $(3\alpha\gamma_{\text{spanner}})$-approximate min-ratio cycle in G.

Proof. Let $\boldsymbol{\Delta}^* \in \mathbb{R}^E$ be the min-ratio cycle on G. We first construct $\boldsymbol{\Delta}^*$ as a circulation on H by canceling out non-spanner edges. That is, consider

$$\boldsymbol{\Delta}^*_H \stackrel{\text{def}}{=} \boldsymbol{\Delta}^* - \sum_{e\in E\setminus H} \boldsymbol{\Delta}^*_e \mathbf{1}_{H[e]}$$

Observe that $\boldsymbol{\Delta}^*_H$ is supported on H because each non-spanner edge $e \in G \setminus H$ appears only in one spanner cycle, which is exactly $H[e]$. Furthermore, we can bound the length of $\boldsymbol{\Delta}^*_H$ using triangle inequality as follows:

$$\begin{aligned}
\|\boldsymbol{L}\boldsymbol{\Delta}^*_H\|_1 &\le \|\boldsymbol{L}\boldsymbol{\Delta}^*\|_1 + \sum_{e\in E\setminus H} |\boldsymbol{\Delta}^*_e| \left\|\boldsymbol{L}\mathbf{1}_{H[e]}\right\|_1 \\
&\le \|\boldsymbol{L}\boldsymbol{\Delta}^*\|_1 + \sum_{e\in E\setminus H} |\boldsymbol{\Delta}^*_e| \gamma_{\text{spanner}} \cdot \boldsymbol{l}_e \\
&\le 2\gamma_{\text{spanner}} \cdot \|\boldsymbol{L}\boldsymbol{\Delta}^*\|_1
\end{aligned}$$

where we use the fact that the length of $H[e]$ is at most $\gamma_{\text{spanner}}\boldsymbol{l}_e$.

Rearrangement yields that

$$\begin{aligned}
\boldsymbol{\Delta}^* &= \boldsymbol{\Delta}^*_H + \sum_{e\in E\setminus H} \boldsymbol{\Delta}^*_e \mathbf{1}_{H[e]}, \text{and} \\
\langle \boldsymbol{g}, \boldsymbol{\Delta}^*\rangle &= \langle \boldsymbol{g}, \boldsymbol{\Delta}^*_H\rangle + \sum_{e\in E} \boldsymbol{\Delta}^*_e \langle \boldsymbol{g}, \mathbf{1}_{H[e]}\rangle
\end{aligned}$$

Using the bound on the length of $\boldsymbol{\Delta}^*_H$, we have

$$\|\boldsymbol{L}\boldsymbol{\Delta}^*_H\|_1 + \sum_{e\in E\setminus H} |\boldsymbol{\Delta}^*_e| \left\|\boldsymbol{L}\mathbf{1}_{H[e]}\right\|_1 \le 3\gamma_{\text{spanner}} \cdot \|\boldsymbol{L}\boldsymbol{\Delta}^*\|_1$$

An averaging argument yields that

$$\begin{aligned}
\min\left\{\frac{\langle \boldsymbol{g}, \boldsymbol{\Delta}^*_H\rangle}{\|\boldsymbol{L}\boldsymbol{\Delta}^*_H\|_1}, \min_{e\in G\setminus H}\left\{\frac{\langle \boldsymbol{g}, \mathbf{1}_{H[e]}\rangle}{\left\|\boldsymbol{L}\mathbf{1}_{H[e]}\right\|_1}\right\}\right\} &\le \frac{\langle \boldsymbol{g}, \boldsymbol{\Delta}^*_H\rangle + \sum_{e\in E} \boldsymbol{\Delta}^*_e \langle \boldsymbol{g}, \mathbf{1}_{H[e]}\rangle}{\|\boldsymbol{L}\boldsymbol{\Delta}^*_H\|_1 + \sum_{e\in E\setminus H} |\boldsymbol{\Delta}^*_e| \left\|\boldsymbol{L}\mathbf{1}_{H[e]}\right\|_1} \\
&\le \frac{1}{3\gamma_{\text{spanner}}} \cdot \frac{\langle \boldsymbol{g}, \boldsymbol{\Delta}^*\rangle}{\|\boldsymbol{L}\boldsymbol{\Delta}^*\|_1}
\end{aligned}$$

That is, either the min-ratio cycle in H or the best spanner cycle is a $(3\gamma_{\text{spanner}})$-approximate min-ratio cycle in G. The lemma follows when considering any α-approximate solution in H. □

Now, we are ready to prove Theorem 3.4 using the dynamic portal routed graph (Lemma 5.9) and the dynamic spanner (Theorem 5.13). We first initialize and maintain a portal routed graph $\mathcal{P}$ using Lemma 5.9. Since the data structure can only deal with $O(m/k)$ updates, we rebuild the whole data structure after every m/k updates. Then, we use Theorem 5.13 to maintain $\widehat{G}$, a spanner of $\mathcal{P}$. The given min-ratio cycle data structure $\mathcal{D}^{\text{MRC}}$ is then used to maintain α-approximate min-ratio cycles on $\widehat{G}$. The cycle maintained by $\mathcal{D}^{\text{MRC}}$ is mapped back to G as discussed in Lemma 5.8.

When maintaining $\widehat{G}$ using the dynamic spanner, notice that $\mathcal{P}$ undergoes the SplitAndMerge operation, which is not supported in Theorem 5.13. However, we can mimic the operation with two vertex splits and map the two new vertices into the same one.

Unfortunately, the portal routed graph might have max degree as large as $\Omega(m)$ and it might lead to efficiency issues when using Theorem 5.13, whose update time depends on the max degree. To deal with the issue, we instead maintain a spanner H on an auxiliary graph $\mathcal{P}^{\text{aux}}$ which is flat in $\mathcal{P}$ and is obtained from $\mathcal{P}$ by split large degree vertices in the first place. We split those vertices in a way aligning with the decomposition of T using the portal set P so that adding a new portal only affects $O(1)$ vertices in $\mathcal{P}^{\text{aux}}$.

Definition 5.15 (Auxiliary portal routed graph $\mathcal{P}^{\text{aux}}$)**.** *Given a graph $G = (V, E)$ consisting of a tree T and a set of off-tree edges, consider a branch-free set of $O(m/k)$ portals P such that each component of $T \setminus P$ is incident to at most $O(k)$ off-tree edges. Let $\mathcal{W}$ be the edge-disjoint decomposition of T induced by P. We construct $\mathcal{P}^{\text{aux}}$ from $\mathcal{P} = \mathcal{P}(G, T, P)$ as follows: For each portal $p \in P$, we split it into p_{root} and p_C for each component $C \in \mathcal{W}$ that contains p. p_{root} is incident to the portal-routed edge whose pre-image is already incident to p. p_C is incident to the portal routed edges whose pre-image is incident to C.*

We do not include tree-path edges in $\mathcal{P}^{\text{aux}}$.

After adding a new portal, $\mathcal{P}^{\text{aux}}$ is still changed by a constant number of SplitVertex operations and a few edge updates.

Claim 5.16. *Given some vertex such that $P \cup \{u\}$ remains branch-free, $\mathcal{P}(G, T, P \cup \{u\})^{\text{aux}}$ is obtained from $\mathcal{P}(G, T, P)^{\text{aux}}$ with 6 SplitVertex operation followed by decreasing edge lengths incident to the newly created vertices and a constant number of edge deletion/insertions.*

Proof. Let $\mathcal{W}$ be the edge-disjoint decomposition of T induced by P. Let $C \in \mathcal{W}$ be the component containing u. After adding u as a new portal, C is further decomposed into two components C_1, C_2 where $u \in \partial C_1$ and $u \in \partial C_2$. In the new auxiliary portal routed graph $\mathcal{P}'$, we create three new vertices u_{root}, u_{C_1}, and u_{C_2}.

For any pre-existing portal $p \in \partial C$, we move some of its incident edge in $\mathcal{P}$ to u. In the $\mathcal{P}^{\text{aux}}$, they goes to either u_{root}, u_{C_1}, or u_{C_2}. As there are at most two such portals p, this creates 6 SplitVertex operations and we identify the 6 new vertices as u_{root}, u_{C_1}, or u_{C_2}. We also add some new edges whose portal routing contains only u but not any pre-existing portals. Then, as adding portals shortcuts portal routing and decreases edge lengths, we update the graph by decreasing some edge lengths incident to these three new vertices u_{root}, u_{C_1}, and u_{C_2}. □

Notice that $\mathcal{P}^{\text{aux}}$ can be maintained easily as we have Lemma 5.9 that maintains $\mathcal{P}$. We use Theorem 5.13 to maintain a spanner on $\mathcal{P}^{\text{aux}}$ which can be made a spanner on $\mathcal{P}$.

Now, we are ready to argue the correctness and performance of Algorithm 2 and prove Theorem 3.4.

Algorithm 2: Dynamic Min-Ratio Tree Cycle via Dynamic Min-Ratio Cycle

global variables
  $\mathcal{D}^{\mathcal{P}}$: dynamic portal routed graph data structure from Lemma 5.9
  $\mathcal{P}$: the portal routed graph maintained by $\mathcal{D}^{\mathcal{P}}$
  $\mathcal{D}^{\mathrm{MRC}}$: the given dynamic min-ratio cycle data structure
  F: the lift forest maintained by $\mathcal{D}^{\mathrm{MRC}}$
  F^G: the lift of F maintained by $\mathcal{D}^{\mathcal{P}}$
  $\mathcal{D}^{\mathrm{spanner}}$: the dynamic spanner data structure from Theorem 5.13
  $\widehat{G}$: the dynamic spanner of $\mathcal{P}$ maintained using $\mathcal{D}^{\mathrm{spanner}}$
procedure *Initialize*$(T, E_{off}, \boldsymbol{l}, \boldsymbol{g}, k)$
  $\mathcal{P} \leftarrow \mathcal{D}^{\mathcal{P}}$.Initialize$(T, E_{\mathrm{off}}, \boldsymbol{l}, \boldsymbol{g}, k)$
  Build the spanner of the portal routed edges of $\mathcal{P}^{\mathrm{aux}}$ as H^{aux} using $\mathcal{D}^{\mathrm{spanner}}$
  Let $\widehat{G}$ be the union of the tree-path edges of $\mathcal{P}$ and H obtained from H^{aux} by identifying vertices for the same portal as one.
  Initialize $\mathcal{D}^{\mathrm{MRC}}$ on $\widehat{G}$
  Use $\mathcal{D}^{\mathcal{P}}$ to maintain F^G, the lift of F in G
procedure *Update*(U)
  if *there have been* m/k *updates since the last initialization* **then**
    Initialize$(T, E_{\mathrm{off}}, \boldsymbol{l}, \boldsymbol{g}, k)$
  Pass the update U to $\mathcal{D}^{\mathcal{P}}$, which maintains $\mathcal{P}$.
  Pass the updates of $\mathcal{P}$ to $\mathcal{D}^{\mathrm{spanner}}$, which maintains H^{aux}.
  Pass the updates of $\widehat{G}$ to $\mathcal{D}^{\mathrm{MRC}}$.
  $\mathcal{D}^{\mathrm{MRC}}$ updates F and outputs a cycle $\boldsymbol{\Delta}$.
  $\mathcal{D}^{\mathcal{P}}$ maintains F^G according to the updates to F.
  Let $\boldsymbol{c}_{\mathrm{spanner}}$ be the best spanner cycle.
  $\boldsymbol{c}^G_{\mathrm{spanner}} \leftarrow \Pi_{\mathcal{P}\mapsto G}(\boldsymbol{c}_{\mathrm{spanner}})$
  $\boldsymbol{\Delta}^G \leftarrow \Pi_{\mathcal{P}\mapsto G}(\boldsymbol{\Delta})$
  Let $\boldsymbol{c}^G_{tree}$ be the best tree cycle $T_{\circlearrowleft}[e]$ that contains no portals.
  return *Best among* $\boldsymbol{c}^G_{\mathrm{spanner}}$, $\boldsymbol{c}^G_{tree}$ *and* $\boldsymbol{\Delta}^G$.

Proof of Theorem 3.4. First, we argue the correctness of the algorithm. Lemma 5.9 correctly maintains the portal routed graph $\mathcal{P}$ under updates. Claim 5.16 and Theorem 5.13 correctly maintains $\widehat{G}$ as a spanner of $\mathcal{P}$. Lemma 5.14 states that either $\boldsymbol{c}_{\mathrm{spanner}}$ or $\boldsymbol{\Delta}$ is a $(3\alpha\gamma_{\mathrm{spanner}})$-approximate min-ratio cycle on $\mathcal{P}$. Since we can correctly lift them back to G using Lemma 5.9 and Lemma 5.8, either $\boldsymbol{c}^G_{\mathrm{spanner}}$, $\boldsymbol{\Delta}^G$, or $\boldsymbol{c}^G_{tree}$ has ratio no worse than the min-ratio tree cycle in G up to a $(3\alpha\gamma_{\mathrm{spanner}})$ factor due to Lemma 5.5. This concludes the correctness of the algorithm.

The lift forest F^G has vertex congestion $\leq 2\gamma_{\mathrm{vcong}}$ due to Lemma 5.7. The representation of the output cycle is guaranteed by Lemma 5.8 and the fact that $\boldsymbol{c}^G_{tree}$ is also a tree cycle on F^G.

Next, we analyze the runtime. The initialization takes time $m\gamma_{\mathrm{spanner}} + T_{init}(m\gamma_{\mathrm{spanner}}/k)$ due to the initialization of the dynamic portal routed graph data structure $\mathcal{D}^{\mathcal{P}}$ (Lemma 5.9), the dynamic spanner $\mathcal{D}^{\mathrm{spanner}}$ (Theorem 5.13), and the given dynamic min-ratio cycle data structure on the graph $\widehat{G}$ which has size $m^{1+o(1)}/k$.

Finally, we analyze the amortized update time. Throughout the course of Q updates, the total

time spent on initialization is

$$\frac{Q}{m/k}\left(m\gamma_{\text{spanner}} + T_{init}\left(\frac{m\gamma_{\text{spanner}}}{k}\right)\right)$$

Total update time due to $\mathcal{D}^{\mathcal{P}}$ and $\mathcal{D}^{\text{spanner}}$ are, by Lemma 5.9 and Theorem 5.13, $O(Qk^2\gamma_{\text{vcong}})$ and $Qk\gamma_{\text{spanner}}$ respectively, as each update changes $\mathcal{P}$ and $\mathcal{P}^{\text{aux}}$ with $O(1)$ operations (Claim 5.12 and Claim 5.16). Across Q updates, the total number of updates to the sparsified portal routed graph $\widehat{G}$ is

$$\left(\frac{Q}{m/k}+1\right)\frac{m\gamma_{\text{spanner}}}{k} = \gamma_{\text{spanner}}\left(Q+\frac{m}{k}\right)$$

by Theorem 5.13 and the fact that $\mathcal{D}^{\text{spanner}}$ is initialized for $Q/(m/k)+1$ times. Each update to $\widehat{G}$ is handled by the given min-ratio data structure in $T_{upd}(m^{1+o(1)}/k)$-time. Therefore, for Q updates, the total update time is

$$\underbrace{\frac{Q}{m/k}\left(m\gamma_{\text{spanner}} + T_{init}\left(\frac{m\gamma_{\text{spanner}}}{k}\right)\right)}_{\text{total rebuild time}} + \underbrace{Qk^2\gamma_{\text{vcong}}\gamma_{\text{spanner}}}_{\text{total update time due to } \mathcal{D}^{\mathcal{P}} \text{ and } \mathcal{D}^{\text{spanner}}} + \underbrace{\gamma_{\text{spanner}}\left(Q+\frac{m}{k}\right)T_{upd}\left(\frac{m\gamma_{\text{spanner}}}{k}\right)}_{\text{total update time due to } \mathcal{D}^{\text{MRC}}}$$

Rearrangement yields

$$\frac{m\gamma_{\text{spanner}}}{k}T_{upd}\left(\frac{m\gamma_{\text{spanner}}}{k}\right) + Q\gamma_{\text{spanner}}\left(k^2\gamma_{\text{vcong}} + \frac{k}{m}T_{init}\left(\frac{m\gamma_{\text{spanner}}}{k}\right) + T_{upd}\left(\frac{m\gamma_{\text{spanner}}}{k}\right)\right)$$

We can charge the term independent of Q to the initialization and conclude the amortized update time bound. □

6 Fully-Dynamic Sparse Neighborhood Cover

In this section, we give an algorithm to maintain a sparse neighborhood cover (SNC) on a fully-dynamic graph. Our algorithm heavily builds on the framework obtained in [KMP23], but nonetheless, is nontrivial in composing various components from the paper.

Theorem 4.1 (Fully-dynamic sparse neighborhood cover)**.** *Given an m-edge constant-degree input graph $G = (V, E, l)$ with polynomially-bounded lengths in $[1, L]$ and a diameter parameter $D \geq 1$ there is a data structure that supports a polynomially bounded number of updates of the following type:*

- *InsertEdge(e)/DeleteEdge(e): inserts/deletes edge e to/from G where insertions preserve that G has constant degree.*

Under these updates, the data structure maintains a forest F, map $\Pi_{V(F)\mapsto V}$, and a subset $S \subseteq V(F)$ that satisfy the following properties, for some $\gamma_{SNC} = e^{O(\log^{41/42} m \log\log m)}$:

1. *F with map $\Pi_{V(F)\mapsto V}$ is a flat embedding of G.*
2. *For any vertex $v \in V$ there is a tree $T \in F$ such that*
$$B_G(v, D/\gamma_{SNC}) \subseteq \Pi_{V(F)\mapsto V}(S \cap V(T)).$$
3. *For any tree $T \in F$ we have that $\mathrm{diam}_F(S \cap V(T)) \leq \gamma_{SNC} \cdot D$.*
4. *The congestion satisfies $\mathrm{vcong}(\Pi_{V(F)\mapsto V}) \leq \gamma_{SNC}$.*

The forest F, map $\Pi_{V(F)\mapsto V}$, and subset $S \subseteq V(F)$ are all maintained explicitly, and each undergoes at most γ_{SNC} amortized changes per update. The algorithm is deterministic, initializes in time $m \cdot \gamma_{SNC}$, and has amortized update time γ_{SNC}.

6.1 Additional Results from [KMP23]

We will build our fully dynamic SNC by combining various pieces of [KMP23], who already designed an algorithm for maintaining a SNC under edge deletions only.

Theorem 6.1 (see [KMP23, Theorem 5.3]). *Given an m-edge constant-degree input graph $G = (V, E, l)$ with polynomially-bounded lengths in $[1, L]$ and a diameter parameter $D \geq 1$, there is a data structure DecrSNC that supports the following update:*

- *DeleteEdge(e): removes edge e from G.*

Under this update, the data structure maintains a set of partitions $\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_k$ for $k = O(\log m)$ such that for some $\gamma_{decrSNC} = e^{O(\log^{20/21} m \log\log m)}$:

1. *every vertex $v \in V$, there is some index $0 \leq i \leq k$, such that $B(v, D/\gamma_{decrSNC}) \subseteq C$ for some cluster $C \in \mathcal{P}_i$, and*
2. *for every $0 \leq i \leq k$, and cluster $C \in \mathcal{P}_i$, we have $\mathrm{diam}(G[C]) \leq D$, and*
3. *for every $0 \leq i \leq k$, the partition $\mathcal{P}_i$ is maintained such that for every such partition $\mathcal{P}_i$, we have that the sizes of all sets C that appear in $\mathcal{P}_i$ and are not a subset of a partition set in the previous version of $\mathcal{P}_i$ is at most $m \cdot \gamma_{decrSNC}$.*

The data structure is deterministic, reports each change to the partitions explicitly, and takes total time $m \cdot \gamma_{decrSNC}$ over any sequence of updates.

Next, we need a result that allows us to maintain a forest F which flatly embeds into each cluster C maintained by the decremental SNC of Theorem 6.1.

Theorem 6.2 (see [KMP23, Theorem 4.9]). *Given an m-edge input graph $G = (V, E, l)$ with polynomial lengths in $[1, L]$ and maximum degree 3. There is a data structure LowDiamTree that maintains a forest F that flatly embeds into G, and supports a polynomially-bounded number of updates of the following type:*

- *InsertEdge(e)/DeleteEdge(e): adds/removed edge e into/from G. If the edge is inserted, its associated length $l(e)$ has to be in $[1, L]$ and the maximum degree is not allowed to exceed 3; if it is deleted, it has to be ensured that thereafter graph G is still connected.*

Under these updates, the algorithm explicitly maintains F and $\Pi_{V(F)\mapsto V}$ that are a flat embedding into G, where l_F of F is defined by $l_F(e) = l(\Pi_{V(F)\mapsto V}(e))$ for every edge $e \in E(F)$, and a vertex map $\Pi_{V\mapsto V(F)}$ such that for $\gamma_{lowDiamTree} = e^{O(\log^{20/21} m \log\log m)}$, at any time:

1. $\mathrm{diam}_F(\Pi_{V\mapsto V(F)}(V)) \leq \gamma_{lowDiamTree} \cdot \mathrm{diam}(G)$, *and*
2. *we have* $\mathrm{vcong}(\Pi_{V(F)\mapsto V}) \leq \gamma_{lowDiamTree}$, *and*
3. *F consists of at most $\gamma_{lowDiamTree} \cdot m$ vertices and edges.*

The algorithm maintains the flat hierarchical forest F and all maps explicitly. Vertex maps are such that once an element is added to the preimage, its image remains fixed until the element is again removed.

The algorithm is deterministic, can be initialized in time $m \cdot \gamma_{lowDiamTree}$, and thereafter processes each edge insertion/deletion in amortized time $\gamma_{lowDiamTree}$.

Finally, we require an algorithm for maintaining a length vertex sparsifier H onto a set of terminals A. Additionally, given any procedure that maintains a forest that flatly embeds into H, the algorithm maintains a forest that flatly embeds into G which preserves distances between vertices in A.

Theorem 6.3 (see [KMP23, Theorem 4.12])**.** *Given an m-edge input graph $G = (V, E, l)$ with polynomial lengths in $[1, L]$ and maximum degree at most 3. Then, for some $\gamma_{vertexSparsifier} = e^{O(\log^{20/21} m \log\log m)}$, there is a data structure* MaintainVertexSparsifier *that initially outputs an empty set A, and graph H consisting of at most $\gamma_{vertexSparsifier}$ vertices and edges, and supports a polynomial number of updates of the following type:*

- InsertEdge(e)/DeleteEdge(e)*: adds/removed edge e into/from G. If the edge is inserted, its associated length $l(e)$ has to be in $[1, L]$ and the maximum degree of G is not allowed to exceed 3.*
- AddTerminalVertex(a)/RemoveTerminalVertex(a)*: adds/ removes the vertex $a \in V(G)$ to/from the terminal set A.*

The algorithm processes the t-th update and outputs a batch of updates $U_H^{(t)}$ consisting of edge insertions/deletions, and isolated vertex insertions/deletions and that when applied to the current vertex sparsifier H, yields the next one, such that at all times

- *we have $A \subseteq V(H) \subseteq V(G)$, and*
- *for all vertices $u, v \in V(H)$, we have $\mathrm{dist}_G(u, v) \leq \mathrm{dist}_H(u, v)$ and further if $u, v \in A$ then we also have $\mathrm{dist}_H(u, v) \leq \gamma_{vertexSparsifier} \cdot \mathrm{dist}_G(u, v)$, and*
- *the number of edges and vertices in H is at most $(1 + |A|) \cdot \gamma_{vertexSparsifier}$, and*
- *we have $\sum_{t' \leq t} |U_H^{(t')}| \leq \gamma_{vertexSparsifier} \cdot t$.*

The algorithm is deterministic, and initially takes time $m \cdot \gamma_{vertexSparsifier}$. Every update is processed in worst-case time $\gamma_{vertexSparsifier}$.

Further, say the algorithm is given as input a dynamic flat forest F over H and vertex maps $\Pi_{V(H)\mapsto V(F)}$, $\Pi_{V(F)\mapsto V(H)}$, along with parameters $\gamma_{congRep}$ and γ_{recRep} such that at any time the vertex congestion $\mathrm{vcong}(\Pi_{V(F)\mapsto V(H)})$ is bounded by $\gamma_{congRep}$ and the number of changes to F caused by an update to G is upper bounded by γ_{recRep}. We require the vertex maps to be such that whenever a vertex is added to the preimage, its image remains constant for the rest of the algorithm.

Then, the algorithm can maintain a flat forest F' over G along with vertex maps $\Pi_{V(H)\mapsto V(F')}$, $\Pi_{V(F')\mapsto V(G)}$ such that at any time $\mathrm{vcong}(\Pi_{V(F')\mapsto V(G)})$ is bounded by $\gamma_{congRep}\cdot\gamma_{vertexSparsifier}$ and the number of changes to F' per update to G is $\widetilde{O}(\gamma_{recRep}+\gamma_{congRep}\cdot\gamma_{vertexSparsifier})$, and we have for any two vertices $u,v\in V(H)$ that

$$l_G(\Pi_{V(F')\mapsto V(G)}(F'[\Pi_{V(H)\mapsto V(F')}(u),\Pi_{V(H)\mapsto V(F')}(v)]) \tag{4}$$

$$\leq\ l_H(\Pi_{V(F)\mapsto V(H)}(F[\Pi_{V(H)\mapsto V(F)}(u),\Pi_{V(H)\mapsto V(F)}(v)]). \tag{5}$$

Further, we have that the vertex maps are such that whenever a vertex is added to the preimage, its image remains constant for the rest of the algorithm.

Given that inputs F, $\Pi_{V(H)\mapsto V(F)}$, and $\Pi_{V(F)\mapsto V(H)}$ are maintained, the algorithm to maintain F', $\Pi_{V(H)\mapsto V(F')}$, and $\Pi_{V(F)\mapsto V(G)}$ requires additional initialization time $\widetilde{O}(m\cdot\gamma_{congRep})$ and processes every update with additional worst-case time $\widetilde{O}(\gamma_{recRep}+\gamma_{congRep}\cdot\gamma_{vertexSparsifier})$.

While [KMP23, Theorem 4.12] bounds edge congestion, we instead write it as a vertex congestion bound. These are equivalent because G has maximum degree 3.

6.2 Maintaining a Sparse Neighborhood Cover

For the algorithm, we assume that the number of updates to G to the data structure is at most m and the number of vertices is at most $2m$. These assumptions can be made without loss of generality as the general result can be obtained by simply rebuilding the data structure after m updates to G and by inserting edges with infinite lengths between connected components in G.

Data Structures. We first describe the algorithm to maintain the data structures required to maintain the hierarchical forest F, described in Theorem 4.1. We again use a standard batching technique over levels $0,1,\ldots,K$ where we take $K=\lceil\log^{1/42}m\rceil$.

Algorithm 3: INIT(G,K,D)

```
1 A_0 ← V; A_1, A_2, ..., A_K ← ∅.
2 foreach 0 ≤ i ≤ K do
3 |   Initialize a data structure H_i as described in Theorem 6.3 on the graph G where we let
  |     H_i denote the corresponding vertex sparsifier maintained by H_i.
4 |   InitLevel(i).
```

In Algorithm 3, we describe how to initialize the data structures. We start by initializing the sets $A_0,A_1,\ldots,A_K$ where we initialize A_0 to be the vertex set of G and all other sets to be empty. We then call for every level $0\leq i\leq K$, the procedure INITLEVEL(i) given in Algorithm 4. In this procedure, we initialize for level i three different data structures: a data structure $\mathcal{H}_i$ that maintains a vertex sparsifier H_i over the vertex set A_i; a data structure $\mathcal{D}_i$ that maintains a decremental sparse

neighborhood cover on $\widehat{H}_i$ where $\widehat{H}_i$ is the decremental version of H_i where insertions are simply ignored; and finally a data structure $\mathcal{T}_{i,j,C}$ that maintains for every cluster C in some partition of the sparse neighborhood cover a tree $T_{i,j,C}$ that spans the cluster C and has low diameter.

Algorithm 4: InitLevel(i)

1 Delete all current terminal vertices from the terminal vertex set maintained by $\mathcal{H}_i$ via the operation RemoveTerminalVertex($\cdot$). Then, add all vertices in $\widehat{A}_i$ as terminals to the data structure $\mathcal{H}_i$ via the operation AddTerminalVertex($\cdot$) where $\widehat{A}_i$ is taken to be the current set A_i.

2 Initialize data structure $\mathcal{D}_i$ as described in Theorem 6.1 on graph $\widehat{H}_i$ with parameter $D_i \stackrel{\text{def}}{=} D/(4\gamma_{decrSnc} \cdot \gamma_{vertexSparsifier})^{K-i+1}$ and let $\mathcal{P}_{i,0}, \mathcal{P}_{i,1}, \ldots, \mathcal{P}_{i,k}$ for some $k = O(\log m)$ denote the partitions maintained by this data structure where $\widehat{H}_i$ is initialized to H_i at the current time and then undergoes all deletions that H_i undergoes but not the insertions.

3 **foreach** $0 \leq j \leq k$ *and cluster* $C \in \mathcal{P}_{i,j}$ **do**

4 Initialize data structure $\mathcal{T}_{i,j,C}$ as described in Theorem 6.2 on the graph $\widehat{H}_{i,j,C}$ and maintain the hierarchical tree $T_{i,j,C}$, and embeddings $\Pi_{V(T_{i,j,C}) \mapsto C}$ and $\Pi_{C \mapsto V(T_{i,j,C})}$, where $\widehat{H}_{i,j,C}$ is initialized to the current graph $\widehat{H}_i[C]$ and then undergoes all deletions that $\widehat{H}_i[C]$ undergoes.

The maintenance of all of these objects is given in Algorithm 5 explains how to process the t-th update to G. The procedure forwards changes to G to all data structures at every level, updating the vertex sparsifier, partitions, and the low-diameter spanning trees to those on the updated graph. However, it handles only the decremental updates, i.e. deletions of edges and vertices. To handle insertions, it adds the endpoints of all inserted edges and vertices to all sets $A_{i+1}, A_{i+2}, \ldots, A_K$ at higher levels (see Line 4).

Whenever too many insertions occur at a certain level, the data structures at the level are rebuilt (see the if-statement starting in Line 13).

Maintaining the objects in Theorem 4.1. We discuss how to take the above Algorithms 3 to 5 and use them to maintain $F, \Pi_{V(F) \to V}$, and subset S as requierd by Theorem 4.1. Then we analyze the overall construction in the next section (Section 6.3). For a cluster $C \in \mathcal{P}_{i,j}$, we maintain a vertex $\pi(i,j,C) \in C \cap A_{i+1}$ if it exists, and otherwise set $\pi(i,j,C) = \perp$. For $i = K, K-1, \ldots, 0$, we will define the flat forest F_i on G for terminals A_i (which comes with a map $\Pi_{V(F_i) \mapsto V(G)}$), and a subset $S_i \subseteq V(F_i)$, which will contain duplicates of vertices in A_i (at most $m^{o(1)}$ times). Ultimately we will set $S = S_0$, $F = F_0$, and $\Pi_{V(F) \mapsto V(G)}$ as $\Pi_{V(F_0) \mapsto V(G)}$. For the base case, let F_{K+1} and S_{K+1} be empty sets.

To go from level $i+1$ to i, first use Theorem 6.3 with F as the disjoint union of $T_{i,j,C}$ with maps $\Pi_{V(T_{i,j,C}) \to V(\widehat{H}_i)}$ (which is a flat embedding of $\widehat{H}_i$ by Theorem 6.2), to get embeddings $\Pi_{V(T'_{i,j,C}) \to V(G)}$ for some flat trees $T'_{i,j,C}$ over G.

Initialize $F_i = F_{i+1}$. Now for each $C \in \mathcal{P}_{i,j}$ for $j = 0, 1, \ldots, k$ add a disjoint copy of $T'_{i,j,C}$ to F_i. If $C \cap A_{i+1} \neq \emptyset$, then $\pi(i,j,C) \neq \perp$. Now, for every vertex $s \in S_{i+1}$ corresponding to a duplicate of $\pi(i,j,C)$, take vertex $\Pi_{V(\widehat{H}_i) \mapsto V(T'_{i,j,C})}(\pi(i,j,C)) \in V(T'_{i,j,C})$ and merge it with F_{i+1} at vertex

Algorithm 5: UPDATE(t)

Let e be the edge in G affected by the t-th update.
foreach $0 \leq i \leq K$ **do**
  Forward the t-th update to G to the data structure $\mathcal{H}_i$ that updates graph H_i via the update batch $U_{H_i}^{(t)}$.
  Add the endpoints of e and all edges and vertices affected by the update in H_i to the sets $A_{i+1}, A_{i+2}, \ldots, A_K$ (recall that $V(H_i) \subseteq V(G)$, as stated in Theorem 6.3).
  Forward the deletions in $U_{H_i}^{(t)}$ to the data structure $\mathcal{D}_i$ that maintains the partitions $\mathcal{P}_{i,0}, \mathcal{P}_{i,1}, \ldots, \mathcal{P}_{i,k}$. We denote by $\mathcal{P}_{i,0}^{OLD}, \mathcal{P}_{i,1}^{OLD}, \ldots, \mathcal{P}_{i,k}^{OLD}$ the partitions before the update was processed and by $\mathcal{P}_{i,0}^{NEW}, \mathcal{P}_{i,1}^{NEW}, \ldots, \mathcal{P}_{i,k}^{NEW}$ the ones obtained after the update.
  foreach $0 \leq j \leq k$ *and cluster* $C \in \mathcal{P}_{i,j}^{OLD}$ **do**
    Forward the deletions in $U_{H_i}^{(t)}$ to the data structure $\mathcal{T}_{i,j,C}$ that maintains the flat tree $T_{i,j,C}$ and embeddings $\Pi_{V(T_{i,j,C}) \mapsto C}$, $\Pi_{C \mapsto V(T_{i,j,C})}$.
  foreach $0 \leq j \leq k$ *and cluster* $C \in \mathcal{P}_{i,j}^{NEW} \setminus \mathcal{P}_{i,j}^{OLD}$ **do**
    if *there is a cluster* C' *in partition* $\mathcal{P}_{i,j}^{OLD}$ *with* $C \subseteq C'$ *and* $|C| > |C'|/2$ **then**
      Let $\mathcal{T}_{i,j,C}$ refer to the data structure $\mathcal{T}_{i,j,C'}$ after deleting all edges incident to vertices in $C' \setminus C$ from the data structure; let $T_{i,j,C}$ denote the corresponding tree.
    else
      Initialize data structure $\mathcal{T}_{i,j,C}$ as described in Theorem 6.2 on the graph $\widehat{H}_{i,j,C}$ and maintain the flat tree $T_{i,j,C}$ and embeddings $\Pi_{V(T_{i,j,C}) \mapsto C}$, $\Pi_{C \mapsto V(T_{i,j,C})}$ where $\widehat{H}_{i,j,C}$ is initialized to the current graph $\widehat{H}_i[C]$ and then undergoes all deletions that $\widehat{H}_i[C]$ undergoes.
if $\exists i$ *s.t.* $0 \leq i < K$ *and* $|A_{i+1}| \geq m^{1-(i+1)/K}$ **then**
  Let i be the smallest index that satisfies the if-condition.
  $A_{i+1}, A_{i+2}, \ldots, A_K \leftarrow \emptyset$.
  foreach $i \leq j \leq K$ **do** INITLEVEL(j). ;

$s \in V(F_{i+1})$. Finally, define S_i to be the union of all copies of vertices in A_i in the copies of $T'_{i,j,C}$, which we can tell by the maps $\Pi_{V(H) \to V(T'_{i,j,C})}$ given by Theorem 6.3.

6.3 Analysis of Fully-Dynamic SNC Algorithm

We start by showing that the ball of radius D/γ_{SNC} around each $v \in V(G)$ is covered by our construction. The remaining aspects of Theorem 4.1 follow more directly by our construction. We start by establishing some auxiliary claims.

Claim 6.4. *Recall that* $D_i \stackrel{\text{def}}{=} D/(4 \cdot \gamma_{decrSnc} \cdot \gamma_{vertexSparsifier})^{K-i+1}$. *Then, for any vertex* $v \in V(G)$, *for every* $0 \leq i \leq K$, *then*

- $A_i \cap B_G(v, D_i/(2\gamma_{decrSnc}\gamma_{vertexSparsifier})) \subseteq C$ *for some cluster* $C \in \mathcal{P}_{i,j}$ *for some* $0 \leq j \leq k$, *or*

- *for each vertex $w \in A_i \cap B_G(v, D_i/(2\gamma_{decrSnc}\gamma_{vertexSparsifier}))$, there is a cluster $C \in \mathcal{P}_{i,j}$ for some $0 \leq j \leq k$ that contains w, and $C \cap A_{i+1} \neq \emptyset$ (we define $A_{K+1} = \emptyset$).*

Proof. The proof is trivial for the case where $A_i \cap B_G(v, D_i) = \emptyset$. Therefore, let us assume that there is at least one such vertex.

We have for every level $0 \leq i \leq K$ that the data structure $\mathcal{D}_i$ as described in Theorem 6.1, with parameter D_i, maintains a collection of partitions $\mathcal{P}_{i,0}, \mathcal{P}_{i,1}, \ldots, \mathcal{P}_{i,k}$ of decremental graph $\widehat{H}_i$ such that for every vertex $w \in V(\widehat{H}_i)$, there is an index $0 \leq j \leq k$, such that $B_{\widehat{H}_i}(w, D_i/\gamma_{decrSnc}) \subseteq C$ for some $C \in \mathcal{P}_{i,j}$.

Here, as can be seen from how $\mathcal{D}_i$ is initialized (see Line 2) and maintained (see Line 4), that $\widehat{H}_i$ differs from H_i in the way that $\widehat{H}_i$ was only updated by the deletions to H_i since $\mathcal{D}_i$ was initialized, and thus insertions to H_i since, are not present in $\widehat{H}_i$. On the other hand, from how we update the set A_{i+1} (see Line 4 and Line 13), we have that every endpoint of an inserted edge to H_i since $\mathcal{D}_i$ was initialized is added to set A_{i+1}.

Now, let w be an arbitrary vertex in $A_i \cap B_G(v, D_i/(2\gamma_{decrSnc}\gamma_{vertexSparsifier}))$. We have that every vertex $w' \in A_i \cap B_G(v, D_i/(2\gamma_{decrSnc}\gamma_{vertexSparsifier}))$ satisfies $\text{dist}_{H_i}(w, w') \leq \gamma_{vertexSparsifier} \cdot \text{dist}_G(w, w') \leq \gamma_{vertexSparsifier} \cdot (\text{dist}_G(w, v) + \text{dist}_G(v, w')) \leq D_i/\gamma_{decrSnc}$.

Now, we either have that $B_{\widehat{H}_i}(w, D_i/\gamma_{decrSnc}) = B_{H_i}(w, D_i/\gamma_{decrSnc})$ in which case, we can conclude from Theorem 6.1 that there indeed is a cluster C in some partition $\mathcal{P}_{i,j}$ that contains all vertices $w' \in A_i \cap B_G(v, D_i/(2\gamma_{decrSnc}\gamma_{vertexSparsifier}))$.

On the other hand, if $B_{\widehat{H}_i}(w, D_i/\gamma_{decrSnc}) \neq B_{H_i}(w, D_i/\gamma_{decrSnc})$, then we have that there was an insertion to H_i with at least one of the endpoints in the ball $B_{\widehat{H}_i}(w, D_i/\gamma_{decrSnc})$. This implies that $B_{\widehat{H}_i}(w, D_i/\gamma_{decrSnc}) \cap A_{i+1} \neq \emptyset$. And therefore, we have from Theorem 6.1 that there indeed is a cluster C in some partition $\mathcal{P}_{i,j}$ that contains $B_{\widehat{H}_i}(w, D_i/\gamma_{decrSnc})$ and thus $C \cap A_{i+1} \neq \emptyset$, as desired. □

Claim 6.5. *Recall that $D_i \stackrel{\text{def}}{=} D/(4 \cdot \gamma_{decrSnc} \cdot \gamma_{vertexSparsifier})^{K-i+1}$. For every vertex $v \in V(G)$ and $0 \leq i \leq K$, there is a tree $T \in F_i$ such that*

$$B_G(v, D_i/(2\gamma_{decrSnc}\gamma_{vertexSparsifier})) \cap A_i \subseteq \Pi_{V(F_i) \to V(G)}(S_i \cap V(T)).$$

Proof. We prove the claim by induction. Fix any vertex $v \in V(G)$ and level $0 \leq i \leq K$. We have from Claim 6.4 that we are in one of two scenarios: in the first scenario, we have that $B_G(v, D_i/(2\gamma_{decrSnc}\gamma_{vertexSparsifier})) \cap A_i \subseteq C$ for some $C \in \mathcal{P}_{i,j}$ for some $0 \leq j \leq k$. And further note that we have a data structure $\mathcal{T}_{i,j,C}$ that maintains a hierarchical tree $T_{i,j,C}$ over C on graph $\widehat{H}_{i,j,C} = \widehat{H}[C]$, and that a disjoint copy of $T_{i,j,C}$ is in F_i by construction. Because S_i contains all the vertices in A_i among $T_{i,j,C}$, the claim follows for $T = T_{i,j,C}$.

Otherwise, we have $i < K$, and that for every $w \in A_i \cap B_G(v, D_i/(2\gamma_{decrSnc}\gamma_{vertexSparsifier}))$, there is a cluster $C_w \in \mathcal{P}_{i,j}$ for some $0 \leq j \leq k$, such that $C_w \cap A_{i+1} \neq \emptyset$. But note that each such cluster C_w is thus represented by vertex $v_w = \pi(i, j, C_w) \in A_{i+1}$. Since $\text{diam}(\widehat{H}_i[C_w]) \leq D_i$ by Theorem 6.1, $\widehat{H}_i \subseteq H_i$, and distances in H_i dominate distances in G by Theorem 6.3, we have $\text{dist}_G(w, v_w) \leq D_i$. Thus, all such vertices v_w over all $w \in A_i \cap B_G(v, D_i/(2\gamma_{decrSnc}\gamma_{vertexSparsifier}))$ have distance at most $D_i + D_i/(2\gamma_{decrSnc}\gamma_{vertexSparsifier}) \leq 2 \cdot D_i = D_{i+1}/(2\gamma_{decrSnc}\gamma_{vertexSparsifier})$ from v in graph G. Thus by the inductive hypothesis, there is at least one tree T in forest F_{i+1} such that all such vertices v_w are contained in $\Pi_{V(F_{i+1}) \mapsto V(G)}(S_{i+1} \cap V(T))$.

Thus, in F_i, we have the tree T where for each $w \in A_i \cap B_G(v, D_i/(2\gamma_{decrSnc}\gamma_{vertexSparsifier}))$ (and possibly some other vertices), we attach to every node $x \in V(T)$ that is identified with v_w,

the tree T'_{i,j,C_w} for the $0 \le j \le k$ where $C_w \in \mathcal{P}_{i,j}$. Thus, the resulting tree $T' \supseteq T$ in F_i has its node set $V(T')$ mapped to a set of vertices in $V(G)$ that contains vertices in $\cup_w C_w \supseteq B_G(v, D_i/(2\gamma_{decrSnc}\gamma_{vertexSparsifier}))$, as desired. □

The following Corollary now follows from Claim 6.5 and the fact that $A_0 = V$ at all times.

Corollary 6.6. *For every vertex $v \in V(G)$, there is a tree $T \in F$ such that $B_G(v, D/(4 \cdot \gamma_{decrSnc} \cdot \gamma_{vertexSparsifier})^{K+2}) \subseteq \Pi_{V(F)\mapsto V(G)}(S \cap V(T))$.*

We establish the Properties listed in Theorem 4.1.

Claim 6.7. *The data structure maintains a forest F along with graph embedding $\Pi_{V(F)\mapsto V}$ and $S \subseteq V(F)$ such that for some $\gamma_{snc} = e^{O(\log^{41/42} m \log\log m)}$, at any time:*

1. *F with map $\Pi_{V(F)\mapsto V}$ is a flat embedding of G.*
2. *For any vertex $v \in V$ there is a tree $T \in F$ such that*
$$B_G(v, D/\gamma_{SNC}) \subseteq \Pi_{V(F)\mapsto V}(S \cap V(T)).$$
3. *For any tree $T \in F$ we have that $\mathrm{diam}_F(S \cap V(T)) \le \gamma_{SNC} \cdot D$.*
4. *The congestion satisfies $\mathrm{vcong}(\Pi_{V(F)\mapsto V}) \le \gamma_{SNC}$.*

Proof. Let us establish the properties one by one:

1. This follows by induction. Indeed, assume by induction that F_{i+1} flatly embeds into G. Now, each $T_{i,j,C}$ and $T'_{i,j,C}$ flatly embeds into G by Theorem 6.2 and Theorem 6.3 (we got $T'_{i,j,C}$ by mapping from $\widehat{H}_i$ to G). Finally, we attach $T'_{i,j,C}$ to F_{i+1} at matching vertices by construction.
2. The first property follows immediately from Corollary 6.6 and by picking $\gamma_{snc} \ge (4 \cdot \gamma_{decrSnc} \cdot \gamma_{vertexSparsifier})^{K+2}$.
3. By Theorem 6.1, Theorem 6.2, and the mapping back procedure from Theorem 6.3, we know:
$$\begin{aligned}\mathrm{diam}_{T'_{i,j,C}}(\Pi_{V(\widehat{H}_i)\mapsto V(T'_{i,j,C})}(C)) &\overset{(a)}{\le} \mathrm{diam}_{T_{i,j,C}}(\Pi_{C\to V(T_{i,j,C})}(C)) \\ &\overset{(b)}{\le} \gamma_{lowDiamTree}\mathrm{diam}(\widehat{H}_i[C]) \le \gamma_{lowDiamTree} D_i,\end{aligned}$$
where (a) follows by the construction of $T'_{i,j,C}$ and Theorem 6.3 (5), and (b) follows from Theorem 6.2. Because each F_i was gotten from F_{i+1} by attaching trees to vertices of F_i, we get that for any $T_i \in F_i$ that contains $T_{i+1} \in F_{i+1}$,
$$\mathrm{diam}_{F_i}(S_i \cap V(T_i)) \le 2\gamma_{lowDiamTree} D_i + \mathrm{diam}_{F_{i+1}}(S_{i+1} \cap V(T_{i+1})),$$
by the fact that we can go from $u \in T_{i,j,C}$ to $v \in T_{i',j',C'}$ via $u \to \pi(i,j,C) \to \pi(i',j',C') \to v$, and that $\pi(i,j,C), \pi(i',j',C')$ are attached to F_{i+1} at vertices in S_{i+1}. Iterating this shows that $\mathrm{diam}_F(S \cap V(T)) \le \sum_{i=0}^{K} 2\gamma_{lowDiamTree} D_i \le 4\gamma_{lowDiamTree} D_i$.

4. By Theorems 6.2 and 6.3 and the fact that there are $k+1$ partitions in the decremental SNC, the vertex congestion increases by a factor of at most $O((k+1)\gamma_{lowDiamTree}\gamma_{vertexSparsifier})$ from F_{i+1} to F_i, because we connect a copy of each $T'_{i,j,C}$ to everything in S_i. Thus, the total vertex congestion is at most $O((k+1)\gamma_{lowDiamTree}\gamma_{vertexSparsifier})^K$.

□

Finally, we bound the total runtime to maintain the forest F and the embedding $\Pi_{F\mapsto G}$.

Claim 6.8. *For some reasonably chosen $\gamma_{snc} = e^{O(\log^{41/42} m \log\log m)}$, the algorithm can be initialized in time $m \cdot \gamma_{snc}$, and thereafter processes each edge insertion/deletion in amortized time γ_{snc}.*

Proof. We start by bounding the number of changes to each set A_i by $m \cdot (5\gamma_{vertexSparsifier})^i$. We prove by induction on i. For the base case, we have that A_0 initially has all vertices added and is thereafter not updated at any point of the algorithm.

For the inductive step, observe that the set A_{i+1} can undergo insertions and deletions. We upper bound the number of vertex insertions to A_{i+1} by $\frac{1}{2}m \cdot (5\gamma_{vertexSparsifier})^{i+1}$ which suffices since the number of deletions from A_{i+1} is at most the number of insertions. We first note that A_{i+1} undergoes at most 2 vertex insertions for every edge update to sparsifier H_j for $j < i+1$ (see Line 4). But we have from Theorem 6.3, that each sparsifier H_j undergoes at most $\gamma_{vertexSparsifier}$ changes for every time the underlying graph G is changed (which occurs at most m times), or when a terminal is added or removed from A_j (which occurs at most $m \cdot (5\gamma_{vertexSparsifier})^j$ times by the induction hypothesis. Thus, the total number of insertions to A_{i+1} can be upper bounded by

$$\begin{aligned}\sum_{j<i+1} \gamma_{vertexSparsifier}\cdot(m + m \cdot (5\gamma_{vertexSparsifier})^j) \\ &\leq (i+1)\gamma_{vertexSparsifier} + 2 \cdot 5^i m \cdot (\gamma_{vertexSparsifier})^{i+1} \\ &< \frac{1}{2} \cdot (5 \cdot \gamma_{vertexSparsifier})^{i+1}\end{aligned}$$

where we use that we have a geometric sum and that $\gamma_{vertexSparsifier}$ is sufficiently large.

Next, we observe that whenever we enter the if-statement in Line 13, where i is selected as the smallest index, we have that the if-statement sets A_{i+1} to the empty set which means that for each such if-statement, we have that at least $m^{1-(i+1)/K}$ updates to A_{i+1} occur which bounds the number of times that the if statement is executed with smallest index i by

$$m \cdot (5\gamma_{vertexSparsifier})^{i+1}/m^{1-(i+1)/K} = m^{(i+1)/K} \cdot (5\gamma_{vertexSparsifier})^{i+1}.$$

Finally, we have from the if-condition that whenever we enter the if-statement and restart a level $j \geq i$ where i is the smallest index for which the if-condition holds, for all such levels the set A_j is of size at most $m^{1-j/K}$ by minimality of i (and the fact that $A_0 = V$).

From the number of restarts of each level and the upper bound on the size of A_i at any time when any data structure is rebuilt, the number of amortized recourse to the sets A_i and the graphs H_i, we can then bound the total update time of all data structures $\mathcal{H}_i, \mathcal{D}_i$ and all data structures $\mathcal{T}_{i,j,C}$ from Theorem 6.1, Theorem 6.2 and Theorem 6.3 by

$$\begin{aligned}m^{1+1/K} \cdot (5\gamma_{vertexSparsifier})^{O(K)} \cdot \gamma_{lowDiamTree} \cdot \gamma_{decrSnc} &= m^{1+1/K}(e^{O(\log^{20/21}(m)\log\log m)})^{O(K)} \\ &= m \cdot e^{O(\log^{41/42}(m)\log\log m)}\end{aligned}$$

where we implicitly use a halving trick for the data structures $\mathcal{T}_{i,j,C}$ which is necessary because sometimes data structures $\mathcal{T}_{i,j,C}$ are newly initialized in Line 12 but each edge in $\widehat{H}_i$ participates in at most $O(\log m)$ such rebuilds as its new cluster C then is incident to at most half the number of edges as its old cluster was.

Finally, in order to maintain the flat forest F, embedding $\Pi_{V(F)\mapsto V(G)}$, and set S, the runtime increases by a factor of

$$O((k+1)\cdot\gamma_{lowDiamTree}\cdot\gamma_{vertexSparsifier})^{O(K)} = e^{O(\log^{41/42}(m)\log\log m)}$$

due to vertex congestion. Thus the total time is $m\cdot e^{O(\log^{41/42}(m)\log\log m)}$ as desired. □

7 Fully-Dynamic Low-Recourse Spanner

In this section, we prove that given a graph $G=(V,E,l)$ undergoing edge deletions and insertions, vertex splits and vertex insertions (possibly non-isolated), we can maintain a spanner H of G with low recourse per update to G. Our result is summarized by the theorem below.

Theorem 5.13. *Given an m-edge n-vertex input graph $G=(V,E,l)$ with lengths in $[1,L]$, a degree threshold Δ such that G initially has maximum degree at most Δ, there is a data structure* DynamicSpanner *that supports a polynomially-bounded number of updates of the following type:*

- InsertEdge(e)*: adds edge e to G, with the guarantee that $\deg^{\text{master}}(v)\le\Delta$ in the graph G where $\deg_{\text{master}}$ is defined as the degree in G ignoring the deletions/splits.*
- DeleteEdge(e)*: removes edge e from G*
- SplitVertex$(v, E_{move}, E_{crossing})$*: we assume that E_{move} is a set of edges incident to vertex v, and $E_{crossing}$ is a set of self-loops incident to v such that $E_{move}\cap E_{crossing}=\emptyset$.*

  The operation splits the vertex $v\in V$ into vertex v and a new vertex v'. It then moves all edges in E_{move} to v' by re-mapping all their endpoints from v to v'. Finally, it re-maps all edges in $E_{crossing}$ such that thereafter each such self-loop at v is mapped to an edge of the same length between v and v'. Returns a pointer to the new vertex v'.
- InsertVertex(v, E_{inc})*: Adds a vertex v to the graph G along with E_{inc} a set of edges that is incident on v.*

For $\gamma_{\text{spanner}}=e^{O(\log^{20/21} m\log\log m)}$, the algorithm maintains a subgraph $H\subseteq G$ and a graph embedding $\Pi_{G\mapsto H}$ such that

1. *at any time, for every $u,v\in V$, $\text{dist}_G(u,v)\le\text{dist}_H(u,v)\le\gamma_{\text{spanner}}\cdot\text{dist}_G(u,v)$, and*
2. *at any time, H consists of at most $\widetilde{O}(n'\log L)$ edges where n' is the number of vertices in the final graph G, and*
3. *the total recourse of H, i.e., the total number of insertions/deletions and isolated vertex insertions to H, over a sequence of $\widehat{q}$ invocations of operations* DeleteEdge, SplitVertex, InsertVertex *and an arbitrary number of (legal) invocations of* InsertEdge *is at most $\gamma_{\text{spanner}}(n'+\widehat{q})\log L$,*

4. *every edge $e = (u, v) \in E(G)$ is mapped to a uv-path $\Pi_{G \mapsto H}(e)$ consisting of at most γ_{spanner} many edges and ensures that at any time $\text{econg}(\Pi_{G \mapsto H}) \leq \gamma_{\text{spanner}} \cdot \Delta \log L$. The number of embedding paths changed over a sequence of q updates to G is at most $\gamma_{\text{spanner}} \cdot \Delta \cdot q \log L$.*

The algorithm maintains H and $\Pi_{G \mapsto H}$ explicitly, is initialized in time $O(m)$ and thereafter processes each update in amortized time $\Delta \cdot \gamma_{\text{spanner}} \log L$.

We develop an incremental spanner that we use in conjunction with the batching scheme obtaining a spanner in a decremental graph developed by [CKL+22, BCK+23a].

7.1 Maintaining a Spanner under Edge Insertions

In this section, we prove the following theorem by giving an algorithm that maintains a spanner H with very low recourse for a graph G undergoing edge insertions. It additionally maintains an embedding Π of the edges in G to paths in H between the same endpoints. We rely on G having small maximum degree at all times to obtain an efficient algorithm.

In the next section, we crucially rely on this result to build our fully-dynamic spanner. To be of use, we need our algorithm to also take vertex splits and deletions under consideration. That is, while we do not require the current spanner and embedding to be adjusted to a vertex split/deletion, we require all embedding paths added in the future to take the vertex split/deletion into account.

Even in this setting, vertex splits are quite challenging to deal with. If split vertices are allowed to reach the maximum degree threshold again, this causes problems for our algorithm. We thus introduce the following definition that formalizes how we further constrain the update sequence.

Definition 7.1 (Master node)**.** *We initially associate each vertex v in the initial vertex set V of G with a unique master node $v = \text{master}(v)$. Then, whenever v is split into v_1 and v_2, we let $\text{master}(v_1) = \text{master}(v)$ and $\text{master}(v_2) = \text{master}(v)$. For every $v \in V$, we define $\deg^{\text{master}}(v) = \sum_{u \in V_G : \text{master}(u) = v} \deg(u)$ and call this the master node degree of v.*

Before we state our result, we also define the length of an embedding path.

Definition 7.2. *(Embedding length) Given an embedding $\Pi_{G \mapsto H}$ for some graphs G and H, we let $\text{length}(\Pi_{G \mapsto H}) = \max_e |\Pi_{G \mapsto H}(e)|$ denote the maximum number of edges on a (possibly broken) embedding path.*

Finally, we summarize the main technical result of this section. To build intuition, we recommend the reader ignore vertex splits when reading the section for the first time.

Theorem 7.3. *Given an integer $1 \leq \Delta \leq n$ and an n-vertex, unweighted, undirected graph $G = (V, E)$ that is initially empty and undergoes up to $n\Delta$ edge insertions and n vertex splits/edge deletions such that the master node degree $\deg^{\text{master}}(v)$ is bounded by Δ for every vertex v throughout.*

For some fixed $\gamma_{\text{incSpanner}} = e^{O(\log^{20/21} m \log\log m)}$, there is an algorithm IncrementalSpanner(G) that maintains a sparse subgraph $H \subseteq G$ such that whenever an edge e is added to H, it remains in H until the end of the algorithm, and an embedding Π that maps each edge $e = (u, v)$ to a uv-path $\Pi(e)$ in the graph G at the time of the edge insertion. Afterwards, the embedding path remains fixed, except that the endpoints of edges that were split are updated to reflect the new endpoints of the edge.

The algorithm ensures that at any time, Π has vertex congestion at most $\gamma_{\text{incSpanner}}\Delta$ and embedding length at most $\gamma_{\text{incSpanner}}$. The algorithm takes total time $\widetilde{O}(n\Delta\gamma_{\text{incSpanner}})$ to process the entire sequence of edge insertions.

Additional Preliminaries: Dynamic APSP. In this section, we use the following result from [KMP23] in our algorithm.

Theorem 7.4. *Given an m-edge input graph $G = (V, E, l)$ with lengths in $[1, L]$, there is a data structure DynamicAPSP that can process a polynomial*[6] *number of edge insertions and deletions to G and at any point in time answers queries where inputted $u, v \in V$, it returns a distance estimate $\widehat{\text{dist}}(u, v)$ such that $\text{dist}_G(u, v) \leq \widehat{\text{dist}}(u, v) \leq \gamma_{approxAPSP} \cdot \text{dist}_G(u, v)$, for some $\gamma_{approxAPSP} = e^{O(\log^{6/7} m \log\log m)}$. It can further report a uv-path P with $l(P) \leq \gamma_{approxAPSP} \cdot \text{dist}_G(u, v)$.*

For some $\gamma_{timeAPSP} = e^{O(\log^{20/21} m \log\log m)}$, the data structure can be initialized in time $m \cdot \gamma_{timeAPSP} \cdot \log L$, processes each edge update in worst-case time $\gamma_{timeAPSP} \cdot \log L$ and each query in worst-case time $O(\log m \log L)$, and reports a path P in worst-case time $O(|P| \log m \log L)$.

Algorithm. We give the pseudocode for our algorithm in Algorithm 6. Upon initialization, our algorithm is given the number of vertices and a bound on the maximum degree Δ a node can attain throughout the insertion sequence. We assume that the graph is initially empty. If there is a non-empty initial graph, we first insert the edges one by one as if they were edge insertions. Our construction consists of $K = 10 \log n$ layers. The goal of each layer i is to embed a large constant fraction of the edges that layer $i - 1$ was not able to embed. To do so, each layer i maintains a spanner H_i, and a sub-graph $\widehat{H}_i$ of H_i. It further maintains an APSP data structure $\mathcal{D}_i$ on the graph $\widehat{H}_i$. All these graphs are initialized to empty graphs on n vertices. We will maintain that the total number of edges seen by layer i is at most $n\Delta/2^i$.

Whenever an edge is inserted we pass it to layer 0. The only time an edge e ever gets passed to a layer is when it is inserted. It then gets passed down to layers of increasing index until a layer j is handling it. Edge e will then forever be handled by layer j and layers $0, \ldots, j$ are said to have seen it.

Whenever an edge $e = (u, v)$ is passed to layer i, the algorithm distinguishes the following three cases:

1. if $\mathcal{D}_i$ returns a uv-path P of length at most $2\gamma_{\text{approxAPSP}} \log n$ in $\widehat{H}_i$: then set the new embedding path $\Pi(e)$ of e equal to the found path P. Further, remove edges e' on P with congestion larger $2\gamma_{\text{approxAPSP}} \cdot \Delta \cdot \log 2n/2^i$ from $\widehat{H}_i$ and $\mathcal{D}_i$.
2. if the returned path P is of greater length **and** $\deg^{\text{master}}_{H_i}(u), \deg^{\text{master}}_{H_i}(v) \leq 32 \cdot 2^i$: then, add e to H_i, $\widehat{H}_i$ and $\mathcal{D}_i$.
3. otherwise: pass e to layer $i + 1$.

Whenever an edge is deleted, it gets deleted from the spanner.

Whenever a vertex is split, we simulate it by removing the smaller side and adding it to some isolated vertex for every layer i. We update H_i, $\widehat{H}_i$, and $\mathcal{D}_i$ accordingly. Notice that we maintain the embedding Π for the re-inserted edges and that we refer to them as the same edges as before. In particular, we do not invoke the procedure Insert on edges that are re-inserted.

[6] In this paper, the term polynomial always refers to a polynomial in m.

Algorithm 6: IncrementalSpanner()

Procedure *Init*(n, Δ)
  // We assume we are given an empty graph. If there is a non-empty initial graph, we simply insert its edges via Insert.
  $K \leftarrow 10 \log n$
  $H_0, H_1, \ldots, H_K \leftarrow (V, \emptyset)$
  $\widehat{H}_0, \widehat{H}_1, \ldots, \widehat{H}_K \leftarrow (V, \emptyset)$
  Let $H := \bigcup_{i=0}^{K} H_i$ throughout.
  Initialize APSP datastructure $\mathcal{D}_i$ on $\widehat{H}_i$ for all $i = 0, \ldots, K$.
  $\Pi \leftarrow \emptyset$
Procedure *Insert*$(e = (u, v))$
  for $i = 0, \ldots, K$ **do**
    if $\mathcal{D}_i.\textsc{Dist}(u, v) \leq 2\gamma_{\text{approxAPSP}} \cdot \log(2n)$ **then**
      $P \leftarrow \mathcal{D}_i.\textsc{Path}(u, v)$.
      $\Pi(e) \leftarrow P$.
      foreach $e' \in P$ *where* $\text{econg}(\Pi, e') \geq 2\gamma_{\text{approxAPSP}} \cdot \Delta \cdot \log n / 2^i$ **do**
        Delete e' from $\widehat{H}_i$ and therefore also from the data structure $\mathcal{D}_i$.
      return H, Π
    else if $\max(\deg_{H_i}^{\text{master}}(u), \deg_{H_i}^{\text{master}}(v)) \leq 32 \cdot 2^i$ **then**
      Add e to H_i and $\widehat{H}_i$ and update $\mathcal{D}_i$ accordingly.
      return H, Π
Procedure *DeleteEdge*(e)
  for $i = 0, \ldots, K$ **do**
    Delete edge e from H_i, $\widehat{H}_i$.
Procedure *Split*$(v, E_{\text{move}}, E_{\text{cross}})$
  // We may assume wlog that there are no self loops, and thus no crossing edges by delaying their insertion.
  for $i = 0, \ldots, K$ **do**
    Assume wlog that $|E_{\text{move}} \cap E_{H_i}| \leq |\deg_{H_i}(v)|/2$.
    Delete all edges in $E_{H_i} \cap (E_{\text{move}} \cup E_{\text{cross}})$ from H_i, $\widehat{H}_i$ and $\mathcal{D}_i$ and re-insert them adjacent to an isolated vertex. Preserve embeddings.
  return H, Π

Proof of Theorem 7.3. We first bound the total number of edges in H_i. We do this by bounding the number of edges in $H_i \setminus \widehat{H}_i$ by using congestion bounds. The number of edges in $\widehat{H}_i$ is $O(n)$ because its girth is at least $2 \log n$.

Claim 7.5. *Assume the total number of edges ever seen by layer i is $\Delta n/2^i$. Then $|E_{H_i}| \leq 9n$ for all $i = 0, \ldots, K$.*

Proof. We first analyse the number of edges in $H_i \setminus \widehat{H}_i$. Every edge that is in $H_i \setminus \widehat{H}_i$ has at least $2\gamma_{\text{approxAPSP}} \cdot \Delta \cdot \log n/2^i$ edges embedded into it. Since the total number of edges passed to layer i is at most $\Delta n/2^i$ and each edge has an embedding path of length at most $2\gamma_{\text{approxAPSP}} \log n$, the total number of edges in $H_i \setminus \widehat{H}_i$ is at most n.

By standard techniques for analysing spanners [ADD+93] we have that the number of edges in $\widehat{H}_i$ is at most $8n$ because it has girth $> 2\log(2n)$ and at most $2n$ vertices (since there are initially n vertices and the at most n vertex splits can increase the number of vertices by at most n). □

We then analyse the total number of edges passed down by layer i.

Claim 7.6. *Assume the total number ever seen by layer i is $\Delta n/2^i$. Then the total number of edges ever seen by layer $i+1$ is at most $\Delta n/2^{i+1}$.*

Proof. Consider the final spanner H_i computed by layer i. Let $S_i = \{v \in V : \deg_{H_i}^{\text{master}}(v) \geq 32 \cdot 2^i\}$. By Claim 7.5 the total number of edges in H_i is at most $16n$. Therefore, the number of master nodes with degree larger than $32 \cdot 2^i$ is at most $16n/(32 \cdot 2^i) = n/2^{i+1}$. For each of these master nodes, we pass down at most Δ edges. Therefore, the number of edges passed down is at most $n\Delta/2^{i+1}$. □

We perform a simple induction on the previous two claims.

Lemma 7.7. *For all $i = 0, \ldots, K$ we have*

1. *$|E_{H_i}| \leq 9n$ and*
2. *the number of edges seen by layer i is at most $\Delta n/2^i$.*

Proof. We show the claim by induction. Consider the base case $i = 0$. The second point is true because the total number of edges inserted and this passed to layer 0 is at most Δn. The first point is true because of Claim 7.5. We then assume that the claim hold for layer i. Then, we have that at most $n\Delta/2^{i+1}$ edges get passed to $i+1$ by Claim 7.6 which shows the second point. The first point then follows from Claim 7.5. □

Next, we show that the vertex congestion of Π is low.

Lemma 7.8. *For every vertex v, we have that $\sum_{e=(v,u)\in H} \text{econg}(\Pi, e) \leq 320\gamma_{approxAPSP}\Delta \log^2 n$.*

Proof. We show the bound $\sum_{e=(v,u)\in H_i} \text{econg}(\Pi, e) \leq 16\gamma_{approxAPSP}\Delta \log n$. The lemma then follows by summing them up.

The total number of edges adjacent to v at layer i is at most $32 \cdot 2^i$, since that is the maximum number of edges adjacent to all vertices with the same master vertex master(v). They can be congested up to $\gamma_{approxAPSP}\Delta \log n/2^i$. Therefore the total congestion is at most $32\gamma_{approxAPSP}\Delta \log n$. □

Next, we show that the total recourse of our incremental spanner algorithm is low.

Lemma 7.9. *The total recourse of* IncrementalSpanner() *is bounded by* $\widetilde{O}(n)$.

Proof. We show the recourse of each layer separately. Excluding vertex splits, the total number of insertions to H_i is at most $10n$ by the bound $9n$ on the final size of the spanner and since vertex splits don't reduce the number of edges in H_i and there are at most n deletions. Therefore, the only thing we have to analyse are the simulation of the vertex splits. To do so, consider the following potential $\Phi_i = \sum_{v \in V_{H_i}} \deg_{H_i}(v) \log \deg_{H_i}(v)$. Whenever a new edge is inserted to H_i the potential increases by at most $\log n$. Therefore the total potential increase is at most $9n \log n$. Whenever a vertex v of current degree D is split into the old vertex v and a new vertex v' such that v' is thereafter incident to R edges, we have that

$$(D-R)\log(D-R) + R\log R \leq (D-R)\log D + R(\log D - 1)) \leq D\log D - R.$$

Therefore, splitting off R edges decreases the potential by R, and the total number of edges split off during n vertex splits can be at most $\widetilde{O}(n)$. We obtain our result by summing up the changes over all layers. □

Lemma 7.10. *The total runtime of* IncrementalSpanner() *is* $O(n\Delta\gamma_{rtSpanner})$ *for* $\gamma_{rtSpanner} = e^{O(\log^{20/21} m \log\log m)}$.

Proof. We analyze the runtime of each layer separately. Given the upper bound Δ on the vertex degree, each vertex split can be implemented via at most Δ insertions and deletions. Since the number of insertions (and deletions to $\widehat{H}_i$) for each layer is bounded by $O(n\Delta)$ as well, the total runtime per layer is at most $O(n\Delta(\gamma_{\text{approxAPSP}} + \gamma_{\text{timeAPSP}}))$. We choose $\gamma_{\text{rtSpanner}} = 10\log n(\gamma_{\text{approxAPSP}} + \gamma_{timeAPSP}) = e^{O(\log^{20/21} m \log\log m)}$. □

We finally prove our main result by assembling the lemmas.

Proof of Theorem 7.3. The theorem follows from Lemmas 7.7 to 7.10, since $H_{10\log n}$ is an empty graph throughout because it contains at most $\Delta n/2^{10\log n} < 1$ edges by Lemma 7.7 where we chose $\gamma_{\text{incSpanner}} = \gamma_{\text{rtSpanner}} + 2\gamma_{\text{approxAPSP}} \log n = e^{O(\log^{20/21} m \log\log m)}$. Notice that the length of the embedding paths is bounded by construction. □

7.2 Fully Dynamic Spanner

Our fully dynamic spanner uses a batching scheme based on the incremental spanner presented in the previous section and the decremental spanner of [CKL+22, BCK+23a].

Preliminaries: Deterministic Vertex Congestion Spanner.

Theorem 7.11 (See Theorem 8.1 in [BCK+23a])**.** *Given undirected, unweighted graphs H and J with $V_J \subseteq V_H$ and an embedding $\Pi_{J\mapsto H}$ from J into H. Then there is a deterministic algorithm* Sparsify$(H, J, \Pi_{J\mapsto H})$ *that returns a sparsifier $\widetilde{J}$ with an embedding $\Pi_{J\mapsto\widetilde{J}}$ such that for $\gamma_c, \gamma_l = e^{O(\log^{2/3} m \log\log m)}$ we have*

1. $\text{vcong}(\Pi_{J\mapsto H} \circ \Pi_{J\mapsto\widetilde{J}}) \leq \gamma_c \cdot (\text{vcong}(\Pi_{J\mapsto H}) + \Delta_J)$ *where Δ_J is the maximum degree of graph J, and*

2. $\text{length}(\Pi_{J\mapsto H} \circ \Pi_{J\mapsto \widetilde{J}}) \leq \gamma_\ell \cdot \text{length}(\Pi_{J\mapsto H})$, *and*

3. $|E(\widetilde{J})| = \widetilde{O}(|V(J)|\gamma_\ell)$.

The algorithm runs in time $\widetilde{O}(|E(J)|\gamma_\ell^2 \text{length}(\Pi_{J\mapsto H}))$.

Overview. Our fully-dynamic spanner algorithm (See Algorithm 7) is obtained via a twist on the batching scheme used in [CKL+22, BCK+23a]. Notice that the spanner algorithms of [CKL+22, BCK+23a] are decremental in nature, and therefore only directly extend to the fully dynamic setting when the number of insertions is small enough to allow all of them to be added to the spanner. Since we have to be able to deal with a much larger number of insertions we use the incremental spanner developed in the previous section to reduce to a setting comparable to [CKL+22, BCK+23a]. Both the description of our algorithm and the analysis closely follows section 5 in [CKL+22]. Without loss of generality, we assume that the graph $G = (V, E)$ is initially empty.

Data structures. Our algorithm maintains graphs $H_{-1}, \dots H_K$ for $K = O(\log^{1/21} m)$ and the maintained spanner is given by $H = \bigcup_{i=-1}^{K} H_i$ throughout. It further maintains embeddings $\Pi_{-1}, \dots, \Pi_K$ so that Π_j maps a subset of E into the graph $H_{\leq j} := \bigcup_{i=-1}^{j} H_i$. Since the domains of the embeddings Π_j are not disjoint, we let $\Pi_{\leq j}$ denote the embedding that maps every edge e via the embedding Π_j with highest index j that has e in its pre-image. Throughout, we will ensure that $\Pi_{\leq j}$ embeds into $H_{\leq j}$ albeit possibly making use of broken paths, and $\Pi_{\leq K}$ embeds into H using only proper paths.

Further, we maintain sets $S_{-1}, \dots, S_K$ of vertices touched by deletions and vertex splits, which we formally define next.

Definition 7.12 (Definition 5.2 in [CKL+22])**.** *We say that the t-th update touches a vertex v if it is an edge deletion of an edge incident to v or it is a vertex split and v is one of the vertices resulting from the split.*

The graph H_{-1} is a spanner maintained via the data structure $\mathcal{D}_{-1} \leftarrow$ IncrementalSpanner(), and the graphs $H_0, \dots, H_K$ are periodically recomputed using Sparsify() (Theorem 7.11). The layers with higher indices are recomputed more often than the layers with lower indices, but we ensure that their progressively smaller size makes these rebuilds computationally cheaper to perform. We next describe how our algorithm reacts to updates.

Insertions. See Line 4 in Algorithm 7 for pseudocode. Given the insertion of an edge e, we simply update $\mathcal{D}_{-1}$.InsertEdge(e) and thus H_{-1}. This changes the embedding Π_{-1} maintained by $\mathcal{D}_{-1}$, but does not cause any changes to the layers $0, \dots, K$ and the embeddings $\Pi_0, \dots \Pi_K$.

Deletions and vertex splits. See Line 6 in Algorithm 7 for pseudocode. Deletions and vertex splits are passed to all layers $-1, \dots, K$. Layer -1 is updated via DeleteEdge()/Split(), whereas the graphs $H_0, \dots, H_K$ are explicityly updated. Vertex splits are implemented by copying the smaller side to a new vertex. Then, some layer gets re-built to repair the broken embedding paths. To describe the repairing, we first define edge embedding projections as in [CKL+22].

Definition 7.13 (Edge-embedding projection, see definition 5.4 in [CKL+22]). *For $i = 0, \ldots, K$ and an edge $(u, v) \in E$ such that $\Pi_{\leq i-1}(e) \cap S_{i-1} \neq \emptyset$ we let $\text{proj}_{i-1}(e) = \widehat{e} \stackrel{\text{def}}{=} (\widehat{u}, \widehat{v})$ be a new edge associated with e. Its endpoints are obtained by taking the closest vertices in S_{i-1} to u and v respectively in the graph $G_{\Pi_{\leq i-1}(e)}$, where $G_{\Pi_{\leq i-1}(e)}$ is obtained by taking the embedding path of edge e and performing all the splits and edge deletions on it that happened since it was created.*

After the t-th split/deletion, we search for the least index j such that t is divisible by $n^{(1-j)/K}$. Then, we re-set all data-structures at layers $j, \ldots K$ to be empty, and let S_{j-1} denote the set of vertices that have been touched (by a vertex split or deletion) since the last time layer $j-1$ got re-built. We then use $\text{proj}_{j-1}(\cdot)$ to project all edges with broken paths to edges on S_j where $\text{proj}_{j-1}(\cdot)$ picks the first touched vertex on both sides of the broken embedding path. We denote this projected graph as J.

Using Theorem 7.11, we then compute a low vertex congestion sparsifier

$$\widetilde{J} \leftarrow \textsc{Sparsify}(H_{<j} \cup E_{\text{affected}}, J, \Pi_{J \mapsto H_{<j} \cup E_{\text{affected}}})$$

and repair the broken paths with $\Pi_j(e) \leftarrow \Pi_{<j}(e)[v, \widehat{v}] \oplus [\Pi_{J \mapsto H_{<j} \cup E_{\text{affected}}} \circ \Pi_{J \mapsto \widetilde{J}}](\widehat{e}) \oplus \Pi_{<j}(e)$ where $\Pi_{J \mapsto H_{<j} \cup E_{\text{affected}}}$ denotes the mapping of edges in J to G obtained via $\Pi_{J \mapsto H_{<j} \cup E_{\text{affected}}}(e) \leftarrow \Pi_{<j}(e)[\widehat{u}, u] \oplus e \oplus \Pi_{<j}(e)[v, \widehat{v}]$. We then add the pre-image of all edges in $\widetilde{J}$ to H_j.

Proof of Theorem 5.13. Our proof follows section 5.1 in [CKL+22], replacing the randomized sparsification procedure with its deterministic version developed in [BCK+23a].

For the purpose of analysis, we let $X^{(t)}$ denote the variable X after the t-th deletion/split. Notice that insertions between t and $t+1$ are only handled by $\mathcal{D}_{-1}$ and do not cause any changes to the layers $0, \ldots, K$. We first establish that $\Pi_{\leq K}$ embeds every edge to a path in H throughout.

Lemma 7.14 (See Lemma 5.6 in [CKL+22]). *For $i = 0, \ldots, K$ and t divisible by $n^{1-i/K}$, $\Pi^{(t)}_{\leq i}$ embeds $G^{(t)}$ into $H^{(t)}_{\leq i}$. In particular, at any stage t, $\Pi^{(t)}_{G \mapsto H} = \Pi_{\leq K}$ embeds $G^{(t)}$ into $H^{(t)}$. The property additionally holds for the intermediate graphs between deletion/split t and $t+1$.*

Proof. The proof is by induction on t. If there have not been any deletions/splits ($t = 0$), then Π_{-1} embeds E into $G^{(0)}$ by Theorem 7.3. All other Π_i are initially empty and therefore the lemma holds for $t = 0$. Now assume the lemma holds for all $t' < t$. We let $j \leftarrow j^{(t)}$, $t_{j-1} = t^{(t)}_{j-1}$ and t' be the moment in time right before the t-th deletion/split (as annotated in Algorithm 7). We first show that edge e had a valid embedding path in $\Pi_{<j}$ at some point between timesteps t_{j-1} and t'. We consider two cases that together establish this claim.

1. $\underline{e \in E^{(t_{j-1})}}$: In this case $\Pi^{(t_{j-1})}_{<j}(e)$ contains a valid embedding to $H^{(t_{j-1})}_{<j}$ by the induction hypothesis.

2. $\underline{e \notin E^{(t_{j-1})}}$: In this case the edge was inserted between time t_{j-1} and t'. Then, Π_{-1} contained a valid embedding of e after it was inserted.

By definition of S_{j-1}, we have that if $\Pi_{<j} \cap S_{j-1} = \emptyset$ then the path $\Pi_{<j}(e)$ still exists in $H^{(t)}$, i.e. if no deletion/vertex split touches the embedding path it is still valid.

Therefore, we consider the case where $\Pi_{<j}(e) \cap S_{j-1} \neq \emptyset$, i.e. a deletion/vertex split touches the path. Then $\widehat{e} = \text{proj}_{j-1}(e)$ in Line 18 and thus $\widehat{e}$ is added to J. We claim that the edge $\widehat{e}$ maps

Algorithm 7: DynamicSpanner()

Procedure $\textit{Init}(n, \Delta)$
  $\mathcal{D}_{-1} \leftarrow$ IncrementalSpanner(); $\mathcal{D}_{-1}$.Init(n, Δ)
  $t \leftarrow 0$
Procedure $\textit{InsertEdge}(e = (u, v))$
  $\mathcal{D}_{-1}$.InsertEdge(e)
Procedure $\textit{DeleteEdge}(e')/\textit{Split}(v', E_1, E_2)$
  $t \leftarrow t + 1$
  // timestep t'
  Update H_{-1} with the update via $\mathcal{D}_{-1}$.DeleteEdge() or $\mathcal{D}_{-1}$.Split() depending on the type of the update.
  Update $H_0, \ldots, H_K$ and $S_{-1}, \ldots, S_K$ with the update.
  $j \leftarrow \min\{j' \in \mathbb{Z}_{\geq 0} | t \text{is divisible by} n^{(1-j')/K}\}$
  $t_{j-1} \leftarrow \left\lfloor t/n^{1-(j-1)/K} \right\rfloor \cdot n^{1-(j-1)/K}$
  // Recompute layers $j, \ldots K$
  for $i = j, \ldots K$ **do**
    $H_i \leftarrow \emptyset$; $\Pi_i \leftarrow \emptyset$; $S_i \leftarrow \emptyset$
  $J \leftarrow (S_{j-1}, \emptyset)$
  $E_{\text{affected}} \leftarrow \{e \in E | \Pi_{<j}(e) \cap S_{j-1} \neq \emptyset\}$
  $\Pi_{J \mapsto H_{<j} \cup E_{\text{affected}}} \leftarrow \emptyset$
  foreach $e = (u, v) \in E_{\text{affected}}$ **do**
    $\widehat{e} = (\widehat{u}, \widehat{v}) \leftarrow \text{proj}_{i-1}(e)$
    Add $\widehat{e}$ to J.
    $\Pi_{J \mapsto H_{<j} \cup E_{\text{affected}}}(\widehat{e}) \leftarrow \Pi_{<j}(e)[\widehat{u}, u] \oplus e \oplus \Pi_{<j}(e)[v, \widehat{v}]$.
  $(\widetilde{J}, \Pi_{J \mapsto \widetilde{J}}) \leftarrow$ Sparsify$(H_{<j} \cup E_{\text{affected}}, J, \Pi_{J \mapsto H_{<j} \cup E_{\text{affected}}})$
  foreach $\widehat{e} \in \widetilde{J}$ **do**
    Add e to H_j
  foreach $e = (u, v) \in E_{\text{affected}}$ **do**
    $\widehat{e} = (\widehat{u}, \widehat{v}) \leftarrow \text{proj}_{j-1}(e)$
    $\Pi_j(e) \leftarrow \Pi_{<j}(e)[v, \widehat{v}] \oplus [\Pi_{J \mapsto H_{<j} \cup E_{\text{affected}}} \circ \Pi_{J \mapsto \widetilde{J}}](\widehat{e}) \oplus \Pi_{<j}(e)[\widehat{u}, u]$
  // timestep t

to a valid path in $H^{(t)}_{\leq j}$ via $[\Pi_{J\mapsto H_{<j}\cup E_{\text{affected}}} \circ \Pi_{J\mapsto \widetilde{J}}](\widehat{e})$. Firstly, the map $\Pi_{J\mapsto \widetilde{J}}$ maps each edge $\widehat{e}$ in J to a valid embedding path in $\widetilde{J}$. Then, the map $\Pi_{J\mapsto H_{<j}\cup E_{\text{affected}}}$ maps these paths to a valid path in $H_{<j}$ since all the edges it uses are pre-images of edges in $\widetilde{J}$ and thus added in Line Line 23.

This implies that the whole path $\Pi_{<j}(e)[v,\widehat{v}] \oplus [\Pi_{J\mapsto H_{<j}\cup E_{\text{affected}}} \circ \Pi_{J\mapsto \widetilde{J}}](\widehat{e}) \oplus \Pi_{<j}(e)$ is in $H^{(t)}_{\leq j}$ by the definition of S_{j-1} and $\text{proj}_{j-1}(e)$. The claim follows since $\Pi^{(t)}_{j+1}, \dots \Pi^{(t)}_{K} = \emptyset$, and insertions between t and $(t+1)'$ (i.e. until the start of the processing of the next deletion/split $t+1$) have a valid embedding path in $\Pi^{(t+1)'}_{-1}$ by Theorem 7.3. □

Now that we established that the embedding $\Pi_{\leq K}$ is proper, we bound the vertex congestion and embedding path length.

Claim 7.15. *For $i = 0, \dots, K$ we have*

1. $\text{length}(\Pi_{\leq i}) \leq 2^{i+1}\gamma_l^{i+1}\gamma_{\text{incSpanner}}$ *and*
2. $\text{vcong}(\Pi_{\leq i}) \leq 8^{i+1}\gamma_c^{i+1}\gamma_{\text{incSpanner}}\Delta$

throughout.

Proof. We prove the items separately.

1. Between times that Π_i is recomputed the embedding path $\Pi_i(e)$ remains fixed. We therefore focus on bounding the length when a re-computation of layer i happens.

   The proof is by induction on t. We have that $\text{length}(\Pi_{-1}) \leq \gamma_{\text{incSpanner}}$ throughout by Theorem 7.3 and the description of our algorithm. Since $\Pi_0, \dots, \Pi_K = \emptyset$ for $t = 0$ the base case follows.

   We let $j \leftarrow j^{(t)}$, $t_{j-1} = t^{(t)}_{j-1}$ and t' be the moment right before the t-th deletion/split. By the induction hypothesis, we have that $\text{length}(\Pi^{t_{j-1}}_{<j}) \leq 2^j\gamma_l^j\gamma_{\text{incSpanner}}$ since $t_{j-1} < t_j$ by the minimality of j in Line 10. Therefore, we have length $\text{length}(\Pi^{t'}_{<j}) \leq 2^j\gamma_l^j\gamma_{\text{incSpanner}}$ by Theorem 7.3 and the description of our algorithm since all newly embedded edges since t_{j-1} have embedding length at most $\gamma_{\text{incSpanner}}$ by Theorem 7.3.

   The segments $\Pi_{<j}(e)[\widehat{u}, u]$ and $\Pi_{<j}(e)[v,\widehat{v}]$ contain at most $2^j\gamma_l^j\gamma_{\text{incSpanner}}$ edges by the induction hypothesis. It remains to bound the length of the segments $[\Pi_{J\mapsto H_{<j}\cup E_{\text{affected}}} \circ \Pi_{J\mapsto \widetilde{J}}](\widehat{e})$. We observe that its length is at most $\gamma_l \cdot \text{length}(\Pi_{J\mapsto H_{<j}\cup E_{\text{affected}}}) \leq 2^j\gamma_l^{j+1}\gamma_{\text{incSpanner}}$ by Theorem 7.11. The claim follows by adding up the segments.

2. We next show the second item, again by induction on the number of deletions/vertex splits t. We have that $\text{vcong}(\Pi_{\leq -1}) \leq \gamma_{\text{incSpanner}}\Delta$ throughout by Theorem 7.3. Therefore the claim follows for $t = 0$ up until right before the first edge deletion/split since then $\Pi_0, \dots, \Pi_K = \emptyset$.

   We let $j \leftarrow j^{(t)}$, $t_{j-1} = t^{(t)}_{j-1}$ and t' be the moment right before the t-th deletion/split. Then, by the induction hypothesis we have that $\text{vcong}(\Pi^{(t_{j-1})}_{<j}) \leq 8^j\gamma_c^j\gamma_{\text{incSpanner}}\Delta$. Therefore, the vertex congestion at moment t' is bounded by $\text{vcong}(\Pi^{(t')}_{<j}) \leq 8^j\gamma_c^j\gamma_{\text{incSpanner}}\Delta + \gamma_{\text{incSpanner}}\Delta \leq 2 \cdot 8^j\gamma_c^j\gamma_{\text{incSpanner}}\Delta$ by Theorem 7.3 since only the embedding Π_{-1} changed in $\Pi_{<j}$ between t_{j-1} and t'.

By the description of our algorithm, the vertex congestion $\text{vcong}(\Pi^{(t')}_{<j})$ upper bounds the degree of J. By Theorem 7.11, we have

$$\text{vcong}(\Pi_{J\mapsto H_{<j}\cup E_{\text{affected}}} \circ \Pi_{J\mapsto \widetilde{J}}) \leq \gamma_c(\text{vcong}(\Pi_{<j}) + \Delta_J) \leq 4 \cdot 8^j \gamma_c^j \gamma_{\text{incSpanner}} \Delta.$$

Adding the extra congestion caused by segments $\Pi_{<j}(e)[\widehat{u}, u]$ and $\Pi_{<j}(e)[v, \widehat{v}]$ we obtain that

$$\text{vcong}(\Pi^{(t)}) \leq 5 \cdot 8^j \gamma_c^j \gamma_{\text{incSpanner}} \Delta.$$

Since the extra congestion added in-between timesteps is at most an additive $\gamma_{\text{incSpanner}}\Delta$ the claim follows. □

We finally establish the runtime and recourse bounds.

Lemma 7.16. *At any stage H consist of $\widetilde{O}(\gamma_l \cdot n) = O(n \cdot e^{O(\log^{1/2} m \log\log m)})$ edges and the amortized number of changes to H is at most $\widetilde{O}(n^{1/K}) = e^{O(\log^{20/21} m)}$ given at least n insertions. The total runtime of the algorithm after k insertions and t deletions/splits is $\widetilde{O}(n^{1/K}\gamma^2_{incSpanner}(\gamma_c\gamma_l)^{O(K)}\Delta t + \gamma_{\text{incSpanner}}\Delta k) \leq (t+k) \cdot e^{O(\log^{20/21} m \log\log m)}$. Further, the total number of embedding changes is $(t+k) \cdot e^{O(\log^{20/21} m \log\log m)}$.*

Proof. Firstly H_{-1} contains at most $\widetilde{O}(n)$ edges throughout, and the total recourse of H_{-1} is $\widetilde{O}(n)$. The total runtime of layer -1 is at most $\widetilde{O}(n\gamma_{\text{incSpanner}}\Delta)$.

We therefore focus on the layers $0, \ldots, K$.

We first show sparsity. Whenever the layer i gets rebuilt, we have that S_{i-1} is of size at most $O(n^{1-(i-1)/K})$. Since the spanner $\widetilde{J}$ built over S_{i-1} is sparse, the number of edges is $\widetilde{O}(\gamma_l|S_{j-1}|)$ by Theorem 7.11. The bounds on the sparsity follow, and the recourse caused by rebuilds is at most $\widetilde{O}(n^{1/K})$ via a simple argument over the frequency of the rebuilds.

We additionally have to bound the recourse caused by vertex splits in-between rebuilds. Whenever a vertex is split, we copy the smaller side over. We do so by layer, and notice that every edge can be on the smaller side at most $\log n$ times before the next rebuild. Therefore, the total recourse is $\widetilde{O}(K \cdot n^{1/K}) = \widetilde{O}(n^{1/K})$.

We then bound the running time. Every time a layer i gets recomputed, the size of J is at most $O(8^i\gamma_c^i\gamma_{\text{incSpanner}}\Delta)$ by Claim 7.15 and the length of the embedding $\Pi_{J\mapsto H_{<j}\cup E_{\text{affected}}}$ is at most $2^i\gamma_l^i\gamma_{\text{incSpanner}}$ by Claim 7.15. Therefore, the total runtime spent for a rebuild of layer i is bounded by

$$\widetilde{O}(8^i\gamma_c^i\gamma_{\text{incSpanner}}|S_{i-1}|\gamma_l^2 2^i\gamma_l^i\gamma_{\text{incSpanner}})$$

and the runtime bound follows since the cost of splits in-between re-computations is proportional to the recourse. Each embedding change can be directly attributed to runtime, and thus analysed in the same way. □

Proof of Theorem 5.13. It follows from Lemma 7.14, Claim 7.15 and Lemma 7.16. □

8 Minimum Cost Flow IPM

In this section, we describe the min-ratio cycle framework of [CKL+22] for solving min-cost flow and its extension to incremental graphs by [BLS23]. Finally, this yields a proof of Theorem 3.10 and Theorem 3.11.

In the following, we consider the problem of finding a min-cost circulation. This is equivalent to min-cost flow via adding high cost edges between sources and sinks, which is a standard reduction. Further, we focus on the proof of Theorem 3.11, since Theorem 3.10 directly follows from Theorem 3.11 via a binary search over the threshold F.

8.1 Minimum Cost Flow via Min-Ratio Cycles

We recall the static min-ratio cycle approach to min-cost flow of [CKL+22] that was adapted to incremental graphs by [BLS23].

For an incremental graph $G = (V, E, \boldsymbol{u}, \boldsymbol{c})$ with polynomially bounded capacities $\boldsymbol{u}(e) \in [1, U]$ and costs $\boldsymbol{c}(e) \in [-C, C]$ we recall the potential

$$\Phi(\boldsymbol{f}) := 20m \log(\boldsymbol{c}^\top \boldsymbol{f} - F) + \sum_{e \in E} (\boldsymbol{f}(e) + \delta)^{-\alpha} + (\boldsymbol{u}(e) - \boldsymbol{f}(e))^{-\alpha} \tag{6}$$

where m is the total number of edge insertions G, $\delta := 1/(20m^2C)$, $\alpha := 1/(5000 \log(mCU))$ and $-\delta \leq \boldsymbol{f}(e) \leq \boldsymbol{u}(e)$ from [BLS23]. This is an modification of the potential introduced by [CKL+22] in the context of static min-cost flow. The additional additive factor δ allows the addition of new edges with flow 0 without increasing the potential by more than a constant.

Lemma 8.1 (See Lemma 4.7 in [BLS23])**.** *Adding an edge e with some capacity $\boldsymbol{u}(e) \leq U$ and cost $\boldsymbol{c}(e) \leq C$ to the graph and setting $\boldsymbol{f}(e) = 0$ increases the potential by $O(1)$.*

Proof. Adding an edge increases the potential by $\delta^{-\alpha} + \boldsymbol{u}(e)^{-\alpha} \leq 3$. □

Next, we define the lengths and gradients as in [BLS23].

Definition 8.2 (Lengths & gradients)**.** *Given the potential $\Phi(\boldsymbol{f})$ for a graph $G = (V, E, \boldsymbol{u}, \boldsymbol{c})$ as in Equation (6), we define the* length *vector $\boldsymbol{l} \in \mathbb{R}^{|E|}$ with*

$$\boldsymbol{l}(e) = (\boldsymbol{f}(e) + \delta)^{-1} + (\boldsymbol{u}(e) - \boldsymbol{f}(e))^{-1}$$

for every $e \in E$ and the gradient *vector $\boldsymbol{g} \in \mathbb{R}^{|E|}$ as*

$$\boldsymbol{g}(e) := 20m(\boldsymbol{c}^\top \boldsymbol{f} - F)^{-1} \boldsymbol{c}(e) + \alpha(\boldsymbol{u}(e) - \boldsymbol{f}(e))^{-1-\alpha} - \alpha(\boldsymbol{f}(e) + \delta)^{-1-\alpha}.$$

We further let $\boldsymbol{L} = \operatorname{diag}(\boldsymbol{l})$.

Next, we state a lemma that shows that a good quality min-ratio cycle always exists if there exists a flow of cost at most F. As in [CKL+22, BLS23], we will crucially instantiate our solver data structure Definition 3.7 with approximate gradients and lengths $\widetilde{\boldsymbol{l}}$ and $\widetilde{\boldsymbol{g}}$. This is necessary, since the exact gradients and lengths can change very frequently. The following lemma bounds the quality of the min-ratio cycle for a sufficiently close approximation of the gradients and lengths.

Lemma 8.3 (Lemma 4.5 in [BLS23], See Lemma 4.7 in [CKL+22])**.** *Let G be a graph such that there is a feasible circulation $\boldsymbol{f}^\star$ with $\boldsymbol{c}^\top \boldsymbol{f}^\star \leq F$. Let $\widetilde{\boldsymbol{g}} \in \mathbb{R}^{|E|}$ satisfy $\left\|\boldsymbol{L}^{-1}(\widetilde{\boldsymbol{g}} - \boldsymbol{g})\right\|_\infty \leq \epsilon$ for some $\epsilon \leq \alpha/2$ and $\widetilde{\boldsymbol{l}} \approx_2 \boldsymbol{l}$. If $\Phi(\boldsymbol{f}) \leq 200m \log mCU$ and $\log(\boldsymbol{c}^\top \boldsymbol{f} - F) \geq -10m \log mCU$, then*

$$\frac{\widetilde{\boldsymbol{g}}^\top (\boldsymbol{f}^\star - \boldsymbol{f})}{\left\|\widetilde{\boldsymbol{L}}(\boldsymbol{f}^\star - \boldsymbol{f})\right\|_1} \leq -\alpha/4.$$

The next lemma shows that a suitably scaled min-ratio cycle reduces the potential by an amount that is comparable to its quality.

Lemma 8.4 (Lemma 4.4 in [CKL+22]). *Let $\widetilde{\boldsymbol{g}} \in \mathbb{R}^{|E|}$ satisfy $\left\|\boldsymbol{L}^{-1}(\widetilde{\boldsymbol{g}} - \boldsymbol{g})\right\|_\infty \leq \kappa/8$ for some $\kappa \in (0,1)$, and $\widetilde{\boldsymbol{l}} \in \mathbb{R}^{|E|}_{\geq 0}$ satisfying $\widetilde{\boldsymbol{l}} \approx_2 \boldsymbol{l}$. Let $\boldsymbol{\Delta}$ satisfy $\boldsymbol{B}^\top \boldsymbol{\Delta} = 0$ and $\widetilde{\boldsymbol{g}}^\top \boldsymbol{\Delta} / \|\boldsymbol{L}\boldsymbol{\Delta}\|_1 \leq -\kappa$. Let η be such that $\eta \widetilde{\boldsymbol{g}}^\top \boldsymbol{\Delta} = -\kappa^2/50$. Then, $\boldsymbol{f} + \eta\boldsymbol{\Delta}$ is feasible and*

$$\Phi(\boldsymbol{f} + \eta\boldsymbol{\Delta}) \leq \Phi(\boldsymbol{f}) - \kappa^2/500$$

To provide a bound on the number of iterations, we state an upper bound the initial potential.

Lemma 8.5 (See Section 4 of [BLS23]). $\Phi(\mathbf{0}) \leq 100m \cdot \log(mCU)$

Proof. We have

$$\Phi(\mathbf{0}) = 20m \log(-F) + \sum_{e \in E} (\boldsymbol{u}(e)^{-\alpha} + \delta^{-\alpha}) \leq 100m \cdot \log(mCU)$$

since $|F| \leq mCU$ as otherwise the flow would never be feasible. □

The next lemma then shows that reducing the potential to $-\widetilde{O}(m)$ suffices to obtain a flow sufficiently close to F.

Lemma 8.6 (Lemma 4.3 in [BLS23]). *If $\Phi_G(\boldsymbol{f}) \leq -200m \log(mCU)$, then $\boldsymbol{c}^\top \boldsymbol{f} \leq F + (mCU)^{-10}$.*

Therefore, given an update $\boldsymbol{\Delta}$ as in Lemma 8.4 we need at most $\widetilde{O}(m/\kappa^2)$ updates.

8.2 Algorithm

Next, we show that the min-ratio cycle solver developed in Section 3 and summarized in Definition 3.7 can be used to extract min-ratio cycles as in Lemma 8.4 efficiently to prove Theorem 3.11.

Our algorithm uses the framework of [BLS23], but replaces the routine for detecting and augmenting along min-ratio cycles. See Algorithm 8 for pseudocode. It is based on the data structure $\mathcal{D} \leftarrow \textsc{Solver}()$ from Definition 3.7. We let $\boldsymbol{f}$ denote the flow maintained by $\mathcal{D}$, while the approximation that our IPM algorithm maintains is denoted $\overline{\boldsymbol{f}}$. We further denote the approximate lengths and gradients passed to Solver as $\widetilde{\boldsymbol{l}}$ and $\widetilde{\boldsymbol{g}}$.

Initialization. Given an incremental graph G with edge capacities $\boldsymbol{u}$ in $[1, U]$ and costs $\boldsymbol{c}$ in $[-C, C]$ and a target cost F, it first initializes the flow $\overline{\boldsymbol{f}} \leftarrow 0$ and $r \leftarrow F$. Then, it computes the lengths and gradients of all edges in G via

$$\widetilde{\boldsymbol{l}}(e) \leftarrow (\overline{\boldsymbol{f}}(e) + \delta)^{-1} + (\boldsymbol{u}(e) - \overline{\boldsymbol{f}}(e))^{-1}$$

and

$$\widetilde{\boldsymbol{g}}(e) \leftarrow r/(\boldsymbol{c}^\top \overline{\boldsymbol{f}} - F)(20m\boldsymbol{c}(e)/r + \alpha(\boldsymbol{u}(e) - \overline{\boldsymbol{f}}(e))^{-1-\alpha} - \alpha(\overline{\boldsymbol{f}}(e) + \delta)^{-1-\alpha})$$

with $\alpha = 1/(5000 \log mCU)$. The algorithm further uses the parameters $q = \alpha/4\gamma_{\text{approx}}$, $\Gamma = (\alpha/4\gamma_{\text{approx}})/800$, and $\epsilon = 1/40\gamma_{\text{approx}}$.

Finally, it initializes the data structure $\mathcal{D} \leftarrow \textsc{Solver}(G, \widetilde{\boldsymbol{l}}, \widetilde{\boldsymbol{g}}, \boldsymbol{c}, \overline{\boldsymbol{f}}, q, \Gamma, \epsilon)$. Notice that $\boldsymbol{f} = \overline{\boldsymbol{f}}$ at this moment.

Cost minimization and edge insertions. The data structure repeatedly calls $E', \boldsymbol{f}'(E') \leftarrow \mathcal{D}.\textsc{ApplyCycle}()$, and updates the maintained flow $\overline{\boldsymbol{f}}(E) \leftarrow \boldsymbol{f}'(E')$, as well as the lengths and gradients of the returned edges in E' using $\mathcal{D}.\textsc{UpdateEdge}()$.

Whenever the total flow cost as changed significantly, it fully re-initializes the data structure $\mathcal{D} \leftarrow \textsc{Solver}(G, \widetilde{\boldsymbol{l}}, \widetilde{\boldsymbol{g}}, \boldsymbol{c}, \overline{\boldsymbol{f}}, q, \Gamma, \epsilon)$ after reading of the full flow $\boldsymbol{f}$ from $\mathcal{D}$ and recomputing all lengths and gradients.

Whenever $\mathcal{D}.\textsc{ApplyCycle}()$ reports that it didn't find a quality q cycle, the algorithm processes the next insertion.

When the maintained flow value $\boldsymbol{c}^\top \boldsymbol{f}$ in $\mathcal{D}$ is less than $F + (mCU)^{-10}$ the algorithm terminates and returns the flow $\mathcal{D}.\textsc{ReportFlow}()$ after rounding it to an exact solution (See e.g. Section 4 of [KP15] and Lemma 4.1 in [CKL+22]).

See Algorithm 8 for detailed pseudocode of our algorithm.

Stability of gradients and lengths. Notice that the data structure $\mathcal{D}$ in $\textsc{MinCostFlow}(G, F)$ does not maintain the exact gradients $\boldsymbol{g}$ and $\boldsymbol{l}$, but maintains a synchronization with some slack to guarantee efficiency. We will refer to the internal lengths and gradients in $\mathcal{D}$ as $\widetilde{\boldsymbol{l}}$ and $\widetilde{\boldsymbol{g}}$ respectively. We then show that we update $\widetilde{\boldsymbol{l}}$ and $\widetilde{\boldsymbol{g}}$ sufficiently regularly to achieve the preconditions of Lemma 8.3 and Lemma 8.4.

We state three lemmas from [BLS23]. They show that the residuals, lengths and gradients are sufficiently stable.

Lemma 8.7 (Lemma 5.24 in [BLS23]). *Let $\widetilde{\boldsymbol{g}} \in \mathbb{R}^{|E|}$ satisfy $\left\|\boldsymbol{L}^{-1}(\widetilde{\boldsymbol{g}} - \boldsymbol{g})\right\|_\infty \leq \epsilon$ for some $\epsilon \in (0, 1/2]$, and let $\widetilde{\boldsymbol{l}} \in \mathbb{R}^{|E|}_{\geq 0}$ satisfy $\widetilde{\boldsymbol{l}} \approx_2 \boldsymbol{l}$. Let $\boldsymbol{\Delta}$ satisfy $\boldsymbol{B}^\top \boldsymbol{\Delta} = \boldsymbol{0}$ and $\widetilde{\boldsymbol{g}}^\top \boldsymbol{\Delta} / \left\|\widetilde{\boldsymbol{L}} \boldsymbol{\Delta}\right\|_1 \leq -\kappa$ for $\kappa \in (0, 1)$. Then*

$$\frac{|\boldsymbol{c}^\top \boldsymbol{\Delta}|}{\boldsymbol{c}^\top \boldsymbol{f} - F} \leq |\widetilde{\boldsymbol{g}}^\top \boldsymbol{\Delta}| / (\kappa m)$$

Lemma 8.8 (Lemma 5.25 in [BLS23]). *If $\left\|\boldsymbol{L}(\boldsymbol{f} - \overline{\boldsymbol{f}})\right\|_\infty \leq \epsilon$ for some $\epsilon \leq 1/100$ then the lengths $\boldsymbol{l}$ and $\overline{\boldsymbol{l}}$ for flow $\boldsymbol{f}$ and $\overline{\boldsymbol{f}}$ respectively satisfy $\boldsymbol{l}(e) \approx_{1+3\epsilon} \overline{\boldsymbol{l}}(e)$*

Lemma 8.9 (Lemma 5.26 in [BLS23]). *If $\left\|\boldsymbol{L}(\boldsymbol{f} - \overline{\boldsymbol{f}})\right\|_\infty \leq \epsilon$ and $r \approx_{1+\epsilon} \boldsymbol{c}^\top \overline{\boldsymbol{f}} - F$ and*

$$\widetilde{\boldsymbol{g}}(e) = r / (\boldsymbol{c}^\top \overline{\boldsymbol{f}} - F)(20m\boldsymbol{c}(e)/r + \alpha(\boldsymbol{u}(e) - \boldsymbol{f}(e))^{-1-\alpha} - \alpha(\boldsymbol{f}(e) + \delta)^{-1-\alpha})$$

for all $e \in E$ satisfies

$$\left\|\boldsymbol{L}^{-1}(\widetilde{\boldsymbol{g}} - \boldsymbol{g})\right\|_1 \leq 10\alpha\epsilon.$$

Proof of Theorem 3.11 We fix $\kappa = q = \alpha/4\gamma_{\text{approx}}$ and $\epsilon = 1/40\gamma_{\text{approx}}$ for this proof. We then show a claim about the approximate lengths and gradients we maintain.

Claim 8.10. *The approximate lengths $\widetilde{\boldsymbol{l}}$ and gradients $\widetilde{\boldsymbol{g}}$ as maintained by $\mathcal{D}$ satisfy the preconditions of Lemma 8.3 and Lemma 8.4, i.e., $\left\|\boldsymbol{L}^{-1}(\widetilde{\boldsymbol{g}} - \boldsymbol{g})\right\|_\infty \leq \min\{\epsilon, \kappa/8\}$ and $\widetilde{\boldsymbol{l}} \approx_2 \boldsymbol{l}$.*

Proof. Note that throughout, $\overline{\boldsymbol{f}}(e)$ is the flow value $\boldsymbol{f}(e)$ that was on edge e the last time $e \in E'$ (or after the last re-initialization of $\mathcal{D}$). By the guarantees of $\mathcal{D}.\textsc{ApplyCycle}()$ in Definition 3.7, we have $\left\|\boldsymbol{L}(\boldsymbol{f} - \overline{\boldsymbol{f}})\right\|_1 \leq \epsilon$. Thus, the lemma follows from Lemma 8.8 and Lemma 8.9. □

Algorithm 8: MinCostFlow($G = (V, E, \boldsymbol{c}, \boldsymbol{u}), F$)

Let $\alpha = 1/(5000 \log mCU)$, γ_{approx} as in Definition 3.7 and $q = \alpha/4\gamma_{\text{approx}}$, $\Gamma = (\alpha/4\gamma_{\text{approx}})/800$, and $\epsilon = 1/40\gamma_{\text{approx}}$.
$r \leftarrow -F$
$\overline{\boldsymbol{f}}, \widetilde{\boldsymbol{g}}, \widetilde{\boldsymbol{l}} \leftarrow \mathbf{0}$
for $e \in E$ **do**
  $\widetilde{\boldsymbol{l}}(e) \leftarrow (\overline{\boldsymbol{f}}(e) + \delta)^{-1} + (\boldsymbol{u}(e) - \overline{\boldsymbol{f}}(e))^{-1}$
  $\widetilde{\boldsymbol{g}}(e) \leftarrow r/(\boldsymbol{c}^\top \overline{\boldsymbol{f}} - F)(20m\boldsymbol{c}(e)/r + \alpha(\boldsymbol{u}(e) - \overline{\boldsymbol{f}}(e))^{-1-\alpha} - \alpha(\overline{\boldsymbol{f}}(e) + \delta)^{-1-\alpha})$
$\mathcal{D} \leftarrow$ Solver($G, \widetilde{\boldsymbol{l}}, \widetilde{\boldsymbol{g}}, \boldsymbol{c}, \overline{\boldsymbol{f}}, q, \Gamma, \epsilon$)
while *true* **do**
  while $E', \boldsymbol{f}'(E') \leftarrow \mathcal{D}.$*ApplyCycle*() *finds circulation* **do** `// Sufficient quality circulation found`
    Update $\overline{\boldsymbol{f}}(E') \leftarrow \boldsymbol{f}'(E')$.
    if $\mathcal{D}.$*FlowCost*() $\leq r/(1+\epsilon)$ *or* $\mathcal{D}.$*FlowCost*() $\geq r(1+\epsilon)$ **then**
      `// Full recompute of lengths and gradients.`
      $r \leftarrow \mathcal{D}$.FlowCost() $- F$; $\overline{\boldsymbol{f}} \leftarrow \mathcal{D}$.ReturnFlow()
      foreach $e \in E$ **do**
        $\widetilde{\boldsymbol{l}}(e) \leftarrow (\overline{\boldsymbol{f}}(e) + \delta)^{-1} + (\boldsymbol{u}(e) - \overline{\boldsymbol{f}}(e))^{-1}$
        $\widetilde{\boldsymbol{g}}(e) \leftarrow r/(\boldsymbol{c}^\top \overline{\boldsymbol{f}} - F)(20m\boldsymbol{c}(e)/r + \alpha(\boldsymbol{u}(e) - \overline{\boldsymbol{f}}(e))^{-1-\alpha} - \alpha(\overline{\boldsymbol{f}}(e) + \delta)^{-1-\alpha})$
      $\mathcal{D} \leftarrow$ Solver($G, \widetilde{\boldsymbol{l}}, \widetilde{\boldsymbol{g}}, \boldsymbol{c}, \overline{\boldsymbol{f}}, q, \Gamma, \epsilon$)
    else
      foreach $e \in E'$ **do**
        $\widetilde{\boldsymbol{l}}(e) \leftarrow (\overline{\boldsymbol{f}}(e) + \delta)^{-1} + (\boldsymbol{u}(e) - \overline{\boldsymbol{f}}(e))^{-1}$
        $\widetilde{\boldsymbol{g}}(e) \leftarrow r/(\boldsymbol{c}^\top \overline{\boldsymbol{f}} - F)(20m\boldsymbol{c}(e)/r + \alpha(\boldsymbol{u}(e) - \overline{\boldsymbol{f}}(e))^{-1-\alpha} - \alpha(\overline{\boldsymbol{f}}(e) + \delta)^{-1-\alpha})$
        $\mathcal{D}$.UpdateEdge($e, \widetilde{\boldsymbol{l}}(e), \widetilde{\boldsymbol{g}}(e)$).
    if $\mathcal{D}.$*FlowCost*() $\leq F + (mCU)^{-10}$ **then**
      return $\mathcal{D}$.ReturnFlow() ; `// Round flow to exact solution`
  Process the next insertion of edge e with capacity $\boldsymbol{u}(e)$ and cost $\boldsymbol{c}(e)$ to G.
  $\overline{\boldsymbol{f}}(e) \leftarrow 0$
  $\widetilde{\boldsymbol{l}}(e) \leftarrow (\overline{\boldsymbol{f}}(e) + \delta)^{-1} + (\boldsymbol{u}(e) - \overline{\boldsymbol{f}}(e))^{-1}$
  $\widetilde{\boldsymbol{g}}(e) \leftarrow r/(\boldsymbol{c}^\top \overline{\boldsymbol{f}} - F)(20m\boldsymbol{c}(e)/r + \alpha(\boldsymbol{u}(e) - \overline{\boldsymbol{f}}(e))^{-1-\alpha} - \alpha(\overline{\boldsymbol{f}}(e) + \delta)^{-1-\alpha})$
  $\mathcal{D}$.InsertEdge($e, \widetilde{\boldsymbol{l}}(e), \widetilde{\boldsymbol{g}}(e)$)

Claim 8.11 (Calls to a solver)**.** *Our algorithm makes the following number of calls to the solver data structure (Definition 3.7).*

1. *The number of calls to* ApplyCycle()*/*UpdateEdge()*/*InsertEdge()*/*ReturnCost() *is bounded by* $\widetilde{O}(m\gamma_{\text{approx}}^{O(1)})$ *in total.*

2. *The number of calls to* ReturnFlow() *is bounded by* $\widetilde{O}(\gamma_{\text{approx}}^{O(1)})$ *and the number of solver data structures* $\mathcal{D}$ *initialized is* $\widetilde{O}(\gamma_{\text{approx}}^{O(1)})$.

Proof. We show the two items separately.

1. Since there are at most m unsuccessful calls to $\mathcal{D}$.ApplyCycle() because there are m insertions total, and each successful one reduces the potential Φ by $\Omega(\kappa^2)$ by Claim 8.10 and Lemma 8.4, the bound follows from the upper bound on the potential (Lemma 8.5), its increase due to insertions (Lemma 8.1), and the lower bound on the potential (Lemma 8.6). From these, we conclude that the algorithm terminates after $\widetilde{O}(m/\kappa^2) = \widetilde{O}(m\gamma_{\text{approx}}^{O(1)})$ calls to ApplyCycle() and thus also ReturnCost().

   After each call to ApplyCycle() some edges E' are returned. The amortized number of returned edges is at most

   $$\Gamma/(\epsilon q) = \widetilde{O}(\gamma_{\text{approx}}^{O(1)}).$$

   by Definition 3.7 and the description of our algorithm. These lead to a call to UpdateEdge() each, and thus to at most $\widetilde{O}(m\gamma_{\text{approx}}^{O(1)})$ calls to UpdateEdge() total. Finally, there are at most m insertions, leading to at most m calls InsertEdge().

2. By Claim 8.10 and Lemma 8.7 we have

   $$\frac{|\boldsymbol{c}^\top \boldsymbol{\Delta}|}{\boldsymbol{c}^\top \boldsymbol{f} - F} \leq |\widetilde{\boldsymbol{g}}^\top \boldsymbol{\Delta}|/(qm) = \Gamma/(qm) \leq 1/800m$$

   for each update to the maintained flow $\boldsymbol{f}$ in $\mathcal{D}$. From $(1 + 1/(800m))^t \geq (1 + \epsilon)$ we get that at least $t \geq m/\epsilon$ steps happened since the last adjustment. The bound on the number of full rebuilds and calls to ReturnFlow() follows from the total number of iterations as bounded in the previous item, since our algorithm only calls ReturnFlow() whenever a full rebuild happens.

□

Claim 8.12 (Correctness)**.** *The algorithm returns a feasible flow* $\boldsymbol{f}$ *of cost at most* F *the first time it exists.*

Proof. Firstly, whenever a flow of cost at most F exists, $\mathcal{D}$.ApplyCycle() finds a cycle of quality less than $-q$ by Theorem 3.8 since $\text{OPT} \cdot \gamma_{\text{approx}} \geq q$ by Lemma 8.3 and Claim 8.10. Secondly, the flow remains feasible by Lemma 8.4 and Claim 8.10. The claim follows. □

Theorem 3.11 follows from Claim 8.11 and Claim 8.12. Given a sufficiently accurate flow, we can round it to an exact solution in near linear time by toggling cycles on a dynamic tree (see e.g. Section 4 of [KP15] and Lemma 4.1 in [CKL+22]).

8.3 Strongly Connected Components

In this section we explain how to use the incremental IPM framework to maintain the strongly connected components (SCCs) in an incremental directed graph, thus showing Theorem 1.8.

The idea is to run the IPM discussed in this section, and then contract edges once their flow values are nontrivial. The correctness of this hinges on the following combinatorial lemma.

Lemma 8.13. *Let G be a directed graph, $\delta \leq \frac{1}{20m^2}$, and let $\boldsymbol{f} \in \mathbb{R}^{E(G)}$ be a circulation in G satisfying $-\delta \leq \boldsymbol{f}(e) \leq 1$ for all $e \in E(G)$. If $\boldsymbol{f}(e) \geq 1/(10m)$, then e is in a SCC.*

Proof. Let H be the condensation of G, i.e., G with all SCCs contracted. Let $\boldsymbol{f}^H$ be the flow on H which has $\boldsymbol{f}^H(e) = \boldsymbol{f}(e)$ for all uncontracted edges e. It is easy to see that $\boldsymbol{f}^H$ is a circulation. We will argue that because H is a DAG, that $\boldsymbol{f}^H(e) < 1/(10m)$ on all edges e. Indeed, perform a cycle decomposition of $\boldsymbol{f}^H$ into at most m cycles. Each cycle has flow at most δ, hence $\boldsymbol{f}^H(e) \leq \delta m < 1/(10m)$ for all e. Hence, any edge with $\boldsymbol{f}(e) \geq 1/(10m)$ is in an SCC. □

Algorithm. Recall that incremental cycle detection is reduced to thresholded mincost flow by setting $\boldsymbol{c}(e) = -1$ and $\boldsymbol{u}(e) = 1$ for every edge e that arrives, and the desired threshold $F = -1$. Now run Algorithm 8, except whenever an edge $e' \in E'$ has $\boldsymbol{f}(e') \geq 1/(10m)$ we contract the endpoints of e' to a single vertex. Now, we remove any edges contracted to a self-loop, and otherwise we keep the flow values the same.

Proof of Theorem 1.8. We first discuss the correctness of the algorithm. By Lemma 8.13 we never contract vertices that are not in a real SCC. By the guarantees of $\mathcal{D}$.ApplyCycle() we know that every uncontracted edge has $\boldsymbol{f}(e) < 1/(5m)$, because the last time $e \in E'$ it satisfied $\boldsymbol{f}(e) < 1/(10m)$. Thus, $\boldsymbol{c}^\top \boldsymbol{f} \geq -1/5$. However, if H has a directed cycle then there exists a circulation $\boldsymbol{f}^*$ with $\boldsymbol{c}^\top \boldsymbol{f}^* \leq -1$. Thus Lemma 8.3 implies that the algorithm can find a min ratio cycle to make progress.

Now we bound the number of iterations of the algorithm. It suffices to bound the potential increase from contracting edges. Note that $-\log(\boldsymbol{f}(e) + \delta) - \log(1 - \boldsymbol{f}(e)) \geq 0$, so removing e does only decreases this piece of the potential. For the $20m \log(\boldsymbol{c}^\top \boldsymbol{f} - F) = 20m \log(\boldsymbol{c}^\top \boldsymbol{f} + 1)$ part, recall that $\boldsymbol{c}^\top \boldsymbol{f} \geq -1/5$ at all times. Thus, removing an edge e with $\boldsymbol{f}(e) \leq 1/(5m)$ can only increase that piece of the potential by

$$20m \log\left(\frac{\boldsymbol{c}^\top \boldsymbol{f} + 1 + 1/(5m)}{\boldsymbol{c}^\top \boldsymbol{f} + 1}\right) \leq 20m \cdot \frac{1/(5m)}{\boldsymbol{c}^\top \boldsymbol{f} + 1} \leq 5,$$

as $\boldsymbol{c}^\top \boldsymbol{f} + 1 \geq 4/5$. So the total potential increase from edge contractions is bounded by $5m$.

Finally, we discuss how to implement contractions as operations to the min-ratio cycle data structure of Theorem 3.8. In particular, we never pass edge contractions to the data structure of Theorem 3.8. Instead, if an edge e is contracted, we just treat it as having $\boldsymbol{g}(e) = 0$ and $\boldsymbol{l}(e) = 1/m^{10}$ from then on. This simulates contracting e. □

References

[ADD+93] Ingo Althöfer, Gautam Das, David P. Dobkin, Deborah Joseph, and José Soares. On sparse spanners of weighted graphs. *Discret. Comput. Geom.*, 9:81–100, 1993. 10, 62

[AF10] Deepak Ajwani and Tobias Friedrich. Average-case analysis of incremental topological ordering. *Discret. Appl. Math.*, 158(4):240–250, 2010. 1, 4

[AFM08] Deepak Ajwani, Tobias Friedrich, and Ulrich Meyer. An $O(n^{2.75})$ algorithm for incremental topological ordering. *ACM Trans. Algorithms*, 4(4):39:1–39:14, 2008. 1, 4

[AHLT05] Stephen Alstrup, Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup. Maintaining information in fully dynamic trees with top trees. *Acm Transactions on Algorithms (talg)*, 1(2):243–264, 2005. 25

[AKPS19] Deeksha Adil, Rasmus Kyng, Richard Peng, and Sushant Sachdeva. Iterative refinement for ℓ_p-norm regression. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1405–1424. SIAM, 2019. 2

[BC18] Aaron Bernstein and Shiri Chechik. Incremental topological sort and cycle detection in $\widetilde{O}(m\sqrt{n})$ expected total time. In Artur Czumaj, editor, *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 21–34. SIAM, 2018. 1, 4

[BCK+23a] Jan van den Brand, Li Chen, Rasmus Kyng, Yang P Liu, Richard Peng, Maximilian Probst Gutenberg, Sushant Sachdeva, and Aaron Sidford. A deterministic almost-linear time algorithm for minimum-cost flow. In *2022 IEEE 64rd Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 2023. 2, 3, 6, 7, 10, 59, 63, 64, 65

[BCK+23b] Jan van den Brand, Li Chen, Rasmus Kyng, Yang P Liu, Richard Peng, Maximilian Probst Gutenberg, Sushant Sachdeva, and Aaron Sidford. Incremental approximate maximum flow on undirected graphs in subpolynomial update time. *arXiv preprint arXiv:2311.03174*, 2023. 1, 2

[BDP21] Aaron Bernstein, Aditi Dudeja, and Seth Pettie. Incremental SCC maintenance in sparse graphs. In Petra Mutzel, Rasmus Pagh, and Grzegorz Herman, editors, *29th Annual European Symposium on Algorithms, ESA 2021, September 6-8, 2021, Lisbon, Portugal (Virtual Conference)*, volume 204 of *LIPIcs*, pages 14:1–14:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. 1, 4

[Ber13] Aaron Bernstein. Maintaining shortest paths under deletions in weighted directed graphs. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 725–734, 2013. 1, 5

[BFG09] Michael A. Bender, Jeremy T. Fineman, and Seth Gilbert. A new approach to incremental topological ordering. In Claire Mathieu, editor, *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2009, New York, NY, USA, January 4-6, 2009*, pages 1108–1115. SIAM, 2009. 1, 4

[BFGT16] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Robert E. Tarjan. A new approach to incremental cycle detection and related problems. *ACM Trans. Algorithms*, 12(2):14:1–14:22, 2016. 1, 4

[BGS22] Aaron Bernstein, Maximilian Probst Gutenberg, and Thatchaphol Saranurak. Deterministic decremental sssp and approximate min-cost flow in almost-linear time. In *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1000–1008. IEEE, 2022. 5, 9

[BK20] Sayan Bhattacharya and Janardhan Kulkarni. An improved algorithm for incremental cycle detection and topological ordering in sparse graphs. In Shuchi Chawla, editor, *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, pages 2509–2521. SIAM, 2020. 1, 4

[BK23] Joakim Blikstad and Peter Kiss. Incremental $(1-\epsilon)$-approximate dynamic matching in $O(\text{poly}(1/\epsilon))$ update time. In Inge Li Gørtz, Martin Farach-Colton, Simon J. Puglisi, and Grzegorz Herman, editors, *31st Annual European Symposium on Algorithms, ESA 2023, September 4-6, 2023, Amsterdam, The Netherlands*, volume 274 of *LIPIcs*, pages 22:1–22:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. 1, 4

[BKS23] Sayan Bhattacharya, Peter Kiss, and Thatchaphol Saranurak. Dynamic algorithms for packing-covering lps via multiplicative weight updates. In Nikhil Bansal and Viswanath Nagarajan, editors, *Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms, SODA 2023, Florence, Italy, January 22-25, 2023*, pages 1–47. SIAM, 2023. 1, 4

[BLS23] Jan van den Brand, Yang P. Liu, and Aaron Sidford. Dynamic maxflow via dynamic interior point methods. In *Proceedings of the 55th Annual ACM Symposium on Theory of Computing*, STOC 2023, page 1215–1228, New York, NY, USA, 2023. Association for Computing Machinery. 1, 2, 3, 5, 21, 68, 69, 70, 71

[BSS22] Sayan Bhattacharya, Thatchaphol Saranurak, and Pattara Sukprasert. Simple dynamic spanners with near-optimal recourse against an adaptive adversary. In Shiri Chechik, Gonzalo Navarro, Eva Rotenberg, and Grzegorz Herman, editors, *30th Annual European Symposium on Algorithms, ESA 2022, September 5-9, 2022, Berlin/Potsdam, Germany*, volume 244 of *LIPIcs*, pages 17:1–17:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. 11

[CFKR13] Edith Cohen, Amos Fiat, Haim Kaplan, and Liam Roditty. A labeling approach to incremental cycle detection. *arXiv*, abs/1310.8381, 2013. 1, 4

[CGH+20] Li Chen, Gramoz Goranci, Monika Henzinger, Richard Peng, and Thatchaphol Saranurak. Fast dynamic cuts, distances and effective resistances via vertex sparsifiers. In *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1135–1146. IEEE, 2020. 16

[Chu21] Julia Chuzhoy. Decremental all-pairs shortest paths in deterministic near-linear time. In Samir Khuller and Virginia Vassilevska Williams, editors, *STOC '21: 53rd Annual*

ACM SIGACT Symposium on Theory of Computing, Virtual Event, Italy, June 21-25, 2021, pages 626–639. ACM, 2021. 9

[CKL+22] Li Chen, Rasmus Kyng, Yang P Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. Maximum flow and minimum-cost flow in almost-linear time. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 612–623. IEEE, 2022. 1, 2, 3, 5, 6, 7, 8, 9, 10, 11, 13, 15, 20, 59, 63, 64, 65, 68, 69, 70, 71, 73

[CZ21] Shiri Chechik and Tianyi Zhang. Incremental single source shortest paths in sparse digraphs. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2463–2477. SIAM, 2021. 1, 5

[CZ23] Julia Chuzhoy and Ruimin Zhang. A new deterministic algorithm for fully dynamic all-pairs shortest paths. In Barna Saha and Rocco A. Servedio, editors, *Proceedings of the 55th Annual ACM Symposium on Theory of Computing, STOC 2023, Orlando, FL, USA, June 20-23, 2023*, pages 1159–1172. ACM, 2023. 9, 11

[DGWN22] Debarati Das, Maximilian Probst Gutenberg, and Christian Wulff-Nilsen. A near-optimal offline algorithm for dynamic all-pairs shortest paths in planar digraphs. In *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 3482–3495. SIAM, 2022. 1, 5

[DHZ00] Dorit Dor, Shay Halperin, and Uri Zwick. All-pairs almost shortest paths. *SIAM Journal on Computing*, 29(5):1740–1759, 2000. 5

[FGNS23] Sebastian Forster, Gramoz Goranci, Yasamin Nazari, and Antonis Skarlatos. Bootstrapping dynamic distance oracles, 2023. 10, 11

[GH22] Gramoz Goranci and Monika Henzinger. Incremental Approximate Maximum Flow in $m^{1/2+o(1)}$ update time, November 2022. 1, 4

[Gup14] Manoj Gupta. Maintaining approximate maximum matching in an incremental bipartite graph in polylogarithmic update time. In *FSTTCS*, volume 29 of *LIPIcs*, pages 227–239. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2014. 1, 4

[GWN20] Maximilian Probst Gutenberg and Christian Wulff-Nilsen. Deterministic algorithms for decremental approximate shortest paths: Faster and simpler. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2522–2541. SIAM, 2020. 5

[HKM+12] Bernhard Haeupler, Telikepalli Kavitha, Rogers Mathew, Siddhartha Sen, and Robert Endre Tarjan. Incremental cycle detection, topological ordering, and strong component maintenance. *ACM Trans. Algorithms*, 8(1):3:1–3:33, 2012. 1, 4

[HKN16] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. Dynamic approximate all-pairs shortest paths: Breaking the o(mn) barrier and derandomization. *SIAM Journal on Computing*, 45(3):947–1006, 2016. 5

[HKNS15] Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *STOC*, pages 21–30. ACM, 2015. 4

[KB06] Irit Katriel and Hans L. Bodlaender. Online topological ordering. *ACM Trans. Algorithms*, 2(3):364–379, 2006. 1, 4

[KL19] Adam Karczmarz and Jakub Lacki. Reliable hubs for partially-dynamic all-pairs shortest paths in directed graphs. In *27th Annual European Symposium on Algorithms (ESA 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019. 5

[KLOS14] Jonathan A. Kelner, Yin Tat Lee, Lorenzo Orecchia, and Aaron Sidford. An almost-linear-time algorithm for approximate max flow in undirected graphs, and its multicommodity generalizations. In Chandra Chekuri, editor, *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 217–226. SIAM, 2014. 2

[KMP22] Rasmus Kyng, Simon Meierhans, and Maximilian Probst Gutenberg. Incremental SSSP for sparse digraphs beyond the hopset barrier. In Joseph (Seffi) Naor and Niv Buchbinder, editors, *Proceedings of the 2022 ACM-SIAM Symposium on Discrete Algorithms, SODA 2022, Virtual Conference / Alexandria, VA, USA, January 9 - 12, 2022*, pages 3452–3481. SIAM, 2022. 1, 5

[KMP23] Rasmus Kyng, Simon Meierhans, and Maximilian Probst Gutenberg. A Dynamic Shortest Paths Toolbox: Low-Congestion Vertex Sparsifiers and their Applications, November 2023. Available at https://arxiv.org/abs/2311.06402. , 5, 7, 9, 11, 49, 50, 51, 52, 60

[KP15] Donggu Kang and James Payor. Flow rounding, 2015. 71, 73

[KPSW19] Rasmus Kyng, Richard Peng, Sushant Sachdeva, and Di Wang. Flows in almost linear time via adaptive preconditioning. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, pages 902–913, 2019. 9, 16

[LC07] Hsiao-Fei Liu and Kun-Mao Chao. A tight analysis of the katriel-bodlaender algorithm for online topological ordering. *Theor. Comput. Sci.*, 389(1-2):182–189, 2007. 1, 4

[MNR96] Alberto Marchetti-Spaccamela, Umberto Nanni, and Hans Rohnert. Maintaining a topological order under edge insertions. *Inf. Process. Lett.*, 59(1):53–58, 1996. 1, 4

[PVW20] Maximilian Probst Gutenberg, Virginia Vassilevska Williams, and Nicole Wein. New algorithms and hardness for incremental single-source shortest paths in directed graphs. In Konstantin Makarychev, Yury Makarychev, Madhur Tulsiani, Gautam Kamath, and Julia Chuzhoy, editors, *Proccedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, Chicago, IL, USA, June 22-26, 2020*, pages 153–166. ACM, 2020. 1, 5

[Räc08] Harald Räcke. Optimal hierarchical decompositions for congestion minimization in networks. In *Proceedings of the fortieth annual ACM symposium on Theory of computing*, pages 255–264, 2008. 6

[RGH+22] Václav Rozhoň, Christoph Grunau, Bernhard Haeupler, Goran Zuzic, and Jason Li. Undirected $(1+\varepsilon)$-shortest paths via minor-aggregates: Near-optimal deterministic parallel & distributed algorithms, 2022. 7, 16, 21, 22, 24, 25, 26, 27, 28, 29

[She13] Jonah Sherman. Nearly maximum flows in nearly linear time. In *54th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 263–269. IEEE Computer Society, 2013. 2

[ST83] Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *J. Comput. System Sci.*, 26(3):362–391, 1983. 13

[ST03] Daniel A Spielman and Shang-Hua Teng. Solving sparse, symmetric, diagonally-dominant linear systems in time $O(m^{1.31})$. In *44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings.*, pages 416–427. IEEE, 2003. 42

[ST04] Daniel A. Spielman and Shang-Hua Teng. Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing (STOC)*, pages 81–90, 2004. 9, 16, 42

[WW10] Virginia Vassilevska Williams and Ryan Williams. Subcubic equivalences between path, matrix and triangle problems. In *2010 IEEE 51st Annual Symposium on Foundations of Computer Science*, pages 645–654. IEEE, 2010. 5