A New Push-Relabel Algorithm for Sparse Networks

Rahul Mehta*
University of Chicago Lab High School
rahulmehta@uchicago.edu

June 12, 2014

Abstract

In this paper, we present a new push-relabel algorithm for the maximum flow problem on flow networks with \(n\) vertices and \(m\) arcs. Our algorithm computes a maximum flow in \(O(mn)\) time on sparse networks where \(m = O(n)\). To our knowledge, this is the first \(O(mn)\) time push-relabel algorithm for the \(m = O(n)\) edge case; previously, it was known that push-relabel implementations could find a max-flow in \(O(mn)\) time when \(m = \Omega(n^{1+\epsilon})\) (King, et. al., SODA ‘92). This also matches a recent general flow decomposition-based algorithm due to Orlin (STOC ‘13), which finds a max-flow in \(O(mn)\) time on near-sparse networks.

Our main result is improving on the Excess-Scaling algorithm (Ahuja & Orlin, 1989) by reducing the number of nonsaturating pushes to \(O(mn)\) across all scaling phases. This is reached by combining Ahuja and Orlin’s algorithm with Orlin’s compact flow networks. A contribution of this paper is demonstrating that the compact networks technique can be extended to the push-relabel family of algorithms. We also provide evidence that this approach could be a promising avenue towards an \(O(mn)\)-time algorithm for all edge densities.

1 Introduction

The maximum flow problem has been studied for several decades in computer science and operations research. One of the most widely researched combinatorial optimization problems of all time, many polynomial-time algorithms have been presented since the groundbreaking research of Ford and Fulkerson [7]. The max-flow problem has numerous theoretical applications in both computer science and operations research, as well as practical applications in transportation, scheduling, and routing problems, and more recently in computer vision. A full discussion of the history of the max-flow problem and its applications can be found in [1]. Many efficient algorithms exist, including ones based on augmenting paths, blocking flows, and the push-relabel method.

Recently, several asymptotically fast algorithms have been presented. Together, the results of King, et. al. [12] and Orlin [14] show that the max-flow problem is solvable in \(O(mn)\) time on general networks with \(n\) vertices and \(m\) arcs. The former is a derandomized version of an algorithm due to Cheriyan, et. al. [4], which relies on a push/relabel/add-edge approach. This runs in \(O(mn)\) time when \(m = \Omega(n^{1+\epsilon})\), and uses the correspondence between nonsaturating pushes and a certain combinatorial game to bound the running time. The latter is based on the augmenting path algorithm of Ford and Fulkerson, but leverages a smaller representation of the flow network known as the “compact network” to reduce the time to run a scaling phase, in a manner similar to Goldberg and Rao [8]. This algorithm runs in \(O(mn)\) time when \(m = O(n^{16/15-\epsilon})\), and in \(O(n^2/\log n)\) time on sparse networks. Our algorithm utilizes these compact networks to achieve our improved running time. There have also been recent

*This work was conducted while the author was a student at the University of Chicago Lab High School, as well as a summer intern at the Toyota Technological Institute of Chicago. The author is currently a student at Princeton University.
developments utilizing electrical flows and fast algorithms for approximately solving Laplacian systems of equations; for more information the reader is referred to [6, 11, 13].

**Our contribution**  We show how to solve the max-flow problem in $O(mn + m^{3/2} \log n)$ time for sparse networks (i.e. when $m = O(n)$). Clearly, since $m = O(n)$, the algorithm runs in $O(mn)$ time. This extends the known range for which max-flow is solvable in $O(mn)$ time using push-relabel algorithms (King, et. al. [12] gave an $O(mn)$-time algorithm when $m = \Omega(n^{1+\varepsilon})$.) This matches the general algorithm of Orlin [14], which solves the problem on general networks in $O(mn + m^{31/16} \log^2 n)$, and $O(mn)$ time on near-sparse networks. Although we do not achieve the $O(n^2/\log n)$ result on sparse networks, we give the first strongly-polynomial algorithm using the push-relabel method for the sparse edge case. We reach the improved running time by running an Ahuja-Orlin scaling phase on a smaller representation of the flow network, known as the compact network. Through this approach, we are able to obtain a strongly-polynomial algorithm for bounded-degree networks; that is, when there is some $k \in \mathbb{Z}_{\geq 0}$ such that for all $u \in V$, $\text{deg}[u] \leq k$. We call this a $k$-bounded-degree network. Moreover, using a well-known technique, we can transform a graph $G$ where $m = O(n)$ into an $(\lfloor m/n \rfloor + 3)$-bounded-degree network; thus, our bounded-degree algorithm can be extended to the more general case of sparse networks. This transformation is described in Appendix A.

An extension of our results to general networks, which would involve resolving certain issues relating to nonsaturating pushes, would imply an algorithm that runs in $O(mn)$ time when $m = O(n^{2-\varepsilon})$. We believe that an algorithm which incorporates one or more of our techniques may be a promising avenue towards developing an algorithm that works for all edge densities.

**Our approach**  Our algorithm uses a modified version of the push-relabel method [9]. We maintain a valid distance labeling $d : V \rightarrow \mathbb{Z}_{\geq 0}$ to estimate the distance of each vertex to the sink, and push excess flow from higher-labeled vertices to lower-labeled vertices. We relabel (increase the distance label) of vertices to allow more pushes in a series of scaling phases, similar to those of Ahuja and Orlin [2]. During the $i$th phase, characterized by the parameter $\Delta_i$, which provides an upper bound on the individual excess of any vertex, we completely discharge $\Delta$-active vertices (i.e. vertices with an excess $\Delta_i/2 < e(u) \leq \Delta_i$), while maintaining the invariant that no excess rises above $\Delta_i$. At the end of a phase, we guarantee that each vertex has an excess no greater than $\Delta_{i+1} \leq \Delta_i/2$. This generic technique, the Excess-Scaling algorithm of Ahuja and Orlin [2], leads to an algorithm that runs in $O(mn + n^2 \log U)$ time, where $U$ denotes the largest arc capacity. A careful analysis shows that by applying the techniques described in the following paragraph, we reduce the number of phases to $O(m^{1/2})$ (Lemma 7.1).

We adapt a recent technique of Orlin [14] to reduce the amount of time needed to run a $\Delta$-scaling phase. At the beginning of each phase, we construct a compact network consisting of a particular subset of the vertices and arcs of the graph, along with an additional set of pseudoarcs. These pseudoarcs represent directed paths in the residual network. We include all vertices adjacent to approximately medium capacity arcs (which we call $\Delta$-favorable and $\Delta$-large), as well as all $\Delta$-active vertices. Then we construct $\Delta$-abundant and $\Delta$-small pseudoarcs (See Section 8), which represent directed paths consisting of possibly many high-capacity (abundant) arcs, and low-capacity (small) arcs as a single arc in the compact network. When a path is found in the residual network, a pseudoarc to the terminal vertex is created with the capacity of the bottleneck arc on the path (that is, the arc of minimal capacity). Then, the capacity of each arc on the path is decreased by the bottleneck capacity, and the process is repeated (this is called “capacity transfer” in Orlin [14]). We perform a slightly modified version of the push-relabel algorithm on the compact network with the above guarantees.

With a careful analysis of the number of phases that specific arcs and vertices are included
in the compact network, we arrive at a main result of our paper, which is that there are $O(m)$ vertices across all scaling phases in the compact network (Theorem 4.1). Using a similar analysis, we limit the number of active vertices at $O(m)$ as well. Using observations about the behavior and size of saturating and nonsaturating pushes, we use several well-known potential functions from [2, 9] to reach the desired bound of $O(mn)$ nonsaturating pushes across all scaling phases.

**Major Results** This paper negotiates several key technical difficulties associated with performing push-relabel on compact networks. First, the residual network must be updated after each capacity transfer. This is possible through the use of the dynamic trees data structure of Sleator and Tarjan [16], and is detailed in Appendix B. Second, we must guarantee that pushes along pseudoarcs do not violate the capacity constraint for any arc in the original residual network. This does not trivially transfer to the generic push-relabel method, since pushes along internal arcs within pseudoarcs may not be “admissible” in the traditional sense (see Section 12). We relax the Goldberg-Taran admissibility criterion from $d(u) = d(v) + 1$ to simply $d(u) > d(v)$, intuitively maintaining the notion that “flow must go downhill,” and relabel the network at the end of each phase, using a technique due to Goldberg, et. al. [5, 10]. We define the *validity condition* for any arbitrary distance label $dist : V \rightarrow \mathbb{Z}_{\geq 0}$ as, for all arcs $(u,v)$, $dist(u) \leq dist(v) + 1$. Since distance labels can increase due to both low-capacity nonsaturating pushes (those that send $\delta \leq \Delta/2$) and relabel operations, we maintain two distance labels $d_h(\cdot)$ and $d_l(\cdot)$ to keep track of the total. For sake of clarity, we let $d(u) = d_h(u) + d_l(u)$ denote the overall distance of $u$. Relabeling the entire network in the manner described above ensures that the overall distance $d$ obeys the validity condition at the end of a $\Delta$-scaling phase. We also maintain the invariant that throughout the execution of a scaling phase, $d_h$ obeys the validity condition as well; this, in turn, guarantees the correctness of our algorithm (Theorem 4.2). $d_l$ does not necessarily obey the validity condition. However, it is relabeled only under specific circumstances, and thus is used to show that there are $O(mn)$ low-capacity nonsaturating pushes across all scaling phases (Lemma 6.3).

What is left to show is that at the end of every phase, we fulfill our promise, namely, that every vertex $u \in V$ that began the phase with excess $e(u) > \Delta/2$ is completely discharged (i.e. $e(u) = 0$), and the excesses of the remaining vertices have fallen below $\Delta/2$ at the termination of the $i$th scaling phase. In order to prove this, we show that pseudoarc pushes that send flow from some $u$ to $v$ such that $d(u) > d(v)$ allow the potential function $\Phi_g = \sum_{u : e(u) > 0} d(u)$ to behave as it would in the generic push-relabel algorithm if we permit pushes along directed paths (this is formalized in Section 5). Finally, we need to show that our novel scheme for counting low-capacity nonsaturating pushes (ones that send $\delta \leq \Delta/2$) correctly discharges active vertices (Lemma 4.10).

The claimed running time of $O(mn + m^{3/2} \log n)$ (Theorem 7.1) follows from the following main bounds:

1. At the end of every $\Delta$-scaling phase, *global-relabel* generates a valid labeling in $O(m)$ time. Across the $O(m^{1/2})$ scaling phases, the overall cost is $O(m^{3/2})$ (Theorem 4.2);
2. There are $O(mn)$ nonsaturating pushes across all $\Delta$-scaling phases (Theorem 6.1);
3. Both the constructing the compact network $G_C$ and transforming it back into the residual network $G_f$ takes at most $O(m \log n)$ per scaling phase. The cost across all $\Delta$-scaling phases is $O(m^{3/2} \log n)$ (Theorem 7.1).
2 Preliminaries

2.1 Definitions

A flow network is a directed graph $G = (V, A)$, with $|V| = n$ vertices and $|A| = m$ arcs. There are two distinguished vertices, namely the source $s$ and the sink $t$. Further, each pair $(u, v) \in V \times V$ has a non-negative, integer-valued capacity $c : V \times V \rightarrow \mathbb{Z}_{\geq 0}$. If $(u, v) \not\in A$, then $c(u, v) = 0$. The largest arc capacity is denoted by $U$.

A flow is an integer-valued function $f : V \times V \rightarrow \mathbb{Z}_{\geq 0}$ satisfying the capacity and conservation constraints, namely that (1) for all $(u, v) \in A$, $f(u, v) \leq c(u, v)$, and (2) for each $u \in V \setminus \{s, t\}$,

$$\sum_{v \in V} f(v, u) - \sum_{v \in V} f(u, v) = 0.$$ 

That is, the amount of flow entering each vertex $u$ is the same as the amount exiting $u$. We denote the value or magnitude of the flow as the amount leaving the source (or equivalently, entering the sink). That is,

$$|f| = \sum_{u \in V} f(s, u) = \sum_{u \in V} f(u, t).$$

We say that an arc $(u, v) \in A$ is saturated if $f(u, v) = c(u, v)$.

The maximum flow problem is to find a flow $f$ of maximum value. That is, we must find a function $f : V \times V \rightarrow \mathbb{Z}_{\geq 0}$ that obeys both the capacity and conservation constraints and maximizes the flow into the sink $t$.

Next, we define an important concept in designing algorithms for the max-flow problem. The residual capacity with respect to a flow $f$ is an integer-valued function $r : V \times V \rightarrow \mathbb{Z}_{\geq 0}$ that is defined as

$$r(u, v) = \begin{cases} c(u, v) - f(u, v) & \text{if } (u, v) \in A \\ f(u, v) & \text{if } (v, u) \in A \\ 0 & \text{otherwise}. \end{cases}$$

Intuitively, this allows us to express how much more flow can be sent on any given arc. We can assume without loss of generality that for each arc $(u, v) \in A$, the flow is nonnegative for at least $(u, v)$ or $(v, u)$. Let the residual network be the flow network $G_f = (V, A_f)$ consisting of pairs $(u, v) \in V \times V$ with nonzero residual capacity.

Next, we define several terms relating to cuts in the flow network. An $s-t$ path is any simple, directed path $\langle s, v_1, v_2, \ldots, v_k, t \rangle$ in the residual network from the source $s$ to the sink $t$.

An $(S, T)$-cut is a bipartition of the vertex set $V$, such that $s \in S$ and $t \in T$, and $T = \overline{S}$. The capacity of the cut, denoted by $c(S, T)$, is equal to the capacity of the arcs crossing the cut from $S$ to $T$.

We now state two classical results of Ford and Fulkerson [7] that will be used in the analysis of the algorithm. The following is a direct consequence of the Max-Flow Min-Cut Theorem;

**Lemma 2.1.** A flow $f$ is a maximum flow if and only if there does not exist an $s-t$ path in the residual network $G_f$.

**Lemma 2.2** (Flow Decomposition Theorem). Any $s-t$ flow $f$ can be decomposed into flows $f_1 \cdot \ldots \cdot f_k$ on paths $p_1, \ldots, p_k$ in the flow network $G$. Moreover, given a feasible flow $f$ and a maximum flow $f_{\text{max}}$, the flow $f' = f_{\text{max}} - f$ can be decomposed into flows $f'_{1} \cdot \ldots \cdot f'_{k'}$ on paths $p'_{1}, \ldots, p'_{k'}$ in the network $G_f$. 

4
2.2 The Push-Relabel Algorithm and Scaling

We must first introduce the notion of a preflow, and then can proceed to outline the generic push-relabel method of Goldberg and Tarjan \cite{GT88}, as well as Ahuja and Orlin’s approach.

A preflow is a function $f : V \times V \rightarrow \mathbb{Z}_{\geq 0}$ that obeys the capacity constraint, but relaxes the conservation constraint; that is,

$$e(u) = \sum_{v \in V} f(v, u) - \sum_{v \in V} f(u, v) \geq 0,$$

for all $u \in V \setminus \{s, t\}$.

Intuitively, vertices may overflow. We define this discrepancy as the excess, denoted by the function $e : V \rightarrow \mathbb{Z}_{\geq 0}$.

A distance labeling is a function $d : V \rightarrow \mathbb{Z}_{\geq 0}$ that associates each vertex with a positive integer. Goldberg and Tarjan define $d$ to be a valid label if $d(s) = n$, $d(t) = 0$, and for all $(u, v) \in A_f$, $d(u) \leq d(v) + 1$. The distance label allows us to push flow “downhill.” More formally, flow is sent along admissible arcs; in Goldberg and Tarjan’s approach these are the arcs $(u, v) \in A_f$ where $d(u) = d(v) + 1$.

We generalize this notion and redefine admissible as simply $d(u) > d(v)$. In Section 5 we show that one can still prove the correctness of the push-relabel algorithm using this definition of admissible arc, and increase the running time by at most a multiplicative constant. In addition, we apply the Goldberg-Tarjan notion of validity only to $d_h$; that is, $d_h$ is valid if, for all arcs $(u, v)$, $d_h(u) \leq d_h(v) + 1$. We do not require such criterion for the labeling $d_e$, and regard it as a “restricted” distance label, of sorts; we only increment $d_e$ under only very specific circumstances, as outlined in Section 4.1. Below, we formalize what we mean when we refer to a label being valid;

**Condition 1** (Validity Condition). A distance label $\ell : V \rightarrow \mathbb{Z}_{\geq 0}$ satisfies the validity condition if, for all arcs $(u, v)$, $\ell(u) \leq \ell(v) + 1$.

The push-relabel family of algorithms, introduced by Goldberg and Tarjan \cite{GT88} maintains a preflow $f$, and iteratively “pushes” flow from some overflowing vertex $u$ to $v$ along admissible arcs. A vertex is “relabelled” when it is not incident to any admissible arcs.

Ahuja and Orlin \cite{AO86} modified the push-relabel algorithm to use a capacity scaling approach. Their algorithm (as well as ours) redefines an active vertex to be any $u \in V$ where $\Delta/2 < e(u) \leq \Delta$, where $\Delta$, the excess dominator, is an upper bound on the excess. By sending flow from $u = \min\{d(w)|\Delta/2 < e(w) \leq \Delta\}$ (the active vertex with the smallest label), we can guarantee that flow is sent to a vertex $v$ such that $e(v) \leq \Delta/2$ (Lemma 4.7).

The Ahuja-Orlin algorithm performs a series of $O(\log U)$ scaling phases to find a max-flow. It maintains the invariant that, at the conclusion of a scaling phase, the excesses of all vertices fall below $\Delta/2$. This discharging scheme takes $O(n^2)$ per scaling phase, yielding an algorithm which takes $O(mn + n^2 \log U)$ time. Under the assumption that $U = \text{poly}(n)$, their algorithm achieves an $O(mn)$ running time on networks that are non-sparse and non-dense (i.e. where $m = \Theta(n^{1+\varepsilon})$). Our main improvement to the Ahuja-Orlin algorithm is reducing the number of nonsaturating pushes to $O(mn)$ across all scaling phases, which we succeed in proving for bounded-degree networks. This is detailed in Section 9. We conclude this section by stating two results of Goldberg and Tarjan \cite{GT88}. The first is used to prove correctness, while the second is a tool in deriving the time bound. The following lemma is reworded in terms of our terminology.

**Lemma 2.3** (Lemma 3.3 of \cite{GT88}). If $f$ is a preflow and $d$ is a distance label obeying the validity condition, then there is no $s-t$ path in the residual network $G_f$.

It is important to note later on that Lemma 2.3 will not be affected by our change in the admissibility criterion to $d(u) > d(v)$. Since the label $d_h$, which is defined and elaborated on in
Section 4.4 is relabeled according to the same procedure in the generic push-relabel algorithm, the proof is not altered.

**Lemma 2.4** (Lemma 3.9 of [9]). The number of saturating pushes is at most $\ell m$, where $d(u) \leq \ell$, for all $u \in V$.

In the original push-relabel algorithm of Goldberg and Tarjan [9], $\ell \leq 2n$. In Section 4.4, we will prove that $\ell \leq 6n$, and that consequently, our algorithm uses at most $6mn$ saturating pushes. Again, the proof of Lemma 2.3 carries over from our result in Section 3, which shows that the potential function of the generic push-relabel algorithm behaves similarly with the new admissibility criterion.

### 3 Abundant Arcs

In order to reduce the running time of each scaling phase, we classify the set of arcs $A$ into several categories based on their residual capacity. This allows us to disregard arcs that are too small or too big to be of use to us when pushing flow from active vertices.

We let the **compaction capacity** of an arc $(u, v) \in A_f$ be defined as $\gamma(u, v) = r(u, v) + r(v, u)$, where $r(u, v)$ is the residual capacity of $(u, v)$. That is, $\gamma(u, v)$ is the residual capacity between two vertices. Recall that $\Delta$ denotes the excess dominator, which provides an upper bound on the excess during each scaling phase, and that $\Delta_i$ denotes the excess dominator for the $i$th scaling phase. We now use the compaction capacity to describe several categories of arcs. An arc $(u, v)$ is $\Delta_i$-favorable if $\Delta_i/4 < \gamma(u, v) \leq \Delta_i/2$. It is $\Delta_i$-large if $\Delta_i/4 \leq \gamma(u, v) \leq 2\Delta_i$, and $r(u, v) \leq \Delta_i$. It is $\Delta_i$-abundant if $r(u, v) > \Delta_i$.

The next lemma follows from our improvement property, namely $\Delta_{i+1} \leq \Delta_i/2$

**Lemma 3.1.** If an arc $(u, v)$ is $\Delta_i$-abundant during the $i$th scaling phase, then it will be $\Delta_{i+1}$-abundant, and consequently $\Delta_{i'}$-abundant for every subsequent $\Delta_{i'}$-scaling phase.

### 4 Compact Networks

The compact network, built at the beginning of each phase, is a modified version of the residual network, with two specific differences. First, we eliminate non-active vertices incident to arcs of high capacity, and construct high-capacity pseudoarcs, which represent paths of these high-capacity arcs, in order to perform a sequence of pushes that would have gone through these vertices. Second, we create low-capacity pseudoarcs, so that we can perform sequences of saturating pushes in one operation. We begin by introducing notation, and then give an algorithm for creating the compact network $G_C$.

A pseudoarc is $\Delta$-abundant if it consists entirely of abundant arcs. A pseudoarc is $\Delta$-small when it contains a non-abundant arc. $V_A$ denotes the set of active vertices; that is, the vertices $u \in V$ that at the beginning of the phase have $\Delta/2 < e(u) \leq \Delta$. $V_{SC}$ denotes the set of vertices not in $V_A$ that are incident to $\Delta$-favorable or $\Delta$-large arcs. We let $V_C = V_{SC} \cup V_A$.

The set $A_1$ denotes the set of original arcs from $A_f$ that are included in the compact network (that is, the $\Delta$-favorable and $\Delta$-large arcs). $A_2$ denotes the set of pseudoarcs. $A_C = A_1 \cup A_2$.

#### 4.1 Creating the Compact Network

We now describe a procedure for creating the compact network $G_C$ (Algorithm 1). Before we can do this, however, we define the abundance graph; $G^{ab} = (V \setminus V_A, A^{ab})$, where $A^{ab}$ contains all arcs $(u, v)$ such that $r(u, v) > \Delta$, and the vertex set excludes any $\Delta$-active vertices. We
use this subgraph of $G$ to efficiently construct pseudoarcs. The motivation behind using this
subgraph stems from the fact that we cannot include any active vertex $u \in V_A$ along a pseudoarc;
consider the situation where create-all-pseudoarcs iteratively transfers capacity away from a
vertex $u \in V_A$, such that there is no original arc left incident to $u \in V_A$ after create-all-
pseudoarcs has terminated. If this occurs, we will not be able to discharge $u$ at all. In Appendix
\[C\] we describe an efficient method to delete active vertices from the network.

**Algorithm 1.** An algorithm for creating compact networks

*Input:* The residual network $G_f$, $\Delta$.

*Output:* The compact network $G_C$.

**Step 1a.** Let $A_1$ denote the set containing all $\Delta$-favorable and $\Delta$-large arcs.

**Step 1b.** Let $V_A$ denote the vertices that are active at the beginning of the phase. Let $V_C$ denote vertices incident to arcs in $A_1$.

**Step 2a.** Construct $\Delta$-abundant pseudoarcs by calling create-all-pseudoarcs on $G_{ab}$ with $\rho = \Delta$.

**Step 2b.** Construct $\Delta$-small pseudoarcs by calling create-all-pseudoarcs on $G_f = (V \setminus V_A, A_f)$ with $\rho = 0$.

**Step 3.** Collect all pseudoarcs in $A_2$ and return $G_C = (V_{SC} \cup V_A, A_1 \cup A_2)$.

**Step 4.** Generate a valid distance label $d_h$ by calling global-relabel on $G_C$.

**Step 5.** For each $u \in V_{SC} \cup V_A$ initialize edge-list $[u]$ to contain admissible arcs $(u, v)$, ordered by $d_h(v)$.

The algorithms for constructing pseudoarcs are detailed in Appendix \[B\]. We first show that
the structure of arcs incident to active vertices is preserved between $G_f$ and $G_C$:

**Lemma 4.1.** For all $u \in V_A$, $\deg_f[u] = \deg_C[u]$, where $\deg_f$ denotes the degree in the residual
network and $\deg_C$ denotes the degree in the compact network.

*Proof.* This follows trivially from Algorithm \[1\]. Only steps 2a and 2b construct pseudoarcs; in
both cases, the set of active vertices $V_A$ is excluded from the algorithm. Therefore, $\deg_f[u] = \deg_C[u]$.

**Lemma 4.2.** During a $\Delta$-scaling phase, if there is an arc $(u, v)$ where $\Delta/4 < \gamma(u, v) \leq 2\Delta$, at
least $(u, v)$ or $(v, u)$ is in the compact network $G_C$ as an original arc.

*Proof.* Consider the case where $\Delta/4 < \gamma(u, v) \leq \Delta/2$. If this is true, then $(u, v)$ is $\Delta$-favorable,
and by Step 1a of the algorithm it is included in the compact network. If $\Delta/2 < \gamma(u, v) \leq 2\Delta$,
then $(u, v)$ will only be in $G_C$ if $r(u, v) \leq \Delta$. But, if neither $r(u, v) \leq \Delta$ nor $r(v, u) \leq \Delta$ then
$r(u, v) + r(v, u) > 2\Delta$, which is a contradiction.

Our next lemma shows that the procedure create-all-pseudoarcs will correctly construct the
abundant and small pseudoarcs. We show how to implement these algorithms using dynamic
trees in Appendix \[B\].

**Lemma 4.3** (Capacity Transfer Lemma). A push on any pseudoarc ($\Delta$-small or $\Delta$-abundant)
created by the procedure create-all-pseudoarcs always corresponds to a push in $G_f$ that does not
violate the capacity constraints.
Proof. The only possible difficulty would be the case where pseudoarcs “share” an arc. More formally, given \( k \) pseudoarcs \( (u_1, w_1) = (u_1, v_1, v_2, \ldots, v_k, w_1), (u_2, w_2) = (u_2, v_1, v_2, \ldots, v_k, w_2), \ldots (u_k, w_k) \), suppose that the arc \((v_i, v_{i+1}) \in (u_1, w)\) is also contained in \((u_2, w_2)\ldots(u_k, w_k)\).

In the procedure transfer-capacity in Appendix\[B\] once we construct a pseudoarc, we transfer capacity to the pseudoarc and then reduce the path capacity for subsequent pseudoarcs to be constructed. In fact, the multiple pseudoarcs created by this procedure that share one or more internal arcs will still correspond to pushes in \( G \) constructed. In fact, the multiple pseudoarcs created by this procedure that share one or more internal arcs will still correspond to pushes in \( G_f \) that do not violate capacity constraints. The arc \((v_i, v_{i+1})\) may have a reduced capacity on \((u_2, w_2)\) after \((u_1, w_1)\) was constructed; this corresponds to the situation where a push along the sequence of paths in \((u_1, w_1)\) occurs before any push from \( u_2, \ldots, u_k \). This will correspond to a possible sequence of \( k \) pushes in the original residual network, according to the decreasing residual capacity of the shared arc \((v_i, v_{i+1})\).

Finally, we will bound the number of \( \Delta \)-favorable and large arcs in the compact network.

**Lemma 4.4.** A \( \Delta_i \)-favorable arc \((u, v)\) during the \( i \)th scaling phase appears in the compact network at most 3 times. Moreover, it will be \( \Delta_{i+3} \)-abundant in 4 scaling phases.

**Proof.** If \((u, v)\) is \( \Delta_i \)-favorable, then we know \( \Delta_i/4 < \gamma(u, v) \leq \Delta_i/2 \). By our improvement property, \( \Delta \) decreases by a factor of at least 2 during each scaling phase. Applying the property, we get \( \Delta_{i+1} \leq \Delta_i/2 \). In the \((i + 1) \)st scaling phase, we can bound the compaction capacity of \((u, v)\) as \( \Delta_{i+1}/2 < \gamma(u, v) \leq \Delta_{i+1} \). Let \( \Delta_{i+3} \leq \Delta_{i+1}/4 \). Thus, we can bound \( \gamma(u, v) \) as \( 2\Delta_{i+3} < \gamma(u, v) \leq 4\Delta_{i+3} \), which is clearly \( \Delta_{i+3} \)-abundant. Therefore, any \( \Delta_i \)-favorable arc \((u, v)\) remains in the compact network at most 3 phases, after which it becomes \( \Delta_{i+3} \)-abundant, and by Lemma 3.1, remains \( \Delta' \)-abundant for every subsequent excess dominator \( \Delta' \).

Recall that, trivially, once an arc is \( \Delta_i \)-abundant, it is not \( \Delta_i \)-favorable, and will not be \( \Delta_{i+1} \)-large, so it will appear in \( G_C \) in at most one more phase. This bound becomes important when bounding the size of the compact network across all phases. But before we are able to accomplish this, we must describe a scheme for discharging active vertices (i.e. those with excess \( \Delta/2 < e(u) \leq \Delta \)).

### 4.2 Discharging Active Vertices

Now that we have bounded the number of \( \Delta \)-favorable and large arcs in the compact network, we must bound the number of active vertices included at the start of a scaling phase (that is, vertices that begin the \( \Delta \)-scaling phase with \( \Delta/2 < e(u) \leq \Delta \)). We first define the basic initialize and push procedures, and then the discharge procedure.

Recall that \( d_h \) and \( d_\ell \) are two distinct vectors that both contribute to the distance labeling of each \( u \in V_A \), and that for all \( u \in V_A \), \( d(u) = d_h(u) + d_\ell(u) \). In the initialize procedure, we will use \( d_h \) (since we maintain the invariant that \( d_h \) is valid within a \( \Delta \)-scaling phase).

**Procedure** initialize\((G)\)

01. Initialize a new preflow \( f \) and distance labels \( d_h, d_\ell \);
02. \( d_h(s) := n \) and \( d_h(u) := 0, \) for all \( u \in V \setminus \{s\} \);
03. for all \( v \in V \setminus \{s\} \) do
   04. if \((u, v) \in A_f \) then \( f(u, v) := c(u, v) \);

**Procedure** push\((u, v)\)

01. //Applies when \( e(u) > 0, d(u) > d(v) \) //
02. \( \delta = \min\{e(u), r(u, v), \Delta - e(v)\} \);
03. \( f(u, v) = f(u, v) + \delta \);
04. \( f(v, u) = f(v, u) - \delta \);

We now can describe the discharge procedure. It will (1) push or relabel \( u \) until \( e(u) < \Delta/2 \) for \( u \in V_{SC} \), and (2) until \( e(u) = 0 \) for \( u \in V_A \). This latter condition ensures that an active vertex in phase \( i \) can only appear in \( G_C \) in phase \( i + 1 \) as a member of \( V_{SC} \), i.e., because it is adjacent to a \( \Delta \)-favorable edge. We present details in Lemma 4.11.

We maintain the invariant throughout the algorithm that within a \( \Delta \)-scaling phase, there is only 1 low-capacity nonsaturating push (i.e. that sends \( \delta \leq \Delta/2 \) units of flow) for each value of the distance label \( d_{\ell} \), for \( u \in V_A \). We enforce this by incrementing the distance label \( d_{\ell} \), and then making the push. We ensure that there is only one low-capacity nonsaturating push for each value of \( d_{\ell} \) by utilizing two data structures; a boolean list \( nonsat_u[\cdot] \) that keeps track of nonsaturating pushes for each value of \( d_{\ell} \). Second, we maintain a current edge list \( edge-list[u] \) that orders the admissible arcs adjacent to \( u \) by \( d_h \). This way, we do not alter the order in which pushes would have been made in the generic push-relabel algorithm, across any sequence of increments to \( d_{\ell} \). These are both maintained as global variables for the duration of a \( \Delta \)-scaling phase. We let \( \delta \) denote the amount of flow sent by a push operation.

Algorithm 2. The algorithm for discharging vertices during a \( \Delta \)-scaling phase.

\begin{verbatim}
Procedure discharge(G_C, u)
01. if u ∈ V_A then
02. while e(u) > 0 do
03.   if e(u) > \Delta/2 then push or relabel u;
04.   else if e(u) ≤ \Delta/2 then
05.     if d_{\ell}(u) = 4n - 1 then
06.       push or relabel u until e(u) = 0;
07.     else if edge-list[u] = ∅ then relabel u;
08.     else if nonsat_u[d_{\ell}(u)] = true and
09.       r(edge-list[u]) > \delta then
10.       d_{\ell}(u) := d_{\ell}(u) + 1;
11.       nonsat_u[d_{\ell}(u)] := false;
12.       Add new admissible edges to
13.       to edge-list[u] ordered by d_h;
14.       while nonsat_u[d_{\ell}(u)] = false do push on
15.         (u, v) := dequeue(edge-list[u]) or relabel u;
16.       if edge-list[u] = ∅ then relabel u;
17.       until there is a nonsaturating push;
18.       nonsat_u[d_{\ell}(u)] := true;
19. else if u ∉ V_A then
20.   while e(u) > \Delta/2 do
21.     push or relabel u;
\end{verbatim}

The next lemma follows directly from the definition of a \( \Delta \)-favorable arc and the discharge procedure;

**Lemma 4.5.** A push along any \( \Delta \)-favorable arc \((u, v)\), when \( u \in V_{SC} \) will always be saturating.

**Proof.** If \( u \in V_{SC} \) during the \( \Delta \)-scaling phase, then no push originating from \( u \) will send less than \( \Delta/2 \) units of flow; this follows directly from discharge, since we do not require any \( u \not\in V_A \).
to satisfy $e(u) = 0$ at the end of the $\Delta$-scaling phase. Thus, we only require that $e(u) < \Delta_k/2$ at the end of the phase. Now, recall that an arc $(v, w)$ is $\Delta_r$-favorable, then $\Delta/4 \leq \gamma(u, v) \leq \Delta/2$. Thus, the capacity of $(v, w)$ during the execution of a scaling phase is at most $\Delta/2$. Since each push from $u$ sends at least $\Delta_i/2$ units of flow, every $\Delta_i$-favorable arc will be saturated by a push from $u \in V_{SC}$. □

In every $\Delta$-scaling phase, we iteratively call discharge until we can proceed to the next scaling phase. Note that for some $u \in V_A$, when the excess is $e(u) \leq \Delta/2$, we modify the manner in which we discharge vertices. For reasons that become clear in the analysis of the algorithm, we divide nonsaturating pushes into high-capacity and low-capacity, when the flow $\delta$ sent is $\delta > \Delta/2$ or $\delta \leq \Delta/2$, respectively.

Note that we limit $d_\ell(u)$ to have the maximum value $4n - 1$. For each $u \in V_A$, we only permit one low-capacity nonsaturating push per value of $d_\ell$, until $d_\ell(u) = 4n - 1$. Then, we allow a second group of nonsaturating pushes to be made. We will show that there are at most $O(n)$ nonsaturating pushes per group of pushes, and that this only happens twice during the execution of a scaling phase (Lemma 4.10). We state this formally in the lemma below:

**Lemma 4.6.** There is at most one low-capacity nonsaturating push (that is, a push that sends $\delta \leq \Delta/2$) for each value of $d_\ell(u)$, for each $u \in V_A$, until $d_\ell = 4n - 1$, after which there will be only $2n$ more such pushes.

The next lemma follows the approach of Lemma 5 from [2];

**Lemma 4.7.** The algorithm obeys the following conditions:

**C1:** Each nonsaturating push from an active vertex from $u \in V_{SC}$ will send at least $\delta > \Delta/2$ units of flow.

**C2:** No excesses rise above $\Delta$.

**Proof.** Consider a push on $(u, v)$. If $u \in V_{SC}$, then $e(u) > \Delta/2$, and $e(v) \leq \Delta/2$, since $u$ is the vertex with the minimum distance label among vertices in $V_{SC}$ such that $e(u)$ is active. Since $d(u) > d(v)$ as well by our admissibility criterion, we send $\delta = \min\{e(u), r(u, v), \Delta - e(v)\} \geq \min\{\Delta/2, r(u, v)\}$. In the case that $\min\{\Delta/2, r(u, v)\} = \Delta/2$, the push will be nonsaturating, but will send $\delta \geq \Delta/2$. This concludes the proof of C1. C2 follows immediately from line 02 in the push procedure, which will $\delta \leq \Delta - e(v)$ for any push along $(u, v)$. □

Next, we will prove two lemmas that ensure each $u \in V_A$ can be discharged at the end of a $\Delta$-scaling phase. We start with a technical lemma regarding path decompositions in the residual network, and then bound $|V_A|$ across all scaling phases.

**Lemma 4.8 (Path Decomposition Lemma).** For each $u \in V_C$ such that $e(u) > 0$, the $s - u$ preflow can be decomposed into a collection of paths $P$ such that $e(u)$ units of flow can be returned to $s$ along paths $p \in P$. Moreover,

$$C = \sum_{p \in P} \min_{(i, j) \in p} r(i, j) \geq e(u).$$

That is, the sum of the minimum-capacity arc in each path $p$ in the decomposition $P$ yields a capacity of at least $e(u)$.

**Proof.** The proof relies on the construction of a new flow network, which we denote as $G_{s,u}$. The construction is as follows; consider all paths $s \rightarrow u$ in $G_C$. Denote the set of these paths as $P_{s,u}$. Now, let $V(G_{s,u})$ consist of the set of all vertices contained in paths $p \in P_{s,u}$, and similarly let $A(G_{s,u})$ consist of the arcs from these paths.
Now, let the source vertex of \( G_{s,u} \) be \( s' = u \), and let the sink \( t' = s \). Finally, for each \( v \in V(G_{s,u}) \backslash \{s', t'\} \), let \( e_{s,u} = 0 \) (where \( e_{s,u} : V \to \mathbb{Z}_{\geq 0} \) denotes the excess for \( G_{s,u} \)). Thus, we eliminate the excess on all vertices in \( G_{s,u} \). We remark that there is an \( s' - t' \) path in \( G_{s,u} \) since \( e(u) > 0 \) in \( G_C \); when \( e(u) \) is positive, there is always a path from \( u \) to \( s \). This ensures the existence of an \( s' - t' \) path.

Finally, we can apply the Flow Decomposition Theorem of Ford and Fulkerson to \( G_{s,u} \). That is, we know that there exists a set of feasible flows \( f_1, \ldots, f_k \) associated with \( s' - t' \) paths \( p_1, \ldots, p_k \) such that \( \forall i, f_i \) sends positive flow only on \( p_i \). This guarantees the existence of paths \( \mathcal{P} = \{p_i \mid 1 \leq i \leq k\} \) such that the excess from \( u \) in the compact network \( G_C \) can be returned to \( s \) along paths in \( \mathcal{P} \).

Moreover, we show that the second claim holds; we will prove that \( e(u) \) places a lower bound on the sum of the capacities of the bottleneck (minimum-capacity) arcs for each \( p \in P \). An arc \( (u, v) \in A_f \) is a bottleneck arc on a path \( p \) if \( r(u, v) = \min_{(i, j) \in p} r(i, j) \). When this arc is saturated, \( p \) is sending the maximum amount of flow possible from \( s \) to \( u \). When \( C = e(u) \), we see that all bottleneck arcs are saturated, since the preflow \( f \) obeys the capacity constraint. When \( C > e(u) \), then some bottleneck arcs are left unsaturated. This guarantees that there always exists a path \( p \) or a collection of paths \( p_1 \ldots p_k \) such that flow can be returned to the source from \( u \) with sufficient relabelings (sufficient relabelings will make paths from the decomposition admissible, so therefore flow can be returned to \( s \)).

Since Lemma 4.8 holds, we can immediately see that flow can be returned to the source from each active vertex after sufficient relabelings. Next, we two crucial lemmas that guarantee the algorithm’s correctness. We first prove that we can efficiently order the current edge list by \( d_h \), and then that we are able to discharge all active vertices at the conclusion of a \( \Delta \)-scaling phase.

**Lemma 4.9.** For each \( u \in V_C \) during a \( \Delta \)-scaling phase, \text{edge-list}[u] \ is in increasing order in terms of \( d_h \). Formally, \( d_h(u_1) \leq d_h(u_2) \leq \ldots \leq d_h(u_k) \) for each of the \( k \) incident arcs on \( u \).

**Proof.** We can use the buckets data structure described in Ahuja and Orlin [2] to initially order the current edge list when we construct \( G_C \). What is left to show is that \text{edge-list}[u] \ remains ordered after a sequence of edge additions. We can maintain this as a priority queue keyed by \( d_h(v) \), for each \( v \in \text{edge-list}[u] \).

We remark that additions can be accomplished in \( O(1) \) time, due to the fact that \( k \), the maximum in-degree, is constant. Thus, we can implement this with a simple buckets data structure described by Ahuja and Orlin [2]. This will cost \( O(mn) \) across all phases. Next, we prove that discharge fulfills its promise, namely that it correctly discharges each active vertex \( u \in V_A \), so that \( e(u) = 0 \) at the termination of a phase.

**Lemma 4.10.** For each \( u \in V_A \), \( e(u) = 0 \) at the conclusion of the \( \Delta \)-scaling phase.

**Proof.** We first make a remark about an invariant that holds for the duration of a “discharge” on some \( u \in V_C \); once an arc \( (u, v) \) incident to \( u \) becomes admissible, it will stay admissible while \( u \) is selected by discharge (since \( d(u) > d(v) \), not \( d(u) = d(v) + 1 \)). Therefore, despite the increments to \( d_h \), we will still be able to select arcs from \text{edge-list}[u] \ in order to send flow on, even if they become admissible under a previous increment.

By Lemma 4.9 we know that the current edge list is in increasing order by \( d_h \). Since \( d_h \) satisfies the validity condition (Lemma 4.13), it provides an estimate of the distance from any \( u \) to the sink \( t \). The ordering of \text{edge-list} becomes important in the following case; consider a vertex \( u \) that is selected by discharge. Then there may be an edge \( (u, s) \) within \( G_C \) that
can become admissible as soon as \( d(u) = 2n + 1 \). However, due to the ordering of the edges, a push along \((u, s)\) will come after all edges that have a chance of sending excess “forward” (i.e. to the sink \( t \)) before it is returned to \( s \). This is important because of our altered labeling \( d = d_h + d_\ell \), we may relabel at different times. Thus, we wish to avoid returning flow to the source \( s \) prematurely.

The final component of the correctness comes from the two groups in which we complete small nonsaturating pushes. We first make one low-capacity nonsaturating push for each value of \( d_\ell \), while \( d_\ell(u) \leq 4n - 1 \). Afterwards, we will discharge \( u \) so that \( e(u) = 0 \) after it reaches the maximum label, ignoring the limitation of one low-capacity nonsaturating push per value of \( d_\ell \). However, since \( \deg[u] = O(1) \) and there are only \( 2n \) distinct values of \( d_h(u) \) from which \( d(v) \) could exceed \( d(u) \), there will be at most \( 2n \) such small nonsaturating pushes made. Therefore, we conclude that all \( u \in V_A \) are completely discharged.

The previous two lemmas underscore the importance that the maximum degree of a vertex is a constant \( k \). This permits us to efficiently maintain the ordering of the current-edge list, so we always send flow towards the sink \( t \) before the source \( s \). If we allowed general networks, where \( \max\{\deg[u] | u \in V\} = O(n) \), then each push would potentially take \( O(\log n) \) time due to the nature of the priority queue implementation and worst-case bounds on the data structure.

We now state a main result of our paper regarding the number of active vertices in the compact network.

**Lemma 4.11** (Active Vertices Lemma). For a vertex \( u \in V_i^C \) such that \( u \) is \( \Delta_i \)-active, if \( u \in V_{i+1}^C \) then \( u \) is incident to a \( \Delta_{i+1} \)-large or \( \Delta_{i+1} \)-favorable arc.

**Proof.** If some vertex \( u \in V_i^C \) is active during the \( \Delta_i \)-scaling phase, then by the discharge procedure, \( e(u) = 0 \) will be true when the phase terminates. That is, \( u \) will not be active in the \( \Delta_{i+1} \)-scaling phase. The only way \( u \in V_i^{C+1} \) holds is if \( u \in V_{i+1}^{SC} \); that is, it was included as an endpoint of a \( \Delta_{i+1} \)-favorable or large arc. Thus, we can see that it will only take one phase for an active vertex \( u \in V_A \) to become a vertex contained in the set \( V_{SC} \) in the next scaling phase.

Note that if we did not completely discharge vertices that were active at the start of a \( \Delta \)-scaling phase, then we could have \( |V_C| = \Omega(n) \) active vertices in the compact network during each scaling phase. Rather, Lemma 4.11 allows us to bound the number of vertices in the compact network across all scaling phases in Theorem 4.1. Further, since we shift flow along each vertex within a pseudoarc, proof of the following lemma is immediate.

**Lemma 4.12.** A push along a pseudoarc \((u, w) = \langle u, v_1, v_2, \ldots, v_k, w \rangle\) will not alter the excesses of any internal vertex \( v_1 \ldots v_k \) that is included within the pseudoarc.

This guarantees that no vertex that appears on a pseudoarc will become active as a result of a pseudoarc push.

### 4.3 Size of the Compact Network

**Theorem 4.1** (Size of the Compact Network). There are \( O(m) \) vertices in the compact network across all scaling phases.

**Proof.** This theorem immediately follows from Lemmas 4.4 and 4.11. In particular, Lemma 4.11 shows that we can convert an active vertex in the \( \Delta_i \) scaling phase to a vertex in \( V_{SC} \) in the \( \Delta_{i+1} \) scaling phase; it remains active for \( O(1) \) phases. Thus, the theorem holds.
This theorem underscores the significance of our algorithm; superficially, it may seem that we maintain paths of “medium” and “large” arcs so that we can conduct sequences of saturating and/or nonsaturating pushes. However, the idea of compaction is what leads to our improved running time. Because we can bound the total number of vertices that will be eligible for flow operations at $O(m)$ across all phases, we are able to reduce the number of nonsaturating pushes to $O(mn)$ across all scaling phases. An important distinction between our algorithm and Orlin’s [14] is that we do not explicitly “compact” active vertices $u \in V_A$ out of the network after $O(1)$ scaling phases. Instead, we show that they are “converted” (in a sense) to a vertex $u \in V_{SC}$ if they remain in the network after they are discharged. This correspondence, while not permitting us to bound the number of active vertices at $O(n)$, does match the number of vertices in $V_{SC}$, and still shows that the network is indeed compact.

4.4 Labelings in Compact Networks

In this section, we describe how to maintain a valid labeling during each $\Delta$-scaling phase. We resolve two difficulties, namely maintaining a valid labeling between $G_C$ and $G_f$. To accomplish this, we use a technique due to Goldberg, et. al. [5, 10], which generates a valid labeling from a BFS-tree. Next, we show how we use distinct labelings for high-capacity nonsaturating pushes $(d_h)$, and for low-capacity nonsaturating pushes $(d_\ell)$ to count the number of nonsaturating pushes.

Procedure `relabel(G, u)`

01. //Applies when $\nexists$ an admissible arc $(u, v) \in A_C$/
02. $d_h(u) := \min\{d_h(v) | (u, v) \in A\} + 1$;
03. if $d(u) < d(v)$ then $d_\ell(u) := d_\ell(v)$;

The `relabel` procedure in our algorithm contains a few notable departures from the analogous procedure in [9, 2]. We note that in lines 02, $d(u)$ may still be less than $d(v)$ after a relabeling. This is due to the fact that $d_\ell(v)$ could have increased a large amount when compared to $d_\ell(u)$, and even though $d_h(u) = d_h(v) + 1$, the discrepancy exists in the overall labeling $d$. We resolve this by performing a “gap relabel” on $d_\ell$ if $d(u) < d(v)$ (line 03 of the `relabel` procedure).

The next lemma shows that $d_h$ is bounded above by $2n$, for all $u \in V$, and is proven in [9].

**Lemma 4.13.** For any $u \in V$, $d_h(u) < 2n$.

**Proof.** Since we know that there exists a path from $u$ to $s$ for each $u \in V_A$, we consider the case that gives us the maximum label. Recall that we only modify by additions of 1 ($d_h(u) = d_h(v) + 1$). Thus, $d_h$ remains a valid labeling during the execution of a $\Delta$-scaling phase. A simple path has length at most $n - 1$, so therefore we have $d(s) = n$, so max $d_h = 2n - 1$. □

Since we generate a valid distance labeling at the beginning of the $\Delta$-scaling phase with `global-relabel`, and only increase $d_h$ when there are no incident admissible arcs, this will behave in the same manner as the labeling in the generic push-relabel algorithm [9], as well as the excess-scaling algorithm [2]. The distance label will only increase by 1 due to each `relabel` operation, so proof of the lemma still follows.

Next, we must consider increments in the distance label from `discharge` that result from a low-capacity nonsaturating push. The following lemma will show that $d_\ell$ also will not increase above $2n$.

**Lemma 4.14.** For all $u \in V$, there is an increase in the overall distance $d(u)$ by $4n - 1$ due to both gap-labelings and low-capacity nonsaturating pushes.
Proof. Each increment in $d_{\ell}$ increases the label by 1. When we need to make an additional nonsaturating push, the discharge procedure will increment the label. Since we limit $d_{\ell}(u)$ at $4n - 1$, it is clear that this value bounds the increase.

The previous two lemmas allow us to state a final conclusion that bounds the total increase in the overall label $d$.

**Lemma 4.15.** For each $u \in V$, $d_h < 2n$, and the increase to $d_{\ell}$ due to low-capacity nonsaturating pushes is $4n$. Moreover, there are $6n^2 = O(n^2)$ relabelings throughout the algorithm.

**Proof.** Proof is immediate from Lemmas 4.13 and 4.14.

It is important to remark that even though we maintain two distinct vectors ($d_h$ and $d_{\ell}$) for the labeling, the algorithm’s correctness remains. More formally, we can ensure that we will never make a nonsaturating push such that the value of the potential function is increased. This is due to our initial definition of $d(u) = d_h(u) + d_{\ell}(u)$. We will allow a push to occur only if $d(u) > d(v) \Rightarrow d_h(u) + d_{\ell}(u) > d_h(v) + d_{\ell}(v)$. We will relabel $u$ until an arc incident to $u$ is admissible, an operation which is bounded at $2n$. Therefore, the potential function will still decrease with each nonsaturating push, and correctness remains.

Next, we describe a variant of the global relabel procedure of Goldberg. This algorithm relies on breadth-first search to traverse the graph and generate a labeling based on the levels in the BFS tree.

**Procedure global-relabel($G$)**

01. Run reverse BFS from $t$ and let $d_t$ denote the distances from the BFS tree;
02. Run BFS from $s$ and let $d_s$ denote the distances from the BFS tree;
03. for all $u \in V$
04.   $d_h(u) := \min\{d_s(u) + n, d_t(u)\}$;
05.   $d_{\ell}(u) := 0$;
06. return $d$;

We will use the global-relabel procedure twice. Once we build the network, we generate a valid labeling, incorporating pseud-arcs. Then, we can simply push and relabel vertices as in the generic algorithm. The second time we apply it is when transforming the compact preflow $f$ into the residual preflow $f'$. The global-relabel procedure costs $O(m)$ time per scaling phase, since we run BFS twice. Now, we prove that the algorithm maintains a valid labeling across all scaling phases.

**Theorem 4.2** (Valid Labeling). Within each $\Delta$-scaling phase, $d_h$ obeys the validity condition. Moreover, at the end of each phase, we generate a labeling for $G_f$ that obeys the validity condition.

**Proof.** First, consider initialization. Since $d(s) = n$ and $d(u) = 0$ for each $u \in V\{s\}$ when the algorithm starts, the labeling is trivially valid. Within each $\Delta$-scaling phase, we claim that $d_h$ obeys the validity condition (i.e. $\forall (u, v) \in A$, $d(u) \leq d(v) + 1$). We clearly see that within relabel, we will increase $d_h(u)$ by at most $d(v) + 1$. Thus, each invocation of relabel maintains the validity of $d_h$. Applying this argument inductively, we see that $d_h$ remains valid across a sequence of push and relabel operations.

Furthermore, in global-relabel, we see that if we retrieve labels directly from the BFS tree, then validity is immediate (this is also discussed in [10]). This ensures that all forward arcs $(u, v) \in A$ are labeled such that $d(u) \leq d(v) + 1$. This means that the labeling $d_h$ in the compact
network is valid while we are running a $\Delta$-scaling phase, and further, the labeling in the residual network is valid once we have transformed the compact preflow into the residual preflow.

5 Validity of Pushes in the Compact Network

When flow is sent along a pseudoarc $(u, w) \in A_2$, there may be pushes that are inadmissible (that is, pushes that send flow from $v_i$ to $w_{i+1}$ when $d(v_i) \leq d(w_{i+1})$). In this section, we show that as long as flow is eventually sent to a lower-labeled vertex, inadmissible pushes within pseudoarcs do not affect termination.

In order to accomplish this, we will consider a generalized version of the original push-relabel algorithm of Goldberg and Tarjan [9]. We use the potential function $\Phi_g$, given below, to show that so long as flow is shifted to a lower-labeled vertex, the potential function behaves exactly the same as in the original push-relabel algorithm.

Let $\Phi_g$ be a potential function defined as follows:

$$\Phi_g = \sum_{u \in V : e(u) > 0} d(u)$$

The following lemma shows that the behavior is indeed what we claim; namely, that relaxing the admissibility constraint to $d(u) > d(v)$ still results in correct behavior of the potential function.

**Lemma 5.1.** If we relax the definition of admissibility to $d(u) > d(v)$, then the generic push-relabel algorithm will terminate.

**Proof.** We first examine the case where the potential function will increase. $\Phi_g$ increases when there is a relabeling or a saturating push. By Lemma 4.15, we see that there are $6n^2$ relabelings. Similarly, we can state that there will be $6mn$ saturating pushes. Therefore, $\Phi_g$ will increase by $6n^2m$.

Now we consider the case where $\Phi_g$ decreases. Clearly, the potential decreases by at least 1 when a push along an original arc is made. Now, consider a simple path $P = \langle u, v_1, v_2, \ldots, v_k, w \rangle$, where $d(u) > d(w)$. We will consider a push along the entirety of $P$. We see that $\Phi_g$ will decrease by at least 1 as well, since flow is still shifted to a vertex with a lower distance label. So long as we enforce $d(u) > d(w)$, the termination of the generic push-relabel algorithm still holds under path pushes. Moreover, we will terminate with $O(n^2m)$ nonsaturating pushes; this shows that the same running-time bounds apply to this stronger form of the generic push-relabel algorithm as the original in [9], and that the algorithm terminates under path pushes.

It is important to note that the previous lemma demonstrates that pseudoarc pushes cause the potential function from the original Goldberg-Tarjan push-relabel algorithm behaves the same if path pushes are permitted, whereas Lemma 4.10 simply shows that the discharge procedure is correct.

We have shown a stronger form of the generic push-relabel algorithm of Goldberg and Tarjan. So long as flow is moved to a lower-labeled vertex, $\Phi_g$ behaves as it does in the generic push-relabel algorithm. This gives us the termination of a $\Delta$-scaling phase:

**Corollary 5.1.** If pushes along directed paths are allowed, each $\Delta$-scaling phase will still terminate.

This proof relies on the assumption that we bound the relabel operations at $O(n^2)$ per phase. A difficulty that we resolve in Section 4.4 is due to the admissibility criterion; since $d(u) > d(v)$,
the bound on $2n - 1$ for vertex labels from [9] no longer applies. We show in the next section that by imposing several constraints on relabeling operations, we can bound the distance labels at $4n - 1$ for each $u \in V$, and at $4n^2 = O(n^2)$ across the vertex set.

6 Analysis of the Algorithm

6.1 Bound on Saturating & Nonsaturating Pushes

Lemma 6.1. There are $O(mn)$ saturating pushes across all scaling phases.

Proof. Proof is immediate from Lemma 2.4.

Lemma 6.2. There are $16cn$ large nonsaturating pushes per scaling phase, and $O(mn)$ across all scaling phases.

Proof. Consider the potential function

$$
\Phi = \sum_{u \in V} \frac{[dh(u) + dl(u)] \cdot e(u)}{\Delta}.
$$

We will first consider $\Phi$ at the beginning of a scaling phase. Since $dh(u) < 2n$, $dl(u) = 0$ at the start of a scaling phase, and $e(u) \leq \Delta$, for all $u \in V$, we see that $\Phi_{\text{init}} = 2cn$.

Next, we see that $\Phi_{\text{max}} = 8cn$. This is immediate from Lemma 4.15, since there is an increase of at most $6n$ in $dh + dl$.

When a nonsaturating (or saturating) push occurs, $dh(u) + dl(u) > dh(v) + dl(u)$ must hold for the push to have been made, the overall cost of distance labels in $\Phi$ will decrease. Since we can guarantee that a large nonsaturating push will send at least $\delta \geq \Delta/2$ (Lemma 4.7), $\Phi$ will decrease by $1/2$. Therefore, there will be $16cn$ large nonsaturating pushes per scaling phase.

Extending this result across all scaling phases, we have

$$
\sum_{i=1}^{K} 12cn = \sum_{i=1}^{K} O(cn) = O(n) \cdot (c_1 + c_2 + \cdots + c_K)
$$

$$
= O(n) \cdot O(c_1 + c_2 + \cdots + c_K) = O(mn).
$$

The final statement is given to us by Theorem 4.1. Clearly, we have $O(mn)$ large nonsaturating pushes across all scaling phases, and the proof holds.

We must finally bound the number of small nonsaturating pushes. Our maintenance of the list $\text{nonsat}_u$ ensures that we only permit a single small nonsaturating push per value of $dl$ in discharge. Moreover, once $dl(u) = 4n - 1$, for some $u \in V_A$, we permit it to make as many nonsaturating pushes as are necessary; we will see that this quantity is $2n$ in the next lemma. This lemma makes use of the fact that the maximum in-degree of a vertex is $k = O(1)$. If general networks were permitted, the overall cost of this procedure may be as large as $O(mn^2)$ across all scaling phases; however, it is efficient in the bounded-degree case.

Lemma 6.3. There are at most $4n - 1$ low-capacity nonsaturating pushes while $dl(u) < 4n - 1$, and at most $2n$ such pushes once $dl(u) = 4n - 1$.

Proof. We enforce the condition that there is a single low-capacity nonsaturating push for each value of $dl$; therefore, it is immediate that when $dl(u) < 4n - 1$, there are at most $4n$ low-capacity nonsaturating pushes. Finally, we must consider the quantity of such pushes when $dl(u) = 4n - 1$; there are at most $2n$ distinct values $dh$ can attain that could cause some
d(u) > d(v). Since \( \forall u \in V_A, \deg[u] = O(1) \), there will be 2n such pushes per vertex. Therefore, there are at most 2n low-capacity nonsaturating pushes once \( d(\ell(u)) = 4n - 1 \), and 6n low-capacity nonsaturating pushes overall.

By Lemma 4.11 we know that there are \( O(m) \) active vertices across all scaling phases. Summing across all scaling phases gives us that there are \( 6mn = O(mn) \) low-capacity nonsaturating pushes.

**Theorem 6.1.** There are \( O(mn) \) nonsaturating pushes across all \( \Delta \)-scaling phases.

### 6.2 Time to Create Compact Networks & Transform the Compact Preflow

In Appendix B we show how we can both construct the compact network and transform the compact preflow into a residual preflow in \( O(m \log n) \) time using the dynamic trees data structure. We restate a theorem proven later that summarizes these bounds.

**Theorem 6.2 (Theorem B.1).** Constructing abundant and small pseudoarcs takes \( O(m \log n) \) time per scaling phase by calling create-all-pseudoarcs. It takes \( O(m \log n) \) time to transform the compact preflow to the residual preflow by calling restore-all-flows.

### 7 A Strongly-Polynomial Variant

In this section, we define our final algorithm, \( \text{max-flow-1} \), and show how this yields a strongly-polynomial running time. We see that by our improvement property, there are clearly \( O(\log U) \) scaling phases. We accomplish this in Lemma 7.1 by bounding the number of scaling phases at \( O(m^{1/2}) \).

#### Algorithm 3. The final algorithm to find a max-flow on a bounded-degree network \( G \).

**Procedure** \( \text{max-flow-1}(G, \Delta) \)

01. Construct the compact network \( G_C \);
02. for all \( u \in V_C \) do initialize \( \text{edge-list}[u] \);
03. while there is a vertex \( u \in V_C \) from which flow can be discharged do
04. Select \( u := \min\{w \in V_A | d(w)\} \)
05. Iteratively call \( \text{discharge} \) on \( u \) and
06. select the next vertex \( u' := \min\{w \in V_A | d(w)\} \);
07. Run \( \text{restore-all-flows} \) and generate \( df \) with \( \text{global-relabel} \);
08. \( \Delta := \min\{\Delta/2, \max\{2^{\lceil \log e(u) \rceil} | u \in V\}\} \);

Correctness of our algorithm is immediate from the previous sections. We *initialize* the preflow \( f \), and then iteratively call \( \text{max-flow-1} \) until \( \Delta = 0 \). We will now show that the algorithm terminates in \( O(m^{1/2}) \) phases.

**Lemma 7.1.** By iteratively calling \( \text{max-flow-1} \), the algorithm terminates after \( O(m^{1/2}) \) scaling phases.

**Proof.** By Theorem 4.1 there are \( O(m) \) vertices in the compact network across all scaling phases. Therefore, there are at least \( O(m^{1/2}) \) scaling phases where \( c > m^{1/2} \).
Now, we must place an upper bound on the number of scaling phases where \( c \leq m^{1/2} \). Let \( \Delta_i \) be the excess dominator for the \( i \)th scaling phase. We will now bound the number of (1) arcs with \( \Delta \)-favorable and \( \Delta \)-large capacity, and (2) the number of active vertices. If we can show that there are \( O(m^{1/2}) \) arcs with \( \Delta \)-favorable and large capacity with respect to \( \Delta_i \), as well as \( \Delta \)-active vertices, we can conclude that both statements are valid for \( O(m^{1/2}) \) scaling phases.

(1) In an argument similar to Lemma 4.4, we can show that any \( \Delta \)-favorable or \( \Delta \)-large arc will become \( \Delta' \)-abundant in 4 scaling phases. Thus, we conclude that this statement is valid for \( O(m^{1/2}) \) scaling phases.

(2) Let \( u \in V_A \) be an active vertex. By Lemma 4.11 we “convert” each active vertex to a vertex in \( V_{SC} \) in the subsequent scaling phase. Therefore, the number of vertices in \( V_{SC} \) provides an upper bound on the number of vertices in \( V_A \), since we can do this “conversion” in \( 1 = O(1) \) scaling phases.

The following theorem is an immediate consequence of Theorem 6.1, Theorem 6.2, and Lemma 7.1.

**Theorem 7.1.** By iteratively calling max-flow-1, the algorithm will find a maximum flow in \( O(mn + m^{3/2}\log n) \) time. For bounded-degree networks, the algorithm runs in \( O(mn) \) time. By the preprocessing step in Appendix A, the algorithm also finds a max-flow in \( O(mn) \) time when \( m = O(n) \).

8 Conclusion

We note that by resolving certain issues we would be able to formulate an \( O(mn) \) time algorithm when \( m = O(n^{2-\varepsilon}) \). Namely, a new scheme for counting low-capacity nonsaturating pushes must be developed. We rely heavily on the fact that the degree of any vertex is \( O(1) \), which allows us to limit low-capacity nonsaturating pushes to one per each value of \( d_l \). We think that a technique described by Ahuja, et. al. \[3\] in which such pushes are “charged against” high-capacity nonsaturating pushes could be a way to resolve this.

If this issue is resolved, we can solve the max-flow problem on general networks in \( O(mn + m^{3/2}\log n) \) time, which implies an \( O(mn) \)-time algorithm for all but very dense networks. Further, the elimination of the \( O(\log n) \) factor would yield an \( O(mn) \) time algorithm for all edge densities; due to limitations with dynamic trees (see \[14\]), a new data structure would likely need to be used.

Acknowledgments

I would like to thank my advisor Janos Simon for his mentorship and guidance throughout this process. I would also like to thank Geraldine Brady and Baker Franke for introducing me to theoretical computer science, as well as for their general support. Finally, I would like to thank Robert Tarjan for suggesting the technique that led to the extension of the algorithm to sparse networks.

References

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network flows: theory, algorithms, and applications*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[2] R. K. Ahuja and J. B. Orlin, *A fast and simple algorithm for the maximum flow problem*, Operations Research 37 (1989), no. 5, pp. 748–759 (English).
A Transforming Sparse Networks into Bounded-Degree Networks

The main idea centers around converting a network $G = (V, A)$ where $m = O(n)$ to a bounded-degree network network $G'$ where $\max\{\deg[u]|u \in V(G')\} \leq k$, for some $k \in \mathbb{Z}_{>0}$. We can accomplish this by splitting vertices of “relatively high” degree into multiple vertices, and then connecting original arcs in $A$ back to the new vertices. If we can bound the increase in $|V'|$ and
\(|A'|\), then we simply can run this pre-processing on \(G\), and then use \(G'\) as an input into our original algorithm for bounded-degree networks. We will pick the number \(d := \lfloor m/n \rfloor + 3\) (the “average degree”) as our goal; our algorithm should produce \(G'\) such that \(d\) is the maximum in/out-degree.

Unless otherwise specified, we call a graph where \(\max\{\deg[u] | u \in V\} \leq k\) a \(k\)-bounded degree graph.

### A.1 The Preprocessing Algorithm

**Algorithm 4.** An algorithm to convert a sparse network into a bounded-degree network.

**Input:** A flow network \(G\).

**Output:** A new flow network \(G'\) such that \(\forall u \in V(G'), \max\{\deg[u]\} \leq k\), for some \(k \in \mathbb{Z}_{>0}\).

**Step 1.** Let \(d := \lfloor m/n \rfloor + 3\), \(G' = \emptyset\).

**Step 2.** Let \(V_d := \{u \in V | \deg[u] > d\}\).

**Step 3.** For all \(u\) in \(V_d\) do:

- a. Find \(k_u\) such that \(\deg[u] \leq k_ud\).
- b. Split \(u\) into \(k_u\) vertices \(u_1 \ldots u_k\), and initialize a binary tree \(T_u\) rooted at \(u_1\).
- c. Connect all arcs originally incident to \(u\) in \(G\) to \(u_1 \ldots u_k\) such that no degree exceeds \(d\).
- d. Connect \(u_2 \ldots u_k\) in the binary tree \(T_u\) with undirected infinite capacity arcs.

**Step 4.** Return \(G' = (V(G) \cup \bigcup_{u \in V_d} V(T_u), A(G) \cup \bigcup_{u \in V_d} A(T_u))\).

### A.2 Running Time & Correctness

We will now state and prove several lemmas regarding the running time and correctness of the procedure. Moreover, we will show that \(|V(G')|\) and \(|A(G')|\) are within a constant factor of \(|V(G)|\) and \(|A(G)|\).

**Lemma A.1.** \(G'\) is a \(d\)-bounded degree network.

**Proof.** Immediate from Step 3c. \(\square\)

For sake of clarity in the following proofs, let \(k = \max\{k_u | u \in V_d\}\). We also note that \(k = O(1)\).

**Lemma A.2.** \(|V(G')}| = O(|V(G)|) = O(n).

**Proof.** For each \(u \in V_d\), we find \(k_u\) such that \(\deg[u] \leq k_ud \leq kd\) and add at most \(k\) vertices to \(G'\). Thus, the total number of vertices in \(G'\) is

\[
|V(G')| = |V(G)| + \sum_{u \in V_d} k \\
\leq |V(G)| + \sum_{i=1}^{n} k = \sum_{i=1}^{n} O(1) = O(n).
\]

Thus \(|V(G')}| \leq O(n)\), which completes our proof. \(\square\)

**Lemma A.3.** \(|A(G')}| = O(|A(G)|) = O(n)\).
Proof. By Step 3d of the algorithm, we will add $k_u - 1 \leq k - 1$ new arcs for the binary tree $T_u$ that is created. This will be done for each $u \in V_d$. Thus,

$$|A(G')| = |A(G)| + \sum_{u \in V_d} k_u - 1 \leq k - 1$$

$$= |A(G)| + \sum_{i=1}^{n} O(1) = O(n) + |A(G)|.$$

Since $m = O(n)$, we have that $|A(G)| = O(n) \implies |A(G')| \leq O(n)$, which concludes our proof.

The algorithm will be run twice; once to ensure that for all $u \in V$, the in-degree does not exceed $d$, and once more for the out-degree.

Lemma A.4. The preprocessing algorithm terminates in $O(n \log m/n)$ steps.

Proof. In the worst case, $|V_d| = n$ (i.e. every vertex has degree greater than $d$). Finding $k$ such that $\deg[u] \leq kd$ will take $\log d = \log m/n$ steps if we use repeated squaring. Finally, connecting original arcs and creating new arcs in $T_u$ both take $O(k) = O(1)$ time. Thus, the overall cost of the pre-processing on $G$ will be $O(n \log m/n)$.

B Manipulating Pseudoarcs with Dynamic Trees

When manipulating $\Delta$-abundant and $\Delta$-small pseudoarcs, we use the dynamic trees data structure of Sleator and Tarjan [16]. In this section, we will list the operations supported by he data structure, and detail several procedures for manipulating preflows in pseudoarcs.

B.1 Tree Operations

The following operations can be carried out in $O(k \log n)$ amortized time, over a sequence of $k$ tree operations. We will see that $k = m$, and all dynamic tree operations take at most $O(m \log n)$ time per scaling phase.

(1) initialize - Creates an empty dynamic tree in $O(1)$ time.

(2) link($u, v$) - Merges two trees containing $u$ and $v$ (assuming $\text{root}(u) \neq \text{root}(v)$). We let $\pi_u := v$. The root of the new tree is $\text{root}(v)$. Let $\text{val}(u) := r(v, \pi_u)$.

(3) cut($u$) - “Breaks” the tree containing $u$ by deleting the arc $(u, \pi_u) \in A_f$. $u$ becomes the root of the new tree, and $r(u, \pi_u) := \text{val}(u)$.

(4) add-val($u, \delta$) - Adds $\delta$ units of flow to each arc on $\text{path}(u)$.

(5) find-min($u$) - Returns $u$ such that $\min\{\text{val}(u) | u \in \text{path}(u)\}$.

B.2 Algorithm for Constructing Pseudoarcs

The dynamic trees data structure is capable of efficiently carrying out a sequence of flow operations. Each tree is rooted at a vertex; for a tree containing vertex $u$, the root is denoted as $\text{root}(u)$. The parent of a vertex $u$ is denoted as $\pi_u$. Let $\text{path}(u)$ denote the simple, directed path $\langle u, v_1, v_2, \ldots, v_k, \text{root}(u) \rangle$. Let $\text{val}(u)$ denote the residual capacity of the arc $(u, \pi_u) \in A_f$. 

Procedure feasible-path($u, \rho$)
A procedure for finding a feasible path in the input graph.

\[ v := \text{root}(u); \]

\[ \textbf{while } v \notin V_C \setminus \{u\} \textbf{ do} \]

\[ \text{Choose } (v, w) \text{ such that } r(v, w) > \rho; \]

\[ \text{link}(v, w); \]

\[ \text{enqueue}(Q, \langle \text{link}, (v, w) \rangle); \]

\[ v := \text{root}(w); \]

\[ \textbf{return} \text{ feasible path } P; \]

When we wish to create abundant pseudoarcs, we select \( \rho = \Delta \). Similarly, when we wish to create \( \Delta \)-small pseudoarcs, we select \( \rho = 0 \).

\textbf{Procedure} transfer-capacity\((u, P)\)

\[ \textbf{while} \ d(v) = \Delta \textbf{ do} \]

\[ \text{link}(v, w); \]

\[ \text{enqueue}(Q, \langle \text{link}, (v, w) \rangle); \]

\[ v := \text{root}(w); \]

\[ \textbf{return} \text{ set } P \text{ of pseudoarcs;} \]

\textbf{Procedure} cut-all-saturated\((u)\)

\[ \text{cut}(u); \]

\[ \text{enqueue}(Q, \langle \text{cut}, u \rangle); \]

\[ w := \text{find-min}(u); \]

\[ \text{val}(w); \]

\[ \text{while } \delta = 0 \textbf{ do} \]

\[ \textbf{return} \text{ set } P \text{ of pseudoarcs;} \]

As a result of \textit{create-all-pseudoarcs}, a pseudoarc constructed from \( G_f \) could end up with capacity greater than \( \Delta \) if there exist many parallel pseudoarcs between \( u, w \in V_C \). In that case, we will refer to it as \( \Delta \)-abundant.

\textbf{B.3 Transforming the Compact Preflow}

All that is left to do now is transform the modified preflow on pseudoarcs in the compact network to the preflow in the residual network. We will accomplish this by sequentially recreating the
forest of dynamic trees used to create the pseudoarcs. We will iteratively execute operations from \( Q \) until there are none left. Doing so will allow us to recreate flows in \( O(k \log n) \) time, over a sequence of \( k \) operations.

**Procedure** \( \text{restore-all-flows}(f) \)

01. //A procedure to transform flows in pseudoarcs to flows in residual arcs. //
02. Initialize a new residual preflow \( f' \);
03. Create an empty dynamic tree;
04. while \( Q \neq \emptyset \) do
05. \( \phi = \text{dequeue}(Q) \);
06. Execute operation \( \phi \) from the queue;
07. if the min-path capacity \( \delta > 0 \) then
08. // Update the residual preflow \( f' \) //
09. \( \text{add-val}(v, \delta) \);
10. \( f'[v, \text{root}(v)] := f'[v, \text{root}(v)] - \delta \);

We conclude this section with a theorem regarding the running time of the procedures described.

**Theorem B.1.** Constructing abundant and small pseudoarcs takes \( O(m \log n) \) time per scaling phase by calling \( \text{create-all-pseudoarcs} \). It takes \( O(m \log n) \) time to transform the compact preflow to the residual preflow by calling \( \text{restore-all-flows} \).

**Proof.** Each operation \( \phi \in Q \) takes \( O(\log n) \) amortized time over a sequence of \( k \) operations. Since we can modify at most \( O(m) \) arcs when constructing pseudoarcs, we see that \( k = m \) and the overall cost is \( O(m \log n) \).

\[\square\]

**C Efficiently Deleting Active Vertices**

Finally, we must handle deletions of active vertices \( u \in V_A \) when constructing abundant pseudoarcs. We can accomplish this in the abundance graph by preprocessing the graph and splitting each vertex \( u \in V \) into \( u_{in} \) and \( u_{out} \). All the in-arcs of \( u \) will be connected to \( u_{in} \), and similarly all out-arcs to \( u_{out} \). We then add an arc of infinite capacity between \( u_{in} \) and \( u_{out} \). When we wish to delete some active vertex \( u' \in V_A \), we can simply delete the arc \((u'_{in}, u'_{out})\).