CS 270 Algorithms **Spring 2021**

Lecture 19: Semidefinite Programming, Maximum Cut

Lecturer: Prasad Raghavendra Scribe: Debayan Bandyopadhyay

19.1 Introduction

Semidefinite Programming can be thought of a generalization of linear programming. Linear programming, as we know, is a language is a way to express problems and solve them. In a similar vein, Semidefinite Programs (SDPs) are a very expressive and powerful convex optimization primitive. Consider an analogy to the programming languages Python and Assembly. Like a high level coding language, we can use Semidefinite Programming to figure out whether a certain problem is solvable, in polynomial time, but our solution will not be very efficient. Thus, we can use SDPs to get a first-cut solution to a particular problem, before thinking more carefully about how to create an efficient, tailored algorithm for that problem, like a low level program with problem-specific optimizations.

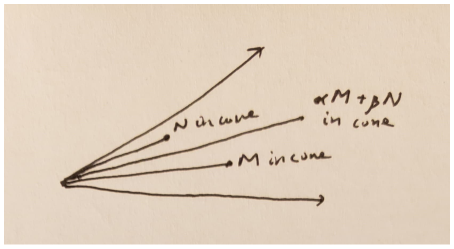
In what sense are SDPs a generalization of linear programming? Linear programming takes linear constraints on variables and provides an assignment of variables that minimizes an objective function. If you take linear programming and add in eigenvalue computations, you get semidefinite programming. In other words, SDPs combine linear programming with spectral techniques. A general semidefinite program can be thought of as a linear program "over" positive semidefinite matrices.

In semidefinite programming, our objective is to find the entries of a real symmetric matrix X which is positive semidefinite, denoted $X = (X_{ij})_{i=1,\dots,n;\, j=1,\dots n} \succcurlyeq 0$.

Definition 19.1. *A real symmetric matrix $M \succcurlyeq 0$, i.e. M is positive semidefinite, is equivalent to the following:*

1. *All eigenvalues of M are nonnegative, $\forall \lambda_i(M), \lambda_i \geq 0$*
2. *The quadratic form is always nonnegative, $x^T M x \geq 0 \, \forall x \in \mathbb{R}^n$*
3. *M can be decomposed as $M = VV^T$ for some $V \in \mathbb{R}^{n\times n}$*
4. *There exists a vector x s.t. the quadratic form is $x^T M x = \sum_i (\sum_j c_{ij} x_j)^2$*
5. *There exist vectors $v_1, \dots, v_n$ s.t. the entries of M are inner products of the vectors, $M_{ij} = \langle v_i, v_j \rangle$*

The set of PSD matrices in $S = \{M | M \succcurlyeq 0,\, M \in \mathbb{R}^{n\times n},\, M = M^T\}$ is a **convex cone**. All this means is that if two matrices M and N are within the cone, i.e. they are positive semidefinite, then the linear combination $\alpha M + \beta N$, for nonnegative constants α and β will also be within the cone. This is not a crucial fact to understand for the rest of the lecture, but it gives a mental picture for the set of all PSD matrices.



In semidefinite programming, we can also provide linear constraints on the matrix.

$$\langle A^{(\ell)}, X\rangle = \sum_{ij} A_{ij}^{(\ell)} X_{ij} \leq b^{(\ell)}\ \forall \ell = 1 \ldots m$$

Definition 19.2. *The inner product of matrices A and X is $\langle A, X\rangle = \sum_{ij} A_{ij} X_{ij}$. This is a natural generalization of the inner product for vectors.*

In contrast to linear programming, we can have any convex objective. In general, we could choose to minimize a convex function of X, or maximize a concave one. For now, let's say that we also have some linear objective function $\min \langle C, X\rangle = \sum C_{ij} X_{ij}$.

$$\begin{gathered}\min \langle C, X\rangle \\ X \succcurlyeq 0 \\ \langle A^{(\ell)}, X\rangle \leq b^{(\ell)}\ \forall \ell = 1 \ldots m\end{gathered}$$

We can also reinterpret this in terms of vectors using the fifth definition of a PSD matrix. We can instead think of the problem as being, find a set of n vectors $v_1, \ldots, v_n \in R^n$ for the objective $\min \sum C_{ij} \langle v_i, v_j\rangle$ subject to $\sum A_{ij}^{(\ell)} \langle v_i, v_j\rangle \leq b^{(\ell)}$.

An important observation is that in the matrix formulation, the constraints were linear in the matrix entries, which correspond to inner products of our vectors. Thus, in this vector formulation, we cannot add constraints on components of these vectors, only on their inner products.

19.2 Maximum Cut

We can use semidefinite programming to approximate solutions to the Maximum Cut (MaxCut) problem.

Input: Graph $G = (V, E)$

Goal: Find a cut $S \cup S^C = V$ that maximizes $|E(S, S^C)|$, the number of crossing edges

MaxCut, unlike its tamer sibling MinCut, is NP-complete, so we have to be satisfied with looking for approximations. A trivial approximation would be to flip a fair coin for each vertex to select its set. This works as a half-approximation, i.e. it gets you at least half of the maximum cut of the graph in expectation. This is because for any edge in the optimal cut, there is a half probability that the edge crosses the cut created by the approximation scheme.

Despite the crudeness of this approximation scheme, it turns out that even Linear Programming approximations don't get you better than a half. Even subexponential size LPs don't do any better than this trivial algorithm. For a long time, there an algorithm which did better than random chance in producing large cuts.

The Goemans-Williamson algorithm finally beat this approximation using SDPs and the excitement surrounding it propelled SDPs as a problem-solving tool. Other problems which used the SDP existed at the time, including the Lovász theta function, but this discovery really kicked things off for the SDPs.

19.3 Goemans-Williamson

19.3.1 SDP formulation

Let's formulate our problem in terms of vectors. What do we want to do? With our graph G, we want to compute vectors $v_1 \dots v_n$ corresponding to the vertices of the graph which denote the cut of G that we care about.

Our intended solution is one in which, for the optimal cut, (S, S^C), our placement of vectors assigns $+1$ for vertices in S and a -1 for vertices in S^C. As we're dealing with vectors, we can envision all of our vectors being unit norm, i.e. $\|v_i\|^2 = 1$, and v_i satisfies the following:

$$v_i = \begin{cases} +1 \cdot \hat{e} \text{ if } i \in S \\ -1 \cdot \hat{e} \text{ if } i \in S^C \end{cases}$$

where $\hat{e}$ is some unit vector.

The number of edges in this cut is easy to analyze in this case:

$$E[S, S^C] = \sum_{(i,j)\in E} \frac{1}{4}\|v_i - v_j\|^2$$

If two vertices are assigned to the same set, then the contribution is zero, but if two vertices are assigned to different sets, then the contribution is 4, hence the fraction of $\frac{1}{4}$ out front.

Thus, the GW-SDP relaxation is:

$$\max \frac{1}{4} \sum_{(i,j)\in E} \|v_i - v_j\|^2$$
$$\langle v_i, v_i \rangle = 1$$

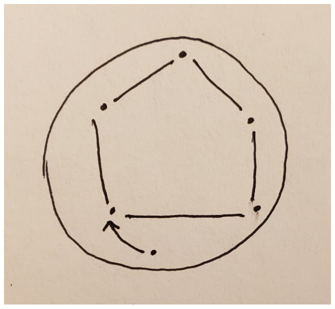
Forcing each vector to be a unit vector only depends on the inner product of a vector with itself, a constraint we can easily enforce in an SDP setting.

For any cut, (S, S^C) or any $\{\pm 1\}^n$ assignment of vertices, we can easily convert that to a feasible point in our SDP formulation. In other words, this is a **relaxation**.

19.3.2 Relaxations

Suppose we have a combinatorial optimization problem, like MaxCut. The space of all solutions is a discrete set, $\{\pm 1\}^n$, assignment of the vertices into S and S^C.

The set of discrete points is hard to optimize over, as it's not a convex set. As such, we can take a convex hull of our space of solutions, Convex Hull(S). We construct the convex hull and optimize over it. But typically convex hulls are exponentially complex, so we would need exponentially many constraints to describe our feasible set. So typically we get a different convex set C that contains this convex hull of S. Now we can optimize over C as we have a simple description of a convex set. We get some point in C. This choice likely won't be one of the original discrete points, and we have to round to one of them to get a meaningful assignment.

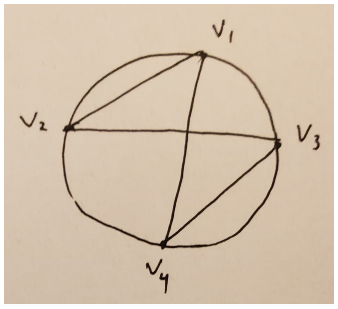


In other words, a relaxation expands the feasible set for which we can use our convex optimization primitives. Once we solve the relaxation, we then round the solution to the relaxation to get an assignment of variables.

Observation: The objective value Optimum GW-SDP(G) $\geq$ MaxCut(G). This is because all cuts, including the maximum cut, are feasible points in the convex set explored by the GW-SDP.

Now that we've solved the $\mathrm{GW}-\mathrm{SDP}(G)$, we have unit vectors which maximize the total squared distance $\frac{1}{4}\sum_{(i,j)\in E}\|v_i - v_j\|^2$. Great. But how do we recover the cut associated with this solution?

19.3.3 Rounding



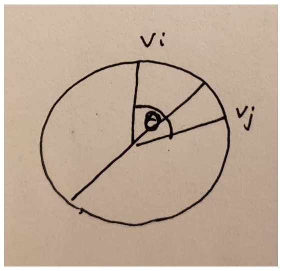
Suppose this is the embedding of our graph from Goemans-Williamson, with vectors on the unit sphere in n dimensions.

We can round these vectors in the following way: Cut the unit sphere by a random hyperplane going through the origin, thereby separating the vectors into two half spaces. Call those sets of vectors S and S^C. We can perform this separation using a random vector $g \in \mathbb{R}^n$ serving as the normal vector to the hyperplane.

So we can use the following assignment with our random vector g:

$$x_i = \begin{cases} +1, & \langle g, v_i \rangle > 0, \quad (i \in S) \\ -1, & \text{else}, \quad (i \in S^C) \end{cases}$$

How well does this rounding do? We can analyze this on an edge by edge basis. Let's look at a particular edge $(i, j) \in E$. We have two unit vectors, v_i and v_j.



What is the probability that the edge is cut, $\mathbb{P}[(i, j) \text{ is cut}]$?

At first, this seems daunting, but if you're only interested in, say, three, vectors in n dimensional space, the subspace of those three vectors is isomorphic to three dimensional space. If we make our lives easier and only look at the plane spanned by v_i and v_j, the hyperplane is going to cut this plane in some way.

The two vectors are separated only if the hyperplane intersects the arc between v_i and v_j. This probability is just $\frac{\theta}{\pi} = \frac{\arccos \langle v_i, v_j \rangle}{\pi}$. (For example, an equivalent way of generating the hyperplane's intersection through this plane is choosing a point on the boundary of a semicircle containing this arc and drawing a diameter from that point.)

The expected number of edges cut is then,

$$\begin{aligned} \mathbb{E}(|E(S, S^C)|) &= \mathbb{E}[\sum_{(i,j) \in E} 1[(i, j) \text{ is cut}]] \\ &= \mathbb{E}[\sum_{(i,j) \in E} \mathbb{P}[(i, j) \text{ is cut}] \\ &= \sum_{(i,j) \in E} \frac{\arccos \langle v_i, v_j \rangle}{\pi} \end{aligned}$$

We want to compare this to the best possible cut, the MaxCut of G. We do not actually know what the Maximum Cut is, but we do have an upper bound on this quantity.

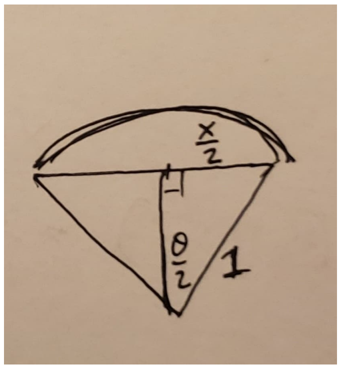
Claim 19.3. *We'll show* $\mathbb{E}(|E(S,S^C)|) \geq \alpha \cdot GW\text{-}SDP(G)$.

$$\sum_{(i,j)\in E} \frac{\arccos\langle v_i, v_j\rangle}{\pi} \geq \alpha \cdot \sum_{(i,j)\in E} \frac{1}{4}\|v_i - v_j\|^2$$

Remember that by relaxation, GW-SDP$(G) \geq$ MaxCut(G). Thus, this will be sufficient to show that there's at least an alpha approximation to the MaxCut of the graph.

Let's do it term by term. Let's look at a pair of vectors on each side.

The term on the left is $\frac{\arccos\langle v_i,v_j\rangle}{\pi} = \frac{\theta}{\pi}$. The term on the right $\frac{1}{4}\|v_i - v_j\|^2 = \sin^2\theta/2$, from geometry (consider the length of the chord in the drawing below).



Let $\alpha_{GW} = \min_{\theta\in(0,\pi)} \frac{\theta/\pi}{\sin^2\theta/2} \approx 0.878\ldots$ Then, $\frac{\arccos\langle v_i,v_j\rangle}{\pi} \geq \alpha_{GW} \cdot \frac{1}{4}\|v_i - v_j\|^2$.

Therefore, the sum over all edges also satisfies this inequality, proving our claim. Thus, we get an alpha approximation to MaxCut, where alpha is around 0.878.

It can be shown NP-hard to approximate the maximum cut to a better approximation under the Unique Games Conjecture. Unfortunately, we don't know whether the conjecture is true. In other words, there is conditional evidence that this value is the best we could hope to do.

On the other hand, this rounding is tight. We cannot improve our rounding of the vectors from SDP formulation to get a better cut.

19.3.4 Runtime Discussion

The runtime for Goemans-Williamson is $O(\text{poly}(n) \cdot \log(1/\epsilon))$ using the ellipsoid method, for example. We could even use interior point methods. Of course, these methods are not very tailored to our problem, so we can do something like Mirror Descent, which in this case would yield matrix multiplicative weights, which solves it in near linear time $\tilde{O}(|E| \cdot \text{poly}(1/\epsilon))$.

One thing to note is that unlike an LP, an SDP with integer constraints need not yield a rational result, so we cannot hope to solve it exactly.

19.3.5 Connection to the Laplacian

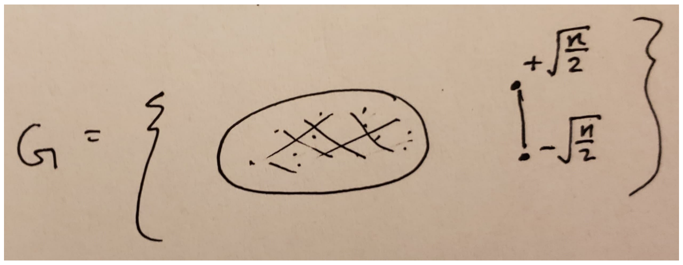
The objective we are maximizing is just a Laplacian of the graph, as a quadratic form. In other words,

$$x^T L_G x = \sum_{(i,j)\in E} (x_i - x_j)^2$$
$$\implies \text{MaxCut}(G) = \max_{x\in\{\pm 1\}^n} x^T L_G x$$

So we could try to write the problem in the following way and just compute the largest eigenvector of the Laplacian:

$$\text{MaxCut}(G) \overset{?}{=} \max_{x\in\mathbb{R}^n,\, \|x\|^2=n} x^T L_G x$$

In fact, you could do this. But what goes wrong is that we get an n dimensional vector x whose length is n, but the vector may not be balanced in any meaningful way. In other words, we want every coordinate to be magnitude unity, but we have no way of enforcing this requirement with the eigenvectors of the Laplacian. The solution could put all the mass on a particular coordinate, giving us a useless solution to our cut problem. Here's a particularly bad example.



The largest eigenvector could put all its mass on the edge disjointed from the large connected component. SDP, on the other hand, lets you combine these eigenvalues and vectors with linear constraints that force each vector to be unit norm as well. The fact that SDPs naturally allow us to convey spectral information and provide linear constraints is what makes them so useful here.

Here's another (cute) application of SDPs:

19.4 Betweenness Problem

A lot of the problems we talk about are Constraint Satisfaction Problems (CSPs) which ask for an assignment of variables that satisfy as many constraints on those variables as possible.

The **Betweenness problem** says:

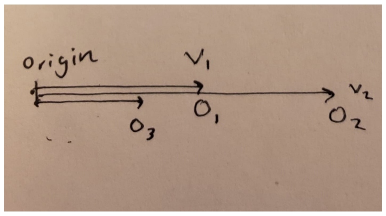
Find a permutation of objects $\{O_1, \ldots, O_n\}$ which satisfies n betweenness constraints, of the following form: Object O_i is positioned between objects O_j and O_k.

Suppose we have a guarantee that there is some permutation of objects that satisfies all of the betweenness constraints.

Goal: Satisfy as many betweenness constraints as possible. (Satisfying all of them is NP complete.)

For each O_i, we have a position $p(i) \in [1, n]$. In essence, we are solving for these positions. The constraint O_i between O_j and O_k is the same as saying $p(j) < p(i) < p(k)$ or $p(k) < p(i) < p(j)$. How do we find a solution like this?

Suppose we had a permutation which satisfied all constraints. We could order the variables O_i on the real line, with the origin somewhere. The vectors v_i will just be the vector from the origin to the position $p(i)$.



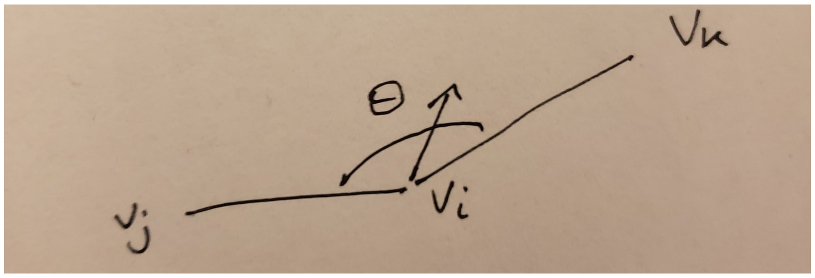
If we want O_i to be between O_j and O_k, then $v_j - v_i$ and $v_k - v_i$ must point in opposite directions. Mathematically, their inner product must be negative. So for each betweenness constraint, create a linear constraint for the SDP formulation, $\langle v_j - v_i, v_k - v_i \rangle = \langle v_j, v_k \rangle - \langle v_j, v_i \rangle - \langle v_i, v_k \rangle + \langle v_i, v_i \rangle < 0$. In other words, we can require that the angle between $v_j - v_i$ and $v_k - v_i$ be obtuse. We want to find the vectors v_i such that this is true. (Note that there is no objective function here, this is a feasibility SDP. This is possible because all betweenness constraints were guaranteed to be satisfied.)

We have n vectors now after solving the following SDP. How can we obtain positions $p(i)$ from these vectors? Choosing random vectors seemed to work well for us, so let's just try our luck.

In other words, pick a random $g \in \mathbb{R}^n$ and compute $x_i = \langle v_i, g \rangle$ for all the vectors v_i we've obtained from our SDP. Now our positions are $p(i)$ can just be determined from an ordering of x_i on the real number line. So now we have a permutation from our SDP. How well does it perform?

Claim 19.4.

$$\mathbb{P}(O_i \text{ between } O_j \text{ \& } O_k) = \frac{\theta}{\pi} \geq \frac{1}{2} \tag{19.1}$$



Why is this claim true? For object O_i to be between O_j and O_k, we need that the vector g shouldn't be oriented between $v_j - v_i$ and $v_k - v_i$ as shown in the image above. Another way of saying is that a vector normal to g should be oriented between the two, which happens with the claimed probability. But because we've guaranteed all of our betweenness constraints lead to obtuse angles, this probability exceeds half.

19.5 Concluding Remarks

In this lecture, we didn't get a chance to show how SDPs are like a higher level programming language. It seems to be the case that in SDP problems, geometry plays a crucial role. But there is no apparent geometry in problems like 3SAT, for example. Is there hope to use SDPs more generally? It turns out there systematic ways of performing these relaxations that we'll discuss next time.