A Deterministic Almost-Linear Time Algorithm for Minimum-Cost Flow

Jan van den Brand
School of Computer Science
Georgia Tech
Atlanta, USA
vdbrand@gatech.edu

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

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

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

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

Aaron Sidford
Department of Management Science & Engineering
and Department of Computer Science
Stanford University
Palo Alto, USA
sidford@stanford.edu

***Abstract*—We give a deterministic $m^{1+o(1)}$ time algorithm that computes exact maximum flows and minimum-cost flows on directed graphs with m edges and polynomially bounded integral demands, costs, and capacities. As a consequence, we obtain the first running time improvement for deterministic algorithms that compute maximum-flow in graphs with polynomial bounded capacities since the work of Goldberg-Rao [J.ACM '98].**

Our algorithm builds on the framework of Chen-Kyng-Liu-Peng-Gutenberg-Sachdeva [FOCS '22] that computes an optimal flow by computing a sequence of $m^{1+o(1)}$-approximate undirected minimum-ratio cycles. We develop a deterministic dynamic graph data-structure to compute such a sequence of minimum-ratio cycles in an amortized $m^{o(1)}$ time per edge update. Our key technical contributions are deterministic analogues of the *vertex sparsification* and *edge sparsification* components of the data-structure from Chen et al. For the vertex sparsification component, we give a method to avoid the randomness in Chen et al. which involved sampling random trees to recurse on. For the edge sparsification component, we design a deterministic algorithm that maintains an embedding of a dynamic graph into a sparse spanner. We also show how our dynamic spanner can be applied to give a deterministic data structure that maintains a fully dynamic low-stretch spanning tree on graphs with polynomially bounded edge lengths, with subpolynomial average stretch and subpolynomial amortized time per edge update.

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

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, NSF Grant CCF-1955039, and a Google Research Fellowship.

Richard Peng was partially supported by NSF CAREER Award CCF-1846218, and the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grant RGPIN-2022-03207. Part of this work was done while at the University of Waterloo.

Sushant Sachdeva was supported by an NSERC Discovery Grant RGPIN-2018-06398, an Ontario Early Researcher Award (ERA) ER21-16-283, and a Sloan Research Fellowship.

Aaron Sidford was supported in part by a Microsoft Research Faculty Fellowship, NSF CAREER Award CCF-1844855, NSF Grant CCF-1955039, a PayPal research award, and a Sloan Research Fellowship.

See https://arxiv.org/abs/2309.16629 for the full version of this paper.

I. Introduction

Given a directed, capacitated graph $G = (V, E, \boldsymbol{u})$ with $n = |V|$ nodes, $m = |E|$ edges, and integer capacities $\boldsymbol{u} \in \mathbb{Z}^E_{\geq 0}$, the *maxflow problem* asks to send as much flow as possible on G from a given source vertex $s \in V$ to a sink vertex $t \in V \setminus \{s\}$ without exceeding the capacity constraints. This problem is foundational in combinatorial optimization and algorithm design. It has been the subject of extensive study for decades, starting from the works [28, 48, 52, 32] and is a key subroutine for solving a variety of algorithmic challenges such as edge-connectivity and approximate sparsest cut (e.g., [45, 56]).

In the standard setting where the capacities are polynomially bounded, a line of work on combinatorial algorithms culminated in a seminal result of Goldberg and Rao in 1998 [42] which showed that the problem can be solved in $\widetilde{O}(m \cdot \min\{m^{1/2}, n^{2/3}\})$ time. The algorithm which achieved this result was deterministic and combinatorial; the algorithm consists of a careful repeated computation of blocking-flows implemented in nearly linear time using dynamic trees. Interestingly, despite advances in randomized algorithms for maxflow ([60, 16, 36, 13]) and deterministic algorithms in special cases (e.g., unit capacity graphs [68] and planar graphs [10, 11]), the runtime in [42] has remained the state-of-the-art among deterministic algorithms in the general case of polynomially bounded capacities.

This gap between state-of-the-art runtimes for deterministic and randomized algorithms for maxflow is particularly striking in light of recent advances: [19] provided an almost linear, $m^{1+o(1)}$, time randomized maxflow algorithm and [15] provided an $\widetilde{O}(m + n^{1.5})$ time randomized algorithm which runs in nearly linear time for dense graphs. Unfortunately, as we discuss in Section I-A, there are key barriers towards efficiently derandomizing both [19] and [15] as well as prior improvements [36, 13, 8].

These results raise key questions about the power of randomization in designing flow algorithms. While there is complexity theoretic evidence that randomization does not affect the polynomial time solvability of decision problems [49] it is less clear what fine-grained effect randomization has on the best achievable runtimes or whether or not a problem can be solved in almost linear time [21]. The problem of obtaining faster deterministic algorithms for maxflow is of particular interest given extensive research over the past decade on obtain faster deterministic algorithms for expander decompositions and flow problems [24, 58], and applications to connectivity problems [54, 64, 62].

In this paper we provide a deterministic algorithm that solves minimum-cost flow and maxflow in $m^{1+o(1)}$ time. We obtain this result by providing an efficient deterministic implementation of the recent flow framework of [19] which reduced the minimum cost flow problem to approximately solving a sequence of structured minimum ratio cycle problems. We also obtain the same running time for deterministically finding flows on graphs that minimize convex edge costs. Further, the techniques we develop have potential broader utility; for example, we show that our techniques can be used to design a deterministic algorithm that dynamically maintains low-stretch trees under insertions and deletions with polynomially bounded lengths (see Section I-B)

a) Paper Organization: In the remainder of this introduction we elaborate on randomized maxflow algorithms and the barriers to their derandomization (Section I-A), present our results (Section I-B), give a coarse overview of our approach (Section I-C), and cover additional related work (Section I-D). We then cover preliminaries in Section II and give a more technical overview in Section III. We present the flow framework in Section IV, build the main dynamic recursive data structure in the full paper.

A. Randomized Maxflow Algorithms

Although the runtime of deterministic algorithms solving maxflow on graphs with polynomially bounded capacities has not been improved since [42], there have been significant advances towards designing randomized maxflow algorithms. Here we provide a brief survey of these advances and discuss the difficulty in obtaining deterministic counterparts of comparable efficiency.

a) Electric Flow Based Interior Point Methods: A number of randomized algorithms over the past decade have improved upon the complexity of maxflow by leveraging and building upon interior point methods (IPMs). IPMs are a broad class of continuous optimization methods that typically reduce continuous optimization problems, e.g., linear programming, to solving a sequence of linear systems. In the special case of maxflow the linear systems typically correspond to electric flow or Laplacian system solving and can be solved in nearly linear time [80].

Combining this approach with improved IPMs, [61] obtained an $\widetilde{O}(m\sqrt{n})$ time maxflow algorithm. Further robustifying this optimization method and using a range of dynamic data structures for maintaining decompositions of a graph into expanders, sparsifiers, and more, [15] obtained an improved $\widetilde{O}(m+n^{1.5})$ time maxflow algorithm. Incorporating additional dynamic data structures for maintaining types of vertex sparsifiers (and more) then led to runtimes of $\widetilde{O}(m^{3/2-1/328})$ [36] and $\widetilde{O}(m^{3/2-1/58})$ [13].

Unfortunately, despite improved understanding of IPMs (in particular deterministic robust linear programming methods [14]) and deterministic Laplacian system solvers [24] it is unclear how to obtain deterministic analogs of these maxflow results. Each result either uses $1/\text{poly}(\log n)$-accurate estimates of effective resistances [61, 15] or edge or vertex sparsifiers of similar accuracy [36, 13]. Obtaining deterministic algorithms for either is an exciting open problem in algorithmic graph theory (and is left unsolved by this paper).

b) Minimum Ratio Cycle Based Interior Point Methods: In a recent breakthrough result [19] leveraged a different type of IPM. This method obtained an almost-linear time algorithm for maxflow and instead used an ℓ_1-counterpart to the more standard ℓ_2-based IPMs that reduce maxflow to electric flow. Using this IPM, [19] essentially reduced solving maxflow to solving a dynamic sequence of minimum ratio cycle problems (e.g., Definition IV.5).

On the one hand, [19] seems to create hope in overcoming the obstacles of faster deterministic maxflow algorithms. Using [19] it is indeed known how to deterministically solve each individual minimum ratio cycle problem to sufficient accuracy in almost linear time. On the other hand, unfortunately [19] required a dynamic data structure for solving these problems in amortized $m^{o(1)}$-per instance and to obtain their runtime, [19] made key use of randomization. In Section I-C we elaborate on the obstacles in avoiding this use of randomization and our main results, which are new algorithmic tools which remove this need.

B. Our Results

We give a deterministic algorithm for computing min-cost flows on graphs.

Theorem I.1 (Min-cost flow)**.** *There is a deterministic algorithm that given a m-edge graph with integral vertex demands and edge capacities bounded by U in absolute value, and integral edge costs bounded by C in absolute value, computes an (exact) minimum-cost flow in time $m^{1+o(1)} \log U \log C$.*

Our algorithm extends to finding flows that minimize convex edge costs to high-accuracy, for example, for matrix scaling, entropy-regularized optimal transport, p-norm flows, and p-norm isotonic regression. See [19, Section 10] for a (deterministic) reduction of these problems to a sequence of minimum ratio cycle problems satisfying the relevant stability guarantees.

Additionally, the components of our data structure can be used to deterministically maintain a low-stretch tree under dynamic updates (see Section II and Theorem II.2 for a formal definition of edge stretch, and a concrete low-stretch tree statement). Previously, deterministic algorithms for maintaining a low-stretch tree with subpolynomial update time were only known for unweighted graphs and those undergoing only edge deletions, achieved by combining the previous result [17] with derandomization techniques in [9, 23]. If randomness is allowed, [33] maintains a low-stretch tree with $n^{1/2+o(1)}$ update time and subpolynomial stretch. [34] gave an algorithm that maintains low-stretch probabilistic tree embedding in subpolynomial update time. Contrary to a conventional low-stretch tree, low-stretch probabilistic tree embeddings may not qualify as a spanning tree, possibly including vertices absent from the input graph. The only way the authors know how to achieve an algorithm that maintains low-stretch trees on graphs with polynomially bounded edge lengths in subpolynomial update time is by adapting the components of [19] to the setting of low-stretch trees.

Theorem I.2 (Dynamic low stretch tree)**.** *There is a deterministic data structure that given a dynamic n-node graph undergoing insertions and deletions of edges with integral lengths bounded by* $\exp((\log n)^{O(1)})$*, maintains a low-stretch tree with average stretch $n^{o(1)}$ in worst-case $n^{o(1)}$ time per update. The data structure maintains the tree in memory with $n^{o(1)}$ amortized recourse per update; the data structure can be be modified to output the changes explicitly with amortized, rather than worst-case, $n^{o(1)}$ time per update.*

C. Our Approach

In this paper we obtain an almost linear time algorithm for maxflow by essentially showing how to eliminate the use of randomness in each of the places it was used [19]. Here we elaborate on these uses of randomness and the techniques we introduce; we provide a more detailed overview of our approach in Section III.

a) Randomization in [19]: At a high level, [19] treats minimum ratio cycle as an instance of the more general minimum cost transshipment problem on undirected graphs. To solve this, [19] applies a time-tested technique of recursively building partial trees (to reduce the number of vertices) and sparsifying (to reduce the number of edges). This approach was pioneered by [80], and has since been used in multiple algorithms [57, 55, 78, 59, 20] and dynamic data structures [18].

More precisely, for a parameter k the partial trees are a collection of $\widetilde{O}(k)$ forests with $O(m/k)$ components where the stretch of every edge in a component is $\widetilde{O}(1)$ on average; here, the stretch of an edge refers to the ratio of the length of routing the edge in the forest to the length of the edge itself. The algorithm uses the partial trees to recursively processes the graphs resulting from contracting each forest. While the forests can be computed and even dynamically maintained deterministically, recursively processing all the partial trees is prohibitively expensive because the total number of components is still $\widetilde{O}(k \cdot m/k) = \widetilde{O}(m)$, i.e., there is no total size reduction. Thus, [19] (motivated in part by [67, 37]) showed that it sufficed to subsample only $\widetilde{O}(1)$ trees to recurse on. This is the first and, perhaps, most critical use of randomness in the [19] algorithm. In particular, it initially seems difficult to design a data structure that maintains all the trees without having a prohibitive runtime.

The dynamic sparsifier constructed in [19] was a spanner with *explicit embedding*, i.e., the algorithm maintained a subgraph $H \subseteq G$, and for each edge $e \in G$, a path in H with few edges that connected its endpoints. This graph H was maintained with low recourse under edge insertions, deletions, and *vertex splits*, where a vertex becomes two vertices, and edges are split between them. The spanner was constructed by maintaining an expander decomposition and uniform sampling edges in each expander. This is the second use of randomness in [19], though it is conceptually easier to circumvent due to recent progress on deterministic expander decomposition and routings [24, 25].

b) Removing randomness from sampling forests: To understand how we remove randomness from sampling the forests, it is critical to discuss how [19] handled the issue of *adaptive adversaries* in the dynamic updates to the data structure (i.e., that the input to the dynamic minimum ratio cycle data structures could depend on the data structure's output). In particular, the future updates to the data structure may depend on the trees that were randomly sampled. To handle this, [19] observed that the IPM provided additional stability on the dynamic minimum ratio cycle problem, in the sense that there was a (sufficiently good) solution $\boldsymbol{\Delta}^*$ to the minimum ratio cycle problem $\arg\min_{\mathbf{B}^\top \boldsymbol{\Delta}=0} \boldsymbol{g}^\top \boldsymbol{\Delta}/\|\mathbf{L}\boldsymbol{\Delta}\|_1$ which changed slowly.

In a similar way, our deterministic min-ratio cycle data structure does *not* work for general dynamic minimum ratio cycle, and instead heavily leverages the stability of a solution $\boldsymbol{\Delta}^*$. As in [19], our algorithm computes the $\widetilde{O}(k)$ partial trees. We know that out of these forests, there exists at least one of them (in fact, at least half of them) that we can successfully recurse on. [19] chooses $\widetilde{O}(1)$ random forests to recurse on, leveraging that at least one of these forests is good with high probability. As discussed, we cannot afford to recurse on all $\widetilde{O}(k)$ forests as this requires dynamically maintaining $\Omega(m)$ trees at every step. Consequently, to obtain a deterministic algorithm we instead show that it suffices to recurse one forest at a time. We recurse on the first forest until we conclude that it did not output a valid solution, then we switch to the next forest, and repeat (wrapping around if necessary). This way, we only maintain one recursive chain and the corresponding spanning tree at each point in time. We argue that the runtime is still acceptable, and more interestingly, that we do not need to switch between branches very frequently. We formalize this, we analyze what we call the *shift-and-rebuild game* in Section 7 of the full paper, and extend the adaptive adversary analysis of [19] to our new algorithm.

c) Deterministically constructing spanners with embeddings: At a high level, [19] gives a deterministic procedure of reducing dynamic spanners to static spanners with embeddings. To construct the static spanner, [19] decomposed G into expanders, sparsified each expander by random sampling, and then embedded G into the sparsifier using a decremental shortest path data structure [25]. The expander decomposition can be computed deterministically using [24]. Thus, the remaining randomized component was the construction of the spanner by subsampling. Instead, we construct the spanner by constructing a deterministic expander W on each piece of the expander decomposition, embedding G into W, and then embedding W back into G (both using the deterministic decremental shortest path data structure [25]). The set of edges in G used to embed W forms the spanner. For our overall maxflow algorithm, we require additional properties of the dynamic spanner algorithm beyond the embedding; see Theorem 8.2 of the full paper.

D. Additional Related Work

a) Derandomization for flow-related problems: Deterministic algorithms for sparsest cut, balanced cut, and expander decomposition [24] can be directly applied to give a variety of deterministic algorithms for flow problems, including solving Laplacian linear systems (electric flows), p-norm flows on unit graphs [59], and more recently, directed Laplacian linear systems [58]. While we utilize deterministic expander decompositions and routings from [24] to give a deterministic spanner with embeddings, these methods seem unrelated to the problem of avoiding subsampling the partial trees.

b) Maxflow / Min-cost flow: Over the last several decades there has been extensive work on the maxflow and minimum cost flow problems [42, 38, 27, 29, 30, 44, 12, 35, 39, 41, 72, 47, 40, 22, 68, 66, 78, 76, 55, 81, 35, 75, 73, 74, 38, 43, 31, 26, 53, 65, 7, 8, 15, 36, 13]. Some of these algorithms, primarily in the instance of unit-capacity maxflow [68, 66, 65, 53], can be made deterministic using deterministic flow primitives from [24].

c) Connectivity problems: There is a long line of work on applications of maxflow to connectivity problems, including sparsest cuts, Gomory-Hu trees, and global mincuts [46, 6, 71, 77, 54, 1, 5, 70, 63, 4, 2, 62]. Some of these algorithms for global mincut can be made deterministic [54, 62], though the techniques often rely on expander decomposition, which, again, does not resolve our issue of sampling partial trees. Since this work was announced, [69] gave a deterministic reduction from k-vertex-connectivity to computing $m^{o(1)}k^2$ maxflows to achieve a deterministic algorithm for k-vertex-connectivity running in $m^{1+o(1)}k^2$ time.

II. Preliminaries

a) General notation: We denote vectors by boldface lowercase letters and matrices by boldface uppercase letters. Often, we use uppercase letters to denote diagonal matrices corresponding to vectors with the matching lowercase letter, e.g., $\mathbf{L} = \text{diag}(\boldsymbol{\ell})$. 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 $|\boldsymbol{x}|_i = |\boldsymbol{x}_i|$. 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 integer h we let $[[h]] \stackrel{\text{def}}{=} \{0, 1, \dots, h\}$, and $[h] \stackrel{\text{def}}{=} \{1, \dots, h\}$. 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 $i \in [n]$.

b) Graphs: We consider multi-graphs G with edge set $E(G)$ and vertex set $V(G)$. When the graph is clear from context, we use E for $E(G)$, V for $V(G)$, $m = |E|$, and $n = |V|$. We assume that each edge $e \in E$ has an implicit direction and overload the notation slightly by writing $e = (u, v)$ where 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 $\mathsf{rev}(e)$ be the edge e reversed: if $e = (u, v)$ points from u to v, then $\mathsf{rev}(e)$ points from v to u.

A *flow vector* is a vector $\boldsymbol{f} \in \mathbb{R}^E$. If $\boldsymbol{f}_e \ge 0$, this means that $\boldsymbol{f}_e$ units flow in the implicit direction of the edge e chosen, and if $\boldsymbol{f}_e \le 0$, then $|\boldsymbol{f}_e|$ units flow in the opposite direction. A *demand vector* is a vector $\boldsymbol{d} \in \mathbb{R}^V$ with $\sum_{v \in V} \boldsymbol{d}_v = 0$. 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, i.e., $\boldsymbol{b}_e$ has a 1 at u, -1 at v, and 0 elsewhere. Define the edge-vertex incidence matrix $\mathbf{B} \in \mathbb{R}^{E \times V}$ as the matrix whose rows are $\boldsymbol{b}_e$. We say that a flow $\boldsymbol{f}$ routes a demand $\boldsymbol{d}$ if $\mathbf{B}^\top \boldsymbol{f} = \boldsymbol{d}$.

We denote by $\deg_G(v)$ the combinatorial 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 G. We define the volume of a set $S \subseteq V$ as $\text{vol}_G(S) \stackrel{\text{def}}{=} \sum_{v \in S} \deg_G(v)$.

Given a set of edges $F \subseteq E(G)$, we define G/F to be the graph where the edges in F are contracted. In this paper, typically this operations is performed for forests F.

c) Dynamic Graphs: In this paper, we say that G is a dynamic graph if it undergoes a sequence of updates. In this paper, the graphs we study will undergo three main types of updates.

- **Edge insertion**: an edge $e = (u, v)$ is added to the graph. The edge is encoded by its endpoints, and when necessary, edge lengths and gradients will also be provided.
- **Edge deletion**: an edge $e = (u, v)$ is deleted from the graph. The edge is encoded by its label in the graph.
- **Vertex split**: a vertex v becomes two vertices v_1 and v_2, and the edges adjacent to v are split between v_1 and v_2. Precisely, every edge $e_i = (v, u_i)$ is assigned to either v_1 or v_2, becoming edge (v_1, u_i) or (v_2, u_i) respectively.

This operation is encoded by listing out the edges moved to the one of v_1, v_2 with a smaller degree. Thus the encoding size is approximately $\min\{\deg(v_1), \deg(v_2)\}$.

In this paper, instead of having our dynamic graphs undergo a single update at a time, we think of them as undergoing *batches* $U^{(1)}, U^{(2)}, \ldots$ of updates, where each batch $U^{(i)}$ denotes a set of updates to apply.

We let $|U^{(t)}|$ denote the total number of updates in the batch, i.e., the total number of edge insertions, deletions, and vertex splits. $\textsc{Enc}(u)$ of an update $u \in U^{(t)}$ denotes its encoding size. As mentioned above, each insertion and deletion can be encoded in size $\widetilde{O}(1)$, while each vertex split can be encoded in size $\widetilde{O}(\min\{\deg(v_1), \deg(v_2)\})$. Finally, the encoding size of a batch $U^{(t)}$ is the sum of the encoding sizes of each of its updates.

Note that $\textsc{Enc}(U^{(t)}) = \Omega(|U^{(t)}|)$, but may be even larger. However, we can bound the total encoding size using the following lemma.

Lemma II.1. *For a dynamic graph G that undergoes batches of updates $U^{(1)}, U^{(2)}, \ldots$ if G initially has m edges then we can bound the total encoding size as $\sum_t \textsc{Enc}(U^{(t)}) = \widetilde{O}\left(m + \sum_t |U^{(t)}|\right)$.*

Proof. Each edge insertion/deletion only contributes $\widetilde{O}(1)$ to the encoding size. Thus, the size of encodings of edge/insertions deletions is at most $\widetilde{O}\left(\sum_t |U^{(t)}|\right)$. In order to account for vertex splits, consider the potential $\Phi = \sum_v \deg(v) \log \deg(v)$. It is straightforward to verify that an edge insertion can only increase the potential by $O(\log m)$. When a vertex v is split into u_1, u_2, the potential decreases by at least $\Omega(\min(\deg(u_1), \deg(u_2)))$. $\square$

d) Paths, Flows, and Trees: Given a path P in G with vertices u, v both on P, then we let $P[u, v]$, which is another path, 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 a a to b path P and a b to c path Q we let $P \oplus Q$ denote the a to c path that is the concatenation of P and Q.

For a forest F, we use $F[u, v]$ to denote the unique simple 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. Additionally, we let $\boldsymbol{p}(F[u, v]) \in \mathbb{R}^{E(G)}$ denote the flow vector which routes one unit from u to v along the path in F. Thus, $|\boldsymbol{p}(F[u, v])|$ 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 and lengths $\boldsymbol{\ell} \in \mathbb{R}^E_{>0}$ is defined as

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

This differs slightly from the more common definition of stretch because due to the additive 1; we choose this definition to ensure that $\mathsf{str}_e^{T,\boldsymbol{\ell}} \geq 1$ for all e. We define the stretch of an edge $e = (u, v)$ with respect to a forest F analogously if u, v are in the same connected component of F. In Section 5 of the full paper, we introduce a notion of stretch when u, v are *not* in the same component of a rooted forest. In this case, the stretch is instead defined as the total distance of u, v to their respective roots divided by the length of e. As stated in the following theorem, it is known how to efficiently construct trees with polylogarithmic average stretch with respect to underlying weights; we call these low-stretch spanning trees (LSSTs).

Theorem II.2 (Static LSST [3]). *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 $\sum_{e \in E} \boldsymbol{v}_e \mathsf{str}_e^{T,\boldsymbol{\ell}} \leq \gamma_{LSST} \|\boldsymbol{v}\|_1$ for some $\gamma_{LSST} \stackrel{\text{def}}{=} O(\log n \log \log n)$.*

In this paper, in contrast to eg., [19, Lemma 6.5], we often use the cruder upper bound of of $\gamma_{LSST} = O(\log^2 n)$. We do this to simplify the presentation as it does not effect the final asymptotic bounds claimed.

e) Graph Embeddings: Given weighted graphs G and H with $V(G) \subseteq V(H)$, we say that $\Pi_{G \to H}$ is a *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. Let $\boldsymbol{w}_G$ be the weight function of G and $\boldsymbol{w}_H$ be the weight function of H. We define the *congestion* of an edge e^H by

$$\text{econg}(\Pi_{G \to H}, e^H) \stackrel{\text{def}}{=} \frac{\sum_{e^G \in E(G) \text{ with } e^H \in \Pi_{G \to H}(e^G)} \boldsymbol{w}_G(e^G)}{\boldsymbol{w}_H(e^H)}$$

and the congestion 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

$$\text{vcong}(\Pi_{G \to H}, v^H) \stackrel{\text{def}}{=} \sum_{e^G \in E(G) \text{ with } v^H \in \Pi_{G \to H}(e^G)} \boldsymbol{w}_G(e^G)$$

and the vertex-congestion of the graph-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 of the embedding by $\text{length}(\Pi_{G \to H}) \stackrel{\text{def}}{=} \max_{e^G \in E(G)} |\Pi_{G \to H}(e^G)|$.

Given graphs A, B, C and graph-embeddings $\Pi_{B \to C}$ from B into C and $\Pi_{A \to B}$ from A to B. We denote by $\Pi_{B \to C} \circ \Pi_{A \to B}$ the graph embedding of A into C obtained by mapping each edge $e^A = (u, v) \in E(A)$ with path $\Pi_{A \to B}(e^A) = e^B_1 \oplus e^B_2 \oplus \ldots \oplus e^B_k$ in B to the path $\Pi_{B \to C}(e^B_1) \oplus \Pi_{B \to C}(e^B_2) \oplus \ldots \oplus \Pi_{B \to C}(e^B_k)$. The following useful fact is straightforward from the definitions.

Fact 1. *Given graphs A, B, C and graph-embeddings $\Pi_{B \to C}$ from B into C and $\Pi_{A \to B}$ from A to B. Then,* $\text{vcong}(\Pi_{B \to C} \circ \Pi_{A \to B}) \leq \text{vcong}(\Pi_{B \to C}) \cdot \text{econg}(\Pi_{A \to B})$.

f) Computational Model: For problem instances encoded with z bits, all algorithms developed in this paper work in fixed-point arithmetic where words have $O(\log^{O(1)} z)$ bits, i.e., we prove that all numbers stored are in $[\exp(-\log^{O(1)} z), \exp(\log^{O(1)} z)]$. In particular, Theorem IV.6 says that the min-ratio cycle problems solved by our algorithm satisfy Definition IV.4, where item 5 says that all weights and lengths are bounded by $\exp(\log^{O(1)} m)$.

III. Technical Overview

Our approach for obtaining a *deterministic* almost-linear time min-cost flow algorithm follows the framework of the recent randomized algorithm in [19]. We start by reviewing the algorithm in [19] and then lay out the challenges in obtaining deterministic analogs of its randomized components. By scaling arguments (see [19, Lemma C.1]), we assume that $U, C \leq m^{O(1)}$.

A. The Randomized Algorithm in [19]

a) The Outer-Loop: An ℓ_1-Interior Point Method: The starting point for the randomized algorithm in [19] is a new ℓ_1-interior point method (IPM), which is actually completely *deterministic*. This method uses a *potential reduction* IPM inspired by [51], where in each iteration, the potential function $\Phi(\boldsymbol{f}) \stackrel{\text{def}}{=} 20m\log(\boldsymbol{c}^\top \boldsymbol{f} - F^*) + \sum_{e\in E}((\boldsymbol{u}_e^+ - \boldsymbol{f}_e)^{-\alpha} + (\boldsymbol{f}_e - \boldsymbol{u}_e^-)^{-\alpha})$ is reduced. Here, $\alpha = 1/\Theta(\log m)$, but the reader can think of the barrier $x^{-\alpha}$ as the more standard $-\log x$ for simplicity.

[19] showed that one can assume that an initial feasible solution $\boldsymbol{f}^{(0)}$ is given that routes the demand and has $\Phi(\boldsymbol{f}^{(0)}) \leq O(m\log m)$ and that the IPM can be terminated once the potential function value is at most $-200m\log m$, as at this point, one can round the flow to an exact solution using an isolation lemma; see [19, Lemma 4.11]. While this step is randomized, it can easily be derandomized using an alternate flow rounding procedure, as is explained later at the start of Section III-B. We next discuss how to achieve a potential reduction of $\Phi(\boldsymbol{f})$ in each iteration by $m^{-o(1)}$. This yields that the IPM terminates within $m^{1+o(1)}$ steps.

To obtain a potential reduction of $m^{-o(1)}$ at each step, given a current feasible flow $\boldsymbol{f}$, the update problem involves finding an update direction $\boldsymbol{\Delta}$ to update the flow to $\boldsymbol{f} + \boldsymbol{\Delta}$ such that (a) $\boldsymbol{\Delta}$ is a circulation, i.e., adding it to $\boldsymbol{f}$ does not change the net routed demands and (b) $\boldsymbol{\Delta}$ approximately minimizes the inner product with a linear function (the gradient of Φ), relative to an ℓ_1-norm that arises from the second derivatives of Φ. Letting $\boldsymbol{g} \in \mathbb{R}^E$ denote this gradient and letting $\boldsymbol{\ell} \in \mathbb{R}_+^E$ be the edge length (both with respect to the current flow $\boldsymbol{f}$), we can write the update problem as

$$\min_{\boldsymbol{\Delta}\in\mathbb{R}^E : \mathbf{B}^\top\boldsymbol{\Delta}=\mathbf{0}} \frac{\boldsymbol{g}^\top\boldsymbol{\Delta}}{\left\|\operatorname{diag}(\boldsymbol{\ell})\boldsymbol{\Delta}\right\|_1}. \tag{1}$$

We refer to this update problem henceforth as the *min-ratio cycle problem*, since, by a cycle-decomposition argument, the optimal value is always realized by a simple cycle. As shown in [19], the update problem has several extremely useful properties:

1) At every time step t, the direction from the current solution $\boldsymbol{f}^{(t)}$ towards the optimal flow $\boldsymbol{f}^*$, henceforth called the *witness* $\boldsymbol{\Delta}^{(t)} \stackrel{\text{def}}{=} \boldsymbol{f}^* - \boldsymbol{f}^{(t)}$, achieves $\frac{\boldsymbol{g}^\top\boldsymbol{\Delta}^{(t)}}{\left\|\operatorname{diag}(\boldsymbol{\ell})\boldsymbol{\Delta}^{(t)}\right\|_1} \leq -\frac{1}{\Theta(\log m)}$.
2) Performing the update with a cycle $\boldsymbol{\Delta}$ with $\frac{\boldsymbol{g}^\top\boldsymbol{\Delta}}{\left\|\operatorname{diag}(\boldsymbol{\ell})\boldsymbol{\Delta}\right\|_1} = -\kappa$ reduces the potential by $\Omega(\kappa^2)$. Thus, even finding an $m^{o(1)}$-approximate min-ratio cycle reduces the potential by $m^{-o(1)}$. After $m^{1+o(1)}$ iterations of updates, the potential becomes small enough, and we can round the current flow to an exact solution.
3) The convergence rate is unaffected if we use approximations $\widehat{\boldsymbol{g}}$ and $\widehat{\boldsymbol{\ell}}$ of the gradient $\boldsymbol{g}$ and the lengths $\boldsymbol{\ell}$ such that both $\widehat{\boldsymbol{g}}$ and $\widehat{\boldsymbol{\ell}}$ are updated only a total of $m^{1+o(1)}$ times (here, by update we mean that a single entry of $\widehat{\boldsymbol{g}}$ and $\widehat{\boldsymbol{\ell}}$ is changed) throughout the entire algorithm.

In this way, the ℓ_1-IPM gives a *deterministic* reduction of (exact) min-cost flow to solving a sequence of stable min-ratio cycle problems.

b) A Data Structure for the Min-Ratio Cycle Problem: Since when solving min-cost flow by approximately solving a sequence of min-ratio cycles, the underlying graph remains the same, and gradient and lengths change sporadically throughout the algorithm, it is useful to think about the repeated solving of the min-ratio cycle problem as a *data structure problem*. This problem is is formalized in Definition IV.5. [19] designs a *randomized* data structure for the min-ratio cycle problem which supports the following operations:

- INITIALIZE($G, \widehat{\boldsymbol{g}}^{(0)}, \widehat{\boldsymbol{\ell}}^{(0)}$): initialize the data structure for graph G and the initial approximate gradients, $\widehat{\boldsymbol{g}}^{(0)}$, and lengths, $\widehat{\boldsymbol{\ell}}^{(0)}$, on the edges of G.
- UPDATE($\widehat{\boldsymbol{g}}^{(t)}, \widehat{\boldsymbol{\ell}}^{(t)}$) : the t-th update replaces current gradient and lengths by $\widehat{\boldsymbol{g}}^{(t)}$ and $\widehat{\boldsymbol{\ell}}^{(t)}$.
- QUERY() : returns a cycle whose ratio with respect to the current gradient $\widehat{\boldsymbol{g}}^{(t)}$ and lengths $\widehat{\boldsymbol{\ell}}^{(t)}$ is within a $m^{o(1)}$ factor of $\boldsymbol{\Delta}^{(t)} = \boldsymbol{f}^* - \boldsymbol{f}^{(t)}$.

In the UPDATE operation, $\widehat{\boldsymbol{g}}^{(t)}, \widehat{\boldsymbol{\ell}}^{(t)}$ are described by their changes from $\widehat{\boldsymbol{g}}^{(t-1)}, \widehat{\boldsymbol{\ell}}^{(t-1)}$. By the above discussion, there are $m^{o(1)}$ coordinate changes on average per instance.

Note that the output cycle returned by QUERY() may have nonzero flow on $\Omega(n)$ edges for each of the $\Omega(m)$ iterations (this is often referred to as the *flow decomposition barrier*). Thus we cannot efficiently, explicitly output the solutions. To overcome this issue, the data structure in [19] maintains a $s = m^{o(1)}$ spanning trees $T_1, T_2, \ldots, T_s$ of the graph G. Each such tree is itself a dynamic object, i.e., these trees undergo changes over time in the form of edge insertions and deletions. However, the total number of such edge insertions and deletions is at most $m^{1+o(1)}$ when amortizing over the sequence of updates generated by the IPM. Using these dynamic trees $T_1, T_2, \ldots, T_s$, whenever the operation QUERY() is invoked, the data structure in [19] finds (with high probability) an approximate min-ratio cycle that consists of $m^{o(1)}$ subpaths of a tree T_i and $m^{o(1)}$ additional edges. Using the start and end points of each tree path, the query operation can encode each solution efficiently, as desired.

As shown in [19], the data structure can overall be implemented to run in amortized $m^{o(1)}$ time per query and update, yielding an almost-linear algorithm for the min-cost flow problem.

c) Maintaining Trees in the Data Structure: It remains to review how the data structure in [19] *efficiently* maintains a set of dynamic trees $T = \{T_1, T_2, \ldots, T_s\}$ such that one of the trees yields a cycle with sufficient ratio with high probability, and how to query this cycle.

To construct the set of trees T, [19] draws on the theory of low-stretch spanning trees (LSSTs). Let $G = G^{(0)}$ be the original graph whose edge lengths are given by the vector $\widehat{\boldsymbol{\ell}}^{(0)}$. [19] applied a standard multiplicative weights argument [67, 78, 55] to construct a set of k (partial) trees $T_1, \ldots, T_k$ such that every edge e had average stretch $\widetilde{O}(1)$ over these k trees. Thus, if $\widehat{\boldsymbol{\ell}}$ is the vector of stretches of a random tree among the T_i, then the *witness* $\boldsymbol{\Delta}^{(0)} = \boldsymbol{f}^* - \boldsymbol{f}^{(0)}$ satisfies in expectation $\|\mathrm{diag}(\widetilde{\boldsymbol{\ell}})\boldsymbol{\Delta}^{(0)}\|_1 \leq m^{o(1)}\|\mathrm{diag}(\widehat{\boldsymbol{\ell}}^{(0)})\boldsymbol{\Delta}^{(0)}\|_1$ By Markov's inequality, this same guarantee (up to constants) must hold with probability at least $1/2$. When this occurs, we say the *stretch of the witness with respect to the tree* is low. By sampling $O(\log m)$ trees among $\{T_1, \ldots, T_k\}$ [19] ensures that this occurs in at least one tree with high probability. A basic flow decomposition result then implies that one of the *fundamental cycles* formed by an off-tree edge and the tree-path (in T) between its endpoints yields an $m^{o(1)}$-approximate solution.

Now, consider what happens after the current flow solution is changed from $\boldsymbol{f}^{(0)}$ to $\boldsymbol{f}^{(1)}$ by adding the first update. This changes the witness from $\boldsymbol{\Delta}^{(0)}$ to $\boldsymbol{\Delta}^{(1)} = \boldsymbol{f}^* - \boldsymbol{f}^{(1)}$ and changes the (approximate) gradient from $\widehat{\boldsymbol{g}}^{(0)}$ to $\widehat{\boldsymbol{g}}^{(1)}$, and lengths from $\widehat{\boldsymbol{\ell}}^{(0)}$ to $\widehat{\boldsymbol{\ell}}^{(1)}$. To solve the next update problem, the sampled trees have to be updated so that the stretch of the new witness $\boldsymbol{\Delta}^{(1)}$ is again low with respect to at least one of the trees (now with respect to $\widehat{\boldsymbol{\ell}}^{(1)}$).

To update the trees, for each sampled tree T some edges are removed and then replaced by new edges. To obtain an efficient implementation, [19] applies the well-established technique of maintaining a hierarchy of partial trees/forests. At each level, a partial tree is computed, and the next level then finds again a partial tree in the graph where edges in the partial tree at the higher levels are contracted. Let us illustrate how such a partial tree is found. At the highest level of the hierarchy, a partial tree/forest F is computed with m/k connected components for some target value $k = m^{o(1)}$. F is computed so that it only undergoes edge deletions, and at most $\widetilde{O}(1)$ per update. Additionally, either a fundamental cycle of F or a cycle in G/F (the graph where F is contracted) has ratio within $\widetilde{O}(1)$ factor of the desired min-ratio cycle in G. This is illustrated in Figure 1. This reduces the problem of finding a min-ratio cycle mainly to finding such a cycle in the graph G/F, which has at most m/k vertices.

We refer to the step of maintaining F and contracting to the graph G/F as the *vertex sparsification* phase. However, G/F might still contain almost all edges of G. To reduce the edge count, [19] computes a spanner G' of G/F that yields a reduction in the number of edges to roughly m/k. We refer to this as *edge sparsification*, and give a more detailed overview of the construction in [19] below. The spanner also allows us to either obtain the solution to the min-ratio cycle problem directly from the spanner construction, or approximately preserve the solution quality in G'. The algorithm then recurses, again building a partial tree (forest) F' on G', finding a spanner, and so on. The tree T is taken as the union of the contracted forests F, F', and the forests found in deeper recursion levels.

Whenever lengths and gradients change, updates are handled by making a few adjustments to the forest and then propagating changes to the deeper levels. By controlling carefully the propagation, the total number of updates across levels remains small.

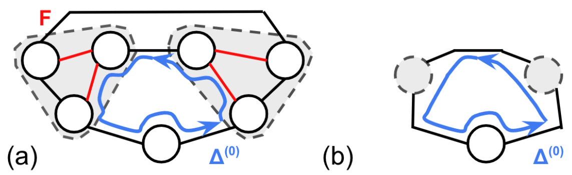

Fig. 1: In (a), we see a graph G and a forest F (the subgraph shown with red edges). The two non-trivial connected components of F are encircled by a dotted border bash. In blue, we show the witness circulation $\boldsymbol{\Delta}^{(0)}$. In (b), we see the graph G/F obtained by contracting the components of G and the circulation $\boldsymbol{\Delta}^{(0)}$ again in blue mapped to G/F. The algorithm ensures that since each contracted edge is approximated well by a path in F, either a solution to (1) is formed by one of the fundamental cycles, or that the mapped circulation in G/F is a good solution to (1).

d) Edge Sparsification: As the forest F undergoes edge deletions, the graph G/F undergoes edge deletions and vertex splits. To design an algorithm to maintain a spanner of G/F, [19] gave a deterministic reduction from maintaining a spanner in an unweighted graph under edge deletions and vertex splits to statically constructing a spanner with low-congestion *edge embeddings*. This means that for a spanner $H \subseteq G$, every edge $e \in E(G) \setminus E(H)$ is mapped into a short path $\Pi_{G \to H}(e)$ in H between its endpoints with at most $m^{o(1)}$ edges, such at every vertex in H has at most $m^{o(1)} \deg_G(v)$ paths through it. Note that at least $\deg_G(v)$ paths go through v in any embedding, so having low vertex congestion means that we match this bound up to the $m^{o(1)}$ factor. It is worth noting that while we only require a spanner of G/F for the algorithm, the reduction only works with a low-congestion embedding. Additionally, our dynamic low-stretch tree data structure makes use of this additional low-congestion property.

Thus, we focus on statically constructing spanners of unweighted graphs with low-congestion embeddings. [19] designed the following algorithm to achieve this. We may assume that G/F has is unweighted by the standard trick of bucketing edges in $\widetilde{O}(1)$ groups whose lengths are within a factor of 2. First, the graph G/F is decomposed into expanders $H_1, H_2, \ldots, H_\ell$, where each vertex appears in $O(\log m)$ expanders, and each expander is almost-uniform-degree in that every degree is within an $O(\log m)$ factor of the average. Thus, for each graph H_i, *random* sampling each edge uniformly with probability about $\widetilde{O}(1)$ divided by the degree of H_i yields a graph H_i' that again is an almost-uniform-degree expander, except now with polylogarithmic degrees. We let the spanner G' of G be the union of all such sampled graphs H_i' and clearly G' is sparse, i.e., has at most $\widetilde{O}(|V(G/F)|)$ edges. This is the only randomized component of the edge sparsification step.

While proving that G' is a spanner of G is rather straightforward, we also must construct a low-congestion embedding of G into G'. In [19], this is achieved by embedding each graph H_i into the corresponding down-sampled graph H_i' for every i. In [19], this is achieved by a deterministic procedure that internally uses the decremental shortest paths data structure on expanders by Chuzhoy and Saranurak [25]. Finally, [19] takes the embedding from G into G' to be the union of the embeddings from H_i to H_i' for all i.

We conclude our discussion on edge sparsification by describing how to find a min-ratio cycle from the spanner construction. Given a spanner G' of G/F with embedding, a flow decomposition arguments shows that either some *spanner cycle* $e \oplus \Pi_{(G/F)\to G'}(e)$ has ratio within $m^{o(1)}$ of $\boldsymbol{\Delta}^{(t)}$, or the circulation in G' achieved by routing $\boldsymbol{\Delta}^{(t)}$ along the paths $\Pi_{(G/F)\to G'}$ into G' has ratio within $m^{o(1)}$ of $\boldsymbol{\Delta}^{(t)}$. By maintaining the paths $\Pi_{(G/F)\to G'}(e)$ explicitly, and recursing on G', our data structure can efficiently query for a min-ratio cycle. This argument is covered in more detail in Section 6 of the full paper.

e) A Note on the Interaction Between Data Structure and Witness: The above description of the data structure is an oversimplification and hides many key details. Perhaps most importantly, the proof of correctness for the data structure in [19] crucially hinges on the existence of the witness $\boldsymbol{\Delta}^{(t)} = \boldsymbol{f}^* - \boldsymbol{f}^{(t)}$ in order to show that the near-optimal cycle $\boldsymbol{\Delta}^{(t)}$ does not ever incur too much stretch even under possibly adaptive updates. Put another way, the data structure does *not* work against general adaptive adversaries, whose updates can depend on the randomness of the data structure, but can be used to solve min-cost flow due to the stability of the witness solution. Similarly, in this paper we do not design a deterministic data structure for general min-ratio cycle instances. Instead, we also require that the update sequence admits a stable witness; leveraging the stable witness in both cases require modifications to both the LSST and spanner data structures, and these are deferred to the main body of the paper.

B. A Deterministic Min-cost Flow Algorithm

Building on the exposition of the algorithm in [19] given in Section III-A, we are now ready to discuss the key changes necessary to obtain our deterministic algorithm. Here, we highlight the parts of [19] that required randomization and outline strategies to remove the randomness.

a) Derandomizing the IPM Framework: The main challenge in derandomizing the framework of [19] is in derandomizing the vertex and edge sparsification routines and solving the requisite dynamic min-ratio cycle problem. Indeed, derandomizing the remainder of the IPM framework is straightforward, because both the IPM presented in the last section and the procedure that maintains the approximate gradient $\widehat{\boldsymbol{g}}$ and the lengths $\widehat{\boldsymbol{\ell}}$ are completely deterministic. The only use of randomness in the above approach, beyond the min-ratio cycle data structure, occurred as [19] rounded the solution when the potential is sufficiently small, i.e. $\Phi(\boldsymbol{f}) \leq -\Omega(m \log m)$, via the Isolation Lemma. However, the use of the Isolation Lemma can be replaced by a deterministic flow rounding procedure using Link-Cut trees [79] as was shown in [50] (see Lemma IV.1).

In the min-ratio cycle data structure of [19], there are two randomized components:

1) $\widetilde{O}(1)$ forests are sampled at each level of the hierarchy, and
2) The spanner of G/F is constructed by decomposing G/F into expanders, and random sampling within each expander.

Below we discuss how to remove the randomness from the first *vertex sparsification* step, and then discuss the second *edge sparsification* step.

b) Derandomizing Vertex Sparsification: Recall that the vertex sparsification construction described above computes a set of k forests $F_1, \dots, F_k$. Of these, $\widetilde{O}(1)$ are sampled, and for each sampled forest F, the algorithm recurses on G/F. A natural approach to derandomize this is to instead recurse on *all* k forests in the collection to deterministically ensure that some forest has low stretch of the witness $\boldsymbol{\Delta}^{(t)}$. Unfortunately this is too expensive, as it leads to $\Omega(m)$ trees being maintained overall. Additionally, every update to the input graph may change every tree, and this approach would therefore lead to linear time per update which is far more than we can afford.

However, we show that, somewhat surprisingly, the following strategy works: instead of directly recursing on all forests, and therefore, on all graphs $G/F_1, G/F_2, \dots, G/F_s$, we can recurse only on the first such tree G/F_1 and check if we find a solution to the min-ratio cycle problem. If we do, we do not need to check $G/F_2, \dots, G/F_s$ at that moment. Otherwise, we move on to G/F_2 (we refer to this as a *shift*), but now know that G/F_1 did at some point not give a solution to the min-ratio cycle and F_1 is therefore not a good forest so that we never have to revisit it. Carefully shifting through these forests $F_1, F_2, \dots, F_s$, it then suffices to only forward the updates to G (in the form of updates to $\widehat{\boldsymbol{\ell}}$) to the forest that is currently used. When we move to the next forest after failing to identify a solution to the min-ratio cycle problem, we then apply all updates that previously happened to G to the contracted graph. We apply this shifting procedure recursively.

Now we need to understand why this improved the amortized update time to $m^{o(1)}$, and perhaps more interestingly, why the algorithm finds an approximate min-ratio cycle without cycling through many graphs G/F_i over the course of the algorithm. At a high level, the runtime is acceptable because the number of dynamic updates to the forests F_i is at most that of the randomized case, as we only maintain a single branch of the recursion at a time. Shifting between forests does not cause dynamic updates, and thus can be charged to the original construction cost.

To understand why the algorithm does not have to shift through several graphs G/F_i every iteration, recall that the witness $\mathbf{\Delta}^{(t)} = \boldsymbol{f}^* - \boldsymbol{f}^{(t)}$ is stable in that only $\widetilde{O}(1)$ edges values change by a constant factor multiplicatively on average per iteration, and that these edges are passed to the data structure. This allows us to show that if we find a forest F_i whose stretch against the witness $\mathbf{\Delta}^{(t)}$ was small, then it stays small until we must rebuild after about m/k updates, or $\|\mathrm{diag}(\boldsymbol{\ell}^{(t)})\mathbf{\Delta}^{(t)}\|_1$ decreases by a constant, which can only happen $\widetilde{O}(1)$ times. This way, over the course of m/k updates, our algorithm only needs to shift $\widetilde{O}(k)$ total times. The major challenge towards formalizing this analysis is that the data structure has multiple levels, which severely complicates the condition that a forest F_i maintains small stretch for several iterations, because we do not know which level caused the failure. We analyze this algorithm through what we call the *shift-and-rebuild game* (Section 7 of the full paper), a generalization of the (simpler) rebuilding game in [19].

c) Derandomizing Edge Sparsification: From the description given above, the main challenge for derandomization of the edge sparsification procedure from [19] is to find a spanner H_i' of an almost-uniform-degree expander H_i such that H_i' consists of few edges and such that we can find a small vertex congestion short-path embedding of the graph H_i into H_i'.

We use the following natural derandomization approach: given H_i with maximum-degree $d_i^{\max}$, we first deterministically construct a constant-degree expander W over the vertex set of H_i. Using the tools from [19], we then compute an embedding $\Pi_{W\to H_i}$ from W into the graph H_i with $m^{o(1)}$ vertex congestion using only short paths. Reusing these tools, we also compute an embedding $\Pi_{H_i\to W}$ from H_i into W with $m^{o(1)} \cdot d_i^{\max}$ *edge congestion* using only short paths.

Now consider the embedding given by $\Pi_{W\to H_i} \circ \Pi_{H_i\to W}$ which maps edges from H_i to paths in W and then back to paths in H_i. We claim that the graph H_i' consisting of the edges in the image of $\Pi_{W\to H_i}$ is a spanner, and $\Pi_{W\to H_i} \circ \Pi_{H_i\to W}$ embeds H_i into H_i' with low vertex congestion and short paths. To see this, we first show that H_i' is sparse, i.e., it has at most $|V(H_i)|m^{o(1)}$ edges. This follows because W has only $O(|V(H_i)|)$ edges by construction, and each edge is mapped to a path of length at most $m^{o(1)}$. Thus the image of $\Pi_{W\to H_i}$ consists of at most $|V(H_i)|m^{o(1)}$ edges. Further, using Fact 1, we immediately obtain that $\Pi_{W\to H_i} \circ \Pi_{H_i\to W}$ has vertex congestion $m^{o(1)} \cdot d_i^{\max}$ and it is not hard to see that each embedding path in $\Pi_{W\to H_i} \circ \Pi_{H_i\to W}$ is short.

There are some additional side constraints that the spanners need to satisfy to work in the framework of our overall algorithm, relating to leveraging the stability of the witness $\mathbf{\Delta}^{(t)}$. Ensuring that these constraints are met requires additional careful analysis, which we give in Section 8 of the full paper.

d) Dynamic Low-Stretch Trees: Our algorithm that dynamically maintains low-stretch trees uses a very similar hierarchical data structure as to our dynamic min-ratio cycle algorithm. At the top level, we statically compute a low-stretch tree, and maintain a partial forest F with $O(m/k)$ connected components under edge updates. We then maintain a spanner of G/F with explicit edge embeddings by applying the deterministic edge sparsification algorithm described above. Finally, we recurse on the spanner of G/F.

IV. Flow Framework

In this section, we discuss our main algorithm for solving flow problems to high accuracy.

We first note that in order to solve a min-cost flow problem exactly, it suffices to find a good enough fractional solution. We use the following result which, as an immediate corollary, shows that to solve min-cost flow it suffices to find a feasible fractional flow $\boldsymbol{f}$ with a cost that is within an additive $1/2$ of the optimal cost.

Lemma IV.1 ([50, Section 4])**.** *There is a deterministic algorithm which when given a feasible fractional flow $\boldsymbol{f}$ in a m-edge n-vertex mincost flow instance with integer capacities outputs a feasible integer flow $\boldsymbol{f}'$ with cost no larger than $\boldsymbol{f}$, in $O(m \log m)$ time.*

To find such an approximate min-cost flow, we use the IPM algorithm introduced in Chen et al. [19] that can find an almost-optimal fractional solution to the min-cost flow problem by solving a sequence of min-ratio cycle problems. In order to state the guarantees of the algorithm, we first define the min-ratio cycle problem and the dynamic variant of it that we consider.

Definition IV.2 (Min-Ratio Cycle)**.** *Given a graph $G(V,E)$, gradients $\boldsymbol{g} \in \mathbb{R}^E$, and lengths $\boldsymbol{\ell} \in \mathbb{R}^E_{>0}$, the* min-ratio cycle problem *seeks a circulation $\mathbf{\Delta}$ satisfying $\mathbf{B}^\top \mathbf{\Delta} = 0$ that (approximately) minimizes $\frac{\langle \boldsymbol{g}, \boldsymbol{f}\rangle}{\|\mathbf{L}\boldsymbol{f}\|_1}$ where $\mathbf{L} = \mathrm{diag}(\boldsymbol{\ell})$.*

Observe that the minimum objective value of the min-ratio cycle problem is non-positive since for any circulation $\mathbf{\Delta}$, the flow $-\mathbf{\Delta}$ is also a circulation.

Extending the problem definition to the dynamic setting, a dynamic min-ratio cycle problem with T instances is described by a dynamic graph $G^{(t)}$, gradients $\boldsymbol{g}^{(t)} \in \mathbb{R}^E$, and lengths $\boldsymbol{\ell}^{(t)} \in \mathbb{R}^E_{>0}$, where the dynamic graph is undergoing a batch of updates $U^{(1)}, \ldots, U^{(T)}$.

The IPM algorithm from [19] requires solving a dynamic min-ratio cycle problem. The data structure from Chen et al. for solving the dynamic min-ratio cycle problem requires a stability condition, roughly requiring that there is a dynamic witness for the problem instances whose length changes slowly across iterations. This condition is captured in the following definitions:

Definition IV.3 (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 $e \in E$.*

Definition IV.4 (Hidden Stable α-Flow Updates). *We say that a dynamic min-ratio cycle instance described by a dynamic graph $G^{(t)}$, gradients $\boldsymbol{g}^{(t)}$, and lengths $\boldsymbol{\ell}^{(t)}$ satisfies the* hidden stable α-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, i.e., $\mathbf{B}_{G^{(t)}}^{\top}\boldsymbol{c}^{(t)} = 0$.*
2) *$\boldsymbol{c}^{(t)}$ and $\boldsymbol{w}^{(t)}$ are a valid pair with respect to $G^{(t)}$.*
3) *$\boldsymbol{c}^{(t)}$ has sufficiently negative objective value relative to $\boldsymbol{w}^{(t)}$, i.e., $\frac{\langle \boldsymbol{g}^{(t)}, \boldsymbol{c}^{(t)} \rangle}{\|\boldsymbol{w}^{(t)}\|_1} \leq -\alpha$.*
4) *For any edge e in the current graph $G^{(t)}$, and any stage $t' \leq t$, if the edge e was not explicitly inserted after stage t', then $\boldsymbol{w}_e^{(t)} \leq 2\boldsymbol{w}_e^{(t')}$. However, between stage t' and t, endpoints of edge e might change due to vertex splits.*
5) *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 IV.4 says that even while $\boldsymbol{g}^{(t)}$ and $\boldsymbol{\ell}^{(t)}$ change, there is a witness circulation $\boldsymbol{c}^{(t)}$ that is fairly stable. In particular, there is an upper bound $\boldsymbol{w}^{(t)}$ on the coordinate-wise lengths of $\boldsymbol{c}^{(t)}$ that stays the same up to 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 to facilitate efficient implementations. For brevity, use the term *Hidden Stability* to refer to Definition IV.4 in the rest of the paper.

Definition IV.5. *The problem of κ-approximate Dynamic Min-Ratio Cycle with Hidden Stability asks for a data structure that, at every stage t, finds a circulation $\boldsymbol{\Delta}^{(t)}$, i.e., $\mathbf{B}_{G^{(t)}}^{\top}\boldsymbol{\Delta}^{(t)} = 0$ such that $\frac{\langle \boldsymbol{g}^{(t)}, \boldsymbol{\Delta} \rangle}{\|\mathbf{L}^{(t)}\boldsymbol{\Delta}\|_1} \leq -\kappa\alpha$. Additionally, we require that the data structure maintains a flow $\boldsymbol{f} \in \mathbb{R}^E$ that is initialized at* $\mathbf{0}$, *and supports the following operations:*

1) Update$(U^{(t)}, \boldsymbol{g}^{(t)}, \boldsymbol{\ell}^{(t)}, \eta)$. *Apply edge insertions/deletions specified in updates $U^{(t)}$ and update gradients $\boldsymbol{g}^{(t)}$ and lengths $\boldsymbol{\ell}^{(t)}$ for these edges. Find a circulation $\boldsymbol{\Delta}^{(t)}$ that approximately solves the min-ratio problem as noted above. Update $\boldsymbol{f} \leftarrow \boldsymbol{f} - \beta\boldsymbol{\Delta}^{(t)}$, where $\beta = \frac{\eta}{(\boldsymbol{g}^{(t)})^{\top}\boldsymbol{\Delta}^{(t)}}$.*
2) Query(e). *Returns the value $\boldsymbol{f}_e$.*
3) Detect$()$. *For a fixed parameter ε, where $\boldsymbol{\Delta}^{(t)}$ is the update vector at stage t, returns*

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

where $\mathsf{last}_e^{(t)}$ is the last stage before t that e was returned by Detect$()$.

Observe that the approximation ratio holds only with respect to the quality of the hidden stable witness circulation $\boldsymbol{c}^{(t)}$, and not with respect to the best possible circulation. As a sanity check, if the data structure could find and return $\boldsymbol{c}^{(t)}$ at each iteration, it would achieve a 1-approximation. 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 $\boldsymbol{\Delta}^{(t)}$ to our flow efficiently by using link-cut trees to efficiently implement Update$(\cdot)$.

The following theorem encapsulates the IPM algorithm presented in [19] and its interface with the dynamic min-ratio cycle data structure.

Theorem IV.6 ([19]). *Assume we have access to a κ-approximate dynamic min-ratio cycle with hidden stability data structure, for some $\kappa \in (0, 1]$ as in Definition IV.5. Then, there is a deterministic IPM-based algorithm that given a min-cost flow problem with integral costs and capacities bounded by $\exp((\log n)^{O(1)})$ in absolute value, solves $\tau = \widetilde{O}(m\kappa^{-2})$ min-ratio cycle instances, and returns a flow with cost within additive $1/2$ of optimal. These $\widetilde{O}(m\kappa^{-2})$ many min-ratio cycle instances satisfy the hidden stable α-flow property for $\alpha = 1/\Theta(\log m)$.*

Over these min-ratio cycle instances, the total sizes of the updates is $\sum_{t \in \tau} |U^{(t)}| = \widetilde{O}(m\kappa^{-2})$, and the algorithm invokes Update, Query, *and* Detect *$\widetilde{O}(m\kappa^{-2})$ times. Furthermore, it is guaranteed that over all these instances, the total number of edges included in any of the* Detect *outputs is $\widetilde{O}(m\kappa^{-2})$.*

The algorithm runs in time $\widetilde{O}(m\kappa^{-2})$ plus the time taken by the data structure.

The above result can be generalized to arbitrary integer costs and capacities at the cost of a $O(\log C \log mU)$ factor in the running time by cost/capacity scaling [19, Lemma C.1].

In the full paper, we build our new deterministic data structure for approximate dynamic min-ratio cycles with hidden stability and prove the following theorem.

Theorem IV.7 (Dynamic Min-Ratio Cycle with Hidden Stability). *There is a deterministic data structure that κ-approximately solves the problem of dynamic min-ratio cycle with hidden stability for $\kappa = \exp(-O(\log^{17/18} m \cdot \log\log m))$. Over τ batches of updates $U^{(1)}, \ldots, U^{(\tau)}$, the algorithm runs in time $m^{o(1)}(m + \sum_{t \in [\tau]} |U^{(t)}|)$.*

The data structure maintains a spanning tree $T \subseteq G^{(t)}$ and returns a cycle $\boldsymbol{\Delta}$ represented as $m^{o(1)}$ paths on T (specified by their endpoints) and $m^{o(1)}$ explicitly given off-tree edges, and supports Update *and* Query *operations in $m^{o(1)}$ amortized time. The running time of* Detect *is $m^{o(1)}|S^{(t)}|$, where $S^{(t)}$ is the set of edges returned by* Detect.

Acknowledgments

We thank the anonymous reviewers for their helpful comments.

References

[1] A. Abboud, R. Krauthgamer, J. Li, D. Panigrahi, T. Saranurak, and O. Trabelsi. "Breaking the Cubic Barrier for All-Pairs Max-Flow: Gomory-Hu Tree in Nearly Quadratic Time". In: *FOCS*. IEEE, 2022, pp. 884–895.

[2] A. Abboud, R. Krauthgamer, and O. Trabelsi. "Subcubic algorithms for Gomory-Hu tree in unweighted graphs". In: *STOC*. ACM, 2021, pp. 1725–1737.

[3] 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.

[4] K. Ameranis, A. Chen, L. Orecchia, and E. Tani. "Efficient Flow-based Approximation Algorithms for Submodular Hypergraph Partitioning via a Generalized Cut-Matching Game". In: *CoRR* abs/2301.08920 (2023).

[5] S. Apers and R. de Wolf. "Quantum Speedup for Graph Sparsification, Cut Approximation, and Laplacian Solving". In: *SIAM J. Comput.* 51.6 (2022), pp. 1703–1742.

[6] S. Arora and S. Kale. "A combinatorial, primal-dual approach to semidefinite programs". In: *STOC*. ACM, 2007, pp. 227–236.

[7] K. Axiotis, A. Mądry, and A. Vladu. "Circulation control for faster minimum cost flow in unit-capacity graphs". In: *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE. 2020, pp. 93–104.

[8] K. Axiotis, A. Mądry, and A. Vladu. "Faster sparse minimum cost flow by electrical flow localization". In: *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE. 2022, pp. 528–539.

[9] A. Bernstein, M. P. Gutenberg, and T. Saranurak. "Deterministic decremental sssp and approximate min-cost flow in almost-linear time". In: *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE. 2022, pp. 1000–1008.

[10] G. Borradaile and P. Klein. "An O (n log n) algorithm for maximum st-flow in a directed planar graph". In: *Journal of the ACM (JACM)* 56.2 (2009), pp. 1–30.

[11] G. Borradaile, P. N. Klein, S. Mozes, Y. Nussbaum, and C. Wulff-Nilsen. "Multiple-source multiple-sink maximum flow in directed planar graphs in near-linear time". In: *SIAM Journal on Computing* 46.4 (2017), pp. 1280–1303.

[12] Y. Boykov and V. Kolmogorov. "An Experimental Comparison of Min-Cut/Max-Flow Algorithms for Energy Minimization in Vision". In: *IEEE Trans. Pattern Anal. Mach. Intell.* 26.9 (2004). Available at: https://arxiv.org/abs/1202.3367, pp. 1124–1137.

[13] J. van den 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: *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing*. 2022, pp. 543–556.

[14] J. v. d. Brand. "A Deterministic Linear Program Solver in Current Matrix Multiplication Time". In: *SODA*. SIAM, 2020, pp. 259–278.

[15] 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*. ACM, 2021, pp. 859–869.

[16] 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: *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE. 2020, pp. 919–930.

[17] S. Chechik and T. Zhang. "Dynamic low-stretch spanning trees in subpolynomial time". In: *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM. 2020, pp. 463–475.

[18] L. Chen, G. Goranci, M. Henzinger, R. Peng, and T. Saranurak. "Fast dynamic cuts, distances and effective resistances via vertex sparsifiers". In: *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE. 2020, pp. 1135–1146.

[19] L. Chen, R. Kyng, Y. P. Liu, R. Peng, M. Probst Gutenberg, and S. Sachdeva. "Maximum flow and minimum-cost flow in almost-linear time". In: *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*. https://arxiv.org/abs/2203.00671. IEEE. 2022, pp. 612–623.

[20] L. Chen, R. Peng, and D. Wang. "2-norm Flow Diffusion in Near-Linear Time". In: *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*. https://arxiv.org/abs/2105.14629. IEEE. 2022, pp. 540–549.

[21] L. Chen and R. Tell. "Simple and fast derandomization from very hard functions: eliminating randomness at almost no cost". In: *STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing, Virtual Event, Italy, June 21-25, 2021*. Ed. by S. Khuller and V. V. Williams. ACM, 2021, pp. 283–291.

[22] 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: *Proceedings of the 43rd ACM Symposium on Theory of Computing, STOC 2011, San Jose, CA, USA, June 6-8 2011*. Available at https://arxiv.org/abs/1010.2921. ACM, 2011, pp. 273–282.

[23] J. Chuzhoy. "Decremental All-Pairs Shortest Paths in Deterministic near-Linear Time". In: *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*. Available at: https://arxiv.org/abs/2109.05621. New York, NY, USA: Association for Computing Machinery, 2021, pp. 626–639.

[24] J. Chuzhoy, Y. Gao, J. Li, D. Nanongkai, R. Peng, and T. Saranurak. "A deterministic algorithm for balanced cut with applications to dynamic connectivity, flows, and beyond". In: *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE. 2020, pp. 1158–1167.

[25] J. Chuzhoy and T. Saranurak. "Deterministic algorithms for decremental shortest paths via layered core decomposition". In: *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM. 2021, pp. 2478–2496.

[26] M. B. Cohen, A. Madry, P. Sankowski, and A. Vladu. "Negative-Weight Shortest Paths and Unit Capacity Minimum Cost Flow in $\widetilde{O}(m^{10/7} \log W)$ Time (Extended Abstract)". In: *SODA*. SIAM, 2017, pp. 752–771.

[27] S. I. Daitch and D. A. Spielman. "Faster approximate lossy generalized flow via interior point algorithms". In: *Proceedings of the fortieth annual ACM symposium on Theory of computing*. 2008, pp. 451–460.

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

[29] E. Dinic. "Algorithm for solution of a problem of maximum flow in networks with power estimation". In: *Soviet Mathematics Doklady* 11 (1970), pp. 1277–1280.

[30] E. Dinic. "Metod porazryadnogo sokrashcheniya nevyazok i transportnye zadachi". In: *Issledovaniya po Diskretnoĭ Matematike* (1973). In Russian. Title translation: Excess scaling and transportation problems.

[31] R. Duan, S. Pettie, and H. Su. "Scaling Algorithms for Weighted Matching in General Graphs". In: *ACM Trans. Algorithms* 14.1 (2018). Available at: https://arxiv.org/abs/1411.1919, 8:1–8:35.

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

[33] S. Forster and G. Goranci. "Dynamic low-stretch trees via dynamic low-diameter decompositions". In: *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*. 2019, pp. 377–388.

[34] S. Forster, G. Goranci, and M. Henzinger. "Dynamic maintenance of low-stretch probabilistic tree embeddings with applications". In: *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM. 2021, pp. 1226–1245.

[35] Z. Galil and É. Tardos. "An $O(n^2(m+n \log n) \log n)$ Min-Cost Flow Algorithm". In: *J. ACM* 35.2 (1988), pp. 374–386.

[36] Y. Gao, Y. P. Liu, and R. Peng. "Fully dynamic electrical flows: Sparse maxflow faster than goldberg-rao". In: *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE. 2022, pp. 516–527.

[37] M. Ghaffari, A. Karrenbauer, F. Kuhn, C. Lenzen, and B. Patt-Shamir. "Near-Optimal Distributed Maximum Flow". In: *SIAM J. Comput.* 47.6 (2018), pp. 2078–2117.

[38] A. Goldberg and R. Tarjan. "Solving minimum-cost flow problems by successive approximation". In: *Proceedings of the nineteenth annual ACM symposium on Theory of computing*. 1987, pp. 7–18.

[39] A. V. Goldberg. "The partial augment–relabel algorithm for the maximum flow problem". In: *European Symposium on Algorithms*. Springer. 2008, pp. 466–477.

[40] A. V. Goldberg. "Scaling Algorithms for the Shortest Paths Problem". In: *SIAM J. Comput.* 24.3 (1995), pp. 494–504.

[41] A. V. Goldberg, S. Hed, H. Kaplan, P. Kohli, R. E. Tarjan, and R. F. Werneck. "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*. Ed. by N. Bansal and I. Finocchi. Vol. 9294. Lecture Notes in Computer Science. Available at: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/11/ghkktw_ESA2015.pdf. Springer, 2015, pp. 619–630.

[42] A. V. Goldberg and S. Rao. "Beyond the Flow Decomposition Barrier". In: *Journal of the ACM* 45.5 (1998). Announced at FOCS'97, pp. 783–797.

[43] A. V. Goldberg and R. E. Tarjan. "Finding Minimum-Cost Circulations by Canceling Negative Cycles". In: *J. ACM* 36.4 (1989), pp. 873–886.

[44] D. Goldfarb and M. D. Grigoriadis. "A computational comparison of the Dinic and network simplex methods for maximum flow". In: *Annals of Operations Research* 13.1 (1988), pp. 81–123.

[45] R. E. Gomory and T. C. Hu. "Multi-terminal network flows". In: *Journal of the Society for Industrial and Applied Mathematics* 9.4 (1961), pp. 551–570.

[46] D. Gusfield. "Very Simple Methods for All Pairs Network Flow Analysis". In: *SIAM J. Comput.* 19.1 (1990), pp. 143–155.

[47] D. S. Hochbaum. "The Pseudoflow Algorithm: A New Algorithm for the Maximum-Flow Problem". In: *Operations Research* 56.4 (2008), pp. 992–1009.

[48] 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.

[49] R. Impagliazzo and A. Wigderson. "*P = BPP* if *E* Requires Exponential Circuits: Derandomizing the XOR Lemma". In: *Proceedings of the Twenty-Ninth Annual ACM Symposium on the Theory of Computing, El Paso, Texas, USA, May 4-6, 1997*. Ed. by F. T. Leighton and P. W. Shor. ACM, 1997, pp. 220–229.

[50] D. Kang and J. Payor. "Flow Rounding". In: *CoRR* abs/1507.08139 (2015).

[51] N. Karmarkar. "A New Polynomial-Time Algorithm for Linear Programming". In: *STOC*. ACM, 1984, pp. 302–311.

[52] 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.

[53] T. Kathuria, Y. P. Liu, and A. Sidford. "Unit Capacity Maxflow in Almost $O(m^{4/3})$ Time". In: *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*. IEEE, 2020, pp. 119–130.

[54] K. Kawarabayashi and M. Thorup. "Deterministic Edge Connectivity in Near-Linear Time". In: *J. ACM* 66.1 (2019), 4:1–4:50.

[55] 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: *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, Portland, Oregon, USA, January 5-7, 2014*. Ed. by C. Chekuri. SIAM, 2014, pp. 217–226.

[56] R. Khandekar, S. Rao, and U. V. Vazirani. "Graph partitioning using single commodity flows". In: *STOC*. ACM, 2006, pp. 385–390.

[57] I. Koutis, G. L. Miller, and R. Peng. "A nearly-m log n time solver for sdd linear systems". In: *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*. IEEE. 2011, pp. 590–598.

[58] R. Kyng, S. Meierhans, and M. Probst. "Derandomizing Directed Random Walks in Almost-Linear Time". In: *63rd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2022, Denver, CO, USA, October 31 - November 3, 2022*. IEEE, 2022, pp. 407–418.

[59] R. Kyng, R. Peng, S. Sachdeva, and D. Wang. "Flows in Almost Linear Time via Adaptive Preconditioning". In: *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*. 2019, pp. 902–913.

[60] Y. T. Lee and A. Sidford. "Efficient Inverse Maintenance and Faster Algorithms for Linear Programming". In: *56th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, October 17-20, 2015*. Available at https://arxiv.org/abs/1503.01752. IEEE Computer Society, 2015, pp. 230–249.

[61] Y. T. Lee and A. Sidford. "Solving Linear Programs with Sqrt(rank) Linear System Solves". In: *CoRR* abs/1910.08033 (2019). arXiv: 1910.08033.

[62] J. Li. "Deterministic mincut in almost-linear time". In: *STOC*. ACM, 2021, pp. 384–395.

[63] J. Li, D. Nanongkai, D. Panigrahi, T. Saranurak, and S. Yingchareonthawornchai. "Vertex connectivity in poly-logarithmic max-flows". In: *STOC*. ACM, 2021, pp. 317–329.

[64] J. Li and D. Panigrahi. "Deterministic Min-cut in Poly-logarithmic Max-flows". In: *FOCS*. IEEE, 2020, pp. 85–92.

[65] Y. P. Liu and A. Sidford. "Faster energy maximization for faster maximum flow". In: *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*. 2020, pp. 803–814.

[66] A. Mądry. "Computing Maximum Flow with Augmenting Electrical Flows". In: *57th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2016, 9-11 October 2016, Hyatt Regency, New Brunswick, New Jersey, USA*. Available at https://arxiv.org/abs/1608.06016. IEEE Computer Society, 2016, pp. 593–602.

[67] A. Mądry. "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, 2010, pp. 245–254.

[68] A. Mądry. "Navigating central path with electrical flows: From flows to matchings, and back". In: *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*. IEEE. 2013, pp. 253–262.

[69] C. Nalam, T. Saranurak, and S. Yingchareonthawornchai. "Deterministic k-Vertex Connectivity in k^2 Max-flows". In: *arXiv preprint arXiv:2308.04695* (2023). Available at https://arxiv.org/pdf/2308.04695.pdf.

[70] D. Nanongkai, T. Saranurak, and S. Yingchareonthawornchai. "Breaking quadratic time for small vertex connectivity and an approximation scheme". In: *STOC*. ACM, 2019, pp. 241–252.

[71] L. Orecchia, L. J. Schulman, U. V. Vazirani, and N. K. Vishnoi. "On partitioning graphs via single commodity flows". In: *STOC*. ACM, 2008, pp. 461–470.

[72] J. B. Orlin and X.-y. Gong. "A fast maximum flow algorithm". In: *Networks* 77.2 (2021), pp. 287–321.

[73] J. B. Orlin. "A Faster Strongly Polynomial Minimum Cost Flow Algorithm". In: *Oper. Res.* 41.2 (1993), pp. 338–350.

[74] J. B. Orlin. "A Polynomial Time Primal Network Simplex Algorithm for Minimum Cost Flows". In: *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '96. Atlanta, Georgia, USA: Society for Industrial and Applied Mathematics, 1996, pp. 474–481.

[75] J. B. Orlin, S. A. Plotkin, and É. Tardos. "Polynomial Dual Network Simplex Algorithms". In: *Math. Program.* 60.1–3 (1993), pp. 255–276.

[76] J. Sherman. "Area-convexity, ℓ_∞ regularization, and undirected multicommodity flow". In: *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*. 2017, pp. 452–460.

[77] J. Sherman. "Breaking the Multicommodity Flow Barrier for $O(\sqrt{\log n})$-Approximations to Sparsest Cut". In: *FOCS*. IEEE Computer Society, 2009, pp. 363–372.

[78] J. Sherman. "Nearly Maximum Flows in Nearly Linear Time". In: *54th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2013, 26-29 October, 2013, Berkeley, CA, USA*. IEEE Computer Society, 2013, pp. 263–269.

[79] 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.

[80] D. A. Spielman and S. Teng. "Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems". In: *Proceedings of the 36th Annual ACM Symposium on Theory of Computing, STOC 2004, Chicago, IL, USA, June 13-16, 2004*. Available at https://arxiv.org/abs/0809.3232, https://arxiv.org/abs/0808.4134, https://arxiv.org/abs/cs/0607105. 2004, pp. 81–90.

[81] É. Tardos. "A Strongly Polynomial Minimum Cost Circulation Algorithm". In: *Combinatorica* 5.3 (1985), pp. 247–255.