A $O(nm\log(U/n))$ Time Maximum Flow Algorithm

Antonio Sedeño-Noda, Carlos González-Martín

Departamento de Estadística, Investigación Operativa y Computación,
Universidad de La Laguna, 38271—La Laguna, Tenerife, Spain

Received July 1998; revised July 1999; accepted 24 February 2000

Abstract: In this paper, we present an $O(nm\log(U/n))$ time maximum flow algorithm. If $U = O(n)$ then this algorithm runs in $O(nm)$ time for all values of m and n. This gives the best available running time to solve maximum flow problems satisfying $U = O(n)$. Furthermore, for unit capacity networks the algorithm runs in $O(n^{2/3}m)$ time. It is a two-phase capacity scaling algorithm that is easy to implement and does not use complex data structures. © 2000 John Wiley & Sons, Inc. Naval Research Logistics 47: 511–520, 2000

1. INTRODUCTION

The maximum flow problem is one of the network optimization problem most intensely studied. The maximum flow problem has numerous applications and some of them are shown in Ahuja, Magnanti, and Orlin [2]. This problem was introduced by Fulkerson and Dantzig in 1955 and solved for the first time by Ford and Fulkerson [11] through the known *Augmenting Path Algorithm*. Since this date, numerous authors have designed different algorithms that, using new and important ideas, solve this problem improving the associated computational complexity.

The most important methods are attributed to Dinic [8], Edmonds and Karp [9], Karzanov [15], Malhotra, Kumar, and Maheshwari [16], Gabow [12], Sleator and Tarjan [18], Goldberg [13], Goldberg and Tarjan [14], Cheriyan and Maheshwari [6], Ahuja and Orlin [1, 3, 4], and Cheriyan, Hagerup, and Melhorn [7]. The theoretical complexities of these algorithms are shown in Table 1. Among the studies of this problem we can emphasize those of Nicoloso and Simeone [17], Fernández-Baca and Martel [10], and the excellent text of Ahuja, Magnanti, and Orlin [5].

In this work, we present an algorithm to solve the maximum flow problem in an $O(nm\log(U/n))$ time. For the last three decades, researchers have not found an algorithm of $O(nm)$ time for all combinations of n and m. This is true even when $U = \Omega(n)$ (Ahuja, Orlin, and Tarjan [3]). In particular, when $m = \Omega(n^2/\log n)$, the algorithm of Cheriyan et al. [7] runs in $O(nm)$ time, independently of the value of U. Our main result is that under the assumption of $U = O(n)$, our algorithm runs in $O(nm)$ time for any combination of n and m. No other algorithm realizes this time for any values of n and m. If $U = O(n^K)$ for some $K > 1$, then the algorithm runs in $O(nm\log n)$ time. This complexity equals the complexity of the algorithm of Sleator and Tarjan [18], without making use of complex data structures. Furthermore, for unit capacity networks the algorithm runs in $O(n^{2/3}m)$ time. This algorithm is based on the two-phase algorithm for

Correspondence to: C. González-Martín

Table 1. Worst-case complexities of maximum flow algorithms.

No.	Authors (year)	Theoretical complexity
1	Ford and Fulkerson (1956)	$O\ (nmU)$
2	Dinic (1970)	$O\ (n^2m)$
3	Edmonds and Karp (1972)	$O\ (nm^2)$
4	Karzanov (1974)	$O\ (n^3)$
5	Maholtra, Kumar, and Maheshwari (1978)	$O\ (n^3)$
6	Sleator and Tarjan (1983)	$O\ (nm \log n)$
7	Gabow (1985)	$O\ (nm \log U)$
8	Goldberg (1985)	$O\ (n^3)$
9	Goldberg and Tarjan (1986)	$O\ (nm \log(n^2/m))$
10	Ahuja, Orlin, and Tarjan (1988)	$O\ (nm + n^2\sqrt{\log U})$
		$O\ (nm + n^2 \log U/ \log\log U)$
		$O\ (nm \log(n\sqrt{\log U}/m + 2))$
11	Cheriyan and Maheshwari (1989)	$O\ (n^2m^{1/2})$
12	Ahuja and Orlin (1989)	$O\ (nm + n^2 \log U)$
13	Ahuja and Orlin (1991)	$O\ (nm \log U)$
		$O\ (n^2m)$
14	Cheriyan, Hagerup, and Melhorn (1996)	$O\ (n^3/ \log n)$
15	Sedeño-Noda and González-Martin (this paper)	$O\ (nm \log(U/n))$

unit capacity networks given by Ahuja and Orlin [4], introducing a scaling in the arc capacities. The behavior in the practice of this algorithm can be attractive, because of the good behavior in the practice of the two-phase algorithm and because it is possible to incorporate strategies to improve it.

After this introduction, in the second section, we give some necessary definitions in order to understand the algorithm, as in the mathematical formalization of the maximum flow problem. Also, in this section we include an explanation of the two procedures on which the two-phase algorithm is based. In the third section, we present the two-phase capacity scaling algorithm. In the fourth section, we analyze the theoretical complexity of the algorithm. Finally, in the fifth section, we make some commentaries about two strategies that can be introduced into the algorithm to improve the practical performance and some commentaries about the comparison between our algorithm and another.

2. NOTATION, DEFINITIONS AND MATHEMATICAL FORMALIZATION

Given a directed network, $G = (V, A)$, let V be a set of n nodes, and let A be the set of m arcs. We distinguish two special nodes in G: a source node s and a sink node t. Given any node i, we define $\mathrm{Pred}(i) = \{j \in V|(j,i) \in A\}$ and $\mathrm{Succ}(i) = \{j \in V|(i,j) \in A\}$. Each arc $(i,j) \in A$ has the following values associated with it: u_{ij}, the capacity that denotes the maximum amount that can flow on the arc (i,j), and l_{ij}, a lower bound that denotes the minimum amount that must flow on the arc. Without loss of generality we can assume that l_{ij} is equal to zero for each arc (i,j). We also define the *arc adjacency list* $A(i) = \{(i,j) \in A : j \in \mathrm{Succ}(i)\}$ for all $i \in V$. The maximum value of the set $\{u_{ij}|(i,j) \in A\}$ will be denoted by U. A *flow* in G is a function

$x\colon A \rightarrow R^{+} \cup \{0\}$ that satisfies

$$\sum_{j\in \mathrm{Succ}(i)} x_{ij} - \sum_{j\in \mathrm{Pred}(i)} x_{ji} = \begin{cases} f & \text{if } i = s, \\ 0 & i \in V - \{s,t\}, \\ -f & \text{if } i = t, \end{cases} \tag{1}$$

$$0 \leq x_{ij} \leq u_{ij}, \qquad (i,j) \in A \tag{2}$$

for some $f \geq 0$. The maximum flow problem consists in finding a flow x such that f is maximal.

A *cut* is a partition of the node set V in two subsets S and $\bar{S} = V - S$. A cut defines a subset of arcs consisting of those arcs that have one endpoint in S and another endpoint in $\bar{S}$. So, we denote this set of arcs by $[S, \bar{S}]$. An *s-t cut* is a cut such that $s \in S$ and $t \in \bar{S}$. The *capacity of an s-t cut* is equal to the sum of the capacities of the arcs that leave set S and arrive at set $\bar{S}$. We denote this capacity by $u[S, \bar{S}] = \sum_{(i,j)\in[S,\bar{S}]} u_{ij}$.

Ford and Fulkerson [11] prove that the maximum amount of flow f that can be sent from the source node s to the sink node t equals the minimum capacity of all s-t cut of the network (the *maximum-flow minimum-cut theorem*).

Give a flow x, the *residual capacity* r_{ij} of any arc $(i,j) \in A$ is the maximum additional flow that can be sent from node i to node j using the arcs (i,j) and (j,i); that is, $r_{ij} = u_{ij} - x_{ij} + x_{ji}$. We refer to the network $R(x)$ consisting of the arcs with positive residual capacities as the *residual network*. The residual network $R(x)$ has the same set of nodes as V but for each arc $(i,j) \in A$ we add two arcs (i,j) and (j,i) with residual capacities r_{ij} and r_{ji} respectively. An *augmenting path* in $R(x)$ is a directed path from node s to node t in $R(x)$. We define the *residual capacity* of an augmenting path as the minimum residual capacity of any arc in the path. The following result given in [11] is well known: A flow x is maximal if in the network $R(x)$ there is no augmenting path. This termination criteria is used to obtain the *Augmenting Path Algorithm*. This algorithm proceeds by identifying augmenting paths in $R(x)$ and augmenting flows on these paths. This algorithm runs in $O(nmU)$ time, where nU is an upper bound of the capacity of the minimum s-t cut. So, in the worst-case, the algorithm identifies nU augmenting paths and sends along each one of them, at least one unit of flow.

Now we consider the distance labels introduced by Goldberg [13]. Let d be a *distance function* $d : V \rightarrow Z^{+} \cup \{0\}$ with respect to the residual capacities r_{ij}. A distance function is *valid* if it satisfies the following two conditions:

(a) $\quad d(t) = 0,$

(b) $\quad d(i) \leq d(j) + 1, \qquad \forall (i,j) \in A, r_{ij} > 0.$

We refer to $d(i)$ as the *distance label* of node i. By induction, it can be proved that $d(i)$ is a lower bound on the length of the shortest path from node i to node t in $R(x)$, where the length of the path is equal to the number of arcs on the path. If, for each node i, $d(i)$ equals the length of the shortest path from node i to node t in $R(x)$, then we say that the distance labels are *exact*. If $d(s) \geq n$, then there is no augmenting path in $R(x)$.

An arc (i,j) in the residual network is called *admissible* if it satisfies the following condition: $d(i) = d(j) + 1$. A path from node s to node t consisting entirely of admissible arcs is called an *admissible path*. An admissible path is a shortest augmenting path from node s to node t.

2. 1. Shortest Augmenting Path Algorithm

Edmonds and Karp [9] proved that if the algorithm of Ford and Fulkerson [11] proceeds by identifying a shortest augmenting path in $R(x)$ in each iteration, then the algorithm solves the maximum flow problem within at most $mn/2$ augmentations. Hence the worst-case complexity of the algorithm is strongly polynomial. The method used to find the shortest augmenting path is a breadth-first search in the residual network. This method requires $O(m)$ time. Therefore, the algorithm runs in $O(m^2n)$ time.

A shortest augmenting path algorithm that uses the distance labels is introduced by Ahuja and Orlin [4] and runs in $O(mn^2)$ time. The mentioned algorithm has the following scheme:

```
Algorithm Shortest_Augmenting_Path;
  begin
    $X := 0; d(t) := 0; i := s$;
    Obtain the exact distance labels by a reverse breadth-first
    search of the residual network starting from the sink node $t$;
    While $d(s) < n$ do
        if $i$ has an admissible arc then
          begin
              Advance($i$);
            if $i = t$ then
              begin
                Augment;
                $i := s$
              end
          end
        else
          Retreat($i$)
  end

  Procedure Advance($i$);
    begin
      Let $(i,j)$ be an admissible arc emanating from $i$;
      pred($j$) := $i$; $i := j$
    end

  Procedure Retreat($i$);
    begin
      $d(i) := \min\{d(j)+1 : (i,j) \in A(i)$ and $r_{ij} > 0\}$;
      if $i \neq s$ then $i :=$ pred($i$)
    end

  Procedure Augment;
    begin
      Uses the predecessor indices to identify the augmenting
      path $P$ from the source to the sink;
      $\delta := \min\{r_{ij} : (i,j) \in P\}$;

      Augment $\delta$ units of flow along $P$
    end
```

The shortest augmenting path algorithm proceeds by augmenting flows along admissible paths. The algorithm first performs a reverse breadth-first search in $R(x)$ starting from the sink node t to compute the exact distance labels $d(i)$. The algorithm keeps a predecessor indices with each node in the residual network so that $\text{pred}(i)$ stores the previous node to node i in the partial admissible path from s to i. The algorithm performs iteratively *Advance* or *Retreat* steps at the last node on the partial admissible path. If the current node i emanates an admissible arc (i, j), then it performs an *Advance* step and adds this arc to the current partial path. If an *Advance* step does not occur, then it performs a *Retreat* step that increases the distance label of the node i, which makes the arc $(\text{pred}(i), i)$ inadmissible and, therefore, backtracks on this arc in the partial admissible path. If in this process the sink node is reached, then the algorithm performs an *Augment* step. The process is repeated until the distance label of the source node becomes greater than or equal to n. At this moment, the flow is a maximum flow.

We now comment on the data structure that is used in the algorithm to identify the admissible arcs emanating from a node. For each node i, the algorithm maintains an arc that belongs to $A(i)$, that is, the current candidate to be examined for admissibility. For each node i, this arc is called the *current arc* of the node. Initially, the current arc of node i is the first arc in $A(i)$. Whenever the algorithm reaches node i, the current arc of this node is tested. If the current arc is inadmissible, then the current arc of the node i becomes the next arc in list $A(i)$. The process is repeated until an admissible arc is found or the end of list $A(i)$ is reached. In case of the latter, it means that in $A(i)$ there is no admissible arc, the algorithm performs a *Retreat* step on node i, and the current arc of node i is again the first arc in $A(i)$.

The discussions about the theoretical complexity of this algorithm will be shown further on, since our algorithm uses this algorithm as subroutine.

3. TWO-PHASE CAPACITY SCALING ALGORITHM

In this section, we propose a capacity scaling algorithm that uses the shortest augmenting path previously introduced in the first phase and Ford and Fulkerson algorithm in the second phase. So, in this way, we obtain an algorithm that runs in $O(nm \log(U/n))$ time.

The first capacity scaling algorithm was proposed by Gabow in [12] and it runs in $O(nm \log U)$ time. Ahuja and Orlin in [4] also propose a capacity scaling algorithm with the same worst-case complexity.

A parameter Δ to define the scaling in the capacities is used in the algorithm. We refer to the network that contains only those arcs whose residual capacity is at least Δ as the Δ-*residual network*, $R(\Delta)$. In this way, all augmenting paths in $R(\Delta)$ have a residual capacity greater than or equal to Δ. A flow is a Δ-*optimal* flow if in the network $R(x)$ there is no augmenting path of residual capacity of at least Δ.

The algorithm performs a number of scaling phases. Each phase starts with a fixed Δ and obtains a Δ-*optimal* flow. We refer to a phase with a specific value of Δ as the Δ-*scaling phase*. When a Δ-scaling phase finishes, the next phase starts with $\Delta = \Delta/2$. In the first phase, $\Delta = 2^{\lfloor \log U \rfloor + 1}$, that is, Δ equals the least integer that is a power of two and that is greater than or equal to U. In the last phase, $\Delta = 1$. In this phase $R(\Delta) = R(x)$ and, therefore, at the end of the algorithm, a maximum flow is obtained. In this way, the algorithm performs $\lfloor \log U \rfloor + 1$ scaling phases.

In each Δ-scaling phase, the algorithm performs two phases. In the first phase, the shortest augmenting path algorithm is applied with the following stop rule: The first phase finishes when the distance label of source node is greater than or equal to $K(\Delta)$. The value of $K(\Delta)$ is fixed in each Δ-scaling phase and its value will be obtained later. In this phase, we enforce each augmentation to send exactly Δ flow units. Therefore, the shortest augmenting path algorithm does not use the *Augment* procedure. So, each augmentation requires $O(1)$ time. To do this, when the algorithm performs an *Advance* step through the current admissible arc (i, j), the residual capacities of the arcs (i, j) and (j, i) are updated in the following way: $r_{ij} = r_{ij} - \Delta$ and $r_{ji} = r_{ji} + \Delta$. If the algorithm performs a *Retreat* step, then the residual capacities of the arcs $(\text{pred}(i), i)$ and $(i, \text{pred}(i))$ are updated in the following way: $r_{\text{pred}(i)i} = r_{\text{pred}(i)i} + \Delta$ and $r_{i\,\text{pred}(i)} = r_{i\,\text{pred}(i)} - \Delta$. In this way, if the sink node is reached, the algorithm has already augmented Δ flow units through the current admissible path from the source node to the sink node.

In the second phase, we use the algorithm of Ford and Fulkerson [11]. The augmenting paths are identified by a depth-first search in $R(\Delta)$ starting at the source node s. In this case, each augmentation sends the residual capacity of the augmenting path. As we have already mentioned, the residual capacity of each augmenting path is greater than or equal to Δ (and less than 2Δ). At the end of the second phase, an Δ-optimal flow is obtained, because in $R(\Delta)$ there is no augmenting path. In addition, the residual network $R(x)$ does not contain any augmenting path with residual capacity of at least Δ.

The method that we propose requires some modifications in the shortest path algorithm. We call an arc (i, j) as Δ-*admissible* if $d(i) = d(j) + 1$ and $r_{ij} \geq \Delta$. In the first phase, the algorithm proceeds by identifying Δ-*admissible paths*. The *Retreat* procedure must be modified in such a way that the distance label of the node i becomes equal to $\min\{d(j) + 1 : (i, j) \in A(i)$ and $r_{ij} \geq \Delta\}$. Because the *Augment* procedure is not used, it is necessary to introduce the *SendFlow* procedure. This procedure updates the residual capacities of the current arc and the reverse current arc as we have mentioned above. Finally, the stop rule is now $d(s) \geq K(\Delta)$.

After all these commentaries, the shortest path algorithm with residual capacity of the augmenting paths equal to Δ is the following:

```
Algorithm  Shortest_Augmenting_Path($\Delta, K(\Delta)$);
  begin
    Let $X$ be the current flow vector; $d(t) := 0$;
    Obtain the exact distance labels by a reverse breadth-first
    search of the $\Delta$-residual network starting from the sink
    node $t$;
    $i := s$;
    While $d(s) < K(\Delta)$ do
        if $i$ has a $\Delta$-admissible arc then
           begin
              Advance($i$);
              SendFlow(pred($i$), $i$, $\Delta$);
              if $i = t$ then $i = s$
           end
        else
           begin
              if $i \neq s$ then  SendFlow(pred($i$), $i$, $-\Delta$);
              Retreat($i$);
           end
  end

Procedure  SendFlow($i, j, \Delta$);
    begin
      $r_{ij} = r_{ij} - \Delta; r_{ji} = r_{ji} + \Delta$
    end
```

In the second Phase of our algorithm it is only necessary to modify the depth-first search in such a way that only the arcs whose residual capacities are at least Δ are considered. The scheme of the two-phase capacity scaling algorithm is the following:

```
Algorithm Two-phase capacity scaling;
  begin
    $X := 0$;
    $\Delta := 2^{\lfloor \log U \rfloor + 1}$;
    While $\Delta > 1$ do
      begin
        Shortest_Augmenting_Path$(\Delta, K(\Delta))$; {first phase}
        While $R(\Delta)$ contains a path $P$ from node $s$ to node $t$ do
          begin {second phase}
            $\delta := \min\{r_{ij} : (i,j) \in P\}$;
            Augment $\delta$ units of flow along $P$
          end;
        $\Delta := \Delta/2$
      end
  end
```

4. COMPLEXITY OF THE TWO-PHASE CAPACITY SCALING ALGORITHM

The complexity of the algorithm depends on the value of $K(\Delta)$. We will obtain the value of $K(\Delta)$ which obtains the smallest running time of the algorithm. Given a Δ-scaling phase, we have already mentioned that our algorithm performs two phases. In the first phase, the shortest augmenting path algorithm runs until the distance label of the source node is greater than or equal to $K(\Delta)$. In the second phase, our algorithm identifies augmenting paths until a Δ-optimal flow is obtained. We are going to obtain the theoretical complexity of both phases as a function of the value of $K(\Delta)$ for a fixed Δ-scaling phase.

LEMMA 1: The first phase of the algorithm performs at most $O(K(\Delta)m)$ augmentations and runs in $O(K(\Delta)m)$ time for a fixed Δ-scaling phase.

PROOF: In the first phase, the modified shortest augmenting path algorithm is used until $d(s) \geq K(\Delta)$. In this way, the algorithm updates the distance label of any node at most $K(\Delta)$ times. That is why if the distance label of a node i is greater than or equal to $K(\Delta)$, this node never falls on the current admissible path $(d(i) < d(s) < K(\Delta))$. Consequently, the distance label of any selected node in a partial path must be less than $K(\Delta)$, and, as each *Retreat* step on node i increases, the distance label by at least 1 unit; then the maximum number of update of $d(i)$ is at most $K(\Delta)$. So the total number of *Retreat* steps is bounded by $O(K(\Delta)n)$ and the total effort spent in the *Retreat* operations is $O(K(\Delta)m)$.

Now, we bound the number of augmentations that the algorithm performs. For that, we call a push of flow on arc (i,j) a Δ-*saturating* push if the residual capacity of the arc becomes less than Δ after the push of flow on arc. In the first phase in a fixed Δ-scaling phase, the algorithm performs at most $K(\Delta)/2$ Δ-saturating pushes, because between two consecutive Δ-saturating pushes on an arc (i,j) both $d(i)$ and $d(j)$ must increase by at least 2 units and the greater value of distance label of a node is $K(\Delta)$. Therefore, the total number of Δ-saturating pushes is $O(K(\Delta)m)$. Since each augmentation Δ-saturates at least one arc, the number of augmentations is $O(K(\Delta)m)$. As each augmentation requires $O(1)$ time, the total effort for the augmentations operations is $O(K(\Delta)m)$. Both the number of *Advance* and *SendFlow* steps is bounded by the augmentation time plus the number of *retreat* steps, that is, $O(K(\Delta)m + K(\Delta)n)$. In summary, the running time of the first phase is $O(K(\Delta)m)$. $\square$

LEMMA 2: The second phase of the algorithm performs at most $4Un^2/(K(\Delta))^2\Delta$ augmentations and runs in $O(4Un^2m/(K(\Delta))^2\Delta)$ time for a fixed Δ-scaling phase.

PROOF: Let $f^{K(\Delta)}$ denote the flow value at the end of the first phase and f^* denote the Δ-optimal flow value. We are going to prove that $f^* - f^{K(\Delta)} \leq 4Un^2/(K(\Delta))^2$. To see this, let V_p be the set $V_p = \{i \in V : d(i) = p\}$. We refer to this set as the set of nodes in the *pth layer*. We consider that the sets $V_{K(\Delta)}, V_{K(\Delta)-1}, \ldots, V_1$ are all nonempty, because if one of the sets is empty then, there is no augmenting path from node s to node t in $R(\Delta)$, and, therefore the flow $f^{K(\Delta)}$ is Δ-optimal. Also, we must note that $|V_1| + |V_2| + \cdots + |V_{K(\Delta)}| \leq n - 1$, because the sink node does not belong to any of these sets. Now, we prove that the network $R(\Delta)$ contains at least two consecutive layers, each with at most $2n/K(\Delta)$ nodes. If this is not true, then every alternate layer contains $H > 2n/K(\Delta)$ nodes, and so the total number of nodes would be $H \cdot (K(\Delta)/2) > n$, leading to a contradiction. Then, we suppose that $|V_p| \leq 2n/K(\Delta)$ and $|V_{p-1}| \leq 2n/K(\Delta)$ for some two alternate layers. In this case, a possible s-t cut is defined by the set of arcs connecting the layer V_p with the layer V_{p-1}, because there are no arcs (i, j) in $R(\Delta)$ with $i \in V_p$ and $j \in V_z$ so that $p > z + 1$ (because the distance function is valid). The capacity of the arcs is bounded by U. Then, the capacity of this s-t cut is $|V_p||V_{p-1}|U \leq 4Un^2/(K(\Delta))^2$, and so, $f^* - f^{K(\Delta)} \leq 4Un^2/(K(\Delta))^2$.

Thus, in the worst case, in the second phase of the algorithm $4Un^2/(K(\Delta))^2$ flow units remain to be sent. But, each augmentation sends at least Δ flow units, that is, the second phase performs at most $4Un^2/(K(\Delta))^2\Delta$ augmentations. Furthermore, after $O(4Un^2/(K(\Delta))^2\Delta)$ augmentations, an Δ-optimal flow is obtained. Let us remember that to identify each augmenting path, in the second phase, a depth-first search is used. Therefore, the running time of the second phase is $O(4Un^2m/(K(\Delta))^2\Delta)$, because the depth-first search runs in $O(m)$ time. $\square$

In summary, in a fixed Δ-scaling phase, the first phase in the algorithm runs in $O(K(\Delta)m)$ time and the second phase runs in $O(4Un^2m/(K(\Delta))^2\Delta)$ time. That is to say, the total running time is $O(K(\Delta)m + 4Un^2m/(K(\Delta))^2\Delta)$. This function reaches a minimum value for $K(\Delta) = 2(Un^2/\Delta)^{1/3}$, and then, we obtain an algorithm that runs in $O((Un^2/\Delta)^{1/3}m)$ time for a fixed Δ-scaling phase. The following theorem gives the theoretical complexity of the two-phase capacity scaling algorithm.

THEOREM 1: The two-phase capacity scaling algorithm for $K(\Delta) = \min(n, 2(Un^2/\Delta)^{1/3})$ runs in $O(nm\log(U/n))$ time.

PROOF: The algorithm performs $q = \lfloor \log U \rfloor + 1$ scaling phases. Note that $U = 2^{q-1}$ and $\Delta = 2^{q-i}$ for each $i = 1, \ldots, q$. Then the theoretical complexity of the algorithm is given by the following sum:

$$\sum_{i=1}^{q} \left(\frac{2^{q-1}n^2}{2^{q-i}} \right)^{1/3} m = n^{2/3}m \sum_{i=1}^{q} (2^{1/3})^{i-1}.$$

But if $K(\Delta) > n$, then, the first phase in the algorithm terminates when $d(s) \geq n$ and finishes with an Δ-optimal flow, because there is no augmenting path. So, let p be the scaling phase such that $K(\Delta)$ becomes greater than or equal to n. Then, from this scaling phase upwards, each scaling phase requires $O(nm)$ time, because $K(\Delta)$ is always greater than n. Now, we are going to compute the value of p by solving the following equality:

$$K(\Delta) = 2\left(\frac{Un^2}{\Delta}\right)^{1/3} = 2\left(\frac{2^{q-1}n^2}{2^{q-p}}\right)^{1/3} = 2(2^{p-1}n^2)^{1/3} = n,$$

and we find that $p = \log n - 2$.

From the scaling phase $p+1$ to the scaling phase $q, K(\Delta)$ is greater than n. Then, the theoretical complexity of the algorithm is given by the following sum:

$$n^{2/3}m\sum_{i=1}^{p}(2^{1/3})^{i-1} + \sum_{p+1}^{q} mn$$

$$= n^{2/3}m\left(\frac{(2^{1/3})^p - 1}{2^{1/3}-1}\right) + mn(q-p) \leq n^{2/3}m(2^{1/3})^p + mn(\log U - \log n + 2)$$

As $(2^{1/3})^p \leq n^{1/3}$, we find that the theoretical complexity of the algorithm is $O(nm \log(U/n))$. $\square$

5. SUMMARY

In this paper we propose an $O(nm \log(U/n))$ maximum flow algorithm. Our algorithm runs in $O(nm)$ time for any combination of n and m provided that $U = O(n)$. This gives the best available running time to solve maximum flow problems satisfying $U = O(n)$. Furthermore, for unit capacity networks the algorithm runs in $O(n^{2/3}m)$ time. The algorithm is a two-phase capacity scaling algorithm that combines other known methods: generic augmenting path algorithm, shortest augmenting path algorithm, and capacity scaling algorithm. To obtain the running time of our algorithm, we modify the shortest augmenting path in such a way that each augmentation requires an effort of $O(1)$ time.

Furthermore, strategies can be incorporated in the algorithm to improve its practical behavior. One of them is attributed to Ahuja and Orlin [4]. This strategy avoids performing a large number of *Retreat* steps after a maximum flow was obtained. The second strategy that we propose is related to the maximum amount of flow that can be sent in the second phase. In the second phase at most $4Un^2/K^2$ flow units are sent from source node to sink node. Therefore, if in the first phase an s-t cut such that the capacity of this is less than or equal to $4Un^2/K^2$ is detected, the first phase terminates, and the algorithm continues with the second phase.

We have carried out a limited computational study where we compare our algorithm with the capacity scaling algorithm attributed to Ahuja and Orlin [4]. We have observed that the capacity scaling algorithm is faster than our algorithm for this particular experiment. In addition, the capacity scaling algorithm performs fewer augmentations than our algorithm. This is due to the fact that in a Δ-scaling phase, the capacity scaling algorithm sends through each shortest augmenting path an amount of flow equal to the residual capacity of the path that is greater than or equal to Δ and strictly less than 2Δ. Instead, our algorithm always sends Δ flow units in the first phase. As a result, the number of augmentations of capacity scaling algorithm is less. We plan to perform an empirical experiment using several augmenting path algorithms to obtain any other classification.

ACKNOWLEDGMENTS

The authors wish to thank the Associate Editor and the anonymous referees for their comments. Their suggestions have resulted in an improved paper.

REFERENCES

[1] R.K. Ahuja and J.B. Orlin, A fast and simple algorithm for the maximum flow problem, Oper Res 37 (1989), 748–759.

[2] R.K. Ahuja, T.L. Magnanti, and J.B. Orlin, ''Network flows,'' Optimization, Handbooks in Operations Research and Management Science, G.L. Nemhauser, A.H.G. Rinnooy Kan and M.J. Todd (Editors), North Holland, Amsterdam, 1989, Vol. 1, pp. 211–369.

[3] R.K. Ahuja, J.B. Orlin, and R.E. Tarjan, Improved time bounds for the maximum flow problem, SIAM J Comput 18 (1989), 939–954.

[4] R.K. Ahuja and J.B. Orlin, Distance-directed augmenting path algorithms for maximum flow and parametric maximum flow problems, Nav Res Logistics 38 (1991), 413–430.

[5] R.K. Ahuja, T.L. Magnanti, and J.B. Orlin, Network flows: Theory, algorithms and applications, Prentice-Hall, Englewood Cliffs, NJ, 1993.

[6] J. Cheriyan and S.N. Maheshwari, Analysis of preflow push algorithms for maximum network flow, SIAM J Comput 18 (1989), 1057–1086.

[7] J. Cheriyan, T. Hagerup, and K. Mehlhorn, An $O(n^3/\log n)$-time maximum-flow algorithm, SIAM J Comput 25 (1996), 1144–1170.

[8] E.A. Dinic, Algorithms for solution of a problem of maximum flow in networks with power estimation, Sov Math Dokl 11 (1970), 1277–1280.

[9] J. Edmonds and R.M. Karp, Theoretical improvements in algorithmic efficiency of network flow problems, J Am Comput Mach 19 (1972), 248–264.

[10] D. Fernández-Baca and C.U. Martel, On the efficiency of maximum flow algorithms on networks with small integer capacities, Algorithmica 4 (1989), 173–189.

[11] L.R. Ford and D.R. Fulkerson, Maximal flow through a network, Can J Math 8 (1956), 399–404.

[12] H.N. Gabow, Scaling algorithms for network flow problems, J Comput Syst Sci 31 (1985), 148–168.

[13] A.V. Goldberg, A new approach to the maximum flow problem, Technical Report MIT/LCS/TM-291, Laboratory for Computer Science, MIT, Cambridge, MA, 1986.

[14] A.V. Goldberg and R.E. Tarjan, A new approach to the maximum flow problem, Proc. 18th ACM Symp. Theory Comput, 1986, pp. 136–146.

[15] A.V. Karzanov, Determining the maximal flow in a network by the method of preflows, Sov Math Dokl 15 (1974), 434–437.

[16] V.M. Malhotra, M.P. Kumar, and S.N. Maheshwari, An $O(|V|^3)$ algorithm for finding maximum flows in networks, Inform Process Lett. 7 (1978), 277–278.

[17] S. Nicoloso and B. Simeone, Classical and contemporary methods in network optimization, Part I: Network Flows, 1992.

[18] D.D. Sleator and R.E. Tarjan, A data structure for dynamic trees, J Comput Syst Sci 24 (1983), 362–391.