A FAST PARAMETRIC MAXIMUM FLOW ALGORITHM

Giorgio Gallo
Michael D. Grigoriadis
Robert E. Tarjan

CS-TR-105-87

November 1986
Revised July 1987

A FAST PARAMETRIC MAXIMUM FLOW ALGORITHM*

Giorgio Gallo**
Dipartimento di Informatica
Universita di Pisa, Pisa, Italy

Michael D. Grigoriadis
Department of Computer Science
Rutgers University, New Brunswick, NJ 08903

Robert E. Tarjan
Department of Computer Science
Princeton University, Princeton, NJ 08544,
and *AT&T Bell Laboratories, Murray Hill, NJ 07974*

Abstract. The classical maximum flow problem sometimes occurs in settings in which the arc capacities are not fixed but are functions of a single parameter, and the goal is to find the value of the parameter such that the corresponding maximum flow or minimum cut satisfies some side condition. Finding the desired parameter value requires solving a sequence of related maximum flow problems. We show that the recent maximum flow algorithm of Goldberg and Tarjan can be extended to solve an important class of such parametric maximum flow problems, at the cost of only a constant factor in its worst case time bound. Faster algorithms for a variety of combinatorial optimization problems follow from our result.

Keywords: Algorithms, data structures, graphs, maximum flow, network flows, networks.

1. Introduction

The well-known *maximum flow problem* calls for finding a maximum flow (or alternatively a minimum cut) in a capacitated network. This problem arises in a variety of situations in which the arc capacities are not fixed but are functions of a single parameter, and the goal is to find the value of the parameter such that the corresponding maximum flow (or

* This research was partially supported by the National Science Foundation under Grants MCS-813503 and DCR-8605962, and by the Office of Naval Research under Contracts N00014-84-K-0444 and N00014-87-K-0467.

** Some of this work was done while this author was on leave at the Department of Computer Science, Rutgers University, New Brunswick, NJ, 4/1986-7/1986.

minimum cut) meets some side condition. The usual approach to solving such problems is to use a maximum flow algorithm as a subroutine and use either binary search, monotonic search, or some other technique, such as Megiddo's [28], to find the desired value of the parameter.

Existing methods take no advantage of the similarity of the successive maximum flow problems that must be solved. In this paper, we address the question of whether this similarity can lead to computational efficiencies. We show that the answer to this question is yes: an important class of parametric maximum flow problems can be solved by extending the new maximum flow algorithm devised by Goldberg and Tarjan [13]. This algorithm is the fastest among all such algorithms for real-valued data, uniformly for all graph densities. The resulting algorithm for the parametric problem has a worst-case time bound that is only a constant factor greater than the time bound to solve a nonparametric problem of the same size. The parametric problems we consider are those in which the capacities of the arcs leaving the source are nondecreasing functions of the parameter, those of arcs entering the sink are nonincreasing functions of the parameter, and those of all other arcs are constant. Our parametric maximum flow algorithm has a variety of applications in combinatorial optimization.

This paper consists of four sections in addition to the introduction. In Section 2 we extend the Goldberg-Tarjan algorithm to find maximum flows in an n-vertex, m-arc network for $O(n)$ ordered values of the parameter in $O(nm \log(n^2/m))$ time. In Section 3 we use the algorithm of Section 2 to compute information about the minimum cut capacity as a function of the parameter, assuming that each arc capacity is a linear function of the parameter. In this case, the minimum cut capacity is a piecewise linear concave function of the parameter. We describe successively more complicated algorithms for finding the smallest (or largest) breakpoint, finding a maximum, and finding all the breakpoints of this function. Each of these algorithms runs in $O(nm \log(n^2/m))$ time.

In Section 4 we discuss applications of our parametric maximum flow algorithm to various combinatorial optimization problems. Depending upon the application, our method is faster than the best previously known method by a factor of between $\log n$ and n. The applications include flow sharing problems [3, 17, 20, 21, 26, 27], zero-one fractional programming problems [4, 5, 11, 12, 23, 24, 29, 32, 33, 34, 38], and others [9, 42]. Section 5 contains a summary of our results and some final remarks.

2. Parametric maximum flows and the preflow algorithm

We begin in this section by reviewing the maximum flow algorithm of Goldberg and Tarjan [13], here called the *preflow algorithm.* Then we extend their method to the parametric maximum flow problem and we analyze three versions of the parametric preflow algorithm. We conclude with some remarks about the parametric problem and our algorithm for solving it.

2.1. Flow terminology

A *network* (see [10,43]) is a directed graph $G = (V, E)$ with a finite vertex set V and arc set E, having a distinguished ***source vertex*** s, a distinguished ***sink vertex*** t, and a nonnegative ***capacity*** $c(v, w)$ for each arc (v, w). We denote the number of vertices by n and the number of arcs by m. We assume that for each vertex v, there is a path from s through v to t; this implies $n = O(m)$. We extend the capacity function to arbitrary vertex pairs by defining $c(v, w) = 0$ if $(v, w) \notin E$. A ***flow*** f on G is a nonnegative function on vertex pairs satisfying the following three constraints:

$$f(v,w) \le c(v,w) \text{ for } (v,w) \in V \times V \qquad \text{(capacity)} \qquad (2.1)$$

$$f(v,w) = -f(w,v) \text{ for } (v,w) \in V \times V \qquad \text{(antisymmetry)} \qquad (2.2)$$

$$\sum_{v \in V} f(u,v) = 0 \text{ for } v \in V - \{s,t\} \qquad \text{(conservation)} \qquad (2.3)$$

The ***value*** of the flow f is $\sum_{v \in V} f(v,t)$. A ***maximum flow*** is a flow of maximum value.

If A and B are two disjoint vertex subsets, the ***capacity*** of the pair A, B is $c(A, B) = \sum_{v \in A, w \in B} c(v, w)$. A ***cut*** $(X, \bar{X})$ is a two-part vertex partition ($X \cup \bar{X} = V$, $X \cap \bar{X} = \emptyset$) such that $s \in X$ and $t \in \bar{X}$. A ***minimum cut*** is a cut of minimum capacity. If f is a flow, the ***flow across the cut*** $(X, \bar{X})$ is $f(X, \bar{X}) = \sum_{v \in X, w \in \bar{X}} f(v, w)$. The conservation constraint implies that the flow across any cut is equal to the flow value. The capacity constraint implies that for any flow f and any cut $(X, \bar{X})$, we have $f(X, \bar{X}) \le c(X, \bar{X})$, which in turn implies that the value of a maximum flow is no greater than the capacity of a minimum cut. The ***max-flow min-cut theorem*** of Ford and Fulkerson [10] states that these two quantities are equal.

2.2. The preflow algorithm

The preflow algorithm computes a maximum flow in a given network. To describe the algorithm we need two additional concepts, those of a *preflow* and a *valid labeling.*

A *preflow* f on G is a real-valued function on vertex pairs satisfying the capacity constraint (2.1), the antisymmetry constraint (2.2), and the following relaxation of the conservation constraint (2.3):

$$\sum_{u \in V} f(u,v) \geq 0 \text{ for all } v \in V - \{s\} \qquad \text{(nonnegativity).} \qquad (2.4)$$

For a given preflow, we define the ***excess*** $e(v)$ of a vertex v to be $\sum_{u \in V} f(u,v)$ if $v \neq s$, or infinity if $v = s$. The ***value*** of the preflow is $e(t)$. We call a vertex $v \notin \{s,t\}$ ***active*** if $e(v) > 0$. A preflow is a flow if and only if (2.4) is satisfied with equality for all $v \notin \{s,t\}$, i.e., $e(v) = 0$ for all $v \notin \{s,t\}$. A vertex pair (v,w) is a ***residual arc*** for f if $f(v,w) < c(v,w)$; the difference $c(v,w) - f(v,w)$ is the ***residual capacity*** of the arc. A pair (v,w) that is not a residual arc is ***saturated.*** A path of residual arcs is a ***residual path.***

A *valid labeling* d for a preflow f is a function from the vertices to the nonnegative integers and infinity, such that $d(t) = 0$, $d(s) = n$, and $d(v) \leq d(w) + 1$ for every residual arc (v,w). The ***residual distance*** $d_f(v,w)$ from a vertex v to a vertex w is the minimum number of arcs on a residual path from v to w, or infinity if there is no such path. A proof by induction shows that if d is a valid labeling, $d(v) \leq \min\{d_f(v,t), d_f(v,s) + n\}$ for any vertex v.

The preflow algorithm maintains a preflow f, initially equal to the arc capacities on arcs leaving s and zero on other arcs. It improves f by pushing flow excess toward the sink along arcs estimated (by using d) to be on shortest residual paths. The value of f gradually becomes larger, and f eventually becomes a flow of maximum value. As a distance estimate, the algorithm uses a valid labeling d, initially defined by $d(s) = n, d(v) = 0$ for $v \neq s$. This labeling increases as flow excess is moved among vertices; such movement causes residual arcs to change.

To implement this approach, the algorithm uses an ***incidence list*** $I(v)$ for each vertex v. The elements of such a list, called *edges*, are the unordered pairs $\{v,w\}$ such that $(v,w) \in E$ or $(w,v) \in E$. Of the edges on $I(v)$, one, initially the first, is designated the *current edge of* v. The incidence lists $I(v)$ for all $v \in V$ can be generated from an arbirarily-ordered arc list E in $O(m)$ time.

The algorithm consists of repeating the following step until there are no active vertices:

> *Push/Relabel.* Select any active vertex v. Let $\{v,w\}$ be the current edge of v. Apply the appropriate one of the following three cases:
>
> *Push.* If $d(v) > d(w)$ and $f(v,w) < c(v,w)$, send $\delta = \min\{e(v), c(v,w) - f(v,w)\}$ units of flow from v to w. This is done by increasing $f(v,w)$ and $e(w)$ by δ, and by decreasing $f(w,v)$ and $e(v)$ by δ. (The push is ***saturating*** if $\delta = c(v,w) - f(v,w)$ and *nonsaturating* otherwise.)
>
> *Get Next Edge.* If $d(v) \leq d(w)$ or $f(v,w) = c(v,w)$, and $\{v,w\}$ is not the last edge on $I(v)$, replace $\{v,w\}$ as the current edge of v by the next edge on $I(v)$.
>
> *Relabel.* If $d(v) \leq d(w)$ or $f(v,w) = c(v,w)$, and $\{v,w\}$ is the last edge on $I(v)$, replace $d(v)$ by $\min\{d(w) \mid \{v,w\} \in I(v) \text{ and } f(v,w) < c(v,w)\} + 1$ and make the first edge on $I(v)$ the current edge of v.

When the algorithm terminates, f is a maximum flow. A minimum cut can be computed as follows. For each vertex v, replace $d(v)$ by $\min\{d_f(v,s) + n,\ d_f(v,t)\}$ for each $v \in V$. (This replacement cannot decrease any distance label.) Then, the cut $(X, \bar{X})$ defined by $X = \{v \mid d(v) \geq n\}$ is a minimum cut whose sink side $\bar{X}$ is of minimum size, a property that follows from Theorem 5.5 of Ford and Fulkerson [10].

The efficiency of this "generic" form of the preflow algorithm depends upon the order in which active vertices are selected for ***push/relabel*** steps. We shall consider this selection issue after we have extended the algorithm to the parametric problem. For the moment, we merely note the bounds derived by Goldberg and Tarjan for the generic algorithm (with any selection order).

Lemma 2.1 [13]. Any active vertex v has $d_f(v,s) < \infty$, which implies $d(v) \leq 2n - 1$. The value of $d(v)$ never decreases during the running of the algorithm. The total number of *relabel* steps is thus $O(n^2)$; together they and all the ***get next edge*** steps take $O(nm)$ time.

Lemma 2.2 [13]. The number of saturating ***push*** steps through any particular residual arc (v,w) is at most one per value of $d(v)$. The total number of saturating ***push*** steps is thus $O(nm)$; each such step takes $O(1)$ time.

Lemma 2.3 [13]. The total number of nonsaturating ***push*** steps is $O(n^2m)$; each such step takes $O(1)$ time.

In all variants of the algorithm, the running time is $O(nm)$ plus $O(1)$ time per nonsaturating ***push*** step; making the algorithm more efficient requires reducing the number of such steps. This is also true in the parametric extension, as we shall see.

2.3. Extension to parametric networks

In a *parametric network*, the arc capacities are functions of a real-valued parameter λ. We denote the capacity function by c_λ and make the following assumptions:

i. $c_\lambda(s,v)$ is a nondecreasing function of λ for all $v \neq t$.

ii. $c_\lambda(v,t)$ is a nonincreasing function of λ for all $v \neq s$.

iii. $c_\lambda(v,w)$ is constant for all $v \neq s$, $w \neq t$.

When speaking of a maximum flow or minimum cut in a parametric network, we mean maximum or minimum for some particular value of the parameter λ.

We shall address the problem of computing maximum flows (or minimum cuts) for each member of an increasing sequence of parameter values $\lambda_1 < \lambda_2 < \cdots < \lambda_l$. Successive values are given *on-line*; that is, λ_{i+1} need not be known until after the maximum flow for λ_i has been computed. In stating time bounds we shall assume that the capacity of an arc can be computed in constant time given the value of λ. (Such is the case if, for example, the arc capacities are linear functions of λ.) We shall also assume that $l = O(n)$; the algorithm we shall describe computes no more than $n-1$ distinct minimum cuts, no matter how many values of λ are given.

We shall now extend the preflow algorithm to the parametric maximum flow problem. Suppose that for some value λ_i of the parameter we have computed a maximum flow f and a valid labeling d for f. What is the effect of changing the value of the parameter to λ_{i+1}? The capacity of each arc (s,v) may increase; that of each arc (v,t) may decrease. Suppose we modify f by replacing $f(v,t)$ by $\min\{c_{\lambda_{i+1}}(v,t), f(v,t)\}$ for each arc $(v,t) \in E$, and replacing $f(s,v)$ by $\max\{c_{\lambda_{i+1}}(s,v), f(s,v)\}$ for each arc $(s,v) \in E$ such that $d(v) < n$. The modified f is a preflow, since $e(v)$ for $v \notin \{s,t\}$ can only have increased. Furthermore, d is a valid labeling for the modified f, since the only new residual arcs are of the form

(s,v) for $d(v) \geq n$ and (v,s) for $d(v) < n$. This means that we can compute a maximum flow and a minimum cut for λ_{i+1} by applying the preflow algorithm beginning with the modified f and the current d.

This idea leads to the following ***parametric preflow algorithm.*** The algorithm finds a maximum flow f_i and a minimum cut $(X_i, \bar{X}_i)$ for each value λ_i of the parameter. It consists of initializing $f = 0$, $d(s) = n$, $d(v) = 0$ for $v \neq s$, and $i = 0$, and repeating the following three steps l times:

Step 1. (Update preflow.) Replace i by $i+1$. For $(v,t) \in E$, replace $f(v,t)$ by $\min\{c_{\lambda_i}(v,t),\ f(v,t)\}$. For $(s,v) \in E$ with $d(v) < n$, replace $f(s,v)$ by $\max\{c_{\lambda_i}(s,v),\ f(s,v)\}$.

Step 2. (Find maximum flow.) Apply the preflow algorithm to the network with arc capacities corresponding to λ_i, beginning with the current f and d. Let f and d be the resulting flow and final valid labeling.

Step 3. (Find minimum cut.) Redefine $d(v) = \min\{d_f(v,s) + n,\ d_f(v,t)\}$ for each $v \in V$. The cut $(X_i, \bar{X}_i)$ is then given by $X_i = \{v \mid d(v) \geq n\}$.

The minimum cuts produced by the algorithm have a ***nesting property*** that was previously observed in the context of various applications by Eisner and Severance [9], Stone [42], and perhaps others. Megiddo [26] has also noted a similar property in a related problem. Here, the result follows directly from our algorithm.

Lemma 2.4. For a given on-line sequence of parameter values $\lambda_1 < \lambda_2 < \cdots < \lambda_l$, the parametric preflow algorithm correctly computes maximum flows $f_1, f_2, \cdots, f_l$ and minimum cuts $(X_1, \bar{X}_1), (X_2, \bar{X}_2), \cdots, (X_l, \bar{X}_l)$ such that $X_1 \subseteq X_2 \subseteq \cdots \subseteq X_l$.

Proof. The correctness of the algorithm is immediate. For any vertex v, the label $d(v)$ cannot decrease in Step 3, which implies that $d(v)$ never decreases during the running of the algorithm. This means that $X_1 \subseteq X_2 \subseteq \cdots \subseteq X_l$, which in turn implies that there can be at most $n-1$ distinct sets X_i. □

2.4. Analysis of the parametric preflow algorithm

In view of Lemma 2.4, Lemmas 2.1 and 2.2 hold without change for the parametric preflow algorithm. Furthermore, the time spent in Steps 1 and 3 is $O(m)$ per iteration for a total of $O(lm) = O(nm)$ time (assuming $l = O(n)$). Thus the parametric preflow algorithm runs in $O(nm)$ time plus $O(1)$ time per nonsaturating push.

The number of nonsaturating pushes depends upon the order in which *push/relabel* steps are performed. We shall analyze three versions of the parametric preflow algorithm. For each, we show that the time bound for the nonparametric case extends to the parametric case with an increase of at most a constant factor. The proofs of the following three theorems are analogous to those given in [13].

We first analyze the generic version, in which *push/relabel* steps take place in any order.

Theorem 2.5. For any order of *push/relabel* step selection, the number of nonsaturating *push* steps, and hence the running time of the parametric preflow algorithm, is $O(n^2m)$.

Proof. Let $\Phi = \sum\{d(v) \,|\, v \text{ is active}\}$ if some vertex is active, and $\Phi = 0$ otherwise. A nonsaturating *push* step decreases Φ by at least one. The function Φ is always in the range 0 to $2n^2$. Step 1 can increase Φ by at most $2n^2$, for a total over all iterations of Step 1 of $O(ln^2) = O(n^3)$. A relabeling step increases Φ by the amount by which the label changes. Thus the total increase in Φ due to relabeling steps is $O(n^2)$. A saturating *push* step can increase Φ by at most $2n$. Thus the total increase in Φ due to such steps is $O(n^2m)$. These are the only ways in which Φ can increase. The total number of nonsaturating *push* steps is bounded by the total increase in Φ over the algorithm, which is $O(n^2m)$. □

Next we consider the *first-in first-out* (FIFO) version of the preflow algorithm, which solves the nonparametric problem in $O(n^3)$ time [13]. In this version, a queue Q is used to select vertices for *push/relabel* steps. Initially Q is empty. At the beginning of Step 2 of the parametric preflow algorithm, every active vertex is appended to Q. The FIFO algorithm consists of repeating the following step until Q is empty:

Discharge. Remove the vertex v on the front of Q. Apply *push/relabel* steps to v until v is no longer active or v is relabeled. If a push from v to another vertex w makes w active, add w to the rear of Q.

Theorem 2.6. For the FIFO version of the parametric preflow algorithm, the number of nonsaturating ***push*** steps, and hence the total running time, is $O(n^3)$.

Proof. We define ***passes*** over the queue Q as follows. The first pass during an iteration of Step 2 consists of the ***discharge*** steps applied to the vertices initially on Q. Each pass after the first in an iteration consists of the ***discharge*** steps applied to the vertices added to Q during the previous pass. There is at most one nonsaturating ***push*** step per vertex v per pass, since such a step reduces $e(v)$ to zero. We claim that the total number of passes is $O(n^2 + ln)$, from which the theorem follows.

To establish the claim, we define $\Phi = \max\{d(v) \mid v \in Q\}$ if $Q \neq \emptyset$, and $\Phi = 0$ if $Q = \emptyset$. Consider the effect on Φ of a pass over Q. If $v \in Q$ at the beginning of a pass and $d(v) = \Phi$, then $v \notin Q$ at the end of the pass unless a ***relabel*** step occurs during the pass. Thus, if Φ is the same at the end as at the beginning of the pass, some vertex label must have increased by at least one. If Φ increases over the pass, some $d(v)$ must increase by at least the amount of the increase in Φ. From the end of one iteration to the beginning of the next, Φ can increase by $O(n)$. Thus (i) the total number of passes in which Φ can increase or stay the same is $O(n^2 + ln)$; (ii) the total number of passes in which Φ can decrease is at most the total increase in Φ between passes and during passes in which it increases, which is also $O(n^2 + ln)$. We conclude that the total number of passes is at most $O(n^2 + ln)$, verifying the claim and hence the theorem. □

A more elaborate version of the preflow algorithm [13] uses the dynamic tree data structure of Sleator and Tarjan [39,40] to reduce the running time to $O(nm \log(n^2/m))$. The corresponding version of the parametric flow algorithm also runs in $O(nm \log(n^2/m))$ time. It uses a queue Q for vertex selection, and it performs ***discharge*** steps exactly as does the FIFO algorithm, but in place of ***push/relabel*** steps it uses more complicated ***tree-push/relabel*** steps. A ***tree-push/relabel step*** can move an amount of flow excess through several arcs at once. Extending the analysis in [13] to the parametric case is straightforward. We shall merely summarize the results.

The dynamic tree algorithm uses a parameter k, the ***maximum tree size***, which can be chosen freely in the range from 2 to n. An easy extension of the analysis in [13] shows that the parametric preflow algorithm runs in $O(nm \log k)$ time plus $O(\log k)$ time per addition of a vertex to Q. Furthermore, the number of additions of a vertex to Q is $O(nm)$

plus $O(n/k)$ per pass over Q, where passes are defined as in the proof of Theorem 2.6. The $O(n^2)$ bound on the number of passes in the proof of Theorem 2.6 remains valid if the dynamic tree algorithm is used in place of the FIFO algorithm. Combining these estimates, we obtain an $O((nm + n^3/k)\log k)$ bound on the total running time. Choosing $k = \max\{2, n^2/m\}$ gives the following theorem:

Theorem 2.7. The parametric preflow algorithm implemented using dynamic trees runs in $O(nm\log(n^2/m))$ time.

2.5. Additional observations

We conclude this section with several observations about the parametric maximum flow problem and our algorithm for solving it. Our first observation concerns variants of the parametric maximum flow problem. Our algorithm remains valid if the arc capacity functions are nonincreasing on arcs out of s and nondecreasing on arcs into t, and the values of the parameter λ are given in decreasing order. To see this, merely substitute $-\lambda$ for λ. The algorithm also applies if the arc capacity functions are nondecreasing on arcs out of s and nonincreasing on arcs into t, and the values of λ are given in decreasing order. In this case, reverse the directions of all the arcs, exchange the source and sink, and apply the original algorithm to this reversed network, which we shall denote by G^R. Each minimum cut $(\bar{X}, X)$ generated for G^R will correspond to a minimum cut $(X, \bar{X})$ in the original (nonreversed) network that will have the source side of minimum size instead of the sink side. Successively generated minimum cuts in G^R will correspond to cuts with successively smaller source sides in the original network.

If we are only interested in computing minimum cuts, there is a variant of the preflow algorithm that does less computation, although it has the same asymptotic time bound [**13**]. This variant, here called the *min-cut preflow algorithm*, computes a preflow of maximum value and a minimum cut, but not a maximum flow. It differs from the original algorithm in that a vertex v is considered to be active only if $e(v) > 0$ and $d(v) < n$. The algorithm terminates having computed a maximum preflow. (A preflow f is maximum if and only if for every vertex v, $v \neq t$ or $e(v) = 0$ or $d_f(v,t) < \infty$.) If this variant is used to compute minimum cuts, a maximum flow for a desired parameter value can be computed by beginning with the corresponding maximum preflow and converting it into a maximum flow using the original preflow algorithm. Most of the applications we shall consider only require the computation of minimum cuts or even minimum cut values and not maximum flows.

So far we have required all arc capacities to be nonnegative, but if we are only interested in computing minimum cuts, we can allow negative capacities on arcs out of s and on arcs into t. This is because there is a simple transformation that makes such arc capacities nonnegative without affecting minimum cuts [34]. For a given vertex v, suppose we add a constant $\Delta(v)$ to $c(s,v)$ and $c(v,t)$. Then the minimum cuts do not change since the capacity of every cut is increased by $\Delta(v)$.

By adding a suitably large $\Delta(v)$ to $c(s,v)$ and $c(v,t)$ for each v, we can make all the arc capacities positive. In the parametric problem, we can choose a new function Δ on the vertices of G for each new value λ_i of λ without affecting the $O(nm\log(n^2/m))$ time bound of our algorithm. It suffices to choose

$$\Delta_{\lambda_1}(v) = \max\{0, c_{\lambda_1}(s,v)\} + \max\{0, -c_{\lambda_1}(v,t)\},$$
$$\Delta_{\lambda_{i+1}}(v) = \Delta_{\lambda_i}(v) + c_{\lambda_i}(v,t) - c_{\lambda_{i+1}}(v,t) \quad \text{for } i \geq 1.$$

With this choice, the transformed arc capacities are nondecreasing functions of λ on arcs leaving s and constant on arcs that enter t. Although the same effect could be obtained by adding a sufficiently large constant to the capacities of these arcs, the modification we have described has the additional advantage of keeping capacities as small as possible. In subsequent sections, when discussing minimum cut problems, we shall allow arbitrary capacities, positive or negative, on arcs leaving s and arcs entering t.

The minimum cuts corresponding to various parameter values have a *nesting property* that is a strengthening of Lemma 2.4. The following lemma is an extension of known results [10, p.13] that we shall need in the next section. To prove the lemma, we shall use the min-cut preflow algorithm discussed above.

Lemma 2.8. Let $(X,\bar{X})$ be any minimum cut for $\lambda = \lambda_1$, and let $(Y,\bar{Y})$ be any minimum cut for $\lambda = \lambda_2$ such that $\lambda_1 \leq \lambda_2$. Then, $(X \cap Y, \bar{X} \cup \bar{Y})$ is a minimum cut for $\lambda = \lambda_1$ and $(X \cup Y, \bar{X} \cap \bar{Y})$ is a minimum cut for $\lambda = \lambda_2$.

Proof. Run the min-cut parametric algorithm for $\lambda = \lambda_1$, followed by $\lambda = \lambda_2$. At the beginning of the computation for $\lambda = \lambda_2$, all vertices $v \in X - \{s\}$ have $d(v) \geq n$. Thus, after a maximum preflow is computed for $\lambda = \lambda_2$, all arcs (v,w) with $v \in X, w \in \bar{X}$ are saturated (their flow has not changed during the computation for $\lambda = \lambda_2$.) Since $(Y,\bar{Y})$ is a minimum cut for $\lambda = \lambda_2$, all arcs (v,w) with $v \in Y, w \in \bar{Y}$ are saturated. Furthermore, if $v \in \bar{Y} - \{t\}$ then $e(v) = 0$, since the net flow across $(Y, \bar{Y})$ must be equal to the excess at t.

Now consider the cut $Z = (X \cup Y)$, $\bar{Z} = (\bar{X} \cap \bar{Y})$. Any arc (v, w) with $v \in Z$, $w \in \bar{Z}$ must be staurated. Since $v \in \bar{Z} - \{t\}$ implies $e(v) = 0$, the cut $(Z, \bar{Z})$ must have capacity $e(t)$, and hence it must be a minimum cut.

The proof for $(X \cap Y, \bar{X} \cup \bar{Y})$ is similar: proceed on G^R, and run the min-cut parametric algorithm for $\lambda = \lambda_2$, followed by $\lambda = \lambda_1$. □

A direct consequence of this lemma is the following corollary, which we shall need in the next section.

Corollary 2.9. Let $(X_1, \bar{X}_1)$ be a minimum cut for $\lambda = \lambda_1$, let $(X_2, \bar{X}_2)$ be a minimum cut for $\lambda = \lambda_2$ such that $X_1 \subseteq X_2$ and $\lambda_1 \leq \lambda_2$, and let λ_3 be such that $\lambda_1 \leq \lambda_3 \leq \lambda_2$. Then there is a cut $(X_3, \bar{X}_3)$ minimum for λ_3 such that $X_1 \subseteq X_3 \subseteq X_2$.

Proof. Let $(X_3', \bar{X}_3')$ be any minimum cut for $\lambda = \lambda_3$. Take $X_3 = (X_3' \cup X_1) \cap X_2$, $\bar{X}_3 = V - X_3$, and apply Lemma 2.8 twice. □

Our last observation is that if the graph G is bipartite, the $O(nm \log(n^2/m))$ time bound for computing parametric maximum flows can be improved slightly. Suppose $V = A \cup B$, $A \cap B \neq \emptyset$, and every arc in G has one vertex in A and one in B. Let $n_A = |A|$ and $n_B = |B|$ and suppose that $n_A \leq n_B$. Then the time to compute maximum flows for l ordered values of λ can be reduced to $O(n_A m \log(n_A^2/m + 2))$ if $l = O(n_A)$. This requires modifying the preflow algorithm so that only vertices in A are active, and modifying the use of the dynamic tree data structure so that such a tree contains as many vertices in A as in B. The details can be found in [41]. The bound is slightly worse if $l = \omega(n_A)$ (see [41]).

3. The min-cut capacity function of a parametric network

For a parametric network, we define the ***min-cut capacity function*** $\kappa(\lambda)$ to be the capacity of a minimum cut as a function of the parameter λ. We shall assume throughout this section that the arc capacities are *linear* functions of λ satisfying the conditions (i)-(iii) of Section 2. It is well known [9,42] and follows from the results of Section 2 that under this assumption $\kappa(\lambda)$ is a piecewise-linear concave function with at most $n-2$ breakpoints. (By a *breakpoint* we mean a value of λ at which the slope of $\kappa(\lambda)$ changes.) The $n-1$ or fewer line segments forming the graph of $\kappa(\lambda)$ correspond to $n-1$ or fewer distinct cuts. We shall develop three algorithms for computing information about $\kappa(\lambda)$. The first computes the smallest (or equivalently the largest) breakpoint. The second computes a value of λ at which $\kappa(\lambda)$ is maximum. The third computes all the breakpoints. Each of these algorithms uses the algorithm of Section 2 as a subroutine and runs in $O(nm \log(n^2/m))$ time. Although the algorithm for computing all breakpoints solves all three problems, we shall present all three algorithms since each is more complicated than the preceding one and since the resulting difference in constant factors may be important in practice.

We shall assume that the capacities $c_\lambda(s,v)$ and $c_\lambda(v,t)$ are given in the form $c_\lambda(s,v) = a_0(v) + \lambda a_1(v)$ and $c_\lambda(v,t) = b_0(v) - \lambda b_1(v)$, with arbitrary coefficients a_0, b_0 and nonnegative coefficients a_1, b_1. A minimum cut $(X_0, \bar{X}_0)$ for some $\lambda = \lambda_0$ gives an equation for a line that contributes a line segment to the function $\kappa(\lambda)$ at $\lambda = \lambda_0$. This line is $L_{X_0}(\lambda) = \alpha_0 + \lambda\beta_0$, where $\alpha_0 = c_{\lambda_0}(X_0, \bar{X}_0) - \lambda_0\beta_0$ and $\beta_0 = \sum_{v \in \bar{X}_0} a_1(v) - \sum_{v \in X_0} b_1(v)$.

3.1. Computing the smallest breakpoint of $\kappa(\lambda)$

To compute the smallest breakpoint of $\kappa(\lambda)$ we use an algorithm stated by Gusfield [17] for an application involving scheduling transmissions in a communication network, discussed in more detail in Section 4.1.

The algorithm consists of the following two steps.

Step 0. Compute λ_1, λ_2 such that the smallest breakpoint λ_0 satisfies $\lambda_1 \le \lambda_0 \le \lambda_2$. Compute a cut $(X_1, \bar{X}_1)$ that is a minimum for λ_1. Go to Step 1.

Step 1. Compute a cut $(X_2, \bar{X}_2)$ that is a minimum for λ_2. If $L_{X_1}(\lambda_1) = L_{X_2}(\lambda_2)$, stop: λ_2 is the smallest breakpoint. Otherwise, replace λ_2 by the value of λ such that $L_{X_1}(\lambda) = L_{X_2}(\lambda)$ and repeat Step 1. (The appropriate value of λ is $(\alpha_2 - \alpha_1)/(\beta_1 - \beta_2)$.)

The values of λ_2 generated by this algorithm are strictly decreasing; thus the parametric preflow algorithm of Section 2 applied to G^R performs all ierations of Step 1 in $O(nm \log(n^2/m))$ time, saving a factor of n over Gusfield's algorithm [17].

In Step 0, it suffices to select λ_1 sufficiently small that for each vertex v such that (s,v) or (v,t) is of nonconstant capacity, $c_{\lambda_1}(s,v) + \sum_{u \in V-\{s,t\}} c(u,v) < c_{\lambda_1}(v,t)$. A suitable value of λ_1 is

$$\lambda_1 = \min_{v \in V-\{s,t\}} \{\, b_0(v) - a_0(v) - \frac{\sum_{u \in V-\{s,t\}} c(u,v)}{a_1(v) + b_1(v)} \mid a_1(v) + b_1(v) > 0 \,\} - 1. \quad (3.1)$$

Similarly, it suffices to select λ_2 sufficiently large that for each vertex v such that (s,v) or (v,t) is of nonconstant capacity, $c_{\lambda_2}(v,t) + \sum_{w \in V-\{s,t\}} c(v,w) < c_{\lambda_2}(s,v)$. A suitable value of λ_2 is

$$\lambda_2 = \max_{v \in V-\{s,t\}} \{\, b_0(v) - a_0(v) + \frac{\sum_{w \in V-\{s,t\}} c(v,w)}{a_1(v) + b_1(v)} \mid a_1(v) + b_1(v) > 0 \,\} + 1. \quad (3.2)$$

Essentially the same algorithm can be used for computing the largest breakpoint; instead of successively decreasing λ_2 and using G^R, successively increase λ_1 and use G.

3.2. Finding a maximum of $\kappa(\lambda)$

Our algorithm for finding the value of λ that maximizes $\kappa(\lambda)$ is based on a simple method of iterative interval contraction for computing the maximum $f(\lambda^*)$ of a strictly concave and continuously differentiable function $f(\lambda)$ on a nonempty interval $[\lambda_1, \lambda_2]$ of the real line. The method is as follows. First, compute the function values and the tangents of $f(\lambda)$ at each end of the given interval. Second, compute the point $\lambda_3 \in [\lambda_1, \lambda_2]$ where the two tangent lines intersect, and also compute $f'(\lambda_3)$. Then, if $f'(\lambda_3) < 0$ replace λ_2 by λ_3 and repeat; if $f'(\lambda_3) > 0$ replace λ_1 by λ_3 and repeat; if $f'(\lambda_3) = 0$ accept λ_3 as the solution. Of course this algorithm need not terminate, but it will converge to the maximum.

The method is seldom used in this general setting because it is inferior to several other algorithms for one-dimensional maximization. But it can be specialized in the obvious way to handle the piecewise-linear concave function $\kappa(\lambda)$ efficiently. A maximum of $\kappa(\lambda)$ can be computed in as many function evaluations as there are linear segments that comprise $\kappa(\lambda)$, namely $n-1$ or fewer. Using the notation introduced above, $\lambda_3 = (\alpha_2 - \alpha_1)/(\beta_1 - \beta_2)$ if the line segments of $\kappa(\lambda)$ at λ_1 and at λ_2 are distinct. Otherwise, the search terminates with a line segment of zero slope and $\lambda^* = \lambda_1$ (or $\lambda^* = \lambda_2$). The algorithm will compute a maximum of $\kappa(\lambda)$ in $O(n^2 m \log(n^2/m))$ time since at most $n-1$ minimum cut problems must be solved.

The running time of this algorithm can be improved by partitioning the sequence of successive values of λ_3 into two subsequences, one increasing and the other decreasing. It is then possible to use two concurrent invocations of the parametric preflow algorithm: Invocation I that starts with λ_1 and computes minimum cuts of G for an increasing sequence of λ values, and Invocation D that starts with λ_2 and computes minimum cuts of G^R for a decreasing sequence of λ values. A new value λ_3 is a member of the increasing sequence if $\beta_3 > 0$, and a member of the decreasing sequence otherwise. The initial values of λ_1 and λ_2 must be such that all breakpoints lie in the interval $[\lambda_1, \lambda_2]$, a property that is assured by the values of λ defined by (3.1) and (3.2).

The algorithm to compute a maximum of $\kappa(\lambda)$ consists of the following four steps.

Step 0. Compute the initial values λ_1 and λ_2 from (3.1) and (3.2). Start concurrent invocations (I and D) of the parametric preflow algorithm of Section 2: For $\lambda = \lambda_1$, Invocation I computes a minimum cut $(X_1, \bar{X}_1)$ having $|X_1|$ maximum; for $\lambda = \lambda_2$, Invocation D computes a minimum cut $(X_2, \bar{X}_2)$ having $|X_2|$ minimum.

Step 1. Compute $\lambda_3 = (\alpha_2 - \alpha_1)/(\beta_1 - \beta_2)$, pass λ_3 to both invocations I and D, and run them concurrently. If invocation I finds a minimum cut $(X_3^I, \bar{X}_3^I)$ first, suspend invocation D and go to Step 2 (the other case is symmetric.) Compute $\beta_3 = \sum_{v \in \bar{X}_3} a_1(v) - \sum_{v \in X_3} b_1(v)$.

Step 2. If $\beta_3 = 0$, stop: $\lambda^* = \lambda_3$. Otherwise, if $\beta_3 > 0$, replace λ_1 by λ_3, back up invocation D to its state before it began processing λ_3, and go to Step 1. Otherwise, go to Step 3.

Step 3 ($\beta_3 < 0$). Finish running the invocation D on λ_3. This produces a minimum cut $(X_3^D, \bar{X}_3^D)$, not necessarily the same as $(X_3^I, \bar{X}_3^I)$. If $\beta_3 \geq 0$, stop: $\lambda^* = \lambda_3$. Otherwise, replace λ_2 by λ_3, back up invocation I to its state before processing λ_3, and go to Step 1.

Backing up invocation D or I as required in Steps 2 and 3 is merely a matter of restoring the appropriate flow and valid labeling, which takes $O(m)$ time. The total time spent during one iteration of Steps 1, 2 and 3 is proportional to the time spent in invocation I or D, whichever one is run to completion on λ_3 and not backed up. The total number of values of λ_3 processed is $O(n)$. Thus the total time is proportional to the time necessary to run the parametric preflow algorithm of Section 2.3 twice, once on an increasing sequence of values and once on a decreasing sequence of values; i.e., $O(nm \log(n^2/m))$.

3.3. Finding all breakpoints of $\kappa(\lambda)$

In some applications it is necessary to produce all the line segments or breakpoints of $\kappa(\lambda)$, possibly along with the corresponding minimum cuts. To do this we extend the maximum-finding algorithm of the previous section. This algorithm uses iterative contraction of the interval $[\lambda_1, \lambda_2]$; it ignores breakpoints that lie in the discarded portion of the interval. We can find all the breakpoints by proceeding as in the algorithm of Section 3.2 but using a divide-and-conquer strategy that recursively examines both of the subintervals $[\lambda_1, \lambda_3]$ and $[\lambda_3, \lambda_2]$ into which the current interval is split by the new value λ_3. This method was proposed by Eisner and Severance [9] for bipartite graphs in the context of a database record segmentation problem (see Section 4.4). Unfortunately, a straighforward implementation of this idea yields an $O(n^2 m \log(n^2/m))$-time algorithm. To obtain a better bound it is necessary to use two concurrent invocations of the parametric preflow algorithm, and also use graph contraction so that recursive invocations of the method compute cuts on smaller and smaller graphs.

If G is a network and X is a set of vertices such that exactly one of s and t is in X, we define $G(X)$, the *contraction of G by X*, to be the network formed by shrinking the vertices in X to a single vertex, eliminating loops, and combining multiple arcs by adding their capacities. The algorithm we present reports only the breakpoints of $\kappa(\lambda)$, although it computes cuts corresponding to the line segments of the graph of $\kappa(\lambda)$. If the actual cuts are needed, they can either be saved as the computation proceeds or computed in a postprocessing step using one application of the method in Section 2.3.

Our algorithm uses a recursive procedure called *slice*. With each network G to which *slice* is applied, we associate four pieces of information: Two values of λ, denoted by λ_1 and λ_2, and two flows f_1 and f_2, such that f_1 is a maximum flow for λ_1, f_2 is a maximum flow for λ_2, the cut $(\{s\}, V - \{s\})$ is the unique minimum cut for λ_1, the cut $(V - \{t\}, \{t\})$ is the unique minimum cut for λ_2, and $\lambda_1 < \lambda_2$. The initial values for λ_1 and λ_2 are computed from (3.1) and (3.2) as before. The *breakpoint algorithm* consists of the following three steps.

Step 1. Compute λ_1 according to (3.1). Compute a maximum flow f_1 and minimum cut $(X_1, \bar{X}_1)$ for λ_1 such that $|X_1|$ is maximum by applying the preflow algorithm to G. Let $G' = G(X_1)$.

Step 2. Compute λ_2 according to (3.2). Compute a maximum flow f_2 and minimum cut $(X_2, \bar{X}_2)$ for λ_2 such that $|X_2|$ is minimum by applying the preflow algorithm to $(G')^R$. Let $G'' = G'(X_2')$.

Step 3. If G'' contains at least three vertices, let f_1'' and f_2'' be the flows in G'' corresponding to f_1 and f_2 respectively; perform *slice* $(G'', \lambda_1, \lambda_2, f_1'', f_2'')$, where *slice* is defined as follows:

Procedure *slice*$(G, \lambda_1, \lambda_2, f_1, f_2)$.

Step S1. Let λ_3 be the value of λ such that $c_{\lambda_3}(\{s\}, V - \{s\}) = c_{\lambda_3}(V - \{t\}, \{t\})$. (This value will satisfy $\lambda_1 \leq \lambda_3 \leq \lambda_2$.)

Step S2. Run the preflow algorithm for the value λ_3 on G starting with the preflow f_1' formed by increasing f_1 on arcs (s, v) to saturate them and decreasing f_1 on arcs (v, t) to meet the capacity constraints. As an initial valid labeling, use $d(v) = \min\{d_{f_1'}(v, t), d_{f_1'}(v, s) + n\}$. Concurrently, run the preflow algorithm for the value λ_3 on G^R starting with the preflow f_2' formed by increasing f_2 on arcs (v, t) to saturate them and decreasing f_2 on arcs (s, v) to meet the capacity constraints. As an initial valid labeling, use $d(v) = \min\{d_{f_2'}(s, v), d_{f_2'}(t, v) + n\}$. Stop when one of the concurrent applications stops, having computed a maximum flow f_3. Suppose the preflow algorithm applied to G stops first. (The other case is symmetric.) Find the minimum cuts $(X_3, \bar{X}_3)$ and $(X_3', \bar{X}_3')$ for λ_3 such that $|X_3|$ is minimum and $|X_3'|$ is maximum. If $|X_3| > n/2$, complete the execution of the preflow algorithm on G^R and let f_3 be the resulting maximum flow.

Step S3. If $c_\lambda(X_3, \bar{X}_3) \neq c_\lambda(X_3', \bar{X}_3')$ for some λ, report λ_3 as a breakpoint.

Step S4. If $X_3 \neq \{s\}$, perform *slice* $(G(\bar{X}_3), \lambda_1, \lambda_3, f_1, f_3)$. If $\bar{X}_3' \neq \{t\}$, perform *slice* $(G(X_3'), \lambda_3, \lambda_2, f_3, f_2)$.

The correctness of this algorithm follows from Corollary 2.9. Note that the minimum cuts computed in Step S2 correspond to minimum cuts for λ_3 in the original network, with the correspondence obtained by expanding the contracted vertex sets. Since each vertex of G is in at most one of the two subproblems in Step S4, there are $O(n)$ invocations of *slice*.

3.4. Analysis of the breakpoint algorithm

Two ideas underly the efficiency of the breakpoint algorithm. To explain them, we need to develop a framework for the analysis of the algorithm. We shall charge to an invocation of *slice* the time spent in the invocation, not including the time spent in nested invocations. The time charged to one invocation is then $O(m)$ plus the time spent running the preflow algorithm in Step S2. Summing $O(m)$ over all $O(n)$ invocations of *slice* gives a bound of $O(nm)$. It remains to estimate the time spent running the preflow algorithm.

The first idea contributing to the speed of the algorithm is that the results of Section 2 allow us to bound the time of a sequence of preflow algorithm applications, not just a single one, by $O(nm\log(n^2/m))$. That is, if we charge this much time for an invocation of *slice*, we can regard certain of the nested invocations as being free. The second idea is that running the preflow algorithm concurrently on G and on G^R allows us to regard the larger of the nested invocations in Step S4 as being free. This leads to a recurrence bounding the running time whose solution is $O(nm\log(n^2/m))$.

Consider an invocation of *slice* $G(\lambda_1, \lambda_2, f_1, f_2)$. Let $G_1 = G(\bar{X}_3)$ as computed in Step S4, and let $G_2 = G(X_3')$; let n_1, m_1 and n_2, m_2 be the numbers of vertices and arcs in G_1 and G_2, respectively. We regard this invocation of *slice* as being a continuation of the algorithm of Section 2.3 applied to G, with λ_1 the most recently processed value of λ and f_1 the resulting maximum flow. Simultaneously, we regard the invocation as being a continuation of the algorithm of Section 2 applied to G^R, with λ_2 the most recently processed value of λ and f_2 the resulting flow.

With this interpretation we can regard the preflow algorithm applications in Step S2 as being free, but if $|X_3| \leq n/2$ we must account for new applications of the Section 2 algorithm on $G(\bar{X}_3)$ and $G^R(\bar{X}_3)$, and otherwise (i.e., $|\bar{X}_3'| \leq n/2$) we must account for new applications of the Section 2 algorithm on $G(X_3')$ and $G^R(X_3')$. Thus we obtain the following bound on the time spent in invocations of the preflow algorithm. If G has n vertices and m arcs, the time spent in such invocations during the computation of $\kappa(\lambda)$ is at most $T(n,m)+$ $O(nm\log(n^2/m))$, where $T(n,m)$ is defined recursively as follows:

$$T(n,m) = \begin{cases} 0 \quad \text{if n} \leq 3; \\ \max\{T(n_1,m_1) + T(n_2,m_2) + O(n_1 m_1 \log({n^2}_1/m_1)) : \\ \qquad n_1, n_2 \geq 3; n_1 + n_2 \leq n+2; \\ \qquad n_1 \leq n_2; m_1, m_2 \geq 1; m_1 + m_2 \leq m+1\} \quad \text{if n} > 3. \end{cases}$$

Remark. In this analysis, the sequence of preflow algorithm invocations associated with a particular application of the algorithm of Section 2.3 is on a sequence of successively smaller graphs, but the analysis in Section 2.4 remains valid. □

To solve the recurrence for $T(n,m)$, we begin by simplifying it. Observe that $T(n,m) = O(T_1(n,m)\log(n^2/m))$, where $T_1(n,m)$ is defined as follows:

$$T_1(n,m) = \begin{cases} 0 \quad \text{if n} \leq 3; \\ \max\{T_1(n_1,m_1) + T_1(n_2,m_2) + n_1 m_1 \ : \\ \quad n_1, n_2 \ \geq \ 3;\ n_1 + n_2 \leq n+2; \\ \quad n_1 \leq n_2;\ m_1,\ m_2 \geq 1;\ m_1 + m_2 \leq m+1\} \quad \text{if n} > 3. \end{cases}$$

By making the change of variables $n' = n-2, m' = m-1$, we obtain $T_1(n,m) = T_2(n-2, m-1)$, where T_2 is defined as follows:

$$T_2(n,m) = \begin{cases} 0 \quad \text{if n} \leq 1; \\ \max\{T_2(n_1,m_1) + T_2(n_2,m_2) + n_1 m_1 + 2m_1 + n_1 + 2 \ : \\ \quad n_1, n_2 \leq 1; n_1 + n_2 \leq n; \\ \quad n_1 \leq n_2;\ m_1, m_2 \ \geq \ 0;\ m_1 + m_2 \leq m\} \quad \text{if n} > 1. \end{cases}$$

Since the recurrence for $T_2(n,m)$ can be unwound at most $n-1$ times before all branches are terminal, the additive term "$2m_1 + n_1 + 2$" contributes at most $(n-1)(2m+n+2)$ to $T_2(n,m)$. That is, $T_2(n,m) = T_3(n,m) + O(nm)$, where T_3 is defined as follows:

$$T_3(n,m) = \begin{cases} 0 \quad \text{if n} \leq 1; \\ \max\ \{T_3(n_1,m_1) + T_3(n_2,m_2) + n_1 m_1 \ : \\ \quad n_1, n_2 \leq 1;\ n_1 + n_2 \leq n; \\ \quad n_1 \leq n_2;\ m_1, m_2 \geq 0; \\ \quad m_1 + m_2 \leq m\} \quad \text{if n} > 1. \end{cases}$$

An easy proof by induction shows that $T_3(n,m) \leq nm$. This implies that $T(n,m)$ is $O(nm\log(n^2/m))$ and in turn that the total running time of the algorithm is of the same order.

3.5. Additional observations

We conclude this section with two observations. First, note that a complete set of minimum cuts for all values of λ can be represented in $O(n)$ space: store with each vertex $v \notin \{s,t\}$ the breakpoint at which v moves from the sink side to the source side of a minimum cut, for a set of minimum cuts whose source sides are nested. The breakpoint algorithm can be augmented to compute this information without affecting its asymptotic time bound. Second, the time bound of the three algorithms in Sections 3.1-3.3 can be improved to

$O(n_A m \log(n_A^2/m + 2))$ if G is bipartite and $\kappa(\lambda)$ has $O(n_A)$ breakpoints. Here n_A is the size of the smaller half of the bipartite partition of V. This bound follows using the bipartite variant of the preflow algorithm mentioned at the end of Section 2.5 (see [41] for details).

4. Applications

In this section, we give a number of applications of the algorithms in Sections 2 and 3. For each application, we obtain an algorithm running in $O(nm \log(n^2/m))$ time, where n is the number of vertices and m is the number of arcs in the graph involved in the problem. For applications in which the graph is bipartite, the bound is $O(n_A m \log(n_A^2/m + 2))$, where n_A is the size of the smaller half of the bipartite partition of the vertex set. (When the latter bound is applicable, we shall state it within square brackets.) Depending upon the application, our bound is a factor of from $\log n$ to n better than the best previously known bound. Our applications fall into four general categories: Flow sharing problems, zero-one fractional programming problems, parametric zero-one polynomial programming problems, and miscellaneous applications.

4.1 Flow sharing

Consider a network with a set of sources $S = \{s_1, s_2, \cdots, s_k\}$ and a single sink t, in which we want to find a flow from the sources in S to t. We require flow conservation at vertices not in $S \cup \{t\}$. We can model this problem as an ordinary one-source, one-sink problem by adding a supersource s and an arc (s, s_i) of infinite capacity for each $i \in \{1, \cdots, k\}$. The resulting network can have many different maximum flows, with different utilizations of the various sources; we define the utilization u_i of source s_i to be the flow through the arc (s, s_i). The question arises of how to compare the quality of such flows. Suppose each source s_i has a positive weight w_i. Several figures of merit have been proposed, leading to the following optimization problems:

i. *Perfect sharing*: Among flows with u_i/w_i equal for all $i \in \{1, \cdots, k\}$, find one that maximizes the flow value $e(t)$.

ii. *Maximin sharing*: Among maximum flows, find one that maximizes the smallest u_i/w_i, $i \in \{1, \cdots, k\}$.

iii. *Minimax sharing*: Among maximum flows, find one that minimizes the largest u_i/w_i, $i \in \{1, \cdots, k\}$.

iv. *Optimal sharing*: Among maximum flows, find one that simultaneously maximizes the smallest u_i/w_i and minimizes the largest u_i/w_i, $i \in \{1, \cdots, k\}$.

v. *Lexicographic sharing*: Among maximum flows, find one that lexicographically maximizes the k-component vector whose j-th component is the j-th smallest u_i/w_i, $i \in \{1, \cdots, k\}$.

Flow sharing problems with one source and multiple sinks are completely analogous to the multiple-source case: merely exchange source and sinks and reverse the network. For the criteria (ii)-(v), one can even allow multiple sources and multiple sinks, and simultaneously optimize one criterion for the sources and a possibly different criterion for the sinks. This is because each of problems (ii)-(v) calls for a maximum flow, and a multiple-source, multiple-sink problem can be decomposed into a multiple-source one-sink problem and a one-source multiple-sink problem, by finding a minimum cut of all sources from all sinks, contracting all vertices on the sink side to give a one-sink problem, and separately contracting all vertices on the source side to give a one-source problem. This observation is due to Megiddo [26].

Perfect sharing arises in a network transmission problem studied by Itai and Rodeh [21] and Gusfield [17] and in a network vulnerability model proposed by Cunningham [5]. We discuss these models below. Brown studied maximin sharing [3], Ichimori, Ishii, and Nishida [20] formulated minimax and optimal sharing, and Megiddo [26,27] studied lexicographic sharing. Motivation for these problems is provided by the following kind of example, which gives rise to a multiple sink problem. During a famine, relief agencies supplying food to the stricken areas want to distribute their available food supplies so that each person receives a fair share. The weight associated with each sink (famine area) is the population in that area, possibly adjusted for differences in food needs between adults and children, and other factors. A perfect sharing solution gives every person in every famine area the same amount of food, but it may be too restrictive since it need not allocate all the available and transportable food supply. A better solution will be provided by solving one of the problems (ii)-(v). There are analogous industrial interpretations of this model.

We shall show that all five flow sharing problems can be solved in $O(nm\log(n^2/m))$ time using the algorithms of Sections 2 and 3. The lexicographic sharing problem requires computing all the breakpoints of a min-cut capacity function by the algorithm of Section 3.3. The other four problems are easier, and can be solved by the algorithm for finding the smallest (or largest) breakpoint given in Section 3.1. Our tool for solving all five problems is the following parametric formulation: for each $s_i \in S$, let arc (s, s_i) have capacity $w_i\lambda_i$, where λ is a real-valued parameter. Since all arc capacities are nonnegative, the range of interest of λ is $[0, \infty)$. There are at most k breakpoints of the min-cut capacity function $\kappa(\lambda)$, one per source s_i.

Perfect sharing. Find the smallest breakpoint λ_s of $\kappa(\lambda)$. Any maximum flow for λ_s solves the perfect sharing problem. This was observed by Gusfield [17] in the context of the network transmission scheduling problem described below. Another application will arise in Section 4.2.

Scheduling transmissions. Itai and Rodeh state the following problem of scheduling transmissions in a "circuit-switched" communication network represented by a directed graph $G = (V, E)$ with fixed positive arc capacities. The capacity $c(v, w)$ is the effective transmission rate of the communication channel (v, w) in the direction from v to w, say in bits per second. A central vertex (sink) $t \in V$ receives all traffic that originates at a subset of vertices $S \subseteq V - \{t\}$ called *emitters* (sources). Each emitter $s_i \in S$ has $w_i > 0$ bits of information that it wishes to send to t. We assume that G has paths from each $s_i \in S$ to t. The communication protocol allows the sharing of arc capacities by several paths, but it requires that at least one path from s_i to t be established before transmission from s_i can begin. Clearly, if transmissions are scheduled from each emitter, one at a time, the entire task can be completed in $T' = \sum_{s_i \in S} w_i / c(X_i, \bar{X}_i)$ seconds, where $(X_i, \bar{X}_i)$ is a minimum cut separating s_i and t. But since arc capacities can be shared, it may be possible to obtain a lower value for T. The objective is to minimize the time T within which all transmissions can be completed.

To see that the problem is a perfect sharing multiple source problem, let $\lambda = 1/T$, in units of 1/second, and assign a capacity of $w_i\lambda$ bits per second to each arc (s, s_i) from the supersource s to an emitter $s_i \in S$. Once λ_s and the corresponding maximum flow have been computed by the algorithm described above, the actual transmission schedule can be constructed from the flow in $O(m)$ time as described in [21]. Itai and Rodeh proposed two algorithms for this problem, with running times of $O(kn^2m)$ and $O(k^2nm\log n)$. These are modifications of known maximum flow algorithms. In comparison, our algorithm runs in $O(nm\log(n^2/m))$ time.

Maximin sharing. Find the largest breakpoint λ_l of $\kappa(\lambda)$. Any maximum flow for λ_l solves the maximin sharing problem.

Minimax sharing. Find the smallest breakpoint λ_s of $\kappa(\lambda)$. Find a maximum flow for λ_s. Construct a residual network in which each arc (v,w) with $s \notin \{v,w\}$ has capacity $c(v,w)-f(v,w)$, each arc (s,s_i) has infinite capacity, and each arc (s_i,s) has zero capacity. Find a maximum flow f' in the residual network. The flow $f+f'$ is a minimax flow in the original network.

Optimal sharing. Find the smallest breakpoint λ_s and the largest breakpoint λ_l of $\kappa(\lambda)$. Find a maximum flow f for λ_s. Construct a residual network in which each arc (v,w) with $s \notin \{v,w\}$ has capacity $c(v,w)-f(v,w)$, each arc (s,s_i) has capacity $w_i(\lambda_l - \lambda_s)$, and each arc (s_i,s) has zero capacity. Find a maximum flow f' in the residual network. The flow $f+f'$ is an optimal flow in the original network.

Lexicographic sharing. Find all the breakpoints of $\kappa(\lambda)$. For each source s_i, let λ_i be the breakpoint at which s_i moves from the sink side to the source side of a minimum cut. For each arc (s,s_i) define its capacity to be $w_i\lambda_i$. Find a maximum flow f with these upper bounds on the capacities of the arcs out of s. Flow f is a lexicographic flow and hence an optimal flow.

The correctness of the first four algorithms above is easy to verify. We shall prove the correctness of the algorithm for the lexicographic sharing problem. Renumber the sources if necessary so that $\lambda_1 \le \lambda_2 \le \cdots \lambda_k$, and let G_∞ denote the parametric network with $\lambda = \infty$.

Theorem 4.1. On G_∞ there is a maximum flow f such that $f(s,s_i) = w_i\lambda_i$ for all i. Such a flow is a lexicographic flow.

Proof. Let $i_1, i_2, \cdots, i_{l-1}$ be the values of i such that $\lambda_{i_j} < \lambda_{i_j+1}$. Let $i_0 = 0$ and $i_l = k$. Then $\lambda_{i_1}, \lambda_{i_2}, \cdots, \lambda_{i_l}$, are the distinct breakpoints in increasing order. Let $\{s\} = X_0 \subset X_1 \subset X_2 \subset \cdots \subset X_l$ be the sets such that $(X_j, \bar{X}_j)$ for $1 \le j \le l$ is the minimum cut with the smallest sink side for $\lambda = \lambda_{i_j}$. Then $s_{i_{j-1}+1}, s_{i_{j-1}+2}, \cdots, s_{i_j} \in X_j - X_{j-1}$. For

$1 \le j \le l$, the cut $(X_{j-1}, \bar{X}_{j-1})$ is a minimum cut for $\lambda = \lambda_{i_j}$ as well, specifically the one with the smallest source side. Thus, $c_{\lambda_{i_j}}(X_{j-1}, \bar{X}_{j-1}) = c_{\lambda_{i_j}}(X_j, \bar{X}_j)$. It follows by induction on j that for $1 \le j \le l$,

$$c_{\lambda_{i_j}}(X_j, \bar{X}_j) = \sum_{i=1}^{i_j} w_i \lambda_i + \sum_{i=i_j+1}^{k} w_i \lambda_{i_j} \tag{4.1}$$

which implies that

$$c_{\infty}(X_j - \{s\}, \bar{X}_j) = \sum_{i=1}^{i_j} w_i \lambda_{i_j} \tag{4.2}$$

Equation (4.2) implies that any flow for G_∞ such that $f(s, s_i) = w_i \lambda_i$ for all i must be a maximum flow (choose $j = l$ in (4.2)). It must also be a lexicographic flow, since for all j, any flow that has $f(s, s_i) \ge w_i \lambda_i$ for $1 \le i \le i_{j-1}$ must either have $f(s, s_i) = w_i \lambda_i$ for $1 \le i \le i_j$ or have some $i \in \{i_{j-1}+1, \cdots, i_j\}$, such that $f(s, s_i) < w_i \lambda_i$.

It remains to show that G_∞ admits a flow f with $f(s, s_i) = w_i \lambda_i$. Consider running the min-cut parametric preflow algorithm presented in Section 2.5 on the parametric network, for the successive λ values $\lambda_{i_1}, \lambda_{i_2}, \cdots, \lambda_{i_l}$. Let $f_1, f_2, \cdots, f_l$ be the successive maximum preflows generated by the algorithm. When the min-cut preflow algorithm is restarted with a new value λ_{i_j} of λ, the flow on each arc (s, s_i) with $i \in \{i_{j-1}+1, \cdots, k\}$ is first increased from $w_i \lambda_{i_{j-1}}$ to $w_i \lambda_{i_j}$. All of this new flow successfully reaches the sink t, because of equation (4.1) and the fact that $(X_j, \bar{X}_j)$ is a minimum cut for $\lambda = \lambda_{i_j}$. It follows by induction on j that f_j is a flow and that $f_j(s, s_i) = w_i \lambda_i$ for $1 \le i \le i_j$. In particular, f_l is the desired flow. □

4.2. Fractional programming applications

Another class of problems that can be solved by the parametric preflow algorithm of Section 2.3 arises from various discrete and network optimization problems with fractional objectives. In general, the *fractional programming* problem is defined as

$$\lambda(x^*) = \max_{x \in S} \{ \lambda(x) = f(x)/g(x) \} , \tag{4.3}$$

where $f(x)$ and $g(x)$ are real-valued functions on a subset S of R^n, and $g(x) > 0$ for all $x \in S$. Isbell and Marlow [22] proposed an elegant solution method for the important case of linear f and g, but their approach has been extended to nonlinear problems (see e.g.

Dinkelbach [7]), and more recently to several classes of combinatorial problems (see e.g. Picard and Queyranne [32, 33], Padberg and Wolsey [30] and Cunningham [5]).

A problem that is intimately related to (4.3) is

$$z(x^*, \lambda) = \max_{x \in S} \{ z(x, \lambda) = f(x) - \lambda g(x) \} , \tag{4.4}$$

where λ is a real-valued constant. These two problems are related in the sense that x^* solves (4.3) if and only if (x^*, λ^*) solves (4.4) for $\lambda = \lambda^* = \lambda(x^*)$ giving the value $z(x^*, \lambda^*) = 0$. Isbell and Marlow's algorithm generates a sequence of solutions until this condition is met. We state their algorithm below in a form useful for our purposes (see e.g. Gondran and Minoux [14, pp.636-641]):

Algorithm FP:

Step 0. Select some $x^0 \in S$. Compute $\lambda_0 = f(x^0)/g(x^0)$. Set $k = 0$.

Step 1. Compute x^{k+1} solving the problem (4.4): $z(x^{k+1}, \lambda_k) = \max_{x \in S} z(x, \lambda_k) = f(x) - \lambda_k g(x)$.

Step 2. If $z(x^{k+1}, \lambda_k) = 0$, stop: $x^* = x^k$. Otherwise, let $\lambda_{k+1} = f(x^{k+1})/g(x^{k+1})$, replace k by $k+1$ and go to Step 1.

Theorem 4.2. Algorithm *FP* is correct. The sequence of values $\{\lambda_k\}$ generated by the algorithm is decreasing.

Proof. For any particular k, $z(x^{k+1}, \lambda_k)$ is nonnegative in Step 1, since $z(x^{k+1}, \lambda_k) \geq z(x^k, \lambda_k) = 0$. If $z(x^{k+1}, \lambda_k) = 0$, the algorithm halts with x^k which solves (4.4) for $\lambda = \lambda_k$ and hence solves (4.3). The algorithm continues only if $z(x^{k+1}, \lambda_k) > 0$, i.e. $f(x^{k+1}) - \lambda_k g(x^{k+1}) > 0$, which implies $\lambda_{k+1} = f(x^{k+1})/g(x^{k+1}) > \lambda_k$. □

If maximization is replaced by minimization in problem (4.3), it suffices to replace maximization by minimization in (4.4) and use the same algorithm. In this case all values of $z(x^{k+1}, \lambda_k)$ except the last one are less than zero, and a decreasing sequence $\{\lambda_k\}$ is generated. Another important observation is that in Step 1, the maximization (4.4) can be taken over a larger set $S' \supset S$, provided that $z(x, \lambda_k) \leq 0$ for all $x \in S' - S$. In the minimization problem, the corresponding requirement is $z(x, \lambda_k) \geq 0$ for all $x \in S' - S$. Several of our applications make use of this extension.

The following Lemma can be used to bound the number of iterations of Algorithm FP in some situations.

Lemma 4.3. $g(x^{k+1}) < g(x^k)$ for $k \geq 1$.

Proof. Consider iterations $k-1$ and k of Algorithm FP, and assume $\lambda(x^k) < \lambda(x^*)$. In iteration $k-1$ we have $z(x^k, \lambda_{k-1}) > 0$ and $\lambda_k = f(x^k)/g(x^k)$. In iteration k we have:

$$\begin{aligned} 0 < z(x^{k+1}, \lambda_k) &= f(x^{k+1}) - \lambda_k g(x^{k+1}) \\ &= f(x^{k+1}) - \lambda_{k-1} g(x^{k+1}) + \lambda_{k-1} g(x^{k+1}) - \lambda_k g(x^{k+1}) \\ &\leq f(x^k) - \lambda_{k-1} g(x^k) + \lambda_{k-1} g(x^{k+1}) - \lambda_k g(x^{k+1}) \\ &= \lambda_k g(x^k) - \lambda_{k-1} g(x^k) + \lambda_{k-1} g(x^{k+1}) - \lambda_k g(x^{k+1}) \\ &= (g(x^k) - g(x^{k+1}))(\lambda_k - \lambda_{k-1}), \end{aligned}$$

which implies that $g(x^k) > g(x^{k+1})$ since $\lambda_k > \lambda_{k-1}$. □

The efficiency of Algorithm FP depends upon the number of times problem (4.4) has to be solved, and on the time spent solving it. For continuous functions f and g defined on a nonempty compact set S, Schaible [37] has shown that the decreasing sequence $\{g(x^k)\}$ for $k \geq 1$ approaches $g(x^*)$ linearly, and the increasing sequence $\{\lambda_k\}$ approaches λ^* superlinearly. Nevertheless, (4.4) may be as hard to solve as the original fractional problem unless some assumptions are made about f, g and S. Fortunately, even the most restrictive assumptions find relevant applications in practice. For instance, if f and g are linear and S is polyhedral (the case in [22]), the algorithm consists of solving a finite number of linear programs (4.4) whose solution is implemented by ***cost-parametric programming*** on intervals $[\lambda_k, \lambda_{k+1}]$, for successive k. This can be specialized to network simplex parametric programming by using the primitives described by Grigoriadis [15]. If f is a negative semidefinite quadratic form and g is linear, the sequence of concave quadratic programs defined by (4.4) can be handled by the parametric algorithm of Grigoriadis and Ritter [16]. If f and g are negative and positive definite quadratic forms respectively, Ritter's parametric quadratic programming method [36] can be used. Approaches for more general nonlinear problems are analyzed in [7, 37].

If S is nonempty and finite, f is real-valued, and g is positive, integer-valued, and bounded above by some integer $p > 0$, Lemma 4.3 implies that Algorithm FP will terminate in $p+1$ or fewer iterations. This observation has been used in various applications where $g(x)$ is a set function, for which usually $p = O(n)$. Such is the case whether (4.3) is a maximization or a minimization problem.

We shall now describe a number of applications of the generic Algorithm FP. In each case the sequence of problems (4.4) that arise can be handled by our parametric preflow algorithm or its min-cut variant described in Section 2.5.

Strength of a directed network. This is an application due to Cunningham [5, Section 6]. Let $G = (V, E)$ be a given directed graph with n vertices, m arcs, nonnegative arc weights and nonnegative vertex weights, and a distinguished vertex $s \in V$. We assume that every $v \in V - \{s\}$ is reachable from s in G. The arc weight $c(v, w)$ represents the cost required to "destroy" the arc $(v, w) \in E$. The node weight d_v is the "value" attributed to having v reachable from s. Destroying a set of edges $A \subseteq E$ (at a total cost of $f(A) = \sum_{(v,w) \in A} c(v, w)$) may cause some subset of vertices $V_A \subseteq V - \{s\}$ to become unreachable from s, resulting in a loss of $g(A) = \sum_{v \in V_A} d_v$ in total value. The ratio $f(A)/g(A)$ is the cost per unit reduction in value. Cunningham defines the ***strength of the network*** to be the minimum of this ratio taken over all subsets $A \subseteq E$ whose removal reduces the value of the network, i.e. such that $g(A) > 0$. This is a problem of the form (4.3):

$$\lambda(A^*) = \min_{A \subseteq E, g(A) > 0} \{ \lambda(A) = f(A)/g(A) \} ,$$

which leads to a sequence of problems (4.4) that Cunningham calls *attack problems*:

$$z(A^{k+1}, \lambda_k) = \max_{A \subseteq E} \{ z(A, \lambda_k) = f(A) - \lambda_k g(A) \} .$$

Each such problem amounts to finding a minimum cut in an expanded network formed by adding to G a sink t and an arc (v, t) with capacity $\lambda_k d_v$ for each $v \in V - \{s\}$. If we solve the strength problem using Algorithm FP and use the algorithm of Section 2.3 to compute minimum cuts for the generated parameter values, we obtain an algorithm running in $O(nm \log(n^2/m))$ time; as Cunningham notes, there can be only $O(n)$ iterations of Step 1. Alternatively, we can make use of his observation that (4.5) is zero if and only if there is flow in the expanded network such that $f(v, t) = \lambda_k d_v$ for each $v \in V$. Equivalently, $\lambda(A^*)$ is the largest value of λ for which the minimum cut is $(V, \{t\})$. That is, the strength problem is a perfect sharing problem, and it can be solved in $O(nm \log(n^2/m))$ time as described in Section 4.1. Either method improves over Cunningham's method, which solves $O(n)$ minimum cut problems without making use of their similarity.

Zero-one fractional programming. An important subclass of (4.3) is the problem for which $f(x) \geq 0$ and $g(x) > 0$ are given polynomials defined for all x in $S = \{0,1\}^n - \{0\}^n$ as follows:

$$f(x) = \sum_{P \in A} a_P \prod_{i \in P} x_i \;+\; \sum_{i=1}^{n} a_i x_i \tag{4.5}$$

$$g(x) = \sum_{Q \in B} b_Q \prod_{i \in Q} x_i \;+\; \sum_{i=1}^{n} b_i x_i. \tag{4.6}$$

The sets A and B are given collections of nonempty nonsingleton subsets of $\{1, \cdots, n\}$, $a_P \geq 0$ for each $P \in A$, and $b_Q \leq 0$ for each $Q \in B$. Since $f(x) \geq 0$ and $g(x) > 0$ for all $x \in S$, we have $a_i \geq 0$ and $b_i > 0$ for each $i \in \{1, \cdots, n\}$. This problem was studied by Picard and Queyranne [32]. For ease in stating time bounds we assume $n = O(|A| + |B|)$.

Algorithm FP leads to a sequence of problems of the form (4.4) for increasing values $\lambda_k \geq 0$ of λ. Each such problem is an instance of the *selection* or *provisioning* problem, characterized by Rhys [35] and Balinski [2] as a minimum cut problem on a bipartite graph.

The entire sequence of these problems can be handled as a parametric minimum cut problem of the kind studied in Section 2. We give two different formulations, one of which works for the special case of $B = \emptyset$ (i.e. $g(x)$ contains no nonlinear terms) and the other of which works for the general case. All the subsequent applications we consider fall into the case $B = \emptyset$.

If $B = \emptyset$, we define a bipartite network G whose vertex set contains one vertex for each set $P \in A$, one vertex for each $i \in \{1, \cdots, n\}$, and two additional vertices, a source s and a sink t. There is an arc (s, v) of capacity a_P for each vertex v corresponding to a set $P \in A$, an arc (i, t) of capacity $\lambda b_i - a_i$ for every $i \in \{1, \cdots, n\}$, and an arc (v, i) of infinite capacity for every vertex corresponding to a set $P \in A$ that has i as one of its elements. Observe that the capacities of all arcs into t are nondecreasing functions of λ, and those of all other arcs are constant. The parametric preflow algorithm operates on G^R instead of G. For a given value of λ, a minimum cut $(X, \bar{X})$ in G corresponds to a solution x to (4.4) defined by $x_i = 1$ if $i \in X, x_i = 0$ if $i \in \bar{X}$.

In the general case $(B \neq \emptyset)$, it is convenient to assume $f(x) > 0$ for some x (otherwise the solution to (4.3) is $\lambda(x) = 0$, attained for any x) and that algorithm FP starts with an

x^0 such that $\lambda_0 > 0$. Then, the entire sequence $\{\lambda_k\}$ is strictly positive. To solve (4.4) we rewrite it as follows:

$$z(x^*, \lambda) = \lambda \max_{x \in S} (f(x)/\lambda - g(x)) .$$

We define the network G to have a vertex set consisting of one vertex for each set $P \in A$, one vertex for each set $Q \in B$, one vertex for each $i \in \{1, \cdots, n\}$, and a source s and a sink t. There is an arc (s, v) of capacity a_P/λ for each v corresponding to a set $P \in A$, an arc (s, v) of capacity $-b_Q$ for each v corresponding to a set $Q \in B$, an arc (v, i) of infinite capacity for each vertex v corresponding to a set $P \in A$ or $Q \in B$ that has i as one of its elements, and an arc (i, t) of capacity $b_i - a_i/\lambda$ for each $i \in \{1, \cdots, n\}$. The capacities of arcs out of s are nonincreasing functions of λ and those of arcs into t are nondecreasing functions of λ; the parametric preflow algorithm operates on G^R. Minimum cuts in G correspond to solutions exactly as described above.

Remark. This formulation differs from that in [32] because of the division by λ. The formulation of [32] gives a parametric minimum cut problem in which the capacities of arcs out of the source and of arcs into the sink are nondecreasing functions of λ, to which the results of Section 2 do not apply.

The following analysis is valid for both of the above network formulations. The nesting property of minimum cuts (Lemma 2.4) implies that the number of iterations of Step 1 of Algorithm FP is $O(n)$, a fact also observed by Picard and Queyranne [32]. To state time bounds, let us denote by n' and m' the number of vertices and edges, respectively, in G; $n' = n + |A| + |B| + 2$ and $m' = n + |A| + |B| + \sum_{P \in A} |P| + \sum_{Q \in B} |Q|$. Algorithm FP, in combination with the parametric preflow algorithm of Section 2.3, yields a time bound of $O(n'm' \log(n'^2/m))$ [or $O(nm' \log(n^2/m' + 2))$], improving over the algorithms of Picard and Queyranne [32] and Gusfield, Martel and Fernandez-Baca [19].

Maximum-ratio closure problem. This problem was considered by Picard and Queyranne [33] and independently by Lawler [24], who only considered acyclic graphs (see the next application). The problem can be solved by a straightforward application of Algorithm FP. Each problem in the sequence of problems (4.4) is a maximum-weight closure problem. The *maximum-weight closure problem* (Picard [31]) is the generalization to nonbipbartite graphs of the selection or provisioning problem [2,35] mentioned above.

These closure problems are defined formally as follows. Let $G = (V, E)$ be a directed graph with vertex weights a_v of arbitrary sign. A subset $U \subseteq V$ is a *closure* in G if for each arc $(v, w) \in E$ with $v \in U$ we also have $w \in U$. A closure $U^* \subseteq V$ is of *maximum weight* if the sum of its vertex weights is maximum among all closures in G. To compute a maximum-weight closure, construct the graph G^* from G as follows. Add a source s and a sink t to G. Create an arc (s, v) of capacity a_v and an arc (v, t) of zero capacity for each $v \in V$. Assign infinite capacity to all arcs in E. A minimum cut $(X, \bar{X})$ of G^* gives the desired closure $U^* = X - \{s\}$.

Now let $a_v \geq 0$ and $b_v > 0$ be given weights on the vertices of $G = (V, E)$. The *maximum-ratio closure* problem is to find a closure U^* that maximizes the ratio $a(U)/b(U)$ over all nonempty closures $U \subseteq V$. To compute a maximum-ratio closure, Picard and Queyranne [33] suggest the use of Algorithm FP. This requires the solution of a sequence of $O(n)$ maximum-weight closure problems, each of which is a minimum cut problem. Thus an $O(n^2 m \log(n^2/m)$-time algorithm results. Lawler's algorithm uses binary search and runs in $O(knm \log(n^2/m))$ time, where $k = \log(\max\{n, a_{\max}, b_{\max}\})$, assuming integer weights.

We can solve the entire sequence of minimum cut problems by the parametric preflow algorithm of Section 2.3 as follows. Modify G^* so that for each vertex $v \in V$ there is an arc (s, v) of capacity $a_v - \lambda b_v$ and an arc (v, t) of capacity zero. All other arcs have infinite capacity. We start with $U^0 = V \cup \{s\}$; or, equivalently, with a sufficiently small value of λ so that the minimum cut is $(\{s\} \cup V, \{t\})$. Such a value is $\lambda_0 = \min_i a_i/b_i$. The capacities of arcs out of the source are nonincreasing functions of the parameter, and the parameter values are given on-line in increasing order. The parametric preflow algorithm operates on $(G^*)^R$ and runs in $O(nm \log(n^2/m))$ time, improving the bound of Picard and Queyranne by a factor of n and that of Lawler by a factor of k.

Remark. Negative arc capacities in the various minimum cut problems can be made non-negative using the transformation suggested in Section 2.5. In the minimum-ratio closure problem, it suffices to assign a capacity of $\max\{0, a_v - \lambda b_v\}$ to each arc (s, v) and a capacity of $\max\{0, \lambda b_v - a_v\}$ to each arc (v, t).

A job sequencing application. Lawler [24] applied his algorithm to a problem studied by Sidney [38] and others: there are n jobs to be scheduled for processing on a single machine subject to a partial order given as an acyclic graph $G = (V, E)$, where V is the set of jobs.

Each job v has a processing time a_v and a "weight" $b_v > 0$ that describes its importance or some measure of profit. Let the completion time of job v as determined by a given feasible sequence be C_v. It is required to find a sequence that minimizes $\sum_{v \in V} b_v C_v$. This problem is NP-complete for an arbitrary partial order even when all $a_v = 1$ or all $b_v = 1$ [24]. Sidney offered the following decomposition procedure. First find a maximum-ratio closure U_1 such that $|U_1|$ is minimum. Remove the subgraph induced by U_1 from G, find a maximum-ratio closure U_2 of the reduced graph, and repeat this process until the entire vertex set is partitioned. Sidney and Lawler call closures ***initial sets*** of V. Once such a decomposition is found, an optimal schedule can be computed by finding an optimal schedule for each closure, for example by a branch-and-bound method, and then concatenating the solutions. The algorithm described above can be used to find each closure. The overall time bound depends upon the size of each closure. (Our algorithm will give closures of minimum cardinality, since the algorithm is applied to the graph $(G^*)^R$; see Section 2.5.)

Maximum density subgraph. A special case of the fractional programming problem (4.3) is that of finding a nonempty subgraph of maximum density in an undirected graph $G = (V, E)$ with n vertices and m edges. The *density* of a subgraph of G induced by a subset of vertices $V' \subseteq V$ is the number of its edges divided by the number of its vertices. For this application, (4.5) and (4.6) have the simpler forms $f(x) = \frac{1}{2}xAx$ and $g(x) = ex$, where e is the vector of all ones, A is the vertex-vertex incidence matrix of G, and $x_i = 1$ if vertex $i \in V'$ and $x_i = 0$ otherwise. Algorithm FP can be used to compute a maximum density subgraph of G. It is not necessary to construct a bipartite network and solve minimum cut problems on it. We can merely modify G by specializing the construction of [34]. Replace each edge of G by two oppositely directed arcs of unit capacity, add a source s and a sink t, and create an arc (s, v) of capacity $\delta_v - \lambda$ and an arc (v, t) of zero capacity for each $v \in V$, where δ_v is one-half the degree of vertex v in G. We can also allow weights on the edges and vertices. The resulting algorithm runs in $O(nm \log(n^2/m))$ time. This bound is better than that of Picard and Queyranne [32] by a factor of n, and better than that of Goldberg [12] by a factor of $\log n$; Goldberg's bound is valid only for the unweighted version of the problem.

4.3. Parametric zero-one polynomial functions

We consider the problem of computing a minimum of the function

$$z(\lambda) = \max_{x \in S} \{ f(x) - \lambda(dx - b) \} . \tag{4.7}$$

where $S = \{0,1\}^n$, $f(x)$ is a polynomial in zero-one variables defined by (4.5), and $d_i > 0$, $i \in \{1, \cdots, n\}$, such that $\sum_i d_i > b > 0$.

The function $z(\lambda)$ differs from the corresponding function (4.4) that arises in the zero-one fractional programming application of Section 4.2 because of the term λb in (4.7). The function $z(\lambda)$ is piecewise linear and convex, and it has at most $n-1$ linear segments and $n-2$ breakpoints. The network formulation of (4.7) is as defined for the zero-one fractional programming application. The breakpoints of $z(\lambda)$ coincide with those of the min-cut capacity function $\kappa(\lambda)$ for this network. In general, no minimum of $z(\lambda)$ coincides with a maximum of $\kappa(\lambda)$. To compute a minimum of $z(\lambda)$ we can use the algorithm of Section 3.2 for finding a maximum of $\kappa(\lambda)$ modified to use the graph of $z(\lambda)$ instead of the graph of $\kappa(\lambda)$ to guide the search. We have $z(0) = \sum_{P \in A} a_P + \sum_{i=1}^{n} a_i > 0$ (for $x = e$). The slope of the leftmost line segment of $z(\lambda)$ is $b - de < 0$, and the slope of the rightmost line segment is $b > 0$. The algorithm consists of the following three steps and finds a minimum of $z(\lambda)$ in $O(n'm' \log(n'^2/m'))$ [or $O(nm' \log(n^2/m' + 2))$] time. A cut $(X, \bar{X})$ in this network defines a solution x by $x_i = 1$ if vertex $i \in X$ and $x_i = 0$ otherwise.

Step 0. Start with initial values $\lambda_1 = 0$, $x^1 = e$, $z(\lambda_1) = f(x^1)$ and $h_1 = b - dx^1$. Choose λ_2 sufficiently large so that $x^2 = 0$; let $z(\lambda_2) = \lambda_2 b$ and $\beta_2 = b$.

Step 1. Compute $\lambda_3 = (z(\lambda_2) - z(\lambda_1)/(\beta_1 - \beta_2)$, pass λ_3 to two invocations, I and D, of the parametric preflow algorithm, and run them concurrently. If invocation I finds a minimum cut $(X_3^I, \bar{X}_3^I)$ first, suspend invocation D and go to Step 2 (the other case is symmetric.) Compute $\beta_3 = b - dx^3$, the slope of the line segment of $z(\lambda)$ derived from this cut.

Step 2. If $\beta_3 = 0$, stop: $\lambda^* = \lambda_3$. Otherwise, if $\beta_3 > 0$, replace λ_2 by λ_3, back up invocation D to its state before it began processing λ_3, and go to Step 1. Otherwise, go to Step 3.

Step 3 ($\beta_3 < 0$). Finish the invocation D for λ_3. This produces a minimum cut $(X_3^D, \bar{X}_3^D)$, not necessarily the same as $(X_3^I, \bar{X}_3^I)$. If $\beta_3 \geq 0$, stop: $\lambda^* = \lambda_3$. Otherwise, replace λ_1 by λ_3, back up invocation I to its state before processing λ_3, and go to Step 1.

We now describe an application of this algorithm.

Knapsack-constrained provisioning problems. We consider the following provisioning problem with a knapsack constraint that limits the weight of the selected items:

$$z(x^*) \;=\; \max_{x \in \{0,1\}^n} \{\; f(x) \;\mid\; dx \leq b \;\}, \tag{4.8}$$

where $f(x)$ is given by (4.5), d is a positive n-vector of item weights and b is a scalar, the knapsack size, such that $\sum_{i \in V} d_i > b > 0$, $V = \{1, \cdots, n\}$. Thus, in addition to the benefit a_i obtained for including an individual item $i \in V$ in the knapsack, the model allows the possibility of an additional reward of $a_P \geq 0$ for including all of the items that comprise a given subset $P \in A$. The (linear) ***knapsack problem*** is a special case of (4.8) in which all subsets $P \in A$ are singletons.

This NP-complete problem was suggested by Lawler [23]. Because of its many practical applications there is interest in the fast computation of bounds on $z(x^*)$. To this end we consider the Lagrangean function for (4.8):

$$L(x, \lambda) \;=\; f(x) - \lambda(dx - b), \quad \text{for } \lambda \geq 0,$$

which has a finite infimum over $x \in \{0,1\}^n$. For each $\lambda \geq 0$, we define the ***dual function***:

$$\Phi(\lambda) = \max_{x \in \{0,1\}^n} L(x, \lambda).$$

$\Phi(\lambda)$ is a piecewise linear convex function of λ, having at most $n-1$ line segments and $n-2$ breakpoints. We wish to solve the following *Lagrangean dual problem*:

$$\Phi(\lambda^*) = \min_{\lambda \geq 0} \Phi(\lambda).$$

This value is an upper bound on $z(x^*)$ and can be used to construct heuristics and search procedures for computing an approximate or exact solution to (4.8). It can be evaluated by the above algorithm in $O(n'm' \log(n'^2/m'))$ [or $O(nm' \log(n^2/m' + 2))$] time.

A special case of considerable practical importance is the ***quadratic knapsack*** problem, for which $f(x) = xAx$ where $A = [a_{ij}]$ is a nonnegative real symmetric matrix having no null rows. For this case, Gallo, Hammer and Simeone [11] proposed an $O(n^2 \log n)$-time algorithm for creating a class of "upper planes" bounding $z(x)$. Chaillou, Hansen and Mahieu [4] showed that its Lagrangean dual can be solved as a sequence of $O(n)$ minimum cut problems in $O(n^2 m \log(n^2/m))$ time.

The problem of evaluating $\Phi(\lambda)$ for a fixed λ can be formulated as a minimum cut problem using a graph construction similar to that described earlier for the maximum density subgraph problem, thereby avoiding the use of a bipartite graph. Let $G = (V, E)$ be a directed graph with vertex set $V = \{1, \cdots, n\}$, arc set $E = \{(v, w) : a_{vw} > 0, v, w \in V\}$, and arc weights $a(v, w) = a_{vw}$. We add to G a source s, a sink t, and an arc (s, v) of capacity $a_v - \lambda d_v$ and an arc (v, t) of zero capacity for each $v \in V$. Using this network formulation, the above algorithm computes the Lagrangean relaxation of a quadratric knapsack problem in $O(nm \log(n^2/m))$ time.

4.4. Miscellaneous applications

Our last two applications both use the algorithm developed in Section **3.3** for computing the min-cut capacity function $\kappa(\lambda)$ of a parametric minimum cut problem. The first application is to a problem of computing critical load factors for modules of a distributed program in a two-processor distributed system [42]. The second application is to a problem of record segmentation between primary and secondary memory in large shared databases [9].

Critical load factors in two-processor distributed systems. Stone [42] modeled this problem by a graph $G = (V, E)$ in which $V = \{1, \cdots, n\}$ is the set of program modules and E is the set of pairs of modules that need to communicate with each other. The capacity of an arc $(v, w) \in E$ specifies the communication cost between modules v and w (it is infinity if the modules must be coresident). The two processors, say A and B, are represented by the source s and the sink t, respectively, that are appended to the network. There is an arc (s, v) of capacity $\lambda b_v > 0$ where b_v is the given cost of executing program module v on processor B. There is an arc (v, t) of capacity $(1 - \lambda)a_v > 0$ where a_v is the given cost of executing program module v on processor A.

The parameter $\lambda \in [0, 1]$ is the fraction of the time processor A delivers useful cycles, commonly known as the *load factor*. For a fixed value of λ, a minimum cut $(X, \bar{X})$ in this network gives an optimum assignment of modules to processors. For $\lambda = 0$, a minimum cut $(X, \bar{X})$ with $|X|$ minimum has $X = \{s\}$, i.e. all modules are assigned to B. For $\lambda = 1$

a minimum cut $(X, \bar{X})$ with $|X|$ maximum has $\bar{X} = \{t\}$, i.e. all modules are assigned to A. The objective is to find the best assignment of program modules to processors for various values of λ, or to generate these assignments for each breakpoint of the min-cut capacity function $\kappa(\lambda)$. Lemma 2.4 implies that, at each breakpoint, one or more modules shift from one side of the cut to the other. By listing, for each module, the breakpoint at which it shifts from one side of the minimum cut to the other, one can determine what Stone calls the *critical load factor* for each module. The operating system can then use this list of critical load factors to do dynamic assignment of modules to processors. The algorithm of Section 3.3 will compute the critical load factors of the modules in $O(nm' \log((n+2)^2/m'))$ time, where $m' = m + 2n$. Stone does not actually propose an algorithm for this computation.

Record segmentation in large shared databases. Eisner and Severance [9] have stated a model for segmenting records in a large shared database between primary and secondary memory. Such a database consists of a set of data items $S = \{1, \cdots, N\}$ and serves a set of users $T = \{1, \cdots, n\}$. Each user $w \in T$ retrieves a nonempty subset $S_w \subseteq S$ of data items and receives a "value" (satisfaction) of $b_w > 0$ whenever all of the items in S_w reside in primary memory. The cost of transporting and storing a data item $v \in S$ in primary memory is $\lambda a_v > 0$, where $a_v > 0$. The scalar $\lambda > 0$ is a conversion factor such that λ units of transportation and storage costs equals one unit of user value. The objective is to find a segmentation that minimizes the total cost minus user satisfaction.

For a fixed value of λ the problem can be formulated as a selection or provisioning problem [2,35] as follows. Construct a bipartite graph having the data items S as its source part and the users T as its sink part. Construct an arc (v, w), $v \in S$, $w \in T$ of infinite capacity if data item v belongs to the set of data items S_w retrieved by user w. Create a supersource s and a supersink t, and append an arc (s, v) of capacity λa_v for each $v \in S$ and an arc (w, t) of capacity b_w for each $w \in T$. A min-cut $(X, \bar{X})$ separating s and t in this network necessarily partitions S and T into $(S_X, \bar{S}_X)$ and $(T_X, \bar{T}_X)$, respectively. It is easy to see that

$$c(X, \bar{X}) = \min_{S_X \cup T_X, \bar{S}_X \cup \bar{T}_X} \{ \lambda \sum_{v \in \bar{S}_X} a_v + \sum_{w \in T_X} b_w \}.$$

The value of λ plays an important role in this linear performance measure, and it depends upon the system load. In practice it is necessary to create a list of primary storage assignments for all critical values of λ. The database inquiry program can then select and implement the best assignment at appropriate times. This table consists of all the breakpoints of the min-cut capacity function $\kappa(\lambda)$ and, for each data item and user, the parameter value at which it moves from one side to the other of a minimum cut. This information can be computed by the breakpoint algorithm of Section **3.3** in $O((n+N)m\log((n+N)^2/m))$ [or $O(\min\{n,N\}m\log((\min\{n,N\})^2/m+2))$] time. The algorithm proposed by Eisner and Severance for solving the parametric problem requires the solution of $O(\min\{n,N\})$ minimum cut problems. Our algorithm improves their method by a factor of $\min\{n,N\}$. They also consider a nonlinear performance measure, for which an algorithm like that in Section 4.3 can be used to derive bounds on an optimum solution. This bounding method gives an approximate solution, and the method can be used in a branch-and-bound algorithm to give an exact solution.

5. Remarks

We have shown how to extend the maximum flow algorithm of Goldberg and Tarjan to solve a sequence of $O(n)$ related maximum flow problems at a cost of only a constant factor over the time to solve one problem. The problems must be instances of the same parametric maximum flow problem and the corresponding parameter values must either consistently increase or consistently decrease. We have further shown how to extend the algorithm to generate the entire min-cut capacity function of such a parametric problem, assuming that the arc capacities are linear functions of the parameter.

We have applied our algorithms to solve a variety of combinatorial optimization problems, deriving improved time bounds for each of the problems considered. Our list of applications is meant to be illustrative, not exhaustive. We expect that more applications will be discovered. Although we have only considered a special form of the parametric maximum flow problem, most of the parametric maximum flow problems we have encountered in the literature can be put into this special form.

We have discussed only sequential algorithms in this paper, but our ideas extend to the realm of parallel algorithms. Specifically, the preflow algorithm has a parallel version that runs in $O(n^2\log n)$ time using n processors on a parallel random-access machine. This version extends to the parametric preflow algorithm in exactly the same way as the sequential algorithm. Thus we obtain $O(n^2\log n)$-time, n-processor parallel algorithms for the problems considered in Sections 2 and 3 and for each of the applications in Section 4, where n is the number of vertices in the network.

There are a number of remaining open problems. One is to find additional applications. Possible applications include computing the arboricity of a graph [29,32] and computing properties of activity selection games [44]. Gusfield [18] has recently found a new application, to a problem considered by Cunningham [5], of solving the sequence of attack problems involved in the computation of the strength of an undirected graph. (This problem is related to the strength problem considered in Section 4.2 but is harder.)

Another area for research is investigating whether an arbitrary maximum flow algorithm can be extended to the parametric problem at a cost of only a constant factor in running time. One algorithm that we have unsuccessfully tried to extend in this way is that of Ahuja and Orlin [1]. Working in this direction, Martel [25] has recently discovered how to modify an algorithm based on the approach of Dinic [6] so that it solves the parametric problem with only a constant factor increase.

6. References

[1] R.K. Ahuja and J.B. Orlin, "A simple $O(nm + n^2 \log c_{\max})$ sequential algorithm for the maximum flow problem", Unpublished report, M.I.T. (1986).

[2] M. L. Balinski, "On a selection problem," *Man. Scie.* 17 (1970), 230-231.

[3] J. R. Brown, "The sharing problem", *Operations Research* 27 (1979), 324-340.

[4] P. Chaillou, P. Hansen, and Y. Mahieu, "Best network flow bounds for the quadratic knapsack problem," presented at the NETFLO 83 International Workshop, Pisa, Italy, 1983. To appear, *Lecture Notes in Mathematics*, Springer-Verlag, Berlin.

[5] W. H. Cunningham, "Optimal attack and reinforcement of a network", *J. Assoc. Comput. Mach.*, 32 (1985), 549-561.

[6] E. A. Dinic, "Algorithm for solution of a problem of maximum flow in networks with power estimation", *Soviet Math. Doklady* 11(1970), 1277-1280.

[7] W. Dinkelbach, "On nonlinear fractional programming", *Man. Scie.* 13 (1967), 492-498.

[8] J. Edmonds, "Minimum partition of a matroid into independent subsets", *J. Res. Nat. Bur. Standards* 69B (1965), 67-72.

[9] M. J. Eisner and D. G. Severance, "Mathematical techniques for efficient record segmentation in large shared databases," *J. Assoc. Comput. Math.* 23 (1976), 619-635.

[10] L. R. Ford, Jr. and D. R. Fulkerson, *Flows in networks*, Princeton University Press, Princeton, NJ 1962.

[11] G. Gallo, P. Hammer, and B. Simeone, "Quadratic knapsack problems," *Math. Programming* 12 (1980), 132-149.

[12] A. V. Goldberg, "Finding a maximum density subgraph," Technical Report No. UCB CSD 84/171, Computer Science Division (EECS), University of California, Berkeley, CA 1984.

[13] A. V. Goldberg and R. E. Tarjan, "A new approach to the maximum flow problem," *J. Assoc. Comput. Mach.* to appear; also *Proc. 18^{t}h Annual ACM Symp on Theory of Computing* (1986), 136-146.

[14] M. Gondran and M. Minoux, *Graphs and algorithms*, (translated by S. Vajda), John Wiley and Sons, New York, (1984).

[15] M. D. Grigoriadis, "An efficient implementation of the network simplex method", *Mathematical Programming Study* 26 (1986), 83-111.

[16] M. D. Grigoriadis and K. Ritter, "A parametric method for semidefinite quadratic programs", *SIAM J. Control* 7 (1969) 559-577.

[17] D. Gusfield, "On scheduling transmissions in a network," Technical Report YALEU DCS TR 481, Department of Computer Science, Yale University, New Haven, CT, 1986.

[18] D. Gusfield, "Computing the strength of a network in $O(|V|^3|E|)$ time", Technical Report CSE-87-2, Department of Electrical and Computer Engineering, University of California, Davis, CA (1987).

[19] D. Gusfield, C. Martel and D. Fernandez-Baca, "Fast algorithms for bipartite network flow", *SIAM J. Computing* 16 (1987), 237-251.

[20] T. Ichimori, H. Ishii and T. Nishida, "Optimal sharing", *Mathematical Programming* 23 (1982) 341-348.

[21] A. Itai and M. Rodeh, "Scheduling transmissions in a network," *J. Algorithms* 6 (1985), 409-429.

[22] J. R. Isbell and H. Marlow, "Attrition games", *Naval Res. Logistics Quart.* 2 (1956), 71-93.

[23] E. L. Lawler, *Combinatorial optimization: Networks and matroids*, Holt, Rinehart and Winston, New York (1976).

[24] E. L. Lawler, "Sequencing jobs to minimize total weighted completion time subject to precedence constraints," *Ann. Discrete Math.* 2 (1978), 75-90.

[25] C. Martel, "A comparison of phase and non-phase network algorithms", Technical Report CSE-87-7, Department of Electrical and Computer Engineering, University of California, Davis, CA (1987).

[26] N. Megiddo, "Optimal flows in networks with multiple sources and sinks", *Mathematical Programming* 7 (1974), 97-107.

[27] N. Megiddo, "A good algorithm for lexicographically optimal flows in multi-terminal networks," *Bull. American Math. Soc.* 83 (1979), 407-409.

[28] N. Megiddo, "Combinatorial optimization with rational objective functions," *Math. of Oper. Res.* 4 (1979), 414-424.

[29] C. St. J. A. Nash-Williams, "Decomposition of finite graphs into forests," *J. London Math. Soc.* 39 (1964), 12.

[30] M. W. Padberg and L. A. Wolsey, "Fractional covers and forests and matchings", *Mathematical Programming*, 29 (1984), 1-14.

[31] J.-C. Picard, "Maximal closure of a graph and applications to combinatorial problems", *Man. Scie.* 11 (1976), 1268-1272.

[32] J.-C. Picard and M. Queyranne, "A network flow solution to some nonlinear $0-1$ programming problems, with applications to graph theory," *Networks* 12 (1982), 141-159.

[33] J.-C. Picard and M. Queyranne, "Selected applications of minimum cuts in networks", *INFOR*, 20 (1982), 394-422.

[34] J.-C. Picard and H. D. Ratliff, "Minimum cuts and related problems," *Networks* 5 (1974) 357-370.

[35] J. M. W. Rhys, "A selection problem of shared fixed costs and network flows," *Man. Scie.* 17 (1970), 200-207.

[36] K. Ritter, "Ein verfahren zur losung parameterabhangiger, nichtlineare maximum probleme", *Unternehmensforchung*, 6 (1962), 149-166.

[37] S. Schaible, "Fractional programming II: On Dinkelbach's algorithm," *Man. Scie.* 22 (1976), 868-873.

[38] J. B. Sidney, "Decomposition algorithm for single-machine sequencing with precedence relations and deferral costs," *Oper. Res.* 23 (1975), 283-298.

[39] D. D. Sleator and R. E. Tarjan, "A data structure for dynamic trees," *J. Comput. System Sci.* 24 (1983), 362-391.

[40] D. D. Sleator and R. E. Tarjan, "Self-adjusting binary search trees," *J. Assoc. Comput. Math.* 32 (1985), 652-686.

[41] C. Stein, "Efficient algorithms for bipartite network flow", unpublished technical report, Deaprtment of Computer Science, Princeton University, Princeton, NJ (1986).

[42] H. S. Stone, "Critical load factors in two-processor distributed systems," *IEEE Trans. on Software Engineering* SE-4 (1978), 254-258.

[43] R. E. Tarjan, *Data structures and network algorithms,* Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[44] D. M. Topkis, "Activity selection games and the minimum-cut problem ", *Networks* 13 (1983), 93-105.