Balancing Applied to Maximum Network Flow Problems (Extended Abstract)

Robert Tarjan* Julie Ward Bin Zhang Yunhong Zhou Jia Mao**

Hewlett-Packard Laboratories
1501 Page Mill Rd, Palo Alto, CA 94304
{robert.tarjan, jward, bin.zhang2, yunhong.zhou}@hp.com

Abstract. We explore *balancing* as a definitional and algorithmic tool for finding minimum cuts and maximum flows in ordinary and parametric networks. We show that a standard monotonic parametric maximum flow problem can be formulated as a problem of computing a particular maximum flow that is *balanced* in an appropriate sense. We present a divide-and-conquer algorithm to compute such a balanced flow in a logarithmic number of ordinary maximum-flow computations. For the special case of a bipartite network, we present two simple, local algorithms for computing a balanced flow. The local balancing idea becomes even simpler when applied to the ordinary maximum flow problem. For this problem, we present a round-robin arc-balancing algorithm that computes a maximum flow on an n-vertex, m-arc network with integer arc capacities of at most U in $O(n^2 m \log(nU))$ time. Although this algorithm is slower by at least a factor of n than other known algorithms, it is extremely simple and well-suited to parallel and distributed implementation.

1 Introduction

In this paper we explore the idea of *balancing* as a definitional and algorithmic tool in the solution of maximum flow and minimum cut problems on capacitated networks. One motivation for introducing this concept is its application to monotone parametric flow problems. For such a problem, we define a λ-*balanced flow*, which is a maximum flow in which the flows on the parametrically-capacitated arcs are balanced in an appropriate way. From a λ-balanced flow it is easy to extract a maximum flow and a minimum cut for *any* value of the parameter. Thus a λ-balanced flow provides a succinct representation of the entire parameterized set of solutions of the parametric problem. We describe a divide-and-conquer algorithm that finds a λ-balanced flow in a logarithmic number of ordinary maximum flow computations.

Balancing can also be used as an algorithmic technique to develop simple, local algorithms for maximum flow problems. For the special case of the maximum flow problem on a bipartite parametric network, we develop two local algorithms

* Also Department of Computer Science, Princeton University.
** Dept. of Comp. Sci., UC San Diego. The work was done at HP Labs as an intern.

to compute a λ-balanced flow. One balances flow across a pair of arcs at a time, and the other balances flows across all arcs incident to a single vertex, a subgraph we call a *star*. For the ordinary maximum flow problem (not restricted to bipartite graphs), we develop an even simpler local algorithm that balances across one arc at a time. Such single-arc balancing was previously used by Awerbuch and Leighton [2, 3] in (more-complicated) approximation algorithms for finding multi-commodity flows. We provide a tight running-time analysis of our arc-balancing algorithm.

Our paper is organized as follows. The remainder of this section gives our basic network flow terminology. More specialized terminology is developed in later sections. Section 2 contains a discussion of some related work. Section 3 discusses the monotone parametric problem, discusses related work, defines λ-balanced flows, and gives our divide-and-conquer algorithm for finding such a flow. Section 4 presents our two local algorithms for finding a λ-balanced flow in a bipartite parametric network. Section 5 describes and analyzes our arc-balancing algorithm for ordinary maximum flow. Section 6 contains final remarks.

A *network* is a directed graph G with two distinguished vertices, a *source* s and a *sink* t, along with a non-negative capacity function c on the arcs. We allow an arc to have infinite capacity. We shall assume that G is *symmetric*: if (v, w) is an arc, so is (w, v). We sometimes refer to the pair (v, w), (w, v) as the *edge* $\{v, w\}$. We can make a network symmetric without affecting the maximum flow problem by adding an arc (w, v) with zero capacity for each arc (v, w) whose reversal (w, v) is not originally present. We assume (without loss of generality) that arcs into the source and out of the sink have zero capacity. In stating resource bounds we shall denote by n the number of vertices, by m the number of arcs, and by U the maximum arc capacity, assuming that the capacities are integers. We assume arbitrary-precision real arithmetic.

A *pseudoflow* on G is a real-valued function f on the arcs that is *antisymmetric*: $f(v, w) = -f(w, v)$ for every arc (v, w), and that obeys the *capacity constraints*: $f(v, w) \leq c(v, w)$ for every arc (v, w). Given a pseudoflow, the *excess* at a vertex v is $\sum\{f(u, v) \mid (u, v) \text{ is an arc}\}$. A pseudoflow is a *preflow* if $e(v)$ is non-negative for every vertex v other than s, and it is a *flow* if $e(v)$ is zero for every vertex v other than s and t. The *value* of a flow is $e(t)$. A flow is *maximum* if it has maximum value over all possible flows. The *maximum flow problem* is that of finding a maximum flow in a given network.

A problem dual to the maximum flow problem is the minimum (source-sink) cut problem. A *cut* is a partition of the vertex set into two parts, one containing the source, the other containing the sink. We shall denote a cut by the set X of vertices in the part of the partition containing the source. An arc (v, w) with v but not w in X *crosses* the cut. The *capacity* of the cut is $\sum\{c(v, w) \mid (v, w) \text{ crosses } X\}$. A cut is *minimum* if it has minimum capacity. The *minimum cut problem* is that of finding a minimum cut in a given network.

Given a pseudoflow, the *residual capacity* of an arc (v, w) is $r(v, w) = c(v, w) - f(v, w)$. An arc (v, w) is *saturated* if $r(v, w) = 0$; otherwise, it is *unsaturated* or *residual*. An *augmenting path* is a path of residual arcs; its capacity is the mini-

mum of the residual capacities of its arcs. A cut is *saturated* if every arc crossing it is saturated. The maximum-flow minimum-cut theorem implies that a flow is maximum and a cut is minimum if and only if the flow saturates every arc across the cut.

2 Related Work

Maximum Flows. The maximum flow problem is a central problem in network optimization. Goldberg and Rao [14] modified the blocking flow approach of Dinitz [7] by using a distance function based on the residual capacities (rather than just assigning each arc a length of one) and obtained an algorithm running in $O(\min\{n^{2/3}, m^{1/2}\}m \log(n^2/m) \log U)$ time on networks with integer arc capacities. For most interesting ranges of n, m and U, their algorithm is fastest. This algorithm uses dynamic trees [22] as well as other involved ideas.

Another approach that has led to both good theoretical bounds and fast practical implementations [6] is the *preflow-push method* of Goldberg and Tarjan [13]. The preflow push method is simple and local, although its theoretically fastest versions use sophisticated ideas. The use of dynamic trees gives an $O(nm \log(n^2/m))$ running time [13]. Combining excess-scaling with the use of dynamic trees gives an $O(nm \log((n/m)\sqrt{\log U} + 2))$ running time [1]. See [14] for a table showing a complete history of complexity improvements for the problem.

A third approach notable for its simplicity is the MA-ordering maximum flow algorithm of Fujishige [10], which runs in $O(n(m + n \log n) \log(nU))$ time, improvable to $O(nm \log U)$ by using scaling. A fast practical version of this algorithm has been obtained by adding the preflow idea [18].

Parametric Flows. The monotone parametric network flow problem is a generalization of the ordinary network flow problem in which each arc incident to the source has its capacity given by a monotone increasing function of λ, and each arc incident to the sink has its capacity given by a monotone decreasing function of λ, where λ is a common value in the range $[0, \infty)$. For any fixed value of λ, a parametric network has a minimum cut. The parametric minimum cut problem is to compute all minimum cuts for *every* possible value of λ. This problem has a variety of applications, including product selection [4, 21], repair kit selection [17], database record segmentation [8], and baseball team elimination [15, 19]. See Gallo et al. [11] and Hochbaum [16] for discussions of some of these applications. Gallo, Grigoriadis, and Tarjan [11] gave a divide-and-conquer method that for certain preflow-push maximum-flow algorithms has a time complexity asymptotically the same as one maximum flow computation. Their fastest algorithm runs in $O(nm \log(n^2/m))$ time. For a parametric bipartite network, their algorithm runs in time $O(km \log(k^2/m + 2))$ where k is the size of the smaller side of the bipartite graph. Gallo et al. also show that the capacities on the arcs into the sink can be made constant, with no loss of generality.

Balancing. The definition of a λ-balanced flow that we present in Section 3, and the two local algorithms for computing such a flow that we give in Section

4, were originally developed by the second and third authors [24, 25]: their work was the genesis of this paper. Awerbuch and Leighton [2] gave a local balancing algorithm for approximating maximum multi-commodity flow. The round-robin arc-balancing algorithm for ordinary single-commodity maximum flow that we give in Section 5 is closely related to (but simpler than) this algorithm. Awerbuch and Leighton [3] later gave a more-complicated but more-efficient balancing algorithm for approximating maximum multi-commodity flows. They did not explore the case of a single commodity, but for this special case of their algorithm a time bound of $O((n^2m/\epsilon^3)\log^3(m/\epsilon))$ to compute a $(1+\epsilon)$-approximation follows from their general bound.

3 Parametric Minimum Cuts and Balanced Flows

For our purposes, a *monotone parametric network* is one in which the capacity of each arc (s, v) is a strictly increasing function $c(\lambda, v)$ of a single parameter λ, for λ in the range $[0, \infty)$, and every other arc has fixed capacity. For simplicity we assume $c(0, v) = 0$ for all arcs (s, v). We further assume that there is a cut S of finite capacity containing every vertex v such that (s, v) is an arc. (In particular, this means that (s, t) is not an arc.) An important special case is $c(\lambda, v) = \lambda$ for every arc (s, v). The problem we wish to solve is to compute a minimum cut for every value of λ. This problem and related ones have a variety of applications [4, 21, 8, 17, 15, 19], with the latest application in webgraph clustering [9]. Even though there are an infinite number of possible λ values, there are only a finite number of possible cuts. Furthermore, in solving this problem it is only necessary to consider nested families of cuts: as the value of λ increases, the capacities of the arcs out of the source increase, and the minimum cut contains more-and-more vertices (more precisely, the minimum cut with fewest vertices contains more-and-more vertices) [8, 23]. Indeed, we can assign to each vertex v a value $\lambda(v)$ such that $S = S(\lambda) = \{v \mid \lambda(v) \leq \lambda\}$ is a minimum cut for λ [23]. (Vertices v with $\lambda(v) = \lambda$ can actually be included in S or not.) Thus the problem we solve is that of computing such a set of vertex values, which we call a *cut function*. Not only does a cut function define a nested family of minimum cuts, it also allows easy computation of the minimum cut capacity as a function of λ.

One way to compute a cut function is to find maximum flows for appropriately chosen values of λ. A straightforward application of this idea requires at most $n-2$ maximum flow computations. Gallo et al. [11] describe a complicated divide-and-conquer method that runs a preflow-push maximum-flow algorithm both forwards and backwards on each subproblem, and starts each subproblem with a preflow given by the solution to an enclosing subproblem. By a clever amortization argument, they show that the overall running time to compute a cut function is only a constant factor larger than the time bound for the preflow push algorithm. In particular, they obtain a bound of $O(nm\log(n^2/m))$ time to compute a cut function. For a bipartite network, they claim a time bound of $O(km\log(k^2/m+2))$, where k is the size of the smaller side of the bipartite graph.

The Gallo et al. algorithm is actually presented as a way to compute the minimum cut capacity function and it is restricted to affine arc capacity functions, but it is easily modified to compute a cut function, and Gusfield and Martel [15] show how to extend the algorithm to nonlinear arc capacity functions.

We reformulate the problem of computing a cut function as one of computing a particular maximum flow in the network formed by replacing all the parameterized capacities by ∞. Specifically, replace all capacities of arcs out of the source by ∞, and let f be a maximum flow of the resulting network. We can compute f by applying any maximum flow algorithm. For each arc (s, v), we define $\lambda(f, v)$ to be the unique value of λ such that $c(\lambda, v) = f(s, v)$. We call the flow f *λ-balanced* if there is no augmenting path avoiding s from a vertex v to a vertex w with $\lambda(f, v) < \lambda(f, w)$. The idea is that a λ-balanced flow is a maximum flow that, to the extent possible, equalizes the flows on the arcs out of s, where equalization is with respect to λ-values. In particular, if $c(\lambda, v) = \lambda$ for every arc (s, v), a λ-balanced flow is a maximum flow that, to the extent possible, equalizes the flows on the arcs out of s. In this special case, the notion of a λ-balanced flow is equivalent to that of a maximum flow that achieves the best possible lexicographic sharing. See Gallo et al. [11], Section 4.1 for a discussion of various notions of flow-sharing, including lexicographic sharing.

Given a λ-balanced flow f, we can obtain a cut function as follows. We set $\lambda(s) = 0$. For each arc (s, v), we let $\lambda(v) = \lambda(f, v)$. For each other vertex w, we let $\lambda(w) = \min\{\lambda(v) \mid \text{there is an augmenting path avoiding } s \text{ from } v \text{ to } w\}$, with the minimum over the empty set defined to be ∞. (In particular, $\lambda(t) = \infty$.) It is straightforward to compute this cut function in $O(n \log n + m)$ time by first sorting the vertices v such that (s, v) is an arc in non-decreasing order by $\lambda(v)$ and then doing an incremental graph search along residual arcs from vertices of small λ, increasing λ to the next possible value when the search runs out of arcs to traverse.

A straightforward way to attempt to compute a λ-balanced flow is to apply the definition. Specifically, begin by finding any maximum flow on the network with parameterized capacities replaced by ∞. Then repeat the following step until it no longer applies: find an augmenting path avoiding s from a vertex v to a vertex w with $\lambda(f, v) < \lambda(f, w)$. Send as much flow as possible along the cycle formed by this path and the arcs (w, s) and (s, v) without causing $\lambda(f, v)$ to exceed $\lambda(f, w)$. The amount of flow increase is the minimum of the capacity of the augmenting path and $c(\lambda, v) - f(s, v)$, where λ is the unique value such that $c(\lambda, v) - f(s, v) = f(s, w) - c(\lambda, w)$. Unfortunately, in general this algorithm does not terminate, although it leads to simple, efficient algorithms for the special case of bipartite networks, as we discuss in Section 4.

On the other hand, the Gallo et al. divide-and-conquer algorithm can be reformulated in a straightforward way to compute a λ-balanced flow, without affecting its asymptotic time bound. We develop a simpler version of this algorithm, which uses a maximum-flow computation as a black box. The initialization is to replace the capacities of the parameterized arcs by ∞ and compute a maximum flow. The main step of the algorithm computes a maximum flow on an

auxiliary network, thereby either making measurable progress toward producing a balanced flow or splitting the problem in two. Specifically, given a maximum flow, compute $\lambda(v)$ for every arc (s, v). If all these values are equal, the flow is balanced. Otherwise, choose a value λ strictly between the maximum and minimum of the $\lambda(v)$'s. Construct a dummy source s' and a dummy sink t', and construct dummy arcs as follows: if (s, v) is an arc with $\lambda(f, v) < \lambda$, construct a dummy arc (s', v) with capacity $c(\lambda, v) - f(s, v)$; if (s, v) is an arc with $\lambda(f, v) > \lambda$, construct a dummy arc (v, t') with capacity $f(s, v) - c(\lambda, v)$. Delete s and t and all incident arcs, and replace the capacity of all arcs not incident to s' and t' by their residual capacity. Find a maximum flow in the resulting auxiliary network. Add this flow (ignoring the flow on the new arcs) to the flow on the original network. Adjust the flow on the arcs out of s so that the new f is indeed a flow. (That is, restore flow conservation at each vertex v such that (s, v) is an arc.)

Note that this flow modification increases $\lambda(f, v)$ for any vertex v with $\lambda(f, v) < \lambda$ to at most λ, and decreases $\lambda(f, v)$ for any vertex v with $\lambda(f, v) > \lambda$ to at least λ. In the auxiliary network, one of three things can happen. The maximum flow on the original network corresponds to a saturated cut. If this cut is $\{s'\}$, then after the flow augmentation all arcs (s, v) have $\lambda(f, v) \geq \lambda$. If the cut is $V - \{t'\}$, then after the augmentation all arcs (s, v) have $\lambda(f, v) \leq \lambda$. In either of these cases the flow augmentation has reduced the range of the λ values, and we repeat the main step. In any other case, the problem can be split into two nontrivial subproblems, as follows. Let the saturated cut in the auxiliary network be S'. Let $S = S' \cup \{s\} - \{s'\}$, and let $\overline{S} = V - S$. Form one subproblem by contracting all vertices in S into s and then deleting all unparameterized arcs out of s. Form the other subproblem by contracting all vertices in $\overline{S}$ into t and deleting all parameterized arcs into t. The current maximum flow on the original network gives maximum flows on the subproblems. Convert these into λ-balanced flows by applying the method (minus the initialization) to each subproblem. The λ-balanced flows on the subproblems, plus the flows on the unparameterized contracted arcs, plus appropriate flows on the parameterized arcs, give a λ-balanced flow on the original network.

It remains to choose λ in each iteration of the main step. Since each subproblem contains an original arc (s, v), the number of subproblem splits is at most $n - 3$. If the value chosen for λ is such that the sum of the capacities of arcs out of s' equals the sum of arc capacities into t', then applying the main step either splits the problem or makes all λ's equal and hence completes the computation. Thus this choice of λ gives a bound of $n-3$ maximum flow computations. On the other hand, choosing λ equal to the average of the maximum and the minimum of the $\lambda(v)$'s guarantees that the difference between the maximum and the minimum of the $\lambda(v)$'s drops by a factor of two with each iteration of the main step, whether or not the problem is split. If we alternate between these two choices, the number of iterations of the main loop is $2 \min\{n - 3, 1 + \log R\}$, where R is the ratio between the initial difference between λ values and the minimum possible difference between λ values. In the important special case where all the

parameterized capacities are equal to λ, R is at most n^2U, giving us a bound of at most $2\min\{n-3, 1+\log(n^2U)\}$ on the number of maximum flow computations needed to find a λ-balanced flow. If we use the Goldberg-Rao algorithm for the maximum flow computations, we obtain a bound of

$$O\left(\min\left\{n^{2/3}, m^{1/2}\right\} \cdot m \cdot \log(n^2/m) \cdot \log U \cdot \min\{n, \log(nU)\}\right)$$

to find a λ-balanced flow. This bound is better than that of Gallo et al. for interesting ranges of the parameters. Goldberg and Rao [14] claim that their maximum flow algorithm in combination with ideas in the Gallo et al. paper [11] gives a bound of $O(\log U)$ times their maximum flow bound to find parametric flows. This bound is better than ours $O(\log(nU))$. But we can find no justification in the Gallo et al. paper for the Goldberg-Rao claim, even for simpler versions of the problem such as computing a maximum or minimum breakpoint of the minimum cut capacity function, and in a private conversation [12] Goldberg has retracted the claim.

As compared to the analysis of the Gallo et al. algorithm, our analysis is straightforward. In particular, it does not rely on starting the maximum flow computations on the subproblems with partially computed solutions, or on the fact that the sizes of the subproblems at a given level of recursion sum to at most the size of the original problem. We wonder if the $\log R$ factor in our time bound can be reduced or eliminated, either by doing a more sophisticated analysis, or by exploiting the internals of certain maximum flow algorithms. Another issue is to bound R for more-complicated parametric capacities, such as arbitrary linear or picewise-linear ones.

4 Local Algorithms for Bipartite Parametric Flow-Balancing

In this section we restrict our attention to the special case of a parametric network that is bipartite; that is, the vertices can be partitioned into two sets A and B such that every arc has one end in A and one end in B. We further assume that the sink t is in A (so that every arc (v,t) has v in B) and the source is in B. We also assume that *every* vertex in A is incident to s. This restriction still covers most of the interesting applications. (See [11].)

In this setting, we can efficiently convert a maximum flow into a λ-balanced flow by doing local balancing operations based on the definition of a λ-balanced flow. We describe two algorithms, one of which uses two-arc augmenting paths, and the other of which balances a star at each step, where a star is the subgraph induced by the set of arcs incident on a single vertex. The main advantage of these algorithms is that they are very simple as compared to algorithms based on standard maximum-flow algorithms.

Two-Arc-Balancing Algorithm: Begin with a maximum flow on the network with parameterized capacities replaced by ∞. Repeat the following step until it no longer applies: find an augmenting path $((v,u),(u,w))$ with $u \neq s$ and

$\lambda(f, v) < \lambda(f, w)$. Add as much flow as possible to this path without violating the capacity constraints or causing $\lambda(f, v)$ to exceed $\lambda(f, w)$. Increase the flow on (s, v) and decrease the flow on (s, w) by the same amount.

If this algorithm terminates (which it need not), the resulting flow is balanced. That is, if the network is bipartite with all vertices in A incident to s, then the definition of a balanced flow need only include two-arc augmenting paths.

We do not explore this algorithm further here, since there is a better alternative, which is to simultaneously consider all two-arc paths for a given intermediate vertex u. This idea gives the following algorithm:

Star-Balancing Algorithm: Begin with a maximum flow on the network with parameterized capacities replaced by ∞. Repeat the following step until it no longer applies: find a vertex $u \neq s$ such that there is an augmenting path $((v, u), (u, w))$ with $\lambda(f, v) < \lambda(f, w)$. Modify the flows on all arcs into u, and on corresponding arcs out of s, so that there is no such augmenting path.

The time required to do a star-balancing step depends on the complexity of the parameterized capacities. In the special case where all such capacities equal λ, or more generally where they are affine functions of λ, it is possible to star-balance a vertex u in time proportional to the degree of u [5]. The *round-robin star-balancing algorithm* is the version of star-balancing that repeatedly iterates over the vertices in B doing star-balancing steps, until an entire pass does not change the flow, or until a suitable stopping condition holds. (Vertices joined by an augmenting path have λ-values close enough that a simple postprocessing phase will complete the computation.)

A variant of round-robin star-balancing that may be more efficient in practice is to maintain a *working set* W of vertices in B whose last examination resulted in a change in the flow. Initially $W = B$. During a pass over W, if an examination of w results in sufficiently small flow change or no flow change, w is dropped from W. When W becomes empty, it is reset to be B. If a pass over $W = B$ results in no flow change, the computation is complete. A more-refined idea is to periodically find saturated cuts, as in the divide-and-conquer algorithm, and to split the working set into separate subsets based on the cuts.

At least for the special case in which all parameterized capacities are equal to λ, the running time of the round-robin star-balancing algorithm can be analyzed using the techniques we use to analyze the round-robin arc-balancing algorithm described in the next section, resulting in a similar running time. We omit this analysis and a discussion of appropriate stopping rules here since the ordinary maximum flow problem provides a simpler analytical setting.

In the important special case in which all arcs from A to B have infinite capacity, it is easy to construct an initial maximum flow in linear time without using a maximum-flow algorithm, by saturating all arcs into the sink, assigning flow to the arcs from A to B to get flow conservation on the vertices in B, for example by averaging the outgoing flow over the incoming arcs, and assigning

flow to the arcs out of s to get flow conservation on the vertices in A. It is also easier to do star balancing steps.

5 Ordinary Maximum Flows via Arc-Balancing

The idea of balancing flows can be fruitfully applied to the ordinary maximum flow problem. We develop an algorithm that computes a maximum flow by maintaining a pseudoflow and repeatedly moving flow along individual arcs so as to balance flow excesses. The algorithm has the virtue of being extremely simple and local, and it provides a simpler setting than the parametric flow problem in which to do algorithmic analysis. The key observation that justifies the algorithm is that if there is no augmenting path from a vertex of positive excess or the source to a vertex of negative excess or the sink, then both a minimum cut and a maximum flow can easily be extracted from the pseudoflow.

Arc-Balancing Algorithm: Construct an initial pseudoflow by saturating every arc out of s and every arc into t, and assigning zero flow to every other arc. Repeat the following step until it no longer applies (or until a suitable stopping rule holds):

> **Move**: Choose a residual arc (v, w) with $e(v) > e(w)$, $w \neq s$, and $v \neq t$. Increase the flow on (v, w) by $\min\{r(v, w), (e(v) - e(w))/2\}$.

A move on an arc (v, w) leaves $e(v) \geq e(w)$. Either it makes $e(v)$ and $e(w)$ equal, in which case we call it a *balanceing move*, or it saturates (v, w), in which case we call it a *saturating move*, or both. An *α-move* is a move on an arc of residual capacity at least α.

It is easy to construct examples on which this algorithm runs forever. Henceforth in this section we shall assume that the arc capacities are integers bounded by U. Given integer arc capacities, it suffices to stop doing moves when there are no $(1/n^2)$-moves. (We discuss another stopping rule below.) Furthermore one can restrict the moves to $(1/n^2)$-moves. Also, $\log(n^2U) + O(1)$ bits of precision suffice in the computations. (Unlike most maximum flow algorithms, our algorithm does *not* maintain integrality of flows.)

Two notions help in understanding the behavior of this algorithm. A *canonical cut* is a cut $S(a) = \{s\} \cup \{v \neq t \mid e(v) \geq a\}$ for some real value a. If the algorithm ever saturates a canonical cut, no move can ever again occur on any arc crossing the cut. A *section* $S(a, b)$ is a non-empty set of vertices $S(a) - S(b)$ such that both $S(a)$ and $S(b)$ are saturated. Once the algorithm creates a section, the minimum excess in the section can never decrease, the maximum excess in the section can never increase, and the *excess difference* in the section, defined to be the maximum excess minus the minimum excess, can never decrease. The initial pseudoflow defines a section containing all the vertices except s and t.

As the algorithm proceeds, it saturates canonical cuts, splitting the network into smaller-and-smaller sections. All moves take place within sections. To find a minimum cut, the only important section is the *active section*, defined to be

the minimal section $S(a, b)$ with $a \leq 0$ and $b > 0$. It suffices to do moves only in the active section. Another stopping rule for integer capacities is to stop when the sum of positive excesses in the active section is less than one or the sum of negative excesses in the active section is greater than minus one. As a special case, it suffices to stop when the maximum excess of the non-sink vertices is non-positive or the minimum excess of the non-source vertices is non-negative.

Once the arc-balancing algorithm stops, a minimum cut can be extracted as follows.

Minimum Cut Computation: If the main loop stops with the sum of positive excesses in the active section less than one, find the saturated cut $S = S(b)$ with minimum $b > 0$. Alternatively, if the main loop stops with the sum of negative excesses in the active section greater than minus one, find the saturated cut $S = S(a)$ with maximum $a \leq 0$.

The desired cut can be found in $O(m)$ time using graph search. It is a little more complicated to extract a maximum flow. The time to do so is $O(m \log n)$ using a dynamic tree data structure [22].

The arc-balancing algorithm is generic in the sense that the order of moves is unspecified. We can restrict the order of moves in a simple way that leads to a good theoretical time bound. We call the resulting method the *round-robin arc-balancing algorithm*. It consists merely of looping over a fixed list of the arcs, applying a move to each active arc, until an entire pass results in no $(1/n^2)$-moves. (Alternatively, we can stop when the active section has positive excesses summing to less than one or negative excesses summing to more than minus one.) The round-robin algorithm is particularly well-suited to massively parallel implementation.

Theorem 1. *The round-robin algorithm stops after $O(n^2 \log(nU))$ passes over the arcs, taking $O(m)$ time per pass, for a total time of $O(n^2 m \log(nU))$.*

The bound on passes in Theorem 1 is tight within a constant factor.

Theorem 2. *For any n and U, there is a network with n vertices and arc capacities bounded by U on which the round-robin algorithm does $\Omega(n^2 \log U)$ passes and $\Omega(n^3 \log U)$ moves.*

The example used to prove Theorem 2 shows that arc-balancing has trouble moving flow along long paths. Essentially the same example is bad for both the two-arc-balancing and the star-balancing round-robin algorithms for finding a λ-balanced flow on a parametric network. This motivates looking for longer-range balancing methods. We can use a modified form of star-balancing in place of arc-balancing in the ordinary maximum flow algorithm: the main change is that the excess of the vertex at the center of the star must be balanced against all those of the adjacent vertices. We have developed a linear-time algorithm for balancing a path of arbitrary length, and an $O(n \log^2 n)$-time algorithm for balancing an n-vertex tree. (That is, given an initial pseudoflow, modifying it so

that no arc-balancing moves are possible.) Both of these methods can be adapted to work for the bipartite parametric problem. These methods could be used as steps in computations on more-general networks, but so far we have not found a way to obtain a better overall running time for the ordinary maximum flow problem, for example.

6 Remarks

As we have discussed in this paper, balancing can be used as a conceptual and algorithmic tool for finding ordinary and parametric maximum flows and minimum cuts. Other types of applications may be able to use balancing ideas. Here we only mention one potential application in bioinformatics. In protein classification, we are given a graph whose edges may have weights. Vertices represent proteins; edges represent associational strengths. Certain proteins are classified by function. We wish to classify the unclassified proteins based on their associational strengths with classified ones. One possible approach is to compute, for an unclassified protein and for each group of classified proteins, an arc-balanced pseudoflow with the classified proteins as sources, and to regard the final excess at the unclassified protein as a measure of its associativity. In applying this idea it makes sense to make the edge capacities increasing functions of their weights and decreasing functions of their distance from classified proteins. See Singh et al. [20] for an approach to this problem using a similar method.

The balanced flow notion offers a succinct description of the solution to a parametric maximum flow or minimum cut problem, and it leads to novel algorithms for solving both parametric and ordinary maximum flow problems that use balancing as a local operation. Although our balancing algorithms are very simple, their time bounds are not competitive with those of existing algorithms. Nevertheless, we are optimistic that our techniques will lead to improved algorithms that are competitive in practice, if not in theory, with existing algorithms. Indeed, preliminary experiments on large real-world datasets suggest that the star-balancing algorithm for bipartite parametric problems computes solutions for all parameter values within a factor of two of the time taken by the fastest ordinary max-flow code [6] to compute a solution for a single parameter value. We intend to do additional experimental work. The idea of balancing in the context of maximum flow offers a rich space for research.

Acknowledgement: We thank Robert Schreiber and Andrew Goldberg for helpful discussions. We thank Qi Feng, a summer intern at HP Labs, for work on the product selection project, initially using maximum flow methods.

References

1. R. K. Ahuja, J. B. Orlin, and R. E. Tarjan. Improved time bounds for the maximum flow problem. *SIAM Journal on Computing*, 18:939–954, 1989.
2. B. Awerbuch and F. T. Leighton. A simple local-control approximation algorithm for multicommodity flow. In *Proc. FOCS*, pages 459–468. IEEE, 1993.

3. B. Awerbuch and T. Leighton. Improved approximation algorithms for the multi-commodity flow problem and local competitive routing in dynamic networks. In *STOC*, pages 487–496, 1994.
4. M. L. Balinski. On a selection problem. *Management Science*, 17(3):230–231, 1970.
5. M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, 1973.
6. B. V. Cherkassky and A. V. Goldberg. On implementing the push-relabel method for the maximum flow problem. *Algorithmica*, 19(4):390–410, 1997.
7. E. A. Dinic. Algorithm for solution of a problem of maximum flow in networks with power estimation. *Soviet Math. Dokl.*, 11:1277–1280, 1970.
8. M. J. Eisner and D. G. Severance. Mathematical techniques for efficient record segmentation in shared databases. *Journal of the ACM*, 23(4):619–635, 1976.
9. G. W. Flake, R. E. Tarjan, and K. Tsioutsiouliklis. Graph clustering and minimum cut trees. *Internet Mathematics*, 1(4):385–408, 2005.
10. S. Fujishige. A maximum flow algorithm using MA ordering. *Operations Research Letters*, 31:176 – 178, 2003.
11. G. Gallo, M. D. Grigoriadis, and R. E. Tarjan. A fast parametric maximum flow algorithm and applications. *SIAM J. Computing*, 18(1):30–55, 1989.
12. A. Goldberg. Private communication, 2006.
13. A. Goldberg and R. Tarjan. A new approach to the maximum flow problem. *Journal of the ACM*, 35(4):921–940, 1988.
14. A. V. Goldberg and S. Rao. Beyond the flow decomposition barrier. *Journal of the ACM*, 45(5):783–797, 1998.
15. D. Gusfield and C. Martel. A fast algorithm for the generalized parametric minimum cut problem and applications. *Algorithmica*, 7:499–519, 1992.
16. D. Hochbaum. Selection, provisioning, shared fixed costs, maximum closure, and implications on algorithmic methods today. *Management Science*, 50(6):709–723, 2004.
17. J. Mamer and S. Smith. Optimizing field repair kits based on job completion rate. *Management Science*, 28(11):1328–1333, 1982.
18. Y. Matsuoka and S. Fujishige. Practical efficiency of maximum flow algorithms using MA orderings and preflows. *J. Oper. Res. Soc. of Japan*, 48:297–307, 2005.
19. S. T. McCormick. Fast algorithms for parametric scheduling come from extensions to parametric maximum flow. *Operations Research*, 47:744–756, 1999.
20. E. Nabieva, K. Jim, A. Agarwal, B. Chazelle, and M. Singh. Whole-proteome prediction of protein function via graph-theoretic analysis of interaction maps. *Bioinformatics*, 21:i302–i310, 2005.
21. J. M. W. Rhys. A selection problem of shared fixed costs and network flows. *Management Science*, 17(3):200–207, 1970.
22. D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(1):362–391, 1983.
23. H. Stone. Critical load factors in two-processor distributed systems. *IEEE Trans. Software Engineering*, 4:254–258, 1978.
24. B. Zhang, J. Ward, and Q. Feng. A simultaneous parametric maximum flow algorithm for finding the complete chain of solutions. Technical report, HP Labs, 2004. `http://www.hpl.hp.com/techreports/2004/HPL-2004-189.html`.
25. B. Zhang, J. Ward, and Q. Feng. Simultaneous parametric maximum flow algorithm with vertex balancing. Technical report, HP Labs, 2005. `http://www.hpl.hp.com/techreports/2005/HPL-2005-121.html`.