MATHEMATICAL PROGRAMMING STUDY 26

A PUBLICATION OF THE MATHEMATICAL PROGRAMMING SOCIETY

Netflow at Pisa

Edited by G. GALLO and C. SANDI

MATHEMATICAL PROGRAMMING STUDY 26

A PUBLICATION OF THE MATHEMATICAL PROGRAMMING SOCIETY

Netflow at Pisa

Edited by G. GALLO and C. SANDI

A. Assad
P.M. Camerini
V. Campos
C.-H. Chen
N. Christofides
A. Colorni
A. Corberán
R.S. Dembo
U. Derigs
M. Desrochers
J. Desrosiers
M. Engquist
L.F. Escudero
M. Florian
G. Gallo
F. Gheysens
F. Glover
R. Glover
B. Golden
M.D. Grigoriadis
K.-D. Hackbarth
A. Haurie
E.L. Johnson
D. Klingman
M. Lucertini
F. Maffioli
T.L. Magnanti
P. Marcotte
M. Minoux
P. Mireault
E. Mota
G. Paletta
S. Pallotino
P. Pieroni
C. Sandi
M. Sauvé
G. Saviozzi
J. Siedersleben
F. Soumis
R.E. Tarjan
V. Valls
D. de Werra
R.T. Wong

1986

NORTH-HOLLAND – AMSTERDAM · NEW YORK · OXFORD

This book is also available in journal format on subscription.

ISBN: 0 444 879854

Published by:
ELSEVIER SCIENCE PUBLISHERS B.V.
P.O. Box 1991
1000 BZ Amsterdam
The Netherlands

Sole distributors for the U.S.A. and Canada:
ELSEVIER SCIENCE PUBLISHING COMPANY, INC.
52 Vanderbilt Avenue
New York, N.Y. 10017
U.S.A.

Library of Congress Cataloging in Publication Data

NETFLOW at Pisa.

(Mathematical programming study ; 26)
NETFLOW, an International Workshop on Network Flow Optimization, Theory and Practice was held in Pisa 3/28-31/83.
1. Network analysis (Planning)--Congresses. 2. Programming (Mathematics)--Congresses. I. Gallo, G. II. Sandi, C. III. International Workshop on Network Flow Optimization, Theory and Practice (1983 : Pisa, Italy) IV. Series.
T57.85.N455 1986 658.4'03'2 86-2347
ISBN 0-444-87985-4

PRINTED IN THE NETHERLANDS

MATHEMATICAL PROGRAMMING STUDIES

NORTH-HOLLAND – AMSTERDAM · NEW YORK · OXFORD

PREFACE

NETFLOW, an 'International Workshop on Network Flow Optimization: Theory and Practice', was held in Pisa from March 28 to 31, 1983. Jointly promoted and organized by the IBM Scientific Center, the Department of Computer Science of the University of Pisa and the Istituto di Elaborazione dell' Informazione (CNR) of Pisa, the workshop intended to provide updated tutorials and advanced research papers on network flow models and optimization techniques.

The increasing interest in mathematical methods for network flow optimization comes from their elegant simplicity, efficiency and from the numerous relevant applications of network flow models. Examples are production-distribution, urban traffic, railway systems, communications, facility location, routing and scheduling, file management, electrical and pipe networks. Optimization algorithms that exploit network structures can be one-hundred times faster than general algorithms, to produce solutions of the same quality. The possibility of displaying the network relationships in two-dimensional drawings greatly simplifies the input of models and the interpretation of output results, especially using the sophisticated graphical equipment now available. All these appealing features have made the network flow optimization theory, since the appearance of a systematic development in the early fifties, a well defined mathematical programming area, provided with a vast literature even in dedicated books and journals.

In so far as its size allows, this volume collects the workshop material, with selection criteria mainly based on the purpose of covering as many topics as possible. To achieve this overall view of the field, many contributions have been included as short papers. Items are presented in the volume according to the workshop structure: the series of full papers is opened and closed by two surveys, on shortest path algorithms and on non linear network flow models respectively. The former is addressed to what can be considered as the simplest, and yet very important, network optimization problem; the latter is concerned with a quite advanced topic, mainly focusing on transportation planning. In between, two expository papers deal with fundamental problems such as max flow and min-cost flow. In these papers, state-of-the-art algorithms are described in full detail with great attention to the data structure needed for their implementation. The remaining full papers contain particular research contributions in relevant areas such as assignment, transportation, vehicle routing and network design. The short papers cover almost all the network flow optimization field, again from the basic topics such as mathematical properties of network flow models, matching algorithms, single and multicommodity solution procedures, to specialized models arising in routing, scheduling, network design, location problems, and to the wide area of nonlinear models.

The organization of the Workshop, which was directed by the Editors of this Study, has relied upon the secretarial and administrative skill of Ms. Elena Avancini and Ms. Giuliana Tedeschi. We wish to take this opportunity to express our gratitude to them. The Editors are also indebted to Ms. Jean E. Cupit for the professional assistance with contributions from non-English speaking authors. A sincere thanks is finally due to the referees, who provided a crucial support to this effort.

CONTENTS

Preface v

R.E. Tarjan, Algorithms for maximum network flow 1

F. Glover, R. Glover and D. Klingman, Threshold assignment algorithm 12

G. Gallo and S. Pallottino, Shortest path methods: A unifying approach 38

C. Sandi, On a nonbasic dual method for the transportation problem 65

M.D. Grigoriadis, An efficient implementation of the network simplex method 83

T.L. Magnanti, P. Mireault and R.T. Wong, Tailoring Benders decomposition for uncapacitated network design 112

N. Christofides, V. Campos, A. Corberán and E. Mota, An algorithm for the rural postman problem on a directed graph 155

M. Florian, Nonlinear cost network models in transportation analysis 167

Short papers

D. de Werra, Variations on the integral decomposition property 197

U. Derigs, A short note on matching algorithms 200

J. Siedersleben, Common aspects of several network flow algorithms 205

N. Christofides and V. Valls, Finding all optimal solutions to the network flow problems 209

K.-D. Hackbarth, A heuristic procedure for calculating telecommunication transmission networks in consideration of network reliability 213

E.L. Johnson and P. Pieroni, A linear programming approach to the optimum network orientation problem 215

M. Engquist and C.-H. Chen, Computational comparison of two solution procedures for allocation/processing networks 218

G. Saviozzi, Advanced start for the multicommodity network flow problem 221

M. Lucertini and G. Paletta, A class of network design problems with multiple demand: Model formulation and an algorithmic approach 225

P.M. Camerini, A. Colorni and F. Maffioli, Some experience in applying a stochastic method to location problems 229

F. Gheysens, B. Golden and A. Assad, A new heuristic for determining fleet size and composition 233

M. Minoux, Solving integer minimum cost flows with separable convex cost objective polynomially 237
L.F. Escudero, A motivation for using the truncated Newton approach in a very large scale nonlinear network problem 240
R.S. Dembo, The performance of NLPNET, a large-scale nonlinear network optimizer 245
J. Desrosiers, F. Soumis, M. Desrochers and M. Sauvé, Vehicle routing and scheduling with time windows 249
A. Haurie, P. Marcotte, A game-theoretic approach to network equilibrium 252

Mathematical Programming Study 26 (1986) 1–11
North-Holland

ALGORITHMS FOR MAXIMUM NETWORK FLOW

Robert E. TARJAN
AT & T Bell Laboratories, Murray Hill, NJ 07974, USA

Received 15 June 1983
Revised manuscript received 3 February 1984

This paper is a survey, from the point of view of a theoretical computer scientist, of efficient algorithms for the maximum flow problem. Included is a discussion of the most efficient known algorithm for sparse graphs, which makes use of a novel data structure for representing rooted trees. Also discussed are the potential practical significance of the algorithms and open problems.

Key word: Maxflow.

1. Introduction

The maximum network flow problem is one of the classical problems of network optimization. From the point of view of the complexity theorist, it is also one of the most intriguing, because of the number, variety, and rich structure of the algorithms that have been proposed to solve it. This paper is a survey of the known maximum flow algorithms and the techniques they use. Our emphasis is on theoretical efficiency, as measured by the worst-case running time of an algorithm on a random-access computer. We shall generally use the 'unit-cost' measure: any operation on real numbers is assumed to take unit time. We shall occasionally use the 'logarithmic-cost' measure, under which any operation on real numbers takes time proportional to the number of bits of precision. For more information on our theoretical framework and in particular on these cost measures, see the books by Aho, Hopcroft and Ullman [1], Papadimitriou and Steiglitz [16], and Tarjan [24].

In Section 2 we define the maximum flow problem and discuss algorithms for solving it. The best methods are based on algorithms for the blocking flow problem, which we consider in Section 3. On sparse graphs, the most efficient known algorithm is due to Sleator and Tarjan [20, 21]. It uses a novel data structure for representing and manipulating rooted trees. Recently Gabow [9] has discovered a new maximum flow algorithm that combines a scaling technique with Dinic's algorithm. If the edge capacities are integers of moderate size, Gabow's method is competitive with that of Sleator and Tarjan. We discuss Gabow's method in Section 4, which also contains additional remarks, including a discussion of the potential practical significance of the theoretically fast algorithms and some open problems. Throughout the paper we omit many of the details and all of the proofs; these may be found in [24].

2. Maximum flow algorithms

Let $G=(V, E)$ be a directed graph with two distinguished vertices, a *source* s and a *sink* t, and a positive real-valued *capacity* $c(v, w)$ on every directed edge $[v, w]$. If $[v, w]$ is not an edge, we define $c(v, w)=0$. We denote the number of vertices by n and the number of edges by m. For ease in stating time bounds we assume $n=O(m)$. A *flow* on G is a real-valued function f on the vertex pairs satisfying

(i) (skew symmetry) $f(v, w)=-f(w, v)$ for all v, w.

(ii) (capacity constraint) $f(v, w) \leq c(v, w)$ for all v, w.

(iii) (flow conservation) For every vertex v other than s and t,

$$\sum_w f(v, w)=0.$$

We impose skew symmetry merely for technical convenience. Note that skew symmetry and the capacity constraint imply $f(v, w)=0$ if neither $[v, w]$ nor $[w, v]$ is an edge. If $f(v, w)>0$, we say there is a *flow from v to w of magnitude $f(v, w)$*. Flow conservation states that the total flow into any vertex other than s and t equals the total outgoing flow. The *value* of a flow f, denoted by $|f|$, is the net flow out of the source, $\sum_v f(s, v)$. The *maximum flow problem* is that of finding a flow of maximum value, called a *maximum flow*.

The classical theory of network flows was developed by Ford and Fulkerson [8]. To understand it, we need one more concept, that of a *cut*. A cut $X, \bar{X}$ is a partition of the vertex set into two parts, such that $s \in X$ and $t \in \bar{X}$. The *capacity* of the cut is

$$c(X, \bar{X})=\sum_{v \in X, w \in \bar{X}} c(v, w).$$

The net flow across the cut is

$$f(X, \bar{X})=\sum_{v \in X, w \in \bar{X}} f(v, w).$$

Flow conservation implies the following lemma:

Lemma 1. *For any flow f, the net flow across any cut X, $\bar{X}$ equals the flow value.*

Ford and Fulkerson's main result is the *max-flow, min-cut* theorem:

Theorem 1. *A flow f is maximum if and only if there is a cut $X, \bar{X}$ such that $|f|=c(X, \bar{X})$.*

The capacity constraint and Lemma 1 imply that, for any flow f and any cut X, $\bar{X}$,

$$|f|=\sum_{v \in X, w \in \bar{X}} f(v, w) \leq c(X, \bar{X}).$$

This gives the easy half of theorem 1. Ford and Fulkerson proved the converse by giving a method to increase the value of a flow if it is not maximum. For any flow

f, let us define the *residual graph* R for f to be the graph with vertex set V, source s, sink t, and an edge $[v, w]$ of capacity $c'(v, w) = c(v, w) - f(v, w)$ for every pair v, w such that $c(v, w) - f(v, w) > 0$. We can push up to $c'(v, w)$ additional units of flow from v to w by increasing $f(v, w)$.[1] An *augmenting path* for f is a path from s to t in R. Given an augmenting path p, we can increase the value of f by increasing $f(v, w)$ for every edge $[v, w]$ on p by any amount up to its *residual capacity*, defined to be $\min\{c'(v, w) | [v, w] \text{ on } p\}$.

Lemma 2. *Let f be any flow and f' a maximum flow on G. If R is the residual graph for f, then the value of a maximum flow on R is $|f'| - |f|$.*

Corollary 1. *A flow is maximum if and only if it has no augmenting path.*

We can find a maximum flow by beginning with the zero flow (the flow that is identically zero on all vertex pairs) and repeating the following step until obtaining a flow with no augmenting path:

Augmenting Step. Find an augmenting path p for the current flow and increase the value of the flow by pushing along the path an amount of flow equal to its residual capacity.

This is Ford and Fulkerson's *augmenting path method.* Once a flow f with no augmenting path is found, the set of vertices reachable from s in the residual graph for f defines a minimum cut. Thus we can compute both a maximum flow and a minimum cut by the augmenting path method.

Remark. Although we can compute a minimum cut from a maximum flow in $O(m)$ time, no similarly easy way is known of computing a maximum flow from a minimum cut.

Finding one augmenting path takes $O(m)$ time with any standard search method, such as depth-first or breadth-first search [24]. If all the capacities are integers, each augmentation increases the flow value by at least one, and the running time of the augmenting path method is $O(m|f|)$. The existence of this method implies the important *integrality theorem*:

Theorem 2. *If all capacities are integers, there is an integral maximum flow (a flow that is an integer on every edge).*

Unfortunately, if the capacities are large integers, the running time of the augmenting path method can be very large. Furthermore, if the capacities are arbitrary real numbers the method need not terminate, and although successive flow values will

[1] Whenever we increase $f(v, w)$, we must decrease $f(w, v)$ by the same amount to maintain skew symmetry. We shall generally not mention this explicitly.

converge they need not converge to the value of the maximum flow [8]. In spite of these theoretical drawbacks, the augmenting path method has been used with practical success for many years.

Edmonds and Karp [7] and independently Dinic [6] were the first to propose maximum flow algorithms efficient in the worst case. Edmonds and Karp's paper contains several interesting results on both the maximum flow and the *minimum-cost flow* problems. (We shall not discuss the latter problem here.) For the maximum flow problem, they proposed two ways of selecting augmenting paths to make Ford and Fulkerson's method efficient. One way is to always choose an augmenting path of maximum residual capacity. With this method, each augmenting step takes $O(m)$ time using a variant of Dial's implementation of Dijkstra's shortest path algorithm [5]. (See the appendix.) If the edge capacities are integers, the number of augmentations is $O(m \log c)$, where c is the maximum edge capacity, and the total running time is $O(m^2 \log c)$. If the edge capacities are arbitrary numbers, the method need not terminate, but the flow value will converge to that of a maximum flow [17].

The other method proposed by Edmonds and Karp is to always augment along a shortest path (one containing fewest edges). With this method each augmentation takes $O(m)$ time using breadth-first search, and there are are $O(nm)$ augmentations, for a total running time of $O(nm^2)$. Thus this algorithm has a worst-case polynomial running time, independent of the magnitudes of the edge capacities (for the uniform-cost complexity measure). It is natural to speculate that one reason Ford and Fulkerson's method works so well in practice is that it is generally implemented to select shortest augmenting paths.

Dinic independently proposed the idea of augmenting along shortest paths but went one step further. He described an algorithm that augments simultaneously along all paths of shortest length. To understand this algorithm we need two new concepts.

A *blocking flow* is a flow such that every path from s to t contains at least one edge $[v, w]$ such that $f(v, w) = c(v, w)$. (We call such an edge *saturated.*) A blocking flow need not be maximum, since it may be possible to increase the flow value by rerouting some of the flow. Let f be a flow and R its residual graph. The *level graph* L for f is the subgraph of R containing only the vertices reachable from s and only the edges $[v, w]$ such that $level(w) = level(v) + 1$, where $level(v)$ for any vertex v is the length of a shortest path from s to v in R. The level graph contains every shortest augmenting path for f and is acyclic.

Dinic's algorithm consists of repeatedly augmenting the current flow using a blocking flow on its level graph. More precisely, we begin with the zero flow and repeat the following step until t is not in the level graph for the current flow:

Blocking Step. Find a blocking flow f' on the level graph for the current flow f. Replace f by the flow $f + f'$ defined by $(f + f')(v, w) = f(v, w) + f'(v, w)$.

Dinic proved that each blocking step increases the distance from s to t in the residual graph. Thus there are at most $n - 1$ blocking steps, each of which requires

finding a blocking flow on an acyclic graph. The overall running time of the algorithm depends upon the time needed to find a blocking flow. In the next section we discuss blocking flow algorithms.

3. Blocking flow algorithms

Dinic's main contribution was in showing that the maximum flow problem can be reduced to $n-1$ blocking flow problems on acyclic graphs. In this section we shall survey blocking flow algorithms. Each of these algorithms gives a corresponding algorithm for maximum flow whose running time is larger by a factor of n. In this section we assume that G is acyclic.

Dinic suggested an algorithm that constructs a blocking flow by pushing flow along one path at a time. The algorithm consists of repeating the following step until there is no path from s to t:

Saturating Step. Find a path p from s to t. Send along p an amount of flow equal to $\Delta = \min\{c(v, w) | [v, w] \text{ on } p\}$. Reduce the capacity of each edge on p by Δ. Clean up the graph by deleting every edge of zero capacity and then deleting every vertex and edge not on a path from s to t.

Cleaning up the graph can be done by deleting all vertices of in-degree zero other than s and all vertices of out-degree zero other than t and repeating until there are no such vertices. With Dinic's method, the time per saturating step is $O(n)$, not counting clean-up, and the time for clean-up summed over all steps is $O(m)$. Since each saturating step removes at least one edge, there are at most m steps, giving a time of $O(nm)$ to find a blocking flow and $O(n^2m)$ to find a maximum flow.

Dinic's blocking flow algorithm saturates one edge at a time, spending $O(n)$ time per edge saturated. On dense graphs there are faster methods that in effect saturate one vertex at a time, attaining an $O(n^2)$ time bound for the blocking flow problem. Karzanov [14] presented the first such algorithm. Although his original method was rather complicated, there is a simplified variant, called the *wave method* [23, 24], that we shall present here.

We need the idea of a *preflow*. A preflow f is a skew-symmetric function on vertex pairs that satisfies the capacity constraint and has a non-negative net flow $\Delta f(v)$ into every vertex v other than s and t, where we define $\Delta f(v) = \sum_u f(u, v)$. A vertex v is *balanced* if $\Delta f(v) = 0$ and *unbalanced* if $\Delta f(v) > 0$. The preflow is *blocking* if it saturates at least one edge on every path from s to t. The wave method starts with a blocking preflow and gradually converts it into a blocking flow by balancing vertices in successive forward and backward passes over the graph.

Each vertex is in one of two states: *unblocked* or *blocked*. An unblocked vertex can become blocked but not vice-versa. We balance an unblocked vertex by increasing the flow out and balance a blocked vertex by decreasing the flow in. More precisely, we balance an unblocked vertex v by repeating the following step until

$\Delta f(v) = 0$ (the balancing succeeds) or there is no unsaturated edge $[v, w]$ such that w is unblocked (the balancing fails):

Increasing Step. Choose an unsaturated edge $[v, w]$ such that w is unblocked. Increase $f(v, w)$ by $\min\{c(v, w) - f(v, w), \Delta f(v)\}$.

We balance a blocked vertex v by repeating the following step until $\Delta f(v) = 0$ (such a balancing always succeeds):

Decreasing Step. Choose an edge $[u, v]$ with positive flow. Decrease $f(u, v)$ by $\min\{f(u, v), \Delta f(v)\}$.

To find a blocking flow, we begin with the preflow that saturates every edge out of s and is zero on every other edge, make s blocked and all other vertices unblocked, and repeat *increase flow* followed by *decrease flow* until there are no unbalanced vertices.

Increase Flow. Scan the vertices other than s and t in topological order (an order such that if $[v, w]$ is an edge, v is scanned before w). When scanning a vertex v, balance v if it is unbalanced and unblocked, and if the balancing fails make v blocked.

Decrease Flow. Scan the vertices other than s and t in reverse topological order. When scanning a vertex v, balance v if it is unbalanced and blocked.

With an appropriate implementation, the wave algorithm will find a blocking flow in $O(n^2)$ time and a maximum flow in $O(n^3)$ time.

Shiloach and Vishkin [19] have proposed a variant of Karzanov's algorithm that is more complicated than the wave method but suited for parallel implementation. Using $k \leq n$ processors, it will find a maximum flow in $O((n^3 \log n)/k)$ time. With one processor its time bound is the same as that of the wave method.

Malhotra, Kumar, and Maheshwari [15] suggested another $O(n^2)$-time blocking flow method that is conceptually very simple. Initially we delete from G every vertex and edge not on a path from s to t. We maintain for each vertex v the *potential throughput* of v, defined by

$$\textbf{thruput}(v) = \min\left\{ \sum_{[u,v]\in E} c(u, v) - f(u, v), \sum_{[v,w]\in E} c(v, w) - f(v, w) \right\}.$$

(To define **thruput**(s) and **thruput**(t), we assume the existence of a dummy edge of infinite capacity from t to s.) To find a blocking flow we repeat the following step until t is not reachable from s:

Saturating Step. Let v be a vertex of minimum potential throughput. Send **thruput**(v) units of flow forward from v to t and backward from v to s by scanning the vertices in topological and reverse topological order. Update all throughputs, delete all newly saturated edges, and delete all vertices and edges not on a path from s to t.

Although this method is simple, it has two drawbacks. When actually implemented it is at least as complicated as the wave method, because of the necessity to maintain

potential throughputs for each vertex. Furthermore it preferentially sends flow through narrow bottlenecks, which may cause it to perform many more augmentations than necessary.

For dense graphs, the $O(n^2)$-time blocking flow algorithms are the fastest known. For sparse graphs, there are faster algorithms that obtain their speed by using sophisticated data structures. Cherkasky [2] discovered an $O(nm^{1/2})$-time method. Galil [10] improved the bound to $O((nm)^{2/3})$ and Galil and Naamad [11] to $O(m(\log n)^2)$. Shiloach [18] independently discovered the $O(m(\log n)^2)$-time method. The fastest known blocking flow algorithm for sufficiently sparse graphs, with a time bound of $O(m \log n)$, was discovered by Sleator and Tarjan [20, 21]. (For sufficiently dense graphs, Galil's $O((nm^{2/3})$-time method is faster by a factor of up to $O(\log n)$.)

The Sleator–Tarjan algorithm is an implementation of Dinic's original method using an appropriate data structure. The data structure allows us to maintain a collection of vertex-disjoint rooted trees, each of whose vertices has a real-valued capacity and at most one outgoing edge (so that the edges are directed toward the tree roots), under the following operations:

maketree(v): Create a new tree containing the single vertex v, of capacity zero.

findroot(v): Return the root of the tree containing vertex v.

findcap(v): Return the pair $[w, x]$, where x is the minimum capacity of a vertex on the tree path from v to **findroot**(v) and w is the vertex on this path of capacity x closest to the root.

addcap(v, x): Add x to the capacity of every vertex on the tree path from v to **findroot**(v).

link(v, w): Combine the two trees containing vertices v and w by adding the edge $[v, w]$ (v must be a root).

cut(v): divide the tree containing vertex v into two trees by deleting the edge out of v (v must not be a root).

We can use such a data structure to find a blocking flow. The forest consists of edges along which it may be possible to send additional flow. Specifically, we maintain for certain vertices v an outgoing current edge $[v, p(v)]$ with positive capacity. These edges form a collection of trees. The capacity of a vertex v is $c(v, p(v))$ if v is not a tree root, **huge** if v is a tree root, where **huge** is a constant chosen larger than the sum of all the edge capacities. The following steps reformulate Dinic's blocking flow algorithm using the five tree operations. We find a blocking flow by first executing **maketree**(v) followed by **addcap**(v, **huge**) for all vertices, then going to *advance* and proceeding as directed.

Advance. Let $v =$ **findroot**(s). If there is no edge out of v, go to *retreat*. Otherwise, let $[v, w]$ be an edge out of v. Perform **addcap**(v, $c(v, w)$-**huge**) followed by **link**(v, w). Define $p(v)$ to be w. If $w \neq t$, repeat *advance*; if $w = t$, go to *augment*.

Augment. Let $[v, \Delta] =$ **findcap**(s). Perform **addcap**($s, -\Delta$). Go to *delete*.

Delete. Perform **cut**(v) followed by **addcap**(v, **huge**). Define $f(v, p(v)) = c(v, p(v))$. Delete $[v, p(v)]$ from the graph. Let $[v, \Delta] =$ **findcap**(s). If $\Delta = 0$, repeat *delete*; if $\Delta = 0$, go to *advance*.

Retreat. If $v = s$, halt. Otherwise, for every edge $[u, v]$, delete $[u, v]$ from the graph and, if $p(u) \neq v$, define $f(u, v) = 0$; if $p(u) = v$, perform **cut**(u), let $[u, \Delta] =$ **findcap**(u), perform **addcap**(u, **huge**-Δ), and define $f(u, v) = c(u, v) - \Delta$. After deleting all edges $[u, v]$, go to *advance*.

Once the algorithm halts, we can use **cut**, **findcap**, and **addcap** as in *retreat* to find the flow f on every remaining edge. With this method, computing a blocking flow takes $O(m)$ tree operations.

The most direct way to implement the tree operations is to store with each vertex v its parent and its capacity. With such a representation each tree operation takes $O(n)$ time and the time to find a blocking flow is $O(nm)$. This method is essentially a reinterpretation of Dinic's algorithm.

Sleator and Tarjan proposed a more sophisticated way of implementing the tree operations that gives a time bound of $O(\log n)$ per tree operation, for a total time of $O(m \log n)$ to find a blocking flow and $O(nm \log n)$ to find a maximum flow. On sparse graphs, this algorithm is asymptotically the fastest known. Details can be found in [20, 21, 22, 24].

4. New directions

The ultimate maximum flow algorithm has yet to be discovered. Recently Gabow [9] proposed an algorithm that combines a scaling technique with Dinic's algorithm. The idea of scaling was first applied to network flow problems by Edmonds and Karp [7], who devised an $O(m^2 \log n \log c)$-time algorithm for the minimum-cost flow problem, assuming integer edge capacities. Gabow's method also requires integer edge capacities and has a time bound of $O(nm \log c)$, where c is the maximum edge capacity. Thus if the edge capacities are of moderate size, the algorithm is competitive in speed with that of Sleator and Tarjan.

Gabow's algorithm is quite simple to describe. Given a graph G with integer capacity function $c(v, w)$, we define a reduced capacity function $c'(v, w) = \lfloor c(v, w)/2 \rfloor$, find a maximum flow on G with respect to the reduced capacities by applying the algorithm recursively, and double the flow. The result is a legal flow for G with respect to the original capacities, with the additional property that any augmenting path for f has residual capacity at most one. This means that $|f|$ is within m units of the value of a maximum flow, and to make f maximum, we need only perform at most m augmentations. If we apply Dinic's algorithm to the residual capacity graph, the total time needed to increase f to a maximum flow is $O(nm)$. An overall time bound of $O(nm \log c)$ for the algorithm follows.

There are both theoretical and practical questions one may ask about the various network flow algorithms. On the theoretical side, we may ask whether the Sleator–Tarjan data structure can be used in a Karzanov-type algorithm to give a method as fast as any on both sparse and dense graphs. We conjecture that the answer is yes and that an $O(m \log n^2/m)$-time blocking flow algorithm can be obtained in this way. We leave this as an open problem. Any more substantial improvement in maximum flow algorithms will probably require major new ideas, such as how to combine the various blocking flow calculations of Dinic's method.

On the practical side, we would like to know which algorithms perform best on actual computers. Experiments are needed to answer this question. The results are likely to depend strongly on the detailed implementation of the algorithms and on the size, density, and structure of the problem graphs. We make no conjecture as to the best practical method, except to observe that as problems become larger and larger in size, data structures as sophisticated as that of Sleator and Tarjan will become important in practice as well as in theory. Experimental studies of most of the older algorithms can be found in [3, 12, 13].

Acknowledgement

My thanks to an anonymous referee for providing several references and proposing the maximum-capacity augmentation algorithm described in the appendix.

Appendix: A fast algorithm for maximum-capacity augmentation

Suppose we wish to carry out Ford and Fulkerson's augmenting path algorithm using Edmonds and Karp's rule of always selecting an augmenting path of maximum residual capacity. We can find augmenting paths efficiently as follows. In a preprocessing step, we sort all the edges by capacity, from largest to smallest. To find a single augmenting path, we number the edges from one to m in order by residual capacity, breaking ties arbitrarily. Then we use Dial's implementation [4] of Dijkstra's shortest path algorithm [5], modified to find a bottleneck shortest path (i.e. a path that minimizes the maximum edge length instead of minimizing the total edge length). As edge lengths, we use the computed edge numbers. Since these are all integers between 1 and m, Dial's algorithm runs in $O(m)$ time (and $O(m)$ space). The bottleneck shortest path is a maximum-capacity augmenting path.

After the augmentation, we must rearrange the edges in decreasing order by residual capacity. The augmentation changes the residual capacities only of edges along the augmenting path and their reversals, and all these capacities change by the same amount (negative for edges along the path, positive for their reversals). Thus the reordering can be performed by merging three sorted lists (of the edges

on the path, the edges whose reversals are on the path, and the remaining edges). This merging takes $O(m)$ time.

We conclude that the time for finding a maximum flow is $O(m \log m)$ plus $O(m)$ per augmentation, for a total of $O(m^2 \log c)$ time, since by the analysis of Edmonds and Karp the number of augmentations is $O(m \log c)$.

References

[1] A.V. Aho, J.E. Hopcroft and J.D. Ullman, *The design and analysis of computer algorithms* (Addison-Wesley, Reading, MA, 1974).

[2] R.V. Cherkasky, "Algorithm of construction of maximal flow in networks with complexity of $O(V^2\sqrt{E}$ operations" (in Russian), *Mathematical Methods of Solution of Economical Problems* 7 (1977) 112–125.

[3] T.-Y. Cheung, "Computational comparison of eight methods for the maximum flow problem", *ACM Transactions on Mathematical Software* 6 (1980) 1–16.

[4] R.B. Dial, "Algorithm 360: Shortest path forest with topological ordering", *Communications of the ACM* 12 (1969) 632–633.

[5] E.W. Dijkstra, "A note on two problems in connexion with graphs", *Numerische Mathematik* 1 (1959) 269–271.

[6] E.A. Dinic, "Algorithm for solution of a problem of maximum flow in a network with power estimation", *Soviet Mathematics Doklady* 11 (1970) 1277–1280.

[7] J. Edmonds and R.M. Karp, "Theoretical improvements in algorithmic efficiency for network flow problems", *Journal of the ACM* 19 (1972) 248–264.

[8] L.R. Ford and D.R. Fulkerson, *Flows in networks* (Princeton University Press, Princeton, NJ, 1962).

[9] H.N. Gabow, "Scaling algorithms for network problems", *Proceedings of the 24th Annual Symposium on Foundations of Computer Science* (1983) 248–258.

[10] Z. Galil, "An $O(V^{5/3}E^{2/3})$ algorithm for the maximal flow problem", *Acta Informatica* 14 (1980) 221–242.

[11] Z. Galil and A. Naamad, "An $O(EV \log^2 V)$ algorithm for the maximal flow problem", *Journal of Computer and System Sciences* 21 (1980) 203–217.

[12] F. Glover, D. Klingman, J. Mote and D. Whitman, "Comprehensive computer evaluation and enhancement of maximum flow algorithms", MS/IS Report 79-1, Graduate School of Business Administration, University of Colorado, Boulder, CO, 1979, extended abstract in *Discrete Applied Mathematics* 2 (1980) 251–254.

[13] H. Hamacher, "Numerical investigations on the maximal flow algorithm of Karzanov", *Computing* 22 (1979) 17–29.

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

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

[16] C.H. Papadimitriou and K. Steiglitz, *Combinatorial optimization: Algorithms and complexity* (Prentice-Hall, Englewood Cliffs, NJ, 1982).

[17] M. Queyranne, "Theoretical efficiency of the algorithm 'capacity' for the maximum flow problem", *Mathematics of Operations Research* 5 (1980) 258–266.

[18] Y. Shiloach, "An $O(n \cdot I \log^2 I)$ maximum-flow algorithm", Technical Report STAN-CS-78-802, Computer Science Department, Stanford University (Stanford, CA, 1978).

[19] Y. Shiloach and U. Vishkin, "An $O(n^2 \log n)$ parallel max-flow algorithm", *Journal of Algorithms* 3 (1982) 128–146.

[20] D.D. Sleator, "An $O(nm \log n)$ algorithm for maximum network flow", Technical Report STAN-CS-80-831, Computer Science Department, Stanford University (Stanford, CA, 1980).