CS261: A Second Course in Algorithms Lecture #20: The Maximum Cut Problem and Semidefinite Programming[*]

Tim Roughgarden[†]

March 10, 2016

1 Introduction

Now that you're finishing CS261, you're well equipped to comprehend a lot of advanced material on algorithms. This lecture illustrates this point by teaching you about a cool and famous approximation algorithm.

In the *maximum cut* problem, the input is an undirected graph $G = (V, E)$ with a nonnegative weight $w_e \geq 0$ for each edge $e \in E$. The goal is to compute a *cut* — a partition of the vertex set into sets A and B — that maximizes the total weight of the cut edges (the edges with one endpoint in each of A and B).

Now, if it were the *minimum* cut problem, we'd know what to do — that problem reduces to the maximum flow problem (Exercise Set #2). It's tempting to think that we can reduce the maximum cut problem to the minimum cut problem just by negating the weights of all of the edges. Such a reduction would yield a minimum cut problem with negative weights (or capacities). But if you look back at our polynomial-time algorithms for computing minimum cuts, you'll notice that we assumed nonnegative edge capacities, and that our proofs depended on this assumption. Indeed, it's not hard to prove that the maximum cut problem is NP-hard. So, let's talk about polynomial-time approximation algorithms.

It's easy to come up with a $\frac{1}{2}$-approximation algorithm for the maximum cut problem. Almost anything works — a greedy algorithm, local search, picking a random cut, linear programming rounding, and so on. But frustratingly, none of these techniques seemed capable of proving an approximation factor better than $\frac{1}{2}$. This made it remarkable when, in 1994, Goemans and Williamson showed how a new technique, "semidefinite programming rounding," could be used to blow away all previous approximation algorithms for the maximum cut problem.

[*]©2016, Tim Roughgarden.

[†]Department of Computer Science, Stanford University, 474 Gates Building, 353 Serra Mall, Stanford, CA 94305. Email: `tim@cs.stanford.edu`.

2 A Semidefinite Programming Relaxation for the Maximum Cut Problem

2.1 A Quadratic Programming Formulation

To motivate a novel relaxation for the maximum cut problem, we first reformulate the problem exactly via a quadratic program. (So solving this program is also NP-hard.) The idea is to have one decision variable y_i for each vertex $i \in V$, indicating which side of the cut the vertex is on. It's convenient to restrict y_i to lie in $\{-1, +1\}$, as opposed to $\{0, 1\}$. There's no need for any other constraints. In the objective function, we want an edge (i, j) of the input graph $G = (V, E)$ to contribute w_{ij} whenever i, j are on different sides of the cut, and 0 if they are on the same side of the cut. Note that $y_i y_j = +1$ if i, j are on the same side of the cut and $y_i y_j = -1$ otherwise. Thus, we can formulate the maximum cut objective function exactly as

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

Note that the contribution of edge (i, j) to the objective function is w_{ij} if i and j are on different sides of the cut and 0 otherwise, as desired. There is a one-to-one and objective-function-preserving correspondence between cuts of the input graph and feasible solutions to this quadratic program.

This quadratic programming formulation has two features that make it a non-linear program: the integer constraints $y_i \in \{\pm 1\}$ for every $i \in V$, and the quadratic terms $y_i y_j$ in the objective function.

2.2 A Vector Relaxation

Here's an inspired idea for a relaxation: rather than requiring each y_i to be either -1 or +1, we only ask that each decision variable is a *unit vector in* $\mathbb{R}^n$, where $n = |V|$ denotes the number of vertices. We henceforth use x_i to denote the (vector-valued) decision variable corresponding to the vertex $i \in V$. We can think of the values +1 and -1 as the special cases of the unit vectors $(1, 0, 0, \ldots, 0)$ and $(-1, 0, 0, \ldots, 0)$. There is an obvious question of what we mean by the quadratic term $y_i \dot{y}_j$ when we switch to decision variables that are n-vectors; the most natural answer is to replace the scalar product $y_i \cdot y_j$ by the inner product $\langle x_i, x_j \rangle$. We then have the following "vector programming relaxation" of the maximum cut problem:

$$\max \frac{1}{2} \sum_{(i,j) \in E} w_{ij} \left(1 - \langle x_i, x_j \rangle\right)$$

subject to

$$\|x_i\|_2^2 = 1 \qquad \text{for every } i \in V.$$

It may seem obscure to write $\|x_i\|_2^2 = 1$ rather than $\|x_i\|_2 = 1$ (which is equivalent); the reason for this will become clear later in the lecture. Since every cut of the input graph G

maps to a feasible solution of this relaxation with the same objective function value, and the vector program only maximizes over more stuff, we have

$$\text{vector } OPT \geq OPT.$$

Geometrically, this relaxation maps all the vertices of the input graph G to the unit sphere in $\mathbb{R}^n$, while attempting to map the endpoints of each edge to points that are as close to antipodal as possible (to get $\langle x_i, x_j \rangle$ as close to -1 as possible).

2.3 Disguised Convexity

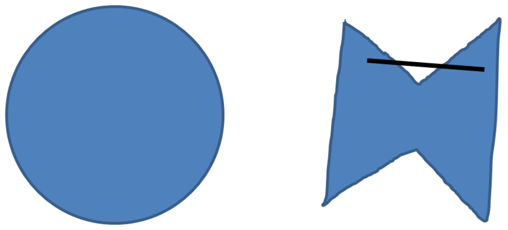

Figure 1: (a) a circle is convex, but (b) is not convex;the chord shown is not contained entirely in the set.

It turns out that the relaxation above can be solved to optimality in polynomial time.[1] You might well find this counterintuitive, given that the inner products in the objective function seem hopelessly quadratic. The moral reason for computational tractability is *convexity*. Indeed, a good rule of thumb very generally is to equate computational tractability with convexity. A mathematical program can be convex in two senses. The first sense is the same as that we discussed back in Lecture #9 — a subset of $\mathbb{R}^n$ is convex if it contains all of its chords. (See Figure 1.) Recall that the feasible region of a linear program is always convex in this sense. The second sense is that the objective function can be a convex function. (A linear function is a special case of a convex function.) We won't need this second type of convexity in this lecture, but it's extremely useful in other contexts, especially in machine learning.

OK. . . but where's the convexity in the vector relaxation above? After all, if you take the average of two points on the unit sphere, you don't get another point on the unit sphere.

We next expose the disguised convexity. A natural idea to remove the quadratic (inner product) character of the vector program above is to linearize it, meaning to introduce a new decision variable p_{ij} for each $i, j \in V$, with the intention that p_{ij} will take on the value $\langle x_i, x_j \rangle$. But without further constraints, this will lead to a relaxation of the relaxation —

[1]Strictly speaking, since the optimal solution might be irrational, we only solve it up to arbitrarily small error.

nothing is enforcing the p_{ij}'s to actually be of the form $\langle x_i, x_j \rangle$ for some collection $x_1, \ldots, x_n$ of n-vectors, and the p_{ij}'s could form an arbitrary matrix instead. So how can we enforce the intended semantics?

This is where elementary linear algebra comes to the rescue. We'll use some facts that you've almost surely seen in a previous course, and also have almost surely forgotten. That's OK — if you spend 20-30 minutes with your favorite linear algebra textbook (or Wikipedia), you'll remember why all of these relevant facts are true (none are difficult).

First, let's observe that a $V \times V$ matrix $P = \{p_{ij}\}$ is of the form $p_{ij} = \langle x_i, x_j \rangle$ for some vectors $x_1, \ldots, x_n$ (for every $i, j \in V$) if and only if we can write

$$P = X^T X \tag{1}$$

for some matrix $X \in \mathbb{R}^{V \times V}$. Recalling the definition of matrix multiplication, the (i, j) entry of $X^T X$ is the inner product of the ith row of X^T and the jth column of X, or equivalently the inner product of the ith and jth columns of X. Thus, for matrices P of the desired form, the columns of the matrix X provide the n-vectors whose inner products define all of the entries of P.

Matrices that are "squares" in the sense of (1) are extremely well understood, and they are called (symmetric) *positive semidefinite (psd)* matrices. There are many characterizations of symmetric psd matrices, and none are particularly hard to prove. For example, a symmetric matrix is psd if and only if all of its eigenvalues are nonnegative. (Recall that a symmetric matrix has a full set of real-valued eigenvalues.) The characterization that exposes the latent convexity in the vector program above is that a symmetric matrix P is psd if and only if

$$\underbrace{z^T P z}_{\text{"quadratic form"}} \geq 0 \tag{2}$$

for every vector $z \in \mathbb{R}^n$. Note that the forward direction is easy to see (if P can be written $P = X^T X$ then $z^T P z = (Xz)^T (Xz) = \|Xz\|_2^2 \geq 0$); the (contrapositive of the) reverse direction follows easily from the eigenvalue characterization already mentioned.

For a fixed vector $z \in \mathbb{R}^n$, the inequality (2) reads

$$\sum_{i,j \in V} p_{ij} z_i z_j \geq 0,$$

which is *linear* in the p_{ij}'s (for fixed z_i's). And remember that the p_{ij}'s are our decision variables!

2.4 A Semidefinite Relaxation

Summarizing the discussion so far, we've argued that the vector relaxation in Section 2.2 is equivalent to the linear program

$$\max \frac{1}{2} \sum_{(i,j) \in E} w_{ij} \left(1 - p_{ij}\right)$$

subject to

$$\sum_{i,j \in V} p_{ij} z_i z_j \geq 0 \qquad \text{for every } z \in \mathbb{R}^n \tag{3}$$

$$p_{ij} = p_{ji} \qquad \text{for every } i, j \in V \tag{4}$$

$$p_{ii} = 1 \qquad \text{for every } i \in V. \tag{5}$$

The constraints (3) and (4) enforce the p.s.d. and symmetry constraints on the p_{ij}'s. Their presence makes this program a *semidefinite program (SDP)*. The final constraints (5) correspond to the constraints that $\|x_i\|_2^2 = 1$ for every $i \in V$ — that the matrix formed by the p_{ij}'s not only has the form $X^T X$, but has this form for a matrix X whose columns are unit vectors.

2.5 Solving SDPs Efficiently

The good news about the SDP above is that every constraint is linear in the p_{ij}'s, so we're in the familiar realm of linear programming. The obvious issue is that the linear program has an infinite number of constraints of the form (3) — one for each real-valued vector $z \in \mathbb{R}^n$. So there's no hope of even writing this SDP down. But wait, didn't we discuss an algorithm for linear programming that can solve linear programs efficiently even when there are too many constraints to write down?

The first way around the infinite number of constraints is to use the ellipsoid method (Lecture #10) to solve the SDP. Recall that the ellipsoid method runs in time polynomial in the number of variables (n^2 variables in our case), provided that there is a polynomial-time *separation oracle* for the constraints. The responsibility of a separation oracle is, given an allegedly feasible solution, to either verify feasibility or else produce a violated constraint. For the SDP above, the constraints (4) and (5) can be checked directly. The constraints (3) can be checked by computing the eigenvalues and eigenvectors of the matrix formed by the p_{ij}'s.[2] As mentioned earlier, the constraints (3) are equivalent to this matrix having only nonnegative eigenvalues. Moreover, if the p_{ij}'s are not feasible and there is a negative eigenvalue, then the corresponding eigenvector serves as a vector z such that the constraint (3) is violated.[3] This separation oracle allows us to solve SDPs using the ellipsoid method.

The second solution is to use "interior-point methods," which were also mentioned briefly at the end of Lecture #10. State-of-the-art interior-point algorithms can solve SDPs both in theory (meaning in polynomial time) and in practice, meaning for medium-sized problems. SDPs are definitely harder in practice than linear programs, though — modern solvers have trouble going beyond thousands of variables and constraints, which is a couple orders of magnitude smaller than the linear programs that are routinely solved by commercial solvers.

[2] There are standard and polynomial-time matrix algorithms for this task; see any textbook on numerical analysis.

[3] If z is an eigenvector of a symmetric matrix P with eigenvalue λ, then $z^T P z = z^T(\lambda z) = \lambda \cdot \|z\|_2^2$, which is negative if and only if λ is negative.

A third option for many SDPs is to use an extension of the multiplicative weights algorithm (Lecture #11) to quickly compute an approximately optimal solution. This is similar in spirit to but somewhat more complicated than the application to approximate maximum flows discussed in Lecture #12.[4]

Henceforth, we'll just take it on faith that our SDP relaxation can be solved in polynomial time. But the question remains: what do we do with the solution to the relaxation?

3 Randomized Hyperplane Rounding

The SDP relaxation above of the maximum cut problem was already known in the 1980s. But only in 1994 did Goemans and Williamson figure out how to round its solution to a near-optimal cut. First, it's natural to round the solution of the vector programming relaxation (Section 2.2) rather than the equivalent SDP relaxation (Section 2.4), since the former ascribes one object (a vector) to each vertex $i \in V$, while the latter uses one scalar for each pair of vertices.[5] Thus, we "just" need to round each vector to a binary value, while approximately preserving the objective function value.

The first key idea is to use randomized rounding, as first discussed in Lecture #18. The second key idea is that a simple way to round a vector to a binary value is to look at which side of some hyperplane it lies on (cf., the machine learning examples in Lectures #7 and #12). See Figure 2. Combining these two ideas, we arrive at randomized hyperplane rounding.

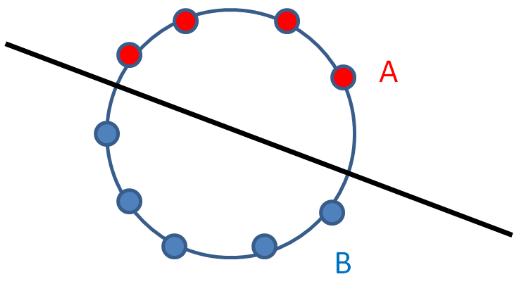

Figure 2: Randomized hyperplane rounding: points with positive dot product in set A, points with negative dot product in set B.

[4]Strictly speaking, the first two solutions also only compute an approximately optimal solution. This is necessary, because the optimal solution to an SDP (with all integer coefficients) might be irrational. (This can't happen with a linear program.) For a given approximation ϵ, the running time of the ellipsoid method and interior-point methods depend on $\log \frac{1}{\epsilon}$, while that of multiplicative weights depends inverse polynomially on $\frac{1}{\epsilon}$.

[5]After solving the SDP relaxation to get the matrix P of the p_{ij}'s, another standard matrix algorithm ("Cholesky decomposition") can be used to efficiently recover the matrix X in the equation $P = X^T X$ and hence the vectors (which are the columns of X).

```
**Randomized Hyperplane Rounding**

given: one vector $x_i$ for each $i \in V$
choose a random unit vector $r \in \mathbb{R}^n$
set $A = \{i \in V : \langle x_i, r\rangle \geq 0\}$
set $B = \{i \in V : \langle x_i, r\rangle < 0\}$
return the cut $(A, B)$
```

Thus, vertices are partitioned according to which side of the hyperplane with normal vector r they lie on. You may be wondering how to choose a random unit vector in $\mathbb{R}^n$ in an algorithm. One simple way is: sample n independent standard Gaussian random variables (with mean 0 and variance 1) $g_1, \ldots, g_n$, and normalize to get a unit vector:

$$r = \frac{(g_1, \ldots, g_n)}{\|(g_1, \ldots, g_n)\|}.$$

(Or, note that the computed cut doesn't change if we don't bother to normalize.) The main property we need of the distribution of r is spherical symmetry — that all vectors at a given distance from the origin are equally likely.

We have the following remarkable theorem.

Theorem 3.1 *The expected weight of the cut produced by randomized hyperplane rounding is at least .878 times the maximum possible.*

The theorem follows easily from the following lemma.

Lemma 3.2 *For every edge $(i, j) \in E$ of the input graph,*

$$\mathbf{Pr}[(i,j) \textit{ is cut}] \geq .878 \cdot \underbrace{\left[\frac{1}{2}(1 - \langle x_i, x_j\rangle)\right]}_{\textit{contribution to SDP}}.$$

Proof of Theorem 3.1: We can derive

$$\begin{aligned}\mathbf{E}[\text{weight of } (A,B)] &= \sum_{(i,j)\in E} w_{ij} \cdot \mathbf{Pr}[(i,j) \text{ is cut}] \\ &\geq .878 \cdot \sum_{(i,j)\in E} \left[\frac{1}{2}(1 - \langle x_i, x_j\rangle)\right] \\ &\geq .878 \cdot OPT,\end{aligned}$$

where the equation follows from linearity of expectation (using one indicator random variable per edge), the first inequality from Lemma 3.2, and the second inequality from the fact that the x_i's are an optimal solution to vector programming relaxation of the maximum cut problem. ■

We conclude by proving the key lemma.

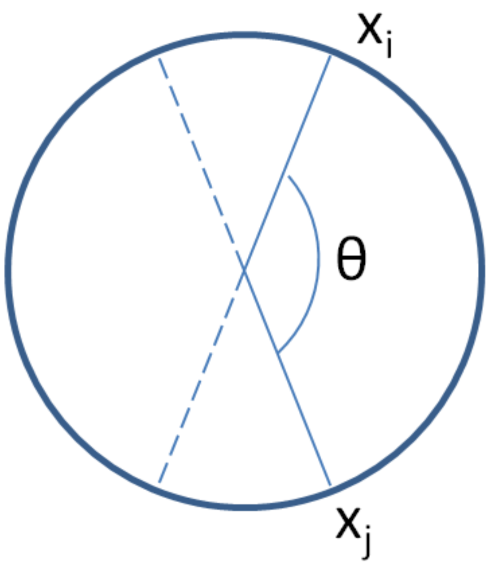

Figure 3: x_i and x_j are placed on different sides of the cut with probability θ/π.

Proof of Lemma 3.2: Fix an edge $(i, j) \in E$. Consider the two-dimensional subspace (through the origin) spanned by the vectors x_i and x_j. Since r was chosen from a spherically symmetric distribution, its projection onto this subspace is also spherically symmetric — it's equally likely to point in any direction. The vertices x_i and x_j are placed on different sides of the cut if and only if they are "split" by the projection of r. (Figure 3.) If we let θ denote the angle between x_i and x_j in this subspace, then 2θ out of the 2π radians of possible directions result in the edge (i, j) getting cut. So we know the cutting probability, as a function of θ:

$$\mathbf{Pr}[(i, j) \text{ is cut}] = \frac{\theta}{\pi}.$$

We still need to understand $\frac{1}{2}(1 - \langle x_i, x_j \rangle)$ as a function of θ. But remember from precalculus that $\langle x_i, x_j \rangle = \|x_i\| \|x_j\| \cos \theta$. And since x_i and x_j are both unit vectors (in the original space and also the subspace that they span), we have

$$\frac{1}{2}(1 - \langle x_i, x_j \rangle) = \tfrac{1}{2}(1 - \cos \theta).$$

The lemma thus boils down to verifying that

$$\frac{\theta}{\pi} \geq .878 \cdot \left[\tfrac{1}{2}(1 - \cos \theta)\right]$$

for all possible values of $\theta \in [0, \pi]$. This inequality is easily seen by plotting both sides, or if you're a stickler for rigor, by computations familiar from first-year calculus. ■

4 Going Beyond .878

For several lectures we were haunted by the number $1 - \frac{1}{e}$, which seemed like a pretty weird number. Even more bizarrely, it is provably the best-possible approximation guarantee for several natural problems, including online bipartite matching (Lecture #14) and, assuming $P \neq NP$, set coverage (Lecture #15).

Now the .878 in this lecture seems like a *really* weird number. But there is some evidence that it might be optimal! Specifically, in 2005 it was proved that, assuming that the "Unique Games Conjecture (UGC)" is true (and $P \neq NP$), there is no polynomial-time algorithm for the maximum cut problem with approximation factor larger than the one proved by Goemans and Williamson. The UGC (which is only from 2002) is somewhat technical to state precisely — it asserts that a certain constraint satisfaction problem is NP-hard. Unlike the $P \neq NP$ conjecture, which is widely believed, it is highly unclear whether the UGC is true or false. But it's amazing that *any* plausible complexity hypothesis implies the optimality of randomized hyperplane rounding for the maximum cut problem.