Submodular Function Minimization

Based on Chapter 7 of the *Handbook on Discrete Optimization* [61]

Version 4*

S. Thomas McCormick †

June 19, 2013

Abstract

This survey describes the submodular function minimization problem (SFM); why it is important; techniques for solving it; algorithms by Cunningham [8, 12, 13], by Schrijver [76] as modified by Fleischer and Iwata [23], by Iwata, Fleischer, and Fujishige [50], by Iwata [46, 48], by Orlin [71], and by Iwata and Orlin [52] for solving it; and extensions of SFM to more general families of subsets.

1 Introduction

We start with a guide for the reader. If you don't know about submodularity, you should start with this Introduction. If you are already familiar with submodular functions but don't know the basic tools needed to build the algorithms, start with Section 2. If you just want to learn about recent algorithms, start with Section 3. Section 4 has a handy summary table comparing the main algorithms, and Section 5 covers some extensions of submodular flow minimization. This survey assumes some familiarity with network flow concepts, particularly those of Max Flow; see, e.g., Ahuja, Magnanti, and Orlin [1] for coverage of these.

1.1 What is Submodularity?

Suppose that our factory has the capability to make any subset of a given set E of potential products. If we decide to produce subset $S \subseteq E$ of products, then we must pay a setup cost $c(S)$ to make the factory ready to produce S. This setup cost is a particular instance of a set function: Given a finite set E (the *ground set*), the notation 2^E stands for the family of all subsets of E. Then a scalar-valued function $f : 2^E \to \mathbb{R}$ is called a *set function*. We write $f(S)$ for the value of f on subset $S \subseteq E$, and use n for $|E|$.

*Version 1 was published as [61] in 2006. Version 2 (May 2006) corrected some errors in the description of Hybrid. Version 3 (August 2007) corrected further Hybrid errors; added material on Orlin's Algorithm, on SFM on ring families, and on finding all SFM solutions; and updated and extended references. Version 3a (June 2008) clarified how the algorithms work on ring families and updated computational results. Version 4 (May 2013) added material on the Iwata-Orlin Algorithm, fixed some minor errors, and rearranged and modified some material.

†Sauder School of Business, University of British Columbia, Vancouver, BC V6T 1Z2 Canada. Supported by an NSERC Operating Grant, and by a visit to LIMOS, Université Blaise Pascal, Clermont-Ferrand.

Suppose that we have tentatively decided to produce subset S in our factory, and that we are considering whether to add product $e \notin S$ to our product mix. Then the incremental (or marginal) setup cost that we would have to pay is $c(S \cup \{e\}) - c(S)$. We deal with a lot of singleton sets, so to unclutter things we use the standard notation that $S + e$ means $S \cup \{e\}$, $S - e$ means $S - \{e\}$, and $f(e)$ means $f(\{e\})$. In this notation the incremental cost of adding e is $c(S + e) - c(S)$. We use $S \subset T$ to mean that $S \subseteq T$ but $S \neq T$.

Now economics suggests that in most real-world situations, this incremental cost is a non-increasing function of S. That is, adding new product e to a larger set should produce an incremental cost no more than adding e to a smaller set. In symbols, for a general function f we should have

$$\text{for all } S \subset T \subset T + e, \quad f(S + e) - f(S) \geq f(T + e) - f(T). \tag{1}$$

When any set function f satisfies (1), then we say that f is *submodular*. The connection between submodularity and economics suggested here is very deep; many more details about this are available in Topkis' book [83].

We say that f is *supermodular* if $-f$ is submodular, and *modular* if it is both sub- and supermodular. It is easy to see that f is supermodular iff it satisfies (1) with the inequality reversed, and modular iff it satisfies (1) with equality. The canonical (and essentially only) example of a modular function is derived from a vector $v \in \mathbb{R}^E$: For $S \subseteq E$, define $v(S) = \sum_{e \in S} v_e$ (so that $v(\emptyset) = 0$), and then $v(S)$ is modular. For example, if π_e is the net present value (NPV) of profits expected from producing product e (the value of the future stream of profits from producing e discounted back to the present), then $\pi(S)$ is the total NPV expected from producing subset S, and $\pi(S) - c(S)$ is the present value of net profits expected from producing S. Note that, because $\pi(S)$ is modular and $c(S)$ is submodular, $\pi(S) - c(S)$ is supermodular.

There is an alternate and more standard definition of submodularity that is sometimes more useful for proofs:

$$\text{for all } X, Y \subseteq E, \quad f(X) + f(Y) \geq f(X \cup Y) + f(X \cap Y). \tag{2}$$

We now show that these definitions are equivalent:

Lemma 1.1 *Set function f satisfies (1) if and only if it satisfies (2).*

Proof: To show that (2) implies (1), apply (2) to the sets $X = S + e$ and $Y = T$ to get $f(S+e) + f(T) \geq f((S+e) \cup T) + f((S+e) \cap T) = f(T+e) + f(S)$, which is equivalent to (1).

To show that (1) implies (2), first re-write (1) as $f(S + e) - f(T + e) \geq f(S) - f(T)$ for $S \subset T \subset T + e$. Now, enumerate the elements of $Y - X$ as $e_1, e_2, \ldots, e_k$ and note that, for $i < k$, $[(X \cap Y) \cup \{e_1, e_2, \ldots, e_i\}] \subset [X \cup \{e_1, e_2, \ldots, e_i\}] \subset [X \cup \{e_1, e_2, \ldots, e_i\}] + e_{i+1}$, so the re-written (1) implies that

$$\begin{aligned}
f(X \cap Y) - f(X) &\leq f((X \cap Y) + e_1) - f(X + e_1) \\
&\leq f((X \cap Y) \cup \{e_1, e_2\}) - f(X \cup \{e_1, e_2\}) \\
&\cdots \\
&\leq f((X \cap Y) \cup \{e_1, e_2, \ldots, e_k\}) - f(X \cup \{e_1, e_2, \ldots, e_k\}) \\
&= f(Y) - f(X \cup Y),
\end{aligned}$$

and this is equivalent to (2). ■

Here are some examples of submodular functions that arise often in practice:

Example 1.2 *Suppose that $G = (N, A)$ is a directed graph with nodes N and arcs A. For $S \subseteq N$ define $\delta^+(S)$ to be the set of arcs $i \to j$ with $i \in S$ but $j \notin S$; similarly, $\delta^-(S)$ is the set of $i \to j$ with $i \notin S$ and $j \in S$, and $\delta(S) = \delta^+(S) \cup \delta^-(S)$ (for an undirected graph, $\delta(S)$ is the set of edges with exactly one end in S). Recall that for $w \in \mathbb{R}^A$, notation $w(\delta^+(S))$ means $\sum_{e \in \delta^+(S)} w_e$. Then if $w \geq 0$, $w(\delta^+(S))$ (or $w(\delta^-(S))$, or $w(\delta(S))$), is a submodular function on ground set N.*

Example 1.3 *Suppose that $\mathcal{M} = (E, r)$ is a matroid (see Welsh [85] for further details) on ground set E with rank function r. Then r is a submodular function on ground set E. More generally, if r is a set function on E, we call r a* polymatroid rank function *if (i) $r(\emptyset) = 0$, (ii) $S \subseteq T \subseteq E$ implies $r(S) \leq r(T)$ (r is* increasing*), and (iii) r is submodular. Then the polyhedron $\{x \in \mathbb{R}^E \mid x \geq 0$ and $x(S) \leq r(S)$ for all $S \subseteq E\}$ is the associated* polymatroid*. For example, let $G = (N, A)$ be a Max Flow network with source s, sink t, and capacities $u \in \mathbb{R}^A$. Define $E = \{i \to j \in A \mid i = s\} = \delta^+(s)$, the subset of arcs with tail s. Then $\{x_{sj} \mid x$ is a feasible flow in $G\}$ (i.e., the projection of the set of feasible flows onto E) is a polymatroid on E. If S is a subset of the arcs with tail s, then $r(S)$ is the max flow value when we set the capacities of the arcs in $E - S$ to zero.*

Example 1.4 *Suppose that we have a set L of potential locations for warehouses. These warehouses are intended to serve the set R of retail stores. There is a fixed cost φ_l for opening a warehouse at $l \in L$, and the benefit to us of serving retail store $r \in R$ from $l \in L$ is b_{rl} (where $b_{rl} = -\infty$ if location l is too far away to serve store r). Thus if we choose to open warehouses $S \subseteq L$, our net benefit would be $f(S) = \sum_{r \in R} \max_{l \in S} b_{rl} - \sum_{l \in S} \varphi_l$. This is a submodular function.*

Example 1.5 *Suppose that we have a system of queues (waiting lines) $E = \{1, 2, \ldots, n\}$. For queue i, let x_i denote its throughput (the amount of work it processes) under some control policy (allocation of resources to the queues). Then the set of feasible throughputs is some set X in $\mathbb{R}^n$. We say that the system satisfies* conservation laws *if the maximum amount of work possible from the set of queues S, namely $f(S) = \max_{x \in X} \sum_{i \in S} x_i$, depends only on whether the queues in S have priority over other queues, and not on the priority order within S. Shanthikumar and Yao [78] show that if the system satisfies conservation laws, then $f(S)$ is submodular. Since any feasible x is non-negative, and this f is clearly increasing, then X is the polymatroid associated with f.*

For some applications f is not defined on all subsets of E. Suppose that $\mathcal{F} \subseteq 2^E$ is a family of subsets of E. If $\mathcal{F}$ is closed under unions and intersections, then we say that $\mathcal{F}$ is a *ring family*, or a *distributive lattice*, or a *lattice family*. If $\mathcal{F}$ is a ring family and we require (2) to hold only for members of $\mathcal{F}$, then we say that f is *ring submodular*. If instead we require that $S \cap T$ and $S \cup T$ are also in $\mathcal{F}$ only for all $S, T \in \mathcal{F}$ with $S \cap T \neq \emptyset$, then we call $\mathcal{F}$ an *intersecting family*. If $\mathcal{F}$ is an intersecting family and we require (2) to hold only for members of $\mathcal{F}$ with non-empty intersection, then we say that f is *intersecting submodular*. Finally, if we require that $S \cap T$ and $S \cup T$ are also in $\mathcal{F}$ only for all $S, T \in \mathcal{F}$ with $S \cap T \neq \emptyset$ and $S \cup T \neq E$, then we call $\mathcal{F}$ a

crossing family. If $\mathcal{F}$ is a crossing family and we require (2) to hold only for members of $\mathcal{F}$ with non-empty intersection and whose union is not E, then we say that f is *crossing submodular*. We consider more general families in Section 5.3.

Here are two examples of these specialized submodular functions:

Example 1.6 *Continuing with our introductory factory example, suppose that we have some precedences among products expressed by a directed graph $G = (E, C)$ on node set E, where arc $i \to j \in C$ means that any set containing product i must also contain product j. Then feasible sets are those $S \subseteq E$ such that $\delta^+(S) = \emptyset$, called* closed *sets. It is easy to see that these sets form a ring family, and reasonable to assume that the cost function $c(S)$ should be ring submodular on this family.*

When dealing with ring families it is necessary to have some sort of compact representation, as otherwise SFM would be hopeless: it could take an exponential number of calls to $\mathcal{E}$ to even discover some $S \in \mathcal{D}$. If $\mathcal{D}$ is a ring family, then $S_{\min} = \bigcap_{S \in \mathcal{D}} S$ and $S_{\max} = \bigcup_{S \in \mathcal{D}} S$ also belong to $\mathcal{D}$. When $S_{\min} \neq \emptyset$ or $S_{\max} \neq E$, then consider the reduced ground set $E' = E - S_{\min} - (E - S_{\max})$ and ring family $\mathcal{D}' = \{S \subseteq E \mid S \cup S_{\min} \in \mathcal{D}\}$, so that $S \in \mathcal{D}$ iff $S - S_{\min} \in \mathcal{D}'$ and $\emptyset$, $E' \in \mathcal{D}'$. Thus we don't lose anything by using E' and $\mathcal{D}'$ in place of E and $\mathcal{D}$, and so we henceforth assume that $\emptyset$, $E \in \mathcal{D}$ for our ring families. Then Birkhoff's Representation Theorem [7] says that all ring families have a representation as the family of closed sets of a directed graph (E, C). This is easy to see: Suppose that $\mathcal{D}$ is a ring family. Define C via $i \to j \in C$ iff j belongs to every $S \in \mathcal{D}$ s.t. $i \in S$, and then indeed the closed sets of (E, C) are $\mathcal{D}$. Conversely, given (E, C), its closed sets form a ring family. When we are dealing algorithmically with a ring family $\mathcal{D}$, we henceforth assume that we are given (E, C) as a representation of $\mathcal{D}$. Section 5.2 gives a general method for adapting an SFM algorithm for 2^E to work on a ring family $\mathcal{D}$.

Example 1.7 *Suppose that we have a connected directed graph $G = (N, A)$ with node $r \in N$ designated as the root, and weights $w \in \mathbb{R}^A$. We want to find a minimum weight arborescence rooted at r (spanning tree such that exactly one arc enters every node besides r, so that the unique path from r to any other node is a directed path). It can be shown (see [77, Section 52.4]) that one way to formulate this as an integer program as follows: Make a decision variable x_a for each $a \in A$ with the intended interpretation that $x_a = 1$ is a is included in the arborescence, and 0 otherwise. Let $\mathcal{T}$ be the family of non-empty subsets of N not containing r. Then the family of constraints $x(\delta^-(S)) \geq 1$ for all $S \in \mathcal{T}$ expresses that each such subset should have at least one arc entering it. The family $\mathcal{T}$ is intersecting, and the right-hand side $f(S) = 1$ for all $S \in \mathcal{T}$ is intersecting supermodular. Note that this is a very common way for submodular functions to arise, as right-hand sides in integer programming formulations (and their linear relaxations) of combinatorial problems.*

It is useful to have a mental model of submodularity to better understand it. Definition (1) tends to suggest that submodularity is related to concavity. Indeed, suppose that $g : \mathbb{R} \to \mathbb{R}$ is a scalar function, and set function f is defined by $f(S) = g(|S|)$. Then it is easy to show that f is submodular iff g is concave.

A deeper result by Lovász [58] suggests instead that submodularity is related to convexity. For $S \subseteq E$ define the *incidence vector* $\chi(S)$ of S as $\chi(S)_e$ equals 1 if $e \in S$, and 0 otherwise (we use χ_u to stand for $\chi(\{u\})$. This is a 1–1 map between 2^E and the vertices of the n-cube

$C_n = [0,1]^n$. If $v = \chi(S)$ is such a vertex, then f gives the value $f(S)$ to v. It is well-known that C_n can be dissected into $n!$ simplices, where the simplex $\sigma(\pi)$ corresponding to permutation π contains all $x \in C_n$ with $0 \leq x_{\pi(1)} \leq x_{\pi(2)} \leq \ldots \leq x_{\pi(n)} \leq 1$. Since f gives values to the vertices of $\sigma(\pi)$, there is a unique way to extend f to the interior of $\sigma(\pi)$ in a linear way. Let $\hat{f} : C_n \to \mathbb{R}$ denote the piecewise linear function which is these $n!$ linear extensions pasted together. This particular piecewise linear extension of f is called the *Lovász extension.*

Theorem 1.8 (Lovász [58]) *Set function f is submodular iff its Lovász extension $\hat{f}$ is convex.* ■

It turns out that this "convex" view of submodularity is much more fruitful than the "concave" view. In particular, Section 2.3 shows that, similar to convexity, minimizing a submodular function is "easy", whereas maximizing one is "hard". In fact, Murota [64, 65] has developed a theory of discrete convexity based on submodularity, in which many of the classic theorems of convexity find analogues.

For a more extensive look at submodular functions and their applications, consult Fujishige's book [29], Lovász's article [58], or Nemhauser and Wolsey [70, Section III.3].

1.2 What is Submodular Function Minimization?

Returning to our factory example, which subset should we choose? Clearly we should choose a subset that maximizes our future NPV minus our costs. That is, among the 2^n subsets of E, we want to find one that maximizes the supermodular function $\pi(S) - c(S)$. Maximizing $\pi(S) - c(S)$ is equivalent to minimizing $-(\pi(S) - c(S)) = c(S) - \pi(S)$, and $c(S) - \pi(S)$ is a submodular function of S. This leads to the core problem of this survey:

Submodular Function Minimization (SFM): $\min_{S \subseteq E} f(S)$, where f is submodular.

Here are some applications of SFM:

Example 1.9 *Let's change Example 1.2 a bit. Now we are given a directed graph $G = (N, A)$ with source $s \in N$ and sink $t \in N$ ($t \neq s$) and with non-negative weights $w \in \mathbb{R}^A$. Let $E = N - \{s, t\}$, and for $S \subseteq E$ define $f(S) = w(\delta^+(S + s))$. This f is again submodular, and SFM with this f is just the familiar s–t Min Cut problem. This also works if G is undirected, by redefining $f(S) = w(\delta(S + s))$.*

Example 1.10 *Continuing with Example 1.3, let $\mathcal{M}_1 = (E, r_1)$ and $\mathcal{M}_2 = (E, r_2)$ be two matroids on the same ground set. Then Edmonds' Matroid Intersection Theorem [16] says that the size of the largest common independent set equals $\min_{S \subseteq E} r_1(S) + r_2(E - S)$. The set function $f(S) = r_1(S) + r_2(E - S)$ is submodular, so this is again SFM. This also works for the intersection of polymatroids.*

Example 1.11 *As a different continuation of Example 1.3, suppose that we have a polymatroid P with rank function r, and that we are given some point $\bar{x} \in \mathbb{R}^E$ that satisfies $\bar{x} \geq 0$. The question is to determine whether $\bar{x} \in P$. To do this we need to verify the exponential number of inequalities $x(S) \leq r(S)$ for all $S \subseteq E$. We could do this by computing $g = \min_{S \subseteq E} r(S) - \bar{x}(S)$ via SFM (note that $r(S) - \bar{x}(S)$ is submodular), because if $g \geq 0$ then $\bar{x} \in P$, and if $g < 0$ then $\bar{x} \notin P$ (and the minimizing S gives a violated constraint). This* separation problem *(see Section 2.3) is a common application of SFM.*

Three recent models in supply chain management use SFM to compute solutions. Shen, Coullard, and Daskin [79] model a facility location-inventory problem related to Example 1.4, which they solve using a linear programming column generation algorithm. The column generation subproblem needs to find optimal subsets of demand points to be served by a facility, and this is an SFM problem. Begen and Queyranne [4] consider a problem of scheduling surgeries in operating rooms, and show that its objective function is discretely convex, which uses SFM in its solution. Huh and Roundy [44] model capacity expansion sequencing decisions in the semiconductor industry, where we trade off the declining cost of buying fabrication tools with the cost of lost sales from buying tools too late. The problem of determining an optimal sequence with general costs uses a (parametric) SFM subroutine. Additional applications to flows over time appear in Baumann and Skutella [3], and to Artificial Intelligence in Jeavons et al. [53].

1.3 Computational Models for SFM

A naive algorithm for SFM is to use brute force to look at the 2^n values of $f(S)$ and select the smallest, but this would take 2^n time, which is exponential, and hence impractical for all but the smallest instances. We would very much prefer to have an algorithm that is polynomial in n. The running time of an algorithm might also depend on the "size" of f as measured by, e.g., some upper bound M on $\max_S |f(S)|$. Since we could scale f to make M arbitrarily small, this makes sense only when we assume that f is integer-valued, and hence we implicitly so assume whenever we use M. An SFM algorithm that is polynomial in n and M is called *pseudo-polynomial*. To be truly polynomial, the running time must be a polynomial in n and $\log M$, leading to a *weakly polynomial* algorithm. If f is real-valued, or if M is very large, then it would be better to have an algorithm whose running time is independent of M, i.e., a polynomial function of n only, which is then called a *strongly polynomial* algorithm.

The first polynomial algorithms for SFM used the Ellipsoid method, see Section 2.3. Algorithms that avoid using Ellipsoid-like methods are called *combinatorial*. There appears to be no intrinsic reason why an SFM algorithm would have to use multiplication or division, so Schrijver [76] asks whether an SFM algorithm exists that is strongly polynomial, and which uses only additions, subtractions, and comparisons (such an algorithm would have to be combinatorial). Schrijver calls such an algorithm *fully combinatorial*. It is sometimes more convenient to hide logarithmic factors in running times, so we use the common notation that $\widetilde{O}(f(n))$ stands for $O(f(n) \cdot (\log n)^k)$ for some positive constant k.

This brings up the problem of how to represent the apparently exponential-sized input f in an algorithm. If we explicitly listed the values of f, then just reading the input would already be super-polynomial. The assumption we make to deal with this is that we have an *evaluation oracle* $\mathcal{E}$ available. We assume that $\mathcal{E}$ is a black box whose input is some set $S \subseteq E$, and whose output is $f(S)$. We use EO to stand for the time needed for one call to $\mathcal{E}$. For Example 1.2 with a reasonable representation for the graph, we would have $\text{EO} = O(|A|)$. Since the input S to EO has size $\Theta(n)$, it is reasonable to assume that $\text{EO} = \Omega(n)$. Section 2.2 shows how to compute a bound M on the size of f in $O(n\text{EO})$ time. Thus our hope is to solve SFM with a polynomial number of calls to $\mathcal{E}$, and a polynomial amount of other work.

1.4 Overview, and Short History of SFM

SFM has been recognized as an important problem since the early days of combinatorial optimization, when in the early 1970s Edmonds [16] established many of the fundamental results

that we use, which we cover in Sections 2.1 and 2.2.

When the Ellipsoid Algorithm arrived, in 1981 Grötschel, Lovász, and Schrijver [42] realized that it is a useful tool for finding polynomial algorithms for problems such as SFM; we cover these developments in Section 2.3. However, this result is ultimately unsatisfactory, since Ellipsoid is not very practical, and does not give much combinatorial insight. The problem shifted from "Is SFM polynomial?" to "Is there a combinatorial (i.e., non-Ellipsoid) polynomial algorithm for SFM?". In 1985 Cunningham [13] said that:

> It is an outstanding open problem to find a practical combinatorial algorithm to minimize a general submodular function, which also runs in polynomial time.

Cunningham made what turned out to be key contributions to this effort in the mid-80s by using a linear programming duality result of Edmonds [16] to set up a Max Flow-style algorithmic framework for SFM. We cover the LPs in Section 2.4, the network flow framework in Section 2.6, and Cunningham's applications of it [8, 12, 13] that yield a pseudo-polynomial algorithm for SFM in Section 3.1.

Then, nearly simultaneously in 1999, two working papers appeared giving quite different combinatorial strongly polynomial algorithms for SFM. These were by Schrijver [76] (formally published in 2000) and Iwata, Fleischer, and Fujishige (IFF) [50] (formally published in 2001). In 2006 Orlin [71] proposed a different SFM algorithm that is organized differently from the Schrijver and IFF algorithms, and which is significantly faster than either; a somewhat similar algorithm by Iwata and Orlin [52] was proposed in 2009. All of these are based on Cunningham's framework. We describe Schrijver's Algorithm in Section 3.2, , various versions of the IFF Algorithm in Section 3.3, and Orlin-type algorithms in Section 3.4.

All of these algorithms maintain a current point y as a convex combination of vertices of a polyhedron: $y = \sum_{i \in I} \lambda_i v^i$. As each algorithm proceeds, new vertices get added to the combination, and from time to time the algorithm needs to call a "Carathéodory" subroutine whose input is the convex combination representation of $y \in \mathbb{R}^E$, and whose output is a set of at most n of the v^i whose convex hull still contains y, see Section 2.5. This can be done using standard linear algebra techniques, but it is esthetically unpleasant. This led Schrijver [76] to pose the question as to whether there exists a fully combinatorial SFM algorithm. Iwata [46] found such an algorithm, based on the IFF Algorithm, which we call IFF-FC and describe in Section 3.3.3; there is also a faster fully combinatorial version of the Iwata-Orlin algorithm [52]. An alternate version of Schrijver's Algorithm using push-relabel ideas from Max Flow is given by Fleischer and Iwata [23] (which we call Schrijver-PR and cover into Section 3.2). A speedup of the IFF Algorithm (which uses ideas from both Schrijver and IFF, and which we call the Hybrid Algorithm) and Iwata's fully combinatorial version of it is given by Iwata [48], which we describe in Section 3.3.4. We compare and contrast these algorithms in Section 4, where we also give some guidelines on solving SFM in practice. We discuss various solvable extensions of SFM in Section 5, and we speculate about the future of SFM algorithms in Section 6. We note that Fleischer [21], Fujishige [27], Iwata [49], and Schrijver [77, Chapter 45] wrote other surveys of submodular function minimization.

We cannot cover it here in detail, but we note that there also exists some work on the structure of solutions to *parametric* SFM problems (where we want to solve a parametrized sequence of SFM problems), notably the work of Topkis [82, 83]. He shows that when a parametric SFM problem satisfies certain properties, then optimal SFM solutions are nested as a function of the parameter. Granot and Veinott [41] later extended this work. Fleischer and Iwata [23] extend

their Push-Relabel version of Schrijver's Algorithm to solve some parametric SFM problems in the same running time, and Nagano [69] extends Orlin's Algorithm in the same way. These are then used by Nagano [68] to minimize separable convex functions over a base polyhedron (see Section 2.1).

The SFM algorithms share a common heritage with algorithms for the Submodular Flow problem, a common generalization of Min Cost Flow and Matroid Intersection developed by Edmonds and Giles [17]; in particular IFF grew out of a Submodular Flow algorithm of Fleischer, Iwata, and McCormick [24]. In return, Fleischer and Iwata were able to show how to solve Submodular Flow in the same time as one call to IFF in [22]. The IFF algorithms have been further extended to minimizing *bisubmodular* functions. These are a directed, or signed, analogue of submodular functions, see Fujishige and Iwata [31], or McCormick and Fujishige [62].

2 Building Blocks for SFM Algorithms

This section builds up some tools that are common to all the SFM algorithms.

2.1 Greedy Optimizes over Submodular Polyhedra

Generalizing the polymatroids of Example 1.3 somewhat, for a submodular function f it is natural to consider the *submodular polyhedron* $P(f) = \{x \in \mathbb{R}^E \mid x(S) \leq f(S) \text{ for all } S \subseteq E\}$. For our arguments to be consistent for every case we need to worry about the constraint $0 = x(\emptyset) \leq f(\emptyset)$. To ensure that this makes sense, from this point forward we re-define $f(S)$ to be $f(S) - f(\emptyset)$ so that $f(\emptyset) = 0$; note that this change does not affect submodularity nor SFM. It turns out to be quite useful to consider the face of $P(f)$ satisfying $x(E) = f(E)$, the *base polyhedron*: $B(f) = \{x \in P(f) \mid x(E) = f(E)\}$. We prove below that $B(f)$ is never empty.

Given weights $w \in \mathbb{R}^E$, it is natural to wonder about maximizing the linear objective $w^T x$ over $P(f)$ and $B(f)$. Note that $y \leq x \in P(f)$ implies that $y \in P(f)$. Hence if $w_e < 0$ for some $e \in E$, then $\max w^T x$ is unbounded on $P(f)$, since we can let $x_e \to -\infty$. If $w \geq 0$, then the results below imply that an optimal x^* must belong to $B(f)$. Hence we can restrict our attention to solving the linear program (with dual variables in red):

$$\begin{array}{rrcll} & \max w^T x & & & \\ \text{s.t.} & \pi_S: \; x(S) & \leq & f(S) & \text{for all } S \subset E \\ & \pi_E: \; x(E) & = & f(E) & \\ & x_e & & \text{free} & \text{for all } e \in E \end{array} \tag{3}$$

It can be seen that LP (3) is bounded, and so we can relax and consider weights w of any sign. The dual of (3) has dual variable π_S for each $\emptyset \subset S \subseteq E$ and is (with primal variables in red):

$$\begin{array}{rrcll} & \min \sum_{S \subseteq E} f(S)\pi_S & & & \\ \text{s.t.} & x_e: \; \sum_{S \ni e} \pi_S & = & w_e & \text{for all } e \in E \\ & \pi_S & \geq & 0 & \text{for all } S \subset E \\ & \pi_E & & \text{free} & \end{array} \tag{4}$$

One remarkable property of submodularity is that the naive *Greedy Algorithm* solves this problem. Given a linear order $\prec$ of the elements of E, index the elements as $e_1, e_2, \ldots, e_n$ such that $e_1 \prec e_2 \prec \ldots \prec e_n$. For any $e \in E$, define $e^{\prec}$ as $\{e' \in E \mid e' \prec e\}$, a subset of E, so that

$e_i^{\prec} = \{e_1, e_2, \ldots, e_{i-1}\}$. Define $e_{n+1}^{\prec} = E$. Then Greedy takes $\prec$ as input, and outputs a vector $v^{\prec} \in \mathbb{R}^E$; component e_i of $v^{\prec}$ is then $v_{e_i}^{\prec}$.

The Greedy Algorithm with Linear Order $\prec$

For $i = 1, \ldots n$
  Set $v_{e_i}^{\prec} = f(e_{i+1}^{\prec}) - f(e_i^{\prec})$ $(= f(e_i^{\prec} + e_i) - f(e_i^{\prec}))$.
Return $v^{\prec}$.

To use this to maximize $w^T x$, index the elements as e_1, e_2, ..., e_n such that $w_{e_1} \geq w_{e_2} \geq \ldots \geq w_{e_n}$, and define $\prec_w$ as the linear order $e_1 \prec_w e_2 \prec_w \ldots \prec_w e_n$. Apply Greedy to $\prec_w$ to get $v^{\prec_w}$. Further define $w_{n+1} = 0$, and dual variables π_S^w as having value $w_{e_{i-1}} - w_{e_i}$ if $S = e_i^{\prec_w}$ $(= \{e_1, e_2, \ldots, e_{i-1}\})$, $i = 2, \ldots, n+1$, and zero otherwise.

Theorem 2.1 *The optimization version of Greedy runs in $O(n \log n + n\mathrm{EO})$ time, $v^{\prec_w}$ is primal optimal, π^w is dual optimal, and $v^{\prec_w}$ is a vertex of $B(f)$.*

Proof: Computing $\prec_w$ involves sorting the weights, which takes $O(n \log n)$ time. Otherwise, Greedy takes $O(n\mathrm{EO})$ time.

Now we prove that $v^{\prec_w} \in B(f)$. Note that $v^{\prec_w}(E) = \sum_{i=1}^n [f(e_{i+1}^{\prec_w}) - f(e_i^{\prec_w})] = f(E) - f(\emptyset) = f(E)$. So we just need to verify that for $\emptyset \subset S \subset E$, $v^{\prec_w}(S) \leq f(S)$. Define k as the largest index such that $e_k \in S$. We proceed by induction on k. For $k = 1$ we must have $S = \{e_1\}$, and $v^{\prec_w}(e_1) = v_{e_1}^{\prec_w} = f(e_2^{\prec_w}) - f(e_1^{\prec_w}) = f(e_1) - 0 = f(e_1)$, so $v^{\prec_w}(e_1) \leq f(e_1)$ is true.

For $1 < k < n$, note that $S \cup e_k^{\prec_w} = e_{k+1}^{\prec_w}$ and $S \cap e_k^{\prec_w} = S - e_k$. Hence (2) gives $f(S) \geq f(e_{k+1}^{\prec_w}) + f(S - e_k) - f(e_k^{\prec_w})$. Now $v^{\prec_w}(S) = f(e_{k+1}^{\prec_w}) - f(e_k^{\prec_w}) + v^{\prec_w}(S - e_k)$. By induction $v^{\prec_w}(S - e_k) \leq f(S - e_k)$, so we get $v^{\prec_w}(S) \leq f(e_{k+1}^{\prec_w}) - f(e_k^{\prec_w}) + f(S - e_k) \leq f(S)$, as required.

Now we prove that π^w is dual feasible. Suppose that $e = e_k$. Then $\sum_{S \ni e} \pi_S^w = \sum_{i=k}^n (w_{e_i} - w_{e_{i+1}}) = w_{e_k} = w_e$ as desired. By the ordering of E, $\pi_S^w \geq 0$ for all $S \subset E$, and it does not matter if π_E^w is negative.

Next, we prove that $v^{\prec_w}$ and π^w are complementary slack. First, $\pi_S^w > 0$ implies that $S = e_k^{\prec_w}$ for some k, and $v^{\prec_w}(e_k^{\prec_w}) = \sum_{i=1}^{k-1} [f(e_{i+1}^{\prec_w}) - f(e_i^{\prec_w})] = f(e_k^{\prec_w})$. Next, if $v^{\prec_w}(S) < f(S)$, then S cannot be one of the $e_k^{\prec_w}$, so $\pi_S = 0$. Hence $v^{\prec_w}$ and π^w are feasible and complementary slack, and thus optimal.

Recall that $v^{\prec_w}$ is a vertex of $B(f)$ if the submatrix of constraints where $\pi_S^w > 0$ is non-singular. This submatrix has rows which are a subset of $\chi(e_2^{\prec_w})$, $\chi(e_3^{\prec_w}), \ldots, \chi(e_{n+1}^{\prec_w})$, and these vectors are clearly linearly independent. ■

Suppose that $y \in P(f)$. We say that $S \subseteq E$ is *tight* for y if $y(S) = f(S)$, and we denote the family of tight sets for y by $\mathcal{T}(y)$. A corollary to this proof is that

$$\text{If } v^{\prec} \text{ is generated by Greedy from } \prec, \text{ then } e^{\prec} \in \mathcal{T}(v^{\prec}) \text{ for all } e \in E. \tag{5}$$

Note that when $w \geq 0$ then we get that $\pi_E^w \geq 0$ also, showing that the given solutions are also optimal over $P(f)$ in this case. We can also conclude from this proof that $B(f) \neq \emptyset$, and that every permutation of E generates a vertex of $B(f)$, and hence that $B(f)$ has a maximum

of $n!$ vertices. Our ability to generate vertices of $B(f)$ as desired is a key part of the SFM algorithms that follow.

The strongly polynomial version of IFF in Section 3.3.2 reduces SFM over 2^E to SFM over a ring family $\mathcal{D}$ represented by the closed sets of the directed graph (E, C), so we need to understand how these concepts generalize in that case. (We therefore henceforth refer to $e \in E$ as "nodes" as well as "elements".) In this case $B(f)$ is in general not bounded (we continue to write $B(f)$ for the base polyhedron over a ring family), because some of the constraints $x(S) \leq f(S)$ needed to bound $B(f)$ do not exist when $S \notin \mathcal{D}$. In particular, if (E, C) has a directed cycle Q and $l \neq k$ are nodes of Q, then for any $z \in B(f)$ we have $z + \alpha(\chi_l - \chi_k) \in B(f)$ for any (positive or negative) value of α, and so $B(f)$ cannot have any vertices. Section 3.3.2 deals with this by contracting strong components of (E, C), so we can assume that (E, C) has no directed cycles. Then we say that linear order $\prec$ is *consistent* with (E, C) (a consistent linear order is called a *linear extension* in [29, 46]) if $k \to l \in C$ implies that $l \prec k$, which implies that $e^{\prec} \in \mathcal{D}$ for every $e \in E$. The proof of Theorem 2.1 shows that when $\prec$ is consistent with $\mathcal{D}$, then $v^{\prec}$ is a vertex of $B(f)$.

If φ is a flow (not necessarily satisfying conservation) on (E, C), define $\partial\varphi : E \to \mathbb{R}$ by $\partial\varphi_k = \sum_l \varphi_{kl} - \sum_j \varphi_{jk}$, the net φ-flow out of node k, or *boundary* of φ. Then it can be shown (see Fujishige [29, Theorem 3.26]) that $w \in B(f)$ iff there is some y which is a convex combination of vertices $v^{\prec}$ for consistent $\prec$, and some flow $\varphi \geq 0$ such that $w = y + \partial\varphi$. Thus the boundaries of non-negative flows in (E, C) are precisely the directions of unboundedness of $B(f)$.

Section 3.3.2 also needs sharper bounds than M on y_e for $y \in B(f)$. For $e \in E$ define D_e, the *descendants* of e, as the set of nodes reachable from e via directed paths in (E, C). We know from (1) and Greedy that the earlier that e appears in $\prec$, the larger the value of $v_e^{\prec}$ is. Any consistent order must have all elements of $D_e - e$ coming before e. Therefore, an order $\prec^e$ putting D_e before all other nodes should maximize y_e, so we should have that $y_e \leq v_e^{\prec^e} = f(D_e) - f(D_e - e)$. The next lemma formalizes this.

Lemma 2.2 *If $y \in B(f)$ and y is in the convex hull of the vertices of $B(f)$, then $y_e \leq f(D_e) - f(D_e - e)$.*

Proof: It suffices to show that, for any $\prec$ consistent with (E, C), that $v_e^{\prec} \leq f(D_e) - f(D_e - e)$. From Greedy, $v^{\prec} = f(e^{\prec} + e) - f(e^{\prec})$. By consistency, $D_e \subseteq e^{\prec} + e$, and so by (1), $f(e^{\prec} + e) - f(e^{\prec}) \leq f(D_e) - f(D_e - e)$. ■

2.2 Algorithmic Tools for Submodular Polyhedra

Here is one of the most useful implications of submodularity:

Lemma 2.3 *If $S, T \in \mathcal{T}(y)$, then $S \cap T$, $S \cup T \in \mathcal{T}(y)$, i.e., the union and intersection of tight sets are also tight.*

Proof: Since $y(S)$ is modular, $f(S) - y(S)$ is submodular. Suppose that $S, T \in \mathcal{T}(y)$. Then by (2) and $y \in P(f)$ we get that $0 = (f(S) - y(S)) + (f(T) - y(T)) \geq (f(S \cup T) - y(S \cup T)) + (f(S \cap T) - y(S \cap T)) \geq 0$, which implies that we have equality everywhere, so we get that $S \cap T$, $S \cup T \in \mathcal{T}(y)$. ■

We use this to prove the useful fact that every vector in $P(f)$ is dominated by a vector in $B(f)$.

Lemma 2.4 *If $z \in P(f)$ and T is tight for z, then there exists some $y \in B(f)$ with $y \geq z$ and $y_e = z_e$ for $e \in T$.*

Proof: Apply the following generalization of the Greedy Algorithm: Start with $y = z$. Then for each $e \notin T$, iterate this step: compute (by brute force) $\alpha = \min\{f(S) - y(S) \mid e \in S\}$, and set $y \leftarrow y + \alpha\chi_{e_i}$. Since we start with $z \in P(f)$ and maintain feasibility throughout, we always have that $\alpha \geq 0$, and the final y must still belong to $P(f)$. Since only $e \notin T$ are changed, for the final y we have $y_e = z_e$ for $e \in T$.

At iteration e we find some set S_e that achieves the minimum. Thus, after iteration e, S_e is tight for y, and S_e remains tight for y for all iterations until the end. Then Lemma 2.3 says that $E = T \cup \bigcup_{e \notin T} S_e$ is also tight, and hence the final y belongs to $B(f)$. ■

The Greedy Algorithm in this proof raises a natural question: Given $y \in P(f)$ and $k \in E$, find the maximum step length we can move in direction χ_k while remaining in $P(f)$. Equivalently, compute $c(k; y) = \max\{\alpha \mid y + \alpha\chi_k \in P(f)\}$, which is easily seen to be equivalent to $\min\{f(S) - y(S) \mid k \in S\}$. A similar problem arises for $y \in B(f)$. In order to stay in $B(f)$ we must lower some component l while raising component k to keep $y(E) = f(E)$ satisfied. Equivalently, compute $c(k, l; y) = \max\{\alpha \mid y + \alpha(\chi_k - \chi_l) \in B(f)\}$, which is easily seen to be equivalent to $\min\{f(S) - y(S) \mid k \in S,\ l \notin S\}$ (which is closely related to Example 1.11). This $c(k, l; y)$ is called an *exchange capacity*. If we choose a large number K and define the modular weight function $w(S)$ to be $-K$ when k but not l is in S, $+K$ if l but not k is in S, and 0 otherwise, then $f(S) - y(S) + w(S)$ is submodular, and solving SFM on this function computes $c(k, l; y)$. The same trick works for $c(k; y)$. (Nagano [67] shows how to use IFF-FC to solve a more general line search problem over $P(f)$.)

In fact it can be shown that the converse is also true: Given an algorithm to compute $c(k, l; y)$ or $c(k, y)$, we can use it solve general SFM. This is unfortunate, as the algorithmic framework we'll see later would like to be able to compute $c(k, l; y)$ and/or $c(k, y)$, but this is as hard as the problem we started out with. However, there is one case where computing $c(k, l; y)$ is easy. We say that (l, k) is *consecutive* in $\prec$ if $l \prec k$ and there is no j with $l \prec j \prec k$. It can be shown [8] that the following result corresponds to a move along an edge of $B(f)$.

Lemma 2.5 *Suppose that $y = v^{\prec}$ is an extreme point of $B(f)$ arising from the Greedy Algorithm using linear order $\prec$. If (l, k) is consecutive in $\prec$, then*

$$c(k, l; y) = [f(l^{\prec} + k) - f(l^{\prec})] - [f(k^{\prec} + k) - f(k^{\prec})] = [f(l^{\prec} + k) - f(l^{\prec})] - v_k^{\prec},$$

which is non-negative.

Proof: Since (l, k) is consecutive in $\prec$, we have $k^{\prec} = l^{\prec} + l$, and so the expression is non-negative by (1).

Let y' be the result of the Greedy Algorithm with the linear order $\prec'$ that matches $\prec$ except that $k \prec' l$ (the same order with l and k switched). Note that y and y' match in every component except that $y_l = f(k^{\prec}) - f(l^{\prec})$ whereas $y'_l = f(k^{\prec} + k) - f(l^{\prec} + k)$, and $y_k = f(k^{\prec} + k) - f(k^{\prec})$, whereas $y'_k = f(l^{\prec} + k) - f(l^{\prec})$. Thus $y' = y + (\chi_k - \chi_l) \cdot ([f(l^{\prec} + k) - f(l^{\prec})] - [f(k^{\prec} + k) - f(k^{\prec})])$. Since the line segment defined by y and y' clearly belongs to $B(f)$, we get that $c(k, l; y) \geq$

$[f(l^{\prec}+k)-f(l^{\prec})]-[f(k^{\prec}+k)-f(k^{\prec})]$. But if $f(k,l;y)$ was strictly larger, then y' would not be an extreme point, so we get the desired result. ■

There is a similar result for $c(k;y)$.

Several of the algorithms avoid the difficulty of computing $c(k,l;y)$ by instead moving in more general directions defined by differences of vertices of $B(f)$. These differences always arise from closely-related linear orders like this: Suppose that we have linear orders $\prec$ and $\prec'$ such that $\prec=(e_1,e_2,\ldots,e_n)$ and $\prec'=(e_1,\,e_2,\,\ldots,\,e_k,\,e'_{k+1},\,e'_{k+2},\,\ldots,\,e'_l,\,e_{l+1},\,\ldots,\,e_n)$, i.e., $\prec'$ differs from $\prec$ only in that we have permuted the elements $B=\{e_{k+1},e_{k+2},\ldots,e_l\}$ of $\prec$ into some other order $e'_{k+1},e'_{k+2},\ldots,e'_l$ in $\prec'$. We call this move from $\prec$ to $\prec'$ a *block* modification of the block of size $b=l-k$. Then

> If we've already computed $v^{\prec}$, we can compute $v^{\prec'}$ using only $O(b)$ calls to $\mathcal{E}$ instead of $O(n)$ calls. (6)

This is because for $j\le k$ and $j>l$, $e_j^{\prec}=e'_j{}^{\prec}$, and so $v_j^{\prec}=v_j^{\prec'}$.

Now let's further suppose that we have a partition of B into non-empty sets Q and R, and that the order of elements in $\prec'$ is all elements of Q first in the same order as in $\prec$, followed by all elements of R in the same order as in $\prec$. For example, if $l=k+10$ and we relabel element e_h of B as q_h if it is in Q, and r_h if it is in R, then $\prec$ might look like

$$\ldots e_{k-1}e_k r_{k+1}r_{k+2}q_{k+3}r_{k+4}r_{k+5}r_{k+6}q_{k+7}q_{k+8}r_{k+9}q_{k+10}e_{l+1}e_{l+2}\cdots,$$

and then $\prec'$ would look like

$$\ldots e_{k-1}e_k q_{k+3}q_{k+7}q_{k+8}q_{k+10}r_{k+1}r_{k+2}r_{k+4}r_{k+5}r_{k+6}r_{k+9}e_{l+1}e_{l+2}\cdots.$$

In such cases we say that $\prec'$ differs from $\prec$ by a *block exchange*. If $|Q|=1$ or $|R|=1$ then we call this a *single element* block exchange, otherwise we call it a *general* block exchange. Then we get:

Lemma 2.6 *For each $q\in Q$ we have $v_q^{\prec'}\ge v_q^{\prec}$, and for each $r\in R$ we have $v_r^{\prec'}\le v_r^{\prec}$.*

Proof: Because of the assumption of how Q and R change in going from $\prec$ to $\prec'$, we have that $q\in Q$ implies that $q^{\prec'}\subseteq q^{\prec}$, and $r\in R$ implies that $r^{\prec}\subseteq r^{\prec'}$. Therefore by (1), for $q\in Q$ we have that $v_q^{\prec'}=f(q^{\prec'}+q)-f(q^{\prec'})\ge f(q^{\prec}+q)-f(q^{\prec})=v_q^{\prec}$, and similarly for $r\in R$. ■

The intuition to draw from Lemma 2.6 is that when we move elements to the left in linear orders we increase that component, and when we move elements to the right in linear orders we decrease that component. When (l,k) is consecutive in $\prec$, then the swap of l and k is a special case of a (single element) block exchange with $Q=\{k\}$ and $R=\{l\}$, and so Lemma 2.6 generalizes the non-negativity result of Lemma 2.5.

For vector v, define v^- via $v_e^-=\min(0,v_e)\le 0$, and $v_e^+=\max(0,v_e)\ge 0$. Computing the exact value $\max_{S\subseteq E}|f(S)|$ is hard (see Section 2.3.1), but we can easily compute a good enough bound M such that $|f(S)|\le M$ for all $S\subseteq E$: Pick any linear order $\prec$ and use Greedy to compute $v=v^{\prec}$. Then for any $S\subseteq E$, by (2) $v^-(E)\le v(S)\le f(S)\le\sum_{e\in E}f(e)^+$. Thus $M=\max(|v^-(E)|,\sum_{e\in E}f(e)^+)$ works as a bound, and takes $O(n\mathrm{EO})$ time to compute.

2.3 Optimization, Separation, and Complexity

Suppose that we have a class $\mathcal{L}$ of linear programs that we want to solve. We say that OPT($\mathcal{L}$) is the problem of computing an optimal solution for any LP in $\mathcal{L}$. The Ellipsoid Algorithm gives a generic way to solve OPT($\mathcal{L}$) as long as we have a subroutine to solve the associated *separation problem* SEP($\mathcal{L}$): Given an LP $L \in \mathcal{L}$ and a point $\bar{x}$, either prove that $\bar{x}$ is feasible for L, or find a constraint $a^T x \leq b$ that is satisfied by all feasible points of L, but violated by $\bar{x}$. Then Ellipsoid says that if SEP($\mathcal{L}$) is polynomial, then OPT($\mathcal{L}$) is also polynomial. In fact, Grötschel, Lovász, and Schrijver [42] were able to use polarity of polyhedra (which interchanges OPT and SEP) to also show the converse (modulo certain technicalities that we skip here):

Theorem 2.7 *OPT($\mathcal{L}$) is solvable in polynomial time iff SEP($\mathcal{L}$) is solvable in polynomial time.* ■

For ordinary LPs, SEP($\mathcal{L}$) is trivially polynomial: just look through all the constraints of L and plug $\bar{x}$ into each one. Either $\bar{x}$ satisfies each one, or we find some constraint violated by $\bar{x}$, and we output that. Thus the Ellipsoid Algorithm is polynomial for ordinary LPs.

However, consider "combinatorial" LPs where the number of constraints is exponential in the number of variables, as is the case for polymatroids in Example 1.3. Here the trivial separation algorithm is no longer polynomial in the number of variables, although Theorem 2.7 is still valid.

This is important for SFM since we can use an idea from Cunningham [11] to reduce SFM to a separation problem over a polymatroid. For $e \in E$ define $\gamma_e = f(E-e) - f(E)$. If $\gamma_e < 0$, then by (1) for any $S \subseteq E$ containing e we have $f(S-e) - f(S) \leq f(E-e) - f(E) = \gamma_e < 0$, or $f(S) > f(S-e)$. Hence e cannot belong to any solution to SFM, and without loss of optimality we can delete e from E and solve SFM on the reduced problem. Thus we can assume that $\gamma \geq 0$. Define $\tilde{f}(S) = f(S) + \gamma(S)$. Clearly $\tilde{f}$ is submodular with $\tilde{f}(\emptyset) = 0$, and for any $S \subset S+e \subseteq E$, $\tilde{f}(S+e) = \tilde{f}(S) + [\tilde{f}(S+e) - \tilde{f}(S)] = \tilde{f}(S) + [(f(S+e) - f(S)) + (f(E-e) - f(E))] \geq \tilde{f}(S)$ by (1), so $\tilde{f}$ is increasing. Thus $\tilde{f}$ is a polymatroid rank function.

Now consider the separation problem over $P(\tilde{f})$ with $\bar{x} = \gamma$. The optimization $\max_S \gamma(S) - \tilde{f}(S)$ yields the set S with maximum violation. But $\gamma(S) - \tilde{f}(S) = -f(S)$, so this also would solve SFM for f. So, if we could solve SEP for $P(\tilde{f})$, we could then use binary search to find a maximum violation, and hence solve SFM for f. But by Theorem 2.7 we can solve SEP for $P(\tilde{f})$ in polynomial time iff we can solve OPT for $P(\tilde{f})$ in polynomial time. But Theorem 2.1 showed that we can in fact solve OPT over $P(\tilde{f})$ in polynomial time. We have proved that the Ellipsoid Algorithm leads to a weakly polynomial algorithm for SFM. (Recently, Fujishige and Iwata [32] showed that there is a direct algorithm that needs only $O(n^2)$ calls to a separation routine to solve SFM.) In fact, later Grötschel, Lovász, and Schrijver were able to extend this result to show how to use Ellipsoid to get a strongly polynomial algorithm for SFM:

Theorem 2.8 (Grötschel, Lovász, Schrijver [43]) *The Ellipsoid Algorithm can be used to construct a strongly polynomial algorithm for SFM that runs in $\widetilde{O}(n^5 \mathrm{EO} + n^7)$ time.* ■

(The running time of this algorithm is quoted as $O(n^4 \mathrm{EO})$ in [74], but Lovász [59] relates that the previous computation was "too optimistic", and that the running time above is correct.)

This theorem establishes that SFM is technically "easy", but it is unsatisfactory in at least two ways:

- The Ellipsoid Algorithm has proven to be very slow in practice.
- This algorithm gives us very little insight into the combinatorial structure of SFM.

2.3.1 Submodular Function Maximization is Hard

Note that in Example 1.4 we are interested in *maximizing* the submodular function, i.e., solving $\max_S f(S)$. However, this example of submodular function maximization is known to be NP Hard (even when all $\varphi_l = 1$ and all b_{rl} are 1 or $-\infty$, since it is a special case of Min Dominating Set in a graph, see Garey and Johnson [34, Problem GT2]), so the general problem is also NP Hard. (However, Shen, Coullard, and Daskin [79] propose a related problem where we do want to solve SFM.) There are also many applications where we want to maximize the (submodular) cut function in Example 1.2, leading to the Max Cut problem (see Laurent [56]), and this is also NP Hard, see [34, Problem ND16]. Nemhauser and Wolsey [70, Section II.3.9], Krause and Golovin [54], and Krause and Guestrin [55] survey other applications and results about maximizing submodular functions, and Feige, Mirrokni, and Vondrák [20] give constant-factor approximation algorithms and hardness of approximation results for variations of the problem.

2.4 A Useful LP Formulation of SFM

Edmonds developed many of the basic concepts and results that led to SFM algorithms. In particular, all combinatorial SFM algorithms to date derive from the following idea from [16] (which considered only polymatroids, but the extension to general submodular functions is easy): Let $\mathbb{1}$ denote the vector of all ones, so that if $z \in \mathbb{R}^E$, then $\mathbb{1}^T z = z(E)$. Suppose that we are given an upper bound vector $x \in \mathbb{R}^E$ (data, not a variable), and we want to find a maximal vector (i.e., a vector $z \in \mathbb{R}^E$ whose sum of components $\mathbb{1}^T z$ is as large as possible) in $P(f)$ subject to this upper bound. This naturally formulates as the following linear program and its dual (with dual variables in red):

$$\begin{array}{rrcll} & \max \mathbb{1}^T z & & & \\ \sigma_e: & z_e & \le & x_e & \text{for all } e \in E, \\ \pi_S: & z(S) & \le & f(S) & \text{for all } S \subseteq E \\ & z_e & & \text{free} & \text{for all } e \in E; \end{array} \qquad \begin{array}{rrcll} & \min \sum_e x_e \sigma_e + \sum_{S \subseteq E} f(S)\pi_S & & & \\ z_e: & \sigma_e + \sum_{S \ni e} \pi_S & = & 1 & \text{for all } e \in E \\ & \sigma_e & \ge & 0 & \text{for all } e \in E \\ & \pi_S & \ge & 0 & \text{for all } S \subseteq E. \end{array}$$

One consequence of submodularity is that LPs like these often have integral optimal solutions when the data is integral. Edmonds saw that these LPs not only have integral optimal solutions, but also have the special property that there is a 0–1 dual solution with exactly one π_S having value 1. Assuming that this is true, let S^* be the subset of E such that $\pi_{S^*} = 1$. Then an optimal solution must have that $\sigma = \chi(E - S^*)$ to satisfy the dual constraint, and the dual objective becomes $x(E - S^*) + f(S^*)$. We now prove this:

Theorem 2.9 *The dual LP has a 0–1 optimal solution with exactly one $\pi_S = 1$. This implies that*

$$\max\{\mathbb{1}^T z \mid z \in P(f),\ z \le x\} = \min_{S \subseteq E}\{f(S) + x(E - S)\}. \tag{7}$$

If f and x are integer-valued, then the primal LP also has an integral optimal solution.

Proof: Note that (weak duality) $z(E) = z(S) + z(E-S) \le f(S) + x(E-S)$. Hence we just need to show that an optimal solution satisfies this with equality.

Recall that $\mathcal{T}(z)$ is the family of tight sets for z. By Lemma 2.3 we have that $S^* = \cup_{T \in \mathcal{T}(z)} T$ is also tight. If z is optimal and $z_e < x_e$, then there must be some $T \in \mathcal{T}(z)$ containing e, else we could feasibly increase z_e. Hence $z_e = x_e$ for all $e \notin S^*$. Thus we have $z(S^*) + z(E - S^*) = f(S^*) + x(E - S^*)$, and so the 0–1 π with $\pi_{S^*} = 1$, $\pi_S = 0$ for $S \neq S^*$, and $\sigma = \chi(E - S^*)$ is optimal.

If f and x are integer-valued, define $M' = \min(-M, \min_e x_e)$, so that $z = M'\mathbb{1}$ satisfies $z \in P(f)$ and $z \le x$. Now apply Greedy starting from this z and ensuring that $z \le x$ is preserved. By induction, z is integral at the current iteration, so that the exchange capacity used to determine the next step is also integral, so the next z is also integral. Hence the final, optimal z is also integral. ∎

One way we could apply this LP to SFM, which we call the *polymatroid approach*, is to recall from Section 2.3 Cunningham's reduction of SFM to a separation problem for the derived polymatroid function $\tilde{f}$ w.r.t. the point γ. Since $\tilde{f}(S) + \gamma(E - S) = f(S) + \gamma(E)$ (and since $\gamma(E)$ is a constant), minimizing $f(S)$ is equivalent to minimizing $\tilde{f}(S) + \gamma(E - S)$. As noted in Section 2.3 we can assume that $\gamma \ge 0$. Since $\tilde{f}$ is a polymatroid function we can use the detailed knowledge about polymatroids developed in [8]. Since $\tilde{f}(S) + \gamma(E - S)$ matches the RHS of (7), we can use Theorem 2.9 and its proof for help. Because we are assuming that $\gamma \ge 0$, we can in fact replace the condition $z \in P(f)$ in the LHS of (7) with $z \in \tilde{P}(\tilde{f}) = \{z \in P(\tilde{f}) \mid z \ge 0\}$, i.e., the polymatroid itself. We can recognize optimality when we have a point $z \in \tilde{P}(\tilde{f})$ and a set $S \subseteq E$ with $z(E) = \tilde{f}(S) + \gamma(E - S)$.

Alternatively, we could use the *base polyhedron approach*, which is to use Theorem 2.9 directly without modifying f, by choosing $x = 0$. Then (7) simplifies to

$$\max\{\mathbb{1}^T z \mid z \in P(f),\ z \le 0\} = \min_{S \subseteq E} f(S). \tag{8}$$

The RHS of this is just SFM. In this approach, it is more convenient to enforce that $z \in B(f)$ instead of $z \in P(f)$. When we switch from $z \in P(f)$ to $y \in B(f)$, in order to faithfully represent $z \le 0$ we change the objective function from $\max \mathbb{1}^T z$ to $\max \sum_e \min(0, y_e)$. The proof of Theorem 2.9 shows that for optimal z and S we have $z(S) = f(S)$, and Lemma 2.4 shows that this z is dominated by some $y \in B(f)$ with $y_e = z_e$ for $e \in S$, so this change does not harm the objective value. Recall that we defined y_e^- to be $\min(0, y_e)$. Then (8) becomes

$$\max\{y^-(E) \mid y \in B(f)\} = \min_{S \subseteq E} f(S). \tag{9}$$

(This result could also be derived directly from LP duality and an argument similar to Theorem 2.9.) It is easy to directly show weak duality for (9): For any $y \in B(f)$ and $S \subseteq E$,

$$\begin{array}{rcll} y^-(E) & \le & y^-(S) & (\text{tight if } y_e < 0 \Rightarrow e \in S) \\ & \le & y(S) & (\text{tight if } e \in S \Rightarrow y_e \le 0) \\ & \le & f(S) & (\text{tight if } y(S) = f(S)) \end{array} \tag{10}$$

Complementary slackness is equivalent to these inequalities becoming equalities as indicated above. Thus joint optimality is equivalent to $y^-(E) = f(S)$. Note that $y(E) = f(E) = y^+(E) + y^-(E)$, or $y^-(E) = f(E) - y^+(E)$, so we can think of the LHS of (9) as $\min y^+(E)$ if we prefer.

2.5 How Do We Know that Our Current Point is Feasible?

In either approach we face a difficult problem: How can the algorithm ensure that either $z \in \tilde{P}(\tilde{f})$ or $y \in B(f)$? Since both are described by an exponential number of constraints, there is no straightforward way to verify these.

A way around this comes from the following facts:

1. Since $B(f)$ and $\tilde{P}(\tilde{f})$ are bounded, a point belongs to them iff it is a convex combination of extreme points.

2. The extreme points $v^{\prec}$ of $B(f)$ and of $\tilde{P}(\tilde{f})$ are available to us from the Greedy Algorithm (or a simple modification of it, in the case of $\tilde{P}(\tilde{f})$).

3. By Carathéodory's Theorem, it suffices to use at most n extreme points for $B(f)$ (since $y \in B(f)$ satisfies the linear constraint $y(E) = f(E)$ the dimension of $B(f)$ is at most $n-1$), or $n+1$ extreme points for $\tilde{P}(\tilde{f})$.

We concentrate on the $B(f)$ case here, as the $\tilde{P}(\tilde{f})$ case is similar. Therefore, to prove that $y \in B(f)$ it suffices to keep linear orders $\prec_i$ with associated extreme points $v^{\prec_i}$ and multipliers $\lambda_i \geq 0$ for i in index set $\mathcal{I}$, such that

$$\sum_{i \in \mathcal{I}} \lambda_i = 1, \quad y = \sum_{i \in \mathcal{I}} \lambda_i v^{\prec_i}. \tag{11}$$

To reduce clutter, we'll usually write $v^{\prec_i}$ as v^i, and we'll abuse notation by considering $i \in \mathcal{I}$ to be both $\prec_i$ and v^i. Since the Greedy Algorithm is a strongly polynomial algorithm for checking if $\prec_i$ truly does generate v^i, as long as $|\mathcal{I}|$ is polynomial we can use this to prove that y really does belong to $B(f)$ in strongly polynomial time. In the language of computational complexity, the linear orders $\prec_i$, $i \in \mathcal{I}$, are a *succinct certificate* that $y \in B(f)$.

Most of our algorithms use such a representation of the current point, and they dynamically change the set $\mathcal{I}$ by adding one or more new vertices v^j to $\mathcal{I}$ to allow a move away from the current point. To keep $|\mathcal{I}|$ small, such algorithms need to reduce the set of v^i to the Carathéodory minimum from time to time. This is a simple matter, handled by subroutine ReduceV. Its input is a representation of y in terms of $\mathcal{I}$ and λ as in (11) with $|\mathcal{I}| \leq 4n$, and the output is a new representation with $|\mathcal{I}| \leq n$. It could happen that a v^j we want to add to $\mathcal{I}$ already belongs to $\mathcal{I}$. We could search $\mathcal{I}$ to detect such duplicates, but this would add an overhead of $O(n^2)$ per addition. The simpler, more efficient method that we use is to allow $\mathcal{I}$ to contain duplicates, which get removed by a later ReduceV.

Let V be the matrix whose columns are the current (too large set of) v^i's, and V' be V with a row of ones added at the top. When we reduce $\mathcal{I}$ (remove columns from V') we must compute and maintain the invariant that there are non-negative multipliers λ_i satisfying (11), which is equivalent to $V'\lambda = \binom{1}{y}$. By standard linear algebra manipulations (essentially converting a feasible solution to a basic feasible solution), ReduceV finds a linearly-independent set of columns of V' with corresponding new λ. Since V' has at most $4n$ columns, the initial reduction of V' to $(I \quad N)$ takes $O(n^3)$ time. Each of the at most $4n$ columns subsequently deleted requires reducing at most one column to a unit vector, which can be done in $O(n^2)$ time. Thus ReduceV takes $O(n^3)$ total time.

Carathéodory Subroutine REDUCEV

Let V be the matrix whose columns are the v^i.
Let V' be $\begin{pmatrix} \mathbb{1} \\ V \end{pmatrix}$, i.e., V with a row of ones added.
While $|\mathcal{I}| > n$ do
  Use linear algebra to reduce V' to $(I \quad N)$, where I is an identity matrix.
  [$(I \quad N)$ might have fewer rows than V'; if $|\mathcal{I}| > n$, N has at least one column]
  Let $\mathcal{B}$ index the columns of I.
  Select a column j of N, call it N^j.
  Compute the vector μ with entries N^j in positions $\mathcal{B}$, and $-\chi_j$ otherwise.
  [thus $(I \quad N)\mu = 0 \Rightarrow V'\mu = 0 \Rightarrow V'(\lambda + \alpha\mu) = \binom{1}{y}$ for any α]
  Compute $\alpha = \min\{-\lambda_i/\mu_i \mid \mu_i < 0\}$, with the min achieved at indices in $\mathcal{M}$.
  Set $\lambda \leftarrow \lambda + \alpha\mu$. [this makes $\lambda_k = 0$ for $k \in \mathcal{M}$ and keeps $\lambda \geq 0$]
  Set $\mathcal{I} \leftarrow \mathcal{I} - \mathcal{M}$, and delete columns in $\mathcal{M}$ from V'.

2.6 From LPs to Network Flow-Like Problems

Our descriptions of the network-like formulations of SFM are somewhat vague, since each algorithm makes different choices about the details of implementation. The two approaches outlined in Section 2.4 lead to two slightly different networks.

2.6.1 The Base Polyhedron Approach

This approach suggests the following generic algorithm: Pick an arbitrary linear order $\prec$ and use it to generate extreme point $y = v^{\prec} \in B(f)$. Define $S^-(y) = \{e \in E \mid y_e < 0\}$, $S^+(y) = \{e \in E \mid y_e > 0\}$, and $S^0(y) = \{e \in E \mid y_e = 0\}$. Then if we could find $k, l \in E$ with $k \in S^-(y)$, $l \in S^+(y)$, and $c(k, l; y) > 0$, then we could update $y \leftarrow y + (\chi_k - \chi_l)c(k, l; y)$ and increase $y^-(E)$ by $c(k, l; y)$. The difficulty with this is that it would require knowing the exchange capacities $c(k, l; y)$, and this is already as hard as SFM, as discussed in Section 2.2.

However, we can at least use Lemma 2.5, which says that $c(k, l; y)$ is easily computable when (l, k) is consecutive in $\prec$. Suppose that $i \in \mathcal{I}$ and (l, k) is consecutive in $\prec_i$, and let $\prec_i'$ be $\prec_i$ with k and l reversed (so that (k, l) is consecutive in $\prec_i'$) with corresponding vertex $v^{i'} = v^{\prec_i'}$. Then Lemma 2.5 says that

$$v^{i'} = v^i + c(k, l; v^i)(\chi_k - \chi_l), \tag{12}$$

and that for all $\theta \in [0, c(k, l; v^i)]$, $v^i + \theta(\chi_k - \chi_l) \in B(f)$. Using (12) we can re-write this as $(1 - \frac{\theta}{c(k,l;v^i)})v^i + (\frac{\theta}{c(k,l;v^i)})v^{i'} \in B(f)$. Then

$$y' = \sum_{j \in \mathcal{I},\ j \neq i} \lambda_j v^j + \lambda_i \left(1 - \frac{\theta}{c(k, l; v^i)}\right) v^i + \lambda_i \left(\frac{\theta}{c(k, l; v^i)}\right) v^{i'} \tag{13}$$

is a convex combination representation of the new point y' over the expanded index set $\mathcal{I}' = \mathcal{I} \cup \{i'\}$. Now (13) implies that $y' = y + \frac{\lambda_i \theta}{c(k,l;v^i)}(v^{i'} - v^i)$, and so from (12) $y' = y + \lambda_i\theta(\chi_k - \chi_l)$. As long as $\lambda_i\theta \leq \min(|y_k|, y_l)$, we would have that $(y')^-(E) = y^-(E) - \lambda_i\theta$, and so our objective value is improved. This is the mechanism by which new vertices are added to $\mathcal{I}$.

More generally (see Figure 1), suppose that $\prec_i$ contains the block $\cdots k_4 k_3 k_2 k_1 \cdots$. Define v^{21} to be generated by $\prec_i$ with k_1 and k_2 exchanged, v^{32} to be generated by $\prec_i$ with k_2 and k_3 exchanged, and v^{43} to be generated by $\prec_i$ with k_3 and k_4 exchanged. Further suppose that $k_1 \in S^-(y)$, $k_2 \in S^0(y)$, and $c(k_1, k_2; v^{21}) > 0$; $k_3 \in S^0(y)$ and $c(k_2, k_3; v^{32}) > 0$; and $k_4 \in S^+(y)$ and $c(k_3, k_4; v^{43}) > 0$. Choose

$$\alpha = \min\left(\frac{\lambda_i}{1/c(1,2;v^i) + 1/c(2,3;v^i) + 1/c(3,4;v^i)}, |y_{k_1}|, y_{k_4} \right) > 0,$$

and

$$y' = y + \frac{\alpha}{c(1,2;v^i)}(v^{21} - v^i) + \frac{\alpha}{c(2,3;v^i)}(v^{32} - v^i) + \frac{\alpha}{c(3,4;v^i)}(v^{43} - v^i).$$

What's going on here is that we are adding v^{21}, v^{32}, and v^{43} to $\mathcal{I}$, reducing the coefficient of v^i from λ_i to $\lambda_i - \alpha[1/c(1,2;v^i) + 1/c(2,3;v^i) + 1/c(3,4;v^i)]$, and putting the new coefficient of v^{21} to be $\frac{\alpha}{c(1,2;v^i)}$, the new coefficient of v^{32} to be $\frac{\alpha}{c(2,3;v^i)}$, and the new coefficient of v^{43} to be $\frac{\alpha}{c(3,4;v^i)}$. Then, despite the fact that none of these three changes by itself improves $y^-(E)$, doing all three changes simultaneously has the net effect of $y' = y + \alpha(\chi_{k_1} - \chi_{k_4})$, which improves $y^-(E)$ by α, at the expense of adding three new vertices to $\mathcal{I}$.

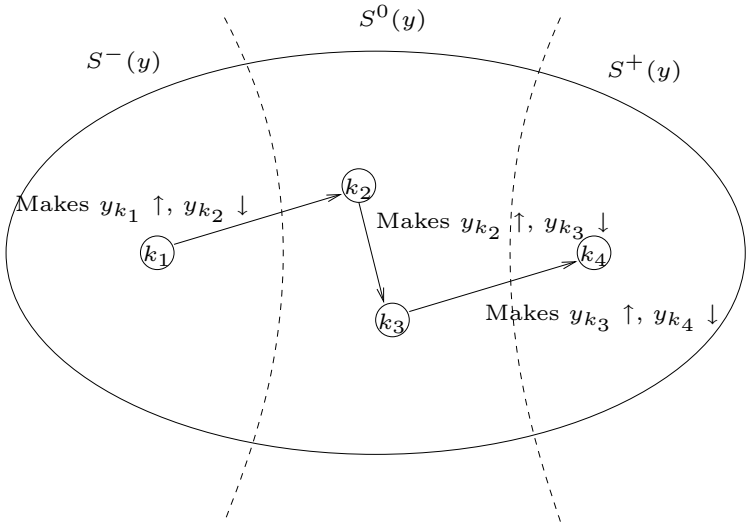

Figure 1: Example showing why we need to consider paths of arcs in the network. None of these three changes improves $y^-(E)$ by itself, but their union does improve $y^-(E)$.

This suggests that we define a network with node set E, and arc $k \to l$ with capacity $c(k, l; v^i)$ whenever there is an $i \in \mathcal{I}$ with (l, k) consecutive in $\prec_i$. (This definition has our arcs in the reverse direction of most of the literature. We choose this convention to get the natural sense of augmenting from $S^-(y)$ towards $S^+(y)$, but somewhat non-intuitively means that arc $k \to l$ corresponds to $l \prec k$.) Then we look for paths from $S^-(y)$ to $S^+(y)$. If we find a path, then we "augment" by making changes as above, and call ReduceV to keep $|\mathcal{I}|$ small.

Schrijver's Algorithm and the Hybrid Algorithm both consider changes to the v^i more general than swaps of consecutive elements. Hence both use this more liberal definition of arcs: $k \to l$

exists whenever there is an $i \in \mathcal{I}$ with $l \prec_i k$. The following lemma is a direct analogue of the classic result that a flow in a Max Flow / Min Cut network is maximum if it has no augmenting paths.

Lemma 2.10 *For either definition of arcs, if no augmenting path exists, then the node subset S defined as $\{e \in E \mid$ there is a partial augmenting path from some node $e' \in S^-(y)$ to node $e\}$ solves SFM.*

Proof: Since no augmenting path exists, $S^-(y) \subseteq S \subseteq S^-(y) \cup S^0(y)$, implying that $y^-(E) = y(S)$. Since no arcs exit S we must have that for each $i \in \mathcal{I}$, there is some $e_i \in E$ such that $S = e_i^{\prec_i}$, hence by (5) $f(S) = v^i(S)$. But then S satisfies all three complementary slackness tightness conditions of (10), and so S is an optimal solution to SFM. ■

Here is another way to think about this. For some v^i in $\mathcal{I}$, consider the pattern of signs of the y_e when ordered by $\prec_i$. If $\oplus$ is a non-negative entry and $\ominus$ is a non-positive entry, we are trying to find an $S \subseteq E$ such that this sign pattern looks like this for every $i \in \mathcal{I}$:

$$(\overbrace{\ominus \ominus \cdots \ominus \ominus}^{S} \oplus \oplus \cdots \oplus \oplus).$$

This picture also illustrates the complementary slackness of (10). If we find such an S, then (5) says that S is tight for v^i, and then by (11) S is tight also for y. Then we must have that $y^-(E) = y(S) = f(S)$, and by (9)–(10) y and S must be optimal. Thus to move closer to optimality we try to move positive components of the v^i to the right, and negative components to the left.

2.6.2 The Polymatroid Approach

This approach suggests a similar generic algorithm: Start with $z = 0$ and try to increase $\mathbb{1}^T z$ while maintaining $z \leq \gamma$ and $z \in P(f)$. In theory, we could do this via the sort of modified Greedy Algorithm used in the proof of Theorem 2.9. The difficulty with this is that it would require knowing the exchange capacities $c(k; z)$, and this is already as hard as SFM, as discussed in Section 2.2.

We define a similar network. This time we add a source s and a sink t to E to get the node set. The arcs not incident to s and t are as in Section 2.6.1. We make arc $s \to e$ if $z_i < \gamma_e$ for some $i \in \mathcal{I}$. We make arc $e \to t$ if there is some $i \in \mathcal{I}$ such that e belongs to no tight set of v^i. Now an s–t augmenting path in this network allows us to bring z closer to γ, and $z(E)$ closer to $\tilde{f}(E)$. When there is no augmenting path, define S as the elements of E reachable from s by augmenting paths. As above, S is tight. Since $e \notin S$ is not reachable, it must have $z_e = \gamma_e$, so we have $z(E) = z(S) + z(E - S) = \tilde{f}(S) + \gamma(S)$, proving that S is optimal for SFM.

2.7 Strategies for Getting Polynomial Bounds

In both cases we end up with generic algorithms that resemble Max Flow / Min Cut: We have a network, we look for augmenting paths, we have a theorem that says that an absence of augmenting paths implies optimality, we have general capacities on the arcs, and we have 0–1 objective coefficients. In keeping with this analogy, we consider the flow problems to be the *primal* problems, and the "min cut" problems to be the *dual* problems, despite the fact that our

original problem of SFM then turns out to be a dual problem. However, note that the analogy with Max Flow / Min Cut is far from exact: whatever "flow" changes we make have to be consistent with, e.g., the representation (11), and we have the option of adding new vertices to (11) (and so new arcs to the "network") as long as we can compute them.

This analogy helps us think about ways in which we might make these generic algorithms have polynomial bounds. There are two broad strategies that have been successful for Max Flow / Min Cut:

1. Give an argument that some *potential function* bounded by a polynomial function of n is monotone non-decreasing, and strictly increases in a polynomial number of iterations. Often the potential function is based on some *distance labels*; the canonical instance of this for Max Flow is Edmonds and Karp's Shortest Augmenting Path [18] bound. They show that the length of the shortest augmenting path from s to each node is monotone non-decreasing, and that each new time an arc is the bottleneck arc on an augmenting path, this shortest distance must strictly increase by 2 at one of its nodes. With $m = |A|$, this leads to their $O(nm^2)$ bound on Max Flow. A more sophisticated version of this argument is used in Goldberg and Tarjan's Push-Relabel Max Flow Algorithm [37] to get an $O(mn \log(n^2/m))$ bound.

   This strategy is attractive since it typically yields a strongly polynomial bound without extra work, and it implies that we don't have to worry about how large the change in objective value is at each iteration. It also doesn't require pre-computing the bound M on the size of f. For Max Flow, these algorithms also seem to work well in practice (see, e.g., Cherkassky and Goldberg [9]).

2. Give a *sufficient decrease* argument that when one iteration changes y to y', the difference in objective value between y and y' is a sufficiently large fraction of the gap between the objective value of y and the optimal objective value that we can get a polynomial bound. The canonical instance of this for Max Flow also comes from Edmonds and Karp [18], the Maximum Capacity Path bound. Here we augment on an augmenting path with maximum residual capacity at each iteration. This can be shown to reduce the gap between the current solution and an optimal solution by a factor of $(1 - 1/m)$, leading to an overall $O(m(m + n \log n) \log(nU))$ bound, where U is the maximum capacity. Capacity scaling algorithms (scaling algorithms were first suggested also by Edmonds and Karp [18], and capacity scaling for Max Flow was suggested by Gabow [33]) can also be seen as a way of achieving sufficient decrease.

   This strategy leads to quite simple proofs of polynomiality. However, it does require starting off with the assumption that all data are integral (so that an optimality gap of less than one implies optimality), and pre-computing the bound M on the size of f. Therefore it leads to algorithms which are naturally only weakly polynomial, not strongly polynomial (in fact, Queyranne [73] showed that Maximum Capacity Path for Max Flow is *not* strongly polynomial). However, it is usually possible to modify these algorithms so they become strongly polynomial, and so can deal with non-integral data. It is generally believed that these algorithms do not perform well in practice, partly because their average-case behavior tends to be close to their worst-case behavior, unlike the potential function-based algorithms.

There are two aspects of these network-based SFM algorithms that are significantly more

difficult than Max Flow. In Max Flow, if we augment flow on s–t path P, then this does not change the residual capacity of any arc not on P. In SFM, augmenting from y to y' along a path P not containing $k \to l$ can cause $c(k, l; y')$ to be positive despite $c(k, l; y) = 0$. A technique that has been developed to handle this is called *lexicographic* augmenting paths (also called *consistent breadth-first search* in [12]), which was discovered independently by Lawler and Martel [57] and Schönsleben [75]. It is an extension of the shortest augmenting path idea. We choose some fixed linear order on the nodes, and we select augmenting paths which are lexicographically minimum, i.e., among shortest paths, choose those whose first node is as small as possible, and among these choose those whose second node is as small as possible, etc. Then, despite the exchange arcs changing dynamically, one can mimic a Max Flow-type potential function-based convergence proof.

Second, the coefficients λ_i in the representation (11) can be arbitrarily small even with integral data. Consider this example due to Iwata: Let L be a large integer. Then f defined by $f(S) = 1$ if $1 \in S$, $n \notin S$, $f(S) = L$ if $n \in S$, $1 \notin S$, and $f(S) = 0$ otherwise is a submodular function. The base polyhedron $B(f)$ is the line segment between the vertices $v^1 = (1, 0, \ldots, 0, -1)$ and $v^2 = (-L, 0, \ldots, 0, L)$. Then the zero vector, i.e., the unique primal optimal solution, has a unique representation as in (11) with $\lambda_1 = 1 - 1/(L+1)$ and $\lambda_2 = 1/(L+1)$. This phenomenon means that it is difficult to carry through a sufficient decrease argument, since we may be forced to take very small steps to keep the λ_i non-negative.

Another choice is whether an algorithm augments along paths as in the classic Edmonds and Karp [18] or Dinic [15] Max Flow Algorithms, or augments arc by arc, as in the Goldberg and Tarjan [37] Push-Relabel Max Flow Algorithm, or is not based on augmentation, as in the Orlin-type algorithms. Augmenting along a path here is tricky since several arcs of the path might correspond to the same v^i, so that tracking the changes to $\mathcal{I}$ is difficult. In terms of worst-case running time, the Dinic [15] layered network approach speeds up the standard Edmonds and Karp shortest augmenting path approach and has been extended to situations akin to SFM by Tardos, Tovey and Trick [81], but the Goldberg and Tarjan approach is even faster. In terms of running time in practice, the evidence shows (see, e.g., Cherkassky and Goldberg [9]) that for Max Flow, the arc by arc approach seems to work better in practice than the path approach. Schrijver's Algorithm uses the arc by arc method. The IFF Algorithm and its variants blend the two methods: A relaxed current point is augmented arc by arc, but the flow mediating the difference between the relaxed point and the feasible point is augmented on paths. Orlin-type algorithms only implicitly use networks, although one could interpret the algorithms as looking at nodes (elements) with $y_e > 0$ and trying to either drive them to be non-positive, or prove that they can never belong to a min cut (SFM solution).

The algorithms have the generic outline of keeping a current point y and moving in some direction to improve $y^-(E)$. This movement is achieved by modifying (11) by adding a new order (or multiple orders) to $\mathcal{I}$ and updating the λ_i. Here is a general idea for developing good movement directions: Suppose that v^i and v^h are two vertices of $B(f)$, where typically $i \in \mathcal{I}$ and $h \notin \mathcal{I}$. Then $x' = x + \alpha(v^h - v^i)$ will also be in $B(f)$ as long as $\alpha \leq \lambda_i$. This move corresponds to changing the convex combination (11) via reducing λ_i by α, and increasing λ_h by α (by adding h to $\mathcal{I}$ if it wasn't there to start with).

A particularly simple version of this arises like this. Suppose that $v^{\prec}$ comes from a linear order with (l, k) consecutive in $\prec$, and $\prec'$ is the same linear order with k and l swapped. Then Lemma 2.5 says that we can compute the exchange capacity $c(k, l; v^{\prec})$ easily and (12) says that $v^{\prec'} - v^{\prec} = c(k, l; v^{\prec})(\chi_k - \chi_l)$. These directions $\chi_k - \chi_l$ for $k, l \in E$ are the *edge directions* of

$B(f)$ (see [8]). Alternatively, we could choose directions from more general *vertex differences*, i.e., $v^h - v^i$. Finally, we could choose directions that are positive linear combinations of vertex differences, or *multiple vertex differences*.

Choosing edge directions has the virtue of having an easy-to-compute exchange capacity, but the vice of being a slow way to make big changes in the linear orders. Alternatively, we could modify larger blocks of elements. This has the vice that exchange capacities are hard to compute (but at least we can use (6) to quickly compute new vertices), but the virtue that big changes in the linear orders are faster. Cunningham's Algorithm uses edge directions and consecutive pairs. Schrijver's Algorithm uses edge directions, but single element blocks; modifying by blocks means that it is complicated to synthesize a edge direction, but it does give a good enough bound on $c(k, l; y)$. Basic IFF uses edge directions and consecutive pairs, but the Hybrid Algorithm changes to vertex differences and general blocks; blocks represent vertex differences easily, and staying within $B(f)$ is easy since we are effectively just replacing v^i by v^h in (11). Orlin's Algorithm uses multiple vertex differences, where the new vertices come from single element block changes, and the Iwata-Orlin algorithm uses single vertex differences coming from general block exchanges..

Cunningham's Algorithm for General SFM [13] uses the polymatroid approach, augmenting on paths, edge directions, modifying consecutive pairs, and the sufficient decrease strategy. However, he is able to prove only a pseudo-polynomial bound. Schrijver's Algorithm [76] and Schrijver-PR use the base polyhedron approach, augmenting arc by arc, edge directions, modifying blocks, and a distance-based strategy, and so they easily get a strongly polynomial bound. Iwata, Fleischer, and Fujishige's Algorithm (IFF) [50] uses the base polyhedron approach, augmenting both on paths and arc by arc, edge directions, modifying consecutive pairs, and the sufficient decrease strategy. IFF are able to modify their algorithm to make it strongly polynomial. Iwata's Algorithm [46] is a fully combinatorial extension of IFF. Iwata's Hybrid Algorithm [48] largely follows IFF, but adds some distance-based ideas that lead to vertex differences and modifying blocks instead of edge directions and consecutive pairs. Orlin's Algorithm [71] uses the base polyhedron approach, augmenting on multiple paths, multiple vertex differences, modifying blocks, and a potential function strategy; Iwata-Orlin is similar, except for using single vertex difference directions. Orlin's Algorithm is naturally strongly polynomial, whereas Iwata-Orlin needs some modifications to become strongly polynomial (though it still avoids scaling).

There is some basis to believe that the potential function-based strategy is more "natural" than scaling for Max Flow-like problems such as SFM. Despite this, the running time for the IFF Algorithm is in most cases faster than the running time for Schrijver's Algorithm. However, Iwata's Hybrid Algorithm, which adds some distance-based ideas to IFF, is even faster than IFF, and Orlin's Algorithm (which is completely potential function-based) is faster yet; see Section 4.

3 The SFM Algorithms

We describe Cunningham's Algorithms in Section 3.1, a version of Schrijver's Algorithm in Section 3.2, various IFF-type algorithms in Section 3.3, and Orlin-type algorithms in Section 3.4.

3.1 Cunningham's SFM Algorithms

We skip most of the detail of these algorithms, as more recent algorithms appear to be better in both theory and practice.

In a series of three papers in the mid-1980s [8, 12, 13] (one with Bixby and Topkis), Cunningham developed the ideas of the polymatroid approach and gave three SFM algorithms. The first [12] is for Example 1.11, for separating point $\bar{x}$ from the matroid polytope defined by rank function r, which is the special case of SFM where $f(S) = r(S) - \bar{x}(S)$. Here Cunningham takes advantage of the special structure of f and carefully analyzes how augmentations happen in a lexicographic shortest augmenting path framework. This allows him to prove that the algorithm needs $O(n^3)$ total augmenting paths; each path adds $O(n)$ new v^i (which are the incidence vectors of independent sets in this case) to $\mathcal{I}$, so when it doesn't call ReduceV the algorithm must manage $O(n^4)$ vertices in $\mathcal{I}$. To construct the graph of augmenting paths, for each of the $O(n^4)$ $i \in \mathcal{I}$ and each of the $O(n^2)$ pairs $k, l \in E$, we must consider whether i implies an arc $k \to l$, for a total of $O(n^6\text{EO})$ time per augmenting path. This yields a total time of $O(n^9\text{EO})$, and a fully combinatorial algorithm for this case (without calling ReduceV). If we do use ReduceV, then the size of $\mathcal{I}$ stays $O(n)$, so the time per augmentation is now only $O(n^3\text{EO})$, for a total of $O(n^6\text{EO})$ (although the resulting algorithm is no longer fully combinatorial, but only strongly polynomial).

In the second paper, Bixby, Cunningham, and Topkis [8] extend some of these ideas to the general case. It uses the polymatroid approach and augmenting on paths. Because of degeneracy, there might be several different linear orders that generate the same vertex v of $\tilde{P}(\tilde{f})$. A given pair (l, k) might be consecutive in some of these orders but not others. They show that, for each vertex v, there is a partial order $\prec_v$ (note that $\prec_v$ is in general *not* a linear order) such that $c(k, l; v) > 0$ iff k *covers* l in $\prec_v$, i.e., if $l \prec_v k$ but there is no $j \in E$ with $l \prec_v j \prec_v k$ (if $\prec_v$ is linear, then k covers l in $\prec_v$ iff (l, k) is consecutive). Furthermore, they gave an $O(n^2\text{EO})$ algorithm for computing $\prec_v$. Finally, they note that if k covers l in $\prec_v$, then $c(k, l; v)$ (and also $c(k; v)$) can be computed in $O(\text{EO})$ time, similar to Lemma 2.5. They define the arcs to include $k \to l$ if there is some $i \in \mathcal{I}$ such that k covers l in v^i, and thus they know that the capacity of every arc is positive. When this is put into the polymatroid approach using ReduceV, it is easy to argue that no set of vertices $\mathcal{I}$ can repeat, leading to a finite algorithm.

In the third paper, Cunningham [13] modified this second algorithm into what we call Cunningham's Algorithm for General SFM. It adds a weak version of the sufficient decrease strategy to the second algorithm. The fact that the λ_i can be arbitrarily small (discussed in Section 2.7) prevents Cunningham from using a stronger sufficient decrease argument. Suppose that we restrict our search for augmenting paths only to arcs $s \to e$ with $\gamma_e - z_e \geq 1/Mn(n+1)^2$ and arcs $k \to l$ with $\lambda_i c(k, l; z) \geq 1/M(n+1)^2$. If we find an augmenting path P of such arcs, then it can be seen that augmenting along P increases $\mathbb{1}^T z$ by at least $1/M(n+1)^2$. Then the key to Cunningham's argument is the following lemma:

Lemma 3.1 ([13, Theorem 3.1]) *If no such path exists, then there is some $S \subseteq E$ with $z(E) > f(S) + \gamma(E - S) - 1$, and because all data are integral, we conclude that S solves SFM.* ∎

Cunningham suggests some speedups, which are essentially variants of implicit capacity scaling (look for augmenting paths of capacity at least K until none are left, then set $K \leftarrow K/2$ until $K < 1/M(n+1)^2$) and maximum capacity augmenting path. These lead to the overall time bound of $O(Mn^6 \log(Mn) \cdot \text{EO})$, which is pseudo-polynomial.

3.2 Schrijver's SFM Algorithm

Schrijver's Algorithm [76] (see also [77, Chapter 45]) uses the base polyhedron approach, augmenting arc by arc, modifying single element blocks, and the distance-based strategy. The algorithm assumes that f is defined on 2^E, but it can be adapted to ring families using the method of Section 5.2. Schrijver's big innovation is to avoid being constrained to consecutive pairs, but to allow arcs $k \to l$ if $l \prec_i k$ for some $i \in \mathcal{I}$, even if l and k are not consecutive in $\prec_i$. This implies that Schrijver has a looser definition of arcs than some other algorithms. Of course, the problem that computing $c(k, l; v)$ is equivalent to SFM still remains; Schrijver's solution is to compute a lower bound on $c(k, l; v)$.

Let's focus on a particular arc $k \to l$, associated with $\prec_h$, which we'd like to include in an augmentation. For simplicity call $\prec_h$ just $\prec$ and v^h just v. Define $(l, k]_\prec = \{e \in E \mid l \prec e \preceq k\}$ (and similarly $[l, k]_\prec$ and $[l, k)_\prec$), so that $(l, k]_\prec = \emptyset$ if $k \preceq l$. Then Lemma 2.5 says that $c(k, l; v)$ is easy to compute if $|(l, k]_\prec| = 1$. In order to get combinatorial progress, we would like to represent the direction we want to move in, $v + \alpha(\chi_k - \chi_l)$, as a combination of new vertices w^j with linear orders $\prec'_j$ with $(l, k]_{\prec'_j} \subset (l, k]_\prec$ for each j. That is, we would like to drive arcs which are not consecutive more and more towards being consecutive.

Schrijver gives a subroutine for achieving this, which we call ExchBd$(k, l; \prec)$ (and describe in Section 3.2.1). It chooses the following linear orders to generate its w^j: For each j with $l \prec j$ define $\prec^{l,j}$ as the linear order with j moved just before l. That is, if $\prec$'s order is

$$\cdots s_{a-1} s_a l t_1 t_2 \ldots t_b j u_1 u_2 \cdots,$$

then $\prec^{l,j}$'s order is

$$\cdots s_{a-1} s_a j l t_1 t_2 \ldots t_b u_1 u_2 \cdots.$$

Note that if $l \prec j \preceq k$, then $(l, k]_{\prec^{l,j}} \subset (l, k]_\prec$, as desired. Also, this is a type of block exchange, with $Q = \{j\}$ and $R = [l, k)_\prec$.

ExchBd$(k, l; \prec)$ has the following properties. The input is linear order $\prec$ and $k, l \in E$ with $l \prec k$. The output is a step length $\alpha \geq 0$, and the collection of vertices $w^j = v^{\prec^{l,j}}$ with coefficients $\mu_j \geq 0$ for $j \in \mathcal{J} = (l, k]_\prec$. This implies that $|\mathcal{J}| \leq |(l, k]_\prec| \leq n$. The μ_j satisfy $\sum_{j \in \mathcal{J}} \mu_j = 1$, and

$$v^\prec + \alpha(\chi_k - \chi_l) = \sum_{j \in \mathcal{J}} \mu_j w^j. \tag{14}$$

That is, $v^\prec + \alpha(\chi_k - \chi_l)$ is a convex combination of the w^j. Also, this implies that $v^\prec + \alpha(\chi_k - \chi_l) \in B(f)$, and hence that $\alpha \leq c(k, l; v)$. We show below that ExchBd takes $O(n^2 \text{EO})$ time.

We now describe Schrijver's Algorithm, assuming ExchBd as a given. We actually present a Push-Relabel variant due to Fleischer and Iwata [23] that we call Schrijver-PR, because it is simpler to describe, and seems to run faster in practice than Schrijver's original algorithm (see Section 4). Schrijver-PR originally also had a faster time bound than Schrijver, but Vygen [84] showed that in fact the time bound for Schrijver's Algorithm is the same as for Schrijver-PR. Roughly speaking, Schrijver's original algorithm is similar to Dinic's Max Flow Algorithm [15], in that it uses exact distance labels to define a layered network, whereas Schrijver-PR is similar to Goldberg and Tarjan's Push-Relabel Max Flow Algorithm [37], in that it uses approximate distance labels to achieve the same thing.

Similar to Goldberg and Tarjan [37], we put non-negative, integer *distance labels* d on the nodes. We call labels d *valid* if

(Sch i) $d_e = 0$ for all $e \in S^-(y)$, and

(Sch ii) we have $d_l \leq d_k + 1$ for every arc $k \to l$ (i.e., whenever $l \prec_i k$ for some $i \in \mathcal{I}$).

This implies that d_e is a lower bound on the number of arcs in a shortest path from $S^-(y)$ to e, so that $d_e < n$; we use $d_e = n$ to signify that no path from $S^-(y)$ to e exists. We choose $d_e = 0$ for all $e \in E$ as an initial valid labeling.

The algorithm defines the set of *active* nodes as $\mathcal{A} = \{e \in S^+(y) \mid d_e < n\}$, i.e., the set of positive nodes which still have a hope of being decreased. The basic idea of the algorithm is to choose a node $l \in \mathcal{A}$ with maximum d_l, and then look for some node k such that $d_k = d_l - 1$ and such that arc $k \to l$ exists due to $l \prec_h k$ for some $h \in \mathcal{I}$. If we have selected l but every arc $k \to l$ has $d_k \geq d_l$ (i.e., no arc $k \to l$ satisfies the distance criterion that $d_k = d_l - 1$), then we apply RELABEL(l).

RELABEL(l) **Subroutine for the Schrijver-PR Algorithm**

Set $d_l \leftarrow d_l + 1$.
If $d_l = n$, then $\mathcal{A} \leftarrow \mathcal{A} - l$.

Thus we can assume that we find such a k, and we then call PUSH, which applies EXCHBD repeatedly to $k \to l$. Each EXCHBD decreases y_l, and makes the $(l,k]_{\prec_i}$ smaller. We apply EXCHBD until either (1) y_l drops to 0, called *non-saturating*, or (2) arc $k \to l$ disappears because $k \prec_i l$ for all $i \in \mathcal{I}$ (i.e., $|(l,k]_{\prec_i}| = 0$ for all $i \in \mathcal{I}$), called *saturating*. To keep combinatorial monotonicity, we always choose an associated $\prec_h$ achieving $\max_{i \in \mathcal{I}} |(l,k]_{\prec_i}|$. To be lexicographic, we scan through the possible nodes k in a fixed linear order.

When we work on arc $k \to l$, we are increasing y_k and decreasing y_l. We enforce that y_l stays non-negative (since $d_l > 0$, if we allowed y_l to become negative, this would violate that $d_e = 0$ for $e \in S^-(y)$), but if y_k is negative, we allow it to become positive. To see the algebraic details of this, note that (11) and (14) imply that

$$y + \alpha\lambda_h(\chi_k - \chi_l) = \sum_{i \neq h} \lambda_i v^i + \lambda_h \sum_j \mu_j w^j. \tag{15}$$

If $\alpha\lambda_h > y_l$, then this would make $y_l < 0$, which we don't allow. So we set $\beta = \min(y_l, \alpha\lambda_h)$, and we want to take the step $y + \beta(\chi_k - \chi_l)$. Note that $\beta = y_l$ means that the new $y_l = 0$, leading to a non-saturating PUSH; and $\beta = \alpha\lambda_h$ means that h leaves $\mathcal{I}$, so there is one less index in $\mathcal{I}$ with a maximum value of $|(l,k]_{\prec_i}|$, so we are closer to being saturating. To get this effect we add $(1 - \beta/(\alpha\lambda_h))$ times (11) to $\beta/(\alpha\lambda_h)$ times (15) to get:

$$y + \beta(\chi_k - \chi_l) \;=\; \sum_{i \neq h} \lambda_i v^i + (\lambda_h - \beta/\alpha) v^h + \sum_j (\beta\mu_j/\alpha) w^j.$$

We put these pieces together into the subroutine PUSH(k,l).

Now we are ready to describe the whole algorithm. For simplicity, assume that $E = \{1, 2, \ldots, n\}$. To get our running time bound, we need to ensure that for each fixed node l, we do at most n saturating PUSHes before RELABELing l. To accomplish this, we do PUSHes

PUSH(k,l) Subroutine for the Schrijver-PR Algorithm

While $y_l > 0$ and arc $k \to l$ exists,
  Select h that solves $\max_{i \in \mathcal{I}} |(l,k]_{\prec_i}|$.
  Call EXCHBD$(k,l;v^h)$ to get α, $\mathcal{J}$, μ_j, w^j.
  Set $\beta = \min(y_l, \alpha\lambda_h)$.
  Update $y \leftarrow y + \beta(\chi_k - \chi_l)$, $\mathcal{I} \leftarrow \mathcal{I} \cup \mathcal{J}$, and $\lambda_h \leftarrow \lambda_h - \beta/\alpha$.
  For $j \in \mathcal{J}$, set $\lambda_j \leftarrow \beta\mu_j/\alpha$.
  Call REDUCEV.

to l from nodes k for each k in order from 1 to n; to ensure that we re-start where we left off if PUSHes to l are interrupted by a non-saturating PUSH, we keep a pointer p_l for each node l that keeps track of the next k where we want to do a PUSH(k,l).

The Schrijver-PR Algorithm for SFM

Initialize by choosing $\prec_1$ to be any linear order, $y = v^1$, and $\mathcal{I} = \{1\}$.
Set $d = 0$ and $p = \mathbb{1}$.
Compute $S^-(y)$ and $S^+(y)$ and set $\mathcal{A} = S^+(y)$.
While $\mathcal{A} \neq \emptyset$ and $S^-(y) \neq \emptyset$,
  Find l solving $\max_{e \in \mathcal{A}} d_e$. [try to push to max distance node l]
  While $p_l \leq n$ do [scan through possible nodes that could push to l]
    If $d_{p_l} = d_l - 1$ then
      PUSH(p_l, l).
      If $y_l = 0$ set $\mathcal{A} \leftarrow \mathcal{A} - l$, and break out of the "While p_l" loop.
    Set $p_l \leftarrow p_l + 1$.
  If $p_l > n$, set $p_l = 1$ and RELABEL(l).
Compute S as the set of nodes reachable from $S^-(y)$, and return S.

We now prove that this works, and give its running time. We give one big proof, but we pick out the key claims along the way in boldface.

Theorem 3.2 *Schrijver-PR correctly solves SFM, and runs in $O(n^7\mathrm{EO} + n^8)$ time.*

Proof:

Distance labels d stay valid: We use induction on the iterations of the algorithm; d starts out being valid. Only PUSH and RELABEL could make d invalid.

PUSH **preserves validity of d:** Suppose that a call to EXCHBD$(k,l;v^h)$ in PUSH(k,l) introduces a new arc $u \to t$. Since $u \to t$ didn't exist before we must have had $u \prec_h t$, and since it does exist now we must have that $t \prec_h^{l,j} u$ for some $j \in (l,k]_{\prec_h}$. The only way for this to happen is if $j = t$ and we had $l \preceq_h u \prec_h t \preceq_h k$ and now have $t \prec_h^{l,t} l \preceq_h^{l,t} u \prec_h^{l,t} k$. Doing PUSH$(k,l)$ means that $d_k + 1 = d_l$. Since d was valid before the PUSH(k,l), we have $d_t \leq d_k + 1 = d_l \leq d_u + 1$, so d is still valid.

RELABEL **preserves validity of d:** We must show that when the algorithm calls RELABEL(t), every arc $u \to t$ has $d_u \geq d_t$. Since RELABEL(t) gets called when $p_t = n+1$, if we can show that

$u < p_t$ and $u \to t$ an arc imply that $d_u \geq d_t$, then we are done. We prove this by induction; it is trivially true when $p_t = 1$, and so also true just after RELABEL(t). A RELABEL(u) for $u \neq t$ also only improves things, so we need worry only about PUSHes. The algorithm increases p_l only when all $p_l \to l$ arcs have been made to disappear in PUSH, so the only problem that could arise is when a call to PUSH(k, l) (with $k = p_l$) creates a new arc $u \to t$. Suppose that the claim remains true until this point. The previous paragraph showed that in this case we had $l \preceq_h u \prec_h t \preceq_h k$, implying that $d_t \leq d_k + 1 = d_l \leq d_u + 1$. If $t = k$ then $d_k = d_t$ which gives that $d_u \geq d_t$; similarly if $u = l$. If $k < p_t$, then $t \prec_h k$ implies that $k \to t$ was an arc, and induction gives that $d_k \geq d_t$, implying that $d_u \geq d_t$. Otherwise, we have $p_l = k \geq p_t$, and we are assuming that $u < p_t$, so we get $u < p_l$. Then $l \prec_h u$ implies that $u \to l$ was an arc, so induction gives $d_u \geq d_l$, again implying that $d_u \geq d_t$.

The algorithm performs at most n^2 total RELABELs: Each RELABEL(l) increases d_l by 1, and $d_l \leq n$, so we call RELABEL(l) at most n times, so that the total is at most n^2.

The algorithm performs at most n^3 total saturating PUSHes: Because of the p_l, for each l we do at most n saturating PUSHes to l before doing a RELABEL(l). Since there are at most n RELABEL(l)s, there are at most n^2 saturating PUSHes to l, or n^3 total saturating PUSHes.

The algorithm performs at most n^3 total non-saturating PUSHes: We have a non-saturating PUSH(k, l) because y_l drops to 0. For the next non-saturating PUSH to happen at l, some other PUSH(l, u) must make $y_l > 0$ first. Since we always PUSH from the highest label, and since distance labels are monotone non-decreasing, we must have that d_u at the time of the PUSH(l, u) is at least one larger than d_l at the time of the non-saturating PUSH, so a RELABEL(u) must have happened in between. Since there are at most n^2 RELABELs, and each RELABEL can re-activate at most n such l's, there are at most n^3 non-saturating PUSHes.

Each call to PUSH(k, l) iterates at most n^2 times: An iteration of the while loop of PUSH(k, l) might cause $y_l = 0$ (a non-saturating PUSH), in which case we exit. Each iteration that does not cause $y_l = 0$ has $\beta = \alpha\lambda_h$, meaning that the new coefficient of v^h is 0, so that h drops out of $\mathcal{I}$. This either reduces $\max_{i \in \mathcal{I}} |(l, k]_{\prec_i}|$, or reduces the number of $i \in \mathcal{I}$ achieving this maximum (calling REDUCEV can only help here). Since $|(l, k]_{\prec_i}| < n$, this implies the claim.

The running time is $O(n^7\text{EO} + n^8)$: There are $O(n^3)$ calls to PUSH, each of which iterates at most n^2 times, and each iteration calls EXCHBD and REDUCEV once each, for a total of $O(n^5)$ calls to EXCHBD and REDUCEV. Each call to EXCHBD costs $O(n^2\text{EO})$ time, and each call to REDUCEV costs $O(n^3)$ time.

The algorithm terminates with an optimal solution: By Lemma 2.10. ∎

3.2.1 The Exchange Capacity Bound Subroutine

Recall that for each $j \in (l, k]_\prec$ we define $\prec^{l,j}$ as the linear order with j moved just before l. The task of EXCHBD($k, l; \prec$) is to find a step length $\alpha \geq 0$ and a representation of $v^\prec + \alpha(\chi_k - \chi_l)$ as a convex combination of vertices $v^{l,j}$ corresponding to the linear orders $\prec^{l,j}$.

Define $q = |(l, k]_\prec|$, enumerate $\prec$ as $\cdots l u_1 u_2 \ldots u_{q-1} k \cdots$, and define $u_q = k$. Define $V^{l,k}$ to be the matrix whose columns are the $v^{l,j}$ for $j \in (l, k]_\prec$, so that $V^{l,k}$ has n rows and q columns, and $V^\prec$ to be the matrix of the same dimension with every column equal to $v^\prec$. Since $\prec^{l,j}$ is the same order as $\prec$ except for $j \in [l, k]_\prec$, by (6) the only places where two columns of $V^{l,k}$ might differ is in the $q + 1$ rows $[l, k]_\prec$. Again using $\oplus$ for non-negative and $\ominus$ for non-positive,

then Lemma 2.6 proves that the sign pattern of this submatrix of $V^{l,k} - V^{\prec}$ is:

$$\begin{array}{r} \\ l \\ u_1 \\ u_2 \\ u_3 \\ \vdots \\ k = u_q \end{array} \begin{array}{c} \begin{array}{ccccc} v^{l,u_1} & v^{l,u_2} & v^{l,u_3} & \dots & v^{l,u_q} \end{array} \\ \begin{pmatrix} \ominus & \ominus & \ominus & \dots & \ominus \\ \oplus & \ominus & \ominus & \dots & \ominus \\ 0 & \oplus & \ominus & \dots & \ominus \\ 0 & 0 & \oplus & \dots & \ominus \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & \oplus \end{pmatrix} \end{array}. \tag{16}$$

Suppose that diagonal element $v_u^{l,u} - v_u^{\prec}$ of (16) equals zero. Then, since $v^{l,u}(E) = v^{\prec}(E) = f(E)$, from (16) we would get that $v^{l,u} = v^{\prec}$. In this case we choose $\alpha = 0$ and represent $v^{\prec} + \alpha(\chi_k - \chi_l) = v^{\prec}$ as $1 \cdot v^{l,u}$ for our convex combination.

Suppose instead that all diagonal elements of (16) are positive. Consider the following equation in unknowns η:

$$(V^{l,k} - V^{\prec})\eta = \chi_k - \chi_l. \tag{17}$$

Since (16) is triangular with positive diagonal, (17) has a unique solution with $\eta > 0$. We then set $\alpha = 1/\eta(E)$ and $\mu = \alpha\eta$, which then satisfy $(V^{l,k} - V^{\prec})\mu = \alpha(\chi_k - \chi_l)$. Since $\mu(E) = 1$, this is equivalent to (14), as desired. Suppose that $q = 1$, i.e., (l, k) is consecutive in $\prec$. Then $\prec^{l,k}$ is just $\prec$ with l and k interchanged. In this case Lemma 2.5 tells us that $v^{l,k} = v^{\prec} + c(k, l; v^{\prec})(\chi_k - \chi_l)$. This implies that when $c(k, l; v^{\prec}) > 0$, the solution of (17) in this case is $\eta = 1/c(k, l; v^{\prec})$, which means that we would compute $\alpha = c(k, l; v^{\prec})$. Thus in this case, as we would expect, ExchBd computes the exact exchange capacity.

Now we consider the running time of ExchBd. Computing the $v^{l,u}$ requires at most n calls to Greedy, which takes $O(n^2\text{EO})$ time (we can save time in practice by using (6), but this doesn't seem to improve the overall bound). Setting up and solving (17) takes only $O(n^2)$ time (because it is triangular), for a total of $O(n^2\text{EO})$ time.

3.3 SFM Algorithms in the Iwata, Fleischer, and Fujishige (IFF) Family

We describe the weakly polynomial version of the IFF algorithm [50] in Section 3.3.1, which assumes for simplicity that f is defined on 2^E. Section 3.3.2 shows how to adapt the algorithm for the case where f is defined on a ring family, and then uses this to get a strongly polynomial version. Iwata's fully combinatorial version [46] is developed in Section 3.3.3, and Iwata's faster Hybrid Algorithm [48] in Section 3.3.4.

3.3.1 The Basic Weakly Polynomial IFF Algorithm

Iwata, Fleischer, and Fujishige's Algorithm (IFF) [50] uses the base polyhedron approach, augmenting both on paths and arc by arc, modifying consecutive pairs, and the sufficient decrease strategy. IFF are able to modify their algorithm to make it strongly polynomial. The IFF Algorithm would like to use capacity scaling. A difficulty is that here the "capacities" are derived from the values of f, and scaling a submodular function typically destroys its submodularity. One way to deal with this is suggested by Iwata [45] in the context of algorithms for Submodular Flow: Add a sufficiently large perturbation to f and the scaled function is submodular. However

this proved to be slow, yielding a run time of $\widetilde{O}(n^7\text{EO})$ compared to $\widetilde{O}(n^4\text{EO})$ for the current fastest algorithm for Submodular Flow [24].

A different approach is suggested by Goldberg and Tarjan's Successive Approximation Algorithm for Min Cost Flow [38], using an idea first proposed by Bertsekas [6]: Instead of scaling the data, relax the data by a parameter δ and scale δ instead. As δ is scaled closer to zero, the scaled problem more closely resembles the original problem, and when the scale factor is small enough and the data are integral, it can be shown that the scaled problem gives a solution to the original problem. Tardos-type [80] proximity theorems can then be applied to turn this weakly polynomial algorithm into a strongly polynomial algorithm.

The idea here is to relax the capacities of arcs by δ. This idea was first used for Min Cost Flow by Ervolina and McCormick [19]. For SFM, every pair of nodes could potentially form an arc, so we introduce a complete directed network on nodes E with *relaxation arcs* $R = \{k \to l \mid k \neq l \in E\}$. We maintain $y \in B(f)$ as before, but we also maintain a flow x in (E, R). We say that x is δ-*feasible* if $0 \leq x_{kl} \leq +\delta$ for all $k \neq l \in E$. We enforce that x is δ-feasible, and that for every $k \neq l \in E$, $x_{kl} \cdot x_{lk} = 0$, i.e., at least one of x_{kl} and x_{lk} is zero. (Some versions of IFF instead enforce that for all $k \neq l \in E$, $x_{kl} = -x_{lk}$, i.e., that x is skew-symmetric, which leads to a simpler description. However, we later sometimes have infinite bounds on some arcs of R which are incompatible with skew-symmetry, so we choose to use this more general representation from the start.) Recall that $\partial x : E \to \mathbb{R}$ is defined as $\partial x_k = \sum_l x_{kl} - \sum_j x_{jk}$. We perturb $y \in B(f)$ by ∂x to get $z = y + \partial x$. If we define $\kappa(S) = |S| \cdot |E - S|$ (which is $|\delta(S)|$ in (E, R), and hence submodular), we could also think of this as relaxing the condition $y \in B(f)$ to $z \in B(f + \delta\kappa)$ (this is the relaxation originated by [45]). The perturbed vector z has enough flexibility that we are able to augment z on paths even though we augment the original vector y arc by arc. The flow x buffers the difference between these two augmentation methods.

The idea of scaling δ instead of $f + \delta\kappa$ is developed for use in Submodular Flow algorithms by Iwata, McCormick, and Shigeno [51], and in an improved version by Fleischer, Iwata, and McCormick [24]. Indeed, some parts of the IFF SFM Algorithm (notably the Swap subroutine below) were inspired by the Submodular Flow algorithm from [24]. It is formally similar to an excess scaling Min Cost Flow algorithm of Goldfarb and Jin [39], with the flow x playing the role of arc excesses.

As $\delta \to 0$, Lemma 3.3 below shows that $\mathbb{1}^T z^-$ converges towards $\mathbb{1}^T y^-$, so we concentrate on maximizing $\mathbb{1}^T z^-$ instead of $\mathbb{1}^T y^-$. We do this by looking for augmenting paths from S^- to S^+ with capacity at least δ (called δ-*augmenting paths*). We modify y arc by arc as needed to try to create further such augmenting paths for z. Roughly speaking, we call z δ-*optimal* if there is no further way to construct a δ-augmenting path. Augmenting on δ-augmenting paths turns out to imply that we make enough progress at each iteration that the number of iterations in a δ-scaling phase is strongly polynomial (only the number of scaling phases is weakly polynomial).

The outline of the outer scaling framework is now clear: We start with $y = v^1$ for an arbitrary order $\prec_1$, and a sufficiently large value of δ (it turns out that $\delta = |y^-(E)|/n^2 \leq 2M/n^2$ suffices). We then cut the value of δ in half, and apply a Refine procedure to make the current values δ-optimal. We continue until the value of δ is small enough that we know that we have an optimal SFM solution (it turns out that $\delta = 1/n^2$ suffices). Thus the number of outer iterations is $1 + \lfloor \log_2 \frac{2M/n^2}{1/n^2} \rfloor = O(\log M)$.

Since the outer scaling framework cuts δ in half, Refine starts by halving the 2δ-feasible flow x to make it a δ-feasible flow.

IFF Outer Scaling Framework

Initialize by choosing $\prec_1$ to be any linear order, $y = v^1$, and $\mathcal{I} = \{1\}$.
Initialize $\delta = |y^-(E)|/n^2$, $x = 0$, and $z = y$. [$z = y + \partial x$ is δ-optimal]
While $\delta \geq 1/n^2$, [when $\delta < 1/n^2$ we are optimal]
    Set $\delta \leftarrow \delta/2$.
    Call REFINE. [converts 2δ-optimality to δ-optimality]
Return last approximate solution from REFINE as optimal SFM solution.

To find δ-augmenting paths, we must restrict the starting and ending nodes to have sufficiently large and small values of z_l, so we define $S^{-\delta}(z) = \{l \in E \mid z_l \leq -\delta\}$, and $S^{+\delta}(z) = \{l \in E \mid z_l \geq +\delta\}$. Further define the subset of arcs of R with residual capacity δ as $R(\delta) = \{k \to l \mid x_{kl} = 0\}$. We look for a directed augmenting path P from some $k \in S^{-\delta}(z)$ to some $l \in S^{+\delta}(z)$ using only arcs of $R(\delta)$. Since P contains only relaxation arcs (no exchange arcs), somewhat surprisingly we do *not* need to ensure that P is a lexicographic shortest path, or even a shortest path at all. Define the set $S = \{l \in E \mid$ there is a path in $(E, R(\delta))$ from $S^{-\delta}(z)$ to $l\}$. If we find such a P (if $S \cap S^{+\delta}(z) \neq \emptyset$), we call AUGMENT($P$) to increase x on arcs in P by δ. If $t \to u \in P$, then $x_{tu} = 0$ and the old contribution of $t \to u$ and $u \to t$ to ∂x_t is $-x_{ut}$. AUGMENT(P) updates $x_{tu} = \delta - x_{ut}$ and $x_{ut} = 0$, so that the new contribution of $t \to u$ and $u \to t$ to ∂x_t is $\delta - x_{ut}$, which is δ larger than before as desired (and their contribution to ∂x_u decreases by δ). Over all arcs of P, this has the effect of increasing ∂x_k by δ, decreasing ∂x_l by δ, and leaving ∂x_h the same for $h \neq k, l$. The corresponding update to $z = y + \partial x$ increases z_k by δ, decreases z_l by δ, and leaves z_h the same for $h \neq k, l$, thereby increasing $\mathbb{1}^T z^-$ by δ. The running time of AUGMENT is dominated by re-computing S, which takes $O(n^2)$ time (since $|R| = O(n^2)$).

IFF Subroutine AUGMENT(P) **for** P **from** $k \in S^{-\delta}(z)$ **to** $l \in S^{+\delta}(z)$

For all $t \to u \in P$ do [augment each arc of P, update $R(\delta)$]
    Set $x_{tu} \leftarrow \delta - x_{ut}$, $x_{ut} \leftarrow 0$.
    If $x_{tu} > 0$ set $R(\delta) \leftarrow R(\delta) - (t \to u)$, and set $R(\delta) \leftarrow R(\delta) \cup (u \to t)$.
Set $\left\{ \begin{array}{l} z_k \leftarrow z_k + \delta \\ z_l \leftarrow z_l - \delta \end{array} \right\}$. [update z, $S^{-\delta}(z)$, $S^{+\delta}(z)$, and S.]
If $\left\{ \begin{array}{l} z_k > -\delta \text{ set } S^{-\delta}(z) \leftarrow S^{-\delta}(z) - k \\ z_l < +\delta \text{ set } S^{+\delta}(z) \leftarrow S^{+\delta}(z) - l \end{array} \right\}$.
Set $S = \{l \in E \mid \exists$ a path in $(E, R(\delta))$ from $S^{-\delta}(z)$ to $l\}$.

What do we do if no augmenting path from $S^{-\delta}(z)$ to $S^{+\delta}(z)$ using only arcs of $R(\delta)$ exists? Suppose that there is some $i \in \mathcal{I}$ such that (l, k) is consecutive in $\prec_i$, $k \in S$ and $l \notin S$. We call such a $(k, l; v^i)$ a *boundary triple*, and let B denote the current set of boundary triples. Note that if $\prec_i$ has no boundary triple, then all $s \in S$ must occur first in $\prec_i$, implying by (5) that

$v^i(S) = f(S)$. Thus

$$\begin{array}{l}\text{If } B = \emptyset, \text{ then } v^i(S) = f(S) \text{ (S is tight for } v^i) \text{ for all } i \in \mathcal{I}, \text{ so that} \\ y(S) = \sum_{i\in\mathcal{I}} \lambda_i v^i(S) = \sum_{i\in\mathcal{I}} \lambda_i f(S) = f(S), \text{ and so } S \text{ is also tight for } y.\end{array} \tag{18}$$

We develop a SWAP$(k, l; v^i)$ procedure below (called *double-exchange* in [24, 50]) to deal with boundary triples.

Note that two different networks are being used here to change two different sets of variables that are augmented in different ways: Augmentations happen on paths, affects variables z, and are defined by and implemented on the network of relaxation arcs. SWAPs happen arc by arc, affects variables y, and are defined by and implemented on the network of arcs of potential boundary triples (where $k \to l$ is an arc iff (l, k) is consecutive in some $\prec_i$). The flow variables x are used to mediate between these different changes.

Let $\prec_j$ be $\prec_i$ with k and l interchanged. Then Lemma 2.5 says that

$$v^j = v^i + c(k, l; v^i)(\chi_k - \chi_l). \tag{19}$$

Then (19) together with (11) implies that

$$y + \lambda_i c(k, l; v^i)(\chi_k - \chi_l) = \lambda_i v^j + \sum_{h\neq i} \lambda_h v^h, \tag{20}$$

so we could take a step of $\lambda_i c(k, l; v^i)$ in direction $\chi_k - \chi_l$ from y. The plan is to choose a step length $\alpha \le \lambda_i c(k, l; v^i)$ and then update $y \leftarrow y + \alpha(\chi_k - \chi_l)$. Then we are sure that the new y also belongs to $B(f)$. This increases y_k and decrease y_l by α. To keep $z = y + \partial x$ invariant, we also modify x_{kl} by α so as to decrease ∂x_k and increase ∂x_l by α.

Recall that x_{kl} was positive (else $k \to l \in R(\delta)$, implying that $l \in S$). As long as $\alpha \le x_{kl}$, updating $x_{kl} \leftarrow x_{kl} - \alpha$ (and keeping $x_{lk} = 0$) modifies ∂x as desired, and keeps x δ-feasible. But there is no reason to use $\alpha > x_{kl}$, since we could instead use $\alpha = x_{kl}$ so that the updated $x_{kl} = 0$, meaning that l would join S, and we would make progress. Thus we choose $\alpha = \min(\lambda_i c(k, l; v^i), x_{kl})$. If $\alpha = x_{kl}$ so that l joins S, we call the SWAP *partial* (since we take only part of the full step from v^i to v^j; *non-saturating* in [50]), else we call it *full* (*saturating* in [50]). Every full SWAP has $\alpha = \lambda_i c(k, l; v^i)$, which implies that $|\mathcal{I}|$ does not change; a partial SWAP increases $|\mathcal{I}|$ by at most one. Since there are clearly at most n partial SWAPs before calling AUGMENT, $|\mathcal{I}|$ can be at most $2n$ before calling REDUCEV.

If $\alpha = x_{kl} < \lambda_i c(k, l; v^i)$ then we'll want to take a step of only x_{kl}. To achieve this, take $x_{kl}/(\lambda_i c(k, l; v^i))$ times (20) plus $(1 - x_{kl}/(\lambda_i c(k, l; v^i)))$ times (11) to get

$$y + x_{kl}(\chi_k - \chi_l) = (x_{kl}/c(k, l; v^i))v^j + (\lambda_i - x_{kl}/c(k, l; v^i))v^i + \sum_{h\neq i} \lambda_h v^h, \tag{21}$$

which shows how to update the λ's in SWAP. The running time of a full SWAP is $O(\mathrm{EO})$. For a partial SWAP, for each h added to S we can update B in $O(n^2)$ time. Thus a partial swap costs $O(\mathrm{EO})$ plus $O(n^2)$ per element added to S. Note that if $x_{kl} = \lambda_i c(k, l; v^i)$ then we have a "degenerate" SWAP that is both partial and full. Although it is partial, $|\mathcal{I}|$ does not change, and although it is full we need to update B anyway. In the complexity analysis we double-count such a SWAP as being both partial and full. The key idea here is trading off (hard to manage)

IFF Subroutine SWAP$(k, l; v^i)$

Set $\beta \leftarrow c(k, l; v^i)$, $\alpha \leftarrow \min(x_{kl}, \lambda_i \beta)$. [compute step length]
If $\alpha = \lambda_i \beta$, [a full SWAP]
    Rename $\prec^i$, v^i as $\prec^j$, v^j, swap k and l in $\prec^j$, add β to v^j_k, subtract β from v^j_l.
    Set $\mathcal{I} \leftarrow \mathcal{I} + j - i$, $\lambda_j \leftarrow \lambda_i$.
Else [$\alpha = x_{kl} < \lambda_i \beta$, a partial SWAP, so $k \to l$ joins $R(\delta)$ and at least l joins S]
    Define $\prec_j$ as $\prec_i$ with k and l interchanged and compute v^j.
    Set $\mathcal{I} \leftarrow \mathcal{I} + j$, $\lambda_i \leftarrow \lambda_i - x_{kl}/\beta$, $\lambda_j \leftarrow x_{kl}/\beta$.
Set $x_{kl} \leftarrow x_{kl} - \alpha$, $\left\{ \begin{array}{l} y_k \leftarrow y_k + \alpha \\ y_l \leftarrow y_l - \alpha \end{array} \right\}$, and update $R(\delta)$ and S.
For each new member h of S do
    Delete any boundary triples $(u, h; v^h)$ from B.
    Add any new boundary triples $(h, u; v^h)$ to B.

exchange capacity for (easy to manage) flow on the relaxation arcs, and this idea comes from [24].

REFINE stops and concludes that the current point is δ-optimal when it can no longer find any augmenting paths and $B = \emptyset$. We show later that the running time of REFINE is $O(n^5 \text{EO})$.

IFF Subroutine REFINE

Set $x \leftarrow x/2$. [make x δ-feasible]
For all $l \in E$ do [update z]
    Set $z_l \leftarrow y_l + \partial x_l$.
Compute $S^{-\delta}(z)$, $S^{+\delta}(z)$, $R(\delta)$, S, and B.
While augmenting paths exist ($S \cap S^{+\delta}(z) \neq \emptyset$), or $B \neq \emptyset$ do
    While $\exists$ path P from $S^{-\delta}(z)$ to $S^{+\delta}(z)$ using arcs from $R(\delta)$, do
        AUGMENT(P) and set B to be boundary triples w.r.t. new S.
    While $\nexists$ path P from $S^{-\delta}(z)$ to $S^{+\delta}(z)$ using arcs from $R(\delta)$ and $B \neq \emptyset$, do
        Find a boundary triple $(k, l; v^i)$ and SWAP$(k, l; v^i)$.
    Call REDUCEV.
Return S as an approximate optimum solution.

Recall from Section 2.6.1 that our optimality condition for S solving SFM is that $y^-(E) = f(S)$. The following lemma (which is a relaxed version of Lemma 2.10) shows for both y and z how close these approximate solutions are to exactly satisfying $y^-(E) = f(S)$ and $z^-(E) = f(S)$, as a function of δ.

Lemma 3.3 *When a δ-scaling phase ends, S is tight for y, and we have $y^-(E) \geq f(S) - n^2\delta$ and $z^-(E) \geq f(S) - n\delta$.*

Proof: When REFINE ends, $B = \emptyset$, and then (18) says that S is tight for y.

Because the δ-scaling phase ended, we have $S^{-\delta}(z) \subseteq S \subseteq E - S^{+\delta}(z)$. This implies that for every $k \in S$, $z_k < +\delta$. Thus $z^-(S) \geq \sum_{e:z_e \leq 0}(z_e - \delta) + \sum_{e:z_e > 0}(z_e - \delta) = z(S) - |S|\delta$. For

every $l \in E - S$, $z_l > -\delta$, and so $z^-(E-S) > -|E-S|\delta$. Since every $k \to l$ with $k \in S$ and $l \notin S$ has $x_{kl} > 0$, $\partial x(S) = \sum_{k \in S,\ l \notin S} x_{kl} > 0$. Therefore $y(S) = z(S) - \partial x(S) < z(S)$. Then $z^-(E) = z^-(S) + z^-(E-S) > (z(S) - |S|\delta) - |E-S|\delta > y(S) - n\delta = f(S) - n\delta$.

Note that for any $l \in E$ and any δ-feasible x, $-(n-1)\delta \leq \partial x_l \leq (n-1)\delta$. Since $y_k = z_k + \partial x_k$, $y_k \geq z_k - (n-1)\delta$, and so $y^-(E) \geq z^-(E) - n(n-1)\delta > (f(S) - n\delta) - n(n-1)\delta = f(S) - n^2\delta$. ■

We now use this to prove correctness and running time. We now formally define z to be *δ-optimal* (for set T) if there is some $T \subseteq E$ such that $z^-(E) \geq f(T) - n\delta$. Lemma 3.3 shows that the z at the end of each δ-scaling phase is δ-optimal for the current approximate solution S. As before, we pick out the main points in boldface.

Theorem 3.4 *The IFF SFM Algorithm is correct for integral data and runs in $O(n^5 \log M \cdot \mathrm{EO})$ time.*

Proof:

The current approximate solution T at the end of a δ-scaling phase with $\delta < 1/n^2$ solves SFM: Lemma 3.3 shows that $y^-(E) \geq f(T) - n^2\delta > f(T) - 1$. But for any $U \subseteq E$, $f(U) \geq y(U) \geq y^-(E) > f(T) - 1$. Since f is integer-valued, T solves SFM.

The first δ-scaling phase calls Augment $O(n^2)$ **times:** Denote initial values with hats. Recall that $\hat{\delta} = |\hat{y}^-(E)|/n^2$. Now $\hat{x} = 0$ implies that $\hat{z} = \hat{y}$, so that $\hat{z}^-(E) = \hat{y}^-(E)$. Since $z^-(E)$ monotonically increases during Refine and is always non-positive, the total increase in $z^-(E)$ is no greater than $|\hat{y}^-(E)| = n^2\hat{\delta}$. Since each Augment increases $z^-(E)$ by δ, there are only $O(n^2)$ calls to Augment.

Subsequent δ-scaling phases call Augment $O(n^2)$ **times:** After halving δ, for the data at the end of the previous scaling phase we had $z^-(E) \geq f(T) - 2n\delta$. Making x δ-feasible at the beginning of Refine changes each x_{kl} by at most δ, and so degrades this to at worst $z^-(E) \geq f(T) - (2n + n^2)\delta$. Now $z^-(E) \leq y^-(E) + n^2\delta$, and $y^-(E)$ cannot be larger than $f(T)$, so that $z^-(E) \leq f(T) + n^2\delta$. Each call to Augment increases $z^-(E)$ by δ, so Augment gets called at most $2n + 2n^2 = O(n^2)$ times.

There are $O(n^3)$ full Swaps **before each call to** Augment**:** Each full Swap$(k, l; v^i)$ replaces v^i by v^j where l is one position higher in v^j than in v^i. Consider one v^i and the sequence of v^j's generated from v^i by full Swaps. Since each such Swap moves an element l of $E - S$ one position higher in its linear order, and no operations before Augment allow elements of $E - S$ to become lower, no pair k, l occurs more than once in a boundary triple. There are $O(n^2)$ such pairs for each v^i, and $O(n)$ v^i's, for a total of $O(n^3)$ full Swaps before calling Augment.

The total amount of work in all calls to Swap **before a call to** Augment **is $O(n^3\mathrm{EO})$:** There are $O(n^3)$ full Swaps before the Augment, and each costs $O(\mathrm{EO})$. Each node added to S by a partial Swap costs $O(n^2)$ time to update B, and this happens at most n times before we must include a node of $S^{+\delta}(z)$, at which point we call Augment. Each partial Swap adds at least one node to S and costs $O(\mathrm{EO})$ other than updating B. Hence the total Swap-cost before the Augment is $O(n^3\mathrm{EO})$.

The time for one call to Refine **is $O(n^5\mathrm{EO})$:** Each call to Refine calls Augment $O(n^2)$ times. The call to Augment costs $O(n^2)$ time, the work in calling Swap before the Augment is $O(n^3\mathrm{EO})$, and the work in calling ReduceV after the Augment is $O(n^3)$, so we charge $O(n^3\mathrm{EO})$ to each Augment.

There are $O(\log M)$ calls to REFINE: For the initial $\hat{y}$, $\hat{y}(E) = f(E) \geq -M$. Let T be the set of elements where $\hat{y}$ is positive. Then $\hat{y}^+(E) = \hat{y}(T) \leq f(T) \leq M$. Thus $\hat{y}^-(E) = \hat{y}(E) - \hat{y}^+(E) \geq -2M$, so $\hat{\delta} = |y^-(E)|/n^2 \leq 2M/n^2$. Since δ's initial value is at most $2M/n^2$, it ends at $1/n^2$, and is halved at each REFINE, there are $O(\log M)$ calls to REFINE.

The total running time of the algorithm is $O(n^5 \log M \cdot \text{EO})$: Multiplying together the factors from the last two paragraphs gives the claimed total time. ■

3.3.2 Making the IFF Algorithm Strongly Polynomial

We now develop a strongly polynomial version of the IFF algorithm that we call IFF-SP. The challenge in making a weakly polynomial scaling algorithm like the IFF Algorithm strongly polynomial is to avoid having to call REFINE for each scaled value of δ, since the weakly polynomial factor $O(\log M)$ is really $\Theta(\log M)$. The rough idea is to find a way for the current data of the problem to reveal a good starting value of δ, and then to apply $O(\log n)$ calls to REFINE to get close enough to optimality that we can "fix a variable", which can happen only a strongly polynomial number of times. Letting the current data determine the value of δ can also be seen as a way to allow the algorithm to make much larger decreases in δ than would be available in the usual scaling framework.

The general mechanism for fixing a variable is to prove a "proximity lemma" as in Tardos [80] that says that if the value of a variable gets too far from a bound, then we can remove that bound, and then reduce the size of the problem. In this case, the proximity lemma below says that if we have some $y \in B(f)$ such that y_l is negative enough w.r.t. δ, then we know that l belongs to every minimizer of f. This is a sort of approximate complementary slackness for LP (9): Complementary slackness for exact optimal solutions y^* and S^* says that $y^*_e < 0$ implies that $e \in S^*$, and the lemma says that for δ-optimal y, $y_e < -n^2\delta$ implies that $e \in S^*$.

Lemma 3.5 *At the end of a δ-scaling phase, if there is some $l \in E$ such that the current y satisfies $y_l < -n^2\delta$, then l belongs to every minimizer of f.*

Proof: By Lemma 3.3, at the end of a δ-scaling phase, for the current approximate solution S, we have $y^-(E) \geq f(S) - n^2\delta$. If S^* solves SFM, we have $f(S) \geq f(S^*) \geq y(S^*) \geq y^-(S^*)$. These imply that $y^-(E) \geq y^-(S^*) - n^2\delta$, or $y^-(E - S^*) \geq -n^2\delta$. Then if $l \in E - S^*$, we could add $-y_l > n^2\delta$ to this to get $y^-(E - S^* - l) > 0$, a contradiction, so we must have $l \in S^*$. ■

There are two differences between how we use this lemma and how IFF [50] use it. First, we apply the lemma in a more "lazy" way than IFF proposed that is shorter and simpler to describe, and which extends to the bisubmodular case [62], whereas the IFF approach seems not to extend [31]. Second, we choose to implement the algorithm taking the structure it builds on the optimal solution explicitly into account (as is done in Iwata [46]) instead of implicitly into account (as is done in [50]), which requires us to slightly generalize Lemma 3.5 into Lemma 3.8 below.

We compute and maintain a set OUT of elements proven to be *out* of every optimal solution, effectively leading to a reduced problem on $E - \text{OUT}$. Previously we used M to estimate the "size" of f. The algorithm deletes "big" elements, so that the reduced problem consists of "smaller" elements, and we need a sharper initial estimate δ^0 of the size of the reduced problem. At first we choose $f(u) = \max_{l \in E} f(l)$ and $\delta^0 = f(u)^+$. Let $\hat{y} \in B(f)$ be an initial point coming from Greedy. Then $\hat{y}^+(E) = \sum_e \hat{y}^+_e \leq n\delta^0$, so that $\hat{y}^-(E) = \hat{y}(E) - \hat{y}^+(E) \geq f(E) - n\delta^0$. Thus,

if we choose $x = 0$, then $\hat{z} = \hat{y} + \partial\hat{x} = \hat{y}$, so that E proves that $\hat{z}$ is δ^0-optimal. Thus we could start calling REFINE with $y = \hat{y}$ and $\delta = \delta^0$.

Suppose we have some set T such that $f(T) \leq -\delta^0$; we call such a set *highly negative*. Then $\lceil \log_2(2n^3) \rceil = O(\log n)$ (a strongly polynomial number) calls to REFINE produces some δ-optimal y with $\delta < \delta^0/n^3$. Subroutine FIX makes these $O(\log n)$ calls to REFINE. But $y(T) \leq f(T) \leq -\delta^0 < -n^3\delta$ implies that there is at least one $t \in T$ with $y_t < -n^2\delta$, and Lemma 3.5 then shows that such t belong to every minimizer of f. We call such a t a *highly negative element*. This would be great, but IFF must go to some trouble to manufacture such a highly negative T.

Instead we adapt a "lazy" version of the IFF idea of considering the set function on $E - u$ defined by $f_u(S) = f(S + u) - f(u) = f(S + u) - \delta^0$. Clearly f_u is submodular on $E - u$ with $f_u(\emptyset) = 0$. Now apply FIX to f_u. Suppose that FIX does not find any highly negative element for f_u. This implies that there cannot be a highly negative set T for f_u. Then we know that for every T not containing u, $-\delta^0 < f_u(T) = f(T + u) - f(u) = f(T + u) - \delta^0$, or $f(T + u) > 0 = f(\emptyset)$. This proves that u cannot belong to any minimizer of f, and so we add u to OUT. On the other hand, suppose that FIX identifies at least one highly negative element t (which is guaranteed if there exists a highly negative set T for f_u). Then t belongs to every minimizer of f_u. Note that any minimizer of f_u actually solves the problem of minimizing $f(S)$ over subsets of E containing u. Therefore we would get the *condition* that every minimizer of f that contains u must also contain t. Note that it is possible that there is no highly negative set for f_u but that FIX identifies some highly negative element t anyway. This is not a problem, since Lemma 3.5 still implies the condition that any minimizer containing u must also contain t. Each new condition arc $u \to t$ means that we no longer need to consider sets containing u but not t as possible SFM solutions, thereby reducing the problem. Only $O(n^2)$ condition arcs can be added before the reduced problem becomes trivial, so this is real progress.

As the algorithm proceeds we need some way of tracking such conditions. We do this by maintaining a set of arcs C on node set E, where arc $k \to l$ means that every minimizer of f containing k must also contain l. We start with $C = \emptyset$, and add arcs to C as we go along. If adding an arc creates a directed cycle Q in (E, C), then the nodes in Q either all belong to every minimizer of f, or none belong to every minimizer of f.

Dealing with (E, C) adds a new layer of complexity to the algorithm. For $u \in E$ define the *descendants* of u as $D_u = \{l \in E \mid$ there is a directed path from u to l in $(E, C)\}$, and the *ancestors* of u as $A_u = \{l \in E \mid$ there is a directed path from l to u in $(E, C)\}$. If FIX finds a highly negative l (so that l belongs to every minimizer of f_u), then we know that D_l must also belong to every minimizer of f_u. Similarly, if we add u to OUT, we must also add all of A_u to OUT. Doing this ensures that whenever we call FIX, the arcs we find for C are indeed new, and so that we make real progress.

Let $\mathcal{C}$ be the set of strongly connected components of $(E - \text{OUT}, C)$. By the above comments, for every $\sigma \in \mathcal{C}$, every solution to SFM either includes all or no nodes of σ. Thus C is better thought of as being a set of arcs on the node subset $\mathcal{C}$. Thus we should re-define descendants (resp. ancestors) from D_u (A_u) for $u \in E - \text{OUT}$ to D_σ (A_σ) for $\sigma \in \mathcal{C}$, again as the set of nodes of $\mathcal{C}$ reachable from σ (that σ can reach) via arcs of C. If $\mathcal{S} \subseteq \mathcal{C}$, define $E(\mathcal{S}) = \cup_{\sigma \in \mathcal{S}} \sigma$, the set of original elements contained in the union of strong components in $\mathcal{S}$. Therefore our general situation is that we have OUT $\subset E$ as the set of nodes out of an optimal solution, and we are essentially solving a reduced SFM problem on the contracted set of elements $\mathcal{C}$, which partitions $E - \text{OUT}$.

Subset $\mathcal{S} \subseteq \mathcal{C}$ can be part of an SFM solution only if no arc of C exits $\mathcal{S}$, i.e., if $\delta^+(\mathcal{S}) = \emptyset$. In this case we call $\mathcal{S}$ *closed* (or an *ideal*). Note that the family $\mathcal{D}$ of closed sets is closed under unions and intersections (it is a ring family), and we say that $(\mathcal{C}, C)$ *represents* $\mathcal{D}$ (in the sense of Birkhoff's Theorem [7]). Thus a solution to SFM for f has the form $E(\mathcal{S})$ for some $\mathcal{S} \in \mathcal{D}$. For $\mathcal{S} \in \mathcal{D}$, define $\hat{f}(\mathcal{S}) = f(E(\mathcal{S}))$, so that $\hat{f}(\emptyset) = 0$ and $\hat{f}$ is submodular on $\mathcal{D}$. Essentially $\hat{f}$ is just f restricted to $E - \textsc{Out}$, and then with each of the components of $\mathcal{C}$ contracted to a single new element. With good data structures for representing $\mathcal{C}$ we can evaluate $\hat{f}$ using just one call to the evaluation oracle $\mathcal{E}$ for f, so we use EO to also count evaluations of $\hat{f}$. We also need to re-define f_u for $u \in E$ to be a set function $\hat{f}_\tau$ for $\tau \in \mathcal{C}$. Since D_τ is closed, $D_\tau \in \mathcal{D}$. Define $\mathcal{D}_\tau$ to be the subsets $\mathcal{S} \subseteq \mathcal{C} - D_\tau$ such that $\mathcal{S} \cup D_\tau$ is closed (again a ring family). The graph representing $\mathcal{D}_\tau$ is $(\mathcal{C} - D_\tau, C)$, which is $(\mathcal{C}, C)$ with the nodes of D_τ (and any incident arcs) deleted. For $\mathcal{S} \in \mathcal{D}_\tau$ define $\hat{f}_\tau(\mathcal{S}) = \hat{f}(\mathcal{S} \cup D_\tau) - \hat{f}(D_\tau)$. Then $\hat{f}_\tau$ is submodular, has $\hat{f}_\tau(\emptyset) = 0$, and can be evaluated using only two calls to the evaluation oracle for $\hat{f}$. Thus we also use EO for $\hat{f}_\tau$.

Instead of restricting $\hat{f}$ to the closed subsets of $\mathcal{C}$, we could define it on all subsets of $\mathcal{C}$ via $\hat{f}(\mathcal{S}) = f(E(\mathcal{S}))$ for any $\mathcal{S} \subseteq \mathcal{C}$ (and similarly for $\hat{f}_\tau$). Since we call Fix on the set of contracted elements $\mathcal{C} - D_\tau$, we would still be sure that any condition arcs found by Fix are new (do not already belong to C), and we could use Lemma 3.5 as it stands. This *implicit* method of handling $\mathcal{D}_\tau$ is used by IFF [50]. Here we use choose to use the slightly more complicated *explicit* method (developed for Iwata's fully combinatorial version of IFF [46]) that does restrict $\hat{f}_\tau$ to $\mathcal{D}_\tau$ because it yields better insight into the structure of the problem, and it is needed for Lemma 3.10 (which is crucial for making the fully combinatorial version work). It also allows us to demonstrate how to modify Refine to work over a ring family, which is needed in Section 5. (The published version of [46] contains an error pointed out by M. Kriesell: It handles flow x as needed for the explicit method, but uses the implicit method Lemma 3.5 instead of the explicit method Lemma 3.8; a corrected version is available at `http://www.sr3.t.u-tokyo.ac.jp/~iwata/`.)

We call the extended version of Refine (that can deal with optimizing over a ring family such as $\mathcal{D}$ instead of 2^E) RefineR. There are only two changes that we need to make to Refine. First, we must ensure that our initial $y = v^\prec$ comes from an order $\prec$ that is consistent with $\mathcal{D}$ (recall that this means that $\sigma \to \rho \in C$ implies that $\rho \prec \sigma$; this change is needed for both the implicit and explicit methods). This is easy to achieve, since we can take any order coming from an acyclic labeling of $(\mathcal{C}, C)$.

Second, we must ensure that all $v^i \in \mathcal{I}$ that arise in the algorithm also have $\prec_i$ consistent with $\mathcal{D}$. We adapt a device suggested by Fujishige [29, Section 14.1 (d)]: we keep a separate flow φ on C. Flows $x_{\sigma\rho}$ have the bounds $0 \le x_{\sigma\rho} \le \delta$, and $\varphi_{\sigma\rho}$ have the bounds $0 \le \varphi_{\sigma\rho} \le \infty$ (we could achieve the same effect without φ by relaxing the upper bounds for x on arcs of C to ∞, but this version is more convenient for the Hybrid Algorithm). Augmentations affect only x, and $R(\delta)$ contains only δ-augmentable arcs w.r.t. x. We now keep the invariant that $z = y + \partial x + \partial\varphi$, and (for the SP and FC versions) define $w = y + \partial\varphi$ so that $z = w + \partial x$. Note that every constraint $y(S) \le \hat{f}(S)$ defining $B(\hat{f})$ comes from some closed $S \in \mathcal{D}$, and each such S has no arcs of C exiting it. Hence for any $S \in \mathcal{D}$ (since $\varphi \ge 0$) $\partial\varphi(S) \le 0$, and so $y \in B(\hat{f})$ implies that $w \in B(\hat{f})$ (recall that $w = y + \partial\varphi$ is how all points in the (now unbounded) $B(\hat{f})$ arise).

While searching for an augmenting path, if there is some $\sigma \to \rho \in C$ with $\sigma \in S$ and $\rho \notin S$ ($\Rightarrow x_{\sigma\rho} > 0$, else $\sigma \to \rho \in R(\delta)$) then we apply FlowSwap: We set $\varphi_{\sigma\rho} \leftarrow \varphi_{\sigma\rho} + x_{\sigma\rho}$ and $x_{\sigma\rho} \leftarrow 0$. If $x_{\sigma\rho} > 0$ and $\rho \to \sigma \in C$ with $\varphi_{\rho\sigma} \ge x_{\sigma\rho}$, then instead FlowSwap sets $\varphi_{\rho\sigma} \leftarrow \varphi_{\rho\sigma} - x_{\sigma\rho}$ and

$x_{\sigma\rho} \leftarrow 0$. Since the new $x_{\sigma\rho} = 0$, $\sigma \to \rho$ joins $R(\delta)$, and so ρ joins S. Note that this update leaves $\partial\varphi + \partial x$ invariant. FlowSwap is the only operation that changes φ. Then $\sigma \to \rho \in C$ implies that $(\sigma, \rho; v^i)$ can never be a boundary triple (since then FlowSwap would ensure that $\rho \in S$), so an inconsistent $\prec_j$ is never created. This also implies that S always belongs to $\mathcal{D}$, so the optimal solution belongs to $\mathcal{D}$. We now need to re-visit Lemmas 3.3 and 3.5 in light of the new representation of z. The next lemma is the ring equivalent of Lemma 3.3, and we express it in terms of $\hat{f}$ defined on generic ring family $\mathcal{D} \subseteq 2^{\mathcal{C}}$.

Lemma 3.6 *When a δ-scaling phase ends, $S \in \mathcal{D}$, S is tight for y, and we have $w^-(\mathcal{C}) \geq \hat{f}(S) - n^2\delta$ and $z^-(\mathcal{C}) \geq \hat{f}(S) - n\delta$.*

Proof: When Refine ends, no arc of C exits S, and so $S \in \mathcal{D}$. Also, $B = \emptyset$, and then (18) says that S is tight for y.

Because the δ-scaling phase ended, we have $S^{-\delta}(z) \subseteq S \subseteq \mathcal{C} - S^{+\delta}(z)$. Similar to the proof of Lemma 3.3 this implies that $z^-(\mathcal{C}) > z(S) - n\delta$. If $\rho \to \sigma \notin C$, interpret $\varphi_{\rho\sigma}$ as 0. Then FlowSwap implies that if $\sigma \in S$ and $\rho \notin S$, then $x_{\sigma\rho} - \varphi_{\rho\sigma} > 0$. Since no arc of C exits S, $\partial x(S) + \partial\varphi(S) = \sum_{\sigma\in S,\ \rho\notin S}(x_{\sigma\rho} - \varphi_{\rho\sigma}) > 0$. Therefore $y(S) = z(S) - \partial x(S) - \partial\varphi(S) < z(S)$. Then $z^-(\mathcal{C}) > z(S) - n\delta > y(S) - n\delta = \hat{f}(S) - n\delta$.

Note that for any $\rho \in \mathcal{C}$ and any δ-feasible x, $-(n-1)\delta \leq \partial x_\rho \leq (n-1)\delta$. Since $w_\sigma = z_\sigma + \partial x_\sigma$, $w_\sigma \geq z_\sigma - (n-1)\delta$, and so $w^-(\mathcal{C}) \geq z^-(\mathcal{C}) - n(n-1)\delta > (\hat{f}(S) - n\delta) - n(n-1)\delta = \hat{f}(S) - n^2\delta$. ■

Weakly Polynomial IFF on Ring Families We can use this lemma and RefineR to immediately adapt the weakly polynomial version of IFF so that it works directly on ring families.

IFF Weakly Polynomial Algorithm for Ring Families

Initialize by choosing $\prec_1$ to be any consistent linear order, $y = v^1$, and $\mathcal{I} = \{1\}$.
Initialize $\delta^0 = \max_\sigma \hat{f}(D_\sigma) - \hat{f}(D_\sigma - \sigma)$, $x = \varphi = 0$, and $z = w = y$.
While $\delta \geq 1/n^2$, [when $\delta < 1/n^2$ we are optimal]
    Set $\delta \leftarrow \delta/2$.
    Call RefineR. [converts 2δ-optimality to δ-optimality]
Return last approximate solution from RefineR as optimal SFM solution.

Theorem 3.7 *When $\hat{f}$ is integer-valued, this Algorithm solves SFM over the ring family in $O(n^5\text{EO}\log(nM))$ time.*

Proof: Lemma 3.6 shows that the S produced at the end of one call to RefineR belongs to $\mathcal{D}$ and proves that the initial z at the next call to RefineR is 2δ-optimal. Similar to Theorem 3.4, if a call to RefineR ends with $\delta < 1/n^2$, then S solves SFM.

By Lemma 2.2, for $y^0 = w^0$ as the initial values of y and w, we have $w^0(E) \leq 2n\delta^0 = 2n\delta^0 - \hat{f}(\emptyset)$, and so w^0 is δ^0-optimal. Clearly $\delta^0 \leq 2M$. Since each call to RefineR halves δ, there are at most $\log_2(2Mn^2) = O(\log(nM))$ such calls. As in Theorem 3.4, each call costs $O(n^5\text{EO})$ time, for a total of $O(n^5\text{EO}\log(nM))$ time. ■

It is disappointing that we lose a factor of $O(\log n)$ in running time when we move from 2^E to $\mathcal{D}$ (compare Theorem 3.4 to Theorem 3.7), as the weakly polynomial bound is interesting precisely when M is "small", and so when $\log n$ is large relative to $\log M$. In practice one would often have an initial solution with a small gap to optimality and so the run time would still be good, or one could instead use the method of Section 5.2.

The next lemma is the ring equivalent of Lemma 3.5. Since we always apply it to $\hat{f}_\tau$, we express it in terms of $\hat{f}_\tau$ defined on ring family $\mathcal{D}_\tau \subseteq 2^{\mathcal{C}-D_\tau}$.

Lemma 3.8 *At the end of a δ-scaling phase, if there is some $\sigma \in \mathcal{C} - D_\tau$ such that the current w satisfies $w_\sigma < -n^2\delta$, then σ belongs to every minimizer of $\hat{f}_\tau$.*

Proof: By Lemma 3.6, at the end of a δ-scaling phase, for the current approximate solution S, we have $w^-(\mathcal{C} - D_\tau) \geq \hat{f}_\tau(S) - n^2\delta$. If S^* solves SFM, we have $\hat{f}_\tau(S) \geq \hat{f}_\tau(S^*) \geq w(S^*) \geq w^-(S^*)$. These imply that $w^-(\mathcal{C} - D_\tau) \geq w^-(S^*) - n^2\delta$, or $w^-((\mathcal{C} - D_\tau) - S^*) \geq -n^2\delta$. Then if $\sigma \in (\mathcal{C} - D_\tau) - S^*$, we could add $-w_\sigma > n^2\delta$ to this to get $w^-((\mathcal{C} - D_\tau) - S^* - \sigma) > 0$, a contradiction, so we must have $\sigma \in S^*$. ■

IFF-SP Subroutine FIX$(\hat{f}_\tau, (\mathcal{C} - D_\tau, C), \delta^0)$
Applies to $\hat{f}_\tau$ defined on closed sets of $(\mathcal{C} - D_\tau, C)$, and $y_\sigma \leq \delta^0$ for all $y \in B(\hat{f}_\tau)$.

Initialize $\prec$ as any linear order consistent with C, $y \leftarrow v^\prec$, $\delta \leftarrow \delta^0$, and $\mathcal{N} = \emptyset$.
Initialize $x = \varphi = 0$ and $z = y + \partial x + \partial\varphi$ $(= y)$.
While $\delta \geq \delta^0/n^3$ do
  Set $\delta \leftarrow \delta/2$.
  Call REFINER.
For $\sigma \in \mathcal{C} - D_\tau$ do [add descendants of highly negative nodes to $\mathcal{N}$]
  If $w_\sigma = y_\sigma + \partial\varphi_\sigma < -n^2\delta$ set $\mathcal{N} \leftarrow \mathcal{N} \cup D_\sigma$.
Return $\mathcal{N}$.

Define $\delta^0 = \max_{\sigma \in \mathcal{C}} \hat{f}(D_\sigma) - \hat{f}(D_\sigma - \sigma)$. Lemma 2.2 shows that δ^0 is an upper bound on the components of any y in the convex hull of the vertices of $B(\hat{f})$, and we show below that if $\delta^0 \leq 0$, then $E - $OUT solves SFM for f (it is not hard to show that δ^0 is monotone non-increasing during the algorithm). So we can assume that $\delta^0 > 0$, and we take this as the "size" of the current solution. Suppose that τ achieves the max for δ^0, i.e., that $\delta^0 = \hat{f}(D_\tau) - \hat{f}(D_\tau - \tau)$. We then apply FIX to $\hat{f}_\tau$. If FIX finds a highly negative σ then we add $\tau \rightarrow \sigma$ to C; if it finds no highly negative elements, then we add $E(A_\tau)$ to OUT.

Theorem 3.9 *IFF-SP is correct, and runs in $O(n^7 \log n \cdot \mathrm{EO})$ time.*

Proof:

If $\delta^0 \leq 0$ then $E - $OUT solves SFM for f: Lemma 2.2 shows that for the current y and any $\sigma \in \mathcal{C}$, $y_\sigma \leq 0$. Thus $y^-(\mathcal{C}) = y(\mathcal{C}) = \hat{f}(\mathcal{C})$, proving that $\mathcal{C}$ solves SFM for $\hat{f}$. We know that any solution T of SFM for f must be of the form $E(\mathcal{T})$ for some $\mathcal{T} \in \mathcal{D}$. By optimality of $\mathcal{C}$ for $\hat{f}$, $\hat{f}(\mathcal{C}) \leq \hat{f}(\mathcal{T})$, or $f(E - \mathrm{OUT}) = f(E(\mathcal{C})) \leq f(E(\mathcal{T})) = f(T)$, so $E - $OUT is optimal for f.

IFF Strongly Polynomial Algorithm (IFF-SP)

Initialize OUT $\leftarrow \emptyset$, $C \leftarrow \emptyset$, $\mathcal{C} \leftarrow E$.
While $|\mathcal{C}| > 1$ do
  Compute $\delta^0 = \max_{\sigma \in \mathcal{C}} \hat{f}(D_\sigma) - \hat{f}(D_\sigma - \sigma)$ and let $\tau \in \mathcal{C}$ attain the maximum.
  If $\delta^0 \leq 0$ then return $E -$ OUT as an optimal SFM solution.
  Else ($\delta^0 > 0$)
    Set $\mathcal{N} \leftarrow Fix(\hat{f}_\tau, (\mathcal{C} - D_\tau, C), \delta^0)$.
    If $\mathcal{N} \neq \emptyset$, for all $\sigma \in \mathcal{N}$ add $\tau \rightarrow \sigma$ to C, update $\mathcal{C}$, and all D_ρ's, A_ρ's.
    Else ($\mathcal{N} = \emptyset$) set OUT $\leftarrow$ OUT $\cup E(A_\tau)$.
Return whichever of $\emptyset$ and $E -$ OUT has a smaller function value.

In FIX$(\hat{f}_\tau, (\mathcal{C}, C), \delta^0)$ with $\delta^0 > 0$, the first call to REFINER calls AUGMENT $O(n)$ times: Lemma 2.2 shows that for the current y and any $\sigma \in \mathcal{C}$, $y_\sigma \leq \delta^0$. In the first call to REFINER we start with $z = y$, so that $z^+(\mathcal{C}) = y^+(\mathcal{C})$. Since $y_\sigma \leq \delta^0$ for each $\sigma \in \mathcal{C}$, we get $z^+(\mathcal{C}) = y^+(\mathcal{C}) \leq n\delta^0$. Each call to AUGMENT reduces $z^+(\mathcal{C})$ by $\delta^0/2$. Thus there are at most $2n$ calls to AUGMENT during the first call to REFINER.

The time for one call to REFINER is $O(n^5 \text{EO})$: By the same argument as in Theorem 3.4.

When a highly negative $\mathcal{T} \in \mathcal{D}_\tau$ exists, a call to FIX$(\hat{f}_\tau, (\mathcal{C} - D_\tau, C), \delta^0)$ results in at least one element added to $\mathcal{N}$: The call to FIX reduces δ from δ^0 to below δ^0/n^3. Then $\mathcal{T}$ highly negative and $\mathcal{T} \in \mathcal{D}_\tau$ imply that $w(\mathcal{T}) \leq y(\mathcal{T}) \leq \hat{f}(\mathcal{T}) \leq -\delta^0 < -n^3\delta$. This implies that there is at least one $\rho \in \mathcal{C}$ with $w_\rho < -n^2\delta$, so at least one element gets added to $\mathcal{N}$.

If FIX$(\hat{f}_\tau, (\mathcal{C} - \mathcal{D}_\tau, C), \delta^0)$ finds no highly negative element, then $E(A_\tau)$ belongs to no minimizer of f: As above, if there were a highly negative set $\mathcal{T}$ for $\hat{f}_\tau$, then the call to FIX would find a highly negative element. Thus for all $\mathcal{T} \in \mathcal{D}_\tau$ we have $-\delta^0 < \hat{f}_\tau(\mathcal{T})$, or $-\hat{f}(D_\tau) + \hat{f}(D_\tau - \tau) < \hat{f}(\mathcal{T} \cup D_\tau) - \hat{f}(D_\tau)$, or $f(E(D_\tau - \tau)) < f(E(\mathcal{T} \cup D_\tau))$. Since $E(\mathcal{T} \cup D_\tau)$ is a generic feasible set containing τ and $E(D_\tau - \tau)$ is a specific set not containing τ, no set containing τ can be optimal. Thus adding $E(A_\tau)$ to OUT is correct.

The algorithm returns a solution to SFM: If some $\delta^0 \leq 0$, then we showed above that the returned $E -$ OUT is optimal. Otherwise the algorithm terminates because $|\mathcal{C}| = 1$. In this case the only two choices left for solving SFM are $E(\mathcal{C}) = E -$ OUT and $\emptyset$, and the algorithm returns the better of these.

FIX **calls** REFINER $O(\log n)$ **times:** Parameter δ starts at δ^0, ends at its first value below δ^0/n^3, and is halved at each iteration. Thus there are $\lceil \log_2(2n^3) \rceil = O(\log n)$ calls to REFINER.

The algorithm calls FIX $O(n^2)$ times: Each call to FIX either (i) adds at least one element to OUT, or (ii) adds at least one arc to C. Case (i) happens at most n times. Since there are only $n(n-1)$ possible arcs for C, case (ii) happens $O(n^2)$ times.

The algorithm runs in $O(n^7 \log n \cdot \text{EO})$ time: Each call to FIX calls REFINER $O(\log n)$ times, so the time for one call to FIX is $O(n^5 \log n \cdot \text{EO})$. The algorithm calls FIX $O(n^2)$ times, for a total time of $O(n^7 \log n \cdot \text{EO})$. ■

3.3.3 Iwata's Fully Combinatorial SFM Algorithm

Iwata's algorithm [46] is a fully combinatorial extension of IFF-SP, and so we call it IFF-FC. Finding a fully combinatorial SFM algorithm answers a natural question. It also turns out to

be useful: So far the only known polynomial algorithm for line search in submodular polyhedra was developed by Nagano [67], and it works by embedding IFF-FC inside Megiddo's parametric framework [63].

Recall that a fully combinatorial algorithm cannot use multiplication or division, and must also be strongly polynomial. This implies that it cannot call ReduceV, since the linear algebra in ReduceV apparently needs to use multiplication and division in a way that cannot be simulated with addition and subtraction. This suggests that we adapt an existing algorithm by avoiding the calls to ReduceV; this would probably degrade the running time since $|\mathcal{I}|$ would be allowed to get much larger than n, but as long as we could show that $|\mathcal{I}|$ remained polynomially-bounded, we should still be ok.

Let's try to imagine a fully combinatorial version of (either version of) Schrijver's Algorithm. A key part of the running time proof of Theorem 3.2 is that Push has $O(n^2)$ iterations since each saturating Push either reduces $\max_i |(l,k]_{\prec_i}|$, or the number of $i \in \mathcal{I}$ attaining this max. Without ReduceV, the first saturating Push could have $|(l,k]_{\prec_i}| = n-1$ and could create $n-2$ v^j's with $|(l,k]_{\prec_i}| = n-2$; these could each cause $n-2$ saturating Pushes, each of which creates $n-3$ v^j's with $|(l,k]_{\prec_i}| = n-3$; these $(n-2)(n-3)$ v_j's could each cause $n-3$ saturating Pushes, each of which creates $n-4$ v^j's with $|(l,k]_{\prec_i}| = n-4$; these $(n-2)(n-3)(n-4)$ v_j's could .... Thus $|\mathcal{I}|$ could become super-polynomial. Also, Schrijver's ExchBd subroutine needs to solve the system (17), and this seems to require using multiplication and division. The same objections apply to Orlin's Algorithm, where there is no apparent way to solve (22) fully combinatorially. For these reasons fully combinatorial versions of Schrijver's Algorithm and Orlin's Algorithm appear to be unattainable; however, the Iwata-Orlin Algorithm does not have this problem, and so has a fully combinatorial version that is the fastest known, see Section 3.4.2.

IFF-SP adds new v^j's only at partial Swaps, and only one new v^j at a time. Since there are at most n partial Swaps per Augment, this means that each Augment creates at most n new v^j's. In the strongly polynomial version of the algorithm, each call to Fix calls RefineR $O(\log n)$ times. Each call to RefineR does $O(n^2)$ Augments, and or a total of $O(n^2 \log n)$ Augments for each call to Fix, for a total of $O(n^3 \log n)$ v^j's added in each call to Fix. Each call to Fix starts out with $|\mathcal{I}| = 1$, so $|\mathcal{I}|$ stays bounded by $O(n^3 \log n)$ when we don't use ReduceV.

When we do use ReduceV, the running time for RefineR comes from ($O(n^2)$ calls to Augment) times ($O(n^3\text{EO})$ work from full Swaps between each Augment). This last term comes from ($O(n^2)$ possible boundary triples per vertex) times ($O(n)$ vertices in $\mathcal{I}$) times ($O(\text{EO})$ work per boundary triple).

When we don't use ReduceV, we instead have $O(n^3 \log n)$ vertices in $\mathcal{I}$. Each one again has $O(n^2)$ possible boundary triples, so now the work from full Swaps between each Augment is $O(n^5 \log n \cdot \text{EO})$. Multiplied times the $O(n^2)$ Augments, this gives $O(n^7 \log n \cdot \text{EO})$ as the time for RefineR. Multiplied times the $O(\log n)$ calls to RefineR per call to Fix, and times the $O(n^2)$ calls to Fix overall, we would get a total of $O(n^9 \log^2 n \cdot \text{EO})$ time for the algorithm without calling ReduceV. Thus there is some real hope for making a fully combinatorial version of IFF-SP.

However, getting rid of ReduceV is not sufficient to make IFF-SP fully combinatorial. There is also the matter of the various other multiplications and divisions in IFF-SP. The only non-trivial remaining multiplication in IFF-SP is the term $\lambda_i c(k,l;v^i)$ that arises in Swap. Below we modify the representation (11) by implicitly multiplying through by a common denominator so that each λ_i is an integer bounded by a polynomial in n. Then this product can be dealt

with using repeated addition.

IFF-SP has two non-trivial divisions. One is the computation of δ^0/n^3 in Fix. We change from halving δ at each iteration to doubling a scaling parameter, and we need another factor of n for technical reasons, so we need to compute instead n^4. This can again be done via $O(n)$ repeated additions. The second is the division $x_{kl}/c(k,l;v^i)$ in (21). We would like to simulate this division via repeated subtractions. To do this we need to know that the quotient $x_{kl}/c(k,l;v^i)$ has strongly polynomial size in terms of a scale factor. Here we take advantage of some flexibility in the choice of the step length α. Recall that when the full step length $\lambda_i c(k,l;v^i)$ is "big", we chose to set $\alpha = x_{kl}$. But (with appropriate modification of the update to x) the analysis of the algorithm remains the same for any α satisfying $x_{kl} \leq \alpha \leq \min(x_{kl} + \delta, \lambda_i c(k,l;v^i))$, since for any such value of α x remains δ-feasible and we can still add l to S. Our freedom to choose α in this range gives us enough flexibility to discretize the quotient. The setup of IFF-SP facilitates making such arguments, since it has the explicit bound δ^0 on the components of y available at all times. Indeed, this is essentially what Iwata [46] does.

IFF-FC adapts IFF-SP as follows: We denote corresponding variables in IFF-FC by tildes, so where IFF-SP has x, y, z, λ, δ, etc., IFF-FC has $\tilde{x}$, $\tilde{y}$, $\tilde{z}$, $\tilde{\lambda}$, $\tilde{\delta}$, etc. Since Fix is always working with $\hat{f}_\tau$ defined on $(\mathcal{C} - D_\tau, C)$, we use σ and ρ in place of k and l. Recall from (11) that IFF-SP keeps $y \in B(\hat{f}_\tau)$ as a convex combination of vertices $y = \sum_{i\in\mathcal{I}} \lambda_i v^i$. The λ_i satisfy $\lambda_i \geq 0$ and $\sum_{i\in\mathcal{I}} \lambda_i = 1$, but are otherwise arbitrary. To make the arithmetic discrete in IFF-FC, we keep a scale factor $\mathrm{SF} = 2^a$ (for a a non-negative integer). We now insist that each λ_i be a fraction with integer numerator, and denominator SF. To clear the fractions we represent $\tilde{y}$ as $\mathrm{SF}y \in B(\mathrm{SF}\hat{f})$ and $\tilde{\lambda}_i = \mathrm{SF}\lambda_i$, so that $\tilde{y} = \sum_{i\in\mathcal{I}} \tilde{\lambda}_i v^i$ with each $\tilde{\lambda}_i$ a positive integer, and $\sum_{i\in\mathcal{I}} \tilde{\lambda}_i = \mathrm{SF}$. At the beginning of each call to Fix, as before we choose an arbitrary $\prec_1$ consistent with $\mathcal{D}$ and set $\tilde{y} = v^1$. Thus we choose $a = 0$, $\mathrm{SF} = 2^0 = 1$, and $\tilde{\lambda}_1 = 1$ to satisfy this initially.

IFF-SP starts each call to Fix with $\delta = \delta^0$ and halves it before each call to RefineR. IFF-FC starts with $\tilde{\delta} = (n+1)\delta^0$, and instead of halving it, IFF-FC doubles SF (increases a by 1). This extra factor of $n+1$ is needed to make Lemma 3.10 work, which in turn is needed to make the fully combinatorial discrete approximation of $\tilde{x}_{\sigma\rho}/c(\sigma,\rho;v^i)$ lead to a $\tilde{\delta}$-feasible update to $\tilde{x}$. The proof of Lemma 3.10 also obliges using the explicit method of handling $\mathcal{D}_\tau$, since it needs to know that all vertices generated during RefineR are consistent with $\mathcal{D}_\tau$, and this may not be true with the implicit method.

Lemma 3.10 also needs that $\hat{f}(\mathcal{C})$ is not too negative, which necessitates changing IFF-SP: If $\hat{f}(\mathcal{C}) \leq -\delta^0$ then it is highly negative, and we can call Fix directly on $\hat{f}$ (instead of $\hat{f}_\tau$) to find some $\sigma \in \mathcal{C}$ that is contained in all SFM solutions via Lemma 3.5, and then we add $E(D_\sigma)$ to a set In of elements *in* all SFM solutions. We then delete D_σ from $\mathcal{C}$ and re-set $\hat{f} \leftarrow \hat{f}_\sigma$. This change clearly does not impair the running time of the algorithm. This also means that we need the same sort of bound for $B(\hat{f})$.

Lemma 3.10 *If $\hat{f}(\mathcal{C}) > -\delta^0$, then for any two vertices v^i and v^j of $B(\hat{f}_\tau)$ and any $\sigma \in \mathcal{C} - D_\tau$, $|v^i_\sigma - v^j_\sigma| \leq \tilde{\delta}$. In particular $c(\sigma,\rho;v^i) \leq \tilde{\delta}$ in $B(\hat{f}_\tau)$ (and also $B(\hat{f})$).*

Proof: Note that $c(\sigma,\rho;v^i)$ equals $|v^i_\sigma - v^j_\sigma|$ for the vertex v^j coming from $\prec_i$ with σ and ρ interchanged, so it suffices to prove the first statement. Lemma 2.2 shows that for any y in $B(\hat{f}_\tau)$, in particular $y = v^\prec$, and any $\rho \in \mathcal{C} - D_\tau$, we have $y_\rho \leq \delta^0$. We have that $y(\mathcal{C} - D_\tau) = \hat{f}_\tau(\mathcal{C} - D_\tau) = \hat{f}(\mathcal{C}) - \hat{f}(D_\tau)$. Then $\hat{f}(\mathcal{C}) > -\delta^0$ and $\hat{f}(D_\tau) \leq \sum_{\sigma\in D_\tau} (\hat{f}(D_\sigma) - \hat{f}(D_\sigma - \sigma)) \leq |D_\tau|\delta^0$ imply that $y(\mathcal{C} - D_\tau) \geq -(|D_\tau| + 1)\delta^0$. Adding $-y_\sigma \geq -\delta^0$ to this for all $\sigma \in \mathcal{C} - D_\tau$ other

than ρ implies that $-n\delta^0 \le y_\rho \le \delta^0$ for any $\rho \in \mathcal{C} - D_\tau$. Thus any exchange capacity is at most $(n+1)\delta^0 = \tilde{\delta}$. A simpler version of the same proof works for $B(\hat{f})$. ∎

IFF Fully Combinatorial Algorithm (IFF-FC)

Initialize In $\leftarrow \emptyset$, Out $\leftarrow \emptyset$, $C \leftarrow \emptyset$, $\mathcal{C} \leftarrow E$.
While $|\mathcal{C}| > 1$ do
  Compute $\delta^0 = \max_{\sigma \in \mathcal{C}} \hat{f}(D_\sigma) - \hat{f}(D_\sigma - \sigma)$ and let $\tau \in \mathcal{C}$ attain the maximum.
  If $\delta^0 \le 0$ then return $E -$ Out as an optimal SFM solution.
  If $\hat{f}(\mathcal{C}) \le -\delta^0$
    Set $\mathcal{N} \leftarrow Fix(\hat{f}, (\mathcal{C}, C), \delta^0)$.
    For each $\sigma \in \mathcal{N}$ add $E(D_\sigma)$ to In, and re-set $\mathcal{C} \leftarrow \mathcal{C} - D_\sigma$, $\hat{f} \leftarrow \hat{f}_\sigma$.
  Else ($\delta^0 > 0$ and $\hat{f}(\mathcal{C}) > -\delta^0$)
    Set $\mathcal{N} \leftarrow Fix(\hat{f}_\tau, (\mathcal{C} - D_\tau, C), \delta^0)$.
    If $\mathcal{N} \neq \emptyset$, for each $\sigma \in \mathcal{N}$ add $\tau \to \sigma$ to C, update $\mathcal{C}$, and all D_ρ's, A_ρ's.
    Else ($\mathcal{N} = \emptyset$) set Out $\leftarrow$ Out $\cup\, E(A_\tau)$.
Return whichever of In and $E -$ Out has a smaller function value.

Thus, where IFF-SP kept δ, IFF-FC keeps the pair $\tilde{\delta}$ and SF, which we could translate into IFF-SP terms via $\delta = \tilde{\delta}/\mathrm{SF}$. Also, in IFF-SP δ dynamically changes during Fix, whereas in IFF-FC $\tilde{\delta}$ keeps its initial value and only SF changes. Since $\tilde{y} = \mathrm{SF}y$, we get the effect of scaling by keeping $\tilde{x} = x$ (so that doubling SF makes x half as large relative to y, implying that we do not need to halve the flow $\tilde{x}$ at each call to RefineR), and continue to keep the invariant that $\tilde{z} = \tilde{y} + \partial\tilde{x}$. However, to keep $\tilde{y} = \mathrm{SF}y$ we do need to double y and each $\tilde{\lambda}_i$ when SF doubles.

When IFF-SP chose the step length α, if $x_{\sigma\rho} \ge \lambda_i c(\sigma, \rho; v^i)$, then we chose $\alpha = \lambda_i c(\sigma, \rho; v^i)$ and took a full step. Since this implied replacing v^i by v^j in $\mathcal{I}$ with the same coefficient, we can translate it directly to IFF-FC without harming discreteness. Because both $\tilde{x}$ and $\tilde{\lambda}$ are multiplied by SF, this translates to saying that if $\tilde{x}_{\sigma\rho} \ge \tilde{\lambda}_i c(\sigma, \rho; v^i)$, then we choose $\tilde{\alpha} = \tilde{\lambda}_i c(\sigma, \rho; v^i)$ and take a full step.

In IFF-SP, if $x_{\sigma\rho} < \lambda_i c(\sigma, \rho; v^i)$, then we chose $\alpha = x_{\sigma\rho}$ and took a partial step. This update required computing $x_{\sigma\rho}/c(\sigma, \rho; v^i)$ in (21), which is not allowed in a fully combinatorial algorithm. To keep the translated $\tilde{\lambda}_i$ and $\tilde{\lambda}_j$ integral, we need to compute an integral approximation to $\tilde{x}_{\sigma\rho}/c(\sigma, \rho; v^i)$. To ensure that $\tilde{x}_{\sigma\rho} - \alpha$ hits zero (so that ρ joins S), we need this approximation to be at least as large as $\tilde{x}_{\sigma\rho}/c(\sigma, \rho; v^i)$.

The natural thing to do is to compute $\tilde{\beta} = \lceil x_{\sigma\rho}/c(\sigma, \rho; v^i) \rceil$ and update λ_i and λ_j to $\lambda_i - \tilde{\beta}$ and $\tilde{\beta}$ respectively, which are integers as required. This implies choosing $\tilde{\alpha} = \tilde{\beta} c(\sigma, \rho; v^i)$. Because $\lceil \tilde{x}_{\sigma\rho}/c(\sigma, \rho; v^i) \rceil < (\tilde{x}_{\sigma\rho}/c(\sigma, \rho; v^i)) + 1$, $\tilde{\alpha}$ is less than $c(\sigma, \rho; v^i)$ larger than α. Hence the increase we make to $\tilde{x}_{\rho\sigma}$ to keep the invariant $\tilde{z} = \tilde{y} + \partial\tilde{x}$ is at most $c(\sigma, \rho; v^i)$. By Lemma 3.10, $c(\sigma, \rho; v^i) \le \tilde{\delta}$, so we would have that the updated $\tilde{x}_{\rho\sigma} \le \tilde{\delta}$, so it remains $\tilde{\delta}$-feasible, as desired. Furthermore, we could compute $\tilde{\beta}$ by repeatedly subtracting $c(\sigma, \rho; v^i)$ from $\tilde{x}_{\sigma\rho}$ until we get a non-positive answer. We started from the assumption that $\tilde{x}_{\sigma\rho} < \tilde{\lambda}_i c(\sigma, \rho; v^i)$, or $\tilde{x}_{\sigma\rho}/c(\sigma, \rho; v^i) < \tilde{\lambda}_i$, implying that $\tilde{\beta} \le \tilde{\lambda}_i \le \mathrm{SF}$. Thus the number of subtractions needed is at most SF, which we show below remains small. In fact, we can do better by using repeated doubling: Initialize $q = c(\sigma, \rho; v^i)$ and set $q \leftarrow 2q$ until $q \ge x_{\sigma\rho}$. The number d of doublings is $O(\log \mathrm{SF}) = O(a)$. Along the

way we save $q_i = 2^i q$ for $i = 0, 1, \ldots, d$. Then set $q \leftarrow q_{d-1}$, and for $i = d-2, d-3, \ldots, 0$, if $q + q_i \le x_{\sigma\rho}$ set $q \leftarrow q + q_i$. If the final $q < x_{\sigma\rho}$, set $q \leftarrow q + 1$. Thus the final q is of the form $pc(\sigma, \rho; v^i)$ for some integer p, we have $q \ge x_{\sigma\rho}$, and $(p-1)c(\sigma, \rho; v^i) < x_{\sigma\rho}$. Thus $q = \tilde{\beta}$, and we have computed this in $O(\log \mathrm{SF})$ time.

IFF-FC Subroutine SWAP$(\sigma, \rho; v^i)$

Define $\prec_j$ as $\prec_i$ with σ and ρ interchanged and compute v^j.
If $\tilde{x}_{\sigma\rho} \ge \tilde{\lambda}_i c(\sigma, \rho; v^i)$ [a full SWAP]
  Set $\tilde{\alpha} = \tilde{\lambda}_i c(\sigma, \rho; v^i)$, and $\tilde{x}_{\sigma\rho} \leftarrow \tilde{x}_{\sigma\rho} - \tilde{\alpha}$.
  Set $\mathcal{I} \leftarrow \mathcal{I} + j - i$ and $\tilde{\lambda}_j \leftarrow \tilde{\lambda}_i$.
Else $(\tilde{x}_{\sigma\rho} < \tilde{\lambda}_i c(\sigma, \rho; v^i))$ [a partial SWAP, so at least ρ joins S]
  Compute $\tilde{\beta} = \lceil \tilde{x}_{\sigma\rho} / c(\sigma, \rho; v^i) \rceil$ and $\tilde{\alpha} = \tilde{\beta} c(\sigma, \rho; v^i)$.
  Set $\tilde{x}_{\rho\sigma} \leftarrow \tilde{\alpha} - \tilde{x}_{\sigma\rho}$ and $\tilde{x}_{\sigma\rho} \leftarrow 0$. [makes ∂x_σ drop by $\tilde{\alpha}$ as required]
  Set $\tilde{\lambda}_j \leftarrow \tilde{\beta}$ and $\mathcal{I} \leftarrow \mathcal{I} + j$.
  If $\tilde{\beta} < \tilde{\lambda}_i$ set $\tilde{\lambda}_i \leftarrow \tilde{\lambda}_i - \tilde{\beta}$, else $(\tilde{\beta} = \tilde{\lambda}_i)$ set $\mathcal{I} \leftarrow \mathcal{I} - i$.
Set $\left\{ \begin{array}{l} \tilde{y}_\sigma \leftarrow \tilde{y}_\sigma + \tilde{\alpha} \\ \tilde{y}_\rho \leftarrow \tilde{y}_\rho - \tilde{\alpha} \end{array} \right\}$, and update $R(\delta)$ and S.
For each new member η of S do
  Delete any boundary triples $(\mu, \eta; v^h)$ from B.
  Add any new boundary triples $(\eta, \mu; v^h)$ to B.

Due to choosing the initial value of $\tilde{\delta} = (n+1)\delta^0$ instead of δ^0, we now need to run FIX for $\lceil \log_2((n+1)2n^3) \rceil$ iterations instead of $\lceil \log_2(2n^3) \rceil$, but this is still $O(\log n)$. This implies that SF stays bounded by a polynomial in n, so that the computation of $\tilde{\beta}$ and our simulated multiplications are fully combinatorial operations.

IFF-FC Subroutine FIX$(\hat{f}_\tau, (\mathcal{C} - D_\tau, C), \tilde{\delta})$
Applies to $\hat{f}_\tau$ defined on closed sets of $(\mathcal{C} - D_\tau, C)$, and $c(\sigma, \rho; v^i) \le \tilde{\delta}$ for all $y \in B(\hat{f}_\tau)$

Initialize $\prec$ as any linear order consistent with C, $\tilde{y} \leftarrow v^\prec$, SF $\leftarrow 1$, and $\mathcal{N} = \emptyset$.
Initialize $\tilde{x} = \tilde{\varphi} = 0$ and $\tilde{z} = \tilde{y} + \partial\tilde{x} + \partial\tilde{\varphi}$ $(= \tilde{y})$.
While SF $\le 2n^4$ do
  Set SF $\leftarrow$ 2SF, $y \leftarrow 2y$, and $\tilde{\lambda}_i \leftarrow 2\tilde{\lambda}_i$ for $i \in \mathcal{I}$.
  Call REFINER.
For $\sigma \in \mathcal{C} - D_\tau$ do [add descendants of highly negative nodes to $\mathcal{N}$]
  If $\tilde{w}_\sigma = \tilde{y}_\sigma + \partial\tilde{\varphi}_\sigma < -n^2\tilde{\delta}$ set $\mathcal{N} \leftarrow \mathcal{N} \cup D_\sigma$.
Return $\mathcal{N}$.

From this point the analysis of IFF-FC proceeds just like the analysis of IFF-SP when it doesn't call REDUCEV that we did at the beginning of this section, so we end up with a running time of $O(n^9 \log^2 n \cdot \mathrm{EO})$.

3.3.4 Iwata's Faster Hybrid Algorithms

In [48] Iwata shows a way to adopt some of the ideas behind Schrijver's SFM Algorithm, in particular the idea of modifying the linear orders by general blocks instead of consecutive pairs, to speed up the IFF Algorithm, including the fully combinatorial version of the previous section. The high-level view of the IFF-based algorithms is that they all depend on the $O(n^5\text{EO})$ running time of Refine: The weakly polynomial version embeds this in $O(\log M)$ iterations of a scaling loop; the strongly polynomial version calls Fix $O(n^2)$ times, and each call to Fix requires $O(\log n)$ calls to Refine (actually RefineR). For the fully combinatorial version we need to look more closely at the running time of RefineR. One term in the bottleneck expression determining the running time of RefineR is $|\mathcal{I}|$. Ordinarily we have $|\mathcal{I}| = O(n)$, but in the fully combinatorial version we don't call ReduceV, so $|\mathcal{I}|$ balloons up to $O(n^3 \log n)$. This makes RefineR run a factor of $O(n^2 \log n)$ slower. Otherwise the analysis is the same as for the strongly polynomial version.

Therefore, if we can make Refine run faster, then all three versions should also run faster. One place to look for an improvement is the action that Refine takes when no augmenting path exists: it finds any boundary triple $(k, l; v^i)$ and does a Swap. Potentially a more constrained choice of boundary triple would lead to a faster running time. The Hybrid Algorithm implements this idea in HRefine by adding distance labels as in Schrijver's Algorithm.

But a problem arises with this: the pair of elements (k, l) picked out by distance labels need not be consecutive in $\prec_i$. Schrijver's Algorithm deals with this by using ExchBd to come up with a representation of $\chi_k - \chi_l$ in terms of vertices with smaller $(l, k]_{\prec_j}$. Indeed, all previous non-Ellipsoid SFM algorithms move in $\chi_k - \chi_l$ directions. The Hybrid Algorithm introduces a new idea (originally suggested by Fujishige as a heuristic speedup for IFF): instead of focusing on $\chi_k - \chi_l$, do a BlockSwap (called *Multiple-Exchange* in Iwata [48]) that makes multiple changes to the block $[l, k]_{\prec_i}$ of $\prec_i$ to get a new $\prec_j$ that is much closer to our ideal (of having all elements of the current set of reachable elements appear consecutively at the beginning of $\prec_j$), and then move in direction $v^j - v^i$. Using such directions means that at most one new vertex (namely v^j) needs to be added to $\mathcal{I}$ at each iteration, so the fully combinatorial machinery still works.

By (6), when we generate $\prec_j$ from $\prec_i$ by rearranging some block of b elements, Greedy needs $O(b\text{EO})$ time to compute v^j. For a(n ordinary) Swap, $b = 2$, so it costs only $O(\text{EO})$ time (plus overhead for updating the set of boundary triples). A BlockSwap is more complicated and costs $O(b\text{EO}) \leq O(n\text{EO})$ time. However, we still come out ahead because the sum of these times over all calls to BlockSwap in one call to HRefine is only $O(n^4\text{EO})$, whereas we called Swap $O(n^5)$ times per Refine. This leads to the improved running time of $O(n^4\text{EO})$ for HRefine, exclusive of calls to ReduceV. As with IFF, the Hybrid Algorithm needs to call ReduceV once per Augment, for a total of $O(n^5)$ linear algebra work (which dominates other overhead). Thus the running time of HRefine is $O(n^4\text{EO} + n^5)$, compared to $O(n^5\text{EO})$ for Refine. Since we can safely assume that EO is at least $O(n)$ (because the length of its input is a subset of size $O(n)$), this is a speedup over all three versions of IFF by a factor of $O(n)$.

The top-level parts of the Hybrid Algorithm look much like the IFF Algorithm: We relax $y \in B(f)$ to $z \in B(f + \delta\kappa)$ via flows x in the relaxation network and keeps the invariant $z = y + \partial x$, and we put this into a loop that scales δ. We again define $S^{-\delta}(z) = \{l \in E \mid z_l \leq -\delta\}$, $S^{+\delta}(z) = \{l \in E \mid z_l \geq +\delta\}$, and $R(\delta) = \{k \to l \mid x_{kl} \leq 0\}$. We look for a directed augmenting

Hybrid Outer Scaling Framework

Initialize by choosing $\prec_1$ to be any linear order, $y = v^1$, and $\mathcal{I} = \{1\}$.
Initialize $\delta = |y^-(E)|/n^2$, $x = \varphi = 0$, and $z = w = y$. [$z = y + \partial\varphi + \partial x$ is δ-optimal]
While $\delta \geq 1/n^2$, [when $\delta < 1/n^2$ we are optimal]
    Set $\delta \leftarrow \delta/2$.
    Call HRefineR. [converts 2δ-optimality to δ-optimality]
Return last approximate solution from HRefineR as optimal SFM solution.

path P from $S^{-\delta}(z)$ to $S^{+\delta}(z)$ using only arcs of $R(\delta)$ and then Augment as before.

Since we no longer require consecutive pairs, we now define the set of arcs available for augmenting y to be $A(\mathcal{I}) = \{k \to l \mid \exists i \in \mathcal{I} \text{ s.t. } l \prec_i k\}$ (the same set of arcs as in Schrijver's Algorithm), which includes many more arcs than in IFF. We use distance labels d w.r.t. $A(\mathcal{I})$ in a similar way as in Schrijver's Algorithm: For now we say that d is *valid* if

(IFF i) $d_s = 0$ for all $s \in S^{-\delta}(z)$, and

(IFF ii) $d_l \leq d_k + 1$ for all $k \to l \in A(\mathcal{I})$ (i.e., $l \prec_i k$).

As usual, d_l is a lower bound on the number of arcs in a path in $(E, A(\mathcal{I}))$ from $S^{-\delta}(z)$ to l, so that $d_l = n$ signifies that no such path exists.

Notice that neither of the arc sets $R(\delta)$ and $A(\mathcal{I})$ is necessarily contained in the other. The set S is defined as nodes reachable from $S^{-\delta}(z)$ w.r.t. $R(\delta)$, and so there could be an arc of $A(\mathcal{I})$ exiting S. In this case there must be some boundary triple $(k, l; v^i) \in B$ with $k \to l \in A(\mathcal{I})$ exiting S, and then IFF does Swaps until S has no arcs of $A(\mathcal{I})$ exiting it. Then (18) ensures that S is tight for y. It might take a lot of time to Swap until $B = \emptyset$, and so Hybrid instead iterates only until $d_t = n$ for all $t \notin S$, and then computes the set S' of nodes reachable from $S^{-\delta}(z)$ w.r.t. $A(\mathcal{I})$ as its approximate solution.

However, there is a problem with this strategy because the explicit method (which is needed for a fully combinatorial version of Hybrid) puts infinite bounds on φ on arcs of C. There is nothing to prevent having an arc $t \to s$ of C entering S' with $\varphi_{ts} \gg \delta$ (note that this could not happen with $t \to s$ entering S in IFF, since $\varphi_{ts} \geq \delta$ implies that $\varphi_{ts} \geq x_{st}$, and so FlowSwap would apply and cause t to join S). Such a *rogue* arc would then invalidate the proof of Lemma 3.6 since it depends on knowing that $x_{kl} - \varphi_{lk} > 0$ for all $k \in S$, $l \notin S$, which might no longer be true. This problem causes the argument for the fully combinatorial version of Hybrid in [48] to be incorrect as it stands.

We adapt a fix suggested by Fujishige (personal communication): We change FlowSwap and the definition of validity of d to ensure that no rogue arcs enter S'. Define $C(\delta) = C \cup \{l \to k \mid k \to l \in C \text{ and } \varphi_{kl} \geq \delta\}$. We now say that d is valid if

(IFF′ i) $d_s = 0$ for all $s \in S^{-\delta}(z)$, and

(IFF′ ii) $d_l \leq d_k + 1$ for all $k \to l \in A(\mathcal{I}) \cup C(\delta)$ (consistency implies that if $k \to l \in C$, then $k \to l \in A(\mathcal{I})$, and so validity on arcs of C is automatic).

Then d_l is a lower bound on the number of arcs in a path in $(E, A(\mathcal{I}) \cup C(\delta))$ from $S^{-\delta}(z)$ to l, and $d_l = n$ signifies that no such path exists. We use this modified explicit method throughout our discussion of Hybrid.

When no augmenting path exists, we use a modified version of FLOWSWAP that we call HFLOWSWAP as follows. Suppose that there $s \to t \in C(\delta)$ with $s \in S$ and $t \notin S$. If $s \to t$ corresponds to $t \to s \in C$ with $\varphi_{ts} \geq \delta$, then validity of d for both $s \to t$, $t \to s \in C(\delta)$ says that $|d_t - d_s| \leq 1$, and so HFLOWSWAP can do an ordinary FLOWSWAP to $t \to s$ (that decreases φ_{ts} to drop x_{st} to zero, allowing t to join S), and d remains valid. If instead $s \to t \in C$ and $d_t < d_s - 1$, then an ordinary FLOWSWAP would make d invalid if it caused φ_{st} to become at least δ so that $t \to s$ joins $C(\delta)$. Hence HFLOWSWAP does FLOWSWAP to $s \to t$ (that increases φ_{st} to drop x_{st} to zero, allowing t to join S) only when $d_t \geq d_s - 1$, and d again remains valid.

Re-define S' as the nodes reachable from $S^{-\delta}$ via $A(\mathcal{I}) \cup C(\delta)$. Since no node t with $d_t = n$ is reachable via $A(\mathcal{I}) \cup C(\delta)$, we have $S^{-\delta}(z) \subseteq S' \subseteq S$. Also, S' clearly has no arcs of $A(\mathcal{I})$ exiting it, so we could use S' in place of S in the proof of Lemma 3.6. Unlike IFF, there can be arcs $t \to s \in C$ entering S' (but only if $\varphi_{ts} < \delta$), and arcs $s \to t \in R(\delta)$ exiting S' (but with $x_{st} \geq -\delta$), and so the bound on the gap between $z^-(E)$ and the value of our approximate solution degrades a bit (compare Lemma 3.6 to Lemma 3.11 below), but it is still small enough for the analysis to go through.

When no augmenting path exists, define the set of nodes not in S with minimum distance label as $D = \{l \notin S \mid d_l = \min_{h \notin S} d_h\}$. If no HFLOWSWAP applies, suppose that there is some arc $k \to l \in A(\mathcal{I})$ (so there is some $i \in \mathcal{I}$ with $l \prec_i k$) with $k \in S$, $l \in D$, and $d_l = d_k + 1$. Then we re-define l as the left-most such element in $\prec_i$, and k as the right-most such element in $\prec_i$, and call the triple $(i; k, l)$ *active*. Suppose that $h \prec_i l$ and $h \notin S$. Then $d_h > d_l = d_k + 1$ so that $d_h \geq d_k + 2$, and then $h \prec_i l \prec_i k$ contradicts validity of d. Thus $h \in S$. Also, $h \succ_i k$ implies that $d_h > d_k$. This definition of active is the only delicate lexicographic choice here.

It is a bit tricky to efficiently find active triples. Define a *re-ordering phase* to be the set of BLOCKSWAPs between consecutive calls to RELABEL or AUGMENT. At each re-ordering phase, we SCAN $\prec_i$ for each $i \in \mathcal{I}$ to find out $Left_{ir}$, the left-most element of $\prec_i$ with distance label r, and $Right_{ir}$, the right-most such element. Then, when we look for an active triple $(i; k, l)$ with $d_l = m$, we can restrict our SEARCH to $[Left_{im}, Right_{i,m-1}]$.

Define $S(i; k, l)$ to be the elements in $[l, k]_{\prec_i}$ in S, and $T(i; k, l)$ to be the elements in $[l, k]_{\prec_i}$ not in S, i.e., $S(i; k, l) = \{h \in S \mid l \prec_i h \preceq_i k\}$ and $T(i; k, l) = \{h \notin S \mid l \preceq_i h \prec_i k\}$. Thus $k \in S(i; k, l)$ and $l \in T(i; k, l)$. Define $\prec_j$ to be $\prec_i$ with all elements of $S(i; k, l)$ moved ahead of the elements of $T(i; k, l)$ (without changing the order within $S(i; k, l)$ and $T(i; k, l)$), i.e., just before l. For example (using t_a to denote elements of $T(i; k, l)$ and s_b to denote elements of $S(i; k, l)$), if $\prec_i$ looks like

$$\ldots u_3 u_4 l t_1 t_2 s_1 t_3 t_4 t_5 s_2 s_3 t_6 s_4 k u_5 u_6 \ldots,$$

then $\prec_j$ looks like

$$\ldots u_3 u_4 s_1 s_2 s_3 s_4 k l t_1 t_2 t_3 t_4 t_5 t_6 u_5 u_6 \ldots.$$

Note that this is just a block exchange. Let v^j be the vertex associated with $\prec_j$ by the Greedy Algorithm. By (6), for $b = |[l, k]_{\prec_i}|$, computing v^j costs $O(b\text{EO})$ time. We ideally want to move y in the direction $v^j - v^i$ by replacing the term $\lambda_i v^i$ in (11) by $\lambda_i v^j$. To do this we need to change x to ensure that $z = y + \partial\varphi + \partial x$ is preserved, and so we must find a flow q to subtract from x whose boundary is $v^j - v^i$.

First we determine the sign of $v^i_u - v^j_u$ depending on whether u is in $S(i; k, l)$ or $T(i; k, l)$ (for $u \notin [l, k]_{\prec_i}$ we have $v^i_u - v^j_u = 0$ since $u^{\prec_j} = u^{\prec_i}$). Lemma 2.6 implies that for $s \in S(i; k, l)$, we have $v^j_s \geq v^i_s$, and for $t \in T(i; k, l)$, we have $v^j_t \leq v^i_t$.

Now set up a transportation problem with left nodes $S(i;k,l)$, right nodes $T(i;k,l)$, and all possible arcs. Make the supply at $s \in S(i;k,l)$ equal to $v_s^j - v_s^i \geq 0$, and the demand at $t \in T(i;k,l)$ equal to $v_t^i - v_t^j \geq 0$. Now use, e.g., the Northwest Corner Rule (see Ahuja, Magnanti and Orlin [1]) to find a basic feasible flow $q \geq 0$ in this network. This can be done in $O(|[l,k]_{\prec_i}|) = O(b)$ time, and the number of arcs with $q_{st} > 0$ is also $O(b)$ [1]. Hence computing q and using it to update x takes only $O(b)$ time. Now re-imagining q as a flow in (E,R) we see that $\partial q = v^j - v^i$, as desired.

Hybrid Subroutine BLOCKSWAP$(i;k,l)$
Applies to active triple $(i;k,l)$

Use l and k to compute $S(i;k,l)$, $T(i;k,l)$, $\prec_j$, and v^j.
Set up the transportation network and compute q.
Compute $\eta = \max_{st} q_{st}$ and $\alpha = \min(\lambda_i, \delta/\eta)$. [compute step length, then update]
Set $y \leftarrow y + \alpha(v^j - v^i)$, $\lambda_j \leftarrow \alpha$, and $\mathcal{I} \leftarrow \mathcal{I} + j$.
If $\alpha = \delta/\eta$ then [a partial BLOCKSWAP, so at least t with $q_{st} = \eta$ joins S]
    Set $\lambda_i \leftarrow \lambda_i - \alpha$.
Else ($\alpha = \lambda_i$) [a full BLOCKSWAP, so i leaves $\mathcal{I}$]
    Set $\mathcal{I} \leftarrow \mathcal{I} - i$.
For $s \rightarrow t$ s.t. $q_{st} > 0$, [update x_{st} and x_{ts}]
    If $\alpha q_{st} \leq x_{st}$, set $x_{st} \leftarrow x_{st} - \alpha q_{st}$;
    Else ($\alpha q_{st} > x_{st}$) set $x_{ts} \leftarrow \alpha q_{st} - x_{st}$, and $x_{st} \leftarrow 0$.
Update $R(\delta)$, S, and D.

As with IFF, the capacities of δ on the x's might prevent us from taking the full step from $\lambda_i v^i$ to $\lambda_i v^j$, and modifying x_{st} and x_{ts} by $\lambda_i q_{st}$. So we choose a step length $\alpha \leq \lambda_i$ and investigate constraints on α. If $\alpha q_{st} \leq x_{st}$ then our update is $x_{st} \leftarrow x_{st} - \alpha q_{st}$, which is no problem. If $\alpha q_{st} > x_{st}$ then our update is $x_{ts} \leftarrow \alpha q_{st} - x_{st}$ and $x_{st} \leftarrow 0$, which requires that $\alpha q_{st} - x_{st} \leq \delta$, or $\alpha \leq (\delta + x_{st})/q_{st}$. Since $x_{st} \geq 0$, if we choose $\eta = \max_{st} q_{st}$ and $\alpha = \min(\lambda_i, \delta/\eta)$, then this suffices to keep x feasible.

Since x is changed only on arcs from S to $E - S$, S can only get bigger after BLOCKSWAP (since z doesn't change, neither $S^{+\delta}(z)$ nor $S^{-\delta}(z)$ changes). If $\alpha = \delta/\eta < \lambda_i$, then for the $s \rightarrow t$ such that $\eta = q_{st}$, $\alpha q_{st} = \delta \geq x_{st}$, so the updated x_{st} is zero. Hence $s \rightarrow t$ joins $R(\delta)$ and so t joins S, and we call such a step *partial* (*non-saturating* in [48]). In this case we need to keep both v^j (with coefficient α) and v^i (if $\alpha < \lambda_i$, with coefficient $\lambda_i - \alpha$) in $\mathcal{I}$ so $|\mathcal{I}|$ possibly goes up by one. Otherwise ($\alpha = \lambda_i$), we call the step *full* (*saturating* in [48]). In this case v^j just replaces v^i (with coefficient $\lambda_i = \alpha$) in $\mathcal{I}$ and $|\mathcal{I}|$ stays the same. Since there are at most n partial BLOCKSWAPs before calling AUGMENT, $|\mathcal{I}| \leq 2n$ before calling REDUCEV.

If there are no active triples for the current D and $d_l < n$ for $l \in D$, then HREFINER does a RELABEL that increases d_l by one for all $l \in D$. HREFINER stops when it can no longer find any augmenting paths and $d_l = n$ for $l \in D$ (and so for all $l \notin S$). Note that HREFINER re-computes $S^{-\delta}(z)$, $S^{+\delta}(z)$, S, and D after every AUGMENT, and S and D during BLOCKSWAP, so that S dynamically changes and is not necessarily monotonic.

Note that if $S^{-\delta}(z) = \emptyset$ in HREFINER, then S is always empty in HREFINER, so that everything soon gets RELABELed to n, and $S' = \emptyset$ gets returned as the approximate optimal

solution. If $S^{+\delta}(z) = \emptyset$, then AUGMENT never gets called in HREFINER, but the algorithm proceeds normally anyway, possibly by eventually adding all elements to S and returning $S' = E$ as its approximate optimal solution.

Hybrid Subroutine HREFINER

Initialize $d = 0$, $x \leftarrow x/2$, $\varphi \leftarrow \varphi/2$, and update z.
Compute $S^{-\delta}(z)$, $S^{+\delta}(z)$, and S.
While augmenting paths exist ($S \cap S^{+\delta}(z) \neq \emptyset$), or $\exists l \notin S$ with $d_l < n$ do
  If $\exists$ path P from $S^{-\delta}(z)$ to $S^{+\delta}(z)$ using arcs from $R(\delta)$, do
    AUGMENT(P), REDUCEV, update $S^{-\delta}(z)$, $S^{+\delta}(z)$, S, and D, SCAN.
  Else ($\nexists$ such a path, but $l \in D$ have $d_l < n$) if HFLOWSWAP applies, call it.
  Else (HFLOWSWAP does not apply) do
    SEARCH for an active triple.
    If $\exists$ an active triple $(i; k, l)$, BLOCKSWAP$(i; k, l)$.
    Else (no active triple) RELABEL: $d_l \leftarrow d_l + 1$ for all $l \in D$, update D, SCAN.
[Now no augmenting paths exist and $d_l = n$ for all $l \notin S$]
Compute $S' =$ nodes reachable from $S^{-\delta}(z)$ via arcs of $A(\mathcal{I}) \cup C(\delta)$.
Return S' as an approximate optimum solution.

Recall that $w = y + \partial\varphi \in B(f)$, and that our optimality condition for S' solving SFM is that $w^-(E) = f(S')$. The following lemma shows for both w and z how close these approximate solutions are to exactly satisfying $w^-(E) = f(S')$ and $z^-(E) = f(S')$ at the end of HREFINER.

Lemma 3.11 *When* HREFINER *ends, S' is tight for y, and we have $w^-(E) \geq f(S') - n^2\delta$ and $z^-(E) \geq f(S') - n(n+1)\delta/2$.*

Proof: We already noted that S' is tight for y, and that (since no augmenting path exists) $S^{-\delta}(z) \subseteq S' \subseteq S \subseteq E - S^{+\delta}(z)$. Similar to the proof of Lemma 3.3 this implies that $z^-(E) > z(S') - n\delta = y(S') + \partial\varphi(S') + \partial x(S') - n\delta = f(S') + \partial x(S') + \partial x(S') - n\delta$.

Since no arcs of $C(\delta)$ exit S' we have that $\partial\varphi(S') > -\delta|S'| \cdot |E - S'| \geq -n^2\delta/4$. Similarly the upper bound of δ on x implies that $\partial x(S') \geq -n^2\delta/4$. This yields $z^-(E) > f(S') - (n + n^2/2)\delta$. Since $w = z + \partial x$, for any $u \in E$ w_u can be at most ∂x_u lower than z_u. Thus the at most $n-1$ arcs $t \to u$ can decrease w_u at most $(n-1)\delta$ below z_u. Furthermore, since $x_{ut} \cdot x_{tu} = 0$, each x_{ut}, x_{tu} pair decreases at most one of w_u and w_t. Thus the total amount by which $w^-(E)$ is smaller than $z^-(E)$ is at most $n(n-1)\delta/2$. Thus we get $w^-(E) > f(S') - n(n+1)\delta/2 - n(n-1)\delta/2 = f(S') - n^2\delta$. ∎

We now use this to prove correctness and running time, as well as consistency for later FC use. As before we pick out the main points in boldface.

Theorem 3.12 *The Hybrid SFM Algorithm generates only consistent vertices, is correct for integral data, and runs in $O((n^4\mathrm{EO} + n^5)\log M)$ time.*

Proof:

The current approximate solution S at the end of a δ-scaling phase with $\delta < 1/n^2$ solves SFM: Lemma 3.11 shows that $w^-(E) \geq f(S) - n^2\delta > f(S) - 1$. But for any $U \subseteq E$, $f(U) \geq w(U) \geq w^-(E) > f(T) - 1$. Since f is integer-valued, T solves SFM.

Distance labels remain valid throughout HRefineR**:** Only Augment changes z, and in such a way that $S^{-\delta}(z)$ only gets smaller. Hence $d_s = 0$ on $S^{-\delta}(z)$ is preserved. We already remarked that HFlowSwap preserves validity on $C(\delta)$. BlockSwap$(i;k,l)$ adds new vertex v^j to $\mathcal{I}$. The only new pairs with $s \prec_j t$ but $s \not\prec_i t$ (that might violate validity) are those with $s \in S(i;k,l)$ and $t \in T(i;k,l)$, and for these we need that $d_s \leq d_t + 1$. Validity applied to $s \preceq_i k$ implies that $d_s \leq d_k + 1 = d_l$. By definition of D, $d_l \leq d_t$, so $d_s \leq d_l \leq d_t < d_t + 1$. Suppose that Relabel made $s \to t \in C(\delta)$ invalid. This means that before the Relabel we had $s \in S$ and $t \in D$ with $d_t = d_s + 1$. But this s, t pair would then define an active triple, contradicting that we are doing a Relabel. Similarly Relabel cannot invalidate any arc of $A(\mathcal{I})$.

Only consistent vertices are generated: Suppose that $s \to t \in C$ and we had $t \prec_i s$ but that BlockSwap uses $\prec_i$ to generate $\prec_j$ with $s \prec_j t$, so that $s \in S(i;k,l)$ and $t \in T(i;k,l)$. The previous paragraph showed that at this point $|d_s - d_t| \leq 1$, and so HFlowSwap would apply to $s \to t$, causing t to join S, contradicting that $t \notin S$.

Each scaling phase calls Augment $O(n^2)$ **times:** At the beginning of HRefineR, for X equal to the final S' from the previous call to HRefineR, Lemma 3.11 shows that $z^-(E) > f(X) - n(n+1)\delta$. This is also true for the first call to HRefineR for $X = \emptyset$ by the choice of the initial value of $|y^-(E)|/n^2 = |z^-(E)|/n^2$ for δ. From the upper bound of δ on x we have $z^-(E) \leq w^-(E) + n^2\delta$. Since $w \in B(f)$ we have $w^-(E) \leq f(X)$, and so $z^-(E) \leq f(X) + n^2\delta$. Thus the total rise in value for $z^-(E)$ during HRefineR is at most $(2n^2 + n)\delta$. Each call to Augment increases $z^-(E)$ by δ, so there are $O(n^2)$ calls to Augment.

There are $O(n^2)$ calls to Relabel **during** HRefineR**:** Each d_k is between 0 and n and never decreases during HRefineR. Each Relabel increases at least one d_k by one, so there are $O(n^2)$ Relabels.

The previous two paragraphs establish that **there are $O(n^2)$ re-ordering phases**.

The common value m of d_l for $l \in D$ is non-decreasing during a re-ordering phase: During a re-ordering phase, d does not change but S, D and $R(\delta)$ do change. However, all arcs where x changes, and hence where $R(\delta)$ can change, are between $S(i;k,l)$ and $T(i;k,l)$. Thus S can only get larger during a re-ordering phase, and so m is non-decreasing in a phase.

The work done by all BlockSwaps **during a re-ordering phase is $O(n^2\text{EO})$:** Suppose that BlockSwap$(i;k,l)$ adds v^j to $\mathcal{I}$. Then, by how k and l were defined in an active triple, for any q with $d_q = d_l$, any p with $d_q = d_p + 1$ must have that $p \prec_j q$, and hence there can be no subsequent active triple $(j;p,q)$ in the phase with $d_q = d_l$. Thus m must increase by at least one before the phase uses a subsequent active triple $(j;p,q)$ involving $\prec_j$. But then $d_q > d_l = d_k + 1$, implying that we must have that $l \prec_i k \prec_i q \prec_i p$. Hence if v^j results from v^i via BlockSwap$(i;k,l)$, and $(j;p,q)$ is the next active triple at j in the same re-ordering phase, it must be that $[l,k]_{\prec_i}$ is disjoint from $[q,p]_{\prec_i}$.

Suppose that $\prec_j$ appears in $\mathcal{I}$ at some point during a re-ordering phase, having been derived by a sequence of BlockSwaps starting with $\prec_{i_1}$ (which belonged to $\mathcal{I}$ at the beginning of the phase), applying active triple $(i_1;k_1,l_1)$ to $\prec_{i_1}$ to get $\prec_{i_2}$, applying active triple $(i_2;k_2,l_2)$ to $\prec_{i_2}$ to get $\prec_{i_3}$, ..., and applying active triple $(i_a;k_a,l_a)$ to $\prec_{i_a}$ to get $\prec_{i_{a+1}}=\prec_j$. Continuing the argument in the previous paragraph, we must have that $l_1 \prec_{i_1} k_1 \prec_{i_1} l_2 \prec_{i_1} k_2 \prec_{i_1} \ldots \prec_{i_1} l_a \prec_{i_1} k_a$. Thus the sum of the sizes of the intervals $[l_1,k_1]_{\prec_{i_1}}$, $[l_2,k_2]_{\prec_{i_1}}$, ..., $[l_a,k_a]_{\prec_{i_1}}$ is $O(n)$. We count all these BlockSwaps as belonging to $\prec_j$, so the total BlockSwap work attributable

to $\prec_j$ is $O(n\text{EO})$. Since $|\mathcal{I}| = O(n)$, the total work during a re-ordering phase is $O(n^2\text{EO})$.

The time for one call to HRefineR **is** $O(n^4\text{EO}+n^5)$**:** The bottleneck in calling Augment is the call to ReduceV, which costs $O(n^3)$ time. There are $O(n^2)$ calls to Augment, for a total of $O(n^5)$ ReduceV work during HRefineR. There are $O(n^2)$ re-ordering phases during HRefineR, so Scan is called $O(n^2)$ times. The BlockSwaps during a phase cost $O(n^2\text{EO})$ time, for a total of $O(n^4\text{EO})$ BlockSwap work in one call to HRefineR. Each call to Scan costs $O(n^2)$ time, for a total of $O(n^4)$ work per HRefineR. As in the previous paragraph, the intervals $[Left_{im}, Right_{i,m-1}]$ are disjoint in $\prec_i$, so the total Search work for $\prec_i$ is $O(n)$, or a total of $O(n^2)$ per phase, or $O(n^4)$ work over all phases. The updates to S and D cost $O(n)$ work per phase, or $O(n^3)$ overall.

There are $O(\log M)$ **calls to** HRefineR**:** As in the proof of Theorem 3.4 the initial $\hat{\delta} = |y^-(E)|/n^2 \leq 2M/n^2$. Each call to HRefineR cuts δ in half, and we terminate when $\delta < 1/n^2$, so there are $O(\log M)$ calls to HRefineR.

The total running time of the algorithm is $O((n^4\text{EO}+n^5)\log M)$**:** Multiplying together the factors from the last two paragraphs gives the claimed total time. ∎

We already specified HRefineR so that it optimizes over a ring family, and this suffices to embed HRefineR into the strongly polynomial framework of Section 3.3.2, getting a running time of $O((n^4\text{EO} + n^5)n^2 \log n)$.

Making the Hybrid Algorithm fully combinatorial is similar to the ideas in Section 3.3.3. The ratio δ/η in BlockSwap is handled in the same way as the ratio $x_{kl}/c(k,l;v^i)$ in (21) of IFF-FC. If $\tilde{\lambda}_i q_{st} \leq \text{SF}\tilde{x}_{st}$ for all $s \to t$ (where SF is the current scale factor), then we can do a full BlockSwap as before. Otherwise we use binary search to compute the minimum integer $\tilde{\beta}$ such that there is some $s \to t$ with $\tilde{\beta} q_{st} \geq \text{SF}\tilde{x}_{st}$. We then update $\tilde{\lambda}_i \leftarrow \tilde{\lambda}_i - \tilde{\beta}$ and $\tilde{\lambda}_j \leftarrow \tilde{\beta}$. Since $\tilde{\beta} = \lceil \text{SF}\tilde{x}_{st}/q_{st} \rceil$, the increase in $\tilde{\beta}$ over the usual value $\text{SF}\tilde{x}_{st}/q_{st}$ is at most 1, so the change in $\partial\tilde{x}_s$ is at most $q_{st} \leq v_s^j - v_s^i \leq \tilde{\delta}$ by Lemma 3.10 (which requires consistent vertices, which we proved above; this is why we need the explicit method here), so the update keeps $\tilde{x}$ $\tilde{\delta}$-feasible. We started from the assumption that there is some $s \to t$ with $\tilde{\lambda}_i q_{st} > \text{SF}\tilde{x}_{st}$, implying that $\tilde{\beta} \leq \tilde{\lambda}_i \leq \text{SF}$, so this binary search is fully combinatorial.

The running time of all versions of the algorithm depends on the $O(n^4\text{EO} + n^5)$ time for HRefineR, which comes from $O(n^2)$ re-ordering phases times $O(n^2\text{EO})$ BlockSwap work plus $O(n^3)$ ReduceV work in each re-ordering phase. The $O(n^2\text{EO})$ BlockSwap work in each re-ordering phase comes from $O(n\text{EO})$ BlockSwap work attributable to each $\prec_i$ in $\mathcal{I}$ times the $O(n)$ size of $\mathcal{I}$. Since $|\mathcal{I}|$ is larger by a factor of $O(n^2 \log n)$ when we don't call ReduceV (it grows from $O(n)$ to $O(n^3 \log n)$), we might expect that the fully combinatorial running time also grows by a factor of $O(n^2 \log n)$, from $O((n^6\text{EO} + n^7)\log n)$ to $O((n^8\text{EO} + n^9)\log^2 n)$. However, the term $O(n^9)$ comes only from the $O(n^3)$ ReduceV work per re-ordering phase: The Scan and Search time in a re-ordering phase is only $O(n^2)$, which is dominated by the BlockSwap work. Thus, since the fully combinatorial version avoids calling ReduceV, the total time is only $O(n^8\text{EO}\log^2 n)$. (The careful implementation of Scan and Search are needed to avoid the extra term of $O(n^9 \log^2 n)$, and this is original to this survey.)

3.4 Orlin-Type SFM Algorithms

Orlin [71] proposed an SFM algorithm in 2006 that was quite different from previous algorithms, in part because it does not use network flow ideas. In 2009 Iwata and Orlin [52] re-used some of

the same ideas in a simpler SFM algorithm that is nearly as fast, and whose fully combinatorial version is the current fastest in that class.

As before, define $S^-(y) = \{e \in E \mid y_e < 0\}$, $S^0(y) = \{e \in E \mid y_e = 0\}$, and $S^+(y) = \{e \in E \mid y_e > 0\}$. Notice that if $S^-(y) = \emptyset$, then $\emptyset$ solves SFM, and if $S^+(y) = \emptyset$, then E solves SFM.

These algorithms do not explicitly reference arcs or a network. Instead they use distance labels that implicitly define the same network as in Schrivjer's Algorithm. Unlike previous distance labels, here each linear order $\prec_i$ has its own distance labels d^i. The label d^i_e of $e \in E$ is an estimate of how far to the right e should appear in $\prec_i$, where a label of zero means that e should be at the left, and $n+1$ means that it should be at the right. As the algorithms proceed they will maintain the following *validity* properties between y, the $\prec_i$ and the d^i:

(OT i) For each $i \in \mathcal{I}$, if $y_e \leq 0$, then $d^i_e = 0$.

(OT ii) For each $i \in \mathcal{I}$, if $e \prec_i g$, then $d^i_e \leq d^i_g$.

(OT iii) For each $i, j \in \mathcal{I}$ and each $e \in E$, $|d^i_e - d^j_e| \leq 1$.

(In [71] the $\prec_i$ are derived from the d^i essentially using property (OT ii), but it can be seen that this definition is equivalent.) Note that (OT i) tends to force $e \in S^-(y)$ to the left of each $\prec_i$, as required by optimality. As the $e \in S^-(y)$ move leftwards, by (1) they increase towards zero (and can become positive), and so the algorithms do increase $y^-(E)$ over time. For $e \in E$ define $d^{\min}_e = \min_{i \in \mathcal{I}} d^i_e$.

As the algorithms proceed, sometimes they will add new linear orders via doing block exchanges on existing linear orders, and the label vectors of the new linear orders match those of the existing ones, except that the labels of some subset of elements is increased by one. We need to ensure that label values do not become too large. Note that if some d^i_e reaches value $n+1$, then by (OT iii) there is some $j \in \mathcal{I}$ with $d^j_e = d^{\min}_e \geq n$, and then by the pigeon-hole principle there is some $k \in (0, n)$ such that there is some e with $d^{\min}_e = k$, but no g with $d^{\min}_g = k - 1$. We call such a k a *distance gap*.

If k is a distance gap, define $T = \{e \mid d^{\min}_e < k\}$, and note that by (OT ii) T is at the left of every $\prec_i$. By (OT i), $y_e \geq 0$ for $e \notin T$. We say that a distance gap satisfies *partial optimality* (because if we also had that $y_e \leq 0$ for $e \in T$, then it is easy to see that this would prove that T is optimal).

Lemma 3.13 *There is an optimal SFM solution S with $S \subseteq T$. If we also know that $y_e > 0$ for $e \notin T$, then $S \subseteq T$ for* every *SFM solution S.*

Proof: By (OT i) and (5), $v^i(T) = f(T)$ for all $i \in \mathcal{I}$, and so $y(T) = f(T)$. Let S^* solve SFM. By (OT i), $S^-(y) \subseteq T$, and so $f(T) = y(T) \leq y(T \cup S^*) \leq f(T \cup S^*)$. Thus $f(T) \leq f(T \cup S^*)$, and so (2) implies that $f(T \cap S^*) \leq f(S^*)$, showing that $T \cap S^* \subseteq T$ also solves SFM.

Now assume that $y_e > 0$ for $e \notin T$. Suppose that S solves SFM but that $S - T \neq \emptyset$. Then $f(T) = y(T) < y(T) + y(S - T) = y(S) \leq f(S)$, contradicting that S solves SFM, so we must have $S \subseteq T$. ∎

This lemma justifies the algorithms in setting $d^i_e = n+1$ for all $i \in T$ as a way of essentially deleting the elements of T from further consideration. This action of deleting elements above distance gaps also ensures that labels stay at most $n+1$. It is possible for $k = 1$ to be a distance gap when $d^{\min}_e \geq 1$ for all $e \in E$ ($\Rightarrow S^-(y) = \emptyset$), and in this case $T = \emptyset$, which solves SFM. The algorithms recognize this. Hence while the algorithms are running, there is always some e with $d^{\min}_e = 0$.

3.4.1 Orlin's SFM Algorithm

Orlin's Algorithm appeared in 2006 [71], and is currently the fastest known strongly polynomial SFM algorithm (faster even than Ellipsoid). The algorithm assumes that f is defined on 2^E, but it can be adapted to ring families using the method of Section 5.2. It is somewhat reminiscent of Schrijver's Algorithm as it also considers directions synthesized from multiple new vertices defined by single element block changes to existing vertices. However, unlike Schrijver, Orlin does not use these new vertices to synthesize an edge direction, but rather synthesizes a direction that keeps $S^0(y)$ monotonically non-decreasing. Also, where Schrijver solves triangular system (17) to synthesize a direction, Orlin solves a more general system that involves an M-matrix (see Berman and Plemmons [5]).

For each $e \in E$ define $d^{[e]}$ (the *primary distance vector* for e) to be a d^i with $d^i_e = d^{\min}_e$, and with minimum $d^i(E)$ value among these. That is, let's define $H^{[e]}$ as the set of $i \in \mathcal{I}$ that lexicographically minimize $(d^i_e, d^i(E))$, and then $d^{[e]} = d^i$ for some $i \in H^{[e]}$. Denote the linear order associated with $d^{[e]}$ by $\prec^{[e]}$. Define $d^{[e+]}$ (the *secondary distance vector* for e, which is usually not one of the d^i) via $d^{[e+]}_g = d^{[e]}_g$ for $g \neq e$, and $d^{[e+]}_e = d^{[e]}_e + 1$. The linear order associated with $d^{[e+]}$ is $\prec^{[e+]}$, which moves e to the right in $\prec$ just until it is larger than every g with $d^{[e]}_g = d^{[e]}_e$, so this is a block exchange. Notice that this construction for $d^{[e+]}$ and $\prec^{[e+]}$ preserves (OT ii) and (OT iii). The associated vertices are $v^{[e]}$ and $v^{[e+]}$. A subset of the $v^{[e+]}$ are the new vertices the algorithm adds to $\mathcal{I}$.

For $e \in E$ define the column $A(e) \in \mathbb{R}^E$ by $A(e) = v^{[e+]} - v^{[e]}$, and its restriction to the components of $S^0(y)$ by $A^0(e)$. Then if $S^0(y) \neq \emptyset$ the $|S^0(y)| \times |S^0(y)|$ *auxiliary matrix* A^* contains the columns $A^0(e)$ for $e \in S^0(y)$. Enumerate the elements of $S^0(y)$ as $e_1, e_2, \ldots, e_q$. Then Lemma 2.6 implies that the sign pattern of A^* looks like:

$$\begin{array}{c} \\ e_1 \\ e_2 \\ e_3 \\ \vdots \\ e_q \end{array} \begin{array}{c} \begin{array}{ccccc} A^0(e_1) & A^0(e_2) & A^0(e_3) & \ldots & A^0(e_q) \end{array} \\ \begin{pmatrix} \ominus & \oplus & \oplus & \ldots & \oplus \\ \oplus & \ominus & \oplus & \ldots & \oplus \\ \oplus & \oplus & \ominus & \ldots & \oplus \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \oplus & \oplus & \oplus & \ldots & \ominus \end{pmatrix} \end{array}.$$

Lemma 3.14 *If A^* is invertible, then $(A^*)^{-1} \leq 0$, and if A^* is singular, then there exists some $\gamma \in \mathbb{R}^{S^0(y)}$, $\gamma \neq 0$, $\gamma \geq 0$, with $A^*\gamma = 0$.*

Proof: Recall that $v^{[e]}(E) = v^{[e+]}(E) = f(E)$. For $e \in S^0(y)$, $A^0(e)$ includes $A(e)_e$, the only possibly negative component, and just a subset of the possibly positive components, and so we get that $\mathbb{1}^T A^0(e) \leq 0$. When A^* is invertible, these facts show that $-A^*$ is an M-matrix. The conclusions then follow from [5] and [71, Theorem 2]. ∎

Now the idea is to choose some $h \in S^+(y)$. In order to get all the $S^-(y)$ elements towards the left, we would like move the labels of h and all the $S^0(y)$ elements to the right, so we'd like to replace $d^{[g]}$ with $d^{[g+]}$ for $g \in S^0(y) + h$. Adding a positive multiple of $A(g)$ to y (adding a new index to $\mathcal{I}$ for $d^{[g+]}$ if necessary) would do this, and would keep $\sum_{i \in \mathcal{I}} \lambda_i = 1$. At the same time, we want to ensure that no $e \in S^0(y)$ leaves $S^0(y)$. Lemma 3.14 gives us tools to help us synthesize a good direction.

If A^* is non-singular, then we can solve $A^*\gamma = -A(h)$ for γ. Lemma 2.6 says that since $h \notin S^0(y)$, $-A^0(h) \leq 0$, and $(A^*)^{-1} \leq 0$, so that the resulting $\gamma \geq 0$. If instead A^* is singular, then Lemma 3.14 ensures that some $\gamma \geq 0$ with $\gamma \neq 0$ exists such that $\gamma A^* = 0$. In the first case we set $\gamma_h = 1$, and in the second case we set $\gamma_h = 0$. We combine both cases by finding some

$$\gamma \geq 0,\ \gamma \neq 0,\ \text{solving} \sum_{g \in S^0(y)+h} \gamma_g A^0(g) = 0. \tag{22}$$

In the degenerate case where $S^0(y) = \emptyset$ we put $\gamma_h = 1$, which has all the properties needed for the algorithm. Hence in all cases, moving in direction $d \equiv \sum_{g \in S^0(y)+h} \gamma_g A(g)$ indeed preserves that $y_e = 0$ for $e \in S^0(y)$ and $\sum_i \lambda_i = 1$, while tending to replace $d^{[g]}$ with $d^{[g+]}$ as desired.

Let α denote the step length of our move, so that we change y to $y + \alpha d$. Note that $\mathcal{I}$ expands to include new indices for $v^{[g+]}$ for each $g \in S^0(y) + h$. Suppose that there was some $e, g \in E$ and $i \in H^{[e]}$ such that $d^i = d^{[g+]}$. Then we could choose $d^{[e]} = d^i$, and we'd have $d^{[e]}_g = d^{[g+]}_g = d^{[g]}_g + 1$, and $d^{[g]}_j = d^{[g+]}_j = d^{[e]}_j$ for $j \neq g$. But then $d^{[g]}(E) < d^{[e]}(E)$, but both have that $d^{[g]}_e = d^{[e]}_e = d^{\min}_e$, and this contradicts that $d^{[e]}$ has the min value of $d^{[e]}(E)$ over all d^i with $d^{[e]}_e = d^{\min}_e$. Hence

> For all $g \in E$, if there is some $i \in \mathcal{I}$ with $d^i = d^{[g+]}$, then i is not in $H^{[e]}$ for any $e \in E$. (23)

Thus for i in the new $\mathcal{I}$ such that there is some $g \in S^0(y) + h$ with $d^i = d^{[g+]}$, there can be no $e \in S^0(y) + h$ with $d^i = d^{[e]}$, and so the new λ_i is $\lambda_i + \alpha \sum_{g: d^{[g+]} = d^i} \gamma_g$; for i in the new $\mathcal{I}$ such that there is some $g \in S^0(y) + h$ with $d^i = d^g$, there can be no $e \in S^0(y) + h$ with $d^i = d^{[e+]}$, and so the new λ_i is $\lambda_i - \alpha \sum_{g: d^{[g]} = d^i} \gamma_g$. Since $\gamma \geq 0$, this ensures that this change causes only λ_i with $d^i = d^{[e]}$ (and not those with $d^i = d^{[e+]}$) to drop towards zero, which is important for the running time proof. We continue moving in direction d until either some λ_i drops to zero for $i \in \mathcal{I}$, or y_e drops to zero for some e that was in $S^+(y)$ (since $\gamma \neq 0$, at least one of these cases must happen). This ensures that we keep $\lambda \geq 0$ and that no new elements join $S^-(y)$.

Once we take this step, we now call subroutine UpdateI, which removes any i with $\lambda_i = 0$, and then recomputes the $d^{[e]}$, $d^{[e+]}$, $S^0(y)$, and $S^+(y)$. It chooses some $h \in S^+(y)$ and for $e \in S^0(y) + h$ such that $d^{[e]}$ has changed, adds $d^{[e+]}$ to $\mathcal{I}$ and computes $A(e)$.

Note that if an exact primal solution is desired, then instead of deleting all elements above a distance gap, one could set their labels to $n+1$ in all d^i to ensure that they are right-most in all d^i and modify the selection of $h \in S^+(y)$ to ensure that $d^i_h \leq n$, and then the algorithm ends with an exact primal solution.

We now prove that this works, and give its running time. We again give one big proof, but we pick out the key claims along the way in boldface.

Theorem 3.15 *Orlin's Algorithm correctly solves SFM, and runs in $O(n^5\text{EO} + n^6)$ time.*

Proof: **If the algorithm terminates, it is with an optimal solution:** It only terminates when $S^+(y) = \emptyset$, and then the remaining E is clearly optimal.

If $y_e = 0$ at some iteration, then $y_e = 0$ at all later iterations: The choice of d and α ensures this.

Orlin's Algorithm for SFM

Initialize by setting $d^1 = 0$, $\prec_1 = 12\ldots n$, $\lambda_1 = 1$, $\mathcal{I} = \{1\}$, and computing $y = v^1$.
While $S^+(y) \neq \emptyset$ do
  Choose some $h \in S^+(y)$. [try to replace $d^{[h]}$ by $d^{[h+]}$]
  If $S^0(y) = \emptyset$, put $\gamma_h = 1$;
  Else compute $\gamma \geq 0$, $\gamma \neq 0$ s.t. $\sum_{e \in S^0(y)+h} \gamma_e A^0(e) = 0$. [(22), Lemma 3.14]
  Compute $d = \sum_{e \in S^0(y)+h} \gamma_e A(e)$.
  Set α the max possible s.t. $y_e + \alpha d_e \geq 0$ $\forall e \in S^+(y)$ and $\alpha \sum_{e:d^{[e]}=d^i} \gamma_e \leq \lambda_i$ $\forall i \in \mathcal{I}$.
  Set $y = y + \alpha d$, add indices for $d^{[g+]}$ for $g \in S^0(y) + h$ to $\mathcal{I}$.
  For $i \in \mathcal{I}$, set $\lambda_i = \lambda_i + \alpha(\sum_{g:d^{[g+]}=d^i} \gamma_g - \sum_{g:d^{[g]}=d^i} \gamma_g)$.
  Call UpdateI.
  If $|\mathcal{I}| \geq 3n$ then call ReduceV and UpdateI.
  If k is a distance gap, then set $E = \{e \mid d_e^{\min} < k\}$, recompute $S^-(y)$, $S^0(y)$, and $S^+(y)$.
Return E as an optimal solution. [E is optimal as $S^+(y) = \emptyset$]

For all $e \in E$, $d_e^{\min}$ is non-decreasing: Dropping i from $\mathcal{I}$ via ReduceV cannot decrease a $d_e^{\min}$. Since any new $d^{[g+]}$ added to $\mathcal{I}$ has $d_e^{[g+]} \geq d_e^{[g]}$ for the $d^{[g]}$ already in $\mathcal{I}$, this also cannot decrease a $d_e^{\min}$.

The algorithm preserves validity and representation (11): If $e \in S^-(y)$ then the algorithm never considers $d^{[e+]}$, and so $d_e^i = 0$ is preserved, i.e., (OT i). We already noted that (OT ii) and (OT iii) are preserved by the definitions of $d^{[e+]}$ and $\prec^{[e+]}$. Since each $A(e) = v^{[e+]} - v^{[e]}$ and d is a linear combination of $A(e)$'s, moving along d preserves that $\sum_i \lambda_i = 1$. The choice of α preserves that $\lambda \geq 0$.

Defining the potential function: For $e \in E$ define $\Phi(e) = |H^{[e]}|$, and $\Phi = \sum_{e \in E} \Phi(e)$. The total decrease (increase) in Φ is the sum of the decreases (increases) at steps where Φ decreases (increases). The eventual goal is to show that the total decrease in Φ is $O(n^4)$. To derive this, define $h(e) = d^{[e]}(E)$ and $\hat{h}(e) = \sum_{g \in E}(d_g^{[e]} - d_g^{\min})$. We first show that the number of changes in $(d_e^{\min}, h(e))$ (which is the lexicographic value for $i \in H^{[e]}$) is $O(n^2)$; to show this we consider changes in $\hat{h}(e)$.

The number of times that $(d_e^{\min}, h(e))$ changes is $O(n^2)$: Since $d_e^{\min}$ is non-decreasing and $d_e^{\min} \leq n$, the number of times that $d_e^{\min}$ changes without a change in $h(e)$ is $O(n)$, so we concentrate on changes in $h(e)$. Note that $h(e) - \hat{h}(e) = \sum_{g \in E} d_g^{\min}$. It is possible for $h(e)$ to change while $\hat{h}(e)$ stays constant, as long as $d_g^{\min}$ increases for some g. The number of changes in $\sum_{g \in E} d_g^{\min}$ is $O(n^2)$, and so if we show that there are $O(n^2)$ changes in $\hat{h}(e)$, this also shows $O(n^2)$ changes for $h(e)$. By removing distance gaps, $0 \leq \hat{h}(e) \leq n$, and so total increases in $\hat{h}(e)$ are bounded by total decreases plus n. The only way for $\hat{h}(e)$ to decrease is if some $d_g^{\min}$ increases. There are only $O(n^2)$ increases of $d_g^{\min}$ over all g, and so only $O(n^2)$ decreases to $\hat{h}(e)$. Hence total changes to $\hat{h}(e)$, and so also $h(e)$, and so also $(d_e^{\min}, h(e))$, are $O(n^2)$.

The total increase and decrease in Φ is $O(n^4)$: Since $\Phi = O(n^2)$, if suffices to bound the total increase in Φ by $O(n^4)$. The only new i added to $\mathcal{I}$ are those with $d^i = d^{[g+]}$ for some $g \in S^0(y) + h$, and by (23) these do not change any $H^{[e]}$. Hence, during a sequence of iterations where $(d_e^{\min}, h(e))$ is constant, $\Phi(e)$ can only decrease (due to λ_i hitting zero for some i with

$d^i = d^{[e]}$, or ReduceV). When $(d_e^{\min}, h(e))$ changes, the new $H^{[e]}$ contains at most $|\mathcal{I}| = O(n)$ indices, and so $\Phi(e)$ increases by $O(n)$. For a fixed e, $(d_e^{\min}, h(e))$ changes $O(n^2)$ times, and so the total increase to $\Phi(e)$ over all iterations is $O(n^3)$. Thus the total increase in Φ over all iterations is $O(n^4)$. Note that this (and all previous paragraphs) remains true when the distance gap step deletes some elements.

The total work in adding columns $A(g)$ to A^* is $O(n^5\text{EO})$: The time to compute one $A(g)$ is $O(n\text{EO})$, so it suffices to show that $O(n^4)$ columns get added to A^*. New column $A(g)$ needs to be added whenever the i with $d^i = d^{[g]}$ disappears from $\mathcal{I}$ due to λ_i hitting zero, or ReduceV. In this case d^i serves as $d^{[g]}$ for the elements $D = \{g \in E \mid d^{[g]} = d^i\}$, and so when d^i disappears we need to add the $|D|$ new columns $A(g)$ for $g \in D$ to A^*. For $g \in D$, losing d^i means that either $\Phi(g)$ decreases by 1, or $(d_g^{\min}, h(g))$ changes. Since the total decrease in Φ is $O(n^4)$, the first case happens $O(n^4)$ times; since $(d_e^{\min}, h(e))$ changes $O(n^2)$ times, the second case happens $O(n^2)$ times, for a total of $O(n^4)$ columns added to A^*.

The total work in solving (22) is $O(n^6)$: Each time a new $A(g)$ gets added to A^*, it takes $O(n^2)$ time to reduce the new A^* to the canonical form needed for solving (22). Since $O(n^4)$ columns are added, total time is $O(n^6)$.

The total work in UpdateI is $O(n^6)$: Each iteration of the loop either makes a new element join $S^0(y)$, or deletes some i with $d^i = d^{[e]}$ for some e from $\mathcal{I}$. Since $S^0(y)$ is monotone, the first case happens $O(n)$ times. The second case either causes some $H^{[e]}$ to decrease, or a change in $(d_e^{\min}, h(e))$; these happen $O(n^4)$ times. Hence there are $O(n^4)$ iterations that call UpdateI. A naive implementation of UpdateI takes $O(n^2)$ time. (It is possible to implement UpdateI using a priority queue [10] with i's key being $d^i(E)$ to reduce its time to $O(n \log n)$.)

The total work in ReduceV is $O(n^6)$: The time for ReduceV is $O(n^3)$ plus $O(n^2)$ for each $i \in \mathcal{I}$ eliminated. ReduceV ensures that $|\mathcal{I}| \leq n$, and the subsequent UpdateI adds $O(n)$ $d^{[e+]}$ to $\mathcal{I}$, and so $|\mathcal{I}| \leq 2n$. Each i added to $\mathcal{I}$ between now and the next call to ReduceV is $d^{[e+]}$ for some e, which generates a new column $A(e)$. At the next call to ReduceV, $|\mathcal{I}| \geq 3n$, and so at least n new columns have been added to A^*. Since $O(n^4)$ columns get added to A^*, ReduceV gets called $O(n^3)$ times. The number of indices deleted from $\mathcal{I}$ is within $O(n)$ of the number added, which is of the same order as the number of columns added to A^*, or $O(n^4)$, for a total of $O(n^6)$ ReduceV work.

The algorithm runs in $O(n^5\text{EO} + n^6)$ time: This is the sum of the times of the major steps in the four previous paragraphs. All other work is clearly bounded by $O(n^2)$ per iteration, and so is dominated by the major steps. ■

3.4.2 Iwata and Orlin's SFM Algorithm

In 2009 Iwata and Orlin [52] gave a simple SFM algorithm that re-uses some of the ideas of Orlin's Algorithm, and a few ideas from the Hybrid Algorithm. It achieves time bounds nearly as good as Orlin's Algorithm, and it has a fully combinatorial version that is the current champion. We call it the Iwata-Orlin, or IO, Algorithm. Unlike Orlin's Algorithm, it is not naturally strongly polynomial, but by making some changes we can get a strongly polynomial version, IO-SP. Also unlike Orlin's Algorithm, it can be make fully combinatorial, which we call IO-FC. We develop each version in turn.

The IO Algorithm is based on a quite different objective function than other SFM algorithms.

The *Min Norm Point (MNP)* problem is to solve

$$\begin{array}{l} \min \sum_{e \in E} y_e^2 \\ \text{s.t. } x \in B(f). \end{array} \tag{24}$$

A proof similar to Lemma 2.3 shows that the union and intersection of SFM solutions are again SFM solutions. Therefore there is a unique minimal SFM solution (namely the intersection of all SFM solutions) and a unique maximal SFM solution (namely the union of all SFM solutions). The next theorem shows the connection between MNP and SFM:

Theorem 3.16 ([25, Theorem 3.3], [26, Proposition 7.1]) *Suppose that y^* solves MNP. Then $S^-(y^*)$ is the minimal SFM solution, and $S^-(y^*) \cup S^0(y^*)$ is the maximal SFM solution.*

This theorem motivate a basic idea of the IO Algorithm: concentrate on the MNP objective (24). Recall that $B(f)$ is contained in the hyperplane $y(E) = f(E)$. It is easy to see that the optimal MNP solution for y in this hyperplane is $\tilde{y} = (\frac{1}{f(E)})\mathbb{1}$, because the quadratic objective function prefers component values that are as equal as possible. Indeed, if $\tilde{y} \in B(f)$ this optimizes MNP over $B(f)$, with the primal solution being E if $f(E) \leq 0$, and $\emptyset$ if $f(E) \geq 0$. (The MNP solution has many strong properties, see Nagano [68].)

This leads to the idea of finding some central value μ and some $S \subseteq E$ and trying to move the $y_e < \mu$ with $e \in S$ upwards towards μ, and the $y_e > \mu$ with $e \in S$ downwards towards μ. We call this a *squish step*. If we can arrange things so that the y_e move a sufficiently long distance towards μ in a squish step, then we could get a large enough decrease in $\sum_E y_e^2$ to get geometric convergence. The algorithm will end up caring only about the positive components of y, and so it will instead concentrate on the potential function $\Phi(y) \equiv \sum_{e \in E}(y_e^+)^2$ (different from the Φ in Orlin's Algorithm). This points up a big difference between Orlin's Algorithm and the IO Algorithm: Orlin's Algorithm aims to monotonically increase $S^0(y)$, whereas the IO Algorithm is happy to have a negative y_e become positive during a squish step (which we take into account in developing (27) below).

There are five versions of the IO Algorithm: (1) The basic weakly polynomial version, that we call *Basic IO*, that is needed for the fully combinatorial version; (2) A faster weakly polynomial version, that we call *Wave IO*; (3) A variant of Wave IO that is strongly polynomial, that we call *IO-SP*; (4) A transitional strongly polynomial version of Basic IO, that we call *Basic IO-SP*; and (5) a fully combinatorial version based on Basic IO-SP that we call *IO-FC*.

The Basic Weakly Polynomial Version of IO The algorithm will use Lemma 3.13 to delete elements from E. For simplicity denote the original E by $\bar{E}$, and the current value by E. Note that all deleted elements e have $d_e^{\min} > 0$, and so by (OT i) they satisfy $y_e > 0$. The proof of Lemma 3.13 shows that at the moment of deletion, $y(E) = f(E)$. Deleted elements are never changed after deletion, and so we maintain that $y(E) = f(E)$ at all times.

Define $y_{\max} = \max_E y_e$. If $y_{\max} \leq 0$, then clearly the current E solves SFM, so we can assume that $y_{\max} > 0$. The next lemma bounds $y_{\max}$ away from 0:

Lemma 3.17 *Suppose that f is integer-valued. If $y_{\max} < 1/n$, then the current E is the unique maximal SFM solution.*

Proof: We have $y_e < 1/n$ for $e \in E$, and $y_e > 0$ for $e \in \bar{E}$. Thus $y^-(\bar{E}) > y(E) - |E|/n \geq x(E) - 1 = f(E) - 1$. For any $S \subseteq E$ and by integrality of f we have $f(S) \geq y^-(\bar{E}) \geq f(E)$, so E is an SFM solution. Since $y_e > 0$ for all $e \in \bar{E} - E$, by Lemma 3.13 in fact E is the maximal SFM solution. ∎

Now define $\delta = y_{\max}/4n$. To aim for sufficient decrease in $\Phi(y)$ during a squish step, we want to find a $\mu \in [\delta, y_{\max})$ such that the interval $(\mu - \delta, \mu + \delta)$ contains no y_e. Since this interval has size 2δ, the total interval $(0, y_{\max})$ has size $4n\delta$, and there are at most n y_e values, by the pigeon-hole principle we must be able to find such a μ. Note that we can find μ by sorting the positive y_e and then doing a linear scan in $O(n \log n)$ time, which will not be a bottleneck.

Now we define the squish step w.r.t. μ. Compute $g = \operatorname{argmin}\{d_e^{\min} \mid e \in E \text{ s.t. } y_e > \mu\}$, $l = d_g^{\min}$, and $k \in \mathcal{I}$ with $d_g^k = l$. Now subroutine NewOrd(k, μ, l) constructs a new linear order $\prec_{k'}$ from $\prec_k$ like this: Define $S = \{e \mid d_e^k = l\}$, and partition S into $Q = \{e \in S \mid y_e < \mu\}$ and $R = \{e \in S \mid y_e > \mu\}$. Then $\prec_{k'}$ is the result of the general block exchange that moves Q to the left of the block and R to the right of the block (as in Section 3.3.4). The labels for k' match those of k, except that $d_e^{k'} = d_e^k + 1$ for $e \in R$ (it's easy to see that this preserves (OT i–iii)). Since $g \in R$, we have that at least one element has its label increased in $d^{k'}$.

Lemma 2.6 says that $v^{k'} - v^k$ is non-negative on Q, non-positive on R, and zero elsewhere, and so moving in the $v^{k'} - v^k$ direction will tend to squish the y_e for $e \in S$ towards μ. The step length of this move is determined by two factors. First, to keep $\lambda \geq 0$ we need to choose $\alpha \leq \lambda_k$. Second, we don't want to overshoot μ. For $e \in Q$ we know that $v_e^{k'} \geq v_e^k$. If $v_e^{k'} > v_e^k$ then we want that $y_e + \alpha(v_e^{k'} - v_e^k) \leq \mu$, or $\alpha \leq (\mu - y_e)/(v_e^{k'} - v_e^k)$, and a similar thing for $e \in R$. Thus we choose

$$\alpha = \min\left\{\lambda_k, \min\left(\frac{\mu - y_e}{v_e^{k'} - v_e^k} \;\middle|\; e \in Q \cup R,\ v_e^{k'} \neq v_e^k\right)\right\}. \tag{25}$$

Moving by α in direction $v^{k'} - v^k$ is done by Squish. When $\alpha = \lambda_k$ we call the Squish *full*, otherwise we call it *partial* (*saturating* and *non-saturating* in [52], respectively). Note that a full Squish does not change $|\mathcal{I}|$ (since $\prec_{k'}$ joins $\mathcal{I}$ but $\prec_k$ drops out), whereas a partial Squish increases $|\mathcal{I}|$ by one (though it does cause a y_e to equal μ for some $e \in S$).

The Squish subroutine for the IO SFM Algorithm

[Input is μ s.t. $\nexists\, y_e \in (\mu - \delta, \mu + \delta)$; and $k \in \mathcal{I}$ and $g \in E$ with $d_g^k = l$.]
Compute $S = \{e \mid d_e^k = l\}$, $Q = \{e \in S \mid y_e < \mu\}$, $R = \{e \in S \mid y_e > \mu\}$.
Call NewOrd(k, μ, l) to get new order $\prec_{k'}$ which is $\prec_k$ with the block exchange that moves Q before R.
Add k' to $\mathcal{I}$ with $d_e^{k'} = d_e^k + 1$ on S, $d_e^{k'} = d_e^k$ elsewhere.
[Lemma 2.6 $\Rightarrow v^{k'} - v^k \geq 0$ on Q, $v^{k'} - v^k \leq 0$ on R, and zero elsewhere.]
Compute α via (25).
Set $\lambda_k \leftarrow \lambda_k - \alpha$, $\lambda_{k'} \leftarrow \alpha$, and update y and $y_{\max}$.
If $\lambda_k = 0$ [since α was λ_k, a full Squish] delete k from $\mathcal{I}$.

Now we are ready for the key lemma that drives all the IO algorithms. It assumes that a sequence of Squishes is substantial enough to cause at least one component of y to change by some multiple k of δ. Although a Squish can cause $y_{\max}$ to decrease, we use $y_{\max}$ to refer to its value at the beginning of the Squish throughout the following lemma and its proof.

Lemma 3.18 *Suppose that y and μ are such that there is no y_e in $(\mu-\delta, \mu+\delta)$, and that we apply a sequence of one or more* SQUISH*es w.r.t. μ that changes y to y'. Further suppose that there is some $e \in E$ with $|y_e^+ - y'_e| \geq k\delta$. Then $\Phi(y) - \Phi(y') \geq k\Phi(y)/16n^3$.*

Proof: Define $Q = \{e \mid y'_e > y_e\}$, $R = \{e \mid y'_e < y_e\}$, and $Q^+ = \{e \in Q \mid y'_e > 0\}$. Due to how SQUISHes work, for $e \in Q$ we have $y_e \leq \mu - \delta$, and for $e \in R$ we have $y_e \geq \mu + \delta$. If $e \in Q$ (or Q^+) then $y_e \leq \mu - \delta$ and $y'_e \leq \mu$, and so

$$y_e + y'_e \leq 2\mu - \delta \text{ for } e \in Q^+; \text{ similarly } y_e + y'_e \geq 2\mu + \delta \text{ for } e \in R. \tag{26}$$

Now $y_e^+ \geq y_e$, and so $\sum_{Q^+}(y_e^+ - y'_e) \geq \sum_{Q^+}(y_e - y'_e)$. Then $Q^+ \subseteq Q$ and $y'_e \geq y_e$ on Q imply that $\sum_{Q^+}(y_e - y'_e) \geq \sum_{Q}(y_e - y'_e)$. Since $y_e \neq y'_e$ only on $Q \cup R$ and $y(E) = y'(E)$ we get $\sum_Q(y_e - y'_e) = \sum_R(y'_e - y_e)$, and together these imply

$$\sum_{Q^+}(y_e^+ - y'_e) \geq \sum_{R}(y'_e - y_e). \tag{27}$$

Now

$$\begin{aligned}
\Phi(y) - \Phi(y') &= \textstyle\sum_{R\cup Q^+}((y_e^+)^2 - (y'_e)^2) \\
&= \textstyle\sum_{R\cup Q^+}(y_e^+ - x'_e)(y_e^+ + y'_e) \\
&\geq \textstyle\sum_{Q^+}(y_e^+ - y'_e)(2\mu - \delta) + \sum_R(y_e - y'_e)(2\mu + \delta) \quad \text{by (26)} \\
&\geq \textstyle\sum_{Q^+}(y'_e - y_e^+)\delta + \sum_R(y_e - y'_e)\delta \qquad\qquad \text{by } 2\mu \cdot (27).
\end{aligned} \tag{28}$$

By hypothesis, at least one term of (28) has $e \in Q^+$ and $y'_e - y_e^+ \geq k\delta$, or $e \in R$ and $y_e - y'_e \geq k\delta$, and so $\Phi(y) - \Phi(y') \geq k\delta^2 = ky_{\max}^2/16n^2$. Since each term of $\Phi(y)$ is at most $y_{\max}^2$, we get $\Phi(y) \leq ny_{\max}^2$, or $y_{\max}^2 \geq \Phi(y)/n$. From this we get $ky_{\max}^2/16n^2 \geq k\Phi(y)/16n^3$. ∎

Notice that a partial SQUISH moves at least one $e \in Q \cup R$ from outside $(\mu-\delta, \mu+\delta)$ to equal μ, a distance of at least δ (and when $e \in Q$, we indeed have that $y'_e - y_e^+ \geq \delta$). Thus we can apply Lemma 3.18 with $k = 1$ to get:

Corollary 3.19 *Suppose that Squish is a partial change from y to y'. Then $\Phi(y) - \Phi(y') \geq \Phi(y)/16n^3$.* ∎

Now we can specify the overall Basic IO Algorithm.

We now prove that this works, and give its running time. We again give one big proof, but we pick out the key claims along the way in boldface.

Theorem 3.20 *The Basic IO Algorithm correctly finds the unique maximal SFM solution, and runs in $O(n^6 \text{EO} \log(nM))$ time.*

Proof: **If the algorithm terminates, it is with the unique maximal SFM solution:** The algorithm terminates when $y_{\max} < 1/n$, and indeed the current E is then the unique maximal SFM solution by Lemma 3.17 and Lemma 3.13.

The Basic IO Algorithm for SFM

Initialize by setting $d^1 = 0$, $\prec_1 = 12\ldots n$, $\lambda_1 = 1$, $\mathcal{I} = \{1\}$, and computing $y = v^1$.
While $y_{\max} \geq 1/n$ [using Lemma 3.17] do
  Find $\mu \geq \delta$ s.t. $\nexists\ y \in (\mu - \delta, \mu + \delta)$.
  Compute $g = \text{argmin}\{d_e^{\min} \mid y_e > \mu\}$, $l = d_g^{\min}$, and $k \in \mathcal{I}$ s.t. $d_g^k = l$.
  Call SQUISH using this μ, k, g, and l.
End while.
Output final E as the unique maximal SFM solution.

Distance labels stay valid: We already saw that (OT i–iii) stay valid during SQUISH.

Doing $O(n^3)$ partial SQUISHes suffices to halve $\Phi(y)$: We can re-express the conclusion of Corollary 3.19 as $\frac{\Phi(y)-\Phi(y')}{\Phi(y)} \geq 1/16n^3$. Therefore $O(n^3)$ partial SQUISHes suffice to halve $\Phi(y)$.

There are $O(n^3 \log(nM))$ partial iterations: The initial value of $\Phi(y)$ is at most $4nM^2$. Once $\Phi(y)$ is below $1/n^2$ then $y_{\max}$ must be below $1/n$, and so by Lemma 3.17 the current E is the unique maximal SFM solution.

There are $O(n^5 \log(nM))$ full iterations: For $i \in \mathcal{I}$ and $e \in E$, define $\text{dgap}_e^i = (n+1) - d_e^i$, the remaining room for d_e^i to increase. Consider a second potential function $\Gamma(\mathcal{I}, d) = \sum_i \sum_e \text{dgap}_e^i$. Each full SQUISH decreases $\Gamma(\mathcal{I}, d)$ by at least one (since new order k' joins $\mathcal{I}$ but old order k exits, and at least $d_g^{k'}$ increases by one), and each partial SQUISH increases $\Gamma(\mathcal{I}, d)$ by at most n^2. Thus the total increase in $\Gamma(\mathcal{I}, d)$ from partial SQUISHes is $O(n^5 \log(nM))$, and so this bounds the number of full SQUISHes.

The running time is $O(n^6 \text{EO} \log(nM))$: Each SQUISH takes $O(n\text{EO})$ time, and the total number of SQUISHes is $O(n^5 \log(nM))$. ∎

The Wave Weakly Polynomial Version of IO Iwata and Orlin modify Basic IO in two ways to speed it up. First, note that so far we have not used REDUCEV. Second, we could repeatedly apply SQUISH (through several full SQUISHes) long enough that we get a partial SQUISH so that Corollary 3.19 applies, and we can do this for every possible μ in $[\delta, y_{\max}]$.

The WAVE subroutine for the IO Algorithm for SFM

[Input is the interval $[\delta, y_{\max}]$.]
  Initialize $\mu = \delta$.
  While $\mu \leq y_{\max}$ and no value of $d_e^{\min}$ changes do
    Increase μ the minimum amount until $\nexists\ y \in (\mu - \delta, \mu + \delta)$.
    While $\nexists\ e$ with $y_e = \mu$ [i.e., keep going until get a partial SQUISH]
      and no value of $d_e^{\min}$ changes do
      Compute $g = \text{argmin}\{d_e^{\min} \mid y_e > \mu\}$, $l = d_g^{\min}$, and $k \in \mathcal{I}$ s.t. $d_g^k = l$.
      Call SQUISH using this μ, k, g, and l.
    End while.
  End while.

We implement this in subroutine WAVE. It starts μ at δ, increases it to a value such that there is no e with $y_e \in (\mu-\delta, \mu+\delta)$ (so that we can apply SQUISH), and then repeatedly applies SQUISH until we get a partial SQUISH that causes $y_e = \mu$ for some e. It then increases μ to the next higher possible value, and repeats. For the running time proof, we also break out of the iterations in case a SQUISH causes $d_e^{\min}$ to increase for some e. Here we assume that $y_{\max}$ changes dynamically during SQUISHes, but that δ maintains its initial value.

Now we can specify the overall Wave IO Algorithm.

The Wave IO Algorithm for SFM

Initialize by setting $d^1 = 0$, $\prec_1 = 12\ldots n$, $\lambda_1 = 1$, $\mathcal{I} = \{1\}$, and computing $y = v^1$.
While $y_{\max} \geq 1/n$ [using Lemma 3.17] do
  Compute $y_{\max} = \max_e y_e$ and set $\delta = y_{\max}/4n$.
  Call WAVE. [this could decrease $y_{\max}$]
  If a distance gap k appears, compute $T = \{e \mid d_e^{\min} \geq k\}$, set $d_e^i = n+1$ $\forall i,\ e \in T$,
    set $E \leftarrow E - T$. [using Lemma 3.13]
  Call REDUCEV.
End while.
Output final E as the unique maximal SFM solution.

We now prove that this works, and give its running time. We again give one big proof, but we pick out the key claims along the way in boldface.

Theorem 3.21 *The Wave IO Algorithm correctly finds the unique maximal SFM solution, and runs in $O((n^4\mathrm{EO} + n^5)\log(nM))$ time.*

Proof: **If the algorithm terminates, it is with the unique maximal SFM solution:** The algorithm terminates when $y_{\max} < 1/n$, and indeed the current E is then the unique maximal SFM solution by Lemma 3.17 and Lemma 3.13.

Distance labels stay valid: We already saw that (OT i–iii) stay valid during SQUISH. REDUCEV could delete some i from $\mathcal{I}$, but this cannot impair validity.

WAVE **returns $O(n^2)$ times due to some $d_e^{\min}$ increasing:** Each d_e^i is non-decreasing during the algorithm, and so the $d_e^{\min}$ are non-decreasing. Each $d_e^{\min}$ starts at 0 and ends up at most $n+1$, and so is increased $O(n)$ times, and so there are $O(n^2)$ times overall that some $d_e^{\min}$ increases.

Split analysis of WAVE **without some $d_e^{\min}$ increasing into two cases:** Recall that WAVE can cause $y_{\max}$ to decrease ($y_{\max}$ is non-increasing during the algorithm). Case A: the value of $y_{\max}$ at the end of WAVE is at most $3n\delta$. Case B: the value of $y_{\max}$ at the end of WAVE is between $3n\delta$ and $4n\delta$.

Let y^0 denote the value of y before calling WAVE, and y' its value after calling WAVE. **In Case A we have $\Phi(y^0) - \Phi(y') \geq \Phi(y^0)/16n^2$:** This follows from applying Lemma 3.18 with $k = n$, since $y_{\max}$ shrinking $n\delta$ implies that some y_e moved at least $n\delta$.

In Case B we also have $\Phi(y^0) - \Phi(y') \geq \Phi(y^0)/16n^2$: We first claim that WAVE processes $\Theta(n)$ different values of μ. Each time μ changes, it increases by at least δ, and ends up at least $3n\delta$, so there are $O(n)$ values of μ. On the other side, each value of y_e can invalidate a

subinterval of width at most 2δ in the overall interval of at least $[\delta, 3n\delta]$, and so there are at least n values of μ that will be considered. For each of these $\Theta(n)$ values of μ, WAVE ends with a partial SQUISH that have some e with $y_e = \mu$, and so Corollary 3.19 applies. Since it applies $\Theta(n)$ times, we get that $\Phi(y')) \leq (1 - 1/16n^3)^n \Phi(y^0)$, and so $\Phi(y^0) - \Phi(y') \leq \Phi(y^0)/16n^2$.

There are $O(n^2 \log(nM))$ calls to WAVE: We showed above that there are $O(n^2)$ WAVEs that return due to some $d_e^{\min}$ increasing. Each other WAVE satisfies $\Phi(y^0) - \Phi(y') \geq \Phi(y^0)/16n^2$. Thus it takes $O(n^2)$ iterations to cut $\Phi(y)$ in half, and so $O(n^2 \log(nM))$ iterations to cut $\Phi(y)$ from its initial value of at most nM^2 to a final value of at most $1/n^3$. When $\Phi(y) < 1/n^3$, then $y_{\max} < 1/n$, and we are optimal.

The size of $\mathcal{I}$ at the end of a WAVE is at most $5n$: Each full SQUISH leaves $|\mathcal{I}|$ unchanged, so we need only show that the number of partial SQUISHes is at most $4n$. Each partial SQUISH causes some y_e to equal the current μ, and we showed above that this happens at most $4n$ times.

Total work in calling REDUCEV is $O(n^5 \log(nM))$: REDUCEV gets called once per WAVE, and there are $O(n^2 \log(nM))$ WAVEs. The number of columns that REDUCEV must process is $O(n)$, and each call costs $O(n^3)$.

The time per call to WAVE is $O(n^2\text{EO})$: Each SQUISH does a block exchange on S, which costs $O(|S|\text{EO})$ by (6), and this is the bottleneck operation. We saw that there are $O(n)$ partial SQUISHes. Each takes $O(|S|\text{EO})$ time, for a total of $O(n^2\text{EO})$ time. Now we consider full SQUISHes. A full SQUISH uses $\prec_k$ and S to create new linear order $\prec_{k'}$. We call $\prec_{k'}$ a *child* of $\prec_k$, and any orders coming from further SQUISHes on $\prec_{k'}$ we call *descendents* of $\prec_k$. Because μ is non-decreasing during SQUISH, and because we choose g to minimize $d_e^{\min}$ among eligible e, the set S' for a descendent of $\prec_k$ must be disjoint and to the right of S. Therefore the total work in computing new vertices from descendents of $\prec_k$ is only $O(n\text{EO})$ (this is similar to an argument in the proof of Theorem 3.12). Since we start with $O(n)$ linear orders in $\mathcal{I}$, the total work from the full SQUISHes is $O(n^2\text{EO})$.

The total time for the algorithm is $O((n^4\text{EO}+n^5)\log(nM))$: The bottleneck operations are calls to REDUCEV (which cost $O(n^5 \log(nM))$ in total) and calls to WAVE (which cost $O(n^2\text{EO})$ each). There are $O(n^2 \log(nM))$ calls to WAVE, for a total of $O((n^4\text{EO}+n^5)\log(nM))$ time. ■

Note that the argument for this running time in [52, Section 4] does not distinguish Case A from Case B as we do above, and this appears to be wrong. The argument that fixes this is original to this survey.

The Strongly Polynomial Version of IO Often weakly polynomial algorithms are scaling-based. The IO Algorithm does not do explicit scaling, but Lemma 3.18 shows that the algorithm is effectively implicitly scaling $\Phi(y)$. In order to turn IO into its strongly polynomial version IO-SP we need a analogue of Lemma 3.5, a proximity lemma adapted to this context.

Lemma 3.22 *If $y_{\max} > 0$ and $y \in B(f)$, then $y_g < -ny_{\max}$ implies that g is contained in all SFM solutions.*

Proof: Define $P = \{e \mid y_e > 0\}$. Now execute a conceptual algorithm to move y towards an SFM solution. For each $e \in P$ in turn use brute force to compute $c(g, e; y)$, and the step length $\alpha = \min(y_e, c(g, e; y))$. Now update $y \leftarrow y + \alpha(\chi_g - \chi_e)$, and go on to the next $e \in P$. Note that $y_e \leq y_{\max}$ for all $e \in P$ and the original $y_g < -ny_{\max}$ imply that the final y_g is still negative.

For $e \in P$, iteration e results in either $y_e = 0$ or there is some y-tight set T_e containing g but not e. Subsequent iterations keep T_e y-tight. Then the intersection T of all these T_e is again y-tight, and satisfies that $y_e \leq 0$ for all $e \in T$ (if no iterations result in a T_e, then y_e ended up equal to zero for all $e \in P$, and then we can use E for T).

Suppose that $g \notin S \subseteq E$. Then $f(T) = y(T) < y(S \cap T) \leq f(S \cap T)$. Submodularity then implies that $f(S) > f(S \cup T)$, and so the subset S not containing g cannot solve SFM. ■

We can no longer use Lemma 3.17 as a termination criterion, as it depends on the assumption that f is integral. But Lemma 3.13 is still valid, and so (as in the weakly polynomial version of IO) when we find a distance gap we exclude the elements above the gap, and so E shrinks during the algorithm. Thus if we have $y_{\max} \leq 0$, then the current E must be the unique maximal SFM solution.

Now we will use the same ring family technology covered in Section 3.3.2, where our current state of knowledge about the solution is represented by the directed graph (E, C). We know that all SFM solutions must correspond to ideals of (E, C). To simplify the exposition, we assume that C contains no directed cycles (if it does, then we can shrink strongly connected components to new vertices, and then use the methods of Section 3.3.2). For $e \in E$, recall that D_e is the set of descendents of e in (E, C), and A_e is the set of ancestors of e in (E, C), both including e. When we delete the elements in T above a distance gap from E, we must also delete $A(T)$, the ancestors of elements in T. This ensures that our current set E is always an ideal.

For $e \in E$ and S an ideal of $(E - D_e, C)$, recall that $f_e(S) = f(S \cup D_e) - f(D_e)$ is the contraction of f at e. Given $y \in B(f)$ (where now we include only the constraints $x(s) \leq f(S)$ for S an ideal of (E, C) in defining $B(f)$) represented via (11) and $e \in E$, we can produce a related $\hat{y} \in B(f_e)$ as follows. Given $\prec_i$ with $i \in \mathcal{I}$, define $\prec_{i'}$ as the linear order on E we get by moving the elements of D_g to the front of $\prec_i$. Suppose that $S \subseteq E - D_e$ is such that $D_e \cup S$ is an initial segment of some $\prec_{i'}$. Then from Greedy $v^{i'}(S) = f(D_e \cup S) - f(D_e) = f_e(S)$. Therefore, if we define $\hat{v}^{i'} \in \mathbb{R}^{E - D_e}$ to be $v^{i'}$ with its first $|D_e|$ coordinates chopped off, we get that $\hat{v}^{i'} \in B(f_e)$. Finally we can define $\hat{y} = \sum_{\mathcal{I}} \lambda_i \hat{v}^{i'}$ (using the same λ_i as y) and know that $\hat{y} \in B(f_e)$.

As the algorithm proceeds, there are three useful events that might happen w.r.t. $y_{\max}$:

1. **We could have $y_{\max} \leq 0$:** This implies that the current E is the unique maximal SFM solution, and we can stop.

2. **We could have some $g \in E$ with $y_g < -n y_{\max}$:** Then Lemma 3.22 implies that g belongs to every SFM solution. We can reduce to the case of the contracted function f_g, recursively find the unique maximal SFM solution S_g for f_g, and return $S_g + g$.

3. **We could have some $g \in E$ with $f(D_g) > n^2 y_{\max}$:** Compute the $\prec_{i'}$ and $\hat{y} \in B(f_g)$. Then $\hat{y}(E - D_g) = f_g(E - D_g) = f(E) - f(D_g) < -(n^2 - n) y_{\max}$, implying that there is some $h \in E - D_g$ with $\hat{y}_h < -n y_{\max}$. We can then apply Lemma 3.22 to f_g and this h to get the condition that any solution to SFM for f_g must include h, which implies that we can add the arc $g \to h$ to C. Such arcs can get added at most $O(n^2)$ times before the algorithm terminates.

It is possible to have both event 2 and event 3 happen, and it is quite possible that none of these events happens. We need to arrange the algorithm so that in a strongly polynomial number

of iterations, we produce either a negative enough element (event 2), or a positive enough D_g (event 3).

At a given point in the algorithm we call element $e \in E$ *big* if $y_e > \delta = y_{\max}/4n$. Lemma 2.2 shows that $y_e \leq f(D_e) - f(D_e - e)$. Therefore e big implies that we have either $f(D_e) > \delta/2$ or $f(D_e - e) < -\delta/2$ (or both). Thus if we accumulate enough geometric decrease using some version of Lemma 3.18 to reduce $\Phi(y)$ by a factor of at least $8n^5$, then we reduce $y_{\max}$ by a factor of at least $8n^3$, and we'd have either that $f(D_e) > n^2 y_{\max}$ (i.e., event 2, so we can add an arc to C), or $f(D_e - e) < -n^2 y_{\max}$, and so there is some $g \in D_e - e$ such that $y_g < -n y_{\max}$ (i.e., event 3, so we can contract g out of E).

Now we can specify the overall IO-SP SFM Algorithm.

The IO-SP Algorithm for SFM

Initialize by setting $d^1 = 0$, $\prec_1 = 12 \ldots n$, $\lambda_1 = 1$, $\mathcal{I} = \{1\}$, and computing $y = v^1$.
While $y_{\max} > 0$ [until event 1] do
  Compute $y_{\max} = \max_e y_e$ and set $\delta = y_{\max}/4n$.
  If there is some $g \in E$ with $y_g < -n y_{\max}$, recursively call the algorithm on f_g
    getting optimal S_g, return $S_g + g$. [event 2, using Lemma 3.22]
  If there is some $g \in E$ with $y(D_g) \geq n^2 y_{\max}$, compute $\hat{y} \in B(f_g)$;
    for each $h \in E - D_g$ with $\hat{y}_h < -n y_{\max}$ add arc $g \to h$ to C. [event 3]
  Call WAVE. [this could decrease $y_{\max}$]
  If a distance gap k appears, compute $T = \{e \mid d_e^{\min} \geq k\}$, set $d_e^i = n + 1 \; \forall i, \; e \in A(T)$,
    set $E \leftarrow E - A(T)$. [using Lemma 3.13]
  Call REDUCEV.
End while.
Output final E as the unique maximal SFM solution.

We now prove that this works, and give its running time. We again give one big proof, but we pick out the key claims along the way in boldface.

Theorem 3.23 *The IO-SP Algorithm correctly finds the unique maximal SFM solution, and runs in* $O((n^5 \mathrm{EO} + n^6) \log n)$ *time.*

Proof: **If the algorithm terminates, it is with the unique maximal SFM solution:** The algorithm terminates when $y_{\max} \leq 0$, and indeed the current E is then the unique maximal SFM solution by Lemma 3.13.

When event 3 happens, at least one arc gets added to C: We have $\hat{y}(E - D_g) = f_g(E - D_g) = f(E) - f(D_g) < -(n^2 - n) y_{\max}$, implying that there is some $h \in E - D_g$ with $\hat{y}_h < -n y_{\max}$.

Define a *phase* as a sequence of WAVEs **that reduce $\Phi(y)$ by half**, and b_j as the number of big elements at the beginning of the j-th WAVE, and K as the total number of phases. An e with $y_{\max} = y_e$ is always big, so $b_j \geq 1$. The proof of Theorem 3.21 showed that there are $O(n^2)$ WAVEs which increase some $d_e^{\min}$, and each WAVE takes $O(n^2 \mathrm{EO})$ time, for a (strongly polynomial) total time of $O(n^4 \mathrm{EO})$. This is not a bottleneck, so for the rest of the proof we focus only on WAVEs that do not increase any $d_e^{\min}$.

A WAVE in phase j reduces $\Phi(y)$ by a factor of at least $(1 - 1/(17n^2b_j))^n$: At the beginning of the WAVE we have $\Phi(y) \leq (n - b_j)\delta^2 + 16n^2b_j\delta^2 \leq 17n^2b_j\delta^2$. A similar Case A/B split as in the proof of Theorem 3.21 then proves the claim.

The total number of WAVEs is $O(n\sum_{j=1}^{K} b_j)$: The previous claim shows that $O(b_j n)$ WAVEs suffices to cut $\Phi(y)$ in half, and so phase j uses $O(b_j n)$ WAVEs, and the total number of WAVEs overall is $O(n\sum_{j=1}^{K} b_j)$.

Each $e \in E$ is big for $O(n \log n)$ phases: Once y_e becomes big, it takes only $O(\log(8n^5)) = O(\log n)$ phases until either event 2 or event 3 happens. If event 2 happens, we switch to f_e, which effectively deletes e from E, so it is no longer big. If event 3 happens at least one new arc with tail e is added to C, and this can happen $O(n)$ times.

The total number of WAVEs is $O(n^3 \log n)$: The previous claim implies that each $e \in E$ contributes $O(n \log n)$ to $\sum_{j=1}^{K} b_j$, and so $\sum_{j=1}^{K} b_j = O(n^2 \log n)$. Thus the total number of WAVEs is $O(n^3 \log n)$.

The total time for the algorithm is $O((n^5\text{EO} + n^6) \log n)$: The bottleneck operations are calls to REDUCEV (which cost $O(n^3)$ each) and calls to WAVE (which cost $O(n^2\text{EO})$ each). There are $O(n^3 \log n)$ calls to WAVE and REDUCEV, for a total of $O((n^5\text{EO}+n^6) \log n)$ time. ■

The Fully Combinatorial Version of IO To the fully combinatorial version IO-FC of the IO Algorithm we need to start with a strongly polynomial version and get rid of multiplications and divisions. This means that we can no longer use REDUCEV, and so $|\mathcal{I}|$ will grow. Thus we first have to convert Basic IO (which doesn't use REDUCEV) to its strongly polynomial version Basic IO-SP using the same techniques we used to convert Wave IO into IO-SP. Now we can specify the overall Basic IO-SP SFM Algorithm by replacing calls to WAVE by simpler calls to SQUISH, and avoiding calling REDUCEV.

The Basic IO-SP Algorithm for SFM

Initialize by setting $d^1 = 0$, $\prec_1 = 12 \ldots n$, $\lambda_1 = 1$, $\mathcal{I} = \{1\}$, and computing $y = v^1$.
While $y_{\max} > 0$ [until event 1] do
  Compute $y_{\max} = \max_e y_e$ and set $\delta = y_{\max}/4n$.
  If there is some $g \in E$ with $y_g < -ny_{\max}$, recursively call the algorithm on f_g
    getting optimal S_g, return $S_g + g$. [event 2, using Lemma 3.22]
  If there is some $g \in E$ with $y(D_g) \geq n^2 y_{\max}$, compute $\hat{y} \in B(f_g)$;
    for each $h \in E - D_g$ with $\hat{y}_h < -ny_{\max}$ add arc $g \to h$ to C. [event 3]
  Find $\mu \geq \delta$ s.t. $\nexists\, y \in (\mu - \delta, \mu + \delta)$.
  Compute $g = \text{argmin}\{d_e^{\min} \mid y_e > \mu\}$, $l = d_g^{\min}$, and $k \in \mathcal{I}$ s.t. $d_g^k = l$.
  Call SQUISH using this μ, k, g, and l. [this could decrease $y_{\max}$]
  If a distance gap k appears, compute $T = \{e \mid d_e^{\min} \geq k\}$, set $d_e^i = n + 1\ \forall i,\ e \in A(T)$,
    set $E \leftarrow E - A(T)$. [using Lemma 3.13]
End while.
Output final E as the unique maximal SFM solution.

We now combine the arguments in the proofs of Theorem 3.20 and Theorem 3.23 to bound the number of iterations of Basic IO-SP.

Lemma 3.24 *Basic IO-SP calls* SQUISH $O(n^6 \log n)$ *times.*

Proof: We adopt the same notion of phases and big elements as in the proof of Theorem 3.23; in particular, b_j is the number of big elements in phase j. By a similar argument as in the proof of Theorem 3.23 we get that each partial SQUISH during phase j reduces $\Phi(y)$ by a factor of at least $(1 - 1/17n^2 b_j)$, and so there are $O(n^2 b_j)$ partial SQUISHes during phase j. As in that proof, since each element is big in at most $O(n^2 \log n)$ phases, overall we have $O(n^4 \log n)$ partial SQUISHes.

Now we consider the secondary potential function Γ from the proof of Theorem 3.20. It showed that the number of full SQUISHes is bounded by $O(n^2)$ times the number of partial SQUISHes, and so there are $O(n^6 \log n)$ total SQUISHes. ■

In order to convert Basic IO-SP into IO-FC we can apply the same sorts of tricks that we used for IFF-FC and Hybrid. The only real remaining source of multiplications and divisions in Basic IO-SP stem from the computation of α in (25). We again bound it away from zero so we can discretize it into a integer multiples of a quantum.

Lemma 2.2 says that for all $y \in B(f)$ we have $y_e \leq f(D_e) - f(D_e - e)$. In Basic IO-SP we know that $f(D_e) \leq n^2 y_{\max}$ and $f(D_e - e) \geq y(D_e - e) \geq -n^2 y_{\max}$, and so $v_e^i \leq 2n^2 y_{\max}$ for all $i \in \mathcal{I}$ and $e \in E$. Since $f(E) \geq -n^2 y_{\max}$ we get $v_e^i \geq -2n^3 y_{\max}$. Thus $|v_e^k - v_e^{k'}| \leq 2(n^3 + n^2) y_{\max} \leq 4n^3 y_{\max}$ for all e. Thus in (25) $\alpha \geq \delta / 4n^3 y_{\max} = 1/16n^4 y_{\max}$ for a partial SQUISH (unless $v^k = v^{k'}$).

Thus if we set $\kappa = 16n^4$ we can do everything in integer multiples of $1/\kappa$. Computing α for a partial SQUISH via repeated doubling and repeated addition/subtraction costs $O(n \log n)$ time, as does updating the λ_i, and this is not a bottleneck operation. Thus we get:

Theorem 3.25 *The IO-FC Algorithm correctly finds the unique maximal SFM solution, and runs in* $O((n^7 \mathrm{EO} + n^8) \log n)$ *time.*

Proof: From Lemma 3.24 there are $O(n^6 \log n)$ SQUISHes. Since we are dealing with the ring submodular case using the technique of Section 5.2, each ring evaluation in IO-FC costs $O(\mathrm{EO} + n)$ time. There are $O(n)$ evaluations per call to SQUISH, and so each call to SQUISH costs $O(n\mathrm{EO} + n^2)$ time. ■

4 Comparing and Contrasting the Algorithms

Table 1 summarizes, compares, and contrasts the six main SFM algorithms we have studied, those of Cunningham for General SFM [13], the Fleischer and Iwata [23] Schrijver-PR Push-Relabel variant of Schrijver [76], Orlin [71], Iwata, Fleischer, and Fujishige [50] and Iwata's fully combinatorial version IFF-FC of it [46], and Iwata's Hybrid Algorithm [48]. The current fastest combinatorial running times are $O((n^4 \mathrm{EO} + n^5) \log M) = \widetilde{O}(n^5)$ (assuming that $\mathrm{EO} = O(n)$) for weakly polynomial [48], $O(n^5 \mathrm{EO} + n^6)$ for strongly polynomial [71], and $O(n^8 \mathrm{EO} \log^2 n)$ for fully combinatorial [48] (compared with $\widetilde{O}(n^5 \mathrm{EO} + n^7)$ for Ellipsoid).

Note that all the algorithms besides Schrijver's and Orlin's Algorithms add just one vertex to $\mathcal{I}$ at each exchange (or at most n vertices per augmenting path). Except for the Hybrid Algorithm, they are able to do this because they all consider only consecutive exchanges; the Hybrid Algorithm considers non-consecutive exchanges, but moves in a $v^i - v^j$ direction instead

	Cunningham for General SFM [13], Sec. 3.1	Schrijver [76, 84], Schrijver-PR [23], Sec. 3.2	Iwata, Fleischer, and Fujishige [50, 46], Sec. 3.3	Iwata Hybrid [48], Sec. 3.3.4	Orlin [71], Sec. 3.4.1	Iwata and Orlin [52], Sec. 3.4.2
Pseudo-polyn. running time	$O(Mn^6 \log(Mn) \cdot$ EO)					
Weakly polyn. running time			$O(n^5 \mathrm{EO} \log M)$ [50], Sec. 3.3.1	$O((n^4 \mathrm{EO} + n^5) \cdot \log M)$ **(*)**		$O((n^4 \mathrm{EO} + n^5) \cdot \log(nM))$
Strongly polyn. running time		$O(n^7 \mathrm{EO} + n^8)$ [23, 84]	$O(n^7 \mathrm{EO} \log n)$ [50], Sec. 3.3.2	$O((n^6 \mathrm{EO} + n^7) \cdot \log n)$	$O(n^5 \mathrm{EO} + n^6)$ **(*)**	$O((n^5 \mathrm{EO} + n^6) \log n)$
Fully comb. running time			$O(n^9 \mathrm{EO} \log^2 n)$ [46], Sec. 3.3.3	$O(n^8 \mathrm{EO} \log^2 n)$		$O((n^7 \mathrm{EO} + n^8) \log n)$ **(*)**
Approach	polymatroid	base polyh.	base polyh.	base polyh.	base polyh.	base polyh.
Convergence strategy	weak suff. decrease, pseudo-polyn.	distance label	str. suff. decrease, strongly polyn.	both distance label and str. suff. decrease	potential function	potential function
Exact primal solution?	no	yes	no	no	possible	no
Scaling?	no	no	relaxation parameter	relaxation parameter	no	implicit on $\sum_e (y_e^+)^2$
Max Flow analogy	Max Capacity Path	Max Dist. Push-Relabel	Relaxed Max Cap. Path for z, push on cut for y	Relaxed Max Cap. Path for z, Push-Relab. on cut for y	find nodes not in min cut	find nodes not in min cut
Arc $k \to l$ for aug. y exists if ...	(l,k) consec., $c(k,l;y) > 0$ (minimal)	$l \prec_i k$ (loosest)	(l,k) consec. (medium)	$l \prec_i k$ (loosest)	$l \prec_i k$ (loosest, but not arc-based)	$l \prec_i k$ (loosest, but not arc-based)
Movement direction, representation	edge, simple repr.	edge, complex repr.	edge, simple repr.	vertex diff, simple repr.	multiple vertex diffs, complex repr.	vertex diff, simple repr.
Modifies $\prec_i$ by ...	consec. pairs	single elt blocks	consec. pairs	gen'l blocks	single elt blocks	gen'l blocks
Augments ...	on paths	arc by arc	z on paths, y arc by arc via Swaps	z on paths, y arc by arc via BlockSwaps	reduce $y_e > 0$	reduce $y_e > 0$
Objective	$\max y^-(E)$	$\max y^-(E)$	$\max z^-(E)$	$\max z^-(E)$	$\max y^-(E)$	$\min \sum_e (y_e^+)^2$
# verts. added per exchange	0 or 1	$\leq n$	0 or 1	0 or 1	$\leq n$	0 or 1

Table 1: Summary comparison table of main results. Running times are expressed in terms of $n = |E|$; M, an upper bound on $|f(S)|$ for any $S \subseteq E$; and EO, the time for one call to the evaluation oracle for f. Current fastest times are marked by **(*)**. For comparison, the running time of the strongly polynomial version of the Ellipsoid Algorithm is $\widetilde{O}(n^5 \mathrm{EO} + n^7)$, see Theorem 2.8.

of a $\chi_k - \chi_l$ direction, thereby allowing it also to add at most one vertex per exchange. By contrast, Schrijver's and Orlin's Algorithms allows non-consecutive exchanges, and thus must pay the price of needing to add as many as n vertices to $\mathcal{I}$ for each exchange.

Only Schrijver's Algorithm and a variant of Orlin's Algorithm yield an exact primal solution y. When f is integer-valued, in the base polyhedron approach we apply Theorem 2.9 with $x = 0$, and in the polymatroid approach we apply it with $x = \gamma$. In either case x is also integral, so Theorem 2.9 shows that there always exists an *integral* optimal y. Despite this fact, none of the algorithms always yields an integral optimal y in this case. However, we could get exact integral primal solutions from n calls to SFM as follows: Use the polymatroid approach, compute γ, and discard any $e \in E$ with $\gamma_e < 0$. Run the modified Greedy Algorithm in the proof of Theorem 2.9 starting with $y = 0$, and look for for a vector $y \in P(\tilde{f})$ with $y \leq \gamma$. At each of the n steps we can compute the maximum step length we can take and stay inside $P(\tilde{f})$ via one call to SFM.

4.1 Solving SFM in Practice

There is very little computational experience with any of these algorithms so far, nor is there any generally accepted test bed of instances of SFM. If we reason by analogy with performance of similar Max Flow algorithms (see, e.g., Cherkassky and Goldberg [9]), then Schrijver-PR should out-perform the IFF Algorithm. The reason is that the Push-Relabel Max Flow Algorithm [37] that is analogous to Schrijver-PR has proven to be more robust and faster in practice than the sort of capacity-scaling algorithms that IFF is based on. However, the superior practical performance of Push-Relabel-based Max Flow algorithms depends on using heuristics to speed up the native algorithm [9], and the relative inflexibility of Schrijver-PR may prevent this.

Iwata [47] did some computational experiments comparing the performance of Schrijver's Algorithm, Schrijver-PR, IFF, and Hybrid (Orlin-type algorithms appeared after these tests). Iwata suggested a family of test problems that are fully dense Min Cut problems perturbed by a modular function in such a way that the optimal SFM solution is always $\{1, 2, \ldots, k\}$ for k equalling about $n/3$. All algorithms were started out with the linear order $(n, n-1, \ldots, 1)$ to ensure that they would have to work hard to move the $\{1, 2, \ldots, k\}$ elements before the $\{k+1, k+2, \ldots, n\}$ elements in the linear orders for the optimal solution. Each algorithm was run on instances with sizes from 50 to 1000 nodes.

Algorithm	Total run time	No. oracle calls
Schrijver	$n^{5.8}$	n^4
Schrijver-PR	$n^{5.5}$	n^4
IFF	$n^{4.0}$	$n^{2.5}$
Hybrid	$n^{3.5}$	$n^{2.5}$

Table 2: Empirical results from Iwata [47]. Estimates of running time and number of evaluation oracle calls come from a log-log regression.

Fujishige, Hayashi, and Isotani [30] performed computational tests using Iwata's codes, and including their own code for their *minimum norm point* (or Fujishige-Wolfe) algorithm for SFM (based on Wolfe [86]; see also Fujishige [29, p. 219–220]). Min Norm Point does not have any known polynomial bound, but it performs well in practice. They tested on Iwata's family of instances, as well as Min Cut instances generated from the DIMACS Netflow Challenge [14] test

sets Genrmf-Wide and Genrmf-Long (which are sparse, with fewer than $5n$ arcs on n nodes). Each algorithm was run on instances with sizes from about 64 to 1000 nodes.

Algorithm	Iwata's family	Genrmf-Long family	Genrmf-Wide family
Min Norm Point	(note 1)	$n^{3.3}$	$n^{3.1}$
Schrijver	(note 2)	$n^{5.3}$	$n^{5.3}$
Schrijver-PR	(note 2)	$n^{4.4}$	$n^{4.4}$
IFF	$n^{4.4}$	$n^{4.7}$	$n^{5.3}$
Hybrid	$n^{3.7}$	$n^{5.4}$	$n^{4.9}$

Table 3: Empirical results from Fujishige, Hayashi, and Isotani [30]. Estimates of running time come from a log-log regression. Note 1: It turns out that Iwata's family is trivially easy for Min Norm Point, and so all times are below 0.01 seconds. Note 2: There were not enough runs that finished within 10,000 seconds to estimate the run time.

Tables 2 and 3 show the empirical estimates of each algorithm's running time or number of evaluation oracle calls as a function of the number of nodes of the Min Cut instance. The empirical performance of Min Norm Point consistently beats the combinatorial algorithms. We see that all four combinatorial algorithms are much faster than their theoretical time bounds, and that none of them has a clear advantage over the other algorithms on all three data sets. However, these tests cover a limited number of types of instances at a limited range of sizes. Also, heuristic improvements to these algorithms (such as the "gap" and "exact distance" heuristics that made such a difference for Max Flow algorithms [9]) have not yet been implemented, and so these results should be taken only as a first indication of empirical performance, not the last word (e.g., the running time estimates for Iwata's instances differs between Table 2 and Table 3).

Iwata's data shows that the dominant factor in determining running time is the number of calls to ReduceV: The big advantage of the IFF-based algorithms is that they ended up calling ReduceV many fewer times than the Schrijver-based algorithms (since Orlin's Algorithm must solve (22) using linear algebra similar to ReduceV, this suggests that it might not perform well in practice, but Iwata-Orlin avoids this and so might be fast).

All of the combinatorial SFM algorithms we consider call the evaluation oracle EO only as part of the Greedy Algorithm (although Hybrid re-computes only an interval of a vertex at each iteration). Greedy calls EO for the nested sequence of subsets $e_1^{\prec}$, $e_2^{\prec}$, ..., $e_i^{\prec}$, ..., $e_n^{\prec}$. In some applications we can take advantage of this and use some incremental algorithm to evaluate $f(e_i^{\prec})$ based on the value of $f(e_{i-1}^{\prec})$ much faster than evaluating $f(e_i^{\prec})$ from scratch. For example, for Min Cut on a graph with n nodes and m arcs, one evaluation of $f(S)$ costs $O(m)$ time, but all n evaluations within Greedy can be done in just $O(m)$ time. This could lead to a heuristic speedup for such applications, although most such applications (such as Min Cut) have specialized algorithms for solving SFM that are much faster than the general algorithms here.

Indeed, it is very rare that true general SFM arises in practice. Nearly all applications of SFM in our experience have some special structure that can be taken advantage of, resulting in much faster special-purpose algorithms than anything covered here. As one example, a naive application of Queyranne's Algorithm (see Section 5.4) to solve undirected Min Cut would take $O(n^3|A|)$ time, since EO $= O(|A|)$ in that case. But in fact Nagamochi and Ibaraki [66] show

how to take advantage of special structure to reduce this to only $O(n|A| + n^2 \log n)$. In the great majority of these cases we end up solving the SFM problem as a sequence of Min Cut problems; see Picard and Queyranne [72] for a list of problems reducible to Min Cut.

A recent example of this is where f is the rank function of a graph (the rank of edge subset S is the maximum size of an acyclic subgraph of S) modified by a modular function, which has applications in physics (see Anglès d'Auriac et al. [2]). Here E is the set of edges of the graph, so we use $n = |E|$, and use $|N|$ for the number of nodes. In this case EO $= O(n)$ so the fastest SFM algorithm here would take $\widetilde{O}(n^5)$ time, but [2] shows how to solve SFM using Min Cuts in only $O(|N| \cdot \mathrm{MF}(|N|, n))$ time, where $\mathrm{MF}(|N|, n)$ is the time to solve a Max Flow problem on a graph with $|N|$ nodes and n edges. One of the best bounds for Max Flow is $O(\min\{|N|^{2/3}, \sqrt{n}\} n \log(|N|^2/n) \log M)$ (Goldberg and Rao [36]), which would give a running time of $\widetilde{O}(\min\{|N|^{2/3}, \sqrt{n}\} n^2) \leq \widetilde{O}(n^{5/2})$, much faster than $\widetilde{O}(n^5)$.

Therefore if you are faced with solving a practical SFM problem, you should look very carefully to see if there is some way to solve it via Min Cut before using one of these general SFM algorithms. If there is no apparent way to reduce to a Min Cut problem, then another possible direction is to try a column generation method (see, e.g., [60]), which pairs linear programming technology (for solving (8) or (9)) with a column generation subroutine that would (in this context) come from the Greedy Algorithm. Although such algorithms do not have polynomial bounds, they often can be made to work well in practice.

5 Solvable Extensions of SFM

5.1 Finding All SFM Solutions

There are many applications of SFM where having access to all optimal solutions is helpful. For example, Jeavons et al. [53] considers the problem from an Artificial Intelligence perspective, and their Theorem 3.5 shows that $O(n^2)$ SFM calls suffice to compute all SFM solutions; Shen, Coullard and Daskin [79] consider a facility location problem where knowing all SFM solutions helps in satisfying real-world constraints; Huh and Roundy [44] use parametric SFM to determine an optimal replacement sequence for semiconductor fabrication tools, and again having all SFM solutions helps in satisfying real-world constraints; and Baumann and Skutella [3] also use parametric SFM to compute flows over time.

We extend Picard and Queyranne's [72] result from on finding all Min Cuts to SFM on a ring family using three tools: (1) Edmonds' duality result Theorem 2.9 for SFM, and its associated complementary slackness; (2) the fact from Section 2.5 that the combinatorial SFM algorithms represent their primal solution as a convex combination of greedy vertices; and (3) the algorithm of Bixby, Cunningham, and Topkis (BCT) [8] for finding the tight sets for a vertex. Recall from Section 2.4 that complementary slackness implies that w and S are jointly optimal if and only if (1) $w_e < 0 \Rightarrow e \in S$; (2) $e \in S \Rightarrow w_e \leq 0$; and (3) S is tight for f. Given a primal optimal w, define index sets $N = \{e \in E \mid w_e < 0\}$ and $Z = \{e \in E \mid w_e = 0\}$. Then to find all optimal SFM solutions, all we need to do is to find all tight sets T for w, and then all such T such that $N \subseteq T \subseteq N \cup Z$ are precisely all optimal SFM solutions.

Given the representation $w = \sum_i \lambda_i v^i + \partial\varphi$, note that T is tight for $w \iff \partial\varphi(T) = 0$ and T is tight for each v^i. BCT's algorithm for finding the tight sets for vertex v^i takes $O(n^2 \mathrm{EO})$ time, and produces a directed graph D^i such that T is tight if and only if it is closed in D^i. Therefore, if we take the graph D whose arc set is $\{s \to t \mid \varphi_{st} > 0\}$ union the $O(n)$ arc sets

of the D^i, then the tight sets for w are precisely the closed sets of D. Computing D takes only $O(n^3\text{EO})$ time (for any algorithm maintaining that $|\mathcal{I}| = O(n)$). Now contract all strong components of D which are descendents of elements of P into a single node, and those which are ancestors of elements of N into a single node. The closed sets of the remaining graph are precisely the tight sets for w which also satisfy (1) and (2) of complementary slackness, and hence are precisely the set of optimal solutions for SFM. Although there can be an exponential number of SFM solutions, this contracted graph compactly represents all of them. The material in this section up to this point is covered by Fujishige [29, Remark at the end of Section 14.2], Murota [65, Note 10.11], and Nagano [67].

Schrijver's Algorithm and a version of Orlin's Algorithm produce an exact optimal primal point w (see also [69] for a direct way to modify Orlin's Algorithm to produce all SFM solutions), but the various IFF Algorithms and Iwata-Orlin do not. It is possible to use $O(n)$ calls to an IFF algorithm to find an exact primal optimal point, though it is not clear how to also produce its required representation as a convex combination of vertices.

However, we note that the strongly polynomial IFF algorithms do carry enough information to get all optimal solutions. As they proceed, they develop the sets In and Out, and they work on the reduced ground set $\mathcal{C}$. They use the scale factor δ and recognize optimality in one of two ways: (1) $|\mathcal{C}| \leq 1$: In this case the only possible solutions to SFM on $\mathcal{C}$ are $\emptyset$ and $\mathcal{C}$. Therefore the only possible solutions to SFM on E are In and In $\cup\, E(\mathcal{C})$, and these are easy to check. (2) Scaling parameter $\delta \leq 0$: Theorem 3.9 shows that the current y proves that $\mathcal{C}$ solves SFM. Therefore we can apply the BCT algorithm to find all tight sets w.r.t. y, which then gives us all SFM solutions to the original problem. This shows that to get all optimal SFM solutions, we do not actually need an exact w defined over all of E. It is enough that IFF-SP (and IFF-FC) supply an exact optimal w defined only on $\mathcal{C}$. This was extended to bisubmodular function minimization by [62].

5.2 SFM on Ring Families and their Unions

We already saw with RefineR in Section 3.3.2 that it is not hard to adapt the IFF algorithms to optimize over ring families instead of 2^E. Orlin [71, Section 8] (see also Schrijver [76, Section 6]) gives a way to use any SFM algorithm that works over 2^E to solve SFM over ring family $\mathcal{D}$. Recall that we assume that $\emptyset$, $E \in \mathcal{D}$, that $\mathcal{D}$ is represented as the closed sets of (E, C), and that for $e \in E$, D_e is the set of descendents of e in (E, C). By the reduction of f on (E, C) to $\hat{f}$ on $(\mathcal{C}, C)$ of Section 3.3.2 we can assume that each strong component of (E, C) is a single node. Denote $m = |C|$, so that $m = O(n^2)$. We can pre-compute all the D_e in $O(nm) \leq O(n^3)$ time. Recall from Section 2.2 that we can compute an upper bound M on $|f|$ in $O(n\text{EO})$ time. These times are negligible compared to the running times of all known SFM algorithms.

For $T \subseteq E$ define $\overline{T} = \bigcap_{S \in \mathcal{D},\ T \subseteq S} S = \bigcup_{e \in T} D_e$, the smallest $R \in \mathcal{D}$ containing T. Then for any $S \subseteq E$ define $g(S) = f(\overline{S}) + 2M|\overline{S} - S|$. Note that g is minimized at some $T \in \mathcal{D}$, as for $T \in \mathcal{D}$, $g(T) = f(T)$; whereas for $T \notin \mathcal{D}$, $g(T) > M$.

We now show that g is submodular by using (1). Suppose that $S \subseteq T \subset T + e \subseteq E$. Define $A = D_e - \overline{S}$ and $B = D_e - \overline{T}$, so that $B \subseteq A$. Then $g(S+e) - g(S) = f(\overline{S}+e) - f(\overline{S}) + 2M(|A|-1)$ and $g(T+e) - g(T) = f(\overline{T}+e) - f(\overline{T}) + 2M(|B|-1)$. Thus $(g(S+e) - g(S)) - (g(T+e) - g(T)) = (f(\overline{S}+e) - f(\overline{S})) - (f(\overline{T}+e) - f(\overline{T})) + 2M(|A| - |B|)$. Since $B \subseteq A$, and by (1) for f, this expression is always non-negative, and so g is submodular.

Thus we can apply any SFM algorithm to g to find a minimizer of f over $\mathcal{D}$. Note that

evaluating $g(S)$ requires computing $\overline{S}$, which takes $O(m)$ time. Therefore any SFM algorithm can be used to minimize over a ring family, at the cost of replacing $O(\text{EO})$ by $O(\text{EO} + m)$ in its running time. However, we can do better. In all the combinatorial SFM algorithms, $\mathcal{E}$ is called only when computing $v^{\prec}$ for some new $\prec$. Recall that when computed w.r.t. g, $v_e^{\prec} = g(e^{\prec} + e) - g(e^{\prec}) = f(\overline{e^{\prec} + e}) - f(\overline{e^{\prec}})$. Notice that $\overline{e^{\prec} + e} = \overline{e^{\prec}} \cup D_e$. Hence as long as we've already computed $\overline{e^{\prec}}$, we can compute $v_e^{\prec}$ in $O(\text{EO} + n)$ time. In general we assume that $\text{EO} = \Omega(n)$, and so computing with g does not inflate running times.

Even without assuming that $\text{EO} = \Omega(n)$, computing v^i for g from $\prec^i$ via Greedy costs $O(n\text{EO} + n^2)$ time. Since the running times of the Schrijver, Orlin, and IO SFM algorithms is of the form $\widetilde{O}(n^{k-2}(\text{calls to Greedy}) + n^k)$ for some $k \geq 5$, replacing the $n\text{EO}$ for f by $n\text{EO} + n^2$ for g does not hurt their overall running times. However, it is unclear whether the overhead of this method would impair running times in practice, or whether it would be preferable to use some version of IFF or IO that can naturally handle ring families.

But sometimes we are interested in optimizing over other families of subsets which are not ring families. For example, in some applications we would like to optimize over non-empty sets, or sets other than E, or both; or given elements s and t, optimize over sets containing s but not t; or optimize over sets S with $|S|$ odd; etc (see Nemhauser and Wolsey [70, Section III] for typical applications). Goemans and Ramakrishnan [35] derive many such algorithms, and give a nice survey of the state of the art.

As we saw in Section 3.3.2, if we want to solve SFM over subsets containing a fixed $l \in E$, then we can consider $E' = E - l$ and $f_l(S) = f(S + l) - f(l)$, a submodular function on E'. If we want to solve SFM over subsets *not* containing a fixed $l \in E$, then we can consider $E' = E - l$ and $\hat{f}(S) = f(S)$, a submodular function on E'.

More generally, Goemans and Ramakrishnan point out that if the family of interest can be expressed as the union of a polynomial number of ring families, then we can run an SFM algorithm on each family and take the minimum answer. For example, suppose that we want to minimize over $2^E - \{\emptyset, E\}$. Define $\mathcal{F}_{st}$ to be the family of subsets of E which contain s but not t. Each $\mathcal{F}_{st}$ is a ring family, so we can apply an SFM algorithm to compute an S_{st} solving SFM on $\mathcal{F}_{st}$. Note that for an ordering of E as $s_1, s_2, \ldots, s_n$ (with $s_{n+1} = s_1$), $2^E - \{\emptyset, E\} = \bigcup_{i=1}^{n} \mathcal{F}_{s,s_{i+1}}$ (since the only non-empty set not in this union must contain all s_i, and so must equal E). Thus we can solve SFM over $2^E - \{\emptyset, E\}$ by taking the minimum of the n values $f(S_{s,s_{i+1}})$, so it costs n calls to SFM to solve this problem.

Suppose that $\mathcal{F}$ is an intersecting family. For $e \in E$ define $\mathcal{F}_e$ as the sets in $\mathcal{F}$ containing e. Then each $\mathcal{F}_e$ is a ring family, and $\mathcal{F} = \bigcup_{e \in E} \mathcal{F}_e$, so we can optimize over an intersecting family with $O(n)$ calls to SFM. If $\mathcal{C}$ is a crossing family, then for each $s \neq t \in E$, $\mathcal{C}_{st}$ is a ring family. Then for any fixed $s \in E$, $\mathcal{C} = \bigcup_{t \neq s} (\mathcal{C}_{st} \cup \mathcal{C}_{ts})$, so we can solve SFM over a crossing family in $O(n)$ calls to SFM.

5.3 SFM on Triple Families and Parity Families

Let $\mathcal{O} = \{S \subseteq E \mid |S| \text{ is odd}\}$ be the family of *odd sets*, and consider SFM over $\mathcal{O}$. This is not a ring family (nor a union of ring families), as the union of two odd sets might be even. However, it does satisfy the following property: If any three of the four sets S, T, $S \cap T$, and $S \cup T$ are not in $\mathcal{O}$ (are even), then the fourth set is also not in $\mathcal{O}$ (is even). Families of sets with this property are called *triple* families, and were considered by Grötschel, Lovász, and Schrijver [43]. A general lemma giving examples of triple families is:

Lemma 5.1 (Grötschel, Lovász, and Schrijver [43]) *Let $\mathcal{R} \subseteq 2^E$ be a ring family, and let a_e for $e \in E$ be a given set of integers. Then for any integers p and q, the family $\{S \in \mathcal{R} \mid a(S) \not\equiv q \pmod{p}\}$ is a triple family.* ∎

Let's consider applications of this where $p = 2$. If we take $\mathcal{R} = 2^E$, $a = \mathbb{1}$, and $q = 0$, then we get that $\mathcal{O}$ is a triple family; taking instead $q = 1$ we get that the family of even sets is a triple family. If we take $a = \chi(T)$ and $q = 0$, then we get that the family of subsets having odd intersection with T is a triple family. If we have two subsets $T_1, T_2 \subseteq E$ and take $q = 0$, $a_e = 1$ on $T_1 - T_2$, $a_e = -1$ on $T_2 - T_1$, and $a_e = 0$ otherwise, then we get that the family of S such that $|S \cap T_1|$ and $|S \cap T_2|$ have different parity is a triple family.

An even more general class of families is considered by Goemans and Ramakrishnan [35]. For ring family $\mathcal{R} \subseteq 2^E$, they call $\mathcal{P} \subseteq \mathcal{R}$ a *parity family* if $S, T \in \mathcal{R} - \mathcal{P}$ implies that $S \cup T \in \mathcal{P}$ iff $S \cap T \in \mathcal{P}$. An important class of parity families is given by:

Lemma 5.2 (Goemans and Ramakrishnan [35]) *Let $\mathcal{R}_1 \subseteq \mathcal{R}_2 \subseteq 2^E$ be ring families. Then $\mathcal{R}_2 - \mathcal{R}_1$ is a parity family.* ∎

Any triple family is clearly a parity family, but the converse is not true. For example, take $E = \{a, b, c\}$, $\mathcal{R}_1 = \{\{a\}, \{a, b\}, \{a, b, c\}\}$, and $\mathcal{R}_2 = 2^E$. Then $\mathcal{R}_1 \subseteq \mathcal{R}_2$ and both $\mathcal{R}_1$ and $\mathcal{R}_2$ are ring families, so the lemma implies that $\mathcal{R}_2 - \mathcal{R}_1$ is a parity family. Taking $S = \{a, b\}$ and $T = \{a, c\}$, we see that $S \in \mathcal{R}_1$, $S \cap T = \{a\} \in \mathcal{R}_1$, and $S \cup T = \{a, b, c\} \in \mathcal{R}_1$, but $T \notin \mathcal{R}_1$, so $\mathcal{R}_2 - \mathcal{R}_1$ is not a triple family.

As an application of Lemma 5.2, note that (2) implies that the union and intersection of solutions of SFM are also solutions of SFM, so the family $\mathcal{S}$ of solutions of SFM is a ring family. Thus $2^E - \mathcal{S}$ is a parity family. The next theorem shows that we can solve SFM over a parity family with $O(n^2)$ calls to SFM over a ring family, so this gives us a way of finding the second-smallest value of any submodular function.

Theorem 5.3 (Goemans and Ramakrishnan [35]) *If $\mathcal{R}$ is a ring family and $\mathcal{P} \subseteq \mathcal{R} \subseteq 2^E$ is a parity family, then we can solve SFM over $\mathcal{P}$ using $O(n^2)$ calls to SFM over ring families.* ∎

Since triple families are a special case of parity families, this gives us a tool that can solve many interesting problems: SFM over odd sets, SFM over even sets, SFM over sets having odd intersection with a fixed $T \subseteq E$, second-smallest value of $f(S)$, etc.

5.4 Symmetric SFM: Queyranne's Algorithm

A special case of SFM arises when f is *symmetric*, i.e., when $f(S) = f(E - S)$ for all $S \subseteq E$. From (2) we get that for any $S \subseteq E$, $2f(\emptyset) = 2f(E) = f(\emptyset) + f(E) \leq f(S) + f(E - S) = 2f(S)$, or $f(\emptyset) = f(E) \leq f(S)$, so that $\emptyset$ and E trivially solve SFM. But in many cases such as undirected Min Cut we would like to minimize a symmetric function over $2^E - \{\emptyset, E\}$. We could apply the procedure in Section 5.2 to solve this in $O(n)$ calls to SFM, but Queyranne [74] developed a special-purpose algorithm that is much faster. It is based on Nagamochi and Ibaraki's Algorithm [66] for finding Min Cuts in undirected graphs.

Queyranne's Algorithm (QA) is *not* based on the LPs from Section 2.4 and so does not have a current primal point y, hence it has no need of $\mathcal{I}$, v^i, and ReduceV. Somewhat similar to IFF-SP, QA maintains a partition $\mathcal{C}$ of E. As it proceeds, it gathers information that allows it to contract subsets in the partition, until $|\mathcal{C}| = 1$. If $\mathcal{S} \subseteq \mathcal{C}$, then we interpret $f(\mathcal{S})$ to be $f(\cup_{\sigma\in\mathcal{S}}\sigma)$, which is clearly submodular on $\mathcal{C}$. It uses a subroutine LeafPair$(\mathcal{C}, f, \eta)$. LeafPair builds up a set S element by element starting with element η; let S_i denote the S at iteration i. Iteration i adds an element σ of $Q = \mathcal{C} - S$ having a minimum value of $\texttt{key}_\sigma = f(S_{i-1} + \sigma) - f(\sigma)$ as the next element of S. The running time of LeafPair is clearly $O(n^2\text{EO})$.

LeafPair$(\mathcal{C}, f, \eta)$ **Subroutine for Queyranne's Algorithm**

Initialize $\tau_1 \leftarrow \eta$, $S_1 \leftarrow \{\tau_1\}$, $Q \leftarrow \mathcal{C} - \tau_1$, $k \leftarrow |\mathcal{C}|$.
For $i = 2, \ldots, k$ do
  For $\sigma \in Q$ set $\texttt{key}_\sigma = f(S_{i-1} + \sigma) - f(\sigma)$.
  Find τ_i in Q with minimum `key` value.
  Set $S_i \leftarrow S_{i-1} + \tau_i$, and $Q \leftarrow Q - \tau_i$.
Return (τ_{k-1}, τ_k).

We say that $\mathcal{S} \subset \mathcal{C}$ *separates* $\sigma, \tau \in \mathcal{C}$ if $\sigma \in \mathcal{S}$ and $\tau \notin \mathcal{S}$, or $\sigma \notin \mathcal{S}$ and $\tau \in \mathcal{S}$. Note that $\mathcal{S}$ separates σ, τ iff $\mathcal{C} - \mathcal{S}$ separates them. The name of LeafPair comes from the *cut equivalent tree* of Gomory and Hu [40], which is a compact way of representing a family of minimum cuts separating any two nodes i and j in a capacitated undirected graph. They give an algorithm that constructs a capacitated tree T on the nodes such that we can construct a Min Cut separating i from j as follows: Find a min-capacity edge e on the unique path from i to j in T. Then $T - e$ has two connected components, which form the two sides of a Min Cut separating i from j, and this cut has value the capacity of e. (Goemans and Ramakrishnan [35] point out that cut trees extend to any symmetric submodular function.) Suppose that i is a leaf of T with neighbor j in T. This implies that $\{i\}$ is a Min Cut separating i from j. We would call such a pair (j, i) a *leaf pair*. The following lemma shows that LeafPair computes a leaf pair in the more general context of SFM:

Lemma 5.4 *If* LeafPair$(\mathcal{C}, f, \eta)$ *outputs* (τ_{k-1}, τ_k)*, then* $f(\tau_k) = \min\{f(\mathcal{S}) \mid \mathcal{S} \subset \mathcal{C}$ *s.t.* $\mathcal{S}$ *separates* τ_{k-1} *and* $\tau_k\}$*.*

Proof: Suppose that we could prove that for all i, all $T \subseteq S_{i-1}$, and all $\sigma \in \mathcal{C} - S_i$ that

$$f(S_i) + f(\sigma) \leq f(S_i - T) + f(T + \sigma). \tag{29}$$

If we take $i = k - 1$, then we must have that $\sigma = \tau_k$. Then, since S_{k-1} and $\{\tau_k\}$ are complementary sets, and since $S_{k-1} - T$ and $T + \tau_k$ are complementary sets, (29) would imply that $f(\tau_k) \leq f(T + \tau_k)$. Since $T + \tau_k$ is an arbitrary set separating τ_k from τ_{k-1}, this shows that τ_{k-1} and τ_k are a leaf pair.

So we concentrate on proving (29). We use induction on i; it is trivially true for $i = 1$. Suppose that $j < i$ is the maximum index such that $\tau_j \in T$. If $j = i - 1$, then $f(S_i - T) + f(T + \sigma) = f(S_{i-1} - T + \tau_i) + f(T + \sigma)$. By the inductive assumption at index $i - 1$, element τ_i, and set $S_{i-1} - T$ we get $f(S_{i-1} - T + \tau_i) + f(T + \sigma) \geq f(S_{i-1}) + f(T + \sigma) -$

$f(T) + f(\tau_i)$. Since $[S_{i-1} \cup (T + \sigma)] = S_{i-1} + \sigma$ and $[S_{i-1} \cap (T + \sigma)] = T$, from (2) we get $f(S_{i-1}) + f(T + \sigma) - f(T) + f(\tau_i) \geq f(S_{i-1} + \sigma) + f(\tau_i)$. By the choice of τ_i in LeafPair we get $f(S_{i-1} + \sigma) + f(\tau_i) \geq f(S_{i-1} + \tau_i) + f(\sigma) = f(S_i) + f(\sigma)$, as desired.

Otherwise ($j < i-1$), by the inductive assumption at index $j+1$, element σ, and set T we get $f(S_i - T) + f(T + \sigma) \geq f(S_i - T) + f(S_{j+1}) - f(S_{j+1} - T) + f(\sigma)$. Since $[(S_i - T) \cup S_{j+1}] = S_i$ and $[(S_i - T) \cap S_{j+1}] = S_{j+1} - T$, from (2) we get $f(S_i - T) + f(S_{j+1}) - f(S_{j+1} - T) + f(\sigma) \geq f(S_i) + f(\sigma)$, as desired. ■

Queyranne's Algorithm for Symmetric SFM over $2^E - \{\emptyset, E\}$

Initialize $\mathcal{C} = E$ and η as an arbitrary element of $\mathcal{C}$.
For $i = 1, \ldots, n-1$ do
  Set $(\sigma, \tau) \leftarrow LeafPair(\mathcal{C}, f, \eta)$.
  Set $T_i \leftarrow E(\tau)$ and $m_i \leftarrow f(T_i)$.
  Contract σ and τ into a new subset of the partition.
Return T_i such that $m_i = \min\{m_j \mid j = 1, \ldots, n-1\}$.

Let S^* solve SFM for f. If S^* separates τ_{k-1} and τ_k, then $E(\tau_k)$ must also solve SFM. If S^* does not separate τ_{k-1} and τ_k, then we can contract τ_{k-1} and τ_k without harming SFM optimality. QA takes advantage of this observation to solve SFM by calling LeafPair $n-1$ times. The running time of QA is thus $O(n^3\text{EO})$. Note that QA is a fully combinatorial algorithm.

5.5 Constrained SFM can be Hard

So far we have seen that SFM remains easy when we consider it over various well-structured families of sets, or the symmetric case. However, there are other important cases of SFM with side constraints that are NP Hard to solve. One such case is *cardinality constrained* SFM, where we want to restrict to the family $\mathcal{C}_k$ of sets of size k. The s–t Min Cut problem Example 1.9 with this constraint is NP Hard [34, Problem ND17]. This example is representative of the fact that SFM often becomes hard when side constraints are added.

6 Future Directions for SFM Algorithms

The history of SFM has been that expectations have continually grown. SFM was recognized early on as being an important problem, and a big question was whether there existed a finite version of Cunningham's "augmenting path" algorithm. In 1985, Bixby, Cunningham, and Topkis [8] found such an algorithm. Then the question became whether one could get a good bound on the running time of an SFM algorithm. Also in 1985, Cunningham [13] found an algorithm with a pseudo-polynomial bound. Then the natural question was whether an algorithm with a (strongly) polynomial bound existed. In 1988, Grötschel, Lovász, and Schrijver [43] showed that the Ellipsoid Algorithm leads to a strongly polynomial SFM algorithm. However, Ellipsoid is slow, so the question became whether there existed a "combinatorial" (non-Ellipsoid) polynomial algorithm for SFM. Simultaneously in 1999, Schrijver [76], and Iwata, Fleischer, and

Fujishige [50] found quite different strongly polynomial combinatorial SFM algorithms. However, both of these algorithms need to use some multiplication and division, leading Schrijver to pose the question of whether there existed a fully combinatorial SFM algorithm. In 2002 Iwata [46] found a way to extend the IFF Algorithm to give a fully combinatorial SFM algorithm. In 2001 Fleischer and Iwata [23] found Schrijver-PR, an apparent speedup for Schrijver's Algorithm (although Vygen [84] showed in 2003 that both variants actually have the same running time), and in 2002 Iwata [48] used ideas from Schrijver's Algorithm to speed up the IFF algorithms. Finally, in 2006 Orlin [71] developed new tools to get a combinatorial strongly polynomial SFM algorithm that is faster than the strongly polynomial version of Hybrid by a factor of $O(n \log n)$, and in 2009 Iwata and Orlin developed further ideas to get a fully combinatorial SFM algorithm that is a factor $O(n \log n)$ faster than IFF-FC.

Is this the end of the road for SFM algorithms? I say "no", for two reasons:

1. The existing SFM algorithms have rather slow running times. Both variants of Schrijver's Algorithm take $O(n^7\mathrm{EO} + n^8)$ time, the strongly polynomial Hybrid Algorithm takes $O((n^6\mathrm{EO} + n^7) \log n)$ time, the weakly polynomial Hybrid Algorithm takes $O((n^4\mathrm{EO} + n^5) \log M)$ time, and Orlin's Algorithm take $O(n^5\mathrm{EO} + n^6)$ time. The progress over time shows that there may be further room for improvement. There is not yet much practical experience with any of these algorithms, but experience in other domains suggests that an $O(n^5)$ algorithm is practically useless for large instances. Therefore it is natural to ask whether we can find significantly faster SFM algorithms.

2. The existing general SFM algorithms use Cunningham's idea of verifying that the current y belongs to $B(f)$ via representing y as $\sum_{i \in \mathcal{I}} \lambda_i v^i$ for vertices v^i coming from Greedy. Naively, this is a rather brute-force way to verify that $y \in B(f)$. However, 30 years of research have not yet produced any better idea.

These two points are closely related. To keep their running times manageable, existing algorithms call ReduceV from time to time to keep $|\mathcal{I}|$ small, and ReduceV costs $O(n^3)$ per call. Thus the key to finding a faster SFM algorithm might be to avoid representing y as a convex combination of vertices. Hybrid, the fastest weakly polynomial SFM algorithm known to this point, runs in $\widetilde{O}(n^4\mathrm{EO})$ time. No formal lower bound on the complexity of SFM exists, but it is hard to imagine an SFM algorithm computing fewer than n vertices, which takes $O(n^2\mathrm{EO})$ time. It is not unreasonable to hope that an $\widetilde{O}(n^3\mathrm{EO})$ SFM algorithm exists.

How far could we go with algorithms based on Push-Relabel technology such as Schrijver's Algorithm and Iwata's Hybrid Algorithm? For networks with $\Theta(n^2)$ arcs (and the networks arising in SFM all can have $\Theta(n^2)$ arcs since each of the $O(n)$ linear orders in $\mathcal{I}$ has $O(n)$ consecutive pairs), the best known running time for a pure Push-Relabel Max Flow algorithm uses $\Theta(n^3)$ pushes (see [1]). Hence such algorithms could not be faster than $\Theta(n^3\mathrm{EO})$ without a breakthrough in Max Flow algorithms. If each such push potentially adds a new vertex to $\mathcal{I}$, then we need to call ReduceV $\Theta(n^2)$ times, for an overhead of $\Theta(n^5)$. Note that the Hybrid Algorithm, at $O((n^4\mathrm{EO} + n^5) \log M)$, comes close to this informal lower bound, losing only the $O(\log M)$ factor due to scaling, and inflating $O(n^3\mathrm{EO})$ to $O(n^4\mathrm{EO})$ since each BlockSwap takes $O(b\mathrm{EO})$ time instead of $O(\mathrm{EO})$ time.

Ideally it would be useful to have a formal lower bound stating that at least some number of oracle calls is needed to solve SFM. It is easy to see the trivial lower bound that $\Omega(n)$ calls are necessary, but so far nothing non-trivial is known.

Here are two other reasons to be dissatisfied with the current state of the art. It is hard to be completely happy with the fully combinatorial SFM algorithms, as their use of repeated subtraction or doubling to simulate multiplication and division is aesthetically unpleasant, and probably impractical. Second, we saw in Section 2.4 that the linear programs have integral optimal solutions. All the algorithms find an integral dual solution (an optimal set S solving SFM), but (when f is integer-valued) none of them directly finds an integral optimal primal solution (a $y \in B(f)$ with $y^-(E) = f(S)$ or a $y \in P(f)$ with $y(E) = f(S) + \gamma(E - S)$). We conjecture that a faster SFM algorithm exists that maintains an integral y throughout the algorithm.

One possibility for making faster SFM algorithms without using $\mathcal{I}$ is suggested by Queyranne's Algorithm for symmetric SFM. Notice that Queyranne's Algorithm does not use a $y = \sum_{i \in \mathcal{I}} \lambda_i v^i$ representation at all, which suggests that it might be possible to find a similar algorithm for general SFM. On the other hand, Queyranne's Algorithm also does not use any of the LP machinery used by the general SFM algorithms, and it does not produce anything resembling a primal solution (a $y \in B(f)$ with $y^-(E) = f(S)$). Also, as Queyranne notes in [74], general SFM is provably not reducible to symmetric SFM, and even SFM with $f(S) = s(S) - u(S)$ with s symmetric and u modular (u a vector in $\mathbb{R}^E$) is not reducible to the symmetric case.

However, we can still dream. A vague outline of an SFM algorithm not representing y as a convex combination of vertices might go like this: Start with $y = v^{\prec}$ for some linear order $\prec$. Then start doing exchanges that increase $y^-(E)$ in such a way that we are assured that y remains in $B(f)$, until we find some S with $y^-(E) = f(S)$, and we are optimal. There would be some lemma, along the lines of our proof that the α from ExchBd is at most $c(k, l; v^i)$, showing inductively that each step remains inside $B(f)$. Then the proof that the final y is in $B(f)$ would be the sequence of steps taken by the algorithm. Alternatively, one could use the framework outlined by Fujishige and Iwata [32]: Their framework needs only a combinatorial strongly polynomial separation routine that either proves that 0 belongs to a submodular polyhedron $P(\tilde{f})$ (for an $\tilde{f}$ derived from f), or gives a subset $S \subseteq E$ such that $\tilde{f}(S) < 0$ (thereby separating 0 from $P(\tilde{f})$). They show that $O(n^2)$ calls to such a routine would suffice for solving SFM. A third possibility would be to derive a polynomial bound on the number of iterations of the Min Norm Point algorithm for SFM proposed by Fujishige [29, p. 219–220] or [30], although this seems to involve other unpleasant linear algebra. A final possibility is to use the "combinatorial hull" representation proposed by Fujishige [28]. We leave these questions for future researchers.

Acknowledgments

I thank two anonymous referees, Yves Crama, Bill Cunningham, Lisa Fleischer, Satoru Fujishige, Satoru Iwata, Hervé Kerevin, Laszlo Lovász, Kazuo Murota, Maurice Queyranne, Alexander Schrijver, Bruce Shepherd, and Fabio Tardella for their substantial help with this material.

References

[1] R. K. Ahuja, T. L. Magnanti and J. B. Orlin (1993). *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, Englewood Cliffs.

[2] J-C. Anglès d'Auriac, F. Iglói, M. Preissmann, and A. Sebő (2002). Optimal Cooperation and Submodularity for Computing Potts' Partition Functions with a Large Number of States. *J. Phys. A: Math. Gen.*, **35** 6973–6983

[3] N. Baumann and M. Skutella (2006). Solving Evacuation Problems Efficiently — Earliest Arrival Flows with Multiple Sources. FOCS 2006 Proceedings, 399–408.

[4] M. A. Begen and M. Queyranne (2011). Appointment Scheduling with Discrete Random Durations. *Mathematics of Operations Research*, **36**, 240–257.

[5] A. Berman and R. J. Plemmons (1994). *Nonnegative Matrices in the Mathematical Sciences*. SIAM, Philadelphia, PA.

[6] D. P. Bertsekas (1986). Distributed Asynchronous Relaxation Methods for Linear Network Flow Problems. Working Paper, Laboratory for Information and Decision Systems, MIT (Cambridge, MA).

[7] G. Birkhoff (1967). *Lattice Theory*. Amer. Math. Soc.

[8] R. E. Bixby, W. H. Cunningham, and D. M. Topkis (1985). The Partial Order of a Polymatroid Extreme Point. *Math. of OR*, **10**, 367–378.

[9] B. V. Cherkassky and A. V. Goldberg (1997). On Implementing Push-Relabel Method for the Maximum Flow Problem. *Algorithmica*, **19**, 390–410. The `PRF` code developed here is available from `http://www.star-lab.com/goldberg/soft.html`.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein (2001). *Introduction to Algorithms*. Second Edition. MIT Press, Cambridge, MA.

[11] W. H. Cunningham (1983). Decomposition of Submodular Functions. *Combinatorica*, **3**, 53–68.

[12] W. H. Cunningham (1984). Testing Membership in Matroid Polyhedra. *JCT Series B*, **36**, 161–188.

[13] W. H. Cunningham (1985). On Submodular Function Minimization. *Combinatorica*, **3**, 185–192.

[14] DIMACS (1990). The First DIMACS International Algorithm Implementation Challenge: The Core Experiments. Available at `ftp://dimacs.rutgers.edu/pub/netflow/generalinfo/core.tex`.

[15] E. A. Dinic (1970). Algorithm for Solution of a Problem of Maximum Flow in a Network with Power Estimation. *Soviet Math. Dokl.*, **11**, 1277–1280.

[16] J. Edmonds (1970). Submodular Functions, Matroids, and Certain Polyhedra. In *Combinatorial Structures and their Applications*, R. Guy, H. Hanani, N. Sauer, and J. Schönheim, eds., Gordon and Breach, 69–87.

[17] J. Edmonds and R. Giles (1977). A Min-Max Relation for Submodular Functions on Graphs. *Ann. Discrete Math.*, **1**, 185–204.

[18] J. Edmonds and R. M. Karp (1972). Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *Journal of ACM* **19**, 248–264.

[19] T. R. Ervolina and S. T. McCormick (1993). Two Strongly Polynomial Cut Canceling Algorithms for Minimum Cost Network Flow. *Discrete Applied Mathematics*. **46**, 133–165.

[20] U. Feige, V. Mirrokni, and J. Vondrák (2007). Maximizing Non-monotone Submodular Functions. *Proceedings of 48th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, 461–471.

[21] L. K. Fleischer (2000). Recent Progress in Submodular Function Minimization. *Optima*, September 2000, 1–11.

[22] L. K. Fleischer and S. Iwata (2000). Improved Algorithms for Submodular Function Minimization and Submodular Flow. *Proceedings of the 32th Annual ACM Symposium on Theory of Computing*, 107–116.

[23] L. K. Fleischer and S. Iwata (2003). A Push-Relabel Framework for Submodular Function Minimization and Applications to Parametric Optimization. "Submodularity" special issue of *Discrete Applied Mathematics*, S. Fujishige ed., **131**, 311-322.

[24] L. K. Fleischer, S. Iwata, and S. T. McCormick (2002). A Faster Capacity Scaling Algorithm for Minimum Cost Submodular Flow. *Math. Prog.*, **92**, 119–139.

[25] S. Fujishige (1980). Lexicographically Optimal Base of a Polymatroid with respect to a Weight Vector. *Math. of OR*, **5**, 186–196.

[26] S. Fujishige (1984). Submodular Systems and Related Topics. *Math. Prog. Study*, **22**, 113–131.

[27] S. Fujishige (2002). Submodular Function Minimization and Related Topics. Discrete Mathematics and Systems Science Research Report 02-04, Osaka University, Japan.

[28] S. Fujishige (2003). Submodular Function Minimization and Related Topics. *Optimization Methods and Software*, **18**, 169–180.

[29] S. Fujishige (2005). *Submodular Functions and Optimization*. Second Edition. North-Holland.

[30] S. Fujishige, T. Hayashi, and S. Isotani (2006). The Minimum-Norm-Point Algorithm Applied to Submodular Function Minimization and Linear Programming. Research Institute for the Mathematical Sciences Preprint RIMS-1571, Kyoto University, Kyoto Japan.

[31] S. Fujishige and S. Iwata (2006). Bisubmodular Function Minimization. *SIAM J. Disc. Math.*, **19**, 1065–1073.

[32] S. Fujishige and S. Iwata (2002). A Descent Method for Submodular Function Minimization. *Math. Prog.*, **92**, 387–390

[33] H. N. Gabow (1985). Scaling Algorithms for Network Problems. *J. of Computer and Systems Sciences*, **31** 148–168.

[34] M. R. Garey and D. S. Johnson (1979). *Computers and Intractability, A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York.

[35] M. X. Goemans and V. S. Ramakrishnan (1995). Minimizing Submodular Functions over Families of Sets. *Combinatorica*, **15**, 499–513.

[36] A. V. Goldberg and S. Rao (1998). Beyond the Flow Decomposition Barrier. *Journal of ACM* 45 753–797.

[37] A. V. Goldberg and R.E. Tarjan (1988). A New Approach to the Maximum Flow Problem. *JACM*, **35**, 921–940.

[38] A. V. Goldberg and R. E. Tarjan (1990). Finding Minimum-Cost Circulations by Successive Approximation. *Mathematics of Operations Research*, **15**, 430–466.

[39] D. Goldfarb and Z. Jin (1999). A New Scaling Algorithm for the Minimum Cost Network Flow Problem. *Operations Research Letters*, **25** 205–211.

[40] R. E. Gomory and T. C Hu (1961). Multiterminal Network Flows. *SIAM J. on Applied Math.*, **9**, 551–570.

[41] F. Granot and A. F. Veinott, Jr. (1985). Substitutes, Complements, and Ripples in Network Flows. *Math. of OR*, **10**, 471–497.

[42] M. Grötschel, L. Lovász, and A. Schrijver (1981). The Ellipsoid Algorithm and its Consequences in Combinatorial Optimization. *Combinatorica*, **1**, 499–513.

[43] M. Grötschel, L. Lovász, and A. Schrijver (1988). *Geometric Algorithms and Combinatorial Optimization*. Springer-Verlag.

[44] W. T. Huh and R. O. Roundy (2005). A Continuous-Time Strategic Capacity Planning Model. *Naval Research Logistics*, **52**, 329–343.

[45] S. Iwata (1997). A Capacity Scaling Algorithm for Convex Cost Submodular Flows. *Math. Programming*, **76**, 299–308.

[46] S. Iwata (2002). A Fully Combinatorial Algorithm for Submodular Function Minimization. *J. Combin. Theory Ser. B*, **84**, 203–212; a corrected version is available at `http://www.sr3.t.u-tokyo.ac.jp/~iwata/`.

[47] S. Iwata (2002). Submodular Function Minimization — Theory and Practice. Talk given at Workshop in Combinatorial Optimization at Oberwolfach, Germany, November 2002.

[48] S. Iwata (2003). A Faster Scaling Algorithm for Minimizing Submodular Functions. *SIAM J. on Computing*, **32**, 833–840.

[49] S. Iwata (2008). Submodular Function Minimization. *Mathematical Programming*, **112**, 45–64.

[50] S. Iwata, L. Fleischer, and S. Fujishige (2001). A Combinatorial, Strongly Polynomial-Time Algorithm for Minimizing Submodular Functions. *J. ACM*, **48**, 761–777.

[51] S. Iwata, S. T. McCormick, and M. Shigeno (2005). A Strongly Polynomial Cut Canceling Algorithm for the Submodular Flow Problem. *SIAM J. on Discrete Math.*, **19**, 304–320.

[52] S. Iwata and J. B. Orlin (2009). A Simple Combinatorial Algorithm for Submodular Function Minimization. Technical report; an extended abstract appears in *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA 2009), pp. 1230–1237.

[53] P. Jeavons, D. Cohen, and M. C. Cooper (1998). Constraints, Consistency and Closure. *Artificial Intelligence*, **101** 251–265.

[54] A. Krause and D. Golovin (2013). Submodular Function Maximization. Chapter in *Tractability: Practical Approaches to Hard Problems* (to appear), Cambridge University Press.

[55] A. Krause and C. Guestrin (2011). Submodularity and its Applications in Optimized Information Gathering. *ACM Transactions on Intelligent Systems and Technology*, **2**, Article 32.

[56] M. Laurent (1997). The Max-Cut Problem. In *Annotated Bibliographies in Combinatorial Optimization*, M. Dell'Amico, F. Maffioli, and S. Martello, eds., Wiley, Chichester.

[57] E. L. Lawler and C. U. Martel (1982). Computing Maximal Polymatroidal Network Flows. *Math. Oper. Res.*, **7**, 334–347.

[58] L. Lovász (1983). Submodular Functions and Convexity. In *Mathematical Programming — The State of the Art*, A. Bachem, M. Grötschel, B. Korte eds., Springer, Berlin, 235–257.

[59] L. Lovász (2002). Email reply to query from S. T. McCormick, 6 August 2002.

[60] M. E. Lübbecke and J. Desrosiers (2005). Selected Topics in Column Generation. *Operations Research*, **53**, 1007–1023.

[61] S. T. McCormick (2006). Submodular Function Minimization. Chapter 7 in the *Handbook on Discrete Optimization*, Elsevier, K. Aardal, G. Nemhauser, and R. Weismantel, eds., 321–391. See `http://www.elsevier.com/wps/find/bookdescription.cws_home/699541/description`

[62] S. T. McCormick and S. Fujishige (2007). Strongly Polynomial and Fully Combinatorial Algorithms for Bisubmodular Function Minimization. Working paper, Sauder School of Business, University of British Columbia, Vancouver, BC; an extended abstract appears in *Proceedings of Nineteenth SODA* (2008), 44–53.

[63] N. Megiddo (1983). Applying parallel computation algorithms in the design of serial algorithms. *Journal of ACM* 30 852–865.

[64] K. Murota (1998). Discrete Convex Analysis. *Math. Programming*, **83**, 313–371.

[65] K. Murota (2003). *Discrete Convex Analysis*. SIAM Monographs on Discrete Mathematics and Applications, Society for Industrial and Applied Mathematics, Philadelphia.

[66] H. Nagamochi and T. Ibaraki (1992). Computing Edge Connectivity in Multigraphs and Capacitated Graphs. *SIAM J. on Discrete Math.*, **5**, 54–66.

[67] K. Nagano (2005). A Strongly Polynomial Algorithm for Line Search in Submodular Polyhedra. *Proceedings of the 4th Japanese-Hungarian Symposium on Discrete Mathematics and Its Applications*, Budapest, Hungary, 234–242.

[68] K. Nagano (2007). On Convex Minimization over Base Polytopes. *Proceedings of IPCO 12*, M. Fischetti and D. Williamson, eds., Ithaca, NY, 252–266.

[69] K. Nagano (2007). A Faster Parametric Submodular Function Minimization Algorithm and Applications. Technical Report, Department of Mathematical Informatics, University of Tokyo, Japan.

[70] G. L. Nemhauser and L. A. Wolsey (1988). *Integer and Combinatorial Optimization.* Wiley, New York.

[71] J.B. Orlin (2007). A Faster Strongly Polynomial Algorithm for Submodular Function Minimization. *Proceedings of IPCO 12*, M. Fischetti and D. Williamson, eds., Ithaca, NY, 240–251.

[72] J-C. Picard and M. N. Queyranne (1982). Selected Applications of Minimum Cuts in Networks. *INFOR*, **20**, 394–422.

[73] M. N. Queyranne (1980). Theoretical Efficiency of the Algorithm Capacity for the Maximum Flow Problem. *Mathematics of Operations Research*, **5**, 258–266.

[74] M. N. Queyranne (1998). Minimizing Symmetric Submodular Functions. *Math. Prog.*, **82**, 3–12.

[75] P. Schönsleben (1980). *Ganzzahlige Polymatroid-Intersektions Algorithmen.* Ph.D. Dissertation, ETH Zürich.

[76] A. Schrijver (2000). A Combinatorial Algorithm Minimizing Submodular Functions in Strongly Polynomial Time. *J. Combin. Theory Ser. B* **80**, 346–355.

[77] A. Schrijver (2003). *Combinatorial Optimization: Polyhedra and Efficiency.* Springer, Berlin.

[78] J. G. Shanthikumar and D. D. Yao (1992). Multiclass Queueing Systems: Polymatroid Structure and Optimal Scheduling Control. *Operations Research*, **40**, S293–299.

[79] Z-J. M. Shen, C. Coullard, and M. S. Daskin (2003). A Joint Location-Inventory Model. *Transportation Science*, **37**, 40–55.

[80] É. Tardos (1985). A Strongly Polynomial Minimum Cost Circulation Algorithm. *Combinatorica*, **5**, 247–256.

[81] É. Tardos, C. A. Tovey, and M. A. Trick (1986). Layered Augmenting Path Algorithms. *Math. Oper. Res.*, **11**, 362–370.

[82] D. M. Topkis (1978). Minimizing a Submodular Function on a Lattice. *Operations Research*, **26**, 305–321.

[83] D. M. Topkis (1998). *Supermodularity and Complementarity.* Princeton University Press, Princeton, NJ.

[84] J. Vygen (2003). A Note on Schrijver's Submodular Function Minimization Algorithm. *JCT B*, **88**, 399–402.

[85] D. J. A. Welsh (1976). *Matroid Theory*, Academic Press, London.

[86] P. Wolfe (1976). Finding the Nearest Point in a Polytope. *Math. Prog.*, **11**, 128–149.