CS 270 Combinatorial Algorithms and Data Structures, Spring 2015

Lecture 18 : Semidefinite programming

We introduce semidefinite programming and see some applications in designing exact and approximation algorithms.

Semidefinite programs

A linear program can be written as

$$\max \ \langle c, x\rangle$$
$$\text{subject to} \quad Ax = b$$
$$x \geq 0 \ ,$$

where $A \in \mathbb{R}^{m\times n}$, $b \in \mathbb{R}^m$. $c, x \in \mathbb{R}^n$.

A semidefinite program can be written in a similar form:

$$\max \quad C \cdot X$$
$$\text{subject to} \quad A_i \cdot X = b_i \quad \forall\ 1 \leq i \leq m$$
$$X \succeq 0 ,$$

where $X \in SYM_n = \{ X \in \mathbb{R}^{n\times n} : x_{ij} = x_{ji}, \ 1 \leq i \leq j \leq n \}$, $A_i \in SYM_n$ for all i, and $C \in SYM_n$.

For two matrices $X, Y \in SYM_n$, $X \cdot Y$ is defined as $\sum_{i,j} X_{ij} \cdot Y_{ij}$. Equivalently, it can be written as $Tr(X^T Y)$ where $Tr(M)$ is the sum of the diagonal entries of M.

The constraint $X \succeq 0$ requires that the matrix X is positive semidefinite, and it is the difference between linear programs and semidefinite programs.

<u>Fact</u> Let $M \in SYM_n$. The following statements are equivalent.

① M is positive semidefinite, i.e. all the eigenvalues of M are non-negative.

② $x^T M x \geq 0$ for all $x \in \mathbb{R}^n$.

③ $M = U^T U$ for some matrix $U \in \mathbb{R}^{n\times n}$

<u>Proof</u> In the proof, we use the fact that a real symmetric matrix has n real eigenvalues $\lambda_1, \dots, \lambda_n$ and a set of orthonormal eigenvectors $v_1, \dots, v_n$ such that $Mv_i = \lambda_i v_i$.

By the fact, $M = VDV^T = \begin{pmatrix} | & | & & | \\ v_1 & v_2 & \cdots & v_n \\ | & | & & | \end{pmatrix}\begin{pmatrix} \lambda_1 & & & \\ & \lambda_2 & & \\ & & \ddots & \\ & & & \lambda_n \end{pmatrix}\begin{pmatrix} — v_1 — \\ — v_2 — \\ \vdots \\ — v_n — \end{pmatrix}$. Note that since v_i are orthonormal, we have $V^T V = I$ and thus $V^T = V^{-1}$, and so $M = VDV^{-1}$, which can be interpreted as switching to the eigen-basis, multiplying by the eigenvalues, and switching back to the standard basis.

① ⇒ ③ Since all eigenvalues are non-negative, we can write $M = VD^{\frac{1}{2}}D^{\frac{1}{2}}V^T$,

thus letting $U = VD^{\frac{1}{2}}$ we get ③.

③ ⇒ ② $x^T M x = x^T U U^T x = \|U^T x\|_2^2 \geq 0$

② ⇒ ① We prove the contrapositive, $\neg$① ⇒ $\neg$②.

If λ_i is negative, then $v_i^T M v_i = -\lambda_i \|v_i\|_2^2 < 0$. □

Condition ② can be interpreted as an infinite number of linear constraints of the form $x^T M x \geq 0$,

one for each $x \in \mathbb{R}^n$, while the variables are the entries of M.

We will see this interpretation more clearly when we talk about Lovász's result later.

Vector programs

Condition ③ says that a matrix $X \succcurlyeq 0$ if and only if it can be written as $X = U^T U$.

This means that there are n-dimensional vectors $u_1, u_2, \ldots, u_n$ such that $X_{i,j} = \langle u_i, u_j \rangle$.

So, we can think of a semidefinite program as a "vector program", in which there are linear constraints and linear objective function of inner products of vectors.

This is a very useful view when we write semidefinite programming relaxations for combinatorial problems.

Cholesky factorization

Because of its connection to vector programming, we usually need to compute the Cholesky factorization of $M \succcurlyeq 0$, i.e. write M as $U^T U$.

We don't need to compute all the eigenvalues and eigenvectors as in the above proof, instead we can compute it by symmetric Gaussian elimination.

Let $M = \begin{pmatrix} \alpha & q^T \\ q & N \end{pmatrix}$.

First, note that $\alpha \geq 0$, as $\alpha = e_1^T M e_1 \geq 0$ by the fact that M is positive semidefinite.

We consider two cases.

The first case is when $\alpha > 0$. Then we can do symmetric Gaussian elimination and write M as follows.

We can get $\begin{pmatrix} \alpha & 0 \\ 0 & N - \frac{qq^T}{\alpha} \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ -\frac{q}{\alpha} & I_{n-1} \end{pmatrix} \begin{pmatrix} \alpha & q^T \\ q & N \end{pmatrix} \begin{pmatrix} 1 & -\frac{q^T}{\alpha} \\ 0 & I_{n-1} \end{pmatrix}$ assuming $\alpha > 0$

$\Rightarrow M = \begin{pmatrix} \alpha & q^T \\ \end{pmatrix} = \begin{pmatrix} \sqrt{\alpha} & 0 \\ \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & N - qq^T \end{pmatrix} \begin{pmatrix} \sqrt{\alpha} & q^T/\sqrt{\alpha} \\ \end{pmatrix}$

[illegible]

$$\Rightarrow M = \begin{pmatrix} \alpha & g^T \\ g & N \end{pmatrix} = \begin{pmatrix} \sqrt{\alpha} & 0 \\ \frac{g}{\sqrt{\alpha}} & I_{n-1} \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & N - \frac{gg^T}{\alpha} \end{pmatrix} \begin{pmatrix} \sqrt{\alpha} & g^T/\sqrt{\alpha} \\ 0 & I_{n-1} \end{pmatrix}.$$

Note also that $N - \frac{1}{\alpha} gg^T \succcurlyeq 0$, because $y^T \begin{pmatrix} 1 & 0 \\ 0 & N - \frac{gg^T}{\alpha} \end{pmatrix} y = (y^T Z^T) M (Zy) \geq 0$, where $Z = \begin{pmatrix} \sqrt{\alpha} & g^T/\sqrt{\alpha} \\ 0 & I_{n-1} \end{pmatrix}^{-1}$.

By induction, $N - \frac{1}{\alpha} gg^T$ can be written as $V^T V$ for some V, and thus M can be written as $U^T U$,

where $U = \begin{pmatrix} 1 & 0 \\ 0 & V \end{pmatrix} \begin{pmatrix} \sqrt{\alpha} & g^T/\sqrt{\alpha} \\ 0 & I_{n-1} \end{pmatrix} = \begin{pmatrix} \sqrt{\alpha} & g^T/\sqrt{\alpha} \\ 0 & V \end{pmatrix}$. and we are done.

The second case is $\alpha = 0$. In this case, we claim that g must be equal to 0, and so $M = \begin{pmatrix} 0 & 0 \\ 0 & N \end{pmatrix}$,

and then we have $N \succcurlyeq 0$ and we can finish by induction as above.

To see that $g = 0$, assume $g \neq 0$, let $x = \begin{pmatrix} x_1 \\ x' \end{pmatrix}$ where $x' \in \mathbb{R}^{n-1}$, then $x^T \begin{pmatrix} 0 & g^T \\ g & N \end{pmatrix} x = 2x_1 \langle x', g \rangle + x'^T N x'$.

Setting $x_1 = +\infty$ or $-\infty$ will get $x^T M x < 0$, contradicting that M is positive semidefinite.

This shows that a Cholesky factorization can be computed in $O(n^3)$ time, assuming each operation can be done in constant time.

But some subtle issues are that there are numerical errors and the intermediate numbers could be very large. This can be handled by choosing α to be the largest diagonal entry.

If M is not positive semidefinite, the above method can be used to find a vector x such that $x^T M x < 0$.

The above method fails to find a Cholesky factorization in only two scenarios: ① $\alpha < 0$ in some intermediate step, or ② $\alpha = 0$ but $g \neq 0$ in some step. In either case, we can find v such that $v^T N v < 0$, and this can be used to find a w such that $w^T M w < 0$ by un-doing the Gaussian elimination.

<u>Solving SDP</u>

Under some fairly general conditions, there are polynomial time algorithms to return arbitrarily close approximation to the optimal solution.

The first observation is that the set of feasible solution is a convex set, as if $X_1 \succcurlyeq 0$ and $X_2 \succcurlyeq 0$ then $\frac{1}{2}(X_1 + X_2) \succcurlyeq 0$ as well.

One can use the ellipsoid algorithm to solve SDP.

To implement a polynomial time separation oracle, one can use a Cholesky factorization algorithm to check whether $X \succcurlyeq 0$.

There are many technical issues. One is numerical error, as the optimal solution can be irrational, and we can only hope for an arbitrarily good approximation (including the Cholesky factorization step).

Another issue is the bound on the initial ellipsoid. There are examples where all the coefficients are small but the solution is doubly-exponentially large (2^{2^n}).

For our purpose, we ignore these issues as for combinatorial problems the set of feasible solution is easily bounded and it is okay to have a good approximation only (e.g. an $(1+\frac{1}{n})$-approximation suffices).

Lovász Theta function

The first application of semidefinite programming in combinatorial optimization was discovered by Lovász.

(It may be surprising to those who know SDP for its applications in designing approximation algorithms.)

His original idea was to map independent sets of vertices to orthonormal sets of vectors, but our presentation in the following is slightly different.

As we already saw, the natural LP for independent sets is not useful.

So, we turn to quadratic programming, and then relax it to an SDP using the viewpoint of vector programming.

It is easy to write the maximum independent set problem as a quadratic program.

$$\max \sum_i x_i^2$$

$$x_i x_j = 0 \quad \forall ij \in E$$

$$x_i^2 = x_i \quad \forall i \in V.$$

The constraint $x_i^2 = x_i$ is to make sure that $x_i \in \{0,1\}$, and the constraint $x_i x_j = 0$ is to make sure that we can't pick both endpoints of an edge.

Recall that SDP can be written as a vector program, and so the idea is to relax this quadratic program to an SDP by replacing each variable $x_i \in \mathbb{R}$ by a vector $v_i \in \mathbb{R}^n$.

There is a little technicality here, as in a vector program we are allowed to have linear constraints on inner products of vectors, but in the above program there is some term x_i that cannot be directly modelet.

So, to relax it to SDP, we do the following little trick to make every term quadratic by introducing a variable.

$$\max \sum_{i \in V} x_i^2$$

$$x_i x_j = 0 \quad \forall ij \in E$$

$$x_i^2 = x_i x_0 \quad \forall i \in V$$

$x_0^2 = 1$

We can relax it to a vector program as follows, which corresponds to an SDP:

$$\begin{aligned} \max \ & \sum_{i\in V} \langle v_i, v_i\rangle \\ & \langle v_i, v_j\rangle = 0 \quad \forall ij\in E \\ & \langle v_i, v_i\rangle = \langle v_i, v_0\rangle \quad \forall i\in V \\ & \langle v_0, v_0\rangle = 1 \end{aligned} \quad \Leftrightarrow \quad \begin{aligned} \max \ & \sum_{i\in V} Y_{i,i} \\ & Y_{ij} = 0 \quad \forall ij\in E \\ & Y_{i,i} = Y_{i,0} \quad \forall i\in V \\ & Y_{0,0} = 1 \\ & Y \succeq 0 \end{aligned}$$

We can think of the SDP as mapping each vertex to a vector, where the vectors of two adjacent vertices are orthogonal.

Before we see the power of this SDP, let us first see the context of the problem.

Perfect graphs

The independence number $\alpha(G)$ of a graph is a fundamental parameter in graph theory and combinatorial optimization.

Unfortunately, it is usually very difficult to bound this number (as we now know, approximating $\alpha(G)$ to a factor of $n^{1-\varepsilon}$ for any $\varepsilon > 0$ is NP-hard, so it is hopeless in general).

One natural upper bound on $\alpha(G)$ is the minimum clique cover number, denoted by $\bar{\chi}(G)$, which is the minimum k such that the vertex set V can be partitioned into k disjoint subsets $V_1, \ldots, V_k$ such that each V_i is a clique, where a clique $C \subseteq V$ is a subset of vertices in which there is an edge xy for any pair of vertices $x, y \in C$.

Note that if $\bar{\chi}(G) = k$, then $\alpha(G) \le k$ because each independent set can take at most one vertex from each clique.

It is of interest to characterize those graphs with $\alpha(G) = \bar{\chi}(G)$, so that the natural upper bound is tight.

However, having such a restriction is not very useful in forcing good structures in the graph, because such a graph can have any structure as long as there is also a large enough independent set.

Therefore, we also restrict the equality to hold for any induced subgraph of G.

Definition A graph G is perfect if $\alpha(H) = \bar{\chi}(H)$ for any induced subgraph H of G.

There are many graph classes that are perfect, e.g. interval graphs, chordal graphs (see [5] for their definitions and other classes).

The above definition is not the usual one. To state the usual definition, we introduce two more parameters.

Let $\omega(G)$ be the size of a maximum clique of the graph, and $\chi(G)$ be the chromatic number, which is defined as the minimum k such that the vertex set V can be partitioned into k disjoint subsets $V_1, \dots, V_k$ such that each is an independent set.

To see the connection with $\alpha(G)$ and $\bar{\chi}(G)$, let $\bar{G}$ be the complement of G, then $\alpha(G) = \omega(\bar{G})$ and $\bar{\chi}(G) = \chi(\bar{G})$.

Definition A graph G is perfect if $\omega(H) = \chi(H)$ for any induced subgraph H of G.

Indeed, the two definitions are equivalent, as shown by the following theorem of Lovász.

Weak perfect graph theorem A graph G is perfect if and only if its complement $\bar{G}$ is perfect.

There is a much stronger theorem characterizing perfect graphs proved by Chudnovsky et.al. in around 2006.

Strong perfect graph theorem A graph is perfect if and only if it does not contain an odd cycle and it does not contain the complement of an odd cycle as an induced subgraph.

One direction is trivial, while the other direction took more than 150 pages, and is considered a major achievement in graph theory.

The sandwich theorem

Let $\theta(G)$ be the optimal value of the SDP.

Lovász showed that $\alpha(G) \le \theta(G) \le \bar{\chi}(G)$, and this implies that the independence number of a perfect graph can be computed in polynomial time, and this is still the only known polynomial time algorithm.

One direction is easy, as $\theta(G)$ is a relaxation of independence number, and so $\theta(G) \geq \alpha(G)$.

To prove the other direction, it suffices to show that $\sum_{i \in C} \|v_i\|^2 \le 1$, as this would imply that if the the vertex set of the graph can be partitioned into k cliques, then $\theta(G) = \sum_{i \in V} \|v_i\|^2 \le k$.

For a clique C, $\sum_{i \in C} \langle v_i, v_i \rangle = \sum_{i \in C} \langle v_i, v_0 \rangle$ by the constraints.

Since C is a clique, the vectors corresponding to vertices in C are orthogonal.

By writing v_0 as a linear combination of the orthonormal vectors $\{ \frac{v_i}{\|v_i\|} \mid i \in C \}$ and it orthonormal complement, such that $v_0 = \sum_{i \in C} a_i \frac{v_i}{\|v_i\|} + \sum_{j \notin C} a_j u_j$ (where we extend $\{ \frac{v_i}{\|v_i\|} \mid i \in C \}$ to a basis using u_j), then

$a_i = \langle v_0, \frac{v_i}{\|v_i\|} \rangle$ for $i \in C$ and we have

$$1 \underset{\substack{\uparrow \\ \text{constraint}}}{=} \langle v_0, v_0 \rangle \underset{\substack{\uparrow \\ \text{orthonormal vectors}}}{\geq} \sum_{i \in C} a_i^2 \underset{\substack{\nwarrow \\ a_i}}{=} \sum_{i \in C} \langle v_0, \frac{v_i}{\|v_i\|} \rangle^2 = \sum_{i \in C} \frac{1}{\|v_i\|^2} \langle v_i, v_i \rangle^2 \underset{\substack{\uparrow \\ \text{constraint}}}{=} \sum_{i \in C} \|v_i\|^2 ,$$

and thus we have the clique constraints in the SDP.

To find a maximum independent set, we can compute $\vartheta(G-v)$ for a vertex v. If $\vartheta(G-v) = \vartheta(G)$, then we can remove v from the graph, as $\alpha(G-v) = \vartheta(G-v) = \vartheta(G)$ as $G-v$ is also perfect. Otherwise, we add v to our independent set. And we repeat the above procedure.

Open question: The above algorithm for finding a maximum independent set is quite dumb and didn't look at the vector solution at all. Is there a smarter "rounding" algorithm for finding a maximum independent set? Even more concretely, is there a rounding algorithm for finding a maximum bipartite matching (which is a special case of the perfect graph result) using SDP?

Maximum cut

A breakthrough result in approximation algorithm is an 0.878- approximation algorithm for maxcut by Goemans and Williamson, who introduced the tool of semidefinite programming in this area.

First, the maxcut problem is formulated as a quadratic program.

$$\max \sum_{ij \in E} \frac{1}{2}(1 - x_i x_j)$$
$$x_i^2 = 1. \qquad \forall i \in V$$

The constraint $x_i^2 = 1$ forces $x_i \in \{+1, -1\}$ and the objective is counting number of edges with different sig

So, this is an exact formulation for maxcut, thus solving quadratic program is NP-hard.

As before, we would like to relax the problem to a vector program so that it is polynomial time solvable.

$$\begin{aligned} &\max \sum_{ij \in E} \frac{1}{2}(1 - \langle y_i, y_j \rangle) \\ &\|y_i\|^2 = 1 \quad \forall v \in V \\ &y_i \in \mathbb{R}^n \end{aligned} \qquad \Leftrightarrow \qquad \begin{aligned} &\max \sum_{ij \in E} \frac{1}{2}(1 - Y_{ij}) \\ &Y_{ii} = 1 \quad \forall i \in V \\ &Y \succcurlyeq 0 \end{aligned}$$

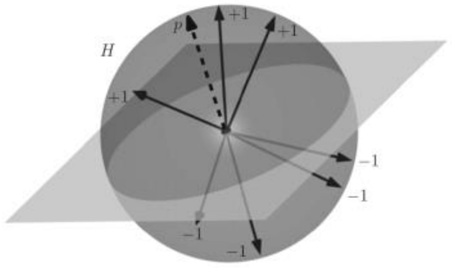
It is clear that it is a relaxation, since a ± 1 solution can be embedded as $(\pm 1, 0, \ldots, 0)$ in the vector solution, and so $\text{SDP} \geq \text{OPT}$.

Now, we want to analyze the performance of this SDP.

To get some intuition about this SDP, we can think of the solution geometrically.

An edge contributes a large value close to 1 to the objective has $\langle y_i, y_j \rangle$ close to -1, meaning that the vectors y_i, y_j are almost opposite to each other.

An optimal SDP solution tries to pull the two vectors of an edge as far as possible.



(picture from Gärtner - Matoušek)

To produce a 0.878-approximate solution, we would like to find an integral solution with $SOL \geq 0.878\ SDP$.

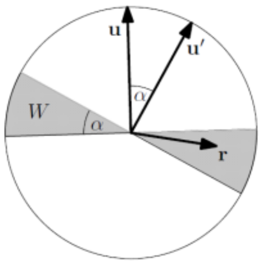
Intuitively, we should use the vector solution as a guideline: we should cut an edge with a probability proportional to $1 - \langle y_i, y_j \rangle = 1 - \cos\theta_{ij}$, where θ_{ij} is the angle between y_i and y_j. If θ_{ij} is larger (closer to π), we should cut it with higher probability.

To achieve this, Goemans and Williamson used a random hyperplane to cut the vectors.

Let p be a random unit vector in $\mathbb{R}^n$.

The algorithm is simple: just set $S = \{ i \mid \langle p, y_i \rangle \geq 0 \}$. See the above picture for an illustration

To analyze its performance, we see that an edge ij is cut with probability θ_{ij}/π.



(picture from Gärtner_Matoušek)

Therefore, $E[\#\text{ edges cut}] = \sum_{uv \in E} Pr(\text{edge } ij \text{ is cut}) = \sum_{ij \in E} \theta_{ij}/\pi$.

Let's compare it to the performance of the SDP solution.

$$\frac{SOL}{OPT} \geq \frac{SOL}{SDP} = \frac{\sum_{ij \in E} \theta_{ij}/\pi}{\sum_{ij \in E} \frac{1}{2}(1-\cos\theta_{ij})} \geq \min_{\theta} \frac{\theta/\pi}{\frac{1}{2}(1-\cos\theta)} \geq 0.8785672,$$

where the last inequality is obtained by plotting the curve of the function...

This completes the proof.

The algorithm performs very well in practice, producing almost optimal solution in most instances.

Graph coloring

Since Goemans-Williamson, there are many SDP-based approximation algorithms, using the random hyperplane rounding method (and its variants). Here we see one more well-known example.

In general, finding a minimum coloring of a graph is extremely difficult to approximate, even an $O(n^{1-\varepsilon})$-approximation algorithm for some $\varepsilon > 0$ would imply P = NP.

Here, we consider the special case of using as few colors as possible to color a 3-colorable graph in polynomial time, and for this problem there is a large gap between the known upper and lower bounds.

Using $O(\sqrt{n})$ colors: First, we see a combinatorial algorithm to use $O(\sqrt{n})$ colors to color a 3-colorable graph.

If the maximum degree is $\sqrt{n}$, then a simple greedy algorithm would do.

Otherwise, if there exists a vertex v with degree $> \sqrt{n}$, then its neighbor set must be 2-colorable.

We know how to color a 2-colorable graph (a bipartite graph) in 2 colors, so we use 3 (new) colors to color v and $N(v)$ and remove them from the graph.

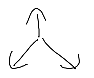
Note that each time we use 3 colors to remove $\sqrt{n}$ nodes, and thus we use at most $3\sqrt{n}$ colors in this reduction step. After that the graph is of maximum degree $\leq \sqrt{n}$, and we are done.

Now, we design a better approximation algorithm using SDP. Consider the following vector relaxation.

$$\langle v_i, v_j \rangle = -\tfrac{1}{2} \qquad \forall\, ij \in E$$

$$\langle v_i, v_i \rangle = 1 \qquad \forall\, i \in V$$

$$v_i \in \mathbb{R}^n$$

It is easy to see that if the graph is 3-colorable, then the SDP is feasible .

Therefore, we assume that there is a feasible solution to the SDP on the right, and we would use the vector solution to find a coloring using fewer (than $O(\sqrt{n})$) colors.

The idea is quite natural. If the graph has such a geometric representation, then a random hyperplane would cut many edges (since each edge has a degree 120°). Let's say we cut the graph into two equal halves. Then, since we cut many edges (significantly more than half), each half is considerably sparser than before. So, if we recursively apply the same idea, then we get sparser and sparser (but smaller and smaller) induced subgraph. and eventually we get a (somewhat) large independent set and use only one color. The formal algorithm is slightly different.

Algorithm: The algorithm would proceed in at most $O(\log n)$ iterations.

In each iteration, we will use C colors to color at least half the vertices, so repeating this will color the graph using $O(C \log n)$ colors.

In each iteration, we will use t random hyperplanes $r_1, r_2, \ldots, r_t$ to divide the vectors into 2^t groups (two vectors v_i and v_j are in the same group if $\operatorname{sign}(r_l, v_i) = \operatorname{sign}(r_l, v_j)$ for $1 \le l \le t$), and we assign one color to all vertices in the same group.

To make sure it is a proper coloring, we uncolor all vertices who have the same color as one of its neighbors.

Analysis:

To prove that there are still at least $n/2$ vertices with colors, we will prove that at most $n/4$ edges are "bad".

What is the probability that an edge is bad (meaning the two endpoints of an edge end up in the same group)?

This happens when all t hyperplanes fail to separate v_i and v_j, and this probability is

$$\left(1 - \frac{\theta_{ij}}{\pi}\right)^t = \left(\frac{1}{3}\right)^t \qquad \text{since } \theta_{ij} = \frac{2\pi}{3} \text{ as } \cos\theta_{ij} = \langle v_i, v_j\rangle = -\frac{1}{2}.$$

So, the expected number of bad edges is at most $m\left(\frac{1}{3}\right)^t$.

By Markov inequality ($\Pr(X \geq N) \leq \frac{E[X]}{N}$), the probability of having $2m\left(\frac{1}{3}\right)^t$ bad edges is at most $\frac{1}{2}$.

Suppose the maximum degree of the graph is Δ, then $m \leq \frac{n\Delta}{2}$.

By setting $t = \log_3 4\Delta$, then the number of bad edges is $\leq 2m\left(\frac{1}{3}\right)^t = n\Delta\left(\frac{1}{3}\right)^t = \frac{n}{4}$ with prob. $\geq \frac{1}{2}$.

Repeating a few times will find such a coloring with high probability.

The number of colors used in each iteration is $2^t = 2^{\log_3 4\Delta} = 4\Delta^{\log_3 2} \approx 4\Delta^{0.632}$, totally $\leq 4\Delta^{0.632} \log n$.

Finally, consider a threshold δ. If max degree $\geq \delta$, then use 3 colors to color it and its neighbors; otherwise, run the SDP algorithm.

The total number of colors used is $\frac{3n}{\delta} + 4\delta^{0.632} \log n$.

Setting $\delta = n^{0.612}$, we color the graph using $O(n^{0.388})$ colors.

Open question: It is a long standing open problem whether there is a polynomial time algorithm to color a 3-colorable graph using say polylog(n) colors. The best known algorithm uses about

$O(n^{0.2})$ colors and is complicated.

References

Most materials are from the book "approximation algorithms and semidefinite programming" by Gärtner and Matoušek, including the introduction (chapter 2), Lovász Theta function (chapter 3, different presentation), maximum cut (chapter 1), and graph coloring (chapter 9, stronger result).

The presentation of the graph coloring algorithm follows that in the book "the design of approximation algorithms" by Williamson and Shmoys (chapter 6.5).

You are referred to these two books if you are interested in learning more about SDP-based approx algos.

A very important result in this direction is an $O(\sqrt{\log n})$-approximation algorithm for sparsest cut by Arora, Rao, and Vazirani. This result raises the analysis of SDP approximation to another level.