A Simple Combinatorial Algorithm for Submodular Function Minimization

Satoru Iwata * James B. Orlin †

Abstract

This paper presents a new simple algorithm for minimizing submodular functions. For integer valued submodular functions, the algorithm runs in $O(n^6 \mathrm{EO} \log nM)$ time, where n is the cardinality of the ground set, M is the maximum absolute value of the function value, and EO is the time for function evaluation. The algorithm can be improved to run in $O((n^4\mathrm{EO}+n^5)\log nM)$ time. The strongly polynomial version of this faster algorithm runs in $O((n^5\mathrm{EO}+n^6)\log n)$ time for real valued general submodular functions. These are comparable to the best known running time bounds for submodular function minimization. The algorithm can also be implemented in strongly polynomial time using only additions, subtractions, comparisons, and the oracle calls for function evaluation. This is the first fully combinatorial submodular function minimization algorithm that does not rely on the scaling method.

1 Introduction

Let V be a finite nonempty set of cardinality n. A function f defined on the subsets of V is *submodular* if it satisfies

$$f(X) + f(Y) \geq f(X \cap Y) + f(X \cup Y), \qquad \forall X, Y \subseteq V.$$

Submodular functions are discrete analogues of concave functions, but they have algorithmic properties that behave similarly to convex functions [13]. Examples include cut capacity functions, matroid rank functions, and entropy functions.

The first polynomial-time algorithm for submodular function minimization is due to Grötschel, Lovász, and Schrijver [7]. A strongly polynomial algorithm has also been described in [8]. These algorithms employ the ellipsoid method.

Recently, combinatorial strongly polynomial algorithms have been developed by [4, 10, 12, 16, 17]. These algorithms build on the works of Cunningham [1, 2]. The current best strongly polynomial bound due to [16] is $O(n^5\mathrm{EO} + n^6)$, where EO is the time for function evaluation.

In this paper, we present a simple combinatorial algorithm for submodular function minimization. The initial variant of the algorithm minimizes integer-valued submodular functions in $O(n^6\mathrm{EO}\log nM)$ time, where M is the maximum absolute value of the function values. The algorithm achieves this complexity without relying on the scaling technique nor on Gaussian elimination. It does not rely on augmenting paths or flow techniques either. Instead, it works with distance labels used in [16] and new potential functions.

With the aid of the Gaussian elimination procedure, the algorithm can be improved to run in $O((n^4\mathrm{EO} + n^5)\log nM)$ time, which matches the best weakly polynomial bound of [10] for instances in which $\log n = O(\log M)$. An advantage of the present algorithm over the previous scaling algorithms is that it obtains the unique maximal minimizer, which is often required in applications of submodular function minimization [4, 6, 15]. The strongly polynomial version of this algorithm runs in $O((n^5\mathrm{EO}+n^6)\log n)$ time, which is quite close to the best known strongly polynomial bound of [16].

These combinatorial algorithms perform multiplications and divisions, although the definition of submodular functions does not involve those arithmetic operations. Schrijver [17] asks if one can minimize submodular functions in strongly polynomial time using only additions, subtractions, comparisons, and oracle calls for the function value. Such an algorithm is called "fully combinatorial." This problem was settled in [9] by developing a fully combinatorial variant of the strongly polynomial algorithm of [12]. A faster version, which runs in $O(n^8 \log^2 n\mathrm{EO})$ time, is presented in [10].

The new algorithm as well as its strongly polynomial version can be turned into fully combinatorial algorithms. The running time bounds of the resulting algorithms are $O(n^6(\mathrm{EO} + \log nM)\log nM)$ and $O((n^7\mathrm{EO} + n^8)\log n)$. These are the first fully combinatorial algorithms that do not rely on the scaling method. Moreover, the latter algorithm improves the best previous bound by a factor of n.

The outline of this paper is as follows. Section 2 provides preliminaries on submodular functions and

*Research Institute for Mathematical Sciences, Kyoto University, Kyoto 606-8502, Japan (iwata@kurims.kyoto-u.ac.jp).

†Sloan School of Management, MIT, Cambridge, MA 02139, USA (jorlin@mit.edu).

base polyhedra. In Section 3, we present our prototype algorithm that runs in weakly polynomial time. In Section 4, we present the faster version of our weakly polynomial algorithm. Section 5 presents an extension to submodular function minimization on ring families, which is then used in the strongly polynomial algorithm presented in Section 6. Finally, in Section 7, we discuss fully combinatorial implementations of these algorithms.

2 Base polyhedra

This section provides preliminaries on submodular functions. See [5, 11, 13, 14] for more details and general background.

For a vector $x \in \mathbf{R}^V$ and a subset $Y \subseteq V$, we denote $x(Y) = \sum_{u \in Y} x(u)$. We also denote by x^+ and x^- the vectors in $\mathbf{R}^V$ with $x^+(u) = \max\{x(u), 0\}$ and $x^-(u) = \min\{x(u), 0\}$, respectively. For each $u \in V$, let χ_u denote the vector in $\mathbf{R}^V$ with $\chi_u(u) = 1$ and $\chi_u(v) = 0$ for $v \in V \setminus \{u\}$. Throughout this paper, we adopt the convention that the maximum over the empty set is $-\infty$ and the minimum over the empty set is ∞.

For a submodular function $f : 2^V \to \mathbf{R}$ with $f(\emptyset) = 0$, we consider the *submodular polyhedron*

$$\mathrm{P}(f) = \{x \mid x \in \mathbf{R}^V, \forall Y \subseteq V : x(Y) \leq f(Y)\}$$

and the *base polyhedron*

$$\mathrm{B}(f) = \{x \mid x \in \mathrm{P}(f), x(V) = f(V)\}.$$

A vector in $\mathrm{B}(f)$ is called a *base*. In particular, an extreme point of $\mathrm{B}(f)$ is called an *extreme base*. An extreme base can be computed by the greedy algorithm of Edmonds [3] and Shapley [18] as follows.

Let $L = (v_1, \cdots, v_n)$ be a linear ordering of V. For any $v_j \in V$, we denote $L(v_j) = \{v_1, \cdots, v_j\}$. The greedy algorithm with respect to L generates an extreme base $y_L \in \mathrm{B}(f)$ by

$$y_L(u) = f(L(u)) - f(L(u) \setminus \{u\}). \tag{2.1}$$

Conversely, any extreme base can be obtained in this way with an appropriate linear ordering.

For any base $x \in \mathrm{B}(f)$ and any subset $Y \subseteq V$, we have $x^-(V) \leq x(Y) \leq f(Y)$. The following theorem shows that these inequalities are in fact tight for appropriately chosen x and Y.

THEOREM 2.1. *For a submodular function $f : 2^V \to \mathbf{R}$, we have*

$$\max\{x^-(V) \mid x \in \mathrm{B}(f)\} = \min\{f(Y) \mid Y \subseteq V\}.$$

Moreover, if f is integer-valued, then the maximizer x can be chosen from among integral bases.

This theorem is immediate from the vector reduction theorem on polymatroids due to Edmonds [3]. It has motivated combinatorial algorithms for minimizing submodular functions. If X and Y are minimizers of a submodular function f, then both $X \cap Y$ and $X \cup Y$ minimize f as well. Therefore, a submodular function has a unique maximal/minimal minimizer.

3 A new weakly polynomial algorithm

This section presents a combinatorial algorithm for minimizing a submodular function $f : 2^V \to \mathbf{Z}$. In order to measure the running time, we use $M = \max\{|f(X)| \mid X \subseteq V\}$.

The algorithm keeps a set Λ of linear orderings of the elements in V. We denote $v \preceq_L u$ if v precedes u in a linear ordering L or $v = u$. Each linear ordering L generates an extreme base $y_L \in \mathrm{B}(f)$ by the greedy algorithm. The algorithm also keeps a base $x \in \mathrm{B}(f)$ as a convex combination $x = \sum_{L \in \Lambda} \lambda_L y_L$ of the extreme bases. Initially, $\Lambda = \{L\}$ with an arbitrary linear ordering L and $\lambda_L = 1$.

The algorithm keeps a label $d_L : V \to \mathbf{Z}$ for each $L \in \Lambda$. The set of labels is *valid* if the following properties are satisfied.

- If $x(u) \leq 0$, then $d_L(u) = 0$ for any $L \in \Lambda$.
- If $u \preceq_L v$, then $d_L(u) \leq d_L(v)$.
- For any $L, K \in \Lambda$ and $u \in V$, $|d_L(u) - d_K(u)| \leq 1$.

For each $u \in V$, we denote $d_{\min}(u) = \min\{d_L(u) \mid L \in \Lambda\}$.

The set of labels is said to have a *gap* at level $k > 0$ if there is an element $v \in V$ with $d_{\min}(v) = k$ and no element $u \in V$ with $d_{\min}(u) = k - 1$. The following lemma is comparable to [16, Lemma 2].

LEMMA 3.1. *Suppose that the set of labels is valid. If there is a gap at level k, then an arbitrary minimizer of f does not contain any element $v \in V$ with $d_{\min}(v) \geq k$.*

Proof. Consider the set $Y = \{v \mid u \in V, d_{\min}(u) < k\}$. For any triple of $u \in Y$, $v \in V \setminus Y$ and $L \in \Lambda$, we have $d_L(u) \leq d_{\min}(u) + 1 < k \leq d_{\min}(v) \leq d_L(v)$, which implies $u \preceq_L v$. Then Y satisfies $y_L(Y) = f(Y)$ for each $L \in \Lambda$, and hence $x(Y) = f(Y)$. Let $X \subseteq V$ be an arbitrary subset with $X \not\subseteq Y$. Since $x(v) > 0$ for any $v \in V \setminus Y$, we have $x(Y) < x(X \cup Y) \leq f(X \cup Y)$. Thus we obtain $f(Y) < f(X \cup Y)$. By the submodularity of f, this implies $f(X) > f(X \cap Y)$. Therefore, any minimizer of f must be a subset of Y.

The algorithm keeps a subset $W \subseteq V$, starting with $W = V$. Whenever there is a gap at some level k, the

algorithm identifies the set $T = \{v \mid v \in W, d_{\min}(v) \geq k\}$ and puts $d_L(v) = n$ for all $v \in T$ and $L \in \Lambda$. The set of labels remains valid. Then the algorithm deletes T from W. Lemma 3.1 guarantees that the resulting W includes all the minimizers of f.

In each iteration, the algorithm computes $\eta = \max\{x(v) \mid v \in W\}$. If $\eta < 1/n$, then the algorithm returns W as the unique maximal minimizer of f. Otherwise, it computes $\delta = \eta/4n$, and finds a value μ with $\delta \leq \mu < \eta$ such that there is no element $v \in W$ that satisfies $\mu - \delta < x(v) < \mu + \delta$. The interval $[0, \eta]$ can be partitioned into $2n$ segments of length 2δ. Obviously, some of these segments do not contain the value of $x(v)$ for any $v \in V$. The midpoint of one of such segments is a desired value of μ. The algorithm then finds an element u that attains the minimum value of $d_{\min}$ among those satisfying $x(u) > \mu$. Let ℓ be this minimum value and let $L \in \Lambda$ be a linear ordering with $d_L(u) = \ell$. The algorithm applies $\mathsf{New_Permutation}(L, \mu, \ell)$ to obtain a linear ordering L', and then it applies $\mathsf{Push}(L, L')$.

The procedure $\mathsf{New_Permutation}(L, \mu, \ell)$ yields a linear ordering L' from L by the following rule. The set $S = \{v \mid v \in V, d_L(v) = \ell\}$ are consecutive elements in L. The set S can be partitioned into $R = \{v \mid v \in S, x(v) > \mu\}$ and $Q = \{v \mid v \in S, x(v) < \mu\}$. The procedure moves the elements in R to the place after the elements in Q without changing the relative orders in Q and in R. The labeling $d_{L'}$ for L' is given by $d_{L'}(v) = d_L(v)$ for $v \in V \setminus R$ and $d_{L'}(v) = d_L(v) + 1$ for $v \in R$. Obviously, $d_{L'}(u) \leq d_{L'}(v)$ holds if $u \preceq_{L'} v$. For any $v \in R$ and $K \in \Lambda$, we have $\ell \leq d_K(v) \leq \ell + 1$, which implies $d_{L'}(v) - 1 \leq d_K(v) \leq d_{L'}(v)$. Thus the set of labels remains valid.

The procedure $\mathsf{Push}(L, L')$ increases $\lambda_{L'}$ and decreases λ_L by the same amount α, which is chosen to be the largest value that is at most λ_L and so that after the modification $x(v) \leq \mu$ for $v \in Q$ and $x(v) \geq \mu$ for $v \in R$. This means $\alpha = \min\{\lambda_L, \beta\}$, where

$$\beta = \min\{\frac{x(v) - \mu}{y_L(v) - y_{L'}(v)} \mid v \in Q \cup R, y_L(v) \neq y_{L'}(v)\}.$$

If α is chosen to be λ_L, then we call this push operation a *saturating push*. Otherwise, it is a *nonsaturating push*.

We are now ready to describe the new algorithm.

Algorithm SFM

Step 0: Let L be an arbitrary linear ordering. Compute an extreme base y_L by the greedy algorithm. Put $x := y_L$, $\lambda_L := 1$, $\Lambda := \{L\}$, $d_L(u) := 0$ for $u \in V$, and $W := V$.

Step 1: Compute $\eta := \max\{x(v) \mid v \in W\}$. If $\eta < 1/n$, then return W as the unique maximal minimizer of f. Find μ with $\delta \leq \mu < \eta$ such that there is no $u \in W$ with $\mu - \delta < x(u) < \mu + \delta$, where $\delta := \eta/4n$.

Step 2: Find $u := \arg\min\{d_{\min}(v) \mid v \in W, x(v) > \mu\}$, and put $\ell := d_{\min}(u)$. Let $L \in \Lambda$ be a linear ordering with $d_L(u) = d_{\min}(u)$.

Step 3: Obtain a linear ordering L' by $\mathsf{New_Permutation}(L, \mu, \ell)$ and apply $\mathsf{Push}(L, L')$.

Step 4: If there is a gap at some level k, then put $T := \{v \mid v \in W, d_{\min}(v) \geq k\}$, $W := W \setminus T$, and $d_L(v) := n$ for all $v \in T$ and $L \in \Lambda$. Go to Step 1.

Whenever the algorithm updates W, the new W satisfies $y_L(W) = f(W)$ for each $L \in \Lambda$, and hence $x(W) = f(W)$. Once an element $v \in V$ is deleted from W, then the algorithm will never change $x(v)$. Thus, $x(W) = f(W)$ holds throughout the algorithm.

When the algorithm terminates with $\eta < 1/n$, we have $x(v) < 1/n$ for $v \in W$ and $x(v) > 0$ for $v \in V \setminus W$. Hence, $x^-(V) > x(W) - |W|/n \geq x(W) - 1 = f(W) - 1$ holds. For any $Y \subseteq V$, we have $f(Y) \geq x^-(V) > f(W) - 1$, which implies by the integrality of f that W minimizes f. Furthermore, since W is shown to include all the minimizers, W itself must be the unique maximal minimizer of f.

We now analyze the running time of this algorithm. The number of linear orderings the algorithm keeps in Λ increases only when the algorithm performs a nonsaturating push. In order to bound the number of nonsaturating pushes, we introduce a potential

$$\Phi(x) = \sum_{v \in W} x^+(v)^2$$

and show its geometric convergence.

LEMMA 3.2. *Suppose that a nonsaturating push moves a base x to x'. Then we have $\Phi(x) - \Phi(x') \geq \Phi(x)/16n^3$.*

Proof. Note that $x(v) \leq x'(v)$ for $v \in Q$ and $x(v) \geq x'(v)$ for $v \in R$. Let Q^+ denote the set $Q^+ = \{v \mid v \in Q, x'(v) > 0\}$. Then we have

$$\begin{aligned}
\Phi(x) - \Phi(x') &= \sum_{v \in R \cup Q^+} [x^+(v)^2 - x'(v)^2] \\
&= \sum_{v \in R \cup Q^+} (x^+(v) - x'(v))(x^+(v) + x'(v)) \\
&\geq \sum_{v \in Q^+} (x^+(v) - x'(v))(2\mu - \delta) \\
&\quad + \sum_{v \in R} (x(v) - x'(v))(2\mu + \delta)
\end{aligned}$$

$$\geq \sum_{v \in Q^+} (x'(v) - x^+(v))\delta + \sum_{v \in R} (x(v) - x'(v))\delta,$$

where the last inequality follows from $x(Q^+ \cup R) \geq x'(Q^+ \cup R)$. Since $|x^+(v) - x'(v)| \geq \delta$ for some $v \in R \cup Q^+$, we obtain $\Phi(x) - \Phi(x') \geq \delta^2$. On the other hand, $\Phi(x) \leq n\eta^2 = 16n^3\delta^2$ holds. Thus we have $\Phi(x) - \Phi(x') \geq \Phi(x)/16n^3$.

At the start of this algorithm, $\Phi(x) \leq 4nM^2$ holds. Therefore, by Lemma 3.2, after $O(n^3 \log nM)$ nonsaturating pushes, $\Phi(x)$ becomes smaller than $1/n^2$. Then η must be smaller than $1/n$ and the algorithm terminates. This implies that the number of linear orderings in Λ is also $O(n^3 \log nM)$.

We now consider another potential

$$\Gamma(\Lambda) = \sum_{L \in \Lambda} \sum_{v \in V} [n - d_L(v)].$$

Each saturating push decreases $\Gamma(\Lambda)$ by at least one. Each nonsaturating push leads to an increase in the size of Λ, and increases $\Gamma(\Lambda)$ by at most n^2. Thus the total increase in $\Gamma(\Lambda)$ over all iterations is $O(n^5 \log nM)$, and the number of saturating pushes is $O(n^5 \log nM)$. Since each execution of Push requires $O(n)$ oracle calls for function evaluation, the algorithm runs in $O(n^6 \mathrm{EO} \log nM)$ time.

THEOREM 3.1. *Algorithm* SFM *finds the unique maximal minimizer in* $O(n^6 \mathrm{EO} \log nM)$ *time.*

4 A faster weakly polynomial algorithm

This section presents a faster version of the algorithm SFM for minimizing a submodular function $f : 2^V \to \mathbf{Z}$.

The algorithm differs from the algorithm SFM in two points. The first one is the use of Reduce(Λ) that computes an expression of x as a convex combination of affinely independent extreme bases chosen from the currently used ones. This procedure is used in common with other combinatorial algorithms for submodular function minimization [1, 2, 4, 10, 12, 16, 17].

The other difference is in the way of selecting μ in Step 1 of SFM. The new algorithm employs the procedure Update(μ) that replaces μ by the smallest value of μ' with $\mu' \geq \mu$ such that there is no $u \in W$ with $\mu' - \delta < x(u) < \mu' + \delta$, where $\delta = \eta/4n$.

The entire algorithm is described as follows.

Algorithm SFMwave

Step 0: Let L be an arbitrary linear ordering. Compute an extreme base y_L by the greedy algorithm. Put $x := y_L$, $\lambda_L := 1$, $\Lambda := \{L\}$, $d_L(u) := 0$ for $u \in V$, and $W := V$.

Step 1: Compute $\eta := \max\{x(v) \mid v \in W\}$. If $\eta < 1/n$, then return W as the unique maximal minimizer of f. Put $\mu := \delta$, where $\delta := \eta/4n$.

Step 2: Repeat the following (2-1) to (2-3) until $\mu > \eta$ or $d_{\min}(v)$ increases for some element $v \in W$.

- **(2-1)** If $x(v) = \mu$ for some $v \in W$, then apply Update(μ).
- **(2-2)** Find an element $u := \arg\min\{d_{\min}(v) \mid v \in W, x(v) > \mu\}$, and put $\ell := d_{\min}(u)$. Let $L \in \Lambda$ be a linear ordering with $d_L(u) = d_{\min}(u)$.
- **(2-3)** Obtain a linear ordering L' by New_Permutation(L, μ, ℓ) and apply Push(L, L').

Step 3: If there is a gap at some level k, then put $T := \{v \mid v \in W, d_{\min}(v) \geq k\}$, $W := W \setminus T$, and $d_L(v) := n$ for all $v \in T$ and $L \in \Lambda$. Apply Reduce(Λ). Go to Step 1.

Each outer iteration is called a *wave*. A wave starts with $\mu = \delta$, and the value of μ never decreases in a wave. As a result of Update(μ), the value of μ increases exactly by δ unless there is some $v \in W$ such that $\mu < x(v) < \mu + 2\delta$. Therefore, Update($\mu$) is applied at most $4n$ times in a wave. In addition, if $d_{\min}(v)$ does not change at any $v \in W$ in the wave, the Update(μ) is applied at least n times.

Suppose that x is the base at the time when current μ is selected, and that x' is the base at the time when the sequence of pushes ends with $x'(v) = \mu$ for some $v \in W$. It follows from the same argument as in the proof of Lemma 3.2 that the potential function Φ has decreased by at least a factor of $1/16n^3$, namely $\Phi(x) - \Phi(x') \geq \Phi(x)/16n^3$. Therefore, if x is the base at the beginning of a wave and x'' is the base at the end with $\mu > \eta$, then we have

$$\Phi(x'') \leq \left(1 - \frac{1}{16n^3}\right)^n \Phi(x).$$

Therefore, the number of waves that do not change $d_{\min}$ is $O(n^2 \log nM)$. The changes in $d_{\min}$ occur $O(n^2)$ times throughout the algorithm. Thus the total number of waves in the entire algorithm is $O(n^2 \log nM)$.

Since Update(μ) is applied at most $4n$ times in a wave, the number of nonsaturating pushes during a wave is $O(n)$. After at most $|\Lambda| = O(n)$ consecutive saturating pushes, the algorithm performs a nonsaturating push or $d_{\min}(u)$ increases for the element $u \in W$

selected in Step 2-2. Thus the number of saturating pushes during a wave is $O(n^2)$.

During a wave, we may create as many as $O(n^2)$ different permutations that get added to Λ. Potentially, each permutation can take $O(n\mathrm{EO})$ steps to create, and thus the bound from this is $O(n^3\mathrm{EO})$ per wave. However, we will show that the time to create all permutations is $O(n^2\mathrm{EO})$ per wave.

The time to create permutations that lead to non-saturating pushes is $O(n^2\mathrm{EO})$ per wave. We now focus on saturating pushes. The algorithm creates a new permutation L' from L by modifying the position of elements in S. We refer to L' as a child of L, and further children of L' are called descendents of L. The time to perform $\mathsf{Push}(L, L')$ is $O(|S|\,\mathrm{EO})$. Since $d_{L'}(v) = d_L(v) + 1$ for all $v \in S$ with $x(v) > \mu$, these elements will not change positions in any descendents of L' during the wave as long as $d_{\min}(v)$ does not increase for any $v \in W$. Moreover, the elements in S with $x(v) < \mu$ will not change positions in any descendent of L' in the wave. Since the algorithm keeps $O(n)$ permutations, the time to perform saturating pushes is $O(n^2\mathrm{EO})$ per wave.

At the end of each wave, the algorithm applies $\mathsf{Reduce}(\Lambda)$, which takes $O(n^3)$ time. Thus each wave takes $O(n^2\mathrm{EO} + n^3)$ time, and the total running time of the entire algorithm is $O((n^4\mathrm{EO} + n^5)\log nM)$.

5 SFM on ring families

This section is devoted to minimization of submodular functions defined on ring families. A similar method has been presented in [16, §8].

A family $\mathcal{D} \subseteq 2^V$ is called a ring family if $X \cap Y \in \mathcal{D}$ and $X \cup Y \in \mathcal{D}$ for any pair of $X, Y \in \mathcal{D}$. A compact representation of $\mathcal{D}$ is given as follows. Let $D = (V, F)$ be a directed graph with the arc set F. A subset $Y \subseteq V$ is called an *ideal* of D if no arc leaves Y in D. Then the set of ideals of D forms a ring family. Conversely, any ring family $\mathcal{D} \subseteq 2^V$ with $\emptyset, V \in \mathcal{D}$ can be represented in this way. Moreover, contracting strongly connected components of D to single vertices, we may assume without loss of generality that D is acyclic. Furthermore, if $(u, v) \in F$ and $(v, w) \in F$, adding an arc (u, w) to F does not change the set of ideals in D. Thus, we may assume that D is transitive. A linear ordering L of V is said to be consistent if $v \preceq_L u$ holds for any $(u, v) \in F$.

Let $\mathcal{D}$ be a ring family represented by a transitive directed acyclic graph $D = (V, F)$. For each vertex $v \in V$, let $R(v)$ denote the set of vertices reachable from v in D. For minimizing a submodular function f on $\mathcal{D}$, we introduce another submodular function $\widehat{f}$ defined on all the subsets of V. Consider first a vector z given by

$$z(v) = f(R(v)) - f(R(v) \setminus \{v\})$$

for each $v \in V$. For any subset $X \subseteq V$, let $\overline{X}$ denote the largest member of $\mathcal{D}$ contained in X. Let $\widehat{f}$ be the function on 2^V defined by

$$\widehat{f}(X) = f(\overline{X}) + z^+(X \setminus \overline{X}).$$

Then it can be shown that

$$\widehat{f}(X) = \min\{f(Y) + z^+(X \setminus Y) \mid Y \subseteq X, Y \in \mathcal{D}\}.$$

Therefore, $\widehat{f}$ is a submodular function. Note that $\widehat{f}(X) \geq f(\overline{X})$ holds for any $X \subseteq V$. In particular, the inequality is tight for $X \in \mathcal{D}$. Thus, minimizing f in $\mathcal{D}$ is equivalent to minimizing $\widehat{f}$ among all the subsets of V.

When applying SFM or $\mathsf{SFMwave}$ to $\widehat{f}$, one needs to compute the function values of $\widehat{f}$. The function value is required in the process of finding the extreme base. For example, in the initialization step, we need to compute an extreme base y_L for an arbitrary linear ordering L. To accomplish this efficiently, we compute $y_L(v)$ in the reverse order of L. Apparently, we have $\overline{V} = V$. If $\overline{L(v)}$ is already known, then it is easy to find $\overline{L(v) \setminus \{v\}}$. In fact, all we have to do is to delete the vertex v and vertices u with $(u, v) \in F$ from $\overline{L(v)}$. This requires only $O(n)$ time. As a result, finding an extreme base y_L can be done in $O(n\mathrm{EO} + n^2)$ time, where EO is the time for evaluating the function value of f. Function evaluation of $\widehat{f}$ in $\mathsf{New_Permutation}$ can be implemented in a similar way, so that the amortized complexity for computing the function value of $\widehat{f}$ is $O(\mathrm{EO} + n)$. Thus, algorithms SFM and $\mathsf{SFMwave}$ are extended to submodular function minimization on ring families. The resulting running time bounds are $O((n^6\mathrm{EO} + n^7)\log nM)$ and $O((n^4\mathrm{EO} + n^5)\log nM)$.

6 A new strongly polynomial algorithm

This section presents a new strongly polynomial algorithm for minimizing a submodular function based on the following proximity lemma.

LEMMA 6.1. *Suppose $x \in \mathrm{B}(f)$ and $\eta = \max\{x(u) \mid u \in V\} > 0$. If $x(v) < -n\eta$ for some $v \in V$, then v is contained in all the minimizers of f.*

Proof. Starting with $y = x$ and $P = \{u \mid x(u) > 0\}$, repeat the following procedure until P becomes empty. Select an element $u \in P$, compute the exchange capacity

$$\widetilde{\mathrm{c}}(y, v, u) = \min\{f(X) - y(X) \mid v \in X \subseteq V \setminus \{u\}\}$$

to determine the step length $\sigma = \min\{y(u), \widetilde{c}(y, v, u)\}$, update $y := y + \sigma(\chi_v - \chi_u)$, and delete u from P. Since $\sigma \leq \eta$ in each iteration, the resulting y satisfies $y(v) < 0$.

At the end of each iteration, we obtain $y(u) = 0$ or a set X such that $v \in X \subseteq V \setminus \{u\}$ and $y(X) = f(X)$. This tight set X remains tight in the rest of the procedure. Therefore, at the end of the procedure, the set S obtained as the intersection of these tight sets satisfies $y(S) = f(S)$, $v \in S$, and $y(u) \leq 0$ for every $u \in S$. If there is no iteration that yields a tight set, then $y(u) \leq 0$ holds for every $u \in V$, and thus the entire set V serves as the set S.

For any subset $Y \subseteq V$ with $v \notin Y$, we have $f(S) = y(S) < y(S \cap Y) \leq f(S \cap Y)$, which implies by the submodularity of f that $f(S \cup Y) < f(Y)$. Thus v is contained in all the minimizers of f.

We now present the algorithm $\mathsf{SPM}(D, f)$ for minimizing a submodular function $f : \mathcal{D} \to \mathbf{R}$ on a ring family $\mathcal{D} \subseteq 2^V$ that consists of the set of ideals of a directed acyclic graph $D = (V, F)$. The algorithm keeps a base $x \in \mathrm{B}(\widehat{f})$ as a convex combination of extreme bases y_L for $L \in \Lambda$ and a subset $W \subseteq V$ that is guaranteed to include all the minimizers of f. The algorithm adds an arc (u, v) to F whenever it detects an implication that a minimizer of f including element u must include element v as well.

At the beginning of each iteration, the algorithm computes $\eta = \max\{x(u) \mid u \in W\}$. If $\eta \leq 0$, then W is the unique maximal minimizer of f. If $x(v) < -n\eta$, for some $v \in V$, then it follows from Lemma 6.1 that v is included in the unique maximal minimizer of f. We then apply the algorithm recursively to the contraction f_v defined by

$$f_v(Y) = f(Y \cup R(v)) - f(R(v))$$

for all ideals Y that include $R(v)$. If $f(R(u)) > n^2\eta$ for some $u \in W$, the algorithm finds an element v that is contained in all the minimizers of f_u. Then the algorithm adds a new arc (u, v) to F. The rest of each iteration is the same as the wave in the algorithm $\mathsf{SFMwave}$.

The algorithm $\mathsf{SPM}(D, f)$ is now described as follows.

Algorithm $\mathsf{SPM}(D, f)$

Step 0: Let L be an arbitrary consistent linear ordering. Compute an extreme base y_L by the greedy algorithm. Put $x := y_L$, $\lambda_L := 1$, $\Lambda := \{L\}$, $d_L(u) := 0$ for $u \in V$, and $W := V$.

Step 1: Compute $\eta := \max\{x(v) \mid v \in W\}$. Do the following (1-1) to (1-3).

- **(1-1)** If $\eta \leq 0$, then return W as the unique maximal minimizer of f.
- **(1-2)** If $x(v) < -n\eta$ for some $v \in W$, then delete all the vertices in $R(v)$ as well as incident arcs from D, apply $\mathsf{SPM}(D, f_v)$ to obtain the unique maximal minimizer Y of f_v, and return $Y \cup R(v)$.
- **(1-3)** If $f(R(u)) > n^2\eta$ for some $u \in W$, then construct a base $x' \in \mathrm{B}(\widehat{f_u})$ by $x' := \sum_{L \in \Lambda} \lambda_L y_{L'}$, where each L' is a linear ordering obtained from L by removing the elements in $R(u)$. For each element $v \in W \setminus R(u)$ with $x'(v) < -n\eta$, add an arc (u, v) to F. If this yields a directed cycle C in D, then contract C to a single vertex v_C, and apply $\mathsf{SPM}(D, f)$ to obtain the unique maximal minimizer Y of f. Then return Y after expanding v_C to C.

Step 2: Put $\mu := \delta$, where $\delta := \eta/4n$. Repeat the following (2-1) to (2-3) until $\mu > \eta$ or $d_{\min}(v)$ increases at some element $v \in W$.

- **(2-1)** If $x(v) = \mu$ for some $v \in W$, then apply $\mathsf{Update}(\mu)$.
- **(2-2)** Find $u := \arg\min\{d_{\min}(v) \mid v \in W, x(v) > \mu\}$, and put $\ell := d_{\min}(u)$. Let $L \in \Lambda$ be a linear ordering with $d_L(u) = d_{\min}(u)$.
- **(2-3)** Obtain a linear ordering L' by $\mathsf{New_Permutation}(L, \mu, \ell)$ and apply $\mathsf{Push}(L, L')$.

Step 3: If there is a gap at some level k, then put $T := \{v \mid v \in W, d_{\min}(v) \geq k\}$, $W := W \setminus T$, and $d_L(v) := n$ for all $v \in T$ and $L \in \Lambda$. Apply $\mathsf{Reduce}(\Lambda)$. Go to Step 1.

In Step (1-2), if $f(R(u)) > n^2\eta$, then $x'(V \setminus R(u)) = f(V) - f(R(u)) < -(n^2 - n)\eta$, and so there is an element $v \in V \setminus R(u)$ with $x'(v) < -n\eta$. Then it follows from Lemma 6.1 that v is contained in all the minimizers of f_u. As this operation increases the size of F, it can happen at most n^2 times over all iterations.

For the sake of analysis, we partition the iterations of this algorithm into phases. Each phase is a block of consecutive iterations that reduces the value of the potential function $\Phi(x)$ by half. After $3 \log n$ phases, the potential function $\Phi(x)$ is decreased by a factor of n^3. As a consequence, the value of η is decreased by at least a factor of n.

We say that an element v is big at an iteration if $x(v) > \delta = \eta/4n$. A big element v satisfies $x(v) \leq \widehat{f}(\{v\}) = f(R(v)) - f(R(v) \setminus \{v\})$, and hence either $f(R(v)) \geq \delta/2$ or $f(R(v) \setminus \{v\}) < -\delta/2$ holds. In

the former case, after $O(\log n)$ phases, we will get $f(R(v)) > n^2\eta$, which leads to an increase of $R(v)$. On the other hand, in the latter case, after $O(\log n)$ phases, there will be an element $u \in R(v) \setminus \{v\}$ such that $x(u) < -n\eta$, which leads to the contraction of $R(u)$.

We now analyze the number of waves in a phase. Let b be the number of elements that are big at some iteration during this phase. At the beginning of each wave, we have $\Phi(x) \leq (n-b)\delta^2 + 16n^2b\delta^2 \leq 17n^2b\delta^2$. Recall that each iteration improves the potential by at least δ^2. If x'' is the base at the end of the wave with $\mu > \eta$, we have

$$\Phi(x'') \leq \left(1 - \frac{1}{17n^2b}\right)^n \Phi(x).$$

Therefore, the number of waves in a phase is $O(nb)$. Let b_j be the number of elements that are big at some wave during the j-th phase. The total number of waves in the entire algorithm is $O(n \sum_j b_j)$.

For each element $v \in V$, once v becomes big, then after $O(\log n)$ phases $R(v)$ will get enlarged or D will get reduced. Therefore, each element can be a big element in $O(n \log n)$ phases, which implies $\sum_j b_j = O(n^2 \log n)$. Thus the algorithm performs $O(n^3 \log n)$ waves in total. Since each wave requires $O(n^2\mathrm{EO}+n^3)$ time, the overall running time of SPM is $O((n^5\mathrm{EO} + n^6)\log n)$, which is within a factor of $\log n$ of the best strongly polynomial bound.

The strongly polynomial scaling algorithms [10, 12] require $O(n^2 \log n)$ scaling phases instead of the $O(\log M)$ phases in their weakly polynomial versions. In contrast, the present result converts the $O(n^2 \log nM)$ bound on the number of waves to $O(n^3 \log n)$. Thus, we improve over the strongly polynomial scaling algorithms by a factor of n.

7 Fully combinatorial algorithms

A fully combinatorial algorithm consists of oracle calls for function evaluation and fundamental operations including additions, subtractions, and comparisons. Such an algorithm is strongly polynomial if the total number of oracle calls and fundamental operations is bounded by a polynomial in the dimension n of the problem. In the design of a fully combinatorial, strongly polynomial algorithm, we are allowed to multiply an integer which is bounded by a polynomial in n. We are also allowed to compute an integer rounding of a ratio of two numbers, provided that the answer is bounded by a polynomial in n.

In this section, we present fully combinatorial implementations of the algorithms SFM and SPM. The key idea is to choose the step length α so that all the coefficients λ_L should be integer multiples of $1/\kappa$ for some specified integer κ.

At the start of SFM, compute $\tau = \max\{f(\{v\}) + f(V \setminus \{v\}) - f(V) \mid v \in V\}$. In $\mathsf{Push}(L, L')$, we have $|y_L(v) - y_{L'}(v)| \leq \tau$ holds for any $v \in V$. Since $\eta \geq 1/n$ during the algorithm, we have $\delta \geq 1/4n^2$. Therefore, $\beta \geq 1/4n^2\tau$ holds unless $y_L = y_{L'}$.

We now set $\kappa = 4n^2\tau$, and redefine the step length α to be the largest integer multiple of $1/\kappa$ that is at most β and λ_L. Then the resulting coefficients are also integer multiples of $1/\kappa$. Thus we can implement SFM in a fully combinatorial manner.

The computation of α in a nonsaturating push requires $O(n \log nM)$ fundamental operations. In a saturating push, however, keeping the convex combination also requires $O(n \log nM)$ fundamental operations. Therefore, the overall time complexity of this fully combinatorial version of SFM is $O(n^6(\mathrm{EO} + \log nM)\log nM)$.

We now turn to the strongly polynomial algorithm obtained from SPM by replacing Step 2 by that of SFM. Since $f(R(u)) \leq n^2\eta$ and $f(R(u) \setminus \{u\}) \geq x(R(u) \setminus \{u\}) \geq -n^2\eta$, we have $y_L(u) \leq 2n^2\eta$ for any element $u \in V$ and any consistent linear ordering L. Since $f(V) \geq -n^2\eta$, this implies $y_L(u) \geq -2n^3\eta$. Thus $|y_L(u) - y_{L'}(u)| \leq 2(n^3 + n^2)\eta \leq 4n^3\eta$ holds, and we have $\beta \geq \delta/4n^3\eta = 1/16n^4$ unless $y_L = y_{L'}$.

We now set $\kappa = 16n^4$, and redefine the step length α to be the largest integer multiple of $1/\kappa$ that is at most β and λ_L. The computation of α in a nonsaturating push requires $O(n \log n)$ fundamental operations. Keeping the convex combination in a saturating push also requires $O(n \log n)$ fundamental operations. Therefore, the total running time of this fully combinatorial version of SPM is $O((n^7\mathrm{EO} + n^8)\log n)$.

Acknowledgements

The authors are grateful to Tom McCormick for helpful comments on the manuscript. The first author thanks the Asahi Glass Foundation for supporting this work. The second author gratefully acknowledges support through ONR grant N00014-08-1-0029.

References

[1] W. H. Cunningham: Testing membership in matroid polyhedra, *J. Combin. Theory*, B36 (1984), 161–188.

[2] W. H. Cunningham: On submodular function minimization, *Combinatorica*, 5 (1985), 185–192.

[3] J. Edmonds: Submodular functions, matroids, and certain polyhedra, *Combinatorial Structures and Their Applications*, R. Guy, H. Hanani, N. Sauer, and J. Schönheim, eds., Gordon and Breach, 69–87, 1970.

[4] L. Fleischer and S. Iwata: A push-relabel framework for submodular function minimization and applications

to parametric optimization, *Discrete Appl. Math.*, 131 (2003), 311–322.

[5] S. Fujishige: *Submodular Functions and Optimization*, Elsevier, 2005.

[6] M. X. Goemans and V. S. Ramakrishnan: Minimizing submodular functions over families of sets, *Combinatorica*, 15 (1995), 499–513.

[7] M. Grötschel, L. Lovász, and A. Schrijver: The ellipsoid method and its consequences in combinatorial optimization, *Combinatorica*, 1 (1981), 169–197.

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

[9] S. Iwata: A fully combinatorial algorithm for submodular function minimization, *J. Combin. Theory*, B84 (2002), 203–212.

[10] S. Iwata: A faster scaling algorithm for minimizing submodular functions, *SIAM J. Comput.*, 32 (2003), 833–840.

[11] S. Iwata: Submodular function minimization, *Math. Programming*, 112 (2008), 45–64.

[12] S. Iwata, L. Fleischer, and S. Fujishige: A combinatorial strongly polynomial algorithm for minimizing submodular functions, *J. ACM*, 48 (2001), 761–777.

[13] L. Lovász: Submodular functions and convexity. *Mathematical Programming — The State of the Art*, A. Bachem, M. Grötschel and B. Korte, eds., Springer-Verlag, 1983, 235–257.

[14] S. T. McCormick: Submodular function minimization, *Discrete Optimization* (K. Aardal, G. Nemhauser, R. Weismantel, eds., Handbooks in Operations Research, 12, Elsevier, 2005), 321–391.

[15] K. Nagano: A strongly polynomial algorithm for line search in submodular polyhedra, *Discrete Optim.*, 4 (2007), 349–359.

[16] J. B. Orlin: A faster strongly polynomial time algorithm for submodular function minimization, *Math. Programming*, to appear.

[17] A. Schrijver: A combinatorial algorithm minimizing submodular functions in strongly polynomial time, *J. Combin. Theory*, B80 (2000), 346–355.

[18] L. S. Shapley: Cores of convex games, *Int. J. Game Theory*, 1 (1971), 11–26.