A SIMPLE VERSION OF KARZANOV'S BLOCKING FLOW ALGORITHM

Robert Endre TARJAN

AT&T Bell Laboratories, Murray Hill, New Jersey 07974, USA

Received March 1983
Revised August 1983

Dinic has shown that the classic maximum flow problem on a graph of n vertices and m edges can be reduced to a sequence of at most $n-1$ so-called 'blocking flow' problems on acyclic graphs. For dense graphs, the best time bound known for the blocking flow problem is $O(n^2)$. Karzanov devised the first $O(n^2)$-time blocking flow algorithm, which unfortunately is rather complicated. Later Malhotra, Kumar and Maheshwari devised another $O(n^2)$-time algorithm, which is conceptually very simple but has some other drawbacks. In this paper we propose a simplification of Karzanov's algorithm that is easier to implement than Malhotra, Kumar and Maheshwari's method.

Maximum flow problem, graph algorithm

We assume some familiarity with algorithms for finding maximum network flows [3,8]. A *network* is a directed graph G with two distinguished vertices, a *source* s and a *sink* t, and a positive capacity $c(v, w)$ on every edge $[v, w]$. We denote by V the vertex set of G, by n the number of vertices, and by m the number of edges. We assume s is reachable from t; thus $n \geq 2$ and $m \geq 1$. (These inequalities simplify our statement of time bounds.) A *flow* f in G is a function on the edges such that the following two properties hold, where for convenience we define $c(v, w) = f(v, w) = 0$ if $[v, w]$ is not an edge:

(i) *Capacity constraint.* For any vertex pair v, w, $0 \leq f(v, w) \leq c(v, w)$.

(ii) *Flow conservation.* For any vertex $v \notin \{s, t\}$, $f_{\text{in}}(v) = f_{\text{out}}(v)$, where $f_{\text{in}}(v) = \sum_u f(u, v)$ and $f_{\text{out}}(v) = \sum_w f(v, w)$.

An edge $[v, w]$ is *saturated* if $f(v, w) = c(v, w)$ and *unsaturated* otherwise; since we assume positive edge capacities any saturated edge has positive flow. A flow f is *maximum* if the net flow out of the source, $f_{\text{out}}(s) - f_{\text{in}}(s)$, is maximum. A flow is *blocking* if there is a saturated edge on every path from s to t. The value of a blocking flow cannot be increased just by increasing the flow on some edges, but it may be possible to increase the flow value by rerouting, i.e. by decreasing the flow on some edges and increasing it on others.

Dinic [2] gave a way of finding a maximum flow in a network by successfully binding blocking flows in at most $n-1$ acyclic networks. He also gave an $O(nm)$-time algorithm for finding a blocking flow, thus obtaining an $O(n^2m)$-time maximum flow algorithm. Other blocking flow algorithms were subsequently discovered by Karzanov [6] ($O(n^2)$), Cherkasky [1] ($O(nm^{1/2})$), Malhotra et al. [9] ($O(n^2)$), Galil [4] ($O((nm)^{2/3})$), Galil and Naamad [5] ($O(m(\log n)^2)$), and Sleator and Tarjan [11,12] ($O(m \log n)$); each gives a bound greater by a factor of n for the maximum flow problem.

On dense graphs ($m = \Theta(n^2)$) the algorithms of Karzanov, Cherkasky, Malhotra et al., and Calil all run in $O(n^2)$ time. Of these algorithms, that of Malhotra et al. is by far the simplest conceptually. Our purpose in this paper is to give a version of Karzanov's algorithm that is not much more complicated conceptually than the algorithm of Malhotra et al. and is at least as easy to implement. We call our method the *wave algorithm.*

Suppose we wish to find a blocking flow on an acyclic network G. Intuitively, the wave method proceeds in alternating forward and backward passes over the graph. During a forward pass, we push as much flow as possible as far as possible

0167-6377/84/$3.00

through the graph. During a backward pass, we back up flow that is blocked from reaching the source, so that the next forward pass will push it forward along alternative paths.

To proceed more formally, we need some definitions. A *preflow* f is a function on the edges of G that satisfies the capacity constraint and has $f_{in}(v) \geq f_{out}(v)$ for every vertex $v \notin \{s, t\}$. A vertex v is *balanced* if $f_{in}(v) = f_{out}(v)$ and *unbalanced* otherwise. A preflow f is *blocking* if it saturates an edge on every path from s to t. The wave algorithm finds a blocking preflow and gradually converts it into a blocking flow by balancing vertices.

Each vertex is in one of two states: *unblocked* or *blocked*. Initially s is blocked and every other vertex is unblocked. 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 attempt to balance an unblocked vertex v by repeating the following step until $f_{in}(v) = f_{out}(v)$ (v is now balanced) or $f_{in}(v) > f_{out}(v)$ but there is no unsaturated edge $[v, w]$ such that w is unblocked (the balancing attempt fails):

Increasing Step. Let $[v, w[$ be an unsaturated edge such that w is unblocked. Increase $f(v, w)$ by $\min\{c(v, w) - f(v, w), f_{in}(v) - f_{out}(v)\}$.

We attempt to balance a blocked vertex v by repeating the following step until $f_{in}(v) = f_{out}(v)$ (v is now balanced):

Decreasing Step. Let $[u, v]$ be an edge of positive flow. Decrease $f(u, v)$ by $\min\{f(u, v), f_{in}(v) - f_{out}(v)\}$.

Whereas an attempt to balance an unblocked vertex may fail, an attempt to balance a blocked vertex always succeeds, since if necessary we decrease the flow on each incoming edge to zero.

To find a blocking flow, we begin with the preflow that saturates every edge $[s, v]$ and is zero on every other edge, and repeat the following steps:

Increase Flow. Scan the vertices other than s and t in topological order (a total order such that if $[v, w]$ is an edge, v is scanned before w; such an order exists since G is acyclic [7]). To scan a vertex v, if v is unblocked and unbalanced, attempt to balance it, and if the attempt fails make it blocked. After scanning all vertices go to *decrease flow* if there is a blocked, unbalanced vertex other than s; otherwise halt.

Decrease Flow. Scan the vertices other than s and t in reverse topological order. To scan a vertex v, if v is blocked and unbalanced, attempt to balance it. After scanning all vertices, go to *increase flow* if there is an unblocked, unbalanced vertex other than t; otherwise halt.

This algorithm maintains the invariant that if v is blocked, there is a saturated edge on every path from v to t. Since s is blocked initially, every preflow constructed by the algorithm is blocking. Because the scanning during a *decrease flow* step is in reverse topological order, each vertex blocked before the step is balanced after the step and remains balanced during the subsequent *increase flow* step, if any. After an *increase flow* step, there are no unbalanced, unblocked vertices. It follows that if the method halts, all vertices are balanced, and it halts with a blocking flow.

At the beginning of a *decrease flow* step, the only unbalanced vertices are those that became blocked during the previous *increase flow* step. Thus every *increase flow* step except the last must block at least one vertex, and there are at most $n - 1$ iterations of the two steps (s is blocked initially; t is never blocked). This means that there are at most $(n - 2)(n - 1)$ attempts to balance a vertex.

An edge $[v, w]$ has its flow increased only if w is unblocked and decreased only if w is blocked; thus the flow in $[v, w]$ first increases, then decreases. Each increasing step either saturates an edge or terminates an attempt to balance a vertex. Similarly, each decreasing step either decreases the flow in an edge to zero or terminates an attempt to balance a vertex. Thus there are at most $2m + (n - 2)(n - 1)$ increasing and decreasing steps.

These bounds on the number of steps mean that an appropriate implementation of the wave algorithm will run in $O(n^2)$ time. We topologically order the vertices using either of the $O(n + m)$-time methods known [7,13]. For each vertex v, we maintain the value $excess(v) = f_{in}(v) - f_{out}(v)$ and a bit indicating whether v is unblocked or blocked. If v is unblocked we maintain a current edge pointer to an edge on a list of edges out of v; initially this pointer indicates the first edge on the list. If v is blocked we maintain a similar pointer to an edge on a list of edges into v. To attempt to balance an unblocked vertex v, we examine successive edges $[v, w]$, starting from the current edge, until finding one to which the increasing step applies; we make

the current edge pointer point to this edge, increase flow on the edge, and update *excess*(v) and *excess*(w). We continue examining edges and increasing the flow on appropriate ones until *excess*(v) = 0 (v is balanced) or the current edge pointer runs off the end of the list (the balance attempt fails). Attempting to balance a blocked vertex is similar. With this implementation each attempt to balance a vertex takes O(1) time plus O(1) time per increasing or decreasing step, and the total time is $O(n^2)$ including the time for topological ordering and repeatedly scanning all the vertices, whether or not they are balanced.

Dinic's maximum flow algorithm implemented using the wave algorithm to find blocking flows runs in $O(n^3)$ time. In this application of the wave algorithm we can use the special structure of the successive networks on which blocking flows must be found to reduce the time to find each blocking flow to $O(n + m + k)$, where k is the number of attempts to balance a vertex, eliminating the $O(n^2)$ overhead for scanning balanced vertices. Although the worst-case bound remains $O(n^2)$, this change may speed up the algorithm in practice.

To obtain this efficiency, we use the fact that each network needing a blocking flow is not only acyclic but *layered:* for each edge $[v, w]$, *level*(w) = *level*(v) + 1, where *level*(v) is the number of edges on a path of fewest edges from s to v. To find a blocking flow, we first compute the level of each vertex. (This computation, which, using breadth-first search, takes $O(n + m)$ time is in fact needed to generate the layered network in the first place [2,3].) Then we proceed with the preflow algorithm using two stacks U and B to hold unbalanced vertices. Stack U contains unblocked, unbalanced vertices in increasing order by level, top to bottom, and Stack B contains blocked, unbalanced vertices in decreasing order by level. Ordering by level gives a topological order.

Just before an increasing step, B is empty and U contains all unblocked, unbalanced vertices except t. To carry out the step, we construct a 'working set' S by removing from the top of U all vertices of lowest level and placing them in S. Then we remove vertices from S one at a time and attempt to balance them. If a vertex w becomes unbalanced while balancing v, we push w onto U (w must be unblocked and of level one greater than v). If an attempt to balance a vertex v fails, we make v blocked and push it onto B. When S is empty, we replenish it from the top of U with all unblocked, unbalanced vertices of the current lowest level. We continue in this way until both S and U are empty. We carry out a decreasing step similarly, using B and S to hold blocked, unbalanced vertices and pushing newly unbalanced, unblocked vertices onto U. This implementation has the claimed $O(n + m + k)$ running time.

We conclude this paper by comparing our blocking flow algorithm with the algorithms of Karzanov and Malhotra, et al. Karzanov's algorithm and other variants of it that have been proposed [1,4,10] are very similar to our method but require storing a stack of incoming flow increments at each vertex and decreasing the incoming flow by reducing the increments in last-in, first-out order. Our method eliminates the need for these stacks by making flow increments and decrements in a different order. Karzanov's original method takes $O(n^2)$ space as well as $O(n^2)$ time, although this space bound can be reduced to $O(m)$ [1,4]. Our algorithm has an $O(n + m)$ space bound; it also treats unblocked and blocked vertices and flow increases and decreases more symmetrically than does Karzanov's method. (In this respect it resembles the algorithm of Shiloach and Vishkin [10], which was designed for parallel implementation and retains the stacks of flow increments.)

Though conceptually simple, the algorithm of Malhotra et al., is more complicated to implement than ours: it needs two numbers and two edge pointers per vertex, whereas our method needs only one bit, one number, and one pointer per vertex. Furthermore, their algorithm preferentially sends flow through vertices with small potential throughput, which may cause the method to run slowly in practice. Our method does not share this drawback.

On sparse graphs, the fastest known blocking flow algorithm is that of Sleator and Tarjan [11,12], which runs in $(m \log n)$ time. We conjecture that Sleator and Tarjan's data structures can be combined with the wave method described here to obtain an $O(m \log(n^2/m))$-time blocking flow algorithm. This time bound is as small as that of any known algorithm, whatever the values of n and m. We leave the construction of such an algorithm as a research problem.

References

[1] 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, 112–125 (1977).

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

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

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

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

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

[7] D.E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, Second Edition, Addison-Wesley, Reading, MA, 1973.

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

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

[10] Y. Shiloach and U. Vishkin, "An $O(n^2\log n)$ parallel max-flow algorithm", *J. Algorithms* 3, 128–46 (1982).

[11] 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.

[12] D.D. Sleator and R.E. Tarjan, "A data structure for dynamic trees", *J. Computer and System Sciences*, to appear; see also *Proc. Thirteenth Annual ACM Symp. on Theory of Computing* (1981), pp. 114–122.

[13] R.E. Tarjan, "Finding dominators in directed graphs", *SIAM J. Comput.* 3, 62–89 (1974).