Algorithmica (2002) 32: 467–473
DOI: 10.1007/s00453-001-0082-y

Algorithmica

Routing Flow Through a Strongly Connected Graph

T. Erlebach[1] and T. Hagerup[2]

Abstract. It is shown that, for every strongly connected network in which every edge has capacity at least Δ, linear time suffices to send flow from source vertices, each with a given supply, to sink vertices, each with a given demand, provided that the total supply equals the total demand and is bounded by Δ. This problem arises in a maximum-flow algorithm of Goldberg and Rao, the binary blocking flow algorithm.

Key Words. Analysis of algorithms, Network flow, Maximum-flow problem, Feasible-flow problem, Strongly connected graph, Depth-first search.

1. Introduction. A *network* is given by a directed graph (V, E) together with a function $c\colon E \to \mathbb{R}_+$ that maps each edge in E to a positive *capacity* and a function $b\colon V \to \mathbb{R}$ that maps each vertex in V to an *import* at v; we denote the network succinctly by the tuple (V, E, c, b). A *pseudoflow* in a network $\mathcal{N} = (V, E, c, b)$ is a function $f\colon E \to \mathbb{R}$ that satisfies $0 \le f(e) \le c(e)$ (the *capacity constraint*) for all $e \in E$. For every edge $e \in E$, $f(e)$ is conveniently thought of as the rate at which a certain commodity flows through e, the capacity of e being the maximum rate possible. The *excess* of a pseudoflow f in $\mathcal{N}$ at a vertex $v \in V$ is defined as

$$e_f(v) = b(v) + \sum_{u\colon (u,v)\in E} f(u, v) - \sum_{w\colon (v,w)\in E} f(v, w),$$

i.e., as the sum of the import at v and the net flow into v through all of its incident edges. An instance of the *feasible-flow problem* is given by a network $\mathcal{N} = (V, E, c, b)$, and the goal is to compute a *flow* in $\mathcal{N}$, i.e., a pseudoflow f in $\mathcal{N}$ that satisfies $e_f(v) = 0$ (the *conservation constraint*) for all $v \in V$. Informally, the feasible-flow problem is to route flow through the network from the vertices with positive imports to the vertices with negative imports.

Summing the conservation constraints for all vertices shows that a network $\mathcal{N} = (V, E, c, b)$ admits no flow unless the total import $\sum_{v\in V} b(v)$ is zero. From this point, we consider only networks with zero total import and strongly connected underlying graphs (V, E). Even such networks may not admit flows, but a flow always exists if the minimum edge capacity $C = \min_{e\in E} c(e)$ is at least as large as the total positive import $\Delta = \sum_{v\in V} \max\{b(v), 0\}$. Flows in networks with this property are quite easy to compute, the challenge being to compute them quickly. Goldberg and Rao [1] gave a linear-time algorithm for the case $C \ge 2\Delta$ and stated that flows in networks with n vertices, m edges,

[1] Computer Engineering and Networks Laboratory, Eidgenössische Technische Hochschule Zürich, CH-8092 Zürich, Switzerland. erlebach@tik.ee.ethz.ch.
[2] Institut für Informatik, Johann Wolfgang Goethe-Universität Frankfurt, D-60054 Frankfurt am Main, Germany. hagerup@ka.informatik.uni-frankfurt.de.

Received December 5, 1999; revised May 27, 2000. Communicated by K. Mehlhorn.
Online publication November 23, 2001.

and $C \geq \Delta$ can be computed in $O(m\alpha(m, n))$ time, where α is an "inverse Ackermann" function, by appealing to the wheels-within-wheels characterization of Knuth [3] and the union-find data structure analyzed by Tarjan [5]. We show the following result.

THEOREM 1. *For every network* $\mathcal{N} = (V, E, c, b)$ *such that* $\sum_{v \in V} b(v) = 0$, $\min_{e \in E} c(e) \geq \sum_{v \in V} \max\{b(v), 0\}$, *and* (V, E) *is strongly connected, a flow in* $\mathcal{N}$ *can be computed in* $O(|V| + |E|)$ *time.*

Our algorithm realizing Theorem 1 is based on depth-first search, and it is simple and fast, comparable in both respects with the algorithm of Goldberg and Rao mentioned above that requires capacities twice as large.

We sketch the relevance of Theorem 1 to a maximum-flow algorithm of Goldberg and Rao, the *binary blocking flow* or *BBF algorithm* [1]. The BBF algorithm maintains a flow that gradually evolves into a maximum flow and repeatedly derives from the current residual network an auxiliary network that, following [2], we call the *core*. The BBF algorithm subsequently contracts each strongly connected component of the core to a single vertex, computes a blocking flow in the resulting acyclic network, and translates this blocking flow to the original core to obtain a flow that is added to the current flow in the full network.

Translating the blocking flow from the contracted network to the core amounts to routing "through" each strongly connected component of the core; i.e., it reduces to solving a collection of instances of the feasible-flow problem in strongly connected networks. The BBF algorithm cancels sufficient flow in the contracted network to ensure that the condition $C \geq 2\Delta$ holds for each instance, with a corresponding detriment to the overall rate of progress. Our new result allows less flow to be canceled and may be used to speed up the BBF algorithm in terms of constant factors.

2. The Algorithm. In this section we prove Theorem 1. Let $\mathcal{N} = (V, E, c, b)$ be a network with the properties mentioned in Theorem 1 and take $n = |V|$, $m = |E|$, $G = (V, E)$, and $\Delta = \sum_{v \in V} \max\{b(v), 0\}$. For every edge $e = (x, y) \in E$, we call x and y the *tail* and the *head* of e, respectively, and write $x = \mathit{tail}(e)$ and $y = \mathit{head}(e)$. We first describe an inefficient algorithm for computing a flow in $\mathcal{N}$ and prove it correct. Subsequently we derive a second algorithm that computes the same output and works in linear time.

Our first algorithm begins by carrying out a depth-first search (DFS) of G that starts at an arbitrary vertex $r \in V$ and is similar to Tarjan's algorithm for computing the strongly connected components of a general directed graph [4]. This constructs a DFS tree $T = (V, E_T)$ of G rooted at r and defines the *preorder number* $\mathit{pre}(v)$ of each $v \in V$ by assigning the numbers $1, \ldots, n$ to the vertices in the order of their discovery. The actual output used by the subsequent processing is the inverse bijection $\mathit{pre}^{-1}\colon \{1, \ldots, n\} \to V$ with, e.g., $\mathit{pre}^{-1}(1) = r$. We frequently identify a vertex $v \in V$ with its preorder number $\mathit{pre}(v)$. The edges in E_T are called *tree edges*. For every $v \in V \backslash \{r\}$, the DFS also computes the parent $\mathit{parent}(v)$ of v in T, the edge $\mathit{parentedge}(v) = (\mathit{parent}(v), v)$, and an edge $\mathit{lowlink}(v)$ called the *lowlink* of v. For all $v \in V$, let T_v be the set of all descendants of v in T (including v itself). Since no confusion results, we will use "T_v"

also to denote the subtree of T induced by this set. For all $v \in V\backslash\{r\}$, *lowlink*(v) is an edge e with *tail*$(e) \in T_v$ that minimizes *head*(e) (i.e., minimizes *pre*(*head*(e))) over all edges with tails in T_v. The root r has no lowlink. For all $v \in V$, take

$$L(v) = \begin{cases} \textit{head}(\textit{lowlink}(v)), & \text{if} \quad v \neq r, \\ 0, & \text{if} \quad v = r. \end{cases}$$

As G is strongly connected, for every $v \in V\backslash\{r\}$ there is an edge leaving T_v; i.e., $L(v) < v$ for all $v \in V$.

The algorithm manipulates a pseudoflow f in $\mathcal{N}$ and begins by setting $f(e) = 0$ for all $e \in E$. For every $v \in V$, we call $\max\{e_f(v), 0\}$ the *supply* of v and $|\min\{e_f(v), 0\}|$ the *demand* of v. Vertices with positive supplies and positive demands are called *sources* and *sinks*, respectively. The basic operation used by the algorithm is increasing $f(e)$ by some quantity $q > 0$ for every edge e on a path p in G from a source u with supply at least q to a vertex v. We call this operation a *push* of value q from u to v along p. The push decreases $e_f(u)$ by q, increases $e_f(v)$ by q, and leaves $e_f(w)$ unchanged for all $w \in V\backslash\{u, v\}$. It does not change the total excess $\sum_{w \in V} e_f(w)$ and creates no new sinks (but may turn v into a source).

After the DFS, the vertices in V are processed in the order of decreasing preorder numbers. Vertices that are not sources are simply skipped. The processing of a source v gradually reduces the supply of v to zero in the following way: As long as v is still a source and the subtree T_v contains at least one sink, an arbitrary sink w in T_v is chosen and a push of value $\min\{e_f(v), -e_f(w)\}$ along the path in T_v from v to w is performed; each such push is called a *tree push*. Subsequently, if v is still a source and *lowlink*$(v) = (x, y)$, a single push of value $e_f(v)$ from v to y along the path in T_v from v to x followed by (x, y) is performed; this push is called a *lowlink push*. After the processing of the last vertex, r, the algorithm terminates and returns the function f.

After its processing, a vertex never again becomes a source. To see this, note that the only source possibly created during the processing of a vertex $v \in V\backslash\{r\}$ is $L(v)$, which, since $L(v) < v$, is processed after v. Because the total excess remains zero throughout, it follows that after the processing of the last vertex, f satisfies the conservation constraint at every vertex. What remains is to show that $f(e) \leq \Delta$ for every $e \in E$, so that f satisfies the capacity constraints as well.

We define the *lowlink path* of each vertex $v \in V$ by induction on the preorder number of v. The lowlink path of r is the empty path. The lowlink path of a vertex $v \neq r$ with lowlink (x, y) is the tree path from v to x, followed by (x, y), followed by the lowlink path of y (which is well defined because $y < v$).

LEMMA 1. *Every lowlink path is simple.*

PROOF. $L(y) < y = L(x)$ for every lowlink (x, y), and $L(x) = L(y)$ for every tree edge (x, y) on a lowlink path. Thus the L values of the vertices on every lowlink path form a nonincreasing sequence, and every lowlink induces a strict decrease. Since no cycle consists of tree edges only, no lowlink path contains a cycle. □

Lemma 1 makes it intuitively clear why $f(e)$ is bounded by the total positive import Δ for every $e \in E$: every "flow atom" follows a simple path from a source to a sink

and therefore cannot contribute more than once to $f(e)$. We provide a formal argument based on a potential function. Fix an arbitrary edge $e = (x, y) \in E$ and define

$$A = \begin{cases} V, & \text{if } e \text{ is a tree edge and } T_y \text{ contains at least one sink,} \\ \{v \in V \mid \text{the lowlink path of } v \text{ contains } e\}, & \text{otherwise.} \end{cases}$$

The set A may change over the course of the execution, but it never acquires new elements. Let $\Phi = \sum_{v \in A} \max\{e_f(v), 0\}$ be the sum of the supplies of the vertices in A.

LEMMA 2. *Φ never increases.*

PROOF. A tree push does not increase any supply and therefore cannot increase Φ. It is easy to see that, just before a lowlink push from v to w, if w belongs to A, then v also belongs to A. Thus a lowlink push cannot increase Φ either. □

LEMMA 3. *Every push that increases $f(e)$ by some value q decreases Φ by at least q.*

PROOF. When a tree push from v to w increases $f(e)$ by q, e is a tree edge and w is a sink in T_y. Therefore v belongs to A before the push, and Φ indeed decreases by at least q. Now consider a lowlink push from v to w that increases $f(e)$ by q. If e is a tree edge, there is no sink in T_y. Thus the second case in the definition of A applies. Since v clearly belongs to A just before the push, w cannot belong to A at that time, as this would cause e to appear twice on the lowlink path of v, contradicting Lemma 1. □

LEMMA 4. *At the end of the execution, $f(e) \leq \Delta$.*

PROOF. Because $f(e) = 0$ initially, Lemma 4 follows from Lemmas 2 and 3 and the fact that Φ is bounded by Δ initially and does not become negative. □

Since e was chosen arbitrarily, Lemma 4 concludes the proof that the function f returned by our first algorithm is a flow; i.e., the algorithm is correct. Due to the pushes along potentially long paths of tree edges, the algorithm is not efficient. The key to obtaining a more efficient algorithm is the simple observation that the flow on the tree edges can be deduced from the flow on the remaining edges. Indeed, let g be the pseudoflow in $\mathcal{N}$ that coincides with f, except that $g(e) = 0$ for every tree edge e. Every tree edge (u, v) is the only tree edge between T_v and the rest of G, and therefore

$$0 = \sum_{w \in T_v} e_f(w) = \sum_{w \in T_v} e_g(w) + f(u, v).$$

Given g, the quantity $E_g(v) = \sum_{w \in T_v} e_g(w)$ can be computed for all $v \in V$ in $O(n)$ time in a bottom-up pass over T, after which it suffices to set $f(u, v) = -E_g(v)$ for every tree edge (u, v). In order to compute g rather than f, we simply refrain from updating $f(e)$ for tree edges e. Then every push can be executed in constant time, and except for the initial DFS, the processing needs only $O(n)$ time.

Algorithm SC-ROUTE:

```
{ Phase 1 }
Choose any vertex r ∈ V;
DFS(r); { compute pre^{-1}, parent, parentedge, and lowlink }
for e ∈ E do f(e) ← 0 od;

{ Phase 2 }
for v ∈ V do E_f(v) ← b(v) od;
for i ← n downto 2 do
    v ← pre^{-1}(i);
    if E_f(v) > 0 then { a lowlink push from v }
        e ← lowlink(v);
        f(e) ← f(e) + E_f(v);
        E_f(head(e)) ← E_f(head(e)) + E_f(v)
    else
        E_f(parent(v)) ← E_f(parent(v)) + E_f(v)
    fi
od;

{ Phase 3 }
for v ∈ V do E_g(v) ← b(v) od;
for every lowlink e = (x, y) do
    E_g(x) ← E_g(x) − f(e);
    E_g(y) ← E_g(y) + f(e)
od;
for i ← n downto 2 do
    v ← pre^{-1}(i);
    f(parentedge(v)) ← −E_g(v);
    E_g(parent(v)) ← E_g(parent(v)) + E_g(v)
od
```

Fig. 1. Algorithm for routing flow through a strongly connected graph.

An efficient algorithm SC-ROUTE based on these considerations is shown in Figure 1. It consists of three phases. Phase 1 carries out the DFS and initializes the pseudoflow f. The main parts of Phases 2 and 3 are traversals of T that run in $O(n)$ time and compute, respectively, the flow on all lowlinks and the flow on all tree edges.

In Phase 2 the flow on the lowlinks is computed essentially as in our first algorithm, but without updates of flow on tree edges. For all $v \in V\backslash\{r\}$, the variable $E_f(v)$ is initialized to $b(v)$ and updated in a bottom-up summation that causes its value to be

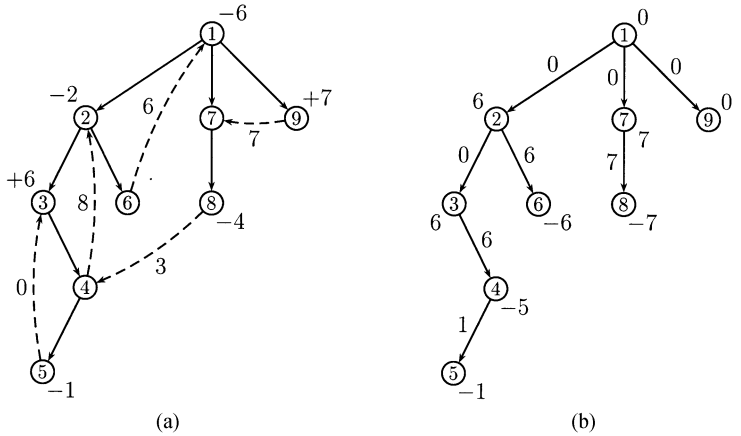

Fig. 2. The execution of the algorithm on an example network. (a) Flow on the lowlinks as calculated in Phase 2. (b) Flow on the tree edges as calculated in Phase 3.

$\sum_{w \in T_v} e_f(w)$ at the start of the processing of v. The tree pushes from v are carried out only implicitly through cancellation in the sum $\sum_{w \in T_v} e_f(w)$. If v is a source after the tree pushes, i.e., if $E_f(v) > 0$ when v is processed, a lowlink push from v is performed. The resulting decrease in $e_f(v)$ is not recorded in $E_f(v)$, since $E_f(v)$ will never again be inspected by the algorithm.

In Phase 3 the first two loops compute the value $e_g(v)$ for each $v \in V$, where g is the pseudoflow introduced above, and assign it to the variable $E_g(v)$. Then T is processed in a bottom-up fashion and, similarly as in Phase 2, during the processing of a vertex $v \in V \backslash \{r\}$, $E_g(v)$ has the value $\sum_{w \in T_v} e_g(w)$, the negative of which is assigned as the flow on the tree edge from the parent of v to v.

It can be seen that SC-ROUTE works in $O(n + m)$ time and computes the same function f as our first algorithm. This concludes the proof of Theorem 1. We illustrate the workings of SC-ROUTE through a small example.

Figure 2(a) shows the DFS tree (solid edges) and lowlinks (dashed edges) of a strongly connected graph, with vertices labeled by their preorder numbers. The lowlink of the vertex 3 is the edge $(4, 2)$, for example. Nonzero import values are shown beside the relevant vertices. During Phase 2, lowlink pushes are performed at the vertices 9, 7, 4, 3, and 2. The edge labels show the resulting flow on the lowlinks. The flow of value 8 on the edge $(4, 2)$, for example, is the result of a lowlink push from 4 to 2 that increases the flow on $(4, 2)$ from 0 to 2 and a lowlink push from 3 to 2 that increases the flow on $(4, 2)$ by another 6 units.

Figure 2(b) shows the DFS tree with edge labels representing the flow on the tree edges calculated in Phase 3. For every vertex v, the label shown next to v is the value $e_g(v)$. The algorithm sets the flow on every tree edge e equal to the negative of the sum of these values in the subtree below e.

Acknowledgment. We are grateful to Peter Sanders and Jesper Träff for many useful discussions.

References

[1] A. V. Goldberg and S. Rao, Beyond the flow decomposition barrier, *J. Assoc. Comput. Mach.*, **45** (1998), 783–797.

[2] T. Hagerup, P. Sanders, and J. L. Träff, An implementation of the binary blocking flow algorithm, in *Proc. 2nd Workshop on Algorithm Engineering* (*WAE* 1998), pp. 143–154. Res. Rep. No. MPI-I-98-1-019, Max-Planck-Institut für Informatik, Saarbrücken, 1998. http://www.mpi-sb.mpg.de/∼wae98/PROCEEDINGS/

[3] D. E. Knuth, Wheels within wheels, *J. Combin. Theory Ser. B*, **16** (1974), 42–46.

[4] R. Tarjan, Depth-first search and linear graph algorithms, *SIAM J. Comput.*, **1** (1972), 146–160.

[5] R. E. Tarjan, Efficiency of a good but not linear set union algorithm, *J. Assoc. Comput. Mach.*, **22** (1975), 215–225.