The preflow algorithm for the maximum flow problem

Karzanov's max-flow algorithm of 1974 given in [1] is based on a concept of *preflow* introduced there, which is a function on the arcs that may violate, in a certain way, the flow conservation condition in nonterminal nodes. The algorithm, called the *preflow algorithm (method)*, takes advantages of handling preflows on intermediate iterations (rather than handling flows, as in the previous algorithms), due to which the running time of the algorithm reduces to $O(n^3)$ (compared with $O(mn^2)$ for Dinitz' algorithm [2]); hereinafter n and m are the numbers of nodes and arcs in the input digraph $G = (V(G), A(G))$. Subsequently preflows and the idea of push operations have been widely used in other max-flow algorithms (having the same or smaller running time), in particular, in Cherkassky's algorithm [3] (which is slightly faster) and in Goldberg's push-relabel algorithm [4] of the same complexity $O(n^3)$.

Similar to Dinitz's algorithm, the preflow algorithm consists of $O(n)$ *stages* (big iterations), each solving the following *auxiliary problem*: find a *blocking flow* in a *layered* network (i.e. a network where all paths from the source to the sink have the same length; such a network is formed by the nodes and arcs contained in shortest source-to-sink paths of the residual network w.r.t. the current flow). The preflow algorithm solves the auxiliary problem in $O(n^2)$ time, thus yielding the time bound $O(n^3)$ for the whole algorithm.

In fact, the algorithm of finding a blocking flow given in [1] can be slightly modified so as to work with an arbitrary acyclic, not necessarily layered, network, and below we give a description just for this more general situation.

We start with specifying definitions and settings. Consider a network $N = (G, s, t, c)$, where $G = (V, A)$ is a directed graph, s and t are two distinguished nodes in G, the *source* and the *sink*, respectively, and $c : A \to \mathbb{R}_+$ is a nonnegative real function of arc *capacities*. For a function $f : A \to \mathbb{R}_+$, the *excess* of f at a node $v \in V$ is defined to be $\mathrm{ex}_f(v) := \sum_{a\in\delta^{\rm in}(v)} f(a) - \sum_{a\in\delta^{\rm out}(v)} f(a)$ (where $\delta^{\rm in}(v)$ ($\delta^{\rm out}(v)$) is the set of arcs in G entering (resp. leaving) a node v). Then f is a *flow* from s to t if $f \le c$ (i.e. $f(a) \le c(a)$ for each $a \in A$), and f satisfies $\mathrm{ex}_f(t) \ge 0$ and $\mathrm{ex}_f(v) = 0$ for all $v \in V - \{s, t\}$. We say that f is a *preflow* in N if $f \le c$ and $\mathrm{ex}_f(v) \ge 0$ for all $v \in V - \{s\}$. A flow or preflow f is called *blocking* if any (directed) path from s to t contains at least one *saturated* arc a, i.e. such that $f(a) = c(a)$.

Theorem. *Let $N = (G = (V, A), s, t, c)$ be an acyclic network. A blocking flow f from s to t in N can be found in $O(n^2)$ time.*

Proof. First of all we order the nodes of G topologically, i.e. label them as $v_1, \dots, v_n$ so that $(v_i, v_j) \in A$ imply $i < j$ (this takes $O(m)$ time). One may assume that $s = v_1$, $t = v_n$, and each arc lies on an s–t path and has nonzero capacity.

The algorithm iteratively handles a preflow $f : A \to \mathbb{R}_+$. For each node $v \in V$, the following data are explicitly maintained:

(i) The excess $\mathrm{ex}(v) = \mathrm{ex}_f(v)$.

(ii) A (double-linked) *list* $\mathrm{Out}(v)$. It is formed by the arcs of G leaving v (each arc occurs in the list exactly once). Each arc e can be either *scanned* or *unscanned.* If e is unscanned, then $f(e) = 0$. One arc in this list is distinguished, called *active* and denoted by $\widetilde{e}_v$. The following condition holds:

(C1) all arcs of $\mathrm{Out}(v)$ before $\widetilde{e}_v$ are scanned, while all arcs after $\widetilde{e}_v$ are unscanned (the arc $\widetilde{e}_v$ itself may be either scanned or not).

Also some arcs in $\mathrm{Out}(v)$ can be labeled as "frozen" (the meaning will be clear later).

(iii) A *stack* $\mathrm{In}(v)$ (to work with on the "last come first serve" basis). Its elements are pairs (e, Δ), where e is an arc entering v and Δ is a nonnegative real. Each arc e entering v may occur in this stack once or several times or it may not occur there at all, and the sum of numbers Δ over the pairs (e, Δ) with the same e is equal to $f(e)$. In particular, if e does not occur in $\mathrm{In}(v)$, then $f(e) = 0$.

The algorithm starts with the function f such that $f(e) = c(e)$ for all arcs e leaving the source s, and $f(e) = 0$ otherwise (in the latter case e is unscanned). Accordingly, for each arc $e = (s, v)$, the pair $(e, f(e))$ is inserted (as a unique element) in the stack $\mathrm{In}(v)$. The initial stacks $\mathrm{In}(v')$ for the remaining nodes v' are empty. Clearly f is a blocking preflow.

The algorithm alternates "pushing" and "balancing" iterations. Although the first iteration is "pushing", it is more convenient for us (and more enlightening) to start with describing a "balancing" iteration.

Balancing. We assume that at the moment of beginning a "balancing" iteration, the following condition holds:

(C2) f is a blocking preflow; moreover, for each node v with $\mathrm{ex}_f(v) > 0$, any v–t path contains a saturated arc e.

Using the topological order on V, we find the node $v = v_i \neq t$ such that $\mathrm{ex}_f(v) > 0$ and $\mathrm{ex}_f(v_j) = 0$ for $j = i+1, \ldots, n-1$. If such a node does not exist, then f is already a blocking flow, and the algorithm terminates.

We perform "balancing" at this v so as to reduce the excess at it to zero. To do so, we use the stack $\mathrm{In}(v)$ and decrease the numbers Δ there step by step in a natural way. More precisely, take the last (chronologically) member (e, Δ) of $\mathrm{In}(v)$ and let $\delta := \min\{\Delta, \mathrm{ex}_f(v)\}$. Update $f(e) := f(e) - \delta$, $\Delta(e) := \Delta(e) - \delta$, and $\mathrm{ex}(v) := \mathrm{ex}(v) - \delta$. If the new $\Delta(e)$ becomes 0, we handle the previous pair (e', Δ) in $\mathrm{In}(v)$ is a similar way (whenever $\mathrm{ex}(v)$ is still nonzero), and so on. Eventually, we obtain f with $\mathrm{ex}_f(v) = 0$.

All arcs of G entering v are labeled as "frozen" (which means that the function f on such arcs should not be changed on subsequent iterations).

Pushing. Note that after performing a balancing iteration, the current function f is a preflow and, moreover, a blocking preflow (which can be easily checked), but the second condition in (C2) need not hold. A "pushing" iteration increases f on certain arcs and restores validity of (C2). Recall that the first iteration in the algorithm is "pushing" as well.

We scan the nodes in the increasing order, starting with v_2 (where $v_1 = s$). Every time we meet a node $v = v_i$ with $\mathrm{ex}_f(v) > 0$ (for a current f), we try to reduce the excess at v as much as possible by increasing f on appropriate arcs leaving v. (Note that a growth of f at an arc (v, u) increases the excess at u (which may be zero before). However, since $u = v_j$ for some $j > i$, the node u will be scanned on a subsequent step of this iteration.)

More precisely, we take the active arc $\widetilde{e}_v$ and do the following:

(P) starting from $\widetilde{e}_v$, we scan step by step the (unscanned) arcs in $\mathrm{Out}(v)$ skipping the "frozen" arcs (if they exist); when scanning a current arc $e = (v, u)$ (which is made "active" at this moment), we increase $f(e)$ as much as possible, i.e. letting $\delta := \min\{c(e) - f(e),\ \mathrm{ex}_f(v)\}$, we update $f(e) := f(e) + \delta$ and $\mathrm{ex}(v) := \mathrm{ex}(v) - \delta$, and accordingly insert the pair (e, δ) in the stack $\mathrm{In}(u)$.

We stop as soon as either the list $\mathrm{Out}(v)$ terminates, or the excess at v becomes zero. In the former case, all arcs in $\mathrm{Out}(v)$ are saturated or "frozen" (and the active arc is formally the last arc), and in the latter case, the current arc $e = (v, u)$ becomes active, the arcs before e are saturated or "frozen", and the arcs after e remain unscanned.

After scanning $v = v_i$, we repeat the procedure with the next vertex $v' = v_j$ $(j > i)$ where the excess w.r.t. the current f is positive. And so on (until we reach the sink t). One can see that the number of operations during a "pushing" iteration is $O(n + q)$, where q is the number of (new) arcs that become saturated at the iteration plus the number of "frozen" arcs skipped when scanning the lists $\mathrm{Out}(v)$.

The following fact is easy.

Lemma 1. *After performing a "pushing" iteration, the obtained f satisfies (C2).*

This implies that the next "balancing" iteration has a correct input, and the whole process of alternating "pushing" and "balancing" iterations is well-defined. The key observation is as follows.

Lemma 2. *Suppose a node $v = v_i$ is handled at some "balancing" iteration. Then for any arc e incident to v, the value $f(e)$ is not changed on subsequent iterations.*

Proof. This relies on the fact that at the moment of balancing f at v, one has $\mathrm{ex}_f(v_j) = 0$ for $j = i + 1, \ldots, n - 1$. Analyzing the algorithm and using this fact, one can realize that for any subsequent function f' and for any arc e' incident to v_j, the value $f'(e')$ cannot be less than $f(e)$. In particular, $f'(e) \geq f(e)$ for each e leaving v.

On the other hand, since all arcs entering v becomes "frozen", the value of a subsequent function f' on an arc e cannot be greater than $f(e)$.

This implies that f preserves on the arcs incident to v (in view of $\mathrm{ex}_f(v) = 0$). ▮

As a result, any node can be balanced at most once, implying that the number of iterations is $O(n)$. Also if an arc $e = (u, v)$ becomes saturated, the number of subsequent operations involving e is $O(1)$ (a possible operation is a decrease of $f(e)$ during balancing v, after which e becomes "frozen" and it can be scanned once during pushing from u).

Thus, using the above-mentioned bound $O(n + q)$ for one "pushing" iteration, we can conclude that to perform all "pushing" iterations takes $O(n^2 + m)$ time. A similar bound is valid for all "balancing" iterations taken together.

Thus, the algorithm runs in $O(n^2 + m)$ time, yielding the theorem. ▮▮

References

[1] A.V. Karzanov, Determining a maximal flow in a network by the method of preflows, *Doklady Akademii Nauk SSSR*, **215**, 1974, 49–52, in Russian. (English translation in *Soviet Math. Dokl.*, **15**, No.2, 1974, 434–437.)

[2] E.A. Dinic, Algorithm for solution of a problem of maximum flow with power estimation, *Doklady Akademii Nauk SSSR*, **194**, 1970, 754–757, in Russian. (English translation in *Soviet Math. Dokl.*, **11**, 1970, 1277–1280.)

[3] B.V. Cherkassky, Algorithm for construction of maximal flow in networks with complexity of $(O(n^2\sqrt{p})$ operations, In: *Mathematical Methods in Economical Research, issue 7*, Nauka, Moscow, 1977, pp. 117–126.

[4] A.V. Goldberg, A new max-flow algorithm, *Technical Report* MIT/LCS/TM-291, Cambridge, 1985.