CSE525: Randomized Algorithms and Probabilistic Analysis

Lecture 9

Lecturer: Anna Karlin *Scribe: Sonya Alexandrova and Keith Jia*

1 Introduction to semidefinite programming

Semidefinite programming is linear programming where variables are entries in a positive semidefinite matrix.

Summary In semidefinite programming, one minimizes a linear function subject to the constraint that an affine combination of symmetric matrices is positive semidefinite. Such a constraint is nonlinear and nonsmooth, but convex, so semidefinite programs are convex optimization problems. Semidefinite programming unifies several standard problems (e.g., linear and quadratic programming) and finds many applications in engineering and combinatorial optimization.

Why do we care Semidefinite programming is a relatively new field of optimization which is of growing interest for several reasons. Many practical problems in operations research and combinatorial optimization can be modeled or approximated as semidefinite programming problems. In automatic control theory, SDP's are used in the context of linear matrix inequalities. SDPs are in fact a special case of cone programming and can be efficiently solved by interior point methods. All linear programs can be expressed as SDPs, and via hierarchies of SDPs the solutions of polynomial optimization problems can be approximated. Semidefinite programming has been used in the optimization of complex systems. In recent years, some quantum query complexity problems have been formulated in terms of semidefinite programs.

1.1 Semidefinite matrix

If X is a symmetric $(n \times n)$ matrix, then the following statements are equivalent:

1. X is positive semidefinite (psd) (denoted by $X \succcurlyeq 0$)
2. $\forall y \in \mathbb{R}^n \; y^\intercal X y \geq 0$
3. X has nonnegative eigenvalues
4. $X = V^\intercal V$ for some $m \times n$ matrix V $(m \leq n)$
5. $X = \sum_{i=1}^{n} \lambda_i w_i w_i^\intercal$ for some $\lambda_i \geq 0$ and orthonormal vectors $w_i \in \mathbb{R}^n$

1.2 Semidefinite program (SDP)

$$\max \text{ or } \min \sum_{i,j} c_{ij} x_{ij}$$
$$\text{subject to} \sum_{i,j} a_{ijk} x_{ij} = b_k \; \forall k$$
$$x_{ij} = x_{ji} \; \forall i, j$$
$$X = (x_{ij}) \succcurlyeq 0$$

Since X is positive semidefinite, $X = V^\intercal V$. If we set v_i to be the i^{th} column of V, we obtain the following vector program that is equivalent to the SDP:

$$\max \text{ or } \min \sum_{i,j} c_{ij}(v_i \cdot v_j)$$
$$\text{subject to} \sum_{i,j} a_{ijk}(v_i \cdot v_j) = b_k \ \forall k$$
$$v_i \in \mathbb{R}^n \ i = 1..n$$

Fact 1. *SDPs can be solved to within additive error ε in time poly(size of input, $\log \frac{1}{\varepsilon}$).*

In our discussions, we ignore the additive error ε, and just assume that SDPs can be solved efficiently for all practical purposes.

2 MAXCUT

The problem of finding maximum cut is formulated as follows: Given a graph $G = (V, E)$ and a set of weights w_{ij} $\forall (i, j) \in E$, partition the vertex set into sets (S_1, S_2), so as to maximize the weigth of edges between S_1 and S_2.

The first approximation algorithm based on an SDP is due to Goemans and Williamson [1], general steps:
1. Relax the integer quadratic program into an SDP.
2. Solve the SDP (to within an arbitrarily small additive error epsilon), which in this class we ignore
3. Round the SDP solution to obtain an approximate solution to the original integer quadratic program.

2.1 IP formulation of MAXCUT

$$\forall i \in V \ \ x_i = \begin{cases} 1 & if\ i \in S_1 \\ 0 & otherwise \end{cases} \tag{1}$$

$$z_{ij} = \begin{cases} 1 & \text{edge (i,j) crosses cut} \\ 0 & otherwise \end{cases} \tag{2}$$

$$max \sum_{(i,j)\in E} w_{ij} z_{ij} \tag{3}$$

$$z_{ij} \leq x_i + x_j \ \forall (i, j) \in E \tag{4}$$

$$z_{ij} \leq 2 - (x_i + x_j) \ \forall (i, j) \in E \tag{5}$$

$$x_i \in \{0, 1\} \ \forall i \in V \tag{6}$$

$$z_{ij} \in \{0, 1\} \ \forall (i, j) \in E \tag{7}$$

Observation One observation is that on any triangle, the number of edges in the cut can be at most 2. Another observation is triangle inequality applies to vertices on the graph. Hence we have 4 and 5.

The constraints on z_{ij} ensure that we only set z_{ij} to 1 if x_i and x_j are on different sides of the cut. Line 5 states that if x_i and x_j are both 0, $z_{ij} = 0$, and line 6 states that if x_i and x_j are both 1, $z_{ij} = 0$. The problem with this formulation is that if we relax it, obtaining a linear program, the LP optimal solution will always be equal to the sum of all edge weights in the graph. The integrality gap of this LP is still 1/2. So, let's reformulate the program so that its optimal solution is the optimal solution of MAXCUT.

2.2 Quadratic Programming Formulation

Since LP relaxations cant get any better than greedy algorithm, we need to look for some other solution. First lets change the indicator function on variables to:

$$\forall y \in V \;\; y_i = \begin{cases} -1 & if\ i \in S_1 \\ 1 & otherwise \end{cases}$$

Obervation If $y(i) \neq y(j)$, $y(i)y(j) = -1$. Using this observation, we can write an Integer Quadratic Programming formulation:

$$max \frac{1}{2} \sum_{(i,j)\in E} w_{ij}[1 - y_i y_j]$$
$$y_i \cdot y_i = 1$$
$$\forall i \in V$$

Here $y_i = 1$ if i is on one side of the cut, and $y_i = -1$ if i is on the other side of the cut. Let's denote by OPT the optimal solution for this program. OPT is exactly equal to the optimal solution to MAXCUT. However, it's NP-hard. In order to obtain a relaxation, we will allow the variables v to be in a higher dimension space.

2.3 Vector Programming Relaxation

Observation We broaden the constraint space to $\mathbb{R}^n$

$$max \frac{1}{2} \sum_{(i,j)\in E} w_{ij}(1 - \overrightarrow{v_i} \cdot \overrightarrow{v_j})$$
$$\overrightarrow{v_i} \cdot \overrightarrow{v_j} = 1 \; \forall i \in V$$
$$\overrightarrow{v_i} \in \mathbb{R}^n$$

Let's denote by Z^* the optimal solution to the vector program.

Claim 2. $Z^* \geq OPT$

Proof To prove it, we just need to show that the vector program is indeed a relaxation for the quadratic program. If we set vectors v_i to $(\pm 1, 0, ..., 0)$, we will get the quadratic program from the vector program. Thus, the space of possible solutions of the quadratic program is a subset of all the possible solutions of the vector program, so the optimal solution to the former is not greater than the optimal solution to the latter. ■

A vector program is equivalent to a SDP program, and we know how to solve these for all practical purposes. But now we need to round the SDP solution. The intuition behind rounding is that we're maximizing $\sum_{(i,j)\in E} w_{ij}(1 - \overrightarrow{v_i} \cdot \overrightarrow{v_j})$, and $1 - \overrightarrow{v_i} \cdot \overrightarrow{v_j}$ is maximized when $\overrightarrow{v_i}$ and $\overrightarrow{v_j}$ point in opposite directions (since $\overrightarrow{v_i}$ and $\overrightarrow{v_j}$ are unit vectors, $\overrightarrow{v_i} \cdot \overrightarrow{v_j} = cos(i, j)$). So, the larger the angle between them, the better.

Fact Ellipsoid solves LPs in polytime if there's a separation oracle. A separation oracle checks in polytime whether proposed solution P satisfies all constraints or else produces a constraint that is violated.
1) Given P, check symmetric
2) Compute eigenvalues
3) If non-negative, done
4) Else eigenvector x with negativie eigen-value is violating constraint

2.4 Random Hyperplane Rounding

Algorithm 1 SPD rounding for MAXCUT

1: Solve SDP, obtain solution: $\overrightarrow{v_1^*}, ..., \overrightarrow{v_n^*}$
2: Pick random hyperplane that passes through origin.
3: Partition vertices based on which side of the hyperplane the corresponding vectors are.

The intuition behind this is that vectors that have a large angle between them are more likely to be partitioned by a random hyperplane.

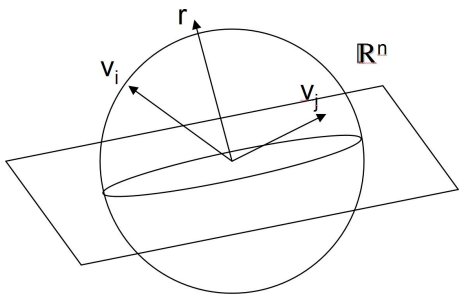

Figure 1: Random hyperplane rounding

Choosing a random hyperplane is equivalent to choosing a random normal to it - so we just need to choose a random unit vector. How to construct a random unit vector? Consider a vector $\overrightarrow{g} = (g_1, ..., g_n), g_i \sim N(0, 1)$, where each element is picked uniformly at random from the normal distribution. If we normalize it: $\overrightarrow{r} = \frac{\overrightarrow{g}}{\|\overrightarrow{g}\|}$, we obtain $\overrightarrow{r}$ which is a random unit vector. In fact, it represents the uniform distribution on a unit sphere.

Fact 3. *A collection of n random Gaussian variables is a spherically symmetric distribution. The probability density function is* $f(g_1, ..., g_n) = \prod_{i=1}^{n} \frac{1}{\sqrt{(2\pi)}} e^{-\frac{g_i^2}{2}} = \frac{1}{(2\pi)^{\frac{n}{2}}} e^{-\frac{\sum g_i^2}{2}}$. *Density of a particular point only depends on the length of the vector, hence after normalization, r will be a uniformly random vector over the unit sphere.*

Now the rounding algorithm looks like this: put i in S_1 if $\overrightarrow{v_i} \cdot \overrightarrow{r} \geq 0$, put i in S_2 if $\overrightarrow{v_i} \cdot \overrightarrow{r} < 0$.
To compute the expected value of the cut, we need to compute the probability that an edge is cut: $Pr((i, j)$ gets cut$)$. Consider a 2D plane containing vectors $\overrightarrow{v_i}$, $\overrightarrow{v_j}$. Since the hyperplane was randomly selected from a rotationally symmetric distribution, the probability that the hyperplane cuts those two vectors is equal to the probability that a random diameter lies in between them. Another way to see that is to write $\overrightarrow{r} = \overrightarrow{r}' + \overrightarrow{r}''$, where $\overrightarrow{r}'$ is the component of $\overrightarrow{r}$ that lies in the 2D plane and $\overrightarrow{r}''$ is orthogonal to the plane. Then, $\overrightarrow{v_i} \cdot \overrightarrow{r}'' = \overrightarrow{v_j} \cdot \overrightarrow{r}'' = 0$, and $\frac{\overrightarrow{r}'}{\|\overrightarrow{r}'\|}$ is uniformly distributed on the unit circle.
Thus, $Pr((i, j)$ gets cut$) = \frac{2\theta}{2\pi} = \frac{\theta}{\pi} = \frac{arccos(\overrightarrow{v_i} \cdot \overrightarrow{v_j})}{\pi}$, where θ is the angle between $\overrightarrow{v_i}$ and $\overrightarrow{v_j}$, and we multiply it by 2 because $\overrightarrow{r}$ could be on either side of the vectors (see Figure 2).

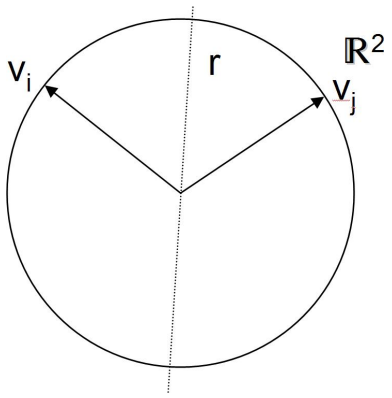

Figure 2: 2D example of RHP

Now we can compute the expected weight of the cut and try to relate to the value of the optimal solution (OPT).

$$\begin{aligned} E(\text{weight of cut}) &= \sum_{(i,j)\in E} w_{ij} Pr((i,j)\text{gets cut}) \\ &= \sum_{(i,j)\in E)} w_{ij} \frac{arccos(\overrightarrow{v_i} \cdot \overrightarrow{v_j})}{\pi} \\ &\geq \alpha \sum_{(i,j)\in E)} w_{ij} \frac{1 - \overrightarrow{v_i} \cdot \overrightarrow{v_j}}{2} \\ &= \alpha\, OPT \end{aligned}$$

To find α, we need to minimize over the dot product:

$$\alpha \geq min_{-1\leq x\leq 1} \frac{\frac{1}{\pi} arccos(x)}{\frac{1}{2}(1-x)} \geq 0.878$$

What we proved is that $E(\text{cut produced by this algorithm}) \geq 0.878\, OPT$. That's the state of the art for MAXCUT [1].
Remark Under the Unique Games Conjecture, this constant (0.878) is optimal.

3 Coloring

Given a graph $G = (V, E)$, we say that it's k-colorable, if $\exists$ assignment f: $V \mapsto [k]$, s.t. $f(i) \neq f(j)$ if $(i, j) \in E$. Or, in other words, it's k-colorable if we can assign one of k colors to each vertex such that no edge has same colors on both ends. Note that coloring the vertices into k colors is the same as dividing them into k sets.
The chromatic number of G is denoted by $\chi(G)$, and it's defined as the smallest k such that G is k-colorable.

3.1 SDP rounding approach for k-colorability

We need to divide vertices into k sets such that every edge between the sets gets cut. Our main approach will be to map vertices to unit vectors in $\mathbb{R}^n$, and then to maximize the distances between endpoints of

edges. We will use a randomized procedure to partition vertices into k sets.
Let's formulate a SDP:

$$\begin{gathered} min\, t \\ \overrightarrow{v_i} \cdot \overrightarrow{v_j} \leq t \; \forall (i,j) \in E \\ \overrightarrow{v_i} \cdot \overrightarrow{v_i} = 1 \; \forall i \in V \\ \overrightarrow{v_i} \in \mathbb{R}^n \; \forall i \end{gathered}$$

By minimizing t, we're minimizing the largest dot product - that is, the angle betweeen two vectors that have an edge between corresponding vertices. So, whenever there's an edge, we want the dot product to be small. While this doesn't have any obvious relationship to colorability, we'll show the connection in a moment.

Lemma 4. *Let t^* be the optimal value of this SDP.*

1. *If G is k-colorable, then $t^* \leq -\frac{1}{k-1}$*
2. *If G has a k-clique, then $t^* \geq -\frac{1}{k-1}$*

Proof

1. We can explicitly construct k vectors $\overrightarrow{w_1}, ..., \overrightarrow{w_k}$ ($\overrightarrow{w_i} \in \mathbb{R}^k$), such that $\overrightarrow{w_i} \cdot \overrightarrow{w_j} = \begin{cases} -\frac{1}{k-1}, i \neq j \\ 1, i = j \end{cases}$
   Then, since t^* is the optimal value of the SDP, it cannot be any larger than the value of our constructed solution.
   Construct the vectors by induction on k:
   Base, $k = 2$: the graph is 2-colorable. Take two unit vectors that point in opposite directions - they will satisfy the desired properties. What this means is that we can map all vertices colored with color 1 to vector 1, and all vertices colored with color 2 to vector 2. Since vertices colored with same color don't have any edges between them, any edge in the graph now corresponds to our two unit vectors. Since they're pointing in opposite directions, their dot product is -1. ✓
   Induction step, $k \to k+1$: we have a set of vectors that satisfy the induction hypothesis, $\overrightarrow{w_1}', ..., \overrightarrow{w_k}'$.
   Construct a new set: $\overrightarrow{w_{k+1}} = (0, ..., 0, 1)$, $\overrightarrow{w_i} = (\sqrt{1 - \frac{1}{k^2}} w_i', -\frac{1}{k})$ $(i = 1..k)$.
   Let's check that the desired properties hold:

$$\begin{aligned} \overrightarrow{w_i} \cdot \overrightarrow{w_i} &= (1 - \frac{1}{k^2}) \overrightarrow{w_i}' \cdot \overrightarrow{w_i}' + \frac{1}{k^2} = 1 \;\; \checkmark \\ \overrightarrow{w_i} \cdot \overrightarrow{w_j} &= (1 - \frac{1}{k^2}) \overrightarrow{w_i}' \cdot \overrightarrow{w_j}' + \frac{1}{k^2} \\ &\leq (1 - \frac{1}{k^2})(\frac{-1}{k-1}) + \frac{1}{k^2} \\ &= -\frac{k^2-1}{k^2} \frac{1}{k-1} + \frac{1}{k^2} \\ &= -\frac{k+1}{k^2} + \frac{1}{k^2} = -\frac{1}{k} \;\; \checkmark \\ \overrightarrow{w_i} \cdot \overrightarrow{w}_{k+1} &= -\frac{1}{k} \;\; \checkmark \end{aligned}$$

2. $i = 1..k$ is a clique $\Rightarrow \forall 1 \leq i < j \leq k \; \overrightarrow{v_i} \cdot \overrightarrow{v_j} \leq t^*$.
   Let's consider the average dot product and show that it's at least $-\frac{1}{k-1}$.
   Then $\exists i, j$ s.t. $\overrightarrow{v_i} \cdot \overrightarrow{v_j} \geq -\frac{1}{k-1} \Rightarrow t^* \geq \overrightarrow{v_i} \cdot \overrightarrow{v_j} \geq -\frac{1}{k-1}$

So this is what we need to prove now: $\overline{a} = \frac{1}{k(k-1)} \sum_{1 \le i,j \le k} (\overrightarrow{v_i} \cdot \overrightarrow{v_j}) \ge -\frac{1}{k-1}$

To prove that, observe

$$(\sum_{i=1}^{k} \overrightarrow{v_i}, \sum_{i=1}^{k}) \overrightarrow{v_i}) = \sum_{i=1}^{k} (\overrightarrow{v_i}, \overrightarrow{v_i}) + k(k-1)\overline{a}$$
$$= k + k(k-1)\overline{a} \ge 0$$

(the last inequality holds because we're computing the dot product of a vector with itself, and that's always ≥ 0)
$k(k-1)\overline{a} \ge 0 \Rightarrow \overline{a} \ge -\frac{1}{k-1}$ ✓

■

Remark Define $\Theta(G) = 1 - \frac{1}{t^*}$ (*Lovasz Theta Function*).
If $t^* = -\frac{1}{k-1}$, then $\Theta(G) = k$ - in that case the graph is called *vector k-colorable*.
We have just proved that

$$\omega(G) \le \Theta(G) \le \chi(G)$$

where $\omega(G)$ is the clique number of G (size of the largest clique), and $\chi(G)$ is the chomatic number of the graph. $\omega(G)$ and $\chi(G)$ can be hugely separated, by n^a. If $\omega(G') = \chi(G') \ \forall G'$, where G' is a vertex-induced subgraph of G, then G is called a *perfect* graph.

3.2 Coloring a 3-colorable graph

Now let's use everything we proved so far to color a 3-colorable graph. We have the same SDP program for t as before, and 3-colorability means that $t^* \le -\frac{1}{2}$ (we can map all vertices to 3 vectors with the angles between any pair of vectors equal to $\frac{2\pi}{3}$; all vertices of same color map to same vector).
Remark At this point in the notes Anna started using λ instead of t, and used t to denote the number of hyperplanes we choose in the algorithm. For the sake of consistency, we stick with using t as part of the SDP, and using m for number of hyperplanes.

Remark We will show how to find a coloring with $\widetilde{O}(n^{0.387})$ colors [2], while the current best is $O(n^{0.211})$ [3]. It is known that coloring a 3-colorable graph using only 4 colors is NP-hard. The Unique Games Conjecture implies that there is no constant factor approximation for 3-coloring. It is also known that approximating k-coloring better than a factor of $n^{1-\varepsilon}$ for arbitrary k is NP-hard.

Denote by Δ the maximal degree in the graph.

Algorithm 2 SDP coloring

1: Solve SDP.
2: Choose $m = 2 + \log_3 \Delta$ random hyperplanes through origin, partition vectors into 2^m subsets, color vectors in each region with a different color.
3: Repeat step 2 until the graph is properly colored.

After step 2, each region has some properly colored vertices (those that don't have any neighbors with same color) and some improperly colored vertices. We apply the algorithm recursively on the improperly colored vertices, using brand new colors - so we don't spoil any properly colored vertices.
Step 2 of the algorithm uses $2^m = 4 \cdot 2^{log_3 \Delta} = 4\Delta^{log_3 2}$ colors.

Claim 5. *Step 2 of the algorithm produces a semi-coloring (coloring of nodes such that $\leq \frac{n}{4}$ edges have same color on both endpoints) with probability at least $\frac{1}{2}$.*

Proof Fix $(i,j) \in E$.

$$\begin{aligned} Pr(\text{i and j get same color}) &= (1 - \frac{arccos(\overrightarrow{v_i} \cdot \overrightarrow{v_j})}{\pi})^m \\ &\leq (1 - \frac{t^*}{\pi})^m \\ &\leq (1 - \frac{-\frac{1}{2}}{\pi})^m \\ &\leq (1 - \frac{1}{\pi}\frac{2\pi}{3})^m = \frac{1}{3^m} \leq \frac{1}{9\Delta} \end{aligned}$$

In line 2, we used the fact that arccos is a decreasing function. In line 3, we used the fact that 3-colorability means that $t^* \leq -\frac{1}{2}$.
This means that $E(\text{number of edges with same color}) \leq \frac{|E|}{9\Delta} \leq \frac{n\Delta}{2 \cdot 9\Delta} = \frac{n}{18}$. By Markov inequality, we have $Pr(\text{number of edges with same color} \geq \frac{n}{4}) \leq \frac{\frac{n}{18}}{\frac{n}{4}} < \frac{1}{2}$ ■

Semicoloring means that $\leq \frac{n}{4}$ have same colors on both endpoints - then at least $\frac{n}{2}$ vertices are properly colored. If k colors are sufficient to get semicoloring, then the graph can be properly colored with $O(k \log n)$ colors (at every iteration, we reduce the number of vertices we need to recolor at least by half). As we just proved, with probability at least $\frac{1}{2}$ we get a semicoloring using $k = 4\Delta^{\log_3 2}$ colors. Then we can get a coloring using $O(\log n \Delta^{\log_3 2})$ colors. If $\Delta = n$, that becomes $\widetilde{O}(n^{\log 32}) = \widetilde{O}(n^{0.631})$ colors, which isn't a very good result, especially since there exists a simple algorithm that uses $O(\sqrt{n})$ to color a 3-colorable graph [4].
This algorithm uses 2 key facts:

1. In a 3-colorable graph, the neighborhood of a vertex is 2-colorable.
2. Graph with maximal degree Δ can be colored with $\Delta + 1$ colors.

Algorithm 3 Simple coloring

1: If there exists a vertex of degree $\geq \sqrt{n}$, find it, 2-color its neighborhood using fresh colors and remove the colored vertices from the graph.
2: Repeat step 1 until there is no more vertices with degree $\geq \sqrt{n}$.
3: Color the rest of the graph using $\sqrt{n}$ additional colors.

Claim 6. *Algorithm 3 uses $O(\sqrt{n})$ colors.*

Proof
At step 1, we remove at least $\sqrt{n}$ vertices from the graph, so step 1 can be repeated at most $\sqrt{n}$ times. For that, we spend at most $2\sqrt{n}$ colors. At the end, maximal degree in the graph is $< n$, so we can color the graph with $\sqrt{n}$ new colors (just greedily go along). All in all, we used $3\sqrt{n}$ colors. ■

Let's use Algorithm 3 to improve on SDP-based coloring (Algorithm 2). The main idea here is that Algorithm 3 is good when the maximal degree in the graph is very large, and it gets worse when the maximal degree diminishes. On the other side, Algorithm 2 works well when the maximal degree is small.

Consider Δ^* to be a parameter of the algorithm.

Algorithm 4 Combined coloring

1: Pick a vertex of degree at least Δ^*, 3-color it and its neighbors, remove them from the graph.
2: Repeat step 1 until all vertices have degree $< \Delta^*$.
3: Run SDP-based coloring (Algorithm 2) to color the rest of the vertices.

For the first 2 steps of the algorithm, we use $\leq \frac{3n}{\Delta^*}$ colors. For the third step, we use $\widetilde{O}(\Delta^{* \log_3 2})$ colors. Choose Δ^* to minimize $\leq \frac{3n}{\Delta^*} + \widetilde{O}(\Delta^{* \log_3 2})$. That gives $\Delta^* = n^{\frac{1}{1.63}}$, and the number of colors $\widetilde{O}(n^{0.39})$.

Remark The question of whether there exists an algorithm for 3-coloring a 3-colorable graph with $\text{poly}(\log n)$ colors remains wide open!

References

[1] M.X. Goemans, D.P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. Journal of the ACM, 1995.

[2] D. Karger, R. Motwani and M. Sudan. Approximate graph coloring by semidefinite programming. In Proceedings of the 35th Annual Symposium on Foundations of Computer Science, 1994.

[3] S. Arora, E. Chlamtac and M. Charikar. New approximation guarantee for chromatic number. In Proceedings of the 38th Annual ACM Symposium on Theory of Computing, 2006.

[4] A. Wigderson. Improving the performance guarantee for approximate graph coloring. Journal of the ACM, 1983.

[5] http://en.wikipedia.org/wiki/Semidefinite_programming

[6] http://en.wikipedia.org/wiki/Matrix_(mathematics)

[7] http://www.stanford.edu/ boyd/papers/pdf/semidef_prog.pdf