CS498: Algorithmic Engineering

Semidefinite Programming and Applications

Guest lecture: Chandra Chekuri

University of Illinois Urbana-Champaign

Outline

Convex Programs

$$\min f(x), \quad x \in S \subseteq \mathcal{R}^n$$

where $f(x)$ is a convex function and S is a convex set

Difficulty and complexity of solving can come from f or from S

Note: For any convex function f and value B, $\{x \mid f(x) \leq B\}$ is a convex set

Convex Programs

$$\min f(x), \quad x \in S \subseteq \mathcal{R}^n$$

where $f(x)$ is a convex function and S is a convex set

To solve the problem efficiently, following are sufficient:

- given x, evaluate $f(x)$ and its gradient ∇f (sub-gradient if f is not smooth)
- given x, output if $x \in S$ or not and if it is not then also output a separating hyper-plane that separates x from S (one always exists for convex set)
- Due to precision issues one cannot get an exact solution but an additive ε approximation for any desired $\varepsilon > 0$ in time that grows with $\log(1/\varepsilon)$

Applications of Convex Programs beyond LPs

- Engineering, statistics, continuous optimization, machine learning, ...
- Fewer direct applications in discrete optimization but some spectacular successes via semidefinite programming.

Semidefinite Programming (SDP)

- A special class of convex programs
- Advantage of SDPs: a natural modeling language for a certain class of quadratic programming problems
- Disadvantage: solving large scale SDPs is still a bit slow

Goal of two lectures: some applications of SDP and its modeling power

SDP: Positive Semidefinite Matrices

SDP is based on the properties of *positive semi-definite* (PSD) matrices:

A $n \times n$ *real symmetric* matrix A is *psd* if any of the following conditions are true:

1. $x^t A x \geq 0$ for all $x \in \mathbb{R}^n$
2. all eigen values of A are real and *non-negative*
3. A can be written as $W^t W$ for a real matrix W

Notation: $X \succeq 0$ to indicate that X is psd

SDP: Positive Semidefinite Matrices

SDP is based on the properties of *positive semi-definite* (PSD) matrices:

A $n \times n$ *real symmetric* matrix A is *psd* if any of the following conditions are true:

1. $x^t A x \geq 0$ for all $x \in \mathbb{R}^n$
2. all eigen values of A are real and *non-negative*
3. A can be written as $W^t W$ for a real matrix W

Notation: $X \succeq 0$ to indicate that X is psd

$X \succ 0$ is positive definite if $x^t A x > 0$ for all $x \in \mathbb{R}^n$

Understanding quadratic form

$x \in \mathbb{R}^n$ and A is $n \times n$ matrix.

$$x^T A x = \sum_{i=1}^{n} \sum_{j=1}^{n} A_{i,j} x_i x_j = \sum_{i=1}^{n} A_{i,i} x_i^2 + \sum_{1 \le i < j \le n} (A_{i,j} + A_{j,i}) x_i x_j$$

A quadratic function on n variables:

$$f(x) = \sum_{i=1}^{n} a_i x_i^2 + \sum_{1 \le i < j \le n} a_{\{i,j\}} x_i x_j + \sum_{i=1}^{n} b_i x_i + c$$

Rewriting above as $f(x) = x^T A x + b^T x + c$ where A is a *symmetric* matrix such that $A_{i,i} = a_i$ and $A_{i,j} = A_{j,i} = \frac{1}{2} a_{\{i,j\}}$

Quadratic functions and convexity

Lemma

A smooth function $f : D \to \mathbb{R}$ *is convex iff the Hessian* $\nabla^2 f(x)$ *is psd for all* $x \in D$.

Proof Sketch.

Recall f is convex iff $f(y) \geq f(x) + (y-x)^T \nabla f(x)$ for all $y, x \in D$. Sufficient to check this for all y close to x. Using Taylor expansion in a small neighborhood:

$$f(y) \simeq f(x) + (y-x)^T \nabla f(x) + \frac{1}{2}(y-x)^T \nabla^2 f(x)(y-x)$$

Hence $\nabla^2 f \succeq 0$ is necessary and sufficient to ensure convexity. □

Convex Programs

$$\min f(x), \quad x \in S \subseteq \mathcal{R}^n$$

where $f(x)$ is a convex function and S is a convex set

Difficulty and complexity can come from f or from S

PSD Matrices and Convexity

Let M_n be the set of all $n \times n$ real matrices

Lemma

Let $A, B \in M_n$. If $A \succeq 0$ and $B \succeq 0$, then for all $a, b \geq 0$, $aA + bB \succeq 0$.

Proof.

Use the characterization $A \succeq 0$ iff $x^T A x \geq 0$ for all $x \in \mathbb{R}^n$. □

PSD Matrices and Convexity

Let M_n be the set of all $n \times n$ real matrices

Lemma

Let $A, B \in M_n$. If $A \succeq 0$ and $B \succeq 0$, then for all $a, b \geq 0$, $aA + bB \succeq 0$.

Proof.

Use the characterization $A \succeq 0$ iff $x^T Ax \geq 0$ for all $x \in \mathbb{R}^n$. □

Lemma

The set of all $n \times n$ psd matrices is a convex cone in $\mathbb{R}^{n^2}$.

Important: We are interpreting M_n as vectors in $\mathbb{R}^{n^2}$.

PSD Matrices and Convexity

Let M_n be the set of all $n \times n$ real matrices

Lemma

The set of all $n \times n$ psd matrices is a convex cone in $\mathbb{R}^{n^2}$.

The cone is denoted by S_n^+.

SDP: Formulation

- The set of all $n \times n$ psd matrices is a convex set in $\mathbb{R}^{n^2}$
- Given two matrices $A, B \in M_n$ define: $A \cdot B = \sum_{i,j} a_{ij} b_{ij}$ (equivalently their dot product when viewed as vectors)
- The SDP problem is given by matrices $C, D_1, D_2, \ldots, D_k \in M_n$ and scalars $d_1, d_2, \ldots, d_k$. The variables are given by a matrix $Y \in M_n$:

$$\begin{aligned} \max \quad & C \cdot Y \\ \text{s.t.} \quad & D_i \cdot Y = d_i \quad 1 \leq i \leq k \\ & Y \succeq 0 \\ & Y \in M_n \end{aligned}$$

SDP: The PSD Constraint

$$
\begin{aligned}
\max \quad & C \cdot Y \\
\text{s.t.} \quad & D_i \cdot Y = d_i \quad 1 \leq i \leq k \\
& Y \succeq 0 \\
& Y \in M_n
\end{aligned}
$$

- The constraint $Y \succeq 0$ is a shorthand to say that Y is constrained to be psd
- We can allow minimization in the objective function and the equalities in the constraints can be inequalities

Claim: SDP is a special case of convex programming.

SDP: The PSD Constraint

$$\begin{aligned} \max \quad & C \cdot Y \\ \text{s.t.} \quad & D_i \cdot Y = d_i \quad 1 \le i \le k \\ & Y \succeq 0 \\ & Y \in M_n \end{aligned}$$

Can write the psd contraint via an infinite set of linear constraints:

$$\begin{aligned} \max \quad & C \cdot Y \\ \text{s.t.} \quad & D_i \cdot Y = d_i \quad 1 \le i \le k \\ & v^T Y v \ge 0 \quad v \in \mathbb{R}^n \\ & y_{i,j} = y_{j,i} \quad 1 \le i < j \le n \\ & Y \in M_n \end{aligned}$$

Solvability of SDP: The Separation Oracle

- The algorithm first checks to see if A is symmetric
- If not then A is not psd and a separating hyper-plane is the constraint $y_{ij} = y_{ji}$
- Then it computes eigenvalues of A : if all are non-negative then A is psd
- Otherwise there is an eigen-vector v of A such that:

$$Av = \lambda v \ \text{ and } \ \lambda < 0 \quad \text{which implies} \quad v^t A v = \lambda < 0$$

- and hence the violated hyper-plane is simply $v^t Y v \geq 0$

Solutions of SDP as Vectors

Given a solution A to an SDP we can interpret A as a collection of vectors $v_1, v_2, \ldots, v_n$ as follows.

- From property 3 of psd matrices, there exists W such that $A = W^t W$
- Let $v_1, v_2, \ldots, v_n$ be the columns of W
- Then it follows that $A_{ij} = v_i \cdot v_j$ with the usual inner product between vectors
- Thus SDP is equivalent to vector programming defined in next slide

Vector Programming

- In vector programming the "variables" are vectors $v_1, v_2, \ldots, v_n$ that are allowed to live in any dimension (although we will restrict them to be in $\mathbb{R}^n$).
- The objective function and constraints are linear in the "actual" variables, namely the inner products $v_i \cdot v_j$.
- Example:

$$\begin{aligned} \max \quad & (1 - v_1 \cdot v_2 + v_2 \cdot v_3) \\ \text{s.t.} \quad & v_1 \cdot v_2 - v_2 \cdot v_3 = 5 \\ & v_1,\ v_2,\ v_3 \in \mathbb{R}^n \end{aligned}$$

SDP and Vector Programming

- From the previous discussion it is easy to see that SDP and vector programming are the same
- To implement vector programming via SDP we use variables y_{ij} for $v_i \cdot v_j$ and constrain Y to be psd
- To implement SDP via vector programming we simply use $v_i \cdot v_j$ for y_{ij}

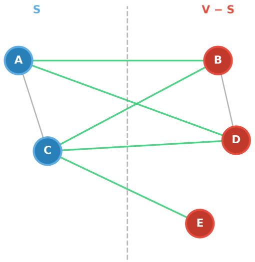
The advantage of vector programming is that it is useful to model certain class of combinatorial problems as we will see.

Max Cut

Input: Edge-weighted graph $G = (V, E)$

Output: partition of V into $(S, V \setminus S)$ to *maximize* $w(\delta(S))$

Another interpretation: find the largest bipartite graph inside G

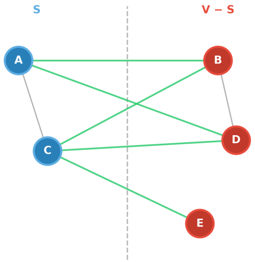


Max Cut

Input: Edge-weighted graph $G = (V, E)$

Output: partition of V into $(S, V \setminus S)$ to *maximize* $w(\delta(S))$

Another interpretation: find the largest bipartite graph inside G



Min Cut is efficiently solvable but Max Cut is NP-Hard

Approximating Max Cut

Question: How well can we approximate Max Cut?

Several simple and easy $\frac{1}{2}$-approximations

- **Random set S:** pick each vertex independently with probability $\frac{1}{2}$
- **Local search:** start with arbitrary S. And keep moving one vertex at a time in or out of S if cut-value improves. Stop when no improvement (local optimum).
- Several LP based algorithms.

Approximating Max Cut

Question: How well can we approximate Max Cut?

Several simple and easy $\frac{1}{2}$-approximations

- **Random set S:** pick each vertex independently with probability $\frac{1}{2}$
- **Local search:** start with arbitrary S. And keep moving one vertex at a time in or out of S if cut-value improves. Stop when no improvement (local optimum).
- Several LP based algorithms.

Is there a better approximation for this simple problem?

Approximating Max Cut

Question: How well can we approximate Max Cut?

Several simple and easy $\frac{1}{2}$-approximations

- **Random set S:** pick each vertex independently with probability $\frac{1}{2}$
- **Local search:** start with arbitrary S. And keep moving one vertex at a time in or out of S if cut-value improves. Stop when no improvement (local optimum).
- Several LP based algorithms.

Is there a better approximation for this simple problem?

Goemans and Williamson 1994: a 0.878-approximation via SDP

First powerful use of SDP in approximation. Ignited many developments.

Quadratic Program for Max-Cut

- For notational ease assume $V = \{1, 2, \ldots, n\}$
- For each $i \in V$, we have a variable $y_i \in \{-1, 1\}$
- $y_i = -1$ implies $i \in S$ and $y_i = 1$ implies $i \in V \setminus S$
- Max Cut is modeled by the following *quadratic* program:

$$\begin{aligned} \max \quad & \sum_{ij \in E} w_{ij}(1 - y_i y_j)/2 \\ \text{s.t.} \quad & y_i \in \{-1, 1\}, \quad i \in V \end{aligned}$$

Instead of writing $y_i \in \{-1, 1\}$ we can write $y_i^2 = 1$

Geometric interpretation

The program

$$\max \sum_{ij \in E} w_{ij}(1 - y_i y_j)/2 \quad \text{s.t.} \quad y_i \cdot y_i = 1, \; i \in V$$

is equivalent to the following *vector program*: interpret y_i as a one-dimensional vector. We have a vector v_i for each $i \in V$:

$$\begin{aligned} \max \quad & \sum_{ij \in E} w_{ij}(1 - v_i \cdot v_j)/2 \\ \text{s.t.} \quad & v_i \cdot v_i = 1 \quad i \in V \\ & v_i \in \mathbb{R}^1 \quad i \in V \end{aligned}$$

Vector Program for Max-Cut: Relaxation

- Solving the 1-d vector program is same as solving Max Cut. NP-Hard
- How do we *relax* to obtain a convex program?
- We relax $v_i \in \mathbb{R}^1$ to $v_i \in \mathbb{R}^n$ (vector in *n*-dimensions)

$$\begin{aligned} \max \quad & \sum_{ij \in E} w_{ij}(1 - v_i \cdot v_j)/2 \\ \text{s.t.} \quad & v_i \cdot v_i = 1 \quad i \in V \\ & v_i \in \mathbb{R}^n \quad i \in V \end{aligned}$$

Vector Program for Max-Cut: Relaxation

- Solving the 1-d vector program is same as solving Max Cut. NP-Hard
- How do we *relax* to obtain a convex program?
- We relax $v_i \in \mathbb{R}^1$ to $v_i \in \mathbb{R}^n$ (vector in *n*-dimensions)

$$
\begin{aligned}
\max \quad & \sum_{ij \in E} w_{ij}(1 - v_i \cdot v_j)/2 \\
\text{s.t.} \quad & v_i \cdot v_i = 1 \quad i \in V \\
& v_i \in \mathbb{R}^n \quad i \in V
\end{aligned}
$$

- Why is this relaxation solvable?
- How good is the relaxation?

Vector Program for Max-Cut: Relaxation

$$
\begin{aligned}
\max \quad & \sum_{ij \in E} w_{ij}(1 - v_i \cdot v_j)/2 \\
\text{s.t.} \quad & v_i \cdot v_i = 1 \quad i \in V \\
& v_i \in \mathbb{R}^n \quad i \in V
\end{aligned}
$$

- Why is this relaxation solvable? Vector program and hence an SDP!
- How good is the relaxation? 0.878-approximation!

SDP for Max-Cut

Vector program:

$$\max \sum_{ij \in E} w_{ij}(1 - v_i \cdot v_j)/2$$

$$\text{s.t.} \quad v_i \cdot v_i = 1 \quad i \in V$$

$$v_i \in \mathbb{R}^n \quad i \in V$$

SDP:

$$\max \sum_{ij \in E} w_{ij}(1 - y_i y_j)/2$$

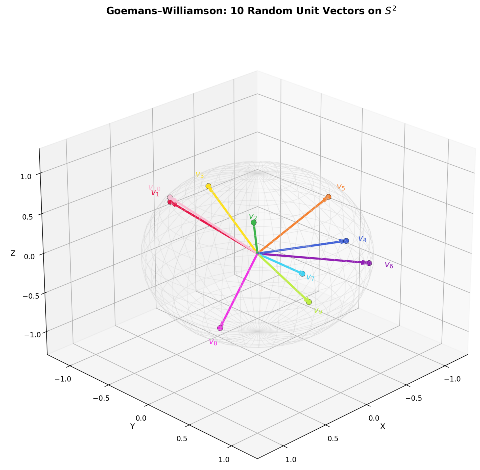
$$\text{s.t.} \quad y_i^2 = 1 \quad i \in V$$

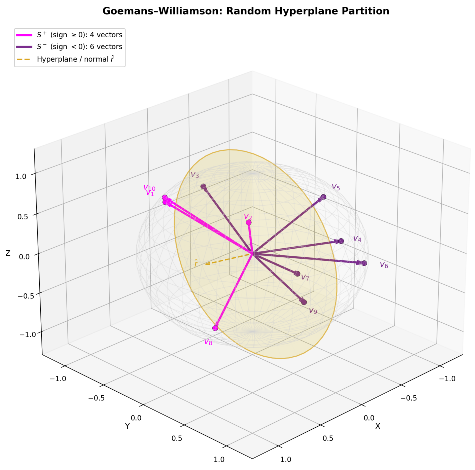
$$Y \succeq 0$$

Rounding: Random Hyperplane Algorithm

- Let OPT_v be an optimum solution value to the vector program. Since it is a relaxation, $\mathrm{OPT}_v \geq \mathrm{OPT}$
- Let the vectors achieving OPT_v be $v_1^*, \ldots, v_n^*$
- Note that each v_i^* is a unit vector in $\mathbb{R}^n$
- How do we produce a cut from the vectors and how do we analyze the quality of the cut?
- The algorithm we use is the **random hyper-plane algorithm**
- Equivalently, pick a random unit vector r
- $S = \{i \mid r \cdot v_i^* > 0\}, \quad V \setminus S = \{i \mid r \cdot v_i^* \leq 0\}$

Rounding: Random Hyperplane Algorithm





Note: pics generated via Claude

Analysis: Probability of Cutting an Edge

- Fix an edge (i, j)
- Let $\theta_{ij} \in [0, \pi]$ be the angle between v_i^* and v_j^*
- Since all vectors are unit vectors, $\cos(\theta_{ij}) = v_i^* \cdot v_j^*$
- Contribution of (i, j) to OPT_v is:

$$w_{ij}(1 - v_i^* \cdot v_j^*)/2 = w_{ij}(1 - \cos(\theta_{ij}))/2$$

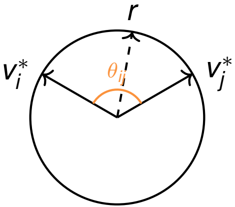
- What is the probability that (i, j) is cut in the algorithm?

Analysis: Probability of Cutting an Edge

- Fix an edge (i, j)
- Let $\theta_{ij} \in [0, \pi]$ be the angle between v_i^* and v_j^*

Lemma

Probability edge (i, j) is cut is θ_{ij}/π.



Analysis: The Goemans–Williamson Bound

- The expected weight of the cut found by the algorithm, by linearity of expectation, is:

$$\sum_{ij \in E} w_{ij}\, \theta_{ij}/\pi$$

- We need to compare this to OPT_v:

$$w_{ij}(1 - v_i^* \cdot v_j^*)/2 = w_{ij}(1 - \cos(\theta_{ij}))/2$$

The Key Inequality Behind 0.878

Recall: For each edge (i, j), let θ_{ij} be the angle between v_i^* and v_j^*.

Algorithm gives:

$$\Pr[\text{edge cut}] = \frac{\theta_{ij}}{\pi}$$

SDP contribution:

$$\frac{1 - \cos\theta_{ij}}{2}$$

We need: find the largest α such that

$$\frac{\theta}{\pi} \geq \alpha \cdot \frac{1 - \cos\theta}{2}$$

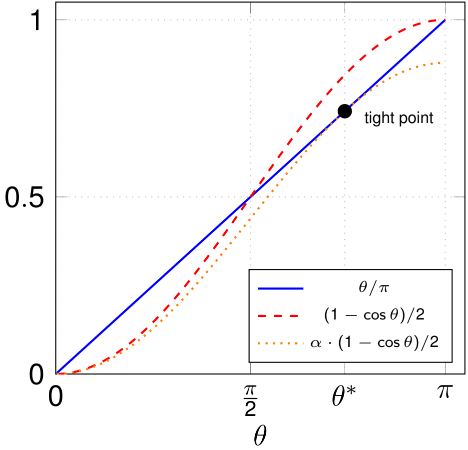
holds for *all* $\theta \in [0, \pi]$.

$$\alpha = \min_{\theta \in (0,\pi]} \frac{2\theta}{\pi(1 - \cos\theta)}$$

Lemma (Goemans–Williamson 1994)

For all $\theta \in [0, \pi]$, $\frac{\theta}{\pi} \geq \alpha \cdot \frac{1-\cos\theta}{2}$ where $\alpha \approx 0.87856$.

Visualizing the 0.878 Constant



Reading the plot:

- Blue: cut probability θ/π.
- Red: SDP contribution $(1 - \cos\theta)/2$.
- Orange: α times SDP contribution

Conclusion:

$$\mathbb{E}[\text{cut}] \geq \alpha \cdot \mathrm{OPT}_v \geq \alpha \cdot \mathrm{OPT}$$

Is expectation good enough?

We obtain a good solution in expectation. What if we are unlucky in the rounding?

- Repeat the rounding several times and take best solution.
- Derandomize the algorithm — technically challenging and not worth it in practice.

Can we do better?

- Is there a better way to round the SDP?
- Can we obtain a better approximation for Max Cut via a different method?

Can we do better?

- Is there a better way to round the SDP?
- Can we obtain a better approximation for Max Cut via a different method?

We know the following:

- The integrality gap of SDP is α
- Under a conjecture called Unique Games Conjecture (UGC), no better worst-case approximation possible for Max Cut

Difficult technical results.

Max-Cut SDP on C_5: Setup (1/2)

Why C_5? Odd cycle, true Max-Cut $= 4$, but $\mathrm{OPT}_v \approx 4.5225$. Gap ≈ 0.8845, close to the 0.878 worst case.

```
import numpy as np
import cvxpy as cp

# C5: pentagon, all weights 1
n = 5
W = np.zeros((n, n))
for i in range(n):
    W[i, (i+1) % n] = 1
    W[(i+1) % n, i] = 1
W_upper = np.triu(W, k=1)

# SDP: X_ij = v_i . v_j
X = cp.Variable((n, n), symmetric=True)
objective = cp.Maximize(cp.sum(cp.multiply(W_upper, (1 - X) / 2)))
constraints = [X >> 0, cp.diag(X) == np.ones(n)]

prob = cp.Problem(objective, constraints)
prob.solve(solver=cp.SCS)
print(f"OPT_v: {prob.value:.4f}")  # ~4.5225
```

Max-Cut SDP on C_5: Rounding (2/2)

```
# Extract vectors via Cholesky: X = L L^T, rows of L^T are v_i
# X = V^T V, so extract vectors v_i as rows of V
w, v = np.linalg.eigh(X.value)
V = v @ np.diag(np.sqrt(np.maximum(w, 0)))

# Random hyperplane rounding
r = np.random.randn(n)
r /= np.linalg.norm(r)
labels = np.sign(V @ r)

# Cut value
cut = np.sum(np.triu(W * (labels[:,None] != labels[None,:]), k=1))
print(f"OPT_v:          {prob.value:.4f}")
print(f"Cut value:      {cut:.4f}")
print(f"Cut / OPT_v:    {cut / prob.value:.4f}")
```

Sample output:

```
OPT_v:        4.5225
Cut value:    4.0000
Cut / OPT_v: 0.8845
```

Takeaway:

- SDP solved in poly time
- One random hyperplane ⇒ cut
- Ratio ≥ 0.878 in expectation