**Beyond Worst Case Analysis** **Fall 2018**

# Lecture 4: Semidefinite Programming and MAXCUT

Scribe: Vishnu Iyer October 1, 2018

## 4.1 Introduction

In this lecture we will explore a problem known as MAXCUT. The problem is defined as follows: given an undirected graph with vertex set $V$, we wish to find a set $S \subseteq V$ such that the number of edges $(i, j)$ for $i \in V$ and $j \in V \setminus S$ is maximized. As one might imagine, this combinatorial optimization problem has a variety of applications. Unfortunately, finding the maximum cut is also NP-hard. In this lecture, we will investigate a technique to estimate the maximum cut.

To approximate an optimal solution to the MAXCUT problem, we will employ an optimaztion model known as Semidefinite Programming (SDP). These programs can be solved in polynomial time to an arbitrarily high degree of accuracy. This quality, along with their expressive power, makes them ideal candidates for approximating solutions to otherwise intractable problems.

## 4.2 Semidefinite Programming

### 4.2.1 Review of Linear Programming

The reader may be familiar with optimization models such as linear programming. Recall that in Linear Programming (LP), we have a set of real-valued variables, an objective function $f(x)$ that is linear in the variables, and a set of constraints, which are also linear in the variables. Our goal is usually either to determine whether the constraints are feasible, or to maximize the objective function subject to the constraints of the constraints.

A general linear program looks like this:

$$\begin{aligned} \text{maximize} \quad & \sum_i c_i x_i \\ \text{subject to} \quad & \sum_i a_i^{(1)} x_i \leq b_1 \\ & \cdots \\ & \sum_i a_i^{(m)} x_i \leq b_m \end{aligned}$$

While linear programs are a powerful method for solving certain problems, algorithms based on linear programming generally do not provide very good performance guarantees for harder problems like MAXCUT.

The general idea behind Linear Programming extends to models such as Quadratic Programming (QP), where the constraints and objectives can be quadratic. Another useful variant is Integer Linear Programming (ILP), where our coefficients and variables are maximized over integers rather than reals. However, the problem of finding a solution to these models is NP-complete. In order to find good approximation algorithm, we must seek a more expressive optimization model that is also efficient to solve.

### 4.2.2 Review of Linear Algebra

Semidefinite Programming makes use of **positive semidefinite** matrices, which are defined below.

**Definition 4.1** (Positive Semidefiniteness)**.** *A symmetric $n \times n$ matrix $A$ is said to be positive semidefinite (PSD), written $A \succeq 0$, if for all $x \in \mathbb{R}^n$, one has that $x^T A x \geq 0$.*

An equivalent definition of positive semidefiniteness is that the matrix $A$ does not have any negative eigenvalues.

**Definition 4.2** (Positive Semidefiniteness)**.** *A symmetric $n \times n$ matrix $A$ is positive semidefinite if it has nonnegative eigenvalues.*

*Proof.* We employ the variational characterization of eigenvalues presented in the first note. The minimum eigenvalue $\lambda_1$ of $A$ is given by

$$\lambda_1 = \min_{x \neq 0} \frac{x^T A x}{x^T x} = \min_{\|x\|=1} x^T A x \geq 0 \qquad \square$$

There is one important property of PSD matrices that will be of use to us later.

**Lemma 4.3.** *The set of positive semidefinite matrices forms a convex cone. That is, for PSD matrices $A,\ B$ and constants $c_1, c_2 \geq 0$, $c_1 A + c_2 B$ is positive semidefinite.*

*Proof.* $x^T(c_1 A + c_2 B)x = c_1 x^T A x + c_2 x^T B x \geq 0$ □

### 4.2.3 SDP Formulation

We now turn to the details of how to formulate semidefinite programs. Like linear programs, we will have a linear objective function and a set of linear constraints. However, we write out our SDP variables as $X_{ij}$ for $1 \leq i, j \leq n$ and requite that the matrix $X = (X_{ij})$ is PSD. Concretely, a general SDP looks like

$$\begin{aligned} \text{maximize} \quad & \sum_{i,j} C_{ij} X_{ij} \\ \text{subject to} \quad & \sum_{i,j} A^{(1)}_{ij} X_{ij} \leq b_1 \\ & \cdots \\ & \sum_{i,j} A^{(m)}_{ij} X_{ij} \leq b_m \\ & X \succeq 0 \end{aligned}$$

At first, one might wonder whether semidefinite programming is any more effective than linear programming. In fact, if anything, it looks more restrictive. However, the following fact sheds light on why semidefinite programming is such a powerful technique.

**Theorem 4.4.** *Semidefinite Programming is equivalent to Vector Programming (VP). That is, every semidefinite programming instance can be formulated as follows:*

$$\begin{aligned} \textit{maximize} \quad & \max \sum_{i,j} C_{ij} \langle x^{(i)}, x^{(j)} \rangle \\ \textit{subject to} \quad & \sum_{i,j} A^{(1)}_{ij} \langle x^{(i)}, x^{(j)} \rangle \leq b_1 \\ & \cdots \\ & \sum_{i,j} A^{(m)}_{ij} \langle x^{(i)}, x^{(j)} \rangle \leq b_m \\ & x^{(i)} \in \mathbb{R}^n \qquad \forall i \end{aligned}$$

In order to prove this, we will first show the following intermediary fact:

**Lemma 4.5.** *An $n \times n$ matrix $X$ is PSD if and only if there exist vectors $x^{(1)}, ...x^{(n)}$ such that $X_{ij} = \langle x^{(i)}, x^{(j)} \rangle$.*

*Proof.* Suppose that there exist vectors $x^{(1)}, ...x^{(n)}$ such that $X_{ij} = \langle x^{(i)}, x^{(j)} \rangle$. Clearly $X$ is symmetric. Now, for any vector $y \in \mathbb{R}^n$, we have

$$y^T X y = \sum_i \sum_j y_i x^{(i)T} x^{(j)} y_j = \left( \sum_i y_i x^{(i)} \right)^T \left( \sum_j y_j x^{(j)} \right) = || \sum_i y_i x^{(i)} ||_2^2 \geq 0$$

and thus $X$ is PSD. Now, conversely, suppose that $X$ is PSD. By the SVD theorem, we can write

$$X = V \Sigma V^T$$

where $\Sigma$ is a matrix with nonnegative diagonal entries and $V$ is a matrix with orthonormal columns. Taking $\tilde{X} = V \Sigma^{\frac{1}{2}} V^T$, where $\Sigma^{\frac{1}{2}}$ is the matrix whose diagonal entries are the square root of those of $\Sigma$, we can write $X = \tilde{X}^T \tilde{X}$. Thus, if we let the columns of $\tilde{X}$ be $x^{(i)}$, then $X_{ij} = \langle x^{(i)}, x^{(j)} \rangle$ □

We are now ready to prove Theorem 4.4.

*Proof.* For a given SDP, we will prove that we can recast it as a vector program, replacing every occurrence of $X_{ij}$ with $\langle x^{(i)}, x^{(j)} \rangle$. Let $s^*$ and $v^*$ be the maximum value that the objective takes in the SDP and VP, respectively. We show that $s^* = v^*$ Suppose $X^*$ is an optimal solution to the *SDP*. By Lemma 4.4 we can construct vectors $x^{(1)}, ..., x^{(n)}$ such that $X_{ij} = \langle x^{(i)}, x^{(j)} \rangle$. Thus we can construct a feasible solution to the VP, and so $v^* \geq s^*$. Conversely, consulting Lemma 4.4 again, we can construct a feasible solution (that is, a PSD matrix) to the SDP from an optimal solution to the VP, so $s^* \geq v^*$. □

Now, it is easier to see where the power of SDP comes in. As a quick aside, SDPs are a rather common technique used in the approximation of QPs. We will see a major example in the next section.

### 4.2.4 Polynomial-time SDP Solvability

We can use a method known as the Ellipsoid Algorithm to solve an optimization model consisting of a linear objective over a convex feasible region. Since the set of PSD matrices form a convex cone, we know that SDPs optimize over convex regions (for more details on

this, consult any convex optimization textbook).

The ellipsoid algorithm works as follows:

1. Obtain a potential solution to the optimization problem.
2. Check whether the solution is feasible.
3. If the solution is not feasible, then construct an inequality that is not satisfied by the candidate solution but is satisfied by every feasible point.

Step 3 "separates" the candidate solution from the feasible region and is thus in general known as a "separation oracle." An SDP separation oracle will be as follows: for a candidate matrix $M$, test for positive semidefiniteness. If this test fails, we must find an inequality

$$\sum_{i,j} a_{ij} X_{ij} \geq 0$$

That is satisfied by all PSD matrices $X$ but not by $M$. Note that since $M$ is not PSD, it has at least one negative eigenvalue. Expressing this in terms of the Rayleigh quotient from lecture one, we have that

$$\min_{\|y\|_2=1} y^T M y = \sum_{i,j} M_{ij} y_i y_j < 0$$

Let $y^*$ be the optimal solution to this minimization problem. We know that for any PSD matrix $X$, we have that $y^{*T} X y \geq 0$. Thus if we take our inequality to be

$$\sum_{i,j} y_i^* y_j^* X_{ij} \geq 0$$

we can readily see that $M$ fails this inequality by any PSD matrix $X$ does not. Therefore, we can use the Ellipsoid algorithm to solve our SDPs to an arbitrarily high degree of accuracy. We omit the details of Ellipsoid algorithm implementation.

## 4.3 The MAXCUT Relaxation

Note that we can solve MAXCUT using the following Quadratic Program

$$\max \sum_{(i,j)\in E} \frac{1}{4}(x_i - x_j)^2$$

$$\forall i,\ x_i^2 = 1$$

The constraints force the values of $x_i$ to be $\pm 1$, with the sign of $x_i$ corresponding to which side of the cut vertex $i$ is in. The sum then counts the number of crossing edges. Again, recall that solving this is NP-hard. As such, we perform a *relaxation*. That is, we optimize over vectors $x^{(i)}$ rather than scalars $x_i$. By a natural extension of the QP above, we can create the following SDP

$$\max \sum_{(i,j)\in E} \frac{1}{4}\|x^{(i)} - x^{(j)}\|_2^2$$

$$\forall i,\ \|x^{(i)}\|_2^2 = 1$$

Intuitively speaking, the QP used the "directionality" of the scalars $\pm 1$ to separate the cuts and increase the value of the objective. The SDP is essentially performing the same task, but the dimension across which we are separating is larger. From this, one might imagine that the $x^{(i)}$ that are pointing in the same direction would be classified under the same cut. This is, in fact, how we will construct our approximation algorithm.

Before we proceed, note that the optimum of the SDP is at least as large as the solution to the Maximum Cut. This is because we can take all the elements of the actual maximum cut and assign their corresponding vectors to the values $\pm v$ (for any unit vector $v$). This is a feasible solution which gives the actual value of the Maximum Cut. Taking the maximum over the feasible set of solutions must return a value with a cut at least as large as that.

**Algorithm 1** MAXCUT Approximation

1. Run the SDP above and obtain vectors $x^{(1)}, ..., x^{(n)}$
2. Generate a vector $u \in \mathbb{R}^n$ from a multivariate Gaussian distribution with mean 0 and covariance $I_n$.
3. For each $x^{(i)}$, if $\langle u, x^{(i)} \rangle \geq 0$, place $i$ in the set $S$. Otherwise, place it in $V \setminus S$

An important point to note is that the vector $u$ is generated from a Gaussian distribution with covariance matrix $I$, so, by a simple symmetry argument, the direction of the sampled vector should be uniformly distributed. We are essentially taking a random hyperplane and labeling the upper halfspace as $S$.

In constructing our algorithm, we used a technique known as randomized rounding. We obtained a solution to the SDP in the form of vectors and to convert it back to the realm of the discrete Max Cut problem, we "rounded" the vectors to all "point" either towards the set $S$ or the set $V \setminus S$. As such, we can be certain that even though our SDP objective is greater than or equal to the true Max Cut, we will always return a legal cut in the graph.

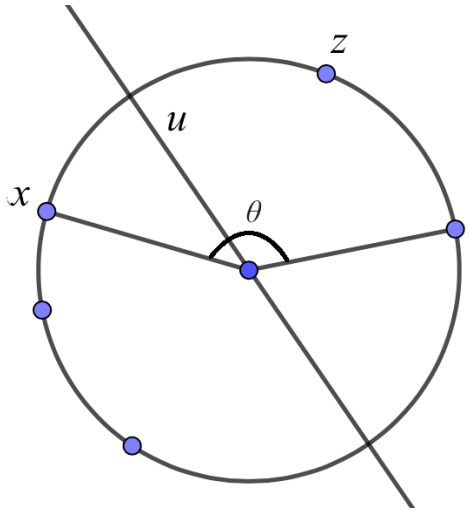


Figure 4.1: A random hyperplane separating vectors that are solutions to the SDP

Furthermore, the randomization helps prevent poor performance on worst-case inputs.

Using Figure 4.1 (previous page) as reference, suppose there was an angle of $\theta$ between two given vectors $x^{(i)}$ and $x^{(j)}$. Then the probability that given hyperplane cuts through them is $\dfrac{\theta}{\pi}$ by a uniformity argument. We are now ready to make a claim about the performance of this algorithm.

**Claim 4.6.** *Algorithm 1 is a* 0.878*-approximation algorithm for the MAXCUT problem.*

*Proof.* Let $N$ be the random variable corresponding to the number of edges cut, with $N_e$ indicating that edge $e$ was cut. We have that

$$E[N] = \sum_{e \in E} E[N_e] = \sum_{e \in E} \Pr[e \text{ is cut}] = \sum_{e \in E} \frac{\theta_e}{\pi}$$

We can use basic calculus to show that

$$\min_{\theta \in [0,\pi]} \frac{2\theta}{\pi(1 - \cos\theta)} \geq 0.878$$

$$\implies \frac{\theta}{\pi} \geq 0.878 \left( \frac{1}{2} - \frac{1}{2} \cos\theta \right)$$

Now, we apply this inequality termwise to bound the expected number of edges cut.

$$\sum_{e \in E} \theta_e \geq 0.878 \sum_{e \in E} \frac{1}{2} - \frac{1}{2} \cos\theta_e = 0.878 \cdot \frac{1}{4} \sum_{(i,j) \in E} 2 - 2\langle x^{(i)}, x^{(j)} \rangle$$

$$= 0.878 \cdot \frac{1}{4} \sum_{(i,j)\in E} \|x^{(i)} - x^{(j)}\|_2^2 = 0.878 \cdot Z_{\text{SDP}} \geq 0.878 \cdot \text{OPT}$$

where $Z_{\text{SDP}}$ denotes the optimum value of the SDP objective and OPT denotes the size of the actual maximum cut. Therefore, we see that the algorithm returns a cut whose expected size is at least 0.878 times the actual maximum cut and our claim is proven. □

## 4.4 Max-Cut in Bounded-Degree Triangle-Free Graphs

We now study the Max-Cut problem in a certain group of graphs which are triangle-free and of bounded-degree. Triangle-free means the graph has no cycles of length 3, and bounded-degree means each vertex has degree of at most $d$. In particular, we will show that given these characteristics, the graph is guaranteed to have a cut of size at least

$$\left(\frac{1}{2} + \Omega\left(\frac{1}{\sqrt{d}}\right)\right)|E|$$

We shall prove this by demonstrating a cut of this size that can be induced from a feasible solution to the SDP discussed previously. Concretely, let us define

$$x_j^{(i)} = \begin{cases} \frac{1}{\sqrt{2}} & i = j \\ -\frac{1}{\sqrt{2 \cdot \deg(i)}} & (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

Assuming, without loss of generality, that there are no isolated vertices, one can easily verify that each $x_j^{(i)}$ satisfies the unit norm criterion and thus the solution above is feasible.

Now we will see how the triangle-free condition comes into play. For any edge $(i,j)$, we know that there is no edge $k$ such that *both* $i$ and $j$ are adjacent to it. Thus for any $k \neq i,j$, we have that $x_k^{(i)} x_k^{(j)} = 0$. Since $i$ and $j$ are adjacent, we know that $x_j^{(i)} = -\frac{1}{2\sqrt{\deg(i)}}$ and $x_i^{(j)} = -\frac{1}{2\sqrt{\deg(j)}}$, so we have that

$$\langle x^{(i)}, x^{(j)} \rangle = -\frac{1}{2\sqrt{\deg(i)}} - \frac{1}{2\sqrt{\deg(j)}} \leq -\frac{2}{2\sqrt{\max(\deg(i), \deg(j))}} \leq -\frac{1}{\sqrt{d}}$$

In order to calculate the probability of being cut, we now find the angle between the vectors.

$$\theta = \arccos\left(\pi - \frac{1}{\sqrt{d}}\right) = \frac{\pi}{2} - \arcsin\left(-\frac{1}{\sqrt{d}}\right) = \frac{\pi}{2} + \arcsin\left(\frac{1}{\sqrt{d}}\right)$$

Where the penultimate inequality results from the angle addition formula and the changes in the arguments of the arcsin and arccos result from domain restrictions. Now we have the probability that an edge is cut is

$$\Pr[(i,j) \text{ is cut}] = \frac{\theta}{\pi} = \frac{1}{2} + \frac{1}{\pi} \arcsin\left(\frac{1}{\sqrt{d}}\right) = \frac{1}{2} + \Omega\left(\frac{1}{\sqrt{d}}\right)$$

The last equality is obvious from the Taylor series of the arcsin function. That is, $\arcsin x = \Omega(x)$. To conclude our argument, we must calculate the expectation of the number of edges cut. Using the same notation as before,

$$E[N] = \sum_{e \in E} E[N_e] = |E| \cdot E[N_e] = \left(\frac{1}{2} + \Omega\left(\frac{1}{\sqrt{d}}\right)\right)|E|$$

Since the expected number of edges is the bound we desired, and this represents only one feasible solution to the SDP, we know the optimum solution must yield a cut at least as good as the one above.