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

Li Chen
School of Computer Science
Georgia Tech
Atlanta, USA
lichen@gatech.edu

Rasmus Kyng
Department of Computer Science
ETH Zurich
Zurich, Switzerland
kyng@inf.ethz.ch

Yang P. Liu
Department of Mathematics
Stanford University
Palo Alto, USA
yangpliu@stanford.edu

Richard Peng
School of Computer Science
University of Waterloo
Waterloo, Canada
y5peng@uwaterloo.ca

Maximilian Probst Gutenberg
Department of Computer Science
ETH Zurich
Zurich, Switzerland
maxprobst@ethz.ch

Sushant Sachdeva
Department of Computer Science
University of Toronto
Toronto, Canada
sachdeva@cs.toronto.edu

***Abstract*—We give 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.

***Index Terms*—Maximum flow, Minimum cost flow, Data structures, Interior point methods, Convex optimization**

See https://arxiv.org/abs/2203.00671 for the full version of this paper.

I. 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 $\boldsymbol{u}_e^- \leq \boldsymbol{f}_e \leq \boldsymbol{u}_e^+$ and satisfies vertex demands $\boldsymbol{d}$. The vertex demand constraints are succinctly captured as $\mathbf{B}^\top \boldsymbol{f} = \boldsymbol{d}$, where $\mathbf{B} \in \mathbb{R}^{E \times V}$ is the edge-vertex incidence matrix defined as $\mathbf{B}_{((a,b),v)}$ 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 $|\boldsymbol{u}_e^+|, |\boldsymbol{u}_e^-| \leq U$ 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 the full version, a brief discussion will help place our work in context. Starting from the first pseudo-polynomial time algorithm by Dantzig [20] 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 $\widetilde{O}(m \min\{m^{1/2}, n^{2/3}\} \log U)$ [29, 31, 23, 27] for max-flow, and $\widetilde{O}(mn \log U)$ [26] for min-cost flow. These bounds stood for decades.

In their breakthrough work on solving Laplacian systems and computing electrical flows, Spielman and Teng [45] 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 [19] demonstrated the power of this paradigm by combining Interior Point methods (IPMs) with fast Laplacian systems solvers to achieve an

Li Chen was supported by NSF Grant CCF-2106444.

The research leading to these results has 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 the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grant RGPIN-2022-03207. Part of this work was done while at the Georgia Institute of Technology.

Sushant Sachdeva's research is supported by an Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grant RGPIN-2018-06398 and an Ontario Early Researcher Award (ERA) ER21-16-284.

978-1-6654-5519-0/22/$31.00 ©2022 IEEE
DOI 10.1109/FOCS54457.2022.00064

$\widetilde{O}(m^{1.5}\log^2 U)$ 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 [16, 33, 43, 41, 42], and an $m^{4/3+o(1)}U^{1/3}$ time algorithm for bipartite matching and unit capacity max-flow [40, 38, 37, 32, 5], and $pm^{1+o(1)}$ time unweighted p-norm minimizing flow for large p [34, 2].

Efficient graph data-structures have played a key role in the development of faster algorithms for flow problems, e.g. dynamic trees [44]. 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 [12, 13, 11], an $\widetilde{O}(n\log U)$ time algorithm for planar graphs [22, 21], 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 [8, 25, 6, 10]. 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 I.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] [2]

Corollary I.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.*

A. Applications

Our result in Theorem I.1 has a wide range of applications. By standard reductions, it gives the first $m^{1+o(1)}$ time algorithm for the bipartite matching problem and $m^{1+o(1)}\log U\log C$ time algorithms for its generalizations, including the worker assignment and optimal transport problems.

In directed graphs with possibly negative edge weights, assuming integral weights bounded by W in absolute value, we obtain the first almost-linear time algorithm to compute single-source shortest paths and to detect a negative cycle, running in $m^{1+o(1)}\log W$ time. In an independent work, Bernstein, Nanongkai, and Wulff-Nilsen [9] claim the first $m\cdot \mathrm{poly}(\log m)\log W$ time algorithm for this problem.

Using recent reductions from various connectivity problems to max-flow, we also obtain the first $m^{1+o(1)}$ time algorithms for various such problems, most prominently to compute vertex connectivity and Gomory-Hu trees in undirected, unweighted graphs, and $(1+\varepsilon)$-approximate Gomory-Hu trees in undirected weighted graphs. We also obtain the fastest current algorithm to find the global min-cut in a directed graph. Finally, we obtain the first almost linear time algorithms to compute approximate sparsest cuts in directed graphs. We defer the discussion of these results and precise statements to the full version.

Additionally, we extend our algorithm to compute flows that minimize general edge-separable convex objectives. This allows us to solve regularized versions of optimal transport (equivalently, matrix scaling), as well as p-norm flow problems and p-norm isotonic regression for all $p \in [1, \infty]$. We state an informal version of our main result on general convex flows.

Informal Theorem I.3. *Consider a graph G with demands $\boldsymbol{d}$, and an edge-separable convex cost function $\mathsf{cost}(f) = \sum_e \mathsf{cost}_e(f_e)$ for "computationally efficient" edge costs cost_e. Then in $m^{1+o(1)}$ time, we can compute a (fractional) flow $\boldsymbol{f}$ that routes demands $\boldsymbol{d}$ and $\mathsf{cost}(\boldsymbol{f}) \le \mathsf{cost}(\boldsymbol{f}^*) + \exp(-\log^C m)$ for any constant $C > 0$, where $\boldsymbol{f}^*$ minimizes $\mathsf{cost}(\boldsymbol{f}^*)$ over flows with demands $\boldsymbol{d}$.*

We remark that the optimal solution $\boldsymbol{f}^*$ to the above convex flow problem can be non-integral, whereas in the case of max-flow and min-cost flow with integral demands/capacities, there exists an integral optimal flow.

II. Overview

Due to space constraints, we only give a technical overview of the key pieces developed in this paper and refer to the full version of the paper on arXiv 2203.00671. Section II-A 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. Below is an informal statement summarizing the IPM guarantees proven in the full version.

Informal Theorem II.1 (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)*

[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.

[2] By classic capacity scaling techniques [24, 28, 3], it suffices to work with graphs with $U, C = \mathrm{poly}(m)$ to show Theorem I.1 and Corollary I.2.

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.

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 $m^{1+o(1)}$.

Informal Theorem II.2 (Hidden Stable-Flow Chasing. Theorem 6.2)**.** *We design a randomized data structure for approximately solving a sequence of "stable" (as defined in Informal Theorem II.1) 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.*

Section II-B 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 II-C we describe how to reduce the number of vertices, before describing the overall recursive data structure in Section II-D. 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 II-E 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 II-F 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 II-G which gives an outline of our dynamic spanner data structure.

Informal Theorem II.3 (Dynamic Spanner w/ Embeddings. Theorem 5.1)**.** *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.*

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.

A. 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:

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

for gradient $\boldsymbol{g} \in \mathbb{R}^E$ and lengths $\mathbf{L} = \operatorname{diag}(\boldsymbol{\ell})$ 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:

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

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 [30], 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 [19]. Thus if we can reduce the potential by $m^{-o(1)}$ per iteration, the method terminates in $m^{1+o(1)}$ iterations.

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 [18] or the tutorial [35]), 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 $(\boldsymbol{u}^+_e - \boldsymbol{f}_e)^{-\alpha} \le \widetilde{O}(m)$, then $\boldsymbol{u}^+_e - \boldsymbol{f}_e \ge \widetilde{O}(m)^{-1/\alpha} \ge$

$\exp(-O(\log^2 Um))$. 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 $\boldsymbol{\ell}(\boldsymbol{f})_e \stackrel{\text{def}}{=} (\boldsymbol{u}_e^+ - \boldsymbol{f}_e)^{-1-\alpha} + (\boldsymbol{f}_e - \boldsymbol{u}_e^-)^{-1-\alpha}$. Now, let $\boldsymbol{\Delta}$ be a circulation with $\boldsymbol{g}(\boldsymbol{f})^\top\boldsymbol{\Delta}/\left\|\mathbf{L}\boldsymbol{\Delta}\right\|_1 \le -\kappa$ for some $\kappa < 1/100$, scaled so that $\left\|\mathbf{L}\boldsymbol{\Delta}\right\|_1 = \kappa/50$. A direct Taylor expansion shows that $\Phi(\boldsymbol{f}+\boldsymbol{\Delta}) \le \Phi(\boldsymbol{f}) - \kappa^2/500$ (Lemma 4.4).

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 $\boldsymbol{g}(\boldsymbol{f})^\top\boldsymbol{\Delta}/\left\|\mathbf{L}\boldsymbol{\Delta}\right\|_1 \le -\widetilde{\Omega}(1)$ (Lemma 4.7).

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 II-B and II-E.

B. 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 $\min_{\mathbf{B}^\top\boldsymbol{\Delta}=0} \boldsymbol{g}^\top\boldsymbol{\Delta}/\|\mathbf{L}\boldsymbol{\Delta}\|_1$ to $m^{o(1)}$ multiplicative accuracy. Because our IPM ensures stability for lengths and gradients (see Lemma 4.9 and 4.10), 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 $m^{1+o(1)}$ total times. Efficiency gains based on leveraging stability has appeared in the earliest works on efficiently maintaining IPM iterates [30, 46] as well as most recent progress on speeding up linear programs.

a) 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 $\mathsf{str}_e^{T,\boldsymbol{\ell}} \stackrel{\text{def}}{=} \frac{\sum_{f\in T[u,v]}\boldsymbol{\ell}(f)}{\boldsymbol{\ell}(e)}$ where $T[u,v]$ is the unique path from u to v along the tree

[3]The reason that path-following IPMs for max-flow [19] 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.

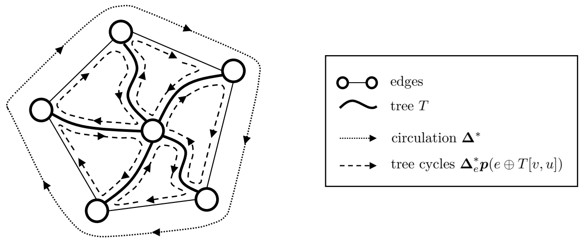

edges
tree T
circulation $\boldsymbol{\Delta}^*$
tree cycles $\boldsymbol{\Delta}_e^*\boldsymbol{p}(e\oplus T[v,u])$

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

T, is $\widetilde{O}(1)$ in expectation. Such a tree can be found in $\widetilde{O}(m)$ time [4, 1].

Let $\boldsymbol{\Delta}^*$ be the witness circulation that minimizes (1), and assume wlog that $\boldsymbol{\Delta}^*$ is a cycle that routes one unit of flow along the cycle. We assume for convenience, that edges on $\boldsymbol{\Delta}^*$ are oriented along the flow direction of $\boldsymbol{\Delta}^*$, i.e. that $\boldsymbol{\Delta}^* \in \mathbb{R}^E_{\ge 0}$. Then, for each edge $e = (u,v)$ on the cycle $\boldsymbol{\Delta}^*$, 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 $\boldsymbol{\Delta}^* = \sum_{e:\boldsymbol{\Delta}_e^*>0}\boldsymbol{\Delta}_e^*\cdot\boldsymbol{p}(e\oplus T[v,u])$. (note that the tree-paths used by adjacent off-tree edges cancel out , see Figure 1). In particular, this implies that $\boldsymbol{g}^\top\boldsymbol{\Delta}^* = \sum_{e:\boldsymbol{\Delta}_e^*>0}\boldsymbol{\Delta}_e^*\cdot\boldsymbol{g}^\top\boldsymbol{p}(e\oplus T[v,u])$.

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 $\mathsf{str}_e^{T,\boldsymbol{\ell}}$ 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 $\boldsymbol{\Delta}^*$ is not stretched by too much. Formally, we have that $\sum_{e:\boldsymbol{\Delta}_e^*>0}\boldsymbol{\Delta}_e^*\cdot\|\mathbf{L}\,\boldsymbol{p}(e\oplus T[v,u])\|_1 \le \gamma\|\mathbf{L}\boldsymbol{\Delta}^*\|_1$ for $\gamma = \widetilde{O}(1)$. Combining our insights, we can thus derive that

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

where the last inequality follows from the fact that $\min_{i\in[n]}\frac{\boldsymbol{x}_i}{\boldsymbol{y}_i} \le \frac{\sum_{i\in[n]}\boldsymbol{x}_i}{\sum_{i\in[n]}\boldsymbol{y}_i}$ (recall also that $\boldsymbol{g}^\top\boldsymbol{\Delta}^*$ 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 $\boldsymbol{\Delta}^*$ 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

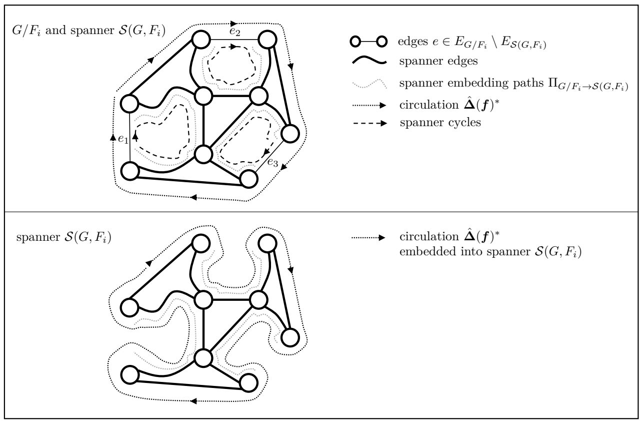
$T_1, T_2, \ldots, T_s$ because the next solution to (1) has to be found with respect to gradients $\boldsymbol{g}(\boldsymbol{f}')$ and lengths $\boldsymbol{\ell}(\boldsymbol{f}')$ depending on $\boldsymbol{f}'$ (instead of $\boldsymbol{g} = \boldsymbol{g}(\boldsymbol{f})$ and $\boldsymbol{\ell} = \boldsymbol{\ell}(\boldsymbol{f})$). 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.

b) 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 $\boldsymbol{\Delta}$ 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 $\widehat{\boldsymbol{\Delta}}$ in the core graph G/F_i, to circulations $\boldsymbol{\Delta}$ 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 $\widehat{\mathbf{L}} = \operatorname{diag}(\widehat{\boldsymbol{\ell}})$) and are chosen to upper bound the length of circulations when mapped back into G such that $\|\widehat{\mathbf{L}}\widehat{\boldsymbol{\Delta}}\|_1 \geq \|\mathbf{L}\boldsymbol{\Delta}\|_1$. Crucially, we must ensure these new lengths $\widehat{\boldsymbol{\ell}}$ do not stretch the witness circulation $\boldsymbol{\Delta}^*$ 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 II-C, 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 $\widehat{\boldsymbol{\Delta}}(\boldsymbol{f})^*$ 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 $\widehat{\boldsymbol{\Delta}}(\boldsymbol{f})^*$, *or* re-routing $\widehat{\boldsymbol{\Delta}}(\boldsymbol{f})^*$ into $\mathcal{S}(G, F_i)$ roughly preserves its quality. Figure 2 illustrates this dichotomy. Thus, 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 $\widehat{\boldsymbol{\Delta}}(\boldsymbol{f})^*$ 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 II-G.



G/F_i and spanner $\mathcal{S}(G, F_i)$
e_2
e_1
e_3
edges $e \in E_{G/F_i} \setminus E_{\mathcal{S}(G,F_i)}$
spanner edges
spanner embedding paths $\Pi_{G/F_i \to \mathcal{S}(G,F_i)}$
circulation $\hat{\boldsymbol{\Delta}}(\boldsymbol{f})^*$
spanner cycles
spanner $\mathcal{S}(G, F_i)$
circulation $\hat{\boldsymbol{\Delta}}(\boldsymbol{f})^*$
embedded into spanner $\mathcal{S}(G, F_i)$

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 $\widehat{\boldsymbol{\Delta}}(\boldsymbol{f})^*$, *or* re-routing $\widehat{\boldsymbol{\Delta}}(\boldsymbol{f})^*$ into $\mathcal{S}(G, F_i)$ roughly preserves its quality.

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 II-D, 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 $\boldsymbol{\Delta}(\boldsymbol{f})^* = \boldsymbol{f}^* - \boldsymbol{f}$ and depends on the updates we make to $\boldsymbol{f}$, 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 $\boldsymbol{f}^* - \boldsymbol{f}$ lets us express the IPM with an adversary that is much more restricted. Our data structure needs to ensure that the flow $\boldsymbol{f}^* - \boldsymbol{f}$ is stretched by $m^{o(1)}$ on average w.r.t. the lengths $\boldsymbol{\ell}$. At a high level, we achieve this by forcing the forests at every level to have stretch 1 on edges where $\boldsymbol{f}_e$ changes significantly and could affect the total stretch of our data structure on $\boldsymbol{f}^* - \boldsymbol{f}$. Section II-E 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 $\|\mathbf{L}\boldsymbol{\Delta}(\boldsymbol{f})^*\|_1$ 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 II-F, 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.

C. Building Core Graphs

In this section, we describe our core graph construction (Definition 6.7), 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 6.5). 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 $F_1, \dots, F_B$ 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 [45] and has been further developed in many works since.

a) 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 $P = e_1, \dots, e_k$ 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:

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

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 $\boldsymbol{\Pi}_F : \mathbb{R}^{E(G)} \to \mathbb{R}^{E(G/F)}$ and $\boldsymbol{\Pi}_F^{-1} : \mathbb{R}^{E(G/F)} \to \mathbb{R}^{E(G)}$. Since we let G/F have self-loops, there is a bijection between edges of G and G/F and thus $\boldsymbol{\Pi}_F$ 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 3.

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 $F[\mathsf{root}^F_{u^G}, u^G] \oplus e^G \oplus F[v^G, \mathsf{root}^F_{v^G}]$. Meanwhile, we define $\widetilde{\mathsf{str}}_e \stackrel{\text{def}}{=} \widehat{\boldsymbol{\ell}}_e^F / \boldsymbol{\ell}_e$, 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. $\left\|\mathbf{L}\boldsymbol{\Pi}_F^{-1}\boldsymbol{f}\right\|_1 \leq \left\|\widehat{\mathbf{L}}^F\boldsymbol{f}\right\|_1$. Let $\boldsymbol{\Delta}^*$ be the optimal solution to (1). We will show later how to build F such that $\left\|\widehat{\mathbf{L}}^F\boldsymbol{\Delta}^*\right\|_1 \leq \gamma \left\|\mathbf{L}\boldsymbol{\Delta}^*\right\|_1$ holds for

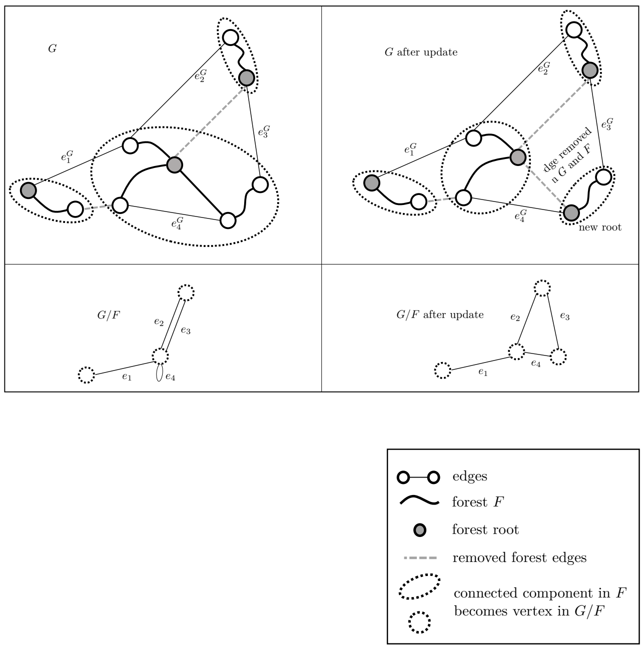


Fig. 3: Illustration of the core graph G/F changing as an edge is deleted in G (and in F).

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 $\boldsymbol{\Delta}$ on G/F and its preimage $\boldsymbol{\Pi}_F^{-1}\boldsymbol{\Delta}$ in G is the same, i.e. $\widehat{\boldsymbol{g}}^\top\boldsymbol{\Delta} = \boldsymbol{g}^\top\boldsymbol{\Pi}_F^{-1}\boldsymbol{\Delta}$. The idea of incorporating gradients into portal routing was introduced in [34]; our version of this construction is somewhat different to allow us to make it dynamic efficiently.

b) 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 [39, 43, 33, 15] and dynamic algorithms [14]. 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) $\widetilde{\mathsf{str}}_e^F = \widehat{\boldsymbol{\ell}}_e^F/\boldsymbol{\ell}_{e^G} = \widetilde{O}(k)$ for any edge $e^G \in G$ with image e in G/F, and
2) The weighted average of $\widetilde{\mathsf{str}}_e^F$ w.r.t. $\boldsymbol{v}$ is only $\widetilde{O}(1)$, i.e. $\sum_{e^G\in G}\boldsymbol{v}_{e^G}\cdot\widetilde{\mathsf{str}}_e^F \leq \widetilde{O}(1)\cdot\|\boldsymbol{v}\|_1$.

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 [39, 43, 33] we compute k different edge weights $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_k$ via multiplicative weight updates (Lemma 6.6) so that the corresponding LSDs $F_1, \ldots, F_k$ have $\widetilde{O}(1)$ average stretch on every edge in G: $\sum_{j=1}^k \widetilde{\mathsf{str}}_e^{F_j} = \widetilde{O}(k)$, for all $e^G \in G$ with image e in G/F.

By Markov's inequality, for any fixed flow $\boldsymbol{f}$ in G, $\left\|\widehat{\mathbf{L}}^{F_j}\boldsymbol{f}\right\|_1 \leq \widetilde{O}(1)\|\mathbf{L}\boldsymbol{f}\|_1$ 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 $\min_{j\in[B]}\left\|\widetilde{\mathsf{str}}^{F_j} \circ \mathbf{L}\boldsymbol{\Delta}^*\right\|_1 \leq \widetilde{O}(1)\|\mathbf{L}\boldsymbol{\Delta}^*\|_1$. 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.

D. 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 the full version. 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 6.5). 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 II-G.

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.

E. 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 4.7 states that for any flow $\boldsymbol{f}$, $\boldsymbol{g}(\boldsymbol{f})^\top \boldsymbol{\Delta}(\boldsymbol{f})/(100m + \|\mathbf{L}(\boldsymbol{f})\boldsymbol{\Delta}(\boldsymbol{f})\|_1) \leq -\widetilde{\Omega}(1)$ holds where $\boldsymbol{\Delta}(\boldsymbol{f}) = \boldsymbol{f}^* - \boldsymbol{f}$. 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 $\boldsymbol{\Delta}(\boldsymbol{f}) = \boldsymbol{f}^* - \boldsymbol{f}$ as long as

$$\min_{j \in [B]} \left\| \widehat{\mathbf{L}}^{F_j} \boldsymbol{\Delta}(\boldsymbol{f}) \right\|_1 \leq \widetilde{O}(1) \left\| \mathbf{L}(\boldsymbol{f}) \boldsymbol{\Delta}(\boldsymbol{f}) \right\|_1 + \widetilde{O}(m). \quad (4)$$

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

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

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 $\boldsymbol{\Delta}(\boldsymbol{f})$ as $\boldsymbol{w}(\boldsymbol{f}) = 100 \cdot \mathbf{1} + |\mathbf{L}(\boldsymbol{f})\boldsymbol{\Delta}(\boldsymbol{f})|$. The name comes from the fact that $\boldsymbol{w}(\boldsymbol{f})_e$ is always at least $|\boldsymbol{\ell}(\boldsymbol{f})_e(\boldsymbol{f}^*_e - \boldsymbol{f}_e)|$ for any edge e. We show that the width is also slowly changing (Lemma 9.2) 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 $\left\| \widehat{\mathbf{L}}^{F_j} \boldsymbol{\Delta}(\boldsymbol{f}) \right\|_1 \leq \left\| \widetilde{\mathsf{str}}^{F_j} \circ \boldsymbol{w}(\boldsymbol{f}) \right\|_1$. Thus, we can strengthen (4) and show that the IPM potential can be decreased by $m^{-o(1)}$ if $\min_{j \in [B]} \left\| \widetilde{\mathsf{str}}^{F_j} \circ \boldsymbol{w}(\boldsymbol{f}) \right\|_1 \leq \widetilde{O}(1) \left\| \boldsymbol{w}(\boldsymbol{f}) \right\|_1$. It 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 $\boldsymbol{w}(\boldsymbol{f} + \boldsymbol{\Delta})_e \approx \boldsymbol{w}(\boldsymbol{f})_e$ 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. $\widetilde{\mathsf{str}}_e^{F_j} = 1$ 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: $\left\| \widetilde{\mathsf{str}}^{F_j} \circ \boldsymbol{w}(\boldsymbol{f} + \boldsymbol{\Delta}) \right\|_1 \lesssim \left\| \widetilde{\mathsf{str}}^{F_j} \circ \boldsymbol{w}(\boldsymbol{f}) \right\|_1 + \left\| \boldsymbol{w}(\boldsymbol{f} + \boldsymbol{\Delta}) \right\|_1$. Using the fact that $\min_{j \in [B]} \left\| \widetilde{\mathsf{str}}^{F_j} \circ \boldsymbol{w}(\boldsymbol{f}) \right\|_1 \leq \widetilde{O}(1) \left\| \boldsymbol{w}(\boldsymbol{f}) \right\|_1$, we have the following: $\min_{j \in [B]} \left\| \widetilde{\mathsf{str}}^{F_j} \circ \boldsymbol{w}(\boldsymbol{f} + \boldsymbol{\Delta}) \right\|_1 \lesssim \widetilde{O}(1) \left\| \boldsymbol{w}(\boldsymbol{f}) \right\|_1 + \left\| \boldsymbol{w}(\boldsymbol{f} + \boldsymbol{\Delta}) \right\|_1$.

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. $\left\| \boldsymbol{w}(\boldsymbol{f} + \boldsymbol{\Delta}) \right\|_1 \leq \widetilde{O}(1) \left\| \boldsymbol{w}(\boldsymbol{f}) \right\|_1$.

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 $\left\| \boldsymbol{w}(\boldsymbol{f} + \boldsymbol{\Delta}) \right\|_1 \geq 100 \left\| \boldsymbol{w}(\boldsymbol{f}) \right\|_1$ 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 II-A), 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 $\mathsf{prev}_i^{(t)}$ 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 $\mathsf{prev}_d^{(t)} = t$. We show in Lemma 7.9 that the cycle output by the data structure in the t-th IPM iteration has length at most $m^{o(1)} \sum_{i=0}^{d} \|\boldsymbol{w}^{(\mathsf{prev}_i^{(t)})}\|_1$. This inequality is a natural generalization of the $\widetilde{O}(1)\left(\|\boldsymbol{w}(\boldsymbol{f})\|_1 + \|\boldsymbol{w}(\boldsymbol{f} + \boldsymbol{\Delta})\|_1\right)$-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 $\sum_{i=0}^{d} \|\boldsymbol{w}^{(\mathsf{prev}_i^{(t)})}\|_1$ is much more than $m^{o(1)}\|\boldsymbol{w}^{(t)}\|_1$. In this way, the data structure can fail because some lower level i has $\|\boldsymbol{w}^{(\mathsf{prev}_i^{(t)})}\|_1 \gg \|\boldsymbol{w}^{(t)}\|_1$. A possible fix is to rebuild the entire data structure which sets $\mathsf{prev}_i^{(t)} = t$ 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.

F. The Rebuilding Game

Our goal in Section 8 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 II-E 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 $\mathsf{prev}_i^{(t)}$ denote the stage when level i was last rebuilt, and refer to $\|\boldsymbol{w}^{(\mathsf{prev}_i^{(t)})}\|_1$ 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 $\sum_{i=0}^{d} \|\boldsymbol{w}^{(\mathsf{prev}_i^{(t)})}\|_1 \gg m^{o(1)}\|\boldsymbol{w}^{(t)}\|_1$. 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 $\mathsf{prev}_i^{(t)} = t$ when necessary. Now, because of well-behaved numerical properties of our IPM, we are guaranteed that $\log(\|\boldsymbol{w}^{(t)}\|_1) \in [-\operatorname{poly}\log(m), \operatorname{poly}\log(m)]$, 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 8.1 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 $\sum_{i=0}^{d} \|\boldsymbol{w}^{(\mathsf{prev}_i^{(t)})}\|_1 \gg m^{o(1)}\|\boldsymbol{w}^{(t)}\|_1$ is larger than $2(d+1)$, we can deduce that if a failure occurs, then $\max_{i=0}^{d} \|\boldsymbol{w}^{(\mathsf{prev}_i^{(t)})}\|_1 > 2\|\boldsymbol{w}^{(t)}\|_1$. Thus, *if* the total width at level i is too large, then a losing rebuild at level i (and hence updating $\boldsymbol{w}^{(\mathsf{prev}_i^{(t+1)})}$ 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 $\operatorname{poly}\log(m)$ 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 $\operatorname{poly}\log(m)$ 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 8.3.

Finally, at the end of Section 8, 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 6.2.

G. Dynamic Embeddings into Spanners of Decremental Graphs

It remains to describe the algorithm to maintain a spanner $\mathcal{S}(G, F_i)$ on the graphs G/F_i. Let us recall the requirements on the spanner given in Section 2.4:

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 B-Branching Tree Chain (see Section II-D).
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 [7] 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 [7] 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) 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 $|E_{H_i}(X, \overline{X})| < \Omega\left(\frac{1}{\log^3(m)}\right) \min\{\mathrm{vol}_{H_i}(X), \mathrm{vol}_{H_i}(\overline{X})\}$ where $E_{H_i}(X, \overline{X})$ is the set of edges in H_i with endpoints in X and $\overline{S}$ and $\mathrm{vol}_{H_i}(Y)$ 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 [45, 7]). 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 [17] 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.

b) 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

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

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

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

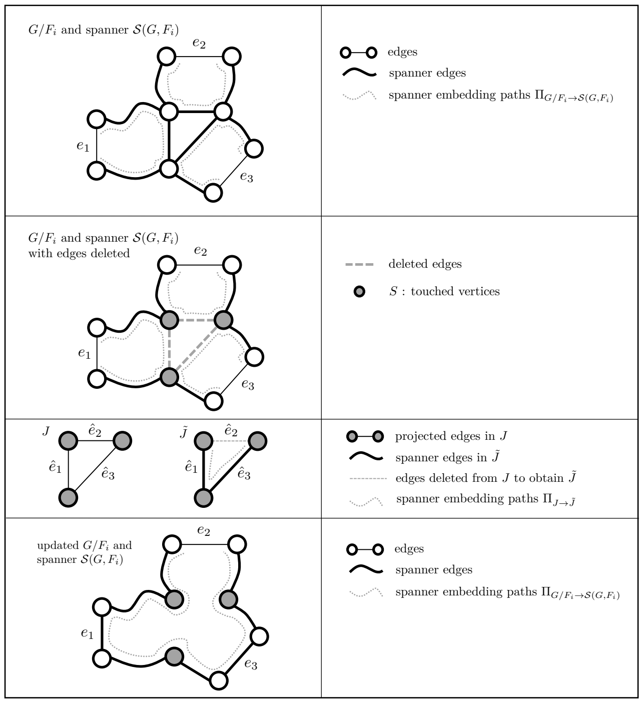
By the guarantees from the previous paragraph, we have that the sparsifier $\widetilde{J}$ has average degree $\widetilde{O}(1)$, and we only added the pre-images of edges in $\widetilde{J}$ to $\mathcal{S}(G, F_i)$. Since J (and $\widetilde{J}$) are taken over the vertex set S, we can conclude that 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 4 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.

c) 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 $km^{o(1)}$ 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


G/F_i and spanner $\mathcal{S}(G, F_i)$
e_2
e_1
e_3
edges
spanner edges
spanner embedding paths $\Pi_{G/F_i \to \mathcal{S}(G,F_i)}$
G/F_i and spanner $\mathcal{S}(G, F_i)$ with edges deleted
e_2
e_1
e_3
deleted edges
S : touched vertices
J
$\hat{e}_2$
$\hat{e}_1$
$\hat{e}_3$
$\tilde{J}$
$\hat{e}_2$
$\hat{e}_1$
$\hat{e}_3$
projected edges in J
spanner edges in $\tilde{J}$
edges deleted from J to obtain $\tilde{J}$
spanner embedding paths $\Pi_{J \to \tilde{J}}$
updated G/F_i and spanner $\mathcal{S}(G, F_i)$
e_2
e_1
e_3
edges
spanner edges
spanner embedding paths $\Pi_{G/F_i \to \mathcal{S}(G,F_i)}$
Fig. 4: Illustration of the procedure for maintaining $\mathcal{S}(G, F_i)$ under edge deletions.

in a rather crude way. But it is not hard to see via the multi-commodity max-flow min-cut theorem [36] 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 [17], 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 5 for the full implementation and analysis.

Acknowledgment

We 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] I. Abraham and O. Neiman. "Using petal-decompositions to build a low stretch spanning tree". In: *SIAM Journal on Computing* 48.2 (2019), pp. 227–248.
[2] D. Adil and S. Sachdeva. "Faster p-norm minimizing flows, via smoothed q-norm problems". In: *SODA*. 2020.

[3] R. K. Ahuja, A. V. Goldberg, J. B. Orlin, and R. E. Tarjan. “Finding minimum-cost flows by double scaling”. In: *Math. Program.* 53 (1992).

[4] N. Alon, R. M. Karp, D. Peleg, and D. West. “A graph-theoretic game and its application to the k-server problem”. In: *SIAM Journal on Computing* (1995).

[5] K. Axiotis, A. Mądry, and A. Vladu. “Circulation control for faster minimum cost flow in unit-capacity graphs”. In: *FOCS*. 2020.

[6] K. Axiotis, A. Mądry, and A. Vladu. “Faster Sparse Minimum Cost Flow by Electrical Flow Localization”. In: *CoRR* abs/2111.10368 (2021).

[7] A. Bernstein, J. v. d. Brand, M. P. Gutenberg, D. Nanongkai, T. Saranurak, A. Sidford, and H. Sun. “Fully-dynamic graph sparsifiers against an adaptive adversary”. In: *arXiv preprint arXiv:2004.08432* (2020).

[8] A. Bernstein, M. P. Gutenberg, and T. Saranurak. “Deterministic decremental sssp and approximate min-cost flow in almost-linear time”. In: *arXiv preprint arXiv:2101.07149* (2021).

[9] A. Bernstein, D. Nanongkai, and C. Wulff-Nilsen. “Negative-Weight Single-Source Shortest Paths in Near-linear Time”. In: *CoRR* abs/2203.03456 (2022). arXiv: 2203.03456.

[10] J. v. d. Brand, Y. Gao, A. Jambulapati, Y. T. Lee, Y. P. Liu, R. Peng, and A. Sidford. “Faster Maxflow via Improved Dynamic Spectral Vertex Sparsifiers”. In: *CoRR* abs/2112.00722 (2021). arXiv: 2112.00722.

[11] J. v. d. Brand, Y. T. Lee, Y. P. Liu, T. Saranurak, A. Sidford, Z. Song, and D. Wang. “Minimum cost flows, MDPs, and ℓ_1-regression in nearly linear time for dense instances”. In: *STOC*. 2021.

[12] J. v. d. Brand, Y. T. Lee, A. Sidford, and Z. Song. “Solving tall dense linear programs in nearly linear time”. In: *STOC*. 2020.

[13] J. v. d. Brand, Y.-T. Lee, D. Nanongkai, R. Peng, T. Saranurak, A. Sidford, Z. Song, and D. Wang. “Bipartite matching in nearly-linear time on moderately dense graphs”. In: *FOCS*. 2020.

[14] L. Chen, G. Goranci, M. Henzinger, R. Peng, and T. Saranurak. “Fast dynamic cuts, distances and effective resistances via vertex sparsifiers”. In: *FOCS*. 2020.

[15] L. Chen, R. Peng, and D. Wang. “2-norm Flow Diffusion in Near-Linear Time”. In: *FOCS*. 2021.

[16] P. Christiano, J. A. Kelner, A. Mądry, D. A. Spielman, and S. Teng. “Electrical flows, Laplacian systems, and faster approximation of maximum flow in undirected graphs”. In: *STOC*. 2011.

[17] J. Chuzhoy and T. Saranurak. “Deterministic algorithms for decremental shortest paths via layered core decomposition”. In: *SODA*. 2021.

[18] M. B. Cohen, Y. T. Lee, and Z. Song. “Solving linear programs in the current matrix multiplication time”. In: *STOC*. 2019.

[19] S. I. Daitch and D. A. Spielman. “Faster Approximate Lossy Generalized Flow via Interior Point Algorithms”. In: *CoRR* abs/0803.0988 (2008). arXiv: 0803.0988.

[20] G. B. Dantzig. “Application of the simplex method to a transportation problem”. In: *Activity analysis and production and allocation* (1951).

[21] S. Dong, Y. Gao, G. Goranci, Y. T. Lee, R. Peng, S. Sachdeva, and G. Ye. “Nested Dissection Meets IPMs: Planar Min-Cost Flow in Nearly-Linear Time”. In: *SODA*. 2022.

[22] S. Dong, Y. T. Lee, and G. Ye. “A nearly-linear time algorithm for linear programs with small treewidth: a multiscale representation of robust central path”. In: *STOC*. 2021.

[23] S. Even and R. E. Tarjan. “Network Flow and Testing Graph Connectivity”. In: *SIAM journal on computing* 4.4 (1975), pp. 507–518.

[24] H. N. Gabow. “Scaling Algorithms for Network Problems”. In: *J. Comput. Syst. Sci.* 31.2 (1985), pp. 148–168.

[25] Y. Gao, Y. P. Liu, and R. Peng. “Fully dynamic electrical flows: sparse maxflow faster than Goldberg-Rao”. In: *FOCS* (2021).

[26] A. Goldberg and R. Tarjan. “Solving minimum-cost flow problems by successive approximation”. In: *STOC*. 1987.

[27] A. V. Goldberg and S. Rao. “Beyond the Flow Decomposition Barrier”. In: *Journal of the ACM* 45.5 (1998), pp. 783–797.

[28] A. V. Goldberg and R. E. Tarjan. “A new approach to the maximum-flow problem”. In: *J. ACM* 35.4 (1988), pp. 921–940.

[29] J. E. Hopcroft and R. M. Karp. “An \$n\^{}{5/2} \$ Algorithm for Maximum Matchings in Bipartite Graphs”. In: *SIAM Journal on Computing* 2.4 (Dec. 1973), pp. 225–231.

[30] N. Karmarkar. “A New Polynomial-Time Algorithm for Linear Programming”. In: *STOC*. 1984.

[31] A. V. Karzanov. “On finding maximum flows in networks with special structure and some applications”. In: *Matematicheskie Voprosy Upravleniya Proizvodstvom* 5 (1973), pp. 81–94.

[32] T. Kathuria, Y. P. Liu, and A. Sidford. “Unit Capacity Maxflow in Almost $O(m^{4/3})$ Time”. In: *FOCS*. 2020.

[33] J. A. Kelner, Y. T. Lee, L. Orecchia, and A. Sidford. “An Almost-Linear-Time Algorithm for Approximate Max Flow in Undirected Graphs, and its Multicommodity Generalizations”. In: *SODA*. 2014.

[34] R. Kyng, R. Peng, S. Sachdeva, and D. Wang. “Flows in Almost Linear Time via Adaptive Preconditioning”. In: *STOC*. 2019.

[35] Y. T. Lee and S. S. Vempala. “Tutorial on the Robust Interior Point Method”. In: *arXiv preprint arXiv:2108.04734* (2021).

[36] T. Leighton and S. Rao. “Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms”. In: *Journal of the ACM (JACM)* 46.6 (1999), pp. 787–832.

[37] Y. P. Liu and A. Sidford. “Faster energy maximization for faster maximum flow”. In: *STOC*. 2020.

[38] A. Mądry. “Computing Maximum Flow with Augmenting Electrical Flows”. In: *FOCS*. 2016.

[39] A. Mądry. “Fast Approximation Algorithms for Cut-Based Problems in Undirected Graphs”. In: *FOCS*. 2010.

[40] A. Mądry. “Navigating central path with electrical flows: From flows to matchings, and back”. In: *FOCS*. 2013.

[41] R. Peng. “Approximate undirected maximum flows in $O(m\text{polylog}(n))$ time”. In: *SODA*. 2016.

[42] J. Sherman. “Area-convexity, ℓ_∞ regularization, and undirected multicommodity flow”. In: *STOC*. 2017.

[43] J. Sherman. “Nearly Maximum Flows in Nearly Linear Time”. In: *FOCS*. 2013.

[44] D. D. Sleator and R. E. Tarjan. “A data structure for dynamic trees”. In: *Journal of computer and system sciences* 26.3 (1983), pp. 362–391.

[45] D. A. Spielman and S. Teng. “Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems”. In: *STOC*. 2004.

[46] P. M. Vaidya. “An Algorithm for Linear Programming which Requires $O(((m+n)n^2+(m+n)^{1.5}n)L)$ Arithmetic Operations”. In: *Math. Program.* 47 (1990), pp. 175–201.