T U M

I N S T I T U T F Ü R I N F O R M A T I K

Routing flow through a strongly connected graph

Thomas Erlebach and Torben Hagerup

TUM–I9917
Oktober 99

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-10-I9917-0/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

Routing flow through a strongly connected graph

Thomas Erlebach
Institut für Informatik
Technische Universität München
D–80290 München, Germany
erlebach@in.tum.de

Torben Hagerup
Fachbereich Informatik
Johann Wolfgang Goethe-Universität Frankfurt
Robert-Mayer-Straße 11–15
D–60054 Frankfurt am Main, Germany
hagerup@informatik.uni-frankfurt.de

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 new maximum-flow algorithm of Goldberg and Rao.

1 Introduction

A *network* is given by a directed graph $G = (V, E)$ together with a function $c : E \to \mathbb{R}_+$ that maps each edge of G to a positive *capacity*; we will denote the network succinctly by the tuple (V, E, c). A *pseudoflow* in a network (V, E, c) is a function $f : E \to \mathbb{R}$ that satisfies $0 \le f(e) \le c(e)$ (the *capacity constraint*) for all $e \in E$. The *excess* of a pseudoflow f in a network (V, E, c) at a vertex $v \in V$ is defined as

$$e_f(v) = \sum_{u:(u,v)\in E} f(u, v) - \sum_{w:(v,w)\in E} f(v, w).$$

It is natural to imagine the value of a pseudoflow f on a particular edge e as the rate at which a certain commodity flows through e, the capacity of e being the maximum rate possible. The excess at a vertex v is then the net flow into v through all of its incident edges.

An instance I of the *feasible-flow* problem is given by a network (V, E, c) and a function $b : V \to \mathbb{R}$ that maps each vertex $v \in V$ to an *import* at v; we will denote I by the tuple (V, E, c, b). In the context of an instance (V, E, c, b) of the feasible-flow problem, a pseudoflow f in the network (V, E, c) is called a *flow* if $e_f(v) + b(v) = 0$ (the *conservation constraint*) holds for all $v \in V$, and the goal is simply to compute an arbitrary flow in the network. Informally, the problem is to route flow through the network from the vertices with *supplies* (positive imports) to the vertices with *demands* (negative imports). Summing the conservation constraints for all vertices shows that an instance $I = (V, E, c, b)$ of the feasible-flow problem has no solution unless $\sum_{v \in V} b(v) = 0$. We will call I *neat* if $\sum_{v \in V} b(v) = 0$ and the underlying graph $G = (V, E)$ is strongly connected. Even if the instance I is neat, it may not have a solution, but a solution always exists if the edge capacities are sufficiently large relative to the imports. As a measure of the difficulty of a neat instance $I = (V, E, c, b)$, we define the *relative throughput* of I as the ratio $(\sum_{v \in V} \max\{b(v), 0\})/\min_{e \in E} c(e)$ of the total supply to the smallest edge capacity.

Neat instances of the feasible-flow problem with relative throughput at most 1 always have solutions and are quite easy to solve, the challenge being to solve them fast. Goldberg and Rao [1] showed how to solve neat instances with relative throughput bounded by $\frac{1}{2}$ in linear time and stated that neat instances with n vertices, m edges, and relative throughput bounded by 1 can be solved 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 *Every neat instance (V, E, c, b) of the feasible-flow problem with relative throughput at most 1 can be solved in $O(|V| + |E|)$ time.*

Our algorithm realizing Theorem 1 is based on depth-first search and is simple and fast, comparable in both respects to the algorithm of Goldberg and Rao mentioned above that can handle only half as much flow.

We sketch the relevance of Theorem 1 to a new 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 will 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 neat instances of the feasible-flow problem. These instances are not necessarily (sufficiently easily) solvable, and the BBF algorithm reduces their relative throughputs as far as necessary by canceling flow in the core before attempting to solve them, 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.

2 The algorithm

In this section we prove Theorem 1. Let a neat instance $I = (V, E, c, b)$ of the feasible-flow problem with relative throughput at most 1 be given and take $G = (V, E)$, $n = |V|$, and $m = |E|$. We show how to solve I in $O(n+m)$ time. We first give a high-level description of our algorithm and prove its correctness. Subsequently we show how to implement the algorithm to run in linear time.

The algorithm works in three phases. Phase 1 runs in $O(n + m)$ time and constructs a depth-first search (DFS) tree $T = (V, E_{\mathrm{T}})$ of G. Phases 2 and 3 are bottom-up traversals of T that run in $O(n)$ time and compute, respectively, the flow on all edges in $E \setminus E_{\mathrm{T}}$ and the flow on all edges in E_{T}.

We use the following notational conventions: When g is a real-valued function defined on V, we denote by g^+ and g^- the positive and negative parts of g; i.e., $g^+(v) = \max\{g(v), 0\}$ and $g^-(v) = \min\{g(v), 0\}$ for all $v \in V$. Moreover, for every subset S of V, we write $g(S)$ for $\sum_{v \in S} g(v)$. For every edge $e = (u, v) \in E$, we call u and v the *tail* and the *head* of e, respectively, and write $u = \mathit{tail}(e)$ and $v = \mathit{head}(e)$.

The high-level description of the algorithm is summarized in Fig. 1. Phase 1 performs a DFS of G starting at an arbitrary vertex $r \in V$ and initializes a pseudoflow f by setting $f(e) = 0$ for all $e \in E$. The DFS is almost identical to Tarjan's algorithm for computing the strongly connected components of a general directed graph [4]. It constructs a DFS tree $T = (V, E_{\mathrm{T}})$ 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 $\mathit{pre}^{-1} : \{1, \ldots, n\} \to V$ with, e.g., $\mathit{pre}^{-1}(1) = r$. Additionally, for every

Algorithm SCC-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 φ(v) ← b(v) od;
for i ← n downto 2 do
    v ← pre^{-1}(i);
    q ← φ(T_v);
    if q > 0 then { a push at v }
        e ← lowlink(v);
        f(e) ← f(e) + q;
        φ(v) ← φ(v) − q;
        φ(head(e)) ← φ(head(e)) + q
    fi
od;

{ Phase 3 }
for v ∈ V \ {r} do f(parentedge(v)) ← −B(T_v) od;
```

Figure 1: Algorithm for routing flow through a strongly connected graph.

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

For $u, v \in V$, we use $u < v$ as a shorthand for $\mathit{pre}(u) < \mathit{pre}(v)$. As G is strongly connected, for every $v \in V \setminus \{r\}$ there is an edge leaving T_v; i.e., $\mathit{head}(\mathit{lowlink}(v)) < v$ for all $v \in V \setminus \{r\}$. Let $E_{\mathrm{L}} = \{\mathit{lowlink}(v) \mid v \in V \setminus \{r\}\}$ be the set of all lowlink edges. For every subset S of V, we use

$$B(S) = b(S) + \sum_{e \in E_{\mathrm{L}} \cap ((V \setminus S) \times S)} f(e) - \sum_{e \in E_{\mathrm{L}} \cap (S \times (V \setminus S))} f(e)$$

to denote the total import at vertices in S plus the net flow into S on lowlink edges.

The purpose of Phase 2 is to reduce the feasible-flow problem in G to a feasible-flow problem in T, which is then easily solved in Phase 3. The vertices are processed in inverse preorder, i.e., in the order $pre^{-1}(n), \ldots, pre^{-1}(1)$. If the current vertex v has more supply than what is needed in its subtree T_v, the surplus is moved to a vertex with smaller preorder number by routing it out of T_v on the edge $lowlink(v)$. We call this operation a *push* at v. To keep track of the effects of push operations, we maintain for every vertex $v \in V$ a quantity $\phi(v)$, initialized to the value $b(v)$. As part of a push at v, $\phi(v)$ is decreased and $\phi(head(lowlink(v)))$ is increased correspondingly.

Phase 3 sets the flow on each edge $(u, v) \in E_{\mathrm{T}}$ to the unique value that ensures that flow conservation holds for the subtree T_v, i.e., to $-B(T_v)$.

When arguing about quantities that change in the course of the execution of the algorithm, we use "initially" to denote the time immediately after the initialization of ϕ in Phase 2, and an overline denotes a final value current at the end of the execution. E.g., for all $v \in V$, $\phi(v) = b(v)$ initially, and $\overline{\phi}(v)$ is the value of $\phi(v)$ at the end of Phase 2 (since $\phi(v)$ is not changed in Phase 3).

We must demonstrate that the function $\overline{f} : E \to \mathbb{R}$ computed by the algorithm is indeed a flow. This is shown in the following lemmas. Let $\Delta = b^+(V)$ be the total supply of I. Since the relative throughput of I is at most 1, $c(e) \geq \Delta$ for all $e \in E$.

Lemma 1 *For all $v \in V$, $\overline{\phi}(T_v) \leq 0$.*

Proof. During the processing of v in Phase 2, $\phi(v)$ is decreased sufficiently to make $\phi(T_v) \leq 0$. Because Phase 2 processes the vertices in inverse preorder, no term in the sum $\sum_{w \in T_v} \phi(w) = \phi(T_v)$ subsequently changes. □

Lemma 2 *For all $v \in V$, $\phi(v) \geq 0$ immediately after every decrease in $\phi(v)$.*

Proof. Only a push at v can decrease $\phi(v)$. By Lemma 1, applied to the children of v, a push at v decreases $\phi(v)$ to a nonnegative value. □

Lemma 3 *For all $(x, y) \in E_{\mathrm{L}}$, $0 \leq \overline{f}(x, y) \leq \Delta$.*

Proof. Since $f(x, y)$ is zero initially and never decreases, $\overline{f}(x, y) \geq 0$. To see that $\overline{f}(x, y) \leq \Delta$, observe that $f(x, y) + \sum_{w > y} \phi^+(w)$ is bounded by Δ initially and, by Lemma 2, never increases. □

Lemma 4 *For all $(u, v) \in E_{\mathrm{T}}$, $0 \leq \overline{f}(u, v) \leq \Delta$.*

Proof. Our task is to show that $-\Delta \le \overline{B}(T_v) \le 0$. Initially, $B(T_v) = \phi(T_v)$, and this relation continues to hold until the end of the processing of v (in Phase 2). Let B_v denote the value of $B(T_v)$ at that time; i.e., $B_v = \overline{\phi}(T_v)$. By Lemma 1, $B_v \le 0$. After the processing of v, $B(T_v)$ never increases, so $\overline{B}(T_v) \le B_v \le 0$.

By what was observed above, the relation $B(T_v) - \phi^+(T_v) = \phi^-(T_v)$ holds initially and until the end of the processing of v. Since $\phi^-(T_v)$ is at least $-\Delta$ initially and, by Lemma 2, never decreases, $B_v \ge -\Delta$. What remains to show is that $B(T_v)$ does not decrease below $-\Delta$ after the processing of v.

Let A denote the set of ancestors of v that have a lowlink edge e with $\mathit{tail}(e) \in T_v$ and let a be the vertex in A with minimal preorder number. The quantity $\Phi = \sum_{w \ge a} \phi^+(w)$ is bounded by Δ initially and, by Lemma 2, never increases. Consider a decrease by q in $B(T_v)$ following the processing of v. The decrease must happen as part of a push at a vertex $z \in A \setminus \{v\}$, and we have $\mathit{head}(\mathit{lowlink}(z)) = \mathit{head}(\mathit{lowlink}(a)) < a$. The decrease in $B(T_v)$ is accompanied by a decrease from q to zero in $\phi(T_z)$ and, by what we just saw, by a decrease by q in Φ. Lemma 1, applied at all siblings of vertices other than z on the path in T from z to v, shows that $\phi(T_z)$ is bounded above by $\Phi + B_v$ immediately before and after the push at z, and therefore the decrease in $B(T_v)$ cannot end with $\Phi + B_v < 0$, i.e., with $\Phi < -B_v$. It follows that the total decrease in $B(T_v)$ after the processing of v is bounded by $\max\{\Delta + B_v, 0\} = \Delta + B_v$, so that $\overline{B}(T_v) \ge B_v - (\Delta + B_v) = -\Delta$. □

Lemma 5 *$\overline{f}$ satisfies the conservation constraint for every vertex in V.*

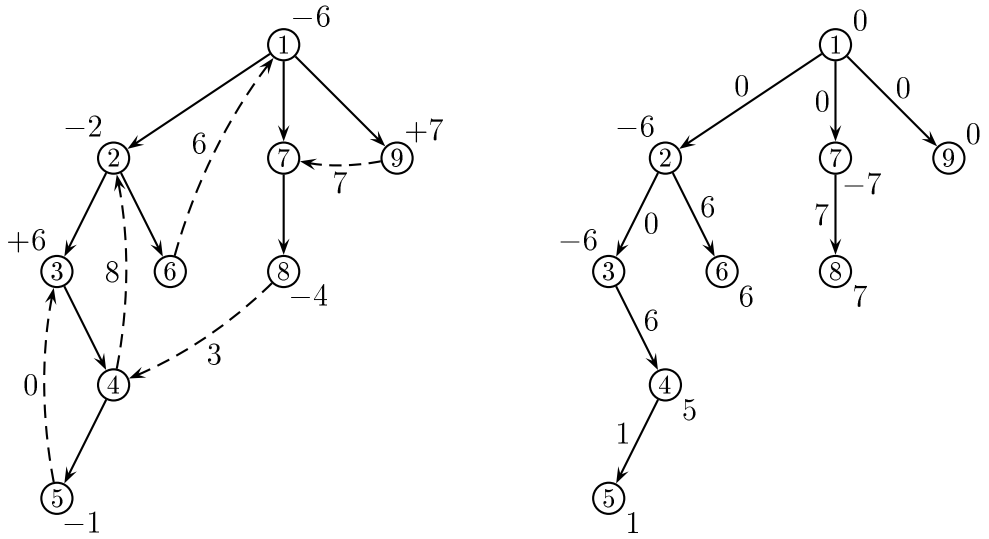
Proof. $\overline{f}(e) = 0$ for all $e \in E \setminus (E_{\mathrm{T}} \cup E_{\mathrm{L}})$. For each vertex $v \in V$ with children $w_1, \ldots, w_d$, we therefore find

$$e_{\overline{f}}(v) + b(v) = \overline{B}(\{v\}) - \overline{B}(T_v) + \sum_{i=1}^{d} \overline{B}(T_{w_i}) = 0.$$

□

Lemmas 3 and 4 show that $\overline{f}$ satisfies the capacity constraints for all edges in E, and Lemma 5 states that $\overline{f}$ satisfies the conservation constraints for all vertices in V. This concludes the correctness proof for the algorithm of Fig. 1. We illustrate the workings of the algorithm through a small example.

Fig. 2(a) shows the DFS tree (solid edges) and lowlink edges (dashed) of a strongly connected graph. For ease of discussion, we identify each vertex with its preorder number, shown inside the corresponding circle. 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, push operations are performed



(a) Flow on the nontree edges as calculated in Phase 2

(b) Flow on the tree edges as calculated in Phase 3

Figure 2: The execution of the algorithm on an example network.

at the vertices 9, 7, 4, 3, and 2. The edge labels show the resulting flow on the lowlink edges. The flow of value 8 on the edge $(4, 2)$, e.g., is the result of a push at the vertex 4 that increases the flow on $(4, 2)$ from 0 to 2 and a push at the vertex 3 that increases the flow on $(4, 2)$ by another 6 units.

Fig. 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 $-\overline{B}(\{v\})$. The algorithm sets the flow on every tree edge e equal to the sum of these values in the subtree below e.

We finally show how to implement Phases 2 and 3 of the algorithm in $O(n)$ time. The values $\phi(T_v)$ needed in Phase 2 (cf. Fig. 1) can be computed on the fly in $O(n)$ total time. It suffices to let every vertex v other than r pass $\phi(T_v)$ to its parent u, where it will be added to the values coming from siblings of v and to $\phi(u)$. The values of ϕ are changed by push operations in Phase 2, but this causes no difficulties: If a push is performed at a vertex v, $\phi(T_v)$ is reduced to zero and no value need be passed to the parent of v; the vertex $z = \mathit{head}(\mathit{lowlink}(v))$ will be processed later, so it suffices to pass the change in $\phi(z)$ to z.

The values $B(T_v)$ that are needed in Phase 3 of the algorithm can be computed in $O(n)$ total time for all $v \in V$ by a single sweep from the leaves to the root of T. At each vertex v, values coming from the children are added to the initial value stored at v and, if v is not the root, the result is sent to

```
void SCC_route() {
  int i,v,e;
  int q; /* flow values are assumed to be integers */
  DFS(0); /* 0 is chosen arbitrarily as the root */
  for (e=0;e<m;e++) f[e]=0;
  for (v=0;v<n;v++) fp[v]=-b[v];
  for (i=n;i>1;i--) {
    v=preinv[i]; /* the vertex of preorder i */
    q=b[v];
    if (q<=0) b[parent[v]]+=q; /* move demand to parent */
    else { /* move supply over lowlink (a push at v) */
      e=lowlink[v];
      f[e]+=q; b[head[e]]+=q; fp[tail[e]]+=q; fp[head[e]]-=q;
    }
  }
  for (i=n;i>1;i--) {
    v=preinv[i];
    fp[parent[v]]+=f[parentedge[v]]=fp[v];
  }
}
```

Figure 3: An efficient implementation of the algorithm in C.

its parent. Rather than initializing the value at v with $B(\{v\})$ once this quantity has reached its final value at the end of Phase 2, it is convenient to maintain the negative of the quantity throughout Phase 2.

A code fragment in executable C that implements our algorithm along the lines suggested above is shown in Fig. 3. The sets V and E are represented by $\{0, \ldots, n-1\}$ and $\{0, \ldots, m-1\}$, respectively, where n and m are global variables. The implementation uses arrays `b`, `parent`, `parentedge`, `lowlink`, and `fp` indexed by vertices, arrays `tail`, `head`, and `f` indexed by edges, and an array `preinv` indexed by preorder numbers that implements the bijection pre^{-1}. Declarations of these arrays are not shown. The (cumulative) values of $-B$ are stored in the array `fp`, and the input array `b` that specifies the imports is also used to store the (cumulative) values of ϕ.

Acknowledgment. The second author is grateful to Peter Sanders and Jesper Träff for many useful discussions and for proving his first algorithm incorrect.

References

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

[2] T. Hagerup, P. Sanders, and J. L. Träff, An implementation of the binary blocking flow algorithm, 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, Germany. http://www.mpi-sb.mpg.de/~wae98/PROCEEDINGS.

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

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

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