23.1 Introduction

Today we're going to introduce a generalization of linear programming which has turned out to be shockingly useful for approximation algorithms: semidefinite programming. There are a few different intuitions about this, but at a high level, think about all the power that we've gotten out of using LP relaxations. One thing we cannot express, though, are products: quadratic programming is NP-hard, so we can't have products of variables. Semidefinite programming is going to let us have some kind of product, but not a regular product – it will actually be the vector inner product. This extra power is extremely useful for certain problems.

23.2 Semidefinite Programming: Definitions

Let's recall some basic linear algebra.

Definition 23.2.1 *A symmetric matrix $X \in \mathbb{R}^{n \times n}$ is* positive semidefinite *(PSD) if and only if $y^Y Xy \geq 0$ for all $y \in \mathbb{R}^n$.*

This is only one possible definition: it turns out that there are many equivalent definitions.

Theorem 23.2.2 *The following are all equivalent statements:*

1. *X is PSD*
2. *$y^T Xy \geq 0$ for all $y \in \mathbb{R}^n$*
3. *$X = V^T V$ for some $V \in \mathbb{R}^{n \times n}$*
4. *For all $i \in [n]$ there exists a vector $v_i \in \mathbb{R}^n$ such that $X_{ij} = v_i \cdot v_j$*

One quick piece of notation: we're going to use $X \succeq 0$ to denote that X is PSD.

Definition 23.2.3 *A* semidefinite program *(SDP) is an LP with the additional constraint that the matrix of variables is PSD.*

Note that usually we think of the variables of an LP as a vector, but here we're going to think of them as a matrix. So, for example, if we have variables x_{ij} for $i, j \in [n]$ (maybe corresponding to edges?), we could write something like

$$\begin{aligned}
\text{maximize:} \quad & \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij} & \\
\text{subject to:} \quad & \sum_{i=1}^{n}\sum_{j=1}^{n} a_{ijk}x_{ij} \leq b_k & \forall k \in [m] \\
& x_{ij} = x_{ji} & \forall i,j \in [n] \\
& X = (x_{ij}) \succeq 0 &
\end{aligned}$$

Theorem 23.2.4 *Under some technical "niceness" conditions, SDPs can be solved to additive error ϵ in time polynomial in the size of the input and* $\log(1/\epsilon)$.

We're not going to do into details of this algorithm (it's even more complicated than solving LPs), but the basic approach is straightforward. We're going to use the Ellipsoid algorithm (as we did with LPs), since the feasible solution set is convex (a polytope intersected with a cone). For a separation oracle, we can check each linear constraint efficiently (we're not talking about exponential size LPs here – everything is polynomial). For the PSD constraint, we compute the smallest eigenvector λ of X with associated eigenvector y. If $\lambda \geq 0$ then X is PSD by definition. Otherwise, $y^T Xy = y^T \cdot (\lambda y) = \lambda(y^T y) < 0$, so $y^T Xy \geq 0$ is a violated constraint (and is in fact a separating hyperplane).

For the rest of the course we're basically going to ignore the niceness conditions (since they're basically always satisfied) and ϵ (since we're doing approximation algorithms anyway). So think of Theorem 23.2.4 as "we can solve SDPs".

Of course, now you might ask the question: why on earth would I want to think of my variables as a matrix and require that it be PSD? The answer lies in one of the equivalent definitions of being PSD: vector programming! By Theorem 23.2.2, the above SDP is equivalent to finding vectors $v_1, v_2, \cdots, v_n \in \mathbb{R}^n$ to solve the following mathematical program:

$$\begin{aligned}
\text{maximize:} \quad & \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}(v_i \cdot v_j) & \\
\text{subject to:} \quad & \sum_{i=1}^{n}\sum_{j=1}^{n} a_{ijk}(v_i \cdot v_j) \leq b_k & \forall k \in [m] \\
& v_i \in \mathbb{R}^n & \forall i \in [n]
\end{aligned}$$

OK, but why would we want to do this kind of vector programming? Because this is a relaxation of (strict) quadratic programming! Consider the following strict quadratic program.

$$\begin{aligned} \text{maximize:} \quad & \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_i x_j & \\ \text{subject to:} \quad & \sum_{i=1}^{n}\sum_{j=1}^{n} a_{ijk}x_i x_j \leq b_k & \forall k \in [m] \\ & x_i \in \mathbb{R} & \forall i \in [n] \end{aligned}$$

Any solution to this is also a solution to the vector program, since we can always extend each x_i into an n-dimensional vector by padding it with $n-1$ 0's. So the vector program / SDP is a polynomial-time solvable relaxation of the (strict) quadratic program!

Another way of thinking of this is the following: LPs are a relaxation of ILPs where where we relax integral variables to fractional variables (allowing us to solve in polytime). SDPs are a relaxation of strict quadratic programs where we relax one-dimensional variables to n-dimensional vector variables, which allows us to solve it in polytime!

So in some sense the "right" way to think about SDPs is that the variables are n-dimensional vectors and we are allowed to write constraints that are linear in the dot products. Note that we are not allowed to write constraints about the vectors themselves, just about their dot products (since under the hood the "actual" variables are the dot products).

23.3 Max Cut

Let's see a relatively straightforward example, which is also the first and most famous application of SDPs to approximation algorithms: the Goemans-Williamson algorithm for Max-Cut [GW95]. Recall the Max-Cut problem:

- Input: graph $G = (V, E)$, weights $w : E \to \mathbb{R}^+$
- Feasible solution: $S \subseteq V$
- Objective: maximize $\sum_{e \in \delta(S)} w(e)$.

There's an obvious 1/2-approximation: just include each $v \in V$ in S with probability 1/2. Then the probability each edge is cut is 1/2, so it's a 1/2-approximation.

Let's see what we can get by going through an SDP. If we were going through an LP, the first step would be to write an ILP and then relax it to an LP. But now we don't necessarily want to write an ILP; instead, we want to write a strict quadratic program. If you think about it for a bit, the following quadratic program is relatively natural. (In order to use classical notation, we'll assume that $V = [n]$).

$$\begin{aligned} \text{maximize:} \quad & \frac{1}{2} \sum_{\{i,j\} \in E} w(i,j)(1 - x_i x_j) & \\ \text{subject to:} \quad & x_i^2 = 1 & \forall i \in [n] \\ & x_i \in \mathbb{R} & \forall i \in [n] \end{aligned}$$

Theorem 23.3.1 *This quadratic program is exactly the Max-Cut problem.*

Proof: The constraints imply that $x_i \in \{-1, 1\}$ for all $i \in [n]$. So given a solution x, let $S = \{i : x_i = 1\}$. Then

$$\frac{1}{2} \sum_{\{i,j\} \in E} w(i,j)(1 - x_i x_j) = \frac{1}{2} \left(\sum_{\{i,j\} \in \delta(S)} 2w(i,j) + \sum_{\{i,j\} \in E \setminus \delta(S)} 0w(i,j) \right) = \sum_{\{i,j\} \in \delta(S)} w(i,j).$$

Similarly, let $S \subseteq V$. Let $x_i = 1$ if $i \in S$, and let $x_i = -1$ if $i \notin S$. Then by the same calculation,

$$\sum_{\{i,j\} \in \delta(S)} w(i,j) = \frac{1}{2} \sum_{\{i,j\} \in E} w(i,j)(1 - x_i x_j).$$

Thus there is a solution to the QP of value α if and only if there is a cut of value α. ■

Of course, we can't actually solve the QP, and the x_i values are technically already allowed to be fractional. So intead of relaxing integrality constraints, we'll relax to vectors. This gives the following SDP:

$$\begin{aligned} \text{maximize:} \quad & \frac{1}{2} \sum_{\{i,j\} \in E} w(i,j)(1 - (v_i \cdot v_j)) & \\ \text{subject to:} \quad & v_i \cdot v_i = 1 & \forall i \in [n] \\ & v_i \in \mathbb{R}^n & \forall i \in [n] \end{aligned}$$

Since any solution to the actual problem gives a solution to the QP of the same value, and any solution to the QP gives a solution to the SDP of the same value, we know that this is a valid relaxation: the optimal SDP value is at least OPT. So our approach will to solve the SDP, and then "round" the vectors we get back to $\{-1, +1\}$. The algorithm we're going to use for this is called "random hyperplane rounding": we're going to choose a random hyperplane, which will divide the vectors from the SDP into two sets, and use that as the cut. More formally, we have the following algorithm.

- Solve the SDP to get vectors $v_i \in \mathbb{R}^n$ for each $i \in [n]$.
- Choose a vector $r \in \mathbb{R}^n$ uniformly at random from $\{v \in \mathbb{R}^n : \|v\| = 1\}$ (i.e., choose a random unit vector). Note: we can do this by choosing each coordinate independently from $N(0,1)$, and then rescaling to get a unit vector.

- Let $S = \{i \in [n] : v_i \cdot r \geq 0\}$.
- Return S.

Theorem 23.3.2 *Random hyperplane rounding is a* $\alpha_{GW} = \inf_{0 \leq \theta \leq \pi} \frac{2}{\pi} \cdot \frac{\theta}{1-\cos\theta} > 0.87856$*-approximation.*

Proof: We'll show that

$$\mathbf{Pr}[\{i,j\} \in \delta(S)] \geq \alpha_{GW} \cdot \frac{1}{2}(1 - v_i \cdot v_j),$$

for every $\{i,j\} \in E$, which then implies the theorem by linearity of expectations and the fact that the SDP optimum is at least OPT.

So fix some $\{i,j\} \in E$. Let P be the plane spanned by $\{v_i, v_j\}$, and let θ_{ij} be the angle between v_i, v_j. If we project r onto P and then rescale it to be a unit vector, then this is still uniformly distributed among the unit vectors in P. So from the perspective of $\{i,j\}$, we can think of the algorithm as randomly choosing a unit vector / line in the plane P, and $\{i,j\}$ is cut if and only if v_i and v_j are on different sides of this line. Thus

$$\mathbf{Pr}[\{i,j\} \in \delta(S)] = \frac{2\theta_{ij}}{2\pi} = \frac{\theta_{ij}}{\pi}.$$

By the definition of α_{GW}, we know that $\alpha_{GW} \leq \frac{2}{\pi} \cdot \frac{\theta_{ij}}{1-\cos\theta_{ij}}$, and thus $\frac{\theta_{ij}}{\pi} \geq \alpha_{GW} \cdot \frac{1-\cos\theta_{ij}}{2}$. Hence

$$\mathbf{Pr}[\{i,j\} \in \delta(S)] = \frac{\theta_{ij}}{\pi} \geq \alpha_{GW} \cdot \frac{1-\cos\theta_{ij}}{2}.$$

Now recall (from either linear algebra or high school trigonometry) that for any two vectors a, b, their dot product is $a \cdot b = \|a\| \cdot \|b\| \cdot \cos\theta_{ab}$, where θ_{ab} is the angle between the vectors. Since $\|v_i\| = \|v_j\| = 1$, this implies that $v_i \cdot v_j = \cos\theta_{ij}$. Thus

$$\mathbf{Pr}[\{i,j\} \in \delta(S)] \geq \alpha_{GW} \cdot \frac{1 - v_i \cdot v_j}{2},$$

which is what we needed to prove. Slightly more formally, we now have

$$\begin{aligned}
\mathbf{E}\left[\sum_{e \in \delta(S)} w(e)\right] &= \sum_{\{i,j\} \in E} w(i,j)\mathbf{Pr}[\{i,j\} \in \delta(S)] \\
&\geq \sum_{\{i,j\} \in E} w(i,j)\alpha_{GW} \cdot \frac{1 - v_i \cdot v_j}{2} \\
&= \alpha_{GW} \cdot \mathrm{OPT(SDP)} \\
&\geq \alpha_{GW} \cdot \mathrm{OPT}
\end{aligned}$$

as desired. ■

A natural question is whether this seemingly crazy value is tight. The best known hardness subject to P $\neq$ NP, due to Johan Håstad [Hås01], still leaves a gap:

Theorem 23.3.3 *Assuming $P \neq NP$, there is no α-approximation for Max-Cut with $\alpha > 16/17 \approx 0.941$.*

Somewhat shockingly, though, under a stronger assumption known as the *Unique Games Conjecture* (UGC), the Goemans-Williamson algorithm is actually tight! This is an extremely famous and important result due to Khot, Kindler, Mossel, and O'Donnell [KKMO07].

Theorem 23.3.4 *Assuming the UGC, there is no α-approximation for Max-Cut with $\alpha > \alpha_{GW}$.*

References

[GW95] Michel X. Goemans and David P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *J. ACM*, 42(6):1115–1145, November 1995.

[Hås01] Johan Håstad. Some optimal inapproximability results. *J. ACM*, 48(4):798–859, July 2001.

[KKMO07] S. Khot, G. Kindler, E. Mossel, and R. O'Donnell. Optimal Inapproximability Results for MAX-CUT and Other 2-Variable CSPs? *SIAM Journal on Computing*, 37(1):319–357, 2007.