CHAPTER 6

Randomized rounding of semidefinite programs

We now turn to a new tool which gives substantially improved performance guarantees for some problems. So far we have used linear programming relaxations to design and analyze various approximation algorithms. In this section, we show how nonlinear programming relaxations can give us better algorithms than we know how to obtain via linear programming; in particular we use a type of nonlinear program called a semidefinite program. Part of the power of semidefinite programming is that semidefinite programs can be solved in polynomial time.

We begin with a brief overview of semidefinite programming. Throughout the chapter we assume some basic knowledge of vectors and linear algebra; see the notes at the end of the chapter for suggested references on these topics. We then give an application of semidefinite programming to approximating the maximum cut problem. The algorithm for this problem introduces a technique of rounding the semidefinite program by choosing a random hyperplane. We then explore other problems for which choosing a random hyperplane, or multiple random hyperplanes, is useful, including approximating quadratic programs, approximating clustering problems, and coloring 3-colorable graphs.

6.1 A brief introduction to semidefinite programming

Semidefinite programming uses symmetric, positive semidefinite matrices, so we briefly review a few properties of these matrices. In what follows, X^T is the transpose of the matrix X, and vectors $v \in \Re^n$ are assumed to be column vectors, so that $v^T v$ is the inner product of v with itself, while vv^T is an n by n matrix.

Definition 6.1: *A matrix $X \in \Re^{n \times n}$ is positive semidefinite iff for all $y \in \Re^n$, $y^T X y \geq 0$.*

Sometimes we abbreviate "positive semidefinite" as "psd." Sometimes we will write $X \succeq 0$ to denote that a matrix X is positive semidefinite. Symmetric positive semidefinite matrices have some special properties which we list below. From here on, we will generally assume (unless otherwise stated) that any psd matrix X is also symmetric.

Fact 6.2: *If $X \in \Re^{n \times n}$ is a symmetric matrix, then the following statements are equivalent:*

1. *X is psd;*

2. *X has non-negative eigenvalues;*

3. *$X = V^T V$ for some $V \in \Re^{m \times n}$ where $m \leq n$;*

4. *$X = \sum_{i=1}^{n} \lambda_i w_i w_i^T$ for some $\lambda_i \geq 0$ and vectors $w_i \in \Re^n$ such that $w_i^T w_i = 1$ and $w_i^T w_j = 0$ for $i \neq j$.*

A *semidefinite program (SDP)* is similar to a linear program in that there is a linear objective function and linear constraints. In addition, however, a square symmetric matrix of variables can be constrained to be positive semidefinite. Below is an example in which the variables are x_{ij} for $1 \leq i, j \leq n$.

$$\begin{aligned} &\text{maximize or minimize} && \sum_{i,j} c_{ij} x_{ij} && && (6.1)\\ &\text{subject to} && \sum_{i,j} a_{ijk} x_{ij} = b_k, && \forall k, \\ & && x_{ij} = x_{ji}, && \forall i, j, \\ & && X = (x_{ij}) \succeq 0. \end{aligned}$$

Given some technical conditions, semidefinite programs can be solved to within an additive error of ϵ in time that is polynomial in the size of the input and $\log(1/\epsilon)$. We explain the technical conditions in more detail in the notes at the end of the chapter. We will usually ignore the additive error when discussing semidefinite programs and assume that the SDPs can be solved exactly, since the algorithms we will use do not assume exact solutions, and one can usually analyze the algorithm that has additive error in the same way with only a small loss in performance guarantee.

We will often use semidefinite programming in the form of *vector programming*. The variables of vector programs are vectors $v_i \in \Re^n$, where the dimension n of the space is the number of vectors in the vector program. The vector program has an objective function and constraints that are linear in the inner product of these vectors. We write the inner product of v_i and v_j as $v_i \cdot v_j$, or sometimes as $v_i^T v_j$. Below we give an example of a vector program.

$$\begin{aligned} &\text{maximize or minimize} && \sum_{i,j} c_{ij} (v_i \cdot v_j) && && (6.2)\\ &\text{subject to} && \sum_{i,j} a_{ijk} (v_i \cdot v_j) = b_k, && \forall k, \\ & && v_i \in \Re^n, && i = 1, \ldots, n. \end{aligned}$$

We claim that in fact the SDP (6.1) and the vector program (6.2) are equivalent. This follows from Fact 6.2; in particular, it follows since a symmetric X is psd if and only if $X = V^T V$ for some matrix V. Given a solution to the SDP (6.1), we can take the solution X, compute in polynomial time a matrix V for which $X = V^T V$ (to within small error, which we again will ignore), and set v_i to be the ith column of V. Then $x_{ij} = v_i^T v_j = v_i \cdot v_j$, and the v_i are a feasible solution of the same value to the vector program (6.2). Similarly, given a solution v_i to the vector program, we construct a matrix V whose ith column is v_i, and let $X = V^T V$. Then X is symmetric and psd, with $x_{ij} = v_i \cdot v_j$, so that X is a feasible solution of the same value for the SDP (6.1).

6.2 Finding large cuts

In this section, we show how to use semidefinite programming to find an improved approximation algorithm for the maximum cut problem, or MAX CUT problem, which we introduced in Section 5.1. Recall that for this problem, the input is an undirected graph $G = (V, E)$, and nonnegative weights $w_{ij} \geq 0$ for each edge $(i, j) \in E$. The goal is to partition the vertex set into two parts, U and $W = V - U$, so as to maximize the weight of the edges whose two endpoints are in different parts, one in U and one in W. In Section 5.1, we gave a $\frac{1}{2}$-approximation algorithm for the maximum cut problem.

We will now use semidefinite programming to give a .878-approximation algorithm for the problem in general graphs. We start by considering the following formulation of the maximum cut problem:

$$\begin{aligned} \text{maximize } & \frac{1}{2} \sum_{(i,j)\in E} w_{ij}(1 - y_i y_j) \\ \text{subject to } & y_i \in \{-1, +1\}, \qquad i = 1, \ldots, n. \end{aligned} \tag{6.3}$$

We claim that if we can solve this formulation, then we can solve the MAX CUT problem.

Lemma 6.3: *The program (6.3) models the maximum cut problem.*

Proof. Consider the cut $U = \{i : y_i = -1\}$ and $W = \{i : y_i = +1\}$. Note that if an edge (i, j) is in this cut, then $y_i y_j = -1$, while if the edge is not in the cut, $y_i y_j = 1$. Thus

$$\frac{1}{2} \sum_{(i,j)\in E} w_{ij}(1 - y_i y_j)$$

gives the weight of all the edges in the cut. Hence finding the setting of the y_i to ± 1 that maximizes this sum gives the maximum-weight cut. □

We can now consider the following vector programming relaxation of the program (6.3):

$$\begin{aligned} \text{maximize } & \frac{1}{2} \sum_{(i,j)\in E} w_{ij}(1 - v_i \cdot v_j) \\ \text{subject to } & v_i \cdot v_i = 1, \qquad i = 1, \ldots, n, \\ & v_i \in \Re^n, \qquad i = 1, \ldots, n. \end{aligned} \tag{6.4}$$

This program is a relaxation of (6.3) since we can take any feasible solution y and produce a feasible solution to this program of the same value by setting $v_i = (y_i, 0, 0, \ldots, 0)$: clearly $v_i \cdot v_i = 1$ and $v_i \cdot v_j = y_i y_j$ for this solution. Thus if Z_{VP} is the value of an optimal solution to the vector program, it must be the case that $Z_{VP} \geq \text{OPT}$.

We can solve (6.4) in polynomial time. We would now like to round the solution to obtain a near-optimal cut. To do this, we introduce a form of randomized rounding suitable for vector programming. In particular, we pick a random vector $r = (r_1, \ldots, r_n)$ by drawing each component from $\mathcal{N}(0, 1)$, the normal distribution with mean 0 and variance 1. The normal distribution can be simulated by an algorithm that draws repeatedly from the uniform distribution on [0,1]. Then given a solution to (6.4), we iterate through all the vertices and put $i \in U$ if $v_i \cdot r \geq 0$ and $i \in W$ otherwise.

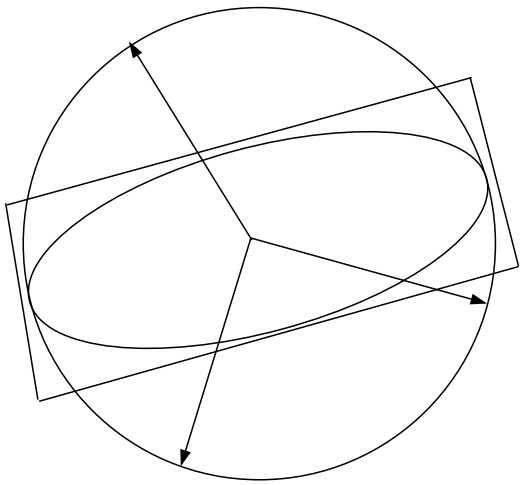

Figure 6.1: An illustration of a random hyperplane.

Another way of looking at this algorithm is that we consider the hyperplane with normal r containing the origin. All vectors v_i lie on the unit sphere, since $v_i \cdot v_i = 1$ and they are unit vectors. The hyperplane with normal r containing the origin splits the sphere in half; all vertices in one half (the half such that $v_i \cdot r \geq 0$) are put into U, and all vertices in the other half are put into W (see Figure 6.1). As we will see below, the vector $r/\|r\|$ is uniform over the unit sphere, so this is equivalent to randomly splitting the unit sphere in half. For this reason, this technique is sometimes called *choosing a random hyperplane*.

To prove that this is a good approximation algorithm, we need the following facts.

Fact 6.4: *The normalization of r, $r/||r||$, is uniformly distributed over the n-dimensional unit sphere.*

Fact 6.5: *The projections of r onto two unit vectors e_1 and e_2 are independent and are normally distributed with mean 0 and variance 1 iff e_1 and e_2 are orthogonal.*

Corollary 6.6: *Let r' be the projection of r onto a two-dimensional plane. Then the normalization of r', $r'/||r'||$, is uniformly distributed on a unit circle in the plane.*

We now begin the proof that choosing a random hyperplane gives a .878-approximation algorithm for the problem. We will need the following two lemmas.

Lemma 6.7: *The probability that edge (i, j) is in the cut is $\frac{1}{\pi} \arccos(v_i \cdot v_j)$.*

Proof. Let r' be the projection of r onto the plane defined by v_i and v_j. If $r = r' + r''$, then r'' is orthogonal to both v_i and v_j, and $v_i \cdot r = v_i \cdot (r' + r'') = v_i \cdot r'$. Similarly $v_j \cdot r = v_j \cdot r'$.

Consider Figure 6.2, where line AC is perpendicular to the vector v_i and line BD is perpendicular to the vector v_j. By Corollary 6.6, the vector r' with its tail at the origin O is oriented with respect to the vector v_i by an angle α chosen uniformly from $[0, 2\pi)$. If r' is to the right of the line AC, v_i will have a nonnegative inner product with r', otherwise not. If r' is above the line BD, v_j will have nonnegative inner product with r', otherwise not. Thus we have $i \in W$ and $j \in U$ if r' is in the sector AB and $i \in U$ and $j \in W$ if r' is in the sector CD. If the angle formed by v_i and v_j is θ radians, then the angles $\angle AOB$ and $\angle COD$ are also θ radians. Hence the fraction of values for which α, the angle of r', corresponds to the event

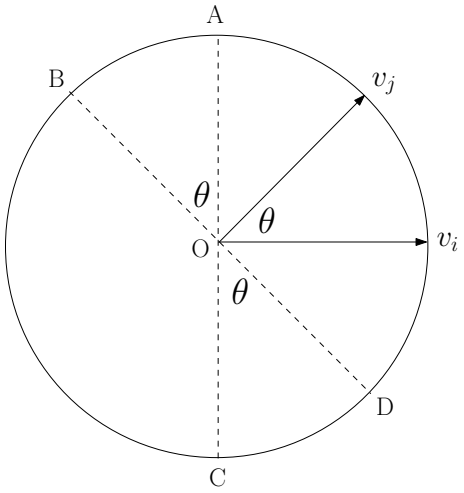

Figure 6.2: Figure for proof of Lemma 6.7.

in which (i, j) is in the cut is $2\theta/2\pi$. Thus the probability that (i, j) is in the cut is $\frac{\theta}{\pi}$. We know that $v_i \cdot v_j = \|v_i\|\|v_j\| \cos\theta$. Since v_i and v_j are both unit length vectors, we have that $\theta = \arccos(v_i \cdot v_j)$, which completes the proof of the lemma.

□

Lemma 6.8: *For $x \in [-1, 1]$,*

$$\frac{1}{\pi}\arccos(x) \geq 0.878 \cdot \frac{1}{2}(1 - x).$$

Proof. The proof follows from simple calculus. See Figure 6.3 for an illustration. □

Theorem 6.9: *Rounding the vector program (6.4) by choosing a random hyperplane is a .878-approximation algorithm for the maximum cut problem.*

Proof. Let X_{ij} be a random variable for edge (i, j) such that $X_{ij} = 1$ if (i, j) is in the cut given by the algorithm, and 0 otherwise. Let W be a random variable which gives the weight of the cut; that is, $W = \sum_{(i,j)\in E} w_{ij} X_{ij}$. Then by Lemma 6.7,

$$E[W] = \sum_{(i,j)\in E} w_{ij} \cdot \Pr[\text{edge } (i, j) \text{ is in cut}] = \sum_{(i,j)\in E} w_{ij} \cdot \frac{1}{\pi}\arccos(v_i \cdot v_j).$$

By Lemma 6.8, we can bound each term $\frac{1}{\pi}\arccos(v_i \cdot v_j)$ below by $0.878 \cdot \frac{1}{2}(1 - v_i \cdot v_j)$, so that

$$E[W] \geq 0.878 \cdot \frac{1}{2} \sum_{(i,j)\in E} w_{ij}(1 - v_i \cdot v_j) = 0.878 \cdot Z_{VP} \geq 0.878 \cdot \text{OPT}\,.$$

□

We know that $Z_{VP} \geq \text{OPT}$. The proof of the theorem above shows that there is a cut of value at least $0.878 \cdot Z_{VP}$, so that $\text{OPT} \geq 0.878 \cdot Z_{VP}$. Thus we have that $\text{OPT} \leq Z_{VP} \leq \frac{1}{0.878}\,\text{OPT}$. It has been shown that there are graphs for which the upper inequality is met with equality. This implies that we can get no better performance guarantee for the maximum cut problem

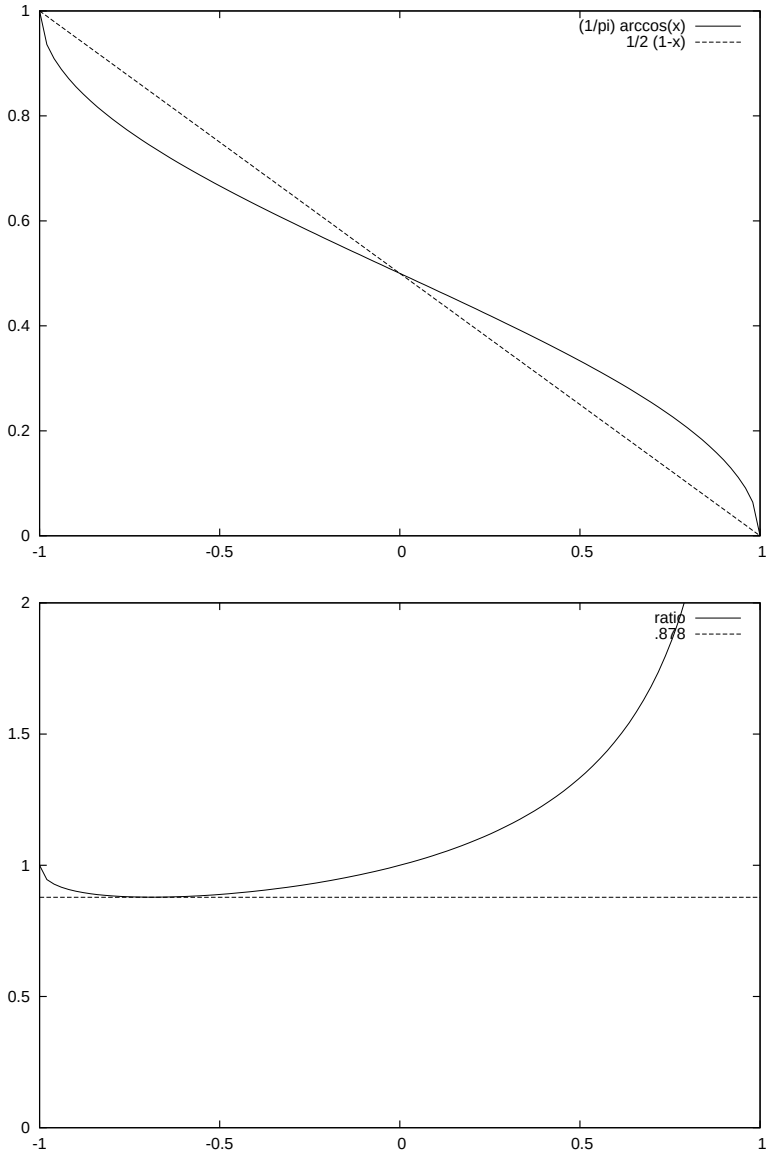

Figure 6.3: Illustration of Lemma 6.8. The upper figure shows plots of the functions $\frac{1}{\pi}\arccos(x)$ and $\frac{1}{2}(1-x)$. The lower figure shows a plot of the ratio of the two functions.

by using Z_{VP} as an upper bound on OPT. Currently, .878 is the best performance guarantee known for the maximum cut problem problem. The following theorems show that this is either close to, or exactly, the best performance guarantee that is likely attainable.

Theorem 6.10: *If there is an α-approximation algorithm for the maximum cut problem with $\alpha > \frac{16}{17} \approx 0.941$, then* P = NP*.*

Theorem 6.11: *Given the unique games conjecture there is no α-approximation algorithm for the maximum cut problem with constant*

$$\alpha > \min_{-1 \le x \le 1} \frac{\frac{1}{\pi}\arccos(x)}{\frac{1}{2}(1-x)} \ge .878$$

unless P = NP*.*

We sketch the proof of the second theorem in Section 16.5.

So far we have only discussed a randomized algorithm for the maximum cut problem. It is possible to derandomize the algorithm by using a sophisticated application of the method of conditional expectations that iteratively determines the various coordinates of the random vector. The derandomization incurs a loss in the performance guarantee that can be made as small as desired (by increasing the running time).

6.3 Approximating quadratic programs

We can extend the algorithm above for the maximum cut problem to the following more general problem. Suppose we wish to approximate the quadratic program below:

$$\begin{aligned} \text{maximize} \quad & \sum_{1 \le i,j \le n} a_{ij} x_i x_j \\ \text{subject to} \quad & x_i \in \{-1, +1\}, \qquad i = 1, \ldots, n. \end{aligned} \tag{6.5}$$

We need to be slightly careful in this case since as stated it is possible that the value of an optimal solution is negative (for instance, if the values of a_{ii} are negative and all other a_{ij} are zero). Thus far we have only been considering problems in which all feasible solutions have nonnegative value so that the definition of an α-approximation algorithm makes sense. To see that the definition might not make sense in the case of negative solution values, suppose we have an α-approximation algorithm for a maximization problem with $\alpha < 1$ and suppose we have a problem instance in which OPT is negative. Then the approximation algorithm guarantees the value of our solution is at least $\alpha \cdot \text{OPT}$, which means that the value of our solution will be greater than that of OPT. In order to get around this difficulty, in this case we will restrict the objective function matrix $A = (a_{ij})$ in (6.5) to itself be positive semidefinite. Observe then that for any feasible solution x, the value of the objective function will be $x^T A x$ and will be nonnegative by the definition of positive semidefinite matrices.

As in the case of the maximum cut problem, we can then have the following vector programming relaxation:

$$\begin{aligned} \text{maximize} \quad & \sum_{1 \le i,j \le n} a_{ij} (v_i \cdot v_j) \\ \text{subject to} \quad & v_i \cdot v_i = 1, \qquad i = 1, \ldots, n, \\ & v_i \in \Re^n, \qquad i = 1, \ldots, n. \end{aligned} \tag{6.6}$$