# Deterministic Maximum Flows in Simple Graphs

**Tianyi Zhang** ✉
Tsinghua University, Beijing, China

## Abstract

In this paper we are interested in deterministically computing maximum flows in undirected simple graphs where edges have unit capacities. When the input graph has $n$ vertices and $m$ edges, and the maximum flow is known to be upper bounded by $\tau$ as prior knowledge, our algorithm has running time [1]$\tilde{O}(m + n^{5/3}\tau^{1/2})$; in the extreme case where $\tau = \Theta(n)$, our algorithm has running time $\tilde{O}(n^{2.17})$. This always improves upon the previous best deterministic upper bound $\tilde{O}(n^{9/4}\tau^{1/8})$ by [Duan, 2013]. Furthermore, when $\tau \geq n^{0.67}$ our algorithm is faster than a classical upper bound of $O(m + n\tau^{3/2})$ by [Karger and Levin, 1998].

**2012 ACM Subject Classification** Theory of computation → Network flows

**Keywords and phrases** graph algorithms, maximum flows, dynamic data structures

**Digital Object Identifier** 10.4230/LIPIcs.ICALP.2021.114

**Category** Track A: Algorithms, Complexity and Games

**Acknowledgements** I want to thank helpful discussions with my advisor Ran Duan as well as my colleague Shucheng Chi.

## 1 Introduction

Let $G = (V, E)$ be an undirected simple graph on $n$ vertices and $m$ edges, where each edge has unit capacity. Fix two special vertices $s, t \in V$ and we are interested in deterministic algorithms that compute an exact maximum flow from $s$ to $t$ in graph $G$. There has been a long line of literature on the study of maximum flows since the 60's. As one of the pioneering works, Ford and Fulkerson [5] introduced the idea of augmenting paths and proposed an algorithm that runs in time $O(m\tau)$; here $\tau$ is an upper bound on the maximum flow value. Subsequent improvements came along in [3, 7], which are known as the blocking flow algorithm and the push-relabel algorithm. For several decades, the best running time was $\tilde{O}(m \min\{m^{1/2}, n^{2/3}\} \log U)$ ($U$ being the largest integer capacity) by [6]. Recently, a new line of developments [17, 14, 18, 16, 15, 13, 19] based on the interior point method surpassed the blocking flow barrier; for large capacities, the best running time is $\tilde{O}((m + n^{1/2}) \log U)$ [19], while for small capacities, the best running time so far is $O(m^{4/3+o(1)}U^{1/3})$ [15, 13].

Another branch of literature focuses on a special case when the maximum flow is known to be small as a prior knowledge. Let $\tau$ be a known upper bound on the maximum flow value. Karger proposed the first such kind of upper bound in [10], which is a randomized algorithm running in time $\tilde{O}(m^{2/3}n^{1/3}\tau)$ for simple graphs. This was improved later in [11] to randomized upper bounds of $\tilde{O}(m + n\tau^{5/4})$ and $\tilde{O}(m + n^{11/9}\tau)$ and deterministic upper bound $O(m + n\tau^{3/2})$. This line of works culminated in [12] as a randomized upper bound $\tilde{O}(m + n\tau)$. In a very recent work [1], the authors obtained a deterministic upper bound of $\tilde{O}(m\tau^{2/3})$ time which is even faster when $\tau$ is small.

A gap between randomized and deterministic algorithms has remained so far. For many years, the best deterministic algorithm has running time $\tilde{O}(m + n\tau^{3/2})$ [11], so in simple graphs this upper bound could be as large as $\Omega(n^{2.5})$. The basic idea of this algorithm is

[1] $\tilde{O}$ hides poly-logarithmic factors.

to sparsify the residual graph so that it always has $O(n\tau^{1/2})$ directed edges, and it uses connectivity structures to handle undirected edges, and in this way each flow augmentation takes time only proportional to the number of directed edges, yielding a total running time of $O(m + n\tau^{3/2})$. It was explicitly asked by the authors in [11] whether one could achieve a full sparsification of $O(n\tau^{1/2})$ edges so that running the blocking flow algorithm of [6] on the sparsified graph would take time $\tilde{O}(m + n^{5/3}\tau^{1/2})$.

To break through the 2.5 exponent even in simple graphs when $\tau = \Theta(n)$, the author of [4] managed to combine the original approach of [11] with the blocking flow technique and achieved a better running time $\tilde{O}(n^{9/4}\tau^{1/8})$ in simple graphs; this upper bound is always at most $\tilde{O}(n^{2.375})$ so it beats the 2.5 exponent in the worst case when $\tau = \Theta(n)$.

## 1.1 Our result

In this chapter, we answer the question from [11] in the affirmative for simple graphs.

▶ **Theorem 1.** *Assume $\tau \geq n^{2/3}$ is an upper bound on the value of the maximum s-t flow, then there is a deterministic algorithm that computes s-t maximum flows in undirected simple graphs in $\tilde{O}(n^{5/3}\tau^{1/2})$ running time.*

Our algorithm is always faster than [4] for any choice of $1 \leq \tau < n$ in simple graphs; in the extreme case where $\tau = \Omega(n)$, we have a running time of $O(n^{2.17})$. For a moderately large $\tau \geq n^{0.67}$, our result is also better than the classical algorithm from [11] which has $O(m + n\tau^{3/2})$ running time.

## 1.2 Technical overview

**Generalizing the blocking flows.** Our algorithm will be based on [4], so let us first try to summarize the benchmark. Throughout iterations, it maintains a set of disjoint connected components $\mathcal{V}$ in the residual graph, and define a binary edge weight function $\mu : E \to \{0, 1\}$, where inter-component edges have weight 1, and intra-component edges have weight 0. In each iteration, the algorithm searches for a blocking flow whose augmentation would increase the $s$-$t$ distance in the residual graph under edge weight $\mu$. To search for a blocking flow in the current residual graph, the algorithm performs a depth-first search only on inter-component edges, and use a connectivity data structure to route flows within components. Therefore, in general their algorithm needs to keep the total number of inter-component edges small.

The first obstacle of this approach is that the total number of directed edges would grow during augmentations. As all directed edges are inter-component ones, this immediately affects the time of finding blocking flows. To work around this issue, the algorithm of [4] was to perform a decycle operation between every pair of consecutive levels in terms of distances from $s$. Although this could keep the number of directed edges small, it merely translates directed edges to inter-component undirected edges, so the total number of inter-component edges remains large. To handle this large number of inter-component edges, their algorithm builds a decremental clustering structure between each pair of consecutive levels that comprises a collection of disjoint clusters and a sparse graph. More specifically, their data structure accepts an integer parameter $h$ and decrementally maintains a set of disjoint star subgraphs, each of size $> h$, such that the number of edges incident tn unclustered vertices is bounded by $O(nh)$. The key motivation of using star subgraphs is that it ensures each flow augmentation uses at most $O(n/h)$ inter-component edges, so that the number of inter-component edges grows slowly.

**A better clustering approach.** One downside of [4]'s decremental clustering algorithm is that each edge deletion could add $O(n)$ more edges incident on unclustered vertices. This substantially limits the usage of the decremental data structure since it needs frequently rebuilding as the number of edges incident on unclustered vertices increases fast. Our new data structure does not need any rebuilding, and furthermore, this advantage allows us to apply it to maintain the set of disjoint components $\mathcal{V}$ directly, instead of maintaining inter-component edges between consecutive levels. In this way, we can directly upper bound the length of each flow, and so we do not need any decycling step, which simplifies the blocking flow approach of [4] as well.

**Comparison with layered core decompositions.** Our new decremental clustering data structure is actually inspired by the layered core decompositions devised in a recent paper [2], but ours is much simpler and weaker in some sense. The total update time of our algorithm is at least quadratic, and it also depends on the pattern of updates, while the layered core decomposition has almost linear total update time in the worst case. Furthermore, in our clustering scheme, each cluster is simply a star subgraph, where in the layered core decomposition each cluster is an expander.

One reason we do not directly apply the layered core decomposition in the maximum flow computations is that it is a heavy tool and may probably make an overkill. When computing maximum flows, edge updates to the clustering scheme always have some benign properties, and so even though star graphs are far less robust than expanders, the clustering scheme does not have to work against strong adversaries. Furthermore, a more technical reason is that paths within the same core may have sub-polynomial lengths, while each star only has diameter 2, which contributes to the length of augmenting paths. Therefore, using the layered core decompositions might induce an extra sub-polynomial factor in the running time of max-flows.

## 2 Preliminaries

Let $G = (V, E)$ be an undirected simple graph with unit-capacities, and let $s, t \in V$ be a source and a sink. For the rest, $G$ will be the input graph where we compute max-flows.

For any graph $H = (X, F)$ and vertex subset $S \subseteq H$, let $H[S]$ be the induced subgraph of $H$ on $S$. An edge orientation on the edge set is represented as $\mathcal{O} : V \times V \to \{-1, 0, 1\}$, such that for each edge $(u, v) \in E$, $\mathcal{O}(u, v)$ is equal to 1 if the edge $(u, v)$ is oriented from $u$ to $v$, and $-1$ if it is from $v$ to $u$, and 0 if it is not oriented. For each $u \in V$, define $\deg_H(u)$ to be the number of neighbors of $u$ in $H$. If an orientation $\mathcal{O} : F \to \{-1, 0, -1\}$ of edges is imposed on $F$, then let $\deg^+_{H,\mathcal{O}}(u), \deg^-_{H,\mathcal{O}}(u)$ be the out/in-degree of $u$ in $G$. Usually, when $H$ and $\mathcal{O}$ are known from context, we would ignore the subscripts.

For any $s$-$t$ flow $f$ in $G$, let $|f|$ denote the value of the flow, and $E_f$ the set of edges carrying the flow. The residual graph of $G$ with respect to $f$ is denoted by $G_f$, which is defined as following.

▶ **Definition 2** (residual graph). *Given a flow $f$ in $G$, a residual graph $G_f$ is defined as following: for each edge $(u, v) \in E$ such that $f(u, v) = 1$, there is a directed edge from $v$ to $u$ with capacity 2.*

Here we emphasize that the orientation of edges have nothing to do with the direction of edges in the residual graph.

Next, we state some lemmas regarding flows from previous works.

▶ **Lemma 3** ([11]). *An acyclic flow $f$ in a graph with integer capacities and no parallel edges uses at most $2n\sqrt{|f|}$ edges.*

▶ **Lemma 4** ([11]). *In a simple undirected graph, it is possible to take a flow $f$ and find an acyclic flow $f'$ of the same value in time $\tilde{O}(|E_f|)$.*

▶ **Lemma 5** ([11]). *Given an acyclic flow $f$, finding $k$ augmenting paths takes $\tilde{O}(m + kn(k + |f|)^{1/2})$ deterministic time.*

We will also be using a dynamic maximum spanning forest algorithm with fast amortized update time.

▶ **Lemma 6** ([8, 9]). *Given an undirected weighted graph on $n$ vertices undergoing edge insertions and deletions, there is a deterministic dynamic algorithm that maintains a maximum spanning forest with $O(\log^4 n)$ amortized update time.*

## 3 Decremental layered clustering

In this section, let $H = (V, F)$ be an arbitrary undirected simple graph on $n$ vertices that undergoes a sequence of edge deletions.

▶ **Definition 7.** *For any undirected simple graph $H = (X, F)$, a subset $A \subseteq X$ is called **d-pruned** with respect to graph $H$, if all edges incident on $A$ could be oriented under an orientation $\mathcal{O}$ of $F$ such that*
**(1)** *All edges between $A$ and $X \setminus A$ are directed outward from $A$.*
**(2)** *Out-degree of every vertex in $A$ is less than $d$.*

▶ **Lemma 8.** *For any simple graph $H = (X, F)$, there exists a $d$-pruned set $A$, such that for any $u \in X \setminus A$, $\deg_{H[X\setminus A]}(u) \geq d$; plus such $A$ can be computed in linear time.*

**Proof.** The set $A$ is constructed in the greedy manner: starting with $A \leftarrow \emptyset$, repeatedly check if there is a vertex $u \in X \setminus A$ such that $\deg_{H[X\setminus A]} < d$; if so, orient all edges incident on $u$ in $H[X \setminus A]$ away from $u$, and move $u$ to $A$. So, in this way, the out-degree is small $\deg^+_H(u) < d$. Clearly, the algorithm can be implemented in linear time. By the stopping condition, all vertices in the induced subgraph $H[X \setminus A]$ have degree at least $d$. ◀

Now, let us define a layering scheme of $H$.

▶ **Definition 9** (layering). *Let $h > 0$ be an integer parameter. A partition $(A_0, A_1, \cdots, A_{\lceil \log(n/h) \rceil})$ of $V$, together with an orientation $\mathcal{O}$ of all edges in $E$, is called a layering scheme of $H$, if the following requirements are satisfied.*
**(1)** *$(u, v) \in E$ such that $u \in A_i, v \in A_j, i < j$, this edge is oriented from $u$ to $v$, namely $\mathcal{O}(u, v) = 1$.*
**(2)** *For each index $b \geq 0$, $u \in A_b$, we have $\deg^+_{H,\mathcal{O}}(u) < 2^{b+1}h$.*

To initialize a layering of $H$, consider the following procedure: initialize $X = V$, and for $i = 0, 1, 2, \cdots, \lceil \log(n/h) \rceil - 1$, apply Lemma 8 on $H[X]$ with parameter $d = 2^i h$ to compute a set $A_i$, and then update $X \leftarrow X \setminus A_i$. After all iterations, define $A_{\lceil \log(n/h) \rceil} = X$ and orient edges in $E \cap (A_{\lceil \log(n/h) \rceil} \times A_{\lceil \log(n/h) \rceil})$ arbitrarily. By construction, for each $u \in A_b$, all its out-neighbors are in $\bigcup_{i \geq b} A_i$, and $\deg^+_{H,\mathcal{O}}(u) \leq 2^b h < 2^{b+1}h$.

On top of the layering scheme, we need to define a clustering scheme of vertices.

▶ **Definition 10** (clustering). *Given a layering* $(A_0, A_1, \cdots, A_{\lceil \log(n/h) \rceil})$ *of* $H$*, a set of tuples* $\{(\mathcal{C}_b, Y_b, Z_b)\}_{b\geq 1}$ *is called a* cluster structure *on layer* $(A_0, A_1, \cdots)$*, where each layer* $A_b, b \geq 1$ *can be divided into two parts* $A_b = Y_b \cup Z_b$*, if the following properties hold.*

**(1)** *$Y_b$ is partitioned into a collection of clusters $\mathcal{C}_b = \{C_1, C_2, \cdots\}$ of subsets, where each $C_i$ has size $\geq 2^{b-1}h + 1$ and is spanned by a star subgraph of $H$. For each $C_i$, its center is defined to be the center of a star subgraph that spans $C_i$.*

**(2)** *Each vertex $z \in Z_b$ is adjacent to some vertices in $Y_b \cup \bigcup_{i>b} A_i$.*

*Besides, define edge weight function $\omega$ associated with this cluster structure as follows.*

**(a)** *All edges of star graphs in $\mathcal{C}_b$ have weight $2b$.*

**(b)** *All non-star edges incident on $Y_b$ in $H[A_b]$, and edges between $A_b$ and $\bigcup_{i>b} A_i$ have weight $2b-1$.*

**(c)** *All edges in $H[Z_b]$ have weight $2b-2$.*

▶ **Lemma 11.** *Given any layering* $(A_0, A_1, \cdots, A_{\lceil \log(n/h) \rceil})$ *of* $H$*, a cluster structure* $\{(\mathcal{C}_b, Y_b, Z_b)\}_{b\geq 1}$ *can be computed in* $\tilde{O}(n^2)$ *time.*

**Proof.** For each $b \geq 1$, construct $Y_b$ in the greedy manner: starting with an empty set $Y_b = \emptyset$, whenever there exists a vertex $c \in A_b \setminus Y_b$ with $\deg_{H[A_b \setminus Y_b]}(u) \geq 2^b h$, take an arbitrary set of $2^b h$ neighbors $u_1, u_2, \cdots, u_{2^b h} \in A_b \setminus Y_b$ and add this cluster $\{c, u_1, u_2, \cdots, u_{2^b h}\}$ to $Y_b$, with $c$ being its center. In the end when the above procedure stops, define $Z_b = A_b \setminus Y_b$. Clearly this procedure takes $O(|F|) = O(n^2)$ time.

To verify property (2), by construction of the layering scheme, for each $z \in Z_b$, $\deg_{H[\cup_{i\geq b} A_i]}(z) \geq 2^b h$. So if $z$ was not added to $Y_b$ as a star center, then $z$ must be adjacent to some vertices from $Y_b \cup \bigcup_{i>b} A_i$. ◀

Next, we try to maintain the layering $(A_0, A_1, \cdots)$ together with a clustering structure when edges are being deleted from $G$. During the execution of our decremental algorithm, we need to ensure a basic requirement.

▶ **Invariant 12.** *During the decremental algorithm, vertices could only move from layers $A_{b+1}$ to $A_b$, $\forall b \geq 0$; in other words, vertices only move from upper layers to lower layers.*

The rest of the section would be devoted to the following statement.

▶ **Lemma 13.** *Suppose $H$ undergoes a sequence of edge deletions. Then a clustering structure $\{(\mathcal{C}_b, Y_b, Z_b)\}_{b\geq 1}$ together with an induced edge weight $\omega$ in Definition 10 can be explicitly maintained using total update time $\tilde{O}(\sum_{b=1}^{\lceil \log(n/h) \rceil} 2^b D_b h + n^2)$, plus that the dynamic algorithm meets Invariant 12; here $D_b$ is an upper bound on the number of edges which are incident on layer $A_b$ right when they get deleted.*

## 3.1 Maintaining clusters under edge deletions

In this subsection we describe the algorithm behind Lemma 13. Let us first focus on a fixed layer $A_b$. Assume the updates to $H[A_b]$ are either edge deletions or vertex transfers from $A_{b+1}$ to $A_b$; if $b = \lceil \log(n/h) \rceil$ then $A_{b+1}$ is always empty. Initialize an arbitrary clustering $(\mathcal{C}_b, Y_b, Z_b)$ as in Lemma 11. We will be maintaining all adjacency lists in the induced subgraph $H[\bigcup_{i\geq b} A_i]$, and more importantly, the following auxiliary data structure based on edge orientation $\mathcal{O}$.

**Auxiliary data structures based on edge orientation $\mathcal{O}$**

- **(i)** For each $u \in A_b$, a list of all out-neighbors in $H$. By Invariant 12, these out-neighbors are all in $\bigcup_{i \geq b} A_i$.
- **(ii)** For each $u \in A_b$, a list of neighbors $Z^-(u) = \{z \mid (u,z) \in E, \mathcal{O}(u,z) = -1, z \in Z_b\}$.
- **(iii)** For each $u \in Z_b$, a list of neighbors $Z^+(u) = \{z \mid (u,z) \in E, \mathcal{O}(u,z) = 1, z \in Z_b\}$.
- **(iv)** A priority queue on all vertices in $Z_b$, which are ordered by degrees $\deg_{H[Z_b]}(u) = |Z^-(u)| + |Z^+(u)|, \forall u \in Z_b$ in the induced subgraph $H[Z_b]$.

In other words, for each vertex in $Z_b$, it knows both its out-neighbors and in-neighbors in $H[Z_b]$, while for each vertex in $Y_b$, it only knows its in-neighbors in $Z_b$. Plus, all vertices in $Z_b$ are ordered by their degrees in the induced subgraph $H[Z_b]$.

**Vertex insertions.** Assume a new vertex $u$ has moved to $A_b$ from upper layer $A_{b+1}$. First, for each edge $(u,v)$ where $v \in \bigcup_{i>b} A_i$, we need to adjust the orientation of $\mathcal{O}(u,v) = 1$, reassign edge weights for all edges $\omega(u,v) = 2b-1$.

By definition of layering, before $u$ came downward from $A_{b+1}$, all its edges incident on $A_b$ were directed inward. So when it joins $A_b$, $Z^+(u)$ should be empty. After insertion of $u$, we add $u$ to $Z_b$, and initialize $Z^-(u)$ by scanning $u$'s adjacency list in $H$, and initialize $Z^+(u) = \emptyset$. Then, for each $v \in A_b$ adjacent to $u$, add $u$ to the list $Z^+(v)$. Next, update the keys of $u$ and its neighbors in $Z_b$ in the priority queue. Finally, if $v \in Y_b$, stay with $\omega(u,v) = 2b-1$, and otherwise reassign $\omega(u,v) = 2b-2$.

After that, repeat the following greedy procedure (1)(2)(3) to form star subgraphs out of $H[Z_b]$. For a more concise description of this procedure, check pseudo-code GreedyCluster.

- **(1)** If there exists $c \in Z_b$ such that $\deg_{H[Z_b]}(c) = |Z^-(c)| + |Z^+(c)| \geq 2^b h$, pick an arbitrary such vertex $c$; if there is none, then halt.
- **(2)** Find all of $c$'s neighbors $u_1, u_2, \cdots, u_k \in Z_b$ by scanning $Z^-(c) \cup Z^+(c)$. Then move all vertices $c, u_1, u_2, \cdots, u_k$ from $Z_b$ to $Y_b$, and add the star subgraph around all $u_i$'s centered at $c$ as a cluster to $\mathcal{C}_b$.
- **(3)** Lastly, we need to maintain all lists $Z^+(\cdot), Z^-(\cdot)$ and edge weights. To maintain lists and edge weights, for every vertex $x \in \{c, u_1, u_2, \cdots, u_k\}$, do the following steps.
  - **(a)** For each $z \in Z^-(x)$, remove $x$ from $Z^+(z)$; for each $z \in Z^+(x)$, remove $x$ from $Z^-(z)$. Then, for each such edge $(x,z)$, reassign $\omega(x,z) = 2b-1$, and for all star edges $(c,u_i), 1 \leq i \leq k$, assign $\omega(c,u_i) = 2b$.
  - **(b)** Scan the out-neighborhood of $x$, then for each $(x,y)$ such that $\mathcal{O}(x,y) = 1$ and $y \in Y_b$, remove $x$ from $Z^-(y)$.

**Non-star edge deletions.** Now assume an edge deletion $(u,v)$ occurs in subgraph $H[\bigcup_{i \geq b} A_i]$, and $u \in A_b$. Let us first study the simpler case where $(u,v)$ is not a star edge in any cluster of $\mathcal{C}_b$.

A preliminary step is updating these four lists $Z^+(u), Z^-(u), Z^+(v), Z^-(v)$ due to $(u,v)$'s deletion. After this, some vertex $z \in Z_b$ might have a small degree in subgraph $H[\bigcup_{i \geq b} A_i]$, and in this case we would have to move it to $A_{b-1}$ to ensure each $z \in Z_b$ is adjacent to some vertices in $Y_b \cup \bigcup_{i>b} A_i$. More specifically, while some vertices in $z \in Z_b$ have degree less than $2^b h$ in subgraph $H[\bigcup_{i \geq b} A_i]$, move $z$ from $Z_b$ to $A_{b-1}$, do the following steps to restore our data structures.

**Algorithm 1** GreedyCluster.

**while** $\exists c \in Z_b$ *such that* $\deg_{H[Z_b]}(c) \geq 2^b h$ **do**
  list all neighbors of $c$ in $H[Z_b]$ as $u_1, u_2, \cdots, u_k$;
  form a star subgraph around $u_1, u_2, \cdots, u_k$ centering at $c$, and add this as a cluster to $\mathcal{C}_b$;
  for each $x \in \{c, u_1, u_2, \cdots, u_k\}$, scan $Z^-(x), Z^+(x)$ to maintain all other affected lists $Z^-(z), Z^+(z), z \in Z_b$, and update the induced edge weights $\omega$;
  for each $x \in \{c, u_1, u_2, \cdots, u_k\}$, scan its out-neighbors in $H[A_b]$ to update all $Z^-(y), \forall y \in Y_b$;

**(1)** Scan $z$'s adjacency list, and for each of its neighbor $y \in A_b$, remove $z$ from $Z^+(y), Z^-(y)$, and update the orientation $\mathcal{O}(z, y) = 1$ if necessary. Finally, remove $z$ from the priority queue on $Z_b$.
**(2)** Besides, for each of $z$'s neighbor $x$ in $H[\bigcup_{i \geq b} A_i]$, reassign $\omega(z, x) = 2b - 3$.

The pseudo-code of the above procedure is presented below as GreedyPrune.

**Algorithm 2** GreedyPrune.

**while** $\exists z \in Z_b$ *such that* $\deg_{H[\bigcup_{i \geq b} A_i]}(z) < 2^b h$ **do**
  move $z$ from $Z_b$ to $A_{b-1}$, and scan its adjacency list to update all affected lists $Z^+(\cdot), Z^-(\cdot)$, and update $\omega$ and $\mathcal{O}$ properly;

**Star edge deletions.** Now consider the harder case where the deleted edge is a star edge of a cluster in $\mathcal{C}_b$. In this case, let $(c, u)$ be the deleted edge and $c$ be the cluster center. After the edge deletion, one or more vertices in the cluster centered at $c$ might have to leave $Y_b$. The two possibilities are the following.

- Excluding $u$, if the star subgraph now has at most $2^{b-1}h$ vertices, the whole cluster should be destroyed and removed from $Y_b$ back to $Z_b$.
- Otherwise, only $u$ needs to join $Z_b$.

Let vertex subset $S$ be the set of all vertices in this cluster that might join $Z_b$ shortly, so $S$ is either a singleton $\{u\}$ or the whole star cluster. For each vertex $v \in S \setminus \{c\}$, a preliminary step is reweighing $\omega(c, v) = 2b - 1$ as $(c, v)$ is no longer a star edge.

Go over all vertices $v \in S$ in an arbitrary order and do the following. Scan the list of all out-neighbors of $v$ to find the set $S_v$ of its out-neighbors in $Z_b$; remember that currently we do not maintain $Z^+(v)$ as $v \notin Z_b$, so instead of directly retrieving $S_v = Z^+(v)$, we have to scan all of $v$'s out-neighbors. Now, as we have already maintained $Z^-(v)$, we can now decide whether $v$ has at least $2^b h$ (undirected) neighbors in $Z_b$ by checking if $|S_v| + |Z^-(v)| \geq 2^b h$. We need to study both possibilities.

- $|S_v| + |Z^-(v)| < 2^b h$. In this case, set $Z^+(v) = S_v$. Move $v$ from $Y_b$ to $Z_b$. To restore orientation-based data structures, first, for each $z \in Z^-(v)$ add $v$ to $Z^+(z)$, and for every $z \in S_v$ add $v$ to $Z^-(z)$; second, scan the out-neighbors of $v$, and for every $(v, y)$ such that $y \in Y_b$, add $v$ to $Z^-(y)$.
  Finally, to restore edge weights, for each of $z \in S_v \cup Z^-(v)$, change the edge weight $\omega(v, z)$ from $2b - 1$ to $2b - 2$.

- $|S_v| + |Z^-(v)| \geq 2^b h$. In this case, we would directly collect a star cluster around $v$ using $S_v \cup Z^-(v)$. Namely, move all vertices in $S_v \cup Z^-(v)$ from $Z_b$ to $Y_b$, and add the star subgraph centered at $v$ as a cluster to $\mathcal{C}_b$. To restore the auxiliary data structures and edge weights, go over all vertices $x \in S_v \cup Z^-(v)$ and follow the same steps (3)(a) and (3)(b) when handling vertex insertions.

After we have iterated over all vertices from $S$, each $v \in S$ either has entered $Z_b$ or formed a new cluster in $\mathcal{C}_b$. Subgraph $H[Z_b]$ could still contain vertices whose degree is at least $2^b h$, so we perform another round of GreedyCluster to exhaustively collect new clusters out of $Z_b$. Finally, to ensure property (2) in Definition 10, we invoke GreedyPrune on $Z_b$ to remove vertices with small degrees.

**Stacking all layers.** In order to maintain the entire clustering structures $\{(\mathcal{C}_b, Y_b, Z_b)\}_{b\geq 1}$ across all layers $(A_0, A_1, \cdots)$ against edge deletions to $H$, we simply apply the above algorithm for each layer $A_b, b \geq 1$. When vertices move from upper layers to lower layers, it is equivalent to vertex insertions to lower layers, which can be handled by the algorithm.

## 3.2 Proof of correctness

The proof of correctness of our algorithm is divided into several lemmas.

▶ **Lemma 14.** *Invariant 12 is preserved by the algorithm, and $(A_0, A_1, \cdots)$ is always a layering satisfying the requirements in Definition 9.*

**Proof.** During the algorithm, vertices never move from lower levels $A_b$ to higher levels $A_{b+1}$, so Invariant 12 is satisfied.

Now let us turn to verify properties of Definition 9. For property (1), when vertices move across different layers, our algorithm always adjusts $\mathcal{O}$ to ensure this requirement, so this property holds. As for property (2), by the algorithm description, whenever a vertex $z \in Z_b$ moves from $A_b$ to $A_{b-1}$, it must be the case that $\deg_{H[\bigcup_{i\geq b} A_i]}(z) < 2^b h$. In the near future, as long as $z$ stays in $A_{b-1}$, its out-neighbors under orientation $\mathcal{O}$ should always be a subset of its current neighbors in $\bigcup_{i\geq b} A_i$, and so the number of its out-neighbors is always bounded by $2^b h$, and thus property (2) holds. ◀

▶ **Lemma 15.** *During edge deletions, the algorithm correctly maintains the auxiliary data structures based on the edge orientation $\mathcal{O}$.*

**Proof.** Maintaining part (i)(iv) is straightforward, so we only focus on part (ii)(iii).

- **Vertex insertions.** $Z_b$ could only change during GreedyCluster. Consider any new cluster $c, u_1, \cdots, u_k$ centered at $c$. To move them from $Z_b$ to $Y_b$, for each $x \in \{c, u_1, \cdots, u_k\}$, on the one hand, we need to go over all of its neighbors in $H[Z_b]$ to fix lists $Z^-(z), Z^+(z)$ for all $z \in Z_b$; on the other hand, we only need to scan $x$'s out-neighbors to fix lists $Z^-(y)$ for all $y \in Y_b$, since we do not require $Z^+(\cdot)$ for vertices in $Y_b$, which is a key point behind the design of the auxiliary data structures.
- **Non-star edge deletions.** In this case, some vertices might move from $Z_b$ downward to $A_{b-1}$. As the maintenance of the auxiliary data structures is done in the straightforward manner, so correctness should follow easily.
- **Star edge deletions.** This case involves two rounds of vertex transfers between $Y_b$ and $Z_b$. The first round is when it enumerates all vertices $v \in S$ and check if $|S_v| + |Z^-(v)| < 2^b h$. In this case, we have discussed two possibilities.

- If $|S_v| + |Z^-(v)| < 2^b h$, then the algorithm moves $v$ from $Y_b$ to $Z_b$. To restore part (ii)(iii), it suffices to scan $Z^-(v)$ and all out-neighbors of $v$, which is what the algorithm does.
- Otherwise if $|S_v| + |Z^-(v)| \geq 2^b h$, the algorithm would collect a star cluster around $v$ with $S_v \cup Z^-(v)$. To move $S_v \cup Z^-(v)$ to $Y_b$, we would use a similar procedure as steps (3)(a)(b) of vertex insertions, and so the auxiliary data structures should be maintained correctly as well.

The second round is when it invokes GreedyCluster to further remove vertices $z \in Z_b$ such that $\deg_{H[Z_b]}(z) \geq 2^b h$. After that, the algorithm calls GreedyPrune to remove vertices $z \in Z_b$ such that $\deg_{H[\bigcup_{i \geq b} A_i]}(z) < 2^b h$. Using the same analysis as before, we know the algorithm always correctly maintains the auxiliary data structures. ◀

▶ **Lemma 16.** *A cluster structure $\{(\mathcal{C}_b, Y_b, Z_b)\}_{b \geq 1}$ from Definition 10 is correctly maintained by our algorithm.*

**Proof.** It is straightforward to verify that our algorithm correctly maintains the edge weight $\omega$. So let us only focus on properties (1)(2).

Property (1) is automatically guaranteed by the algorithm, since the algorithm only collects star clusters of size $\geq 2^b h$, and whenever a star cluster becomes smaller than $2^{b-1} h$ after losing too many of its leaves due to edge deletions, the algorithm would try to move it back to $Z_b$. Now, let us turn to property (2).

▷ Claim 17. After every update, it holds that for each vertex $z \in Z_b$, $\deg_{H[\bigcup_{i \geq b} A_i]}(z) \geq 2^b h$, $\deg_{H[Z_b]}(z) < 2^b h$.

Proof of claim. When the cluster structure has just been initialized, this claim holds by construction of the layers $(A_0, A_1, \cdots)$. Next, consider vertex insertions and edge deletions separately.

- **Vertex insertions.** After a vertex insertion from upper layers, the degrees of vertices in $Z_b$ in subgraph $H[\bigcup_{i \geq b} A_i]$ remains unchanged. By the GreedyCluster procedure, all vertices in $Z_b$ should have degree less than $2^b h$ in $H[Z_b]$ afterwards.
- **Edge deletions.** After an edge deletion, by the end of the algorithm, it performs one round of GreedyCluster and then GreedyPrune, which first moves vertices in $Z_b$ whose degree in $H[Z_b]$ is at least $2^b h$ to $Y_b$, and then moves vertices whose degree in $H[\bigcup_{i \geq b} A_i]$ is less than $2^b h$ downward to $A_{b-1}$. So the claim should hold when it finishes. ◀

By this claim, for any vertex $z \in Z_b$, on the one hand we know $\deg_{H[\bigcup_{i \geq b} A_i]}(z) \geq 2^b h$, and on the other hand $\deg_{H[Z_b]}(z) < 2^b h$, so there exists $y \in \bigcup_{i \geq b} A_i \setminus Z_b = Y_b \bigcup_{i > b} A_i$ adjacent to $z$, which is property (2). ◀

## 3.3 Running time analysis

▶ **Lemma 18.** *The total time of the algorithm for handling layer $A_b$ is bounded by $O(2^b D_b h + n^2)$.*

**Proof.** Let us first analyze the total time of GreedyPrune throughout edge deletions. During this subroutine, the algorithm repeatedly picks $z \in Z_b$ such that $\deg_{H[\bigcup_{i \geq b} A_i]}(z) < 2^b h$ and move it to $A_{b-1}$. This requires scanning the adjacency list of $z$. Since each $z$ can transfer from $A_b$ to $A_{b-1}$ for at most once, the total time is bounded by $O(|F|) = O(n^2)$.

For each cluster $C \in \mathcal{C}_b$, define $\mathsf{del}(C)$ to be the current number of star edges that have been deleted since $C$ joined $\mathcal{C}_b$. Define a potential function, where $K$ is a large constant:

$$\Phi = |F \cap (Z_b \times Z_b)| + 2^b h|Z_b| + K \cdot 2^b h \sum_{C \in \mathcal{C}_b} \mathsf{del}(C) = O(K \cdot 2^b D_b h + n^2)$$

▷ Claim 19. The total time of GreedyCluster across all updates is bounded by $O(2^b h D_b + n^2)$.

Proof of claim. In the while-loop, locating a $c \in Z_b$ such that $\deg_{H[Z_b]}(c) = |Z^-(c)| + |Z^+(c)| \geq 2^b h$ takes $O(\log n)$ time using the priority queue. By the algorithm, forming a cluster around $c$ requires scanning all neighbors in $H[Z_b]$ for each $x \in \{c, u_1, u_2, \cdots, u_k\}$, plus $x$'s out-neighbors in $H[\bigcup_{i \geq b} A_i]$. Suppose the total number of edges in $H[Z_b]$ incident on $\{c, u_1, u_2, \cdots, u_k\}$ is $m_0$. Then, on the one hand, the time cost would be $O(m_0 + 2^b h(k+1))$, as each $x$ has out-neighbors at most $2^{b+1}h$ by Definition 9; on the other hand, the potential decrease of $\Phi$ is exactly $m_0 + 2^b h(k+1)$ as well, which cancels out the time cost. Hence, the total time of GreedyCluster across all updates is bounded by $O(2^b h D_b + n^2)$. ◁

A dominant part of the overall running time is handling star edge deletions, where we need to iterate over the set $S$. Let $C$ be the star cluster centered at $c$. For any $v \in S$, computing the number of $v$'s neighbors in $Z_b$ takes time at most $2^{b+1}h$ as out-degree is bounded $\deg^+_{H,\mathcal{O}}(v) \leq 2^{b+1}h$ and $Z^-(v)$ is accessible as a list. Now consider two possibilities.

- $|S_v \cup Z^-(v)| < 2^b h$. In this case, the algorithm moves $v$ back to $Z_b$ by scanning $S_v \cup Z^-(v)$ plus all out-neighbors, and the running time is bounded by $O(2^b h)$. As for potential change, moving $v$ to $Z_b$ increases $\Phi$ by $|S_v \cup Z^-(v)| + 2^b h \leq 2^{b+1}h$, so the amortized cost is bounded by $O(2^b h)$.
- $|S_v \cup Z^-(v)| \geq 2^b h$. In this case, we would collect a new cluster around $v$. Similar to the analysis of GreedyCluster, the amortized cost of this part is zero.

If $S = \{v\}$, then this part of the amortized cost incurred by a star edge deletion is bounded by $O(2^b h)$. Plus that $\mathsf{del}(C)$ increases by 1, the amortized cost would be $K 2^b h$.

Otherwise, if $S$ is the entire cluster centered at $c$, then on the one hand, summing up the above two cases, the amortized cost of iterating over $S$ is $O(2^b h|S|)$; on the other hand, since the star $C$ centered at $c$ is canceled from $\mathcal{C}_b$, the potential decrease would be $K \cdot 2^b h \cdot \mathsf{del}(C)$. So the amortized cost of handling $S$ would be $O(2^b h|S|) - K \cdot 2^b h \cdot \mathsf{del}(C)$.

A key point is that, by the algorithm, when $C$ joined $\mathcal{C}_b$ it had at least $2^b h$ leaves by then, but now when $C$ leaves it has $2^{b-1}h$ leaves. Therefore, $\mathsf{del}(C) \geq |S|$. Hence, if we choose $K$ to be a sufficiently large constant, the amortized cost $O(2^b h|S|) - K \cdot 2^b h \cdot \mathsf{del}(C)$ would be negative.

To summarize, we have proved that each star edge deletion has amortized cost at most $K 2^b h$, plus that other costs are bounded by $O(2^b h D_b + n^2)$. So the total update time is $O(2^b h D_b + n^2)$. ◀

## 4 The main algorithm

**Initialization.** Let $f$ be an empty flow, and we will keep augmenting $f$ throughout $O(n^{2/3})$ iterations. Apply Lemma 13 on $G$ to initialize a layering $(A_0, A_1, \cdots)$ as well as a clustering $\{(\mathcal{C}_b, Y_b, Z_b)\}_{b \geq 1}$ together with edge weights $\omega$, where $h = \lceil \tau^{1/2} \rceil$; we can enforce $s, t \in A_0$ at the beginning since this would only increase $O(n)$ edges incident on $A_0$. We maintain two disjoint edge sets $F_0$ and $F_1$, where initially $F_0$ includes all edges in $G[V \setminus A_0]$, and let $F_1$ be the rest. So initially $F_1$ contains at most $O(nh)$ edges.

Throughout the iterations, $(V, F_0 \cup F_1)$ will be the residual graph $G_f$, and $F_0$ will be the set of all undirected edges used by the clustering structure, and $F_1$ could contain both undirected and directed edges in the residual graph $G_f$.

**Iterations.** In each iteration, we try to find a blocking flow with respect to $f$ with increasing path (unweighted) length. Denote by $G_f$ the residual graph with respect to $f$.

For each edge $(u, v) \in F_0 \cup F_1$, associate it with a binary weight $\mu$ such that $\mu(u, v) = 0$ if $(u, v) \in F_0$, and $\mu(u, v) = 1$ otherwise. To keep track of all connected components of $(V, F_0)$, maintain a fully dynamic maximum weight spanning forest $\mathsf{MSF}$ on $(V, F_0)$ according to weights $\omega$ using standard approach [8], and for each $u \in V$, let $\mathsf{mst}(u)$ denote the maximum weight spanning tree in $\mathsf{MSF}$ containing $u$ (if $u \in A_0$, then $\mathsf{mst}(u)$ refers to $u$ itself).

The following invariant regarding the $s$-$t$ distance in $G_f$ under edge weight $\mu$ will be guaranteed by our algorithm.

▶ **Invariant 20.** *At the beginning of the $l$-th iteration, the $s$-$t$ distance in $G_f$ is at least $l$.*

At the beginning of each iteration, perform Dijkstra's algorithm on $G_f = (V, F_0 \cup F_1, \mu)$ from $s$, and so each vertex $u \in V$ has a distance label $\mathsf{level}(u)$ which is set to the distance from $s$ to $u$ under edge weight $\mu$. Since all edges in $F_0$ are undirected and have zero $\mu$-weight, for shortest paths computations we could contract all edges in $F_0$, and so Dijkstra's algorithm takes time $O(|F_1| + n \log n)$; plus, vertices in any tree component in $\mathsf{MSF}$ can share the same label.

Later on when we search for augmenting paths, more edge weights $\mu(\cdot)$ would turn from 0 to 1 as edges are deleted from $F_0$. However, during this iteration, we will keep the labels $\mathsf{level}(\cdot)$ unchanged and ensure the following property.

▶ **Invariant 21.** *For any vertex $u \in V$, the distance from $s$ to $u$ in the residual graph $G_f$ under edge weight $\mu$ is always at least $\mathsf{level}(u)$.*

**Depth first search.** Next, let focus on a single iteration. Each vertex in $V$ has two phases: active or inactive. At the beginning of the search, activate all vertices whose label is $< l$ as well as terminal $t$. Starting with source vertex $u = s$. Assume inductively that we have found a sequence of vertices $s = u_0, u_1, \cdots, u_k$. As long as $k < l$ and $s$ is active, repeat the following steps.

**(1)** Enumerate all vertices in $v \in \mathsf{mst}(u_k)$. For each $v$, try to find an edge $(v, w)$ in $G_f$ such that:
  **(a)** $w$ is active, and $\mathsf{level}(w) = \mathsf{level}(v) + 1$.
  **(b)** Either the edge $(v, w)$ is undirected or it is from $v$ to $w$.
  If such an edge $(v, w)$ is found, then assign $u_{k+1} \leftarrow w, k \leftarrow k + 1$.

**(2)** If no such edge $(v, w)$ can be found, then deactivate all vertices in $\mathsf{mst}(u_k)$ and $k \leftarrow k - 1$.

Now, suppose at some point $k = l$. Since $u_k$ is active, we know $u_k = t$. Then we try to send flows from $s$ to $t$ in $G_f$ on an augmenting path of length $l$ under edge weights $\mu$. Recalling the way we found all of $u_0, u_1, \cdots, u_l$, each $\mathsf{mst}(u_i)$ and $\mathsf{mst}(u_{i+1})$ are connected by an edge $(v_{i-1}, u_i)$ with positive capacity in $G_f$, and each pair of $u_i, v_i$ are in the same tree component of $(V, F_0)$, so we can route one unit of flow from $s$ to $t$ going through $u_0, u_1, \cdots, u_l$.

Let the augmenting path be $s = p_0, p_1, \cdots, p_e = t$, and augment $f$ by sending one unit of flow along this path. After that, we need to update the residual graph $G_f$, the subgraph $H$ on which the layered clustering is maintained, the maximum spanning forest $\mathsf{MSF}$. Go over all $0 \le i < e$, and consider the following two cases.

- If $(p_i, p_{i+1})$ is an undirected edge in $F_0$, then delete it from $F_0$, and add a directed edge $(p_{i+1}, p_i)$ to $F_1$ with residual capacity 2.
- If $(p_i, p_{i+1})$ is an undirected edge in $F_1$, then it is connecting two vertices with consecutive labels. In this case, add a directed edge $(p_{i+1}, p_i)$ to $F_1$ with residual capacity 2, and delete the old edge from $F_1$.
- If $(p_i, p_{i+1})$ is a directed edge in $F_1$, then remove it from $F_1$ and add an undirected edge $(p_i, p_{i+1})$ with unit capacity to $F_1$.

After $G_f$ is updated, we should reset $k \leftarrow 0$, and update the layered clustering structure, the maximum spanning forest accordingly.

The whole blocking flow computation is summarized as pseudo-code BlockingFlow.

**Algorithm 3** BlockingFlow$(f, l)$.

```
maintain layered clustering {(C_b, Y_b, Z_b)}_{b≥1}, maximum spanning forest MSF;
compute distance labels level(·);
activate all vertices whose label is < l as well as terminal t;
u_0 ← s, k ← 0;
while s is active do
    if k < l then
        if exists (v, w) such that w is active, v ∈ mst(u_k), level(w) = level(v) + 1 then
            u_{k+1} ← w and k ← k + 1;
        else
            deactivate the entire mst(u_k) and backtrack k ← k − 1;
    else
        send one unit of s-t flow;
        reset k ← 0, update residual graph, layered clustering, maximum spanning
         forest;
```

Now we can summarize our main algorithm as a piece of pseudo-code MaxFlow below.

**Algorithm 4** MaxFlow$(G, \tau)$.

```
initialize empty flow f;
initialize a layered clustering structure on G parameterized by h = ⌈τ^{1/2}⌉;
for l = 1, 2, ··· , ⌈2n^{2/3}⌉ do
    call BlockingFlow(f, l);
decycle f using Lemma 4;
apply Lemma 5 on f exhaustively;
return f;
```

## 4.1 Proof of correctness

▶ **Lemma 22.** *Invariant 21 is preserved after $G_f$ is updated.*

**Proof.** When a connected component in MSF is split due to edge deletions in $F_0$, the current distances in $G_f$ cannot increase. If vertices move from $V \setminus A_0$ to $A_0$ due to degree losses, some inter-component edges would be inserted to $G_f$. However, such kind of edge insertions can never decrease distances, as they were connecting vertices with the same label value level$(\cdot)$. ◀

After the depth-first search is completed, we need to prove that the distance from $s$ to $t$ in the contracted graph $G_f$ is at least $l+1$, namely preserving Invariant 20 for the next iteration.

▶ **Lemma 23.** *After the depth-first search which augments $f$, the distance from $s$ to $t$ in the residual graph $G_f$ under edge weights $\mu$ is at least $l+1$.*

**Proof.** Suppose after all the augmentations by a blocking flow, there still exists an $s$-$t$ flow with length at most $l$ under edge weights $\mu$. So there exists a path $s = p_0, p_1, \cdots, p_e = t$ in $G_f$ such that $e \leq l$. Consider two possibilities.

- $\mathsf{level}(p_i) = i, \forall 0 \leq i \leq e$. Then it must be $e = l$. For any edge $(p_i, p_{i+1})$, the capacity from $p_i$ to $p_{i+1}$ should always be positive throughout the depth-first search, since the algorithm never pushes flows from level $i+1$ to $i$. Hence, the depth-first search could not have terminated, which is a contradiction.
- There exists $0 \leq i < e$ such that $\mathsf{level}(p_{i+1}) > \mathsf{level}(p_i) + 1$. By the algorithm, the depth-first search never adds flows directly across nonconsecutive levels, so the capacity from $p_i$ to $p_{i+1}$ should be positive at the beginning as well. Then, by the shortest paths computation, $\mathsf{level}(p_{i+1})$ should be at most $\mathsf{level}(p_i) + 1$, which makes a contradiction as well. ◀

By the above lemma together with Lemma 4.6 from [4], we have the following corollary.

▶ **Corollary 24** ([4]). *After $2n^{2/3}$ iterations, the residual flow of $f$ (namely, maximum flow in $G_f$) becomes at most $n^{2/3}$.*

**Proof.** By the pigeon-hole principle, there exists a pair of consecutive levels in the residual graph $G_f$ whose union contains at most $n^{1/3}$ vertices. Therefore, the capacity of the cut between these two levels is at most $2 \cdot (\frac{n^{1/3}}{2})^2 < n^{2/3}$. ◀

## 4.2 Running time analysis

Finally, let us analyze the running time of our max flow algorithm. To analyze the total time of maintaining the layered clusters, first we need some properties regarding the maximum spanning forest $\mathsf{MSF}$.

▶ **Lemma 25.** *For each index $b$, all star edges in $\mathcal{C}_b$ are tree edges in the maximum spanning forest. Plus, for each vertex $z \in Z_b$, there exists $y \in Y_b \cup \bigcup_{i>b} A_i$ such that $(y, z)$ is also a tree edge.*

**Proof.** Consider any star edge $(c, u)$ with $c$ being the center. If $(c, u)$ is not a tree edge, then the tree path between $u, c$ connects $u$ to another vertex $v \neq c$. By definition of $\omega$, $\omega(u, v) \leq 2b - 1$, so switching $(c, u)$ for $(u, v)$ in the spanning forest gives a strictly larger total weight, which makes a contradiction.

Now consider the second half of the statement. By property (2) of the clustering structure, there exists $y \in Y_b \cup \bigcup_{i>b} A_i$ adjacent to $z$, and so $\omega(y, z) = 2b-1$. If $(y, z)$ is not a tree edge, then there exists a tree path that connects $z$ to $y$. Consider the first edge $(z, w)$ on this path. As the spanning forest has maximum total weight, it must be $\omega(z, w) \geq \omega(z, y) = 2b - 1$. As $z \notin Y_b$, it can only be the case that $\omega(z, w) = 2b - 1$, and so $w$ cannot be in $Z_b$ or $\bigcup_{i<b} A_i$. Hence, $w \in Y_b \bigcup_{i>b} A_i$, which completes the proof. ◀

▶ **Corollary 26.** *For each $u \in A_b$, there exists a tree path of at most $\log n$ edges connecting $u$ to a star center in $\bigcup_{i \geq b} Y_i$.*

▶ **Lemma 27.** *For each index $b$, and for each spanning tree $T \in \mathsf{MSF}$, suppose it contains $k$ centers in $\bigcup_{i \geq b} Y_i$, then any tree path of $T$ contains at most $O(k \log n)$ vertices in $A_b$.*

**Proof.** Suppose otherwise a tree path contains $20k \log n$ vertices in $A_b$, then there exists a sub-sequence of this tree path $u_1, u_2, \cdots, u_{2k}$ vertices in $A_b$, such that for each $1 \leq j < 2k$, the tree path from $u_j$ to $u_{j+1}$ has at least $9 \log n$ tree edges. By Corollary 26, there exists a center $v_j \in \bigcup_{i \geq b} Y_i$ such that $u_j, v_j$ are connected by a tree path of length at most $\log n$ in $\mathsf{MSF}$. By the pigeon-hole principle, there exists distinct indices $j_1 < j_2$ such that $v_{j_1} = v_{j_2}$, which means $u_{j_1}, u_{j_2}$ are connected by a tree path of at most $2 \log n$ edges, contradiction. ◀

▶ **Lemma 28.** *The overall running time throughout all iterations of maintaining the layered clusters*

$$\{(\mathcal{C}_b, Y_b, Z_b)\}_{b \geq 1}$$

*together with its induced edge weight $\omega$ is bounded by $\tilde{O}(n^2 + n\tau)$. Also, the total update time of maintaining the maximum spanning forest $\mathsf{MSF}$ is also bounded by $\tilde{O}(n^2 + n\tau)$.*

**Proof.** Consider any unit flow that we send from $s$ to $t$ during this round. Focus on the part within the same component of $\mathsf{MSF}$. By the routing scheme, the flow always uses the tree edges of the maximum spanning tree. By Lemma 27, the length of the tree path is at most proportional to the number of centers. Therefore, for the $b$-th layer, the number of vertices in $A_b$ on the unit flow is bounded by $O(\frac{n}{2^b h} \log n)$. Since the augmenting path is a simple path, the number of edge deletions incident on $A_b$ is at most $O(\frac{n}{2^b h} \log n)$ as well. Summing over at most $\tau$ augmentations, the total edge deletions incident on $A_b$ is at most $\tilde{O}(\frac{n\tau}{2^b h})$. By Lemma 13, the total update time of maintaining the layered clustering is $\tilde{O}(n^2 + n\tau)$.

As for the dynamic maximum spanning forest $\mathsf{MSF}$, the number of changes to $G_f$ and $\omega$ is always upper bounded by the time to maintain the layered clustering, so the total time of maintaining $\mathsf{MSF}$ is bounded similarly. ◀

Next we bound the total number of new inter-component edges added to $F_1$ throughout all $2n^{2/3}$ iterations for a single round.

▶ **Lemma 29.** *During the for-loop of MaxFlow, the total number of edges in $F_1$ is bounded by $O(n\tau^{1/2} \log n)$.*

**Proof.** Similar to the previous lemma, we can prove that each augmentation turns at most $O(n/h \log n)$ undirected edges from $F_0$ to be directed edges in $F_1$. Since $F_1$ contains at most $O(nh)$ initially before the first iteration, the overall edges to $F_1$ is bounded by $O(nh + \frac{n\tau \log n}{h}) = O(n\tau^{1/2} \log n)$. ◀

The above lemma helps us bounding the total time of computing blocking flows through all iterations.

▶ **Lemma 30.** *The running time of a single iteration takes time $\tilde{O}(n\tau^{1/2})$.*

**Proof.** First, invoking Dijkstra's algorithm takes time $O(|F_1| + n \log n) = \tilde{O}(n\tau^{1/2})$. Next, we need to argue that in each iteration, the blocking flow procedure takes time $\tilde{O}(|F_1| + n)$ as well. In fact, we claim that each edge $(v, w) \in F_1$ can be visited by at most twice during the depth-first search. If $\mathsf{level}(w) \neq \mathsf{level}(v) + 1$, then the algorithm does to need to consider it when searching for augmenting paths, as $\mathsf{level}(\cdot)$ does not change within this iteration. Now let us assume $\mathsf{level}(w) = \mathsf{level}(v) + 1$. If $(v, w)$ is a directed edge, then after two augmentations it would be a directed edge from $w$ to $v$, and so it would never be visited again; if $(v, w)$ is an undirected edge, then after one augmentation it would be a directed one from $w$ to $v$ as well.

Finally, we efficiently enumerate edges incident on any tree in MSF under edge deletions, we could arrange all vertices in this tree according to the Euler-tour which supports fast link and cut operations on trees. ◀

By the above lemmas, the total time of for-loop is bounded by $\tilde{O}(n^2 + n\tau + n^{5/3}\tau^{1/2}) = \tilde{O}(n^{5/3}\tau^{1/2})$. After the while-loop, the total residual flow is bounded by $O(n^{2/3})$, so by Lemma 5 the rest takes time $\tilde{O}(n^{5/3}\tau^{1/2})$ as well.

## References


1. Julia Chuzhoy, Yu Gao, Jason Li, Danupon Nanongkai, Richard Peng, and Thatchaphol Saranurak. A deterministic algorithm for balanced cut with applications to dynamic connectivity, flows, and beyond. In *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1158–1167. IEEE, 2020.
2. Julia Chuzhoy and Thatchaphol Saranurak. Deterministic algorithms for decremental shortest paths via layered core decomposition. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2478–2496. SIAM, 2021.
3. Efim A Dinic. Algorithm for solution of a problem of maximum flow in networks with power estimation. In *Soviet Math. Doklady*, volume 11, pages 1277–1280, 1970.
4. Ran Duan. Breaking the $\mathcal{O}(n^{2.5})$ deterministic time barrier for undirected unit-capacity maximum flow. In *Proceedings of the twenty-fourth annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1171–1179. SIAM, 2013.
5. Lester Randolph Ford Jr and Delbert Ray Fulkerson. *Flows in networks*. Princeton university press, 2015.
6. Andrew V Goldberg and Satish Rao. Beyond the flow decomposition barrier. *Journal of the ACM (JACM)*, 45(5):783–797, 1998.
7. Andrew V Goldberg and Robert E Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM (JACM)*, 35(4):921–940, 1988.
8. 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 (JACM)*, 48(4):723–760, 2001.
9. Jacob Holm, Eva Rotenberg, and Christian Wulff-Nilsen. Faster fully-dynamic minimum spanning forest. In *Algorithms-ESA 2015*, pages 742–753. Springer, 2015.
10. David R Karger. Using random sampling to find maximum flows in uncapacitated undirected graphs. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 240–249, 1997.
11. David R Karger and Matthew S Levine. Finding maximum flows in undirected graphs seems easier than bipartite matching. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 69–78, 1998.
12. David R Karger and Matthew S Levine. Random sampling in residual graphs. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 63–66, 2002.
13. Tarun Kathuria. A potential reduction inspired algorithm for exact max flow in almost $\tilde{O}(m^{4/3})$ time. *arXiv preprint*, 2020. `arXiv:2009.03260`.
14. Yin Tat Lee and Aaron Sidford. Path finding methods for linear programming: Solving linear programs in $\tilde{O}(\sqrt{rank})$ iterations and faster algorithms for maximum flow. In *2014 IEEE 55th Annual Symposium on Foundations of Computer Science*, pages 424–433. IEEE, 2014.
15. Yang P Liu and Aaron Sidford. Faster divergence maximization for faster maximum flow. *arXiv preprint*, 2020. `arXiv:2003.08929`.
16. Yang P Liu and Aaron Sidford. Faster energy maximization for faster maximum flow. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, pages 803–814, 2020.

**17** Aleksander Madry. Navigating central path with electrical flows: From flows to matchings, and back. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pages 253–262. IEEE, 2013.

**18** Aleksander Madry. Computing maximum flow with augmenting electrical flows. In *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 593–602. IEEE, 2016.

**19** Jan van den Brand, Yin Tat Lee, Yang P Liu, Thatchaphol Saranurak, Aaron Sidford, Zhao Song, and Di Wang. Minimum cost flows, mdps, and *ell*_1-regression in nearly linear time for dense instances. *arXiv e-prints*, 2021. `arXiv:2101.05719`.