Partially-Dynamic Maximum Flow in Dense Graphs

Egor Kravchenko
ETH Zurich, Switzerland

Maximilian Probst Gutenberg
ETH Zurich, Switzerland

Abstract

We give the first algorithms that, with high probability, maintain $(1-\epsilon)$-approximate s-t maximum flow in an n-vertex undirected, capacitated graph undergoing either only edge insertions or only edge deletions in total update time $\tilde{\mathcal{O}}_\varepsilon(n^2)$.[1] For dense graphs, this yields polylogarithmic amortized update time, which was previously only obtained for the special case of uncapacitated graphs undergoing edge insertions.

We develop the following two algorithms:

- For graphs undergoing deletions, we generalize the congestion-balancing framework from [6], which was developed for maximum matching. We then show that this framework can be simulated on cut sparsifiers, which yields significant speed-ups.
- For graphs undergoing insertions, we show that the sparsification techniques by Eppstein et al. [13] can be combined more directly with the techniques from Henzinger and Goranci [15]. We thereby bypass the need to dynamize the more involved residual graph sparsification approach by Levin and Karger [25] suggested in [17], and extend their result to capacitated graphs.

2012 ACM Subject Classification Theory of computation → Dynamic graph algorithms

Keywords and phrases Maximum Flow, Dynamic Graph Algorithm, Data Structure

Digital Object Identifier 10.4230/LIPIcs.ICALP.2026.133

Category Track A: Algorithms, Complexity and Games

Funding *Maximilian Probst Gutenberg*: The research leading to these results has received funding from grant no. 200021 204787 of the Swiss National Science Foundation.

Acknowledgements The authors would like to thank Simon Meierhans for various insightful discussions.

1 Introduction

Computing the s–t maximum flow is a central problem in network optimization. Given a graph with capacities on its edges, the goal is to compute the largest amount of flow that can be routed from a designated source s to a designated sink t, subject to capacity constraints and flow conservation at all intermediate vertices. Improvements in maximum flow algorithms have a broad impact, since the problem underlies many classic reductions and applications, including maximum bipartite matching, finding a maximum collection of edge-disjoint s–t paths, and a variety of cut and connectivity tasks [27, 26, 8, 2, 1, 21, 20]. Beyond its theoretical role, maximum flow also appears in numerous real-world settings, such as scheduling, image segmentation, and transportation and logistics. Recently, the first algorithm achieving almost-linear running time in the number of edges was obtained in [10, 31].

[1] We use $\tilde{\mathcal{O}}_\varepsilon(X)$ to hide polylogarithmic factors in the size of the input graph and polynomial factors in ϵ, e.g. $f(n) \cdot \log^8(n)/\epsilon^5 \in \tilde{\mathcal{O}}_\varepsilon(f(n))$.

This progress naturally raises the question of maintaining s–t max flow in *dynamic graphs*, where the input graph undergoes edge insertions and deletions. For directed graphs, lower bounds from [3, 22, 12] essentially rule out algorithms to maintain the exact s-t max flow value even in partially-dynamic graphs, where the update sequence only consists of *either* edge insertions or edge deletions. We refer to graphs undergoing only edge insertion as *incremental* and refer to graphs only undergoing edge deletions as *decremental*.

Since exact algorithms even for partially-dynamic are ruled out by conditional lower bounds, research has largely focused on $(1-\varepsilon)$-approximation algorithms. In our discussion, we let n denote the number of vertices, and m denote the maximum number of edges in the graph at any point in time. Note that m serves as an upper bound on the number of updates for partially-dynamic graphs. To avoid clutter, we assume throughout that capacities are polynomially-bounded in n and the number of updates m does not exceed n^2.[2] Table 1 gives a more detailed summary of the landscape of results in the partially-dynamic setting. For a discussion of the literature on the static max flow problem, we refer the reader to [30].

In [32, 11, 30], $(1-\varepsilon)$-approximate partially-dynamic algorithms are given with total update time $\tilde{\mathcal{O}}_\varepsilon(m^{1+o(1)})$ even for directed graphs. In [15, 17], it was shown that for *undirected, uncapacitated* graphs undergoing edge insertions, total update time $\tilde{\mathcal{O}}_\varepsilon(n^2)$ can be achieved, which is near-optimal for dense graphs.

In this paper, we give near-optimal algorithms for maintaining approximate s–t max flow in undirected partially-dynamic graphs with capacities.

▶ **Theorem 1.** *For any partially-dynamic graph G, vertices $s,t \in V$, and $\varepsilon \in (0,1)$, there exists a randomized algorithm with total update time $\tilde{\mathcal{O}}_\varepsilon(n^2)$, that, after each update, outputs a $(1-\varepsilon)$-approximation to the s–t maximum flow in G.*[3]

The algorithm succeeds with high probability and works against an adaptive adversary.

For dense graphs, this implies polylogarithmic amortized time per update, improving over the subpolynomial update time of the state-of-the-art algorithms in [11, 30]. Our bound also matches the polylogarithmic update time achieved in [17], while extending also to graphs that undergo edge deletions and additionally support general capacities.[4]

Our algorithms are simple and combinatorial in nature. This contrasts starkly with [11, 30], which rely on a sophisticated combination of interior-point methods and dynamic data structures. We note that the decremental algorithm in Theorem 1 uses the algorithm from [28], which is based on first-order optimization methods. While the notion of "combinatorial" is not well-defined, this algorithm is typically considered to be combinatorial. However, even when replaced by the classic, unambiguously "combinatorial" algorithm in [14], we still obtain $\tilde{\mathcal{O}}_\varepsilon(n^{5/2})$ total runtime.

We give two separate algorithms, one for decremental graphs and another for incremental graphs:

- **Decremental:** The original paper [6] introduced the congestion-balancing framework and used it to obtain a $\tilde{\mathcal{O}}_\varepsilon(m)$ time algorithm for decremental bipartite matching. The main idea is to find a fractional matching that is well-spread, so that a substantial amount of

[2] Our algorithm, like all previous results, has polylogarithmic dependency on the aspect ratio of the graph.

[3] In graphs undergoing edge insertions, the algorithm maintains an approximate flow explicitly, in graphs undergoing edge deletions, only the approximate max flow value is maintained. Note that [11, 32, 30] all only maintain an approximate flow value, only the algorithms for graphs undergoing insertions in [15, 17] can maintain such an approximate flow.

[4] [17] obtains a bound of $\tilde{\mathcal{O}}_\varepsilon(m+nF^*)$ total update time, which is tighter in simple uncapacitated graphs since there $F^* = O(n)$. We show in Theorem 16 that the same bound can be obtained in capacitated graphs.

the matching can only be removed by deleting either many edges or some very "important" edges. This allows them to show that only few, i.e. $\tilde{\mathcal{O}}_\varepsilon(1)$, fractional matchings have to be computed over the course of the algorithm.

We show how to extend their approach to the more general problem of decremental maximum flow. However, this extension by itself yields only a $\tilde{\mathcal{O}}_\varepsilon(mn)$-time bound, since it can be shown to be essential to recompute up to $\Omega(n)$ fractional flow certificates. To overcome this, we combine congestion balancing with cut sparsification: we maintain a sparse cut sparsifier with $\tilde{\mathcal{O}}_\varepsilon(n)$ edges and run the framework's updates on this sparsified graph. Combined with a simple data structure, this reduces the algorithm's dependence on m to the sparsifier size $\tilde{\mathcal{O}}_\varepsilon(n)$.

- **Incremental:** We build on the idea from [15] to maintain the residual network under further insertions, along with a data structure that checks s-t reachability. Whenever the data structure detects s-t reachability, we augment along a new s-t path. A direct implementation yields $\mathcal{O}(mF \log m)$ total time: each augmentation requires an $\mathcal{O}(m \log m)$ time for reachability and path search, and we perform at most F augmentations. We accelerate this by running the same procedure on a cut-sparsified representation of the residual network with only $\tilde{\mathcal{O}}_\varepsilon(n)$ edges, which replaces the m factor by the sparsifier size and gives $\tilde{\mathcal{O}}_\varepsilon(nF)$ total time; this is sufficient when $F = \mathcal{O}(n)$.

  While it seems at first rather straightforward to simply maintain a cut sparsifier of G and then simulate the above procedure for $O(n)$ edge insertions until re-sparsification is necessary, the fluctuations in the max flow value of the sparsifier graph between re-computations can cause $\Omega(\varepsilon F)$ augmentations for every $O(n)$ edge insertions resulting in a catastrophic runtime.

  We show that the simple hierarchical re-sparsification technique from [13] can be adapted to update the sparsifier of G such that the max flow value monotonically increases over time, thus avoiding any fluctuations. We achieve this simply by scaling up cut sparsifiers slightly and careful composition. While much like in [17], the key to our algorithm is a re-sparsification framework, our approach is arguably much more direct, as we only require sparsification of undirected graphs as opposed to residual networks. In this way, our algorithmic framework can also deal with capacities while retaining comparable simplicity.

  For the regime where the maximum flow value F is large, i.e., $F = \Omega(n)$, we use a standard truncation argument that allows us to discard edges of small residual capacity while preserving a $1 - \varepsilon$ approximation, obtaining a total running time of $\tilde{\mathcal{O}}_\varepsilon(n^2)$.

2 Preliminaries

Logarithms. We denote by $\log x$, the logarithm of x with base 2.

Flows. Our algorithm works on undirected, capacitated graphs $G = (V, E, u)$, where V denotes the set of vertices, E the set of edges, and u the capacity function $E \to \mathbb{Z}_+$. For two vertices $s, t \in V$, an s-t flow $f \in \mathbb{R}^m$ assigns a value $f_e \in \mathbb{R}$ to each edge such that $\sum_{e \sim v} f_e = 0$ for all $v \notin \{s, t\}$ with $|f_e| \leq u(e)$. The value of a flow f is $F(f) = \sum_{e \sim s} f_e$ and a maximum flow $f^* = \operatorname{argmax}_{f \text{ is } s-t \text{ flow}} F(f)$, with value $F^*(G) = F(f^*)$. When the flow is clear from the context, we use F without the argument f. We call a flow f an α-approximate max flow for $\alpha \in (0, 1)$ if $F(f) \geq \alpha F^*$.

Table 1 Results on dynamic maximum flow. We give total update times for handling m updates on a graph with m edges (where $m \leq n^2$).

Setting	Apx. Factor	Directed	Weighted	Update Time	Reference
Incremental	1	Yes	Yes	$\tilde{\mathcal{O}}_\varepsilon(mF^*)$	[23, 19]
	1	No	No	$\tilde{\mathcal{O}}_\varepsilon(n^{2.5})$	[15]
	$1-\varepsilon$	Yes	No	$\tilde{\mathcal{O}}_\varepsilon(m^{1+0.5+o(1)})$	[15]
	$1-\varepsilon$	Yes	Yes	$\tilde{\mathcal{O}}_\varepsilon(mn^{0.5+o(1)})$	[33]
	$1-\varepsilon$	No	Yes	$\tilde{\mathcal{O}}_\varepsilon(m^{1+o(1)})$	[32]
	$1-\varepsilon$	Yes	Yes	$\tilde{\mathcal{O}}_\varepsilon(m^{1+o(1)})$	[11]
	$1-\varepsilon$	No	No	$\tilde{\mathcal{O}}_\varepsilon(n^2)$	[17]
	$1-\varepsilon$	No	Yes	$\tilde{\mathcal{O}}_\varepsilon(n^2)$	**Theorem 1**
Decremental	$1-\varepsilon$	Yes	Yes	$\tilde{\mathcal{O}}_\varepsilon(m^{1+o(1)})$	[30]
	$1-\varepsilon$	No	Yes	$\tilde{\mathcal{O}}_\varepsilon(n^2)$	**Theorem 1**
Fully dynamic	$\mathcal{O}(\log n)$	No	Yes	$mn^{0.667+o(1)}$	[9]
	$\mathcal{O}(\text{polylog } n)$	No	Yes	$m^\delta, \forall \delta > 0$	[16]
	$m^{o(1)}$	No	No	$m^{1+o(1)}$	[18]
	$m^{o(1)}$	No	Yes	$m^{1+o(1)}$	[30]

Cuts. For any cut $(S, \bar{S} = V \setminus S)$, we denote by $E(S, \bar{S})$ the set of edges crossing the cut and by $u(E(S, \bar{S})$ the capacities of the edges in the cut. For distinct vertices s, t, we say $(S, \bar{S})$ is an s-t cut if $s \in S, t \in \bar{S}$. In this paper, we extensively (and sometimes implicitly) use the max-flow min-cut theorem, which states that the maximum s-t flow value is equal to the capacity of the minimum s-t cut.

Cut Sparsifier. We use the following definition of cut sparsifiers. This definition is slightly non-standard but obtained straightforwardly from classic constructions.

▶ **Definition 2.** *Let $G = (V, E, u)$ be a capacitated graph and given any $\varepsilon > 0$. We call a graph $H = (V, E', u')$ an $(1-\varepsilon)$ cut sparsifier for G if for every cut $(S, \bar{S})$, $(1-\varepsilon)u'(E_H(S, \bar{S})) \leq u(E_G(S, \bar{S})) \leq u'(E_H(S, \bar{S}))$.*

3 Approximated Decremental Max Flow

In this section, we give our main technical contribution: a simple algorithm to maintain max flow in a graph undergoing edge deletions.

▶ **Theorem 3** (Decremental Max Flow). *For any decremental graph G, vertices $s, t \in V$, and $\varepsilon \in (0, 1)$, there exists a randomized algorithm with total update time $\tilde{\mathcal{O}}_\varepsilon\left(n^2\right)$, that, after each update, outputs a $(1-\varepsilon)$-approximation to the s–t maximum flow in G.*

The algorithm succeeds with high probability and works against an adaptive adversary.

3.1 Thresholded Approximated Decremental Max Flow

We first observe that it essentially suffices to solve a thresholded version of the decremental max flow problem.

▶ **Lemma 4.** *For any decremental graph G, threshold F and $\varepsilon \in (0,1)$, there exists a randomized algorithm* Decremental-Thresholded-Flow(G, F, ε) *with total update time $\tilde{\mathcal{O}}_\varepsilon\left(n^2+m\right)$, that, after each update, certifies that either*
1. *there is a flow f in G with $F(f) \geq (1-O(\varepsilon))F$, or*
2. *$F^*(G) \leq (1+\varepsilon)F$.*

The algorithm succeeds with high probability and works against an adaptive adversary.

Only $\tilde{\mathcal{O}}_\varepsilon(1)$ instantiations of Decremental-Thresholded-Flow(G, F, ε) suffice to implement Theorem 3 via standard techniques (see for example [33]).

We therefore focus in this section on proving Lemma 4. To this end, we extend the *congestion-balancing flow framework* introduced in [6]. [6] uses it to solve the approximate decremental matching problem in time $\tilde{\mathcal{O}}_\varepsilon(m)$. We apply the same framework to the more general decremental maximum flow problem and combine it with a cut-sparsification technique. The congestion-balancing flow framework bounds the number of re-computations of the flow by $\tilde{O}(n)$. Each round requires computing a static max flow of value at least $(1-\epsilon)F$ or returns a cut of capacity less than F. We use the algorithm Maxflow(G, F) from [29, 28] which runs in $\tilde{O}(m)$ time. However, this would result in overall time $\tilde{\mathcal{O}}_\varepsilon(mn)$. To improve runtime, we show that we only need to compute flows on cut sparsifiers, which reduces the runtime of each iteration to $\tilde{\mathcal{O}}_\varepsilon(n^2)$, which yields the desired runtime.

We use the theorem below, which is implicit in [7], and maintains a partition of the edges of a fully-dynamic graph into $O(\log n)$ expander decompositions of almost uniform degree. For such uniform degree expanders, it is well-known that uniform down-sampling yields $(1-\varepsilon)$ cut sparsifiers. The sparsifiers can be sampled implicitly at cost of $\tilde{\mathcal{O}}_\varepsilon(1)$ per edge that is added to the sparsifier. The overall sparsifier has $\tilde{\mathcal{O}}_\varepsilon(n)$ edges.

▶ **Theorem 5** (see [7]). *There exists a randomized data structure* Sparsifier *which takes a fully-dynamic graph $G=(V,E,u)$ and $\varepsilon > 0$, and processes updates and returns an $(1-\varepsilon)$ cut sparsifier H of G on query at any time. The data structure takes time $\tilde{\mathcal{O}}_\varepsilon(|E(G)|)$ on initialization, processes each update with amortized time $\tilde{\mathcal{O}}_\varepsilon(1)$, and queries in time $\tilde{\mathcal{O}}_\varepsilon(n)$. The algorithm succeeds with high probability and works against an adaptive adversary.*

The Algorithm. We are now ready to describe our high-level algorithm (see Algorithm 1). We assume throughout w.l.o.g. that n is a power of 2 and so are all capacities [5]. Our algorithm proceeds in phases. Each edge e is assigned a capacity $\kappa(e) \leq u(e)$, which captures the importance of the edge. We use the function κ to scale capacities in G down as much as possible, so that a maxflow of value $(1-O(\varepsilon))F$ can still be routed. This results in the corresponding flow being spread out as much as possible, which makes it harder for the adversary to delete it. Initially, we set $\kappa(e) = \frac{1}{n}$ for every edge e. Later, we repeatedly double the capacities of some edges. Importantly, we never double the scaled capacity κ if the original capacity was achieved, i.e. at all time, for all $e \in E$, we have $\kappa(e) \leq u(e)$. Throughout the algorithm, we maintain two sparsifiers: the graph $H_{original}$, which is a cut sparsifier of G, and the graph H_{scaled}, which is a cut sparsifier of G with scaled capacities κ. To this end, we use the data structure from Theorem 5 with accuracy ε. As G and κ undergo updates, so do $H_{original}$ and H_{scaled}.

At the beginning of each phase, we run Maxflow$(H_{original}, F)$ to ensure that $F^*(G) \geq (1-O(\varepsilon))F$; if this is not true, we terminate. Otherwise, we call Maxflow(H_{scaled}, F) until it returns a flow. If the procedure returns a cut $(S, \bar{S})$, we double capacities $\kappa(e)$ for all edges

[5] If edge e has capacity $u(e)$ with binary representation $u(e) = 2^{r_1} + 2^{r_2} + \cdots + 2^{r_p}$, we can split it into at most $\log u(e) = \mathcal{O}(\log n)$ edges with capacities $2^{r_1}, 2^{r_2}, \ldots, 2^{r_p}$.

Algorithm 1 DECREMENTAL-THRESHOLDED-FLOW(G, F, ε).

```
1: Procedure Init
2:     Initialise κ(e) = 1/n for all edges e ∈ E.
3:     Maintain H_original to be a (1 − ε) cut sparsifier of G via Theorem 5.
4:     Maintain H_scaled to be a (1 − ε) cut sparsifier of G with capacities κ via Theorem 5.
5:     Set Flow-Deleted to 0.
6:     BeginNewPhase()
7: end Procedure
8: Procedure BeginNewPhase
9:     if Maxflow(H_original, F) returns cut (S, S̄) then
10:        Terminate
11:    end if
12:    while Maxflow(H_scaled, (1 − 3ε)F) returns a cut (S, S̄) do
13:        for each edge e ∈ E_G(S, S̄) do
14:            if κ(e) < 1 then
15:                κ(e) ← 2κ(e)                          ▷ Updates H_scaled.
16:            end if
17:        end for
18:    end while
19:    Set Flow-Deleted to 0.
20: end Procedure
21: Procedure Process deletion of edge e
22:    Remove edge e from G, H_original, H_scaled.
23:    Increase Flow-Deleted by κ(e).
24:    if Flow-Deleted ≥ εF then
25:        BeginNewPhase()
26:    end if
27: end Procedure
```

$e \in E_G(S, \bar{S})$ in the cut with $\kappa(e) < u(e)$ (and therefore $\kappa(e) \leq u(e)/2$). Once this process terminates, the algorithm finds a witness flow f to certify that the maximum flow is of value at least $(1 - O(\varepsilon))F$. Then, we wait until the adversary deletes edges that carry at least εF flow in f. Once this threshold is reached, we start a new phase.

Analysis of Congestion-Balancing. In our analysis, we assume that the algorithms that maintain $H_{original}$ and H_{scaled} succeed, i.e., the two graphs are $(1 - \varepsilon)$ cut sparsifiers of the original graphs G, and the graph G with capacities κ at any time.

We start with the following simple observation.

▶ **Observation 6.** *Throughout Algorithm 1, the function κ only changes via doubling in line 15, and we always have $\frac{1}{n} \leq \kappa(e) \leq u(e)$.*

For our analysis, we introduce the following potential function.

▶ **Definition 7** (Potential function)**.** *Let $\mathcal{F}$ be the collection of all s-t flows f in the current graph G for which $F(f) \geq (1 - 2\varepsilon)F$. Define the **cost** of an edge e by*

$$c(e) = \log\big(n \cdot \kappa(e)\big),$$

which is always nonnegative. For any (possibly fractional) s-t flow f, define its cost by

$$c(f) = \sum_{e \in E(G)} |f(e)| \cdot c(e).$$

Define the potential of the graph as

$$\Pi(G, \kappa, F) = \min_{f \in \mathcal{F}} c(f).$$

If $\mathcal{F}$ is empty, then we set $\Pi(G, \kappa, F) = +\infty$.

Again, we make some simple observations about the potential.

▶ **Observation 8.** *If $\kappa(e)$ increases for some edge e, then $\Pi(G, \kappa, F)$ cannot decrease as a result. Moreover, edge deletions do not decrease $\Pi(G, \kappa, F)$.*

▶ **Observation 9.** *At the beginning of Algorithm 1, we have $c(e) = 0$ for every edge $e \in E(G)$, so $\Pi(G, \kappa, F) = 0$. Moreover, throughout the algorithm, the potential can only increase, and once it becomes $+\infty$, it will remain $+\infty$ forever.*

We have established that the potential starts at 0 and only increases. We now show that as long as the algorithm does not terminate, it is never too large.

▶ **Proposition 10.** *Let $\Pi(G, \kappa, F) < +\infty$. Then $\Pi(G, \kappa, F) = \mathcal{O}(nF \log n)$.*

Proof. Note that for every flow $f \in \mathcal{F}$ we have

$$c(f) = \sum_{e \in E(G)} c(e) \cdot |f(e)| = \sum_{e \in E(G)} \log(n\kappa(e)) \cdot |f(e)| \leq O(\log(n)) \cdot \sum_{e \in E(G)} |f(e)|$$

where we use $\kappa(e) \leq u(e)$ and that all capacities are polynomially-bounded in n. Consider next that f is acyclic. Then, we can decompose the flow into flow paths. Clearly, each unit of flow contributes to at most $n - 1$ edges at cost at most $O(\log n)$, and thus the cost is bounded from above by $O(F \cdot n \log n)$. Since we can remove cycles from any flow $f \in \mathcal{F}$ without decreasing its value and while monotonically removing its cost, we have that some minimizer for $\Pi(G, \kappa, F)$ is acyclic. This yields the proposition. ◀

Next, we bound the increase in potential and total cost of all edges with each scaling κ.

▶ **Definition 11.** *Let E_0 be the edge set of the initial graph G. We define $\kappa(E_0) = \sum_{e \in E_0} \kappa(e)$. If $e \in E_0$ was deleted, then $\kappa(e)$ refers to the version right before the deletion of edge e.*

▶ **Lemma 12.** *Consider a call of MaxFlow on H_{scaled} in Line 12 that returns a cut $(S, \bar{S})$. Let κ be the capacities before the doubling step in line 15, and let κ' be the capacities after this step. Then the following two statements hold simultaneously:*
1. $\kappa'(E_0) \leq \kappa(E_0) + F$,
2. $\Pi(G, \kappa', F) \geq \Pi(G, \kappa, F) + \varepsilon F$.

Proof. When a cut $(S, \bar{S})$ is found in Line 12, we have $\kappa(E_{H_{scaled}}(S, \bar{S})) < (1 - 3\varepsilon)F$ by definition. Since H_{scaled} is a $(1 - \varepsilon)$ cut sparsifier of G with capacities κ, we have $\kappa(E_G(S, \bar{S})) < (1 - 3\varepsilon)F < F$. Since each edge in $e \in E_G(S, \bar{S})$ has $\kappa(e)$ either doubled (if $\kappa(e) \leq u(e)/2$) or unchanged (if $\kappa(e) = u(e)$), we thus have that the increase in κ is at most F overall.

We prove the second claim in two steps:

- **$\Pi(G, \kappa, F)$ is finite:** In Line 9, the algorithm returned proving that for $H_{original}$, we have that all s-t cuts $(S, \bar{S})$ have capacity at least $(1-\varepsilon)F$. Since $H_{original}$ is a $(1-\varepsilon)$ cut sparsifier of G, we thus have

  $$u(E_G(S, \bar{S})) \geq (1-\varepsilon)(1-\varepsilon)F \geq (1-2\varepsilon)F.$$

  Using the max-flow min-cut theorem and Definition 7, $\mathcal{F}$ is non-empty and $\Pi(G, \kappa, F)$ finite.
- **Every flow in $\mathcal{F}$ has increased cost:** consider any flow $f \in \mathcal{F}$. Let E_{full} be the set of all edges $e \in E_G(S, \bar{S})$ such that $\kappa(e) = u(e)$, and let $E_{\text{double}} = E_G(S, \bar{S}) \setminus E_{\text{full}}$, i.e. the edge for which $\kappa'(e) = 2 \cdot \kappa(e)$.
  Since the cut was outputted in Line 12, we have that it has capacity in H_{scaled} at most $(1-4\varepsilon)F$. Since H_{scaled} is a $(1-\varepsilon)$ cut sparsifier of G with capacities κ, we have

  $$\kappa(E_G(S, \bar{S})) \leq (1-3\varepsilon)F.$$

  Thus, $E_{\text{full}} \subseteq E_G(S, \bar{S})$ have capacity at most $(1-3\varepsilon)F$ under κ and since they have $\kappa(e) = u(e)$, we have indeed that $u(E_{\text{full}}) \leq (1-3\varepsilon)F$. Since flow f sends at least $(1-2\varepsilon)F$ units of flow, at least εF units of flow are sent along edges in E_{double}. Since $\sum_{e \in E_{\text{double}}} |f(e)| \geq \varepsilon F$, we have

  $$c'(f) - c(f) = \sum_{e \in E_{\text{double}}} \Big(\log(2n\kappa(e)) - \log(n\kappa(e))\Big) \cdot |f(e)| = \sum_{e \in E_{\text{double}}} |f(e)| \geq \varepsilon F.$$

  Therefore, $\Pi(G, \kappa', F) \geq \Pi(G, \kappa, F) + \varepsilon F$. ◀

▶ **Corollary 13.** *In any execution of Algorithm 1, the total number of times that the invocation of* Maxflow *in line 12 returns a cut is $\tilde{\mathcal{O}}_\varepsilon(n)$. Moreover, we always have $\kappa(E_0) = \tilde{\mathcal{O}}_\varepsilon(nF)$.*

Proof. By Lemma 12, every time Maxflow returns a cut, $\Pi(G, \kappa, F)$ increases by at least εF. By Proposition 10, this quantity is always $\tilde{\mathcal{O}}_\varepsilon(nF)$, so the number of such increases is at most $\tilde{\mathcal{O}}_\varepsilon(n/\varepsilon)$. By Lemma 12, every call of Maxflow that returns a cut incurs increases in $\kappa(E_0)$ by at most F, so in the end we have $\kappa(E_0) = \tilde{\mathcal{O}}_\varepsilon(nF)$. ◀

▶ **Corollary 14.** *The total number of phases before Algorithm 1 terminates is $\tilde{\mathcal{O}}_\varepsilon(n)$.*

Proof. Denote the set of all deleted edges by E_{del} and consider the quantity $\kappa(E_{\text{del}}) = \sum_{e \in E_{\text{del}}} \kappa(e)$. We begin a new phase only when the counter Flow-Deleted exceeds εF, so the total number of phases is bounded by $\kappa(E_{\text{del}})/(\varepsilon F)$. Using $\kappa(E_{\text{del}}) \leq \kappa(E_0)$, that fact that initially $\kappa(E_0)$ is $|E_0| \cdot \frac{1}{n} \leq O(n)$ and Corollary 13, we obtain

$$\frac{\kappa(E_{\text{del}})}{\varepsilon F} \leq \frac{\tilde{\mathcal{O}}_\varepsilon(nF)}{\varepsilon F} = \tilde{\mathcal{O}}_\varepsilon(n)\,.$$ ◀

Runtime Analysis. It remains to analyze the runtime. Before we delve into the full analysis, we still need to clarify an implementation detail: when the Maxflow algorithm is executed in Line 12 terminates with a cut $(S, \bar{S})$, it subsequently requires enumerating the edges in $E_G(S, \bar{S})$. Notably, it finds the cut in the graph H_{scaled}. We use the following fact.

▷ Claim 15. Given a fully-dynamic graph G, at anytime, given a cut $(S, \bar{S})$, it can output the edges $E_G(S, \bar{S})$. The algorithm is deterministic, requires $\tilde{\mathcal{O}}_\varepsilon(|E(G)|)$ initialization time, $\tilde{\mathcal{O}}_\varepsilon(1)$ update time and time $\tilde{\mathcal{O}}_\varepsilon(n + |E_G(S, \bar{S})|)$ to report all edges in $E_G(S, \bar{S})$ given the vertices in $(S, \bar{S})$.

Proof Sketch. We use a dynamic connectivity data structure as given in [24]. Such data structures maintain a maximal spanning forest F of the current graph, which can be stored efficiently in a dynamic tree data structure such as [4].

Consider now any vertices $u \in S, v \in \bar{S}$ and the u-v path $F[u,v]$ along F. Via the dynamic tree data structure, one can efficiently query the midpoint x on $F[u,v]$ (ties broken arbitrarily). Since we can check efficiently if $x \in S$, we set $u = x$; otherwise, we set $v = x$. Then, we recurse again on $F[u,v]$. It is not hard to verify that after $O(\log n)$ such steps, (u,v) is an edge in $E_G(S,\bar{S})$.

To enumerate the cut, we use the above process, then delete the edge (u,v) from G. This yields for an adapted forest F that does not use edge (u,v) and thus yields a new edge upon repeating this process. Upon finding all edges in $E_G(S,\bar{S})$ via this process, we simply add the deleted edges back into the connectivity data structure. The tree data structure can also be used to remove vertices $u \in S$ that are not connected to any vertex in $\bar{S}$ or otherwise return a vertex $v \in \bar{S}$ in the same connected component.

Since the connectivity data structure and dynamic tree data structure support all of the above operations in $\tilde{\mathcal{O}}_\varepsilon(1)$ time, the claimed runtime is achieved. ◁

Finally, we are ready to prove Lemma 4.

Proof. We argue correctness first. Algorithm 1 terminates only if MAXFLOW returns a cut in Line 9. This means that $F^*(H_{\text{original}}) < F$. Since $F^*(G) \leq F^*(H_{\text{original}})$, we get $F^*(G) < F$.

On the other hand, the initialization of a new phase terminates after the MAXFLOW procedure in Line 12 certifies that an s-t flow of value at least $(1-\varepsilon)(1-3\varepsilon)F \geq (1-4\varepsilon)F$ can be routed in H_{scaled} and thus at least $(1-\varepsilon)(1-4\varepsilon)F \geq (1-5\varepsilon)F$ flow can be routed in G with capacities κ.

Since throughout a phase, edges of total scaled capacity κ at most εF are deleted, we have that at least $(1-5\varepsilon)F - \varepsilon F = (1-6\varepsilon)F$ flow remaining, as desired.

Finally, we argue about the time complexity. In particular, we need to bound the cost of max-flow calls and the subroutine for enumerating cut edges. All other costs are subsumed by these bounds. To this end, we observe that any edge that was enumerated in a cut has its capacity thereafter doubled, and thus appears at most $O(\log n)$ times in such cuts. Since by Corollaries 14 and 13, the number of MAXFLOW calls and cut enumerations is $\tilde{\mathcal{O}}_\varepsilon(n)$, and since MAXFLOW calls run in time linear in the number of edges of the cut sparsifiers that they run on, we obtain total runtime $\tilde{\mathcal{O}}_\varepsilon(n^2)$. ◀

4 Incremental Max Flow

This section is devoted to the Incremental Max Flow problem. We first present an algorithm with total update time $\tilde{\mathcal{O}}_\varepsilon(m + nF^*)$, and then show how to obtain $\tilde{\mathcal{O}}_\varepsilon(m+n^2)$ time via a simple trick.

▶ **Theorem 16** (Incremental Max Flow). *For any incremental graph G, vertices $s,t \in V$, and $\varepsilon \in (0,1)$, there exists a randomized algorithm with total update time $\tilde{\mathcal{O}}_\varepsilon\left(m + \min\{nF^*, n^2\}\right)$, that, after each update, outputs a $(1-\varepsilon)$-approximation to the s–t maximum flow in G. The algorithm succeeds with high probability and works against an adaptive adversary.*

On a high level, we partition the update sequence into $\lceil m/n \rceil$ phases, each consisting of at most n edge insertions. At the beginning of each phase, we invoke the procedure SPARSIFY, which computes a $(1-\varepsilon)$ cut sparsifier of the input graph with $\tilde{\mathcal{O}}_\varepsilon(n)$ edges. Such a procedure is a standard primitive and can be computed in near-linear time [5]. We then initialize the residual network with respect to this flow, and for the rest of the phase, we

forward newly inserted edges directly to the residual network. After every insertion, we test whether t is reachable from s in the residual network; if so, we find an s–t augmenting path and augment along it, updating the residual network accordingly.

A key difficulty is that we cannot treat cut sparsification as a black box: a naive approach that (re)computes a sparsifier and then runs a static max-flow algorithm from scratch at the start of every phase yields too large a total running time. Instead, we maintain the sparsifier together with auxiliary state that supports the required flow/residual updates efficiently.

4.1 Cut Sparsifier Segment Tree

In this section, we develop a data structure that efficiently computes cut sparsifiers after edge insertions and maintains a maximum flow on them. Consider the following problem.

We are given a sequence of edge sets $E_0, \dots, E_{k-1}$, each on the same vertex set V and each of size $\mathcal{O}(n)$. After observing each set E_i, we want to return a $(1-\varepsilon)$ cut sparsifier of the graph with edges $E_0 \cup E_1 \cup \cdots \cup E_i$ and a maximum flow on this sparsifier in total time $\tilde{\mathcal{O}}_\varepsilon(n(F^* + k))$[6], where F^* is the maximum flow value in the graph on edges $E_0 \cup \cdots \cup E_{k-1}$.

We develop a data structure Sparsifier-Tree to solve this problem. We build a complete binary segment tree over the time indices $[0, k)$ (assume k is a power of two; otherwise pad with empty edge sets). Each node u corresponds to a half-open interval $I(u) = [a, b)$, with children representing the left and right halves. For every interval $I = [a, b)$ we define

$$E_I := \bigcup_{t=a}^{b-1} E_t \qquad \text{and} \qquad G_I := (V, E_I).$$

Invariant. For every node with interval $I = [a, b)$ we store a $(1-\varepsilon_0)$ cut sparsifier H_I of G_I.

Update. Upon receiving E_i, we set the leaf structure for $[i, i+1)$ (e.g. $H_{[i,i+1)} := (V, E_i)$), and then update the nodes on the path to the root. For a parent interval $I = I_L \cup I_R$ we set

$$H_I \leftarrow \textsc{Sparsify}\big(H_{I_L} \cup H_{I_R}\big),$$

where $\cup$ denotes weighted union (we assume that weights add on duplicates).

▶ **Proposition 17.** *Let G_1 and G_2 be graphs on the same vertex set V, and let H_1 and H_2 be their $(1-\varepsilon)$ cut sparsifiers. Then the graph $H := H_1 \cup H_2$ is a $(1-\varepsilon)$ cut sparsifier of $G := G_1 \cup G_2$ (with weights added on duplicates).*

Proof. Fix a cut $S \subseteq V$. By assumption, for each $i \in \{1, 2\}$ we have

$$(1-\varepsilon)\, u'(E_{H_i}(S, \bar{S})) \le u(E_{G_i}(S, \bar{S})) \le u'(E_{H_i}(S, \bar{S})).$$

Adding the two inequalities and using $u(E_G(S, \bar{S})) = u(E_{G_1}(S, \bar{S})) + u(E_{G_2}(S, \bar{S}))$ and $u'(E_H(S, \bar{S})) = u'(E_{H_1}(S, \bar{S})) + u'(E_{H_2}(S, \bar{S}))$, we obtain

$$(1-\varepsilon)\, u'(E_H(S, \bar{S})) \le u(E_G(S, \bar{S})) \le u'(E_H(S, \bar{S})). \qquad \blacktriangleleft$$

[6] We assume that $\log k = \mathcal{O}(\log n)$.

Now set $\varepsilon_0 = \Theta(\varepsilon/\log k)$. Since each $H_{[a,b)}$ is produced by at most $\log k$ applications of Sparsify along a root-to-leaf path, the total distortion accumulates multiplicatively: for every cut S,

$$(1-\varepsilon_0)^{\mathcal{O}(\log k)}\, u'(E_{H_{[a,b)}}(S,\bar{S})) \;\le\; u(E_{G_{[a,b)}}(S,\bar{S})) \;\le\; u'(E_{H_{[a,b)}}(S,\bar{S})).$$

Choosing the hidden constant in $\Theta(\cdot)$ appropriately yields

$$(1-\varepsilon)\, u'(E_{H_{[a,b)}}(S,\bar{S})) \le u(E_{G_{[a,b)}}(S,\bar{S})) \le u'(E_{H_{[a,b)}}(S,\bar{S})).$$

Together with Proposition 17, this leads to the following result.

▶ **Proposition 18.** *For every tree node with interval $[a,b)$, $H_{[a,b)}$ is a $(1-\varepsilon)$ cut sparsifier of the graph with edges $E_{[a,b)}$.*

The time complexity of maintaining the segment tree is straightforward. The tree has k leaves and at most $2k-1$ nodes. By the invariant, each stored sparsifier has $\tilde{\mathcal{O}}_\varepsilon(n)$ edges, so each merge

$$H_I \leftarrow \textsc{Sparsify}(H_{I_L} \cup H_{I_R})$$

takes $\tilde{\mathcal{O}}_\varepsilon(n)$ time. Each update triggers $\mathcal{O}(\log k)$ merges along the path to the root, hence the total time over k updates is $\mathcal{O}(nk\log k) = \tilde{\mathcal{O}}_\varepsilon(nk)$.

▶ **Proposition 19.** *The total runtime needed to maintain Sparsifier-Tree over k updates is $\tilde{\mathcal{O}}_\varepsilon(nk)$.*

Now consider any interval $[a,b)$. It can be decomposed (in the standard way) into a disjoint union of at most $\log k$ intervals $I_1,\dots,I_s$ associated with tree vertices. Then we let $H_{[a,b)} = \bigcup_{j=1}^{s} H_{I_j}$ (with weighted union). Finally, when we are asked for a sparsifier for the graph $G_{[0,a)}$ for some a, we return the graph $H_{[0,a)}$, which is a $(1-\varepsilon)$ cut sparsifier of $G_{[0,a)}$ by Propositions 17 and 18.

Our data structure also needs to maintain a maximum flow for every prefix sparsifier $H_{[0,a)}$. To do so, for every tree node with interval $[a,b)$ we additionally store a flow $f_{[a,b)}$.

Invariant 1 (prefix optimality). For every prefix $[0,a)$ with canonical decomposition

$$[0,a) \;=\; \bigcup_{i=1}^{s} I_i \qquad (s \le \log k),$$

define the flow $f_{[0,a)}$ by

$$f_{[0,a)} \;:=\; \bigcup_{i=1}^{s} f_{I_i}.$$

Then $f_{[0,a)}$ is a maximum s–t flow in the graph $H_{[0,a)}$.

Invariant 2 (left-sibling augmentation). Let $[a,b)$ be a node that is the left child of its parent, and let $[0,a)$ be the prefix interval corresponding to all time indices strictly before a. Denote by $R_{[0,a)}$ the residual network of $H_{[0,a)}$ with respect to the flow $f_{[0,a)}$. Then the stored flow $f_{[a,b)}$ is a maximum s–t flow in the graph $R_{[0,a)} \cup H_{[a,b)}$.

Update. Upon receiving E_b (thus initializing the leaf $[b, b+1)$), we update the flows for nodes affected by the segment-tree update. In particular, for every node of the form $[a, b+1)$ that is a left child, we compute a maximum flow in the graph

$$R_{[0,a)} \cup H_{[a,b+1)},$$

where $R_{[0,a)}$ is the residual network of $H_{[0,a)}$ with respect to $f_{[0,a)}$. Together with the flow $f_{[0,a)}$, this yields a maximum flow in

$$H_{[0,a)} \cup H_{[a,b+1)} \;=\; H_{[0,b+1)}.$$

To compute each maximum flow $f_{[a,b)}$ we use the Ford–Fulkerson algorithm on the graph $R_{[0,a)} \cup H_{[a,b)}$.

Time Complexity. Let us next bound the runtime of this procedure.

▶ **Lemma 20.** *The total runtime of computing the flows $f_{[a,b)}$ over all tree nodes $[a, b)$ is $\tilde{\mathcal{O}}_\varepsilon(nF^*)$.*

Proof. Fix a node $[a, b)$. The graph on which we compute $f_{[a,b)}$ is of the form $R_{[0,a)} \cup H_{[a,b)}$. By construction, every sparsifier stored in the tree has $\tilde{\mathcal{O}}_\varepsilon(n)$ edges, and the residual network adds at most a $\log k$ factor more edges. Hence $R_{[0,a)} \cup H_{[a,b)}$ has $\tilde{\mathcal{O}}_\varepsilon(n)$ edges.

We compute $f_{[a,b)}$ using the Ford–Fulkerson algorithm. Under our capacity model, each augmentation increases the flow value by at least one unit, and each augmentation can be found in time linear in the number of edges. Therefore, the time to compute $f_{[a,b)}$ is

$$\tilde{\mathcal{O}}_\varepsilon\big(n \cdot \mathrm{val}(f_{[a,b)})\big).$$

Thus it suffices to bound $\sum_{[a,b)} \mathrm{val}(f_{[a,b)})$ over all nodes in the tree.

We prove the following stronger statement by induction on the interval length $(b-a)$. Let

$$\Sigma([a, b)) \;:=\; \sum_{[c,d) \text{ in the subtree of } [a,b)} \mathrm{val}(f_{[c,d)}).$$

Then

$$\Sigma([a, b)) \;\le\; \big(\log(b-a)+1\big) \cdot F^*\big(R_{[0,a)} \cup H_{[a,b)}\big)\,.$$

Base case. $b - a = 1$. Then the subtree contains only the node $[a, a+1)$, and by definition

$$\Sigma([a, a+1)) = \mathrm{val}(f_{[a,a+1)}) = F^*\big(R_{[0,a)} \cup H_{[a,a+1)}\big)\,.$$

Induction step. Let $m = (a+b)/2$, and let $L = [a, m)$ and $R = [m, b)$ be the left and right children. By the induction hypothesis applied to L and R,

$$\Sigma(L) \le \big(\log(m-a)+1\big) \cdot F^*\big(R_{[0,a)} \cup H_L\big)\,,$$
$$\Sigma(R) \le \big(\log(b-m)+1\big) \cdot F^*\big(R_{[0,m)} \cup H_R\big)\,.$$

Since $m - a = b - m = (b-a)/2$, we have $\log(m-a)+1 \le \log(b-a)$ and $\log(b-m)+1 \le \log(b-a)$.

Moreover, by definition,

$$\mathrm{val}(f_{[a,b)}) = F^*\big(R_{[0,a)} \cup H_{[a,b)}\big)\,.$$

Finally, since $R_{[0,m)}$ is the residual network after sending a maximum flow in $H_{[0,m)} = H_{[0,a)} \cup H_L$, the additional maximum flow obtainable after adding H_R satisfies

$$F^*\big(R_{[0,m)} \cup H_R\big) = F^*\big(R_{[0,a)} \cup H_L \cup H_R\big) - F^*\big(R_{[0,a)} \cup H_L\big)\,.$$

Combining these bounds, we obtain

$$\begin{aligned}\Sigma([a,b)) &= \Sigma(L) + \Sigma(R) + \mathrm{val}(f_{[a,b)}) \\ &\le \log(b-a)\cdot F^*\big(R_{[0,a)} \cup H_L\big) \\ &\qquad + \log(b-a)\cdot\Big(F^*\big(R_{[0,a)} \cup H_L \cup H_R\big) - F^*\big(R_{[0,a)} \cup H_L\big)\Big) \\ &\qquad + F^*\big(R_{[0,a)} \cup H_{[a,b)}\big) \\ &= \log(b-a)\cdot F^*\big(R_{[0,a)} \cup H_L \cup H_R\big) + F^*\big(R_{[0,a)} \cup H_{[a,b)}\big)\,.\end{aligned}$$

Since $H_{[a,b)}$ is a cut sparsifier for $H_L \cup H_R$, every s–t cut in $R_{[0,a)} \cup (H_L \cup H_R)$ has capacity at most the corresponding cut in $R_{[0,a)} \cup H_{[a,b)}$. Hence

$$F^*\big(R_{[0,a)} \cup H_L \cup H_R\big) \le F^*\big(R_{[0,a)} \cup H_{[a,b)}\big)\,,$$

and therefore

$$\Sigma([a,b)) \le \big(\log(b-a)+1\big)\cdot F^*\big(R_{[0,a)} \cup H_{[a,b)}\big)\,,$$

which completes the induction.

Applying the bound to the root interval $[0,k)$ (where $R_{[0,0)}$ is empty) yields

$$\sum_{[a,b)} \mathrm{val}(f_{[a,b)}) \le (\log k + 1)\cdot F^*(H_{[0,k)}).$$

Since $H_{[0,k)}$ is a $(1-\varepsilon)$ cut sparsifier of the final graph, its maximum flow value satisfies $F^*(H_{[0,k)}) = \mathcal{O}(F^*)$. Using $\log k = \mathcal{O}(\log n)$ and that such polylogarithmic factors are hidden in $\tilde{\mathcal{O}}_\varepsilon(\cdot)$, we conclude that the total time is $\tilde{\mathcal{O}}_\varepsilon(n \cdot \sum_{[a,b)} \mathrm{val}(f_{[a,b)})) = \tilde{\mathcal{O}}_\varepsilon(nF^*)$. ◀

4.2 $\tilde{\mathcal{O}}_\varepsilon(m + nF^*)$ Incremental Algorithm

By a standard trick, it suffices to only maintain the approximate value of the max flow (see Appendix A for details on how to recover an approximate flow). Our algorithm works in phases, each of which corresponds to n edge insertions. Denote $E_i = \{e_{in}, \dots, e_{in+n-1}\}$. At the beginning of phase j, we call Sparsifier-Tree data structure to get the cut sparsifier H for the graph on edges $E_{[0,j)} = E_0 \cup \dots \cup E_{j-1}$ and the maxflow f on it. Then we build the residual network R for the flow f in H and insert edges from E_i into R one by one. During edge insertions, we maintain a reachability data structure, which allows us to check whether there exists an s-t path. If it exists, we return such a path. Such a data structure processes q edge insertions in a graph with n vertices and m edges in time $\mathcal{O}(q+n+m)$. Once we found an s-t path, we augment the current flow f along this path, rebuild the residual network and reinitialize the reachability data structure. For pseudocode we refer to Algorithm 2.

Note that the flow itself is not a flow in the original graph, so for now, we can only output the flow value. First, we argue the correctness of Algorithm 2.

▶ **Lemma 21.** *For every $j \in \{0, \dots, \frac{m}{n} - 1\}$ and $i \in \{1, \dots, n\}$, the flow value ans_{i+jn} is always a $(1-\varepsilon)$ approximation of the maxflow in G_{i+jn}.*

Algorithm 2 Simple Incremental Algorithm.

Input: $e_1, \ldots, e_m, \varepsilon$
Output: $ans_1, \ldots, ans_m$ ▷ flow values approximations after edge inserting

```
1: SPARSIFIER-TREE.INITIALIZE(ε)
2: for j ← 0, ..., ⌈m/n⌉ − 1 do
3:     H, f ← SPARSIFIER-TREE(j − 1)        ▷ sparsifier and maxflow for edges from E_[0,j)
4:     R ← Residual network of H with flow f.
5:     REACHABILITY.INITIALIZE(R)
6:     for i = 1 ... n do
7:         R ← Add Edge(R, e_{i+jn})
8:         while REACHABILITY.EXISTS-PATH() do
9:             P ← s-t path in R
10:            Augment f across P
11:            Update capacities on edges of R
12:            REACHABILITY.INITIALIZE(R)
13:        end while
14:        ans_{i+jn} ← F(f)
15:    end for
16: end for
```

Proof. After execution of the **while**-loop, there is no s-t path in R, so the current flow f is a maxflow in the graph $H_{cur} := H \cup e_{jn+1} \cup \cdots \cup e_{jn+i}$. By Proposition 18, H is a $(1-\varepsilon)$ cut sparsifier of G_{jn}. Proposition 17 says that then the union H_{cur} is a $(1-\varepsilon)$ cut sparsifier of $G_{jn+i} := G_{jn} \cup e_{jn+1} \cup \cdots \cup e_{jn+i}$, so the maxflow f is a $(1-\varepsilon)$ approximation of a maxflow in G_{jn+i}. ◀

To analyze the time complexity of Algorithm 2, we first bound the total number of iterations of the **while**-loop.

▶ **Lemma 22.** *The total number of iterations of the* **while***-loop in Algorithm 2 is* $1/(1-\varepsilon)F^*(G)$.

Proof. Consider the j-th phase. Every iteration of the **while**-loop increases the value of the flow f by at least one. Thus, the total number of iteration during the j-th phase is at most $F^*(H \cup E_j) - F^*(H)$. Recalling notation from the Sparsifier Segment Tree, H is $H_{[0,j)}$, so this quantity becomes $F^*(H_{[0,j)} \cup E_j) - F^*(H_{[0,j)})$. Since $H_{[0,j+1)}$ is a cut sparsifier of $H_{[0,j)} \cup E_j$, we have

$$F^*(H_{[0,j)} \cup E_j) - F^*(H_{[0,j)}) \leq F^*(H_{[0,j+1)}) - F^*(H_{[0,j)}).$$

Summing up this quantities for all $j \in \{0, \frac{m}{n} - 1\}$, results in

$$F^*(H_{[0,\frac{m}{n})}) - F^*(H_{[0,0)]}) \leq 1/(1-\varepsilon)F^*.$$ ◀

Now we are ready to prove the time complexity of Algorithm 2.

▶ **Lemma 23.** *The total runtime of Algorithm 2 is* $\tilde{\mathcal{O}}_\varepsilon(m + nF^*)$.

Proof. By Lemma 20, the total runtime of all calls of SPARSIFIER-TREE is $\tilde{\mathcal{O}}_\varepsilon(m + nF^*)$. Adding edges to the residual network takes $\tilde{\mathcal{O}}_\varepsilon(n)$ time in each phase, which results in total time $\tilde{\mathcal{O}}_\varepsilon(m)$ across m/n phases. Every iteration of a **while**-loop takes $\tilde{\mathcal{O}}_\varepsilon(n)$ time, since the size of the graph R is $\tilde{\mathcal{O}}_\varepsilon(n)$. By Lemma 22, the total number of iterations is $1/(1-\varepsilon)F^*$, so the total runtime required by all **while**-loop executions is $\tilde{\mathcal{O}}_\varepsilon(nF^*)$, which yields runtime $\tilde{\mathcal{O}}_\varepsilon(m + nF^*)$. ◀

Lemmata 21 and 23 combined together prove the first part of Theorem 16.

4.3 $\tilde{\mathcal{O}}_\varepsilon(m+n^2)$ incremental algorithm

Finally, we describe how to obtain total runtime $\tilde{\mathcal{O}}_\varepsilon(m+n^2)$ even when $F^*(G)$ exceeds $\tilde{\mathcal{O}}_\varepsilon(n)$. To this end, we want to use a simple trick: given an m-edge graph with maximum flow $F^*(G)$, we can round all capacities down to the nearest multiple of $\varepsilon \cdot F^*(G)/m$ and retain $(1-\varepsilon)F^*(G)$ of the original flow since the total amount of capacity erased is at most $m \cdot \varepsilon \cdot F^*(G)/m = \varepsilon \cdot F^*(G)$.

In our case, we have to apply this trick quite carefully. We run the cut sparsifier segment trees as before. However, we ensure that all capacities considered are powers of 2. We can assume this for all edges in G, and it is not hard to adapt [5] enforce that the cut sparsifier scales capacities of sampled edges again by powers of 2 without increasing the number of sampled edges significantly. Finally, since there are only $O(\log n)$ different capacities, we can also assume that the cut sparsification procedure is implemented by taking the union of the $O(\log n)$ cut sparsifiers obtained from sampling a cut sparsifier for each capacity class separately. Again, this only increases the sparsity by a factor of $O(\log n)$.

Recall that it was previously important that cut sparsifiers dominate all cuts and thus have at least the same amount of maxflow as they evolve. It is not hard that due to the above changes, this property is still guaranteed, even when rounding all capacities in the respective cut sparsifiers down to the nearest multiple of some value η. We point out that, crucially, we do not apply the rounding to cut sparsifiers and then run further cut sparsification and round down again!

Finally, whenever we run Algorithm 2, we run it as before but round down all capacities to the nearest multiple of some value η. Since the graphs that we run this algorithm on have at most $\tilde{\mathcal{O}}_\varepsilon(n)$ edges, we can choose $\eta = \tilde{\mathcal{O}}_\varepsilon(F^*(G)/n)$ while loosing at most $\varepsilon F^*(G)$ in total capacity. Now, our previous analyses can be extended to yield almost the same bounds: every augmentation increases the flow by at least $F^*(G)/\tilde{\mathcal{O}}_\varepsilon(n)$ and thus there can be at most $\tilde{\mathcal{O}}_\varepsilon(n)$ such augmentations. All other arguments extend almost seamlessly to yield the desired total runtime of $\tilde{\mathcal{O}}_\varepsilon(n^2)$.

References

1 Amir Abboud, Rasmus Kyng, Jason Li, Debmalya Panigrahi, Maximilian Probst Gutenberg, Thatchaphol Saranurak, Weixuan Yuan, and Wuwei Yuan. Deterministic almost-linear-time gomory-hu trees. In *66th IEEE Symposium on Foundations of Computer Science (FOCS 2025)*, 2025.

2 Amir Abboud, Jason Li, Debmalya Panigrahi, and Thatchaphol Saranurak. All-pairs max-flow is no harder than single-pair max-flow: Gomory-hu trees in almost-linear time. In *2023 IEEE 64th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 2204–2212. IEEE, 2023. doi:10.1109/FOCS57990.2023.00137.

3 Amir Abboud and Virginia Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *2014 IEEE 55th Annual Symposium on Foundations of Computer Science*, pages 434–443. IEEE, 2014. doi:10.1109/FOCS.2014.53.

4 Stephen Alstrup, Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup. Maintaining information in fully dynamic trees with top trees. *Acm Transactions on Algorithms (talg)*, 1(2):243–264, 2005. doi:10.1145/1103963.1103966.

5 András A Benczúr and David R Karger. Randomized approximation schemes for cuts and flows in capacitated graphs. *SIAM Journal on Computing*, 44(2):290–319, 2015. doi:10.1137/070705970.

6 Aaron Bernstein, Maximilian Probst Gutenberg, and Thatchaphol Saranurak. Deterministic decremental reachability, SCC, and shortest paths via directed expanders and congestion balancing. In *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1123–1134. IEEE, 2020. doi:10.1109/FOCS46700.2020.00108.

7 Aaron Bernstein, Jan van den Brand, Maximilian Probst Gutenberg, Danupon Nanongkai, Thatchaphol Saranurak, Aaron Sidford, and He Sun. Fully-Dynamic Graph Sparsifiers Against an Adaptive Adversary. In *49th International Colloquium on Automata, Languages, and Programming (ICALP 2022)*, volume 229 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 20:1–20:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.ICALP.2022.20.

8 Ruoxu Cen, William He, Jason Li, and Debmalya Panigrahi. Steiner connectivity augmentation and splitting-off in poly-logarithmic maximum flows. In *Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2449–2488. SIAM, 2023. doi:10.1137/1.9781611977554.CH95.

9 Li Chen, Gramoz Goranci, Monika Henzinger, Richard Peng, and Thatchaphol Saranurak. Fast dynamic cuts, distances and effective resistances via vertex sparsifiers. In *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1135–1146. IEEE, 2020. doi:10.1109/FOCS46700.2020.00109.

10 Li Chen, Rasmus Kyng, Yang Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. Maximum flow and minimum-cost flow in almost-linear time. *Journal of the ACM*, 72(3):1–103, 2025. doi:10.1145/3728631.

11 Li Chen, Rasmus Kyng, Yang P. Liu, Simon Meierhans, and Maximilian Probst Gutenberg. Almost-linear time algorithms for incremental graphs: Cycle detection, sccs, s-t shortest path, and minimum-cost flow. In *Proceedings of the 56th Annual ACM Symposium on Theory of Computing*, STOC '24, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3618260.3649745.

12 Søren Dahlgaard. On the Hardness of Partially Dynamic Graph Problems and Connections to Diameter. In Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi, editors, *43rd International Colloquium on Automata, Languages, and Programming (ICALP 2016)*, volume 55 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 48:1–48:14, Dagstuhl, Germany, 2016. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ICALP.2016.48.

13 David Eppstein, Zvi Galil, Giuseppe F Italiano, and Amnon Nissenzweig. Sparsification—a technique for speeding up dynamic graph algorithms. *Journal of the ACM (JACM)*, 44(5):669–696, 1997. doi:10.1145/265910.265914.

14 Andrew V Goldberg and Satish Rao. Beyond the flow decomposition barrier. *Journal of the ACM (JACM)*, 45(5):783–797, 1998. doi:10.1145/290179.290181.

15 Gramoz Goranci and Monika Henzinger. Efficient data structures for incremental exact and approximate maximum flow. In *50th International Colloquium on Automata, Languages, and Programming, ICALP 2023*, page 69, 2023.

16 Gramoz Goranci, Monika Henzinger, Peter Kiss, Ali Momeni, and Gernot Zöcklein. Dynamic hierarchical j-tree decomposition and its applications. In *Proceedings of the 2026 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1128–1180. SIAM, 2026. doi:10.1137/1.9781611978971.45.

17 Gramoz Goranci, Monika Henzinger, Harald Räcke, and A. R. Sricharan. Incremental Approximate Maximum Flow via Residual Graph Sparsification. In *52nd International Colloquium on Automata, Languages, and Programming (ICALP 2025)*, volume 334 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 91:1–91:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPIcs.ICALP.2025.91.

18 Gramoz Goranci, Harald Räcke, Thatchaphol Saranurak, and Zihan Tan. The expander hierarchy and its applications to dynamic graph algorithms. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2212–2228. SIAM, 2021. doi:10.1137/1.9781611976465.132.

19 Manoj Gupta and Shahbaz Khan. Simple dynamic algorithms for maximal independent set, maximum flow and maximum matching. In *Symposium on Simplicity in Algorithms (SOSA)*, pages 86–91. SIAM, 2021. doi:10.1137/1.9781611976496.10.

20 Zhongtian He, Shang-En Huang, and Thatchaphol Saranurak. Cactus representation of minimum cuts: Derandomize and speed up. In *Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1503–1541. SIAM, 2024. doi:10.1137/1.9781611977912.61.

21 Zhongtian He, Shang-En Huang, and Thatchaphol Saranurak. Cactus representations in polylogarithmic max-flow via maximal isolating mincuts. In *Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1465–1502. SIAM, 2024. doi:10.1137/1.9781611977912.60.

22 Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*, pages 21–30, 2015. doi:10.1145/2746539.2746609.

23 Monika Rauch Henzinger. A static 2-approximation algorithm for vertex connectivity and incremental approximation algorithms for edge and vertex connectivity. *Journal of Algorithms*, 24(1):194–220, 1997. doi:10.1006/JAGM.1997.0855.

24 Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *Journal of the ACM*, 48(4):723–760, 2001. doi:10.1145/502090.502095.

25 David R Karger and Matthew S Levine. Fast augmenting paths by random sampling from residual graphs. *SIAM Journal on Computing*, 44(2):320–339, 2015. doi:10.1137/070705994.

26 Jason Li, Danupon Nanongkai, Debmalya Panigrahi, Thatchaphol Saranurak, and Sorrachai Yingchareonthawornchai. Vertex connectivity in poly-logarithmic max-flows. *Journal of the ACM*, 72(4):1–34, 2025. doi:10.1145/3743670.

27 Jason Li and Debmalya Panigrahi. Deterministic min-cut in poly-logarithmic max-flows. In *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 85–92. IEEE, 2020. doi:10.1109/FOCS46700.2020.00017.

28 Richard Peng. Approximate undirected maximum flows in o (m polylog (n)) time. In *Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 1862–1867. SIAM, 2016. doi:10.1137/1.9781611974331.CH130.

29 Jonah Sherman. Nearly maximum flows in nearly linear time. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pages 263–269. IEEE, 2013. doi:10.1109/FOCS.2013.36.

30 Jan Van Den Brand, Li Chen, Rasmus Kyng, Yang P Liu, Simon Meierhans, Maximilian Probst Gutenberg, and Sushant Sachdeva. Almost-linear time algorithms for decremental graphs: Min-cost flow and more via duality. In *2024 IEEE 65th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 2010–2032. IEEE, 2024. doi:10.1109/FOCS61266.2024.00120.

31 Jan Van Den Brand, Li Chen, Rasmus Kyng, Yang P Liu, Richard Peng, Maximilian Probst Gutenberg, Sushant Sachdeva, and Aaron Sidford. A deterministic almost-linear time algorithm for minimum-cost flow. In *2023 IEEE 64th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 503–514. IEEE, 2023. doi:10.1109/FOCS57990.2023.00037.

32 Jan van den Brand, Li Chen, Rasmus Kyng, Yang P Liu, Richard Peng, Maximilian Probst Gutenberg, Sushant Sachdeva, and Aaron Sidford. Incremental approximate maximum flow on undirected graphs in subpolynomial update time. In *Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2980–2998. SIAM, 2024. doi:10.1137/1.9781611977912.106.

33 Jan van den Brand, Yang P Liu, and Aaron Sidford. Dynamic maxflow via dynamic interior point methods. In *Proceedings of the 55th Annual ACM Symposium on Theory of Computing*, pages 1215–1228, 2023. doi:10.1145/3564246.3585135.

A Finding a flow approximation, given the approximate flow value

Algorithm 2 returns approximations only for the flow values. In this section, we discuss how to find a $(1-\varepsilon)$ approximation of the flow itself after each edge insertion. Consider the following algorithm: we start with an empty flow and let $\delta = \frac{\varepsilon}{4}$. After each edge insertion, we add the edge to the residual network (without any additional data structure). Then, we find a $(1-\delta)$-approximate flow value f_{approx} and, if the value $(1-\delta)^2 f_{current}$ for our current flow is smaller, we find the flow in the residual network using the algorithm of Karger and Levine [25] with runtime $\tilde{\mathcal{O}}_\varepsilon(m+nF)$. We present the pseudocode in Algorithm 3.

Algorithm 3 Finding a $1-\varepsilon$ flow approximation.

Input: $\varepsilon, e_1, \ldots, e_m$
Output: $ans_1, \ldots, ans_m$ ▷ approximations of the flow after edge inserting
$R \leftarrow \emptyset$ ▷ Initialize residual network
$F \leftarrow \emptyset, f_{current} \leftarrow 0$ ▷ Initialize flow
$\delta \leftarrow \frac{\varepsilon}{4}$
for $i \leftarrow 1, \ldots, m$ **do**
    $R \leftarrow R \cup e_i$
    $F_{approx} \leftarrow$ Get approximated $(1-\delta)$ flow value after k edge insertions ▷ Algorithm 2
    if $F(f_{current}) < (1-\delta)^2 \cdot F_{approx}$ **then**
        $f_{new}, R \leftarrow \text{Karger} - \text{Levine}(R)$ ▷ F_{new} is the found flow
        $f_{current} \leftarrow f_{current} \cup f_{new}$
    end if
    $ans_i \leftarrow f_{current}$
end for

▶ **Proposition 24.** *Let $0 < \varepsilon < 1$. Then for every i, the flow ans_i is a $(1-\varepsilon)$-approximation of F_i^*.*

Proof. Consider the i-th iteration of the **for** cycle. If the condition of the **if** operator is not satisfied, then we have

$$f_{current} \geq (1-\delta)^2 f_{approx} \geq (1-\delta)^3 f_i^* \geq (1-\varepsilon) f_i^*.$$

If the condition of the **if** operator is satisfied, we find an exact max flow in the residual network, and therefore, $f_{current} = f_i^*$.

On the other hand, the flow F is always a legal flow in the graph G_i, so it always holds that $f_{current} \leq f_i^*$. ◀

▶ **Proposition 25.** *The running time of Algorithm 3 is $\tilde{\mathcal{O}}_\varepsilon(m + \frac{nF^*}{\varepsilon^2})$.*

Proof. The algorithm has two time-consuming subroutines. Finding an approximate flow costs $\tilde{\mathcal{O}}_\varepsilon(m + \frac{nF^*}{\varepsilon^2})$ in general. Each Karger–Levine call takes $\tilde{\mathcal{O}}_\varepsilon(m + nF(R))$ time, where $F(R)$ is the max flow value in R.

Since $(1-\delta)^2 f_{approx} \geq (1-\varepsilon) f_i^*$ and after every Karger–Levine call we obtain $f_{current} = f_i^*$, the variable $f_{current}$ scales by a factor of at least $1/(1-\varepsilon)$. At the end of the algorithm we have $f_{current} \leq F^*$, so there are no more than $\log_{1/(1-\varepsilon)} F^* = \mathcal{O}\left(\frac{\log F}{\varepsilon}\right)$ calls of Karger–Levine. Thus, the total runtime of Karger–Levine is bounded by $\tilde{\mathcal{O}}_\varepsilon(m + \frac{nF^*}{\varepsilon^2})$. ◀