OPERATIONS RESEARCH
Vol. 56, No. 4, July–August 2008, pp. 992–1009
ISSN 0030-364X | EISSN 1526-5463 | 08 | 5604 | 0992

informs®
DOI 10.1287/opre.1080.0524
© 2008 INFORMS

The Pseudoflow Algorithm: A New Algorithm for the Maximum-Flow Problem

Dorit S. Hochbaum
Department of Industrial Engineering and Operations Research and Walter A. Haas School of Business,
University of California, Berkeley, California 94720, hochbaum@ieor.berkeley.edu

We introduce the pseudoflow algorithm for the maximum-flow problem that employs only pseudoflows and does not generate flows explicitly. The algorithm solves directly a problem equivalent to the minimum-cut problem—the maximum blocking-cut problem. Once the maximum blocking-cut solution is available, the additional complexity required to find the respective maximum-flow is $O(m \log n)$. A variant of the algorithm is a new parametric maximum-flow algorithm generating all breakpoints in the same complexity required to solve the constant capacities maximum-flow problem. The pseudoflow algorithm has also a simplex variant, pseudoflow-simplex, that can be implemented to solve the maximum-flow problem. One feature of the pseudoflow algorithm is that it can initialize with any pseudoflow. This feature allows it to reach an optimal solution quickly when the initial pseudoflow is "close" to an optimal solution. The complexities of the pseudoflow algorithm, the pseudoflow-simplex, and the parametric variants of pseudoflow and pseudoflow-simplex algorithms are all $O(mn \log n)$ on a graph with n nodes and m arcs. Therefore, the pseudoflow-simplex algorithm is the fastest simplex algorithm known for the parametric maximum-flow problem. The pseudoflow algorithm is also shown to solve the maximum-flow problem on s, t-tree networks in linear time, where s, t-tree networks are formed by joining a forest of capacitated arcs, with nodes s and t adjacent to any subset of the nodes.

Subject classifications: flow algorithms; parametric flow; normalized tree; lowest label; pseudoflow algorithm; maximum flow.
Area of review: Optimization.
History: Received May 2001; revisions received December 2002, June 2003, June 2004, April 2005, May 2007; accepted May 2007.

1. Introduction

The maximum-flow problem is to find in a network with a source, a sink, and arcs of given capacities, a flow that satisfies the capacity constraints and flow-balance constraints at all nodes other than source and sink, so that the amount of flow leaving the source is maximized.

The past five decades have witnessed prolific developments of algorithms for the maximum-flow problem. These algorithms can be classified in two major classes: feasible-flow algorithms and preflow algorithms. The feasible-flow algorithms work with augmenting paths incrementing the flow at every iteration. The first algorithm of this type was devised by Ford and Fulkerson (1957). A *preflow* is a flow that may violate the restriction on the balance of the incoming flow and the outgoing flow into each node other than source and sink by permitting excesses (more inflow than outflow). It appears that the first use of preflow for the maximum-flow problem was in work by Boldyreff (1955), in a push-type algorithm with heuristically chosen labels. Boldyreff's technique, named "flooding technique", does not guarantee an optimal solution. The push-relabel algorithm of Goldberg and Tarjan (1988) uses preflows, and is efficient both theoretically and empirically.

The literature on maximum-flow algorithms includes numerous algorithms. Most notable for efficiency among feasible-flow algorithms is Dinic's (1970) algorithm of complexity, $O(n^2 m)$. An improved version of this algorithm runs in time $O(n^3)$ (Karzanov 1974, Malhorta et al. 1978). Goldberg and Rao (1998) based their algorithm on an extension of Dinic's algorithm for unit capacity networks with run time of $O(\min\{n^{2/3}, m^{1/2}\} m \log(n^2/m) \log U)$ for U the largest arc capacity. Goldberg and Tarjan's (1988) push-relabel algorithm with dynamic trees implementation has complexity of $O(mn \log n^2/m)$, and King et al. (1994) devised an algorithm of complexity $O(mn \log_{m/n \log n} n)$.

We describe here a novel approach for solving the maximum-flow and minimum-cut problems which is based on the use of *pseudoflow* permitting excesses and deficits.[1] Source and sink nodes have no distinguished role in this algorithm, and all arcs adjacent to source and sink in the maximum-flow problem instance are maintained saturated throughout the execution of the algorithm. The method seeks a partition of the set of nodes to subsets, some of which have excesses, and some have deficits, so that all arcs going from excess subsets to deficit subsets are saturated. The partition with this property is provably a minimum cut in the graph.

The pseudoflow algorithm uses a certificate of optimality inspired by the algorithm of Lerchs and Grossmann (1965) for the maximum-closure problem defined on a digraph with node weights. That algorithm was devised for the purpose of finding the optimal solution (contour)

of the open-pit mining problem. The algorithm of Lerch and Grossmann (1965) does not work with flows but rather with a concept of *mass* representing the total sum of node weights in a given subset. It is shown here that the concept of mass generalizes in capacitated networks to the notion of pseudoflow. The reader interested in further investigation of the conceptual link between our algorithm and Lerchs and Grossmann's algorithm is referred to Hochbaum (2001).

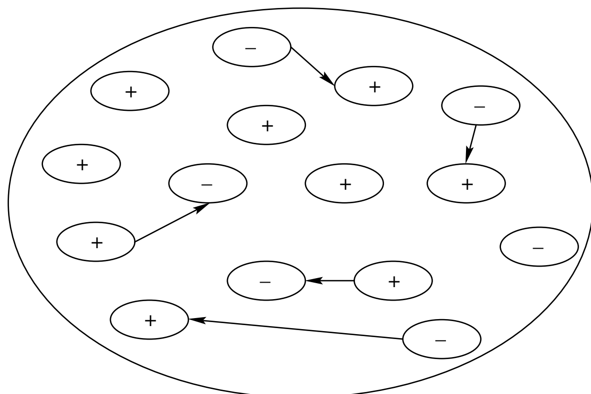
The pseudoflow algorithm solves, instead of the maximum-flow problem, the *maximum blocking-cut problem* (Radzik 1993). The blocking-cut problem is defined on a directed graph with arc capacities and node weights that does *not* contain source and sink nodes. The objective of the blocking-cut problem is to find a subset of the nodes that maximizes the sum of node weights, minus the capacities of the arcs leaving the subset. This problem is equivalent to the minimum s, t-cut problem (see §3) that is traditionally solved by deriving a maximum flow first.

At each iteration of the pseudoflow algorithm, there is a partition of the nodes to subsets with excesses and subsets with deficits such that the total excess can only be greater than the maximum blocking-cut value. In that sense, the union of excess subsets forms a *superoptimal solution*. The algorithm is thus interpreted as a dual algorithm for the maximum blocking-cut problem. If there are no unsaturated arcs between excess subsets and deficit subsets, then the union of the excess subsets forms an optimal solution to the maximum blocking-cut problem. A schematic description of the partition at an iteration of the algorithm is given in Figure 1.

The pseudoflow algorithm works with a tree structure called a *normalized tree*. This tree preserves some information about residual paths in the graph. The normalized tree is used as a basic arcs tree in a simplex-like variant of the pseudoflow algorithm, described in §10. We call this variant the *pseudoflow-simplex*.

A part of the investigation here is on the sensitivity analysis of the maximum-flow problem using the pseudoflow algorithm. The goal of sensitivity analysis, or parametric analysis, is to find the maximum flow as a function of a parameter when source and sink adjacent arc capacities are monotone nondecreasing and nonincreasing functions of the parameter, respectively. We distinguish two types of sensitivity analysis: In *simple sensitivity analysis*, we are given k parameter values for the arc capacities functions, and the problem is to find the optimal solution for each of these values. In *complete parametric analysis*, the goal is to find all the maximum flows (or minimum cuts) for any value of the parameter.

Figure 1. A schematic description of the graph during the execution of the pseudoflow algorithm with a partition to subsets of nodes of excess/deficit marked $+/-$, respectively.



Martel (1989) showed that a variant of Dinic's algorithm can solve the simple sensitivity analysis in $O(n^3 + kn^2)$. Gallo et al. (1989) showed that simple sensitivity analysis for $k = O(n)$ and complete parametric analysis problems for linear functions of the parameter can be solved in the same time as a single run of the push-relabel preflow algorithm, $O(mn \log n^2/m + km \log n^2/m)$. The simple sensitivity analysis was subsequently improved by Gusfield and Tardos (1994) to $O(mn \log n^2/m + kn \log n^2/m)$ permitting the increase of the number of parameter values to $k = O(m)$, while still maintaining the same complexity. We show here that both simple sensitivity analysis for $k = O(m \log n)$ and complete parametric analysis can be performed using the pseudoflow algorithm or the pseudoflow-simplex algorithm in the same time as a single run, $O(mn \log n + kn)$. The complete parametric analysis algorithm can be extended to any monotone functions, for the pseudoflow algorithm and its variants, and also for the push-relabel algorithm by adding $O(n \log U)$ steps, where U is the range of the parameter (Hochbaum 2003). The pseudoflow and the pseudoflow-simplex algorithms are thus the only alternatives to the push-relabel algorithm known to date that can solve the complete parametric analysis efficiently.

The contributions here include:

(1) A new algorithm for the maximum-flow problem of complexity, $O(mn \log n)$. This is the first algorithm specifically designed for the maximum-flow problem that makes use of pseudoflows.

(2) A new pseudoflow-based simplex algorithm for maximum flow, matching the best complexity of a simplex algorithm for the problem (Goldberg et al. 1991).

(3) A simple sensitivity analysis algorithm for the maximum-flow problem on k parameter values with the pseudoflow algorithm of complexity, $O(mn \log n + kn)$. The pseudoflow-simplex algorithm for simple sensitivity analysis also runs in $O(mn \log n + kn)$ time. This improves on the previously best-known simplex-based algorithm of Goldfarb and Chen (1997) for simple sensitivity analysis with complexity $O(mn^2 + kn)$.

(4) A complete parametric analysis with the pseudoflow or pseudoflow-simplex algorithms generating all breakpoints for any monotone functions of the parameter in $O(mn \log n + n \log U)$ steps and in $O(mn \log n)$ steps for linear functions. The pseudoflow-simplex is the first

simplex-based algorithm that performs the complete parametric analysis problem in the same complexity as a single constant capacities instance.

(5) An efficient procedure for "warm starting" the algorithm when the graph arcs and capacities are modified arbitrarily.

(6) A linear-time algorithm for maximum flow on an s, t-tree network, which is a network with tree topology (in the undirected sense) appended by source and sink nodes that are connected to any subset of the nodes of the tree. This is used as a subroutine, e.g., in solving the minimum-cost network flow (see Vygen 2002).

This paper is organized as follows. The next section introduces notations and relevant definitions. In §3, we discuss the relationship of the maximum blocking-cut problem to the minimum-cut problem, the maximum-flow problem, and the maximum-closure problem. Section 4 introduces the normalized tree and its properties. Section 5 describes the pseudoflow algorithm and establishes its correctness as a maximum blocking-cut algorithm. In §6, the generic pseudoflow algorithm is shown to have pseudopolynomial run time, a scaling variant is shown to have polynomial run time, and a labeling variant is shown to have strongly polynomial run time. Section 7 presents several strongly polynomial variants of the pseudoflow algorithm. In §8, it is demonstrated how to recover from a normalized tree a feasible flow in time $O(m \log n)$ and $O(m \log n + n \log U)$, respectively. At optimality, the flow amount is equal to the capacity of the cut arcs and thus we conclude that the pseudoflow algorithm is also a maximum-flow algorithm. In §9, we discuss the parametric features of the algorithm and show that simple sensitivity analysis and complete parametric analysis can be implemented in $O(mn \log n)$ for linear functions, and with an additive factor of $O(n \log U)$ for arbitrary monotone functions. The pseudoflow-simplex and its parametric implementation are presented in §10. Section 11 describes an implementation of the push-relabel algorithm as a pseudoflow-based method. The methodologies of pseudoflow, the pseudoflow-simplex, and the push-relabel algorithms are compared and contrasted in §12. The online appendices contain a new algorithm for normalizing any given tree in a network in $O(m)$ steps. The implications for efficient warm starts, minimum directed cuts, and a linear-time maximum-flow algorithm for s, t-tree networks are discussed in the appendices as well. An electronic companion to this paper is available as part of the online version that can be found at http://or.journal.informs.org/.

2. Preliminaries and Notations

For a directed graph $G = (V, A)$, the number of arcs is denoted by $m = |A|$ and the number of nodes by $n = |V|$. A graph is called an s, t-graph if its set of nodes contains two distinguished nodes s and t.

For $P, Q \subset V$, the set of arcs going from P to Q is denoted by $(P, Q) = \{(u, v) \in A \mid u \in P \text{ and } v \in Q\}$. The capacity of an arc $(u, v) \in A$ is a nonnegative real number c_{uv}, and the flow on that arc is f_{uv}. For simplicity, we set all lower bound capacities to zero, yet all results reported apply also in the presence of nonzero lower bounds. A *pseudoflow* f in an arc capacitated s, t-graph is a mapping that assigns to each arc (u, v) a real value f_{uv} so that $0 \leqslant f_{uv} \leqslant c_{uv}$.

For a given pseudoflow f in a simple s, t-graph (containing at most one arc for each pair of nodes), the residual capacity of an arc $(u, v) \in A$ is $c^f_{uv} = c_{uv} - f_{uv}$ and the residual capacity of the backwards arc (v, u) is $c^f_{vu} = f_{uv}$. An arc or a backwards arc is said to be a *residual arc* if its residual capacity is positive. So, the set of residual arcs A_f is $A_f = \{(i, j) \mid f_{ij} < c_{ij} \text{ and } (i, j) \in A \text{ or } f_{ji} > 0 \text{ and } (j, i) \in A\}$.

For $P, Q \subset V$, $P \cap Q = \varnothing$, the capacity of the cut separating P from Q is $C(P, Q) = \sum_{(u, v) \in (P, Q)} c_{uv}$. For a given pseudoflow f, the total flow from a set P to a set Q is denoted by $f(P, Q) = \sum_{(u, v) \in (P, Q)} f_{uv}$. For a given pseudoflow f, the total capacity of the residual cut from a set P to a set Q is denoted by $C^f(P, Q) = \sum_{(u, v) \in (P, Q)} c^f_{uv}$.

Even though the underlying graph considered is directed, the directions of the arcs are immaterial in parts of the algorithm and discussion. An arc (u, v) of an unspecified direction is referred to as *edge* $[u, v]$. So, we say that $[u, v] \in A$ if either (u, v) or $(v, u) \in A$. The capacity of an edge e is denoted by c_e. The flow on an edge e is denoted by f_e.

The ordered list $(v_1, v_2, \ldots, v_k)$ denotes a *directed path* from v_1 to v_k with $(v_1, v_2), \ldots, (v_{k-1}, v_k) \in A$. A directed path is said to be a *residual path* if $(v_1, v_2), \ldots, (v_{k-1}, v_k) \in A_f$. An *undirected path* from v_1 to v_k is denoted by $[v_1, v_2, \ldots, v_k]$ with $[v_1, v_2], \ldots, [v_{k-1}, v_k] \in A$.

An s, t-graph is called a *closure graph* if the only arcs of finite capacities are those adjacent to the source and sink nodes.

A *rooted tree* is a collection of arcs that forms an undirected acyclic connected graph T with one node designated as a root. A rooted tree is customarily depicted with the root above and the tree nodes suspended from it below. A node u is called an *ancestor* of node v if the (unique) path from v to the root contains u. All nodes that have node u as an ancestor are called the *descendants* of node u. T_v denotes the subtree suspended from node v that contains v and all the descendants of v in T. $T_{[v, p(v)]} = T_v$ is the subtree *suspended* from the edge $[v, p(v)]$. The *parent* of a node v, denoted by $p(v)$, is the unique node that follows v on the path from v to the root of the tree. All nodes that have node v as a parent are the immediate descendants of v and are called children of v. A *child* of v is denoted by $ch(v)$. We will occasionally refer to the nodes or the arcs of a tree T as the set T whenever there is no risk of ambiguity.

We introduce three related equivalent representations of a graph: G, G_{st}, and G^{ext}.

(1) The directed graph $G = (V, A)$ has real node weights w_i for $i \in V$ and (positive) arc capacities c_{ij} for $(i, j) \in A$.

(2) The graph $G_{st} = (V_{st}, A_{st})$ is an s, t-graph with only arc capacities. It is constructed from the graph G as

follows: The set of nodes is $V_{st} = V \cup \{s, t\}$, and the set of arcs A_{st} comprises of the arcs of A appended by sets of arcs adjacent to s and t, $A(s)$, and $A(t)$. The arcs of $A(s) = \{(s, j) \mid w_j > 0\}$ connect s to all nodes of positive weight, each of capacity equal to the weight of the respective node, $c_{sj} = w_j$. Analogously, $A(t) = \{(j, t) \mid w_j < 0\}$ and $c_{jt} = -w_j = |w_j|$ for $(j, t) \in A(t)$. Zero weight nodes are connected neither to the source nor to the sink. Thus, $G_{st} = (V_{st}, A_{st}) = (V \cup \{s, t\}, A \cup A(s) \cup A(t))$.

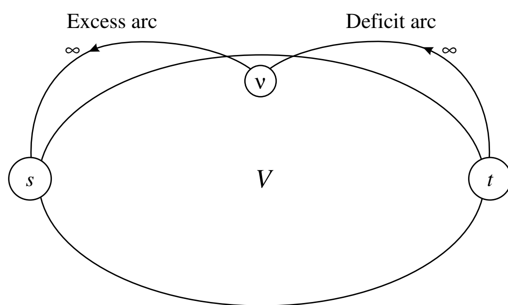
The inverse map from a graph G_{st} to a graph G is as follows: A node weighted graph $G = (V, A)$ is constructed by assigning to every node v adjacent to s a weight $w_v = c_{sv}$, and every node u adjacent to t is assigned a weight $w_u = -c_{ut}$. Nodes that are adjacent neither to the source nor to the sink are assigned the weight zero. For a node v adjacent to both s and t in G_{st}, the lower capacity arc among the two of value $\delta = \min\{c_{sv}, c_{vt}\}$ is removed, and the value δ is subtracted from the other arc's capacity. Therefore, each node can be assumed to be adjacent to either source or sink or to neither. The source and sink nodes are then removed from V_{st}.

(3) The *extended network*, G^{ext}, corresponds to an s, t-graph G_{st} by adding to A_{st}, for each node v, two arcs of infinite capacity—(t, v) and (v, s)—and then shrinking s and t into a single node r called *root*. We refer to the appended arcs from sink t as the *deficit arcs* and the appended arcs to the source s as the *excess arcs* and denote them by $A_\infty = \bigcup_{v \in V}\{(v, r) \cup (r, v)\}$. Figure 2 provides a schematic description of such a network. The *extended network* is the graph $G^{\text{ext}} = (V \cup \{r\}, A^{\text{ext}})$, where $A^{\text{ext}} = A \cup A(s) \cup A(t) \cup A_\infty$.

The maximum-flow problem is defined on a directed s, t-graph, $G_{st} = (V_{s,t}, A_{st})$. An arc of infinite capacity is added from sink to source (t, s) to turn the problem into a circulation problem. The standard formulation of the maximum-flow problem with variable f_{ij} indicating the amount of flow on arc (i, j) is

$$\begin{aligned} &\text{Max} \quad f_{ts} \\ &\text{subject to} \quad \sum_i f_{ki} - \sum_j f_{jk} = 0 \quad \forall k \in V_{s,t}, \\ &\qquad 0 \leqslant f_{ij} \leqslant c_{ij} \quad \forall (i, j) \in A_{st}. \end{aligned}$$

Figure 2. An extended network prior to shrinking source and sink into r.



The objective function value, which is also the total flow leaving the source (or arriving at the sink) is denoted by $|f|$. In this formulation, the first set of (equality) constraints is called the *flow-balance constraints*. The second set of (inequality) constraints is called the *capacity constraints*. A "preflow" violates the flow-balance constraints in one direction permitting nonnegative excess $\sum_i f_{ki} - \sum_j f_{jk} \leqslant 0$. A "pseudoflow" may violate the flow-balance constraints in both directions. Capacity constraints are satisfied by both preflow and pseudoflow.

Claim 2.1. *For any pseudoflow in G_{st}, there is a corresponding feasible flow on the graph G^{ext}.*

Proof. The feasible flow is constructed by sending the excess or deficit of each node v back to node r via the excess arc (v, r) or the deficit arc (r, v). □

Let f be a pseudoflow vector in G_{st} with $0 \leqslant f_{ij} \leqslant c_{ij}$ and let *inflow*(D), *outflow*(D) be the total amount of flow incoming and outgoing to and from the set of nodes D. For each subset of nodes, $D \subseteq V$, the excess of D is the net inflow into D,

$$\begin{aligned} \mathit{excess}(D) &= \mathit{inflow}(D) - \mathit{outflow}(D) \\ &= \sum_{(u,v) \in (V \cup \{s\} \setminus D,\, D)} f_{uv} - \sum_{(v,u) \in (D,\, V \cup \{t\} \setminus D)} f_{vu}. \end{aligned}$$

The excess of a singleton node v is *excess*(v). A negative-valued excess is called *deficit*, with *deficit*$(D) = -$*excess*(D).

For $S \subseteq V$, we let the *complement* of S be $\bar{S} = V \setminus S$. The complement sets used here are only with respect to V, even for the graphs G_{st} and G^{ext}.

Next, we introduce the maximum blocking-cut problem and the concept of surplus of a set.

Problem Name: Maximum blocking-cut.

Instance: Given a directed graph $G = (V, A)$, node weights (positive or negative) w_i for all $i \in V$, and nonnegative arc weights c_{ij} for all $(i, j) \in A$.

Optimization Problem: Find a subset of nodes $S \subseteq V$ such that

$$\mathit{surplus}(S) = \sum_{i \in S} w_i - \sum_{i \in S,\, j \in \bar{S}} c_{ij} \text{ is maximum.}$$

The notion of surplus of a set of nodes in G has related definitions in the graphs G_{st} and G^{ext}.

Definition 2.1. The surplus of $S \subseteq V$ in the graph G is $\sum_{j \in S} w_j - \sum_{i \in S,\, j \in \bar{S}} c_{ij}$. The surplus of $S \subseteq V$ in the graph G_{st} is $C(\{s\}, S) - C(S, \bar{S} \cup \{t\})$. The surplus of $S \subseteq V$ in the graph G^{ext} is $\sum_{j \in S,\, (r,j) \in A(s)} c_{rj} - \sum_{j \in S,\, (j,r) \in A(t)} c_{jr} - \sum_{(i,j) \in (S, \bar{S})} c_{ij}$.

These definitions of surplus in the corresponding graphs G^{ext}, G_{st}, and G are equivalent: in G^{ext} and G_{st}, the definitions are obviously the same because the capacity of

the arcs in $A(s)$ with endpoints in S is $C(\{s\}, S)$, the capacity of the arcs in $A(t)$ with endpoints in S is $C(S, \{t\})$, and $C(S, \bar{S} \cup \{t\}) = C(S, \{t\}) + C(S, \bar{S})$. To see the equivalence in G_{st} and G, observe that the sum of weights of nodes in S is also the sum of capacities $C(\{s\}, S) - C(S, \{t\})$, where the first term corresponds to positive weights in S, and the second term to negative weights in S. Therefore,

$$\sum_{j\in S} w_j - \sum_{i\in S,\, j\in \bar{S}} c_{ij} = C(\{s\}, S) - C(S, \{t\}) - C(S, \bar{S})$$
$$= C(\{s\}, S) - C(S, \bar{S} \cup \{t\}).$$

The expression on the left-hand side is the surplus of S in G, whereas the expression on the right-hand side is the surplus of S in G_{st}.

3. The Maximum-Flow and the Maximum Blocking-Cut Problems

The blocking-cut problem is closely related to the maximum-flow and minimum-cut problems as shown next. This relationship was previously noted by Radzik (1993).

Lemma 3.1. *For $S \subseteq V$, $\{s\} \cup S$ is the source set of a minimum cut in G_{st} if and only if $(S, \bar{S})$ is a maximum blocking cut in the graph G.*

Proof. We rewrite the objective function in the maximum blocking-cut problem for the equivalent graph G_{st}:

$$\max_{S\subseteq V}[C(\{s\}, S) - C(S, \bar{S} \cup \{t\})]$$
$$= \max_{S\subseteq V}[C(\{s\}, V) - C(\{s\}, \bar{S}) - C(S, \bar{S} \cup \{t\})]$$
$$= C(\{s\}, V) - \min_{S\subseteq V}[C(\{s\}, \bar{S}) + C(S, \bar{S} \cup \{t\})].$$

In the last expression, the term $C(\{s\}, V)$ is a constant from which the minimum-cut value is subtracted. Thus, the set S maximizing the surplus is also the source set of a minimum cut and, vice versa—the source set of a minimum cut also maximizes the surplus. □

We note that the blocking-cut problem has appeared in several forms in the literature. The Boolean quadratic minimization problem with all the quadratic terms having positive coefficients is a restatement of the blocking-cut problem.[2] More closely related is the feasibility condition of Gale (1957) for a network with supplies and demands, or of Hoffman (1960) for a network with lower and upper bounds. Verifying feasibility is equivalent to ensuring that the maximum blocking cut is zero in a graph with node weights equal to the respective supplies and demands with opposite signs. If the maximum blocking cut is positive, then there is no feasible flow satisfying the supply and demand balance requirements. The names *maximum blocking cut* or *maximum surplus cut* were used for the problem by Radzik (1993).

3.1. The Blocking-Cut Problem and the Maximum-Closure Problem

The pseudoflow algorithm is a generalization of the algorithm solving the maximum-closure problem in closure graphs, described in Hochbaum (2001). Indeed, the blocking-cut problem generalizes the closure problem. The maximum-closure problem is to find in a node weighted directed graph a maximum weight subset of nodes that forms a closed set, i.e., a set of nodes that contains all successors of each node in the closed set. Picard (1976) demonstrates how to formulate the maximum-closure problem on a graph $G = (V, A)$ as a flow problem on the respective G_{st} graph: All arc capacities in A are set to infinity, a source node s and a sink node t are added to V and arcs $A(s)$ and $A(t)$ are appended as in the description of how to derive G_{st} from G: $A(s)$ contains arcs from s to all nodes of positive weight with capacity equal to that weight, and $A(t)$ contains arcs from all nodes of negative weights to t with capacities equal to the absolute value of the weight. In G_{st}, any finite cut $C(S, \bar{S} \cup \{t\})$ must have $(S, \bar{S}) = \varnothing$. This implies that for such a finite cut, the set S is closed. It follows that $C(S, \bar{S} \cup \{t\})$ is equal to $C(S, \{t\})$, and thus Lemma 3.1 demonstrates that the source set of a minimum cut is also a maximum weight closed set.

The blocking-cut problem generalizes the maximum closure problem by *relaxing* the closure requirement: Nodes that are successors of nodes in S, i.e., nodes that are the head of arcs with tails in nodes of S, may be excluded from the set S, but at a penalty equal to the capacity of the respective arcs.

From Lemma 3.1, we conclude that any maximum-flow or minimum-cut algorithm solves the maximum blocking-cut problem: The source set of a minimum cut is a maximum surplus set. The pseudoflow algorithm works in a reverse direction by solving the maximum blocking-cut problem first, which provides a minimum cut, and then recovering a maximum flow.

4. A Normalized Tree

The pseudoflow algorithm maintains a construction called *a normalized tree* after the use of this term by Lerchs and Grossmann (1965). A normalized tree $T = (V \cup \{r\}, E_T)$ is defined on a spanning tree in G^{ext} rooted in r so that $E_T \subseteq A \cup \bigcup_{v\in V}\{(v, r) \cup (r, v)\}$. The children of r in such a spanning tree are denoted by r_i, and called the *roots* of their respective *branches* (also referred to as trees or subtrees). In a normalized tree, only the branch roots r_i are permitted to carry nonzero deficits or excesses. For a given pseudoflow f, a branch T_{r_i} is called *strong* if $excess(T_{r_i}) = excess(r_i) = f_{r_i, r} > 0$, and *weak* otherwise. All nodes of strong branches are considered strong, and all nodes of weak branches are considered weak. Branches with zero excess are called zero-deficit branches, and are considered to be weak.

The tree T induces a forest of all the branches in $G = (V, A)$ formed by the set of arcs $E_T \cap A$. The arcs in the set $E_T \cap A$ are called *in-tree arcs*, and the arcs in the set $A \backslash E_T$ are called *out-of-tree arcs*.

Recall that the root is placed at the top of the tree, so that the reference to the downwards direction is equivalent to "pointing away from the root," and the upwards direction is equivalent to "pointing toward the root." The topology of a normalized tree with three branches is illustrated in Figure 3. The branch rooted at r_1 is strong because the amount of excess of the branch T_{r_1} (and of r_1) is positive. Branch T_{r_3} has nonpositive excess and is thus considered weak.

Definition 4.1. A spanning tree T in G^{ext} with a pseudoflow f in G_{st} is called *normalized* if it satisfies Properties 1, 2, 3, and 4.

Property 1. *The pseudoflow f saturates all source-adjacent arcs and all sink-adjacent arcs $A(s) \cup A(t)$.*

Property 2. *The pseudoflow f on out-of-tree arcs is equal to the lower or the upper bound capacities of the arcs.*

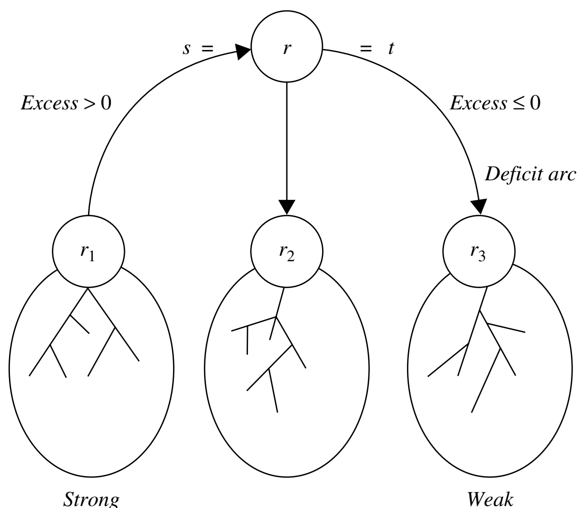
Property 3. *In every branch, all downwards residual capacities are strictly positive.*

Property 4. *The only nodes that do not satisfy flow-balance constraints in G_{st} are the roots of their respective branches.*

Property 4 means that for T to be a normalized tree, the arcs connecting the roots of the branches to r in G^{ext} are the only excess and deficit arcs permitted. It also means that the excess of a branch is equal to the excess of its root, $excess(T_{r_i}) = excess(r_i)$.

It is shown next that a crucial property—the superoptimality property—is satisfied by any normalized tree and is thus a corollary of Properties 1, 2, 3, and 4.

Figure 3. A schematic description of a normalized tree.



Note. Each r_i is a root of a branch.

Property 5 (Superoptimality). *The set of strong nodes of a normalized tree T is a superoptimal solution to the blocking-cut problem. That is, the sum of excesses of the strong branches is an upper bound on the maximum surplus.*

Proof. To establish the superoptimality property, we first prove two lemmata. Recall that for a pseudoflow f and any $D \subseteq V$, the capacity of the residual cut from D to $\bar{D} = V \backslash D$ is $C^f(D, \bar{D}) = \sum_{(i,j)\in A\cap(D,\bar{D})}(c_{ij} - f_{ij}) + \sum_{(j,i)\in A\cap(\bar{D},D)} f_{ji}$.

Lemma 4.1. *For a pseudoflow f saturating $A(s) \cup A(t)$, $surplus(D) = excess(D) - C^f(D, \bar{D})$.*

Proof.

$$\begin{aligned} excess(D) &= C(\{s\}, D) - C(D, \{t\}) + f(\bar{D}, D) - f(D, \bar{D}) \\ &= C(\{s\}, D) - C(D, \{t\}) + f(\bar{D}, D) \\ &\quad - \left(C(D, \bar{D}) - \sum_{(i,j)\in A\cap(D,\bar{D})} c_{ij}^f \right) \\ &= C(\{s\}, D) - C(D, \{t\}) - C(D, \bar{D}) + C^f(D, \bar{D}) \\ &= surplus(D) + C^f(D, \bar{D}). \quad \square \end{aligned}$$

Definition 4.2. For a pseudoflow f saturating $A(s) \cup A(t)$ and a normalized tree T, we define $surplus^T(D) = excess(D) - C^f((D, \bar{D}) \cap T)$.

From this definition and Lemma 4.1, it follows that $excess(D) \geqslant surplus^T(D) \geqslant surplus(D)$. All three terms are equal when $C^f(D, \bar{D}) = 0$, which happens when $f(\bar{D}, D) = 0$ and $f(D, \bar{D}) = C(D, \bar{D})$.

Lemma 4.2. *For a normalized tree T with pseudoflow f, strong nodes S and weak nodes $\bar{S}$,* $\max_{D\subseteq V}$ *$surplus^T(D) = surplus^T(S)$.*

Proof. The maximum of $excess(D)$ is attained for a set of nodes D^* that contains all the nodes of positive excess—the roots of the strong branches—and excludes all nodes of negative excess—the roots of weak branches. Recall that all nonroot nodes have zero excess.

Suppose that for a strong branch $B \subseteq S$, only a proper subset $B_1 \subset B$ is contained in D^*, where B_1 contains the root of the branch. Then, the set of residual arcs in $(B_1, \bar{B}_1) \cap T$ is nonempty due to Property 3, and $surplus(B_1) \leqslant surplus(B)$. Thus, B maximizes the value $surplus^T(D)$ for any $D \subseteq B$. Similarly, including any subset of a weak branch that does not include the weak root cannot increase the value of $\text{surplus}^T(D^*)$. Therefore, the maximum is attained for the set of all strong nodes S. $\square$

In particular, observing that $(S, \bar{S}) \cap T = \varnothing$, it follows that

$$excess(S) = \max_{D\subseteq V} surplus^T(D) \geqslant \max_{D\subseteq V} surplus(D).$$

Thus, the excess of the strong nodes S is an upper bound on the value of the optimal solution to the blocking-cut problem, and the superoptimality is proved. $\square$

When the residual capacity of arcs in $(S, \bar{S})$ is zero, then $excess(S) = surplus^T(S) = surplus(S)$. With this and Lemma 4.2, we have:

COROLLARY 4.1 (OPTIMALITY CONDITION). *For a normalized tree T with a pseudoflow f saturating $A(s)$ and $A(t)$ and a set of strong nodes S, if $C^f(S, \bar{S}) = 0$, then S is a maximum surplus set in the graph and $(S, \bar{S})$ is a maximum blocking cut.*

DEFINITION 4.3. A normalized tree with pseudoflow f is *optimal* if for the set of strong nodes S in the tree $(S, \bar{S}) \cap A_f = \varnothing$.

COROLLARY 4.2 (MINIMALITY). *If S is the set of strong nodes for an optimal normalized tree, then it is a minimal maximum surplus set in the graph.*

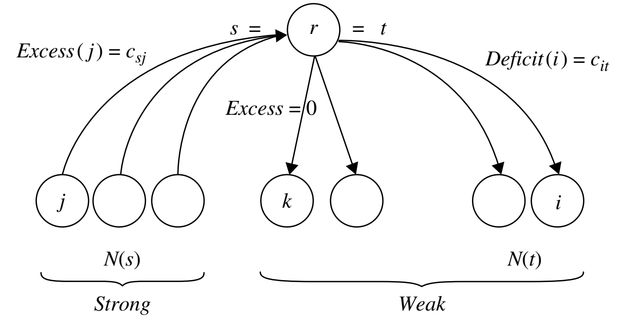
PROOF. From the proof of Lemma 4.2, the maximum surplus set contains all strong nodes. Therefore, a set strictly contained in S cannot be a maximum surplus set. □

5. The Description of the Generic Pseudoflow Algorithm

The algorithm begins with a normalized tree and an associated pseudoflow saturating source and sink adjacent arcs in G_{st}. An iteration of the algorithm consists of seeking a residual arc from a strong node to a weak node—a *merger arc*. If such an arc does not exist, the normalized tree is optimal. Otherwise, the selected merger arc is appended to the tree, the excess arc of the strong merger branch is removed, and the strong branch is merged with the weak branch. The *entire* excess of the respective strong branch is then pushed along the unique path from the root of the strong branch to the root of the weak branch. Any arc encountered along this path that does not have sufficient residual capacity to accommodate the amount pushed is *split* and the tail node of that arc becomes a root of a new strong branch with excess equal to the difference between the amount pushed and the residual capacity. The process of pushing excess and splitting is called *normalization*. The residual capacity of the split arc is pushed further until it either reaches another arc to split or the deficit arc adjacent to the root of the weak branch.

To initialize the algorithm, we need a normalized tree. One normalized tree, called a *simple normalized tree*, corresponds to a pseudoflow in G_{st} saturating all arcs $A(s)$ and $A(t)$ and zero on all other arcs. In a simple normalized tree, each node is a singleton branch for which the node serves as a root as in Figure 4. Thus, all nodes adjacent to source are strong nodes, and all those adjacent to sink are weak nodes. All the remaining nodes have zero inflow and outflow, and are thus of zero deficit and set as weak. The following procedure outputs a simple normalized tree.

Figure 4. A simple normalized tree where $N(s)/N(t)$ are the nodes adjacent to s/t, respectively.



procedure Initialize $\{G_{st}\}$
$S = W = \varnothing$.
begin
$\forall (i, j) \in A$, $f_{ij} = 0$.
$\forall (s, j) \in A(s)$, $f_{sj} = c_{sj}$; $r_j = j$; $excess(r_j) = f_{r_j, r} = c_{sj}$;
  $S \leftarrow S \cup \{j\}$.
$\forall (j, t) \in A(t)$, $f_{jt} = c_{jt}$; $r_j = j$; $excess(r_j) = -f_{r, r_j} = -c_{jt}$;
  $W \leftarrow W \cup \{j\}$.
$\forall j \in V \backslash \{S \cup W\}$; $W \leftarrow W \cup \{j\}$; $r_j = j$;
  $excess(r_j) = f_{r, r_j} = 0$.
Output $T = \bigcup_{j \in V} [r, r_j]$, S, W.
end

Another type of normalized tree, the *saturate-all tree*, is generated by a pseudoflow saturating *all* arcs in the graph G_{st}. Here, the branches are also singletons, and the strong nodes are those that have incoming capacity greater than outgoing capacity. Other types of initial normalized trees are described in Anderson and Hochbaum (2002).

We first present the generic version of the pseudoflow algorithm which does not specify which merger arc to select. We later elaborate on selection rules for a merger arc that lead to more efficient variants of the algorithm.

The input parameters to procedure pseudoflow consist of a pseudoflow f in G_{st} saturating $A(s) \cup A(t)$, a normalized tree T associated with f, and the respective sets of strong and weak nodes, S and W. As before, A_f is the set of residual arcs in A with respect to f.

procedure pseudoflow $\{G_{st}, f, T, S, W\}$
begin
  while $(S, W) \cap A_f \neq \varnothing$ do
    Select $(s', w) \in (S, W)$;
    Let $r_{s'}$, r_w be the roots of the branches containing s' and w, respectively.
    Let $\delta = excess(r_{s'}) = f_{r_{s'}, r}$;
    Merge $T \leftarrow T \backslash [r, r_{s'}] \cup (s', w)$;
    Renormalize {Push δ units of flow along the path $[r_{s'}, \ldots, s', w, \ldots, r_w, r]$:}
    $i = 1$;
    Until $v_{i+1} = r$;
      Let $[v_i, v_{i+1}]$ be the ith edge on the path;

{Push flow} If $c^f_{v_i, v_{i+1}} \geqslant \delta$ augment flow by δ,
$f_{v_i, v_{i+1}} \leftarrow f_{v_i, v_{i+1}} + \delta$;
Else, split $\{(v_i, v_{i+1}), \delta - c^f_{v_i, v_{i+1}}\}$;
Set $\delta \leftarrow c^f_{v_i, v_{i+1}}$;
Set $f_{v_i, v_{i+1}} \leftarrow c_{v_i, v_{i+1}}$;
$i \leftarrow i+1$
end
end
end

procedure split $\{(a, b), M\}$
$T \leftarrow T \backslash (a, b) \cup (a, r)$; $excess(a) = f_{ar} = M$;
{a is a root of a strong branch}
$A_f \leftarrow A_f \cup \{(b, a)\} \backslash \{(a, b)\}$;
end

An illustration of the steps at one iteration is given in Figure 5. The merger arc is (s, w). In Figure 5(a), the weak and strong merger branches, T_{r_w} and T_{r_s}, are shown. The subtrees T_w and T_s are contained in those branches. Figure 5(b) shows the inversion of the path $[r_s, \ldots, s]$ in the strong branch that follows a merger. Edges e_1 and e_2 are the only blocking edges found in the normalization process, and the new partition following the split of those edges is shown in Figure 5(c). The incoming flow into b is the excess $f_{r_s, r}$ and the incoming flow into d is the amount that got through (b, a), c^f_{ba}.

The correctness of the generic algorithm follows first from Lemma 6.1, shown in §6, proving that the algorithm terminates. Second, at each iteration the excess is pushed and arcs split so that the resulting tree is normalized and all nonroot nodes satisfy flow-balance constraints in G_{st}. The algorithm thus terminates with a normalized tree and no residual arc between strong and weak nodes. This is the certificate of optimality given in Corollary 4.1. Therefore, at termination the set of strong nodes is the source set of a minimum-cut and a maximum-surplus set.

6. The Complexity of the Pseudoflow Algorithm

6.1. The Pseudopolynomiality of the Generic Algorithm

The next lemma demonstrates that for integer capacities, the generic algorithm is finite.

Lemma 6.1. *At each iteration, either the total excess of the strong nodes is strictly reduced, or at least one weak node becomes strong.*

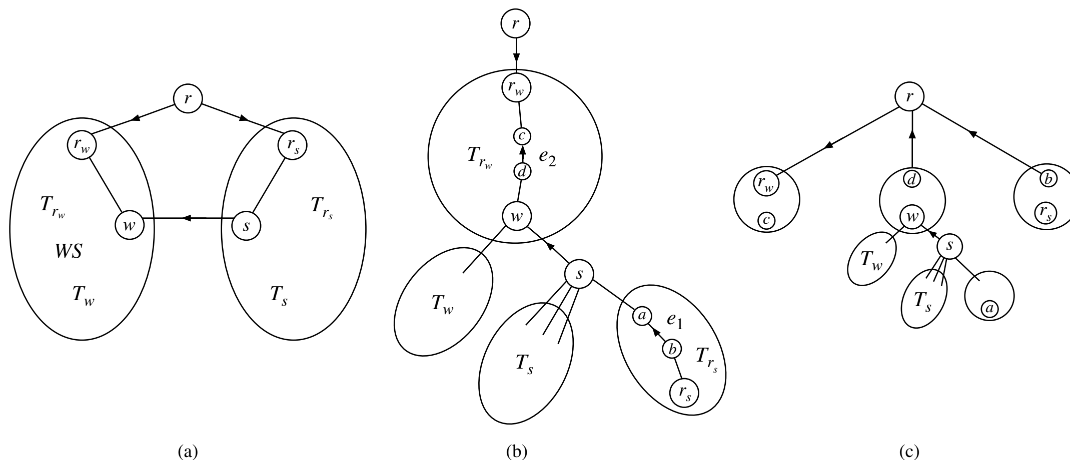
Proof. Because downwards residual capacities are positive, a positive portion of the excess pushed arrives at the weak branch. Then, either a positive amount of excess arrives at r_w, or some upwards arc $(u, p(u))$ in the weak branch has zero residual capacity. In the first case, a positive amount of excess arrives at node r_w and the total excess is strictly reduced. In the latter case, there is no change in excess but the nodes of the subtree $T_{[u, p(u)]}$ that include the head of the merger arc, w, become strong. □

Let $M^+ = C(\{s\}, V)$ be the sum of arc capacities in $A(s)$ and $M^- = C(V, \{t\})$ be the sum of arc capacities in $A(t)$. These quantities are the total excess of the strong nodes and the total deficit of the weak nodes in the initial simple normalized tree. From Lemma 6.1, it follows that two consecutive reductions in total excess may be separated by at most n mergers because each merger that is not associated with a strict reduction in excess must result in a decrease in the number of weak nodes. Thus, we have for integer capacities:

Corollary 6.1. *The complexity of the algorithm is $O(nM^+)$ iterations.*

The complexity expression in Corollary 6.1 depends on the total sum of excesses in the initial tree. It is thus

Figure 5. The merger arc is (s, w); r_s and r_w are the roots of the strong and weak branches respectively; $e_1 = (b, a)$ is the first split arc on the merger path; $e_2 = (d, e)$ is the second split arc. (a) Merger arc identified. (b) Inverting the strong branch. (c) After renormalization.



possible to take advantage of the symmetry of source and sink and solve the problem on the reverse graph—reversing all arcs and the roles of source and sink—resulting in $O(nM^-)$ iterations. Therefore, the total number of iterations is $O(n\min\{M^+, M^-\})$.

Corollary 6.2. *The complexity of the algorithm is* $O(n\min\{M^+, M^-\})$ *iterations.*

It is important to note that even though the algorithm terminates when total excess is zero, this is only a sufficient condition for termination, not a necessary condition. At termination, both excess and deficit may be positive, as long as the cut arcs (S, W) are all saturated. This observation leads to another corollary.

A procedure for feasible flow recovery is given in §8. The feasible flow recovered in G_{st} has the flow on all out-of-tree arcs unchanged. In particular, for an optimal normalized tree, the flow saturates all arcs in (S, W) and (S, W) is a minimum cut. We conclude that for optimum minimum-cut value $C(S, W) = C^*$, the remaining excess at termination is $M^+ - C^*$. We then get a somewhat tighter complexity expression,

Corollary 6.3. *Let the minimum-cut capacity be* C^*. *Then, the complexity of the algorithm is* $O(nC^*)$ *iterations.*

6.2. A Scaling Polynomial-Time Improvement

Using a standard technique for scaling integer arc capacities, the running time of the algorithm becomes polynomial. This works as follows: Let $P = \lceil \log_2(\max_{(i,j)\in A_{st}}\{c_{ij}\})\rceil - 1$. Then, at each scaling iteration p, $p = P, P-1, \ldots, 1, 0$, the problem is solved with arc capacities, $\bar{c}_{ij} = \bar{c}_{ij} \leftarrow \lfloor c_{ij}/2^p \rfloor$ for all $(i, j) \in A_{st}$. At the next iteration, the value of p is reduced by one, thus adding one more significant bit to the value of the capacities.

Now between two consecutive scaling iterations, the value of the residual cut is increased by at most m scaling units. This is because the residual cut capacity at the end of the previous scaling iteration is zero, and when the additional bit is added to the capacities of at most m arcs, the value of the minimum cut in the graph is bounded by this residual cut. With Corollary 6.3, this implies a running time of $O(mn)$ iterations per scaling step. Because there are $O(\log(\min\{M^+, M^-\}))$ scaling steps, we have

Corollary 6.4. *The complexity of the scaling pseudoflow algorithm is* $O(mn\log\min\{M^+, M^-\})$ *iterations.*

6.3. A Strongly Polynomial Labeling Scheme

We describe here a labeling scheme for the pseudoflow algorithm, and show that it satisfies four invariants. The pseudoflow algorithm satisfying these invariants is shown to have complexity of $O(mn\log n)$, which is strongly polynomial.

In this section, a merger arc (s, w) has node s (which is unrelated to the source node), the strong endpoint of the merger arc, called the *strong merger node*, and w, called the *weak merger node*.

Our labeling scheme is similar, but not identical, to the distance labeling used in Goldberg and Tarjan (1988). The basic labeling scheme restricts the selection of a merger arc (s, w) so that w is a lowest label weak node among all possible residual arcs in (S, W).

Initially, all nodes are assigned the label 1, $l_v = 1$ $\forall v \in V$. After a merger iteration involving the merger arc (s, w), the label of all strong nodes including the strong merger node s can only increase to the label of w plus 1. Formally, the statement Select $(s, w) \in (S, W)$ is replaced by

Select $(s, w) \in (S, W)$, so that $l_w = \min_{(u,v)\in(S,W)} l_v$;
{relabel} $\forall v \in S$, $l_v \leftarrow \max\{l_v, l_w + 1\}$.

The labels satisfy the following invariants throughout the execution of the algorithm:

Invariant 1. For every arc $(u, v) \in A_f$, $l_u \leqslant l_v + 1$.

Invariant 2 (Monotonicity). Labels of nodes on any downwards path in a branch are nondecreasing.

Invariant 3. The labels of all nodes are lower bounds on their distance to the sink. Furthermore, the difference between the labels of any pair of nodes u and v, $l_u - l_v$, is a lower bound on the shortest residual path distance from u to v.

Invariant 4. Labels of nodes are nondecreasing over the execution of the algorithm.

We now prove the validity of the invariants:

Proof of Invariant 1. Assume by induction that the invariant holds through iteration k, and prove that it holds through iteration $k+1$ as well. Obviously, the invariant is satisfied at the outset, when all labels are equal to one. Consider a residual arc (u, v) after the relabeling at iteration $k+1$ and let arc (s, w) be the merger arc in that iteration. Let l_u, l_v be the labels prior to the relabeling in iteration $k+1$, and l'_u, l'_v be the labels after the relabeling is complete. There are four different cases corresponding to the status of the nodes at the beginning of iteration $k+1$.

u strong, v weak: At iteration $k+1$, only the label of node u can change because weak nodes are not relabeled. The lowest label choice of w implies that $l_v \geqslant l_w$, and therefore $l'_u = \max\{l_u, l_w + 1\} \leqslant l_v + 1 = l'_v + 1$.

u strong, v strong: Here, the inequality can potentially be violated only when the label of u goes up and the label of v does not. Suppose that the label of u has increased. Then, $l'_u = l_w + 1$. If the label of v has not likewise increased, then $l_v \geqslant l_w + 1$ and $l'_v = l_v \geqslant l_w + 1 = l'_u$, so the inequality is satisfied.

u weak, v strong: Only the label of v can change, and then only upwards.

u weak, v weak: Weak nodes do not get relabeled, so, by induction, the inequality is still satisfied at the end of iteration $k+1$. □

Proof of Invariant 2. Assume, by induction, that monotonicity is satisfied through iteration k. The operations that might affect monotonicity at iteration $k+1$ are relabeling, merging, and splitting of branches. As a result of relabeling, the nodes on the strong section of the merger path $[r_s, \ldots, s]$ are all labeled with the label $l_w + 1$ because previously all the labels of these nodes were $\leqslant l_s$ by the inductive assumption of monotonicity. After merging and inverting the strong branch, that path has the roles of parents and children reversed along the path. But because the nodes along the strong section of the path all have the same labels, the monotonicity still holds. Monotonicity also holds for all subtrees that are suspended from the merger path nodes because the parent/child relationship is not modified there, and all labels $\geqslant l_w + 1$. □

Proof of Invariant 3. This is a corollary of Invariant 1. Along a residual path, labels increase by at most one unit for each arc on the path. Therefore, the difference between the labels of the endpoints is less than or equal to the distance between them along any residual path. Formally, for a residual path on k nodes $(v_1, v_2, \ldots, v_k)$, we have

$$l_{v_1} \leqslant l_{v_2} + 1 \leqslant \cdots \leqslant l_{v_k} + k - 1.$$

Therefore, $l_{v_1} - l_{v_k} \leqslant k - 1$. □

Invariant 4 is obviously satisfied because relabeling can only increase labels.

Lemma 6.2. *Between two consecutive mergers using merger arc (s, w), the labels of s and w must increase by at least one unit each.*

Proof. Let the label of w be $l_w = L$ during the first merger using (s, w). After the merger's relabeling, $l_s \geqslant L+1$. Prior to (s, w) serving again as a merger arc, flow must be pushed back on (w, s) so that (s, w) may become an out-of-tree residual arc. This can happen if

- w is above s in a strong branch and the strong merger node is either s or a descendant of s. After such a merger, the label of w must satisfy $l_w = l_s \geqslant L+1$. Or,
- (w, s) serves as a merger arc. But then w is relabeled to be at least $l_s + 1$ and $l_w \geqslant l_s + 1 \geqslant L + 2$.

In either case, $l_w \geqslant l_s \geqslant L+1$.

Upon repeated use of (s, w) as a merger arc, $l_s \geqslant l_w + 1 \geqslant L+2$. Thus, the labels of s and w must increase by at least one each between the consecutive mergers. □

Invariants 3 and 4 imply that no label can exceed n because a label of a node increases only if there is a residual path from the node to a weak root. It follows that each arc can serve as merger arc at most $n-1$ times throughout the algorithm.

Corollary 6.5. *The labeling pseudoflow algorithm executes at most $O(mn)$ mergers.*

We now bound the number of edge splits throughout the algorithm.

Corollary 6.6. *There are at most $O(mn)$ calls to procedure* split *throughout the execution of the pseudoflow algorithm.*

Proof. At any iteration, there are at most n branches because there are only n nodes in the graph. Each call to split creates an additional branch. The number of branches after a merger can:

- Increase, when there are at least two splits.
- Remain unchanged, when there is exactly one split.
- Decrease by one when there is no edge split. In this case, all the strong nodes in the branch containing the strong merger node become weak.

Because there are only $O(mn)$ iterations, the total accumulated decrease in the number of branches can be at most $O(mn)$ throughout the algorithm. Therefore, there can be at most $O(mn+n)$ edge splits. □

6.3.1. Data Structures. We maintain a set of n buckets, where bucket k, B_k contains all the k-labeled branch roots. The buckets are updated in the following cases:

(1) There is a merger and the root of the strong branch no longer serves as root. Here, this root is removed from its bucket.

(2) A root is relabeled, and then moved to another, higher label bucket.

(3) A split operation creates a new root node. Here, the new root node is inserted into the bucket of its label.

The number of bucket updates is thus dominated by the complexity of the number of mergers and the number of edge splits $O(mn)$.

For the tree operations, we use the data structure called *dynamic trees* devised by Sleator and Tarjan (1983, 1985). Dynamic trees is a data structure that manages a collection of node disjoint trees. Among the operations enabled by the dynamic trees data structure are:

findroot(v)—find the root of the tree that v belongs to.

findmin(v)—find the minimum key value on the path from v to the root of its tree.

addcost(v, δ)—add the value δ to the keys of all nodes on the path from v to the root of its tree.

invert(v)—invert the tree that v belongs to so it is rooted at v instead of at *findroot*(v).

merge(u, v)—link a tree rooted at u with node v of another tree so that v becomes the parent of u.

split$(u, p(u))$—split the tree that u and $p(u)$ belong to so that the descendants of u form a separate tree T_u.

All these operations, and several others, can be performed in time $O(\log n)$ per operation (either in amortized time or in worst case depending on the version of dynamic trees implemented; see Sleator and Tarjan 1983, 1985). The only operation required for the pseudoflow algorithm which is not directly available is the operation of finding the next edge along the merger path that has residual capacity less than the amount of excess δ. We call the operation that finds the first arc on the path from v to the root of the tree with residual capacity less than δ, FIND-FIRST(v, δ).

We note that FIND-FIRST can be implemented as a minor modification of *findmin*(v), and in the same complexity as *findmin*(v), $O(\log n)$.

LEMMA 6.3. *The complexity of the labeling pseudoflow algorithm is* $O(mn \log n)$.

PROOF. The number of merger iterations is at most $O(mn)$ and the number of splits is $O(mn)$. Each iteration requires:

(1) Identifying a merger arc (s, w), if one exists, with w of lowest label.
(2) Inverting and merging trees.
(3) Updating residual capacities along the merger path.
(4) Finding the next split edge on the merger path.

For operation (1), a merger arc can be identified efficiently using several possible data structures. We show, in the next section, that in all the labeling variants the search for a merger arc initiates at a *strong* node of a specific label. It is then established that scanning all arcs adjacent to all strong nodes of label l requires only $O(m)$ operations. This results in $O(mn)$ steps to identify all merger arcs throughout the algorithm, or $O(1)$ steps on average per merger arc.

Operations (2) and (3) use dynamic trees, and operation (4) is the FIND-FIRST operation. The complexity of each of these operations is $O(\log n)$. The total complexity is therefore $O(mn \log n)$. ☐

7. Strongly Polynomial Variants

7.1. The Lowest Label Variant

Under the *lowest label variant*, the selection rule depends on the labels of strong nodes. The merger arc is selected between a lowest label strong node of label l and a node of label $l-1$:

Select $(s, w) \in A_f$ for $s \in S$ satisfying $l_s = \min_{v \in S} l_v$ and $l_w = l_s - 1$;
{relabel} If no such arc exists and $\forall\, ch(s)$, $l_{ch(s)} \geqslant l_s + 1$, relabel $l_s \leftarrow l_s + 1$.

Because s is of lowest label among the strong nodes, the node labeled $l-1$ is necessarily weak. The relabel is an increase of the label of a single strong node by one unit when it has no neighbor of lower label and all its children in the strong branch have larger labels.

LEMMA 7.1. *Invariants* 1, 2, 3, *and* 4 *hold for the lowest label variant.*

PROOF. Invariant 1 holds because the relabel of u occurs only when for all residual arcs, (u, v) $l_u < l_v + 1$. After increasing the label of u by one unit, the invariant inequality still holds.

The relabel operation satisfies the monotonicity invariant, Invariant 2, by construction. Invariant 3 is satisfied because it is a corollary of Invariant 1, and Invariant 4 is satisfied by construction. ☐

The search for a merger arc is implemented efficiently utilizing the monotonicity property. The strong nodes are scanned in depth-first-search (DFS) order starting from a lowest label root of a strong branch. Such a root node is found easily in the lowest label nonempty bucket. Each *backtrack* in the strong branch is associated with a relabel of the corresponding node. For each node, we maintain a *neighbor-pointer* to the last scanned neighbor in the adjacency list since the last relabel. When a node is relabeled, this pointer is reset to the start position. A node that has its pointer at the end position and has had all its neighbors scanned for a potential merger is a candidate for relabeling.

Maintaining the status of out-of-tree arcs is easy as an arc changes its status either in a merger, when the merger arc becomes an in-tree arc; or in a split, when an in-tree arc becomes out-of-tree. Either one of these cases happens only $O(mn)$ times throughout the algorithm and the update of status for each takes $O(1)$. To summarize,

LEMMA 7.2. *Finding all merger arcs throughout a phase requires at most* $O(m)$ *operations for a total of* $O(mn)$ *operations for the entire algorithm.*

PROOF. Let *phase* l be the collection of iterations when the lowest label among the strong nodes is l. From Invariant 3, it follows that there are no more than n phases. The DFS process scans all arcs in the normalized tree at most twice per phase, and all out-of-tree arcs are scanned at most once per phase. Therefore, during a single phase, the total complexity of searching for merger arcs is $O(m)$. ☐

7.2. The Highest Label Variant

In the highest label variant, we select as strong merger branch the one that has a *highest label root node*. The mergers are still performed from a node of lowest label l in the branch (rather than among all strong nodes) to a node of label $l-1$, and the relabeling rule is identical. Unlike the lowest label variant, the head of the merger arc may not be a weak node. Still, all the invariants hold as the proof of Lemma 7.1 applies, and so does the complexity analysis. Note that the search for merger arcs at each phase is accomplished in $O(m)$ as it relies only on the monotonicity invariant.

7.3. A Hybrid Variant

Here, any strong branch can be selected to initiate the mergers from. The merger sought at each iteration is from a lowest label strong node in the selected branch, and thus of label identical to the label of the root of the branch. As before, all the variants and the complexity analysis are the same as for the lowest label variant.

7.4. Free Arcs Normalized Tree

An arc (i, j) is said to be *free* with respect to a pseudoflow f if it is residual in both directions, $0 < f_{ij} < u_{ij}$. In the pseudoflow algorithm, a split occurs on arc (i, j)

on the merger path when the amount of pushed excess δ strictly exceeds the residual capacity c^f_{ij}. Therefore, when $\delta = c^f_{ij}$, then arc (i, j) remains in the tree but is not free. The free arcs variant of the algorithm splits an arc if the excess pushed is greater or equal to the residual capacity. The split branch resulting when the excess is equal to the residual capacity has zero excess and is considered weak. With this splitting rule, all in-tree arcs are free.

The free arcs variant tends to create smaller branches. Note that the weakly polynomial complexity bound resulting from Lemma 6.1 does not apply for this case. The free arcs variant was used for the bipartite matching pseudoflow algorithm in Hochbaum and Chandran (2004), leading to particularly small (up to three nodes) branches.

7.5. Global Relabeling

Because labels are bounded by the residual distance to the sink, we can use a process of assigning labels that are equal to the respective distances to the sink, provided that monotonicity is satisfied in each branch. These labels are found using breadth-first-search (BFS) from the sink in the residual network. To preserve monotonicity, the label assigned to each node is the minimum between its distance to sink and the maximum label of its children in the branch. We call this labeling process *global relabeling*, after an analogous process used for push-relabel. The frequency of use of global relabeling should be balanced against the increased complexity of performing BFS in the residual network to identify the distance labels.

7.6. Delayed Normalization

A heuristic idea is to process a merger through merger arc (s, w) while normalizing for the strong portion of the path only, from r_s to s and w. The normalization process in the weak section of the merger path is delayed and the excess at node w is recorded. After several mergers have been performed, the normalization of the weak branches is executed jointly for all the weak nodes by a single scan of the weak branches. If several strong branches were merged to the same weak branch, the weak sections of their merger paths overlap. In that case, instead of normalizing each path separately, we normalize them jointly for a potential improvement in the running time. The extent of improvement in performance depends on the magnitude of the overlap and on the overhead required.

8. Flow Recovery

At termination of the pseudoflow algorithm, we have a minimum cut, but not a maximum feasible flow. We show in this section how to construct a feasible flow from any pseudoflow and a corresponding normalized tree, not necessarily optimal.

Definition 8.1. A feasible flow f' is said to be *associated* with a normalized tree T and pseudoflow f if all free arcs of f' are in T, and if for all out-of-tree arcs $(i, j) \in A\backslash T$, $f'_{ij} = f_{ij}$.

If the normalized tree is optimal, then the cut between strong and weak nodes is saturated and therefore the corresponding feasible flow is maximum.

Theorem 8.1. *Every normalized tree with pseudoflow f has a feasible flow in G_{st} associated with it that can be constructed in $O(m \log n)$ time.*

The following lemma is needed for the proof of the theorem. The concept of "strictly strong" or "strictly weak" node refers to a node in a strong (respectively, weak) branch with strictly positive excess (respectively, deficit).

Lemma 8.1. *For any strictly strong node, there exists a residual path either to the source or to some strictly weak node.*

Proof. Suppose not. Let v be a strictly strong node for which there is no residual path to the source or to a strictly weak node. Therefore, the set of nodes reachable from v in the residual graph, $R(v)$, includes only nodes with nonnegative excess.

Because no node is reachable from $R(v)$ in the residual graph, $\mathit{inflow}(R(v)) = 0$, and thus in particular, $\mathit{inflow}(R(v)) - \mathit{outflow}(R(v)) \leqslant 0$. On the other hand, for all $u \in R(v)$, the excess is nonnegative, $\mathit{inflow}(u) - \mathit{outflow}(u) \geqslant 0$. Now, because $v \in R(v)$,

$$0 \geqslant \mathit{inflow}(R(v)) - \mathit{outflow}(R(v)) = \sum_{u \in R(v)} [\mathit{inflow}(u) - \mathit{outflow}(u)] > 0.$$

This leads to a contradiction of the assumption. □

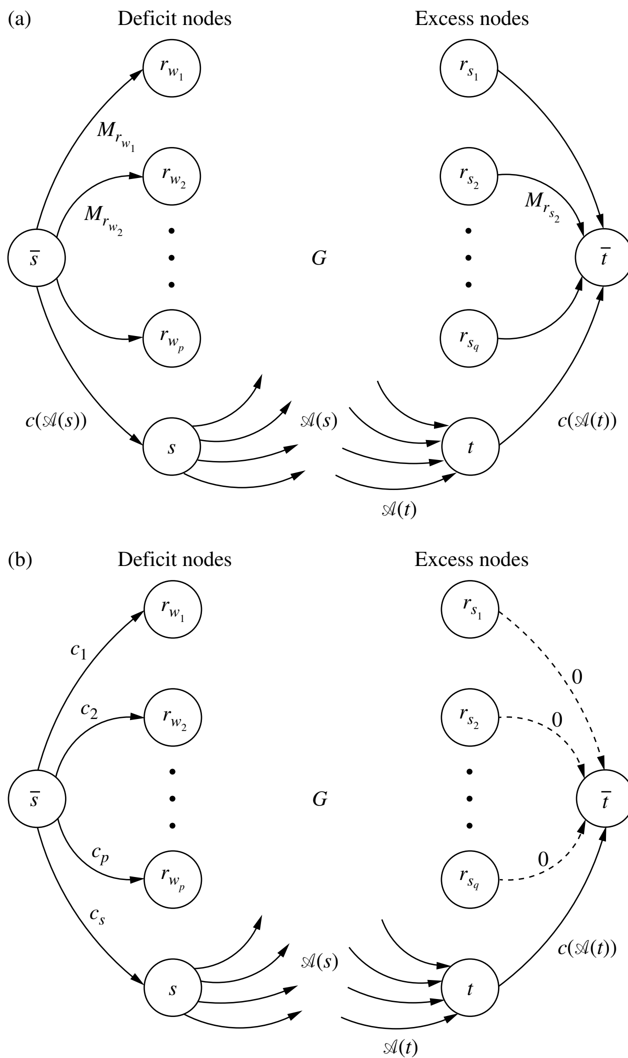
An analogous argument proves that for any strictly weak node, there exists a residual path either to the sink or to a strictly strong node.

Proof of Theorem 8.1. Given the pseudoflow f corresponding to the normalized tree, a feasible flow is constructed by eliminating the positive excess at the roots of strong branches and the negative excess at the roots of (strictly) weak branches. This is done by using a process analogous to *flow decomposition*, which decomposes a feasible flow in an s, t-graph into a collection of up to m simple s, t-paths and cycles. To use flow decomposition, we construct a graph with source and sink nodes in which the pseudoflow f is feasible by appending nodes and arcs to the graph G_{st} as follows: let all strictly strong nodes in G_{st} with respect to f be adjacent to a sink node $\bar{t}$ with arcs from the excess nodes to $\bar{t}$ carrying the amount of excess. These excess nodes include the node t with excess equal to the total capacity of $A(t)$, $C(A(t))$. All strictly weak nodes have arcs incoming to them from a node $\bar{s}$ carrying the deficit flow. The deficit nodes include the source node s with deficit equal to the capacity of the arcs $A(s)$. The resulting graph has a feasible flow from $\bar{s}$ to $\bar{t}$. Such a graph

is shown in Figure 6(a) (where the quantity $|excess(v)|$ is denoted by M_v).

We first decompose the sum of all excesses (which is the portion of the flow from the excess nodes other than t to $\bar{t}$). This is done by reversing the graph and the flows on each arc with the amount of flow becoming the residual capacity of each arc in the opposite direction. This graph contains a residual path from every excess node to $\bar{s}$ as proved in Lemma 8.1. Once all the excesses have been disposed of, there may be some strictly positive deficits left. These are denoted by the flows c_j in Figure 6(b). All these deficits must now reach $\bar{t}$ via t in the residual graph because there are no other positive excess nodes left. Again, flow decomposition is employed to send these deficits to $\bar{t}$.

Figure 6. The graph in which flow decomposition generates the corresponding feasible flow. Here, $M_v = |excess(v)|$. (a) The graph in which excesses are eliminated. (b) The graph after excesses have been eliminated and before applying flow decomposition to eliminate deficits.



Finding the flow paths can be done using DFS, where at each iteration the procedure identifies a cycle or a path along which flow is pushed back and eliminates at least one bottleneck arc. The complexity of this DFS procedure is $O(mn)$. A more efficient algorithm for flow decomposition using dynamic trees was devised by Sleator and Tarjan (1985). The complexity of that algorithm is $O(m \log n)$. □

In an optimal normalized tree, with a set of strong nodes S, there are no residual arcs in $(S, \bar{S})$. In that case, it follows from Lemma 8.1 that all positive excess is sent back to the source s using paths traversing strong nodes only, and all positive deficit is sent back to sink t via weak nodes only. Thus, for an optimal normalized tree, the pseudoflow saturates $(S, \bar{S})$, and the associated feasible flow also saturates the arcs $(S, \bar{S})$. So, the flow on $(S, \bar{S})$ is equal to the capacity of the cut $C(S, \bar{S})$. Given the maximum blocking-cut solution and minimum cut, the maximum flow is therefore found in time $O(m \log n)$, as proved in Theorem 8.1. This is an alternative proof to the optimality of a normalized tree with $C^f(S, \bar{S}) = 0$.

Remark 8.1. For closure graphs—graphs that have all arcs not adjacent to source and sink, with infinite capacity—the pseudoflow on the normalized tree is transformed to a feasible flow in $O(n)$ time (Hochbaum 2001). That algorithm is more efficient for finding feasible flow than the algorithm for general graphs because for closure graphs, all out-of-tree arcs must have zero flow on them and the flow decomposition involves only the arcs within the respective branch. This is the case also for s, t-tree networks discussed in the online appendix.

9. The Parametric Pseudoflow Algorithm

Whereas parametric analysis for general linear programming problems is restricted to a sequence of changes of one parameter at a time, for the maximum-flow problem parametric changes can be made simultaneously to all source adjacent and sink adjacent capacities provided that the changes are monotone nondecreasing on one side and monotone nonincreasing on the other. In our discussion, the capacities of arcs adjacent to source are monotone nondecreasing functions of the parameter λ, e.g., $a_{sj} + \lambda b_{sj}$ for $b_{sj} \geqslant 0$, and the capacities of arcs adjacent to sink are monotone nonincreasing functions, e.g., $a_{it} - \lambda b_{it}$ for $b_{it} \geqslant 0$. Gallo et al. (1989) showed how to solve the parametric maximum-flow problem for a sequence of k parameter values with the push-relabel algorithm in $O(mn \log n^2/m + km \log n^2/m)$ steps. For $k = O(n)$, this complexity is the same as required for solving a constant capacity maximum-flow instance (referred to here as a *single instance*).

As in the introduction, the parametric analysis for a given set of sorted parameter values, $\lambda_1 < \lambda_2 < \cdots < \lambda_k$, is referred to as the *simple sensitivity analysis*. The *complete parametric analysis* is more general and generates *all* possible breakpoints $b_1 < b_2 < \cdots < b_q$, at which the

minimal source set of a minimum cut is changing. (From Lemma 9.1, $q \leqslant n$.) A complete parametric analysis solution provides the solutions for any parameter value λ. Specifically, if $\lambda \in [b_l, b_{l+1})$, then $(S_{b_l}, \bar{S}_{b_l})$—a minimum cut for the parameter b_l—is also a minimum cut for λ. In other words, the optimal solution for any given parameter value is found from the complete parametric analysis output by identifying the consecutive pair of breakpoints between which the value lies.

To date, only the push-relabel algorithm has been shown to solve the complete parametric analysis efficiently, in the same complexity as a single instance. This was shown in Gallo et al. (1989) for linear functions of the parameter. In Hochbaum (2003), we demonstrated that the complete parametric analysis can be implemented for *any* monotone functions (for both push-relabel and pseudoflow) with an additive run time of $O(n \log U)$, where U is the range of the parameter. This additive run time is the complexity of finding zeroes of the monotone functions, which is provably impossible to avoid (Hochbaum 2003).

We first show that the pseudoflow algorithm solves the simple sensitivity analysis in $O(mn \log n + kn)$ time, which is the complexity of solving a single instance for $k = m \log n$. The pseudoflow algorithm is then shown to solve the complete parametric analysis in $O(mn \log n)$ for linear functions, and with an additive run time of $O(n \log U)$ for arbitrary monotone functions. The pseudoflow-simplex algorithm is later shown to solve the respective parametric problems in the same complexities as the pseudoflow algorithm. It is the only known simplex algorithm that solves the simple sensitivity analysis in the same complexity as a single instance, and substantially faster than other simplex algorithms. Moreover, the pseudoflow-simplex algorithm is the only simplex algorithm known to solve the complete parametric analysis efficiently.

Let S_λ be a minimal (maximal) source set of a minimum cut in the graph G_λ in which the capacities are set as a function of λ. It is well known (e.g., Gale 1957) that as λ increases, so does the set S_λ. This can also be deduced from the construction of an optimal normalized tree: As λ increases, all excesses of branches can only go up. So, strong branches can only become "stronger," while some weak branches may become strong or have lesser deficit. (The procedure of adjusting the normalized tree for changes of capacities, *renormalization*, is given in the online appendix.)

Lemma 9.1 (Nestedness). *For k parameter values $\lambda_1 < \lambda_2 < \cdots < \lambda_k$, the corresponding minimal (maximal) source set minimum cuts satisfy $S_{\lambda_1} \subseteq S_{\lambda_2} \subseteq \cdots \subseteq S_{\lambda_k}$.*

Corollary 9.1 (Contraction Corollary). *For $\lambda \in (\lambda_1, \lambda_2)$, a source set of a minimum cut S_λ in the graph G_λ in which the set S_{λ_1} is contracted with the source and $\bar{S}_{\lambda_2} - S_{\lambda_1}$ is contracted with the sink, is also a source set of a minimum cut in G_λ.*

The key to the efficiency of the parametric solution is to leave the distance labels unchanged between consecutive calls to monotonically increasing values of the parameter. Adjusting the graph for a new parameter value in the push-relabel algorithm requires $O(m \log n^2/m)$ time for a total of $O(mn \log n^2/m + km \log n^2/m)$. For the pseudoflow algorithm, the normalized tree remains the same except that it may require renormalization at a complexity of $O(n)$ when the value of the parameter is increased. The running time is then $O(mn \log n + kn)$.

We now sketch briefly the main concepts of the complete parametric analysis algorithm of Gallo et al. (1989), mimicked here for the pseudoflow algorithm. Each call to the algorithm is made with respect to a value λ^* and an interval $[\lambda_1, \lambda_2]$, where $\lambda^* \in [\lambda_1, \lambda_2]$ and where the graph has the maximal source set of the minimum cut on G_{λ_1} shrunk with the source, and the maximal sink set of the minimum cut on G_{λ_2} shrunk with the sink. Each such call is provided with two *free runs*, one starting from G_{λ_1} and the other on the reverse graph starting from the solution on G_{λ_2}. For a given interval where we search for breakpoints, we run the algorithm twice: from the lower endpoint of the interval where the maximal source set of the cut obtained at that value is shrunk into the source, and from the highest endpoint of the interval where the maximal sink set of the cut is shrunk into the sink. The runs proceed for the graph and reverse graph until the first one is done. The newly found cut subdivides the graph into a source set and a sink set, $G^{(1)}$ and $G^{(2)}$, with n_1 and n_2 nodes, respectively, and m_1 and m_2 edges, respectively. Assuming that $n_1 \leqslant n_2$, then $n_1 \leqslant \frac{1}{2}n$. In the smaller interval, corresponding to the graph on n_1 nodes, two new runs are initiated from both endpoints. In the larger interval, however, we continue the previous runs using two properties:

Reflectivity: The complexity of the algorithm remains the same whether running it on the graph or reverse graph.

Monotonicity: Running the algorithm on a monotone sequence of parameter values has the same complexity as a single run.

The implementation of the complete parametric analysis with pseudoflow requires two minimum-cut solutions, one with minimal and one with maximal source sets. From Corollary 4.2, the set of strong nodes at the optimal solution is a minimal source set. To find a *maximal* source set minimum cut, we solve the problem in the reverse graph. Alternatively, if using the free arcs variant, the set of strong nodes appended with all zero-deficit branches that do not have residual arcs to strictly weak branches forms a maximal source set. This collection of zero-deficit branches can also be generated by a process of normalization, even if the free arcs variant is not used.

Using the reflectivity and monotonicity properties for the labeling pseudoflow algorithm, we choose a quantity Q to be $Q = c \log n$ for some constant c, $m_1 \leqslant m$. $m_2 \leqslant m$, where $n_1 + n_2 \leqslant n + 1$ and $n_1 \leqslant \frac{1}{2}n$. Let $T(m, n)$ be the running time of the labeling parametric pseudoflow algorithm on

a graph with m arcs and n nodes. The recursive equation satisfied by $T(m, n)$ is

$$T(m, n) = T(m_1, n_1) + T(m_2, n_2) + 2Qm_1n_1.$$

The solution to this recursion is $T(m, n) = O(Qmn) = O(mn \log n)$. This omits the operation of finding the value of λ^*, which is done as the intersection of the two cut functions for the parameter value λ_1 and the parameter value λ_2. This intersection is computed at most $O(n)$ times, each at $O(1)$ steps for the linear functions. Computing the intersection of arbitrary monotone functions can be done in $O(\log U)$ steps each with binary search, thus requiring an additive factor of $O(n \log U)$.

10. Pseudoflow-Simplex

A generic simplex network-flow algorithm works with a feasible flow and a spanning tree of the arcs in the basis, called the *basis tree*. All free arcs are basic, and are thus part of the basis tree. A simplex iteration is characterized by having one arc *entering* the basis, and one arc *leaving* the basis.

A normalized tree can serve as a basis tree in the extended network for a simplex algorithm that solves the maximum blocking-cut problem. A merger arc qualifies as an entering arc, but in the pseudoflow algorithm there are potentially numerous leaving arcs (split arcs). The simplex version of the pseudoflow algorithm removes the first arc with the bottleneck residual capacity on the cycle beginning at $r, r_s, \ldots, s, w, \ldots, r_w, r$. At each iteration, there is precisely one split edge that may be such that the strong merger branch is eliminated altogether when the excess arc is the bottleneck (the residual capacities of all arcs on the cycle are greater or equal to the amount of excess), or the weak merger branch is eliminated when the deficit arc is the bottleneck. The requirement to remove the first bottleneck arc is shown below to be essential to retain the downwards positive residual capacities in each branch—a property of normalized trees. Removing a first bottleneck arc on the cycle has been used previously in the concept of a *strong basis* introduced by Cunningham (1976).

Our simplex algorithm for solving the maximum blocking-cut problem is called the *pseudoflow-simplex algorithm*. Although the algorithmic steps taken by the pseudoflow algorithm and the pseudoflow-simplex on the same normalized tree can lead to different outcomes in the subsequent iteration (see §12), the complexity of both algorithms is the same.

Let an out-of-tree residual arc between a strong node s' and a weak node w be referred to as an *entering arc*. The cycle created by adding an entering arc to a normalized tree is $[r, r_{s'}, \ldots, s', w, \ldots, r_w, r]$, where r represents, as before, the root in the extended network. The largest amount of the flow that can be augmented along the cycle is the bottleneck residual capacity along the cycle. The first arc (u, v) along the cycle starting with $(r, r_{s'})$ attaining this bottleneck capacity δ is the *leaving arc*, where

$$\delta = c^f_{u,v} = \min\Big\{excess(r_{s'}), \min_{e\in[r_{s'},\ldots,s',w,\ldots,r_w]} c^f_e, deficit(r_w)\Big\}$$
$$= \min_{e\in[r, r_{s'},\ldots,s',w,\ldots,r_w, r]} c^f_e.$$

The new basis tree is $T \cup \{(s', w)\}\backslash\{[u, v]\}$. The roots of the branches remain unchanged provided that neither u nor v is the node r. If $u = r$, then the excess arc is the bottleneck arc, and the strong branch T_{r_s} is eliminated and joins the weak branch T_{r_w}. If $v = r$, then the deficit arc is the bottleneck arc, and the weak branch is eliminated and joined with T_{r_s}. This means that throughout the algorithm, the number of branches is nonincreasing.

procedure pseudoflow-simplex$\{G_{st}, f, T, S, W\}$
`begin`
  `while` $(S, W) \cap A_f \neq \varnothing$ `do`
    Select $(s', w) \in (S, W)$; $\{(s', w)$ is the entering arc.$\}$
    {Leaving arc and flow update:}
    Let δ be the minimum residual capacity along the
      cycle $[r, r_{s'}, \ldots, s', w, \ldots, r_w, r]$
      attained first for arc (u, v):
    {If $\delta > 0$ push δ units of flow along the path
      $[r_{s'}, \ldots, s', w, \ldots, r_w, r]$:}
    `Until` $v_{i+1} = r$;
      Let $[v_i, v_{i+1}]$ be the ith edge on the path;
      {Push flow} Set $f_{v_i, v_{i+1}} \leftarrow f_{v_i, v_{i+1}} + \delta$;
      If $f_{v_i, v_{i+1}} = c_{v_i, v_{i+1}}$ then
        $A_f \leftarrow A_f \cup \{(v_{i+1}, v_i)\}\backslash\{(v_i, v_{i+1})\}$;
      $i \leftarrow i + 1$
    `end`
    Set $T \leftarrow T\backslash\{(u, v)\} \cup \{(s', w)\}$.
  `end`
`end`

The four properties of a normalized tree apply at the end of each iteration: the choice of the leaving arc as the first bottleneck arc ensures that all downwards residual capacities remain positive (Property 3 of a normalized tree). This is because other arcs on this path, which after the flow update have zero residual capacity, are all in the weak side of the tree, and the zero residual capacity is in the upwards direction.

Lemma 6.1 applies to pseudoflow-simplex, so a simplex iteration results in either a strict decrease in the total (positive) excess, or at least one weak node becomes strong. The complexity of pseudoflow-simplex is thus $O(nM^+)$ iterations. Furthermore, the termination rule and all complexity improvements still apply. The use of the labeling-based selection rule for an entering arc with the dynamic trees data structure leads to precisely the same complexity as that of the labeling algorithm, $O(mn \log n)$, although some modification is required as noted next.

Although Invariants 1, 3, and 4 hold, the monotonicity invariant does not hold for pseudoflow-simplex. The

reason is that the tail of the split bottleneck arc may not become the new root of the strong tree, and no inversion takes place. So, when the bottleneck arc is within the weak branch, the labels of the weak nodes that join the strong branch are smaller than that of the strong merger node which becomes their ancestor. This lack of monotonicity disables the efficient scanning for merger arcs with DFS. Instead, we use a dynamic tree representation of the normalized tree with an additional set of key values indicating the labels of the respective nodes. Finding a minimum key-valued node in a branch requires $O(\log n)$ time at most, and the updating of these keys all can be performed within the run time dominated by the run time of the overall algorithm, and therefore without increasing the complexity.

Because all properties of pseudoflow-simplex needed to prove the complexity of solving the maximum-flow problem and its parametric versions are the same as those of the pseudoflow algorithm, it follows that the complexity of pseudoflow-simplex is the same as that of the pseudoflow algorithm for these problems.

11. A Pseudoflow Variant of the Push-Relabel Algorithm

It is possible to set up the push-relabel algorithm as a pseudoflow-based algorithm. The advantage is that with pseudoflow, the algorithm can be initialized with any pseudoflow.[3]

We sketch the push-relabel algorithm: the algorithm solves the maximum-flow problem in the graph G_{st} initializing with a preflow saturating source adjacent arcs $A(s)$, and setting all other flows to zero. The source is labeled n, the sink is labeled zero, and all other nodes are labeled one. An iteration of the algorithm consists of finding an excess node of label $\leqslant 2n$ (or $\leqslant n$ if we only search for a minimum cut) and pushing, along a residual arc, the minimum of the excess and of the residual capacity to a neighbor of lower label. If no such neighbor exists, then the node is relabeled to the minimum label of its out-neighbors in the residual graph plus one. When no excess node of label $\leqslant 2n$ exists, the algorithm terminates with a maximum flow.

The push-relabel algorithm works with a set of excess nodes, but does not permit nodes with deficits. However, in the extended network G^{ext}, any pseudoflow is a feasible flow. In particular, deficit nodes can be represented as balanced nodes with flow on the deficit arcs equal to the deficit amount. In G^{ext}, it is possible to start the push-relabel algorithm with any pseudoflow that also saturates sink-adjacent arcs. This requires adding deficit arcs from the sink to all generated deficit nodes by the given pseudoflow. Adding excess arcs that go to the source from all excess nodes not adjacent to source is not necessary because the push-relabel algorithm can work with any arbitrary set of excess nodes.

With the added deficit arcs, the push-relabel algorithm works without modification. Because the algorithm does not generate strict deficits, there is no need to add new deficit arcs during execution. At termination of the push-relabel algorithm, some of the deficit arcs may carry positive flows on them. In that case, generating a feasible flow can be done by the procedure described in §8.

Running push-relabel with pseudoflows permits the use of arbitrary initial pseudoflows, notably, the "saturate-all" initialization. This version of push-relabel allows the use of warm starts. Also, experimental studies we conducted show that for some classes of problems, the pseudoflow-push-relabel utilizing some of the initialization schemes runs faster than the standard push-relabel algorithm (Chandran and Hochbaum 2003).

12. Comparing the Simplex, Pseudoflow, and Push-Relabel Algorithms for Maximum Flow

Here, we compare the strategies of three generic algorithms for the maximum-flow problem: simplex (network simplex), push-relabel, and pseudoflow. We also describe the differences between the pseudoflow algorithm and pseudoflow-simplex.

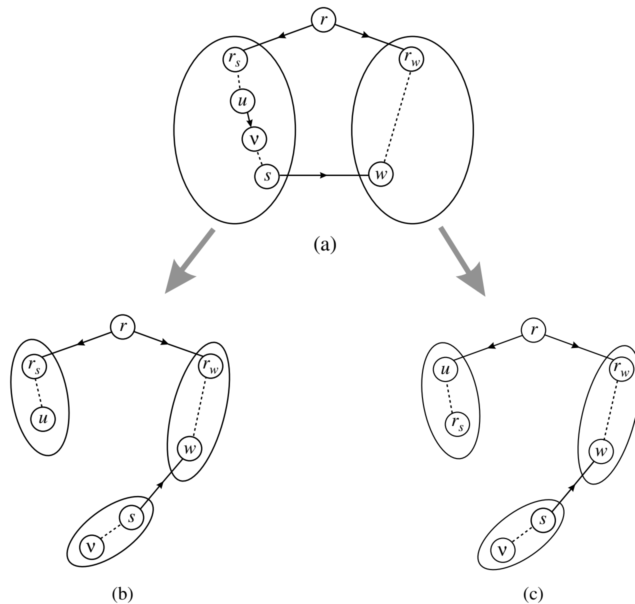
Two extreme strategies are manifested in the simplex and push-relabel algorithms. Simplex is a "global" algorithm that maintains a spanning tree in the graph and each iteration involves the entire graph. Push-relabel, on the other hand, is a "local" algorithm that can execute operations based only on information at a node and its neighbors. In that sense, push-relabel is a suitable algorithm a for distributed mode of computation, whereas simplex is not.

Our algorithm is positioned in a middle ground between simplex and push-relabel: instead of maintaining individual nodes as push-relabel does, it maintains subsets of nodes with feasible flows within the subset or branch. These subsets retain information on where the flow should be pushed to—along the paths going between the roots of the respective branches. This path information is not captured in a push-relabel algorithm that pushes flow guided only by distance labels toward lower label nodes without paths or partial paths information.

The subsets of nodes maintained by the pseudoflow algorithm tend to be smaller in size than those maintained by simplex which are the subtrees rooted at a source-adjacent or a sink-adjacent node (this latter point is made clearer by contrasting the split process; see below). Moreover, compared to pseudoflow-simplex, the pseudoflow algorithm does not strive to increase flow feasibility. Pseudoflow-simplex sends only an amount of excess that does not violate the feasibility of any residual arc along the augmentation path (the merger path in our terminology). In contrast, the pseudoflow algorithm pushes the entire excess of the branch until it gets blocked. This quantity is always larger or equal to the amount pushed by simplex.

To contrast the split process in pseudoflow as compared to the one in simplex, consider Figure 7. In Figure 7(a), the merger arc is added, and edge (u, v) is identified for

Figure 7. Comparing the split of an edge with an update step in the simplex algorithm.



pseudoflow as the first infeasible edge, or as leaving arc for simplex. Figure 7(b) shows the resulting branches following "simplex-split" and Figure 7(c) shows the branches after split. (In general, the choice of the split arc for simplex may be different from the choice in the corresponding iteration of the pseudoflow algorithm.) Note that the rooting of the branch on the left is different. For pseudoflow-simplex, the set of roots of the normalized tree with which the procedure works is always a subset of the initial set of roots, whereas for pseudoflow, the set of the strong roots can change arbitrarily.

Another aspect in which the two algorithms differ is that pseudoflow performs several arc exchanges in the tree during a single-merger iteration, whereas simplex performs exactly one.

13. Electronic Companion

An electronic companion to this paper is available as part of the online version that can be found at http://or.journal.informs.org/.

Endnotes

1. We note that the concept of pseudoflow has been used previously in algorithms solving the minimum-cost network-flow problem.
2. We thank Michel Minoux for mentioning the Boolean quadratic minimization problem's relationship to the blocking-cut problem, and the anonymous referees for pointing out the references Radzik (1993), Gale (1957), and Hoffman (1960).
3. We are grateful to an anonymous referee for pointing out this possibility.

Acknowledgments

This research was supported in part by NSF award DMI-0620677.

References

Anderson, C., D. S. Hochbaum. 2002. The performance of the pseudoflow algorithm for the maximum flow and minimum cut problems. Manuscript, University of California, Berkeley.

Boldyreff, A. W. 1955. Determination of the maximal steady state flow of traffic through a railroad network. *J. Oper. Res. Soc. Amer.* **3**(4) 443–465.

Chandran, B., D. S. Hochbaum. 2003. Experimental study of the pseudoflow push-relabel algorithm. Manuscript, University of California, Berkeley.

Cunningham, W. H. 1976. A network simplex method. *Math. Programming* **1**(1) 105–116.

Dinic, E. A. 1970. Algorithm for solution of a problem of maximal flow in a network with power estimation. *Soviet Math. Dokl.* **11** 1277–1280.

Ford, L. R., Jr., D. R. Fulkerson. 1957. A simple algorithm for finding maximal network flows and an application to the Hitchcock problem. *Canad. J. Math.* **9** 210–218.

Gale, D. 1957. A theorem of flows in networks. *Pacific J. Math.* **7**(1057) 1073–1082.

Gallo, G., M. D. Grigoriadis, R. E. Tarjan. 1989. A fast parametric maximum flow algorithm and applications. *SIAM J. Comput.* **18**(1) 30–55.

Goldberg, A. V., S. Rao. 1998. Beyond the flow decomposition barrier. *J. ACM* **45** 783–797.

Goldberg, A. V., R. E. Tarjan. 1988. A new approach to the maximum flow problem. *J. ACM* **35** 921–940.

Goldberg, A. V., M. D. Grigoriadis, R. E. Tarjan. 1991. The use of dynamic trees in a network simplex algorithm for the maximum flow problem. *Math. Programming* **50** 277–290.

Goldfarb, D., W. Chen. 1997. On strongly polynomial dual algorithms for the maximum flow problem. Special issue of *Math. Programming, Ser. B* **78**(2) 159–168.

Gusfield, D., E. Tardos. 1994. A faster parametric minimum-cut algorithm. *Algorithmica* **11**(3) 278–290.

Hochbaum, D. S. 2001. A new-old algorithm for minimum-cut and maximum-flow in closure graphs. *Networks* **37**(4) 171–193.

Hochbaum, D. S. 2003. Efficient algorithms for the inverse spanning tree problem. *Oper. Res.* **51**(5) 785–797.

Hochbaum, D. S., B. G. Chandran. 2004. Further below the flow decomposition barrier of maximum flow for bipartite matching and maximum closure. Submitted.

Hoffman, A. J. 1960. Some recent applications of the theory of linear inequalities to extremal combinatorial analysis. R. Bellman, M. Hall Jr., eds. *Proc. Sympos. Appl. Math.*, Vol. 10. *Combinatorial Analysis*. American Mathematical Society, Providence, 113–127.

Karzanov, A. V. 1974. Determining the maximal flow in a network with a method of preflows. *Soviet Math. Dokl.* **15** 434–437.

King, V., S. Rao, R. Tarjan. 1994. A faster deterministic maximum flow algorithm. *J. Algorithms* **17**(3) 447–474.

Lerchs, H., I. F. Grossmann. 1965. Optimum design of open-pit mines. *Trans. Canad. Inst. Mining, Metallurgy, Petroleum* **68** 17–24.

Malhorta, V. M., M. P. Kumar, S. N. Maheshwari. 1978. An $O(|V|^3)$ algorithm for finding maximum flows in networks. *Inform. Proc. Lett.* **7**(6) 277–278.

Martel, C. 1989. A comparison of phase and nonphase network flow algorithms. *Networks* **19**(6) 691–705.

Picard, J.-C. 1976. Maximal closure of a graph and applications to combinatorial problems. *Management Sci.* **22**(11) 1268–1272.

Radzik, T. 1993. Parametric flows, weighted means of cuts, and fractional combinatorial optimization. P. M. Pardalos, ed. *Complexity in Numerical Optimization*. World Scientific, Hackensack, NJ, 351–386.

Sleator, D. D., R. E. Tarjan. 1983. A data structure for dynamic trees. *J. Comput. System Sci.* **24** 362–391.

Sleator, D. D., R. E. Tarjan. 1985. Self-adjusting binary search trees. *J. ACM* **32** 652–686.

Vygen, J. 2002. On dual minimum cost flow algorithms. *Math. Methods Oper. Res.* **56** 101–126.