Efficient Data Structures for Incremental Exact and Approximate Maximum Flow

Gramoz Goranci
Faculty of Computer Science, Universität Wien, Austria

Monika Henzinger
Institute of Science and Technology Austria (ISTA), Klosterneuburg, Austria

Abstract

We show an $(1+\epsilon)$-approximation algorithm for maintaining maximum s-t flow under m edge insertions in $m^{1/2+o(1)}\epsilon^{-1/2}$ amortized update time for directed, unweighted graphs. This constitutes the first sublinear dynamic maximum flow algorithm in general sparse graphs with arbitrarily good approximation guarantee.

Furthermore we give an algorithm that maintains an exact maximum s-t flow under m edge insertions in an n-node graph in $\tilde{O}(n^{5/2})$ total update time. For sufficiently dense graphs, this gives to the first exact incremental algorithm with sub-linear amortized update time for maintaining maximum flows.

2012 ACM Subject Classification Theory of computation → Dynamic graph algorithms

Keywords and phrases dynamic graph algorithms, maximum flow, data structures

Digital Object Identifier 10.4230/LIPIcs.ICALP.2023.69

Category Track A: Algorithms, Complexity and Games

Related Version *Previous Version*: https://arxiv.org/abs/2211.09606

Funding This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (Grant agreement No. 101019564 "The Design of Modern Fully Dynamic Data Structures (MoDynStruct)" and from the Austrian Science Fund (FWF) project "Static and Dynamic Hierarchical Graph Decompositions", I 5982-N, and project "Fast Algorithms for a Reactive Network Layer (ReactNet)", P 33775-N, with additional funding from the *netidee SCIENCE Stiftung*, 2020–2024.

Acknowledgements This work was done in part while Gramoz Goranci was at Institute for Theoretical Studies, ETH Zurich, Switzerland. There, he was supported by Dr. Max Rössler, the Walter Haefner Foundation and the ETH Zürich Foundation. We also thank Richard Peng, Thatchaphol Saranurak, Sebastian Forster and Sushant Sachdeva for helpful discussions, and the anonymous reviewers for their insightful comments.

1 Introduction

The maximum flow problem and its dual, the minimum cut problem, are one of the cornerstones problems in combinatorial optimization. They are often used as subroutine for solving other prominent graph problems (e.g., Gomory-Hu Trees [11], Sparsest Cut [24]), performing divide-and-conquer on graphs [8] and have found several applications across many areas including computer vision [2], clustering [29] and scientific computing. Designing fast maximum flow algorithms has been an active area of research for decades, with recent advances making tremendous progress towards the quest of designing a near-linear time algorithm [6, 30, 26, 23, 28, 27, 31, 25, 4]. This has culminated in a recent breakthrough result due to Chen, Kyng, Liu, Peng, Probst Gutenberg, and Sachdeva [4], which computes a maximum flow in $m^{1+o(1)}$ time, where m is the number of edges of the input graph.

Recently, we have witnessed a growing interest in designing dynamic algorithms for computing maximum flows in dynamically changing graphs [20, 5, 1, 7, 14, 13, 3, 21]. Despite this, the current fastest algorithms either incur super-constant approximation factors [13, 3] or achieve competitive update times only for sufficiently dense graphs [1]. Moreover, all previous works on dynamic flows are restricted to undirected graphs. From a (conditional) lower bound perspective, for any $\delta > 0$, it is known [7] that no algorithm can exactly maintain maximum flow in $O(m^{1-\delta})$ amortized time per operation, even when restricted to algorithms that support edge insertions, unless the OMv conjecture [16] is false. Nevertheless, the lower bound construction from [7] is on dense graphs, i.e., $m = \Omega(n^2)$, and thus for sparse graphs yields only an $\Omega(m^{1/2})$ lower bound on the update time.

In this paper, we show a simple generic algorithmic framework for maintaining *approximate* maximum flows under edge insertions.

▶ **Theorem 1.** *Let $G = (V, E)$ be an initially empty, directed, unweighted n-vertex graph, s and t be any two vertices, and $\epsilon > 0$, $\mu \in [0, n]$ be two parameters. If there is*

(a) *an incremental algorithm* IncBMF(G, s, t, μ) *for inserting m edges and maintaining s-t maximum flow whose value is bounded by μ in $t_{total}(m, n, \mu)$ total update time and $q(m, n)$ query time, and*

(b) *a static algorithm for computing exact s-t maximum flow in an n-vertex, m'-edge graph in $t_{static}(m', n)$ time, where $m' \leq m$,*

then we can design an incremental algorithm for maintaining a $(1 + \epsilon)$-approximate s-t maximum flow under m edge insertions in

$$\frac{t_{total}(m, n, \mu + 1)}{m} + \frac{t_{static}(m, n)}{\epsilon\mu} + q(m, n)$$

amortized update time and $q(m, n)$ query time.

For maintaining exact maximum flows whose value is bounded by μ, we slightly adapt an incremental version of the Ford-Fulkerson [9] algorithm, which was initially observed by Henzinger [17] and later by Gupta and Khan [14] (cf. Lemma 9). This gives an incremental algorithm with $O(m\mu)$ total update time and $O(1)$ query time. The recent breakthrough result due to Chen, Kyng, Liu, Peng, Probst Gutenberg, and Sachdeva [4] (cf. Theorem 6) gives a static exact maximum flow algorithm that runs in $m^{1+o(1)}$ time. Plugging these bounds in Theorem 1 and choosing $\mu = m^{1/2+o(1)}\epsilon^{-1/2}$ yields the main result of this paper, which we summarize in the theorem below.

▶ **Theorem 2.** *Given an initially empty, directed, unweighted graph $G = (V, E)$, any two vertices s and t in V, and any $\epsilon > 0$, there is an incremental algorithm that maintains a $(1 + \epsilon)$-approximate maximum s-t flow in G under m edge insertions in $m^{1/2+o(1)}\epsilon^{-1/2}$ amortized update time. The algorithm supports queries about the value of the maintained flow in $O(1)$ time.*

When the underlying graph is undirected and unweighted, we additionally show an improved incremental version of an algorithm due to Karger and Levine [22] for maintaining exact maximum flows whose value is bounded by μ. Concretely, our algorithm achieves $\tilde{O}(m + n\mu^{3/2})$ total update time for handling m edge insertions (cf. Lemma 15). Since $\mu \leq n$ always holds in unweighted graphs, we immediately obtain the following result.

▶ **Theorem 3.** *Given an initially empty, undirected, unweighted graph $G = (V, E)$, any two vertices s and t in V, and any $\epsilon > 0$, there is an incremental algorithm that maintains an* exact *maximum s-t flow in G under m edge insertions in $\tilde{O}(n^{5/2})$ total update time. The algorithm supports queries about the value of the maintained flow in $O(1)$ time.*

For sufficiently dense graphs, this gives to the first exact incremental algorithm with sub-linear amortized update time for maintaining maximum flows.

We believe that our approach to dynamic flows may serve as the basis for designing new fully-dynamic maximum flow algorithms with competitive approximation ratio.

Independet Work. A recent independent work by Brand, Liu and Sidford [32] provides an algorithm for incremental approximate maximum flow with $n^{1/2+o(1)}\epsilon^{-1}$ amortized update time on directed graphs. They achieve this result by implementing a dynamic variant of the recent maximum flow algorithm based on the Interior Point Method (IPM) [4]. For comparison, their result extends to capacitated graphs with polynomially bounded capacitites, and achieves a speed up on the running time (albeit only on dense graphs). However, these improvements come at the cost of employing the complicated machinery of IPMs. Our result from Theorem 2 is simpler, matches their running time guarantee on sparse graphs and gives a slightly better dependncy on the accuracy parameter ϵ.

2 Preliminaries

In the following, we settle some basic notation, as well as review definitions and algorithms for computing flows on graphs.

Maximum Flow

Let $G = (V, E)$ be a directed, unweighted graph with n vertices and m edges, let $s \in V$ be a *source* vertex, and let $t \in V$ be a *target* vertex. A *flow* from s to t in G is a function $f : E \to \mathbb{R}^+$ that maps each edge to a non-negative real number; the value $f(e)$ represents the amount of flow sent along e. A flow must satisfy the following properties: (i) for each $e \in E$, we have $f(e) \leq 1$, known as *capacity constraints*, and (ii) for each $v \in V \setminus \{s, t\}$, $\sum_{(u,v)\in E} f(u, v) = \sum_{(v,u)} f(v, u)$, known as *conservation constraints*. The *value* of a flow f is the amount of flow leaving the source s minus the amount flow entering s, i.e., $v(f) = \sum_{(s,u)\in E} f(s, u) - \sum_{(u,s)\in E} f(u, s)$. In the maximum s-t flow problem, the goal is to find a flow f with the largest $v(f)$, called the *maximum s-t flow*. Let $F^* = \max\{v(f) \mid f \text{ is a flow in } G\}$. Note that while F^* is unique, there might be multiple maximum s-t flows attaining F^*.

Residual graph and Augmenting Paths

Given a directed, unweighted graph $G = (V, E)$, and a flow f from s to t in G, we let $G_f = (V, E_f)$ be the *residual graph* of G with respect to f, where E_f contains all edges of E, except that their direction is reversed if $f(e) = 1$. An edge whose direction in G_f is reversed is referred to as a *backward* edge. Otherwise, the edge is a *forward* edge. An augmenting path P from s to t in G is a simple directed path from s to t in G_f. We next review a powerful result relating the residual graphs and optimal flows.

▶ **Lemma 4** ([9]). *If there is no directed path from s to t in the residual graph G_f, then the flow f is a maximum s-t flow.*

Another useful fact is given in the lemma below.

▶ **Lemma 5.** *If there exists a directed path P from s to t in G_f, pushing flow along P in G increases the value of the flow f by at exactly one.*

Exact Maximum Flow in Directed Graphs

Our incremental approximate algorithm heavily relies on the ability to compute a maximum s-t flow quickly. Hence, we use the current fastest result [4] that achieves an exact, almost-linear algorithm for the maximum flow problem on directed graphs.[1]

▶ **Theorem 6** ([4]). *For any directed, unweighted graph $G = (V, E)$ and any two vertices s and t, there is an algorithm that computes an* exact *maximum s-t flow in G in $m^{1+o(1)}$ time.*

Approximate Maximum Flow

To measure the quality of approximate maximum flows, we will use the notion of α-approximations, which indicates that the value of the current flow solution is at least $1/\alpha$ of the optimum value. In other words, a flow f is α-approximate if $F := v(f) \geq \frac{1}{\alpha}F^*$.

3 A framework for Incremental Approximate Maximum Flow

In this section we show a simple generic algorithmic framework for maintaining a $(1+\epsilon)$-approximate maximum s-t flow under edge insertions, i.e., prove Theorem 1. Our construction is based on two important components: (i) incrementally maintaining maximum s-t flows whose value is upper bounded by some parameter μ (one should think of μ being small relative to the size of the network) and (ii) performing periodical rebuilds whenever the maximum s-t flow of the current flow is larger than the parameter μ. The latter is a common approach in dynamic algorithms and was used by Gupta and Peng [15] in their dynamic algorithm for maintaining approximate matchings, and also recently leveraged in the context of *exact* algorithms for the dynamic minimum cut problem [12]. We next further elaborate on the precise requirements of both components and discuss how they lead to our algorithm.

To implement component (i), given a directed, unweighted n-vertex, m-edge graph G, any two vertices s, t, and a parameter $\mu \in [0, n]$, the goal is to construct a data structure denoted by IncBMF(G, s, t, μ), or simply IncBMF, that supports the following operations

- Initialize(G, s, t, μ): Initializes the data structure.
- Insert(u, v): Insert the edge (u, v) to G.
- MaxFlow(s, t): Return the value of the maximum s-t flow in the current graph G if this value is smaller than μ.

Ideally, we would like that IncBMF(G, s, t, μ) supports edge insertions in amortized time proportional to the parameter μ, and queries in constant time. This would alone lead to an efficient incremental maximum flow algorithm whenever the current flow value is bounded by μ.

When the maximum flow is large (i.e., component (ii)), we can make use of the *stability* of maximum flow and periodically invoke a fast static algorithm. Concretely, first note that the value of the maximum s-t flow changes by at most one per insertion. Therefore, if we have a large flow that is close to the maximum one, it will remain close to the maximum flow over a large number of updates. This naturally leads to the following simple but powerful approach: compute a flow at a certain time in the update sequence and do nothing for a certain number of updates as long as the flow is a good approximation to the maximum flow. This idea together with the data structure IncBMF(G, s, t, μ) yields an incremental algorithm for approximating s-t maximum flow, which we formally describe below.

[1] The algorithm extends to graphs with polynomially bound weights, but for the purposes of this paper and simplifying the presentation, we only state the unweighted version of this result

Given a directed, unweighted graph $G = (V, E)$, any two vertices s, t, and two parameters $\epsilon > 0, \mu \in [0, n]$, our data structure maintains:

- a flow estimate F to the maximum s-t flow F^*,
- a counter τ indicating the number of operations since the last rebuild,
- an incremental algorithm IncBMF for maintaining graphs with the maximum flow bounded by some parameter $(\mu + 1)$.

Initially, G is an empty graph, $F \leftarrow 0$, $\tau \leftarrow 0$, and we invoke the operation Initialize of IncBMF with $(G, s, t, \mu + 1)$ as an input. Upon insertion of an edge (u, v) to G, we query IncBMF to determine whether the s-t maximum flow in the current graph is at most μ. If so, we pass the edge insertion to the IncBMF data structure and update F accordingly.

On the other hand, if the current maximum s-t flow is larger than μ (and our algorithm always correctly detects this since IncBMF run with the parameter $(\mu + 1)$ returns the correct answer), we increment τ, which counts the number of insertions since the last reset of τ that fall into this case. If $\tau \geq \epsilon\mu$, we compute an exact maximum flow F^* for the current graph from scratch using a static algorithm, update F using the value of F^* and set $\tau = 0$. We call such a step a *rebuild* step. Observe that since we are in the insertions-only setting, once a maximum flow is larger than μ, it will always remain larger than that value. Finally, to answer a query about the maximum s-t flow, we return F as an estimate. These procedures are summarized in Algorithm 1.

Algorithm 1 Incremental Approximate Maximum Flow (IncApproxMF.)

```
1  Procedure Initialize(G = (V, E), s, t, ϵ, μ)
2      Set E ← 0 and F ← 0
3      Invoke IncBMF.Initialize(G, s, t, μ + 1)
4  Procedure Insert(u, v)
5      E ← E ∪ {(u, v)}
6      if IncBMF.MaxFlow(s, t) ≤ μ then
7          Invoke IncBMF.Insert(u, v)
8          Set F ← IncBMF.MaxFlow(s, t)
9      else
10         Set τ ← τ + 1
11         if τ ≥ ϵμ then
12             Compute an s-t maximum flow in G using a static algorithm
13             Set F to be the value of the flow computed in the previous step
14             Set τ ← 0
15 Procedure MaxFlow(s, t)
16     return F
```

▶ **Theorem 7** (Restatement of Theorem 1). *Let $G = (V, E)$ be an initially empty, directed, unweighted n-vertex graph, s and t be any two vertices, and $\epsilon > 0$, $\mu \in [0, n]$ be two parameters. If there is*

(a) *an incremental algorithm* IncBMF(G, s, t, μ) *for inserting m edges and maintaining s-t maximum flow whose value is bounded by μ in total update time $t_{total}(m, n, \mu)$ and $q(m, n)$ query time, and*

(b) *a static algorithm for computing exact s-t maximum flow in an n-vertex, m'-edge graph in $t_{static}(m', n)$ time, where $m' \leq m$,*

then we can design an incremental algorithm for maintaining a $(1+\epsilon)$-approximate s-t maximum flow under m edge insertions in

$$\frac{t_{total}(m, n, \mu+1)}{m} + \frac{t_{static}(m, n)}{\epsilon\mu} + q(m, n)$$

amortized update time and $q(m, n)$ query time.

Proof. We first prove the correctness of the algorithm. Let G be the current graph and let F be the estimate maintained by the algorithm to the value of the maximum s-t flow F^* in G. We will show that F is an $(1+\epsilon)$-approximation to F^*. To this end, we distinguish the following three cases.

(1) If $F^* \leq \mu$, then by assumption of the theorem, the data structure IncBMFensures that $F = F^*$ and thus our claim trivially holds.

(2) If $F^* = \mu + 1$, the call IncBMF.MaxFlow(s, t) returns the value $\mu + 1$ and, thus, the algorithm reaches the else-case for the first time (here we slightly abuse the notation and denote this as a *rebuild* step).

(3) If $F^* > \mu + 1$, then this is not the first time that the algorithm reaches the else-case and, thus, there was a prior rebuild. Note that F corresponded to the value of some s-t maximum flow at the last prior rebuild. This in turn implies that F must be larger than μ. Let F_0^* be the value of the maximum s-t flow of the graph at that rebuild. Since each edge insertion can increase the value of the maximum flow by at most 1 and we recompute a new maximum flow every $\epsilon\mu$ insertions, we have that $F^* \leq F_0^* + \epsilon\mu$. Since $F_0^* > \mu$ and $F = F_0^* \geq 1$, bringing these together yields:

$$\frac{F^*}{F} \leq \frac{F_0^* + \epsilon\mu}{F} \leq \frac{(1+\epsilon)F_0^*}{F_0^*} \leq 1 + \epsilon,$$

which proves our claimed approximation guarantee.

We next study the running time. Note that our algorithm passes the edge insertions to the incremental algorithm IncBMF (invoked with the parameter $\mu + 1$) only if the value of the maximum flow in the current graph is bounded by μ. Hence, by the theorem assumption, the total update time to handle these insertions is $t_{\text{total}}(m, n, \mu+1) + mq(m, n)$. Amortizing the latter over m insertions gives an amortize cost of $t_{\text{total}}(m, n, \mu+1)/m + q(m, n)$, which in turn gives the first and the third term of our claimed running time guarantee.

It remains to analyze the cost of periodical rebuilds. Note that if the current maximum flow value is larger than μ, our algorithm updates the estimate F every $\epsilon\mu$ operations. By assumption of the theorem, the time to compute an exact maximum flow is $t_{\text{static}}(m', n) \leq t_{\text{static}}(m, n)$ as $m' \leq m$. Charging this time over $\epsilon\mu$ insertions, yields an amortized cost of $t_{\text{static}}(m, n)/(\epsilon\mu)$, which in turn gives the second term our of claimed runtime guarantee and completes the proof of the theorem. ◀

4 Incremental Bounded Maximum Flow

In this section we give two incremental algorithms for exactly maintaining the maximum flow as long as its value is bounded by a predefined parameter μ. The first is an incremental version of the Ford-Fulkerson [9] algorithm, applies to directed graphs, and runs in $O(m\mu)$ total update time, while the second one is an incremental version of an algorithm due to Karger and Levine [22], applies to undirected graphs and runs in $\tilde{O}(m + n\mu^{3/2})$ total update time.

4.1 Directed Graphs

The algorithm we are about to discuss applies to directed, unweighted graphs, was initially observed by Henzinger [17] and later by Gupta and Khan [14], and can be thought of as an incremental version of the celebrated Ford-Fulkserson algorithm [9]. We review it below and slightly adapt it for our purposes.

Henzinger [17] showed how to incrementally maintain maximum s-t flow in $O(F^*)$ amortized update time, where F^* is the value of the maximum flow in the final graph. As there are graphs where $F^* = \Omega(n)$, her running time guarantee is competitive only when F^* is small, e.g., sub-linear on the size of the graph. We next show to slightly adapt her algorithm so that it maintains a maximum flow as long as its value is bounded by a parameter μ.

The key observation behind this algorithm is that the insertions of an (unit-capacitated) edge can only increase the maximum flow value by at most 1. To check whether this value has increased, she uses Lemma 4 as a certificate, i.e., one determines whether the insertion of the (forward) edge in the residual graph G_f creates a directed path from s to t in G_f. A naive way to determine this is to run a graph search algorithm on G_f after each insertion, which requires $\Omega(m)$ for a single update and is thus prohibitively expensive for our purposes. However, one can exploit a data structure due to Italiano [19] for incrementally maintaining single source reachability information from a source s which requires $O(m)$ total update time for handling m insertions. Let us briefly review this data structure before presenting the incremental algorithm.

Incremental Single Source Reachability

In the incremental single source reachability problem, given an (initially empty) directed, unweighted graph $G = (V, E)$ and a distinguished vertex s, the goal is to construct a data structure IncSSR that supports the following operations: (i) Initialize(G, s): initialize the data structure in G with source s, (ii) Insert(u, v): insert the edge (u, v) in G, and (iii) Reach(u): return `True` if u is reachable from s, and `False` otherwise.

Italiano [19] observed that an incremental version of graph search leads to an efficient incremental IncSSR data structure. The main idea is to maintain a reachability tree T from s. Initially, the tree is initialized to $\{s\}$. Upon insertion of an edge (u, v) to G, we need to update T iff $u \in T$ and $v \notin T$. If this is the case, we add (u, v) to T and make the v the child of u. Moreover, the algorithm examines all outgoing neighbors w incident to v, and if $w \notin T$, processes the edge (v, w) recursively using the same procedure. To answer queries, we return `True` if $u \in T$, and `False` otherwise. These procedures are summarized in Algorithm 2.

The correctness of the data structure immediately follows by construction as we always maintain a correct rechability tree T from s for the current graph. For the running time, note that the total time over all insertions is $O(m)$ as each edge is processed at most $O(1)$ times; once when it is inserted into the graph and once when it is added to T.

▶ **Lemma 8** ([19]). *Given an initially empty directed, unweighted graph $G = (V, E)$ and a source vertex s, the incremental algorithm* IncSSR *maintains reachability information from s to every other node in V while supporting insertions in $O(1)$ amortized update time and queries in $O(1)$ in worst-case time.*

The Algorithm

We now have all the necessary tools to present an incremental algorithm maintaining the maximum flow whose value is bounded by μ. Let F^* denote the maximum s-t flow value on the current graph.

Algorithm 2 Incremental Single Source Reachability (IncSSR).

```
1  Procedure Initialize(G = (V, E), s)
2      Set T ← {s} and E ← ∅
3  Procedure Insert(u, v)
4      E ← E ∪ {(u, v)}
5      UpdateTree(u, v)
6  Procedure UpdateTree(u, v)
7      if u ∈ T and v ∉ T then
8          Make v a child of u in T
9          foreach (v, w) ∈ E do
10             UpdateTree(v, w)
11 Procedure Reach(u)
12     if u ∈ T then
13         return True
14     else
15         return False
```

Initially, G and the residual graph G_f are empty graphs, $F^* \leftarrow 0$ and $f(e) \leftarrow 0$ for each $e \in E$. The algorithm proceeds in μ rounds, where a round ends when the value of the current maximum flow increases by one. Each round starts by initializing an incremental single source reachability data structure IncrSSR from the source s (Lemma 8) on the residual graph G_f. Upon an edge insertion (u, v) to G, we pass the directed edge (u, v) to the data structure IncrSSR and test whether t is reachable from s using this data structure. If the latter holds, then we find a simple directed s-t path P in G_f, which in turn serves as an augmenting path for G. We then send one unit of flow along the path P in G and update the current flow and its value accordingly. To answer a query about the maximum flow between s and t, we simply return F^*. These procedures are summarized in Algorithm 3.

The correctness of this algorithm is immediate by Lemma 4, which correctly tells us when to increase the value of the maximum flow, and Lemma 5, which asserts that the sending one unit of flow along an augmenting path increases the value of the flow by exactly one.

For the running time, note that each round requires $O(m)$ total time. As there are exactly μ rounds, we get a total update time of $O(m\mu)$. The query time is $O(1)$ as we simply return the value of current maximum flow.

▶ **Lemma 9.** *Given an initially empty directed, unweighted graph $G = (V, E)$ with n vertices, any two vertices s and t, and a parameter $\mu \in [0, n]$, the algorithm* IncBMF(G, s, t, μ) *exactly maintains, under m edge insertions, the maximum s-t flow in G whose value is bounded by μ in $O(m\mu)$ total update time and $O(1)$ query time.*

4.2 Undirected Graphs

We next give an incremental variant of the deterministic maximum flow algorithm for unweighted, *undirected* graphs due to Karger and Levine [22]. For a threshold parameter μ on the maximum flow value, we obtain a total update time of $\tilde{O}(m + n\mu^{3/2})$ for handling m insertions.

Algorithm 3 Incremental Bounded Maximum Flow (IncBMF).

```
1  Procedure Initialize(G = (V, E), s, t, μ)
2      Set E ← ∅
3      Set f(e) ← 0 for each e ∈ E, G_f ← (V, E) and F* ← 0
4      Invoke IncSSR.Initialize(G_f, s)
5  Procedure Insert(u, v)
6      if F* ≤ μ then
7          Set E ← E ∪ {(u, v)}
8          Invoke IncSSR.Insert(u, v)
9          if IncSSR.Reach(t) then
10             Find a simple directed s-t path P in G_f
11             Augment f along the path P in G and let f' be the resulting flow
12             Set f ← f' and G_f ← G_f'
13             Set F* ← F* + 1
14             Invoke IncSSR.Initialize(G_f, s)
15 Procedure MaxFlow(s, t)
16     return F*
```

The basic idea behind this improvement is to sparsify the residual graph on a flow problem so that the augmenting paths can be found more efficiently than paying $O(m)$ per path, as we did in the incremental version of the Ford-Fulkerson algorithm. Two core components that allow for a faster algorithm are: (i) using spanning forests for edges that do not carry any flow in the residual graph (i.e, edges that remain undirected) and (ii) removing cycles from the current flow after each augmentation step to make sure that the flow does not use too many edges.

We next elaborate more on these two components. First, since we will need a different treatment for directed and undirected edges, setting up some additional notation is useful. For a graph $G = (V, E)$ and a flow f on the edges of G, we let E_f^u denote the "undirected edges" of G, i.e., edges e for which $f(e) = 0$, and let E_f^d denote the "directed edges" of G, i.e., edges e for which $f(e) = 1$. Component (i) involves replacing the edges in E_f^u with a spanning forest T. It is known that T captures the connectivity information among any pair of vertices in E_f^u, and thus whenever searching for an augmenting path, it suffices to do so in the graph induced by edge edges E_f^d and T. Another advantage is that T can have at most $(n-1)$ edges, which is potentially much smaller than the size of E_f^u. One challenge with this approach is that E_f^u evolves over time, i.e., edges might have flow added to it or flow is sent on the reserve direction during an augmentation step. Fortunately, we have efficient data structures to maintain such dynamic updates.

▶ **Lemma 10** ([18]). *Given an undirected graph $G = (V, E)$, there is an algorithm* DynSpanF *to maintain a spanning forest T of G that supports operations edge insertions and deletions (i.e., operations Insert(u, v) and Delete(u, v)) in $O(\log^2 n)$ amortized time per operation.*

Unfortunately the above idea alone is not sufficient. The problem is that we do not have any control on the size of E_f^d. It can be well the case that all edges in the graph become eventually directed, which defeats the purpose of treating undirected edges differently. To get around this, we first introduce the notion of acyclic flows and then review a result that shows that integral acyclic flows use very few edges. This lays the foundations of component (ii).

Algorithm 4 Incremental Bounded Maximum Flow for Undirected Graphs (IncBMFU).

```
1  Procedure Initialize(G = (V, E), s, t, μ)
2      Set f(e) ← 0 for each e ∈ E, E_f^u ← ∅, E_f^d ← ∅ and F* ← 0
3      Invoke DynSpanF.Initialize(G = (V, E_f^u)) to maintain a spanning forest T
4      Invoke IncSSR.Initialize(E_f^d ∪ T, s)
5  Procedure Insert(u, v)
6      if F* ≤ μ then
7          Set E_f^u ← E_f^u ∪ {(u, v)}
8          Invoke DynSpanF.Insert(u, v)
9          if (u, v) ∈ T then
10             Invoke IncSSR.Insert(u, v) and IncSSR.Insert(v, u)
11             if IncSSR.Reach(t) then
12                 Find a simple directed s-t path P in E_f^d ∪ T
13                 Augment f along the path P in G, let f' be the resulting flow and set
                    f ← f'
14                 Set f ← Decycle(f)
                   // delete any edge no longer in E_f^u because flow added
15                 for each e ∈ E_f^u with f(e) > 0 do
16                     Set E_f^u ← E_f^u \ {e} and E_f^d ← E_f^d ∪ {e}
17                     Invoke DynSpanF.Delete(e)
                   // insert an edge to E_f^u because flow removed
18                 for each e ∈ E_f^d with f(e) = 0 do
19                     Set E_f^d ← E_f^d \ {e} and E_f^u ← E_f^u ∪ {e}
20                     Invoke DynSpanF.Insert(e)
21                 Set F* ← F* + 1
22                 Invoke IncSSR.Initialize(E_f^d ∪ T, s)
23 Procedure MaxFlow(s, t)
24     return F*
```

▶ **Definition 11.** *We say that a flow f is* acyclic *if there is no directed cycle on which every edge has positive flow in the direction of the cycle.*

▶ **Lemma 12** ([10]). *Any integral acylic flow f uses at most $O(n\sqrt{v(f)})$ edges.*

Taking cue from the lemma above, our goal would be to ensure that at any time, the current flow we maintain is acyclic. Note that even if a flow is initially acylic, an augmentation step may destroy this property. This suggests that we need a *decycling* step to bring back the flow to the desired state. More importantly, for unweighted, undirected graphs, the *decycling* procedure takes time that is proportional to the number of edges that carry non-zero flow on the current graph.

▶ **Lemma 13** ([22]). *Let G be an unweighted, undirected graph, and let f be a flow of G that is non-zero on exactly x edges. Then there is an algorithm* Decycle(f) *that returns an* acyclic *flow f' with $v(f) = v(f')$ and runs in $O(x)$ time.*

The Algorithm

We now show how the above ideas lead to an incremental algorithm that maintains a maximum flow whose value is bounded by μ. As before, let F^* denote the maximum s-t flow value on the current graph.

Initially, G and the edges sets E_f^u, E_f^d are empty, $F^* \leftarrow 0$ and $f(e) \leftarrow 0$ for each $e \in E$. The algorithm initializes a dynamic spanning forest data structure DynamicSpanF on E_f^u to maintain a spanning forest T (Lemma 10). There are μ rounds, and each round ends when the value of the current maximum flow increases by one. Each round starts by initializing an incremental single source reachability data structure IncrSSR on $E_f^d \cup T$ from the source s (Lemma 8).

Upon an edge insertion (u, v) to G, we first pass this insertion to the data structure DynamicSpanF. If the edge (u, v) ends up being added to T, we then pass this insertion as two edge insertions (u, v) and (v, u) to the data structure IncrSSR and test whether t is reachable from s using this data structure. If the latter holds, then we find a simple directed s-t path in $E_f^d \cup T$, which in turn serves as an augmenting path for G. We then send one unit of flow along the path P in G. To make sure that the flow remains acyclic, we invoke procedure Decycle to remove potential directed cycles and update the current flow to be acyclic. Using the dynamic data structure DynamicSpanF, we delete all edges that no longer belong to E_f^u (because they now carry non-zero flow), and insert all new edges to E_f^u (because flow was removed from them). Finally, we increment the current flow by exactly 1.

To answer a query about the maximum flow between s and t, we simply return F^*. These procedures are summarized in Algorithm 4.

We next argue about the correctness of the algorithm. We start by reviewing the result below which shows that it is safe to restrict our attention to the graph $E_f^d \cup T$ when searching for an augmenting path.

▶ **Lemma 14** ([22]). *Let G_f be the residual graph of an undirected, unweighted graph G with respect to the flow f. Then $E_f^d \cup T$ has an augmenting path if and only if G_f does.*

In light of the lemma above, Lemma 4 and Lemma 5, it suffices to show that our incremental algorithm correctly maintains $E_f^d \cup T$. To this end, observe that this directly follows from (i) the correctness of DynSpanF data structure for maintaining T (Lemma 10) and (ii) by Lines 18-20 in Algorithm 4 which makes sure that the set E_f^d is correctly updated after each augmentation step. This completes the correctness argument.

We prove the running time complexity of the algorithm in the lemma below.

▶ **Lemma 15.** *Given an initially empty undirected, unweighted graph $G = (V, E)$ with n vertices, any two vertices s and t, and a parameter $\mu \in [0, n]$, the algorithm* IncB-MFU(G, s, t, μ) *exactly maintains, under m edge insertions, the maximum s-t flow in G whose value is bounded by μ in $\tilde{O}(m + n\mu^{3/2})$ total update time and $O(1)$ query time.*

Proof. Let us first study the work done to find augmenting paths. Since we decycle flows after each augmentation and the spanning forest T can have at most $2(n-1)$ edges (two edges in reverse direction for each undirected edge), by Lemma 12, each augmentation step is done on a graph with $O(n\sqrt{\mu})$ edges and thus takes $O(n\sqrt{\mu})$ time. Similarly, note that before an augmentation step, the set E_f^d does not change, and we only report to IncSSR data structure the edge insertions that ended up being added to T. There can be at most $2(n-1)$ such edge insertions. Therefore, the total cost of running IncSSR per round is $O(|E_f^d \cup T|) = O(n\sqrt{\mu})$. Since there are μ rounds, the total time is $O(n\mu^{3/2})$.

It remains to account for the dynamic operations handled by DynSpanF data structure. Consider the cost of deletions. An edge is deleted from the data structure whenever we put some non-zero flow on it. Since an augmenting path can have at most n edges, and there are at most μ rounds, this can happen to at most $n\mu$ edges. The latter in turn leads to at most $n\mu$ deletions for a total time of $\tilde{O}(n\mu)$ for handling them (Lemma 10).

We now turn our attention to the cost of insertions. Over the course of the incremental algorithm we pass m edges insertions to DynSpanF, for a total time of $\tilde{O}(m)$ (Lemma 10)). We also also pass insertions to DynSpanF whenever flow has been removed on the edges. However, for flow to be removed from an edge, it must have been first added an on edge, i.e., this edge was passed as a deletion to the data structure. Therefore, we can charge the total cost of these insertions to the total cost of deletions, which we bounded by $\tilde{O}(n\mu)$. This completes the proof of the lemma. ◀

5 Conclusion

In this paper we showed two algorithms for maintaining approximate and exact flows in dynamic graphs undergoing edge insertions. Our dynamic approximation algorithm first showed how to maintain small maximum flows efficiently in the incremental setting, and then employed the well-known technique of periodical rebuilds. For the exact result, we showed that the sparsifiers of residual graphs in the undirected setting can be maintained efficiently under edge insertions.

In general, the dynamic complexity of maximum flows is a largely unexplored area, with many fundamental questions remaining unanswered. For example, do there exist *decremental* algorithms achieving comparable guarantees to the ones we obtained in the incremental setting? Our framework from Theorem 6 readily extends to the graphs undergoing edge deletions only. However, it is not known how to maintain small maximum flows in the decremental setting.

Another fundamental open question is the existence of a fast *fully dynamic* algorithm that approximates maximum flows up to a constant factor. For general undirected graphs, recent research suggests that this question is intimately connected to efficient sparsifiers constructions that (approximately) preserve the cut structure between terminal subset of vertices on graphs. Thus, beyond dynamic graphs, any progress in answering this question would potentially lead to understanding other fundamental problems in graph algorithms.

References

1 Ittai Abraham, David Durfee, Ioannis Koutis, Sebastian Krinninger, and Richard Peng. On fully dynamic graph sparsifiers. In *Symposium on Foundations of Computer Science (FOCS)*, pages 335–344, 2016.

2 Yuri Boykov, Olga Veksler, and Ramin Zabih. Fast approximate energy minimization via graph cuts. *IEEE Transactions on pattern analysis and machine intelligence*, 23(11):1222–1239, 2001.

3 Li Chen, Gramoz Goranci, Monika Henzinger, Richard Peng, and Thatchaphol Saranurak. Fast dynamic cuts, distances and effective resistances via vertex sparsifiers. In *Symposium on Foundations of Computer Science (FOCS)*, 2020.

4 Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. Maximum flow and minimum-cost flow in almost-linear time. In *Annual Symposium on Foundations of Computer Science (FOCS)*, pages 612–623, 2022.

5 Ho Yee Cheung, Tsz Chiu Kwok, and Lap Chi Lau. Fast matrix rank algorithms and applications. *J. ACM*, 60(5):31:1–31:25, 2013.

6 Paul Christiano, Jonathan A. Kelner, Aleksander Madry, Daniel A. Spielman, and Shang-Hua Teng. Electrical flows, laplacian systems, and faster approximation of maximum flow in undirected graphs. In *Symposium on Theory of Computing (STOC)*, pages 273–282, 2011.

7 Søren Dahlgaard. On the hardness of partially dynamic graph problems and connections to diameter. In *International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 48:1–48:14, 2016.

8 Gary William Flake, Robert E Tarjan, and Kostas Tsioutsiouliklis. Graph clustering and minimum cut trees. *Internet Mathematics*, 1(4):385–408, 2004.

9 L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

10 Zvi Galil and Xiangdong Yu. Short length versions of menger's theorem (extended abstract). In Frank Thomson Leighton and Allan Borodin, editors, *Symposium on Theory of Computing (STOC)*, pages 499–508. ACM, 1995.

11 R. E. Gomory and T. C. Hu. Multi-terminal network flows. *Journal of the Society for Industrial and Applied Mathematics*, 9(4):551–570, 1961.

12 Gramoz Goranci, Monika Henzinger, Danupon Nanongkai, Thatchaphol Saranurak, Mikkel Thorup, and Christian Wulff-Nilsen. Fully dynamic exact edge connectivity in sublinear time. In *Symposium on Discrete Algorithms (SODA) 2023*, pages 70–86, 2023.

13 Gramoz Goranci, Harald Räcke, Thatchaphol Saranurak, and Zihan Tan. The expander hierarchy and its applications to dynamic graph algorithms. In *Symposium on Discrete Algorithms (SODA)*, 2021.

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

15 Manoj Gupta and Richard Peng. Fully dynamic (1+ e)-approximate matchings. In *Symposium on Foundations of Computer Science (FOCS)*, pages 548–557, 2013.

16 Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *Symposium on Theory of Computing (STOC)*, pages 21–30, 2015.

17 Monika Rauch Henzinger. A static 2-approximation algorithm for vertex connectivity and incremental approximation algorithms for edge and vertex connectivity. *J. Algorithms*, 24(1):194–220, 1997.

18 Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760, 2001.

19 Giuseppe F. Italiano. Amortized efficiency of a path retrieval data structure. *Theor. Comput. Sci.*, 48(3):273–281, 1986.

20 Giuseppe F. Italiano, Yahav Nussbaum, Piotr Sankowski, and Christian Wulff-Nilsen. Improved algorithms for min cut and max flow in undirected planar graphs. In *Symposium on Theory of Computing (STOC)*, pages 313–322, 2011.

21 Adam Karczmarz. Fully dynamic algorithms for minimum weight cycle and related problems. In Nikhil Bansal, Emanuela Merelli, and James Worrell, editors, *International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 198 of *LIPIcs*, pages 83:1–83:20, 2021.

22 David R. Karger and Matthew S. Levine. Finding maximum flows in undirected graphs seems easier than bipartite matching. In *Symposium on the Theory of Computing (STOC)*, pages 69–78. ACM, 1998.

23 Jonathan A. Kelner, Yin Tat Lee, Lorenzo Orecchia, and Aaron Sidford. An almost-linear-time algorithm for approximate max flow in undirected graphs, and its multicommodity generalizations. In *Symposium on Discrete Algorithms (SODA)*, pages 217–226, 2014.

24 Rohit Khandekar, Satish Rao, and Umesh V. Vazirani. Graph partitioning using single commodity flows. *J. ACM*, 56(4):19:1–19:15, 2009.

25 Yang P. Liu and Aaron Sidford. Faster divergence maximization for faster maximum flow. In *Symposium on Foundations of Computer Science (FOCS)*, 2020.

26 Aleksander Madry. Navigating central path with electrical flows: From flows to matchings, and back. In *Symposium on Foundations of Computer Science (FOCS)*, pages 253–262, 2013.

27 Aleksander Madry. Computing maximum flow with augmenting electrical flows. In *Symposium on Foundations of Computer Science (FOCS)*, pages 593–602, 2016.

28 Richard Peng. Approximate undirected maximum flows in $O(m\text{polylog}(n))$ time. In *Symposium on Discrete Algorithms (SODA)*, pages 1862–1867, 2016.

29 Thatchaphol Saranurak and Di Wang. Expander decomposition and pruning: Faster, stronger, and simpler. In Timothy M. Chan, editor, *Symposium on Discrete Algorithms (SODA)*, pages 2616–2635. SIAM, 2019.

30 Jonah Sherman. Nearly maximum flows in nearly linear time. In *Symposium on Foundations of Computer Science (FOCS)*, pages 263–269, 2013.

31 Jonah Sherman. Area-convexity, l_∞ regularization, and undirected multicommodity flow. In *Symposium on Theory of Computing (STOC)*, pages 452–460, 2017.

32 Jan van den Brand, Yang P. Liu, and Aaron Sidford. Dynamic maxflow via dynamic interior point methods. *CoRR*, abs/2212.06315, 2022. to appear at STOC'23. `arXiv:2212.06315`.