LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TM-291

A NEW MAX-FLOW ALGORITHM

Andrew V. Goldberg

November 1985

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

A New Max-Flow Algorithm

Andrew V. Goldberg[†*]

Laboratory for Computer Science
M.I.T.
Cambridge, MA 02139

ABSTRACT

All previously known max-flow algorithms worked by finding augmenting paths, either one path at a time (Ford and Fulkerson algorithm), or all shortest augmenting paths at once (by using the level network technique of Dinic). We introduce an alternative way of dealing with the problem. Our method is to push flow through the original network. The algorithm and its analysis are simple and intuitive, yet the algorithm does as well as any other network flow algorithm on dense graphs, achieving $O(n^3)$ running time.

The algorithm admits distributed and parallel implementations as well as a sequential implementation. The algorithm requires less storage then the only other parallel max-flow algorithm known (due to Shiloach and Vishkin), and its parallel running time is the same, $O(n^2 \log n)$. In fact, our algorithm uses constant amount of storage for every edge or vertex of the network, allowing an implementation under a more realistic distributed model.

Keywords:

network flow, combinatorial algorithm, distributed algorithm, parallel algorithm.

1. Historical Background

The problem of finding a maximum flow in a network with capacity constraints on edges arises in many applications in the fields of computer science and operations research. It is not surprising that efficient algorithms for the problem have received a great deal of attention.

[†]Supported by a Fannie and John Hertz Foundation Fellowship.
[*] Part of this research has been done while the author was at GTE Laboratories, Inc.

As early as 1956, Ford and Fulkerson [FF 1956] formulated an algorithm to solve the problem based on finding flow-augmenting paths. The complexity of the algorithm was not analyzed until 1969, when Edmonds and Karp [EK 1972] proved that if the shortest flow-augmenting path is used at each iteration of the algorithm, the time complexity is $O(nm^2)$ (throughout this paper, n denotes the number of vertices and m the number of edges in the network).

In 1970, Dinic introduced the level network method [Din 1970], which reduces the maximum flow problem to the problem of finding a *maximal* flow in the level network. Using this method, he constructed an $O(n^2m)$ max-flow algorithm. In 1974, Karsanov [Kars 1974] gave a new method of finding a maximal flow in a level network, which results in an $O(n^3)$ max-flow algorithm. This upper bound is still the best known for dense graphs. Karsanov's idea was to find maximal flow in a level network by pushing flow through it rather then by looking for augmenting paths one by one.

Several algorithms have been developed for sparse graphs; the best so far is the $(nm \log n)$ algorithm of Sleator and Tarjan [Sl 1980].

In 1982, Shiloach and Vishkin [SV 1982] investigated a distributed version of the problem. They constructed an $O(n^2 \log n)$ parallel time algorithm with $O(n^3)$ sequential implementation. We will refer to this algorithm as the S&V algorithm.

All network flow algorithms with complexity $O(n^3)$ (or better for sparse graphs) were based on Dinic's level network method; their analysis was quite involved, except for the $O(n^3)$ algorithm due to Malhotra, Kumar, and Maheshwari [MKM 1978].

The algorithm presented in this paper abandons Dinic's method; the new method is to push the flow through the original network. The algorithm does not use global concepts like augmenting path or level network, allowing natural implementations under distributive and parallel models of computation. The distributed and parallel implementations of the algorithm have the same time complexity as the S&V algorithm, but require less memory, allowing a more realistic distributed model.

For more information on the history of the problem, its applications, and underlying theory see [Even 1979], [Lawl 1976], [PS 1982], and [Tarj 1983].

2. Definitions and Notation

Let $G = (V, E)$ be a directed graph with a positive capacity $c(v, w)$ for every $(v, w) \in E$. Let $|V| = n$ and $|E| = m$. Define $c(v, w) = 0$ for $(v, w) \notin E$ to extend the capacity function to $V \times V$. Let *source* s and *sink* t be two distinguished vertices of the graph. A flow on G is a function $f : V \times V \rightarrow \mathbf{R}$ such that

$$0 \leq f(v, w) \leq c(v, w) \quad \forall (v, w) \in E \tag{1}$$

$$\sum_{w \in V} f(v, w) = 0 \quad \forall v \in V - \{s, t\} \tag{2}$$

$$f(v, w) = -f(w, v) \tag{3}$$

The *value* $|f|$ of a flow f is the net flow into the sink (or out of the source),

$$|f| = \sum_{w} f(w, t)$$

The *maximum flow* is the flow with the maximum value.

An important concept in the max-flow problem is the *cut*. A cut $S, \overline{S}$ is a partitioning of the vertices such that $S \bigcup \overline{S} = V$, $S \bigcap \overline{S} = \emptyset$, $s \in S$, and $t \in \overline{S}$. The capacity of the cut and flow across the cut are defined in a natural way:

$$c(S, \overline{S}) = \sum_{v \in S,\, w \in \overline{S}} c(v, w)$$

$$c(S, \overline{S}) = \sum_{v \in S,\, w \in \overline{S}} f(v, w)$$

A *pseudo-flow* is a function $g: V \times V \rightarrow \mathbf{R}$ which satisfies conditions (1) and (3), and a condition (4) given below (which is a relaxation of condition (2)).

$$\sum_{w \in V} f(v, w) \geq 0 \quad \forall v \in V \tag{4}$$

The algorithm computes a sequence of pseudo-flows converging to the maximum flow. Given a pseudo-flow g, for each vertex v we define *flow excess* e_v to be $e_v = \sum_{w} f(v, w)$.

The *residual capacity* with respect to a pseudo-flow g is a function $r_g: V \times V \rightarrow \mathbf{R}$ given by $r_g(v, w) = c(v, w) - f(v, w)$. The *residual graph* is $G_g = (V, E_g)$, where E_g contains all pairs of vertices with positive residual capacity: $E_g = \{(v, w) \mid r_g(v, w) > 0\}$.

3. Algorithm Description

The best way to understand the algorithm is to look at it from a distributed computational point of view. Assume that processors are located at vertices of the network, and they use local information to determine what to do next. Processors communicate with their neighbors along the network edges. Complexity measures are parallel time and communication; local computation is assumed to be instantaneous.

The algorithm proceeds in pulses; each pulse consists of three phases. At the beginning of pulse p, each vertex v contains two pieces of information: flow excess $e_v(p)$ and distance label $l_v(p)$. At the beginning of each pulse, there is a pseudo-flow $g(p)$ on G. As we will see later, the distance label $l_v(p)$ is a lower bound on the distance from v to

the sink in $G_g(p)$ (the distance is in the residual graph with each edge of unit length). The excess $e_v(p)$ is the flow excess with respect to $g(p)$; excess is always non-negative.

```
procedure STAGE_1;
begin
    INITIALIZE;
    repeat
        ∀v ≠ t do PUMP(v);
        p ← p + 1;
    until ∀v (l_v(p) = l_v(p-1)) & (e_v(p) = e_v(p-1));
end;

procedure INITIALIZE;
begin
    p ← 1;
    g(1) ← zero pseudo-flow;
    Do breadth-first search from the sink; ∀v set l_v(1) to the distance from v to the sink;
    e_v(1) ← 0 for v ∈ V - {s}; e_s(1) ← ∞;
end;

procedure PUMP(v);
begin
    (* phase 1 *)
    if l_v(p) ≠ ∞ then begin
        W ← {w | r_g(p)(v, w) > 0, l_w(p) < l_v(p)};
        while W ≠ ∅ and e_v(p) > 0 do begin
            Pick w ∈ W;
            W ← W - {w};
            PUSH(v, w);
        end;
    end;
    (* phase 2 *)
    S ← {w | r_g(p)(v, w) > 0};
    if S = ∅ then l_v(p+1) ← ∞
    else l_v(p+1) ← min_{w∈S} l_w(p) + 1;
    if l_v(p+1) ≥ n then l_v(p+1) ← ∞;
    (* phase 3 *)
    if l_v(p) ≠ l_v(p+1) then broadcast the new label to the neighbors;
end;

procedure PUSH(v, w);
    e_v(p) ← max(0, e_v(p) - r_g(p)(v, w));
    r_g(p)(v, w) ← max(0, r_g(p)(v, w) - e_v(p));
end;
```

Figure 1: Summary of the Algorithm

Figure 1 summarizes the algorithm, which is described below.

The intuition behind the algorithm is simple: each vertex tries to get rid of its flow excess by sending flow to its lower-labeled neighbors, i.e. towards the sink. This changes the residual flow and the residual graph, so the vertex updates its label based on the labels of vertices reachable from it in the new residual graph.

We start with a null flow and infinite excess at the source. The breadth-first search is done from the sink to initialize the distance labels. During the first phase of a pulse, each vertex with a finite label and non-zero excess tries to get rid of the excess by sending flow to its neighbors with smaller distance labels. The vertex does this by selecting such a neighbor and sending as much of the excess to the neighbors as the residual capacity of the corresponding edge allows. The vertex repeats this while it can, i.e. until its excess becomes zero or until the residual capacity of every edge from the vertex to lower-labeled vertices is zero.

During the second phase, each vertex updates its label by adding 1 to the minimum of the current labels of the vertices reachable from it. If no vertices are reachable, the label is set to infinity. If a label becomes n or greater, it is also set to infinity. During the third stage, vertices whose labels have changed broadcast the new labels to their neighbors.

This concludes the first stage of the algorithm. At the end of this stage, only vertices with distance label infinity can have positive excess. Then the second stage is performed.

The second stage works the same way as the first stage, but initial pseudo-flow and flow excess at vertices is taken from the end of the first stage, and distances are computed from the source rather then to the sink.

Remark: All previously known max-flow algorithms first find the maximum flow, then the minimal cut; however the algorithms do not "know" that max-flow has been found until they find min-cut. Our algorithm, on the other hand, first finds min-cut (in the first stage) and then max-flow (in the second stage). Also, min-cut is found gradually: vertices that get infinite distance labels at the first stage will be at the source side of the minimum cut found by the algorithm.

4. Correctness of the Algorithm

In this section we prove that the algorithm terminates and finds the maximum flow.

Claim 1: Distance labels of a vertex are non-decreasing:

$$\forall v \;\; l_v(p) \leq l_v(p+1)$$

Proof: Define $l_v(0) = l_v(1)$, and $g(0) = g(1)$. This definition is consistent with the way the algorithm computes distance labels. Then the claim is true for $p = 0$. Inductively assume that the claim is true for $p < k$, and prove it for $p = k$. Note that if a vertex gets an infinite label, it will retain it until the end of the stage; therefore the claim holds for infinite labeled vertices. We restrict ourselves to finite $l_v(p)$ and $l_v(p+1)$.

Fix a vertex v. Define $A = \{w \mid (v, w) \in E_{g(p)}\}$ and $B = \{w \mid (v, w) \in E_{g(p+1)}\}$. A is the set of vertices reachable from v at the beginning of pulse p, and B is the set of vertices reachable at the beginning of pulse $p+1$. Because flow is pushed to a lower-labeled vertices, we have

$$a \in A - B \Rightarrow l_a(p) < l_v(p) \tag{5}$$

$$b \in B - A \Rightarrow l_b(p) > l_v(p) \tag{6}$$

Using the induction assumption and unequalities (5) and (6) we obtain

$$l_v(p) - 1 = \min_{w \in A} l_w(p-1) \leq \min_{w \in A} l_w(p) \leq \min_{w \in B} l_w(p) = l_v(p+1).$$

This proves the claim. •

Claim 2: If in n pulses no distance labels change, stage terminates.

Proof: Assume that the distance labels stay the same from pulse $k+1$ to pulse $k+n$. The algorithm will terminate if all vertices with non-zero excess have infinite distance label at some pulse from $k+1$ to $k+n$. Let $S(p)$ be the set of all finite labeled vertices with non-zero excess at pulse p:

$$S(p) = \{v \mid l_v(p) < \infty \ \& \ e_v(p) > 0\}$$

Let $d(p) = \max_{v \in S(p)} l_v$. For $k+1 \leq p \leq k+n$ distance labels do not change, and therefore each vertex always succeeds in getting rid of its excess; new access can come only from higher-labeled vertices. Therefore for $k+1 \leq p \leq k+n$, $d(p+1) < d(p)$. It follows that by pulse $d(k+1+d(k+1))$, $S(p) = \{t\}$, and the algorithm terminates. •

Claim 3: The total amount of distance label change does not exceed $(n-1)^2$.

Proof: The distance label of the sink never changes; other distance labels start from a positive value and increase until they reach or exceed n; the total amount of change is $(n-1) \times (n-1)$. •

Lemma 1: The algorithm terminates.

Proof: From claims 1-3 it follows that the first stage terminates. By a similar argument we can show that the second stage also terminates. •

Definition: By $dist_{G_{g(p)}}(a, b)$ we denote the distance from a to b in $G_{g(p)}$.

Claim 4: Distance labels are lower bounds on the distance to the sink:

$$\forall v \in V,\ l_v(p) \leq dist_{G_{g(p)}}(v, t).$$

Proof: We induct on the value of $k = dist_{G_{g(p)}}(v, t)$. For $k = 0$, v must be the sink, and the claim holds. Assume claim holds for $k < i$. Let $k = i$. Then there is a v–t path, $v - w_1 - \cdots - w_k = t$, of length k in $G_{g(p)}$. By claim 1 and the inductive assumption, we have

$$l_{w_1}(p-1) \leq l_{w_1}(p) \leq k-1.$$

But $(v, w_1) \in E_{g(p)}$, so $l_v(p) \leq l_{w_1}(p-1) \leq k$, Q.E.D. •

The previous claim assures that distance labels are lower bounds on the distance to the sink in the current residual graph. The next claim is that at the end of the first stage, the distance labels are true distances to the sink.

Claim 5: Let T be the last pulse of the first stage. Then

$$l_v(T) = dist_{G_{g(T)}}(v, t)$$

Proof: By claim 4, $l_v(T) \leq dist_{G_{g(T)}}(v, t)$. It remains to show that the set

$$S = \{v \mid l_v(T) < dist_{G_{g(T)}}(v, t)\}$$

is empty.

Assume for contradiction that $S \neq \emptyset$, let w be an element of S with the smallest distance label: $l_w(T) = \min_{v \in S} dist_{G_{g(T)}}(v, t)$. Since $t \notin S$, and t is the only vertex with zero label, $l_w(T) > 0$. Since T is the last pulse, distance labels do not change during T; in particular $l_w(T) = l_w(T-1)$. It follows that there is a w' such that

$$l_{w'}(T) = l_{w'}(T-1) = l_w(T) - 1$$

By the choice of w, $l_{w'}(T) = dist_{G_{g(T)}}(w', t)$, and

$$dist_{G_{g(T)}}(w, t) \leq dist_{G_{g(T)}}(w', t) + 1 = l_w(T) \Rightarrow w \notin S.$$

Contradiction. •

Claim 6: Let T be the last pulse, and define

$$S = \{v \mid l_v(T) = \infty\}$$

$$\overline{S} = \{w \mid l_w(T) < \infty\}$$

Then $\forall v \in S,\ w \in \overline{S}\ \ c_{G_g(T)}(v, w) = 0$.

Proof: By the previous lemma, vertices in $\overline{S}$ have paths to t in the residual graph with respect to $g(T)$, and vertices in S do not have such paths. •

The next claim follows easily from the previous one.

Claim 7: $\forall v \in S,\ w \in \overline{S}$ the residual capacity of (v, w) does not change during the second phase of the algorithm.

The following fact is true for every pseudo-flow.

Claim 8: Let g be a pseudo-flow and let $S \subset V$ be a subset of vertices. Then

$$\sum_{v \in S, w \in V-S} c_g(v, w) \geq \sum_{v \in S} e_g(v).$$

Proof: Using (1), (2), and (4) we obtain

$$\sum_{v \in S} e_v \leq \sum_{v \in S} \sum_{w \in V} g(v, w) = \sum_{v \in S} \sum_{w \in V-S} g(v, w) \leq \sum_{v \in S} \sum_{w \in V-S} c(v, w)$$

Q.E.D. •

Claim 9: At the end of the second stage, all vertices other then s and t have a zero excess.

Proof: Let $V_\infty = \{v \in V - \{s, t\} \mid l_v = \infty$ after the second stage$\}$. Then $\forall s \in V_\infty$ and $v \in V - V_\infty$, $c_g(s, v) = 0$. By claims 6 and 9, the vertices in V_∞ must have zero excess. •

Lemma 2: The algorithm computes a maximum flow.

Proof: Claim 9 implies that the pseudo-flow computed by the algorithm is really a flow. Claims 6 and 7 imply that in the residual graph constructed with respect to the final flow, the sink is not reachable from the source, so by the augmenting path theorem the flow is maximum.

5. Analysis and Implementation Details

5.1. Bound on the Number of Pulses

By refining of the termination argument of the previous section we obtain an upper bound on the number of pulses in the algorithm. We provide the bound for the first stage of the algorithm; the same bound can be proven for the second stage by a similar argument.

Theorem 1: The first stage of the algorithm takes at most $2n^2$ pulses.

Proof: Let a *flow atom* be a piece of flow that stays together during the execution of the algorithm. The notion of flow atom is well-defined because the algorithm terminates. At each pulse, one of the following three possibilities can occur to a flow atom:

(1) An atom is sent to a lower labeled vertex, and the distance label of the vertex does not change.

(2) An atom is sent to a lower labeled vertex, and the distance label of the vertex increases by $d > 0$.

(3) An atom is not sent; in this case the distance label of the vertex at which the atom is must increase by $d > 0$.

Consider an atom that moves last. During a pulse, in case (1) the atom moves closer to the sink, and in cases (2) and (3) it moves away by d or $d-1$, respectively. Since the initial distance from the sink is no greater then $n-1$, and the number of steps back is no greater then $(n-1)^2$, the atom moves for at most $(n-1)^2 + (n-1)^2 + (n-1) < 2n^2 - n$ pulses. After atoms stop moving, the residual graph remains fixed, and the algorithm must terminate in n pulses. •

5.2. Sequential Implementation

In this section, we propose specific data structures for sequential implementation of the algorithm, and show that the complexity of the implementation is $O(n^3)$ (under the RAM model of computation [AHU 1974]).

In the proposed sequential implementation, vertices are processed in a round-robin fashion. There are n buckets at each vertex; the i-th bucket contains a linked list of neighboring vertices with distance label i. Each vertex maintains the smallest non-empty bucket index. When this bucket becomes empty, the index is updated. Since distance labels on vertices never decrease, the index never decreases, so the index update time is $O(n)$ per vertex, or $O(n^2)$ per stage. Index updates are not the bottleneck of the algorithm, so we will ignore them in further analysis. This also takes care of the distance label computations performed during phase 2 of the algorithm, since the distance label can be easily computed when the index is computed.

Lemma 3: The number of executions of the PUSH procedure is $O(n^3)$.

Proof: We distinguish two kinds of PUSH operations: one that saturates the residual edge (v, w) and one that does not saturate the edge. When a residual edge is saturated, no flow is moved through the corresponding edge of G until its head vertex moves back so that its distance label exceeds the distance label of the tail vertex. For each edge of G, the corresponding residual edge can be saturated n times. The total number of saturating edge operations does not exceed $nm \in O(n^3)$. During each pulse, at most one non-saturating PUSH operation is performed from each vertex. Since the number of pulses is $O(n^2)$, the total number of non-saturating PUSH operations does not exceed $O(n^3)$. •

Phase 3 of the PUSH procedure is implemented as follows. When a label of a vertex changes from i to j, it removes itself from the i-th bucket of its neighbors and inserts itself into the j-th buckets. Using right data structures, remove and insert operations take constant time. Given that, we can prove the following claim by an argument similar to the proof of the previous one.

Claim 10: Broadcasting the new distance labels in phase 3 of the algorithm costs $O(nm)$ operations per stage.

Lemma 3 and claim 10 give the following theorem:

Theorem 2: The sequential implementation of the algorithm runs in time $O(n^3)$.

5.3. Distributed Implementation

We consider the following model of distributed computation. The model assumes that local computation is much faster then inter-processor communication, and that the amount of local memory of a processor is proportional to the number of its neighbors in the network. The processors are located at the vertices of the network, and communicate through the links. Each processor has unlimited computational speed, and its local memory is proportional to the degree of the corresponding vertex. We will consider both synchronous and asynchronous cases. In either case, we will be interested in two complexity measures of an algorithm: parallel time and the number of messages (communication complexity). For more details on the model, see [Awer 1984].

Theorem 1 gives an $O(n^2)$ running time under the synchronous model; $O(n^3)$ communication complexity bound follows from the Proof of Theorem 2. It has been noted by Awerbuch [Awer 1985] that using the synchronizer protocol [Awer 1984], the algorithm can be implemented under asynchronous model with $O(n^2 \log n)$ time complexity

and $O(n^3)$ communication complexity.

If the restriction on processor memory is omitted, the S&V algorithm can be implemented with the same time and communication complexity bounds. However the S&V algorithm seems to require $\Omega(n^2)$ storage at each vertex (or $\Omega(n)$ storage at for each edge) and can not be implemented under the restricted memory model, which is better in many practical situations.

5.4. Parallel Implementation

The parallel computation model used is PRAM [FW 1978] without concurrent writing. The implementation of the algorithm under this model is very close to synchronous distributed implementation, except that trees of processors are added at vertices to enable data access. Because of these trees, each pulse takes $O(\log n)$ time, and the total parallel running time of the algorithm is $O(n^2 \log n)$.

Again, the implementation is very similar to the S&V algorithm implementation, but requires less memory - in fact, it requites only constant amount of memory at each processor.

6. Conclusions

We have described a new algorithm for computing max-flow. The algorithm is very natural: it is both intuitive and robust with respect to an implementation machine. Performance of the algorithm on dense graphs and under parallel models of computation is as good as that of the best max-flow algorithms known before, and its parallel implementation uses less memory then other parallel max-flow algorithms.

Acknowledgement

I would like to thank Baruch Awerbuch, David Shmoys, and Robert Tarjan for many suggestions and stimulating discussions, and Charles Leiserson for comments on a draft of the paper.

References

[AHU 1974] A.V. Aho, J.E. Hopcroft, J.D. Ullman, *The Design and Analysis of Computer Algorithms,* Addison-Wesley, (1974).

[Awer 1984] B. Awerbuch, *An Efficient Network Synchronization Protocol,* Proc. of 16th ACM Symp. on Theory of Comp. (1984), pp. 522-525.

[Awer 1985] B. Awerbuch, personal communication.

[Din 1970] E.A. Dinic, *Algorithm for Solution of a Problem of Maximum Flow in Networks with Power Estimation,* Soviet Math. Dokl., 11 (1970), pp. 1277-1280.

[Even 1979] S. Even, *Graph Algorithms,* Computer Science Press, Potomac, MD, (1979).

[FF 1956] L.R. Ford, Jr. and D.R. Fulkerson, *Maximal Flow Through a Network,* Can. J. Math., 8 (1956), pp. 399-404.

[FW 1978] S. Fortune and J. Willie, *Parallelism in Random Access Machines,* Proc. 10th ACM Symp. on Theory of Comp. (1978), pp. 114-118.

[Gal 1981] Z. Galil, *On the Theoretical Efficiency of Various Network Flow Algorithms,* Theoretical Comp. Sc., 14 (1981), pp. 103-111.

[Kars 1974] A.V. Karsanov, *Determining the Maximal Flow in a Network by the Method of Preflows,* Soviet Math. Dokl., 15 (1974), pp. 434-437.

[Lawl 1976] E.L. Lawler, *Combinatorial Optimization: Networks and Matroids,* Holt, Rinehart, and Winston, New York, (1976).

[MKM 1978] V.M. Malhotra, M.P. Kumar, and S.N. Maheshwari, *An $O(|V|^3)$ Algorithm for Finding Maximum Flows in Networks,* Inform. Proc. Letters, 7 (1978), pp. 277-278.

[PS 1982] C.H. Papadimitriou and K. Sleiglitz, *Combinatorial Optimization: Algorithms and Complexity,* Prentice-Hall, Englewood Cliffs, NJ, (1982).

[Tarj 1983] R.E. Tarjan, *Data Structures and Network Algorithms,* Society for Industr. and Appl. Math., (1983).

[SL 1980] D.D. Sleator, *An $O(nm \log n)$ Algorithm for Maximum Network Flow,* Tech. Rep. STAN-CS-80-831 (1980), Computer Science Dept., Stanford Univ., Stanford, CA.

[SV 1982] Y. Shiloach and U. Vishkin, *An $O(n^2 \log n)$ Parallel Max-Flow Algorithm,* J. of Algorithms, 3 (1982), pp. 126-146.