$O(nm)$-TIME ALGORITHM FOR MAX FLOW

MAX ZIMET AND ADAM EAGLE
MAXZIMET@MIT.EDU, AREAGLE@MIT.EDU

ABSTRACT. In this paper, we describe the recent work by Orlin[1] which gives a strongly-polynomial $O(nm)$-time algorithm for max flow, where m is the number of edges in our graph and n is the number of nodes. This is the best asymptotic running time that has been attained for the max-flow problem. The algorithm's central idea is to reduce the number of nodes in the graph we are concerned with by an amount depending on the capacities of edges, and then to run the fastest-known scaling algorithm – that of Goldberg and Rao[2] – on the new graph. The max-flow on the new graph can then be used to find a max-flow on the original graph.

1. INTRODUCTION

The max flow problem is an important problem in combinatorial optimization. We state the problem as follows:

Consider a directed graph $G = (N, A)$ with $n = |N|$ nodes, $m = |A|$ edges, and non-negative integer-valued edge capacities u_{ij} in which the largest capacity is bounded by U. There are two special nodes in G, a source s and a sink t. The flow on edge (i, j) is denoted as x_{ij}, where for each edge (i, j) we require $x_{ij} \geq 0$, and the flow satisfies the conservation constraint:

$$\sum_{(i,j)\in A} x_{ij} - \sum_{(j,i)\in A} x_{ij} = 0 \quad \forall i \in N \setminus (s, t).$$

A flow is feasible if it also satisfies the capacity constraints, $x_{ij} \leq u_{ij} \, \forall (i, j) \in A$. The value of a flow x is the net flow out of the source, which is equal to the net flow into the sink. In the max flow problem, we seek a feasible flow with the largest possible value.

The fastest strongly polynomial algorithm was published by King et. al. [3]; this has a running time of $O(nm \log_{m/(n \log n)} n)$. When $m = \Omega(n^{1+\epsilon})$ for any $\epsilon > 0$, the running time of this algorithm is $O(nm)$. The fastest weakly polynomial algorithm, published by Goldberg and Rao[2], has a running time of $O(\Lambda m \log (n^2/m))$ per scaling phase, where $\Lambda = \min(n^{2/3}, m^{1/2})$. This algorithm uses $\log U$ scaling phases, each of which reduces the difference between the current s-t cut capacity and the minimum s-t cut capacity by a factor of $1/2$. Karzanov[4] proves using the method of preflows that the max flow problem can be solved in $O(n^3)$ time.

Date: December 9, 2013.

Orlin[1] shows that max flows can always be found in $O(nm)$ time by providing an $O(nm + m^{31/16} \log^2 n)$-time algorithm; when $m = O(n^{(16/15)-\epsilon})$, this running time is $O(nm)$. Since King et. al. [3] find a max flow in $O(nm)$ time for all other m, namely, $m > n^{1+\epsilon}$, Orlin's algorithm establishes that the max flow problem can be solved in $O(nm)$ time for all n and m.

This paper summarizes and clarifies some of the ideas in Orlin's paper without going into too much technical detail regarding specific implementations of subroutines and data structures. References are provided for the reader who wishes to investigate these matters more closely.

2. Additional Notation and Assumptions

We will now define a few properties that will used frequently in the rest of this paper. The residual capacity of an edge (i, j) is $r_{ij} = u_{ij} - x_{ij} + x_{ji}$, and it expresses the amount of additional flow that can be sent from i to j, given a flow x. We will denote the vector of residual capacities as r and the residual graph by G_r.

An s-t cut $[S, T]$ is a partition of N into S and T such that $s \in S$ and $t \in T$. An edge (i, j) is a forward edge of $[S, T]$ if $i \in S$ and $j \in T$. Similarly, an edge (i, j) is a backward edge of $[S, T]$ if $j \in S$ and $i \in T$. The capacity of an s-t cut is $u(S, T) = \sum_{i \in S, j \in T} u_{ij}$ and the residual capacity of an s-t cut is $r(S, T) = \sum_{i \in S, j \in T} r_{ij}$.

Finally, we assume that there is, in fact, a feasible flow in the network we are studying. If there were not such a flow, then we could detect this condition using a breadth-first search; thus, this case is uninteresting.

3. Overview of Algorithm

We now give an overview of the algorithm.

3.1. **Improvement Phases.**

Our algorithm will proceed via a sequence of improvement steps. Each improvement step will take as input a flow, specified by a vector r of residual capacities (so for each edge (i, j), r contains an entry r_{ij}); and an s-t cut, specified by the subset of nodes S which contains s and the subset of nodes T which contains t. We denote this input as (r, S, T). The procedure *Improve*(r, S, T), detailed in Section 5, implements each improvement phase. The output of this procedure is a new flow, specified by a vector r' of residual capacities, and a new s-t cut, specified by subsets $S', T' \subset V$.

We want to quantify the extent of improvement accomplished by each improvement phase. To do so, we let Δ be the residual capacity of the s-t cut (S, T), which we denote by $r(S, T)$. We note that the max flow-min cut theorem implies that Δ is an upper bound on the max-flow in the residual graph; we therefore refer to Δ as the flow bound. For reasons that will become clear later, we also associate to each phase a number Γ, which we call the compactness parameter. We will choose Γ so that $0 \leq \Gamma \leq \Delta$. We also require that each improvement phase satisfy the improvement property: $\Delta' \leq \Gamma/(8m)$, where $\Delta' = r'(S', T')$. In particular, $\Delta' \leq \Delta/(8m)$, so the flow bound improves by a factor of at least $8m$ after each phase.

3.2. **Critical Nodes, Compact Network.** The essential idea used to make this algorithm fast is, in each phase, to single out some nodes of our graph G as being Γ-*critical,* or simply *critical* if the value of Γ is clear. (That is, while talking about a particular improvement phase, which has a fixed value of Γ, we may simply use the term *critical*, leaving the value of Γ implicit). These are, intuitively, the nodes adjacent to edges with significant amounts of residual capacity. Let C be the number of Γ-critical nodes in a particular improvement phase. We show in Theorem 3 that the number of such nodes, over all improvement phases, is $O(m)$. We then just need to upper bound the time spent by the algorithm per Γ-critical node in each improvement phase. That is, let T be the time taken by the algorithm during a particular improvement phase. We want to prove the upper bound $T/C = \tilde{O}(m^{15/16})$ (where the tilde means we ignore log factors of m and n). Note that given this upper bound, the time to run the total algorithm is $\tilde{O}(m \cdot m^{15/16}) = \tilde{O}(m^{31/16})$, as there are $O(m)$ critical nodes over all improvement phases. Finally, using our assumption that $m = O(n^{16/15-\epsilon})$ for some positive ϵ, we find that the running time of our algorithm is $\tilde{O}(m \cdot n^{1-\epsilon(15/16)}) = O(mn)$.

We prove the upper bound $T/C = \tilde{O}(m^{15/16})$ by considering two cases. The first case is if there are many critical nodes ($C \geq m^{9/16}$), in which case we can accomplish the goal of this improvement phase by running $O(\log n)$ Goldberg and Rao scaling phases, and T/C is upper bounded because C is large. The second case is if there are few critical nodes ($C < m^{9/16}$). In this case we create a new network, called the compact network, whose nodes are the critical nodes, and which has $O(m)$ edges. We find a flow on this network, and use this to find an approximate max-flow on the original network. We use the fact that the compact network has only C nodes and m edges to upper bound T/C.

There is an intuitive way to see why the running time of this algorithm is independent of U. The output of each improvement phase is an s-t cut whose capacity is at most $1/(8m)$ times that of the s-t cut which was the input to this phase. Since the edge capacities are integers, the smallest capacity an s-t cut can have is 1 (assuming there is a possible flow). The largest capacity an s-t cut can possibly have is mU. Therefore, there are at most $\log_{8m} mU \leq 1 + \log U$ improvement phases. If we need few improvement phases, we may as well have used Goldberg-Rao. So, assume that there are $\Theta(\log U)$ improvement phases, so *the average number of critical nodes per improvement phase is* $O(m/\log U)$. We emphasize this statement because it is the key to the strongly-polynomial runtime of this algorithm: we construct compact networks with C nodes, where C scales inversely with $\log U$. If $\log U \leq m^{7/16}$, then again the Goldberg-Rao algorithm completes in time $O(mn)$. So, assume that $\log U > m^{7/16}$. Since the time needed to run the Goldberg-Rao algorithm on a graph with C nodes and $O(C^2)$ edges is $\tilde{O}(C^{8/3} \log U)$, it follows that the total time needed to run the algorithm is $\tilde{O}((m/\log U)^{8/3} \log U) = \tilde{O}(m^{8/3} (\log U)^{-5/3}) = \tilde{O}(m^{8/3-(7/16)\cdot(5/3)}) = \tilde{O}(m^{31/16})$.

As we noted above, our assumption that $m = O(n^{16/15-\epsilon})$ implies that this is $O(mn)$.

4. Constructing the Compact Network

We denote the compact network by $G_c = (N_c, A_c)$. In order that the max-flow value in the compact network be close to the max-flow value in the original network, we choose the Γ-critical nodes to be those with significant residual capacities coming in or out of them. To make this precise, we introduce a number of definitions.

4.1. **Cuts in the Compact Network.**

Because improvement steps in our algorithm, and scaling steps in the Goldberg-Rao algorithm, require a cut as input, we explain how we will obtain a cut in the compact network from an *s*-*t* cut in the original network, and vice versa. This will guide our construction of the compact network.

First, we show how to get a cut in the compact network from an *s*-*t* cut in the original network. Take an *s*-*t* cut $[S, T]$ of the original network. We call the cut $[S_c, T_c]$ of G_c defined by $S_c = S \cap N_c$ and $T_c = T \cap N_c$ the cut *induced by* the *s*-*t* cut $[S, T]$.

Next, we show how to obtain an *s*-*t* cut $[S, T]$ in the original network from a cut $[S_c, T_c]$ in the compact network. We note that our algorithm proceeds by finding a cut $[S_c, T_c]$ in the compact graph with small capacity, and then uses this to find an *s*-*t* cut in the original network with small capacity. So, the capacity of the induced cut in the original network should not be much more than that of $[S_c, T_c]$. We introduce the concept of abundant edges in order to satisfy this condition.

Recall that Δ is the flow bound. So, if an edge has capacity greater than Δ, then we will not include it in the *s*-*t* cut which is the output of the current improvement phase. (If we did, then the *s*-*t* cut would have capacity greater than Δ). This motivates the following definition.

Definition 1. An edge (i, j) is said to be Δ-abundant if $r_{ij} \geq 2\Delta$. An edge (i, j) is said to be Δ-anti-abundant if $r_{ji} \geq 2\Delta$. A path in G_r is an abundant path if it consists entirely of abundant edges. Denote the set of abundant edges by A_{ab} and the set of anti-abundant edges by A_{-ab}.

We frequently drop the Δ's if the value of Δ is obvious; for instance, we may simply write that an edge is abundant.

Using this definition, we can give a procedure for inducing *s*-*t* cuts in the original network from cuts $[S_c, T_c]$ in the compact network. Namely, let S be the subset of N consisting of nodes which are reachable from a node in S_c via an abundant path in G_r; let $T = N \backslash S$.

We will now use these procedures for inducing cuts to guide our construction of the compact network. Orlin[1] shows that the capacities of the induced cuts do not differ significantly from the capacities of the original cuts.

4.2. **Contraction.** As we have argued, we frequently do not need to consider abundant edges. Therefore, our first step in creating the compact graph is to contract many such edges. The concept of contraction was introduced by Goldberg and Rao[2, 1]. We need the following definition.

Definition 2. The abundance graph G_{ab} is the subgraph of G whose nodes are N and whose edges are the Δ-abundant edges.

At the beginning of an improvement phase, we contract each directed cycle of the abundance graph to a single node. We also contract abundant external edges, where external edges are edges adjacent to either s or t. Any edge coming into / going out of one of these contracted nodes in the original abundance graph will have a corresponding edge coming into / going out of the new node. We also have to modify the flow in the new graph. The exact details of contraction are discussed in Goldberg and Rao. The important facts about this process, as stated by Orlin[1], are:

(1) the flows in the expanded and contracted graphs have the same value;
(2) we can expand a contracted graph (essentially reversing the process of contraction);
(3) the time taken, per improvement phase, for contraction and for expansion is $O(m)$.

As we show in Lemma 5, the number of improvement phases is $O(m^{2/3})$, so the total time on contractions and expansions is $O(m^{5/3}) = o(m^{31/16})$, and therefore these operations are not bottleneck operations.

Note that one consequence of contracting cycles is that if (i, j) is in the modified abundance graph (which I henceforth simply refer to as the abundance graph), then (j, i) is not.

4.3. **Compaction.** We will now provide the definitions necessary for identifying Γ-critical nodes.

Definition 3. An edge (i, j) is said to be Γ-medium if $\Gamma/(64m^3) \leq u_{ij} + u_{ji}$, and neither (i, j) nor (j, i) is Δ-abundant.

Definition 4. Let $\hat{r}_{out}(i)$ and $\hat{r}_{in}(i)$ denote, respectively, the total residual capacity of anti-abundant edges going out of/into node i. Then, node i is Γ-critical if it is incident to a Γ-medium edge or if $|\hat{r}_{out}(i) - \hat{r}_{in}(i)| > \Gamma/(16m^2)$. Nodes which are not Γ-critical are called Γ-compactible.

With these definitions, we can now define the compact graph. The nodes of the compact graph are the Γ-critical nodes. Denote the set of these nodes by N_c. The edges may be divided into three classes. Writing A_c for the edge set of the compact graph, we write $A_c = A_1 \cup A_2 \cup A_3$. Edges in A_1 are called original edges, and they correspond to edges between critical nodes which are in the original network; the residual capacity of these edges are unchanged from their residual capacity in the original network. Edges in A_2 are called abundant pseudo-edges. There is such an edge between critical nodes i and j if there is a path in the abundance graph from i to j. The residual capacity of this edge is 2Δ. Finally, edges in A_3, called anti-abundant pseudo-edges, are created by the following procedure, adapted from [1].

```
1  **Input**: (r, $\Gamma$)
2  **Result**: Some capacity is transferred from anti-abundant edges in the
            original graph to anti-abundant pseudo-edges in the compact
            graph
3  Let $H$ be the subset of edges of $A_{-ab}$ incident to $\Gamma$-compactible nodes;
4  foreach $(i, j) \in H$ do
5      $q_{ij} = r_{ij}$;
6  end
7  while $H \neq \emptyset$ do
8      select a node $i$ of $H$ with no incoming edges;
9      use depth first search to find a path $P$ that starts at node $i$ and ends
       at a node $\ell$ such that $\ell \in N_c$ or $\ell$ has no outgoing edge (or both);
10     let $\delta = \min(q_{jk} : (j, k) \in P)$;
11     if $(i, \ell) \in N_c$ then
12         $A_3 = A_3 \cup (i, \ell)$;
13         ${r^c}_{i\ell} = \delta$;
14     end
15     forall the $(j, k) \in P$ do
16         $q_{jk} = q_{jk} - \delta$;
17     end
18     delete each edge $(j, k)$ from $H$ such that $q_{jk} = 0$;
19     forall the pairs $(i, j) \in N_c$ do
20         aggregate all edges in $A_3$ from $i$ to $j$ in a single edge $(i, j)$;
21     end
22 end
```

Algorithm 1: Transfer Capacity

Note: $r^c_{i\ell}$ in line 13 is the residual capacity of edge (i, ℓ) in the compact graph (before we aggregate the edges together in line 20).

This algorithm keeps the capacity of certain paths between critical nodes which travel only over anti-abundant edges: namely, those paths which travel only through compactible nodes, except for at the endpoints of the path. So, we lose capacity when going from the original network to the compact network when we have a path of anti-abundant edges which cannot be extended to a path of anti-abundant edges that starts and ends on a critical node. We call this capacity lost capacity.

Let **P** be the set of paths found in this procedure which start and end at critical nodes (so their capacity is not lost), and let **Q** be the set of paths found in this procedure which do not start and end at critical nodes (so their capacity is lost). We want to bound the difference in min-cut capacities between the original and compact networks. Our first step toward this is to bound the amound of residual capacity that was lost.

Lemma 1. *The amount of lost residual capacity was less than* $\Gamma/(16m)$.

Proof. Let $q_{out}(j)$ and $q_{in}(j)$, respectively, be the residual capacity out of / into compactible node j at some iteration of the capacity transfer procedure. Let $\Phi(j,q) = q_{out}(j) - q_{in}(j)$. When we begin the capacity transfer, $|\Phi(j,q)| = |\hat{r}_{out}(j) - \hat{r}_{in}(j)| \leq \Gamma/(16m^2)$. In the end, $\Phi(j,q) = 0$ because $q = 0$. Note that Φ only changes when we transfer capacity from paths in $\mathbf{Q}$.

First, suppose that $\hat{r}_{out}(j) - \hat{r}_{in}(j) > 0$. Because j is not in N_c, we will only ever end a path at j if there are no outgoing edges from j. But, this will never be the case, as every path that does not start at j will change $r_{out}(j)$ and $r_{in}(j)$ by the same amount. So, if there are incoming edges to j, then there are outgoing edges from j. Now, suppose we start a path at j. When j is selected to begin a path in line 8, we must have $q_{in}(j) = 0$, so $q_{out}(j) = \hat{r}_{out}(j) - \hat{r}_{in}(j) \leq \Gamma/(16m^2)$. So, the total amount of residual capacity lost due to paths starting at some compactible node is at most $\Gamma/(16m^2)$. A similar argument shows that the total amount of residual capacity lost due to paths ending at some compactible node is at most $\Gamma/(16m^2)$. Since every path with lost capacity either begins or ends at some compactible node, the total amount of lost capacity is less than or equal to $n\Gamma/(16m^2) \leq \Gamma/(16m)$. □

4.4. **Time to Create the Compact Network.** We now upper bound the time needed to create the compact network. Before we do so, however, we must briefly mention a number of subroutines we use in this process.

First, we mention how we are able to quickly find which nodes are connected by paths in the abundance graph.

Definition 5. The transitive closure of a graph $G = (V, E)$ is $\overline{G} = \big(V, \overline{E}\big)$, where there is an edge $(i,j) \in \overline{E}$ if there is a path in G from i to j.

Our algorithm will maintain the transitive closure of G_{ab} over the course of all improvement phases. The following lemma shows that edges are only ever added to $\overline{G_{ab}}$ over the course of our improvement phases, so that we may use the algorithm of Italiano[5] to maintain this transitive closure.

Lemma 2. *If* (i,j) *is* Δ*-abundant at the beginning of the* Δ*-improvement phase, then* (i,j) *is also* Δ'*-abundant at the beginning of the* Δ'*-improvement phase if* $\Delta' < \Delta$.

Proof. This is obvious from $r_{ij} \geq 2\Delta$, since the change in flow across (i,j) is at most Δ, so after the Δ-improvement phase $r_{ij} \geq \Delta > 2\Delta'$. □

Next, we note that our current construction of the compact network allows for $|A_2|$ to become large. We want the number of edges in the compact graph to be

$O(m)$. Note that $|A_1| \leq m$ and $|A_3| \leq m$, since each iteration of the capacity transfer algorithm deletes one edge from the original graph. Orlin[1] describes a technique that allows him to reduce the size of A_2 to be less than or equal to $C = O(m)$ (as is shown in Theorem 3).

Theorem 1. *Suppose that each compact network has at most $m^{9/16}$ nodes. Then, the time it takes to create G_c is $O(m^{9/8})$.*

From the discussion of Section 3.2, we see that the hypothesis of this theorem is indeed satisfied.

Proof. As already mentioned, contraction and expansion takes $O(m)$ time. Determining which nodes are critical takes time $O(m)$. Then, we find A_1 in $O(m)$ time. We can find A_2 in $O(C^2) = O(m^{9/8})$ time by inspecting the transitive closure of the abundance graph. Orlin[1] proves that he can find A_3 in $O(m \log n)$ time. □

4.5. **Transferring Flows from Compact to Original Network.** Orlin[1] gives a procedure for efficiently obtaining a flow on the original network, given a flow on the compact network. The details are not very interesting. However, he also shows that if the compact graph flow is close to optimal, then so is the flow on the original graph – shortly, we will clarify the sense in which this is true. First, we need a couple of lemmas.

Lemma 3. *Suppose that P is a path in the original graph from i to j which contains only anti-abundant edges. Suppose that $[S, T]$ is an s-t cut in G_r with no abundant forward edge. If $i \in S$ and $j \in T$, then P has exactly one edge crossing the cut $[S, T]$. If $i \notin S$ or $j \notin T$, then no edge of P crosses $[S, T]$.*

Proof. If $i \in S$ and $j \in T$, then there is obviously one edge in P crossing the cut. If there were more than one, then there would also be a backward edge in P – that is, there is a backward anti-abundant edge, so there is a forward abundant edge.

The proof for the other cases is similar. □

This lemma is used to prove the next lemma:

Lemma 4. *Suppose that $[S_c, T_c]$ is an s-t cut in G_c with no abundant forward edges. Let $[S, T]$ be the induced cut of G. Then $r_c(S_c, T_c) \leq r(S, T) \leq r_c(S_c, T_c) + \Gamma/(16m)$, where r_c is the vector of reduced capacities in the compact graph.*

Proof. Use Lemma 3 to create a correspondence between edges crossing the cut $[S, T]$ and paths in $\mathbf{P} \cup \mathbf{Q}$ which cross the cut. Each such edge in $\mathbf{P}$ transferred its capacity to an edge in the cut (S_c, T_c), while the edges in $\mathbf{Q}$ lost their capacity. By Lemma 1, the amount of lost residual capacity from the edges in $\mathbf{Q}$ is less than $\Gamma/(16m)$. □

The following theorem follows easily:

Theorem 2. *Take (y_c, S_c, T_c), where y_c is a flow in the Γ-compact network G_c and $[S_c, T_c]$ is a cut in G_c with $r(S_c, T_c) \leq v + \alpha$, where v is the value of y_c. Let y, S, T be the induced flow and s-t cut in G. Then, y has a flow value of v and $r(S, T) \leq r(S_c, T_c) + \Gamma/(16m) = v + \alpha + \Gamma/(16m)$.*

4.6. **Number of Critical Nodes.** Before moving on to discuss the improvement phase, we prove the following important theorem.

Theorem 3. *The number of Γ-critical nodes over all improvement phases is $O(m)$.*

Proof. We do not count the number of Γ-special nodes, as the proof that the number of Γ-special nodes, over all improvement phases, is $O(m)$ consists of much unenlightening algebra. The reader is referred to [1].

On the other hand, the proof that the number of nodes adjacent to Γ-medium edges over all phases is $O(m)$ is elegant. In fact, we prove that an edge can be Γ-medium for at most 3 consecutive phases. Suppose that there is an edge (i, j) that is Γ-medium: $u_{ij} + u_{ji} \geq \Gamma/(64m^3)$. Let Δ' be the flow bound at the next phase. Then, $u_{ij} + u_{ji} \geq \Delta'/(8m^2)$. Continuing this way, we see that if Δ^* is the flow bound two phases after Δ', then $u_{ij} + u_{ji} \geq 8\Delta^*$. At this point, (i, j) or (j, i) must be Δ^*-abundant, so (i, j) is not Γ^*-medium (where Γ^* is the compactness parameter associated to Δ^*). □

5. Improvement Phase

We begin our algorithm with an arbitrary feasible flow and s-t cut .

We now describe the steps taken during an improvement phase. This algorithm is adapted from Orlin[1]. We first define a Δ-optimal flow to be a flow whose s-t cut capacity differs from that of the minimum s-t cut capacity by a factor of Δ.

```
1  **Input**: $(r, S, T)$
2  **Result**: An improved approximate residual flow
3  $\Delta := r(S, T)$;
4  let C' be the number of $\Delta$-critical nodes ($C$ is the number of $\Gamma$-critical
   nodes);
5  **if** $C \geq m^{9/16}$ **then**
6      let $\Gamma = \Delta$;
7      find a $\Gamma/(8m)$-optimal flow in $G_r$ using $O(\log n)$ Goldberg-Rao scaling
       phases;
8  **else if** $m^{1/3} \leq C < m^{9/16}$ **then**
9      let $\Gamma = \Delta$;
10     let $G_c$ denote the $\Gamma$-compact network;
11     find a $\Gamma/(16m)$-optimal flow $y$ on $G_c$ using $O(\log n)$ Goldberg-Rao
       scaling phases;
12     let $y$ be the induced $\Gamma/(8m)$-opt flow on $G_r$;
13     update the residual capacities;
14 **else if** $C < m^{1/3}$ **then**
15     choose the minimum value $\Gamma$ such that the number $C$ of $\Gamma$-critical
       nodes in the network is less than $m^{1/3}$;
16     let $G_c$ denote the $\Gamma$-compact network;
17     find an optimal flow $y$ on $G_c$ using an $O(C^3)$-time max-flow algorithm;
18     let $y$ be the induced $\Gamma/(16m)$-opt flow on $G_r$;
19     update the residual capacities;
20
```

Algorithm 2: Improve Approximate Flow

Lemma 5. *The number of improvement phases is $O(m^{2/3})$.*

Proof. There are $O(m)$ Γ-critical and $(\Gamma/2)$-critical nodes over all improvement phases. The algorithm makes it clear that in all 3 cases, the number of $\Gamma/2$-critical nodes is at least $m^{1/3}$. So, the number of improvement phases is $O(m^{2/3})$. □

Lemma 6. *The time to create all of the compact networks is $O(nm + m^{43/24})$.*

Proof. This proof is largely uninteresting. We prove the interesting part: we can find Γ in $O(m + n \log n)$ time, when we are in the third case. For each node i, look at the adjacent edges and find the greatest Γ for which i is in the Γ-compact network. Sort all of the nodes by these values and select the smallest Γ so that the compact network contains at most $m^{1/3}$ nodes. □

Lemma 7. *The algorithm determines the optimal or approximately optimal flows in all of the compact networks in $\tilde{O}(m^{31/16})$ time.*

Proof. In the first case, we need $O(\log n)$ Goldberg-Rao scaling phases, each of which takes time $T = \tilde{O}(m^{3/2})$, so $T/C = \tilde{O}(m^{15/16})$. In the second case, we need $O(\log n)$ Goldberg-Rao scaling phases, each of which takes time $T = \tilde{O}(C^{2/3} \cdot C^2) = \tilde{O}(C^{8/3})$, so $T/C = \tilde{O}(C^{5/3}) = \tilde{O}(m^{15/16})$. In the third case, we need time $T = O(C^3)$, and $T/C = O(C^2) = O(m^{2/3})$. Since, over all phases, there are $O(m)$ critical nodes, the total time is $\tilde{O}(m^{31/16})$. □

Lemma 8. *The total time it takes to transform the flows in compact networks to the flows in the residual networks is $O(nm + m^{5/3} \log n)$.*

Proof. This is based on the procedure for transforming the flows, which we have not provided. However, this proof is not difficult and can be read in Lemma 10 of Orlin's paper[1]. □

All of these results, plus the discussion in Section 3.2, prove the following theorem:

Theorem 4. *The running time to find a maximum flow is $\tilde{O}(m^{31/16})$. If $m = O(n^{16/15-\epsilon})$ for any $\epsilon > 0$, the running time is $O(nm)$.*

6. Conclusion

Orlin[1] has provided a strongly-polynomial algorithm that finds a max-flow in $O(mn)$ time when $m = O(n^{16/15-\epsilon})$ for any $\epsilon > 0$. Together with the $O(mn)$ time algorithm of King et. al.[3], this shows that for any m and n, there is an $O(mn)$ time algorithm for max-flow with integer capacities.

References

[1] J.B. Orlin. Max flows in $O(nm)$ time, or better. *Proceedings of the 2013 Symposium on the Theory of Computing*, pages 765–774.

[2] A. V. Goldberg and S. Rao. Beyond the flow decomposition barrier. *Journal of the ACM*, 45:783–797, 1998.

[3] V. King, S. Rao, and R. Tarjan. A faster deterministic maximum flow algorithm. *J. Algorithms*, 23:447–474, 1994.

[4] A. V. Karzanov. Determining the maximal flow in a network by the method of preflows. *Soviet Mathematics Doklady*, 15:434–437, 1974.

[5] G. Italiano. Amortized efficiency of a path retrieval data structure. *Theoretical Computer Science*, 48(0):273–281, 1986.

Acknowledgements

The authors are grateful to Professor Karger for his amazing course in advanced algorithms.