EFFICIENT MAXIMUM FLOW ALGORITHMS

Robert E. Tarjan

CS-TR-311-90

March 1991

Efficient Maximum Flow Algorithms

Robert E. Tarjan*

Department of Computer Science
Princeton University
Princeton, NJ 08544

and

NEC Research Institute
Princeton, NJ 08540

Abstract

Discovering efficient algorithms to compute flows in networks has been a goal of researchers in operations research and computer science for over 35 years. In this paper we review some recent developments in algorithms for the single-commodity maximum flow problem, and we comment on possible additional improvements. Our criterion for efficiency is theoretical worst-case running time on large sparse problems, ignoring constant factors.

*Research at Princeton University partially supported by DIMACS (Center for Discrete Mathematics and Theoretical Computer Science) a National Science Foundation Science and Technology Center, grant NSF-STC88-09648, and by National Science Foundation grant No. CCR-8920505.

1. Terminology

A *directed graph* $G = (V, E)$ consists of a vertex set V and an edge set E, each element of which is an ordered pair of distinct vertices. We shall allow only *symmetric* graphs, which are graphs such that $(v, w) \in E$ if and only if $(w, v) \in E$; this assumption entails no loss of generality. A *network* is a symmetric directed graph with a real-valued *capacity function* u on the edges. We denote the capacity of an edge (v, w) by $u(v, w)$, deleting the redundant set of parentheses, and similarly for other functions on the edges. We denote by U the maximum absolute value of any edge capacity.

A *pseudoflow* on a network is a real-valued function f on the edges, satisfying the following constraints for every edge (v, w):

(1) $$f(v, w) \leq u(v, w) \quad \text{(flow capacity constraint);}$$

(2) $$f(v, w) = -f(w, v) \quad \text{(flow antisymmetry constraint).}$$

The antisymmetry constraint is somewhat nonstandard but simplifies the statements of certain definitions and results; for some discussion of this see [18].

For a given pseudoflow f, the *flow excess* at a vertex v is

(3) $$e(v) = \sum_{(u,v) \in E} f(u, v) .$$

A pseudoflow is a *preflow* (with respect to a designated source vertex s) if $e(v) \geq 0$ for every vertex $v \neq s$. A pseudoflow is a *flow* (with respect to a designated source vertex s and a designated sink vertex t) if $e(v) = 0$ for every vertex $v \notin \{s, t\}$. The constraint $e(v) = 0$ for $v \notin \{s, t\}$ is called *flow conservation*. The *value* of a flow is $e(t)$. It is easy to show that the value of a flow is equal to the negative of the excess of the source, $-e(s)$. A flow is *maximum* if its value is as large as possible. The *maximum flow problem* is that of computing a maximum flow for a given network with a given source and sink.

In the remaining two sections we shall discuss algorithms for the maximum flow problem, along with some possible directions for future research. Classical results on network flow theory can be found in the books [13, 24, 27]; two more-recent surveys of work in the area are [1, 18]. Section 2 discusses the classical *augmenting path method* and algorithms based on it. Section 3 discusses the *preflow push method*, a relatively recent discovery that leads to simplified algorithms and improved running times.

2. The Augmenting Path Method

The maximum flow problem is a special case of linear programming, and can thus be solved by applying any linear programming algorithm, such as the simplex method [8] or an interior point method [21]. The fastest known methods for this problem are, however, specialized combinatorial algorithms. But, as we shall see, a specialized version of the simplex method is also quite efficient.

We shall briefly examine some general approaches to computing maximum flows and discuss what is known about each. To simplify matters, we assume that the function that is identically zero on all edges is a flow; that is, $0 \leq u(v, w)$ for every edge (v, w) (capacity nonnegativity constraint). We use the zero flow as a starting flow and gradually improve it to a maximum flow. If the nonnegativity constraint is violated, the problem of finding a starting flow can itself be solved as a maximum flow problem satisfying the nonnegativity constraint [24]; thus there is no loss of generality in assuming nonnegativity.

The classical combinatorial algorithm for the maximum flow problem is the *augmenting path method* of Ford and Fulkerson [12, 13]. To define this method we need a few more concepts. The *residual capacity* of an edge (v, w) with respect to a pseudoflow f is $u_f(v, w) = u(v, w) - f(v, w)$. An edge is *saturated* if its residual capacity is zero and *nonsaturated* otherwise. An *augmenting path* is a path from the source s to the sink t consisting entirely of nonsaturated edges. Given an augmenting path with respect to some flow, the value of the flow can be increased by adding to the flow of each edge along the path any positive value δ not more than the minimum of the residual capacities of the edges on the path (and subtracting δ from the flows of the reversals of the edges on the path). The augmenting path method consists of beginning with the zero flow (or any flow) and repeatedly finding an augmenting path and sending as much flow as possible along it, continuing until there are no augmenting paths.

With arbitrary selection of augmenting paths, the augmenting path method can be very inefficient. Indeed, it need not even terminate after a finite number of augmentations if the capacities are irrational numbers [13]. A careful choice of augmenting paths does, however, lead to a polynomial time bound. Edmonds and Karp [11] showed that if each augmentation is on a path of fewest edges, then the augmenting path algorithm runs in $O(nm^2)$ time; successive augmenting paths are nondecreasing in length. Dinic [9] independently made essentially the same discovery, and noted further that all augmenting

paths of the same length can be found in a single computation, called a *phase*, resulting in an overall time bound of $O(n^2m)$. Sleator and Tarjan [25, 26] discovered a way to make Dinic's algorithm run in $O(nm \log n)$ time using a sophisticated data structure called a *dynamic tree*, and Dinic [10] and Gabow [14] independently found a relatively simple capacity-scaling algorithm that runs in $O(nm \log U)$ time on a network with integer edge capacities.

Recently, Goldfarb and Hao [20] have taken a careful look at the simplex algorithm as applied to the maximum flow problem. They have devised a version of the primal network simplex algorithm in which pivots are chosen using a shortest path strategy as in the Dinic and Edmonds-Karp versions of the augmenting path method. The time bound of their algorithm is $O(n^2m)$. Using dynamic trees, Goldberg, Grigoriadis, and Tarjan [17] have improved the running time of this algorithm to $O(nm \log n)$.

3. The Preflow Push Method

Somewhat better bounds have been obtained by using alternative algorithmic approaches. We shall describe the *preflow push method*, invented by Goldberg [15] and developed more fully by Goldberg and Tarjan [16, 19] and others. Not only does this method give improved worst-cast time bounds, it leads to very simple computer codes that run fast in practice.

The preflow push method is based on two ideas. The first is to break the flow movement into smaller steps. Instead of moving an amount of flow all the way from the source to the sink at once, flow is moved through one edge at a time. The second idea is to provide directionality to the flow movement by maintaining an estimate at each vertex of the distance to the sink via a path of residual edges. Flow is moved toward the sink with respect to these estimates, and the estimates are intermittently updated as they become less accurate.

The distance estimates are defined by a *valid labeling* of the vertices. A valid labeling is a nonnegative integer function on the vertices, with the following properties:

$$d(s) = n\,, \tag{4}$$

$$d(t) = 0\,, \tag{5}$$

$$d(v) \leq d(w) + 1 \quad \text{for every residual edge}(v, w)\,. \tag{6}$$

For two vertices v, w, let $\ell(v, w)$ be the minimum number of edges on a path of residual edges from v to w. Then (4), (5), and (6) imply that, for any valid labeling d, $d(v) \leq \min\{\ell(v,t), \ell(v,s)+n\}$. The labeling given by $d(v) \leq \min\{\ell(v,t), \ell(v,s)+n\}$ is a valid labeling, and is the largest possible valid labeling.

We call an edge (v, w) *admissible* with respect to a valid labeling if (v, w) is residual and $d(v) = d(w) + 1$. The preflow push method begins by choosing an initial preflow f that is zero on all edges except those incident to s and that saturates every edge out of s. That is, $f(v, w) = 0$ if $s \notin \{v, w\}$; $f(s, v) = u(s, v)$; $f(v, s) = -u(s, v)$. Then the method chooses an initial valid labeling d. One possibility is $d(v) = 0$ if $v \neq s$, $d(s) = n$; another possibility is $d(v) = \min\{\ell(v,t), \ell(v,s)+n\}$. The latter labeling, which we call the *tight labeling*, can be computed by performing breadth-first searches from s and from t. The main part of the algorithm consists of repeating the following two steps in any order, until the preflow f becomes a flow.

push (v, w):

applicability: $e(v) > 0$, $v \neq t$, (v, w) is eligible.

action: increase $f(v, w)$ by $\min\{e(v), u_f(v, w)\}$.

relabel (v):

applicability: $e(v) > 0$, $v \neq t$, and no edge of the form (v, w) is eligible.

action: set $d(v) = \min\{d(w) + 1 \mid (v, w) \text{ is an eligible edge}\}$.

A push is called *saturating* if $u_f(v, w)$ units of flow are moved and *nonsaturating* otherwise.

In discussing various versions of the preflow push algorithm we omit implementation details; see [16, 18, 19]. Ignoring certain overhead issues, the running time of the algorithm depends on the number of saturating and nonsaturating pushes and the number of relabelings. The following lemmas, whose proofs we omit (see [16, 18, 19]), provide a running time analysis of the generic algorithm.

Lemma 1. *For any active vertex v, $d(v) \leq 2n - 1$.*

Lemma 2. *The label number of relabeling operations is $O(n^2)$, taking a total time of $O(nm)$.*

Lemma 3. *The total number of saturating pushes is* $O(nm)$.

Lemma 4. *The total number of nonsaturating pushes is* $O(n^2m)$.

The nonsaturating pushes dominate the time of the generic algorithm, and result in an overall time bound of $O(n^2m)$. In order to obtain a faster version of the algorithm, either the number of pushes must be reduced or the time per push must be reduced.

Goldberg [15] (see also [16, 19]) suggested a strategy for selecting pushes called the FIFO (first-in, first-out) method. The FIFO method maintains a queue of all the vertices other than t with positive excess. The method consists of repeatedly selecting the first vertex on the queue, pushing flow from it until it can be relabeled, relabeling it, and adding it to the end of the queue if it still has positive excess. Vertices made newly active by pushes are also added to the end of the queue.

The FIFO method reduces the number of nonsaturating pushes, and the overall running time of the algorithm, to $O(n^3)$. This matches a bound previously obtained by others [22, 23, 28], and it leads to a simple computer program that is fast in practice. A parallel version of the FIFO method, in which all active vertices are processed at once, runs in $O(n^2 \log n)$ time with n processors [16, 19].

A variant of the FIFO method, the *largest label method* [16, 19], always processes a vertex of largest label. This method has a running time of $O(n^2\sqrt{m})$ [7]. Ahuja and Orlin [2] found an *excess-scaling* version of the preflow push algorithm that runs in $O(nm + n^2 \log U)$ time for networks with integer capacities. An improved version of this algorithm runs in $O(nm + n^2(\log U)^{1/2})$ time [3].

The use of the dynamic tree data structure further improves some of these algorithms. With dynamic trees, the FIFO algorithm runs in $O(nm \log(n^2/m))$ time, as does the largest label algorithm [16, 19]. The improved excess-scaling algorithm runs in $\left(nm \log\left(\frac{n}{m}(\log U)^{1/2} + 2\right)\right)$ time on integer-capacity networks [3].

Recently, Cheriyan and Hagerup [5] and others have explored ways to reduce the running time of the preflow push method by using randomization, excess scaling, dynamic trees, and array-based table lookup techniques – in short, by combining most of the previously known techniques for the problem with several new ideas. The first step in this line of research was a randomized algorithm with an expected running time of $O(nm + n^2(\log n)^3)$ devised by Cheriyan and Hagerup [5]. The time bound for this algorithm was reduced to $O(nm + n^2(\log n)^2)$ by Tarjan in unpublished work. Alon [4]

found a way to make the Cheriyan-Hagerup algorithm deterministic, but with an increase in the time bound to $O(nm + n^{8/3} \log n)$. Cheriyan, Hagerup, and Mehlhorn [6] devised a simpler algorithm giving the same time bounds; in addition, they found a way to use table lookup techniques to obtain an $O(n^3/\log n)$ deterministic running time on a unit-cost random-access machine.

At the heart of the Cheriyan-Hagerup approach is a way of reducing pushes by analyzing a combinatorial game that arises in the operation of the preflow push algorithm. This game is still only incompletely understood. and it seems likely that a better analysis of the game will lead to improved running times for the maximum flow problem for both randomized and deterministic algorithms. Although the maximum flow problem has been extensively studied for over three decades, it has not yet revealed all its secrets.

References

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, "Network flows," in *Handbook in Operations Research and Management Science, Volume 1: Optimization*, G. L. Nemhauser, A. H. G. Rinnooy Kan, and M. J. Todd, eds., North-Holland, Amsterdam, 1990, 211–360.

[2] R. K. Ahuja and J. B. Orlin, "A fast and simple algorithm for the maximum flow problem," *Operations Research* **37** (1989), 748–759.

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

[4] N. Alon, "Generating pseudo-random permutations and maximum flow algorithms," manuscript, IBM Research Center, San Jose, CA, 1989.

[5] J. Cheriyan and T. Hagerup, "A randomized maximum-flow algorithm," *Proc. 30^{th} Annual IEEE Symp. on Foundations of Computer Science* (1989), 118–123.

[6] J. Cheriyan, T. Hagerup, and K. Mehlhorn, "Can a maximum flow be computed in $o(nm)$ time," *Automata, Languages and Programming, 17^{th} Internat. Colloquium, Lecture Notes in Computer Science* **443**, Springer-Verlag, New York (1990), 235–248.

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

[8] J. B. Dantzig, *Linear Programming and Extensions*, Princeton University Press, Princeton, NJ, 1962.

[9] E. A. Dinic, "Algorithm for solution of a problem of maximum flow in networks with power estimation," *Soviet Math. Dokl.* **11** (1970), 1277–1280.

[10] E. A. Dinic, "The method of scaling and transportation problems," (in Russian), *Issledovaniya po Diskretnoi Matematike*, Science, Moscow (1973).

[11] J. Edmonds and R. M. Karp, "Theoretical improvements in algorithmic efficiency for network flow problems," *J. Assoc. Computing Mach.* **19** (1972), 248–264.

[12] L. R. Ford, Jr. and D. R. Fulkerson, "Maximal flow through a network," *Canadian J. Math.* **8** (1956), 399–404.

[13] L. R. Ford, Jr. and D. R. Fulkerson, *Flows in Networks*, Princeton University Press, Princeton, NJ, 1962.

[14] H. N. Gabow, "Scaling algorithms for network problems," *J. Computer and System Sciences* **31** (1985), 148–168.

[15] A. V. Goldberg, "A new max-flow algorithm," Technical Report MIT/LCS/TM-291, Laboratory for Computer Science, M.I.T., Cambridge, MA, 1985.

[16] A. V. Goldberg, "Efficient graph algorithms for sequential and parallel computers," Technical Report TR-374, Laboratory for Computer Science, M.I.T., Cambridge, MA, 1987.

[17] A. V. Goldberg, M. D. Grigoriadis, and R. E. Tarjan, "Use of dynamic trees in a network simplex algorithm for the maximum flow problem," *Math. Prog.*, to appear.

[18] A. V. Goldberg, E. Tardos, and R. E. Tarjan, "Network flow algorithms," in *Paths, Flows, and VLSI-Layout*, B. Korte, L. Lovász, H. J. Prömel, and A. Schriver, eds., Springer-Verlag, Berlin, 1990, 101–164.

[19] A. V. Goldberg and R. E. Tarjan, "A new approach to the maximum flow problem," *J. Assoc. Computing Mach.* **35** (1988), 921–940.

[20] D. Goldfarb and J. Hao, "A primal simplex algorithm that solves the maximum flow problem in at most nm pivots and $O(n^2m)$ time," *Math. Prog.* **47** (1990), 353–365.

[21] N. Karmakar, "A new polynomial-time algorithm for linear programming," *Combinatorica* **4** (1984), 373–395.

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

[23] V. M. Malhotra, M. Pramodh Kumar, and S. N. Maheshwari, "An $O(|V|^3)$ algorithm for finding maximum flows in networks," *Information Processing Letters* **7** (1978), 277–278.

[24] C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, NJ, 1982.

[25] D. D. Sleator and R. E. Tarjan, "A data structure for dynamic trees," *J. Computer and System Sciences* **26** (1983), 362–391.

[26] D. D. Sleator and R. E. Tarjan, "Self-adjusting binary search trees," *J. Assoc. Computing Mach.* **32** (1985), 652–686.

[27] R. E. Tarjan, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[28] R. E. Tarjan, "A simple version of Karzanov's blocking flow algorithm," *Operations Research Letters* **2** (1984), 265–268.