Maximum Flow and Minimum-Cost Flow in Almost-Linear Time

LI CHEN, Georgia Institute of Technology, USA
RASMUS KYNG, ETH Zurich, Switzerland
YANG P. LIU, Stanford University, USA
RICHARD PENG*, University of Waterloo, Canada
MAXIMILIAN PROBST GUTENBERG, ETH Zurich, Switzerland
SUSHANT SACHDEVA, University of Toronto, Canada

We present an algorithm that computes exact maximum flows and minimum-cost flows on directed graphs with m edges and polynomially bounded integral demands, costs, and capacities in $m^{1+o(1)}$ time. Our algorithm builds the flow through a sequence of $m^{1+o(1)}$ approximate undirected minimum-ratio cycles, each of which is computed and processed in amortized $m^{o(1)}$ time using a new dynamic graph data structure.

Our framework extends to algorithms running in $m^{1+o(1)}$ time for computing flows that minimize general edge-separable convex functions to high accuracy. This gives almost-linear time algorithms for several problems including entropy-regularized optimal transport, matrix scaling, p-norm flows, and p-norm isotonic regression on arbitrary directed acyclic graphs.

CCS Concepts: • **Theory of computation → Graph algorithms analysis**; **Network flows**.

Additional Key Words and Phrases: Maximum flow, Minimum cost flow, Data structures, Interior point methods, Convex optimization

ACM Reference Format:
Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. 2018. Maximum Flow and Minimum-Cost Flow in Almost-Linear Time. In . ACM, New York, NY, USA, 101 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

The maximum flow problem and its generalization, the minimum-cost flow problem, are classic combinatorial graph problems that find numerous applications in engineering and scientific computing. These problems have been studied extensively over the last seven decades, starting from the work of Dantzig and Ford-Fulkerson, and several important algorithmic problems can be reduced to min-cost flows (e.g. max-weight bipartite matching, min-cut, Gomory-Hu trees). The origin of numerous significant algorithmic developments such as the simplex method, graph sparsification, and link-cut trees, can be traced back to seeking faster algorithms for max-flow and min-cost flow.

Formally, we are given a directed graph $G = (V, E)$ with $|V| = n$ vertices and $|E| = m$ edges, upper/lower edge capacities $\boldsymbol{u}^+, \boldsymbol{u}^- \in \mathbb{R}^E$, edge costs $\boldsymbol{c} \in \mathbb{R}^E$, and vertex demands $\boldsymbol{d} \in \mathbb{R}^V$ with $\sum_{v \in V} \boldsymbol{d}_v = 0$. Our goal is to find a flow $\boldsymbol{f} \in \mathbb{R}^E$ of minimum cost $\boldsymbol{c}^\top \boldsymbol{f}$ that respects edge capacities $\boldsymbol{u}_e^- \le \boldsymbol{f}_e \le \boldsymbol{u}_e^+$ and satisfies vertex demands $\boldsymbol{d}$. The vertex demand constraints are succinctly captured as $\boldsymbol{B}^\top \boldsymbol{f} = \boldsymbol{d}$, where $\boldsymbol{B} \in \mathbb{R}^{E \times V}$ is the edge-vertex incidence matrix defined

*Part of this work was done while at the Georgia Institute of Technology, USA.

Manuscript submitted to ACM

as $\boldsymbol{B}_{((a,b),v)}$ is 1 if $v = a$, -1 if $v = b$, and 0 otherwise. To compare running times, we assume that all $\boldsymbol{u}_e^+, \boldsymbol{u}_e^-, \boldsymbol{c}_e$ and $\boldsymbol{d}_v$ are integral, and $\left|\boldsymbol{u}_e^+\right|, \left|\boldsymbol{u}_e^-\right| \leq U$ and $|\boldsymbol{c}_e| \leq C$.

There has been extensive work on max-flow and min-cost flow. While we defer a longer discussion of the related works to Section 3, a brief discussion will help place our work in context. Starting from the first pseudo-polynomial time algorithm by Dantzig [34] that ran in $O(mn^2U)$ time, the approach to designing faster flow algorithms was primarily combinatorial, working with various adaptations of augmenting paths, cycle cancelling, blocking flows, and capacity/cost scaling. A long line of work led to a running time of $\widetilde{O}(m \min\{m^{1/2}, n^{2/3}\} \log U)$ [47, 65, 76, 79] for max-flow, and $\widetilde{O}(mn \log U)$ [60] for min-cost flow. These bounds stood for decades.

In their breakthrough work on solving Laplacian systems and computing electrical flows, Spielman and Teng [123] introduced the idea of combining continuous optimization primitives with graph-theoretic constructions for designing flow algorithms. This is often referred to as the *Laplacian Paradigm*. Daitch and Spielman [33] demonstrated the power of this paradigm by combining Interior Point methods (IPMs) with fast Laplacian systems solvers to achieve an $\widetilde{O}(m^{1.5} \log^2 U)$ time algorithm for min-cost flow, the first progress in two decades. A key advantage of IPMs is that they reduce flow problems on directed graphs to problems on undirected graphs, which are easier to work with. The Laplacian paradigm achieved several successes, including $\widetilde{O}(m\epsilon^{-1})$ time $(1+\epsilon)$-approximate undirected max-flow and multicommodity flow [26, 82, 113, 119, 120], and an $m^{4/3+o(1)}U^{1/3}$ time algorithm for bipartite matching and unit capacity max-flow [10, 81, 96, 100, 101], and $pm^{1+o(1)}$ time unweighted p-norm minimizing flow for large p [5, 86].

Efficient graph data-structures have played a key role in the development of faster algorithms for flow problems, e.g. dynamic trees [122]. Recently, the development of special-purpose data-structures for efficient implementation of IPM-based algorithms has led to progress on min-cost flow for some cases – including an $\widetilde{O}(m \log U + n^{1.5} \log^2 U)$ time algorithm [18–20], an $\widetilde{O}(n \log U)$ time algorithm for planar graphs [37, 38], and small improvements for general graphs, resulting in an $\widetilde{O}(m^{3/2-1/58} \log^{O(1)} U)$ time algorithm for min-cost flow [11, 15, 59, 130]. Yet, despite this progress, the best running time bounds in general graphs are far from linear. We give the first almost-linear time algorithm for min-cost flow, achieving the optimal running time up to subpolynomial factors.

Theorem 1.1. *There is an algorithm that, on a graph $G = (V, E)$ with m edges, vertex demands, upper/lower edge capacities, and edge costs, all integral with capacities bounded by U and costs bounded by C, computes an exact min-cost flow in $m^{1+o(1)} \log U \log C$ time with high probability.*

Our algorithm implements a new IPM that solves min-cost flow via a sequence of slowly-changing undirected min-ratio cycle subproblems. We exploit randomized tree-embeddings to design new data-structures to efficiently maintain approximate solutions to these subproblems.

A direct reduction from max-flow to min-cost flow gives us an algorithm for max-flow with only a $\log U$ dependence on the capacity range U.[1]

Corollary 1.2. *There is an algorithm that on a graph G with m edges with integral capacities in $[1, U]$ computes a maximum flow between two vertices in time $m^{1+o(1)} \log U$ with high probability.*

By classic capacity scaling techniques [6, 53, 66], it suffices to work with graphs with $U, C = \text{poly}(m)$ to show Theorem 1.1 and Corollary 1.2. For completeness, we include our version of the reductions in Appendix B, as we could not find a readily citable version.

[1] s, t max-flow can be reduced to min cost circulation by adding a new edge $t \to s$ with lower capacity 0 and upper capacity mU. Set all demands to be 0. The cost of the $t \to s$ edge is -1. All other edges have zero cost.

1.1 Key Technical Contributions

Towards proving our results, we make several algorithmic contributions. We informally describe the key pieces here, and present a more detailed overview in Section 4.

Our first contribution is a new potential reduction IPM for min-cost flow, inspired by [78], that reduces min-cost flow to a sequence of $m^{1+o(1)}$ slowly-changing instances of undirected minimum-ratio cycle. Each instance of undirected min-ratio cycle is specified by an undirected graph where every edge e is assigned a positive length $\boldsymbol{\ell}_e$ and a signed gradient $\boldsymbol{g}_e$, and the goal is to find a circulation $\boldsymbol{c} \in \mathbb{R}^E$, i.e. $\boldsymbol{c}$ satisfies $\boldsymbol{B}^\top \boldsymbol{c} = 0$, with the smallest ratio $\boldsymbol{g}^\top \boldsymbol{c} / \|\boldsymbol{L}\boldsymbol{c}\|_1$, where $\boldsymbol{L} = \mathrm{diag}(\boldsymbol{\ell})$ is the diagonal length matrix. Note that the graph is undirected in the sense that each edge can be traversed in either direction, and has the same length in either direction, however, the contribution of the edge gradient changes sign depending on the direction that the edge is traversed in.

Below is an informal statement summarizing the IPM guarantees proven in Section 6.

Informal Theorem 1.3 (ℓ_1 IPM Algorithm)**.** *We give an IPM algorithm that reduces solving min-cost flow exactly to sequentially solving $m^{1+o(1)}$ instances of undirected min-ratio cycle, each up to an $m^{o(1)}$ approximation. Further, the resulting problem instances are "stable", i.e. they satisfy, 1) the direction from the current flow to the (unknown) optimal flow is a good enough solution for each of the instances, and, 2) the length and gradient input parameters to the instances change only for an amortized $m^{o(1)}$ edges every iteration.*

The standard IPM approach reduces min-cost flow to solving $\widetilde{O}(\sqrt{m})$ instances of electrical flow, which is an ℓ_2 minimization problem, to constant accuracy. At the cost of solving a larger number of resulting subproblems, our algorithm offers several advantages – undirected min-ratio cycle is an ℓ_1 minimization problem which is hopefully simpler (e.g. note that the optimal solution must be a simple cycle) and we can afford a large $m^{o(1)}$ approximation factor in the subproblems. Most analogous to our approach is an early interior point method by [132][2] which solved minimum cost flow using (exact) ℓ_1 min-ratio cycle subproblems. Their subproblems, however, do not satisfy the stability guarantees that are essential for our approach to quickly solving the subproblems approximately. Our IPM is robust to updates with much worse approximation factors than those required in the the recent works on *robust interior point methods* ([30] and many later works) and establishes a different notion of stability w.r.t. gradients, lengths, and solution witnesses. This perspective may be of independent interest.

In contrast to most IPMs that work with the log barrier, our IPM uses a power barrier which aggressively penalizes constraints that are very close to being violated, more so than the usual log barrier. This ensures polylogarithmic bit-complexity throughout our algorithm.

Since a large approximation suffices, one can use a probabilistic low stretch spanning tree T [2, 8] computed with respect to the lengths $\boldsymbol{\ell}$ and use a fundamental tree cycle to find an $\widetilde{O}(1)$ approximate solution in time $\widetilde{O}(m)$ (see Section 4.2). However, the changes to gradient and lengths by the IPM due to the flow updates during the IPM iterations forces us to compute a new probabilistic low stretch spanning tree T' with respect to the new edge lengths. But computing a new tree in time $\Omega(m)$ per iteration results in much too large a runtime.

Our approach instead rebuilds only parts of the probabilistic low-stretch spanning tree after each IPM iteration to adapt to the changes in lengths. To implement this, we design a data structure which maintains a recursive sequence of instances of the min-ratio cycle problem on graphs with fewer vertices and fewer edges. These smaller instances give worse approximate solutions, but are cheaper to maintain. We use a j-tree style approach [99] where we interleave

[2]We thank an attentive reader for making us aware of this connection.

vertex reduction by partial embeddings into trees with edge reductions via spanners, and exploit the stability of the IPM. However, using a j-tree as in [99] naïvely still requires $m^{1+o(1)}$ time per instance. Our second contribution is to push this approach much further, to give a randomized data structure that can return $m^{o(1)}$ approximate solutions to *all* $m^{1+o(1)}$ undirected min-ratio cycle instances generated by the IPM in $m^{1+o(1)}$ total time. Our approach leads to a strong form of a dynamic vertex sparsifier (in the spirit of [23]). The stability of the instances generated by our IPM algorithm is essential to achieve low amortized time per instance.

Informal Theorem 1.4 (Hidden Stable-Flow Chasing. Theorem 8.2)**.** *We design a randomized data structure for approximately solving a sequence of "stable" (as defined in Informal Theorem 1.3) undirected min-ratio cycle instances. The data structure maintains a collection of $m^{o(1)}$ spanning trees and supports the following operations with high probability in amortized $m^{o(1)}$ time: 1) Return an $m^{o(1)}$-approximate min-ratio cycle (implicitly represented as the union of $m^{o(1)}$ off-tree edges and tree paths on one of the maintained trees), 2) route a circulation along such a cycle 3) insert/delete edge e, or update $\boldsymbol{g}_e$ and $\boldsymbol{\ell}_e$, and 4) identify edges that have accumulated significant flow.*

To achieve efficient edge reduction over the entire sequence of subproblems, we give an algorithm that can efficiently maintain a spanner of a given graph (a sparse subgraph that can embed the original graph using short paths) with explicit embeddings under edge deletions/insertions and vertex splits. Removing edges can *completely* destroy the min-ratio cycles in the graph. However, in that case, we can find a good approximate min-ratio cycle using the removed edges along with their explicit spanner embeddings. This spanner is our third key contribution.

Informal Theorem 1.5 (Dynamic Spanner w/ Embeddings. Theorem 7.1)**.** *We give a randomized data-structure that for an unweighted, undirected graph G undergoing edge updates (insertions/deletions/vertex splits), maintains a subgraph H with $\widetilde{O}(n)$ edges, along with an explicit path embedding of every $e \in G$ into H of length $m^{o(1)}$. The amortized number of edge changes in H is $m^{o(1)}$ for every edge update. Moreover, the set of edges that are embed into a fixed edge $e \in H$ is decremental for all edges e, except for an amortized set of $m^{o(1)}$ edges per update.*

This algorithm can be implemented efficiently.

By designing a spanner which changes very little under input graph modifications including edge insertions/deletions and vertex splits, we make it possible to dynamically combine edge and vertex sparsification very efficiently, even in a recursive construction.

Finally, note that our data-structures for hidden stable-flow chasing and spanner maintenance are utilized to efficiently implementing the ℓ_1 IPM algorithm. Thus, the subsequent undirected min-ratio cycle instances can change depending on the approximately optimal cycles returned by our algorithm. In the terminology of dynamic graph algorithms, the sequence of undirected min-ratio cycle problems we need to solve is not oblivious (to the answers returned by the algorithm). This adaptivity creates significant additional challenges for the data-structures that need addressing.

1.2 Paper Organization

The remainder of the paper is organized as follows. We first provide in Section 2 an extensive description of applications of our algorithms. We then give in an overview of previous max-flow and min-cost flow approaches in Section 3,

In Section 4 we elaborate on each major piece of our algorithm: the ℓ_1-IPM based on undirected minimum-ratio cycles, the construction of the data structure for maintaining undirected minimum-ratio cycles for "stable" update sequences, and a spanner with explicit path embeddings in dynamic graphs.

In Section 5 we give the preliminaries.

The algorithm to obtain our main result (Theorem 1.1), the min-cost flow algorithm, is given on pages 27-74 in Sections 6-11, with some omitted proofs in Appendix A. The rest of the paper addresses generalization to convex costs, connections to the broader flow literature, and applications.

In Section 6 we give an iterative method which shows that a minimum cost flow can be computed to high accuracy in $m^{1+o(1)}$ iterations, each of which augments by a $m^{o(1)}$-approximate undirected minimum-ratio cycle. In Section 7 we construct our dynamic spanner with path embeddings. The goal of Sections 8 to 10 is to show our main data structure (Theorem 8.2) for maintaining undirected minimum-ratio cycles. Section 8 sets up the framework for describing "stable" update sequences, and describes the main data structure components. Section 9 formally constructs the data structure modulo a technical issue, which we resolve by introducing and solving the *rebuilding game* in Section 10. In Section 11 we combine all the pieces we have developed to give a min-cost flow algorithm running in time $m^{1+o(1)}$.

In the last part of the paper, Section 12, we extend the IPM analysis to handle general edge-separable convex, nonlinear objectives, such as normed flows, isotonic regression, entropy-regularized optimal transport, and matrix scaling.

The appendix contains omitted proofs in Appendix A, and a proof of capacity scaling for min-cost flows in Appendix B.

2 APPLICATIONS

Our results directly imply faster running times for algorithms that invoke network flow primitives.

Extensions of Theorem 1.1. Our main result can be generalized to take vertex capacities and costs by standard transformations (for lower capacities on vertices being zero, one can simple split each vertex v into v_{in} and v_{out} such that all in-going edges to v are incident to v_{in} and all out-going edges to v_{out} after the split, and then insert and edge (v_{in}, v_{out}) with the desired capacity and cost). Further, we can generalize our algorithm to handle the *flow diffusion* problems [24, 50, 133] where $\boldsymbol{d} \in \mathbb{R}^V, \boldsymbol{d}^\top \mathbf{1} \geq \mathbf{0}$, is considered a vertex capacity vector instead of a demand and one wants to find a flow $\boldsymbol{f}$ that satisfies $\boldsymbol{B}^\top \boldsymbol{f} \leq \boldsymbol{d}$ while minimizing over a cost function on $\boldsymbol{f}$. This can be realized by adding special vertices s, t and an edge (s, v) (resp. (v, t)) for each vertex $v \in V$ where $\boldsymbol{b}_v \leq 0$ (resp. $\boldsymbol{b}_v > 0$), with lower capacity 0, upper capacity $|\boldsymbol{b}_v|$ and cost 0.

Previously, considerable effort [15, 27, 28] was directed towards obtaining approximate max-flow algorithms that can handle vertex-capacities in undirected graphs where the above mentioned transformations do not translate. Diffusion has been considered for the cost function taken to be the ℓ_2-norm [24, 73]. We recover using simple reductions a simple almost linear time algorithm that can handle a wide range of cost function.

We can also obtain an algorithm that runs in near-linear time to compute p-norm flows, i.e. flow problems where one is given a weight matrix $\boldsymbol{W}$ and solves the problem $\min_{\boldsymbol{B}^\top \boldsymbol{f} = \boldsymbol{d}} \|\boldsymbol{W}\boldsymbol{f}\|_p^p$ up to a polynomially small error. An even more general problem is considered in Theorem 12.14. Previous work, either achieved super-linear run-time [3, 4] or was only able to solve the problem when $\boldsymbol{W}$ was taken to be the identity matrix [5, 86].

Bipartite Matching & Optimal Transport. Many popular variations of matching problems are well-known to be reducible to min-cost flow in bipartite graphs, i.e. graphs $G = (V, E)$ where there is a partition V_1, V_2 of V such that each edge has exactly one endpoint in V_1 and one in V_2.

In the standard matching problem, one is given the task of maximizing the number of edges without common vertex in an undirected graph. In the *perfect* matching problem, the algorithm has to output a matching of size $|V|/2$ or conclude that such a matching does not exist. A substantial generalization of perfect matching problems is the *worker assignment* problem: given upper capacities $\boldsymbol{u}^+ \in \mathbb{R}^E_{\geq 0}$ and costs $\boldsymbol{c} \in \mathbb{R}^E$ over the edges and has $\boldsymbol{b} \in \mathbb{N}^V_{\geq 0}$, the goal

is to either compute a weight $\boldsymbol{w} \in \mathbb{N}^E$ such that each vertex $v \in V$ has edges of total weight $\boldsymbol{b}_v$ incident and where $\boldsymbol{c}^\top \boldsymbol{w}$ is minimized over all such choices, or decide that no such weight $\boldsymbol{w}$ exists. Our result implies that the the worker assignment problem can be solved in time $m^{1+o(1)} \log^2 U$ in bipartite graphs. We refer the reader to [55] for an in-depth description of the reduction to min-cost flow.

Our result can further also be used to solve the optimal transportation problem, even with entropic regularization (see [40, 70]), which is crucial for applications in machine learning. In this problem, one is given a bipartite graph $G = (V_1 \cup V_2, E)$, demand $\boldsymbol{d}$, where $\boldsymbol{d}$ is non-negative on V_1 and non-positive on V_2, costs $\boldsymbol{c}$, and the goal is to find a flow $\boldsymbol{f}$ that satisfies $\boldsymbol{B}^\top \boldsymbol{f} = \boldsymbol{d}$ and minimizes $\boldsymbol{c}^\top \boldsymbol{f} + H(\boldsymbol{f})$ where $H(\boldsymbol{f}) = \sum_{e \in E} \boldsymbol{f}_e \log(\boldsymbol{f}_e)$. We can use our result in Theorem 12.16 to obtain the first almost-linear time algorithm to obtain an optimal flow $\boldsymbol{f}$ to high accuracy (also called transportation plan). This improves even over the run-time of $\widetilde{O}(n^2)$ taken by current state-of-the-art low accuracy solvers [9, 12, 32, 40]. Without the entropic regularization the problem is reducible directly to the worker assignment problem.

The matrix scaling problem [7, 31] asks: given a matrix $\boldsymbol{A} \in \mathbb{R}^{n \times n}_{\geq 0}$ with non-negative polynomially bounded entries, to compute positive diagonal matrices $\boldsymbol{X}, \boldsymbol{Y}$ such that all row and column sums of $\boldsymbol{XAY}$ are 1. As shown in Section 12.2, the dual of the matrix scaling problem is optimal transport with entropic regularization. Hence we achieve an algorithm for solving matrix scaling to high accuracy in almost-linear time even when the entries of the matrices $\boldsymbol{X}, \boldsymbol{Y}$ may be exponentially large.

Negative Shortest-Paths and Cycle Detection. We obtain a almost linear time algorithm to compute the Single-Source Shortest Paths from a dedicated source vertex s in a directed, possibly negatively weighted graph by invoking Corollary 1.2 with costs set to edge weights, $\boldsymbol{u}^- = \boldsymbol{0}, \boldsymbol{u}^+ = n \cdot \boldsymbol{1}$ and $\boldsymbol{d}_s = n$ and $\boldsymbol{d}_v = -1$ for all $v \in V$. For a graph with weights bounded by W in absolute value, this gives an algorithm with running time $m^{1+o(1)} \log W$. Further, we can find a negative directed cycle in a graph by choosing $\boldsymbol{u}^- = \boldsymbol{d} = \boldsymbol{0}, \boldsymbol{u}^+ = \boldsymbol{1}$, letting the cost vector equal the weights and check whether the computed flow $\boldsymbol{f}$ is non-zero. If it is not then $\boldsymbol{f}$ is a negative cost circulation and using Cut-Link Trees [122] on can recover a negative cycle. For both problems, we give the first almost linear time algorithm.

Connectivity & Gomory-Hu Trees. Another family of classic combinatorial problems are connectivity problems where many reductions to maximum flow have been found during the last years. It is well-known that from a (s, t) maximum flow, i.e. the maximum amount of flow that can be sent in a unit-weighted graph from a vertex s to vertex t, one can find an (s, t) min-cut in almost linear time, that is a bipartition V_1, V_2 of the vertex set V of the graph with $s \in V_1, t \in V_2$ such that the number of edges with tail in V_1 and head in V_2 is minimized.

Our algorithm implies an algorithm that finds the *global* min-cut obtained by minimizing over (s, t) cuts for all pairs $s, t \in V$, in time $mn^{1/2+o(1)}$ time in directed graphs [21]. For undirected graphs, using a reduction from [94], we obtain the first almost linear algorithm to compute a Steiner min-cut which is the minimum (s, t)-cut for $s, t \in S$ for a fixed input set $S \subseteq V$.

Our result also implies the first $m^{1+o(1)}$ time algorithm to compute a global vertex min-cut in undirected graphs via [93], i.e. a tripartition A, B, S of V such that there is no edge from A to B where the size of S is minimized. It further gives $m^{1+o(1)}\text{poly}(k)$ time algorithm to construct a k-vertex connectivity oracle (see [114]).

Finally, we consider algorithms to compute Gomory-Hu trees that is a weighted tree T over the vertex set of G such that for any two vertices $s, t \in V$, the (s, t) min-cut in G has the same value as in T. Our result gives the first $m^{1+o(1)}$ time algorithm to compute Gomory-Hu trees in unweighted graphs (via [1, 134]), or to a $(1 + \epsilon)$-approximation in weighted graphs (via [95]) for arbitrarily small constant ϵ.

We point out that we improve for all cited problems the run-time by polynomial factors (in m).

Directed Expanders. We say a cut $(S, V \setminus S)$ in a digraph G is ϕ-out-sparse if $\frac{|E(S,V\setminus S)|}{\min\{\mathrm{vol}(S),\mathrm{vol}(V\setminus S)\}} < \phi$ where $E(S, V \setminus S)$ is the set of edges with tail in S and head in $V \setminus S$ and $\mathrm{vol}(X)$ is the sum of degrees of vertices in X in G. A graph G is called a ϕ-expander if G allows no ϕ-out-sparse cut.

Applying our max-flow algorithm to a straightforward extension of the cut-matching game [84, 97] gives a $m^{1+o(1)}$ time algorithm that given any graph G and parameter $\phi \in (0, 1/O(\log^2 m)]$, either outputs a $O(\phi \log^2 m)$-out-sparse cut or certifies that G is a ϕ-expander. The algorithm also works when a ϕ-out-sparse cut is redefined to be a cut $(S, V \setminus S)$ with $\frac{|E(S,V\setminus S)|}{\min\{|S|,|V\setminus S|\}} < \phi$. This improves over the previously best run-time of $\widetilde{O}(m/\phi)$ for sparse graphs for a wide range of values for ϕ.

As a concrete application, we obtain a $mn^{0.5+o(1)}$ total time algorithm for the problem of maintaining strongly-connected graphs in a graph undergoing edge deletions that works against an adaptive adversary (via [14]), improving on the previously best time of $mn^{2/3+o(1)}$.

Isotonic Regression. Isotonic regression is a classic shape-constrained nonparametric regression method. The problem is formulated as follows: we are given a DAG (Directed Acyclic Graph) $G = (V, E)$ and a vector $\boldsymbol{y} \in \mathbb{R}^V$. The goal is to find project $\boldsymbol{y}$ on to the space of vectors that are *isotonic* with respect to G. A vector $\boldsymbol{x} \in \mathbb{R}^V$ is said to be isotonic with respect to G if the embedding of V into $\mathbb{R}$ given by $\boldsymbol{x}$ is weakly order-preserving with respect to the partial order described by G. The projection is usually computed using a weighted ℓ_p norm. This can be captured as the following convex program, $\min_{\boldsymbol{x}} \|\boldsymbol{W}(\boldsymbol{x} - \boldsymbol{y})\|_p$ subject to the constraints $\boldsymbol{x}_i \le \boldsymbol{x}_j$ for all $(i, j) \in E$.

We give an almost linear time algorithm for computing a $1/\mathrm{poly}(n)$ additive approximate solution to Isotonic regression for all $p \in [1, \infty)$. The previous best time bounds were $\widetilde{O}(m^{1.5})$ for $p \in [1, \infty)$ [87], $O(nm \log \frac{n^2}{m})$ for $p \in (1, \infty)$ [74], and $O(nm + n^2 \log n)$ for $p = 1$ [125]. We stress that this running time is almost-linear in the number of edges in the underlying DAG, which could be significantly smaller than the number of edges in the transitive closure, which determines the running time of some algorithms [126].

3 RELATED WORKS

We give a brief overview of the many approaches toward the max-flow and min-cost flow problems. A more detailed description of many of these approaches, and more, can be found the CACM article by Goldberg and Tarjan [68]. As there is a vast literature on flow algorithms, this list is by no means complete: we plan to update this section in subsequent works, and would greatly appreciate any pointers.

3.1 Maximum Flow

The max-flow problem, and its dual, the min-cut problem were first studied by Dantzig [34], who gave an $O(mn^2U)$ time algorithm. Ford and Fulkerson introduced the notion of residual graphs and augmenting paths, and showed the convergence of the successive augmentation algorithm via the max-flow min-cut theorem [49].

Proving faster convergence of flow augmentations has received much attention since the 1970s due to weighted network flow being a special case of linear programs. Works by Edmonds-Karp [43] and Karzanov [79] gave weakly, as well as strongly polynomial time algorithms for finding maximum flows on capacitated graphs.

Partly due to the connection with linear programming, the strongly polynomial case, as well as its generalizations to min-cost flows and lossy generalized flows, subsequently received significantly more attention.

To date, there have been three main approaches for solving max-flow in the strongly-polynomial setting:

(1) Augmenting paths [17, 35, 36, 43, 69, 79].
(2) Push-relabel [62, 63, 66, 111].
(3) Pseudo-flows [22, 48, 75].

These flow algorithms in turn motivated the study of dynamic tree data structures [57], which allows for the quick identification of bottleneck edges in dynamically changing trees. Suitably applying these dynamic trees gives a max-flow algorithm in the strongly-polynomial setting with runtime $\widetilde{O}(nm)$, which is within polylog factors of the flow decomposition barrier. This barrier lower bounds the combinatorial complexity of representing the final flow as a collection of paths.

Obtaining faster algorithms hinges strongly upon handling paths using data structures and measuring progress more numerically [33, 43, 53, 65]. Such views date back to the Edmonds-Karp [43] weakly polynomial algorithm based on finding bottleneck shortest paths which takes $\widetilde{O}(m^2 \log U)$ time. Karzanov [79] and independently Even-Tarjan [47] further showed that in unit capacity graphs, maximum flow can be solved in time $O(m \min(\sqrt{m}, n^{2/3}))$ by combining a fast bottleneck finding approach with a dual-based convergence argument. A related algorithm by Hopcroft-Karp [76] showed that maximum bipartite matching can be solved in $O(m\sqrt{n})$ time.

Our algorithm in some sense can be viewed as implementing a data structure that identifies approximate bottlenecks in $n^{o(1)}$ time per update, except we use a much more complicated definition of 'bottleneck' motivated by interior point methods. Subquadratic running times using numerical methods started with the study of scaling algorithms for weighted matchings [53] and negative length shortest paths/negative cycle detection [61]. In these directions, Goldberg and Rao [65] used binary blocking flows to obtain a runtime of $O(m \min(\sqrt{m}, n^{2/3} \log U)$ for max-flow.

More systematic studies of numerical approaches to network flows took place via the Laplacian paradigm [123]. Daitch and Spielman [33] made the critical observation that when interior point methods are applied to single commodity flow problems, the linear systems that arise are graph Laplacians, which can be solved in nearly-linear time [123]. This immediately implied $\widetilde{O}(m^{1.5} \log U)$ time algorithms for min-cost flow problems with integral costs/capacities, and provided the foundations for further improvements. Christiano, Kelner, Madry, Spielman, and Teng then gave the first exponent beyond 1.5 for max-flow: an algorithm that computes $(1+\varepsilon)$-approximate max-flows in undirected graphs in time $\widetilde{O}(mn^{1/3}\varepsilon^{-8/3})$ [26]. This motivated substantial progress on numerically driven flow algorithms, which broadly fall into two categories:

- Obtaining faster approximation algorithms for max-flow and its multi-commodity generalizations in undirected graphs through first-order methods [82, 83, 113, 119, 120], leading to a runtime of $\widetilde{O}(m\epsilon^{-1})$ for $(1+\varepsilon)$-approximate max-flow.
- Reducing the iteration complexity of high accuracy methods such as interior point methods: from $m^{1/2}$ to $n^{1/2}$, or $m^{1/3+o(1)}$ for unit capacity max-flow [26, 81, 88, 88, 100, 101].

Over the past two years, further progress took place via data structured tailored to electrical flows arising in interior point methods. These led to near-optimal runtimes for max-flow and min-cost flow on dense graphs [18, 19] as well as improvements over $m^{1.5}$ in sparse capacitated settings [59, 130]. Our approach broadly falls into this category, except we use dynamic tree-like data structures as the starting point as opposed to electrical flow data structures, and modify our interior point methods towards them. Notably, we use interior point methods based on undirected min-ratio cycles instead of electrical flows. Hence, our methods use $\Omega(m)$ iterations instead of $m^{1/2}$ or $n^{1/2}$ which is common to all algorithms subsequent to [33].

3.2 Minimum-Cost Flows

Work on the minimum cost flow problem can be traced back to the Hungarian algorithm for the assignment problem [85]. This problem is a special case of minimum cost flow on bipartite graphs with unit capacity edges. When generalized to graphs with arbitrary integer capcities, the algorithm runs in $\widetilde{O}((n+F)m)$ time where F is the total units of flow sent. Algorithms with similar running time guarantees include many variants of network simplex [6], and the out-of-kilter algorithm [51].

Strongly polynomial time algorithms for min-cost flows have been extensively studied [44, 45, 58, 71, 72, 109, 110, 112, 127], with the fastest runtime also about $\widetilde{O}(nm)$. Many these algorithms are also based on augmenting minimum mean cycles, which are closely related to our undirected minimum-ratio cycles. However, the admissible cycles in these algorithms are directed, and their analysis are with obtaining strongly polynomial time as goal.

The assignment problem has been a focal point for studying scaling algorithms that obtain high accuracy solutions numerically [39, 60, 67]. This is partly due to the negative-length shortest path problem also reducing to it [53, 61]. These scaling algorithms obtain runtimes of the form of $\widetilde{O}(m^{1.5}\log U)$, but also extend to matching problems on non-bipartite graphs [39, 53]. However, to date scaling arguments tend to work on only one of capacities or costs (similar to the reductions in Appendix B), and all previous runtimes beyond the $\Theta(nm)$ flow decomposition barrier for computing minimum-cost flows have been via interior point methods [11, 18, 33, 88, 130].

3.3 Future Directions

One implication of our result is the first almost-linear time, and hence essentially optimal, algorithm for bipartite matching. However, whether a maximum matching in general (non-bipartite) graphs [42] can be constructed in almost-linear time remains a major open problem. There is a long history of work [46, 52, 54, 64, 80, 103, 115] (please see [98] for a detailed history) on the general graph matching problem which has led to the development of several algorithmic techniques and paradigms in computing [41, 77, 104, 121, 129].

For general graph matching, the best algorithm thus far is due to the celebrated work of Micali and Vazirani [102] that runs in time $O(m\sqrt{n})$ (see [131] for a proof). Analogous to the roughly $m^{1.5}$ time bound for bipartite matching and maximum-flow [65, 76, 79] that stood until recent years, the bound due to Micali and Vazirani [102] for general graph matching has stood for over four decades. Whether there is an almost-linear time algorithm for general matching, and its minimum-cost generalizations [56], remains an outstanding open problem.

A significant methodological question raised by our work concerns the reach of the ℓ_1-interior point method we introduce. Can other (graph) problems be solved using the ℓ_1-IPM? One natural problem to attack this way is general graph matching, although this immediately runs into the issue that general graph matching lacks a polynomial-size linear program.

4 OVERVIEW

In this section, we give a technical overview of the key pieces developed in this paper. Section 4.1 describes an optimization method based on interior point methods that reduces min-cost flow to a sequence of $m^{1+o(1)}$ undirected minimum-ratio cycle computations. In particular, we reduce the problem to computing approximate min-ratio cycles on a slowly changing graph. This can be naturally formulated as a data structure problem of maintaining min-ratio cycles approximately on a dynamic graph.

We build a data structure for solving this dynamic min-ratio cycle problem and solve it with $m^{o(1)}$ amortized time per cycle update for our IPM, giving an overall running time of $m^{1+o(1)}$. Section 4.2 gives an overview of our data structure for this dynamic min-ratio cycle problem, with pointers to the rest of the overview which provides a more in-depth picture of the construction. The data structure creates a recursive hierarchy of graphs with fewer and fewer vertices and edges. In Section 4.4 we describe how to reduce the number of vertices, before describing the overall recursive data structure in Section 4.5. Naïvely, the resulting data structure works only against oblivious adversaries where updates and queries to the data structure are fixed beforehand. We cannot utilize it directly because the optimization routine updates the dynamic graph based on past outputs from the data structure. Therefore, the cycles output by the data structure may not be good enough to make progress. Section 4.6 discusses the interaction between the optimization routine and the data structure when we directly apply it. It turns out one can leverage properties of the interaction and adapt the data structure for the optimization routine. Section 4.7 presents an online algorithm that manipulates the data structure so that it always outputs cycles that are good enough to make progress in the optimization routine. Finally, the overview ends with Section 4.3 which gives an outline of our dynamic spanner data structure. We use this spanner to reduce the number of edges at each level of our recursive hierarchy, one of the main algorithmic elements of our data structure.

4.1 Computing Min-Cost Flows via Undirected Min-Ratio Cycles

The goal of this section is to describe an optimization method which computes a min-cost flow on a graph $G = (V, E)$ in $m^{1+o(1)}$ computations of $m^{o(1)}$-approximate min-ratio cycles:

$$\min_{\boldsymbol{B}^\top \boldsymbol{\Delta}=0} \frac{\boldsymbol{g}^\top \boldsymbol{\Delta}}{\|\boldsymbol{L}\boldsymbol{\Delta}\|_1} \tag{1}$$

for gradient $\boldsymbol{g} \in \mathbb{R}^E$ and lengths $\boldsymbol{L} = \operatorname{diag}(\boldsymbol{\ell})$ for $\boldsymbol{\ell} \in \mathbb{R}^E_{>0}$. Note that the value of this objective is negative, as $-\boldsymbol{\Delta}$ is a circulation if $\boldsymbol{\Delta}$ is.

Towards this, we work with the linear-algebraic setup of the min-cost flow problem:

$$\boldsymbol{f}^* \in \operatorname*{arg\,min}_{\substack{\boldsymbol{B}^\top \boldsymbol{f}=\boldsymbol{d} \\ \boldsymbol{u}_e^- \le \boldsymbol{f}_e \le \boldsymbol{u}_e^+ \text{ for all } e \in E}} \boldsymbol{c}^\top \boldsymbol{f} \tag{2}$$

for demands $\boldsymbol{d} \in \mathbb{R}^E$, lower and upper capacities $\boldsymbol{u}^-, \boldsymbol{u}^+ \in \mathbb{R}^E$, and cost vector $\boldsymbol{c} \in \mathbb{R}^E$. Our goal is to compute an optimal flow $\boldsymbol{f}^*$. Let $F^* = \boldsymbol{c}^\top \boldsymbol{f}^*$ be the optimal cost.

Our algorithm is based on a potential reduction interior point method [78], where each iteration we reduce the value of the potential function

$$\Phi(\boldsymbol{f}) \stackrel{\text{def}}{=} 20m \log(\boldsymbol{c}^\top \boldsymbol{f} - F^*) + \sum_{e \in E} \left((\boldsymbol{u}_e^+ - \boldsymbol{f}_e)^{-\alpha} + (\boldsymbol{f}_e - \boldsymbol{u}_e^-)^{-\alpha} \right) \tag{3}$$

for $\alpha = 1/(1000 \log mU)$. The reader can think of the barrier $x^{-\alpha}$ as the more standard $-\log x$ for simplicity instead. We use $x^{-\alpha}$ to ensure that all lengths/gradients encountered during the algorithm can be represented using $\widetilde{O}(1)$ bits, and explain why this holds later in the section. When $\Phi(\boldsymbol{f}) \le -200m \log mU$, we can terminate because then $\boldsymbol{c}^\top \boldsymbol{f} - F^* \le (mU)^{-10}$, at which point standard techniques let us round to an exact optimal flow [33]. Thus if we can reduce the potential by $m^{-o(1)}$ per iteration, the method terminates in $m^{1+o(1)}$ iterations.

Previous analyses of IPMs used ℓ_2 subproblems, i.e. replacing the ℓ_1 norm in (1) with an ℓ_2 norm, which can be solved using a linear system. [78] shows that using ℓ_2 subproblems such a method converges in $\widetilde{O}(m)$ iterations. Later

analyses of path-following IPMs [117] showed that a sequence of $\widetilde{O}(\sqrt{m})$ ℓ_2 subproblems suffice to give a high-accuracy solution. Surprisingly, we are able to argue that a solving sequence of $\widetilde{O}(m)$ ℓ_1 minimizing subproblems of the form in (1) suffice to give a high accuracy solution to (2). In other words, changing the ℓ_2 norm to an ℓ_1 norm does not increase the number of iterations in a potential reduction IPM. The use of an ℓ_1-norm-based subproblem gives us a crucial advantage: Problems of this form must have optimal solutions in the form of cycles—and our new algorithm finds approximately optimal cycles vastly more efficiently than any known approaches for ℓ_2 subproblems.

There are several reasons we choose to use a potential reduction IPM with this specific potential. The most important reason is the flexibility of a potential reduction IPM allows our data structure for maintaining solutions to (1) to have large $m^{o(1)}$ approximation factors. This contrasts with recent works towards solving min-cost flow and linear programs using a *robust IPM* (see [30] or the tutorial [90]), which require $(1 + o(1))$–approximate solutions for the iterates.

Finally, we use the barrier $x^{-\alpha}$ as opposed to the more standard logarithmic barrier in order to guarantee that all lengths/gradients encountered during the method are bounded by $\exp(\log^{O(1)} m)$ throughout the method. This follows because if $(\boldsymbol{u}_e^+ - \boldsymbol{f}_e)^{-\alpha} \leq \widetilde{O}(m)$, then

$$\boldsymbol{u}_e^+ - \boldsymbol{f}_e \geq \widetilde{O}(m)^{-1/\alpha} \geq \exp(-O(\log^2 Um)).$$

Such a guarantee does not hold for the logarithmic barrier.[3]

To conclude, we discuss a few specifics of the method, such as how to pick the lengths and gradients, and how to prove that the method makes progress. Given a current flow $\boldsymbol{f}$ we define the gradient and lengths we use in (1) as $\boldsymbol{g}(\boldsymbol{f}) \stackrel{\text{def}}{=} \nabla\Phi(\boldsymbol{f})$ and $\boldsymbol{\ell}(\boldsymbol{f})_e \stackrel{\text{def}}{=} \left(\boldsymbol{u}_e^+ - \boldsymbol{f}_e\right)^{-1-\alpha} + \left(\boldsymbol{f}_e - \boldsymbol{u}_e^-\right)^{-1-\alpha}$. Now, let $\boldsymbol{\Delta}$ be a circulation with $\boldsymbol{g}(\boldsymbol{f})^\top \boldsymbol{\Delta}/\|\boldsymbol{L}\boldsymbol{\Delta}\|_1 \leq -\kappa$ for some $\kappa < 1/100$, scaled so that $\|\boldsymbol{L}\boldsymbol{\Delta}\|_1 = \kappa/50$. A direct Taylor expansion shows that $\Phi(\boldsymbol{f} + \boldsymbol{\Delta}) \leq \Phi(\boldsymbol{f}) - \kappa^2/500$ (Lemma 6.4).

Hence it suffices to show that such a $\boldsymbol{\Delta}$ exists with $\kappa = \widetilde{\Omega}(1)$, because then a data structure which returns an $m^{o(1)}$-approximate solution still has $\kappa = m^{-o(1)}$, which suffices. Fortunately, the *witness circulation* $\boldsymbol{\Delta}(\boldsymbol{f})^* = \boldsymbol{f}^* - \boldsymbol{f}$ satisfies $\boldsymbol{g}(\boldsymbol{f})^\top \boldsymbol{\Delta}/\|\boldsymbol{L}\boldsymbol{\Delta}\|_1 \leq -\widetilde{\Omega}(1)$ (Lemma 6.7).

We emphasize that the fact that $\boldsymbol{f}^* - \boldsymbol{f}$ is a good enough witness circulation for the flow $\boldsymbol{f}$ is essential for proving that our randomized data structures suffice, even though the updates seem adaptive. At a high level, this guarantee helps because even though we do not know the witness circulation $\boldsymbol{f}^* - \boldsymbol{f}$, we know how it changes between iterations, because we can track changes in $\boldsymbol{f}$. We are able to leverage such guarantees to make our data structures succeed for the updates coming from the IPM. To achieve this, we end up carefully designing our adversary model with enough power to capture our IPM, but with enough restrictions that our min-ratio cycle data structure to win against the adversary. We elaborate on this point in Sections 4.2 and 4.6.

4.2 High Level Overview of the Data Structure for Dynamic Min-Ratio Cycle

As discussed in the previous section, our algorithm computes a min-cost flow by solving a sequence of $m^{1+o(1)}$ min-ratio cycle problems $\min_{\boldsymbol{B}^\top\boldsymbol{\Delta}=0} \boldsymbol{g}^\top\boldsymbol{\Delta}/\|\boldsymbol{L}\boldsymbol{\Delta}\|_1$ to $m^{o(1)}$ multiplicative accuracy. Because our IPM ensures stability for lengths and gradients (see Lemmas 6.9 and 6.10), and is even robust to approximations of lengths and gradients, we can show that over the course of the algorithm we only need to update the entries of the gradients $\boldsymbol{g}$ and lengths $\boldsymbol{\ell}$ at most $m^{1+o(1)}$

[3] The reason that path-following IPMs for max-flow [33] do not encounter this issue is because one can show that primal-dual optimality actually guarantees that the lengths/resistances are polynomially bounded. We do not maintain any dual variables, so such a guarantee does not hold for our algorithm.

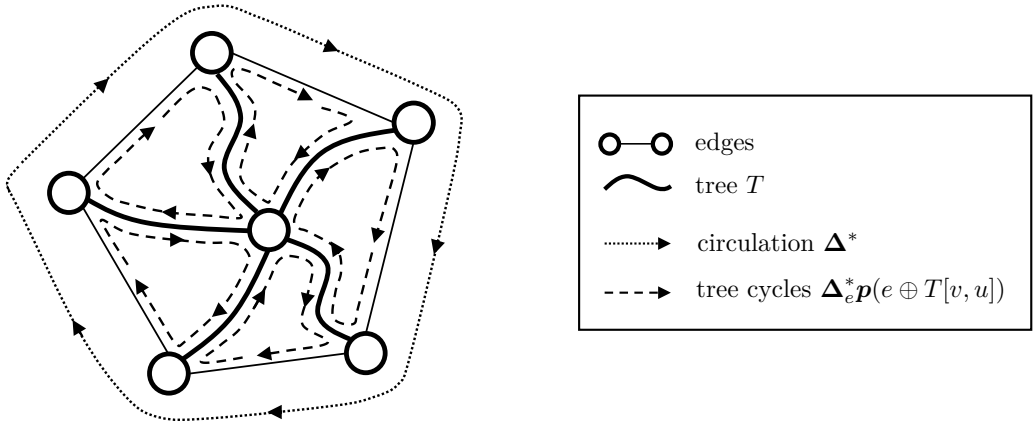

Fig. 1. Illustrating the decomposition $\Delta^* = \sum_{e:\Delta^*_e>0} \Delta^*_e \cdot \boldsymbol{p}(e \oplus T[v,u])$ of a circulation into tree cycles given by off-trees and the corresponding tree paths.

total times. Efficiency gains based on leveraging stability has appeared in the earliest works on efficiently maintaining IPM iterates [78, 128] as well as most recent progress on speeding up linear programs.

Warm-Up: A Simple, Static Algorithm. A simple approach to finding an $\widetilde{O}(1)$-approximate min-ratio cycle is the following: given our graph G, we find a probabilistic low stretch spanning tree T, i.e., a tree such that for each edge $e = (u,v) \in G$, the stretch of e, defined as $\mathrm{str}_e^{T,\boldsymbol{\ell}} \stackrel{\text{def}}{=} \frac{\sum_{f \in T[u,v]} \boldsymbol{\ell}(f)}{\boldsymbol{\ell}(e)}$ where $T[u,v]$ is the unique path from u to v along the tree T, is $\widetilde{O}(1)$ in expectation. Such a tree can be found in $\widetilde{O}(m)$ time [2, 8].

Let Δ^* be the witness circulation that minimizes (1), and assume wlog that Δ^* is a cycle that routes one unit of flow along the cycle. We assume for convenience, that edges on Δ^* are oriented along the flow direction of Δ^*, i.e. that $\Delta^* \in \mathbb{R}^E_{\geq 0}$. Then, for each edge $e = (u,v)$ on the cycle Δ^*, the fundamental tree cycle of e in T denoted $e \oplus T[v,u]$, representing the cycle formed by edge e concatenated with the path in T from its endpoint v to u. To work again with vector notation, we denote by $\boldsymbol{p}(e \oplus T[v,u]) \in \mathbb{R}^E$ the vector that sends one unit of flow along the cycle $e \oplus T[v,u]$ in the direction that aligns with the orientation of e. A classic fact from graph theory now states that $\Delta^* = \sum_{e:\Delta^*_e>0} \Delta^*_e \cdot \boldsymbol{p}(e \oplus T[v,u])$ (note that the tree-paths used by adjacent off-tree edges cancel out, see Figure 1). In particular, this implies that $\boldsymbol{g}^\top \Delta^* = \sum_{e:\Delta^*_e>0} \Delta^*_e \cdot \boldsymbol{g}^\top \boldsymbol{p}(e \oplus T[v,u])$.

This fact will allow us to argue that with probability at least $\frac{1}{2}$, one of the tree cycles is an $\widetilde{O}(1)$-approximate solution to (1). Therefore, since the stretch $\mathrm{str}_e^{T,\boldsymbol{\ell}}$ of edges $e \in E$ is small in expectation, we can, by Markov's inequality, argue that with probability at least $\frac{1}{2}$, the circulation Δ^* is not stretched by too much. Formally, we have that $\sum_{e:\Delta^*_e>0} \Delta^*_e \cdot \|\boldsymbol{L}\, \boldsymbol{p}(e \oplus T[v,u])\|_1 \leq \gamma \|\boldsymbol{L}\Delta^*\|_1$ for $\gamma = \widetilde{O}(1)$. Combining our insights, we can thus derive that

$$\frac{\boldsymbol{g}^\top \Delta^*}{\|\boldsymbol{L}\Delta^*\|_1} \geq \frac{1}{\gamma} \cdot \frac{\sum_{e:\Delta^*_e>0} \Delta^*_e \cdot \boldsymbol{g}^\top \boldsymbol{p}(e \oplus T[v,u])}{\sum_{e:\Delta^*_e>0} \Delta^*_e \cdot \|\boldsymbol{L}\, \boldsymbol{p}(e \oplus T[v,u])\|_1} \geq \frac{1}{\gamma} \min_{e:\Delta^*_e>0} \frac{\boldsymbol{g}^\top \boldsymbol{p}(e \oplus T[v,u])}{\|\boldsymbol{L}\, \boldsymbol{p}(e \oplus T[v,u])\|_1}$$

where the last inequality follows from the fact that $\min_{i \in [n]} \frac{\boldsymbol{x}_i}{\boldsymbol{y}_i} \leq \frac{\sum_{i \in [n]} \boldsymbol{x}_i}{\sum_{i \in [n]} \boldsymbol{y}_i}$ (recall also that $\boldsymbol{g}^\top \Delta^*$ is negative). But this exactly says that for the edge e minimizing the expression on the right, the tree cycle $e \oplus T[v,u]$ is a γ-approximate solution to (1), as desired.

Since the low stretch spanning tree T stretches circulation Δ^* reasonably with probability at least $\frac{1}{2}$, we could boost the probability by sampling $\widetilde{O}(1)$ trees $T_1, T_2, \ldots, T_s$ independently at random and conclude that w.h.p. one of the fundamental tree cycles gives an approximate solution to (1).

Unfortunately, after updating the flow $\boldsymbol{f}$ to $\boldsymbol{f}'$ along such a fundamental tree cycle, we cannot reuse the set of trees $T_1, T_2, \ldots, T_s$ because the next solution to (1) has to be found with respect to gradients $\boldsymbol{g}(\boldsymbol{f}')$ and lengths $\boldsymbol{\ell}(\boldsymbol{f}')$ depending on $\boldsymbol{f}'$ (instead of $\boldsymbol{g} = \boldsymbol{g}(\boldsymbol{f})$ and $\boldsymbol{\ell} = \boldsymbol{\ell}(\boldsymbol{f})$). But $\boldsymbol{g}(\boldsymbol{f}')$ and $\boldsymbol{\ell}(\boldsymbol{f}')$ depend on the randomness used in trees $T_1, T_2, \ldots, T_s$. Thus, naively, we have to recompute all trees, spending again $\Omega(m)$ time. But this leads to run-time $\Omega(m^2)$ for our overall algorithm which is far from our goal.

A Dynamic Approach. Thus we consider the data structure problem of maintaining an $m^{o(1)}$ approximate solution to (1) over a sequence of at most $m^{1+o(1)}$ changes to entries of $\boldsymbol{g}, \boldsymbol{\ell}$. To achieve an almost linear time algorithm overall, we want our data structure to have an amortized $m^{o(1)}$ update time. Motivated by the simple construction above, our data structure will ultimately maintain a set of $s = m^{o(1)}$ spanning trees $T_1, \ldots, T_s$ of the graph G. Each cycle Δ that is returned is represented by $m^{o(1)}$ off-tree edges and paths connecting them on some T_i.

To obtain an efficient algorithm to maintain these trees T_i, we turn to a recursive approach. In each level of our recursion, we first reduce the number of vertices, and then the number of edges in the graphs we recurse on. To reduce the number of vertices, we produce a *core graph* on a subset of the original vertex set, and we then compute a *spanner* of the core graph which reduces the number of edges. Both of these objects need to be maintained dynamically, and we ensure they are very stable under changes in the graphs at shallower levels in the recursion. In both cases, our notion of stability relies on some subtle properties of the interaction between the data structure and the hidden witness circulation.

We maintain a recursive hierarchy of graphs. At the top level of our hierarchy, for the input graph G, we produce $B = O(\log n)$ core graphs. To obtain each such core graph, for each $i \in [B]$, we sample a (random) forest F_i with $\widetilde{O}(m/k)$ connected components for some size reduction parameter k. The associated core graph is the graph G/F_i which denotes G after contracting the vertices in the same components of F_i. We can define a map that lifts circulations $\widehat{\Delta}$ in the core graph G/F_i, to circulations Δ in the graph G by routing flow along the contracted paths in F_i. The lengths in the core graph $\widehat{\boldsymbol{\ell}}$ (again let $\widehat{\boldsymbol{L}} = \mathrm{diag}(\widehat{\boldsymbol{\ell}})$) and are chosen to upper bound the length of circulations when mapped back into G such that $\|\widehat{\boldsymbol{L}}\widehat{\Delta}\|_1 \geq \|\boldsymbol{L}\Delta\|_1$. Crucially, we must ensure these new lengths $\widehat{\boldsymbol{\ell}}$ do not stretch the witness circulation Δ^* when mapped into G/F_i by too much, so we can recover it from G/F_i. To achieve this goal, we choose F_i to be a low stretch forest, i.e. a forest with properties similar to those of a low stretch tree. In Section 4.4, we summarize the central aspects of our core graph construction.

While each core graph G/F_i now has only $\widetilde{O}(m/k)$ vertices, it still has m edges which is too large for our recursion. To overcome this issue we build a spanner $\mathcal{S}(G, F_i)$ on G/F_i to reduce the number of edges to $\widetilde{O}(m/k)$, which guarantees that for every edge $e = (u, v)$ that we remove from G/F_i to obtain $\mathcal{S}(G, F_i)$, there is a u-to-v path in $\mathcal{S}(G, F_i)$ of length $m^{o(1)}$. Ideally, we would now recurse on each spanner $\mathcal{S}(G, F_i)$, again approximating it with a collection of smaller core graphs and spanners. However, we face an obstacle: removing edges could destroy the witness circulation, so that possibly no good circulation exists in any $\mathcal{S}(G, F_i)$. To solve this problem, we compute an explicit embedding $\Pi_{G/F_i \to \mathcal{S}(G,F_i)}$ that maps each edge $e = (u, v) \in G/F_i$ to a short u-to-v path in $\mathcal{S}(G, F_i)$. We can then show the following dichotomy: Let $\widehat{\Delta}(\boldsymbol{f})^*$ denote the witness circulation when mapped into the core graph G/F_i. Then, *either* one of the edges $e \in E_{G/F_i} \setminus E_{\mathcal{S}(G,F_i)}$ has a spanner cycle consisting of e combined with $\Pi_{G/F_i \to \mathcal{S}(G,F_i)}(e)$ which is almost as good as $\widehat{\Delta}(\boldsymbol{f})^*$, *or* re-routing $\widehat{\Delta}(\boldsymbol{f})^*$ into $\mathcal{S}(G, F_i)$ roughly preserves its quality. Figure 2 illustrates this dichotomy. Thus,

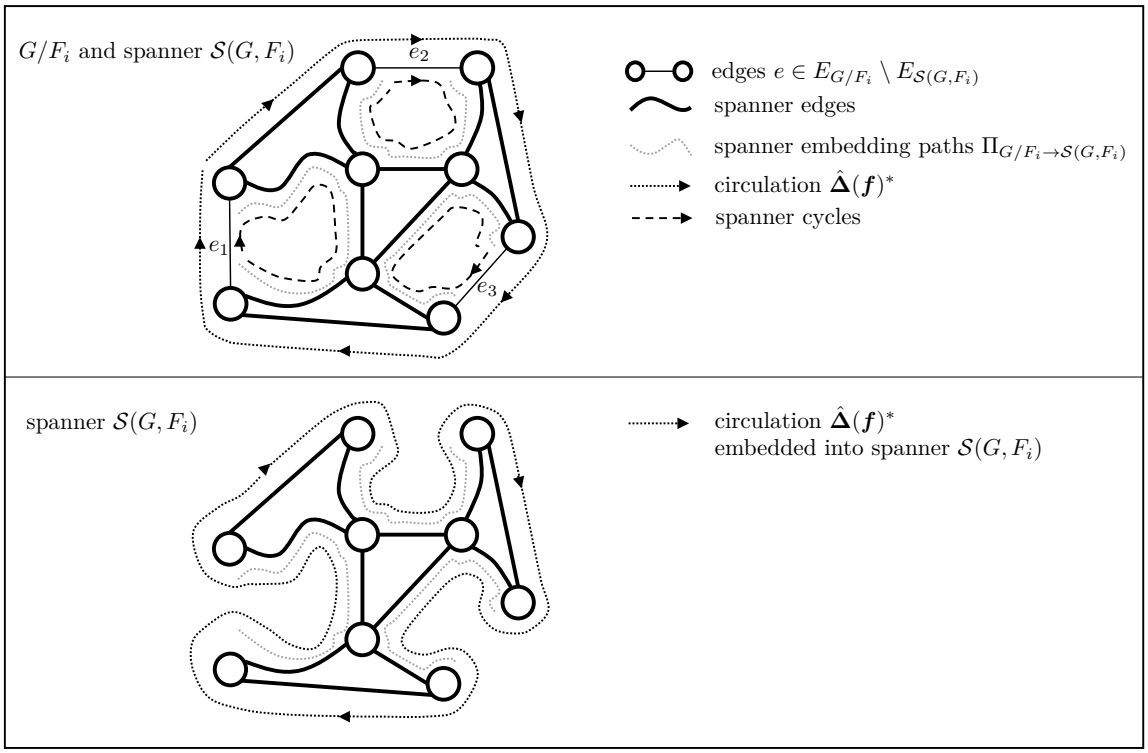

Fig. 2. Illustration of a dichotomy: either one of the edges $e \in E_{G/F_i} \setminus E_{\mathcal{S}(G,F_i)}$ has a spanner cycle consisting of e combined with $\Pi_{G/F_i \to \mathcal{S}(G,F_i)}(e)$ which is almost as good as $\widehat{\Delta}(\boldsymbol{f})^*$, *or* re-routing $\widehat{\Delta}(\boldsymbol{f})^*$ into $\mathcal{S}(G, F_i)$ roughly preserves its quality.

either we find a good cycle using the spanner, or we can recursively find a solution on $\mathcal{S}(G, F_i)$ that almost matches $\widehat{\Delta}(\boldsymbol{f})^*$ in quality. To construct our dynamic spanner with its strong stability guarantees under changes in the input graph, we use a new approach that diverges from other recent works on dynamic spanners; we give an outline of the key ideas in Section 4.3.

Our recursion uses d levels, where we choose the size reduction factor k such that $k^d \approx m$ and the bottom level graphs have $m^{o(1)}$ edges. Note that since we build B trees on G and recurse on the spanners of $G/F_1, G/F_2, \ldots, G/F_B$, our recursive hierarchy has a branching factor of $B = O(\log n)$ at each level of recursion. Thus, choosing $d \leq \sqrt{\log n}$, we get $B^d = m^{o(1)}$ leaf nodes in our recursive hierarchy. Now, consider the forests $F_{i_1}, F_{i_2}, \ldots, F_{i_d}$ on the path from the top of our recursive hierarchy to a leaf node. We can patch these forests together to form a tree associated with the leaf node. Each of these trees, we maintain as a link-cut tree data structure. Using this data structure, whenever we find a good cycle, we can route flow along it and detect edges where the flow has changed significantly. The cycles are either given by an off-tree edge or a collection of $m^{o(1)}$ off-tree edges coming from a spanner cycle. We call the entire construction a *branching tree chain*, and in Section 4.5, we elaborate on the overall composition of the data structure.

What have we achieved using this hierarchical construction compared to our simple, static algorithm? First, consider the setting of an oblivious adversary, where the gradient and length update sequences and the optimal circulation after each update is fixed in advance. In this setting, we can show that our spanner-of-core graph construction can survive through $m^{1-o(1)}/k^i$ updates at level i. Meanwhile, we can rebuild these constructions in time $m^{1+o(1)}/k^{i-1}$, leading

to an amortized cost per update of $km^{o(1)} \leq m^{o(1)}$ at each level. This gives the first dynamic data structure for our undirected min-ratio problem with $m^{o(1)}$ query time against an oblivious adversary.

However, our real problem is harder: the witness circulation in each round is $\Delta(f)^* = f^* - f$ and depends on the updates we make to f, making our problem adaptive. Instead of modelling our IPM as giving rise to a fully-dynamic problem against an adaptive adversary, the promise that the witness circulation can always be written as $f^* - f$ lets us express the IPM with an adversary that is much more restricted. Our data structure needs to ensure that the flow $f^* - f$ is stretched by $m^{o(1)}$ on average w.r.t. the lengths ℓ. At a high level, we achieve this by forcing the forests at every level to have stretch 1 on edges where f_e changes significantly and could affect the total stretch of our data structure on $f^* - f$. Section 4.6 describes the guarantees we achieve using this strategy. However, the data structure at this point is not yet guaranteed to succeed. Instead, we very carefully characterize the failure condition. In particular, to induce a failure, the adversary must create a situation where the current value of $\|L\Delta(f)^*\|_1$ is significantly less than the value when the levels of our data structure were last rebuilt. This means we can counteract from this failure by rebuilding the data structure levels. Due to the high cost of rebuilding the shallowest levels of the data structure, naïvely rebuilding the entire data structure is much too expensive, and we need a more sophisticated strategy. We describe this strategy in Section 4.7, where we design a game that expresses the conflict between our data structure and the adversary, and we show how to win this game without paying too much runtime for rebuilds.

4.3 Dynamic Embeddings into Spanners of Decremental Graphs

We start by describing the algorithm to maintain a spanner $\mathcal{S}(G, F_i)$ on the graphs G/F_i. Let us first give the requirements of the spanner:

(1) Sparsity: at all times the spanner should be sparse, i.e. consist of at most $\widetilde{O}(|V(\mathcal{S}(G, F_i))|)$ edges. This is crucial for reducing the problem size and as we ensure that F_i has only $\widetilde{O}(m/k)$ connected components, we have that $\mathcal{S}(G, F_i)$ consists of $\widetilde{O}(m/k)$ edges, reducing the problem size by a factor of almost k.
(2) Low Recourse: we further require that for each update to G/F_i, there are at most $\gamma_r = m^{o(1)}$ changes to $\mathcal{S}(G, F_i)$ on average. This is crucial as otherwise the updates to $\mathcal{S}(G, F_i)$ could trigger even more updates in the branching tree chain (see Section 4.5 for more details).
(3) Short Paths with Embedding: we maintain the spannner such that for every edge e in G, its endpoints in $\mathcal{S}(G, F_i)$ are at distance at most $\gamma_l \cdot \ell(e)$ and even maintain witness paths $\Pi_{G \to \mathcal{S}(G,F_i)}(e)$ between the endpoints consisting of γ_l edges. This is crucial as we need an explicit way to check whether $e \oplus \Pi_{G/F_i \to \mathcal{S}(G,F_i)}(e)$ is a good solution to the min-ratio cycle problem.
(4) Small Set of New Edges That We Embed Into: we ensure that after each update, we return a set D consisting of $m^{o(1)}$ edges such that each edge e in G/F_i is embedded into a path $\Pi_{G/F_i \to \mathcal{S}(G,F_i)}(e)$ consisting of the edges on the path of the old embedding path $\Pi_{G/F_i \to \mathcal{S}(G,F_i)}(e)$ of e and edges in D.
(5) Efficient Update Time: we show how to maintain $\mathcal{S}(G, F_i)$ with amortized update time $km^{o(1)}$.

We note that additionally, we need our spanner to work against adaptive adversaries since the update sequence is influenced by the output spanner. Although spanners have been studied extensively in the dynamic setting, there is currently only a single result that works against adaptive adversaries. While this spanner given in [13] appears promising, it does not ensure our desired low recourse property for vertex splits and this seems inherent to the algorithm (additionally, it also does not maintain an embedding $\Pi_{G/F_i \to \mathcal{S}(G,F_i)}$).

While we use similar elements as in [13] to obtain spanners statically, we arrive at a drastically different algorithm that can deal well with vertex splits. We focus first on obtaining an algorithm with low recourse and discuss afterwards how to implement it efficiently.

A Static Algorithm. We first consider the static version of the problem on a graph G/F_i, i.e. to give a static algorithm that computes a spanner with short path embeddings. By using a simple bucketing scheme over edge lengths, we can assume wlog that all lengths have unit-weight. We partition the graph into edge-disjoint expander graphs $H_1, H_2, \ldots, H_k$ where each H_i has roughly uniform degree, i.e. its average degree is at most a polylogarithmic factor larger than its minimum degree $\Delta_{min}(H_i)$, and each vertex v in G is in at most $\widetilde{O}(1)$ graphs H_i. Here, we define an expander to be a graph H_i that has no cut $(X, \overline{X})$ where $\overline{X} = V(H_i) \setminus X$ with $|E_{H_i}(X, \overline{X})| < \Omega\left(\frac{1}{\log^3(m)}\right) \min\{\text{vol}_{H_i}(X), \text{vol}_{H_i}(\overline{X})\}$ where $E_{H_i}(X, \overline{X})$ is the set of edges in H_i with endpoints in X and $\overline{S}$ and $\text{vol}_{H_i}(Y)$ is the sum of degrees over the vertices $y \in Y$.

Next, consider any such expander H_i. It is well-known that sampling edges in expanders with probability $p_i \sim \frac{\log^4(m)}{\Delta_{min}(H_i)}$ gives a cut-sparsifier $\mathcal{S}_i$ of H_i, i.e. a graph such that for each cut $(X, \overline{X})$, we have $|E_{H_i}(X, \overline{X})| \approx |E_{\mathcal{S}_i}(X, \overline{X})|/p_i$ (see [13, 123]). This ensures that also $\mathcal{S}_i$ is an expander. It is well-known that any two vertices in the same expander are at small distance, i.e. there is a path of length at most $\widetilde{O}(1)$ between them. We use a dynamic shortest paths data structure [29] for expander graphs on $\mathcal{S}_i$ to find such short paths between the endpoints of each edge e in G/F_i and take them to be the embedding paths (here we lose an $m^{o(1)}$ factor in the length of the paths due to the data structure).

It remains to observe that each spanner $\mathcal{S}_i$ has a nearly linear number of edges because each graph H_i has average degree close to its minimum degree, and edges are sampled independently with probability p_i. Thus, letting $\mathcal{S}(G, F_i)$ be the union of all graphs $\mathcal{S}_i$ and using that each vertex is in at most $\widetilde{O}(1)$ graphs H_i, we conclude the desired sparsity bound on $\mathcal{S}(G, F_i)$. We take $\Pi_{G/F_i \to \mathcal{S}(G,F_i)}$ to be the union of the embeddings constructed above and observe that the length of embedding paths is at most $m^{o(1)}$ as desired.

The Dynamic Algorithm. To make the above algorithm dynamic, let us assume that there is a spanner $\mathcal{S}(G, F_i)$ with corresponding embedding $\Pi_{G/F_i \to \mathcal{S}(G,F_i)}$ and after its computation, a batch of updates U is applied to G/F_i (consisting of edge insertions/deletions and vertex splits). Clearly, after forwarding the updates U to the current spanner $\mathcal{S}(G, F_i)$, by deleting edges that were deleted from G/F_i and splitting vertices, we have that for some edges $e \in G/F_i$, the updated embedding $\Pi_{G/F_i \to \mathcal{S}(G,F_i)}(e)$ might no longer be a proper path.

We therefore need to add new edges to $\mathcal{S}(G, F_i)$ and fix the embedding. We start by defining S to be the vertices that are touched by an update in U, meaning for the deletion/insertion of edge (u, v) we add u and v to S and for a vertex split of v into v and v', we add v and v' to S. Note that $|S| \leq 2|U|$ and that all $\Pi_{G/F_i \to \mathcal{S}(G,F_i)}(e)$ that are no longer proper paths intersect with S.

We now fix the embedding by constructing a new static spanner on a special graph J over the vertices of S. More precisely, for each $e = (a, b)$ in G/F_i where $\Pi_{G/F_i \to \mathcal{S}(G,F_i)}(e)$ intersects with S, we find the vertices $\widehat{a}, \widehat{b}$ in S that are closest to a and b on $\Pi_{G/F_i \to \mathcal{S}(G,F_i)}(e)$, and then insert an edge $\widehat{e} = (\widehat{a}, \widehat{b})$ into the graph J. We say that e is the pre-image of $\widehat{e}$ (and $\widehat{e}$ the image of e in J).

Finally, we run the static algorithm from the last paragraph to find a sparsifier $\widetilde{J}$ of J and let $\Pi_{J \to \widetilde{J}}$ be the corresponding embedding. Then, for each edge $\widehat{e}$ that was sampled into $\widetilde{J}$, we add its pre-image e to the current sparsifier $\mathcal{S}(G, F_i)$.

To fix the embedding, for each $\widehat{e} = (\widehat{a}, \widehat{b}) \in \widetilde{J}$, we observe that since $e = (a, b)$ was added to $\mathcal{S}(G, F_i)$, we can simply embed the edge into itself. We define for each such edge $\widehat{e}$ the path

$$P_{\widehat{e}} = \Pi_{G/F_i \to \mathcal{S}(G,F_i)}(e)[\widehat{a}, a] \oplus (a, b) \oplus \Pi_{G/F_i \to \mathcal{S}(G,F_i)}(e)[b, \widehat{b}]$$

which is a path between the endpoints of $\widehat{e}$. This path is in the current graph $\mathcal{S}(G, F_i)$ since we added (a, b) to the spanner and by definition of $\widehat{a}$, we have that $\Pi_{G/F_i \to \mathcal{S}(G,F_i)}(e)[\widehat{a}, a]$ is still a proper path, the same goes for $\widehat{b}$.

But this means we can embed each edge $f = (c, d)$ even if its image $\widehat{f} = (\widehat{c}, \widehat{d}) \notin \widetilde{J}$, since we can simply set it to the path

$$\Pi_{G/F_i \to \mathcal{S}(G,F_i)}(f)[c, \widehat{c}] \oplus \left(\bigoplus_{\widehat{e} \in \Pi_{J \to \widetilde{J}}(\widehat{f})} P_{\widehat{e}} \right) \oplus \Pi_{G/F_i \to \mathcal{S}(G,F_i)}(f)[\widehat{d}, d].$$

By the guarantees from the previous paragraph, we have that the sparsifier $\widetilde{J}$ has average degree $\widetilde{O}(1)$, and we only added the pre-images of edges in $\widetilde{J}$ to $\mathcal{S}(G, F_i)$. Since J (and $\widetilde{J}$) are taken over the vertex set S, we can conclude that we only cause $\widetilde{O}(|S|) = \widetilde{O}(|U|)$ recourse to the spanner. Further, since each new path $\Pi_{G \to \mathcal{S}(G,F_i)}(e)$ for each e now consists of $\widetilde{O}(1)$ path segments from the old embedding $\Pi_{G \to \mathcal{S}(G,F_i)}$ (plus $\widetilde{O}(1)$ edges), the maximum length of the the embedding paths has only increased by a factor of $\widetilde{O}(1)$ overall. Finally, we take D to be the set of edges on $P_{\widehat{e}}$ for all $\widehat{e} \in \widetilde{J}$. Clearly, each edge f embeds into a subpath of its previous embedding path (to reach the first and last vertex in S) and into some paths $P_{\widehat{e}}$ all of which now have edges in D. To bound the size of D, we observe that also each path $P_{\widehat{e}}$ is of short length since it is obtained from combining two old embedding paths (which were short) and a single edge. Thus, we have $|D| = |\bigcup_{\widehat{e} \in \widetilde{J}} P_{\widehat{e}}| = \widetilde{O}(|\widetilde{J}|) = \widetilde{O}(|U|)$ which again is only $\widetilde{O}(1)$ when amortizing over the number of updates. Figure 3 gives an example of this spanner maintenance procedure in action.

By using standard batching techniques, we can also deal with sequences of update batches $U^{(1)}, U^{(2)}, \ldots$ to the spanner and ensure that we cause only $m^{o(1)}$ amortized recourse per update/ size of D to the spanner.

An Efficient Implementation. While the algorithm above achieves low recourse, so far, we have not reasoned about the run-time. To do so, we enforce low *vertex-congestion* of $\Pi_{G/F_i \to \mathcal{S}(G,F_i)}$ defined to be the maximum number of paths $\Pi_{G/F_i \to \mathcal{S}(G,F_i)}(e)$ that any vertex $v \in V(G/F_i)$ occurs on. More precisely, we implement the algorithm above such that the vertex congestion of $\Pi_{G/F_i \to \mathcal{S}(G,F_i)}(e)$ remains of order $\gamma_c \Delta_{max}(G/F_i)$ for some $\gamma_c = m^{o(1)}$ over the entire course of the algorithm. We note that by a standard transformation, we can assume wlog that $\Delta_{max}(G/F_i) = \widetilde{O}(k)$.

Crucially, using our bound on the vertex congestion, we can argue that the graph J has maximum degree $\gamma_c \Delta_{max}(G/F_i)$. Since we can implement the static spanner algorithm in time near-linear in the number of edges, this implies that the entire algorithm to compute a sparsifier $\widetilde{J}$ only takes time $\sim |U| \gamma_c \Delta_{max}(G/F_i) \approx |U| m^{o(1)} k$, and thus in amortized time $k m^{o(1)}$ per update.

It remains to obtain this vertex congestion bound. Let us first discuss the static algorithm. Previously, we exploited that each sparsifier $\mathcal{S}_i$ is expander since it is a cut-sparsifier of H_i in a rather crude way. But it is not hard to see via the multi-commodity max-flow min-cut theorem [92] that this property can be used to argue the existence of an embedding $\Pi_{H_i \to \mathcal{S}_i}$ that uses each edge in $\mathcal{S}_i$ on at most $\widetilde{O}(1/p_i)$ embedding paths and therefore each path has average length $\widetilde{O}(1)$. In fact, using the shortest paths data structures on expanders [29], we can find such an embedding and turn the average length guarantee into a worst-case guarantee.

This ensures that each edge has congestion at most $\widetilde{O}(1/p_i) = \widetilde{O}(\Delta_{max}(G/F_i))$ and because $\mathcal{S}(G, F_i)$ has average degree $\widetilde{O}(1)$, this also bounds the vertex congestion. We need to refine this argument carefully for the dynamic version but can then argue that due to the batching we only increase the vertex congestion slightly. We refer the reader to Section 7 for the full implementation and analysis.

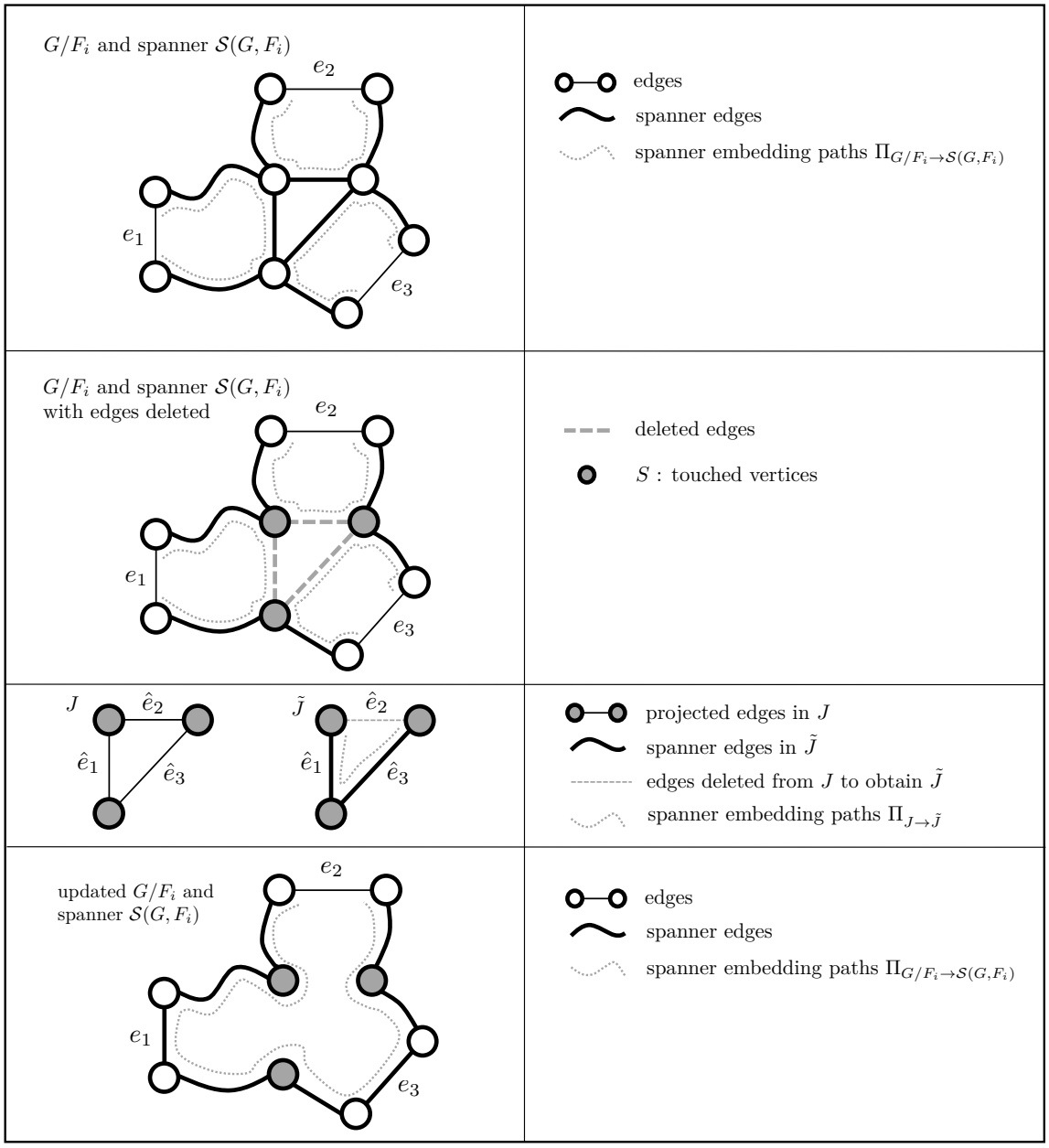

Fig. 3. Illustration of the procedure for maintaining $\mathcal{S}(G, F_i)$ under edge deletions.

4.4 Building Core Graphs

In this section, we describe our core graph construction (Definition 8.7), which maps our dynamic undirected min-ratio cycle problem on a graph G with at most m edges and vertices into a problem of the same type on a graph with only

$\widetilde{O}(m/k)$ vertices and m edges, and handles $\widetilde{O}(m/k)$ updates to the edges before we need to rebuild it. Our construction is based on constructing low-stretch decompositions using forests and portal routing (Lemma 8.5). We first describe how our portal routing uses a given forest F to construct a core graph G/F. We then discuss how to use a collection of (random) forests $F_1, \ldots, F_B$ to produce a low-stretch decomposition of G, which will ensure that one of the core graphs G/F_i preserves the witness circulation well. Portal routings played a key role in the ultrasparsifiers of [123] and has been further developed in many works since.

Forest Routings and Stretches. To understand how to define the stretch of an edge e with respect to a forest F, it is useful to define how to *route* an edge e in F. Given a spanning forest F, every path and cycle in G can be mapped to G/F naturally (where we allow G/F to contain self-loops). On the other hand if every connected component in F is rooted, where root_u^F denotes the root corresponding to a vertex $u \in V$, we can map every path and cycle in G/F back to G as follows. Let $P = e_1, \ldots, e_k$ be any (not necessarily simple) path in G/F where the preimage of every edge e_i is $e_i^G = (u_i^G, v_i^G) \in G$. The preimage of P, denoted P^G, is defined as the following concatenation of paths:

$$P^G \stackrel{\text{def}}{=} \bigoplus_{i=1}^{k} F[\text{root}_{u_i^G}^F, u_i^G] \oplus e_i^G \oplus F[v_i^G, \text{root}_{v_i^G}^F],$$

where we use $A \oplus B$ to denote the concatenation of paths A and B, and $F[a, b]$ to denote the unique ab-path in the forest F. When P is a circuit (i.e. a not necessarily simple cycle), P^G is a circuit in G as well. One can extend these maps linearly to all flow vectors and denote the resulting operators as $\mathbf{\Pi}_F : \mathbb{R}^{E(G)} \to \mathbb{R}^{E(G/F)}$ and $\mathbf{\Pi}_F^{-1} : \mathbb{R}^{E(G/F)} \to \mathbb{R}^{E(G)}$. Since we let G/F have self-loops, there is a bijection between edges of G and G/F and thus $\mathbf{\Pi}_F$ acts like the identity function.

To make our core graph construction dynamic, the key operation we need to support is the dynamic addition of more root nodes, which results in forest edges being deleted to maintain the invariant each connected component has a root node. Whenever an edge is changing in G, we ensure that G/F approximates the changed edge well by forcing both its endpoints to become root notes, which in turn makes the portal routing of the new edge trivial and this guarantees its stretch is 1. An example of this is shown in Figure 4.

For any edge $e^G = (u^G, v^G)$ in G with image e in G/F, we set $\widehat{\boldsymbol{\ell}}_e^F$, the edge length of e in G/F, to be *an upper bound* on the length of the *forest routing* of e, i.e. the path $F[\text{root}_{u^G}^F, u^G] \oplus e^G \oplus F[v^G, \text{root}_{v^G}^F]$. Meanwhile, we define $\widetilde{\text{str}}_e \stackrel{\text{def}}{=} \widehat{\boldsymbol{\ell}}_e^F / \boldsymbol{\ell}_e$, as an overestimate on the stretch of e w.r.t. the forest routing. A priori, it is unclear how to provide a single upper bound on the stretch of every edge, as the root nodes of the endpoints are changing over time. Providing such a bound for every edge is important for us as the lengths in G/F could otherwise be changing too often when the forest changes. We guarantee these bounds by scheme that makes auxiliary edge deletions in the forest in response to external updates, with these additional roots chosen carefully to ensure the length upper bounds.

Now, for any flow $\boldsymbol{f}$ in G/F, its length in G/F is at least the length of its pre-image in G, i.e. $\left\|\boldsymbol{L}\mathbf{\Pi}_F^{-1}\boldsymbol{f}\right\|_1 \leq \left\|\widehat{\boldsymbol{L}}^F\boldsymbol{f}\right\|_1$. Let $\mathbf{\Delta}^*$ be the optimal solution to (1). We will show later how to build F such that $\left\|\widehat{\boldsymbol{L}}^F\mathbf{\Delta}^*\right\|_1 \leq \gamma \left\|\boldsymbol{L}\mathbf{\Delta}^*\right\|_1$ holds for some $\gamma = m^{o(1)}$, solving (1) on G/F with edge length $\widehat{\boldsymbol{\ell}}$ and properly defined gradient $\widehat{\boldsymbol{g}}$ on G/F yields an $\frac{1}{\gamma}$-approximate solution for G. The gradient $\widehat{\boldsymbol{g}}$ is defined so that the total gradient of any circulation $\mathbf{\Delta}$ on G/F and its preimage $\mathbf{\Pi}_F^{-1}\mathbf{\Delta}$ in G is the same, i.e. $\widehat{\boldsymbol{g}}^\top\mathbf{\Delta} = \boldsymbol{g}^\top\mathbf{\Pi}_F^{-1}\mathbf{\Delta}$. The idea of incorporating gradients into portal routing was introduced in [86]; our version of this construction is somewhat different to allow us to make it dynamic efficiently.

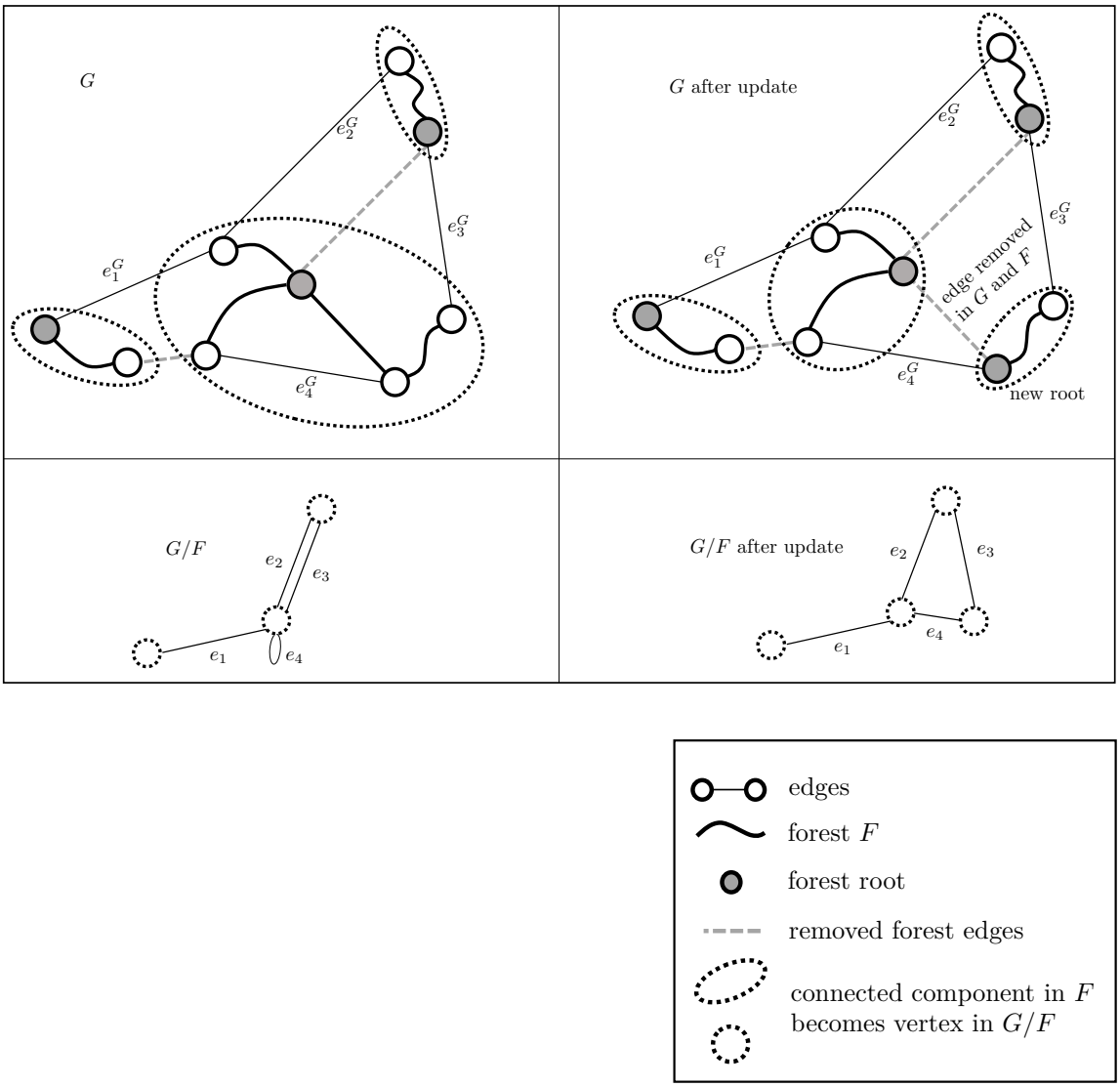

Fig. 4. Illustration of the core graph G/F changing as an edge is deleted in G (and in F).

Collections of Low Stretch Decompositions (LSD).. The first component of the data structure is constructing and maintaining forests of F that form a *Low Stretch Decomposition (LSD)* of G. Variations of which (such as j-trees) have been used to construct several recursive graph preconditioners [24, 82, 99, 119] and dynamic algorithms [23]. Informally, a k-LSD is a rooted forest $F \subseteq G$ that decomposes G into $O(m/k)$ vertex disjoint components. Given some positive edge weights $\boldsymbol{v} \in \mathbb{R}^E_{>0}$ and reduction factor $k > 0$, we compute a k-LSD F and length upper bounds $\widehat{\boldsymbol{\ell}}^F$ of G/F that satisfy two properties:

(1) $\widetilde{\text{str}}_e^F = \widehat{\boldsymbol{\ell}}_e^F / \boldsymbol{\ell}_{e^G} = \widetilde{O}(k)$ for any edge $e^G \in G$ with image e in G/F, and

(2) The weighted average of $\widetilde{\mathrm{str}}_e^F$ w.r.t. $\boldsymbol{v}$ is only $\widetilde{O}(1)$, i.e. $\sum_{e^G \in G} \boldsymbol{v}_{e^G} \cdot \widetilde{\mathrm{str}}_e^F \leq \widetilde{O}(1) \cdot \|\boldsymbol{v}\|_1$.

Item 1 guarantees that the solution to (1) for G/F yields a $\widetilde{O}(k)$-approximate one for G. However, this guarantee is not sufficient for our data structure, as our B-branching tree chain has $d \approx \log_k m$ levels of recursion and the quality of the solution from the deepest level would only be $\widetilde{O}(k)^d \approx m^{1+o(1)}$-approximate.

Instead, like [82, 99, 119] we compute k different edge weights $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_k$ via multiplicative weight updates (Lemma 8.6) so that the corresponding LSDs $F_1, \ldots, F_k$ have $\widetilde{O}(1)$ average stretch on every edge in G: $\sum_{j=1}^k \widetilde{\mathrm{str}}_e^{F_j} = \widetilde{O}(k)$, for all $e^G \in G$ with image e in G/F.

By Markov's inequality, for any fixed flow $\boldsymbol{f}$ in G, $\left\|\widehat{\boldsymbol{L}}^{F_j}\boldsymbol{f}\right\|_1 \leq \widetilde{O}(1)\,\|\boldsymbol{L}\boldsymbol{f}\|_1$ holds for at least half the LSDs corresponding to $F_1, \ldots, F_k$. Taking $\widetilde{O}(1)$ samples uniformly from $F_1, \ldots, F_k$, say $F_1, \ldots, F_B$ for $B = \widetilde{O}(1)$ we get that with high probability

$$\min_{j \in [B]} \left\|\widetilde{\mathrm{str}}^{F_j} \circ \boldsymbol{L}\boldsymbol{\Delta}^*\right\|_1 \leq \widetilde{O}(1) \left\|\boldsymbol{L}\boldsymbol{\Delta}^*\right\|_1. \tag{4}$$

That is, it suffices to solve (1) on $G/F_1, \ldots, G/F_B$ to find an $\widetilde{O}(1)$-approximate solution for G.

We provide all details including definitions and construction of the core graph in Section 8.

4.5 Maintaining a Branching Tree Chain

The goal of this section is to elaborate on how we combine core graphs and spanners to produce our overall data structure for our undirected min-ratio cycle problem, the B-branching tree chain. We also describe how the data structure is maintained under dynamic updates, which is more formally shown in Section 9. A central reason our hierarchical data structure works is that the components, both core graphs and spanners, are designed to remain very stable under dynamic changes to the input graphs they approximate. In the literature on dynamic graph algorithms, this is referred to as having *low recourse*.

(1) Sample and maintain $B = O(\log n)$ k-LSDs $F_1, F_2, \ldots, F_B$, and their associated core graphs G/F_i. Over the course of $O(m/k)$ updates at the top level, the forests F_i are *decremental*, i.e. only undergo edge deletions (from root insertions), and will have $\widetilde{O}(m/k)$ connected components.
(2) Maintain spanners $\mathcal{S}(G, F_i)$ of the core graphs G/F_i, and embeddings $\Pi_{E(G/F_i) \to \mathcal{S}(G,F_i)}$, say with length increase $\gamma_\ell = m^{o(1)}$.
(3) Recursively process the graphs $\mathcal{S}(G, F_i)$, i.e. maintains LSDs and core graphs on those, and spanners on the contracted graphs, etc. Go for d total levels, for $k^d = m$.
(4) Whenever a level i accumulates m/k^i total updates, hence doubling the number of edges in the graphs at that level, we rebuild levels $i, i+1, \ldots, d$.

Recall that on average, the LSDs stretch lengths by $\widetilde{O}(1)$, and the spanners $\mathcal{S}(G, F_i)$ stretch lengths by γ_ℓ. Hence the overall data structure stretches lengths by $\widetilde{O}(\gamma_\ell)^d = m^{o(1)}$ (for appropriately chosen d).

We now discuss details on how to update the forests G/F_i and spanners $\mathcal{S}(G, F_i)$. Intuitively, every time an edge $e = (u, v)$ is changed in G, we will delete $\widetilde{O}(1)$ additional edges from F_i. This ensures that no edge's total stretch/routing-length increases significantly due to the deletion of e (Lemma 8.5). As the forest F_i undergoes edge deletions, the graph G/F_i undergoes *vertex splits*, where a vertex has a subset of its edges moved to a newly inserted vertex. Thus, a key component of our data structure is to maintain spanners and embeddings of graphs undergoing vertex splits (as well as edge insertions/deletions). It is important that the amortized recourse (number of changes) to the spanner $\mathcal{S}(G, F_i)$ is

$m^{o(1)}$ independent of k, even though the average degree of G/F_i is $\Omega(k)$, and hence on average $\Omega(k)$ edges will move per vertex split in G/F_i. We discuss the more precise guarantees in Section 4.3.

Overall, let every level have recourse $\gamma_r = m^{o(1)}$ (independent of k) per tree. Then each update at the top level induces $O(B\gamma_r)^d$ (as each tree branches into B trees) updates in the data structure overall. Intuitively, for the proper choice of $d = \omega(1)$, both the total recourse $O(B\gamma_r)^d$ and approximation factor $\widetilde{O}(\gamma_\ell)^d$ are $m^{o(1)}$ as desired.

4.6 Going Beyond Oblivious Adversaries by using IPM Guarantees

The precise data structure in the previous section only works for *oblivious adversaries*, because we used that if we sampled $B = O(\log n)$ LSDs, then whp. there is a tree whose average stretch is $\widetilde{O}(1)$ with respect to a *fixed flow* $\boldsymbol{f}$. However, since we are updating the flow along the circulations returned by our data structure, we influence future updates, so the optimal circulations our data structure needs to preserve are not independent of the randomness used to generate the LSDs. To overcome this issue we leverage the key fact that the flow $\boldsymbol{f}^* - \boldsymbol{f}$ is a good witness for the min-ratio cycle problem at each iteration.

Lemma 6.7 states that for any flow $\boldsymbol{f}$, $\boldsymbol{g}(\boldsymbol{f})^\top \Delta(\boldsymbol{f}) / (100m + \|\boldsymbol{L}(\boldsymbol{f})\Delta(\boldsymbol{f})\|_1) \le -\widetilde{\Omega}(1)$ holds where $\Delta(\boldsymbol{f}) = \boldsymbol{f}^* - \boldsymbol{f}$. Then, the best solution to (1) among the LSDs $G/F_1, \ldots, G/F_B$ maintains an $\widetilde{O}(1)$-approximation of the quality of the witness $\Delta(\boldsymbol{f}) = \boldsymbol{f}^* - \boldsymbol{f}$ as long as

$$\min_{j \in [B]} \left\| \widehat{\boldsymbol{L}}^{F_j} \Delta(\boldsymbol{f}) \right\|_1 \le \widetilde{O}(1) \left\| \boldsymbol{L}(\boldsymbol{f})\Delta(\boldsymbol{f}) \right\|_1 + \widetilde{O}(m). \tag{5}$$

In this case, let $\widehat{\Delta}$ be the best solution obtained from $G/F_1, \ldots, G/F_B$. We have

$$\frac{\boldsymbol{g}(\boldsymbol{f})^\top \widehat{\Delta}}{\left\| \boldsymbol{L}(\boldsymbol{f})\widehat{\Delta} \right\|_1} \le \frac{\boldsymbol{g}(\boldsymbol{f})^\top \Delta(\boldsymbol{f})}{\widetilde{O}(1) \left\| \boldsymbol{L}(\boldsymbol{f})\Delta(\boldsymbol{f}) \right\|_1 + \widetilde{O}(m)} = -\widetilde{\Omega}(1).$$

The additive $\widetilde{O}(m)$ term is there for a technical reason discussed later.

To formalize this intuition, we define the *width* $\boldsymbol{w}(\boldsymbol{f})$ of $\Delta(\boldsymbol{f})$ as $\boldsymbol{w}(\boldsymbol{f}) = 100 \cdot \mathbf{1} + |\boldsymbol{L}(\boldsymbol{f})\Delta(\boldsymbol{f})|$. The name comes from the fact that $\boldsymbol{w}(\boldsymbol{f})_e$ is always at least $\left|\boldsymbol{\ell}(\boldsymbol{f})_e(\boldsymbol{f}^*_e - \boldsymbol{f}_e)\right|$ for any edge e. We show that the width is also slowly changing (Lemma 11.2) across IPM iterations, in that if the width changed by a lot, then the residual capacity of e must have changed significantly. This gives our data structure a way to predict which edges' contribution to the length of the witness flow $\boldsymbol{f}^* - \boldsymbol{f}$ could have significantly increased.

Observe that for any forest F_j in the LSD of G, we have $\left\| \widehat{\boldsymbol{L}}^{F_j} \Delta(\boldsymbol{f}) \right\|_1 \le \left\| \widetilde{\mathrm{str}}^{F_j} \circ \boldsymbol{w}(\boldsymbol{f}) \right\|_1$. Thus, we can strengthen (5) and show that the IPM potential can be decreased by $m^{-o(1)}$ if

$$\min_{j \in [B]} \left\| \widetilde{\mathrm{str}}^{F_j} \circ \boldsymbol{w}(\boldsymbol{f}) \right\|_1 \le \widetilde{O}(1) \left\| \boldsymbol{w}(\boldsymbol{f}) \right\|_1. \tag{6}$$

(6) also holds with w.h.p if the collection of LSDs are built after knowing $\boldsymbol{f}$. However, this does not necessarily hold after augmenting with $\boldsymbol{\Delta}$, an approximate solution to (1).

Due to stability of $\boldsymbol{w}(\boldsymbol{f})$, we have $\boldsymbol{w}(\boldsymbol{f} + \Delta)_e \approx \boldsymbol{w}(\boldsymbol{f})_e$ for every edge e whose length does not change a lot. For other edges, we update their edge length and force the stretch to be 1, i.e. $\widetilde{\mathrm{str}}^{F_j}_e = 1$ via the dynamic LSD maintenance, by shortcutting the routing of the edge e at its endpoints. This gives that for any $j \in [B]$, the following holds:

$$\left\| \widetilde{\mathrm{str}}^{F_j} \circ \boldsymbol{w}(\boldsymbol{f} + \Delta) \right\|_1 \lesssim \left\| \widetilde{\mathrm{str}}^{F_j} \circ \boldsymbol{w}(\boldsymbol{f}) \right\|_1 + \left\| \boldsymbol{w}(\boldsymbol{f} + \Delta) \right\|_1.$$

Using the fact that $\min_{j\in[B]} \left\|\widetilde{\mathrm{str}}^{F_j} \circ \boldsymbol{w}(\boldsymbol{f})\right\|_1 \le \widetilde{O}(1)\left\|\boldsymbol{w}(\boldsymbol{f})\right\|_1$, we have the following:

$$\min_{j\in[B]} \left\|\widetilde{\mathrm{str}}^{F_j} \circ \boldsymbol{w}(\boldsymbol{f}+\boldsymbol{\Delta})\right\|_1 \lesssim \widetilde{O}(1)\left\|\boldsymbol{w}(\boldsymbol{f})\right\|_1 + \left\|\boldsymbol{w}(\boldsymbol{f}+\boldsymbol{\Delta})\right\|_1.$$

Thus, solving (1) on the updated $G/F_1, \ldots, G/F_B$ yields a good enough solution for reducing IPM potential as long as the width of $\boldsymbol{w}(\boldsymbol{f}+\boldsymbol{\Delta})$ has not increased significantly, i.e. $\left\|\boldsymbol{w}(\boldsymbol{f}+\boldsymbol{\Delta})\right\|_1 \le \widetilde{O}(1)\left\|\boldsymbol{w}(\boldsymbol{f})\right\|_1$.

If the solution on the updated graphs $G/F_1, \ldots, G/F_B$ does not have a good enough quality, we know by the above discussion that $\left\|\boldsymbol{w}(\boldsymbol{f}+\boldsymbol{\Delta})\right\|_1 \ge 100\left\|\boldsymbol{w}(\boldsymbol{f})\right\|_1$ must hold. Then, we re-compute the collection of LSDs of G and solve (1) on the new collection of $G/F_1, \ldots, G/F_B$ again. Because each recomputation reduces the ℓ_1 norm of the width by a constant factor, and all the widths are bounded by $\exp(\log^{O(1)} m)$ (as discussed in Section 4.1), there can be at most $\widetilde{O}(1)$ such recomputations. At the top level, this only increases our runtime by $\widetilde{O}(1)$ factors.

The real situation is much more complicated since we recursively maintain the solutions on the spanners of each $G/F_1, \ldots, G/F_B$. Hence, it is possible that lower levels in the data structure are the "reason" that the quality of the solution is poor. More formally, let T be the total number of IPM iterations. We use $t \in [T]$ to index each iteration and use superscript $x^{(t)}$ to denote the state of any variable x after t-th iteration. For example, $\boldsymbol{f}^{(t)}$ is the flow computed so far after t IPM iterations and we define $\boldsymbol{w}^{(t)} \stackrel{\text{def}}{=} \boldsymbol{w}(\boldsymbol{f}^{(t)})$ to be the width w.r.t. $\boldsymbol{f}^{(t)}$. Recall that every graph maintained in the dynamic B-Branching Tree Chain re-computes its collection of LSDs after certain amount of updates. When some graph at level i re-computes, we enforce every graph at the same level to re-compute as well. Since there's only $m^{o(1)}$ such graphs at each level, this scheme results in a $m^{o(1)}$ overhead on the update time which is tolerable. For every level $i = 0, \ldots, d$, we define $\mathrm{prev}_i^{(t)}$ to be the most recent iteration at or before t that a re-computation of LSDs occurs at level i. For graphs at level d which contain only $m^{o(1)}$ vertices, we enforce a rebuild everytime and always have $\mathrm{prev}_d^{(t)} = t$. We show in Lemma 9.9 that the cycle output by the data structure in the t-th IPM iteration has length at most

$$m^{o(1)} \sum_{i=0}^{d} \left\|\boldsymbol{w}^{(\mathrm{prev}_i^{(t)})}\right\|_1.$$

This inequality is a natural generalization of the $\widetilde{O}(1)\left(\left\|\boldsymbol{w}(\boldsymbol{f})\right\|_1 + \left\|\boldsymbol{w}(\boldsymbol{f}+\boldsymbol{\Delta})\right\|_1\right)$-bound when taking recursive structure into account.

At this point, we want to emphasize that the fact that we can prove this guarantee depends on certain "monotonicity" properties of both our core and spanner graph constructions. In the core graph construction, it is essential that we can provide a fixed length upper bound for most edges. In the spanner construction, we crucially use that the set of edges routing into any fixed edge in the spanner is *decremental* for most spanner edges. This allows us to produce an initial upper bound on the width for edges in the spanner and continue using this bound as long as the spanner edge routes a decremental set.

The cycle output by the data structure yields enough decrease in the IPM potential if its 1-norm is small enough. Otherwise, the 1-norm of the output cycle is large and we know that $\sum_{i=0}^{d} \left\|\boldsymbol{w}^{(\mathrm{prev}_i^{(t)})}\right\|_1$ is much more than $m^{o(1)}\left\|\boldsymbol{w}^{(t)}\right\|_1$. In this way, the data structure can fail because some lower level i has $\left\|\boldsymbol{w}^{(\mathrm{prev}_i^{(t)})}\right\|_1 \gg \left\|\boldsymbol{w}^{(t)}\right\|_1$. A possible fix is to rebuild the entire data structure which sets $\mathrm{prev}_i^{(t)} = t$ at any level i. However, this costs linear time per rebuild, and this may need to happen almost every iteration because there are multiple levels. In the next section we show how to leverage that lower levels have cheaper rebuilding times (levels $i, i+1, \ldots, d$ can be rebuilt in time approximately $m^{1+o(1)}/k^i$) to design a more efficient rebuilding schedule.

4.7 The Rebuilding Game

Our goal in Section 10 is to develop a strategy that finds approximate min-ratio cycles without spending too much time rebuilding our data structure when it fails to do so. In the previous overview section, we carefully characterized the conditions under which our data structure can fail against adversarial updates, given the promise that $\boldsymbol{f}^* - \boldsymbol{f}$ remains a good witness circulation. In this section, we set up a game which abstracts the properties of the data structure and the adversary. The player in this game wants to ensure our data structure works correctly by rebuilding levels of it when it fails. We show that the player can win without spending too much time on rebuilding.

Recall $\boldsymbol{w}^{(t)} \stackrel{\text{def}}{=} \boldsymbol{w}(\boldsymbol{f}^{(t)})$ is a hidden vector that we use to upper bound the ℓ_1 cost of the hidden witness circulation $\boldsymbol{\Delta}(\boldsymbol{f})$. We will refer to $\|\boldsymbol{w}^{(t)}\|_1$ as the total width at time t. We argued in the previous Section 4.6 that our branching-tree data structure can find a good cycle whenever the total width $\|\boldsymbol{w}^{(t)}\|_1$ is not too small compared to the total widths at the times when the levels $0, 1, \ldots, d$ of the data structure were last initialized or rebuilt. We let $\text{prev}_i^{(t)}$ denote the stage when level i was last rebuilt, and refer to $\|\boldsymbol{w}^{(\text{prev}_i^{(t)})}\|_1$ as the total width at level i. As we saw in the previous section, the only way our cycle-finding data structure can fail to produce a good enough cycle is if $\sum_{i=0}^{d} \|\boldsymbol{w}^{(\text{prev}_i^{(t)})}\|_1 \gg m^{o(1)}\|\boldsymbol{w}^{(t)}\|_1$. We can estimate the quality of the cycles we find, and if we fail to find a good cycle we can conclude this undesired condition holds. However, even if the condition holds, we might still find a good cycle "by accident", so finding a cycle does not prove that the data structure currently estimates the total width well. Because the total widths $\|\boldsymbol{w}^{(t)}\|_1$ are hidden from us, we do not know which level(s) cause the problem when we fail to find a cycle.

We turn this into a game that abstracts the data structure and IPM and supposes that total width $\|\boldsymbol{w}^{(t)}\|_1$ is an arbitrary positive number chosen by an adversary, while a player (our protagonist) manages the data structure by rebuilding levels of the data structure to set $\text{prev}_i^{(t)} = t$ when necessary. Now, because of well-behaved numerical properties of our IPM, we are guaranteed that $\log(\|\boldsymbol{w}^{(t)}\|_1) \in [-\text{poly}\log(m), \text{poly}\log(m)]$, and we impose this condition on the total width in our game as well. By developing a strategy that works against any adversary choosing such total widths, we ensure our data structure will work with our IPM as a special case. In Definition 10.1 we formally define our rebuilding game.

In our branching tree data structure, level i can be rebuilt at a cost of $m^{1+o(1)}/k^i$ and it can last through roughly $m^{1-o(1)}/k^i$ cycle updates before we have to rebuild it because the core graph has grown too large (we call this a "winning rebuild"). But, if we are unable to find a good cycle, we are forced to rebuild sooner (we call this a "losing rebuild"). Which level should we rebuild if we are unable to find a good cycle? The answer is not immediately clear, because any level could have too large total width. However, by tuning our parameters such that the $m^{o(1)}$ factor in our condition $\sum_{i=0}^{d} \|\boldsymbol{w}^{(\text{prev}_i^{(t)})}\|_1 \gg m^{o(1)}\|\boldsymbol{w}^{(t)}\|_1$ is larger than $2(d+1)$, we can deduce that if a failure occurs, then $\max_{i=0}^{d} \|\boldsymbol{w}^{(\text{prev}_i^{(t)})}\|_1 > 2\|\boldsymbol{w}^{(t)}\|_1$. Thus, *if* the total width at level i is too large, then a losing rebuild at level i (and hence updating $\boldsymbol{w}^{(\text{prev}_i^{(t+1)})}$ to $\boldsymbol{w}^{(t)}$) will reduce its total width by at least a factor 2.

This means that for any level i, if we do a losing rebuild of level i $\text{poly}\log(m)$ times before a winning rebuild of level i, we can conclude that the too-large total width is not at level i. This leads to the following strategy: Starting at the lowest level, do a losing rebuild of each level i up to $\text{poly}\log(m)$ times after each winning rebuild, and then move to rebuilding level $i-1$ in case of more failures. We state this strategy more formally in Algorithm 6. This leads to a cost of $O(m^{o(1)}(m+T))$ to process T cycle updates in the rebuilding game, as we prove in Lemma 10.3.

Finally, at the end of Section 10, we combine the data structure designed in the previous sections with our strategy for the rebuilding game to create a data structure that handles successfully finds update cycles in our hidden stable-flow chasing setting in amortized $m^{o(1)}$ cost per cycle update, which is encapsulated in Theorem 8.2.

5 PRELIMINARIES

Model of Computation. In this article, for problem instances encoded with z bits, all algorithms work in fixed-point arithmetic where words have $O(\log^{O(1)} z)$ bits, i.e. we prove that all numbers stored are in $[\exp(-\log^{O(1)} z), \exp(\log^{O(1)} z)]$.

General notions. We denote vectors by boldface lowercase letters. We use uppercase boldface to denote matrices. Often, we use uppercase matrices to denote the diagonal matrices corresponding to lowercase vectors, such as $\boldsymbol{L} = \mathrm{diag}(\boldsymbol{\ell})$. For vectors $\boldsymbol{x}, \boldsymbol{y}$ we define the vector $\boldsymbol{x} \circ \boldsymbol{y}$ as the entrywise product, i.e. $(\boldsymbol{x} \circ \boldsymbol{y})_i = \boldsymbol{x}_i \boldsymbol{y}_i$. We also define the entrywise absolute value of a vector $|\boldsymbol{x}|$ as $|\boldsymbol{x}|_i = |\boldsymbol{x}_i|$. We use $\langle \cdot, \cdot \rangle$ as the vector inner product: $\langle \boldsymbol{x}, \boldsymbol{y} \rangle = \boldsymbol{x}^\top \boldsymbol{y} = \sum_i \boldsymbol{x}_i \boldsymbol{y}_i$. We elect to use this notation when $\boldsymbol{x}, \boldsymbol{y}$ have superscripts (such as time indexes) to avoid cluttering. For positive real numbers a, b we write $a \approx_\alpha b$ for some $\alpha > 1$ if $\alpha^{-1} b \le a \le \alpha b$. For positive vectors $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^{[n]}_{>0}$, we say $\boldsymbol{x} \approx_\alpha \boldsymbol{y}$ if $\boldsymbol{x}_i \approx_\alpha \boldsymbol{y}_i$ for all $i \in [n]$. This notion extends naturally to positive diagonal matrices. We will need the standard Chernoff bound.

Theorem 5.1 (Chernoff Bound). *Suppose $X_1, X_2, \ldots, X_k \in [0, W]$ are independent random variables, $X = \sum_i X_i$ and $\mu = \mathbb{E}X$. For any $\delta \ge 1/2$, we have $\mathbb{P}[X \in [(1-\delta)\mu, (1+\delta)\mu]] \ge 1 - 2e^{-\frac{\delta\mu}{6W}}$.*

Graphs. In this article, we consider multi-graphs G, with edge set $E(G)$ and vertex set $V(G)$. When the graph is clear from context, we use the short-hands E for $E(G)$, V for $V(G)$, $m = |E|, n = |V|$. We assume that each edge $e \in E$ has an implicit direction, used to define its edge-vertex incidence matrix $\boldsymbol{B}$. Abusing notation slightly, we often write $e = (u, v) \in E$ where e is an edge in E and u and v are the tail and head of e respectively (note that technically multi-graphs do not allow for edges to be specified by their endpoints). We let $\mathrm{rev}(e)$ be the edge e reversed: if $e = (u, v)$ points from u to v, then $\mathrm{rev}(e)$ points from v to u.

We say a flow $\boldsymbol{f} \in \mathbb{R}^E$ routes a demand $\boldsymbol{d} \in \mathbb{R}^V$ if $\boldsymbol{B}^\top \boldsymbol{f} = \boldsymbol{d}$. For an edge $e = (u, v) \in G$ we let $\boldsymbol{b}_e \in \mathbb{R}^V$ denote the demand vector of routing one unit from u to v.

We denote by $\deg_G(v)$ the degree of v in G, i.e. the number of incident edges. We let $\Delta_{\max}(G)$ and $\Delta_{\min}(G)$ denote the maximum and minimum degree of graph H. We define the volume of a set $S \subseteq V$ as $\mathrm{vol}_G(S) \stackrel{\text{def}}{=} \sum_{v \in S} \deg_G(v)$.

Dynamic Graphs. We say G is a *dynamic* graph, if it undergoes *batches* $U^{(1)}, U^{(2)}, \ldots$ of updates consisting of edge insertions/ deletions and/or vertex splits that are applied to G. We stress that results on dynamic graphs in this article often only consider a subset of the update types and we therefore explicitly state for each dynamic graph which updates are allowed. We say that the graph G, after applying the first t update batches $U^{(1)}, U^{(2)}, \ldots, U^{(t)}$, is at *stage* t and denote the graph at this stage by $G^{(t)}$. Additionally, when G is clear, we often denote the value of a variable x at the end of stage t of G by $x^{(t)}$, or a vector $\boldsymbol{x}$ at the end of stage t of G by $\boldsymbol{x}^{(t)}$.

For each update batch $U^{(t)}$, we encode edge insertions by a tuple of tail and head of the new edge and deletions by a pointer to the edge that is about to be deleted. We further also encode vertex splits by a sequence of edge insertions and deletions as follows: if a vertex v is about to be split and the vertex that is split off is denoted v^{NEW}, we can delete all edges that are incident to v but should be incident to v^{NEW} from v and then re-insert each such edge via an insertion (we allow insertions to new vertices, that do not yet exist in the graph).

For technical reasons, we assume that in an update batch $U^{(t)}$, the updates to implement the vertex splits are last, and that we always encode a vertex split of v into v and v^{NEW} such that $\deg_{G^{(t+1)}}(v^{\mathrm{NEW}}) \le \deg_{G^{(t+1)}}(v)$. We let the vertex set of graph $G^{(t)}$ consist of the union of all endpoints of edges in the graph (in particular if a vertex is split, the new vertex v^{NEW} is added due to having edge insertions incident to this new vertex v^{NEW} in $U^{(t)}$).

$\textsc{Enc}(u)$ of an update $u \in U^{(t)}$ be the size of the encoding of the update and note that for edge insertions/ deletions, we have $\textsc{Enc}(u) = \widetilde{O}(1)$ and for a vertex split of v into v and v^{NEW} as described above we have $\textsc{Enc}(u) =$

$\widetilde{O}(\deg_{G^{(t+1)}}(v^{\text{NEW}}))$. For a batch of updates U, we let $\textsc{Enc}(U) = \sum_{u\in U}\textsc{Enc}(u)$. In this article, we only consider dynamic graphs where the total size of the encodings of all update batches is polynomially bounded in the size of the initial graph $G^{(0)}$.

We point out in particular that the number of updates $|U|$ in an update batch U can be completely different from the actual encoding size $\textsc{Enc}(U)$ of the update batch U.

Paths, Flows, and Trees. Given a path P in G with vertices u, v both on P, then we let $P[u, v]$ denote the path segment on P from u to v. We note that if v precedes u on P, then the segment $P[u, v]$ is in the reverse direction of P. For forests F, we similarly define $F[u, v]$ as the path from u to v along edges in the forest F. We ensure that u, v are in the same connected component of F whenever this notation is used.

We let $\boldsymbol{p}(F[u, v]) \in \mathbb{R}^{E(G)}$ denote the flow vector routing one unit from u to v along the path in F. In this way, $|\boldsymbol{p}(F[u, v])|$ is the indicator vector for the path from u to v on F. Note that $\boldsymbol{p}(F[u, v]) + \boldsymbol{p}(F[v, w]) = \boldsymbol{p}(F[u, w])$ for any vertices $u, v, w \in V$.

The *stretch* of $e = (u, v)$ with respect to a tree T is defined as

$$\text{str}_e^{T,\boldsymbol{\ell}} \stackrel{\text{def}}{=} 1 + \frac{\langle \boldsymbol{\ell}, |\boldsymbol{p}(T[u, v])|\rangle}{\boldsymbol{\ell}_e} = 1 + \frac{\sum_{e'\in T[u,v]} \boldsymbol{\ell}_{e'}}{\boldsymbol{\ell}_e}.$$

This differs slightly from the more common definition of stretch because of the 1+ term – we do this to ensure that $\text{str}_e^{T,\boldsymbol{\ell}} \geq 1$ for all e. It is known how to efficiently construct trees with polylogarithmic average stretch with respect to underlying weights. These are called low-stretch spanning trees (LSSTs).

Theorem 5.2 (Static LSST [2]). *Given a graph $G = (V, E)$ with lengths $\boldsymbol{\ell} \in \mathbb{R}^E_{>0}$ and weights $\boldsymbol{v} \in \mathbb{R}^E_{>0}$ there is an algorithm that runs in time $\widetilde{O}(m)$ and computes a tree T such that $\sum_{e\in E} \boldsymbol{v}_e \text{str}_e^{T,\boldsymbol{\ell}} \leq O(\|\boldsymbol{v}\|_1 \log n \log\log n)$.*

We let $\gamma_{LSST} \stackrel{\text{def}}{=} O(\log n \log\log n)$.

Graph Embeddings. Given graphs G and H with $V(G) \subseteq V(H)$, we say that $\Pi_{G\to H}$ is an *graph-embedding* from G into H if it maps each edge $e^G = (u, v) \in E(G)$ to a u-v path $\Pi_{G\to H}(e^G)$ in H. We define congestion of an edge e^H by $\text{econg}(\Pi_{G\to H}, e^H) \stackrel{\text{def}}{=} |\{e^G \in E(G) \mid e^H \in \Pi_{G\to H}(e^G)\}|$ and of the embedding by $\text{econg}(\Pi_{G\to H}) \stackrel{\text{def}}{=} \max_{e^H\in E(H)} \text{econg}(\Pi_{G\to H}, e^H)$. Analogously, the congestion of a vertex $v^H \in V(H)$ is defined by $\text{vcong}(\Pi_{G\to H}, v^H) \stackrel{\text{def}}{=} |\{e^G \in E(G) \mid v^H \in \Pi_{G\to H}(e^G)\}|$ and the vertex-congestion of the embedding by $\text{vcong}(\Pi_{G\to H}) \stackrel{\text{def}}{=} \max_{v^H\in V(H)} \text{vcong}(\Pi_{G\to H}, v^H)$. We define the length by $\text{length}(\Pi_{G\to H}) \stackrel{\text{def}}{=} \max_{e^G\in E(G)} |\Pi_{G\to H}(e^G)|$. We let (boldface) $\boldsymbol{\Pi}_{G\to H}(e) \in \mathbb{R}^E$ for $e = (u, v)$ denote a vector representing the flow from $u \to v$. Thus $\boldsymbol{B}^\top \boldsymbol{\Pi}_{G\to H}(e) = \boldsymbol{b}_e$.

For a path p_1 with endpoints $u \to v$ and p_2 with endpoints $v \to w$, we define $p_1 \oplus p_2$ as the concatenation, which is a path from $u \to w$.

Sometimes we consider the edges that route into an edge $e \in E(H)$. Given graphs G, H and embedding $\Pi_{G\to H}$, for edge $e \in E(H)$ we define $\Pi^{-1}_{G\to H}(e) \stackrel{\text{def}}{=} \{e' \in G : e \in \Pi(e')\}$. The notation is natural if we think of Π as a function from an edge e to the set of edges in its path, and hence Π^{-1} is the inverse/preimage of a one-to-many function.

Dynamic trees. Our algorithms make heavy use of dynamic tree data structures, so we state a lemma describing the variety of operations that can be supported on a dynamic tree. This includes path updates either of the form adding a directed flow along a tree path, or adding a positive value to each edge on a tree path. Additionally, the data structure can support changing edges in the tree, and querying flow values on edge. Each of these operations can be performed in amortized $\widetilde{O}(1)$ time.

LEMMA 5.3 (DYNAMIC TREES, SEE [122]). *There is a deterministic data structure $\mathcal{D}^{(T)}$ that maintains a dynamic tree $T \subseteq G = (V, E)$ under insertion/deletion of edges with gradients $\boldsymbol{g}$ and lengths $\boldsymbol{\ell}$, 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 $\boldsymbol{\ell}_e$. The amortized time is $\widetilde{O}(1)$ per change.*
2. *For a path vector $\Delta = \boldsymbol{p}(T[u, v])$ for some $u, v \in V$, return $\langle \boldsymbol{g}, \Delta \rangle$ or $\langle \boldsymbol{\ell}, |\Delta| \rangle$ in time $\widetilde{O}(1)$.*
3. *Maintain a flow $\boldsymbol{f} \in \mathbb{R}^E$ under operations $\boldsymbol{f} \leftarrow \boldsymbol{f} + \eta\Delta$ for $\eta \in \mathbb{R}$ and path vector $\Delta = \boldsymbol{p}(T[u, v])$, 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 $\boldsymbol{f} \leftarrow \boldsymbol{f} + \eta|\Delta|$ for $\eta \in \mathbb{R}_{\geq 0}$ and path vector $\Delta = \boldsymbol{p}(T[u, v])$, 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 $\Delta^{(t)}$ is the update vector at time t, returns*

$$S^{(t)} \stackrel{\text{def}}{=} \left\{ e \in E : \boldsymbol{\ell}_e \sum_{t' \in [\text{last}_e^{(t)}+1, t]} |\Delta_e^{(t')}| \geq \varepsilon \right\} \tag{7}$$

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

PROOF. Every operation described is standard except for DETECT, which we now give an algorithm for. Note that (7) is equivalent to the following:

$$\sum_{t' \in [\text{last}_e^{(t)}+1, t]} |\Delta_e^{(t')}| - \frac{\varepsilon}{\boldsymbol{\ell}_e} \geq 0.$$

This value can be maintained using positive flow updates (item 4), i.e. $|\Delta|$ to a tree path. We reset the value of an edge e to $-\varepsilon/\boldsymbol{\ell}_e$ once it is detected. Locating and collecting edges satisfying (7) is reduced to finding edges with nonnegative values, which can be done in $\widetilde{O}(|S^{(t)}|)$ time by repeatedly querying the largest value on the tree, and checking whether it is nonnegative. □

The DETECT operation allows our algorithm to decide when we need to change the gradients and lengths of an edge e in our IPM.

6 POTENTIAL REDUCTION INTERIOR POINT METHOD

The goal of this section is to present a primal-only potential reduction IPM [78] that solves the min-cost flow problem on a graph $G = (V, E)$ with demands $\boldsymbol{d} \in \mathbb{Z}^V$, lower and upper capacities $\boldsymbol{u}^-, \boldsymbol{u}^+ \in \mathbb{Z}^E$, and costs $\boldsymbol{c} \in \mathbb{Z}^E$ such that all integers are bounded by U:

$$\boldsymbol{f}^* \stackrel{\text{def}}{=} \operatorname*{arg\,min}_{\substack{\boldsymbol{B}^\top \boldsymbol{f} = \boldsymbol{d} \\ \boldsymbol{u}_e^- \leq \boldsymbol{f}_e \leq \boldsymbol{u}_e^+ \text{ for all } e \in E}} \boldsymbol{c}^\top \boldsymbol{f}. \tag{8}$$

Instead of using the standard logarithmic barrier, we elect to use the barrier $x^{-\alpha}$ for small α. This is because we do not know how to prove that the lengths encountered during the algorithms are quasipolynomially bounded for the logarithmic barrier. Precisely, we consider the following potential function, where $F^* \stackrel{\text{def}}{=} \boldsymbol{c}^\top \boldsymbol{f}^*$ is the optimal value for

(8), and $\alpha \stackrel{\text{def}}{=} 1/(1000 \log mU)$. We assume that we know F^*, as running our algorithm allows us to binary search for F^*.

$$\Phi(\boldsymbol{f}) \stackrel{\text{def}}{=} 20m \log(\boldsymbol{c}^\top \boldsymbol{f} - F^*) + \sum_{e \in E} \left((\boldsymbol{u}_e^+ - \boldsymbol{f}_e)^{-\alpha} + (\boldsymbol{f}_e - \boldsymbol{u}_e^-)^{-\alpha} \right) \tag{9}$$

We show in Section 6.3 that we can initialize a flow $\boldsymbol{f}$ on a larger graph (still with $O(m)$ edges) such that the potential $\Phi(\boldsymbol{f})$ is initially $O(m \log mU)$ (Lemma 6.12). Additionally, given a nearly optimal solution, we can recover an exactly optimal solution to the original min-cost flow problem in linear time (Lemma 6.11). A simple observation is that if the potential is sufficiently small, then the cost of the flow is nearly optimal.

Lemma 6.1. *We have* $\boldsymbol{c}^\top \boldsymbol{f} - F^* \leq \exp\left(\Phi(\boldsymbol{f})/(20m)\right)$. *In particular, if* $\Phi(\boldsymbol{f}) \leq -200m \log mU$ *then* $\boldsymbol{c}^\top \boldsymbol{f} - F^* \leq (mU)^{-10}$.

Proof. From (9) and the fact that $\boldsymbol{u}_e^- \leq \boldsymbol{f}_e \leq \boldsymbol{u}_e^+$, we get

$$\Phi(\boldsymbol{f}) \geq 20m \log(\boldsymbol{c}^\top \boldsymbol{f} - F^*).$$

Rearranging this gives the desired result. □

Given a flow $\boldsymbol{f} \in \mathbb{R}^E$ we define lengths $\boldsymbol{\ell} \in \mathbb{R}^E_{>0}$ and gradients $\boldsymbol{g} \in \mathbb{R}^E$ to capture the next ℓ_1 problem we solve to decrease the potential.

Definition 6.2 (Lengths and gradients)**.** *Given a flow* $\boldsymbol{f} \in \mathbb{R}^E$ *we define lengths* $\boldsymbol{\ell} \in \mathbb{R}^E$ *as*

$$\boldsymbol{\ell}(\boldsymbol{f})_e \stackrel{\text{def}}{=} \left(\boldsymbol{u}_e^+ - \boldsymbol{f}_e\right)^{-1-\alpha} + \left(\boldsymbol{f}_e - \boldsymbol{u}_e^-\right)^{-1-\alpha} \tag{10}$$

and gradients $\boldsymbol{g} \in \mathbb{R}^E$ *as* $\boldsymbol{g}(\boldsymbol{f}) \stackrel{\text{def}}{=} \nabla\Phi(\boldsymbol{f})$. *More explicitly,*

$$\boldsymbol{g}(\boldsymbol{f})_e \stackrel{\text{def}}{=} [\nabla\Phi(\boldsymbol{f})]_e = 20m(\boldsymbol{c}^\top \boldsymbol{f} - F^*)^{-1}\boldsymbol{c}_e + \alpha\left(\boldsymbol{u}_e^+ - \boldsymbol{f}_e\right)^{-1-\alpha} - \alpha\left(\boldsymbol{f}_e - \boldsymbol{u}_e^-\right)^{-1-\alpha} \tag{11}$$

The remainder of the section is split into three parts. In Section 6.1 we show that approximately solving the cycle problem induced by gradients and lengths approximating those in Definition 6.2 allows us to decrease the potential additively by an almost constant quantity in a single iteration. Then in Section 6.2 we bound how such iterations affect the lengths and gradients in order to show that approximate versions of them only need to be modified $m^{1+o(1)}$ times across the entire algorithm, and in Section 6.3 we discuss how to get an initial flow and extract an exact min-cost flow from a nearly optimal flow.

The following theorem summarizes the results of this section.

Theorem 6.3. *Suppose we are given a min-cost flow instance given by Equation (8). Let* $\boldsymbol{f}^*$ *denote an optimal solution to the instance.*

For all $\kappa \in (0, 1)$, *there is a potential reduction interior point method for this problem, that, given an initial flow* $\boldsymbol{f}^{(0)} \in \mathbb{R}^E$ *such that* $\Phi(\boldsymbol{f}^{(0)}) \leq 200m \log mU$, *the algorithm proceeds as follows:*

The algorithm runs for $\widetilde{O}(m\kappa^2)$ *iterations. At each iteration, let* $\boldsymbol{g}(\boldsymbol{f}^{(t)}) \in \mathbb{R}^E$ *denote that gradient and* $\boldsymbol{\ell}(\boldsymbol{f}^{(t)}) \in \mathbb{R}^E_{>0}$ *denote the lengths given by Definition 6.2. Let* $\widetilde{\boldsymbol{g}} \in \mathbb{R}^E$ *and* $\widetilde{\boldsymbol{\ell}} \in \mathbb{R}^E_{>0}$ *be any vectors such that* $\left\| \boldsymbol{L}(\boldsymbol{f}^{(t)})^{-1}\left(\widetilde{\boldsymbol{g}} - \boldsymbol{g}(\boldsymbol{f}^{(t)})\right)\right\|_\infty \leq \kappa/8$ *and* $\widetilde{\boldsymbol{\ell}} \approx_2 \boldsymbol{\ell}(\boldsymbol{f})$.

(1) *At each iteration, the hidden circulation* $\boldsymbol{f}^* - \boldsymbol{f}^{(t)}$ *satisfies*

$$\frac{\widetilde{\boldsymbol{g}}^\top(\boldsymbol{f}^* - \boldsymbol{f}^{(t)})}{100m + \left\|\widetilde{\boldsymbol{L}}(\boldsymbol{f}^* - \boldsymbol{f}^{(t)})\right\|_1} \leq -\alpha/4.$$

(2) *At each iteration, given any* Δ *satisfying* $B^\top \Delta = 0$ *and* $\widetilde{g}^\top \Delta / \left\|\widetilde{L}\Delta\right\|_1 \le -\kappa$, *it updates* $f^{(t+1)} \leftarrow f^{(t)} + \eta\Delta$ *for* $\eta \leftarrow \kappa^2/(50 \cdot |\widetilde{g}\Delta|)$.

(3) *At the end of* $\widetilde{O}(m\kappa^2)$ *iterations, we have* $c^\top f^{(t)} \le c^\top f^* + (mU)^{-10}$.

Intuitively, the algorithm will compute a sequence of flows $f^{(0)}, f^{(1)}$, and maintain approximations $\widetilde{g}, \widetilde{\ell}$ of $g(f^{(t)}), \ell(f^{(t)})$ respectively. Each iteration, the algorithm will call an oracle for approximating the minimum-ratio cycle, i.e. $\min_{B^\top\Delta=0} \widetilde{g}^\top\Delta/\left\|\widetilde{L}\Delta\right\|_1$. The first item shows that the optimal ratio is at most $-\alpha/4$. Thus if the oracle returns an $m^{o(1)}$ approximation, the returned circulation has $\kappa \ge m^{-o(1)}$. Scaling Δ appropriately and adding it to $f^{(t)}$ decreases the potential by $\Omega(\kappa^2)$, hence the potential drops to $-O(m \log m)$ within $\widetilde{O}(m\kappa^{-2})$ iterations.

In Section 11 we will give a formal description of the interaction of the algorithm of Theorem 6.3 and our data structures to implement each step in amortized $m^{o(1)}$ time. As part of this, we argue that we can change $\widetilde{g}$ and $\widetilde{\ell}$ only $\widetilde{O}(m\kappa^{-2})$ total times. This is encapsulated in Lemma 11.4.

6.1 One Step Analysis

Consider a current flow f and lengths/gradients $\ell(f), g(f)$ defined in Definition 6.2, with $L = \mathrm{diag}(\ell)$. The problem we will solve approximately in each iteration will be

$$\min_{B^\top\Delta=0} \frac{g(f)^\top\Delta}{\|L(f)\Delta\|_1}. \tag{12}$$

Alternatively, this can be viewed as constraining $B^\top\Delta = 0$ and $g(f)^\top\Delta = -1$, and then minimizing $\|L(f)\Delta\|_1$. Our first goal is to show that an approximate solution to (12) for approximations of the gradient and lengths allows us to decrease the potential.

Lemma 6.4. *Let* $\widetilde{g} \in \mathbb{R}^E$ *satisfy* $\left\|L(f)^{-1}(\widetilde{g} - g(f))\right\|_\infty \le \kappa/8$ *for some* $\kappa \in (0,1)$, *and* $\widetilde{\ell} \in \mathbb{R}^E_{>0}$ *satisfy* $\widetilde{\ell} \approx_2 \ell(f)$. *Let* Δ *satisfy* $B^\top\Delta = 0$ *and* $\widetilde{g}^\top\Delta/\left\|\widetilde{L}\Delta\right\|_1 \le -\kappa$. *Let* η *satisfy* $\eta\widetilde{g}^\top\Delta = -\kappa^2/50$. *Then*

$$\Phi(f + \eta\Delta) \le \Phi(f) - \frac{\kappa^2}{500}.$$

Before showing this, we need simple bounds on the Taylor expansion of the logarithmic barrier and $x^{-\alpha}$ in the region where the second derivative is stable.

Lemma 6.5 (Taylor expansion for $x^{-\alpha}$). *If* $|\Delta_e| \le \frac{1}{10}\min\left(u_e^+ - f_e, f_e - u_e^-\right)$ *for* $e \in E$ *then*

$$\begin{aligned}&\left((u_e^+ - f_e - \Delta_e)^{-\alpha} + (f_e + \Delta_e - u_e^-)^{-\alpha}\right) \le \left((u_e^+ - f_e)^{-\alpha} + (f_e - u_e^-)^{-\alpha}\right)\\ &+ \alpha\left(\left(u_e^+ - f_e\right)^{-1-\alpha} - \left(f_e - u_e^-\right)^{-1-\alpha}\right)\Delta_e + \alpha\left(\left(u_e^+ - f_e\right)^{-2-\alpha} + \left(f_e - u_e^-\right)^{-2-\alpha}\right)\Delta_e^2\end{aligned} \tag{13}$$

Also we have that

$$\left|(u_e^+ - f_e - \Delta_e)^{-1-\alpha} - (u_e^+ - f_e)^{-1-\alpha}\right| \le 2|\Delta_e|(u_e^+ - f_e)^{-2-\alpha} \tag{14}$$

and

$$\left|(f_e + \Delta_e - u_e^-)^{-1-\alpha} - (f_e - u_e^-)^{-1-\alpha}\right| \le 2|\Delta_e|(f_e - u_e^-)^{-2-\alpha}. \tag{15}$$

(14) and (15) are useful for analyzing how a step improves the value of the potential function $\Phi(f)$, as well as showing that the gradients $g(f)$ and lengths $\ell(f)$ are stable, i.e. change only $m^{1+o(1)}$ times over $m^{1+o(1)}$ iterations.

Proof. Define $\phi(x) \stackrel{\text{def}}{=} (\boldsymbol{u}_e^+ - x)^{-\alpha} + (x - \boldsymbol{u}_e^-)^{-\alpha}$. ϕ is a convex function with derivative

$$\phi'(x) = \alpha\left((\boldsymbol{u}_e^+ - x)^{-1-\alpha} - (x - \boldsymbol{u}_e^-)^{-1-\alpha}\right)$$

and second derivative

$$\phi''(x) = \alpha(1+\alpha)\left((\boldsymbol{u}_e^+ - x)^{-2-\alpha} + (x - \boldsymbol{u}_e^-)^{-2-\alpha}\right).$$

In particular note that $\phi''(\boldsymbol{f}_e + \delta) \approx_{1.3} \phi''(\boldsymbol{f}_e)$ for any $|\delta| \le \frac{1}{10}\min\left(\boldsymbol{u}_e^+ - \boldsymbol{f}_e, \boldsymbol{f}_e - \boldsymbol{u}_e^-\right)$, because $1.1^{2+\alpha} \le 1.3$ by the choice of α. Thus by Taylor's theorem we get that

$$\phi(\boldsymbol{f}_e + \Delta_e) \le \phi(\boldsymbol{f}_e) + \phi'(\boldsymbol{f}_e)\Delta_e + \frac{1}{2}\max_{y\in[\boldsymbol{f}_e, \boldsymbol{f}_e+\Delta_e]}\phi''(y)\Delta_e^2 \le \phi(\boldsymbol{f}_e) + \phi'(\boldsymbol{f}_e)\Delta_e + 1.3\phi''(\boldsymbol{f}_e)\Delta_e^2$$
$$\le \phi(\boldsymbol{f}_e) + \phi'(\boldsymbol{f}_e)\Delta_e + \alpha\left(\left(\boldsymbol{u}_e^+ - \boldsymbol{f}_e\right)^{-2-\alpha} + \left(\boldsymbol{f}_e - \boldsymbol{u}_e^-\right)^{-2-\alpha}\right)\Delta_e^2,$$

which when expanded yields the desired bound. (14), (15) follow from a similar application of Taylor's theorem on a first order expansion. □

Lemma 6.6 (Taylor expansion for $\log x$). *If $|y| \le x/10$ for $x > 0$ then*

$$\log(x+y) \le \log(x) + y/x + y^2/x^2. \tag{16}$$

Proof. This is equivalent to $\log(1 + y/x) \le y/x + y^2/x^2$ for $|y/x| \le 1/10$, which follows from the Taylor expansion $\log(1+z) = \sum_{k\ge 0} z^k/k$ for $|z| < 1$. □

Proof of Lemma 6.4. We first bound $\boldsymbol{g}(\boldsymbol{f})^\top\Delta$ by

$$\left|\boldsymbol{g}(\boldsymbol{f})^\top\Delta - \widetilde{\boldsymbol{g}}^\top\Delta\right| \stackrel{(i)}{\le} \left\|\boldsymbol{L}(\boldsymbol{f})^{-1}(\widetilde{\boldsymbol{g}} - \boldsymbol{g}(\boldsymbol{f}))\right\|_\infty \|\boldsymbol{L}(\boldsymbol{f})\Delta\|_1 \stackrel{(ii)}{\le} 2\varepsilon/\kappa \cdot |\widetilde{\boldsymbol{g}}^\top\Delta| \le \widetilde{\boldsymbol{g}}^\top\Delta/2,$$

where (i) follows from Hölder's inequality with the ℓ_1, ℓ_∞ norms, (ii) follows from the lemma hypotheses and $\|\boldsymbol{L}(\boldsymbol{f})\Delta\|_1 \le 2\|\widetilde{\boldsymbol{L}}\Delta\|_1$, and the final inequality follows from $\varepsilon \le \kappa/8$. **[Sushant: There seems to be no ε in the statement of the lemma.]** Hence

$$2\widetilde{\boldsymbol{g}}^\top\Delta \le \boldsymbol{g}(\boldsymbol{f})^\top\Delta \le \widetilde{\boldsymbol{g}}^\top\Delta/2. \tag{17}$$

We can also bound $\boldsymbol{c}^\top\Delta$ by

$$20m(\boldsymbol{c}^\top\boldsymbol{f} - F^*)^{-1}\left|\boldsymbol{c}^\top\Delta\right| = \left|\boldsymbol{g}(\boldsymbol{f})^\top\Delta - \alpha\sum_{e\in E}\left(\left(\boldsymbol{u}_e^+ - \boldsymbol{f}_e\right)^{-1-\alpha} - \left(\boldsymbol{f}_e - \boldsymbol{u}_e^-\right)^{-1-\alpha}\right)\Delta_e\right| \tag{18}$$
$$\stackrel{(i)}{\le} \left|\boldsymbol{g}(\boldsymbol{f})^\top\Delta\right| + \alpha\|\boldsymbol{L}(\boldsymbol{f})\Delta\|_1 \stackrel{(ii)}{\le} (2 + 2\alpha/\kappa)|\widetilde{\boldsymbol{g}}^\top\Delta|. \tag{19}$$

where (i) follows from the triangle inequality, and (ii) from (17) plus the problem hypotheses. In particular, we deduce that

$$\left|\boldsymbol{c}^\top\Delta\right| \le \frac{\boldsymbol{c}^\top\boldsymbol{f} - F^*}{20m} \cdot (2 + 2\alpha/\kappa)|\widetilde{\boldsymbol{g}}^\top\Delta|. \tag{20}$$

Let $\overline{\Delta} \stackrel{\text{def}}{=} \eta\Delta$, the circulation that we add. From (20) we get that

$$|\boldsymbol{c}^\top\overline{\Delta}| \le \eta \cdot \frac{\boldsymbol{c}^\top\boldsymbol{f} - F^*}{20m} \cdot (2 + 2\alpha/\kappa)|\widetilde{\boldsymbol{g}}^\top\Delta| \le \frac{\boldsymbol{c}^\top\boldsymbol{f} - F^*}{20m} \cdot 4\eta/\kappa|\widetilde{\boldsymbol{g}}^\top\Delta| \le \frac{\kappa}{500m}\left(\boldsymbol{c}^\top\boldsymbol{f} - F^*\right)$$

by the choice of η in the problem hypothesis. Additionally, we have

$$\left\|\boldsymbol{L}(\boldsymbol{f})\overline{\Delta}\right\|_1 \le 2\left\|\widetilde{\boldsymbol{L}}\overline{\Delta}\right\|_1 \le 2/\kappa \cdot \left\|\widetilde{\boldsymbol{g}}\overline{\Delta}\right\|_1 \le 2\eta/\kappa \cdot \|\widetilde{\boldsymbol{g}}\Delta\|_1 \le \kappa/25 \tag{21}$$

by the choice of η. This implies that

$$|\overline{\Delta}_e| \le \kappa/25 \cdot (\boldsymbol{u}_e^+ - \boldsymbol{f}_e)^{1+\alpha} \le \kappa/25 \cdot (2U)^{\alpha} \cdot (\boldsymbol{u}_e^+ - \boldsymbol{f}_e) \le \kappa/10 \cdot (\boldsymbol{u}_e^+ - \boldsymbol{f}_e),$$

where the last inequality follows from the choice $\alpha = 1/(1000 \log mU)$. $|\overline{\Delta}_e| \le \kappa/10 \cdot (\boldsymbol{f}_e - \boldsymbol{u}_e^-)$ follows similarly.

This bound allows us to apply:

- Lemma 6.5 on the current $\boldsymbol{u}^+$, $\boldsymbol{u}^-$, $\boldsymbol{f}$, and Δ, and
- Lemma 6.6 for $x = \boldsymbol{c}^\top \boldsymbol{f} - F^*$ and $y = \boldsymbol{c}^\top \overline{\Delta}$

to get

$$\begin{aligned}\Phi(\boldsymbol{f}+\overline{\Delta}) - \Phi(\boldsymbol{f}) &\le \boldsymbol{g}(\boldsymbol{f})^\top \overline{\Delta} + 20m\left(\frac{\kappa}{500m}\right)^2 + \sum_{e\in E} \alpha\left(\left(\boldsymbol{u}_e^+ - \boldsymbol{f}_e\right)^{-2-\alpha} + \left(\boldsymbol{f}_e - \boldsymbol{u}_e^-\right)^{-2-\alpha}\right)\overline{\Delta}_e^2 \\ &\overset{(i)}{\le} \widetilde{\boldsymbol{g}}^\top \overline{\Delta}/2 + \frac{\kappa^2}{10000} + \alpha\kappa/10 \cdot \left\|\boldsymbol{L}(\boldsymbol{f})\overline{\Delta}\right\|_1 \overset{(ii)}{\le} -\frac{\kappa^2}{100} + \frac{\kappa^2}{10000} + \frac{\alpha\kappa}{250} \le -\frac{\kappa^2}{500}.\end{aligned}$$

Here, (i) follows from (17), and the bound $|\overline{\Delta}_e| \le \kappa/10 \cdot \min\left(\boldsymbol{u}_e^+ - \boldsymbol{f}_e, \boldsymbol{f}_e - \boldsymbol{u}_e^-\right)$, and (ii) from $\widetilde{\boldsymbol{g}}^\top \overline{\Delta} = \eta \widetilde{\boldsymbol{g}}^\top \Delta = -\kappa^2/50$ and (21). □

Our next goal is to show that a straight line to $\boldsymbol{f}^*$, i.e. $\Delta = \boldsymbol{f}^* - \boldsymbol{f}$ satisfies the guarantees of Lemma 6.4 for some $\kappa \ge \widetilde{\Omega}(1)$. This has two purposes. First, it shows that an $m^{o(1)}$-optimal solution to (12) allows us to decrease the potential by $m^{-o(1)}$ per step, so that the algorithm terminates in $m^{1+o(1)}$ steps. Second, it shows that the problems (12) encountered during the method are not fully adaptive, and we are able to use this guarantee on a good solution to inform our data structures.

Lemma 6.7 (Quality of $\boldsymbol{f}^* - \boldsymbol{f}$). *Let $\widetilde{\boldsymbol{g}} \in \mathbb{R}^E$ satisfy $\left\|\boldsymbol{L}(\boldsymbol{g})^{-1}\left(\widetilde{\boldsymbol{g}} - \boldsymbol{g}(\boldsymbol{f})\right)\right\|_\infty \le \varepsilon$ for some $\varepsilon < \alpha/2$, and $\widetilde{\boldsymbol{\ell}} \in \mathbb{R}^E_{>0}$ satisfy $\widetilde{\boldsymbol{\ell}} \approx_2 \boldsymbol{\ell}(\boldsymbol{f})$. If $\Phi(\boldsymbol{f}) \le 200m \log mU$ and $\log(\boldsymbol{c}^\top \boldsymbol{f} - F^*) \ge -10 \log mU$, then*

$$\frac{\widetilde{\boldsymbol{g}}^\top(\boldsymbol{f}^* - \boldsymbol{f})}{100m + \left\|\widetilde{\boldsymbol{L}}(\boldsymbol{f}^* - \boldsymbol{f})\right\|_1} \le -\alpha/4.$$

The additional $100m$ in the denominator is for a technical reason, and intuitively says that the bound is still fine even if we force every edge to pay at least 100 towards the ℓ_1 length of the circulation.

Proof. We can bound $\boldsymbol{g}(f)^\top(f^*-f)$ using

$$\begin{aligned}
\boldsymbol{g}(f)^\top(f^*-f) &= 20m\frac{\boldsymbol{c}^\top(f^*-f)}{\boldsymbol{c}^\top f - F^*} + \sum_{e\in E}\left(\alpha\left(\boldsymbol{u}_e^+ - f_e\right)^{-1-\alpha} - \alpha\left(f_e - \boldsymbol{u}_e^-\right)^{-1-\alpha}\right)(f_e^*-f_e)\\
&\overset{(i)}{\le} -20m - \alpha\sum_{e\in E}\left(\left(\boldsymbol{u}_e^+ - f_e\right)^{-1-\alpha} + \left(f_e-\boldsymbol{u}_e^-\right)^{-1-\alpha}\right)|f_e^*-f_e|\\
&\quad + 2\alpha\sum_{e\in E}\left(\left(\boldsymbol{u}_e^+ - f_e\right)^{-\alpha} + \left(f_e-\boldsymbol{u}_e^-\right)^{-\alpha}\right)\\
&= -20m - \alpha\left\|\boldsymbol{L}(f)(f^*-f)\right\|_1 + 2\alpha\left(\Phi(f) - 20m\log(\boldsymbol{c}^\top f - F^*)\right)\\
&\overset{(ii)}{\le} -20m - \alpha\left\|\boldsymbol{L}(f)(f^*-f)\right\|_1 + 2\alpha\cdot 400m\log mU\\
&\overset{(iii)}{\le} -19m - \alpha\left\|\boldsymbol{L}(f)(f^*-f)\right\|_1.
\end{aligned}$$

where (i) follows from the bound $\boldsymbol{u}_e^- - f_e \le f_e^* - f_e \le \boldsymbol{u}_e^+ - f_e$ for all $e\in E$, so

$$\begin{aligned}
(\boldsymbol{u}_e^+ - f_e)^{-1-\alpha}(f_e^*-f_e) &= (\boldsymbol{u}_e^+ - f_e)^{-1-\alpha}(f_e^* - \boldsymbol{u}_e^+ + \boldsymbol{u}_e^+ - f_e)\\
&= (\boldsymbol{u}_e^+ - f_e)^{-\alpha} - (\boldsymbol{u}_e^+ - f_e)^{-1-\alpha}|f_e^* - \boldsymbol{u}_e^+|\\
&\le 2(\boldsymbol{u}_e^+ - f_e)^{-\alpha} - (\boldsymbol{u}_e^+ - f_e)^{-1-\alpha}\left|f_e^* - f_e\right|,
\end{aligned}$$

and similar for the $-(\boldsymbol{u}_e^+ - f_e)^{-1-\alpha}(f_e^*-f_e)$ term, (ii) follows from the lemma hypotheses, and (iii) from the choice $\alpha = 1/(1000\log mU)$. We now bound

$$\begin{aligned}
\widetilde{\boldsymbol{g}}^\top(f^*-f) &= \boldsymbol{g}(f)^\top(f^*-\boldsymbol{g}) + (\widetilde{\boldsymbol{g}}-\boldsymbol{g}(f))^\top(f^*-f)\\
&\overset{(i)}{\le} -19m - \alpha\left\|\boldsymbol{L}(f)(f^*-f)\right\|_1 + \left\|\boldsymbol{L}(f)^{-1}(\widetilde{\boldsymbol{g}}-\boldsymbol{g}(f))\right\|_\infty\left\|\boldsymbol{L}(f)(f^*-f)\right\|_1\\
&\overset{(ii)}{\le} -19m - \alpha\left\|\boldsymbol{L}(f)(f^*-f)\right\|_1 + \varepsilon\left\|\boldsymbol{L}(f)(f^*-f)\right\|_1\\
&\le -19m - \alpha/2\cdot\left\|\boldsymbol{L}(f)(f^*-f)\right\|_1,
\end{aligned}$$

where (i) follows from the above bound on $\boldsymbol{g}(f)^\top(f^*-f)$ and Hölder for the ℓ_1/ℓ_∞ norms, and (ii) follows from the the conditions on $\widetilde{\boldsymbol{g}}$. Thus we get that

$$\frac{\widetilde{\boldsymbol{g}}^\top(f^*-f)}{100m + \left\|\widetilde{\boldsymbol{L}}(f^*-f)\right\|_1} \le \frac{-19m - \alpha/2\cdot\|\boldsymbol{L}(f)(f^*-f)\|_1}{100m + 2\|\boldsymbol{L}(f)(f^*-f)\|_1} \le -\alpha/4,$$

where we have used the above bound on $\widetilde{\boldsymbol{g}}^\top(f^*-f)$ and $\boldsymbol{\ell}\approx_2\widetilde{\boldsymbol{\ell}}$. □

6.2 Stability Bounds

Our algorithm ultimately approximately solves (12) by using approximations $\widetilde{\boldsymbol{g}}$ of $\boldsymbol{g}(f)$ and $\widetilde{\boldsymbol{\ell}}$ of $\boldsymbol{\ell}(f)$ satisfying the conditions of Lemma 6.4. The goal of this section is to show that $\boldsymbol{g}(f)$ and $\boldsymbol{\ell}(f)$ are slowly changing relative to the lengths, so that our dynamic data structure can only update their values on $m^{o(1)}$ edges per iteration.

We start by showing that the residual cost is very slowly changing, by about $1/m$ per iteration.

Lemma 6.8 (Residual stability). *Let $\widetilde{g} \in \mathbb{R}^E$ satisfy $\left\|\boldsymbol{L}(f)^{-1}(\widetilde{g} - g(f))\right\|_\infty \le \varepsilon$ for some $\varepsilon \in (0, 1/2]$, and $\widetilde{\ell} \in \mathbb{R}^E_{>0}$ satisfy $\widetilde{\ell} \approx_2 \ell(f)$. Let Δ satisfy $B^\top \Delta = 0$ and $\widetilde{g}^\top \Delta / \left\|\widetilde{L}\Delta\right\|_1 \le -\kappa$ for $\kappa \in (0, 1)$. Then*

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

Proof. We can write

$$\begin{aligned}\frac{|c^\top \Delta|}{c^\top f - F^*} &\overset{(i)}{\le} \frac{1}{20m}(|g^\top \Delta| + \alpha\|L(f)\Delta\|_1) \\ &\overset{(ii)}{\le} \frac{1}{20m}(|\widetilde{g}^\top \Delta| + \|L(f)^{-1}(\widetilde{g} - g(f))\|_\infty \|L(f)\Delta\|_1 + \alpha\|L(f)\Delta\|_1) \\ &\overset{(iii)}{\le} (|\widetilde{g}^\top \Delta| + 2\|\widetilde{L}\Delta\|_1)/(20m) \le |\widetilde{g}^\top \Delta|/(\kappa m),\end{aligned}$$

where (i) uses the triangle inequality, (ii) uses the triangle inequality and $|x^\top y| \le \|x\|_\infty \|y\|_1$, and (iii) uses the hypotheses and $\widetilde{\ell} \approx_2 \ell(f)$. **[Sushant: We seem to have dropped the middle term? Also, how did we get κ in the demoninator? The last line is misssing some details.]** □

Hence if $|g(f)^\top \Delta| \le O(\kappa^2)$ and $\|L(f)\Delta\|_1 = O(\kappa)$ such as in the hypotheses of Lemma 6.4, the residual cost changes by at most a $1/m$ factor per iteration.

We show that if the residual capacity of an edge does not change much, then its length is stable.

Lemma 6.9 (Length stability). *If $\|L(f)(f - \overline{f})\|_\infty \le \varepsilon$ for some $\varepsilon \le 1/100$ then $\ell(f) \approx_{1+3\varepsilon} \ell(\overline{f})$.*

Proof. Because $\left\|L(f)(f - \overline{f})\right\|_\infty \le 1/100$, we have for all $e \in E$

$$|f_e - \overline{f}_e| \le \varepsilon(u_e^+ - f_e)^{1+\alpha} = \varepsilon(u_e^+ - f_e)(2U)^\alpha \le 2\varepsilon(u_e^+ - f_e).$$

Similarly, $|f_e - \overline{f}_e| \le 2\varepsilon(f_e - u_e^-)$. Hence $u_e^+ - f_e \approx_{\exp(2\varepsilon)} u_e^+ - \overline{f}_e$ and $f_e - u_e^- \approx_{\exp(2\varepsilon)} \overline{f}_e - u_e^-$, so we get

$$\begin{aligned}\ell(\overline{f})_e &= (u_e^+ - \overline{f}_e)^{-1-\alpha} + (\overline{f}_e - u_e^-)^{-1-\alpha} \\ &\approx_{\exp(2\varepsilon(1+\alpha))} (u_e^+ - f_e)^{-1-\alpha} + (f_e - u_e^-)^{-1-\alpha} = \ell(f)_e.\end{aligned}$$

This completes the proof, as $2\varepsilon(1 + \alpha) \le 3\varepsilon$. □

Next we show a similar stability claim for gradients. Here, we scale by the residual cost $c^\top f - F^*$ to ensure that the leading term is $20mc$. Thus, the gradient is stable if the residual capacity of an edge does not change much, and if the residual cost is stable. We know that the residual cost is stable over $\widetilde{O}(m)$ iterations by Lemma 6.8.

Lemma 6.10 (Gradient stability). *If $\|L(f)(f - \overline{f})\|_\infty \le \varepsilon$ and $r \approx_{1+\varepsilon} c^\top \overline{f} - F^*$ then $\widetilde{g}$ defined as*

$$\widetilde{g}_e = 20mc_e/r + \alpha(u_e^+ - f_e)^{-1-\alpha} - \alpha(f_e - u_e^-)^{-1-\alpha} \text{ for all } e \in E$$

satisfies

$$\left\|L(f)^{-1}\left(\widetilde{g} - (c^\top \overline{f} - F^*)/r \cdot g(\overline{f})\right)\right\| \le 6\alpha\varepsilon.$$

Proof. We can first compute that

$$\left[\widetilde{g} - (c^\top \overline{f} - F^*)/r \cdot g(\overline{f})\right]_e$$

$$= \left(1 - \frac{c^\top \overline{f}}{r}\right)\left(\alpha(u_e^+ - \overline{f}_e)^{-1-\alpha} - \alpha(\overline{f}_e - u_e^-)^{-1-\alpha}\right) \tag{22}$$

$$+ \alpha((u_e^+ - f_e)^{-1-\alpha} - (u_e^+ - \overline{f}_e)^{-1-\alpha}) - \alpha((f_e - u_e^-)^{-1-\alpha} - (\overline{f}_e - u_e^-)^{-1-\alpha}). \tag{23}$$

We bound the terms in (22) and (23) separately. To bound (22) we write

$$\left(1 - \frac{c^\top \overline{f}}{r}\right)\alpha \left\| L(f)^{-1}\left((u^+ - \overline{f})^{-1-\alpha} - (\overline{f} - u^-)^{-1-\alpha}\right)\right\|_\infty$$

$$= \left(1 - \frac{c^\top \overline{f}}{r}\right)\alpha \left\| L(f)^{-1}\ell(\overline{f})\right\|_\infty \overset{(i)}{\leq} 1.1\varepsilon\alpha \cdot (1+3\varepsilon) \leq 2\varepsilon\alpha,$$

where (i) follows from the hypothesis and Lemma 6.9. To bound (23) we write

$$\alpha \left\| L(f)^{-1}\left(((u^+ - f)^{-1-\alpha} - (u^+ - \overline{f})^{-1-\alpha}) - ((f - u^-)^{-1-\alpha} - (\overline{f} - u^-)^{-1-\alpha})\right)\right\|_\infty$$

$$\overset{(i)}{\leq} 2\alpha \left\| L(f)^{-1}((u^+ - f)^{-2-\alpha} + (f - u^-)^{-2-\alpha})(f - \overline{f})\right\|_\infty$$

$$\leq 2\alpha \max_{e \in E}\{(u_e^+ - f_e)^\alpha, (f_e - u_e^-)^\alpha\}\|L(f)(f - \overline{f})\|_\infty \leq 2\alpha(2U)^\alpha\varepsilon \leq 4\alpha\varepsilon$$

where (i) follows from Lemma 6.5, specifically (14), (15). Summing these gives the desired bound. □

We now show the main result of this section, Theorem 6.3.

Proof of Theorem 6.3. The first item follows from Lemma 6.7. To show the third item, note that the update in the second item exactly corresponds to Lemma 6.4, so $\Phi(f^{(t)}) \leq \Phi(f^{(t-1)}) - \Omega(\kappa^2)$. Once the potential has reduced to $-O(m \log m)$, then $c^\top f^{(t)} - c^\top f \leq (mU)^{-10}$ (Lemma 6.1), so the algorithm takes $\widetilde{O}(m\kappa^{-2})$ total iterations. □

6.3 Initial and Final Point

In this section we discuss how to initialize our method and how to get an exact optimal solution from a nearly optimal solution. For the latter piece, we can directly cite previous work which gives a rounding method using the Isolation Lemma.

Lemma 6.11 ([19, Lemma 8.10]). *Consider a min-cost flow instance $\mathcal{I} = (G, d, c)$ on a graph $G = (V, E)$ with demands $d \in \{-U, \ldots, U\}^V$ and cost $c \in \{-U, \ldots, U\}^E$. Assume that all optimal flows have congestion at most U on every edge.*

Consider a perturbed instance $\widetilde{\mathcal{I}} = (G, d, \widetilde{c})$ on the same graph $G = (V, E)$ and demand d, but with modified cost vector $\widetilde{c} \in \mathbb{R}^E$ defined as $\widetilde{c}_e = c_e + z_e$ for independent, random $z_e \in \left\{\frac{1}{4m^2U^2}, \frac{2}{4m^2U^2}, \ldots, \frac{2mU}{4m^2U^2}\right\}$ for all $e \in E$. Let $\widetilde{f}$ be a solution for $\widetilde{\mathcal{I}}$ whose cost is at most $\frac{1}{12m^3U^3}$ from optimal. Let f be obtained by rounding $\widetilde{f}$ to the nearest integer on every edge. Then f is an optimal flow for the instance $\mathcal{I}$ with probability at least $1/2$.

It is worth noting that scaling up the cost vector $\widetilde{c}$ of the perturbed instance $\widetilde{\mathcal{I}}$ in Lemma 6.11 by $4m^2U^2$ results in a min-cost flow instance with integral demands and costs again.

Now we describe how to augment our original graph with additional edges without affecting the optimal solution, but allows us to initialize a solution with bounded potential. The proof is deferred to Appendix A.1.

Lemma 6.12 (Initial Point). *There is an algorithm that given a graph $G = (V, E)$ and min-cost flow instance $\mathcal{I} = (G, \boldsymbol{d}, \boldsymbol{c}, \boldsymbol{u}^+, \boldsymbol{u}^-)$ with demands $\boldsymbol{d} \in \{-U, \ldots, U\}^V$, and costs and lower/upper capacities $\boldsymbol{c}, \boldsymbol{u}^-, \boldsymbol{u}^+ \in \{-U, \ldots, U\}^V$, constructs a min-cost flow instance $\widetilde{\mathcal{I}} = (\widetilde{G}, \widetilde{\boldsymbol{d}}, \widetilde{\boldsymbol{c}}, \widetilde{\boldsymbol{u}}^+, \widetilde{\boldsymbol{u}}^-)$ with $O(m)$ edges and $\widetilde{\boldsymbol{d}} \in \{-2mU, \ldots, 2mU\}^{V(\widetilde{G})}$, $\widetilde{\boldsymbol{c}}, \widetilde{\boldsymbol{u}}^+, \widetilde{\boldsymbol{u}}^- \in \{-4mU^2, \ldots, 4mU^2\}^{E(\widetilde{G})}$, and a flow $\boldsymbol{f}^{(\text{init})}$ on $\widetilde{G}$ routing $\widetilde{\boldsymbol{d}}$ such that $\Phi(\boldsymbol{f}^{(\text{init})}) \leq 200m \log mU$.*

Also, given an optimal flow $\widetilde{\boldsymbol{f}}$ for $\widetilde{\mathcal{I}}$, the algorithm can either compute an optimal flow $\boldsymbol{f}$ for $\mathcal{I}$ or conclude that $\mathcal{I}$ admits no feasible flow. The algorithm runs in time $O(m)$.

7 DECREMENTAL SPANNER AND EMBEDDING

The main result of this section is summarized in the following theorem. Intuitively, the theorem states that given a low-degree graph G, one can maintain a sparsifier H of G and embed G with short paths and low congestion into H.

Theorem 7.1. *Given an m-edge n-vertex unweighted, undirected, dynamic graph G undergoing update batches $U^{(1)}, U^{(2)}, \ldots$ consisting only of edge deletions and $\widetilde{O}(n)$ vertex splits. There is a randomized algorithm with parameter $1 \leq L \leq o\left(\frac{\sqrt{\log(m)}}{\log\log m}\right)$, that maintains a spanner H and an embedding $\Pi_{G \to H}$ such that*

1. *Sparsity and Low Recourse: initially $H^{(0)}$ has sparsity $\widetilde{O}(n)$. At any stage $t \geq 1$, the algorithm outputs a batch of updates $U_H^{(t-1)}$ that when applied to $H^{(t-1)}$ produce $H^{(t)}$ such that $H^{(t)} \subseteq G^{(t)}$, $H^{(t)}$ consists of at most $\widetilde{O}(n)$ edges and $\sum_{t' \leq t} \textsc{Enc}(U_H^{(t')}) = \widetilde{O}\left(n + \sum_{t' \leq t} |U^{(t')}| \cdot n^{1/L}\right)$, and*
2. *Low Congestion, Short Paths Embedding: $\text{length}(\Pi_{G \to H}) \leq (\gamma_l)^{O(L)}$ and $\text{vcong}(\Pi_{G \to H}) \leq (\gamma_c)^{O(L)} \Delta_{\max}(G)$, for $\gamma_l, \gamma_c = \exp(O(\sqrt{\log m} \cdot \log\log m))$, and*
3. *Low Recourse Re-Embedding: the algorithm further reports after each update batch $U^{(t)}$ at stage t is processed, a (small) set $D^{(t)} \subseteq E(H^{(t)})$ of edges, such that for all other edges $e \in E(H^{(t)}) \setminus D^{(t)}$, there exists **no** edge $e' \in E(G^{(t)})$ whose embedding path $\Pi^{(t)}_{G \to H}(e')$ contains e at the current stage but did not before the stage. The algorithm ensures that at any stage t, we have $\sum_{t' \leq t} |D^{(t')}| = \widetilde{O}\left(\sum_{t' \leq t} |U^{(t)}| \cdot n^{1/L} (\gamma_c \gamma_l)^{O(L)}\right)$, i.e. that the sets $D^{(t)}$ are roughly upper bounded by the size of $U^{(t)}$ on average.*

The algorithm takes initialization time $\widetilde{O}(m\gamma_l)$ and processing the t-th update batch $U^{(t)}$ takes amortized update time $\widetilde{O}(\textsc{Enc}(U^{(t)}) \cdot n^{1/L} (\gamma_c \gamma_l)^{O(L)} \Delta_{\max}(G))$, and succeeds with probability at least $1 - n^{-C}$ for any constant $C > 0$, specified before the procedure is invoked.

Taking $L = (\log m)^{1/4}$ in Theorem 7.1 gives a parameter $\gamma_s = \exp(O(\log^{3/4} m \log\log m))$ such that the amortized runtime, lengths of the embeddings, and amortized size of D are all $O(\gamma_s)$. We emphasize that the guarantees 1 and 3 are with respect to the number of updates in each batch $U^{(t)}$ and *not* with respect to the (possibly much larger) encoding size of $U^{(t)}$. This is of utmost importance for our application.

In this section, we will prove Theorem 7.1 under the assumption that the update sequence is bounded by $n - 1$ and that each update batch consists only of a single update. This is without loss of generality as one can restart the algorithm every n updates without affecting any of the bounds.

7.1 The Algorithm

Data Structures. Our algorithm to implement Theorem 7.1 works with a rather standard batching approach with $L + 1$ levels. The algorithm therefore maintains graphs $H_0, H_1, \ldots, H_L$ which form the sparsifier $H \stackrel{\text{def}}{=} \bigcup H_j$ implicitly. The algorithm recomputes each graph H_j every so often, with shallower levels being recomputed less often than deeper

levels. However, each H_j undergoes frequent changes since after each stage we apply the updates to G to each graph H_j (if applicable) such that the graphs remain subgraphs of G with the same vertex set at any stage.

It further maintains embeddings $\Pi_0, \Pi_1, \ldots, \Pi_L$ where each embedding Π_j maps a subset of $E(G)$ into the graph $H_{\leq j} \stackrel{\text{def}}{=} \bigcup_{i \leq j} H_i$. In the algorithm, the pre-image of the embeddings $\Pi_0, \Pi_1, \ldots, \Pi_L$ is not disjoint, and in fact, we let Π_0 always have the full set $E(G)$ in its pre-image. We define the embedding $\Pi_{\leq j}$ which to be the embedding that maps each edge $e \in E(G)$ via the embedding Π_i with the largest $i \leq j$ that has e in its pre-image.

Whenever we recompute a graph H_j, we also recompute Π_j such that after recomputation $\Pi_{\leq j}$ embeds the current graph G into the current graph $H_{\leq j}$. As for the graphs H_j, we apply updates to G to the embedding paths in Π_j which means that eventually an edge $e \in E(G)$ with endpoints a, b might not be mapped by $\Pi_j(e)$ to an actual a-b path; either because edges on $\Pi_j(e)$ are deleted, or vertices are split or both. However, the algorithm ensures that most edges are correctly mapped via $\Pi_{\leq j}$ at all times and the small fraction of edges that is not properly dealt with are then dealt with by embeddings $\Pi_{j+1}, \Pi_{j+2}, \ldots, \Pi_L$ on deeper levels. The embedding $\Pi_{G \to H}$ is again maintained implicitly and defined $\Pi_{G \to H} \stackrel{\text{def}}{=} \Pi_{\leq L}$.

Finally, we maintain sets $S_0, S_1, \ldots, S_L$. Each set S_j consists of the vertices that are touched by the updates to G since the last time that H_j was recomputed. We give a formal definition for touched vertices below.

Definition 7.2. *We say that the t-th update to G* touches *a vertex u if the update is an edge deletion and u is one of its endpoints, or if the update is a vertex split and u is one of the resulting vertices from the split.*

Initialization. We start our algorithm by running the sparsification procedure below to initialize H_0 and Π_0.

Theorem 7.3. *Given an unweighted, undirected graph G. There is a procedure $\textsc{Sparsify}(G)$ that produces a sparse subgraph $H_0 \subseteq G$ and an embedding Π_0 of vertex-congestion at most $2\gamma_c \Delta_{\max}(G)$ and length at most γ_l, with high probability. The algorithm takes time $\widetilde{O}(m\gamma_l)$.*

For $j > 0$, we initialize H_j to the empty graph and Π_j to be an empty map. We set all sets S_j (including S_0) to the empty vertex set.

Updates. At each stage $t = 1, 2, \ldots$, we invoke the procedure $\textsc{Update}(t)$ that is given in Algorithm 1. As mentioned earlier, the algorithm works by batching updates made to the graph G. In Line 3, the algorithm determines the current batch level j which in turn determines the batch size that has to be handled at the current stage t. Once j is determined, we also find the last stage t_{j-1} that a level-$(j-1)$ update occurred.

The algorithm then sets all graphs and embeddings of level $j, j+1, \ldots, L$ to the empty graphs/ embeddings in Line 6.

It then forms the graph J which is the key object in this section. This graph is constructed by finding all edges e whose embedding path in $\Pi_{<j}(e)$ was affected by updates since the last recomputations of $\Pi_0, \Pi_1, \ldots, \Pi_{j-1}$ and adds some projection $\widehat{e}$ of e onto the vertex set S_{j-1} to J.

The idea behind this projection is that as S_{j-1} is the set of vertices touched since this stage, we have that an affected edge e has some vertex of S_{j-1} on its path. Assuming for the moment that the edge e itself is not incident to a vertex that was split since the recomputation stage, we essentially project the endpoints a, b of e to the nearest vertices $\widehat{a}, \widehat{b}$ in S_{j-1} on the old embedding path $\Pi_{<j}(e)$. Note in particular that by definition the path $\Pi_{<j}(e)[a, \widehat{a}]$ and $\Pi_{<j}(e)[\widehat{b}, b]$ are then still in G.

We give the following more formal definition that also defines a projection for the slightly more involved case where the edge e is incident to a vertex that splits over time.

Definition 7.4 (Edge-Embedding Projection). *For any $0 < j \leq L$, embedding $\Pi_{<j}$, set S_{j-1} being the set of all vertices touched by updates to G since the last stage that $\Pi_{<j}$ was modified, and edge $e \in E(G)$ such that $\Pi_{<j}(e) \cap S_{j-1} \neq \emptyset$. Then, we let $\text{proj}_{j-1}(e)$ be a new edge $\widehat{e}$ that is associated with e and has endpoints $\widehat{a}$ and $\widehat{b}$ being the closest vertices in S_{j-1} to the endpoints of e in the current graph $\Pi_{<j}(e)$, respectively. Here $\Pi_{<j}(e)$ refers to the graph over the entire vertex set $V(G)$ obtained from adding the edges that are on $\Pi_{<j}(e)$ and then applying the relevant updates that took place on G since $\Pi_{<j}$ was last modified.*

As previously mentioned, we project these edges e whose embedding path $\Pi_{<j}(e)$ was affected by updates onto S_{j-1} to obtain edge $\widehat{e}$ which is added to the graph J. Note that as projected edges are associated with edges e in G, the graph J can be a multigraph.

Algorithm 1: UPDATE(t)

1 Update all sparsifiers $H_0, H_1, \ldots, H_L$ with the t-th update if it applies.;
2 Add the vertices touched by the t-th update to each of the sets $S_0, S_1, \ldots, S_L$.;
3 $j \leftarrow \min\{j' \in \mathbb{Z}_{\geq 0} \mid t \text{ is divisible by } n^{1-j'/L}\}$.; `// Determine j level of stage t.`
4 $t_{j-1} \leftarrow \lfloor t/n^{1-(j-1)/L} \rfloor \cdot n^{1-(j-1)/L}$. ; `// t_{j-1} is the most recent level-(j-1) stage.`
5 **for** $i = j, j+1, \ldots, L$ **do** `// Re-set level-≥ j sparsifiers.`
6 $H_i \leftarrow (V, \emptyset)$; Set Π_i to be an empty map; $S_i \leftarrow \emptyset$.
`// Auxiliary Graph and embedding by projecting embedding paths onto S_{j-1}.`
7 $J \leftarrow (S_{j-1}, \emptyset)$; $\Pi_{J \to H_{<j} \cup E_{affected}} \leftarrow \emptyset$.;
8 $E_{affected} \leftarrow \{e \in E(G) \text{ with } \Pi_{<j}(e) \cap S_{j-1} \neq \emptyset\}$.;
9 **foreach** *edge* $e \in E_{affected}$ **do**
10 $\widehat{e} \leftarrow \text{proj}_{j-1}(e)$.; `// Find Projected Edge of e.`
11 Let a and b be the endpoints of e, and $\widehat{a}$ and $\widehat{b}$ be the endpoints of $\widehat{e}$.;
12 Add $\widehat{e}$ to J.;
13 $\Pi_{J \to H_{<j} \cup E_{affected}}(\widehat{e}) \leftarrow \Pi_{<j}(e)[\widehat{a}, a] \oplus e \oplus \Pi_{<j}(e)[b, \widehat{b}]$.
`// Sparsify Auxiliary Graph and translate back to re-build H.`
14 $(\widetilde{J}, \Pi_{J \to \widetilde{J}}) \leftarrow \text{SPARSIFY}(H_{<j} \cup E_{affected}, J, \Pi_{J \to H_{<j} \cup E_{affected}})$.;
15 **foreach** *edge* $e \in E_{affected}$ *and* $\widehat{e} = \text{proj}_{j-1}(e) \in \widetilde{J}$ **do** Add e to H_j.;
16 **foreach** *edge* $e \in E_{affected}$ **do**
17 $\widehat{e} = \text{proj}_{j-1}(e)$;
18 Let a and b be the endpoints of e, and $\widehat{a}$ and $\widehat{b}$ be the endpoints of $\widehat{e}$.;
19 $\Pi_j(e) \leftarrow \Pi_{<j}(e)[a, \widehat{a}] \oplus [\Pi_{J \to H_{<j} \cup E_{affected}} \circ \Pi_{J \to \widetilde{J}}](\widehat{e}) \oplus \Pi_{<j}(e)[\widehat{b}, b]$.
20 **return** $D = \Pi_{J \to H_{<j} \cup E_{affected}}(\widetilde{J})$.

Along with J, there is also a natural embedding of the projected edge $\widehat{e}$ into the sparsifier $H_{<j}$: we can simply take the path $\Pi_{<j}(e)[\widehat{a}, a] \oplus e \oplus \Pi_{<j}(e)[b, \widehat{b}]$ which we already argued to exist in the current graph $H_{<j}$. This embedding is constructed in Line 13.

Finally, the procedure SPARSIFY is invoked on the graph J. While we have seen this procedure before in the initialization stage, here, we use a generalized version that incorporates the embedding from J into H constructed above. The guarantees of our generalized procedure SPARSIFY are summarized below. Note that letting $J = G$; and letting $\Pi_{J \to G}$ be the identity function, we recover Theorem 7.3 as a corollary.

Theorem 7.5. *Given unweighted, undirected graphs H' and J with $V(J) \subseteq V(H')$ and an embedding $\Pi_{J\to H'}$ from J into H'. Then, there is a randomized algorithm $\textsc{Sparsify}(H', J, \Pi_{J\to H'})$ that returns a sparsifier $\widetilde{J} \subseteq J$ with $|E(\widetilde{J})| = \widetilde{O}(|V(J)|)$ and an embedding $\Pi_{J\to\widetilde{J}}$ from J to $\widetilde{J}$ such that*

(1) $\operatorname{vcong}(\Pi_{J\to H'} \circ \Pi_{J\to\widetilde{J}}) \le \gamma_c \cdot \big(\operatorname{vcong}(\Pi_{J\to H'}) + \Delta_{\max}(J)\big)$, *and*
(2) $\operatorname{length}(\Pi_{J\to H'} \circ \Pi_{J\to\widetilde{J}}) \le \gamma_l \cdot \operatorname{length}(\Pi_{J\to H'})$.

The algorithm runs in time $\widetilde{O}(|E(J)| \cdot \gamma_l)$ and succeeds with probability at least $1 - n^{-C}$ for any constant C, specified before the procedure is invoked.

We defer the proof of the theorem to Section 7.2 and finish the description of Algorithm 1. Given the sparsified graph $\widetilde{J}$ (along with the embedding map), we find the pre-images of the edges in $\widetilde{J}$ and add them to H. We then re-embed all edges (a, b) in G that were no longer properly embedded into H by using the embedding $\Pi_{J\to H} \circ \Pi_{J\to\widetilde{J}}$ to get from $\widehat{a}$ to $\widehat{b}$ and then prepend (append) the path $\Pi_{J\to\widetilde{J}}(e)$ from a to $\widehat{a}$ (from $\widehat{b}$ to b). To gain better intuition for the resulting embedding, we recommend the reader to follow the analysis in Lemma 7.6.

Proof of Theorem 7.1. As a first part of the analysis, we establish that the algorithm indeed maintains an actual embedding.

Lemma 7.6. *For any $0 \le i \le L$, and stage t divisible by $n^{1-i/L}$, $\Pi^{(t)}_{\le i}$ embeds $G^{(t)}$ into $H^{(t)}_{\le i}$. In particular, at any stage t, $\Pi^{(t)}_{G\to H} = \Pi^{(t)}_{\le L}$ embeds $G^{(t)}$ into $H^{(t)}$.*

Proof. We prove by induction on the stage t. For the base case ($t = 0$), we observe that in the initialization phase Π_0 embeds $G^{(0)}$ into H_0 via the algorithm in Theorem 7.3. For all other $j > 0$, Π_j is an empty embedding and therefore, the claim follows.

Let us next consider the inductive step $t - 1 \mapsto t$: We let $j = j^{(t)}$ and $t_{j-1} = t^{(t)}_{j-1}$. Consider any edge $e \in E(G^{(t)})$. We first note that the path $\Pi^{(t_{j-1})}_{<j}(e)$ in $H^{(t_{j-1})}_{<j}$ exists since we can invoke the induction hypothesis by the fact that $t_{j-1} < t$ which follows from the minimality of j (see Line 3). By definition of S_{j-1} being the vertices touched by all updates to G since t_{j-1}, we have that if $\Pi^{(t)}_{<j}(e) \cap S^{(t)}_{j-1} = \emptyset$, that $\Pi^{(t)}_{<j}(e)$ still embeds e properly into $H^{(t)}_{<j}$. Since the foreach-loop in Line 16 is not entered for such edges e, we further have that $\Pi^{(t)}_{\le j}(e) = \Pi^{(t)}_{<j}(e)$.

It remains to analyze the case $\Pi^{(t)}_{<j}(e) \cap S^{(t)}_{j-1} \neq \emptyset$ where we note that $\widehat{e} = \operatorname{proj}_{j-1}(e)$ is well-defined. We start with the observation that $\Pi_{J\to H_{<j}\cup E_{affected}}(\widehat{e})$ restricted to the edges in $\widetilde{J}$ only maps to the edges in $H^{(t)}_{<j}$ and the pre-images of edges $\widehat{e} \in \widetilde{J}$ under the $\operatorname{proj}_{j-1}(\cdot)$ map as can be seen easily from its construction in Line 13. But $H^{(t)}_j$ consists exactly of the pre-images of $\widetilde{J}$ as can be seen from Line 15, so each such embedding path is in $H^{(t)}_{\le j}$. Since $\Pi_{J\to\widetilde{J}}$ maps all edges $\widehat{e}$ in J to paths in $\widetilde{J}$, we thus have $[\Pi_{J\to H_{<j}\cup E_{affected}} \circ \Pi_{J\to\widetilde{J}}](\widehat{e})$ in $H^{(t)}_{\le j}$.

By the way the algorithm sets the new embedding path $\Pi^{(t)}_j(e)$ in Line 19, we thus only have to argue about the path segments $\Pi^{(t)}_{<j}(e)[a, \widehat{a}]$ and $\Pi^{(t)}_{<j}(e)[\widehat{b}, b]$ where a, b are the endpoints of e and $\widehat{a}, \widehat{b}$ are the endpoints of $\widehat{e}$. But again, by definition of $\widehat{e}$ (see Definition 7.4), we have that both of these path segments are contained in $H^{(t)}_{<j}$.

Finally, for all $i > j$, $\Pi^{(t)}_i$ is set to be empty and therefore $\Pi^{(t)}_{\le i} = \Pi^{(t)}_{\le j}$. □

It turns out that the proof of Lemma 7.6 is already the most complicated part of the analysis. We next bound congestion and length of the embedding.

Claim 7.7. *For any $0 \le i \le L$, we have $\operatorname{vcong}(\Pi^{(t)}_{\le i}) \le 4^i \gamma_c^{i+1} \Delta_{\max}(G)$.*

Proof. Again, we prove by induction on stage t. We have for $i = 0$, that $\Pi_0^{(0)}$ as computed in the initialization stage has vertex-congestion at most $\gamma_c \Delta_{\max}(G)$ by Theorem 7.3. For $i > 0$, we have that $\Pi_i^{(0)}$ is empty; therefore its congestion is 0.

For $t - 1 \mapsto t$, we define $j = j^{(t)}$ and $t_{j-1} = t_{j-1}^{(t)}$. Observe that for each edge e considered in the first foreach-loop starting in Line 9, $\Pi_{J \to H_{<j} \cup E_{affected}}(e)$ consists only of the edges in $\Pi_{<j}^{(t)}(e)$ and the edge e itself, it follows that every embedding path that contributes to vertex congestion of a vertex v in $\mathrm{vcong}(\Pi_{J \to H_{<j} \cup E_{affected}})$ also contributes to the vertex congestion of v in $\mathrm{vcong}(\Pi_{<j}^{(t)})$, and hence $\mathrm{vcong}(\Pi_{J \to H_{<j} \cup E_{affected}}) \leq \mathrm{vcong}(\Pi_{<j}^{(t)})$. Further, we can see from the construction of the graph J that $\Delta(J) \leq \mathrm{vcong}(\Pi_{<j}^{(t)})$.

Let us next analyze $\mathrm{vcong}(\Pi_{<j}^{(t)})$. By minimality of j (see Line 3), we have $t_{j-1} < t$ and we can use the induction hypothesis to get $\mathrm{vcong}(\Pi_{<j}^{(t_{j-1})}) \leq 4^{j-1}\gamma_c^j \Delta_{\max}(G)$. It is further immediate to see that since the embedding $\Pi_{<j}$ was not affected by any recomputations since stage t_{j-1} that the vertex congestion can only have dropped ever since.

Thus, when the graph J is sparsified in Line 14, by Theorem 7.5, we can conclude

$$\mathrm{vcong}\left(\Pi_{J \to H_{<j} \cup E_{affected}} \circ \Pi_{J \to \widetilde{J}}\right) \leq 2 \cdot 4^{j-1}\gamma_c^{j+1} \Delta_{\max}(G).$$

Finally, when we construct the embedding Π_j in Line 19, the path segments $[\Pi_{J \to H_{<j} \cup E_{affected}} \circ \Pi_{J \to \widetilde{J}}](\widehat{e})$ incur vertex congestion at most $\mathrm{vcong}(\Pi_{J \to H_{<j} \cup E_{affected}} \circ \Pi_{J \to \widetilde{J}})$, and the path segments $\Pi_{<j}(e)[a, \widehat{a}]$ and $\Pi_{<j}(e)[\widehat{b}, b]$ incur total vertex congestion at most $\mathrm{vcong}(\Pi_{<j}^{(t_j)})$.

As congestion is additive, we can upper bound the total congestion of $\Pi_j^{(t)}$ by $(4^{j-1}\gamma_c^j + 2 \cdot 4^{j-1}\gamma_c^{j+1})\Delta_{\max}(G)$ and can finally use $\mathrm{vcong}(\Pi_{\leq j}^{(t)}) \leq \mathrm{vcong}(\Pi_j^{(t)}) + \mathrm{vcong}(\Pi_{<j}^{(t)}) \leq 4^j \gamma_c^{j+1} \Delta_{\max}(G)$. For all $i > j$, we note that $\Pi_i^{(t)}$ is empty and therefore, $\mathrm{vcong}(\Pi_{\leq i}^{(t)}) \leq \mathrm{vcong}(\Pi_{\leq j}^{(t)})$. □

Claim 7.8. *For any $0 \leq i \leq L$ and stage t divisible by $n^{1-i/L}$, we have* $\mathrm{length}(\Pi_{\leq i}^{(t)}) \leq 2^i \gamma_l^{i+1}$.

Proof. We again take induction over time t. For $t = 0$, we note that Π_0 has length γ_l by Theorem 7.3. For $t - 1 \to t$, for $j = j^{(t)}$ and $t_{j-1} = t_{j-1}^{(t)}$, we have by induction hypothesis that $\Pi_{<j}^{(t_{j-1})}(e) \leq 2^{j-1}\gamma_l^j$. But note that when we set the path $\Pi_j^{(t)}(e)$ in Line 19, then the segments $\Pi_{<j}^{(t)}(e)[a, \widehat{a}]$ and $\Pi_{<j}^{(t)}(e)[\widehat{b}, b]$ (combined) are of length at most $2^{j-1}\gamma_l^j$ because they survived from $\Pi_{<j}^{(t_{j-1})}(e)$ by definition of S_{j-1}. Further, the segment $[\Pi_{J \to H_{<j} \cup E_{affected}} \circ \Pi_{J \to \widetilde{J}}](\widehat{e})$ has length at most $\gamma_l \cdot \mathrm{length}(\Pi_{J \to H_{<j} \cup E_{affected}})$ by Theorem 7.5. But by construction in Line 13, the embedding $\Pi_{J \to H_{<j} \cup E_{affected}}$ has length at most $\mathrm{length}(\Pi_{<j}^{(t_{j-1})}(e)) + 1$.

Combining these insights, we have $\mathrm{length}(\Pi_{\leq j}^{(t)}) \leq 2^{j-1}\gamma_l^j + \gamma_l(\mathrm{length}(\Pi_{<j}^{(t_{j-1})}(e)) + 1) \leq 2^j \gamma_l^{j+1}$. For $i > j$, we have $\mathrm{length}(\Pi_{\leq j}) = \mathrm{length}(\Pi_{\leq i})$. □

Now that we established all properties of the embedding, it remains to analyze the sparsifier H.

Lemma 7.9. *At any stage, H consists of at most $\widetilde{O}(n)$ edges and the amortized number of changes to the edge set of H per update is $\widetilde{O}(n^{1/L})$. $D^{(t)}$ is of amortized size $\widetilde{O}(n^{1/L}(\gamma_c\gamma_l)^{O(L)})$. Initialization time of the algorithm is $\widetilde{O}(m\gamma_l)$ and it has amortized update time $\widetilde{O}(n^{1/L}(\gamma_c\gamma_l)^{O(L)}\Delta_{\max}(G))$.*

Proof. The graph H_0 is computed during initialization and remains fixed and therefore consists of $\widetilde{O}(n)$ edges by Theorem 7.3 and contributes no recourse. For each $j > 0$, H_j is initially empty and only has edges added in stages t divisible by $n^{1-j/L}$ (but not by $n^{1-(j-1)/L}$) in Line 15. In each such stage t, the graph J is formed over the vertices S_{j-1}.

It is straight-forward to see by Definition 7.2 and Line 6 that S_{j-1} is of size at most $n^{1-(j-1)/L}$ at any stage. Thus, when the graph $\widetilde{J}$ is computed, it consists of at most $\widetilde{O}(n^{1-(j-1)/L})$ edges by Theorem 7.5. The bounds on overall sparsity of H follow.

For the claim on the recourse, we note that in stages t divisible by $n^{1-j/L}$ (but not by $n^{1-(j-1)/L}$), we recompute a spanner on the vertices S_{j-1} which are a subset of the vertices in $G^{(t)}$ and add $\widetilde{O}(n^{1-(j-1)/L})$ edges. For the graphs $H_{j'}^{(t)}$ for $j' > j$, the graphs are empty after the algorithm finishes. Using an inductive argument, we can argue that the number of edge deletions at stage t can also be upper bound by $\widetilde{O}(n^{1-(j-1)/L})$. Thus, it is not hard to see that at most $\widetilde{O}(n^{1/L})$ amortized changes to the edge set of H are made. It remains to argue about a rather subtle detail: if the update is a vertex split applied to $G^{(t-1)}$ to obtain $G^{(t)}$, then we also need to account for the recourse caused by the vertex split to the graphs $H_{j'}^{(t-1)}$ for $j' < j$. But note that we only pay in recourse cost for edges that are moved from a vertex v to a vertex v' if the degree of v' after the vertex split is at most half the degree of v's degree. Thus, we can charge each edge that is moved this way. Further, if v''s degree is then again increased by a factor of $3/2$, we can further re-pay that cost of moving by charging the newly inserted edges. Following this charging scheme, we can argue that each edge can be charged to pay $O(\log(n))$ on insertion and an additional $O(\log(n))$ in recourse for the halving of degrees (after being recompensated if the degree goes up again). Since there are $\widetilde{O}(n)$ edges initially in H and at most $\widetilde{O}(n^{1/L})$ new edges after each update appear, our recourse bound follows.

To obtain the bound on $D^{(t)}$, we first observe that for each path $\Pi_j(e')$ constructed in Line 19, by induction over time, we can straight-forwardly establish that $\Pi_{<j}(e')[a, \widehat{a}]$ and $\Pi_{<j}(e')[\widehat{b}, b]$ are subpaths of $\Pi_{G\to H}^{(t-1)}(e')$ by the properties of set S_{j-1}. Thus, the only edges e on any such $\Pi_j(e')$ not already on the path $\Pi_{G\to H}^{(t-1)}(e')$ are the edges in the subpath $[\Pi_{J\to H_{<j}\cup E_{affected}} \circ \Pi_{J\to\widetilde{J}}](\widehat{e'})$. But clearly, $[\Pi_{J\to H_{<j}\cup E_{affected}} \circ \Pi_{J\to\widetilde{J}}](\widehat{e'}) \subseteq \Pi_{J\to H_{<j}\cup E_{affected}}(\widetilde{J})$. It remains to use our bound on the number of edges in $\widetilde{J}$ and the fact that the map $\Pi_{J\to H_{<j}\cup E_{affected}}$ maps edges to paths of length $\gamma_c^{O(L)}$ in H by Claim 7.8.

For the running time, we use Theorem 7.3 for the initialization, and observe that each vertex in J as analyzed above has degree at most $O(\gamma_c^{O(L)}\Delta_{\max}(G))$ as discussed in the proof of Claim 7.7. Thus, using Theorem 7.5 computing each sparsifier $\widetilde{J}$ of J only takes time $\widetilde{O}(|V(J)|\gamma_c^{O(L)}\Delta_{\max}(G))$. By standard amortization arguments and the fact that the time to compute the sparsifier dominates the update time of Update(t) asymptotically, the lemma follows. □

To complete the proof of Theorem 7.1, we only have to analyze the success probability, which is straight-forward as the only random event at each stage is the invocation of the procedure Sparsify. Thus, taking a simple union bound over these events at all stages gives the desired result.

7.2 Implementing the Sparsification Procedure

It remains to prove the procedure that statically sparsifies graphs.

Theorem 7.5. *Given unweighted, undirected graphs H' and J with $V(J) \subseteq V(H')$ and an embedding $\Pi_{J\to H'}$ from J into H'. Then, there is a randomized algorithm Sparsify$(H', J, \Pi_{J\to H'})$ that returns a sparsifier $\widetilde{J} \subseteq J$ with $|E(\widetilde{J})| = \widetilde{O}(|V(J)|)$ and an embedding $\Pi_{J\to\widetilde{J}}$ from J to $\widetilde{J}$ such that*

(1) $\text{vcong}(\Pi_{J\to H'} \circ \Pi_{J\to\widetilde{J}}) \le \gamma_c \cdot \big(\text{vcong}(\Pi_{J\to H'}) + \Delta_{\max}(J)\big)$, *and*
(2) $\text{length}(\Pi_{J\to H'} \circ \Pi_{J\to\widetilde{J}}) \le \gamma_l \cdot \text{length}(\Pi_{J\to H'})$.

The algorithm runs in time $\widetilde{O}(|E(J)| \cdot \gamma_l)$ and succeeds with probability at least $1 - n^{-C}$ for any constant C, specified before the procedure is invoked.

Additional Tools. At a high level, the proof of Theorem 7.5 follows by performing an expander decomposition, uniformly subsampling each expander to produce a sparsifier, and then embedding each expander into its sparsifier by using a data structure for outputting short paths between vertices in decremental expanders. To formalize this, we start by surveying some tools on expander graphs. Recall the definiton of expanders.

Definition 7.10 (Expander). *Let G be an unweighted, undirected graph and $\phi \in (0, 1]$, then we say that G is a ϕ-expander if for all $\emptyset \neq S \subseteq V$, $|E_G(S, V \setminus S)| \geq \phi \cdot \min\{\mathrm{vol}_G(S), \mathrm{vol}_G(V \setminus S)\}$.*

We can further get a collection of expander decomposition with near uniform degrees in the expanders. The proof of this statement follows almost immediately from [118] and is therefore deferred to Appendix A.2.

Theorem 7.11. *Given an unweighted, undirected graph G, there is an algorithm $\textsc{Decompose}(G)$ that computes an edge-disjoint partition of G into graphs $G_0, G_1, \ldots, G_\ell$ for $\ell = O(\log n)$ such that for each $0 \leq i \leq \ell$, $|E(G_i)| \leq 2^i n$ and for each nontrivial connected component X of G_i, $G_i[X]$ is a ψ-expander for $\psi = \Omega(1/\log^3(m))$, and each $x \in X$ has $\deg_{G_i}(x) \geq \psi 2^i$. The algorithm runs correctly in time $O(m \log^7(m))$, and succeeds with probability at least $1 - n^{-C}$ for any constant C, specified before the procedure is invoked.*

Further, we use the following result from [29]. Given a ϕ-expander undergoing edge deletions the data structure below implicitly maintains a subset of the expander that still has large conductance using standard expander pruning techniques (see for example [105, 118]). Further on the subset of the graph that still has good conductance, it can output a path of length $m^{o(1)}$ between any pair of queried vertices.

Theorem 7.12 (see Theorem 3.9 in arXiv v1 in [29]). *Given an unweighted, undirected graph G that is ϕ-expander for some $\phi > 0$. There is a deterministic data structure $\mathcal{DS}_{ExpPath}$ that explicitly grows a monotonically increasing "forbidden" vertex subset $\widehat{V} \subseteq V(G)$ while handling the following operations:*

- *$\textsc{Delete}(e)$: Deletes edge e from $E(G)$ and then explicitly outputs a set of vertices that were added to $\widehat{V}$ due to the edge deletion.*
- *$\textsc{GetPath}(u, v)$: for any $u, v \in V(G) \setminus \widehat{V}$ returns a path consisting of at most $\gamma_{ExpPath}$ edges between u and v in the graph $G[V(G) \setminus \widehat{V}]$. Each path query can be implemented in time $\gamma_{ExpPath}$, where $\gamma_{ExpPath} = (\log(m)/\phi)^{O(\sqrt{\log(m)})}$. The operation does not change the set $\widehat{V}$.*

The data structure ensures that after t edge deletions $\mathrm{vol}_{G^{(0)}}(\widehat{V}) \leq \gamma_{del} t/\phi$ for some constant $\gamma_{del} = O(1)$. The total update time taken by the data structure for initialization and over all deletions is $O(|E(G^{(0)})|\gamma_{ExpPath})$.

The Algorithm. We can now use these tools to give Algorithm 2 that implements the procedure $\textsc{Sparsify}(H', J, \Pi_{J \to H'})$. The algorithm has two key steps:

(1) Sampling: The graph J is first decomposed via Theorem 7.11. Then, the algorithm iterates over ψ-expanders $J_i[X]$ with near-uniform degrees. It is well-known that to obtain a sparsifier $\widetilde{J}_{X,i}$ of such graphs, one can simply sample each edge with probability roughly $\frac{\log(m)}{\psi \Delta_{min}(J_i[X])}$. To obtain the final sparsifier $\widetilde{J}$, we only have to take the union over all samples $\widetilde{J}_{X,i}$.

(2) Embedding: We then proceed to find an embedding for edges in $J_i[X]$ into $\widetilde{J}_{X,i}$. The sampled edges can be handled trivially by embedding them into themselves. To embed the remaining edges $e \in E(J)$ into $\widetilde{J}_{X,i}$, we exploit that $\widetilde{J}_{X,i}$ is an expander graph which allows us to employ the data structure from Theorem 7.12 on $\widetilde{J}_{X,i}$ to query for a path between the endpoints of e in $\widetilde{J}_{X,i}$ efficiently. We further keep track of the congestion of

Algorithm 2: SPARSIFY$(H', J, \Pi_{J\to H'})$

$J_0, J_1, \ldots, J_\ell \leftarrow$ DECOMPOSE(J).;
$\widetilde{J} \leftarrow (V, \emptyset)$.;
foreach $e \in E(J)$ **do** $\Pi_{J\to\widetilde{J}}(e) \leftarrow \emptyset$.;
foreach $i \in [0, \ell]$ *and connected component* X *in* J_i **do**
  /* Sample the edges that are added to the sparsifier $\widetilde{J}$. */
  $p_{X,i} \stackrel{\text{def}}{=} \min\left\{\frac{96C\log(m)}{\psi\Delta_{min}(J_i[X])}, 1\right\}$. ;
  Construct graph $\widetilde{J}_{X,i}$ by sampling each edge $e \in E(J_i[X])$ independently with probability $p_{X,i}$.;
  Add all edges in $\widetilde{J}_{X,i}$ to $\widetilde{J}$.;
  /* Embed all edges in $J_i[X]$ into the sampled local graph $\widetilde{J}_{X,i}$. */
  foreach $e \in \widetilde{J}_{X,i}$ **do** $\Pi_{J\to\widetilde{J}}(e) \leftarrow e$. ;
  while *there exists an edge* $e \in E(J[X_i])$ *with* $\Pi_{J\to\widetilde{J}}(e) = \emptyset$ **do**
    $\widetilde{J}_{APSP} \leftarrow$ a copy of $\widetilde{J}_{X,i}$.;
    Initialize $\mathcal{DS}_{ExpPath}$ on graph $\widetilde{J}_{APSP}$ with parameter $\phi \stackrel{\text{def}}{=} \psi/4$ maintaining set $\widehat{V}$.;
    foreach $e \in E(\widetilde{J}_{X,i})$ **do** $cong(e) \leftarrow 0$.;
    while *there exists an edge* $e \in E(J[V \setminus \widehat{V}])$ *with* $\Pi_{J\to\widetilde{J}}(e) = \emptyset$ **do**
      Let u and v be the endpoints of edge e.;
      $\Pi_{J\to\widetilde{J}}(e) \leftarrow \mathcal{DS}_{ExpPath}$.GETPATH$(u, v)$.;
      foreach $e \in \Pi_{J\to\widetilde{J}}(e)$ **do**
        $cong(e) \leftarrow cong(e) + 1$;
        if $cong(e) \geq \tau \stackrel{\text{def}}{=} \frac{\gamma_{ExpPath}\gamma_{del}}{\psi p_{X,i}}$ **then**
          Remove edge e from $\widetilde{J}_{APSP}$ via $\mathcal{DS}_{ExpPath}$.DELETE(e).
return $(\widetilde{J}, \Pi_{J\to\widetilde{J}})$

each edge in $\widetilde{J}_{X,i}$ by our embedding and remove edges that are too congested (at least until we cannot embed anymore in any other way).

We point out that Algorithm 2 in no way uses the embedding $\Pi_{J\to H'}$. Still, we show that due to the structure given, we can tightly upper bound the congestion and length of the embedding given by the composition $\Pi_{J\to H'} \circ \Pi_{J\to\widetilde{J}}$.

Proof of Theorem 7.5. We start by proving the following structural claim. For the rest of the section, we condition on the event that it holds for each relevant i and X.

Claim 7.13. *For each i, and connected component X in J_i, the corresponding sample $\widetilde{J}_{X,i}$ satisfies for each $S \subseteq X$ that $\frac{1}{2}|E_{\widetilde{J}_{X,i}}(S, X\setminus S)|/p_{X,i} \leq |E_{J_i}(S, X\setminus S)| \leq 2|E_{\widetilde{J}_{X,i}}(S, X\setminus S)|/p_{X,i}$ with probability at least $1 - n^{-2C}$.*

PROOF. Since for $i = 0$, $J_i[X] = \widetilde{J}_{X,i}$ and $p_{X,i} = 1$, the claim is vacuously true. For $i > 0$, consider any cut $(S, X\setminus S)$ in $J_i[X]$ and assume wlog $k = |S| \leq |X\setminus S|$. Since $J_i[X]$ is a ψ-expander, we have that $|E_{J_i}(S, X\setminus S)| \geq \psi\Delta_{min}(J_i[X])|S|$ by Definition 7.10. The algorithm samples each such edge e into the sample $\widetilde{J}_{X,i}$ independently with probability $p_{X,i}$. Thus, $\mathbb{E}|E_{\widetilde{J}_{X,i}}(S, X\setminus S)| = |E_G(S, X\setminus S)| \cdot p_{X,i} \geq 48C\log(m)|S|$.

Using a Chernoff bound as in Theorem 5.1 on the random variable $|E_{\widetilde{J}_{X,i}}(S, X \setminus S)|$, we can thus conclude that our claim is correct on the cut $(S, X \setminus S)$ with probability at least $1 - 2m^{-4Ck}$. Since there are at most $\binom{|X|}{k} \leq |X|^{2k}$ cuts where the smaller side has exactly k vertices, we can finally use a union bound over all cuts to complete the proof. □

By the claim above, and the fact that each graph $J_i[X]$ (for $i > 0$) is a ψ-expander by Theorem 7.11, we can conclude that each $\widetilde{J}_{X,i}$ is a $\psi/4$-expander.

Corollary 7.14. *For any $i > 0$ and X as used in Algorithm 2, $\widetilde{J}_{X,i}$ is a $\psi/4$-expander.*

This implies that our initializations of the data structure $\mathcal{DS}_{ExpPath}$ in Line 11 are legal according to Theorem 7.12. Next, let us give an upper bound on the congestion of the embedding $\Pi_{J \to \widetilde{J}}$.

Claim 7.15. *For any $i \in [0, \ell]$ and X as used in Algorithm 2, we have $\operatorname{econg}(\Pi_{J \to \widetilde{J}}|_{E(J_i[X])}) \leq \gamma_{X,i} = O\left(\frac{\gamma_{ExpPath}\gamma_{del}\log(m)}{\psi p_{X,i}}\right)$ where $\Pi_{J \to \widetilde{J}}|_{E(J_i[X])}$ denotes the embedding $\Pi_{J \to \widetilde{J}}$ restricted to edges in $J_i[X]$.*

Proof. We first observe that only in the foreach loop iteration on i and X, can any congestion be added to edges in $\widetilde{J}_{X,i}$ by the disjointness of the graphs J_j (see Theorem 7.11). Further, in the particular iteration on i and X, up to the while-loop starting in Line 9, the congestion of $\Pi_{J \to \widetilde{J}}|E(J_i[X])$ is at most 1 since only edges sampled into $\widetilde{J}_{X,i}$ are embedded into themselves.

It is straight-forward to observe that the congestion of the partial embedding $\Pi_{J \to \widetilde{J}}$ (restricted to $E(J_i[X])$) throughout each iteration of the outer-while loop starting in Line 9 is increased by at most τ as the algorithm track congestion of the current iteration explicitly and removes edges that are too congested in Line 19. It thus remains to bound the number of iterations of the outer-while loop starting in Line 9 by by $O(\log(m))$. We can then conclude that the total congestion on the edges is at most $O(\tau \log(m))$.

To bound this number of iterations, let us analyze a single outer while-loop iteration (starting at Line 9), and fix the end of such an iteration. Let t be the number of deletions processed by the data structure $\mathcal{DS}_{ExpPath}$ throughout the iteration and $\widehat{V}^{FINAL}$, the set $\widehat{V}$ at the end of the while-loop iteration. Using that the while-loop terminates, we can further conclude that the only edges not embedded after the current iteration are those outside $E(J_i[X \setminus \widehat{V}^{FINAL}])$. By Theorem 7.12, $\operatorname{vol}_{\widetilde{J}_{X,i}}(\widehat{V}^{FINAL}) \leq 4\gamma_{del}t/\psi$ and therefore by Claim 7.13, we have $|E(J_i[X]) \setminus E(J_i[X \setminus \widehat{V}^{FINAL}])| \leq \operatorname{vol}_{J_i[X]}(\widehat{V}^{FINAL}) \leq \frac{8\gamma_{del}t}{p_{X,i}\psi}$. But at the same time, we know that at least $t \cdot \tau/\gamma_{ExpPath}$ edges have been embedded in the current while-loop iteration, since each edge embedding adds at most $\gamma_{ExpPath}$ units to the total congestion. We conclude that each iteration, we embed at least a $\frac{1}{16}$-fraction of the edges in $J_i[X]$ that where not embedded before the current while-loop iteration. It follows that there are at most $O(\log(m))$ iterations, which establishes our claim. □

Claim 7.16. $\operatorname{vcong}(\Pi_{J \to H'} \circ \Pi_{J \to \widetilde{J}}) \leq \gamma_c \cdot \left(\operatorname{vcong}(\Pi_{J \to H'}) + \Delta_{\max}(J)\right)$ *with probability at least $1 - n^{-2C}$.*

Proof. Let us fix any vertex $v \in V(H')$. We define $E_v = \{e \in E(J) \mid v \in \Pi_{J \to H'}(e)\}$ to be the edges in J whose embedding path contains v. By definition of vertex congestion for embeddings, $|E_v| \leq \operatorname{vcong}(\Pi_{J \to H'})$.

Next, for each edge $e \in E_v$, let e be in J_i after the decomposition in Line 1 in the component X, we define the random variable

$$Y_e = \begin{cases} \gamma_{X,i} & \text{if } e \in E(\widetilde{J}) \\ 0 & \text{otherwise} \end{cases}$$

Note that the random variables Y_e are independent as edges are sampled independently at random into $\widetilde{J}$ in Line 6. Further, by Claim 7.15, we have for each edge e, $\operatorname{econg}(\Pi_{J \to \widetilde{J}}|_{E(J_i[X])}, e) \leq Y_e$ and thus $\operatorname{vcong}(\Pi_{J \to H'} \circ \Pi_{J \to \widetilde{J}}, v) \leq \sum_{e \in E_v} Y_e$.

We will bound this sum using a Chernoff bound. We first observe that every variable $Y_e \in [0, W]$ for $W = \gamma_{var}\gamma_{ExpPath}\Delta_{\max}(J)$ for some scalar γ_{var} which follows from the definitions of $\gamma_{X,i}$ in Claim 7.15 and $p_{X,i}$ in Line 5. Across all the edge congestion variables Y_e, we have a uniform bound μ_{edge} on the expectation given by $\mathbb{E}[Y_e] = p_{X,i} \cdot \gamma_{X,i} = \mu_{\text{edge}}$. Therefore $\mathbb{E}[\sum_{e \in E_v} Y_e] \leq \mu_{\text{edge}} \cdot \text{vcong}(\Pi_{J \to H'})$. We can conclude by Theorem 5.1 that

$$\mathbb{P}\left[\sum_{e \in E_v} Y_e \leq 24C \log(n) \cdot W + 2\mu_{\text{edge}} \text{vcong}(\Pi_{J \to H'})\right] > 1 - 2n^{-4C}.$$

We can now set $\gamma_c = 24C \log(n) \cdot \gamma_{ExpPath} \cdot \gamma_{var} + 2\mu_{\text{edge}} = (\gamma_{ExpPath})^{O(1)}$ which is consistent with our requirements on γ_c. Finally, it remains to take a simple union bound over all vertices $v \in V(H')$. □

Claim 7.17. $\text{length}(\Pi_{J \to H'} \circ \Pi_{J \to \widetilde{J}}) \leq \gamma_l \cdot \text{length}(\Pi_{J \to H'})$.

Proof. Consider any edge $e \in E(J)$. If e is sampled into $\widetilde{J}$, then $\Pi_{J \to \widetilde{J}}(e) = e$, as can be seen in Line 8. Otherwise, $\Pi_{J \to \widetilde{J}}(e)$ is of length at most $\gamma_{ExpPath}$ as can be seen from Line 15 and Theorem 7.12. Setting $\gamma_l = \gamma_{ExpPath}$ thus ensures our claim. □

Combining Claim 7.16 and Claim 7.17, we have established the properties claimed in Theorem 7.5. The success probability follows by taking a straight-forward union bound over the events used in the analysis above and the success of Theorem 7.11. The run-time of the algorithm can be seen from inspecting Algorithm 2, Theorem 7.11, the fact that the while-loop in Line 9 runs at most $O(\log(n))$ times for each iteration of the outer foreach-loop (established in the proof of Claim 7.15) and finally the run-time guarantees on the data structure in Theorem 7.12.

8 DATA STRUCTURE CHAIN

The goal of Sections 8 to 10 is to build a data structure to dynamically maintain $m^{o(1)}$-approximate undirected minimum-ratio cycles under changing costs and lengths, i.e. for gradients $\boldsymbol{g} \in \mathbb{R}^E$ and lengths $\boldsymbol{\ell} \in \mathbb{R}^E_{>0}$ return a (compactly represented) cycle $\boldsymbol{\Delta}$ satisfying $\boldsymbol{B}^\top \boldsymbol{\Delta} = 0$ and

$$\frac{\langle \boldsymbol{g}, \boldsymbol{\Delta} \rangle}{\|\boldsymbol{L}\boldsymbol{\Delta}\|_1} \leq m^{-o(1)} \min_{\boldsymbol{B}^\top \boldsymbol{f} = 0} \frac{\langle \boldsymbol{g}, \boldsymbol{f} \rangle}{\|\boldsymbol{L}\boldsymbol{f}\|_1}. \tag{24}$$

Our data structure does not work against fully adaptive adversaries. However, it works for updates coming from the IPM. We capture this notion with the following definition.

Definition 8.1 (Hidden Stable-Flow Chasing Updates)**.** *Consider a dynamic graph $G^{(t)}$ undergoing batches of updates $U^{(1)}, \ldots, U^{(t)}, \ldots$ consisting of edge insertions/deletions and vertex splits. We say the sequences $\boldsymbol{g}^{(t)}, \boldsymbol{\ell}^{(t)}$, and $U^{(t)}$ satisfy the* hidden stable-flow chasing *property if there are hidden dynamic circulations $\boldsymbol{c}^{(t)}$ and hidden dynamic upper bounds $\boldsymbol{w}^{(t)}$ such that the following holds at all stages t:*

1. *$\boldsymbol{c}^{(t)}$ is a circulation: $\boldsymbol{B}^\top_{G^{(t)}} \boldsymbol{c}^{(t)} = 0$.*
2. *$\boldsymbol{w}^{(t)}$ upper bounds the length of $\boldsymbol{c}^{(t)}$: $|\boldsymbol{\ell}_e^{(t)} \boldsymbol{c}_e^{(t)}| \leq \boldsymbol{w}_e^{(t)}$ for all $e \in E(G^{(t)})$.*
3. *For any edge e in the current graph $G^{(t)}$, and any stage $t' \leq t$, if the edge e was already present in $G^{(t')}$, i.e. $e \in G^{(t)} \setminus \bigcup_{s=t'+1}^{t} U^{(s)}$, then $\boldsymbol{w}_e^{(t)} \leq 2\boldsymbol{w}_e^{(t')}$.*
4. *Each entry of $\boldsymbol{w}^{(t)}$ and $\boldsymbol{\ell}^{(t)}$ is quasipolynomially lower and upper-bounded:*

$$\log \boldsymbol{w}_e^{(t)} \in [-\log^{O(1)} m, \log^{O(1)} m] \text{ and } \log \boldsymbol{\ell}_e^{(t)} \in [-\log^{O(1)} m, \log^{O(1)} m] \text{ for all } e \in E(G^{(t)}).$$

Intuitively Definition 8.1 says that even while $\boldsymbol{g}^{(t)}$ and $\boldsymbol{\ell}^{(t)}$ change, there is a witness circulation $\boldsymbol{c}^{(t)}$ that is fairly stable. More precisely, there is some upper bound $\boldsymbol{w}^{(t)}$ on the coordinate-wise lengths of $\boldsymbol{c}^{(t)}$ that increases by at most a factor of 2, except on edges that are explicitly updated. Interestingly, even though both $\boldsymbol{c}^{(t)}$ and $\boldsymbol{w}^{(t)}$ are hidden from the data structure, their existence is sufficient.

The IPM guarantees in Section 6 can be connected to Definition 8.1 by setting $\boldsymbol{c}^{(t)} = \boldsymbol{f}^* - \boldsymbol{f}^{(t)}$ and $\boldsymbol{w}^{(t)} = 10 + |\boldsymbol{\ell}(\boldsymbol{f}^{(t)}) \circ \boldsymbol{c}^{(t)}|$, where $\boldsymbol{f}^{(t)}$ is the current flow maintained by our algorithm. The guarantees of Definition 8.1 then hold by a combination of Lemmas 6.7, 6.9 and 6.10. This is formalized in Lemma 11.2 in Section 11.

Our main data structure dynamically maintains min-ratio cycles under hidden stable-flow chasing updates.

Theorem 8.2 (Dynamic Min-Ratio Cycle with Hidden Stable-Flow Chasing Updates)**.** *There is a data structure that on a dynamic graph $G^{(t)}$ maintains a collection of $s = O(\log n)^d$ spanning trees $T_1, T_2, \ldots, T_s \subseteq G^{(t)}$ for $d = O(\log^{1/8} m)$, and supports the following operations:*

- *$\textsc{Update}(U^{(t)}, \boldsymbol{g}^{(t)}, \boldsymbol{\ell}^{(t)})$: Update the gradients and lengths to $\boldsymbol{g}^{(t)}$ and $\boldsymbol{\ell}^{(t)}$. For the update to be supported, we require that $U^{(t)}$ contains only edge insertions/deletions and $\boldsymbol{g}^{(t)}, \boldsymbol{\ell}^{(t)}$ and $U^{(t)}$ satisfy the hidden stable-flow chasing property (Definition 8.1) with hidden circulation $\boldsymbol{c}^{(t)}$ and upper bounds $\boldsymbol{w}^{(t)}$, and for a parameter α,*
$$\frac{\langle \boldsymbol{g}^{(t)}, \boldsymbol{c}^{(t)} \rangle}{\|\boldsymbol{w}^{(t)}\|_1} \leq -\alpha.$$
- *$\textsc{Query}()$: Return a tree T_i for $i \in [s]$ and a cycle $\boldsymbol{\Delta}$ represented as $m^{o(1)}$ paths on T_i (specified by their endpoints and the tree index) and $m^{o(1)}$ explicitly given off-tree edges such that for $\kappa = \exp(-O(\log^{7/8} m \cdot \log\log m))$,*
$$\frac{\langle \boldsymbol{g}^{(t)}, \boldsymbol{\Delta} \rangle}{\|\boldsymbol{L}^{(t)}\boldsymbol{\Delta}\|_1} \leq -\kappa\alpha.$$

Over τ stages the algorithm succeeds whp. with total runtime $m^{o(1)}(m + Q)$ for $Q = \sum_{t\in[\tau]} |U^{(t)}|$.

To interpret Theorem 8.2, note that $\boldsymbol{\Delta} = \boldsymbol{c}^{(t)}$ would be a valid output by the guarantees in Definition 8.1, i.e. $\left\|\boldsymbol{L}^{(t)}\boldsymbol{\Delta}\right\|_1 \leq \left\|\boldsymbol{w}^{(t)}\right\|_1$ from Item 2. Thus the data structure guarantee can be interpreted as efficiently representing and returning a cycle whose quality is within a $m^{o(1)}$ factor of $\boldsymbol{c}^{(t)}$. Eventually, we will add $\boldsymbol{\Delta}$ to our flow efficiently by using link-cut trees.

Section 8 focuses on introducing the general layout of the data structure, and is definition-heavy. Section 9 explains how to plug in the circulations $\boldsymbol{c}$ and upper bounds $\boldsymbol{w}$ in our data structure, and shows a weaker version of Theorem 8.2 in Theorem 9.1. We use the weaker Theorem 9.1 to show the full cycle-finding result Theorem 8.2 by defining a rebuilding game in Section 10.

8.1 Dynamic Low-Stretch Decompositions (LSD)

In the following subsections we describe the components of the data structure we maintain to show Theorem 8.2. At a high level, our data structure consists of d levels, each of which has approximately a factor of $k = m^{1/d}$ fewer edges than the previous level. The edge reduction is achieved in two parts. First, we reduce the number of vertices to $\widetilde{O}(m/k)$ by maintaining a spanning forest F of G with $\widetilde{O}(m/k)$ connected components, and then recurse on G/F, the graph where each connected component of F in G is contracted to a single vertex. While G/F now has $\widetilde{O}(m/k)$ vertices, it still potentially has up to m edges, so we need to employ the dynamic sparsification procedure in Theorem 7.1 to reduce the number of edges to $\widehat{O}(m/k)$.

We start by defining a rooted spanning forest and its induced stretch.

Variable	Definition
$\boldsymbol{\ell}^{(t)}, \boldsymbol{g}^{(t)}$	Lengths and gradients on a dynamic graph $G^{(t)}$
$\boldsymbol{c}^{(t)}, \boldsymbol{w}^{(t)}$	Hidden circulation & upper bounds with $\lvert\boldsymbol{\ell}^{(t)} \circ \boldsymbol{c}^{(t)}\rvert \le \boldsymbol{w}^{(t)}$ (Definition 8.1)
F	Rooted spanning forest of G (Definition 8.3).
$\boldsymbol{p}(F[u,v])$	Path vector from $u \to v$ in a forest F
$\mathrm{str}_e^{F,\boldsymbol{\ell}}$	Stretch of edge e with respect to spanning forest F and lengths $\boldsymbol{\ell}$ (Definition 8.4)
$\widetilde{\mathrm{str}}_e$	Stretch overestimates stable under edge deletions (Lemma 8.5)
$\mathcal{C}(G,F)$	Core graph from a spanning forest F (Definition 8.7)
$\widehat{e}$	Image of edge $e \in E(G)$ into the core graph $\mathcal{C}(G,F)$
$\mathcal{S}(G,F)$	Sparsified core graph $\mathcal{S}(G,F) \subseteq \mathcal{C}(G,F)$ (Definition 8.9)
$\mathcal{G}_0, \ldots, \mathcal{G}_d$	B-branching tree chain (Definition 8.10)
$G_0, \ldots, G_d$	Tree chain (Definition 8.10)
$T^{G_0,\ldots,G_d}$	Tree in G corresponding to tree chain $G_0, \ldots, G_d$ (Definition 8.11)
$\mathcal{T}^G$	Collection of B^d trees on G from B-branching tree chain (Definition 8.11)
$\mathrm{prev}_i^{(t)}$	Previous rebuild times of branching tree chain (Definition 8.12)

Table 1. Important definitions and notation to describe the data structure. In general a (t) superscript is the corresponding object at time t of a sequence of updates.

Definition 8.3 (Rooted Spanning Forest)**.** *A* rooted spanning forest *of a graph $G = (V, E)$ is a forest F on V such that each connected component of F has a unique distinguished vertex known as the* root*. We denote the root of the connected component of a vertex $v \in V$ as root_v^F.*

Definition 8.4 (Stretches of F)**.** *Given a rooted spanning forest F of a graph $G = (V, E)$ with lengths $\boldsymbol{\ell} \in \mathbb{R}^E_{>0}$, the stretch of an edge $e = (u, v) \in E$ is given by*

$$\mathrm{str}_e^{F,\boldsymbol{\ell}} \stackrel{\text{def}}{=} \begin{cases} 1 + \langle \boldsymbol{\ell}, |\boldsymbol{p}(F[u,v])| \rangle / \boldsymbol{\ell}_e & \text{if } \mathrm{root}_u^F = \mathrm{root}_v^F \\ 1 + \left\langle \boldsymbol{\ell}, |\boldsymbol{p}(F[u,\mathrm{root}_u^F])| + |\boldsymbol{p}(F[v,\mathrm{root}_v^F])| \right\rangle / \boldsymbol{\ell}_e & \text{if } \mathrm{root}_u^F \neq \mathrm{root}_v^F, \end{cases}$$

where $\boldsymbol{p}(F[\cdot,\cdot])$, as defined in Section 5, maps a path to its signed indicator vector.

When F is a spanning tree Definition 8.4 coincides with the definition of stretch for a LSST.

The goal of the remainder of this section is to give an algorithm to maintain a *Low Stretch Decomposition (LSD)* of a dynamic graph G. As a spanning forest decomposes a graph into vertex disjoint connected subgraphs, a LSD consists of a spanning forest F of low stretch. The algorithm produces stretch upper bounds that hold throughout all operations, and the number of connected components of F grows by amortized $\widetilde{O}(1)$ per update. At a high level, for any edge insertion or deletion, the algorithm will force both endpoints to become roots of some component of F. This way, any inserted edge will actually have stretch 1 because both endpoints are roots.

Lemma 8.5 (Dynamic Low Stretch Decomposition). *There is a deterministic algorithm with total runtime $\widetilde{O}(m)$ that on a graph $G = (V, E)$ with lengths $\boldsymbol{\ell} \in \mathbb{R}^E_{>0}$, weights $\boldsymbol{v} \in \mathbb{R}^E_{>0}$, and parameter k, initializes a tree T spanning V, and a rooted spanning forest $F \subseteq T$, a edge-disjoint partition $\mathcal{W}$ of F into $O(m/k)$ sub trees and stretch overestimates $\widetilde{\mathrm{str}}_e$. The algorithm maintains F decrementally against τ batches of updates to G, say $U^{(1)}, U^{(2)}, \ldots, U^{(\tau)}$, such that $\widetilde{\mathrm{str}}_e \stackrel{\text{def}}{=} 1$ for any new edge e added by either edge insertions or vertex splits, and:*

(1) *F has initially $O(m/k)$ connected components and $O(q \log^2 n)$ more after t update batches of total encoding size $q \stackrel{\text{def}}{=} \sum_{t=1}^{\tau} \textsc{Enc}(U^{(i)})$ satisfying $q \le \widetilde{O}(m)$.*

(2) $\mathrm{str}_e^{F,\boldsymbol{\ell}} \le \widetilde{\mathrm{str}}_e \le O(k\gamma_{LSST}\log^4 n)$ *for all* $e \in E$ *at all times, including inserted edges* e.
(3) $\sum_{e\in E^{(0)}} \boldsymbol{v}_e \widetilde{\mathrm{str}}_e \le O(\|\boldsymbol{v}\|_1 \gamma_{LSST}\log^2 n)$, *where* $E^{(0)}$ *is the initial edge set of* G.
(4) *Initially,* $\mathcal{W}$ *contains* $O(m/k)$ *subtrees. For any piece* $W \in \mathcal{W}$, $W \subseteq V$, $|\partial W| \le 1$ *and* $\mathrm{vol}_G(W \setminus R) \le O(k\log^2 n)$ *at all times, where* $R \supseteq \partial\mathcal{W}$ *is the set of roots in* F. *Here,* ∂W *denotes the set of* boundary vertices *that are in multiple partition pieces.*

Intuitively, the first property says that F has $O(m/k)$ roots initially and each update x adds $\widetilde{O}(\textsc{Enc}(x))$ roots to it. For example, each edge update adds $\widetilde{O}(1)$ roots to F. This allows us to satisfy the second property, which is that the stretch of e with respect to F (Definition 8.4) is upper bounded by some global upper bound $\widetilde{\mathrm{str}}_e$. Note that $\widetilde{\mathrm{str}}_e$ stays the same for any edge e across the execution of the algorithm. The third property says that these global upper bounds are still good on average with respect to the weights $\boldsymbol{v}$ up to $\widetilde{O}(1)$ factors. The final property is useful for interacting with our sparsifier in Theorem 7.1 whose runtime and congestion depend on the maximum degree of the input graphs.

We defer the proof of Lemma 8.5 to Appendix A.3.

8.2 Worst-Case Average Stretch via Multiplicative Weights

By doing a multiplicative weights update procedure (MWU) on top of Lemma 8.5, we can build a distribution over partial spanning tree routings whose average stretch on every edge is $\widetilde{O}(1)$. This is very similar to MWUs done in works of [82, 116] for building ℓ_∞ oblivious routings, and cut approximators [99, 119].

Lemma 8.6 (MWU). *There is a deterministic algorithm that on a graph* $G = (V, E)$ *with lengths* $\boldsymbol{\ell}$ *and a positive integer* k *computes* t *spanning trees, rooted spanning forests, and stretch overestimates* $\{(T_i, F_i \subseteq T_i, \widetilde{\mathrm{str}}_e^i)\}_{i=1}^t$ *(Lemma 8.5) for some* $t = \widetilde{O}(k)$ *such that*

$$\sum_{i=1}^{t} \boldsymbol{\lambda}_i \widetilde{\mathrm{str}}_e^i \le O(\gamma_{LSST}\log^2 n) \text{ for all } e \in E, \tag{25}$$

where $\boldsymbol{\lambda} \in \mathbb{R}_{>0}^{[t]}$ *is the uniform distribution over the set* $[t]$, *i.e.* $\boldsymbol{\lambda} = \vec{1}/t$.

The algorithm runs in $\widetilde{O}(mk)$*-time.*

The proof is standard and deferred to Appendix A.4.

If we sample a single tree/index from the distribution $\boldsymbol{\lambda}$, then any fixed flow will be stretched by $O(\gamma_{LSST}\log^2 n)$ on average. Hence any fixed flow will be stretched by $O(\gamma_{LSST}\log^2 n)$ by at least one of $O(\log n)$ trees sampled from $\boldsymbol{\lambda}$ with high probability. We will leverage this fact to analyze how the witness circulation $\boldsymbol{c}^{(t)}$ in Definition 8.1 and Theorem 8.2 is stretched by a random forest.

8.3 Sparsified Core Graphs and Path Embeddings

Given a rooted spanning forest F, we will recursively process the graph G/F where each connected component of F is contracted to a single vertex represented by the root. We call this the *core graph*, and define the lengths and gradients on it as follows. Below, we should think of G as the result of edge insertions/deletions to an earlier graph $G^{(0)}$, so $\widetilde{\mathrm{str}}_e = 1$ for edge inserted to get from $G^{(0)}$ to G, as enforced in Lemma 8.5.

Definition 8.7 (Core graph)**.** *Consider a tree* T *and a rooted spanning forest* $E(F) \subseteq E(T)$ *on a graph* G *equipped with stretch overestimates* $\widetilde{\mathrm{str}}_e$ *satisfying the guarantees of Lemma 8.5. We define the* core graph $C(G, F)$ *as a graph with the*

same edge and vertex set as G/F. For $e = (u, v) \in E(G)$ with image $\widehat{e} \in E(G/F)$ we define its length as $\boldsymbol{\ell}_{\widehat{e}}^{C(G,F)} \stackrel{\text{def}}{=} \widetilde{\text{str}}_e \boldsymbol{\ell}_e$ and gradient as $\boldsymbol{g}_{\widehat{e}}^{C(G,F)} \stackrel{\text{def}}{=} \boldsymbol{g}_e + \langle \boldsymbol{g}, \boldsymbol{p}(T[v, u]) \rangle$.

Remark 8.8. *In our usage, we maintain $C(G, F)$ where G is a dynamic graph and F is a decremental rooted spanning forest. In particular, T, F, and $\widetilde{\text{str}}$ are initialized and maintained via Lemma 8.5. As G undergoes dynamic updates such as edge deletions or vertex splits which adds new vertices to G, T won't be a spanning tree of G anymore. Definition 8.7 responds to such situation by allowing T not being a spanning tree nor a subgraph of G.*

Thus, for $e = (u, v) \in E(G)$, u and v may not be connected in T. In this case, the value of $\boldsymbol{g}_{\widehat{e}}^{C(G,F)}$ is simply $\boldsymbol{g}_e$. Also, the support of the gradient vector $\boldsymbol{g}$ is $E(G) \cup E(T)$. This corresponds to the case when some edge in T is removed from G, we keep the gradient on that edge as it is.

Note that the length and gradient of the image of an edge $e \in E(G)$ in Definition 8.7 do not change under edge deletions to F, because they are defined with respect to the tree T. This important property will be useful later in efficiently maintaining a sparsifier of the core graph, which we require to reduce the number of edges in the sparsified core graph to $\widehat{O}(m/k)$.

Definition 8.9 (Sparsified core graph)**.** *Given a graph G, forest F, and parameter k, define a $(\gamma_s, \gamma_c, \gamma_\ell)$-*sparsified core graph with embedding *as a subgraph $\mathcal{S}(G, F) \subseteq C(G, F)$ and embedding $\Pi_{C(G,F)\to\mathcal{S}(G,F)}$ satisfying*

1. *For any $\widehat{e} \in E(C(G, F))$, all edges $\widehat{e}' \in \Pi_{C(G,F)\to\mathcal{S}(G,F)}(\widehat{e})$ satisfy $\boldsymbol{\ell}_{\widehat{e}}^{C(G,F)} \approx_2 \boldsymbol{\ell}_{\widehat{e}'}^{C(G,F)}$.*
2. $\text{length}(\Pi_{C(G,F)\to\mathcal{S}(G,F)}) \le \gamma_l$ *and* $\text{econg}(\Pi_{C(G,F)\to\mathcal{S}(G,F)}) \le k\gamma_c$.
3. *$\mathcal{S}(G, F)$ has at most $m\gamma_s/k$ edges.*
4. *The lengths and gradients of edges in $\mathcal{S}(G, F)$ are the same as in $C(G, F)$ (Definition 8.7).*

In Section 9 we give a dynamic algorithm for maintaining a sparsified core graph of a graph G undergoing edge insertions and deletions. We defer the formal statement to Lemma 9.8 where we not only maintain a sparsified core graph but also show that the witness circulation $\boldsymbol{c}^{(t)}$ and upper bounds $\boldsymbol{w}^{(t)}$ from Definition 8.1 are preserved approximately.

8.4 Full Data Structure Chain

Our data structure has d levels. The graphs at the i-th level have about m/k^i edges, and each such graph branches into $O(\log n)$ graphs sampled from the distribution $\boldsymbol{\lambda}$ from Lemma 8.6.

Definition 8.10 (Branching Tree-Chain)**.** *For a graph G, parameter k, and branching factor B, a B-*branching tree-chain *consists of collections of graphs $\{\mathcal{G}_i\}_{0 \le i \le d}$, such that $\mathcal{G}_0 \stackrel{\text{def}}{=} \{G\}$, and we define $\mathcal{G}_i$ inductively as follows,*

1. *For each $G_i \in \mathcal{G}_i$, $i < d$, we have a collection of B trees $\mathcal{T}^{G_i} = \{T_1, T_2, \ldots, T_B\}$ and a collection of B forests $\mathcal{F}^{G_i} = \{F_1, F_2, \ldots, F_B\}$ such that $E(F_j) \subseteq E(T_j)$ satisfy the conditions of Lemma 8.5.*
2. *For each $G_i \in \mathcal{G}_i$, and $F \in \mathcal{F}^{G_i}$, we maintain $(\gamma_s, \gamma_c, \gamma_l)$-sparsified core graphs and embeddings $\mathcal{S}(G_i, F)$ and $\Pi_{C(G_i,F)\to\mathcal{S}(G_i,F)}$.*
3. *We let $\mathcal{G}_{i+1} \stackrel{\text{def}}{=} \{\mathcal{S}(G_i, F) : G_i \in \mathcal{G}_i, F \in F^{G_i}\}$.*

Finally, for each $G_d \in \mathcal{G}_d$, we maintain a low-stretch tree F.

We let a tree-chain *be a single sequence of graphs $G_0, G_1, \ldots, G_d$ such that G_{i+1} is the $(\gamma_s, \gamma_c, \gamma_l)$-sparsified core graph $\mathcal{S}(G_i, F_i)$ with embedding $\Pi_{C(G_i,F_i)\to\mathcal{S}(G_i,F_i)}$ for some $F_i \in \mathcal{F}^{G_i}$ for $0 \le i < d$, and a low-stretch tree F_d on G_d.*

In general, we will have $B = O(\log n)$ throughout, and we will omit B when discussing branching tree-chains. Note that level i of a branching tree-chain, i.e. the collection of graphs in $\mathcal{G}_i$, has at most $B^i = O(\log n)^i$ graphs for

$B = O(\log n)$. A branching tree-chain can alternatively be viewed as a set of $O(\log n)^d$ tree-chains, each of which naturally corresponds to a spanning tree of the top level graph G.

Definition 8.11 (Trees from Tree-Chains)**.** *Given a graph G and tree-chain $G_0, G_1, \ldots, G_d$ where $G_0 = G$, define the corresponding spanning tree $T^{G_0,G_1,\ldots,G_d} \stackrel{\text{def}}{=} \bigcup_{i=0}^{d} F_i$ of G as the union of preimages of edges of F_i in $G = G_0$.*

Define the set of trees corresponding to a branching tree-chain of graph G as the union of $T^{G_0,G_1,\ldots,G_d}$ over all tree-chains $G_0, G_1, \ldots, G_d$ where $G_0 = G$:

$$\mathcal{T}^G \stackrel{\text{def}}{=} \{T^{G_0,G_1,\ldots,G_d} : G_0, G_1, \ldots, G_d \text{ s.t. } G_{i+1} = \mathcal{S}(G_i, F_i) \text{ for all } 0 \le i < d\}$$

We can dynamically maintain a branching tree-chain such that we rebuild $\mathcal{G}_{i+1}$ from $\mathcal{G}_i$ every approximately m/k^i updates. Between rebuilds, the trees $\mathcal{T}^{G'}$ of graphs $G' \in \mathcal{G}_i$ stay the same, while the forests in $\mathcal{F}^{G'}$ are decremental as guaranteed in Lemma 8.5.

Definition 8.12 (Previous Rebuild Times)**.** *Given a dynamic graph $G^{(t)}$ with updates indexed by times $t = 0, 1, \ldots$ and corresponding dynamic branching tree-chain (Definition 8.10), we say that nonnegative integers $\text{prev}_0^{(t)} \le \text{prev}_1^{(t)} \le \cdots \le \text{prev}_d^{(t)} = t$ are* previous rebuild times *if $\text{prev}_i^{(t)}$ was the most recent time at or before t that $\mathcal{G}_i$ was rebuilt, i.e. for $G \in \mathcal{G}_i$ the set of trees $\mathcal{T}^G$ was reinitialized and sampled.*

We will assume that our algorithm rebuilds all $G_i \in \mathcal{G}_i$ at the same time: if we recompute a set of trees $\mathcal{T}^G$ for some $G_i \in \mathcal{G}_i$, then we also recompute the trees $\mathcal{T}^{G_i'}$ for all other $G_i' \in \mathcal{G}_i$. In the following Section 9 we show Theorem 9.1, we gives a data structure whose guarantee is weaker than Theorem 8.2. Precisely, the quality of the cycle returned depends on the previous rebuild times. We later boost this to an algorithm for Theorem 8.2 by solving a *rebuilding game* in Section 10.

9 ROUTINGS AND CYCLE QUALITY BOUNDS

The goal of this section is to explain how to route the witness circulations $\boldsymbol{c}^{(t)}$ and length upper bounds $\boldsymbol{w}^{(t)}$ through the branching tree-chain, and eventually recover an approximately optimal flow $\boldsymbol{\Delta}$. The main theorem we show in this section is the following.

Theorem 9.1. *Let $G = (V, E)$ be a dynamic graph undergoing τ batches of updates $U^{(1)}, \ldots, U^{(\tau)}$ containing* only *edge insertions/deletions with edge gradient $\boldsymbol{g}^{(t)}$ and length $\boldsymbol{\ell}^{(t)}$ such that the update sequence satisfies the hidden stable-flow chasing property (Definition 8.1) with hidden dynamic circulation $\boldsymbol{c}^{(t)}$ and width $\boldsymbol{w}^{(t)}$. There is an algorithm on G that maintains a $O(\log n)$-branching tree chain corresponding to $s = O(\log n)^d$ trees $T_1, T_2, \ldots, T_s$ (Definition 8.11), and at stage t outputs a circulation $\boldsymbol{\Delta}$ represented by $\exp(O(\log^{7/8} m \log\log m))$ off-tree edges and paths on some $T_i, i \in [s]$.*

The output circulation $\boldsymbol{\Delta}$ satisfies $\boldsymbol{B}^\top \boldsymbol{\Delta} = 0$ and for some $\kappa = \exp(-O(\log^{7/8} m \log\log m))$

$$\frac{\langle \boldsymbol{g}^{(t)}, \boldsymbol{\Delta} \rangle}{\left\| \boldsymbol{\ell}^{(t)} \circ \boldsymbol{\Delta} \right\|_1} \le \kappa \frac{\langle \boldsymbol{g}^{(t)}, \boldsymbol{c}^{(t)} \rangle}{\sum_{i=0}^{d} \|\boldsymbol{w}^{(\text{prev}_i^{(t)})}\|_1},$$

where $\text{prev}_i^{(t)}, i \in [d]$ are the previous rebuild times (Definition 8.12) for the branching tree chain.

The algorithm succeeds w.h.p. with total runtime $(m+Q)m^{o(1)}$ for $Q \stackrel{\text{def}}{=} \sum_{t=1}^{\tau} \left| U^{(t)} \right| \le \text{poly}(n)$. Also, levels $i, i+1, \ldots, d$ of the branching tree chain can be rebuilt at any point in $m^{1+o(1)}/k^i$ time.

The final sentence about rebuilding levels $i, i+1, \ldots, d$ allows us to force $\text{prev}_i^{(t)} = \text{prev}_{i+1}^{(t)} = \cdots = \text{prev}_d^{(t)} = t$. This is necessary because it is possible that $\|\boldsymbol{w}^{(\text{prev}_i^{(t)})}\|_1$ is much larger than $\|\boldsymbol{w}^{(t)}\|_1$ for some $0 \le i \le d$. This could result in

$\frac{\langle \boldsymbol{g}^{(t)}, \boldsymbol{c}^{(t)} \rangle}{\sum_{i=0}^{d} \|\boldsymbol{w}^{(\mathrm{prev}_i^{(t)})}\|_1}$ being much more than $\frac{\langle \boldsymbol{g}^{(t)}, \boldsymbol{c}^{(t)} \rangle}{\|\boldsymbol{w}^{(t)}\|_1}$. This is not sufficient to show Definition 8.1, which only guarantees that the latter quality is at most $-\alpha$, but does not assume a bound on the former. We will resolve this issue in Section 10 by carefully using our ability to rebuild levels $i, i+1, \ldots, d$ periodically whenever the cycle Δ returned by Theorem 9.1 is not good enough, and we deduce that $\|\boldsymbol{w}^{(\mathrm{prev}_i^{(t)})}\|_1$ is much larger than $\|\boldsymbol{w}^{(t)}\|_1$ for some $0 \le i \le d$.

9.1 Passing Circulations and Length Upper Bounds Through a Tree-Chain

Towards proving Theorem 9.1 we define how to pass the witness circulation $\boldsymbol{c}$ and length upper bounds $\boldsymbol{w}$ downwards in a tree-chain. It is convenient to define a *valid pair* of $\boldsymbol{c}, \boldsymbol{w}$ with respect to a graph G with lengths $\boldsymbol{\ell}$. Essentially, this means that $\boldsymbol{c}$ is indeed a circulation and $\boldsymbol{w}$ are valid length upper bounds, i.e. items 1 and 2 of the hidden stable-flow chasing property Definition 8.1.

Definition 9.2 (Valid pair)**.** *For a graph $G = (V, E)$ with lengths $\boldsymbol{\ell} \in \mathbb{R}^E_{>0}$, we say that $\boldsymbol{c}, \boldsymbol{w} \in \mathbb{R}^E$ are a* valid pair *if $\boldsymbol{c}$ is a circulation and $|\boldsymbol{\ell}_e \boldsymbol{c}_e| \le \boldsymbol{w}_e$ for all $e \in E$.*

9.1.1 Passing Circulations and Length Upper Bounds to the Core Graph. We first describe how to pass $\boldsymbol{c}, \boldsymbol{w}$ from G to a core graph $C(G, F)$ (Definition 8.7).

Definition 9.3 (Passing $\boldsymbol{c}, \boldsymbol{w}$ to core graph)**.** *Given a graph $G = (V, E)$ with a tree T, arooted spanning forest $E(F) \subseteq E(T)$, and a stretch overestimates $\widetilde{\mathrm{str}}_e$ as in Lemma 8.5, circulation $\boldsymbol{c} \in \mathbb{R}^E$ and length upper bounds $\boldsymbol{w} \in \mathbb{R}^E_{>0}$, we define vectors $\boldsymbol{c}^{C(G,F)} \in \mathbb{R}^{E(C(G,F))}$ and $\boldsymbol{w}^{C(G,F)} \in \mathbb{R}^{E(C(G,F))}_{>0}$ as follows. For $\widehat{e} \in E(C(G, F))$ with preimage $e \in E$, define $\boldsymbol{c}^{C(G,F)}_{\widehat{e}} \stackrel{\text{def}}{=} \boldsymbol{c}_e$ and $\boldsymbol{w}^{C(G,F)}_{\widehat{e}} \stackrel{\text{def}}{=} \widetilde{\mathrm{str}}_e \boldsymbol{w}_e$.*

We verify that $\boldsymbol{c}^{C(G,F)}$ is a circulation on $C(G, F)$ and that $\boldsymbol{w}^{C(G,F)}$ are length upper bounds.

Lemma 9.4 (Validity of Definition 9.3). *Let $\boldsymbol{c}, \boldsymbol{w}$ be a valid pair (Definition 9.2) on a graph G with lengths $\boldsymbol{\ell}$. As defined in Definition 9.3, $\boldsymbol{c}^{C(G,F)}, \boldsymbol{w}^{C(G,F)}$ are a valid pair on $C(G, F)$ with lengths $\boldsymbol{\ell}^{C(G,F)}$ (Definition 8.7), and*

$$\left\|\boldsymbol{w}^{C(G,F)}\right\|_1 \le \sum_{e \in E(G)} \widetilde{\mathrm{str}}_e \boldsymbol{w}_e.$$

Proof. The proof is primarily checking the definitions. Recall that the edge set of $C(G, F)$ is G/F. Contracting vertices preserves circulations, hence $\boldsymbol{c}^{C(G,F)}$ is a circulation (as $\boldsymbol{c}$ is).

In Definition 8.7 we define $\boldsymbol{\ell}^{C(G,F)}_{\widehat{e}} = \widetilde{\mathrm{str}}_e \boldsymbol{\ell}_e$. So

$$|\boldsymbol{\ell}^{C(G,F)}_{\widehat{e}} \boldsymbol{c}^{C(G,F)}_{\widehat{e}}| = \widetilde{\mathrm{str}}_e |\boldsymbol{\ell}_e \boldsymbol{c}_{\widehat{e}}| = \widetilde{\mathrm{str}}_e |\boldsymbol{\ell}_e \boldsymbol{c}_e| \le \widetilde{\mathrm{str}}_e \boldsymbol{w}_e = \boldsymbol{w}^{C(G,F)}_{\widehat{e}},$$

where the inequality holds because $\boldsymbol{c}, \boldsymbol{w}$ are a valid pair.

The bound on $\|\boldsymbol{w}^{C(G,F)}\|_1$ follows trivially by definition. The reason for the inequality (instead of equality) is that some edges may be contracted and disappear. □

Finally we state an algorithm which takes hidden stable-flow chasing updates on a dynamic graph $G^{(t)}$ and produces a dynamic core graph. Below, we let $\boldsymbol{c}^{(t),C(G,F)}, \boldsymbol{w}^{(t),C(G,F)}$ denote the result of using Definition 9.3 for $\boldsymbol{c} = \boldsymbol{c}^{(t)}$ and $\boldsymbol{w} = \boldsymbol{w}^{(t)}$, and similar definitions for $\boldsymbol{g}^{(t),C(G,F)}, \boldsymbol{\ell}^{(t),C(G,F)}$ used later in the section.

Lemma 9.5 (Dynamic Core Graphs). *Algorithm 3 takes as input a parameter k, a dynamic graph $G^{(t)}$ undergoes τ batches of updates $U^{(1)}, \ldots U^{(\tau)}$ with gradients $\boldsymbol{g}^{(t)}$, and lengths $\boldsymbol{\ell}^{(t)}$ at stage $t = 0, \ldots, \tau$ that satisfies $\sum_{t=1}^{\tau} \mathrm{Enc}(U^{(t)}) \le m/(k \log^2 n)$ and the hidden stable-flow chasing property with the hidden circulation $\boldsymbol{c}^{(t)}$ and width $\boldsymbol{w}^{(t)}$.*

For each $j \in [B]$ with $B = O(\log n)$, the algorithm maintains a static tree T_j, a decremental rooted forest $F_j^{(t)}$ with $O(m/k)$ components satisfying the conditions of Lemma 8.5, and a core graph $C(G^{(t)}, F_j^{(t)})$:

(1) *Core Graphs have Bounded Recourse: the algorithm outputs update batches $U_j^{(t)}$ that produce $C(G^{(t)}, F_j^{(t)})$ from $C(G^{(t-1)}, F_j^{(t-1)})$ such that $\sum_{t' \le t} |U_j^{(t')}| = O\left(\sum_{t' \le t} \textsc{Enc}(U^{(t')}) \cdot \log^2 n\right)$.*

(2) *The Widths on the Core Graphs are Small: whp. there is an $j^* \in [B]$ only depending on $\boldsymbol{w}^{(0)}$ such that for $\boldsymbol{w}^{(t), C(G^{(t)}, F_j^{(t)})}$ as defined in Definition 9.3 for $E(F_j) \subseteq E(T_j)$, and all stages $t \in \{0, \ldots, \tau\}$,*

$$\|\boldsymbol{w}^{(t), C(G^{(t)}, F_{j^*}^{(t)})}\|_1 \le O(\gamma_{LSST} \log^2 n)\|\boldsymbol{w}^{(0)}\|_1 + \|\boldsymbol{w}^{(t)}\|_1. \tag{26}$$

The algorithm runs $\widetilde{O}(mk)$-time.

Algorithm 3: Dynamically maintains a core graph (Definition 8.7). Procedure Initialize initializes all variables, and DynamicCore takes updates to $G^{(t)}$.

global variables
- $B \leftarrow O(\log n)$: number of instantiations of Lemma 8.5
- $\mathcal{A}_j^{(LSD)}$ for $j \in [B]$: algorithms implementing Lemma 8.5
- T_j for $j \in [B]$: trees initialized by $\mathcal{A}_j^{(LSD)}$
- $\widetilde{\text{str}}^j$ for $j \in [B]$: stretch overestimates initialized by $\mathcal{A}_j^{(LSD)}$.

procedure *Initialize*$(G = G^{(0)}, \boldsymbol{\ell}, k)$
- Let $\boldsymbol{\lambda}$ and $\{T_1', \ldots, T_t'\}$ be returned by Lemma 8.6 on G with lengths $\boldsymbol{\ell}$ and $t = \widetilde{O}(k)$
- For $j \in [B]$ sample $i_j \in [t]$ proportional to $\boldsymbol{\lambda}$, and $T_j \leftarrow T_{i_j}'$
- Initialize $\mathcal{A}_j^{(LSD)}$ on T_j for $j \in [B]$

procedure *DynamicCore*$(G^{(t)}, U^{(t)}, \boldsymbol{g}^{(t)}, \boldsymbol{\ell}^{(t)})$
- **for** $j \in [B]$ **do**
  - Pass $U^{(t)}$ to $\mathcal{A}_j^{(LSD)}$ which updates $F_j^{(t-1)}$ to $F_j^{(t)}$.
  - `// All edges` $e \in G^{(t)} \cap \left(\bigcup_{i \le t} U^{(i)}\right)$ `have` $\widetilde{\text{str}}_e^j = 1$ `by Lemma 8.5`
  - Let $U_j^{(t)}$ be the batch of vertex splits that updates $C(G^{(t-1)}, F_j^{(t-1)})$ to $C(G^{(t-1)}, F_j^{(t)})$.
  - Append $U_j^{(t)}$ with $U^{(t)}$ which updates $C(G^{(t-1)}, F_j^{(t)})$ to $C(G^{(t)}, F_j^{(t)})$.

Algorithm 3 initializes B trees $T_1, \ldots, T_B$ from the MWU distribution output by Lemma 8.6. For each of these B trees, we maintain a forest $E(F_j^{(t)}) \subseteq E(T_j)$ satisfying the conditions of Lemma 8.5 with the goal of forcing the stretch of every newly appeared edge e in $G^{(t)}$ to be 1, i.e. $\widetilde{\text{str}}_e^j = 1$ for all $j \in [B]$.

Given an update batch $U^{(t)}$, Algorithm 3 first updates the forest $F_j^{(t-1)}$ to $F_j^{(t)}$ for any $j \in [B]$ using the algorithm of Lemma 8.5. For any update $x \in U^{(t)}$, if x updates some edge e, both endpoints are roots in the forest $F_j^{(t)}$ and they appear in the core graph. If x splits some vertex u, u is made a root in the forest and it appears in the core graph as well. In both cases, the update x can be performed in the core graph $C(G^{(t-1)}, F_j^{(t)})$ (notice that it is not $C(G^{(t)}, F_j^{(t)})$). Thus, we can apply the entire batch $U^{(t)}$ to produce $C(G^{(t)}, F_j^{(t)})$ from $C(G^{(t-1)}, F_j^{(t)})$.

When a vertex $u \in V(G^{(t-1)})$ is split, the algorithm of Lemma 8.5 treats it as a sequence of one isolated vertex insertion and $O(\deg_{G^{(t)}}(u^{NEW})) = O(\textsc{Enc}(x))$ edge insertions/deletions. The newly added isolated vertex stays isolated in the forests $F_j, j \in [B]$ as they are maintained decrementally edge-wise.

We adapt the reduction when applying $U^{(t)}$ to produce $C(G^{(t)}, F_j^{(t)})$ from $C(G^{(t-1)}, F_j^{(t)})$. Thus, the number of updates in the core graph is at least the *total encoding size* of updates in the original graph. As we will show, it is upper-bounded by the total encoding size as well.

Proof of Lemma 9.5. Note that $\sum_{t\in[\tau]} \textsc{Enc}(U^{(t)}) = m/(k\log^2 n)$, we can take $q = O(m/(k\log^2 n))$ in Lemma 8.5. Thus by item 1 of Lemma 8.5, $F_j^{(t)}$ has $O(m/k)$ connected components.

Next, we prove Item 1 that bounds the number of updates to the core graph. After t batches of updates $U^{(1)}, \ldots, U^{(t)}$, $\mathcal{A}_j^{(LSD)}$ increases the number of components in F_j by $O\left(\sum_{t'\le t} \textsc{Enc}(U^{(t')}) \cdot \log^2 n\right)$ according to Item 1 of Lemma 8.5. Every new components appeared in F_j splits a vertex in the core graph $C(G, F_j)$. Thus, there will be $O\left(\sum_{t'\le t} \textsc{Enc}(U^{(t')}) \cdot \log^2 n\right)$ vertex splits happened in the core graph. After updating F_j, every update batch $U^{(t')}$ updates $C(G, F_j)$ as it updates G. Thus, we can bound the number of updates to $C(G, F_j)$ up to first t stages by $O\left(\sum_{t'\le t} \textsc{Enc}(U^{(t')}) \cdot \log^2 n\right)$.

To show Item 2, by Lemma 9.4 we first get that

$$\left\|\boldsymbol{w}^{(t),C\left(G^{(t)},F_j^{(t)}\right)}\right\|_1 \le \sum_{e\in E(G^{(t)})} \widetilde{\mathrm{str}}_e^j \boldsymbol{w}_e^{(t)} \overset{(i)}{\le} \sum_{e\in G^{(t)}\setminus\left(\bigcup_{s=1}^t U^{(s)}\right)} \widetilde{\mathrm{str}}_e^j \boldsymbol{w}_e^{(t)} + \sum_{e\in G^{(t)}\cap\left(\bigcup_{s=1}^t U^{(s)}\right)} \boldsymbol{w}_e^{(t)}$$
$$\overset{(ii)}{\le} 2\sum_{e\in G^{(0)}} \widetilde{\mathrm{str}}_e^j \boldsymbol{w}_e^{(0)} + \left\|\boldsymbol{w}^{(t)}\right\|_1, \tag{27}$$

where (i) follows because every edge e appeared in $G^{(t)}$ due to some update in some $U^{(i)}$ has $\widetilde{\mathrm{str}}_e^j = 1$ by a condition of Lemma 8.5. (ii) follows because the hidden stable-flow chasing property (Definition 8.1 item 3) gives that any edge $e \in G^{(t)} \setminus \bigcup_{s=1}^t U^{(s)} \subseteq G^{(0)}$ has $\boldsymbol{w}_e^{(t)} \le 2\boldsymbol{w}_e^{(0)}$.

Now recall that T_j is sampled from the collection $\{T_1', \ldots, T_t'\}$ of trees given by Lemma 8.6, with probabilities proportional to $\boldsymbol{\lambda}$. Hence

$$\mathbb{E}_{T_j}\left[\sum_{e\in G^{(0)}} \widetilde{\mathrm{str}}_e^j \boldsymbol{w}_e^{(0)}\right] = \sum_{e\in G^{(0)}} \boldsymbol{w}_e^{(0)} \sum_{i=j}^t \boldsymbol{\lambda}_j \widetilde{\mathrm{str}}_e^j \le O(\gamma_{LSST}\log^2 n)\|\boldsymbol{w}^{(0)}\|_1,$$

by the guarantees of Lemma 8.6, so by Markov's inequality

$$\Pr_{T_j}\left[\sum_{e\in G^{(0)}} \widetilde{\mathrm{str}}_e^j \boldsymbol{w}_e^{(0)} \le O(\gamma_{LSST}\log^2 n)\|\boldsymbol{w}^{(0)}\|_1\right] \ge 1/2.$$

Since we sample B independent trees T, for $B = \Theta(\log n)$, we get that there exists an i^* satisfying (26) with probability at least $1 - 2^{-B} \ge 1 - n^{-\Theta(1)}$.

Finally, the algorithm runs in total time $\widetilde{O}(mk)$ by Lemma 8.6 and Lemma 8.5. □

9.1.2 Passing Circulations and Length Upper Bounds to the Sparsified Core Graph. We describe how to pass $\boldsymbol{c}^{C(G,F)}, \boldsymbol{w}^{C(G,F)}$ on a core graph to a sparsified core graph $\mathcal{S}(G, F)$.

Definition 9.6 (Passing $\boldsymbol{c},\boldsymbol{w}$ to sparsified core graph)**.** *Consider a graph G with spanning forest F, and circulation $\boldsymbol{c}^{C(G,F)} \in \mathbb{R}^{E(C(G,F))}$ and upper bound $\boldsymbol{w}^{C(G,F)} \in \mathbb{R}^{E(C(G,F))}_{>0}$, and embedding $\Pi_{C(G,F)\to\mathcal{S}(G,F)}$ for a $(\gamma_s,\gamma_c,\gamma_l)$-sparsified core graph $\mathcal{S}(G,F) \subseteq C(G,F)$. Define*

$$\boldsymbol{c}^{\mathcal{S}(G,F)} = \sum_{\widehat{e}\in E(C(G,F))} \boldsymbol{c}^{C(G,F)}_{\widehat{e}} \boldsymbol{\Pi}_{C(G,F)\to\mathcal{S}(G,F)}(\widehat{e}) \tag{28}$$

$$\boldsymbol{w}^{\mathcal{S}(G,F)} = 2 \sum_{\widehat{e}\in E(C(G,F))} \boldsymbol{w}^{C(G,F)}_{\widehat{e}} \left|\boldsymbol{\Pi}_{C(G,F)\to\mathcal{S}(G,F)}(\widehat{e})\right|. \tag{29}$$

We check that $\boldsymbol{c}^{\mathcal{S}(G,F)}$ is a circulation on $\mathcal{S}(G,F)$ and $\boldsymbol{w}^{\mathcal{S}(G,F)}$ are length upper bounds.

Lemma 9.7 (Validity of Definition 9.6). *Let $\boldsymbol{c}^{C(G,F)},\boldsymbol{w}^{C(G,F)}$ be a valid pair on graph $C(G,F)$ with lengths $\boldsymbol{\ell}^{C(G,F)}$. As defined in Definition 9.6, $\boldsymbol{c}^{\mathcal{S}(G,F)},\boldsymbol{w}^{\mathcal{S}(G,F)}$ is a valid pair on $\mathcal{S}(G,F)$ with lengths $\boldsymbol{\ell}^{\mathcal{S}(G,F)}$ (Definition 8.9). Also,*

$$\|\boldsymbol{w}^{C(G,F)}\|_1 \le \|\boldsymbol{w}^{\mathcal{S}(G,F)}\|_1 \le O(\gamma_l)\|\boldsymbol{w}^{C(G,F)}\|_1.$$

Proof. Let $\boldsymbol{B}_{\mathcal{S}},\boldsymbol{B}_C$ be the incidence matrices of $\mathcal{S}(G,F), C(G,F)$ respectively. To see that $\boldsymbol{c}^{\mathcal{S}(G,F)}$ is a circulation, we write

$$\boldsymbol{B}_{\mathcal{S}}^\top \boldsymbol{c}^{\mathcal{S}(G,F)} = \sum_{\widehat{e}\in E(C(G,F))} \boldsymbol{c}^{\mathcal{S}(G,F)}_{\widehat{e}} \boldsymbol{B}_{\mathcal{S}}^\top \boldsymbol{\Pi}_{C(G,F)\to\mathcal{S}(G,F)}(\widehat{e}) = \sum_{\widehat{e}\in E(C(G,F))} \boldsymbol{c}^{\mathcal{S}(G,F)}_{\widehat{e}} \boldsymbol{b}_{\widehat{e}} = \boldsymbol{B}_C^\top \boldsymbol{c}^{C(G,F)} = 0.$$

To see that $\boldsymbol{w}^{\mathcal{S}(G,F)}$ are valid upper bounds, for all $\widehat{e}' \in E(\mathcal{S}(G,F))$

$$\begin{aligned}|\boldsymbol{\ell}^{\mathcal{S}(G,F)}_{\widehat{e}'} \boldsymbol{c}^{\mathcal{S}(G,F)}_{\widehat{e}'}| &= \boldsymbol{\ell}^{\mathcal{S}(G,F)}_{\widehat{e}'} \Big| \sum_{\widehat{e}:\widehat{e}'\in\Pi_{C(G,F)\to\mathcal{S}(G,F)}(\widehat{e})} \boldsymbol{c}^{C(G,F)}_{\widehat{e}} \Big| \overset{(i)}{\le} 2 \sum_{\widehat{e}:\widehat{e}'\in\Pi_{C(G,F)\to\mathcal{S}(G,F)}(\widehat{e})} |\boldsymbol{\ell}^{\mathcal{S}(G,F)}_{\widehat{e}} \boldsymbol{c}^{C(G,F)}_{\widehat{e}}| \\ &\le 2 \sum_{\widehat{e}:\widehat{e}'\in\Pi_{C(G,F)\to\mathcal{S}(G,F)}(\widehat{e})} \boldsymbol{w}^{C(G,F)}_{\widehat{e}} = 2\boldsymbol{w}^{\mathcal{S}(G,F)}_{\widehat{e}'}.\end{aligned}$$

Throughout, we used several properties guaranteed in Definition 8.9, and (i) specifically follows by item 1. The final equality follows by the definition of $\boldsymbol{w}^{\mathcal{S}(G,F)}$ in (29).

Finally we upper-bound $\|\boldsymbol{w}^{\mathcal{S}(G,F)}\|_1$ by

$$\begin{aligned}\|\boldsymbol{w}^{\mathcal{S}(G,F)}\|_1 &\le 2 \sum_{\widehat{e}\in E(C(G,F))} \boldsymbol{w}^{C(G,F)}_{\widehat{e}} \|\boldsymbol{\Pi}_{C(G,F)\to\mathcal{S}(G,F)}(\widehat{e})\|_1 \\ &\le 2\|\boldsymbol{w}^{C(G,F)}\|_1 \mathrm{length}(\Pi_{C(G,F)\to\mathcal{S}(G,F)}) \le O(\gamma_l)\|\boldsymbol{w}^{C(G,F)}\|_1,\end{aligned}$$

because $\mathcal{S}(G,F)$ is a $(\gamma_s,\gamma_c,\gamma_l)$ sparsified core graph. $\|\boldsymbol{w}^{C(G,F)}\|_1 \le \|\boldsymbol{w}^{\mathcal{S}(G,F)}\|_1$ follows directly from the definition. □

We can now give an algorithm that takes a dynamic graph $G^{(t)}$ undergoing hidden stable-flow chasing updates, and maintain a sparsified core graph also undergoing hidden stable-flow chasing updates, such that the total size of updates increases by a factor of at most $m^{o(1)}$. This shows how to pass from level i to $i+1$ in a tree-chain (Definition 8.10).

Lemma 9.8 (Dynamic Sparsified Core Graphs). *Algorithm 4 takes as input a parameter k, a dynamic graph $G^{(t)}$ undergoes τ batches of updates $U^{(1)},\dots,U^{(\tau)}$ with gradients $\boldsymbol{g}^{(t)}$, lengths $\boldsymbol{\ell}^{(t)}$ at stage $t = 0,\dots,\tau$ that satisfies $\sum_{t=1}^{\tau} \mathrm{Enc}(U^{(t)}) \le m/(k\log^2 n)$ and the hidden stable-flow chasing property with the hidden circulation $\boldsymbol{c}^{(t)}$ and width $\boldsymbol{w}^{(t)}$.*

The algorithm maintains for each $j \in [B]$ (for $B = O(\log n)$), a decremental forest $F_j^{(t)}$, a static tree T_j satisfying the conditions of Lemma 8.5, and a $(\gamma_s, \gamma_l, \gamma_c)$-sparsified core graph $\mathcal{SC}(G^{(t)}, F_j^{(t)})$ for parameters $\gamma_s = \gamma_c = \gamma_l = \exp(O(\log^{3/4} m \log\log m))$ with embedding $\Pi_{C(G^{(t)},F_j^{(t)})\to\mathcal{S}(G^{(t)},F_j^{(t)})}$:

(1) *Sparsified Core Graphs have Low Recourse: the algorithm outputs update batches $U_{\mathcal{S},j}^{(t)}$ that produce $\mathcal{S}(G^{(t)}, F_j^{(t)})$ from $\mathcal{S}(G^{(t-1)}, F_j^{(t-1)})$ such that $\sum_{t'\le t} \textsc{Enc}(U_{\mathcal{S},j}^{(t')}) = \gamma_r \cdot \sum_{t'\le t} \textsc{Enc}(U^{(t')})$ for some $\gamma_r = \exp(O(\log^{3/4} m \log\log m))$,*

(2) *Sparsified Core Graphs undergo Hidden Stable-Flow Chasing Updates: for each $j \in [B]$, the update batches $U_{\mathcal{S},j}^{(t)}$ to the sparsified core graph along with the associated gradients $\boldsymbol{g}^{(t),\mathcal{S}(G^{(t)}F_j^{(t)})}$, and lengths $\boldsymbol{\ell}^{(t),\mathcal{S}(G^{(t)},F_j^{(t)})}$ as defined in Definition 8.9 satisfy the hidden stable-flow chasing property (see Definition 8.1) with the hidden circulation $\boldsymbol{c}^{(t),\mathcal{S}(G^{(t)},F_j^{(t)})}$, and width $\boldsymbol{w}^{(t),\mathcal{S}(G^{(t)},F_j^{(t)})}$ as defined in Definition 9.6, and*

(3) *The Widths on the Sparsified Core Graphs are Small: for each $j \in [B]$, the width on the sparsified core graph $\mathcal{SC}(G^{(t)}F_j^{(t)})$ is bounded as follows:*

$$\left\|\boldsymbol{w}^{(t),\mathcal{SC}(G^{(t)},F_j^{(t)})}\right\|_1 \le \widetilde{O}(\gamma_l)\left(\|\boldsymbol{w}^{(0),C(G^{(0)},F_j^{(0)})}\|_1 + \left\|\boldsymbol{w}^{(t)}\right\|_1\right).$$

Also, whp. there is an $j^* \in [B]$ only depending on $\boldsymbol{w}^{(0)}$ such that

$$\left\|\boldsymbol{w}^{(t),\mathcal{S}(G^{(t)},F_{j^*}^{(t)})}\right\|_1 \le \widetilde{O}(\gamma_l)\left(\left\|\boldsymbol{w}^{(0)}\right\|_1 + \left\|\boldsymbol{w}^{(t)}\right\|_1\right). \tag{30}$$

The algorithm runs in total time $\widetilde{O}(mk \cdot \gamma_r)$.

Algorithm 4 essentially maintains the sparsified core graphs $\mathcal{S}(G^{(t)}, F_j^{(t)})$ by passing the core graphs $C(G^{(t)}, F_j^{(t)})$ into the dynamic spanner Theorem 7.1. Intuitively, because $F_j^{(t)}$ is decremental, the graph $C(G^{(t)}, F_j^{(t)})$ changes by undergoing vertex splits, plus additional edge insertions and deletions induced by the update batch $U^{(t)}$.

Similar to Lemma 8.5, Algorithm 4 treats each update batch $U^{(t)}$ to G as $O(\textsc{Enc}(U^{(t)}))$ edge insertions/deletions and isolated vertex insertions. In particular, for any update $x \in U^{(t)}$ that splits a vertex $u \in G^{(t-1)}$, it is treated as an update sequence of inserting one isolated vertex u^{NEW} and then deleting/inserting $\deg_{G^{(t)}}(u^{NEW})$ edges.

However, each edge insertion/deletion causes $\widetilde{O}(1)$ vertex splits in the core graph $C(G^{(t)}, F_j^{(t)})$. As vertices in $C(G^{(t)}, F_j^{(t)})$ could have degree $\Omega(k)$, we cannot afford treating vertex splits in the core graph as a sequence of edge insertions/deletions. This would represent $\mathcal{SC}(G^{(t)}, F_j^{(t)})$ using updates of total encoding size $O(k \cdot \sum_t \textsc{Enc}(U^{(t)})) = O(m)$ instead of $O(m^{1+o(1)}/k)$. Using the dynamic spanner of Theorem 7.1 resolves the issue as it handles vertex splits with low recourse. In particular, $\mathcal{SC}(G^{(t)}, F_j^{(t)})$ can be represented using a sequence of updates with total encoding size $O(m^{o(1)} \cdot \sum_t \textsc{Enc}(U^{(t)}))$.

Formalizing this approach requires discussion of several technical points. First, we cannot simply maintain the spanner of $C(G^{(t)}, F_j^{(t)})$ using Theorem 7.1 which does not support edge insertions. Instead of modifying the dynamic spanner algorithm, we deal with edge insertions naïvely by inserting each of them to $\mathcal{SC}(G^{(t)}, F_j^{(t)})$. As the total number of edge insertion is at most $\sum_{t\in[\tau]} \textsc{Enc}(U^{(t)}) = o(m/k)$, $\mathcal{SC}(G^{(t)}, F_j^{(t)})$ is still sparse enough.

Second, vertices in core graphs $C(G^{(t)}, F_j^{(t)})$, $j \in [B]$ might have degree $\Omega(k)$. To ensure a maximum degree bound of $\widetilde{O}(k)$ which is required by Theorem 7.1, we artificially split vertices in $C(G^{(t)}, F_j^{(t)})$ to create a graph $\widetilde{C}_j^{(t)}$ on which we maintain the spanner. Precisely, we create a new vertex u_W in $\widetilde{C}_j^{(0)}$ for each piece W in the partition $\mathcal{W}$ of the

Algorithm 4: Dynamically maintains a sparsified core graph (Definition 8.7). Procedure INITIALIZE initializes all variables, and DYNAMICSPARSECORE takes updates to $G^{(t)}$.

global variables
  $B \leftarrow O(\log n)$: number of instantiations of Lemma 8.5 in Lemma 9.5
  $\mathcal{A}^{(Core)}$: algorithm implementing Lemma 9.5
  $\mathcal{A}_j^{(Spanner)}$ for $j \in [B]$: algorithms implementing Theorem 7.1
procedure *INITIALIZE*$(G = G^{(0)}, \boldsymbol{\ell}, k)$
  $\mathcal{A}^{(Core)}$.INITIALIZE$(G, \boldsymbol{\ell}, k)$
  for $j \in [B]$ **do**
    Let $\mathcal{W}$ be the partition in Lemma 8.5 item 4, and $R \supseteq \partial\mathcal{W}$ an initial set of roots obtained from running the algorithm in Lemma 8.5.
    Create graph $\widetilde{C}_j$ by splitting vertices of $C(G, F_j)$ into vertices u_r for each $r \in R$, and a vertex u_W for the set of vertices $W \setminus R$ for each $W \in \mathcal{W}$. `// The vertices` u_r `will not be split further, and` u_W `all have degree at most` $\widetilde{O}(k)$`. Also, a deletion to` F_j `will only split a single vertex.`
    Let $\Lambda_j \stackrel{\text{def}}{=} \Lambda_{\widetilde{C}_j \to C(G,F_j)}$ be the bijection between $E\left(\widetilde{C}_j\right)$ and $E\left(C(G, F_j)\right)$.
    Initialize $\mathcal{A}_j^{(Spanner)}$ on $\widetilde{C}_j$, the split version of $C(G, F_j)$.
    Let $\widetilde{\mathcal{SC}}_j$ be the spanner maintained by $\mathcal{A}_j^{(Spanner)}$ and $\mathcal{SC}(G, F_j) \stackrel{\text{def}}{=} \Lambda_j(\widetilde{\mathcal{SC}}_j)$.
procedure *DYNAMICSPARSECORE*$(G^{(t)}, U^{(t)}, \boldsymbol{g}^{(t)}, \boldsymbol{\ell}^{(t)})$
  $\mathcal{A}^{(Core)}$.DYNAMICCORE$(G^{(t)}, U^{(t)}, \boldsymbol{g}^{(t)}, \boldsymbol{\ell}^{(t)})$
  for $j \in [B]$ **do**
    Let $U_j^{(t)}$ be the update batch that produce $C(G^{(t)}, F_j^{(t)})$ from $C(G^{(t-1)}, F_j^{(t-1)})$.
    Let $U_j^{(t)+} \subseteq U_j^{(t)}$ contain all edge insertions.
    Let $U_j^{(t)-} \subseteq U_j^{(t)}$ contain the rest.
    Update $\widetilde{\mathcal{SC}}_j^{(t-1)}$ to $\widetilde{\mathcal{SC}}_j^{(t-0.5)}$ with $\Lambda_j^{-1}(U_j^{(t)-})$ using $\mathcal{A}_j^{(Spanner)}$.
    Update $\widetilde{\mathcal{SC}}_j^{(t-0.5)}$ to $\widetilde{\mathcal{SC}}_j^{(t)}$ via inserting edges of $\Lambda_j^{-1}(U_j^{(t)+})$ directly.
    Let $R_j^{(t)} \subseteq E(\widetilde{\mathcal{SC}}_j^{(t)})$ be the re-embedded set output by $\mathcal{A}_j^{(Spanner)}$.
    Let $U_{\mathcal{SC},j}^{(t)}$ be the corresponding update batch that produce $\mathcal{SC}(G^{(t)}, F_j^{(t)})$ from $\mathcal{SC}(G^{(t-1)}, F_j^{(t-1)})$.
    Append $U_{\mathcal{SC},j}^{(t)}$ with $\Lambda_j(R_j)$ and output $U_{\mathcal{SC},j}^{(t)}$. `// Despite edges in` $\Lambda_j(R_j)$ `remain unchanged in` $\mathcal{SC}(G^{(t)}, F_j^{(t)})$`, we force re-insertions on them in the output batch of updates.`

forest $F_j^{(0)}$, and a vertex u_r for each root in the initial forest $F_j^{(0)}$. Throughout the execution, we ensure that every vertex of $\widetilde{C}_j^{(t)}$ is either u_r for some r being a root in the current forest, or u_X for some connected component $X \subseteq W$ of an initial piece $W \in \mathcal{W}$. In the former case, u_r corresponds to a single vertex in the original graph $G^{(t)}$ and thus it is never split due to edge removals from the forest $F_j^{(t)}$. In the later case, u_X corresponds to the set of vertices $X \setminus R$ and thus its degree is bounded by $\widetilde{O}(k)$ due to 4 of Lemma 8.5.

Proof of Lemma 9.8. We first argue that the graph $\widetilde{C}_j$ for all $j \in [B]$ has maximum degree $\widetilde{O}(k)$ and $O(m/k)$ vertices, and undergoes a total of $O(m/k)$ vertex splits, and edge insertions/deletions. This shows that the application of the dynamic spanner algorithm in Theorem 7.1 is efficient.

For any $j \in [B]$, each vertex of $\widetilde{C}_j$ is either u_r for some root $r \in R$ or u_X for some connected component X of an initial piece $W \in \mathcal{W}$. In the case of u_r, it will not be split further. In the case of u_X, it corresponds to the set of vertices $X \setminus R$. Since X is a connected component in F of an initial piece $W \in \mathcal{W}$, the degree of u_X is at most $\deg_G(W \setminus R)$ which is $\widetilde{O}(k)$ due to 4 of Lemma 8.5.

The data structure implementing Lemma 8.5 inside $\mathcal{A}^{(Core)}$ ensures that F_j is decremental. Edge deletions in F_j does not affect either gradients nor lengths of edges in $C(G, F_j)$ (Definition 8.7). Thus, one edge deletion in F_j corresponds to only a single vertex split in $C(G, F_j)$. The total number of vertex splits happened to $C(G, F_j)$ can be bounded by the number of edge removals in F_j. The number is $O(m/k + q \log^2 n) = O(m/k)$ for $q \stackrel{\text{def}}{=} \sum_{t=1}^{\tau} \textsc{Enc}(U^{(t)}) \le m/(k \log^2 n)$ by item 1 of Lemma 8.5. Similarly, each edge deletion to F_j causes one vertex split in $\widetilde{C}_j$. To see this, first note that no root vertices $u_r \in \widetilde{C}_j$ are ever split. For the deletion of an edge e to $F_j^{(t)}$, let $W \in \mathcal{W}$ be the partition piece containing e. The vertex u_W may have been split further already, so let e be currently inside the connected component $X \subseteq W$. Now, because $\partial W \subseteq R$ at all times, we get that only u_X was split in $\widetilde{C}_j$, as desired.

After updating $F_j^{(t)}$ and the enlarged vertex set of $C(G^{(t)}, F_j^{(t)})$, we process every update of $\bigcup_t U^{(t)}$ naively as $O(q)$ edge updates. As each edge update to $C(G^{(t)}, F_j)$ corresponds to one edge update to $\widetilde{C}_j$, the number of edge updates happened to $\widetilde{C}_j$ is also $O(q) = O(m/(k \log^2 n))$. It remains to bound the initial number of vertices in $\widetilde{C}_j$ by $O(m/k)$. As noted in Algorithm 4, there is one vertex per root of the initial forest F_j and one vertex per cluster of the partition $\mathcal{W}$. The number of roots initially is $O(m/k)$ (Lemma 8.5). The number of clusters in $\mathcal{W}$ is also $O(m/k)$ (Lemma 8.5). Thus, $\widetilde{C}_j$ has $O(m/k)$ vertices initially.

As noted in Algorithm 4, it is at most twice the initial number of roots in F_j which is $O(m/k)$.

Bounding the total size of $U_{SC,j}^{(t)}$ (Item 1): Fix some $j \in [B]$. As discussed above, processing all updates in the data structure of Lemma 8.5 causes at most $O(m/k + \sum_t \textsc{Enc}(U^{(t)}) \log^2 m) = O(m/k)$ vertex splits to $\widetilde{C}_j$. So, the graph $\widetilde{C}_j$ undergoes at most $O(m/k)$ vertex splits and edge insertions/deletions. By item 3 of Theorem 7.1, the data structure $\mathcal{A}_i^{(Spanner)}$ outputs the re-embedded set $R_j^{(t)}$ of amortized size at most γ_r, by taking $L = (\log m)^{1/4}$ in Theorem 7.1. Thus, the total size of re-embedded edges $\sum_t |R_j^{(t)}|$ is bounded by $O(m\gamma_r/k)$. Similarly, Theorem 7.1 also shows that $\mathcal{S}(G^{(t)}, F_j^{(t)})$ are $(\gamma_s, \gamma_c, \gamma_l)$ sparsified core graphs with the embeddings $\Pi_j^{(t)}$.

Now, we move towards checking the remaining conditions: showing the hidden stable-flow chasing property of the outputs on $\mathcal{S}(G^{(t)}, F_j^{(t)})$ for all $j \in [B]$, and (30). For simplicity, we use $\Pi_j^{(t)}$ to denote the embedding $\Pi_{C(G^{(t)}, F_j^{(t)}) \to \mathcal{SC}(G^{(t)}, F_j^{(t)})}$ throughout the remainder of this proof.

Showing hidden stable-flow chasing property (Item 2): $\boldsymbol{c}^{(t), \mathcal{S}(G^{(t)}, F_j^{(t)})}$ and $\boldsymbol{w}^{(t), \mathcal{SC}(G^{(t)}, F_j^{(t)})}$ form a valid pair by Lemma 9.7. Therefore, items 1 and 2 of Definition 8.1 are satisfied.

Next, we prove item 3 of Definition 8.1. At any stage $t \in [\tau]$ and any edge $e \in \mathcal{SC}(G^{(t)}, F_j^{(t)})$ for some $j \in [B]$, suppose e also appears in an earlier stage t', i.e. $e \in \mathcal{SC}(G^{(t')}, F_j^{(t')})$ for some $t' < t$. e is not included in any of $U_{\mathcal{SC},j}^{(s)}, s \in (t', t]$. Thus, we have $(\Pi_j^{(t)})^{-1}(e) \subseteq (\Pi_j^{(t')})^{-1}(e)$ otherwise e is included in some $U_{\mathcal{SC},j}^{(s)}, s \in (t', t]$ due to the definition of re-embedded set (Item 3 of Theorem 7.1).

For any edge $e' \in (\Pi_j^{(t)})^{-1}(e)$, it exists in the core graph at both stage t and t', i.e. $e' \in C(G^{(t)}, F_j^{(t)})$ and $C(G^{(t')}, F_j^{(t')})$. Let e'^G be its pre-image in G. e'^G also exists in G at both stage t and t'. Since G is undergoing hidden stable-flow chasing updates, by item 3 of Definition 8.1 we have

$$\boldsymbol{w}_{e'^G}^{(t),G^{(t)}} \leq 2 \cdot \boldsymbol{w}_{e'^G}^{(t'),G^{(t')}}.$$

Definition 9.3 and the immutable nature of $\widetilde{\mathrm{str}}$ from Lemma 8.5 yields

$$\boldsymbol{w}_{e'}^{(t),C(G^{(t)},F_j^{(t)})} = \widetilde{\mathrm{str}}_{e'^G}^{T_j,\boldsymbol{\ell}} \boldsymbol{w}_{e'^G}^{(t),G^{(t)}} \leq 2 \cdot \widetilde{\mathrm{str}}_{e'^G}^{T_j,\boldsymbol{\ell}} \boldsymbol{w}_{e'^G}^{(t'),G^{(t')}} = 2 \cdot \boldsymbol{w}_{e'}^{(t'),C(G^{(t')},F_j^{(t')})}. \tag{31}$$

Combining with the fact that $(\Pi_j^{(t)})^{-1}(e) \subseteq (\Pi_j^{(t')})^{-1}(e)$ and Definition 9.6 yields the following and proves item 3 of Definition 8.1:

$$\begin{aligned}
\boldsymbol{w}_e^{(t),\mathcal{SC}(G^{(t)},F_j^{(t)})} &= 2 \cdot \sum_{e' \in \left(\Pi_j^{(t)}\right)^{-1}(e)} \boldsymbol{w}_{e'}^{(t),C(G^{(t)},F_j^{(t)})} \\
&\leq 2 \cdot 2 \cdot \sum_{e' \in \left(\Pi_j^{(t)}\right)^{-1}(e)} \boldsymbol{w}_{e'}^{(t'),C(G^{(t')},F_i^{(t')})} \\
&\leq 2 \cdot 2 \cdot \sum_{e' \in \left(\Pi_j^{(t')}\right)^{-1}(e)} \boldsymbol{w}_e^{(t'),C(G^{(t')},F_j^{(t')})} = 2 \cdot \boldsymbol{w}_e^{(t'),\mathcal{SC}(G^{(t')},F_i^{(t')})}.
\end{aligned}$$

Item 4 follows directly from the definition of $\boldsymbol{\ell}^{(t),\mathcal{SC}(G^{(t)},F_j^{(t)})}$ and $\boldsymbol{w}^{(t),\mathcal{SC}(G^{(t)},F_j^{(t)})}$.

Upper-bounding $\|\boldsymbol{w}^{(t),\mathcal{SC}(G^{(t)},F_{i^*}^{(t)})}\|_1$ *(Item 3):* For any i, Lemma 9.7 yields

$$\left\|\boldsymbol{w}^{(t),\mathcal{S}(G^{(t)},F_i^{(t)})}\right\|_1 \leq O(\gamma_l) \left\|\boldsymbol{w}^{(t),C(G^{(t)},F_i^{(t)})}\right\|_1.$$

Lemma 9.5 gives that there is an $i^* \in [B]$ such that for all t,

$$\left\|\boldsymbol{w}^{(t),C(G^{(t)},F_{i^*}^{(t)})}\right\|_1 \leq \widetilde{O}(\gamma_{LSST} \log^2 n)\|\boldsymbol{w}^{(0)}\|_1 + \|\boldsymbol{w}^{(t)}\|_1.$$

Combining these gives the desired bound.

Runtime: Time spent on the data structure implementing Lemma 8.5 is $\widetilde{O}(mk)$. Before using the dynamic spanner of Theorem 7.1, we split each $C(G, F_j), j \in [B]$ in Line 9. This makes the max degree of the input graph to each of $\mathcal{A}_j^{(Spanner)}, j \in [B]$ being $\Theta(k)$. By Lemma 9.5 none of these vertices is split in an update to $C(G, F_j)$, so we may still apply Theorem 7.1. Thus, the time spent on every dynamic spanner is $O(mk\gamma_r)$. □

9.1.3 Maintaining a Branching Tree-Chain. Note that definitions Definitions 9.3 and 9.6 give a way to pass $\boldsymbol{c}^{(t)}, \boldsymbol{w}^{(t)}$ from the top level graph G downwards through a tree-chain (Definition 8.10). We formalize this by proving that we can dynamically maintain a branching tree-chain (Definition 8.10).

Lemma 9.9 (Dynamic Branching Tree-Chain). *Algorithm 5 takes as input a parameter d, a dynamic graph $G^{(t)}$ undergoes τ batches of updates $U^{(1)}, \ldots, U^{(\tau)}$ with gradients $\boldsymbol{g}^{(t)}$, length $\boldsymbol{\ell}^{(t)}$ at stage $t = 0, \ldots, \tau$ that satisfies the hidden stable-flow chasing property (Definition 8.1) with hidden circulation $\boldsymbol{c}^{(t)}$, and width $\boldsymbol{w}^{(t)}$. The algorithm explicitly maintains a B-branching tree-chain (Definition 8.10) with previous rebuild times* $\mathrm{prev}_0^{(t)}, \ldots, \mathrm{prev}_d^{(t)}$ *(Definition 8.12). If* $\boldsymbol{c}^{(t),G}, \boldsymbol{w}^{(t),G}$

Algorithm 5: Dynamically maintains a B-branching tree chain (Definition 8.10). Procedure Initialize initializes all variables, Rebuild rebuilds the data structure of level at least d_s at stage t_0, and DynamicBranchingChain takes updates to $G^{(t)}$.

global variables
  $d \leftarrow \log^{1/8} n$: number of levels in the maintained branching tree chain.
  $k \leftarrow m^{1/d}$: reduction factor used in Lemma 8.5.
  $B \leftarrow O(\log n)$: number of sparsified core graphs maintained in Lemma 9.8.
procedure *Initialize*$(G^{(0)}, \boldsymbol{\ell})$
  Initialize $\mathcal{G}_0 = \{G^{(0)}\}$.
  Rebuild$(0, 0)$
procedure *Rebuild*(i_0, t_0)
  for $i = i_0, \ldots, d-1$ **do**
    $\text{prev}_{i+1} \leftarrow t_0$.
    $\mathcal{G}_{i+1} \leftarrow \{\}$
    for $G \in \mathcal{G}_i$ **do**
      $\mathcal{A}_G^{(SparseCore)}$.Initialize$(G, \boldsymbol{\ell}_G)$
      For $j \in [B]$, add $\mathcal{SC}(G, F_j)$ to $\mathcal{G}_{i+1}$.
procedure *DynamicBranchingChain*$(G^{(t)}, U^{(t)}, \boldsymbol{g}^{(t)}, \boldsymbol{\ell}^{(t)})$
  $U^{(t)}_{G^{(t)}} \leftarrow U^{(t)}$
  for $i = 0, \ldots, d-1$ **do**
    if *The accumulated encoding size of updates of any* $G \in \mathcal{G}_i$ *exceeds* $m(\gamma_s/k)^{i+1}/\log^2 n$ **then**
      Rebuild(i, t)
    for $G \in \mathcal{G}_i$ **do**
      $\left\{U^{(t)}_{SC(G,F_j)} \;\middle|\; j \in [B]\right\} \leftarrow \mathcal{A}_G^{(SparseCore)}$.DynamicSparseCore$(G, U^{(t)}_G)$

for $G \in \mathcal{G}_i^{(t)}$ *for all* $0 \le i \le d$ *are recursively defined via Definition 9.3, 9.6 then there is a tree-chain* $G_0, \ldots, G_d$ *with*

$$\|\boldsymbol{w}^{(t),G_i}\|_1 \le \widetilde{O}(\gamma_l)^i \left(\sum_{j=0}^{i} \|\boldsymbol{w}^{(\text{prev}_j^{(t)})}\|_1 + \|\boldsymbol{w}^{(t)}\|_1 \right) \text{ for all } i \in \{0, 1, \ldots, d\}. \tag{32}$$

The algorithm succeeds with high probability and runs in total time $m^{1/d}\widetilde{O}(\gamma_s\gamma_r)^{O(d)}(m+Q)$ *for* $Q \stackrel{\text{def}}{=} \sum_t \textit{Enc}(U^{(t)}) \le$ poly(n).

Remark 9.10. *Theorem 9.1 maintains the data structure implementing Lemma 9.9 on dynamic graphs undergoing only edge insertions/deletions. However, it can be modified to also support vertex splits since it is built using Lemmas 8.5 and 9.8 and Theorem 7.1, all which support vertex splits.*

Algorithm 5 initializes a B-branching tree chain as in Definition 8.10. For every graph $G \in \mathcal{G}_i$ for some level i, it maintains a collection of forests, trees, and sparsified core graph using the dynamic data structure from Lemma 9.8.

However, the data structure of Lemma 9.8 can only take up to $m/(k \log^2 n)$ updates if the input graph has at most m edges at all time. This forces us to rebuild the data structure every once in a while. In particular, we rebuild everything

at every level $i \geq i_0$ if any of the data structures of Lemma 9.8 on some level i_0 graph $G \in \mathcal{G}_{i_0}$ has accumulated too many updates (approximately m/k^{i_0}). We will show that the cost for rebuilding amortizes well across dynamic updates.

Proof of Lemma 9.9. At any level $i = 0, \ldots, d$, there are at most $O(\log n)^i$ graphs maintained at i-th level at any given stage t due to Lemma 9.8. That is, we have $|\mathcal{G}_i^{(t)}| \leq O(\log n)^i$ for any t and i. At any stage t and level $i > 0$, every graph $G \in \mathcal{G}_i^{(t)}$ has at most $m\gamma_s^{i-1}/k^i$ vertices and $m(\gamma_s/k)^i$ edges. This is again due to Lemma 9.8.

To analyze the runtime, note that every $m\gamma_s^i/\left(k^{i+1}\log^2 n\right)$ updates to some graph $G \in \mathcal{G}_i$ create $O(m\gamma_s^i\gamma_r/k^{i+1})$ updates to every $\mathcal{SC}(G, F_j)$ for $j \in [B]$ by Lemma 9.8. Therefore, over the course of Q updates to the top level graph G, the total number of updates to the $B = O(\log n)$ graphs at level i is

$$Q \cdot O\left(\gamma_r \log^3 n\right)^i.$$

Next we analyze the runtime cost of rebuilding level i_0. By Lemma 9.8, it takes $O(m(\gamma_s/k)^i k\gamma_r)$ time to initialize $\mathcal{A}^{(9.8)}$ for any graph $G \in \mathcal{G}_i$ at any level i. Therefore, the cost of rebuilding the graphs of every level $i \geq i_0$ is

$$\sum_{i=i_0}^{d} O(\log n)^i \cdot m(\gamma_s/k)^i k\gamma_r = \frac{m}{k^{i_0}} \cdot \widetilde{O}(\gamma_s)^d k\gamma_r.$$

However, the rebuild happens at most every $m(\gamma_s/k)^{i_0}/\log^2 n$ total updates to graphs at level i_0. Thus, over the course of at most $Q \cdot O\left(\gamma_r \log^3 n\right)^{i_0}$ updates to every graph at level i_0, the total runtime cost spent on rebuilding level i_0 is at most

$$(m+Q)\, k\gamma_r \widetilde{O}(\gamma_s\gamma_r)^{O(d)}.$$

The overall runtime bound follows because $k = m^{1/d}$.

We now show (32) by induction on t and the the level i, and prove the result for level $i+1$ at a time t given a partial tree chain $G_0, \ldots, G_i$ satisfying (32). Let $G_i^{(\mathrm{prev}_i^{(t)})}$ be the version of graph G_i when it was rebuilt at time $\mathrm{prev}_i^{(t)}$.

If $\mathrm{prev}_{i+1}^{(t)} < t$, we can use the same chain $G_0, \ldots, G_{i+1}$ as the change from stage $\mathrm{prev}_{i+1}^{(t)}$ because Lemma 9.8 guarantees that there is an index $j^* \in [B]$ which satisfies (30) for all stages in $[\mathrm{prev}_{i+1}^{(t)}, t]$, where $G_{i+1} = \mathcal{S}(G_i, F_{j^*})$. By induction and Lemma 9.8, we deduce that

$$\begin{aligned}
\left\|\boldsymbol{w}^{(t),G_{i+1}}\right\|_1 &\overset{(i)}{\leq} \widetilde{O}(\gamma_l)\left(\left\|\boldsymbol{w}^{\left(\mathrm{prev}_i^{(t)}\right),G_i^{\left(\mathrm{prev}_i^{(t)}\right)}}\right\|_1 + \left\|\boldsymbol{w}^{(t),G_i}\right\|_1\right) \\
&\overset{(ii)}{\leq} \widetilde{O}(\gamma_l)\left[\widetilde{O}(\gamma_l)^i \sum_{j=0}^{i}\left\|\boldsymbol{w}^{\left(\mathrm{prev}_j^{(t)}\right)}\right\|_1 + \widetilde{O}(\gamma_l)^i\left(\sum_{j=0}^{i}\left\|\boldsymbol{w}^{\left(\mathrm{prev}_j^{(t)}\right)}\right\|_1 + \left\|\boldsymbol{w}^{(t)}\right\|_1\right)\right] \\
&\leq \widetilde{O}(\gamma_l)^{i+1}\left(\sum_{j=0}^{i}\left\|\boldsymbol{w}^{\left(\mathrm{prev}_j^{(t)}\right)}\right\|_1 + \left\|\boldsymbol{w}^{(t)}\right\|_1\right).
\end{aligned}$$

(i) is because the vector $\boldsymbol{w}^{(0)}$ in Lemma 9.8 corresponds to $\boldsymbol{w}^{(\mathrm{prev}_i^{(t)}),G_i^{(\mathrm{prev}_i^{(t)})}}$, as $\mathrm{prev}_i^{(t)}$ is the initialization time of level i. (ii) is by induction on the stage t and level i. Thus we have shown (32) by induction, which completes the proof. □

9.2 Finding Approximate Min-Ratio Cycles in a Tree-Chain

In this section we explain how to extract a cycle $\mathbf{\Delta}$ from a branching tree-chain (such as the one maintained in Lemma 9.9) with large quality $|\boldsymbol{g}^\top \Delta|/\|\boldsymbol{L}\Delta\|_1$, satisfying the guarantees of Theorem 9.1. As a branching tree-chain consists of $O(\log n)^d$ tree-chains, we focus on getting a cycle $\mathbf{\Delta}$ out of a single tree-chain. More formally, our setting for much of this section will be a tree-chain $G_0, G_1, \ldots, G_d$ (Definition 8.10), with a corresponding tree $T \stackrel{\text{def}}{=} T^{G_0,\ldots,G_d}$ as defined in Definition 8.11. For $\boldsymbol{g}, \boldsymbol{\ell}$ and a valid pair $\boldsymbol{c}, \boldsymbol{w}$ (Definition 9.2), we can define $\boldsymbol{c}^{G_0} \stackrel{\text{def}}{=} \boldsymbol{c}$ and $\boldsymbol{w}^{G_0} \stackrel{\text{def}}{=} \boldsymbol{w}$, and $\boldsymbol{c}^{G_i}$ and $\boldsymbol{w}^{G_i}$ recursively for $1 \le i \le d$ via Definitions 9.3 and 9.6. Let $\boldsymbol{\ell}^{G_i}, \boldsymbol{g}^{G_i}$ be the lengths and gradients on the graphs G_i, and $\boldsymbol{\ell}^{C(G_i,F_i)}, \boldsymbol{g}^{C(G_i,F_i)}$ be the lengths and gradients on the core graphs.

Note that every edge $e^G \in E(G) \setminus E(T)$ has a "lowest" level that the image of it (which we call e) exists in a tree chain, after which it is not in the next sparsified core graph. In this case, the edge plus its path embedding induce a cycle, which we call the *sparsifier cycle* associated to e. In the below definition we assume that the path embedding of a self-loop e in $C(G_i, F_i)$ is empty.

Definition 9.11. *Consider a tree-chain $G_0 = G, \ldots, G_d$ (Definition 8.10) with corresponding tree $T \stackrel{\text{def}}{=} T^{G_0,\ldots,G_d}$ where for every $0 \le i \le d$, we have a core graph $C(G_i, F_i)$ and sparsified core graph $\mathcal{S}(G_i, F_i) \subseteq C(G_i, F_i)$, with embedding $\Pi_{C(G_i,F_i)\to\mathcal{S}(G_i,F_i)}$.*

We say an edge $e^G \in E(G)$ is at level $\mathsf{level}_{e^G} = i$ if its image e is in $E(C(G_i, F_i))\setminus E(\mathcal{S}(G_i, F_i))$. Define the sparsifier cycle *$a(e)$ of such an edge $e = e_0 \in C(G_i, F_i)$ to be the cycle $a(e) = e_0 \oplus \mathsf{rev}(\Pi_{C(G_i,F_i)\to\mathcal{S}(G_i,F_i)}(e_0)) = e_0 \oplus e_1 \oplus \cdots \oplus e_L$. We define the preimage of this sparsifier cycle in G to be the* fundamental chain cycle

$$a^G(e^G) = e_0^G \oplus T[v_0^G, u_1^G] \oplus e_1^G \oplus T[v_1^G, u_2^G] \oplus \cdots \oplus e_L^G \oplus T[v_L^G, u_{L+1}^G],$$

where $e_i^G = (u_i^G, v_i^G)$ is the preimage of edge e_i in G for each $i \in [L]$ and where we define $u_{L+1}^G = u_0^G$.

We let $\boldsymbol{a}(e)$ and $\boldsymbol{a}^G(e^G)$ be the associated flow vectors for the sparsifier cycle $a(e)$ and fundamental chain cycle $a^G(e^G)$.

At a high level, our algorithm will maintain the total gradient of every fundamental chain cycle explicitly. Note that this implies that the gradient of at most $m^{o(1)}$ fundamental chain cycles change per iteration on average. Also, the algorithm maintains length *overestimates* of each fundamental chain cycle, as maintaining the true length dynamically is potentially expensive. The algorithm will return the overall best quality fundamental chain cycle.

Definition 9.12. *Consider a tree-chain $G = G_0, \ldots, G_d$ with corresponding spanning tree $T \stackrel{\text{def}}{=} T^{G_0,\ldots,G_d}$. For any edge $e^G \in E(G) \setminus E(T)$ at level i with image e in $C(G_i, F_i) \setminus \mathcal{S}(G_i, F_i)$ we define $\widetilde{\mathsf{len}}^G_{e^G}$, an overestimate on the length of e^G's fundamental chain cycle, as $\widetilde{\mathsf{len}}_{e^G} \stackrel{\text{def}}{=} \langle \boldsymbol{\ell}^{C(G_i,F_i)}, |\boldsymbol{a}(e)| \rangle$.*

Because the lengths and gradients on all edges in all the G_i, and embeddings $\Pi_{C(G_i,F_i)\to\mathcal{S}(G_i,F_i)}$ are maintained explicitly in Lemma 9.9, we can store length overestimates $\widetilde{\mathsf{len}}_{e^G}$ for all fundamental chain cycles, and their total gradients with a constant overhead.

There are two more important pieces to check. First, we need to check that the gradients defined on the core graphs Definition 8.7 indeed given the correct total gradient for each cycle, and that the values $\widetilde{\mathsf{len}}_{e^G}$ are indeed overestimates for the lengths of all the fundamental chain cycles $\boldsymbol{a}^G(e^G)$. Then we will show that using the length overestimates $\widetilde{\mathsf{len}}_{e^G}$ still allows us to return a sufficiently good fundamental chain cycle.

Lemma 9.13 (Gradient correctness). *Let $e^G \notin E(T)$ be an edge with $\mathsf{level}_{e^G} = i$ and let e be its image in G_i. Then the total gradient of the cycle $a(e)$ and its preimage $a^G(e^G)$ are the same, i.e. $\langle \boldsymbol{g}^{C(G_i,F_i)}, \boldsymbol{a}(e) \rangle = \langle \boldsymbol{g}, \boldsymbol{a}^G(e^G) \rangle$.*

Lemma 9.14 (Length overestimates). *Let $e^G \notin E(T)$ be an edge with $\mathsf{level}_{e^G} = i$ and let e be its image in G_i. Then the values $\widetilde{\mathsf{len}}_{e^G}$ overestimate the length of the preimage cycle $a^G(e^G)$, i.e. $\widetilde{\mathsf{len}}_{e^G} \geq \langle \boldsymbol{\ell}, |\boldsymbol{a}^G(e^G)| \rangle$.*

It is useful to define the concept of *lifting* a cycle back from $C(G_i, F_i)$ to G_i in order to show Lemmas 9.13 and 9.14.

Definition 9.15 (Lifted cycle). *Consider a cycle $\widehat{C}$ in $C(G_i, F_i)$ with edges $\widehat{e_1} \oplus \widehat{e_2} \oplus \cdots \oplus \widehat{e_L}$, such that $e_i = (u_i, v_i)$ is the preimage of $\widehat{e_i}$ in G_i. We define the lift of $\widehat{C}$ into G_i as the cycle*

$$e_1 \oplus F_i[v_1, u_2] \oplus e_2 \oplus \cdots \oplus e_L \oplus F_i[v_L, u_1].$$

Now we can show Lemmas 9.13 and 9.14 by repeatedly lifting cycles until we are back in the top level graph G.

Proof of Lemma 9.13. It suffices to show that any cycle $\widehat{C}$ in $C(G_i, F_i)$ and its lift C in G_i have the same gradient. Precisely, if we let $\widehat{\boldsymbol{c}}$ and $\boldsymbol{c}$ denote the flow vectors of $\widehat{C}$ and C respectively, we wish to show $\langle \boldsymbol{g}^{C(G_i,F_i)}, \widehat{\boldsymbol{c}} \rangle = \langle \boldsymbol{g}^{G_i}, \boldsymbol{c} \rangle$. To see this, recall that by the definition of $\boldsymbol{g}^{C(G_i,F_i)}$ in Definition 8.7, for $\widehat{C} = \widehat{e_1} \oplus \cdots \oplus \widehat{e_L}$ for $e_j = (u_j, v_j)$ and $E(F_i) \subseteq E(T_i)$ for some tree T_i (namely the tree used to initialize the forest F_i of $C(G_i, F_i)$),

$$\begin{aligned}
\langle \boldsymbol{g}^{C(G_i,F_i)}, \widehat{\boldsymbol{c}} \rangle &= \sum_{j=1}^{L} \boldsymbol{g}^{C(G_i,F_i)}_{\widehat{e_j}} = \sum_{j=1}^{L} \boldsymbol{g}^{G_i}_{e_j} + \langle \boldsymbol{g}^{G_i}, \boldsymbol{p}(T_i[v_j, u_j]) \rangle \\
&\overset{(i)}{=} \sum_{j=1}^{L} \boldsymbol{g}^{G_i}_{e_j} + \langle \boldsymbol{g}^{G_i}, \boldsymbol{p}(T_i[v_j, u_{j+1}]) \rangle \\
&\overset{(ii)}{=} \sum_{j=1}^{L} \boldsymbol{g}^{G_i}_{e_j} + \langle \boldsymbol{g}^{G_i}, \boldsymbol{p}(F_i[v_j, u_{j+1}]) = \langle \boldsymbol{g}^{G_i}, \boldsymbol{c} \rangle,
\end{aligned}$$

where (i) follows because $\sum_{j=1}^{L} \boldsymbol{p}(T_i[v_j, u_j]) = \sum_{j=1}^{L} \boldsymbol{p}(T_i[v_j, u_{j+1}])$ (for $u_{L+1} \overset{\text{def}}{=} u_1$) because both sides route the same demand on a tree T_i. (ii) follows because $F_i \subseteq T$, and v_j, u_{j+1} are in the same connected component of F_i. The last equality follows by the definition of C. □

Proof of Lemma 9.14. Similar to the above proof of Lemma 9.13, by repeatedly lifting until we get to G, it suffices to show that the length of a cycle $\widehat{C}$ in $C(G_i, F_i)$ is larger than that of its lift C. Formally, if $\widehat{\boldsymbol{c}}$ and $\boldsymbol{c}$ denote the flow vectors of $\widehat{C}$ and C respectively, we wish to show $\langle \boldsymbol{\ell}^{C(G_i,F_i)}, |\widehat{\boldsymbol{c}}| \rangle \geq \langle \boldsymbol{\ell}^{G_i}, |\boldsymbol{c}| \rangle$. For $\widehat{C} = \widehat{e_1} \oplus \cdots \oplus \widehat{e_L}$ for $e_j = (u_j, v_j)$ and forest F_i, we have

$$\begin{aligned}
\langle \boldsymbol{\ell}^{C(G_i,F_i)}, |\widehat{\boldsymbol{c}}| \rangle &= \sum_{j=1}^{L} \boldsymbol{\ell}^{C(G_i,F_i)}_{\widehat{e_j}} \overset{(i)}{=} \sum_{j=1}^{L} \widetilde{\mathsf{str}}^{i}_{e_j} \boldsymbol{\ell}^{G_i}_{e_j} \overset{(ii)}{\geq} \sum_{j=1}^{L} \mathsf{str}^{F_i, \boldsymbol{\ell}^{G_i}}_{e_j} \boldsymbol{\ell}^{G_i}_{e_j} \\
&= \sum_{j=1}^{L} \boldsymbol{\ell}^{G_i}_{e_j} + \langle \boldsymbol{\ell}^{G_i}, |\boldsymbol{p}(F_i[u_j, \mathsf{root}^{F_i}_{u_j}])| \rangle + \langle \boldsymbol{\ell}^{G_i}, |\boldsymbol{p}(F_i[v_j, \mathsf{root}^{F_i}_{v_j}])| \rangle \\
&= \sum_{j=1}^{L} \boldsymbol{\ell}^{G_i}_{e_j} + \langle \boldsymbol{\ell}^{G_i}, |\boldsymbol{p}(F_i[v_j, \mathsf{root}^{F_i}_{v_j}])| \rangle + \langle \boldsymbol{\ell}^{G_i}, |\boldsymbol{p}(F_i[u_{j+1}, \mathsf{root}^{F_i}_{u_{j+1}}])| \rangle \\
&\overset{(iii)}{\geq} \sum_{j=1}^{L} \boldsymbol{\ell}^{G_i}_{e_j} + \langle \boldsymbol{\ell}^{G_i}, |\boldsymbol{p}(F_i[v_j, u_{j+1}])| \rangle = \langle \boldsymbol{\ell}^{G_i}, |\boldsymbol{c}| \rangle,
\end{aligned}$$

where (i) follows from the definition of $\boldsymbol{\ell}^{C(G_i,F_i)}$ in Definition 8.7, (ii) follows from Lemma 8.5 item 2, and (iii) follows from $\mathsf{root}^{F_i}_{v_j} = \mathsf{root}^{F_i}_{u_{j+1}}$ and the triangle inequality. The final equality is from the definition of C as the lift of $\widehat{C}$. □

We now show that it suffices to maintain the "best quality" fundamental chain cycle, i.e. $\max_{e^G \in E(G)\setminus E(T)} |\langle \boldsymbol{g}^G, \boldsymbol{a}^G(e^G)\rangle|/\widetilde{\mathrm{len}}_{e^G}$. To show this, we first explain how to express a cycle $\boldsymbol{c}$ as the combination of fundamental chain cycles.

Lemma 9.16. *Given a circulation $\boldsymbol{c}$ in graph G, recursively define $\boldsymbol{c}^{G_i}$ for all $i = 0, \ldots, d$ via Definitions 9.3 and 9.6. Then*

$$\boldsymbol{c} = \sum_{i=0}^{d} \sum_{e^G:\mathsf{level}_{e^G}=i} \boldsymbol{c}^{G_i}_{e^i} \boldsymbol{a}^G(e^G).$$

Proof. Define $\boldsymbol{y} \stackrel{\text{def}}{=} \sum_{i=0}^{d} \sum_{e^G:\mathsf{level}_{e^G}=i} \boldsymbol{c}^{G_i}_{e^i} \boldsymbol{a}^G(e^G)$. We will show that $\boldsymbol{c}_{e^G} = \boldsymbol{y}_{e^G}$ for any edge $e^G \in G \setminus T$.

First, define for any $i = 0, \ldots, d$, Π_i as the embedding $\Pi_{C(G_i,F_i)\to\mathcal{SC}(G_i,F_i)}$ and e^i as the image in G_i for any edge $e^G \in G$. Clearly, e^i is well-defined if $i \le \mathsf{level}_{e^G}$. We also denote the image of e^i in the core graph $C(G_i, F_i)$ as $\widehat{e}^i$.

At any level i, observe that if $e^{i+1} \in G_{i+1}$, we have $\widehat{e}^i = e^{i+1}$, $\Pi_i(\widehat{e}^i) = \{e^{i+1}\}$ and therefore $\boldsymbol{c}^{G_{i+1}}_{e^{i+1}} = \boldsymbol{c}^{G_i}_{e^i}$. Otherwise, $\boldsymbol{c}^{G_i}_{e^i}$ is added to $\boldsymbol{c}^{G_{i+1}}_{f}$, $f \in \Pi_i(\widehat{e}^i)$. Let e^G be any edge in $G \setminus T$ at level i. Following from Definition 9.6, we can express $\boldsymbol{c}^{G_i}_{e^i}$ as

$$\boldsymbol{c}^{G_i}_{e^i} = \boldsymbol{c}_{e^G} + \sum_{j=0}^{i-1} \sum_{f^G:\mathsf{level}_{f^G}=j} \boldsymbol{c}^{G_j}_{f^j} \cdot \boldsymbol{\Pi}_j(\widehat{f}^j)_{\widehat{e}^j}. \tag{33}$$

On the other hand, we know that e^G does not appear in any of the sparsifier cycle of f^G at level $j \ge i$. Thus, $[\boldsymbol{a}^G(f^G)]_{e^G} = 0$. If $\mathsf{level}_{f^G} = j < i$, $[\boldsymbol{a}^G(f^G)]_{e^G} = -\boldsymbol{\Pi}_j(\widehat{f}^j)_{\widehat{e}^j}$ where the -1 term comes from that the sparsifier cycle takes $\widehat{f}^j$ and the reverse of the path $\Pi_j(\widehat{f}^j)$. This yields that

$$\boldsymbol{y}_{e^G} = \boldsymbol{c}^{G_i}_{e^i} - \sum_{j=0}^{i-1} \sum_{f^G:\mathsf{level}_{f^G}=j} \boldsymbol{c}^{G_j}_{f^j} \boldsymbol{\Pi}_j(\widehat{f}^j)_{\widehat{e}^j} \stackrel{(i)}{=} \boldsymbol{c}_{e^G},$$

where (i) follows by rearranging (33).

The lemma follows via the fact that a circulation is uniquely determined by the amount of flows on non-tree edges. □

Lemma 9.17. *Let $\boldsymbol{c}, \boldsymbol{w}$ be a valid pair. Let $T = T^{G_0,\ldots,G_d}$ for a tree-chain $G_0, \ldots, G_d$. Then*

$$\max_{e^G \in E(G)\setminus E(T)} \frac{|\langle \boldsymbol{g}, \boldsymbol{a}^G(e^G)\rangle|}{\widetilde{\mathrm{len}}_{e^G}} \ge \frac{1}{\widetilde{O}(k)} \frac{|\langle \boldsymbol{g}, \boldsymbol{c}\rangle|}{\sum_{i=0}^{d} \|\boldsymbol{w}^{G_i}\|_1}.$$

Proof. Recall that for an edge $\widehat{e} \in C(G_i, F_i)$ with preimage e in G_i, its length is $\boldsymbol{\ell}^{C(G_i,F_i)}_{\widehat{e}} \stackrel{\text{def}}{=} \widetilde{\mathrm{str}}^i_e \boldsymbol{\ell}^{G_i}_e$ defined in Definition 8.7. Thus by the definition of $\widetilde{\mathrm{len}}_{e^G}$ in Definition 9.12,

$$\sum_{i=0}^{d} \sum_{e^G:\mathsf{level}_{e^G}=i} |\boldsymbol{c}^{G_i}_e| \widetilde{\mathrm{len}}_{e^G} = \sum_{i=0}^{d} \sum_{e^G:\mathsf{level}_{e^G}=i} \left(\widetilde{\mathrm{str}}^i \boldsymbol{\ell}^{G_i}_e |\boldsymbol{c}^{G_i}_e| + \sum_{e' \in \Pi_{C(G_i,F_i)\to\mathcal{S}(G_i,F_i)}(\widehat{e})} \boldsymbol{\ell}^{G_{i+1}}_{\widehat{e}} |\boldsymbol{c}^{G_i}_e| \right)$$
$$\stackrel{(i)}{\le} \sum_{i=0}^{d} \sum_{e^G:\mathsf{level}_{e^G}=i} \left(\widetilde{O}(k) \boldsymbol{w}^{G_i}_e + \boldsymbol{w}^{G_{i+1}}_{\widehat{e}} \right) \le \widetilde{O}(k) \sum_{i=0}^{d} \|\boldsymbol{w}^{G_i}_e\|_1,$$

where (i) follows $\widetilde{\mathrm{str}}^i \le \widetilde{O}(k)$ from Lemma 8.5 item 2, and the fact that $\boldsymbol{c}^{G_i}, \boldsymbol{w}^{G_i}$ are all valid pairs (see Lemmas 9.4 and 9.7) so $\boldsymbol{\ell}^{G_i}_e |\boldsymbol{c}^{G_i}_e| \le |\boldsymbol{w}^{G_i}_e|$, and the definition of $\boldsymbol{w}^{G_{i+1}}_e$ in Definition 9.6. Additionally, note by the triangle inequality

and Lemma 9.16 that

$$|\langle \boldsymbol{g}, \boldsymbol{c}\rangle| \le \sum_{i=0}^{d} \sum_{e^G:\mathsf{level}_{e^G}=i} |\boldsymbol{c}_e^{G_i}||\langle \boldsymbol{g}, \boldsymbol{a}^G(e^G)\rangle|.$$

Hence, we get by the fact that

$$\max_{i\in[n]} \frac{\boldsymbol{x}_i}{\boldsymbol{y}_i} \ge \frac{\sum_{i\in[n]} \boldsymbol{x}_i}{\sum_{i\in[n]} \boldsymbol{y}_i}$$

for $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n_{\ge 0}$ that

$$\max_{e^G\in E(G)\setminus E(T)} \frac{|\langle \boldsymbol{g}, \boldsymbol{a}^G(e^G)\rangle|}{\widetilde{\mathsf{len}}_{e^G}} \ge \frac{\sum_{i=0}^{d}\sum_{e^G:\mathsf{level}_{e^G}=i} |\boldsymbol{c}_e^{G_i}||\langle \boldsymbol{g}, \boldsymbol{a}^G(e^G)\rangle|}{\sum_{i=0}^{d}\sum_{e^G:\mathsf{level}_{e^G}=i} |\boldsymbol{c}_e^{G_i}|\widetilde{\mathsf{len}}_{e^G}} \ge \frac{1}{\widetilde{O}(k)} \frac{|\langle \boldsymbol{g}, \boldsymbol{c}\rangle|}{\sum_{i=0}^{d} \|\boldsymbol{w}^{G_i}\|_1}.$$

□

Remark. *We can adapt the statement and proof of Lemma 9.17 to remove the $\widetilde{O}(k)$ by being more careful. However, this further complicates the statements of Lemma 9.17 and its interaction with Section 8, and the extra $\widetilde{O}(k)$ does not meaningfully affect our runtimes.*

We can now complete the proof of Theorem 9.1.

Proof of Theorem 9.1. The first part is to dynamically maintain an explicit $O(\log n)$ branching tree chain with path embeddings using Lemma 9.9. With $O(1)$ overhead, the algorithm can also maintain the values of $\langle \boldsymbol{g}, \boldsymbol{a}^G(e^G)\rangle$, $\widetilde{\mathsf{len}}_{e^G}$ for all fundamental chain cycles of the $O(\log n)^d$ trees in the branching tree chain, because the branching tree chain maintains all edge gradients/lengths explicitly, and Definition 9.12 and Lemma 9.13. Hence in $\widetilde{O}(1)$ overhead it can maintain the maximizer $\arg\max_{e^G\in E(G)\setminus E(T)} \frac{|\langle \boldsymbol{g}, \boldsymbol{a}^G(e^G)\rangle|}{\widetilde{\mathsf{len}}_{e^G}}$ as desired in Lemma 9.17 for each tree. To show that the best out of these works, note that by Lemma 9.9 with high probability there is a tree-chain with

$$\frac{1}{\widetilde{O}(k)} \frac{|\langle \boldsymbol{g}, \boldsymbol{c}\rangle|}{\sum_{i=0}^{d} \|\boldsymbol{w}^{G_i}\|_1} \ge \frac{1}{\widetilde{O}(k)\widetilde{O}(\gamma_l)^{O(d)}} \frac{|\langle \boldsymbol{g}, \boldsymbol{c}^{(t)}\rangle|}{\sum_{i=0}^{d} \|\boldsymbol{w}^{(\mathsf{prev}_i^{(t)})}\|_1} \ge \kappa \frac{|\langle \boldsymbol{g}, \boldsymbol{c}^{(t)}\rangle|}{\sum_{i=0}^{d} \|\boldsymbol{w}^{(\mathsf{prev}_i^{(t)})}\|_1},$$

for $\kappa = 1/(\widetilde{O}(k)\widetilde{O}(\gamma_l)^{O(d)})$ as desired.

The total runtime is $(m+Q)m^{o(1)}$ for $Q = \sum_{t\in[\tau]} \textsc{Enc}(U^{(t)})$ by Lemma 9.9 for the choice $\gamma_s = \gamma_r = \exp(\log^{3/4}\log\log m)$, $d = \log^{1/8} m$ and $k = m^{1/d}$. Thus $\kappa = \exp(-O(\log^{7/8} m \log\log m))$. Also, Q approximates $\sum_{t\in[\tau]} |U^{(t)}|$ up to a polylog factor since $U^{(t)}$ contains only edge insertions/deletions.

Finally, we can rebuild levels $i, i+1, \ldots, d$ in time $m^{1+o(1)}/k^i$ time because the graphs on level i have $m(\gamma_s/k)^i$ edges, there are $O(\log n)^d$ such graphs, and the initialization time is almost linear. □

10 REBUILDING DATA STRUCTURE LEVELS

The goal of this section is to use Theorem 9.1 to get Theorem 8.2 through a *rebuilding game* to handle the cases where $\|\boldsymbol{w}^{(\mathsf{prev}_i^{(t)})}\|_1$ is much larger than $\|\boldsymbol{w}^{(t)}\|_1$ for some $0 \le i \le d$. The surprising aspect is that this is doable despite the fact that the $\boldsymbol{w}^{(t)}$ are all hidden. We now introduce the *rebuilding game* that captures these notions. Each round of the rebuilding game corresponds to our algorithm successfully returning a good enough cycle. When is this does not happen, we instead have to rebuild part of our data structure. The rebuilding game is designed to let us formally reason about strategies for rebuilding the data structure when it fails to find a good cycle.

Rebuilding game parameters and definition. The rebuilding game has several parameters: integers parameters size $m > 0$ and depth $d > 0$, update frequency $0 < \gamma_g < 1$, a rebuilding cost $C_r \geq 1$, a weight range $K \geq 1$, and a recursive size reduction parameter $k \stackrel{\text{def}}{=} m^{1/d} \geq 2$, and finally an integer round count $T > 0$.

Definition 10.1 (Rebuilding Game)**.** *The rebuilding game is played between a player and an advesary and proceeds in rounds $t = 1, 2, \ldots, T$. Additionally, the steps (moves) taken by the player are indexed as $s = 1, 2, \ldots$. Every step s is associated with a tuple $\text{prev}^{(s)} := (\text{prev}_0^{(s)}, \ldots, \text{prev}_d^{(s)}) \in [T]^{d+1}$. Both the player and adversary know $\text{prev}^{(s)}$. At the beginning of the game, at round $t = 1$ and step $s = 1$, we initially set $\text{prev}_i^{(1)} = 1$ for all levels $i \in \{0, 1, \ldots, d\}$.*

At the beginning each round $t \geq 1$,

(1) *The adversary first chooses a positive real weight $W^{(t)}$ satisfying $\log W^{(t)} \in (-K, K)$. This weight is **hidden** from the player.*

(2) *Then, **while** either of the following conditions hold,*

$$\sum_{i=0}^{d} W^{(\text{prev}_i^{(s)})} > 2(d+1)W^{(t)} \tag{34}$$

or

For some level l, at least $\gamma_g m/k^l$ rounds have passed since the last rebuild of level l. (35)

the adversary can (but does not have to) force the player to perform a fixing step*. The player may also choose to perform a fixing step, regardless of whether the adversary forces it or not. In a fixing step, the player picks a level $i \in \{0, 1, \ldots, d\}$, and we then set $\text{prev}_j^{(s+1)} \leftarrow t$ for $j \in \{i, i+1, \ldots, d\}$, and $\text{prev}_j^{(s+1)} \leftarrow \text{prev}_j^{(s)}$ for $j \in \{0, \ldots, i-1\}$. We call this a* fix *at level i, and we say the levels $j \geq i$ have been* rebuilt*. This move costs $C_r m/k^i$ time.*

(3) *When the player is no longer performing fixing steps, the round finishes.*

The goal of the player is to complete all T rounds in total time cost $O(\frac{C_r K d}{\gamma_g}(m + T))$.

Remark 10.2. *We emphasize an important point about our terminology in the rebuilding game: A* fix *at level i causes a* rebuild *of all levels $j \geq i$. The adversary can force a rebuild of a level l if it has participated in $\gamma_g m/k^l$ rounds since it was last rebuilt – and the latest rebuild may have been triggered by a fix at level l or by a fix at some level $i < l$.*

To translate the rebuilding game to the setting of Theorems 9.1 and 8.2 we can set $W^{(t)} \stackrel{\text{def}}{=} \|\boldsymbol{w}^{(t)}\|_1$. We give an algorithm where the player completes all T rounds with total time cost $O(\frac{C_r K d}{\gamma_g}(m + T))$. For our choice of parameters, this will be almost linear. Note that it is trivial for the player to finish all T rounds in time $O(C_r m T)$, as they could just always do a fix at level $i = 0$, as this sets all $\text{prev}_j^{(s)} \leftarrow t$.

The following lemma tells us that a fairly simple strategy can deterministically[4] ensure that the player always wins the rebuilding game.

Lemma 10.3 (Strategy for Rebuilding Game). *There is a deterministic strategy given by Algorithm 6 for the player to finish T rounds of the rebuilding game in time $O(\frac{C_r K d}{\gamma_g}(m + T))$.*

[4]Note that when we employ the rebuilding game strategy in our overall data structure, the data structure uses randomization. However, the randomized steps succeed with high probability union-bounded across the entire algorithm. The rebuilding game strategy corresponds to the behavior of the data structure assuming all these data structure randomization steps are successful. We address this formally in the proof of Theorem 8.2.

Algorithm 6: Strategy for the rebuilding game.

foreach $i = 0, \ldots, d$. **do**
  We maintain a "fixing count", $\texttt{fix}_i$, initialized to zero.;
  And we maintain a "round count", $\texttt{round}_i$, also initialized to zero.;
foreach *round* $t = 1, 2, \ldots, T$ *of the game* **do**
  if *there is a level* l *with* $\texttt{round}_l \geq \gamma_g m/k^l$ **then**
    Find the smallest level i such that $\texttt{round}_i \geq \gamma_g m/k^i$;
    Fix level i, thus rebuilding levels $j \geq i$.;
    For levels $j = i, i+1, \ldots, d$ set $\texttt{fix}_j \leftarrow 0$ and $\texttt{round}_j \leftarrow 0$. ;
    `// We call this a WIN at level i.`
  while *the adversary continues to force a fixing step* **do**
    Let i be the smallest level in $0, \ldots, d$ s.t. for all $j > i$, $\texttt{fix}_j = 2K$;
    Fix level i, thus rebuilding levels $j \geq i$.;
    Set $\texttt{fix}_i \leftarrow \texttt{fix}_i + 1$;
    `// We call this a LOSS at level i.`
  For all levels $j = 0, 1, \ldots, d$, set $\texttt{round}_j \leftarrow \texttt{round}_j + 1$.;

Before we state the proof, we first introduce some important terminology for understanding Algorithm 6 and its analysis.

The rebuilding game algorithm. Overall, our goal is the following: we (as the player) want to ensure that our vector of weights $W^{(\text{prev}_i^{(s)})}$ at different levels $i \in \{0, 1, \ldots, d\}$ is such that we must frequently succeed in completing a round without making too many fixing steps, and we need to ensure we do not spend too much time on fixing steps. To implement our strategy in Algorithm 6, we maintain two counters $\texttt{round}_i$ and $\texttt{fix}_i$ for each level i. These two counters are used to decide which level to rebuild in each step s of the game.

The first counter, $\texttt{round}_i$, is very simple. It counts the number of rounds that have occurred since level i was last rebuilt. Ideally, we would like to complete as many rounds as each level can handle, before we reset it. The adversary can force a fixing step if it has been more than $\gamma_g m/k^i$ rounds since level i was last rebuilt. In the setting of Theorem 9.1, this corresponds to a level of the branching tree chain accumulating enough updates that it should be rebuilt. We preempt the adversary by always rebuilding a level if it has been through this many rounds, regardless of whether the adversary forces us to or not. When this occurs, the level can "pay for itself", since the cost of fixing is low when amortized across the rounds since the last rebuild. Thus we declare a "WIN" at level i and rebuild levels $j \geq i$.

The second counter, $\texttt{fix}_i$, is the more interesting one. When a fixing step occurs and we decide to fix level i (thus rebuiling levels $j \geq i$) we say a "LOSS" occurred at level i. The $\texttt{fix}_i$ counter tracks how many times a fixing step occured and we had a LOSS at level i, counted since the last time we rebuilt level i due to a WIN at some level $l \leq i$. In a fixing step, we always decide to let the LOSS occur at the largest level index i where $\texttt{fix}_i < 2K$.

10.1 Analyzing the rebuilding game algorithm

Before we start our formal proof of Lemma 10.3, we will outline the main the main elements of the analyses of the time cost of using Algorithm 6 to play the rebuilding game.

Ideally, we would like to say that "if a LOSS occurs at level i, then the weight $W^{(\text{prev}_i^{(s)})}$ must be large compared to the current round t weight $W^{(t)}$", because then rebuilding would reduce $W^{(\text{prev}_i^{(s+1)})}$. However, this is not true, as it

may be that some other level's weight $W^{(\text{prev}_j^{(s)})}$ for $j < i$ is large enough to make Equation (34) hold, allowing the adversary to force a fixing step. Instead, the invariant we maintain is this: We ensure that when a fixing step occurs, if we choose to rebuild level i, it must be that either.

Case (A) either some level $l < i$ has an even larger weight than all levels $j \geq i$ and thus can be "blamed" for the fixing step, or

Case (B) no level $j > i$ has a weight large enough to force a fixing step.

Handling Case (B).. In case (B), we significantly reduce the weight $W^{(\text{prev}_i^{(s+1)})}$ compared to $W^{(\text{prev}_i^{(s)})}$ at level i for the next step $s+1$. Once we have $\texttt{fix}_j = 2K$ for all $j \geq i$, there must be level $l < i$ with weight larger than level i, as repeated occurrence of (B) ensures this.

Handling Case (A).. Now, the remaining key point is to make sure that in case (A), we do not waste too much time rebuilding at level i before moving to rebuilding at level $i-1$, so that we eventually start rebuilding the most problematic level $l < i$ with larger weight. Fortunately, our threshold of $2K$ fixes at level i is low enough to ensure this. Finally, to help us formalize that we make progress on reducing the weight at level i specifically in Case (A), we introduce a notion of "prefix maximizing" levels. At any step s, we say a level i is "prefix maximizing" if its weight $W^{(\text{prev}_i^{(s)})}$ is strictly larger than the weight $W^{(\text{prev}_j^{(s)})}$ at all levels $j < i$ This leads to the following definition.

Definition 10.4. *In the Rebuilding Game, at the start of each step s, we define a set $\mathcal{I}^{(s)}$ of "prefix maximizing" levels, given by*

$$\mathcal{I}^{(s)} = \left\{ i \in \{0, 1, \ldots, d\} \;\middle|\; W^{(\text{prev}_i^{(s)})} > W^{(\text{prev}_j^{(s)})} \text{ for all } j < i \right\}.$$

Note that $0 \in \mathcal{I}^{(s)}$ for all s, and the player does not know which levels are prefix maximizing.

This next lemma shows formally that strategy of Algorithm 6 successfully implements the kind of weight tracking we described above. Concretely, the lemma tells us that when a level i is prefix maximizing, the weight of the level must be pushed down as $\texttt{fix}_i$ increases.

Lemma 10.5 (Bound on fixing step count). *In the rebuilding game, suppose the player uses the strategy of Algorithm 6, then we always have $\texttt{fix}_0 < 2K$.*

This lemma tells use that we never have $2K$ LOSSes at level 0 before the next WIN at level 0. Since Algorithm 6 trivially ensures $\texttt{fix}_j \leq 2K$ for all levels $j > 0$, this gives us tight control over the number of fixing steps that can be forced by the adversary.

Proof of Lemma 10.5. We consider an instance of the rebuilding game and suppose the player uses the strategy given by Algorithm 6. To prove our lemma, we first introduce a condition which must be satisfied in each step by each level i which is prefix maximizing. This condition essentially states that the $\texttt{fix}_i$ counter is correctly tracking an upper bound on the level weight $W^{(\text{prev}_i^{(s)})}$. For convenience of our analysis, we define the condition for levels regardless of whether they are prefix maximizing, although we only need to show that it holds for such levels.

Definition. *At the start of step s, if for some level i we have,*

$$\log_2 \left(W^{(\text{prev}_i^{(s)})} \right) < K - \texttt{fix}_i \qquad (\texttt{fix}_i \textit{ correctness condition}) \tag{36}$$

we say that level i satisfies the $\texttt{fix}_i$ correctness condition at step s.

Given this notion, we can now state the induction hypothesis.

Inductive Hypothesis. *At the start of step s, Condition (36) holds for each $i \in \mathcal{I}^{(s)}$.*

We will prove this claim by induction on the step count s. First, we establish that proving this claim is sufficient to prove the lemma. By assumption, we have $\log_2\left(W^{(\mathrm{prev}_i^{(s)})}\right) > -K$, and hence the above claim would imply $\mathtt{fix}_i < 2K$ for all $i \in \mathcal{I}^{(s)}$, for all s. Since $0 \in \mathcal{I}^{(s)}$ for all s, we get that $\mathtt{fix}_0 < 2K$ always.

We first establish the base case $s = 1$. Note that trivially for all $i \in \{0, 1, 2, \ldots, d\}$, we have $\mathtt{fix}_i = 0$, and by assumption we have $\log_2(W^{(\mathrm{prev}_i^{(s)})}) < K = K - \mathtt{fix}_i$. This implies Condition (36) holds for every level i, and hence it holds for each level $i \in \mathcal{I}^{(s)}$. This establishes the base case.

Now, we now assume the induction hypothesis at the *start* of step s and prove it for the *start* of step $s+1$, i.e. we want to show Condition (36) holds for each $i \in \mathcal{I}^{(s+1)}$ at the start of step $s+1$. We break the analysis into two main cases, depending on what happens in step s. The first case (1) is when a WIN occurs. The second case (2) is when a LOSS occurs at some level i. We further break the second case into two sub-cases, separately handling when (2A) i is not a prefix maximizing level at step s and when (2B) i is a prefix maximizing level at step s. Conceptually, the key case is (2B), when we have a LOSS and $i \in \mathcal{I}^{(s)}$, which means we have to ensure that we make progress by reducing $W^{(\mathrm{prev}_i^{(s+1)})}$ compared to the earlier value $W^{(\mathrm{prev}_i^{(s)})}$.

Case 1: a WIN occurs. In this case, a WIN must occur at some level $i \in \{0, 1, 2, \ldots, d\}$ (since $\gamma_g m / k^d \le 1$). Let i denote the level at which the WIN occurs. In this case, for all levels $j \ge i$, we rebuild and set $\mathtt{fix}_i = 0$, and by the definition of K, we thus have (for the updated value of $\mathtt{fix}_i$) that $\log_2(W^{(\mathrm{prev}_i^{(s+1)})}) < K = K - \mathtt{fix}_i$. Thus, for each $j \ge i$, we have that Condition (36) holds, and thus, in particular, it must hold for each $j \in \mathcal{I}^{(s+1)}$.

For each level $l < i$, $\mathtt{fix}_l$ does not change and $W^{(\mathrm{prev}_l^{(s+1)})} = W^{(\mathrm{prev}_l^{(s)})}$. The latter implies that for each $l < i$, $l \in \mathcal{I}^{(s+1)}$ if and only if $l \in \mathcal{I}^{(s)}$. We also conclude that Condition (36) holds at the beginning of step $s+1$ if it held for level l at the beginning of step s. Hence, by the induction hypothesis at step s, Condition (36) holds for all $l < i$ with $l \in \mathcal{I}^{(s+1)}$.

This proves the induction hypothesis for step $s+1$ in the case where a WIN occurs.

Case 2: a LOSS occurs. We next consider the case when a *LOSS* occurs in step s, and we let the current round be denoted by t. The *LOSS* occurs at some level i. In order to analyze this case, we are going to split it further into two subcases 2A and 2B, depending on whether the LOSS occurs at level i which is in the prefix maximizing set or not (2B and 2A respectively). However, first we make some observations that are common to both cases 2A and 2B.

To start, we deal with levels $l < i$. As in the case of a WIN, we again have that, for each $l < i$, $\mathtt{fix}_l$ does not change and $W^{(\mathrm{prev}_l^{(s+1)})} = W^{(\mathrm{prev}_l^{(s)})}$. The latter implies that for each $l < i$, $l \in \mathcal{I}^{(s+1)}$ if and only if $l \in \mathcal{I}^{(s)}$. Hence, by the induction hypothesis, Condition (36) holds for each $l < i$ with $l \in \mathcal{I}^{(s+1)}$.

Next we need to deal with levels $j \ge i$. As a LOSS occurs in step s, we must have that $\sum_{j=0}^{d} W^{(\mathrm{prev}_j^{(s)})} \ge 2(d+1)W^{(t)}$. This implies

$$\max_{j=0}^{d} W^{(\mathrm{prev}_j^{(s)})} \ge \frac{2(d+1)}{d+1} W^{(t)} \ge 2W^{(t)}. \tag{37}$$

We claim that in this case, we must have

$$\mathcal{I}^{(s)} \cap \{i+1, \ldots, d\} = \emptyset. \tag{38}$$

Suppose for a contradiction that for some $j > i$ we have $j \in \mathcal{I}^{(s)}$. As $\texttt{fix}_j = 2K$, we conclude by the induction hypothesis, that at the start of step s, we have $\log_2(W^{(\text{prev}_j^{(s)})}) < K - \texttt{fix}_j = -K$, and hence $\log_2(W^{(\text{prev}_j^{(s)})}) < -K$. But, this is impossible, as $\log_2(W^{(\text{prev}_j^{(s)})}) > -K$ by the game definitions.

Subcase 2A: LOSS at level $i \notin \mathcal{I}^{(s)}$. We now further restrict to the case when at the start of step s, we have $i \notin \mathcal{I}^{(s)}$. We thus have at the start of step s, by the condition observed in Equation (38), that $\mathcal{I}^{(s)} \cap \{i, i+1, \ldots, d\} = \emptyset$.

This allows us to conclude that

$$\text{for all } j \in \{i, i+1, \ldots, d\} \text{ there exists } l < j \text{ with } W^{(\text{prev}_l^{(s)})} > W^{(\text{prev}_j^{(s)})}. \tag{39}$$

By Equation (39), we conclude that there exists $l < i$ with $W^{(\text{prev}_l^{(s)})} > \max_{j \in \{i, i+1, \ldots, d\}} W^{(\text{prev}_j^{(s)})}$. Furthermore, we can conclude that $\max_{h \in \{0,\ldots,i-1\}} W^{(\text{prev}_h^{(s)})} \geq 2W^{(t)}$, since the maximum in Equation (37) is not achieved by an index $\geq i$. Consequently, when the LOSS at level i occurs and we rebuild, for all levels $j \in \{i, i+1, \ldots, d\}$, we set $\text{prev}_j^{(s+1)} \leftarrow t$, and hence at the start of step $s+1$, we have for all $j \geq i$ that

$$W^{(\text{prev}_j^{(s+1)})} = W^{(t)} \leq \frac{1}{2} \max_{h \in \{0,\ldots,i-1\}} W^{(\text{prev}_h^{(s)})} = \frac{1}{2} \max_{h \in \{0,\ldots,i-1\}} W^{(\text{prev}_h^{(s+1)})}$$

and hence for all $j \geq i$ we conclude that $j \notin \mathcal{I}^{(s+1)}$. Altogether, this proves the induction hypothesis for step $s+1$ in the case where a LOSS occurs at level i and $i \notin \mathcal{I}^{(s)}$.

Subcase 2B: LOSS at level $i \in \mathcal{I}^{(s)}$. We now consider the case when at the start of step s, we have $i \in \mathcal{I}^{(s)}$. This is the most important case, where we ensure a reduction in the weight $W^{(\text{prev}_i^{(s+1)})}$ compared to $W^{(\text{prev}_i^{(s)})}$. By Equation (38), we have $\mathcal{I}^{(s)} \cap \{i+1, \ldots, d\} = \emptyset$, and hence we conclude that $W^{(\text{prev}_i^{(s)})} = \max_{j=0}^{d} W^{(\text{prev}_j^{(s)})} \geq 2W^{(t)}$. By the induction hypothesis, we have (labelling $\texttt{fix}_i$ explicitly by step for clarity)

$$\log_2(W^{(\text{prev}_i^{(s)})}) < K - \texttt{fix}_i^{(s)}$$

hence,

$$\log_2(W^{(\text{prev}_i^{(s+1)})}) = \log_2(W^{(t)}) \leq \log_2(W^{(\text{prev}_i^{(s)})}) - 1 < K - \texttt{fix}_i^{(s)} - 1 = K - \texttt{fix}_i^{(s+1)}. \tag{40}$$

Thus, Condition (36) holds for i at the end of step $s+1$, regardless of whether $i \in \mathcal{I}^{(s+1)}$.

Furthermore, for all $j > i$, we set $W^{(\text{prev}_j^{(s+1)})} = W^{(t)} = W^{(\text{prev}_i^{(s+1)})}$, and hence $j \notin \mathcal{I}^{(s+1)}$.

Again, recall that we already dealt with established Condition (36) for $l < i$ above in Equation (39). Thus the induction hypothesis holds for step $s+1$ when $i \in \mathcal{I}^{(s)}$,

This completes our case analysis, establishing the inductive hypothesis, and hence the lemma. □

At this point, armed with the conclusion of Lemma 10.5, we are ready to analyze the running time of the rebuilding game strategy given by Algorithm 6, to prove the main lemma of this section, Lemma 10.3.

Before starting the proof, we will briefly outline its main elements. The costs of the rebuilding game occur during fixes as part of either a WIN or a LOSS in Line 6 and Line 10 respectively.

We use a standard amortization argument to account for the cost of fixes that occur during WINs. We can count the cost occurred during WINs separately at each level, and finally add it up across these. In each level, the cost per round can be bounded by C_r/γ_g.

Next, we have to account for the cost of fixes carried out during LOSSes. These fixes are all accounted for by increases in some $\texttt{fix}_j$ counter. We then bound the total cost of fixes that later have their $\texttt{fix}_j$ counter reset by amortizing the

cost toward the rounds that cause the reset of the $\texttt{fix}_j$ counter through a WIN at some level $i \le j$. Because Lemma 10.5 guarantees the $\texttt{fix}_j$ counters are bounded by $2K$, we can bound the additional cost amortized toward the rounds during the WIN at level i by $\frac{4KC_r}{\gamma_g}$ per step. Finally the bound on the $\texttt{fix}_j$ counters from Lemma 10.5 also tells us that the leftover cost unaccounted for by amortization through resets is also bounded, this time by $4KC_rm$ in total.

Proof of Lemma 10.3. To bound the running time, we use a simple amortized analysis across the steps of the rebuilding game.

The round counter at level i, i.e., $\texttt{round}_i$ increases by 1 in each round and hence the sum of the increases is T. If a WIN occurs at level i, we incur a time cost through a fix of level i, with a cost of C_rm/k^i (Line 6). At the same time, we reduce the round counter at i and all deeper levels $j > i$, and in particular, we reduce $\texttt{round}_i$ by $\gamma_g m/k^i$. We will amortize the cost of this fix toward the rounds that increased $\texttt{round}_i$ from zero to the threshold $\gamma_g m/k^i$, and thus the amortized cost from fixes during WINs at level i per round is at most C_r/γ_g. When we add this up across T rounds, the total cost from fixes during WINs at level i is $T \cdot C_r/\gamma_g$. Thus the total cost added across our $d+1$ levels from fixes during WINs is

$$\text{cost from fixes during WINs} \le T(d+1)C_r/\gamma_g. \tag{41}$$

All the cost incurred during a LOSS at some level i (Line 10) leads to an increase of the fixing step counter $\texttt{fix}_i$ by 1, and has an associated time cost of C_rm/k^i. We will break the cost form LOSSes into two parts:

(1) Cost accounted for by a $\texttt{fix}_i$ counter increase where the fix counter is later reset to 0.
(2) Cost accounted for by a $\texttt{fix}_i$ counter increase where the fix counter is not reset.

We can bound the cost arising from Part 2 very easily: By Lemma 10.5, $\texttt{fix}_i \le 2K$, and so the cost from fixing of level i without a reset of the counter following is bounded by $\texttt{fix}_i \cdot C_rm/k^i \le 2K \cdot C_rm/k^i$. Adding this cost across all levels we get that the total cost from Part 2 is upper bounded by

$$\text{cost from fixes during LOSSes with no } \texttt{fix} \text{ counter reset} \le \sum_{i=0}^{d} 2K \cdot C_rm/k^i \le 4KC_rm. \tag{42}$$

Finally, we bound the cost from Part 1. Consider the resetting of some counter $\texttt{fix}_j$ associated with a level j. Any such counter is reset during a WIN at some level $i < j$. We will bound the cost part by amortizing it toward the rounds that caused this WIN at level i. In particular, note that the level i experienced least $\gamma_g m/k^i$ rounds since it was last rebuilt and at this point $\texttt{fix}_j$ was reset (though it may also have been reset again since). This means we can count the cost associated with the increases in $\texttt{fix}_j$ toward the WIN at level i. The total cost we need to account for in this way toward the WIN at level i is then

$$\sum_{j=i}^{d} \texttt{fix}_j \cdot C_rm/k^j \le 2K \sum_{j=i}^{d} C_rm/k^j \le 4KC_rm/k^i.$$

Thus, the amortized cost per round associated with WINs at level i through these $\texttt{fix}_j$ resets is at most

$$\frac{4KC_rm/k^i}{\gamma_g m/k^i} = \frac{4KC_r}{\gamma_g}.$$

As we have T rounds, the total cost associated with WINs at level i through $\texttt{fix}_j$ resets is then $\frac{4KC_r}{\gamma_g}T$. Since a round can contribute toward a WIN at each of our $d+1$ levels, this means the cost amortized to toward a round across all

levels is .

$$\text{cost from fixes during LOSSes with } \texttt{fix} \text{ counter reset } \leq \sum_{i=0}^{d} T \cdot \frac{4KC_r}{\gamma_g} = \frac{4KC_r(d+1)}{\gamma_g} T. \tag{43}$$

Finally, adding together the costs accounted for in Equations (41), (42), and (43), and the cost of executing rounds, we get a bound on the total cost of $O(\frac{KC_r d}{\gamma_g}(T+m))$ as desired. □

10.2 Dynamic Min-Ratio Cycle Using the Rebuilding Game

In this section we combine Theorem 9.1 and Lemma 10.3 to show Theorem 8.2 which gives a data structure for returning min-ratio cycles in dynamic graphs with hidden stable-flow chasing updates.

Proof of Theorem 8.2. Let $W^{(t)} \stackrel{\text{def}}{=} \|\boldsymbol{w}^{(t)}\|_1$. The adversary plays the following strategy. They feed the inputs $\boldsymbol{g}^{(t)}, \boldsymbol{\ell}^{(t)}, U^{(t)}$ to Theorem 9.1 and get a cycle Δ. Let $\kappa^{(9.1)}$ be the approximate parameter from Theorem 9.1. Let $\kappa \stackrel{\text{def}}{=} \kappa^{(9.1)}/(2d+2)$. The adversary checks whether

$$\langle \boldsymbol{g}^{(t)}, \Delta \rangle / \|\boldsymbol{\ell}^{(t)} \circ \Delta\|_1 \leq -\kappa\alpha. \tag{44}$$

Because Δ is represented using $m^{o(1)}$ edges on a tree T, this can be performed in amortized $m^{o(1)}$ time by using a dynamic tree (Lemma 5.3). If (44) holds, the adversary allow a progress step, and this completes the Query() operation of Theorem 8.2. Otherwise, they force the player to perform a fixing step. This is valid because by Theorem 9.1 we must have

$$\frac{\kappa^{(9.1)}}{2d} \frac{\langle \boldsymbol{g}^{(t)}, \boldsymbol{c}^{(t)} \rangle}{\|\boldsymbol{w}^{(t)}\|_1} \leq -\kappa\alpha \leq \frac{\langle \boldsymbol{g}^{(t)}, \Delta \rangle}{\|\boldsymbol{\ell}^{(t)} \circ \Delta\|_1} \leq \kappa^{(9.1)} \cdot \frac{\langle \boldsymbol{g}^{(t)}, \boldsymbol{c}^{(t)} \rangle}{\sum_{i=0}^{d} \|\boldsymbol{w}^{(\text{prev}_i^{(t)})}\|_1},$$

so $\sum_{i=0}^{d} \|\boldsymbol{w}^{(\text{prev}_i^{(t)})}\|_1 \geq 2(d+1)\|\boldsymbol{w}^{(t)}\|_1$. Our algorithm for Theorem 8.2 is then to implement the player's strategy in Lemma 10.3 on top of Theorem 9.1.

Correctness of Query() follows by definition. To bound the runtime, by Theorem 9.1 we can take the constants $d = \log^{1/8} m$, $k = \exp(O(\log^{7/8} m))$, $\gamma_g = \exp(-O(\log^{7/8} m \log\log m))$, $C_r = \exp(O(\log^{7/8} m \log\log m))$, $T = (m+Q)\exp(O(\log^{7/8} m \log\log m))$. Thus by Lemma 10.3 the total runtime to execute the player's algorithm is $(m+Q)\exp(O(\log^{7/8} m \log\log m))$ as desired. □

11 COMPUTING THE MIN-COST FLOW VIA MIN-RATIO CYCLES

In this section we given the full pseudocode for proving Theorem 1.1, modulo getting an initial point and final point, which are explained in Lemmas 6.11 and 6.12.

We explain the implementation in procedure MinCostFlow given in Algorithm 7. The algorithm maintains approximate lengths $\boldsymbol{\ell}^{(t)}$ and gradients $\boldsymbol{g}^{(t)}$, updating them when the dynamic tree data structures $\mathcal{D}^{(T_i)}$ report that some edge has accumulated many changes. It updates these lengths and gradients, and passes the result to a data structure $\mathcal{D}^{(HSFC)}$ which dynamically maintains the trees $T_1, \ldots, T_s$ and a min-ratio cycle on them under hidden stable-flow chasing updates. We check that the updates to $\boldsymbol{\ell}^{(t)}, \boldsymbol{g}^{(t)}$ are indeed hidden stable-flow chasing in Lemma 11.2. Finally, the $\mathcal{D}^{(HSFC)}$ rebuilds itself every εm iterations, after which the residual cost $\boldsymbol{c}^\top \boldsymbol{f} - F^*$ might have changed by a $1+\varepsilon$ factor. It terminates when $\boldsymbol{c}^\top \boldsymbol{f} - F^* \leq (mU)^{-10}$, which happens within $m^{1+o(1)}$ iterations.

Algorithm 7: MinCostFlow$(G, \boldsymbol{d}, \boldsymbol{c}, \boldsymbol{u}^+, \boldsymbol{u}^-, \boldsymbol{f}^{(0)}, F^*)$. Takes graph G, demands $\boldsymbol{d}$, costs $\boldsymbol{c}$, upper/lower capacities $\boldsymbol{u}^+, \boldsymbol{u}^-$, initial feasible flow $\boldsymbol{f}^{(0)}$ (Lemma 6.12), and guess of the optimal flow F^*

global variables
- $\alpha \leftarrow 1/(1000 \log mU)$
- $\kappa \leftarrow \exp(-O(\log^{7/8} m \log\log m))$; // Approximation quality in Theorem 8.2
- $d \leftarrow O(\log^{1/8} n)$; // Data structure depth
- $\mathcal{D}^{(HSFC)}$; // Hidden Stable-Flow Chasing (HSFC) data structure in Theorem 8.2
- $T_1, T_2, \ldots, T_s$ for $s \leftarrow O(\log n)^d$; // Trees maintained by data structure $\mathcal{D}^{(HSFC)}$
- $\varepsilon \leftarrow \kappa\alpha/(1000s)$. ; // Error tolerated within each tree.
- $\mathcal{D}^{(T_i)}$; // Dynamic tree data structure for trees T_i
- $\boldsymbol{f}_1, \ldots, \boldsymbol{f}_s \leftarrow \vec{0} \in \mathbb{R}^E$ and $\boldsymbol{f} \stackrel{\text{def}}{=} \boldsymbol{f}^{(0)} + \sum_{i\in[s]} \boldsymbol{f}_i$; // Flows on trees T_i
- $\widetilde{\boldsymbol{f}}^{(t)}$; // Approximate flow at stage t, remembers which edges have been updated.
- $\boldsymbol{f}^{(t)} \leftarrow \boldsymbol{f}^{(0)}$; // Total flow at stage t, implicitly stored
- $r \leftarrow \infty$; // Estimate of cost difference from optimal.

procedure MinCostFlow$(G, \boldsymbol{d}, \boldsymbol{c}, \boldsymbol{u}^+, \boldsymbol{u}^-, \boldsymbol{f}^{(0)}, F^*)$
- **while** $\boldsymbol{c}^\top \boldsymbol{f}^{(t)} - F^* \geq (mU)^{-10}$ **do**
  - **if** t *is a multiple of* $\lfloor \varepsilon m \rfloor$ **then**
    - Explicitly compute $\boldsymbol{f}^{(t)} \leftarrow \boldsymbol{f}^{(0)} + \sum_{i\in[s]} \boldsymbol{f}_i$, $\widetilde{\boldsymbol{f}}^{(t)} \leftarrow \boldsymbol{f}^{(t)}$.
    - $r \leftarrow \boldsymbol{c}^\top \widetilde{\boldsymbol{f}}^{(t)} - F^*$. ; // Cost difference from optimal.
    - $\boldsymbol{g}^{(t)} \leftarrow \boldsymbol{g}(\widetilde{\boldsymbol{f}}^{(t)}), \boldsymbol{\ell}^{(t)} \leftarrow \boldsymbol{\ell}(\widetilde{\boldsymbol{f}}^{(t)})$; // Definition 6.2
    - Rebuild $\mathcal{D}^{(HSFC)}$ and update the T_i. ; // Because r may have changed by a $1+\varepsilon$ factor.
  - $U^{(t)} \leftarrow \bigcup_{i\in[s]} \mathcal{D}^{(T_i)}$.Detect() ; // Lemma 5.3
  - **foreach** $e \in U^{(t)}$ **do**
    - Set $\widetilde{\boldsymbol{f}}_e^{(t)} \leftarrow \boldsymbol{f}_e^{(t)} = \boldsymbol{f}_e^{(0)} + \sum_{i\in[s]} (\boldsymbol{f}_i)_e$, $\boldsymbol{\ell}_e^{(t)} \leftarrow \boldsymbol{\ell}(\widetilde{\boldsymbol{f}}^{(t)})_e$; // Definition 6.2
    - $\boldsymbol{g}_e^{(t)} \leftarrow 20m\boldsymbol{c}_e/r + \alpha(\boldsymbol{u}_e^+ - \widetilde{\boldsymbol{f}}_e^{(t)})^{-1-\alpha} - \alpha(\widetilde{\boldsymbol{f}}_e^{(t)} - \boldsymbol{u}_e^-)^{-1-\alpha}$

  /* No change to $e \notin U^{(t)}$ */
  - **foreach** $e \notin U^{(t)}$ **do** $\boldsymbol{g}_e^{(t)} \leftarrow \boldsymbol{g}_e^{(t-1)}, \boldsymbol{\ell}_e^{(t)} \leftarrow \boldsymbol{\ell}_e^{(t-1)}, \widetilde{\boldsymbol{f}}_e^{(t)} \leftarrow \widetilde{\boldsymbol{f}}_e^{(t-1)}$.;
  - $\mathcal{D}^{(HSFC)}$.Update$(U^{(t)}, \boldsymbol{g}^{(t)}, \boldsymbol{\ell}^{(t)})$, and update the T_i for $i \in [s]$; // Theorem 8.2
  - $(i, \Delta) \leftarrow \mathcal{D}^{(HSFC)}$.Query(), where $i \in [s]$ and $\Delta = (u_1, v_1) \oplus T_i[v_1, u_2] \oplus (u_2, v_2) \oplus \cdots \oplus (u_l, v_l) \oplus T_i[v_l, u_1]$ for edges (u_i, v_i) and $l \leq m^{o(1)}$. ; // Δ represented via $m^{o(1)}$ off-tree edges and paths on T_i
  - $\Delta \leftarrow \eta\Delta$ for $\eta \leftarrow -\kappa^2\alpha^2/(800\langle \boldsymbol{g}^{(t)}, \Delta\rangle)$; // Scale Δ so $\langle \boldsymbol{g}^{(t)}, \Delta\rangle = -\kappa^2\alpha^2/800$
  - $\boldsymbol{f}_i \leftarrow \boldsymbol{f}_i + \Delta$ using $\mathcal{D}^{(T_i)}$, Lemma 5.3 item 3 ; // Implicitly set $\boldsymbol{f}^{(t)} \stackrel{\text{def}}{=} \boldsymbol{f}^{(t-1)} + \Delta$
  - $t \leftarrow t + 1$.
- **return** $\boldsymbol{f}^{(t)}$

To analyze the progress of the algorithm, we will show that MinCostFlow (Algorithm 7) satisfies the hypotheses of our main IPM result Theorem 6.3. Thus, applying Theorem 6.3 shows that MinCostFlow (Algorithm 7) computes a mincost flow to high accuracy in $\widetilde{O}(m\kappa^{-2})$ iterations for some $\kappa = m^{-o(1)}$.

We first note that $\boldsymbol{\ell}^{(t)}$ and $\boldsymbol{g}^{(t)}$ are approximately correct lengths and gradients at all times.

Lemma 11.1 (Stability in MinCostFlow). *During a call to MinCostFlow (Algorithm 7), for $\boldsymbol{f}^{(t)} \leftarrow \boldsymbol{f}^{(0)} + \sum_{i\in[s]} \boldsymbol{f}_i$, we have $\boldsymbol{\ell}^{(t)} \approx_{1.1} \boldsymbol{\ell}(\boldsymbol{f}^{(t)})$, for r defined in line 17, $r \approx_{1+\varepsilon} \boldsymbol{c}^\top \boldsymbol{f}^{(t)} - F^*$, and*

$$\left\| \boldsymbol{L}(\boldsymbol{f}^{(t)})^{-1}(\boldsymbol{g}^{(t)} - (\boldsymbol{c}^\top \boldsymbol{f}^{(t)} - F^*)/r \cdot \boldsymbol{g}(\boldsymbol{f}^{(t)})) \right\|_\infty \le 10s\varepsilon = \alpha\kappa/100.$$

Proof. To show $\boldsymbol{\ell}^{(t)} \approx_{1.1} \boldsymbol{\ell}(\boldsymbol{f}^{(t)})$ it suffices to check that $\boldsymbol{f} \leftarrow \widetilde{\boldsymbol{f}}^{(t)}$ and $\overline{\boldsymbol{f}} \leftarrow \boldsymbol{f}^{(t)}$ satisfy the hypotheses of Lemma 6.9, precisely $\|\boldsymbol{L}(\boldsymbol{f}^{(t)})(\boldsymbol{f}^{(t)} - \widetilde{\boldsymbol{f}}^{(t)})\|_\infty \le s\varepsilon$. Indeed, this follows directly by the guarantees of Detect in Lemma 5.3 and the fact there are s trees, because if no tree returned e, then the total error is at most $s\varepsilon$.

To show the bound on the gradient, we use Lemma 6.10. Because we have argue above that $\|\boldsymbol{L}(\boldsymbol{f}^{(t)})^{-1}(\boldsymbol{f}^{(t)} - \widetilde{\boldsymbol{f}}^{(t)})\|_\infty \le s\varepsilon$, it suffices to check that $r \approx_{1+\varepsilon} \boldsymbol{c}^\top \boldsymbol{f}^{(t)} - F^*$. Recall that r is reset every $\lfloor \varepsilon m \rfloor$ iterations in line 17 of Algorithm 7. For the scaled circulation $\boldsymbol{\Delta}$ in line 27, Lemma 6.8, for $\widetilde{\boldsymbol{g}} = \boldsymbol{g}^{(t)}$ and $\widetilde{\boldsymbol{\ell}} = \boldsymbol{\ell}^{(t)}$ in Algorithm 7, tells us

$$\frac{|\boldsymbol{c}^\top \boldsymbol{\Delta}|}{\boldsymbol{c}^\top \boldsymbol{f}^{(t)} - F^*} \le |\boldsymbol{g}^{(t)} \boldsymbol{\Delta}|/(\kappa m) \le \alpha^2\kappa/(800m) \le 1/(800m),$$

where the hypotheses of Lemma 6.8 are satisfied because of the guarantee of $\mathcal{D}^{(HSFC)}$.Query() (Theorem 8.2), and we used the bound on $|\boldsymbol{g}^{(t)}\boldsymbol{\Delta}|$ from line 27 of Algorithm 7. Hence over εm iterations, $\boldsymbol{c}^\top \boldsymbol{f}^{(t)} - F^*$ can change by at most a $(1 + 1/(800m))^{\varepsilon m} \le 1 + \varepsilon$ factor, as desired. □

Our next goal is to define circulations $\boldsymbol{c}^{(t)}$ and upper bounds $\boldsymbol{w}^{(t)}$ to make $\boldsymbol{g}^{(t)}, \boldsymbol{\ell}^{(t)}, \boldsymbol{c}^{(t)}, \boldsymbol{w}^{(t)}$ as defined in MinCostFlow (Algorithm 7) satisfy the hidden stable-flow chasing property. This shows that the solutions $\boldsymbol{\Delta}$ returned by the data structure have a good ratio.

Lemma 11.2. *Let $\boldsymbol{g}^{(t)}, \boldsymbol{\ell}^{(t)}, U^{(t)}$ be defined as in an execution of MinCostFlow (Algorithm 7). For $\boldsymbol{f}^* \stackrel{\text{def}}{=} \arg\min_{\boldsymbol{B}^\top \boldsymbol{f} = \boldsymbol{d}} \boldsymbol{c}^\top \boldsymbol{f}$, let $\boldsymbol{c}^{(t)} \stackrel{\text{def}}{=} \boldsymbol{f}^* - \boldsymbol{f}$ and $\boldsymbol{w}^{(t)} = 50 + |\boldsymbol{\ell}^{(t)} \circ \boldsymbol{c}^{(t)}|$. Then $\boldsymbol{g}^{(t)}, \boldsymbol{\ell}^{(t)}, U^{(t)}$ satisfy the hidden stable-flow chasing (Definition 8.1) with circulations $\boldsymbol{c}^{(t)}$ and upper bounds $\boldsymbol{w}^{(t)}$.*

Proof. We check each item of Definition 8.1 carefully. For the circulation condition in item 1, note that $\boldsymbol{B}^\top \boldsymbol{c}^{(t)} = \boldsymbol{B}^\top \boldsymbol{f}^* - \boldsymbol{B}^\top \boldsymbol{f}^{(t)} = \boldsymbol{d} - \boldsymbol{d} = 0$ because $\boldsymbol{f}^*$ and $\boldsymbol{f}^{(t)}$ both route the demand $\boldsymbol{d}$. For the width condition in item 2, by the definition of $\boldsymbol{w}^{(t)} = 50 + |\boldsymbol{\ell}^{(t)} \circ \boldsymbol{c}^{(t)}|$ we trivially have $|\boldsymbol{\ell}^{(t)} \circ \boldsymbol{c}^{(t)}| \le \boldsymbol{w}^{(t)}$ coordinate-wise.

To check that the upper bounds $\boldsymbol{w}^{(t)}$ are stable (item 3), for an edge e let $t' \in [\text{last}_e^{(t)}, t]$ be so that e was not updated by any $U^{(t)}$ since stage t'. By the guarantees of Detect we know that

$$\boldsymbol{\ell}_e^{(t)} |\boldsymbol{f}_e^{(t)} - \boldsymbol{f}_e^{(t')}| \le \varepsilon.$$

Hence we get that

$$|\boldsymbol{w}_e^{(t)} - \boldsymbol{w}_e^{(t')}| \stackrel{(i)}{=} \boldsymbol{\ell}_e^{(t)} |\boldsymbol{f}_e^{(t)} - \boldsymbol{f}_e^{(t')}| \le \varepsilon \stackrel{(ii)}{\le} 1/100 |\boldsymbol{w}_e^{(t')}|,$$

Here, (i) follows because $\boldsymbol{\ell}_e^{(t)} = \boldsymbol{\ell}_e^{(\text{last}_e^{(t)})}$ because e was not updated in any $U^{(t)}$ since stage $\text{last}_e^{(t)}$ and (ii) is because $\boldsymbol{w}_e^{(t')} \ge 50$ for all t'. Hence $|\boldsymbol{w}_e^{(t)}| \le 1.1 |\boldsymbol{w}_e^{(t')}|$ as desired.

To check that the lengths and widths are quasipolynomially bounded for item 4, note that

$$\min_{e\in E} \{\boldsymbol{u}_e^+ - \boldsymbol{f}_e, \boldsymbol{f}_e - \boldsymbol{u}_e^-\} \ge \Phi(\boldsymbol{f}^{(t)})^{-1/\alpha} \ge \exp(-O(\log^2 mU)),$$

by our assumption that $\Phi(\boldsymbol{f}^{(t)}) \le \widetilde{O}(m)$ always. Also, $|\boldsymbol{c}_e^{(t)}| \le O(U)$ for all $e \in E$. This shows that $\log \boldsymbol{\ell}_e^{(t)}, \log \boldsymbol{w}_e^{(t)} \le O(\log^2 mU)$ for all $e \in E$. The lower bound $\boldsymbol{w}_e^{(t)} \ge 50$ is by definition. Also, the lower bound $\boldsymbol{\ell}_e^{(t)} \ge 1/U^{1+\alpha} \ge 1/U^2$ is trivial by the definition of $\boldsymbol{\ell}_e^{(t)}$. □

As a result, we deduce that $\mathcal{D}^{(HSFC)}$ succeeds whp. This allows us to prove that MinCostFlow satisfies the hypotheses of Algorithm 7, and allows us to bound the total number of iterations.

Lemma 11.3. *An execution of MinCostFlow (Algorithm 7) runs for $\widetilde{O}(m\kappa^{-2}\alpha^{-2})$ iterations.*

Proof. We will define $\widetilde{\boldsymbol{g}}, \widetilde{\boldsymbol{\ell}}, \boldsymbol{\Delta}, \eta$ and flows $\boldsymbol{f}^{(t)}$ to show that an execution of MinCostFlow (Algorithm 7) satisfies the hypotheses of Theorem 6.3, which implies that MinCostFlow (Algorithm 7) terminates in $\widetilde{O}(m\kappa^{-2}\alpha^{-2})$ iterations.

For $\boldsymbol{f}^{(t)}$ as defined in MinCostFlow (Algorithm 7), note that $\Phi(\boldsymbol{f}^{(t)}) \le 200m\log(mU)$ at all times. This is because it holds at the initial point $\boldsymbol{f}^{(0)}$ (Lemma 6.12) and the potential is decreasing.

Next we define $\widetilde{\boldsymbol{g}}, \widetilde{\boldsymbol{\ell}}$, the approximate gradients and lengths. Let $\widetilde{\boldsymbol{g}} = r/(\boldsymbol{c}^\top \boldsymbol{f}^{(t)} - F^*) \cdot \boldsymbol{g}^{(t)}$ and $\widetilde{\boldsymbol{\ell}} = \boldsymbol{\ell}^{(t)}$ for $\boldsymbol{g}^{(t)}, \boldsymbol{\ell}^{(t)}$ as defined in MinCostFlow (Algorithm 7). By Lemma 11.1 we know that $\widetilde{\boldsymbol{\ell}} \approx_{1.1} \boldsymbol{\ell}(\boldsymbol{f}^{(t)})$ and

$$\begin{aligned}\|\boldsymbol{L}(\boldsymbol{f}^{(t)})^{-1}(\widetilde{\boldsymbol{g}} - \boldsymbol{g}(\boldsymbol{f}^{(t)}))\|_\infty &= r/(\boldsymbol{c}^\top \boldsymbol{f}^{(t)} - F^*)\left\|\boldsymbol{L}(\boldsymbol{f}^{(t)})^{-1}(\boldsymbol{g}^{(t)} - (\boldsymbol{c}^\top \boldsymbol{f}^{(t)} - F^*)/r \cdot \boldsymbol{g}(\boldsymbol{f}^{(t)}))\right\|_\infty \\ &\overset{(i)}{\le} (1+\varepsilon)\alpha\kappa/100 \le \alpha\kappa/50,\end{aligned}$$

where (i) uses the bounds on r and $\boldsymbol{g}^{(t)}$ in Lemma 11.1. Thus for $\boldsymbol{c}^{(t)} = \boldsymbol{f}^* - \boldsymbol{f}^{(t)}$ as in Lemma 11.2,

$$\frac{\boldsymbol{g}^{(t)\top}\boldsymbol{c}^{(t)}}{\|\boldsymbol{w}^{(t)}\|_1} = (\boldsymbol{c}^\top \boldsymbol{f}^{(t)} - F^*)/r \cdot \frac{\widetilde{\boldsymbol{g}}^\top(\boldsymbol{f}^* - \boldsymbol{f}^{(t)})}{50m + \|\boldsymbol{L}^{(t)}\boldsymbol{c}^{(t)}\|_1} \overset{(i)}{\le} -(1-\varepsilon)\alpha/4 \le -\alpha/8,$$

where (i) follows from the first item of Theorem 6.3 (Lemma 6.7) and $r \approx_{1+\varepsilon} \boldsymbol{c}^\top \boldsymbol{f}^{(t)} - F^*$ from Lemma 11.1. Hence, by the guarantees of $\mathcal{D}^{(HSFC)}$.Query() (Theorem 8.2) as called in line 26 of MinCostFlow (Algorithm 7), we know that whp.

$$\frac{\boldsymbol{g}^{(t)\top}\boldsymbol{\Delta}}{\|\boldsymbol{L}^{(t)}\boldsymbol{\Delta}\|_1} \le -\kappa\alpha/8. \tag{45}$$

Thus, for the scaling η as in MinCostFlow (Algorithm 7), Theorem 6.3 (where we change κ to $\alpha\kappa/8$ for this setting) shows that the algorithm computes a high-accuracy flow in $\widetilde{O}(m\kappa^{-2}\alpha^{-2})$ iterations. □

The final piece is to analyze the runtime of MinCostFlow (Algorithm 7) by bounding the total size of the update batches $U^{(t)}$ as defined in line 20.

Lemma 11.4. *Consider a call to MinCostFlow (Algorithm 7) and let $U^{(t)}$ be as in line 20. Then $\sum_t |U^{(t)}| \le \widetilde{O}(m\kappa^{-2}\alpha^{-2}\varepsilon^{-1}) \le m^{1+o(1)}$.*

Proof. Because of the guarantee in line 27 of Algorithm 7, we know that $|\boldsymbol{g}^{(t)\top}\boldsymbol{\Delta}| = \kappa^2\alpha^2/800$. Additionally by the guarantees of $\mathcal{D}^{(HSFC)}$.Query() (Theorem 8.2) as called in line 26 of MinCostFlow (Algorithm 7), we get that (see (45))

$$\|\boldsymbol{L}^{(t)}\boldsymbol{\Delta}\|_1 \le 8/(\kappa\alpha)|\boldsymbol{g}^{(t)\top}\boldsymbol{\Delta}| \le 1.$$

Hence the sum of $\|\boldsymbol{L}^{(t)}\boldsymbol{\Delta}\|_1$ over all iterations is $\widetilde{O}(m\kappa^{-2}\alpha^{-2})$ by our bound on the number of iterations in Lemma 11.3. Each time an update on edge e in $U^{(t)}$ it contributes $\Omega(\varepsilon)$ to this sum by the guarantees of Detect in Lemma 5.3. Hence $\sum_t |U^{(t)}| \le \widetilde{O}(m\kappa^{-2}\alpha^{-2}\varepsilon^{-1})$. □

Combining these pieces shows our main result Theorem 1.1 on computing min-cost flows.

Proof of Theorem 1.1. Given a min-cost flow instance, we will first use Lemmas 6.11 and 6.12 to compute the initial flow $\boldsymbol{f}^{(0)}$. We then run MinCostFlow with initial flow $\boldsymbol{f}^{(0)}$, and use Lemma 6.11 to round to an exact min-cost flow.

The only remaining piece to analyze is the runtime. The main component of the runtime is the data structure $\mathcal{D}^{(HSFC)}$ (Theorem 8.2). The inputs $\boldsymbol{g}^{(t)}, \boldsymbol{\ell}^{(t)}, U^{(t)}$ to $\mathcal{D}^{(HSFC)}$ satisfy the hidden stable-flow chasing property by Lemma 11.2. Hence the data structure $\mathcal{D}^{(HSFC)}$ runs in total time $\varepsilon^{-1}(m+Q)m^{o(1)} = m^{1+o(1)}$ time by Theorem 8.2, because the data structure reinitializes $\widetilde{O}(\varepsilon^{-1})$ times (in line 15), and $Q = \sum_t |U^{(t)}| \leq m^{1+o(1)}$ by Lemma 11.4.

The remaining runtime components can be handled in $sm^{o(1)} = m^{o(1)}$ time per operation by using dynamic trees (Lemma 5.3), as there are s trees, and the fact that the cycle in line 26 of MinCostFlow (Algorithm 7) is represented by $\exp(O(\log^{7/8} m \log\log m)) \leq m^{o(1)}$ paths, so the total runtime is $m^{1+o(1)}$ as desired. □

12 GENERAL CONVEX OBJECTIVES

The goal of this section is to extend our algorithms to the setting of optimizing single commodity flows for general decomposable convex objectives.

12.1 General Setup for General Convex Objectives

Formally, for a graph $G = (V, E)$ let $h_e : \mathbb{R} \to \mathbb{R} \cup \{+\infty\}$ be convex functions. For a flow $\boldsymbol{f}$ let $h(\boldsymbol{f}) \stackrel{\text{def}}{=} \sum_{e\in E} h_e(\boldsymbol{f}_e)$. Our goal is to minimize $h(\boldsymbol{f})$ over all flows $\boldsymbol{f}$ routing a demand $\boldsymbol{d}$, i.e. $\boldsymbol{B}^\top \boldsymbol{f} = \boldsymbol{d}$.

We cast this in the setting of empirical risk minimization (see [89]) by introducing new variables $\boldsymbol{y} \in \mathbb{R}^E$ and convex sets $\mathcal{X}_e \stackrel{\text{def}}{=} \{(f, y) : y < h_e(f)\}$:

$$\min_{\boldsymbol{B}^\top \boldsymbol{f}=\boldsymbol{d}} h(\boldsymbol{f}) = \min_{\substack{\boldsymbol{B}^\top \boldsymbol{f}=\boldsymbol{d} \\ \boldsymbol{y}\in\mathbb{R}^E : \boldsymbol{y}_e \leq h(\boldsymbol{f}_e) \text{ for all } e\in E}} \mathbf{1}^\top \boldsymbol{y} = \min_{\substack{\boldsymbol{B}^\top \boldsymbol{f}=\boldsymbol{d} \\ (\boldsymbol{f}_e, \boldsymbol{y}_e)\in\mathcal{X}_e \text{ for all } e\in E}} \mathbf{1}^\top \boldsymbol{y}. \tag{46}$$

Let $F^* \stackrel{\text{def}}{=} \min_{\boldsymbol{B}^\top \boldsymbol{f}=\boldsymbol{d}} h(\boldsymbol{f})$. We will assume that we have access to gradients and Hessians of ν-self-concordant barriers for $\mathcal{X}_e, \psi_e : \mathcal{X}_e \to \mathbb{R}$. Explicit self-concordant barriers are known for several natural objectives h_e (see e.g. Chapter 9.6. of [16], or Section 4 of [107]), and it is known that every subset $\mathcal{X} \subseteq \mathbb{R}^n$ admits an n-self-concordant barrier [25, 91, 107, 108].

We now formally introduce the definition of self-concordance.

Definition 12.1 (ν-self-concordance [108, Definition 4.2.2]). *We say that a function $\psi : \mathcal{X} \to \mathbb{R}$ on an open set $\mathcal{X} \subseteq \mathbb{R}^n$ is a self-concordant barrier if ψ is convex, $\psi(\boldsymbol{x}) \to \infty$ as $\boldsymbol{x}$ approaches the boundary of $\mathcal{X}$, and for all $\boldsymbol{x} \in \mathcal{X}$ and $\boldsymbol{v} \in \mathbb{R}^n$*

$$\left|\nabla^3 \psi(\boldsymbol{x})\left[\boldsymbol{v}, \boldsymbol{v}, \boldsymbol{v}\right]\right| \leq 2\left(\boldsymbol{v}^\top \nabla^2 \psi\left(\boldsymbol{x}\right) \boldsymbol{v}\right)^{3/2}.$$

We say that f is ν-self-concordant for some $\nu > 0$ if f is self-concordant and for all $\boldsymbol{x} \in \mathcal{X}$ and $\boldsymbol{v} \in \mathbb{R}^n$ we have

$$\left\langle \nabla \psi\left(\boldsymbol{x}\right), \boldsymbol{v}\right\rangle^2 \leq \nu \boldsymbol{v}^\top \nabla^2 \psi\left(\boldsymbol{x}\right) \boldsymbol{v}.$$

Analyzing the runtime of our algorithm requires assuming that various quantities are quasipolynomially bounded such as the starting flow, demands, and convex objectives, and the underlying self-concordant barriers.

Assumption 12.2. *We make the following assumptions for our method, for a parameter $K = \widetilde{O}(1)$.*

(1) *We have access in $\widetilde{O}(1)$ time to gradients/Hessians of the self-concordant barriers $\psi_e(\boldsymbol{f}_e, \boldsymbol{y}_e)$.*

(2) *All capacities, demands, and costs are polynomially bounded, i.e.* $|f_e| \le m^K$ *for all* e, $\|\boldsymbol{d}\|_\infty \le m^K$, *and* $|h_e(x)| \le O(m^K + |x|^K)$ *for all* $x \in \mathbb{R}$.

(3) *We shift the barriers* $\Psi(\boldsymbol{f}_e, \boldsymbol{y}_e)$ *such that* $\inf_{|f_e|, |\boldsymbol{y}_e| \le m^K} \Psi(\boldsymbol{f}_e, \boldsymbol{y}_e) = 0$. *We can shift the barriers because that does not affect self-concordance. This implies that* $\overline{\zeta}_e(\boldsymbol{f}_e) \ge 1$ *on the whole domain.*

(4) *There is a feasible flow* $\boldsymbol{f}^{(0)}$ *and variables* $|\boldsymbol{y}_e^{(0)}| \le m^K$ *such that* $m^{-K}\boldsymbol{I} \preceq \nabla^2 \psi_e(\boldsymbol{f}_e^{(0)}, \boldsymbol{y}(0)_e) \preceq m^K \boldsymbol{I}$ *for all* e, *and* $\psi_e(\boldsymbol{f}_e^{(0)}, \boldsymbol{y}_e^{(0)}) \le K$.

(5) *The parameters* $\alpha, \varepsilon, \kappa$ *used throughout are all less than* $1/(1000\nu)$.

(6) *The Hessian is quasipolynomially bounded as long as the function value is* $\widetilde{O}(1)$ *bounded, i.e. for all points* $|f_e|, |\boldsymbol{y}_e| \le m^K$ *with* $\psi_e(\boldsymbol{f}_e, \boldsymbol{y}_e) \le \widetilde{O}(1)$, *we have* $\nabla^2 \psi_e(\boldsymbol{f}_e, \boldsymbol{y}_e) \preceq \exp(\log^{O(1)} m)\boldsymbol{I}$.

We assume everything stated above for the remainder of the section. The final assumption in item 6 is to ensure that all lengths/gradients encountered in the algorithm are bounded by $\exp(\log^{O(1)} m)$. This holds for all explicit $O(1)$-self-concordant barriers we have encountered, such as those for entropy-regularized optimal transport, matrix scaling, and normed flows.

We make direct use of the following lemmas from [108].

Lemma 12.3 ([108, Theorem 4.2.4], first part). *For a self concordant function* f, *and any* $\boldsymbol{x}$ *and* $\boldsymbol{y}$ *in its domain, we have*

$$\langle \nabla f(\boldsymbol{x}), \boldsymbol{y} - \boldsymbol{x} \rangle < \nu.$$

Lemma 12.4 ([108, Theorem 4.1.7], first part). *For a self concordant function* f, *and any* $\boldsymbol{x}$ *and* $\boldsymbol{y}$ *in its domain, we have*

$$\langle \nabla f(\boldsymbol{y}) - \nabla f(\boldsymbol{x}), \boldsymbol{y} - \boldsymbol{x} \rangle \ge \frac{\|\boldsymbol{x} - \boldsymbol{y}\|^2_{\nabla^2 f(\boldsymbol{x})}}{1 + \|\boldsymbol{x} - \boldsymbol{y}\|_{\nabla^2 f(\boldsymbol{x})}}.$$

Lemma 12.5 ([108, Theorem 4.1.6]). *For a self concordant function* f, *and any* $\boldsymbol{x}$ *and* $\boldsymbol{y}$ *in its domain such that*

$$\|\boldsymbol{x} - \boldsymbol{y}\|_{\nabla^2 f(\boldsymbol{x})} < 1$$

we have

$$\left(1 - \|\boldsymbol{x} - \boldsymbol{y}\|_{\nabla^2 f(\boldsymbol{x})}\right)^2 \nabla^2 f(\boldsymbol{x}) \preceq \nabla^2 f(\boldsymbol{y}) \preceq \frac{1}{\left(1 - \|\boldsymbol{x} - \boldsymbol{y}\|_{\nabla^2 f(\boldsymbol{x})}\right)^2} \nabla^2 f(\boldsymbol{x})$$

Fix some $\alpha \in (0, 1/10)$, set a path parameter t and minimize the following objective

$$\Psi_t(\boldsymbol{f}, \boldsymbol{y}) \stackrel{\text{def}}{=} t \cdot \mathbf{1}^\top \boldsymbol{y} + \sum_{e \in E} \exp(\alpha \psi_e(\boldsymbol{f}_e, \boldsymbol{y}_e)) = \sum_{e \in E} \left(t\boldsymbol{y}_e + \exp(\alpha \psi_e(\boldsymbol{f}_e, \boldsymbol{y}_e))\right),$$

over $\boldsymbol{B}^\top \boldsymbol{f} = \boldsymbol{d}$. This is analogous to our α-power potential in Equation 9 at the start of Section 6.

Note that for a fixed flow $\boldsymbol{f}$, we can eliminate the variables $\boldsymbol{y}$ in the following way. We should set $\boldsymbol{y}_e = y_e(\boldsymbol{f}_e)$ for $y_e(\boldsymbol{f}_e) \stackrel{\text{def}}{=} \arg\min_y ty + \exp(\alpha \psi_e(\boldsymbol{f}_e, y))$. Thus we can write

$$\min_{\boldsymbol{B}^\top \boldsymbol{f} = \boldsymbol{d}, \boldsymbol{y}} \Psi_t(\boldsymbol{f}, \boldsymbol{y}) = \min_{\boldsymbol{B}^\top \boldsymbol{f} = \boldsymbol{d}} \sum_{e \in E} \left(t y_e(\boldsymbol{f}_e) + \exp(\alpha \psi_e(\boldsymbol{f}_e, y_e(\boldsymbol{f}_e)))\right),$$

Let $\overline{\zeta}_e(\boldsymbol{f}_e) \stackrel{\text{def}}{=} \exp(\alpha \psi_e(\boldsymbol{f}_e, y(\boldsymbol{f}_e)))$, and $\zeta_e(\boldsymbol{f}_e) = t y_e(\boldsymbol{f}_e) + \overline{\zeta}_e(\boldsymbol{f}_e)$ and define the potential

$$Z_t(\boldsymbol{f}) \stackrel{\text{def}}{=} \sum_{e \in E} \zeta_e(\boldsymbol{f}_e). \tag{47}$$

Our first main lemma (Lemma 12.8) will be that up to scaling, the function ζ_e is self-concordant. To show this, we start by studying the derivatives of the function $ty_e(f_e) + \zeta_e(f_e)$.

Definition 12.6. *For a function $\psi : \mathbb{R}^n \to \mathbb{R}$, and a sequence $(i_1, i_2, \ldots, i_k) \in [n]^k$, define the mixed partials*

$$\psi_{x_{i_1},\ldots,x_{i_k}} = \frac{\partial}{\partial x_{i_1}} \cdots \frac{\partial}{\partial x_{i_k}} \psi.$$

Lemma 12.7. *Let $f : \mathcal{X} \to \mathbb{R}$ be a convex function on an open set $\mathcal{X} \subseteq \mathbb{R}^2$. For $x \in \mathbb{R}$ let $y(x) \stackrel{\text{def}}{=} \operatorname{argmin}_y \psi(x, y)$. Let $\zeta(x) \stackrel{\text{def}}{=} \psi(x, y(x))$. Then for $\boldsymbol{v} \stackrel{\text{def}}{=} \begin{bmatrix} 1 \\ y'(x) \end{bmatrix}$,*

$$\zeta'(x) = \langle \nabla \psi(x, y(x)), \boldsymbol{v} \rangle = f_x(x, y(x)), \tag{48}$$

$$\zeta''(x) = \boldsymbol{v}^\top \nabla^2 \psi(x, y(x)) \boldsymbol{v}, \tag{49}$$

$$\zeta'''(x) = \nabla^3 \psi(x, y(x))[\boldsymbol{v}, \boldsymbol{v}, \boldsymbol{v}]. \tag{50}$$

$$y'(x) = -\psi_{xy}(x, y(x)) / \psi_{yy}(x, y(x)). \tag{51}$$

Proof. Note that $\psi_y(x, y(x)) = 0$ by the optimality of $y(x)$. By the chain rule for total derivatives

$$\zeta'(x) = f_x(x, y(x)) + \psi_y(x, y(x)) y'(x) = \langle \nabla \psi(x, y(x)), \boldsymbol{v} \rangle$$

which shows the first equality (48).

Taking the derivative of the first equality of (48) gives us

$$\begin{aligned} \zeta''(x) &= \psi_{xx}(x, y(x)) + 2\psi_{xy}(x, y(x)) y'(x) + \psi_{yy}(x, y(x)) y'(x)^2 + f_y(x, y(x)) y''(x) \\ &= \psi_{xx}(x, y(x)) + 2\psi_{xy}(x, y(x)) y'(x) + \psi_{yy}(x, y(x)) y'(x)^2 = \boldsymbol{v}^\top \nabla^2 \psi(x, y(x)) \boldsymbol{v}, \end{aligned}$$

where we have used that $\psi_y(x, y(x)) = 0$. This shows (49).

Taking the derivative of (49) gives

$$\begin{aligned} \zeta'''(x) = {} & \psi_{xxx}(x, y(x)) + 3\psi_{xxy}(x, y(x)) y'(x) + 3\psi_{xyy}(x, y(x)) y'(x)^2 + \psi_{yyy}(x, y(x)) y'(x)^3 \\ & + 2 \left(\psi_{xy}(x, y(x)) + \psi_{yy}(x, y(x)) y'(x) \right) y''(x). \end{aligned}$$

However, note that taking the derivative of the identity $\psi_y(x, y(x)) = 0$ gives us

$$\psi_{xy}(x, y(x)) + \psi_{yy}(x, y(x)) y'(x) = 0.$$

Plugging this into the above gives us

$$\begin{aligned} \zeta'''(x) &= \psi_{xxx}(x, y(x)) + 3\psi_{xxy}(x, y(x)) y'(x) + 3\psi_{xyy}(x, y(x)) y'(x)^2 + \psi_{yyy}(x, y(x)) y'(x)^3 \\ &= \nabla^3 \psi(x, y(x))[\boldsymbol{v}, \boldsymbol{v}, \boldsymbol{v}] \end{aligned}$$

as desired.

To show (51), recall that $f_y(x, y(x)) = 0$. Taking a derivative of this in x gives

$$\psi_{xy}(x, y(x)) + \psi_{yy}(x, y(x)) y'(x) = 0,$$

which rearranges to (51) as desired. □

Now we show that the ζ_e functions are self-concordant. Note that we do not claim that ζ_e is ν-self-concordant, just self-concordant.

Lemma 12.8. *For all $e \in E$, $\alpha^{-1}\zeta_e/4$ is a self-concordant function.*

Proof. We calculate

$$\begin{aligned}\alpha^{-1}\zeta_e'''(\boldsymbol{f}_e) &= \alpha^{-1}\nabla^3(\exp(\alpha\psi_e(\boldsymbol{f}_e, y_e(\boldsymbol{f}_e))))[\boldsymbol{v},\boldsymbol{v},\boldsymbol{v}] \\ &= \Big(\nabla^3\psi_e(\boldsymbol{f}_e, y_e(\boldsymbol{f}_e))[\boldsymbol{v},\boldsymbol{v},\boldsymbol{v}] \\ &\quad + 3\alpha\nabla^2\psi_e(\boldsymbol{f}_e, y_e(\boldsymbol{f}_e))[\boldsymbol{v},\boldsymbol{v}] + \alpha^2\langle\nabla\psi_e(\boldsymbol{f}_e, y_e(\boldsymbol{f}_e)),\boldsymbol{v}\rangle^3\Big)\overline{\zeta}_e(\boldsymbol{f}_e),\end{aligned}$$

where $\boldsymbol{v} = \begin{bmatrix}1\\ y_e'(\boldsymbol{f}_e)\end{bmatrix}$. By ν-self-concordance of ψ_e, we can bound the previous expression by

$$\alpha^{-1}\zeta_e'''(\boldsymbol{f}_e) \le 4(\nabla^2\psi_e(\boldsymbol{f}_e, y_e(\boldsymbol{f}_e))[\boldsymbol{v},\boldsymbol{v}]\overline{\zeta}_e(\boldsymbol{f}_e))^{3/2} \le 4(\alpha^{-1}\zeta_e''(\boldsymbol{f}_e))^{3/2},$$

where the last inequality follows by the formula for $\boldsymbol{\ell}(\boldsymbol{f}_e)$. Scaling by a factor of 4 completes the proof. □

Our algorithm for solving (46) will fix a value of t and reduce the value of the potential (47) until $1^\top\boldsymbol{y} \le F^* + 50\nu m/t$. Once this holds, we will double the value of t and start a new phase. Each phase will requires approximately $m^{1+o(1)}\alpha^{-2}$ iterations.

We now formally define the gradients and lengths. The gradient is

$$\boldsymbol{g}(\boldsymbol{f})_e \stackrel{\text{def}}{=} [\nabla Z_t(\boldsymbol{f})]_e = \zeta_e'(\boldsymbol{f}_e), \tag{52}$$

and the lengths we define as

$$\begin{aligned}\boldsymbol{\ell}(\boldsymbol{f})_e &\stackrel{\text{def}}{=} \sqrt{\alpha^{-1}\zeta_e''(\boldsymbol{f}_e)} \\ &= \sqrt{(\boldsymbol{v}^\top\nabla^2\psi_e(\boldsymbol{f}_e, y_e(\boldsymbol{f}_e))\boldsymbol{v} + \alpha\langle\nabla\psi_e(\boldsymbol{f}_e, y_e(\boldsymbol{f}_e)),\boldsymbol{v}\rangle^2)\overline{\zeta}_e(\boldsymbol{f}_e)} \quad \text{for} \quad \boldsymbol{v} \stackrel{\text{def}}{=} \begin{bmatrix}1\\ y_e'(\boldsymbol{f}_e)\end{bmatrix}.\end{aligned} \tag{53}$$

Here, the equality starting line 2 follows from Lemma 12.7, (49) applied to the function $\zeta_e(\boldsymbol{f}_e)$.

Define $\boldsymbol{f}_t^* \stackrel{\text{def}}{=} \arg\min_{\boldsymbol{B}^\top\boldsymbol{f}=\boldsymbol{d}} Z_t(\boldsymbol{f})$. We now bound the optimality gap of $\boldsymbol{f}_t^*$. This will ultimately show that if $Z_t(\boldsymbol{f}_t^*) - F^*$ is much larger than $4m$, then we can reduce the potential by $m^{-o(1)}$ in a single step.

Lemma 12.9 (Optimality gap). *For sufficiently small $\alpha = \widetilde{\Omega}(1/\log\max(t,2))$, we have that*

$$Z_t(\boldsymbol{f}_t^*) - F^* \le 4m.$$

Proof. Recall that $\boldsymbol{f}^{(0)}, \boldsymbol{y}^{(0)}$ are the initially feasible points. Let $\boldsymbol{f}^*$ be the optimal flow and $\boldsymbol{y}_e^* = h_e(\boldsymbol{f}_e^*)$. We will upper bound $Z_t(\boldsymbol{f})$ for a flow $\boldsymbol{f} = \beta\boldsymbol{f}^{(0)} + (1-\beta)\boldsymbol{f}^*$ and $\boldsymbol{y} = \beta\boldsymbol{y}^{(0)} + (1-\beta)\boldsymbol{y}^*$ for a parameter $\beta \in [0,1]$ chosen later. Define $Q = h(\boldsymbol{f}^{(0)}) - F^*$, the optimality gap of the original flow. By our assumptions, we know that $\log Q = \widetilde{O}(1)$. We set $\beta = \min(1, m/(tQ))$. For $s \in [0,1]$ let $\boldsymbol{f}^{(s)} \stackrel{\text{def}}{=} \boldsymbol{f}^{(0)} + s(\boldsymbol{f}^* - \boldsymbol{f}^{(0)})$ and $\boldsymbol{y}^{(s)} \stackrel{\text{def}}{=} \boldsymbol{y}^{(0)} + s(\boldsymbol{y}^* - \boldsymbol{y}^{(0)})$.

Define the function $\overline{\psi}_e(s) \stackrel{\text{def}}{=} \psi_e(\boldsymbol{f}_e^{(s)}, \boldsymbol{y}_e^{(s)})$, which is ν-self-concordant as it is the restriction of ψ_e onto a line. By self-concordance, $\overline{\psi}_e''(s) \ge \overline{\psi}_e'''(s)/(2\sqrt{\overline{\psi}_e''(s)})$, so integrating both sides gives

$$\overline{\psi}_e'(1-\beta) - \overline{\psi}_e'(0) \ge \sqrt{\overline{\psi}_e''(1-\beta)} - \sqrt{\overline{\psi}_e''(0)}.$$

By ν-self-concordance and [108, Theorem 4.2.4] (Lemma 12.3), we know $\beta\overline{\psi}_e'(1-\beta) \le \nu$, and $\overline{\psi}_e'(0) \ge -\sqrt{\nu\overline{\psi}_e''(0)}$. Rearranging this gives us

$$\sqrt{\overline{\psi}_e''(1-\beta)} \le 2\sqrt{\nu\overline{\psi}_e''(0)} + \nu/\beta. \tag{54}$$

We use this to bound $\overline{\psi}_e(\boldsymbol{f}_e, \boldsymbol{y}_e) = \overline{\psi}_e(1-\beta)$. We can assume that $\overline{\psi}_e'(0) \ge 0$ because $\overline{\psi}_e'(s)$ is an increasing function, so we might as well start our integration at the minimizer on the line.

Now, by rearranging the ν-self-concordance condition we get

$$\overline{\psi}_e'(s) \le \frac{\nu\overline{\psi}_e''(s)}{\overline{\psi}_e'(s)+1} + 1.$$

Integrating both sides gives us

$$\overline{\psi}_e(1-\beta) - \overline{\psi}_e(0) \le \nu \log\left(\frac{\overline{\psi}_e'(1-\beta)+1}{\overline{\psi}_e'(0)+1}\right) + 1 \le \nu\log\left(\sqrt{\nu\overline{\psi}_e''(1-\beta)}+1\right)+1.$$

Recall by our assumption that $\overline{\psi}_e(0) = \psi_e(\boldsymbol{f}_e^{(0)}, \boldsymbol{y}_e^{(0)}) \le K = \widetilde{O}(1)$. Additionally by (54) the RHS of the above expression is also bounded by $\widetilde{O}(1) + \log(\nu/\beta) \le \widetilde{O}(1) + \max(0, O(\log t))$ because $\log Q = \widetilde{O}(1)$. Thus, we get that $\overline{\psi}_e(1-\beta) = \widetilde{O}(1) + \max(0, O(\log t))$, and in turn for $\alpha = \widetilde{\Omega}(1/\log\max(t,2))$,

$$Z_t(\boldsymbol{f}) \le t \cdot 1^\top \boldsymbol{y} + \sum_{e\in E} \exp(\alpha\psi_e(\boldsymbol{f}_e, \boldsymbol{y}_e)) \le t\beta Q + 2m \le 4m.$$

□

Using this, we will bound the quality of the solution $\boldsymbol{f}_t^* - \boldsymbol{f}$, i.e. how negative its gradient is compared to its total length.

Lemma 12.10. *Let α be set as in Lemma 12.9. If $1^\top \boldsymbol{y}(\boldsymbol{f}) - F^* \ge 10m/t$ and $\|\boldsymbol{L}(\boldsymbol{f})^{-1}(\widetilde{\boldsymbol{g}} - \boldsymbol{g}(\boldsymbol{f})\|_\infty \le \varepsilon$ for $\varepsilon \le \alpha/100$ then*

$$\widetilde{\boldsymbol{g}}^\top(\boldsymbol{f}_t^* - \boldsymbol{f}) \le -\alpha/4 \cdot \|\boldsymbol{L}(\boldsymbol{f})(\boldsymbol{f}_t^* - \boldsymbol{f})\|_1 - m/4.$$

Proof. We first handle the case where $\alpha\|\boldsymbol{L}(\boldsymbol{f})(\boldsymbol{f}_t^* - \boldsymbol{f})\|_1 \le 10m$. In this case, by the convexity of $Z_t(\boldsymbol{f})$, we get

$$\boldsymbol{g}(\boldsymbol{f})^\top(\boldsymbol{f}_t^* - \boldsymbol{f}) \le Z_t(\boldsymbol{f}_t^*) - Z_t(\boldsymbol{f}) = \left(Z_t(\boldsymbol{f}_t^*) - F^*\right) - \left(Z_t(\boldsymbol{f}) - F^*\right)$$

which upon applying Lemma 12.9 to the first term, and the assumption of $1^\top\boldsymbol{y}(\boldsymbol{f}) - F^* \ge 10m/t$ to the second gives

$$\le 4m - t \cdot 10m/t \le -6m.$$

Thus, we get

$$\begin{aligned}\boldsymbol{g}(\boldsymbol{f})^\top(\boldsymbol{f}_t^* - \boldsymbol{f}) &\le \boldsymbol{g}(\boldsymbol{f})^\top(\boldsymbol{f}_t^* - \boldsymbol{f}) + \|\boldsymbol{L}(\boldsymbol{f})^{-1}(\widetilde{\boldsymbol{g}} - \boldsymbol{g}(\boldsymbol{f}))\|_\infty \|\boldsymbol{L}(\boldsymbol{f})(\boldsymbol{f}_t^* - \boldsymbol{f})\|_1 \\ &\le -6m + \varepsilon \cdot 10m/\alpha \le -5m \le -\alpha/4 \cdot \|\boldsymbol{L}(\boldsymbol{f})(\boldsymbol{f}_t^* - \boldsymbol{f})\|_1 - m/4.\end{aligned}$$

Now, we handle the case where $\alpha\|\boldsymbol{L}(\boldsymbol{f})(\boldsymbol{f}_t^* - \boldsymbol{f})\|_1 \ge 10m$. Consider the function

$$\widehat{\zeta}_e(\boldsymbol{f}_e) = \alpha^{-1}\zeta_e(\boldsymbol{f}_e)/4,$$

which by Lemma 12.8 is self-concordant. Invoking [108, Theorem 4.1.7] (Lemma 12.4) on this function, we get for all e,

$$(\widehat{\zeta}_e'([f_t^*]_e) - \widehat{\zeta}_e'(f_e))([f_t^*]_e - f_e) \geq \frac{\widehat{\zeta}_e(f_e)''|[f_t^*]_e - f_e|^2}{1 + \sqrt{\widehat{\zeta}_e(f_e)''}|[f_t^*]_e - f_e|}$$
$$\geq \sqrt{\widehat{\zeta}_e(f_e)''}|[f_t^*]_e - f_e| - 1.$$

Rearranging the above equation gives us

$$\zeta_e'(f_e)([f_t^*]_e - f_e) \leq 4\alpha \left(\widehat{\zeta}_e'([f_t^*]_e)([f_t^*]_e - f_e) - \sqrt{\widehat{\zeta}_e''(f_e)}|[f_t^*]_e - f_e| + 1\right)$$
$$= 4\alpha\widehat{\zeta}_e'([f_t^*]_e)([f_t^*]_e - f_e) - \alpha/2 \cdot \boldsymbol{\ell}(f_e)|[f_t^*]_e - f_e| + 4\alpha.$$

By the optimality of f_t^* we know that $\boldsymbol{g}(f_t^*) = \boldsymbol{B}\boldsymbol{z}$ for some $\boldsymbol{z}$, so the first term is 0 because the difference $f_t^* - f$ is a circulation. Hence

$$\boldsymbol{g}(f)^\top(f_t^* - f_e) \leq -\alpha/2 \cdot \|\boldsymbol{L}(f)(f_t^* - f)\|_1 + 4\alpha m.$$

which upon incorporating errors from the approximate gradient gives

$$\widetilde{\boldsymbol{g}}^\top(f_t^* - f) \leq -\alpha/2 \cdot \|\boldsymbol{L}(f)(f_t^* - f)\|_1 + 4\alpha m + \|\boldsymbol{L}(f)^{-1}(f_t^* - f)\|_1 \|\boldsymbol{L}(f)(\widetilde{\boldsymbol{g}} - \boldsymbol{g}(f))\|_\infty$$
$$\leq (-\alpha/2 + \varepsilon) \cdot \|\boldsymbol{L}(f)(f_t^* - f)\|_1 + 4\alpha m \leq -\alpha/4 \cdot \|\boldsymbol{L}(f)(f_t^* - f)\|_1 - m/4$$

as long as $\alpha\|\boldsymbol{L}(f)(f_t^* - f)\|_1 \geq 10m$, for the choice of α. □

We move towards analyzing how a step $\boldsymbol{\Delta}$ decreases the potential Z_t. We start by showing that the gradients and lengths are stable in a Hessian ball.

Lemma 12.11 (Stability bounds). *For a flow f and vector $\boldsymbol{f} \in \mathbb{R}^E$ satisfying $\|\boldsymbol{L}(f)(\boldsymbol{f} - \overline{f})\|_\infty \leq \varepsilon$ for $\varepsilon < 1/1000$, then $\boldsymbol{\ell}(f) \approx_{1+5\varepsilon} \boldsymbol{\ell}(\overline{f})$, $\|\boldsymbol{L}(f)^{-1}(\boldsymbol{g}(f) - \boldsymbol{g}(\overline{f}))\|_\infty \leq \varepsilon$.*

Proof. The stability of lengths follows from self-concordance of $\alpha^{-1}\zeta_e/4$ shown in Lemma 12.8, plus the Hessian stability of such functions shown in [107, Theorem 4.1.6] (Lemma 12.5). To analyze gradient stability, let $\phi(s) \stackrel{\text{def}}{=} \zeta_e'(f^{(s)})$. Now

$$|\phi(s)'| = |\overline{f}_e - f_e|\zeta_e''(f^{(s)}) = \alpha|\overline{f}_e - f_e|\boldsymbol{\ell}(f^{(s)})^2 \leq 2\alpha\boldsymbol{\ell}(f_e)^2|\overline{f}_e - f_e| \leq 2\alpha\varepsilon\boldsymbol{\ell}_e(f).$$

Hence $|\boldsymbol{\ell}_e(f)^{-1}(\phi(1) - \phi(0))| \leq 2\alpha\varepsilon \leq \varepsilon$ as desired. □

We can use Lemma 12.11 to show that a good quality circlation $\boldsymbol{\Delta}$ decreases the potential Z_t.

Lemma 12.12. *Let $\widetilde{\boldsymbol{L}} \approx_2 \boldsymbol{L}(f)$ and $\|\boldsymbol{L}(f)^{-1}(\widetilde{\boldsymbol{g}} - \boldsymbol{g}(f))\|_\infty \leq \varepsilon$ for $\varepsilon < \kappa/100$. If circulation $\boldsymbol{\Delta}$ satisfies $\widetilde{\boldsymbol{g}}^\top\boldsymbol{\Delta}/\|\widetilde{\boldsymbol{L}}\boldsymbol{\Delta}\|_1 \leq -\kappa$, then for $\eta > 0$ chosen so that $\eta\widetilde{\boldsymbol{g}}^\top\boldsymbol{\Delta} = -\kappa^2/50$ satisfies*

$$Z_t(f + \eta\boldsymbol{\Delta}) \leq Z_t(f) - \kappa^2/100.$$

Proof. Let $\overline{\Delta} = \eta\Delta$. Define $\boldsymbol{f}^{(s)} = \boldsymbol{f} + s\overline{\Delta}$, and $\phi(s) = Z_t(\boldsymbol{f}^{(s)})$. By Taylor's theorem, we know

$$\begin{aligned} Z_t(\boldsymbol{f}+\eta\Delta) - Z_t(\boldsymbol{f}) = \phi(1) - \phi(0) &\le \phi'(0) + \max_{s\in[0,1]} \phi''(s)^2/2 \\ &\overset{(i)}{\le} \eta \boldsymbol{g}(\boldsymbol{f})^\top \Delta + \overline{\Delta}^\top \nabla^2 Z_t(\boldsymbol{f})\overline{\Delta} \\ &\le \eta\overline{\boldsymbol{g}}^\top\Delta + (\boldsymbol{g}(\boldsymbol{f}) - \overline{\boldsymbol{g}})^\top\overline{\Delta} + \alpha\|\boldsymbol{L}(\boldsymbol{f})\overline{\Delta}\|_2^2 \\ &\le -\kappa^2/50 + \|\boldsymbol{L}(\boldsymbol{f})^{-1}(\boldsymbol{g}(\boldsymbol{f}) - \overline{\boldsymbol{g}})\|_\infty \|\boldsymbol{L}(\boldsymbol{f})\overline{\Delta}\|_1 + \|\boldsymbol{L}(\boldsymbol{f})\overline{\Delta}\|_1^2 \\ &\le -\kappa^2/50 + \varepsilon\kappa/50 + (\kappa/50)^2 \le -\kappa^2/100, \end{aligned}$$

where (i) follows from length stability in Lemma 12.11. □

We can now state and show our main result on optimizing flows under general convex objectives.

Theorem 12.13 (General convex flows). *Let G be a graph with m edges, and let $\boldsymbol{d}$ be a demand. Given convex functions $h_e : \mathbb{R} \to \mathbb{R} \cup \{+\infty\}$ and ν-self-concordant barriers $\psi_e(f, y)$ on the domain $\{(f, y) : y \le h_e(f)\}$ satisfying the guarantees of Assumption 12.2, there is an algorithm that runs in $m^{1+o(1)}$ time and outputs a flow $\boldsymbol{f}$ with $\boldsymbol{B}^\top\boldsymbol{f} = \boldsymbol{d}$ and for any fixed constant $C > 0$,*

$$h(\boldsymbol{f}) \le \min_{\boldsymbol{B}^\top\boldsymbol{f}^* = \boldsymbol{d}} h(\boldsymbol{f}^*) + \exp(-\log^C m).$$

Proof. Initialize $t = m^{-\widetilde{O}(1)}$, and set $\alpha = \widetilde{\Omega}(1)$ as in Lemma 12.9. For this fixed value of t run the analogue of Algorithm 7, and we repeat the same analysis as in Section 11. We will store the approximate values of $\boldsymbol{f}, \boldsymbol{y}(\boldsymbol{f})$. Every $\widetilde{\Omega}(m)$ iterations, we recompute $\boldsymbol{f}, \boldsymbol{y}(\boldsymbol{f})$ exactly and check whether $1^\top\boldsymbol{y}(\boldsymbol{f}) - F^* \le 20m/t$. If so, we double t and proceed to the next phase. We stop when $t = m^{\widetilde{O}(1)}$, so there are at most $\widetilde{O}(1)$ phases.

By Lemmas 12.10 and 12.12, the value of Z_t decreases by $\kappa^{-2}\alpha^{-2} = m^{-o(1)}$ per iteration. When t doubles, because we know that $1^\top\boldsymbol{y}(\boldsymbol{f}) - F^* \le 20m/t$ by the stopping condition, $Z_{2t}(\boldsymbol{f}) \le 20m + Z_t(\boldsymbol{f})$, i.e. the potential increases by at most $20m$. Hence over all $\widetilde{O}(1)$ phases, the total potential increase is $\widetilde{O}(m)$. So the algorithm runs in at most $m^{1+o(1)}$ iterations. The number of gradient/length changes is bounded by $m^{1+o(1)}$ if they are updated lazily by Lemma 12.11.

Because $Z_t(\boldsymbol{f}) \le \widetilde{O}(m)$ always, by the choice of α we know that $\psi_e(\boldsymbol{f}_e, \boldsymbol{y}_e) \le \widetilde{O}(1)$ at all times. Thus, by item 6 of Assumption 12.2, all lengths are quasipolynomially bounded during the algorithm. Additionally, Lemma 12.10 and an identical analysis to Lemma 11.2 for $\boldsymbol{c} \overset{\text{def}}{=} \boldsymbol{f}_t^* - \boldsymbol{f}$ and $\boldsymbol{w} \overset{\text{def}}{=} 50 + \|\boldsymbol{\ell}(\boldsymbol{f}) \circ \boldsymbol{c}\|_1$ shows that the updates to $\boldsymbol{g}, \boldsymbol{\ell}$ satisfy the hidden stable-flow chasing property (Definition 8.1). Hence our min-ratio cycle data structure Theorem 8.2 succeeds whp. in total time $m^{1+o(1)}$ as desired. □

12.2 Applications: p-Norms, Entropy-Regularized Optimal Transport, and Matrix Scaling

Using our main result Theorem 12.13 we can give algorithms for the problems of normed flows, isotonic regression, entropy-regularized optimal transport, and matrix scaling. We start by discussing p-norm flows. In this case, we allow the convex functions on our edges to be the sum of arbitrarily weighted power functions where the power is at most $\widetilde{O}(1)$.

Theorem 12.14. *Consider a graph $G = (V, E)$ and demand $\boldsymbol{d}$ whose entries are bounded by $\exp(\log^{O(1)} m)$, and convex functions h_e which are the sum of $\widetilde{O}(1)$ p-norm terms, i.e. $h_e(x) = \sum_{i=1}^{c_e} w_i|x|^{p_i}$ for $c_e \le \widetilde{O}(1)$ and $p_i \le \widetilde{O}(1)$, and $w_i \in [0, \exp(\log^{O(1)} m)]$ for all $i \in [c_e]$. Let $h(\boldsymbol{f}) \overset{\text{def}}{=} \sum_{e\in E} h_e(\boldsymbol{f}_e)$. Then in $m^{1+o(1)}$ time we can compute a flow $\boldsymbol{f}$*

satisfying $\boldsymbol{B}^\top \boldsymbol{f} = \boldsymbol{d}$ *and for any constant* $C > 0$

$$h(\boldsymbol{f}) \le \min_{\boldsymbol{B}^\top \boldsymbol{f}^* = \boldsymbol{d}} h(\boldsymbol{f}^*) + \exp(-\log^C m).$$

Proof. By splitting up an edge e into c_e edges in a path we can assume that each $h_e(x) = w|x|^p$ for some $p \le \widetilde{O}(1)$. It is known that the function $\psi(x,y) \stackrel{\text{def}}{=} -2\log y - \log(y^{2/p} - x^2)$ is 4-self-concordant for the region $\{(x,y) : y \ge |x|^p\}$ [106, Example 9.2.1]. For this barrier, all items in Assumption 12.2 hold by observation, except those involving $\nabla^2\psi(x,y)$ which we now calculate.

$$\nabla^2\psi(x,y) = \begin{bmatrix} \frac{4x^2}{(y^{2/p}-x^2)^2} + \frac{2}{y^{2/p}-x^2} & -\frac{4xy^{2/p-1}}{p(y^{2/p}-x^2)^2} \\ -\frac{4xy^{2/p-1}}{p(y^{2/p}-x^2)^2} & \frac{2}{y^2} + \frac{4y^{4/p-2}}{p^2(y^{2/p}-x^2)^2} + \frac{(2p-4)y^{2/p-2}}{p^2(y^{2/p}-x^2)} \end{bmatrix}.$$

Clearly, if $-\log y, -\log(y^{2/p}-x^2) \le \widetilde{O}(1)$, and $|y|, |x| \le m^{\widetilde{O}(1)}$, then all terms of $\nabla^2\psi(x,y)$ are bounded by $\exp(\log^{O(1)} m)$ as desired, which verifies item 6 of Assumption 12.2. Thus all assumptions are satisfied, so the result follows by Theorem 12.13. □

The same barriers allow us to solve the problem of ℓ_p isotonic regression [87]. Given a directed acyclic graph $G = (V, E)$ and a vector $\boldsymbol{y} \in \mathbb{R}^V$, the ℓ_p isotonic regression problems asks to return a vector $\boldsymbol{x}$ that satisfies $\boldsymbol{x}_u \le \boldsymbol{x}_v$ for all directed arcs $(u,v) \in E$, and minimizes $\|\boldsymbol{W}(\boldsymbol{x}-\boldsymbol{y})\|_p$ for a weight vector $\boldsymbol{W} \ge 0$. Linear algebraically, this is $\min_{\boldsymbol{x}\in\mathbb{R}^V, \boldsymbol{B}\boldsymbol{x}\ge\vec{0}} \|\boldsymbol{W}(\boldsymbol{x}-\boldsymbol{y})\|_p$. We show that the dual of this problem is a flow problem. Let q be the dual norm of p. Then

$$\begin{aligned} \min_{\boldsymbol{x}\in\mathbb{R}^V, \boldsymbol{B}\boldsymbol{x}\ge\vec{0}} \|\boldsymbol{W}(\boldsymbol{x}-\boldsymbol{y})\|_p &= \min_{\boldsymbol{x}\in\mathbb{R}^V} \max_{\boldsymbol{f}\ge\vec{0}, \|\boldsymbol{z}\|_q \le 1} \boldsymbol{z}^\top \boldsymbol{W}(\boldsymbol{x}-\boldsymbol{y}) - \boldsymbol{f}^\top \boldsymbol{B}\boldsymbol{x} \\ &= \max_{\boldsymbol{f}\ge\vec{0}, \|\boldsymbol{z}\|_q \le 1} \min_{\boldsymbol{x}\in\mathbb{R}^V} \boldsymbol{z}^\top \boldsymbol{W}(\boldsymbol{x}-\boldsymbol{y}) - \boldsymbol{f}^\top \boldsymbol{B}\boldsymbol{x} \\ &= \max_{\boldsymbol{f}\ge\vec{0}, \|\boldsymbol{W}^{-1}\boldsymbol{B}^\top \boldsymbol{f}\|_q \le 1} -\boldsymbol{f}^\top(\boldsymbol{B}\boldsymbol{y}). \end{aligned}$$

Let $\boldsymbol{c} = \boldsymbol{B}\boldsymbol{y}$. By rescaling, the objective becomes computing

$$\min_{\boldsymbol{f}\ge\vec{0}} \boldsymbol{c}^\top \boldsymbol{f} + \|\boldsymbol{W}^{-1}\boldsymbol{B}^\top \boldsymbol{f}\|_q^q.$$

Given a high-accuracy solution to this objective, we can extract the desired original potentials $\boldsymbol{x}$ by taking a gradient of the objective. To turn this objective into the q-norm of a flow, add a few vertex v^* to the graph G, and an undirected edge between (v, v^*) for all $v \in V$. Assign this edge the convex function $\boldsymbol{w}_v|x|^q$, and for every other original edge $e \in E$, assign it the convex function $\boldsymbol{c}_e \boldsymbol{f}_e$, and restrict $\boldsymbol{f} \ge 0$ (eg. using a logarithmic barrier). Finally, force $\boldsymbol{f}$ to to have 0 demand on the graph G with the extra vertex v^*. This is now a clearly equivalent flow problem. As we have already described the self-concordant barriers for linear objectives, $\boldsymbol{f}_e \ge 0$, and q-norms in the proof of Theorem 12.14, we get:

Theorem 12.15. *Given a directed acyclic graph* G*, vector* $\boldsymbol{y} \in \mathbb{R}^V$*, and* $p \in [1, \infty]$*, we can compute in* $m^{1+o(1)}$ *time vertex potentials* $\boldsymbol{x}$ *with* $\boldsymbol{B}\boldsymbol{x} \ge 0$ *and for any constant* $C > 0$

$$\|\boldsymbol{x}-\boldsymbol{y}\|_p \le \min_{\boldsymbol{B}\boldsymbol{x}^*\ge 0} \|\boldsymbol{x}^*-\boldsymbol{y}\|_p + \exp(-\log^C m).$$

Next we discuss the pair of problems of entropy-regularized optimal transport/min-cost flow and matrix scaling, which are duals. The former problem is

$$\min_{\boldsymbol{B}^\top \boldsymbol{f} = \boldsymbol{d}} \sum_{e\in E} \boldsymbol{c}_e \boldsymbol{f}_e + \boldsymbol{f}_e \log \boldsymbol{f}_e.$$

The matrix scaling problem asks to given a matrix $\boldsymbol{A} \in \mathbb{R}^{n\times n}_{\geq 0}$ with non-negative entries to compute positive diagonal matrices $\boldsymbol{X}, \boldsymbol{Y}$ so that $\boldsymbol{XAY}$ is doubly stochastic, i.e. all row and column sums are exactly 1. By [31, Theorem 4.6], it suffices to minimize the objective

$$\sum_{(i,j):\boldsymbol{A}_{ij}\neq 0} \boldsymbol{A}_{ij} \exp(\boldsymbol{x}_i - \boldsymbol{y}_j) - \sum_{i=1}^{n} \boldsymbol{x}_i - \sum_{i=1}^{n} \boldsymbol{y}_i$$

to high accuracy. Consider turning the pair (i, j) with $\boldsymbol{A}_{ij} \neq 0$ into an edge $e = (i, j + n)$ in a graph G with $2n$ vertices with weight $\boldsymbol{w}_e \stackrel{\text{def}}{=} \boldsymbol{A}_{ij}$. Let $\boldsymbol{z} = \begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{y} \end{bmatrix}$ be the concatenation of the x, y vectors. Then the above problem becomes $\sum_{e\in E(G)} \boldsymbol{w}_e \exp(\boldsymbol{z}_i - \boldsymbol{z}_j) - \sum_{i=1}^{2n} \boldsymbol{z}_i$. The optimality conditions for this objective are $\boldsymbol{B}^\top \boldsymbol{W} \exp(\boldsymbol{Bz}) = \boldsymbol{d}$, where $\boldsymbol{d}$ is $+1$ on the vertices $\{1, \ldots, n\}$ and -1 on $\{n + 1, \ldots, 2n\}$. Rearranging this gives us $\boldsymbol{B}^\top \boldsymbol{f} = \boldsymbol{d}$ for $\boldsymbol{f} = \boldsymbol{W} \exp(\boldsymbol{Bz})$, or $\boldsymbol{Bz} = \log(\boldsymbol{W}^{-1}\boldsymbol{f})$. This is the exact optimality condition for the flow problem

$$\min_{\boldsymbol{B}^\top \boldsymbol{f}=\boldsymbol{d}} \sum_{e\in E} (-1 - \log \boldsymbol{w}_e)\boldsymbol{f}_e + \boldsymbol{f}_e \log(\boldsymbol{f}_e),$$

which is exactly entropy-regularized optimal transport for $\boldsymbol{c}_e \stackrel{\text{def}}{=} (-1 - \log \boldsymbol{w}_e)$. If the entries of $\boldsymbol{A}$ are polynomially lower and upper bounded, then given an (almost) optimal flow $\boldsymbol{f}$, we know by KKT conditions that $\boldsymbol{f} = \boldsymbol{W} \exp(\boldsymbol{Bz})$ for some dual variable $\boldsymbol{z}$ which we can then efficiently recover. So it suffices to give high-accuracy algorithms for the entropy regularized OT problem.

Theorem 12.16. *Given a graph $G = (V, E)$, demands $\boldsymbol{d}$, costs $\boldsymbol{c} \in \mathbb{R}^E$, and weights $\boldsymbol{w}_e \in \mathbb{R}_{\geq 0}$, all bounded by $\exp(\log^{O(1)} m)$, let $h(\boldsymbol{f}) \stackrel{\text{def}}{=} \sum_{e\in E} \boldsymbol{c}_e \boldsymbol{f}_e + \boldsymbol{w}_e \boldsymbol{f}_e \log \boldsymbol{f}_e$. Then in $m^{1+o(1)}$ time we can find a flow $\boldsymbol{f}$ with $\boldsymbol{B}^\top \boldsymbol{f} = \boldsymbol{d}$ and for any constant $C > 0$*

$$h(\boldsymbol{f}) \leq \min_{\boldsymbol{B}^\top \boldsymbol{f}^*=\boldsymbol{d}} h(\boldsymbol{f}^*) + \exp(-\log^C m).$$

Proof. By splitting the edge e into two edges, we can handle the terms $\boldsymbol{c}_e \boldsymbol{f}_e$ and $\boldsymbol{w}_e \boldsymbol{f} \log \boldsymbol{f}_e$ separately. For the $\boldsymbol{c}_e \boldsymbol{f}_e$ term, we can handle it using the self-concordant barrier $\psi(x, y) \stackrel{\text{def}}{=} -\log(y - \boldsymbol{c}_e x)$. For the term $\boldsymbol{w}_e \boldsymbol{f} \log \boldsymbol{f}_e$, we use the 2-self-concordant barrier $\psi(x, y) \stackrel{\text{def}}{=} -\log x - \log(y - x \log x)$ [106, Example 9.2.4]. As in the proof of Theorem 12.14, all items of Assumption 12.2 hold directly, except that we must compute $\nabla^2 \psi(x, y)$.

$$\nabla^2 \psi(x, y) = \begin{bmatrix} \frac{1}{x^2} + \frac{(1+\log x)^2}{(y - x\log x)^2} + \frac{1}{x(y - x\log x)} & -\frac{1+\log x}{(y - x\log x)^2} \\ -\frac{1+\log x}{(y - x\log x)^2} & \frac{1}{(y - x\log x)^2} \end{bmatrix}.$$

Indeed, if $-\log x, -\log(y - x \log x) \leq \widetilde{O}(1)$, and $|x|, |y| \leq \exp(\log^{O(1)} m)$, then all terms in the Hessian are quasipolynomially bounded. This verifies item 6 of Assumption 12.2, so the result follows by Theorem 12.13. □

Corollary 12.17 (Matrix scaling). *Given a matrix $\boldsymbol{A} \in \mathbb{R}^{n\times n}_{\geq 0}$ whose nonzero entries are in $[n^{-O(1)}, n^{O(1)}]$, we can find in time $\mathrm{nnz}(\boldsymbol{A})^{1+o(1)}$ positive diagonal matrices $\boldsymbol{X}, \boldsymbol{Y}$ such that all row and column sums of $\boldsymbol{XAY}$ are within $\exp(-\log^C n)$ of 1 for any constant $C > 0$.*

ACKNOWLEDGMENTS

Li Chen was supported by NSF Grant CCF-2106444. Rasmus Kyng and Maximilian Probst Gutenberg have received funding from the grant "Algorithms and complexity for high-accuracy flows and convex optimization" (no. 200021 204787) of the Swiss National Science Foundation. Yang P. Liu was supported by NSF CAREER Award CCF-1844855

and NSF Grant CCF-1955039. Richard Peng was partially supported by NSF CAREER Award CCF-1846218 and NSERC Discovery Grant RGPIN-2022-03207. Sushant Sachdeva's research is supported by an Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grant RGPIN-2018-06398, an Ontario Early Researcher Award (ERA) ER21-16-283, and a Sloan Research Fellowship.

The authors thank the 2021 Hausdorff Research Institute for Mathematics Program Discrete Optimization during which part of this work was developed. The authors are very grateful to Yin Tat Lee and Aaron Sidford for several useful discussions on general convex flow problems and empirical risk minimization in graphical settings.

REFERENCES

[1] Amir Abboud, Robert Krauthgamer, Jason Li, Debmalya Panigrahi, Thatchaphol Saranurak, and Ohad Trabelsi. 2021. Gomory-Hu Tree in Subcubic Time. *CoRR* abs/2111.04958 (2021). arXiv:2111.04958 Available at: https://arxiv.org/abs/2111.04958.

[2] Ittai Abraham and Ofer Neiman. 2019. Using petal-decompositions to build a low stretch spanning tree. *SIAM J. Comput.* 48, 2 (2019), 227–248.

[3] Deeksha Adil, Brian Bullins, Rasmus Kyng, and Sushant Sachdeva. 2021. Almost-Linear-Time Weighted ℓ_p-Norm Solvers in Slightly Dense Graphs via Sparsification. In *48th International Colloquium on Automata, Languages, and Programming (ICALP 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

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

[5] Deeksha Adil and Sushant Sachdeva. 2020. Faster p-norm minimizing flows, via smoothed q-norm problems. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 892–910.

[6] Ravindra K. Ahuja, Andrew V. Goldberg, James B. Orlin, and Robert Endre Tarjan. 1992. Finding minimum-cost flows by double scaling. *Math. Program.* 53 (1992), 243–266. https://doi.org/10.1007/BF01585705

[7] Zeyuan Allen-Zhu, Yuanzhi Li, Rafael Mendes de Oliveira, and Avi Wigderson. 2017. Much Faster Algorithms for Matrix Scaling. In *FOCS*. IEEE Computer Society, 890–901.

[8] Noga Alon, Richard M Karp, David Peleg, and Douglas West. 1995. A graph-theoretic game and its application to the k-server problem. *SIAM J. Comput.* 24, 1 (1995), 78–100.

[9] Jason Altschuler, Jonathan Niles-Weed, and Philippe Rigollet. 2017. Near-linear time approximation algorithms for optimal transport via Sinkhorn iteration. *Advances in neural information processing systems* 30 (2017).

[10] Kyriakos Axiotis, Aleksander Mądry, and Adrian Vladu. 2020. Circulation control for faster minimum cost flow in unit-capacity graphs. In *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 93–104.

[11] Kyriakos Axiotis, Aleksander Mądry, and Adrian Vladu. 2022. Faster sparse minimum cost flow by electrical flow localization. In *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 528–539.

[12] Jean-David Benamou, Guillaume Carlier, Marco Cuturi, Luca Nenna, and Gabriel Peyré. 2015. Iterative Bregman projections for regularized transportation problems. *SIAM Journal on Scientific Computing* 37, 2 (2015), A1111–A1138.

[13] Aaron Bernstein, Jan van den Brand, Maximilian Probst Gutenberg, Danupon Nanongkai, Thatchaphol Saranurak, Aaron Sidford, and He Sun. 2020. Fully-dynamic graph sparsifiers against an adaptive adversary. *arXiv preprint arXiv:2004.08432* (2020).

[14] Aaron Bernstein, Maximilian Probst Gutenberg, and Thatchaphol Saranurak. 2020. Deterministic decremental reachability, SCC, and shortest paths via directed expanders and congestion balancing. In *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 1123–1134.

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

[16] Stephen Boyd and Lieven Vandenberghe. 2004. Convex Optimization. (2004).

[17] Yuri Boykov and Vladimir Kolmogorov. 2004. An Experimental Comparison of Min-Cut/Max-Flow Algorithms for Energy Minimization in Vision. *IEEE Trans. Pattern Anal. Mach. Intell.* 26, 9 (2004), 1124–1137. Available at: https://arxiv.org/abs/1202.3367.

[18] Jan van den Brand, Yin Tat Lee, Yang P. Liu, Thatchaphol Saranurak, Aaron Sidford, Zhao Song, and Di Wang. 2021. Minimum cost flows, MDPs, and ℓ_1-regression in nearly linear time for dense instances. In *STOC*. ACM, 859–869.

[19] Jan van den Brand, Yin-Tat Lee, Danupon Nanongkai, Richard Peng, Thatchaphol Saranurak, Aaron Sidford, Zhao Song, and Di Wang. 2020. Bipartite matching in nearly-linear time on moderately dense graphs. In *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 919–930.

[20] Jan van den Brand, Yin Tat Lee, Aaron Sidford, and Zhao Song. 2020. Solving tall dense linear programs in nearly linear time. In *Proccedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing (STOC)*. ACM, 775–788.

[21] Ruoxu Cen, Jason Li, Danupon Nanongkai, Debmalya Panigrahi, Kent Quanrud, and Thatchaphol Saranurak. 2021. Minimum Cuts in Directed Graphs via Partial Sparsification. *CoRR* abs/2111.08959 (2021). arXiv:2111.08959 https://arxiv.org/abs/2111.08959

[22] Bala G Chandran and Dorit S Hochbaum. 2009. A computational study of the pseudoflow and push-relabel algorithms for the maximum flow problem. *Operations research* 57, 2 (2009), 358–376.

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

[24] Li Chen, Richard Peng, and Di Wang. 2021. ℓ_2-norm Flow Diffusion in Near-Linear Time. *arXiv preprint arXiv:2105.14629* (2021). Available at https://arxiv.org/abs/2105.14629.

[25] Sinho Chewi. 2021. The entropic barrier is n-self-concordant. arXiv:2112.10947 [math.MG] Available at: https://arxiv.org/abs/2112.10947.

[26] Paul Christiano, Jonathan A. Kelner, Aleksander Mądry, Daniel A. Spielman, and Shang-Hua Teng. 2011. Electrical flows, Laplacian systems, and faster approximation of maximum flow in undirected graphs. In *Proceedings of the 43rd ACM Symposium on Theory of Computing (STOC)*. ACM, 273–282.

[27] Julia Chuzhoy. 2021. *Decremental All-Pairs Shortest Paths in Deterministic near-Linear Time.* Association for Computing Machinery, New York, NY, USA, 626–639. Available at: https://arxiv.org/abs/2109.05621.

[28] Julia Chuzhoy and Sanjeev Khanna. 2019. A New Algorithm for Decremental Single-Source Shortest Paths with Applications to Vertex-Capacitated Flow and Cut Problems. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing* (Phoenix, AZ, USA) *(STOC 2019)*. Association for Computing Machinery, New York, NY, USA, 389–400. Available at https://arxiv.org/abs/1905.11512.

[29] Julia Chuzhoy and Thatchaphol Saranurak. 2021. Deterministic algorithms for decremental shortest paths via layered core decomposition. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM, 2478–2496.

[30] Michael B. Cohen, Yin Tat Lee, and Zhao Song. 2019. Solving linear programs in the current matrix multiplication time. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019*. ACM, 938–942. Available at https://arxiv.org/abs/1810.07896.

[31] Michael B. Cohen, Aleksander Mądry, Dimitris Tsipras, and Adrian Vladu. 2017. Matrix Scaling and Balancing via Box Constrained Newton's Method and Interior Point Methods. In *FOCS*. IEEE Computer Society, 902–913.

[32] Marco Cuturi. 2013. Sinkhorn distances: Lightspeed computation of optimal transport. *Advances in neural information processing systems* 26 (2013).

[33] Samuel I Daitch and Daniel A Spielman. 2008. Faster approximate lossy generalized flow via interior point algorithms. In *Proceedings of the fortieth annual ACM symposium on Theory of computing*. 451–460.

[34] George B Dantzig. 1951. Application of the simplex method to a transportation problem. *Activity analysis and production and allocation* (1951).

[35] E.A. Dinic. 1970. Algorithm for solution of a problem of maximum flow in networks with power estimation. *Soviet Mathematics Doklady* 11 (1970), 1277–1280.

[36] E.A. Dinic. 1973. Metod porazryadnogo sokrashcheniya nevyazok i transportnye zadachi. *Issledovaniya po Diskretnoĭ Matematike* (1973). In Russian. Title translation: Excess scaling and transportation problems..

[37] Sally Dong, Yu Gao, Gramoz Goranci, Yin Tat Lee, Richard Peng, Sushant Sachdeva, and Guanghao Ye. 2022. Nested Dissection Meets IPMs: Planar Min-Cost Flow in Nearly-Linear Time. In *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 124–153. https://doi.org/10.1137/1.9781611977073.7 arXiv:https://epubs.siam.org/doi/pdf/10.1137/1.9781611977073.7

[38] Sally Dong, Yin Tat Lee, and Guanghao Ye. 2021. A nearly-linear time algorithm for linear programs with small treewidth: a multiscale representation of robust central path. In *STOC*. ACM, 1784–1797.

[39] Ran Duan, Seth Pettie, and Hsin-Hao Su. 2018. Scaling Algorithms for Weighted Matching in General Graphs. *ACM Trans. Algorithms* 14, 1 (2018), 8:1–8:35. Available at: https://arxiv.org/abs/1411.1919.

[40] Pavel Dvurechensky, Alexander Gasnikov, and Alexey Kroshnin. 2018. Computational optimal transport: Complexity by accelerated gradient descent is better than by Sinkhorn's algorithm. In *International conference on machine learning*. PMLR, 1367–1376.

[41] Jack Edmonds. 1965. Maximum matching and a polyhedron with 0, 1-vertices. *Journal of research of the National Bureau of Standards B* 69, 125-130 (1965), 55–56.

[42] Jack Edmonds. 1965. Paths, trees, and flowers. *Canadian Journal of mathematics* 17 (1965), 449–467.

[43] Jack Edmonds and Richard M. Karp. 1972. Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *J. ACM* 19, 2 (1972), 248–264. https://doi.org/10.1145/321694.321699

[44] Thomas R Ervolina and S Thomas McCormick. 1993. Canceling most helpful total cuts for minimum cost network flow. *Networks* 23, 1 (1993), 41–52.

[45] Thomas R Ervolina and S Thomas McCormick. 1993. Two strongly polynomial cut cancelling algorithms for minimum cost network flow. *Discrete Applied Mathematics* 46, 2 (1993), 133–165.

[46] Shimon Even and Oded Kariv. 1975. An $O(n^{2.5})$ algorithm for maximum matching in general graphs. In *16th Annual Symposium on Foundations of Computer Science (sfcs 1975)*. IEEE, 100–112.

[47] Shimon Even and R. Endre Tarjan. 1975. Network Flow and Testing Graph Connectivity. *SIAM journal on computing* 4, 4 (1975), 507–518.

[48] Barak Fishbain, Dorit S. Hochbaum, and Stefan Muller. 2010. Competitive Analysis of Minimum-Cut Maximum Flow Algorithms in Vision Problems. *CoRR* abs/1007.4531 (2010). arXiv:1007.4531 Available at: http://arxiv.org/abs/1007.4531.

[49] L. R. Ford and D. R. Fulkerson. 1954. *Maximal Flow through a Network.* RAND Corporation, Santa Monica, CA.

[50] Kimon Fountoulakis, Di Wang, and Shenghao Yang. 2020. p-Norm Flow Diffusion for Local Graph Clustering. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event (Proceedings of Machine Learning Research, Vol. 119)*. PMLR, 3222–3232.

[51] Delbert R Fulkerson. 1961. An out-of-kilter method for minimal-cost flow problems. *J. Soc. Indust. Appl. Math.* 9, 1 (1961), 18–27.

[52] Harold N Gabow. 1976. An efficient implementation of Edmonds' algorithm for maximum matching on graphs. *Journal of the ACM (JACM)* 23, 2 (1976), 221–234.

[53] Harold N. Gabow. 1985. Scaling Algorithms for Network Problems. *J. Comput. Syst. Sci.* 31, 2 (1985), 148–168. https://doi.org/10.1016/0022-0000(85)90039-X

[54] Harold N. Gabow. 2017. The weighted matching approach to maximum cardinality matching. *Fund. Inform.* 154, 1-4 (2017), 109–130. https://doi.org/10.3233/FI-2017-1555

[55] Harold N. Gabow and Robert Endre Tarjan. 1989. Faster Scaling Algorithms for Network Problems. *SIAM J. Comput.* 18, 5 (1989), 1013–1036. https://doi.org/10.1137/0218069

[56] Harold N. Gabow and Robert E. Tarjan. 1991. Faster scaling algorithms for general graph-matching problems. *J. Assoc. Comput. Mach.* 38, 4 (1991), 815–853. https://doi.org/10.1145/115234.115366

[57] Zvi Galil and Amnon Naamad. 1980. An $O(EV \log^2 V)$ Algorithm for the Maximal Flow Problem. *J. Comput. Syst. Sci.* 21, 2 (1980), 203–217. https://doi.org/10.1016/0022-0000(80)90035-5

[58] Zvi Galil and Éva Tardos. 1988. An $O(n^2(m + n \log n) \log n)$ Min-Cost Flow Algorithm. *J. ACM* 35, 2 (1988), 374–386.

[59] Yu Gao, Yang P Liu, and Richard Peng. 2022. Fully dynamic electrical flows: Sparse maxflow faster than goldberg-rao. In *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 516–527.

[60] Andrew Goldberg and Robert Tarjan. 1987. Solving minimum-cost flow problems by successive approximation. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*. 7–18.

[61] Andrew V. Goldberg. 1995. Scaling Algorithms for the Shortest Paths Problem. *SIAM J. Comput.* 24, 3 (1995), 494–504.

[62] Andrew V Goldberg. 2008. The partial augment–relabel algorithm for the maximum flow problem. In *European Symposium on Algorithms*. Springer, 466–477.

[63] Andrew V. Goldberg, Sagi Hed, Haim Kaplan, Pushmeet Kohli, Robert Endre Tarjan, and Renato F. Werneck. 2015. Faster and More Dynamic Maximum Flow by Incremental Breadth-First Search. In *Algorithms - ESA 2015 - 23rd Annual European Symposium, Patras, Greece, September 14-16, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9294)*, Nikhil Bansal and Irene Finocchi (Eds.). Springer, 619–630. Available at: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/11/ghkktw_ESA2015.pdf.

[64] Andrew V Goldberg and Alexander V Karzanov. 2004. Maximum skew-symmetric flows and matchings. *Mathematical Programming* 100 (2004), 537–568.

[65] Andrew V. Goldberg and Satish Rao. 1998. Beyond the Flow Decomposition Barrier. *J. ACM* 45, 5 (1998), 783–797. https://doi.org/10.1145/290179.290181 Announced at FOCS'97.

[66] Andrew V. Goldberg and Robert Endre Tarjan. 1988. A new approach to the maximum-flow problem. *J. ACM* 35, 4 (1988), 921–940. https://doi.org/10.1145/48014.61051

[67] Andrew V. Goldberg and Robert E. Tarjan. 1989. Finding Minimum-Cost Circulations by Canceling Negative Cycles. *J. ACM* 36, 4 (1989), 873–886.

[68] Andrew V Goldberg and Robert E Tarjan. 2014. Efficient maximum flow algorithms. *Commun. ACM* 57, 8 (2014), 82–89. Available at https://cacm.acm.org/magazines/2014/8/177011-efficient-maximum-flow-algorithms.

[69] Donald Goldfarb and Michael D Grigoriadis. 1988. A computational comparison of the Dinic and network simplex methods for maximum flow. *Annals of Operations Research* 13, 1 (1988), 81–123.

[70] Wenshuo Guo, Nhat Ho, and Michael Jordan. 2020. Fast algorithms for computational optimal transport and wasserstein barycenter. In *International Conference on Artificial Intelligence and Statistics*. PMLR, 2088–2097.

[71] Refael Hassin. 1983. The minimum cost flow problem: a unifying approach to dual algorithms and a new tree-search algorithm. *Mathematical Programming* 25 (1983), 228–239.

[72] Refael Hassin. 1992. Algorithms for the minimum cost circulation problem based on maximizing the mean improvement. *Operations research letters* 12, 4 (1992), 227–233.

[73] Monika Henzinger, Satish Rao, and Di Wang. 2020. Local Flow Partitioning for Faster Edge Connectivity. *SIAM J. Comput.* 49, 1 (2020), 1–36. Available at: https://arxiv.org/abs/1704.01254.

[74] D.S. Hochbaum and M. Queyranne. 2003. Minimizing a Convex Cost Closure Set. *SIAM Journal on Discrete Mathematics* 16, 2 (2003), 192–207.

[75] Dorit S. Hochbaum. 2008. The Pseudoflow Algorithm: A New Algorithm for the Maximum-Flow Problem. *Operations Research* 56, 4 (2008), 992–1009. https://doi.org/10.1287/opre.1080.0524

[76] John E. Hopcroft and Richard M. Karp. 1973. An $n^{5/2}$ Algorithm for Maximum Matchings in Bipartite Graphs. *SIAM J. Comput.* 2, 4 (Dec. 1973), 225–231. https://doi.org/10.1137/0202019

[77] Mark R Jerrum, Leslie G Valiant, and Vijay V Vazirani. 1986. Random generation of combinatorial structures from a uniform distribution. *Theoretical computer science* 43 (1986), 169–188.

[78] Narendra Karmarkar. 1984. A New Polynomial-Time Algorithm for Linear Programming. In *STOC*. ACM, 302–311.

[79] Alexander V Karzanov. 1973. On finding maximum flows in networks with special structure and some applications. *Matematicheskie Voprosy Upravleniya Proizvodstvom* 5 (1973), 81–94.

[80] Aleksandr V Karzanov. 1976. Efficient implementations of Edmonds' algorithms for finding matchings with maximum cardinality and maximum weight. *Studies in Discrete Optimization* (1976), 306–327.

[81] Tarun Kathuria, Yang P. Liu, and Aaron Sidford. 2020. Unit Capacity Maxflow in Almost $O(m^{4/3})$ Time. In *61st IEEE Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 119–130. https://doi.org/10.1109/FOCS46700.2020.00020

[82] Jonathan A. Kelner, Yin Tat Lee, Lorenzo Orecchia, and Aaron Sidford. 2014. An Almost-Linear-Time Algorithm for Approximate Max Flow in Undirected Graphs, and its Multicommodity Generalizations. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, Chandra Chekuri (Ed.). SIAM, 217–226.

[83] Jonathan A. Kelner, Gary L. Miller, and Richard Peng. 2012. Faster approximate multicommodity flow using quadratically coupled flows. In *Proceedings of the 44th Symposium on Theory of Computing Conference, STOC 2012, New York, NY, USA, May 19 - 22, 2012*, Howard J. Karloff and Toniann Pitassi (Eds.). ACM, 1–18. Available at: https://arxiv.org/abs/1202.3367.

[84] Rohit Khandekar, Satish Rao, and Umesh Vazirani. 2009. Graph partitioning using single commodity flows. *Journal of the ACM (JACM)* 56, 4 (2009), 1–15.

[85] Harold W. Kuhn. 2012. A tale of three eras: The discovery and rediscovery of the Hungarian Method. *Eur. J. Oper. Res.* 219, 3 (2012), 641–651.

[86] Rasmus Kyng, Richard Peng, Sushant Sachdeva, and Di Wang. 2019. Flows in Almost Linear Time via Adaptive Preconditioning. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*. 902–913.

[87] Rasmus Kyng, Anup Rao, and Sushant Sachdeva. 2015. Fast, provable algorithms for isotonic regression in all l_p-norms. *Advances in neural information processing systems* 28 (2015).

[88] Yin Tat Lee and Aaron Sidford. 2019. Solving Linear Programs with Sqrt(rank) Linear System Solves. *CoRR* abs/1910.08033 (2019). arXiv:1910.08033 http://arxiv.org/abs/1910.08033

[89] Yin Tat Lee, Zhao Song, and Qiuyi Zhang. 2019. Solving Empirical Risk Minimization in the Current Matrix Multiplication Time. In *Conference on Learning Theory, COLT 2019, Phoenix, AZ, USA, June 25-28, 2019 (Proceedings of Machine Learning Research, Vol. 99)*. PMLR, 2140–2157. Available at https://arxiv.org/abs/1905.04447.

[90] Yin Tat Lee and Santosh S Vempala. 2021. Tutorial on the Robust Interior Point Method. *arXiv preprint arXiv:2108.04734* (2021).

[91] Yin Tat Lee and Man-Chung Yue. 2021. Universal Barrier Is n-Self-Concordant. *Math. Oper. Res.* 46, 3 (2021), 1129–1148. Available at: https://arxiv.org/abs/1809.03011.

[92] Tom Leighton and Satish Rao. 1999. Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *Journal of the ACM (JACM)* 46, 6 (1999), 787–832.

[93] Jason Li, Danupon Nanongkai, Debmalya Panigrahi, Thatchaphol Saranurak, and Sorrachai Yingchareonthawornchai. 2021. Vertex Connectivity in Poly-Logarithmic Max-Flows. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing* (Virtual, Italy) *(STOC 2021)*. Association for Computing Machinery, New York, NY, USA, 317–329.

[94] Jason Li and Debmalya Panigrahi. 2020. Deterministic Min-cut in Poly-logarithmic Max-flows. In *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 85–92.

[95] Jason Li and Debmalya Panigrahi. 2021. Approximate Gomory–Hu Tree is Faster than n – 1 Max-Flows. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing* (Virtual, Italy) *(STOC 2021)*. Association for Computing Machinery, New York, NY, USA, 1738–1748. https://doi.org/10.1145/3406325.3451112

[96] Yang P Liu and Aaron Sidford. 2020. Faster energy maximization for faster maximum flow. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*. 803–814.

[97] Anand Louis. 2010. Cut-matching games on directed graphs. *arXiv preprint arXiv:1010.1047* (2010).

[98] László Lovász and Michael D Plummer. 2009. *Matching theory*. Vol. 367. American Mathematical Soc.

[99] Aleksander Mądry. 2010. Fast Approximation Algorithms for Cut-Based Problems in Undirected Graphs. In *51th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2010, October 23-26, 2010, Las Vegas, Nevada, USA*. IEEE Computer Society, 245–254. https://doi.org/10.1109/FOCS.2010.30

[100] Aleksander Mądry. 2013. Navigating central path with electrical flows: From flows to matchings, and back. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*. IEEE, 253–262.

[101] Aleksander Mądry. 2016. Computing Maximum Flow with Augmenting Electrical Flows. In *57th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE Computer Society, 593–602.

[102] Silvio Micali and Vijay V Vazirani. 1980. An O (v| v| c| E|) algoithm for finding maximum matching in general graphs. In *21st Annual Symposium on Foundations of Computer Science (sfcs 1980)*. IEEE, 17–27.

[103] Marcin Mucha and Piotr Sankowski. 2004. Maximum matchings via Gaussian elimination. In *45th Annual IEEE Symposium on Foundations of Computer Science*. IEEE, 248–255.

[104] Ketan Mulmuley, Umesh V Vazirani, and Vijay V Vazirani. 1987. Matching is as easy as matrix inversion. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*. 345–354.

[105] Danupon Nanongkai, Thatchaphol Saranurak, and Christian Wulff-Nilsen. 2017. Dynamic Minimum Spanning Forest with Subpolynomial Worst-Case Update Time. In *58th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2017, Berkeley, CA, USA, October 15-17, 2017*, Chris Umans (Ed.). IEEE Computer Society, 950–961. Available at: https://arxiv.org/abs/1708.03962.

[106] Arkadi Nemirovski. 2004. Interior point polynomial time methods in convex programming. *Lecture notes* 42, 16 (2004), 3215–3224. Available at https://www2.isye.gatech.edu/~nemirovs/Lect_IPM.pdf.

[107] Yu Nesterov. 1998. Introductory lectures on convex programming.

[108] Yurii E. Nesterov. 2004. *Introductory Lectures on Convex Optimization - A Basic Course*. Applied Optimization, Vol. 87. Springer. Available at: https://wwwfr.uni.lu/content/download/92121/1121193/file/NesB.pdf.
[109] James B. Orlin. 1993. A Faster Strongly Polynomial Minimum Cost Flow Algorithm. *Oper. Res.* 41, 2 (1993), 338–350.
[110] James B. Orlin. 1996. A Polynomial Time Primal Network Simplex Algorithm for Minimum Cost Flows. In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms* (Atlanta, Georgia, USA) *(SODA '96)*. Society for Industrial and Applied Mathematics, USA, 474–481.
[111] James B Orlin and Xiao-yue Gong. 2021. A fast maximum flow algorithm. *Networks* 77, 2 (2021), 287–321.
[112] James B. Orlin, Serge A. Plotkin, and Éva Tardos. 1993. Polynomial Dual Network Simplex Algorithms. *Math. Program.* 60, 1–3 (1993), 255–276.
[113] Richard Peng. 2016. Approximate undirected maximum flows in $O(m\text{polylog}(n))$ time. In *Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete algorithms*. SIAM, 1862–1867.
[114] Seth Pettie, Thatchaphol Saranurak, and Longhui Yin. 2022. Optimal Vertex Connectivity Oracles. *Accepted to STOC'22* (2022).
[115] Michael O Rabin and Vijay V Vazirani. 1989. Maximum matchings in general graphs through randomization. *Journal of algorithms* 10, 4 (1989), 557–567.
[116] Harald Räcke. 2008. Optimal hierarchical decompositions for congestion minimization in networks. In *Proceedings of the 40th Annual ACM Symposium on Theory of Computing, Victoria, British Columbia, Canada, May 17-20, 2008*, Cynthia Dwork (Ed.). ACM, 255–264. https://doi.org/10.1145/1374376.1374415
[117] James Renegar. 1988. A polynomial-time algorithm, based on Newton's method, for linear programming. *Mathematical programming* 40, 1 (1988), 59–93.
[118] Thatchaphol Saranurak and Di Wang. 2019. Expander Decomposition and Pruning: Faster, Stronger, and Simpler. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, Timothy M. Chan (Ed.). SIAM, 2616–2635. Available at: https://arxiv.org/abs/1812.08958.
[119] Jonah Sherman. 2013. Nearly Maximum Flows in Nearly Linear Time. In *54th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*. IEEE Computer Society, 263–269. https://doi.org/10.1109/FOCS.2013.36
[120] Jonah Sherman. 2017. Area-convexity, ℓ_∞ regularization, and undirected multicommodity flow. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*. 452–460.
[121] Alistair Sinclair and Mark Jerrum. 1989. Approximate counting, uniform generation and rapidly mixing Markov chains. *Information and Computation* 82, 1 (1989), 93–133.
[122] Daniel D Sleator and Robert Endre Tarjan. 1983. A data structure for dynamic trees. *Journal of computer and system sciences* 26, 3 (1983), 362–391.
[123] Daniel A. Spielman and Shang-Hua Teng. 2004. 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)*. 81–90.
[124] Daniel A Spielman and Shang-Hua Teng. 2003. 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.* IEEE, 416–427.
[125] Q. F. Stout. 2013. Isotonic Regression via Partitioning. *Algorithmica* 66, 1 (2013), 93–112. https://doi.org/10.1007/s00453-012-9628-4
[126] Quentin F. Stout. 2021. ℓ_p Isotonic Regression Algorithms Using an ℓ_0 Approach. *ArXiv* abs/2107.00251 (2021).
[127] Éva Tardos. 1985. A Strongly Polynomial Minimum Cost Circulation Algorithm. *Combinatorica* 5, 3 (1985), 247–255.
[128] Pravin M. Vaidya. 1990. An Algorithm for Linear Programming which Requires $O(((m+n)n^2+(m+n)^{1.5}n)L)$ Arithmetic Operations. *Math. Program.* 47 (1990), 175–201.
[129] Leslie G Valiant. 1979. The complexity of computing the permanent. *Theoretical computer science* 8, 2 (1979), 189–201.
[130] Jan van den Brand, Yu Gao, Arun Jambulapati, Yin Tat Lee, Yang P Liu, Richard Peng, and Aaron Sidford. 2022. Faster maxflow via improved dynamic spectral vertex sparsifiers. In *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing*. 543–556.
[131] Vijay V Vazirani. 2023. A Theory of Alternating Paths and Blossoms, from the Perspective of Minimum Length. (2023). Submitted to *Mathematics of Operations Research*.
[132] C. Wallacher and U. Zimmermann. 1992. A Combinatorial Interior Point Method for Network Flow Problems. *Mathematical Programming* 56, 1-3 (Aug. 1992), 321–335. https://doi.org/10.1007/BF01580905
[133] Di Wang, Kimon Fountoulakis, Monika Henzinger, Michael W Mahoney, and Satish Rao. 2017. Capacity releasing diffusion for speed and locality. In *International Conference on Machine Learning*. PMLR, 3598–3607.
[134] Tianyi Zhang. 2021. Gomory-Hu Trees in Quadratic Time. *CoRR* abs/2112.01042 (2021). arXiv:2112.01042 Available at: https://arxiv.org/abs/2112.01042.

A OMITTED PROOFS

A.1 Proof of Lemma 6.12

Proof. The graph $\widetilde{G}$ will have one more vertex than G, denoted by v^*. Additionally, we will add a directed edge between v^* and v for all $v \in V(G)$, where the direction will be decided later. Thus, $\widetilde{G}$ will have at most $m+n$ edges.

Initially define $\boldsymbol{f}_e^{(\text{init})} = (\boldsymbol{u}_e^- + \boldsymbol{u}_e^+)/2$ for all $e \in E(G)$. However, $\boldsymbol{f}^{(\text{init})}$ will not route the demand $\boldsymbol{d}$, and we denote the demand it routes by $\overline{\boldsymbol{d}} \stackrel{\text{def}}{=} \boldsymbol{B}^\top \boldsymbol{f}^{(\text{init})}$. Now, we will describe how to generate the edge between v^* and v. If $\overline{\boldsymbol{d}}_v = \boldsymbol{d}_v$, then we do not add any edge between v and v^*. If $\overline{\boldsymbol{d}}_v > \boldsymbol{d}_v$, then add an edge $e_v = (v \to v^*)$ with upper capacity $2(\overline{\boldsymbol{d}}_v - \boldsymbol{d}_v)$ (and lower capacity 0), and set $\boldsymbol{f}_{e_v}^{(\text{init})} = \overline{\boldsymbol{d}}_v - \boldsymbol{d}_v$. If $\overline{\boldsymbol{d}}_v < \boldsymbol{d}_v$ then add an edge $e_v = (v^* \to v)$ of upper capacity $2(\boldsymbol{d}_v - \overline{\boldsymbol{d}}_v)$ (and lower capacity 0), and set $\boldsymbol{f}_{e_v}^{(\text{init})} = \boldsymbol{d}_v - \overline{\boldsymbol{d}}_v$. Finally, set $\widetilde{\boldsymbol{d}}_{v^*} = 0$ and $\widetilde{\boldsymbol{d}}_v = \boldsymbol{d}_v$ for all $v \in V(G)$, and $\widetilde{\boldsymbol{c}}_{e_v} = 4mU^2$ for the new edges e_v, and $\widetilde{\boldsymbol{c}}_e = \boldsymbol{c}_e$ for all $e \in E(G)$.

It is direct to check that all capacities of edges in $\widetilde{G}$ are integral and bounded by $2mU$, and costs are bounded by $4mU^2$. It suffices then to prove that if G supports a feasible flow, then the optimal flow in G must put 0 units on any of the e_v edges. Indeed, note that the e_v edges only can contribute nonnegative cost as $\widetilde{\boldsymbol{f}}_{e_v} \geq 0$ by definition, and if any of them supports one unit of flow (i.e. $\widetilde{\boldsymbol{f}}_{e_v} \geq 1$), then that contributes $4mU^2$ to the cost. The maximum possible cost of edges $e \in E$ is bounded by mU^2, as there are m edges of capacity at most U, each with cost at most U, and the minimum is at least $-mU^2$. Hence if G supports a feasible flow, $\widetilde{\boldsymbol{f}}_{e_v} = 0$ for all the e_v edges.

We conclude by bounding $\Phi(\boldsymbol{f}^{(\text{init})})$. Note that $\widetilde{\boldsymbol{f}}_e$ are all half-integral, and $\widetilde{\boldsymbol{f}}_e \leq mU$ for all $e \in E(\widetilde{G})$. Also, $F^* \geq -mU^2$ by the above discussion, and because all costs are bounded by $4mU^2$, $\widetilde{\boldsymbol{c}}^\top \boldsymbol{f}^{(\text{init})} \leq 2m \cdot 4mU^2 \cdot mU = 8m^3U^3$. Thus

$$\Phi(\boldsymbol{f}^{(\text{init})}) \stackrel{(i)}{\leq} 20m \log(8m^3U^3 + mU^2) + \sum_{e \in E} \left((1/2)^{-\alpha} + (1/2)^{-\alpha}\right) \leq 200m \log mU,$$

where (i) used the above bounds and that $\boldsymbol{u}_e^+ - \boldsymbol{f}_e^{(\text{init})}, \boldsymbol{f}_e^{(\text{init})} - \boldsymbol{u}_e^-$ are all half-integral. □

A.2 Proof of Theorem 7.11

The goal of this section is to show Theorem 7.11. To obtain our result, we require the following result from [118].

Theorem A.1 (see [118, Theorem 1.2]). *Given an unweighted, undirected m-edge graph G, there is an algorithm that finds a partition of $V(G)$ into sets $V_1, V_2, \ldots, V_k$ such that for each $1 \leq j \leq k$, $G[V_j]$ is a ψ-expander for $\psi = \Omega(1/\log^3(m))$ and there are at most $m/4$ edges that are not contained in one of the expander graphs. The algorithm runs in time $O(m \log^7(m))$ and succeeds with probability at least $1 - n^{-C}$ for any constant C fixed before the start of the algorithm.*

We run Algorithm 8 (given below) to obtain the graphs G_i as desired in Theorem 7.11.

Algorithm 8: DECOMPOSE(G)

1 $\ell \leftarrow \lceil \log_2 \Delta_{max}(G) \rceil + 1$; $G_\ell = G$;
2 **for** $i = \ell, \ell - 1, \ldots, 1$ **do**
3 Let $G_i^{\circlearrowleft}$ denote the graph G_i after adding 2^i self-loops to each vertex.;
4 Compute an expander decomposition $V_0, V_1, \ldots, V_k$ of $G_i^{\circlearrowleft}$ as described in Theorem A.1.;
5 $G_{i-1} \leftarrow \left(\bigcup_{0 \leq j \leq k} E_{G_i}(V_j, V \setminus V_j)\right)$.;
6 $G_i \leftarrow G_i \setminus G_{i-1}$.

Claim A.2. *For each i, the graph G_i has at initialization in Line 1 or Line 5 at most $2^i n$ edges.*

PROOF. We prove by induction on i. For the base case, $i = \ell$, observe that $2^\ell \geq \Delta_{max}(G)$ and since G_ℓ is a subgraph of G, we have $|E(G_\ell)| \leq 2^\ell n$.

For $i \mapsto i-1$, we observe that G_i is unchanged since its initialization until at least after G_{i-1} was defined in Line 5. Thus, using the induction hypothesis and the fact above, we can conclude that $G_i^{\circlearrowleft}$ (defined in Line 3) consists of at most $2^i n$ edges from G_i plus $2^i n$ edges from all self-loops. But by Theorem A.1, this implies that $|\bigcup_{0\le j\le k} E_{G_i}(V_j, V\setminus V_j)| = |\bigcup_{0\le j\le k} E_{G_i^{\circlearrowleft}}(V_j, V\setminus V_j)| \le 2^{i+1}n/4 = 2^{i-1}n$, and since this is exactly the edge set of G_{i-1}, the claim follows. □

Proof of Theorem 7.11. Using Claim A.2 and the insight that each graph G_i, after initialization, can only have edges deleted from it, we conclude that $|E(G_i)| \le 2^i n$ for each i.

For the minimum degree property of each G_i with $i > 0$, we observe by Theorem A.1, that for $G_i^{\circlearrowleft}$ and vertex v in expander V_j, $\deg_{G_i}(v) = |E_{G_i}(v, V_j \setminus \{v\})| = |E_{G_i^{\circlearrowleft}}(v, V_j \setminus \{v\})| \ge \psi 2^i$. □

A.3 Proof of Lemma 8.5

We show Lemma 8.5 using the following steps. First, we assume for the majority of the section that the weights $\boldsymbol{\upsilon} = \mathbf{1}$, i.e. the all ones vector. We explain later a standard reduction to this case. Given a low stretch tree T on a graph with lengths $\boldsymbol{\ell}$, and a target set of roots R, we explain how to find a forest F (depending on R) that has low total stretch (Definition 8.4). This involves defining a notion of congestion on edges $e \in E(T)$. Then we explain how to handle dynamic edge insertions and deletions by adding new roots to the tree, and decrementally maintain the forest. The trickiest part is to explain how to add roots so that we can return valid stretch *overestimates*. At a high level, this is done by computing a heavy-light decomposition of T, and using it to inform our root insertions.

It is useful to maintain the invariant that our set of roots is *branch-free* at all times, i.e. that the lowest common ancestor (LCA) of any two roots $r_1, r_2 \in R$ is also in R. This is necessary to make it easier to construct the forest F given R. Intuitively, forcing our set of roots to be branch-free is not a big restriction, as any set of roots can be made branch free by at most doubling its size.

Definition A.3 (Branch-free sets). *For a rooted tree T on vertices V, we say that a set $R \subseteq V$ is* branch-free *if the LCA of any vertices $r_1, r_2 \in R$ is also in R.*

Given a branch-free set of roots R, we build a forest F in the following way. We start with some total ordering/permutation π on the tree edges $E(T)$, and for any two "adjacent" roots $r_1, r_2 \in R$, we delete the smallest edge with respect to π from T. Here, adjacent means that no root is on the path between r_1, r_2. It is crucial in this construction that R is branch-free, so that there are exactly $|R|-1$ adjacent pairs of roots.

Definition A.4 (Forest given roots). *Given a rooted tree T, a branch-free set of roots $R \subseteq V$, and a total ordering π on $E(T)$, define $F_T(R, \pi)$ as the forest obtained by removing the smallest tree edge with respect to π from every path between adjacent roots in T.*

It is direct to verify that $F_T(R, \pi)$ has exactly $|R|$ connected components, each of which contains a unique vertex in R. We now explain how to construct π. π sorts the edges by their *congestions*.

Definition A.5 (Congestion). *Given a graph $G = (V, E)$ with lengths $\boldsymbol{\ell}$, tree T we define the* congestions *of edges $e \in E(T)$ as*

$$\mathsf{cong}_e^{T,\boldsymbol{\ell}} \stackrel{\text{def}}{=} \sum_{\substack{e'=(u,v)\in E(G)\\ \text{s.t. } e\in T[u,v]}} 1/\boldsymbol{\ell}_{e'}.$$

We show that if π is ordered by increasing congestions, then $F = F_T(R, \pi)$ has low total stretch.

Lemma A.6 (Valid π). *For a graph $G = (V, E)$ with lengths $\boldsymbol{\ell}$, a rooted tree T, and a branch-free set of roots R, let π be a total ordering on $E(T)$ sorted by increasing $\mathrm{cong}_e^{T,\boldsymbol{\ell}}$ (Definition A.5). Then for $F = F_T(R, \pi)$, we have $\sum_{e\in E} \mathrm{str}_e^{F,\boldsymbol{\ell}} \le 2\sum_{e\in E} \mathrm{str}_e^{T,\boldsymbol{\ell}}$.*

Proof. Let $\widehat{E}$ be the set of edges deleted from $E(T)$ to get $F_T(R, \pi)$. For an edge $e \in \widehat{E}$ on a path between adjacent roots $r_1(e), r_2(e) \in R$, define $L_e = \langle \boldsymbol{\ell}, |\boldsymbol{p}(T[r_1(e), r_2(e)])| \rangle$ as the length of the path between $r_1(e), r_2(e)$ in T. First note by Definition 8.4 that for an edge $e' = (u, v) \in E(G)$

$$\mathrm{str}_{e'}^{F,\boldsymbol{\ell}} \le \mathrm{str}_{e'}^{T,\boldsymbol{\ell}} + \sum_{e\in\widehat{E}\cap T[u,v]} L_e/\boldsymbol{\ell}_{e'}. \tag{55}$$

Thus, we can bound

$$\begin{aligned}
\sum_{e'\in E(G)} \mathrm{str}_{e'}^{F,\boldsymbol{\ell}} &\overset{(i)}{\le} \sum_{e'\in E(G)} \mathrm{str}_{e'}^{T,\boldsymbol{\ell}} + \sum_{e'=(u,v)\in E(G)} \sum_{e\in\widehat{E}\cap T[u,v]} L_e/\boldsymbol{\ell}_{e'} \\
&= \sum_{e'\in E(G)} \mathrm{str}_{e'}^{T,\boldsymbol{\ell}} + \sum_{e\in\widehat{E}} L_e \mathrm{cong}_e^{T,\boldsymbol{\ell}} = \sum_{e'\in E(G)} \mathrm{str}_{e'}^{T,\boldsymbol{\ell}} + \sum_{e\in\widehat{E}} \sum_{f\in T[r_1(e),r_2(e)]} \boldsymbol{\ell}_f \mathrm{cong}_e^{T,\boldsymbol{\ell}} \\
&\overset{(ii)}{\le} \sum_{e'\in E(G)} \mathrm{str}_{e'}^{T,\boldsymbol{\ell}} + \sum_{e\in\widehat{E}} \sum_{f\in T[r_1(e),r_2(e)]} \boldsymbol{\ell}_f \mathrm{cong}_f^{T,\boldsymbol{\ell}} \overset{(iii)}{\le} \sum_{e'\in E(G)} \mathrm{str}_{e'}^{T,\boldsymbol{\ell}} + \sum_{e\in E(T)} \boldsymbol{\ell}_e \mathrm{cong}_e^{T,\boldsymbol{\ell}} \\
&= 2 \sum_{e'\in E(G)} \mathrm{str}_{e'}^{T,\boldsymbol{\ell}}.
\end{aligned}$$

Here (i) follows from (55), (ii) follows from the fact that π is sorted by increasing $\mathrm{cong}_e^{T,\boldsymbol{\ell}}$ so $\mathrm{cong}_e^{T,\boldsymbol{\ell}} \le \mathrm{cong}_f^{T,\boldsymbol{\ell}}$ for all $f \in T[r_1(e), r_2(e)]$, and (iii) follows as $T[r_1(e), r_2(e)]$ are disjoint paths. □

To handle item 4 of Lemma 8.5, we initialize the set of roots R to have size $O(m/k)$ to already satisfy item 4. This set of roots R exists by a standard decomposition result due to [123].

Lemma A.7 (Tree Decomposition, [123, 124]). *There is a deterministic linear-time algorithm that on a graph $G = (V, E)$ with weights $\boldsymbol{w} \in \mathbb{R}^E_{>0}$, 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. *$R \overset{\text{def}}{=} \partial\mathcal{W} \subseteq V$, defined as the subset of vertices appear in multiple components, is branch-free.*
3. *For every component $C \subseteq V$ of $\mathcal{W}$, the total weight of edges adjacent to non-boundary vertices of C is at most $O(\|\boldsymbol{w}\|_1 \cdot k/m)$, i.e. $\sum_{e:e\cap C\not\subseteq\partial\mathcal{W}} \boldsymbol{w}_e \le 40 \cdot \|\boldsymbol{w}\|_1 \cdot k/m$.*

Item 4 of Lemma 8.5 then follows from Lemma A.7 by taking $\boldsymbol{w}_e = 1$ for all $e \in E$.

We now explain how to add roots to R under insertions and deletions of edges $e = (u, v)$. While a naïve approach is to add both u, v to the set R, i.e. $R \leftarrow R \cup \{u, v\}$ (and then add more roots to make it branch free), this does not work because $\mathrm{str}_e^{T,\boldsymbol{\ell}}$ might fluctuate significantly, and not be globally upper bounded as we want. To fix this, we introduce a more complex procedure that adds $\widetilde{O}(1)$ additional roots to R to control the number of potential roots that an edge e is assigned too.

Formally, we will construct an auxiliary tree with the same root and vertex set as T. This tree is constructed via a heavy-light decomposition on T. Then we replace each heavy chain with a balanced binary tree. Thus, this auxiliary tree has height $O(\log^2 n)$. When a vertex u is added to R, we walk up the tree induced by the heavy-light decomposition

and add all the ancestors of u. Thus, at most $O(\log^2 n)$ additional vertices will be added to R, but each edge will only be assigned to $O(\log^2 n)$ distinct roots.

We introduce one additional piece of notation. Given a rooted tree T_H (the tree defined by the heavy-light decomposition on T) and vertex u, define $u^{\uparrow T_H}$ as the set of its ancestors in T_H plus itself. We extend the notation to any subset of vertices by defining $R^{\uparrow T_H} = \bigcup_{u \in R} u^{\uparrow T_H}$.

Lemma A.8 (Heavy-Light Decomposition of Trees, [122]). *There is a linear-time algorithm that given a rooted tree T with n vertices outputs a collection of vertex disjoint tree paths $\{P_1, \ldots, P_t\}$ (called* heavy chains*), such that the following hold for every vertex u:*

1. *There is exactly one heavy chain P_i containing u.*
2. *If P_i is the heavy chain containing u, at most one child of u is in P_i.*
3. *There are at most $O(\log n)$ heavy chains that intersect with $u^{\uparrow T}$.*

In addition, edges that are not covered by any heavy chain are called light edges.

Lemma A.9. *There is a linear-time algorithm that given a tree T rooted at r with n vertices outputs a rooted tree T_H supported on the same vertex set such that*

1. *The height of T_H is $O(\log^2 n)$.*
2. *For any subset of vertices R in T, $R^{\uparrow T_H}$ is branch-free in T.*
3. *Given any total ordering on tree edges π and nonempty vertex subset $R \subseteq V(T)$, $\text{root}_u^F \in u^{\uparrow T_H}$ for every vertex u where $F = F_T(R^{\uparrow T_H}, \pi)$.*
4. *Given any total ordering on tree edges π and nonempty vertex subsets $R_1, R_2 \subseteq V(T)$, $\text{root}_u^{F_1} = \text{root}_u^{F_2}$ if $R_1^{\uparrow T_H} \cap u^{\uparrow T_H} = R_2^{\uparrow T_H} \cap u^{\uparrow T_H}$ where $F_i = F_T(R_i^{\uparrow T_H}, \pi), i = 1, 2$. That is, the root of u in any rooted spanning forest of the form $F_T(R^{\uparrow T_H}, \pi)$ is determined by the intersection of u's ancestors and forest roots.*

Proof. We first present the construction of the rooted tree T_H. We root T_H at the root r of T and compute its heavy-light decomposition in linear-time via Lemma A.8. Let $\{P_1, \ldots, P_t\}$ be the resulting decomposition. For every path P_i, we build a balanced binary search tree (BST) T_i over its vertices, $V(P_i)$, with respect to their depth in T. The depth of a vertex in T is defined as the distance to the root r. In addition, we make the vertex with minimum depth the root of the BST T_i. T_H is then obtained from T by replacing every path P_i by BST T_i.

To show condition 1, observe that the path $T_H[u, r]$ consists of $O(\log n)$ node-to-root paths in some balanced BSTs and $O(\log n)$ light edges. Each node-to-root path in some balanced BST has length at most $O(\log n)$. Therefore, the $T_H[u, r]$-path has length at most $O(\log^2 n)$.

Next, we prove condition 2. For any two vertices u and v in $R^{\uparrow T_H}$, let w be their lowest common ancestor in T. Let P_w be the heavy chain containing w. Thus, for at least one of $T[u, r]$ and $T[v, r]$, w must be the first vertex of P_w that appears on that path or else w has two distinct children that belong to P_w. Then, w appears in either $T_H[u, r]$ or $T_H[v, r]$ as well and thus is included in $R^{\uparrow T_H}$.

We prove Condition 3 by induction on the depth of u in T_H, i.e. the size of $u^{\uparrow T_H}$. If $|u^{\uparrow T_H}| = 1$, u is the root of T_H and $R^{\uparrow T_H}$ contains u for any nonempty R. Thus, root_u^F is u itself. Next, we consider the case where $|u^{\uparrow T_H}| = k + 1$. Let v be the first vertex in $R^{\uparrow T_H}$ on the path $T_H[u, r]$. Let P be the heavy chain containing v and b be the first vertex in P on the path $T_H[u, r]$.

If P does not contain u, let a be the vertex before b on the path $T_H[u, r]$. The sequence u, a, b, v shows up in the same order as in the path $T_H[u, r]$. Since $R^{\uparrow T_H}$ does not contain a, $R^{\uparrow T_H}$ does not contain any vertex in the subtree of

T_H rooted at a as well as the subtree of T rooted at a. Thus, u, a, and b are connected in the forest $F_T(R^{\uparrow T_H}, \pi)$ and share the same root. The size of $b^{\uparrow T_H}$ is less than the size of $u^{\uparrow T_H}$ and we can apply induction hypothesis to argue that $\text{root}_u^F = \text{root}_b^F \in b^{\uparrow T_H} \subsetneq u^{\uparrow T_H}$.

If P contains u, we will show that u is connected to some other vertex $w \in P \cap u^{\uparrow T_H}$ in the rooted forest $F_T(R^{\uparrow T_H}, \pi)$. Let C be the set of vertices in P connected to u. Observe that C forms a contiguous subpath of P and contains one root from $R^{\uparrow T_H}$. Recall that the subtree in T_H corresponding to P is a balanced binary search tree keyed by depth in T. Let B be such binary search tree. It is known that given a binary search tree and a range on keys, the set of nodes in the tree within the range is closed under taking lowest common ancestor. Let w be the lowest common ancestor of all vertices in C in the BST B. w must be an element of $R^{\uparrow T_H}$ and therefore $\text{root}_u^F = w$. This concludes the proof of Condition 3.

To prove Condition 4, it suffices to argue the case where $R_2 = R_1 \cup \{r\}$ and $R_1^{\uparrow T_H}$ contains every ancestor of r. Specifically, we prove that $\text{root}_u^{F_2} = \text{root}_u^{F_1}$ for every vertex u which does not have r as its ancestor. Let C be the component of F_1 in which r lives and w be the root of C. Adding r as a new root removes some edge between r and w and divides C into two components C_1 and C_2. Suppose that $r \in C_1$ and $w \in C_2$. Condition 3 says that r is ancestor w.r.t. T_H to every vertex in C_1. However, only vertices in C_1 have their root changed. This concludes the proof of Condition 4. □

We now provide an algorithm for Lemma 8.5 and prove that it works. At a high level, the algorithm will first compute an LSST. Then it will compute global stretch overestimates based on the tree T_H from Lemma A.9. Then, it will initialize a set of roots of size $O(m/k)$ by adding all endpoints of large stretch edges as terminals, and by calling Lemma A.7 to bound the degree of each component of F as needed in item 4 of Lemma 8.5.

Proof of Lemma 8.5. For the weights $\boldsymbol{v}$, we first construct a graph $G_{\boldsymbol{v}}$ that has $\lceil m\boldsymbol{v}_e / \|\boldsymbol{v}\|_1 \rceil$ unweighted copies of the edge e for each $e \in E(G)$. Note that $G_{\boldsymbol{v}}$ has at most $2m$ edges:

$$\sum_{e \in E} \left\lceil \frac{m\boldsymbol{v}_e}{\|\boldsymbol{v}\|_1} \right\rceil \le \sum_{e \in E} \left(1 + \frac{m\boldsymbol{v}_e}{\|\boldsymbol{v}\|_1}\right) = 2m.$$

Let T be a LSST on $G_{\boldsymbol{v}}$ computed using Theorem 5.2, with an arbitrarily chosen root r, and let T_H be the tree in Lemma A.9. Let π be the permutation sorted by increasing congestions (Lemma A.6).

Notice that $G \subseteq G_{\boldsymbol{v}}$ and thus every spanning tree/forest of $G_{\boldsymbol{v}}$ is also a spanning tree/forest of G. Furthermore, either the tree stretch or forest stretch of edge $e \in E(G)$ is equal to the one of any of e's copy in $G_{\boldsymbol{v}}$.

We now explain how to compute stretch overestimates $\widetilde{\text{str}}_e$. For $i \ge 0$, let B_i be the set of vertices within distance i of the root r in T_H. Let $D = O(\log^2 n)$ be the height of T_H. We define

$$\widetilde{\text{str}}_e \stackrel{\text{def}}{=} 2 \sum_{i=0}^{D} \text{str}_e^{F_T(B_i, \pi), \boldsymbol{\ell}}. \tag{56}$$

In this definition, $\widetilde{\text{str}}_e$ take identical values among copies of e in $G_{\boldsymbol{v}}$. By Lemma A.6 we know that

$$\sum_{e \in E(G_{\boldsymbol{v}})} \widetilde{\text{str}}_e = 2 \sum_{i=0}^{D} \sum_{e \in E(G_{\boldsymbol{v}})} \text{str}_e^{F_T(B_i, \pi), \boldsymbol{\ell}} \le O(D \cdot m\gamma_{LSST}) = O(m\gamma_{LSST} \log^2 n). \tag{57}$$

Thus the total stretch bound is fine. Shortly, we will explain the full algorithm for maintaining the set of roots and why $\widetilde{\text{str}}_e$ are valid stretch overestimates for our algorithm.

To explain how we maintain the set of roots, we first explain how to initialize a set of roots. First run Lemma A.7 on T with uniform weights $\boldsymbol{w}_e = 1$ for all $e \in E(G)$ to output a set $|\mathcal{W}^T| = O(m/(k \log^2 n))$, and such that each component

has total adjacent weight k (minus the boundaries), i.e. at most k adjacent edges (as $\boldsymbol{w}_e = 1$ for all e). Start by defining $R_0 \leftarrow \partial \mathcal{W}^T$. So far, $|R_0| = O(m/(k \log^2 n))$.

Also for any edge $e \in E(G)$ with $\widetilde{\mathrm{str}}_e \geq O(k\gamma_{LSST} \log^4 n)$, add both endpoints of e to R_0. As

$$\sum_{e \in E(G)} \widetilde{\mathrm{str}}_e \leq \sum_{e \in E(G)} \left\lceil \frac{m\boldsymbol{v}_e}{\|\boldsymbol{v}\|_1} \right\rceil \widetilde{\mathrm{str}}_e = \sum_{e \in E(G_{\boldsymbol{v}})} \widetilde{\mathrm{str}}_e \leq O(m\gamma_{LSST} \log^2 n),$$

Markov's inequality tells us that the number of edges $e \in E(G)$ with $\widetilde{\mathrm{str}}_e \geq O(k\gamma_{LSST} \log^4 n)$ is bounded by $O(m/(k \log^2 n))$. Thus, overall $|R_0| = O(m/(k \log^2 n))$. Now, define our initial sets of branch-free roots as $R \stackrel{\text{def}}{=} R_0^{\uparrow T_H}$. Because the height of T_H is $O(\log^2 n)$ (Lemma A.9), we know $|R| = O(m/k)$.

We now handle edge insertions and deletions. When an edge $e = (u, v)$ is inserted or deleted, we add $u^{\uparrow T_H} \cup v^{\uparrow T_H}$ to R, i.e. $R \leftarrow R \cup (u^{\uparrow T_H} \cup v^{\uparrow T_H})$. This is branch free by Lemma A.9. If e was inserted, assign it to have $\widetilde{\mathrm{str}}_e = 1$, as both endpoints are roots in R. We also update the forest $F \stackrel{\text{def}}{=} F_T(R, \pi)$. Because R is incremental and π is a total ordering, F is decremental.

We now verify all items of Lemma 8.5. Item 1 follows because initially $|R| = O(m/k)$, and the height of T_H is $O(\log^2 n)$, so each edge insertion/deletion increases the size of R by $O(\log^2 n)$. Item 3 follows because

$$\sum_{e \in E(G)} \boldsymbol{v}_e \widetilde{\mathrm{str}}_e \leq \frac{\|\boldsymbol{v}\|_1}{m} \sum_{e \in E(G)} \left\lceil \frac{m\boldsymbol{v}_e}{\|\boldsymbol{v}\|_1} \right\rceil \widetilde{\mathrm{str}}_e \leq \frac{\|\boldsymbol{v}\|_1}{m} \sum_{e \in E(G_{\boldsymbol{v}})} \widetilde{\mathrm{str}}_e = O(\|\boldsymbol{v}\|_1 \gamma_{LSST} \log^2 n),$$

where the last inequality follows from (57).

Let $F_0 \stackrel{\text{def}}{=} F_T(R, \pi)$ be the initial rooted spanning forest to be output. Let $\mathcal{W}$ be a refinement of $\mathcal{W}^T$ induced by the connectivity in F_0. We output $\mathcal{W}$ as the desired edge-disjoint partition of F_0 into $O(m/k)$ subtrees. $\mathcal{W}$ contains at most $O(m/k)$ subtrees because F_0 is obtained from T by removing $|R| - 1$ edges and $\mathcal{W}^T$ is a edge-disjoint partition of T. Item 4 follows because $R \supseteq \partial \mathcal{W}$, where $\mathcal{W}$ was a partition of G into pieces of total degree $O(k)$.

We conclude by checking item 2, i.e. that $\widetilde{\mathrm{str}}_e$ upper-bounds $\mathrm{str}_e^{F,\boldsymbol{\ell}}$ at any moment for every edge $e = (u, v)$. If u, v are in the same connected component of F, then $\mathrm{str}_e^{F,\boldsymbol{\ell}} = \mathrm{str}_e^{T,\boldsymbol{\ell}} \leq \widetilde{\mathrm{str}}_e$ by noting that $T = F_T(B_0^{\uparrow T_H}, \pi)$. In the other case where u and v are disconnected, let $R^{\uparrow T_H}$ be the current set of roots of F. There must be some non-negative integer i (and j) such that $u^{\uparrow T_H} \cap R^{\uparrow T_H} = u^{\uparrow T_H} \cap B_i$ (and $v^{\uparrow T_H} \cap R^{\uparrow T_H} = v^{\uparrow T_H} \cap B_j$ respectively). To finish, note that item 4 of Lemma A.9 ensures that

$$\mathrm{root}_u^F = \mathrm{root}_u^{F_i}, \mathrm{root}_v^F = \mathrm{root}_v^{F_j}, \text{ and therefore}$$
$$\mathrm{str}_e^{F,\boldsymbol{\ell}} \leq \mathrm{str}_e^{F_i,\boldsymbol{\ell}} + \mathrm{str}_e^{F_j,\boldsymbol{\ell}} \leq \widetilde{\mathrm{str}}_e. \qquad \square$$

Finally, it can be checked that the total runtime is $\widetilde{O}(m)$, as every operation can be implemented efficiently.

A.4 Proof of Lemma 8.6

Proof. Let $W = O(\gamma_{LSST} \log^2 n)$ be such that items 2, 3 of Lemma 8.5 imply

$$\sum_{e \in E} \boldsymbol{v}_e \widetilde{\mathrm{str}}_e \leq W \|\boldsymbol{v}\|_1, \text{ and}$$
$$\max_{e \in E} \widetilde{\mathrm{str}}_e \leq kW \log^2 n.$$

Let $t = 10kW \log^2 n = \widetilde{O}(k)$. The algorithm sequentially constructs edge weights $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_t$ in a multiplicative weight update fashion and trees $T_1, \ldots, T_t$, forests $F_1, \ldots, F_t$, and stretch overestimates $\widetilde{\mathrm{str}}^1, \ldots, \widetilde{\mathrm{str}}^t$ via Lemma 8.5.

Initially, $\boldsymbol{v}_1 \stackrel{\text{def}}{=} \mathbf{1}$ is an the all 1's vector. After computing T_i, $\boldsymbol{v}_{i+1}$ is defined as

$$\boldsymbol{v}_{i+1,e} \stackrel{\text{def}}{=} \boldsymbol{v}_{i,e} \exp\left(\frac{\widetilde{\text{str}}_e^i}{t}\right) = \exp\left(\frac{1}{t}\sum_{j=1}^{i}\widetilde{\text{str}}_e^j\right) \text{ for all } e \in E.$$

Finally we define the distribution $\boldsymbol{\lambda}$ to be uniform over the set $\{1, \ldots, t\}$.

To show the desired bound (25), we first relate it with $\|\boldsymbol{v}_{t+1}\|$ using the following:

$$\max_{e\in E}\frac{1}{t}\sum_{i=1}^{t}\widetilde{\text{str}}_e^i \le \log\left(\sum_e \exp\left(\frac{1}{t}\sum_{i=1}^{t}\widetilde{\text{str}}_e^i\right)\right) = \log\|\boldsymbol{v}_{t+1}\|_1,$$

where $\boldsymbol{v}_{t+1}$ is defined similarly even though it is never used in the algorithm.

Next, we upper bounds $\|\boldsymbol{v}_i\|_1$ inductively for every $i = 1, \ldots, t+1$. Initially, $\boldsymbol{v}_1 = \mathbf{1}$ and we have $\|\boldsymbol{v}_1\|_1 = m$. To bound $\|\boldsymbol{v}_{i+1}\|$, we plug in the definition and have the following:

$$\begin{aligned}\|\boldsymbol{v}_{i+1}\|_1 &= \sum_e \boldsymbol{v}_{i,e}\exp\left(\frac{\widetilde{\text{str}}_e^i}{t}\right) \le \sum_e \boldsymbol{v}_{i,e}\left(1 + 2\cdot\frac{\widetilde{\text{str}}_e^i}{t}\right)\\ &= \|\boldsymbol{v}_i\|_1 + \frac{2}{t}\sum_e \boldsymbol{v}_{i,e}\widetilde{\text{str}}_e^i \le \|\boldsymbol{v}_i\|_1 + \frac{2}{t}W\|\boldsymbol{v}_i\|_1 = \left(1+\frac{2W}{t}\right)\|\boldsymbol{v}_i\|_1,\end{aligned}$$

where the first inequality comes from the bound $\widetilde{\text{str}}_e^i \le kW = 0.1t$ and $e^x \le 1 + 2x$ for $0 \le x \le 0.1$. Applying the inequality iteratively yields

$$\exp\left(\max_{e\in E}\frac{1}{t}\sum_{i=1}^{t}\widetilde{\text{str}}_e^i\right) = \boldsymbol{v}_{t+1,e} \le \|\boldsymbol{v}_{t+1}\|_1 \le \left(1+\frac{2W}{t}\right)^t\|\boldsymbol{v}_1\|_1 \le \exp(2W)m.$$

The desired bound (25) now follows by taking the logarithm of both sides. □

B COST AND CAPACITY SCALING FOR MIN-COST FLOWS

In this section, we describe a cost and capacity scaling scheme [6, 53, 55] that reduces the min-cost flow problem to $\widetilde{O}(\log mU \log C)$ instances with polynomially bounded cost and capacity. We prove the following lemma:

Lemma B.1. *Suppose there is an algorithm $\mathcal{A}$ that solves (8) on any m-edge graph and $\text{poly}(m)$-bounded integral demands, costs, and lower/upper capacities in $T_{\mathcal{A}}(m)$ time. There is an algorithm that on a graph $G = (V, E)$ and a min-cost flow instance $\mathcal{I} = (G, \boldsymbol{d}, \boldsymbol{c}, \boldsymbol{u}^-, \boldsymbol{u}^+)$ with integral demands $\boldsymbol{d}$, integral lower/upper capacities $\boldsymbol{u}^-, \boldsymbol{u}^+ \in \{-U, \ldots, U\}^E$, and integral costs $\boldsymbol{c} \in \{-C, \ldots, C\}^E$, solves $\mathcal{I}$ exactly in $O(T_{\mathcal{A}}(m)\log m \log mU \log C)$-time.*

Instead of (8), we consider the equivalent *min-cost circulation* problem:

$$\min_{\substack{\boldsymbol{B}^\top\boldsymbol{f}=0\\ 0\le\boldsymbol{f}_e\le\boldsymbol{u}_e \text{ for all } e\in E}} \boldsymbol{c}^\top\boldsymbol{f}, \tag{58}$$

where cost $\boldsymbol{c} \in \{-C, \ldots, C\}^E$ and capacity $\boldsymbol{u} \in \{1, \ldots, U\}^E$. It satisfies strong duality with dual problem:

$$\max_{\substack{\boldsymbol{B}\boldsymbol{y}+\boldsymbol{s}^- - \boldsymbol{s}^+ = \boldsymbol{c}\\ \boldsymbol{s}^-,\boldsymbol{s}^+\in\mathbb{R}^E_{\ge 0}}} -\boldsymbol{s}^{+\top}\boldsymbol{u}. \tag{59}$$

Given a (directed) graph $G = (V, E)$ with costs $\boldsymbol{c}$, capacities $\boldsymbol{u}$, and some feasible circulation $\boldsymbol{f}$ to (58), we can define its residual graph $G(\boldsymbol{f}) = (V, E(\boldsymbol{f}))$ with costs $\boldsymbol{c}(\boldsymbol{f})$, and capacities $\boldsymbol{u}(\boldsymbol{f})$ as follows. For any arc (directed edge)

$e = (u, v) \in E$, we include e with cost $\boldsymbol{c}_e$ and capacity $\boldsymbol{u}_e - \boldsymbol{f}_e$ if it's not saturated, i.e. $\boldsymbol{f}_e < \boldsymbol{u}_e$. We also include its reverse arc $\mathrm{rev}(e) = (v, u)$ with cost $-\boldsymbol{c}_e$ and capacity $\boldsymbol{f}_e$ if $\boldsymbol{f}_e > 0$. Given any directed graph G, we use $\boldsymbol{B}(G) \in \{-1, 0, 1\}^{E \times V}$ to denote its edge-vertex incidence matrix that respects the edge orientation.

Given some positive integers m, C, U, we define $T_{MCC}(m, C, U)$ to be the time for exactly solving (58) on a graph of at most m arcs with costs $\boldsymbol{c} \in \{-C, \ldots, C\}^E$, capacities $\boldsymbol{u} \in \{1, \ldots, U\}^E$ w.h.p.. A direct implication of Theorem 1.1 shows that

Corollary B.2. $T_{MCC}(m, \mathrm{poly}(m), \mathrm{poly}(m)) = m^{1+o(1)}$.

B.1 Reduction to Polynomially Bounded Cost Instances

In this section, we present a cost scaling scheme (Algorithm 9) for reducing to $O(\log C)$ instances with polynomially bounded cost.

Lemma B.3. *Suppose there is an algorithm $\mathcal{A}$ that gives an integral exact minimizer to (58) on any m-edge graph and m^{10}-bounded integral costs, U-bounded integral capacities in $T_{\mathcal{A}}(m, U)$ time. Algorithm 9 takes as input a graph $G = (V, E)$ and a instance of (58) $\mathcal{I} = (G, \boldsymbol{c}, \boldsymbol{u})$ with costs $\boldsymbol{c} \in \{-C, \ldots, C\}^E$ and capacities $\boldsymbol{u} \in \{1, \ldots, U\}^E$, solves $\mathcal{I}$ exactly in $O(T_{\mathcal{A}}(m, U) \log C + m \log C)$-time. In other words, $T_{MCC}(m, C, U) = O((T_{MCC}(m, m^{10}, U) + m) \log C)$.*[5]

Algorithm 9: Cost Scaling Scheme for Solving (58)

procedure CostScaling($G = (V, E), \boldsymbol{c} \in \{-C, \ldots, C\}^E, \boldsymbol{u} \in \{1, \ldots, U\}^E$)
- $\boldsymbol{f}^{(0)} \leftarrow 0$.
- $T \leftarrow O(\log C)$
- **for** $t = 0, \ldots, T - 1$ **do**
  - Let $\widetilde{G}(\boldsymbol{f}^{(t)}), \widetilde{\boldsymbol{c}}(\boldsymbol{f}^{(t)}), \boldsymbol{u}(\boldsymbol{f}^{(t)})$ be the cost rounded residual graph (Definition B.6) of $\boldsymbol{f}^{(t)}$.
  - Solve (58) on $\widetilde{G}(\boldsymbol{f}^{(t)}), \widetilde{\boldsymbol{c}}(\boldsymbol{f}^{(t)}), \boldsymbol{u}(\boldsymbol{f}^{(t)})$
  - Let $\boldsymbol{\Delta}_{\boldsymbol{f}}$ be the primal optimal.
  - Extract dual optimal $\boldsymbol{\Delta}_{\boldsymbol{y}}$ via Lemma B.9.
  - $\boldsymbol{f}^{(t+1)} \leftarrow \boldsymbol{f}^{(t)} + \boldsymbol{\Delta}_{\boldsymbol{f}}$
  - $\boldsymbol{y}^{(t+1)} \leftarrow \boldsymbol{y}^{(t)} + \boldsymbol{\Delta}_{\boldsymbol{y}}$
- Output $\boldsymbol{f}^{(T)}$

In (58), the problem is equivalent under any perturbation to the cost with $\boldsymbol{B}\boldsymbol{y}$ for any real vector $\boldsymbol{y} \in \mathbb{R}^V$. To see this, given any circulation $\boldsymbol{f}$ (not even feasible), the cost $\boldsymbol{c}^\top \boldsymbol{f}$ is equal to $(\boldsymbol{c} - \boldsymbol{B}\boldsymbol{y})^\top \boldsymbol{f}$ for any $\boldsymbol{y}$ because $\boldsymbol{B}^\top \boldsymbol{f} = 0$. Given such $\boldsymbol{y}$, we define the *reduced cost* of $\boldsymbol{c}$ w.r.t. $\boldsymbol{y}$ as $\boldsymbol{c} - \boldsymbol{B}\boldsymbol{y}$.

Here we introduce the idea of ε-*optimality* which will be used to characterize exact minimizers to integral instance of (58).

Definition B.4. *Given a parameter $\varepsilon > 0$, and a feasible circulation $\boldsymbol{f}$ to (58), we say $\boldsymbol{f}$ is* ε-optimal *if there is some vertex potential $\boldsymbol{y} \in \mathbb{R}^V$ such that $\min_e (\boldsymbol{c}(\boldsymbol{f}) - \boldsymbol{B}(\boldsymbol{f})\boldsymbol{y})_e > -\varepsilon$, where $G(\boldsymbol{f})$ is the residual graph, $\boldsymbol{c}(\boldsymbol{f})$ is the residual cost w.r.t. $\boldsymbol{f}$, and $\boldsymbol{B}(\boldsymbol{f})$ is the edge-vertex incidence matrix of $G(\boldsymbol{f})$.*

[5] In the proof, we do not make any effort on reducing the exponent of the polynomial bound on costs.

From Definition B.4, the 0 circulation is C-optimal as initial cost of any edge is at least $-C$. In the integral case, a flow $\boldsymbol{f}$ is an exact minimizer if it is $1/(n+1)$-optimal:

Lemma B.5. *In the case where the costs $\boldsymbol{c}$ in (58) is from $\{-C,\ldots,C\}^E$, a feasible integral circulation $\boldsymbol{f}$ is an exact minimizer if it is $1/(n+1)$-optimal.*

Proof. Let $\widehat{\boldsymbol{c}} = \boldsymbol{c}(\boldsymbol{f}) - \boldsymbol{B}(\boldsymbol{f})\boldsymbol{y}$ be the witness of $1/(n+1)$-optimality of $\boldsymbol{f}$. The cost of every circulation in $G(\boldsymbol{f})$ is identical between $\boldsymbol{c}(\boldsymbol{f})$ and $\widehat{\boldsymbol{c}}$. In the residual network of $\boldsymbol{f}$, every simple cycle has cost at least $-n/(n+1)$ due to $\min_e \widehat{\boldsymbol{c}}_e > -1/(n+1)$. However, since $\boldsymbol{f}$ is integral, so do its residual graph, costs, and capacities. Every negative cycle in $G(\boldsymbol{f})$ should have cost at most -1. The fact that every simple cycle in $G(\boldsymbol{f})$ has cost at least $-n/(n+1) > -1$ denies the existence of negative cycles in $G(\boldsymbol{f})$. Thus, 0 is the exact minimizer to the residual problem w.r.t. $\boldsymbol{f}$ and $\boldsymbol{f}$ is an optimal solution to (58). □

Intuitively, Algorithm 9 works by computing a augmenting flow $\boldsymbol{\Delta}$ given a ε-optimal solution $\boldsymbol{f}$ such that $\boldsymbol{f}+\boldsymbol{\Delta}$ is $(\varepsilon/2)$-optimal. Thus, the algorithm runs for $O(\log C + \log n)$ iterations until it reaches a $1/(n+1)$-optimal solution. $\boldsymbol{\Delta}$ is computed via solving (58) with costs rounded to polynomial size. That is, given an integral feasible circulation $\boldsymbol{f}$, we define a *rounded residual graph* $\widetilde{G}_{\boldsymbol{f}}$ as follows:

Definition B.6. *Given a ε-optimal integral circulation $\boldsymbol{f}$ w.r.t. a vertex potential $\boldsymbol{y} \in \mathbb{R}^V$, we define its* cost rounded residual graph $\widetilde{G}(\boldsymbol{f}) = (V, \widetilde{E}(\boldsymbol{f}))$ *with costs $\widetilde{\boldsymbol{c}}(\boldsymbol{f})$ and capacities $\boldsymbol{u}(\boldsymbol{f})$ as follows: Let $\widehat{\boldsymbol{c}}(\boldsymbol{f})$ be the reduced cost of $\boldsymbol{c}(\boldsymbol{f})$ w.r.t. $\boldsymbol{y}$, i.e. $\widehat{\boldsymbol{c}}(\boldsymbol{f}) = \boldsymbol{c}(\boldsymbol{f}) - \boldsymbol{B}(\boldsymbol{f})\boldsymbol{y}$. For any arc $e = (u,v) \in G(\boldsymbol{f})$, include e in $\widetilde{G}(\boldsymbol{f})$ with the same residual capacity $\boldsymbol{u}(\boldsymbol{f})_e$ and cost $\widetilde{\boldsymbol{c}}(\boldsymbol{f})_e$ obtained by rounding $\widehat{\boldsymbol{c}}(\boldsymbol{f})_e$ to the nearest integral multiple of ε/m^8.*

Remark B.7. *When solving (58) on $\widetilde{G}(\boldsymbol{f})$, we can always ignore edges whose cost is more than εm. Any simple cycle containing such edge has non-negative cost because costs are at least $-\varepsilon$. There will be an optimal solution that does not use any of such edge.*

Thus, every edge we care about is an integral multiple of ε/m^8 within the range $[-\varepsilon, \varepsilon m]$. Via dividing the costs by ε/m^8, all the costs are integers within $\{-m^{10},\ldots,m^{10}\}$.

Given a current ε-optimal integral circulation $\boldsymbol{f}$ w.r.t. $\boldsymbol{y}$, Algorithm 9 finds a augmenting flow $\boldsymbol{\Delta}_{\boldsymbol{f}}$ via solving (58) on the instance $\mathcal{I} = (\widetilde{G}(\boldsymbol{f}), \widetilde{\boldsymbol{c}}(\boldsymbol{f}), \boldsymbol{u}(\boldsymbol{f}))$ with polynomially bounded costs (Remark B.7). Let $\boldsymbol{\Delta}_{\boldsymbol{y}}$ be the corresponding dual (59) optimal to the instance $\mathcal{I}$. We show that $\boldsymbol{f}+\boldsymbol{\Delta}_{\boldsymbol{f}}$ is $\varepsilon/2$-optimal w.r.t. $\boldsymbol{y}+\boldsymbol{\Delta}_{\boldsymbol{y}}$. This is formulated as the following lemma:

Lemma B.8. *Given an ε-optimal integral circulation $\boldsymbol{f}$ w.r.t. $\boldsymbol{y}$, let $\boldsymbol{\Delta}_{\boldsymbol{f}}$ and $\boldsymbol{\Delta}_{\boldsymbol{y}}$ be the optimal primal dual solution to (58) on the instance $\mathcal{I} = (\widetilde{G}(\boldsymbol{f}), \widetilde{\boldsymbol{c}}(\boldsymbol{f}), \boldsymbol{u}(\boldsymbol{f}))$. $\boldsymbol{f}+\boldsymbol{\Delta}_{\boldsymbol{f}}$ is $\varepsilon/2$-optimal w.r.t. $\boldsymbol{y}+\boldsymbol{\Delta}_{\boldsymbol{y}}$.*

Proof. Clearly, $\boldsymbol{f}+\boldsymbol{\Delta}_{\boldsymbol{f}}$ is an integral feasible circulation. Let $\boldsymbol{\Delta}_{\boldsymbol{y}}, \boldsymbol{s}^-, \boldsymbol{s}^+$ be the corresponding dual solution to (59). We have that $\boldsymbol{B}(\boldsymbol{f})\boldsymbol{\Delta}_{\boldsymbol{y}} + \boldsymbol{s}^- - \boldsymbol{s}^+ = \widetilde{\boldsymbol{c}}(\boldsymbol{f})$. By complementary slackness, we know that for any arc $e = (u,v)$

(1) If $\Delta_{\boldsymbol{f},e} < \boldsymbol{u}(\boldsymbol{f})_e$, $\boldsymbol{s}^+_e = 0$, and
(2) If $\Delta_{\boldsymbol{f},e} > 0$, $\boldsymbol{s}^-_e = 0$.

For any arc $e = (u,v)$, it is included in the residual graph $G(\boldsymbol{f}+\boldsymbol{\Delta}_{\boldsymbol{f}})$ if $\Delta_{\boldsymbol{f},e} < \boldsymbol{u}(\boldsymbol{f})_e$. Therefore, we have $\boldsymbol{s}^+_e = 0$ and $(\boldsymbol{B}(\boldsymbol{f})\boldsymbol{\Delta}_{\boldsymbol{y}})_e \le \widetilde{\boldsymbol{c}}(\boldsymbol{f})_e$. The reduced residual cost on e w.r.t. $\boldsymbol{y}+\boldsymbol{\Delta}_{\boldsymbol{y}}$ is

$$\widehat{\boldsymbol{c}}(\boldsymbol{f})_e - (\boldsymbol{B}(\boldsymbol{f})\boldsymbol{\Delta}_{\boldsymbol{y}})_e \ge \widehat{\boldsymbol{c}}(\boldsymbol{f})_e - \widetilde{\boldsymbol{c}}(\boldsymbol{f})_e \ge \frac{-\varepsilon}{m^8}.$$

Its reverse, $\mathrm{rev}(e) = (v, u)$, is included in the residual graph $G(\boldsymbol{f} + \boldsymbol{\Delta}_{\boldsymbol{f}})$ if $\Delta_{f,e} > 0$. Therefore, we have $\boldsymbol{s}^- = 0$ and $(\boldsymbol{B}(\boldsymbol{f})\boldsymbol{\Delta}_{\boldsymbol{y}})_{\mathrm{rev}(e)} = -(\boldsymbol{B}(\boldsymbol{f})\boldsymbol{\Delta}_{\boldsymbol{y}})_e \le -\widetilde{\boldsymbol{c}}(\boldsymbol{f})_e = \widetilde{\boldsymbol{c}}(\boldsymbol{f})_{\mathrm{rev}(e)}$. The reduced residual cost on $\mathrm{rev}(e)$ w.r.t. $\boldsymbol{y} + \boldsymbol{\Delta}_{\boldsymbol{y}}$ is

$$\widehat{\boldsymbol{c}}(\boldsymbol{f})_{\mathrm{rev}(e)} - (\boldsymbol{B}(\boldsymbol{f})\boldsymbol{\Delta}_{\boldsymbol{y}})_{\mathrm{rev}(e)} \ge \widehat{\boldsymbol{c}}(\boldsymbol{f})_{\mathrm{rev}(e)} - \widetilde{\boldsymbol{c}}(\boldsymbol{f})_{\mathrm{rev}(e)} \ge \frac{-\varepsilon}{m^8}.$$

□

However, the algorithm implementing Theorem 1.1 only gives primal optimal solution. We need a separate routine for extracting the dual solution from the primal one.

Lemma B.9. *There is an algorithm that given an instance $\mathcal{I} = (G = (V, E), \boldsymbol{c}, \boldsymbol{u})$ of (58) where G has m edges, costs $\boldsymbol{c} \in \{-C, \ldots, C\}^E$, and capacities $\boldsymbol{u} \in \{-U, \ldots, U\}$, computes an optimal primal and dual solution $\boldsymbol{f}, \boldsymbol{y}, \boldsymbol{s}^-, \boldsymbol{s}^+$ to (58) and (59) in $O(T_{MCC}(m, C, U))$-time.*

Proof. First, we compute $\boldsymbol{f}$, the optimal primal solution to (58), in $T_{MCC}(m, C, U)$-time. Due to the optimality of $\boldsymbol{f}$, the residual graph $G(\boldsymbol{f})$ has no negative cycles. Then, we can compute a distance label on $G(\boldsymbol{f})$ as follows: Add a supervertex s to $G(\boldsymbol{f})$ with arcs toward every vertex in $G(\boldsymbol{f})$ of 0 costs. Then, we can compute a shortest path tree rooted at s by solving an un-capacitated min-cost flow with demands $\boldsymbol{d}_s = n$, $\boldsymbol{d}_u = -1, u \in V$ in $O(T_{MCC}(m, C, U))$-time using standard reduction.

Let $\boldsymbol{y}_u, u \in V$ be the distance from s to u. Since $\boldsymbol{y}$ is a valid distance label on $G(\boldsymbol{f})$, we have $\boldsymbol{B}(\boldsymbol{f})\boldsymbol{y} \le \boldsymbol{c}(\boldsymbol{f})$ where $\boldsymbol{B}(\boldsymbol{f})$ is the edge-vertex incidence matrix of the residual graph $G(\boldsymbol{f})$. Then, we will construct $\boldsymbol{s}^-, \boldsymbol{s}^+ \ge 0$ such that $(\boldsymbol{y}, \boldsymbol{s}^-, \boldsymbol{s}^+)$ is the optimal dual solution.

For any arc $e = (u, v) \in G$, if $0 < \boldsymbol{f}_e < \boldsymbol{u}_e$, we set both $\boldsymbol{s}^-_e = \boldsymbol{s}^+_e = 0$. If $\boldsymbol{f}_e = 0$, we set $\boldsymbol{s}^-_e = \boldsymbol{c}_e - (\boldsymbol{y}_v - \boldsymbol{y}_u)$ and $\boldsymbol{s}^+_e = 0$. If $\boldsymbol{f}_e = \boldsymbol{u}_e$, we set $\boldsymbol{s}^- = 0$ and $\boldsymbol{s}^+ = \boldsymbol{y}_v - \boldsymbol{y}_u - \boldsymbol{c}_e$.

Next, we check that $(\boldsymbol{y}, \boldsymbol{s}^-, \boldsymbol{s}^+)$ is a feasible dual solution. For any arc $e = (u, v) \in G$, if $0 < \boldsymbol{f}_e < \boldsymbol{u}_e$, both e and $\mathrm{rev}(e)$ appears in $G(\boldsymbol{f})$. Thus, we have both $\boldsymbol{y}_v - \boldsymbol{y}_u \le \boldsymbol{c}_e$ and $\boldsymbol{y}_u - \boldsymbol{y}_v \le -\boldsymbol{c}_e$ and therefore $\boldsymbol{y}_v - \boldsymbol{y}_u = \boldsymbol{c}_e$. Otherwise, $\boldsymbol{y}_v - \boldsymbol{y}_u + \boldsymbol{s}^-_e - \boldsymbol{s}^+_e = \boldsymbol{c}_e$ holds by the definitions of $\boldsymbol{s}^-_e$ and $\boldsymbol{s}^+_e$.

Finally, we check that $\boldsymbol{c}^\top \boldsymbol{f} = -\boldsymbol{s}^{+\top} \boldsymbol{u}$. Complementary slackness and the fact that $\boldsymbol{f}$ is a circulation yield

$$\boldsymbol{y}^\top \boldsymbol{B}^\top \boldsymbol{f} + \boldsymbol{s}^{-\top} \boldsymbol{f} + \boldsymbol{s}^{+\top} (\boldsymbol{u} - \boldsymbol{f}) = 0.$$

Rearrangement yields

$$-\boldsymbol{s}^{+\top} \boldsymbol{u} = (\boldsymbol{B}\boldsymbol{y} - \boldsymbol{s}^- + \boldsymbol{s}^+)^\top \boldsymbol{f} = \boldsymbol{c}^\top \boldsymbol{f},$$

where the last equality comes from dual feasibility of $(\boldsymbol{y}, \boldsymbol{s}^-, \boldsymbol{s}^+)$. □

Proof of Lemma B.3. Initially, we start with a C-optimal flow $\boldsymbol{f}^{(0)} = 0 \in R^E$ w.r.t. potential $0 \in \mathbb{R}^V$. At any iteration $t+1$, $\boldsymbol{f}^{(t+1)}$ is an $\varepsilon/2$-optimal flow w.r.t. $\boldsymbol{y}^{(t+1)}$ if $\boldsymbol{f}^{(t)}$ is an ε-optimal flow w.r.t. $\boldsymbol{y}^{(t)}$ (Lemma B.8). Thus, after T iterations, $\boldsymbol{f}^{(T)}$ is an $C/2^T$-optimal flow w.r.t. $\boldsymbol{y}^{(T)}$. Taking $T = O(\log C)$, we have $C/2^T < 1/(n+1)$ and hence $\boldsymbol{f}^{(T)}$ is an optimal solution to (58) due to Lemma B.5.

Every iteration, we solve for an optimal primal dual solution to (58) on a cost-rounded residual graph using Lemma B.9. Such instance has m edges, polynomially bounded costs (Remark B.7), U-bounded capacities and thus takes $O(T_{\mathcal{A}}(m, U))$-time given an algorithm $\mathcal{A}$ for solving (58). There is also an $O(m)$-overhead for constructing cost-rounded residual graph and updating $\boldsymbol{f}^{(t+1)}$ and $\boldsymbol{y}^{(t+1)}$ in each iteration. Overall, the runtime is $O(T_{\mathcal{A}}(m, U) \log C + m \log C)$ since $C = \Omega(\mathrm{poly}(m))$. □

B.2 Reduction to Polynomially Bounded Capacity Instances

It remains to address the case of min-cost flows with polynomially bounded costs, but possibly large capacity. Here we use capacity scaling.

Lemma B.10. *Suppose there is an algorithm $\mathcal{A}$ that gives an integral exact minimizer to (58) on any m-edge graph and m^{10}-bounded integral costs, m^{40}-bounded integral capacities in $T_{\mathcal{A}}(m)$ time. Algorithm 10 takes as input a graph $G = (V, E)$ and an instance of (58) $\mathcal{I} = (G, \boldsymbol{c}, \boldsymbol{u})$ with costs $\boldsymbol{c} \in \{-m^{10}, \ldots, m^{10}\}^E$ and capacities $\boldsymbol{u} \in \{1, \ldots, U\}^E$, solves $\mathcal{I}$ exactly in $O(T_{\mathcal{A}}(m) \log m \log U + m \log m \log U)$-time. In other word, $T_{MCC}(m, m^{10}, U) = O(T_{MCC}(m, m^{10}, m^{40}) \log m \log mU)$.*

In each iteration, Algorithm 10 augments the current integral circulation $\boldsymbol{f}$ with $\boldsymbol{\Delta}$, a constant approximate integral solution to (58) on the residual graph. After $O(\log(CmU))$ iterations, the optimal objective value on the residual graph is at most -0.1. This indicates that the value is 0 and we have reached an optimal solution because the residual graph is always an integral instance and has optimal value either 0 or at most -1.

To find a constant approximate solution, Algorithm 10 first finds a $\text{poly}(m)$-approximate solution of value $-x, x > 0$. Then, one can round the residual capacities down to integral multiples of $x/\text{poly}(m)$ and show that the optimal solution to the rounded residual instance is a constant approximation. Solving (58) on the rounded residual instance is equivalent to solving with polynomially bounded capacities, which can be done using Corollary B.2.

The first component is an algorithm that computes a $\text{poly}(m)$-approximate solution to (58). In particular, we find a Cm-approximate solution when the costs $\boldsymbol{c} \in \{-C, \ldots, C\}^E$. Given an instance $\mathcal{I} = (G, \boldsymbol{c}, \boldsymbol{u})$, Roughly speaking, the algorithm finds a negative weight cycle in G with largest bottleneck. This is done via performing binary search over the bottleneck, which has m different values, and then detecting negative cycle by solving an unit capacity version of (58).

Lemma B.11. *Suppose there is an algorithm $\mathcal{A}$ that gives an integral exact minimizer to (58) on any m-edge graph and m^{10}-bounded integral costs, m^{40}-bounded integral capacities in $T_{\mathcal{A}}(m)$ time. There is an algorithm that takes as input a graph $G = (V, E)$ and a instance of (58) $\mathcal{I} = (G, \boldsymbol{c}, \boldsymbol{u})$ with costs $\boldsymbol{c} \in \{-m^{10}, \ldots, m^{10}\}^E$ and capacities $\boldsymbol{u} \in \{1, \ldots, U\}^E$, outputs an m^{12}-approximate solution $\boldsymbol{f}$ such that*

$$\frac{1}{m^{12}} \boldsymbol{c}^\top \boldsymbol{f}^* \geq \boldsymbol{c}^\top \boldsymbol{f},$$

where $\boldsymbol{f}^*$ is the optimal solution to (58). The algorithm runs in $O(T_{\mathcal{A}}(m) \log m)$-time.

Proof. For any directed cycle C in G, we define its bottleneck as $\boldsymbol{u}(C) \stackrel{\text{def}}{=} \min_{e \in C} \boldsymbol{u}_e$. The algorithm finds a negative weighted cycle C^* in G with maximum bottleneck. This is done by first performing binary search over all possible bottleneck capacities, which has m of them. Let u be the bottleneck capacity we want to check. We construct graph G_u from G by removing all edges with capacities smaller than u. Via a standard reduction to unit capacity min-cost circulation on the instance $\mathcal{I}' = (G_u, \boldsymbol{c}, 1)$, we can either find a negative cost cycle in G_u or determine there's none in $O(T_{\mathcal{A}}(m))$-time. There are $O(\log m)$ stages in binary search and each stage is done in $O(T_{\mathcal{A}}(m))$-time using the given min-cost circulation algorithm $\mathcal{A}$. Overall, we can find a negative cost cycle C^* with maximum bottleneck in $O(T_{\mathcal{A}}(m) \log m)$-time.

Let $\boldsymbol{p}(C^*)$ be the flow vector corresponding to C^*. We show that $u(C^*)\boldsymbol{p}(C^*)$ is a m^{12}-approximate solution to (58) on instance $\mathcal{I} = (G, \boldsymbol{c}, \boldsymbol{u})$. Let $\boldsymbol{f}^*$ be the optimal solution to (58). Decompose $\boldsymbol{f}^*$ as a non-negative linear combination of

edge-disjoint directed cycles in G, i.e.

$$\boldsymbol{f}^* = \sum_{i=1}^{m} a_i^* \boldsymbol{p}(C_i),$$

where $\{C_1, \ldots, C_m\}$ is a edge disjoint collection of cycles in G, a_i^* is an non-negative coefficient, and $\boldsymbol{p}(C_i)$ denotes the flow vector corresponding to cycle C_i for $i = 1, \ldots, m$. Due to optimality of $\boldsymbol{f}^*$, we can assume that $a_i^* > 0$ only if C_i is a negative cost cycle.

Observe that a_i^* is at most $u(C_i)$, the bottleneck of C_i. And the cost of each cycle is at least $-m^{11}$. Thus, we can bound the cost of $\boldsymbol{f}^*$ by

$$\boldsymbol{c}^\top \boldsymbol{f}^* = \sum_{i:\boldsymbol{c}^\top \boldsymbol{p}(C_i)<0} a_i^* \boldsymbol{c}^\top \boldsymbol{p}(C_i) \geq - \sum_{i:\boldsymbol{c}^\top \boldsymbol{p}(C_i)<0} u(C_i) m^{11} \geq -m^{12} u(C^*),$$

where the last inequality comes from the definition of C^* being the maximum bottleneck of all negative cost cycles in G.

On the other hand, the weight of C^* is at most -1 because costs are integers. Thus, the cost of $u(C^*)\boldsymbol{p}(C^*)$ is at most $-u(C^*)$ and the proof concludes. □

Given any $B > 0$, we can round edge capacities down to integral multiples of B/m^{20} and remove edges of capacity over mB. The optimal circulation for the rounded instance is feasible in the original instance. Furthermore, it is a constant approximation. First, let us define the rounded instance formally.

Definition B.12. *Given a graph $G = (V, E)$ with costs $\boldsymbol{c} \in \{-m^{10}, \ldots, m^{10}\}^E$ and capacities $\boldsymbol{u} \in \{1, \ldots, U\}^E$ and a positive value $B > 0$. We define its* capacity rounded graph $G^B = (V, E^B)$ *with costs $\boldsymbol{c}$ and capacities $\boldsymbol{u}^B$ as follows: We include every arc $e \in E$ to G^B and assign its capacity $\boldsymbol{u}_e^B$ to be the nearest integral multiple of $\lceil B/m^{20} \rceil$ below $\boldsymbol{u}_e$ or $\lceil Bm^{20} \rceil$ if $\boldsymbol{u}_e > Bm^{20}$.*

Remark B.13. *By scaling down the rounded capacities by $\lceil B/m^{20} \rceil$, the capacity of every edge is a positive integer at most m^{40}. Thus, solving (58) on the capacity rounded instance $\mathcal{I}^B = (G^B, \boldsymbol{c}, \boldsymbol{u}^B)$ is equivalent to solving on an instance with m^{40}-bounded capacities. In addition, one can recover the optimal solution for the capacity rounded instance by scaling up with $\lceil B/m^{20} \rceil$, which is still integral.*

Lemma B.14. *Given a graph $G = (V, E)$ with costs $\boldsymbol{c} \in \{-m^{10}, \ldots, m^{10}\}^E$ and capacities $\boldsymbol{u} \in \{1, \ldots, U\}^E$ and a positive value $B > 0$. Suppose that $-B$ is m^{12}-approximation to the optimal value of (58) on the instance $\mathcal{I} = (G, \boldsymbol{c}, \boldsymbol{u})$. Let $\boldsymbol{f}^B$ be the optimal solution for (58) on the capacity rounded instance $\mathcal{I}^B = (G^B, \boldsymbol{c}, \boldsymbol{u}^B)$. $\boldsymbol{f}^B$ is an integral 1.1-approximate solution for the original instance $\mathcal{I} = (G, \boldsymbol{c}, \boldsymbol{u})$.*

Proof. Clearly, $\boldsymbol{f}^B$ is a feasible solution for $\mathcal{I}$ because $\boldsymbol{f}^B \leq \boldsymbol{u}^B \leq \boldsymbol{u}$ and G^B is a subgraph of G. Let $\boldsymbol{f}^*$ be the optimal solution for (58) on the instance $\mathcal{I} = (G, \boldsymbol{c}, \boldsymbol{u})$.

One can decompose $\boldsymbol{f}^*$ as a non-negative linear combination of edge-disjoint directed cycles in G, i.e.

$$\boldsymbol{f}^* = \sum_{i=1}^{m} a_i^* \boldsymbol{p}(C_i),$$

where $\{C_1, \ldots, C_m\}$ is a edge disjoint collection of cycles in G, a_i^* is an non-negative coefficient, and $\boldsymbol{p}(C_i)$ denotes the flow vector corresponding to cycle C_i for $i = 1, \ldots, m$. Due to optimality of $\boldsymbol{f}^*$, we can assume that $a_i^* > 0$ only if C_i is a negative cost cycle. Also, a_i^* is at most be bottleneck capacity of the cycle C_i, i.e. $a_i^* \leq \min_{e \in C_i} \boldsymbol{u}_e$.

First, we claim that $f^*_e \le \lceil Bm^{20} \rceil$ for any arc e. Otherwise, there is a negative cost cycle C_i in the decomposition with $a^*_i > \lceil Bm^{20} \rceil$. The cost of C_i is at least -1 due to integral costs. In this case, we use the fact that $-m^{12}B \le \boldsymbol{c}^\top \boldsymbol{f}^*$ and deduce

$$\boldsymbol{c}^\top \boldsymbol{f}^* < -\lceil Bm^{20} \rceil \le -Bm^{20} \le \frac{\boldsymbol{c}^\top \boldsymbol{f}^*}{m^{12}} m^{20} = m^8 \boldsymbol{c}^\top \boldsymbol{f}^* < 0,$$

which leads to a contradiction.

If $B < 2m^{20}$, we have $\boldsymbol{u}^B = \boldsymbol{u}$ because $\lceil B/m^{20} \rceil = 1$. In this case, $\boldsymbol{f}^B$ is exactly $\boldsymbol{f}^*$. Otherwise, round down the cycle decomposition of $\boldsymbol{f}^*$ to integral multiples of $\lceil B/m^{20} \rceil$. That is, we define

$$\widetilde{\boldsymbol{f}} = \sum_{i=1}^{m} \widetilde{a}_i \boldsymbol{p}(C_i),$$

where $\widetilde{a}_i$ is the nearest integral multiple of $\lceil B/m^{20} \rceil$ at most a^*_i. We have that $\widetilde{a}_i \le \min_{e \in C_i} \boldsymbol{u}^B_e$ and hence $\widetilde{\boldsymbol{f}}$ is a feasible solution for the capacity rounded instance. In addition, we have $\widetilde{a}_i \ge a^*_i - \lceil B/m^{20} \rceil$ for any i. Using these facts, we have

$$\begin{aligned}
\boldsymbol{c}^\top \widetilde{\boldsymbol{f}} = \sum_{i=1}^{m} \widetilde{a}_i \boldsymbol{c}^\top \boldsymbol{p}(C_i) &\overset{(i)}{\le} \sum_{i=1}^{m} a^*_i \boldsymbol{c}^\top \boldsymbol{p}(C_i) - \sum_{i=1}^{m} \lceil \frac{B}{m^{20}} \rceil \boldsymbol{c}^\top \boldsymbol{p}(C_i) \\
&\overset{(ii)}{\le} \sum_{i=1}^{m} a^*_i \boldsymbol{c}^\top \boldsymbol{p}(C_i) + \sum_{i=1}^{m} \lceil \frac{B}{m^{20}} \rceil m^{11} \\
&\overset{(iii)}{\le} \sum_{i=1}^{m} a^*_i \boldsymbol{c}^\top \boldsymbol{p}(C_i) + \sum_{i=1}^{m} 2\frac{B}{m^{20}} m^{11} \\
&= \boldsymbol{c}^\top \boldsymbol{f}^* + 2\frac{B}{m^8} \\
&\overset{(iv)}{\le} \boldsymbol{c}^\top \boldsymbol{f}^* + \frac{-2\boldsymbol{c}^\top \boldsymbol{f}^*}{m^8} \\
&= (1 - 2m^{-8}) \boldsymbol{c}^\top \boldsymbol{f}^*,
\end{aligned}$$

where (i) comes from $\widetilde{a}_i \ge a^*_i - \lceil B/m^{20} \rceil$, (ii) comes from that any simple cycle has cost at least $-m^{11}$, (iii) comes from $B \ge 2m^{20}$ and $\lceil B/m^{20} \rceil \le 2B/m^{20}$, and (iv) comes from $\boldsymbol{c}^\top \boldsymbol{f}^* \le -B$ as $-B$ is the value of a m^{12}-approximate solution. We conclude the proof by observing that

$$\boldsymbol{c}^\top \boldsymbol{f}^* \le \boldsymbol{c}^\top \boldsymbol{f}^B \le \boldsymbol{c}^\top \widetilde{\boldsymbol{f}} \le (1 - 2m^{-8}) \boldsymbol{c}^\top \boldsymbol{f}^*.$$

□

Proof of Lemma B.10. Let $\boldsymbol{f}^*$ be the optimal solution. For any t, $\boldsymbol{f}^{(t)}$ is integral since the augmenting circulation $\Delta_{\boldsymbol{f}}$ is always integral (Remark B.13). Therefore, the optimal solution $\boldsymbol{f}^* - \boldsymbol{f}^{(t)}$ to the residual instance w.r.t. $\boldsymbol{f}^{(t)}$ is integral and have cost at most -1 or 0.

Lemma B.14 states that the augmenting circulation $\Delta_{\boldsymbol{f}}$ is always a 2-approximation to the residual instance. Thus, at any iteration t, we have

$$\boldsymbol{c}^\top (\boldsymbol{f}^* - \boldsymbol{f}^{(t)}) \le \boldsymbol{c}^\top \Delta_{\boldsymbol{f}} \le \frac{1}{2} \boldsymbol{c}^\top (\boldsymbol{f}^* - \boldsymbol{f}^{(t)}) \le 0.$$

Algorithm 10: Capacity Scaling Scheme for Solving (58)

procedure CapacityScaling($G = (V, E), \boldsymbol{c} \in \{-m^{10}, \ldots, m^{10}\}^E, \boldsymbol{u} \in \{1, \ldots, U\}^E$)
  $\boldsymbol{f}^{(0)} \leftarrow 0$.
  $T \leftarrow O(\log U)$
  for $t = 0, \ldots, T-1$ **do**
    Compute $-x \leq 0$ to be the value of an m^{12}-approximate solution to (58) via Lemma B.11. // $x \geq 0$.
    if $x = 0$ **then**
      $\boldsymbol{f}^{(t)}$ is an optimal solution and we end the for loop here.
    Let $G^x(\boldsymbol{f}^{(t)}), \boldsymbol{c}(\boldsymbol{f}^{(t)}), \boldsymbol{u}^x(\boldsymbol{f}^{(t)})$ be the capacity rounded graph (Definition B.12) of the residual graph $G(\boldsymbol{f}^{(t)})$ with costs $\boldsymbol{c}(\boldsymbol{f}^{(t)})$and capacities $\boldsymbol{u}(\boldsymbol{f}^{(t)})$.
    Solve (58) on $G^x(\boldsymbol{f}^{(t)}), \boldsymbol{c}(\boldsymbol{f}^{(t)}), \boldsymbol{u}^x(\boldsymbol{f}^{(t)})$
    Let $\boldsymbol{\Delta}_f$ be the primal optimal.
    $\boldsymbol{f}^{(t+1)} \leftarrow \boldsymbol{f}^{(t)} + \boldsymbol{\Delta}_f$
  Output $\boldsymbol{f}^{(T)}$

Using the definition of $\boldsymbol{f}^{(t+1)}$ and induction yield

$$0 \geq \boldsymbol{c}^\top(\boldsymbol{f}^* - \boldsymbol{f}^{(t+1)}) = \boldsymbol{c}^\top(\boldsymbol{f}^* - \boldsymbol{f}^{(t)}) - \boldsymbol{c}^\top\boldsymbol{\Delta}_f \geq \frac{1}{2}\boldsymbol{c}^\top(\boldsymbol{f}^* - \boldsymbol{f}^{(t)}) \geq \frac{1}{2^{t+1}}\boldsymbol{c}^\top\boldsymbol{f}^*.$$

Since the costs and capacities are bounded by m^{10} and U, $\boldsymbol{c}^\top\boldsymbol{f}^*$ is at least $-m^{11}U$. After $T = O(\log m + \log U) = O(\log mU)$ iterations, we have

$$0 \geq \boldsymbol{c}^\top(\boldsymbol{f}^* - \boldsymbol{f}^{(T)}) \geq \frac{1}{2^T}\boldsymbol{c}^\top\boldsymbol{f}^* \geq \frac{-m^{11}U}{2^T} \geq \frac{-1}{2}.$$

Combining with the previous observation that $\boldsymbol{c}^\top(\boldsymbol{f}^* - \boldsymbol{f}^{(T)})$ is either 0 or at most -1, we have that $\boldsymbol{c}^\top(\boldsymbol{f}^* - \boldsymbol{f}^{(T)}) = 0$ and hence $\boldsymbol{f}^{(T)}$ is an optimal solution.

Each iteration spends $O(T_{\mathcal{A}}(m)\log m)$-time for computing an m^{12}-approximate solution, plus $O(T_{\mathcal{A}}(m))$-time for computing $\boldsymbol{\Delta}_f$ on a capacity rounded instance, and $O(m)$ for constructing instances and computing $\boldsymbol{f}^{(t+1)}$. Overall, the runtime is $O(T_{\mathcal{A}}(m)\log m \log mU + m \log m \log mU)$. □

Now, we can prove Lemma B.1 by combining Lemma B.3 and Lemma B.10.

Proof of Lemma B.1. As any instance of (8) can be reduced to (58) with linear overhead in the number of edges and poly(m) scaling on costs and capacities, the lemma follows directly from Lemma B.3 and Lemma B.10. □

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009