Approximate MaxFlow in undirected graphs: a survey of [Pen16]

Eliezer Zimble Uni:ez2313

May 16, 2023

Abstract

Recent progress in approximating maximum flow problems for undirected graphs has achieved $O(m^{1+o(1)})\epsilon^{-2}$ time through the use of congestion-approximators [She13] and oblivious routing schemes [KLOS13]. [RST14] demonstrated that in turn approximate maximum flows can be employed to more effectively construct oblivious routing schemes/congestion approximators. The resulting interdependency creates a "chicken-and-egg" setup: calling oblivious routing schemes/congestion approximators can produce approximate maximum flow at a cost of $polylog(n)$, and vice versa. The setup creates a bottleneck as the initial calls cost $O(m^{1+o(1)})$ time. We survey the approach of [Pen16] in resolving the interdependency by combining the above techniques together with ultra-sparsification. The recursive size reductions avoid the costly initial calls and ultimately achieve an improved $O(m\text{polylog}(n))$ time for approximate MaxFlow in undirected graphs.

1 Introduction

1.1 Background and Initial Problem Statement:

The maximum flow problem and the dual min-cut problem are fundamental graph theory problems whose study has been influential in developing many algorithmic tools such as augmenting paths, scaling algorithms, graph sparisification and fast Laplacian solvers.
Recent $(1+\epsilon)$ MaxFlow approximation algorithms have achieved $O(m^{1+o(1)})$ time by using congestion-approximators [She13] and oblivious routing schemes [KLOS13]

Throughout this paper, we will survey the results of [Pen16] which offer an improved runtime of $O(m\text{polylog(n)})$ for approximate maximum flow.

Initial Problem statement:
We will focus on the problem of approximate maximum flow across undirected graphs with demands for each vertex. Namely:

Consider a capacitated undirected graph $G = (V, E, \mathbf{u})$ with $|V| = n$ vertices, $|E| = m$ edges and edge capacity vector $\mathbf{u}$ where $\mathbf{u}_e = c_e$ for $e \in E$. A flow vector $\mathbf{f} \in \mathbb{R}^{\mathbf{m}}$ satisfies a demand vector $\mathbf{b} \in \mathbb{R}^{\mathbf{n}}$ if the total amount of flow entering/leaving $v \in V$ is $\mathbf{b_v}$.

The decision version asks whether for a fixed $\mathbf{b}$, a flow f can satisfy the demands without violating any capacity constraints. Through a reduction using binary search, we can use the decision version to find the optimal flow f^*.

Outline of paper:
In the remainder of section 1 we present preliminary definitions and outline a reformulation of the problem statement in terms of congestion.
In section 2 we will present the 3 main tools used in [Pen16]'s algorithm:

1. Use of congestion approximators to output approximate maximum flow in undirected graphs [She13].
2. Use of approximate maximum flow to produce oblivious routing schemes [RST14]. The oblivious routing schemes serve as the congestion approximators needed for (1).
3. Use of ultra-sparsifiers to reduce the size of the graph G while (approximately) maintaining the necessary flow properties.

In section 3 we will present the overall algorithm of the main result of [Pen16].
In section 4 we will prove the correctness of the algorithm (by using the results of section 2).
In section 5 we will perform the runtime analysis of the algorithm. We will use proof by induction to show that the runtime is $O\left(m\text{polylog}(n)\right)$.

1.2 Preliminaries and Notation:

We present a number of preliminaries and notation that will allow us to reformulate the problem statement and simplify the later analysis.

Definition 1.1 *(congestion):* For edge $e \in E$ with flow f_e and capacity c_e, [She13] defines congestion on e as $|f_e/c_e|$. The congestion of the network is the maximum congestion over any edge.

Definition 1.2 *(notation):* Following [She13], for a graph G let C be the $m \times m$ diagonal matrix of edge capacities. Let B be the $m \times n$ divergence matrix such that $(Bf)_v$ is the excess at vertex v.

Remark 1: From definition 1.2, the condition that $\mathbf{f}$ satisfies demand vector $\mathbf{b}$ can now be expressed as $B\mathbf{f} = \mathbf{b}$.

Definition 1.3 *(Cut):* A cut is defined by a subset of vertices $S \subseteq V$. The demand of the cut, $\mathbf{b}(S)$, is given by the total demand of all vertices $v \in S$. The capacity of S (in vector form) is $\mathbf{u}(S)$ is the total capacity of edges leaving S.

Definition 1.4 *(Approx)*: We define the notation for approximation as $x \approx_\kappa y$ to mean:

$$x \leq y \leq \kappa x$$

1.3 Reformulation of Problem statement:

Lemma 1: *Finding the maximum flow* $\mathbf{f}$ *that satisfies demands* $\mathbf{b}$ *is equivalent to finding a flow which minimizes* $||C^{-1}\mathbf{f}||_\infty \leq 1$ *and satisfies demands* $\mathbf{b}$.
Proof:

1. For a flow f to be feasible, the congestion of each edge $|f_e/c_e|$ must be ≤ 1. Equivalently, the maximum congestion over all edges must be ≤ 1.
2. From (1) and *Definition 1.2*, we can express this as: f is feasible if $||C^{-1}\mathbf{f}||_\infty \leq 1$.
3. From (2), we can find a flow f that meets demands $\mathbf{b}$ by minimizing:

$$||C^{-1}\mathbf{f}||_\infty$$

such that the following constraints hold:

$$B\mathbf{f} = \mathbf{b}$$

$$||C^{-1}\mathbf{f}||_\infty \leq 1$$

From *Lemma 1*, we can reframe the exact MaxFlow problem in terms of finding the optimal congestion $opt(b)$. Namely, let f^* be the maximum flow on G satisfying demands $\mathbf{b}$. We define $opt(b)$ as the congestion for flow f^*.

We can express our goal of approximate maximum flow as finding a flow $\mathbf{f}$ such that:

$$||C^{-1}\mathbf{f}||_\infty \leq (1+\epsilon)opt(\mathbf{b})$$

or certifying that the demands cannot be met with congestion less than 1.

2 Algorithmic tools

We will outline the 3 main tools required for our algorithm. We will formulate each tool as a subroutine that will be called by the overall algorithm. For each subroutine, we will develop a theorem to prove its correctness.

2.1 Congestion-approximators

We will sketch the main results of [She13] needed for our algorithm:

Definition 1.2: An $\alpha-$congestion approximator R is a matrix which satisfies:

$$||R\mathbf{b}||_\infty \approx_\alpha opt(\mathbf{b})$$

Theorem 1 (ApproximatorMaxFlow): *The main result of [She13] proves that given fixed demands* $\mathbf{b}$ *and access to an* $\alpha-$*congestion approximator* R *then there is an algorithm* ApproximatorMaxFlow *that:*

1. *Makes $O\left(\alpha^2 log^2(n)\epsilon^{-3}\right)$ number of iterations.*
2. *Each iteration takes $O(m)$ time in addition to the time for computing matrix-vector multiplication with $\mathbf{R}$ and $\mathbf{R^T}$*
3. *Returns an approximate maximum flow $\mathbf{f}$ such that $B\mathbf{f} = \mathbf{b}$ and $||C^{-1}\mathbf{f}||_\infty \leq (1+\epsilon)opt(\mathbf{b})$.*

Proof sketch: [She13] transforms the constrained minimization problem $min||C^{-1}\mathbf{f}||_\infty$ s.t. $B\mathbf{f} = \mathbf{b}$ into an unconstrained problem:

$$min||C^{-1}\mathbf{f}||_\infty + 2\alpha||R(\mathbf{b} - B\mathbf{f})||_\infty$$

The unconstrained problem can approximated very well with softmax which in turn is approximated with its gradient. The flow $\mathbf{f_t}$ at iteration t may not satisfy $\mathbf{b}$ perfectly due to the introduction of the second "residual" term. [She13] shows that the size of the residual term is bounded and decreases on each iteration. We can therefore iteratively employ the softmax approximation to route the residual until the final remainder is small enough to be routed directly. The iterative steps of approximately solving for routing $\mathbf{b}$ ultimately allow us to exactly satisfy the demands $\mathbf{b}$.

2.2 Oblivious routing scheme

Definition 2.1 *Oblivious routing scheme*: As in [RS18], on a graph G a routing scheme predetermines the path of a unit flow f between nodes s and t for all pairs nodes. The scheme is oblivious when routes are predetermined without knowledge of the demand vector b. With an oblivious routing scheme in hand, a demand vector b is satisfied by scaling the flow along the predetermined paths.

Definition 2.2 *Competitive Ratio*: We can evaluate an oblivious routing scheme in terms of its competitive ratio. Let $oblv(b)$ be the congestion of the flow produced by the oblivious routing scheme for demands $\mathbf{b}$. We define the competitive ratio as:

$$\text{competitive ratio} = \frac{oblv(\mathbf{b})}{opt(\mathbf{b})}$$

Namely, the ratio of the congestion of the flow produced by the oblivious routing scheme compared to the optimal possible congestion for a flow $\mathbf{f}^*$ satisfying $\mathbf{b}$.

Remark 2: *[She13] (in the second paragraph of the abstract) notes that oblivious routing schemes with competitive ratio α serve as α-congestion approximators.*

Proof: From **Definition 2.2** for competitive ratio α we have $oblv(\mathbf{b}) = \alpha \cdot opt(\mathbf{b})$. We therefore have that $oblv(\mathbf{b}) \approx_\alpha opt(\mathbf{b})$.

Lemma 2.1: [RS18] construct an oblivious routing scheme R by way of a single hierarchical decomposition tree of a graph G that achieves a competitive ratio of $O\left(log^4(n)\right)$. [Pen16] highlights that by virtue of R being a tree, matrix-vector products using R and R^T can be performed in $O(n)$ time.

Lemma 2.2: The hierarchical decomposition to construct R is through computation of approximate maximum flow (with error $\frac{1}{\Theta(log^3(n))}$) on graphs of size $m_1, \cdots, m_N$ such that:

$$\sum_{i=1}^{N} m_i \leq O\left(mlog^4(n)\right)$$

in addition to running time $O\left(mlog^6(n)\right)$.

Theorem 2: *There is a routine* CongestionApproximator *that given approximate maximum flow of a graph G constructs an $O\left(log^4(n)\right)$-congestion approximator* $\mathbf{R}$ *with high probability. The construction computes approximate maximum flow (with error $\frac{1}{\Theta(log^3(n))}$) on graphs of size $m_1, \cdots, m_N$ such that:*

$$\sum_{i=1}^{N} m_i \leq O\left(mlog^4(n)\right)$$

Proof:

1. From **Lemma 2.1** and **Remark 1**, we can construct R as an $O\left(log^4(n)\right)$-congestion approximator by way of hierarchical decomposition.
2. From (1) and **Lemma 2.2**, we can use approximate maximum flow to construct R.
3. **Lemma 2.2** directly gives the size of the graphs used in approximate maximum flow.

Remark 3: *[RST14] (in the abstract) notes that the approximate maximum flow of [She13] provides an efficient method to construct $O\left(log^4(n)\right)$-congestion approximator* $\mathbf{R}$. *Together with remark 2 this highlights the "chicken-and-egg" setup.*

2.3 Ultra Sparsifiers

We outline the processes used to reduce the original graph G into tree-like sparse structures using edge and vertex reductions.

Lemma 3.1 *(Edge reductions)*: There is a routine UltraSparsify that takes a graph G and parameter $\kappa > 1$ that constructs a graph H with the same set of vertices and $n-1+O\left(mlog^2(n)log(log(n))/\kappa\right)$ edges such that **for all** $S \subseteq V$:

$$\mathbf{u_H(S) \approx_k u_G(S)}$$

with runtime $O\left(mlog(n)log(log(n))\right)$.

Lemma 3.2: *An α-congestion approximator for H is also a $\kappa\alpha$-congestion approximator for G*
Proof:

1. From the problem reformulation, the finding an approximate congestion approximator is equivalent to finding an approximate maximum flow.
2. From the MaxFlow-MinCut theorem, finding the maximum flow is equivalent to finding the minimum cut.

3. From (1) and (2), we have that finding an approximate congestion approximator is equivalent to finding an approximate minimum cut.
4. A minimum cut aims to minimize $\frac{\mathbf{u(S)}}{\mathbf{b(S)}}$.
5. From (4) and **Lemma 3.1**, we have:
$$min\frac{\mathbf{u_G(S)}}{\mathbf{b(S)}} \approx_\kappa min\frac{\mathbf{u_H(S)}}{\mathbf{b(S)}}$$
6. From (3) and (5) we have:
$$opt_H(\mathbf{b}) \approx_\kappa \mathbf{opt_G}(\mathbf{b})$$
7. From **Definition 1.2***:* for an α-congestion approximator for H labelled R_H we have:
$$||R_H\mathbf{b}||_\infty \approx_\alpha opt_H(\mathbf{b})$$
From (6) we then have:
$$||R_H\mathbf{b}||_\infty \approx_{\alpha\kappa} opt_G(\mathbf{b})$$
Therefore we have proven that R_H is a $\kappa\alpha$-congestion approximator for G.

Lemma 3.3 *(Vertex reductions)*: From [Mad10] there is a routine $H' = \text{Reduce}(H)$ which reduces the number of vertices of while maintaining graph cut properties such that any α-congestion approximator for H' can be converted into an $O(\alpha)$-congestion approximator for H. When H has $m = n - 1 + m'$ edges, the output graph H' has $O(m')$ number of edges.

We can use now use the edge and vertex reductions to present the theorem needed for our ultra-sparsifiers.

Theorem 3 *(UltraSparisification): There are routines* UltraSparsifyAndReduce *and* Convert *such that on input graph G, with n vertices and m edges, and approximation parameter κ, calling* UltraSparsifyAndReduce(G, κ) *outputs a graph G' such that with high probability:*

1. *G' has at most $O\left(mlog^2(n)log(log(n))/\kappa\right)$ edges.*
2. *Given an α-congestion approximator $R_{G'}$ for G', $R_G =$ Convert$(G, G', R_{G'}$ is an $O(\kappa\alpha)$-congestion approximator for G.*

Proof Sketch:

1. Step 1 of UltraSparsifyAndReduce follows from **Lemma 3.1** and **Lemma 3.3**. Namely:
   (a) Per **Lemma 3.1**, we can produce a graph $H = \text{UltraSparsify}(G)$ with $n - 1 + O\left(mlog^2(n)log(log(n))/\kappa\right)$ edges.
   (b) Then per **Lemma3.3** we can produce $H' = \text{Reduce}(H)$ with $O\left(mlog^2(n)log(log(n))/\kappa\right)$ edges.
2. Step 2 of UltraSparsifyAndReduce follows from **Lemma 3.2** and **Lemma 3.3**:

(a) Per **Lemma 3.3**, an α-congestion approximator $R_{H'}$ for H' can be converted into an $O(\alpha)$-congestion approximator R_H for H.

(b) Then per **Lemma 3.2**, we have that R_H is an $O(\kappa\alpha)$-congestion approximator of G (for $H = \text{UltraSparsify}(G)$).

3 Algorithm

We now present [Pen16]'s recursive algorithm for finding the approximate maximum flow **f** by calling the subroutines defined in section 2. See figure below for pseudocode. In step 2 the algorithm performs size reduction of the graph using UltraSparsifyAndReduce to produce G'. In step 3 the algorithm recursively calls RecursiveApproxMaxFlow and terminates with a congestion approximator $R_{G'}$ for G'. In step 4 the algorithm converts $R_{G'}$ into a congestion approximator R_G for the original graph G. In step 5 the algorithm uses R_G and the result of [She13] to return a $(1+\epsilon)$ approximate maximum flow that satisfies **b**.

```
f = RecursiveApproxMaxFlow($G, \epsilon, \mathbf{b}$)
1  Set $\kappa \leftarrow K log^6(n) log(log(n))$ for some absolute constant $K$.

2  $G' \leftarrow$ UltraSparsifyAndReduce($G, \kappa$).

3  $R_{G'} \leftarrow$ CongestionApproximator($G'$),
   which in turn makes recursive calls to RecursiveApproxMaxFlow.

4  $R_G \leftarrow$ Convert($G, G', R_{G'}$)

5  return ApproximatorMaxFlow($G, R_G, \epsilon$)
```

4 Proof of correctness

We will now prove the correctness of the algorithm RecursiveApproxMaxFlow by bounding the number of recursive calls.

Lemma 4.1 *(Bound of recursive graph size): We have that $|E_{G'}| \le O\left(\frac{|E_G|}{(K\cdot log^4(n))}\right)$ during each recursive call.*
Proof:

1. From **Theorem 3** , we have that $|E_{G'}| \le O\left(mlog^2(n)log(log(n))/\kappa\right)$. Plugging in for $\kappa = Klog^6(n)log(log(n))$ gives us:

$$|E_{G'}| \le O\left(\frac{mlog^2(n)log(log(n))}{Klog^6(n)log(log(n))}\right)$$

$$= O\left(\frac{m}{K\cdot log^4(n)}\right)$$

Plugging in $m = |E_G|$ gives:

$$|E_{G'}| \leq O\left(\frac{|E_G|}{(K \cdot log^4(n))}\right)$$

Theorem 4.1*: The total number of recursive calls is bound by $O(m)$ with high probability.*
Proof:

1. From **Theorem 2** we have that the total collective size of all of the graphs created by the recursive calls is bounded by:
$$\leq O\left(|E_{G'}| log^4(n)\right)$$

2. Plugging in from **Lemma 4.1** we then have:
$$\leq O\left(\frac{|E_G|}{(K \cdot log^4(n))} \cdot log^4(n)\right)$$
$$= O\left(\frac{|E_G|}{K}\right)$$

3. For large enough constant K the bound is $O(m)$ with high probability. The total number of recursive calls is therefore $O(m)$.

Theorem 4.2*: The algorithm* RecursiveApproxMaxFlow *is correct with high probability.*
Proof:

1. The correctness with high probability of the subroutines ApproximatorMaxFlow, CongestionApproximator, UltraSparsifyAndReduce and Convert are given in section 2. The overall correctness of the algorithm therefore follows by proving that for the choice of $\kappa := K log^6(n) log(log(n))$ (step 1) the recursive calls terminate and the aggregate failure probability over all recursive calls remains small.

2. From **Theorem 4.1**, the number of recursive calls is bound by $O(m)$. From a union bound, we can therefore bound the collective failure probability over all recursive calls such that there is a high probability of success for the algorithm overall.

The algorithm therefore with high probability correctly outputs a $(1 + \epsilon)$ approximate maximum flow that satisfies demands $\mathbf{b}$.

5 Runtime analysis:

We will now use proof by induction to prove that the overall algorithm RecursiveApproxMaxFlow terminates in $O(m\text{polylog}(n))$ time with high probability.

Lemma 5.1 *(bound step (5) of* ApproximatorMaxFlow*): For a graph with m edges, the runtime of the final call (in step (5)) of the algorithm is $O\left(m log^{22}(n) log^2(log(n)) \epsilon^{-3}\right)$.*
Proof:

1. From **Theorem 2**, we have that $R_{G'}$ is an $O\left(log^4(n)\right)$-congestion approximator for G' (with high probability).
2. From **Theorem 3**, line (4) of the algorithm then uses $R_{G'}$ to construct R_G as an α' -congestion approximator for G with (with high probability). Where $\alpha' = O\left(\kappa \cdot log^4(n)\right)$
$$= O\left(Klog^6(n)log(log(n)) \cdot log^4(n)\right)$$
$$O\left(K \cdot log^{10}(n)log(log(n))\right)$$
3. From **Theorem 1** we have that the total number of iterations for an instance of ApproximatorMaxFlow is:
$$O\left((\alpha')^2 log^2(n)\epsilon^{-3}\right)$$
$$= O\left(log^{20}(n)log^2(log(n)) \cdot log^2(n)\epsilon^{-3}\right)$$
$$= O\left(log^{22}(n)log^2(log(n)) \cdot \epsilon^{-3}\right)$$
4. From **Theorem 1**, we have that each iteration of ApproximatorMaxFlow takes $O(m)$ time.
5. From (4) and (5) we have that the runtime contribution of the final call of ApproximatorMaxFlow is:
$$O\left(m \cdot log^{22}(n)log^2(log(n))\epsilon^{-3}\right)$$

Lemma 5.2: *The runtime of* RecursiveApproxMaxFlow *must be at least* $O\left(m \cdot log^{31}(n)log^2(log(n)\right)$ *time*
Proof:

1. From **Theorem 2** we have that the error parameter is at most $\epsilon = \frac{1}{\Theta(log^3(n))}$ for all calls. Therefore the factor of ϵ^{-3} contributes at most $log^9(n)$.
2. From (1) and **Lemma 5.1**, the runtime contribution of the final call of ApproximatorMaxFlow is $O\left(m \cdot log^{31}(n)log^2(log(n))\right)$.
3. As the overall runtime of RecursiveApproxMaxFlow is $\geq$ the runtime of the subroutine ApproximatorMaxFlow, we then have that the runtime of RecursiveApproxMaxFlow is at least $O\left(m \cdot log^{31}(n)log^2(log(n)\right)$.

Theorem 5: *With high probability,* RecursiveApproxMaxFlow *terminates in:*
$O\left(m \cdot log^{31}(n)log^2(log(n)\right)$ *time*:
From **Lemma 5.2** we have a "lower bound" on the runtime. We will use a proof by induction to prove the bound as an upperbound for the runtime of RecursiveApproxMaxFlow.

1. Let $T(m)$ be the runtime of RecursiveApproxMaxFlow on a graph with m edges and error parameter$\epsilon = \frac{1}{\Theta(log^3(n))}$
2. The **base case** of $m \leq log^{10}n$ holds by using existing approximate maximum flow algorithms.
3. For the inductive step, let $m_1, \cdots, m_N$ be the sizes of the graphs called in the recursive calls.

4. From **Theorem 2** and **Lemma 5.1** we have the following recurrence relation:

$$T(m) \leq \sum_{i=1}^{N} T(m_i) + O\left(mlog^6(n)\right) + O\left(m \cdot log^{31}(n)log^2(log(n))\right)$$

   As the $O\left(mlog^6(n)\right)$ term is dominated:

$$= \sum_{i=1}^{N} T(m_i) + O\left(m \cdot log^{31}(n)log^2(log(n))\right)$$

5. From **Lemma 4.1** we have that the graph in each recursive call is a subgraph of G such that the inductive hypothesis holds. Therefore we have:

$$T(m) \leq \sum_{i=1}^{N} K' \cdot m_i \cdot log^{31}(n)log^2(log(n)) + O\left(m \cdot log^{31}(n)log^2(log(n))\right)$$

6. From **Theorem 2** and **Theorem 3** we have:

$$\sum_{i=1}^{N} m_i \leq O\left(|E_{G'}|log^4(n)\right) \leq O\left(mlog^6(n)log(log(n))/\kappa\right)$$

   For large enough K, we can bound this as:

$$\leq \frac{m}{2}$$

7. From (5) and (6) we have:

$$T(m) \leq \frac{m}{2} \cdot K' \cdot log^{31}(n)log^2(log(n) + O\left(m \cdot log^{31}(n)log^2(log(n))\right)$$

$$= O\left(m \cdot log^{31}(n)log^2(log(n))\right)$$

   when K' is set to the (hidden) constant of the last term.

(7) proves the inductive step and thereby proves overall that RecursiveApproxMaxFlow runs in $O(m\text{polylog}(n))$ time.

From section 4 and section 5, we have that with high probability, RecursiveApproxMaxFlow returns a $(1+\epsilon)$ maximum flow solution satisfying demands $\mathbf{b}$ in $O(m \cdot \text{polylog}(n))$ time.

References

[KLOS13] 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, 2013.

[Mad10] Aleksander Madry. Fast approximation algorithms for cut-based problems in undirected graphs, 2010.

[Pen16] Richard Peng. Approximate undirected maximum flows in $O(m\text{polylog}(n))$ time. In Robert Krauthgamer, editor, *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, pages 1862–1867. SIAM, 2016.

[RS18] Harald Räcke and Stefan Schmid. Compact oblivious routing, 2018.

[RST14] Harald Räcke, Chintan Shah, and Hanjo Täubig. Computing cut-based hierarchical decompositions in almost linear time. 01 2014.

[She13] Jonah Sherman. Nearly maximum flows in nearly linear time. *CoRR*, abs/1304.2077, 2013.