18.438 Advanced Combinatorial Optimization	Updated April 29, 2012

Lecture 20

Lecturer: Michel X. Goemans *Scribe: Claudio Telha (Nov 17, 2009)*

Given a finite set V with n elements, a function $f : 2^V \to \mathbb{Z}$ is *submodular* if for all $X, Y \subseteq V$, $f(X \cup Y) + f(X \cap Y) \leq f(X) + f(Y)$. Submodular functions frequently arise in combinatorial optimization. For example, the cut function in a weighted undirected graph and the rank function of a matroid are both submodular.

Submodular function minimization is the problem of finding the global minimum of f. It is polynomial time solvable (assuming that f can be efficiently computed), as it was shown by Grötschel, Lovász and Schrijver [2] using the ellipsoid method. In this lecture, we describe[1] a recent efficient combinatorial algorithm for submodular function minimization by Iwata and Orlin [1].

1 Preliminaries

Given a submodular function f such that $f(\emptyset) = 0$, the base polyhedron $B(f)$ is defined as follows:

$$B(f) = \{x \in \mathbb{R}^V : x(S) \leq f(S), \forall S \subseteq V \quad \text{and} \quad x(V) = f(V)\}.$$

For any linear linear ordering L of the elements in V, we define $\underline{L}(v) = \{u : u \leq_L v\}$. It can be shown that $y_L(v) \equiv f(\underline{L}(v)) - f(\underline{L}(v) - \{v\})$ defines a vertex of $B(f)$. Moreover, any vertex of $B(f)$ can be represented in this way. We will make extensive use of this representation.

Let us define the positive part function $x^+(v) = \max\{x(v), 0\}$ and the negative part function $x^-(v) = \min\{x(v), 0\}$. The following min-max relation establishes the connection between the base polyhedron and submodular function minimization:

$$\min_{W \subseteq V} f(W) = \max_{x \in B(f)} \sum_{v \in V} x^-(v).$$

The Iwata and Orlin's algorithm solves the maximization problem $\max_{x \in B(f)} x^-(v)$, and simultaneously finds a minimizer W for f, that is maximal inclusion-wise. This is how we will prove the correctness of the minimax relation. Instead of using a combinatorial algorithm like the one we describe, we could also use the ellipsoid algorithm as we can separate efficiently over $B(f)$ and the function $\sum_{v \in V} x^-(v)$ is concave (lots of details are missing here, including how to start and when to stop).

We assume in this lecture that f is given by a value oracle. In other words, we are only allowed to query the value $f(S)$ for any $S \subseteq V$. We express the running time of the algorithm as a function of the number of oracle calls to f and the time to execute all the arithmetic operations.

Section 2 describes the algorithm, and Section 3 analyzes its running time.

2 Iwata and Orlin's algorithm

One question is how to certify that the x that the algorithm delivers is in $B(f)$. The key here is to express it as a convex combination of a polynomial number of extreme points of $B(f)$ (which we know correspond to linear orderings).

The algorithm maintains 4 objects:

[1] The description given here closely follows [1].

- A set Λ of linear orderings of V representing vertices of $B(f)$. Initially, Λ has a unique (arbitrary) ordering L.
- An element $x \in B(f)$ defined by a convex combination $x = \sum_{L\in\Lambda} \lambda_L y_L$ of vertices in Λ. Initially, $\lambda_L = 1$ for our unique ordering L in Λ.
- Distance labels $d_L : V \to \mathbb{Z}_+$ for each linear ordering $L \in \Lambda$. Initally, $d_L(v) = 0$ for all $v \in V$.
- A set $W \subseteq V$ containing every minimizer of f. At the end of the algorithm, W will be the maximal minimizer, inclusion-wise. Initially, $W = V$.

During the execution of the algorithm, the following properties of the distance labels are preserved:

1. If $x(u) \leq 0$ then $d_L(u) = 0$ for all $L \in \Lambda$.
2. If $u \leq_L v$ then $d_L(u) \leq d_L(v)$. (This can be restated as $d_L(v) < d_L(u)$ implies $v <_L u$.
3. For all $L, L' \in \Lambda$, $|d_L(u) - d_{L'}(u)| \leq 1$.

We need some additional definitions. We say that the distance labels $\{d_L\}_{L\in\Lambda}$ are *valid* if the three properties of the distance labels hold. We also define the *minimum distance labels*:

$$d_{\min}(u) \equiv \min_{L\in\Lambda} d_L(u).$$

Note that Property 3 of the distance labels ensures that

$$d_L(u) \in \{d_{\min}(u), d_{\min}(u) + 1\}.$$

We group the elements $v \in V$ according to their value $d_{\min}(v)$, and we say that the minimum distance labels *have a gap at* $k \geq 1$ if there exists an element $v \in V$ with $d_{\min}(v) = k$ and there are no elements $u \in V$ with $d_{\min}(u) = k - 1$. The importance of this gap notion is the following lemma, that allow us to delete all the elements in V after a gap.

Lemma 1 *If the distance labels are valid and they have a gap at $k \geq 1$, then any minimizer X of f satisfies $X \subseteq W := \{v \in V : d_{\min}(v) < k\}$*

Proof: Consider any $u \in W, v \notin W$. For each $L \in \Lambda$, we have that $d_L(u) \leq d_{\min}(u) + 1 \leq (k-2) + 1 < d_{\min}(v) \leq d_L(v)$. By property 2, we have that $u <_L v$ for all $L \in \Lambda$. In particular, $y_L(W) = \sum_{w\in W} y_w = f(W)$ for every $L \in \Lambda$ and therefore $x(W) = f(W)$. Note that any element $v \in V - W$ satisfies $d_{\min}(v) > 0$, so by property 1, $x(v) > 0$. This implies that for any set $X \not\subseteq W$, we have

$$f(X \cup W) \geq x(X \cup W) = x(X - W) + x(W) > x(W) = f(W)$$

which, by submodularity implies:

$$f(X \cap W) \leq f(X) + f(W) - f(X \cup W) < f(X),$$

so X cannot be a minimizer. □

Whenever there is a gap at some level $k \geq 1$, the algorithm updates W. Lemma 1 ensures that every minimizer of f is still contained in W. If $v \in V$ is removed from W, we set $d_L(v) = n$ for all $L \in \Lambda$.

We need to discuss when to stop the algorithm. We will use the value $\eta = \max_{v\in W} x(v)$ as a measure of the progress of the algorithm towards an optimal solution. We can stop when $\eta \leq 0$ as this would imply equality between $f(W)$ and $\sum_{v\in V} x^-(v)$. If we further assume that the submodular function takes integer values, we can stop when $\eta < 1/n$. Indeed, every possible minimizer satisfies $X \subseteq W$, and therefore, if $\eta < 1/n$, then $f(W) = x(W) = x(X) + x(W - X) < f(X) + 1$, so $f(W) \leq f(X)$, by integrality. Thus, W is a minimizer.

We have the main elements to describe the structure algorithm. Step 4 will be described in detail in the next subsection.

1. Choose an arbitrary order L_0 and initialize $\Lambda = \{L_0\}$, $x = y_{L_0}$, $d_{L_0} \equiv 0$ and $W = V$.

2. Compute $\eta = \max_{v \in W} x(v)$. If $\eta < 1/n$, return W as the global minimizer of f. Otherwise, define $\delta = \eta/4n$ and find μ such that $\delta \leq \mu \leq \eta - \delta$ and there is no $u \in W$ with $\mu - \delta < x(u) < \mu + \delta$.

3. Find $u = \arg\min\{d_{\min}(v) : v \in W, x(v) > \mu\}$ and set $l = d_{\min}(u)$. Let L_1 be a linear ordering in Λ with $d_{L_1}(u) = d_{\min}(u)$.

4. Using the push procedure described in Section 2.1, build a new linear ordering L_2 from L_1. Update Λ and x.

5. If there is a gap at any $k \geq 1$, update W and set the distance labels $d_L(v)$ to n, for every vertex removed and every $L \in \Lambda$. Go to Step 2.

2.1 The push procedure

We now the describe the push procedure (Step 4). This step updates x by adding one linear order L_2 to Λ and possibly removing one linear order L_1 from Λ. If the current x is given by $x = \sum_{L \in \Lambda} \lambda_L x_L$, the updated x' satisfies

$$x' - x = \epsilon y_{L_2} - \epsilon y_{L_1},$$

so this means that λ_{L_1} is decreased by ϵ and λ_{L_2} is set to ϵ. ϵ will be set as large as possible under some conditions.

First, let us see how to choose L_1. We find a value $\mu \in [\delta, \eta - \delta]$ such that the interval $(\mu - \delta, \mu + \delta)$ does not contain any value in $\{x(v)\}_{v \in W}$ (this μ is guaranteed to exists by the pigeonhole principle). Finally, we find u and L_1 such that

$$u = \arg \min_{u : x(u) > \mu} d_{\min}(u), \quad d_{L_1}(u) = d_{\min}(u).$$

Once L_1 is fixed, we define L_2. Consider the sets $S = \{v \in W : d_{L_1}(v) = d_{\min}(u)\}$ and $R = \{v \in S : x(v) > \mu\}$. It is clear that $u \in R \subseteq S$ and that the elements in S are consecutive in L_1. We obtain L_2 from L_1 by preserving the position of the elements in $V - S$, and then moving all the elements in R just after all the elements in $S \setminus R$, preserving the relative order in both sets. See Figure 2.1 for an illustration.

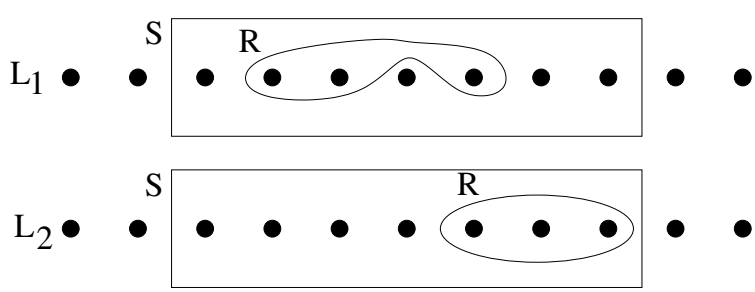

Figure 1: The elements in R are moved backwards inside of S. The order in R and $S \setminus R$ is preserved.

We also need to define new distance labels for L_2. This actually justifies our choice of R. Indeed, observe that we can increase the label of all elements of R by 1 unit and still stay feasible (since (i)

all elements of S have the same label and (ii) all elements v of R have $d_{L_1}(v) = d_{\min}(v)$ because of the choice of u). Thus, the following labels are valid for L_2:

$$d_{L_2}(v) = \begin{cases} d_{L_1}(v) + 1 & \text{if } v \in R, \\ d_{L_1}(v) & \text{if } v \notin R. \end{cases}$$

In order to conclude the procedure, we need to define ϵ. We choose ϵ as the larger ϵ satisfying two conditions. The first one is to keep λ_{L_1} nonnegative. This imposes the condition $\epsilon \leq \lambda_{L_1}$. The second condition requires that $x'(v) \geq \mu$ in R and $x'(v) \leq \mu$ in $S \setminus R$, both valid conditions for the original x. We choose ϵ as the larger value such that both conditions hold. If $\epsilon = \lambda_{L_1}$, we call the update process a *saturating* push. Otherwise, we call it a non-saturating push. Note that a saturating push replaces L_1 by L_2 in the convex combination defining x.

This completes the definition of the algorithm. We conclude this section with a simple observation about how the vector x changes with each push. The total change is

$$x' - x = \epsilon y_{L_2} - \epsilon y_{L_1}.$$

It is not hard to see that $y_{L_2}(v) = y_{L_1}(v)$ when $v \notin S$. By submodularity we also have that $y_{L_2}(v) \leq y_{L_1}(v)$ when $v \in R$, and $y_{L_2}(v) \geq y_{L_1}(v)$ when $v \in S \setminus R$. It follows that

$$\begin{aligned} x(v) &= x'(v) \text{ for } v \notin S \\ x(v) &\geq x'(v) \text{ for } v \in R \\ x(v) &\leq x'(v) \text{ for } v \in S \setminus R. \end{aligned}$$

Intuitively, we are thus making progress as we are decreasing some large $x(v)$ (and increasing some smaller ones) and we are increasing some distance labels. In the next section we analyze the running time of the algorithm.

3 Running time

We first prove that the number of saturating and non-saturating pushes is $\text{poly}(n, \log M)$, where M is the maximum absolute value of f. In both cases, we use a potential function argument.

Lemma 2 *The number of non-saturating pushes is $O(n^3 \log(nM))$.*

Proof: Consider the potential function $\Phi(x) = \sum_{v \in W} (x^+(v))^2$. Let us prove that if x' is an update of x using a non-saturating push, then

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

Let $Q^+ = \{v : v \in S \setminus R, x'(v) > 0\}$. Then,

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

Note that $x^+(v) \geq x'(v)$ for $v \in S \setminus (Q^+ \cup R)$, and thus we can write:

$$\Phi(x) - \Phi(x') \quad \geq \quad 2\mu \left(\sum_{v \in S} (x^+(v) - x'(v)) \right) + \delta \left(\sum_{v \in Q^+} (x'(v) - x^+(v)) + \sum_{v \in R} (x(v) - x'(v)) \right)$$

But the first term is ≥ 0 as $x^+(v) \geq x(v)$ and $\sum_{v \in S} x(v) = \sum_{v \in S} x'(v)$. Therefore:

$$\Phi(x) - \Phi(x') \geq \delta \sum_{v \in Q} (x'(v) - x_+(v)) + \delta \sum_{v \in R} (x(v) - x'(v)).$$

All these terms are ≥ 0 and, by construction, we know that $|x_+(v) - x'(v)| \geq \delta$ for some v, and therefore $\Phi(x) - \Phi(x') \geq \delta^2$ for a non-starurating push and nonnegative for a saturating push. Furthermore, $\Phi(x) \leq n\eta^2 = 16n^3\delta^2$, and thus $\Phi(x') \leq \Phi(x)\left(1 - \frac{1}{16n^3}\right)$, for every saturating push.

Note that if we start the algorithm with $x = x_0$, then $\Phi(x_0) \leq nM^2$, so after $O(n^3 \log(nM))$ non-saturating pushes, the current x satisfies $\Phi(x) < 1/n^2$. Since this condition implies the termination condition $\eta < 1/n$, the result follows. □

Lemma 3 *The number of saturating pushes is $O(n^5 \log(nM))$.*

Proof: We introduce another potential function $\phi(\Lambda) = \sum_{L \in \Lambda} \sum_{v \in V} (n - d_L(v))$. Note that in any push operation, the distance labels of L_2 are no smaller than the distance labels of L_1, and at least one of the labels increases. It follows that a saturating push increases the potential function by at least one. A non-saturating push adds a new linear order to Λ, and therefore, it increases Λ by at most n^2. From this relation, and Lemma 2, it follows that the number of saturating pushes is $O(n^5 \log(nM))$, otherwise the potential function would be negative. □

Both pushes can be carried out in polynomial time, using $O(n)$ oracle calls, so the algorithm runs in polynomial time in the oracle model.

References

[1] Iwata, S. and J. Orlin (2009). A simple combinatorial algorithm for submodular function minimization. *Accepted for the 2009 SODA Conference.*

[2] Grötschel, M. and Lovász, L. and Schrijver, A. The ellipsoid method and its consequences in combinatorial optimization. Combinatorica (1) 2:169-177, 1981.