Convex and Semidefinite Programming for Approximation

We have seen linear programming based methods to solve NP-hard problems. One perspective on this is that linear programming is a meta-method since it allows modeling of wide variety of problems (via integer programming) and further linear programs can be solved in polynomial time.

In mathematical programming, a more general polynomial time solvable methodology is convex programming which allows us to solve the following problem:

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

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

Convex Programs

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

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

To solve the problem we need to be able to do things efficiently:

given x, evaluate $f(x)$ in polynomial time

given x, output if $x \in S$ or not and it is it 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 $\log (1/\varepsilon)$

Convex Programs

One can apply convex programming the same way as linear programming to obtain a *relaxation* that can be solved in polynomial time. Although convex programming is more general, it is not as widely used as linear programming in approximation algorithms.

There are several reasons. First, integer programs are natural and easy to write for NPO problems and their relaxations turn out to be linear programs. Second, the duality theory of linear programs helps in understanding and rounding the relaxation. No general duality theory exists for convex programming. Third, our mathematical toolkit is perhaps not sophisticated enough yet!

Semidefinite Programming (SDP)

A special class of convex programs that are more general than linear programs have made their way into approximation algorithms starting with the simple but spectacular result of Goemans and Williamson on approximating Max-Cut.

The advantage of SDPs is that they provide a natural modeling language for a certain class of quadratic programming problems and they are closer to linear programming in that they have duality theory (although we haven't really been able to exploit it like we do with linear programs).

We will see a few applications of SDP methods for approx.

SDP

SDP is based on the properties of positive semi-definite matrices:

A $n \times n$ *real symmetric* matrix A is *positive semi-definite* if one of the following equivalent conditions holds

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

An important fact from property 1 above is that given two psd matrices A and B, $a A + b B$ is also psd for $a, b \geq 0$

Thus the set of all $n \times n$ psd matrices is a convex set in R^{n^2}

SDP

Thus the set of all $n \times n$ psd matrices is a convex set in R^{n^2}

Let M_n denote the set of $n \times n$ real matrices

Given two $n \times n$ matrices A and B define

$A \cdot B = \sum_{i,j} a_{ij} b_{ij}$

The SDP problem is given by matrices $C, D_1, D_2, .., D_k$ from M_n and scalars $d_1, d_2, ..., d_k$. The variables are given by a matrix Y

$\max C \cdot Y$

$D_i \cdot Y = d_i \quad 1 \leq i \leq k$

$Y \succcurlyeq 0$

$Y \in M_n$

SDP

$\max C \cdot Y$

$D_i \cdot Y = d_i \quad 1 \leq i \leq k$

$Y \succcurlyeq 0$

$Y \in M_n$

The constraint $Y \succcurlyeq 0$ 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

The claim is that sdp can be solved in polynomial time

Solvability of SDP

SDP can be solved in polynomial time to an arbitrary desired accuracy from the fact that it is a special case of convex programming.

To see this we note that the "actual" variables in sdp are y_{ij} , $1 \leq i \leq n$, $1 \leq j \leq n$

Note that the objective function and constraints are linear in these variables

The only complex constraint is that $Y \succcurlyeq 0$

Since the set of all psd matrices in M_n is a convex set, this simply enforces a constraint that the variables y_{ij} belong to this convex set

Solvability of SDP

To see that SDP can be solved in poly-time the main thing to see is that we have a separation oracle for the convex set defining psd matrices.

That is, we need to give a polynomial time algorithm that given a matrix $A \in M_n$ checks if A is psd or not and if A is not psd, it outputs a hyperplane (in R^{n^2}) that separates the convex set that contains all psd matrices from A

The algorithm is simple from the characterization of psd matrices

Solvability of SDP

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 the eigenvalues of A and if they are all non-negative then A is psd

Otherwise there is an eigen-vector v of A such that

$Av = \lambda v$ and $\lambda < 0$ which implies $v^t A v = \lambda < 0$

and hence the violating hyper-plane is simply $v^t Y v \geq 0$ (which is valid for all psd matrices by property 1)

Solvability of SDP

Thus the constraint $Y \succcurlyeq 0$ can be thought of as imposing the following *infinite* family of *linear* constraints on the variables y_{ij}

$y_{ij} = y_{ji}$ for all i,j

$v^t Y v \geq 0$ for all $v \in R^n$

We can, in essence, separate over the above system in polynomial time

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 R^n)

The objective function and constraints are linear in the "actual" variables, namely the inner products $v_i \cdot v_j$.

Example:

$\max (1 - v_1 \cdot v_2 + v_2 \cdot v_3)$

s.t

$v_1 \cdot v_2 - v_2 \cdot v_3 = 5$

$v_1, v_2, v_3 \in R^n$

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.

SDP for Max-Cut

We now give an approximation algorithm for Max-Cut via SDP methods

Recall that Max-Cut is the problem of partitioning an edge weighted graph $G=(V,E)$ into two parts $(S, V\backslash S)$ to maximize $w(\delta(S))$

Such cut problems are sometime easy to express as *quadratic programs*

It turns out that in some cases the relaxation of a quadratic program to a vector program yields good (fantastic) relaxations

Quadratic program for Max-Cut

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 $i \in V \backslash S$

Then it is easy to see that Max-Cut is modeled by the following quadratic program

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

s.t

$$y_i \in \{-1, 1\}, \quad i \in V$$

The above constraint is equivalent to

$$y_i \cdot y_i = 1$$

Vector program for Max-Cut

The program

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

s.t

$y_i \cdot y_i = 1 \quad i \backslash in V$

is equivalent to the following vector program where we constrain the vectors to be in R^1. We have a vector v_i for each $i \in V$

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

s.t

$v_i \cdot v_i = 1$

$v_i \in R^1$

Vector program for Max-Cut

Since we cannot solve the constrained vector program we let the vectors lie in any dimension (in particular R^n since that is equivalent to SDP)

Thus we obtain the following relaxation

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

s.t

$$v_i \cdot v_i = 1$$

$$v_i \in R^n$$

From our previous discussion we can solve above to an arbitrary precision using SDP

Rounding

Let OPT_v be an optimum solution value to the vector program. Since it is a relaxation, $OPT_v \geq OPT$

Let the vectors achieving OPT_v be $v_1^*, \ldots, v_n^*$

Note that each v_i^* is a unit vector in 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

Set $S = \{ i \mid r \cdot v_i^* > 0 \}$, $V \setminus S = \{ i \mid r \cdot v_i^* \leq 0 \}$

Analysis

To analyze this we need to understand the probability that an edge (i,j) is cut in the random hyper-plane algorithm and the value that (i,j) contributes to OPT_v

Let θ_{ij} be the angle between v^*_i and v^*_j

Since all vectors are unit vectors, $\cos(\theta_{ij}) = v^*_i \cdot v^*_j$

Note that the 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?

We claim that it is precisely equal to θ_{ij}/π

See figure

Analysis

Thus the total expected weight of the cut found by the algorithm, by linearity of expectation, is

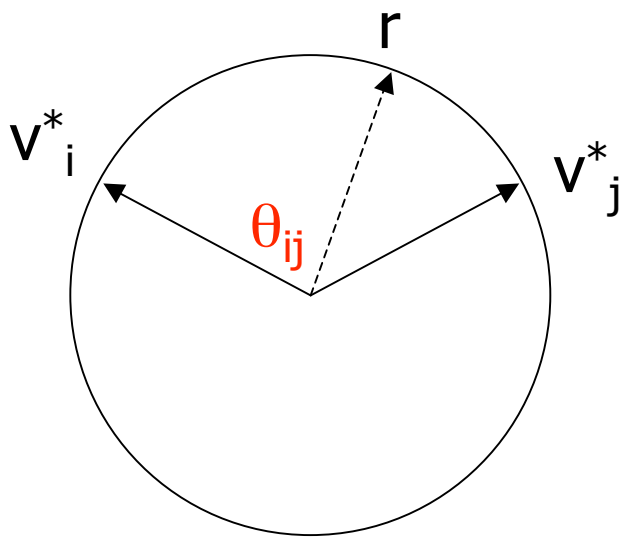
$\sum_{ij \in E} w_{ij}\ \theta_{ij}/\pi$

A simple claim from elementary calculus shows that

$\theta/\pi \geq \alpha\ (1\text{-}\cos(\theta))/2$

for all $\theta \in [0, \pi]$ where

$\alpha \simeq 0.87856$



Analysis

Thus the expected weight of the cut is at least α OPT_v which implies an α randomized approx for Max-Cut

A technical issue is how to pick a random unit vector in R^n. This is not completely obvious or trivial

We do this by picking n independent identical Gaussian random variables $x_1, x_2, .., x_n$ with 0 mean and std deviation 1 (with density function $e^{t^2/2}/\text{sqrt}(2\pi)$)

We let $r = (x_1, x_2, ..., x_n)/||x||$ where

$||x|| = \text{sqrt}(x_1^2 + x_2^2 + ... + x_n^2)$ is the length of x

Analysis

We let $r = (x_1, x_2, \ldots, x_n)/||x||$ where

$||x|| = \text{sqrt}(x_1^2 + x_2^2 + \ldots + x_n^2)$ is the length of x

The reason the above works is because the density function for the resulting vector is

$f(y_1, y_2, \ldots, y_n)$ is proportional to $e^{\sum_i y_i^2/2}$

Note that the density function is dependent only on the length of the vector and hence is centrally symmetric - all directions are equally likely

Thus normalizing the vector gives a random unit vector

SDP for Graph Coloring

Graph coloring:

Given graph $G=(V,E)$, color vertices of G using colors from say $\{1,2,\ldots,n\}$ such that for every edge (u,v) the colors of u and v are distinct.

Goal: minimize the number of distinct colors used

$\chi(G)$: chromatic number of G is the min # of colors needed for G

Thus graph coloring is to find/approximate $\chi(G)$

Hardness of coloring

It is known that coloring is very hard to approximate in general. Unless P=NP there is no $n^{1-\varepsilon}$ approximation for any fixed $\varepsilon > 0$!

The hardness comes from a gap reduction (using PCP technology) that show that distinguishing graphs with $\chi(G) \leq n^{\varepsilon}$ and $\chi(G) > n^{1-\varepsilon}$ is NP-hard

However in many cases we are interested in the situation when $\chi(G)$ is small and the above hardness does not directly apply

The following is also known. For all $k \geq k_0$ for some sufficiently large constant k_0, it is NP-hard to color a graph with $\chi(G) = k$ with less than $f(k)$ colors where $f(k)$ is a polynomial in k. With stronger assumptions $f(k)$ can be made quasi-poly in k.

3-coloring

The simplest case of interest is when $\chi(G) = 3$

Note that if $\chi(G) = 2$ then G is a bipartite graph and it can relatively easily be checked whether G is bipartite or not (how?)

Deciding if $\chi(G) = 3$ is NP-hard, one of the problems in Karp's list (the hardness holds even for planar graphs which can always be 4-colored by the famous 4-color theorem)

What can we say about hardness of 3-coloring?

What is known is the following. It is NP-hard to decide if $\chi(G) = 3$ or $\chi(G) \geq 5$

Coloring 3-colorable graphs and promise problems

We now consider approximation algorithms that color a 3-colorbale graph with "few" colors

We first observe that the problem we are considering is somewhat non-standard in the following sense. We cannot check if a graph is 3-colorable so what does it mean to color a 3-colorable graph with "few" colors?

Technically, we are working on what is called a *promise problem.* These are problems for which the input is guaranteed to satisfy some property: the algorithm can produce garbage if input does not satisfy property

Coloring 3-colorable graphs with $O(n^{1/2})$ colors

Any graph G can be colored with $\Delta(G) + 1$ colors where $\Delta(G)$ is the maximum degree (how?)

Another simple observation is the following.

Given a vertex u, let $N(u) = \{ v \mid uv \in E \}$ be the neighbors of u

It is easy to see that $\chi(G[N(u)]) \leq \chi(G) - 1$, that is the neighbors of u can be colored with $\chi(G)-1$ colors (why?)

Thus if $\chi(G) = 3$ then for every u, G[N(u)] is bipartite and hence we can check and color them using 2 colors

Coloring 3-colorable graphs with $O(n^{1/2})$ colors

$\Delta(G)+1$ coloring is good when $\Delta(G)$ is small

Otherwise there is a vertex of large degree but then its neighbors can be colored using 2 colors

We take advantage of the above two in the following algorithm

Let δ be a parameter that will be fixed later

$G' = G$

While ($\Delta(G') > \delta$)

  let $v \in V(G')$ be a vertex of degree $\Delta(G')$

  color $N(v)$ with 2 *fresh* colors, color v with a fresh color

  remove v and $N(v)$ from G'

endwhile

Color G' with $\delta+1$ *fresh* colors

Coloring 3-colorable graphs with $O(n^{1/2})$ colors

The total number of colors used is seen to be at most $3t + (\delta+1)$ where t is the number iterations of the while loop

In each iteration we remove at least $(\delta+1)$ vertices and hence $t \leq n/(\delta+1)$

Therefore the number of colors used is at most

$3n/(\delta+1) + (\delta+1)$

This is minimized when we choose $\delta = \Theta(n^{1/2})$. This gives a coloring with $O(n^{1/2})$ colors.

SDP for coloring

We now give an algorithm based on SDP that colors a 3-colorable graph with $O(\Delta^a \log n)$ colors where $a = \log 2/\log 3 \simeq 0.631$

This can be plugged into the previous algorithm to optimize δ and obtain an $O(n^{0.387})$ coloring

We will then improve the algorithm to color the graph using $O(\Delta^{1/3} (\log \Delta)^{1/2} \log n)$ colors which will yield an $O(n^{1/4} \log^{1/2} n)$ coloring

The approach generalizes to k-colorable graphs. One can color them with $O(\Delta^{1-2/k} (\log \Delta)^{1/2} \log n)$ colors or $O(n^{1-3/(k+1)} \log^{1/2} n)$ colors

Vector chromatic number

The vector chromatic number of G is a solution to the following SDP

$$\min z$$

$$v_i \cdot v_j \leq z \qquad (i,j) \in E$$

$$v_i \cdot v_i = 1 \qquad i \in V$$

$$v_i \in R^n \qquad i \in V$$

That is, we assign unit vectors to vertices so that for any edge the inner product is at most z

Vector chromatic number

Claim: The vector chromatic number of a k-colorable graph is at most $-1/(k-1)$

To see this, let $V_1, V_2, \ldots, V_k$ be the color classes of G

We assign the same k-dimensional vector for each vertex in V_i as $(\beta, \beta, \ldots, \alpha, \ldots, \beta)$ where the α is in the i'th position

We set $\alpha = -(k-1)^{1/2}/k^{1/2}$ and $\beta = 1/(k(k-1))^{1/2}$

The k-dimensional vectors can be lifted to n-dimensinoal vectors by adding zeroes

It is easy to check that these achieve $z = -1/(k-1)$ for SDP

Vector chromatic number

For $k = 3$, the the three vectors that achieve $-1/2$ are simply the three unit vectors in the plane spaced 120 degrees apart

Note that the scheme in the previous slide embedded the vectors in R^k but the vectors actually live in a $k-1$ dimensional sub-space since they are all orthogonal to the vector $(1,1,\ldots,1)$ and that is why for $k=3$ we can embed in the plane

Using vector chromatic number

We focus on 3-colorable case

The algorithm first computes a vector chromatic number for G which we can assume is at most -1/2

Let $v_1, v_2, \ldots, v_n$ be the vectors from the SDP

We first give an algorithm that colors G using $O(\Delta^a \log n)$ colors where $a = \log 2/\log 3 \simeq 0.631$

Let θ_{ij} be the angle between v_i and v_j

It follows that for each edge (i,j), $\theta_{ij} \geq 2\pi/3$ (120 degrees)

Using vector chromatic number

Let t be a parameter to be chosen later
The algorithm picks t random unit vectors $r_1, r_2, \ldots, r_t$
Let $\mathrm{sgn}(u \cdot v)$ be -1 if $u \cdot v < 0$ and 1 otherwise

For each vertex i, we obtain a color as
$\mathrm{color}(i) = (\mathrm{sgn}(v_i \cdot r_1), \mathrm{sgn}(v_i \cdot r_2), \ldots, \mathrm{sgn}(v_i \cdot r_t))$

Note that the total number of colors used is 2^t

Analysis

To understand the algorithm let us analyze the probability that an edge (i,j) is *not* properly colored, that is color(i) = color(j) (we say the edge is monochromatic)

Since each r_i is an independent random unit vector

$\Pr[\text{color}(i) = \text{color}(j)] = \alpha^t$ where α is the probability that $\text{sgn}(r \cdot v_i) = \text{sgn}(r \cdot v_j)$ for a random unit vector

$\Pr[\text{sgn}(r \cdot v_i) = \text{sgn}(r \cdot v_j)] = 1 - \theta_{ij}/\pi$

For an edge (i,j) by the SDP, $\theta_{ij}/\pi \geq 2/3$

therefore $\Pr[\text{color}(i) = \text{color}(j)] \leq 1/3^t$

Analysis

We set $t = 2 + \log_3 \Delta$

Then an edge (i,j) is monochromatic with probability at most $1/3^t \leq 1/(9\Delta)$

Thus the expected number of monochromatic edges is at most $m/(9\Delta) \leq n\Delta/(18\Delta) \leq n/18$

Suppose we remove all vertices incident to a monochromatic edge - in expectation we lose at most $2n/18 \leq n/9$ vertices

The graph induced on the remaining $n-n/9$ vertices is properly colored!

Analysis

Thus using $2^t \simeq 4\ \Delta^a$ colors we have colored a constant fraction of the graph (here $a = \log 2/\log 3$)

We repeat this $\log n$ times to color the entire graph (note that the vector coloring need not be recomputed, the same vectors would work)

Note that we need to use a fresh set of 2^t colors in each iteration and hence the total number of colors used is $O(\Delta^a \log n)$

Analysis

To make this a rigorous argument we need to ensure that in each iteration we color a constant fraction of the graph. In the analysis we showed that the number of vertices incident to a monochromatic edge is in expectation at most $n/9$. Thus with probability at least $1/2$ it is at most $2n/9$.

We can repeat the rounding several times to ensure that with high probability at most $2n/9$ vertices are incident to a monochromatic edge.

We could also do an overall expectation analysis

An improved algorithm

We now describe an improved algorithm and its analysis which obtains a coloring using $O(\Delta^{1/3} (\log \Delta)^{1/2} \log n)$ colors.

The idea is to to find an *independent set* of size

$\Omega(n / (\Delta^{1/3} (\log \Delta)^{1/2}))$ using the fact that the graph has vector chromatic number 3. This can be iterated to obtain the desired coloring.

The algorithm and analysis rely on some useful properties of projections of vectors on to random lines. We first review some of these properties.

Properties of Normal Distribution

In the following we use some properties of the *normal distribution (or guassian distribution)*. The density function of a standard one dimensional normal distribution with mean μ and variance σ^2 is given by

$f(y) = e^{-(y-\mu)^2/2\sigma^2}/((2\pi)^{1/2}\ \sigma)$

Here we will be concerned only with the simple setting with $\mu=0$ and $\sigma = 1$ in which case the above simplifies to

$f(y) = e^{-y^2/2}/\ (2\pi)^{1/2}$

A basic fact about normal distributions is that the sum of two *independent* normal distributed random variables with means μ_1 and μ_2 and variances σ_1^2 and σ_2^2 is itself normally distributed with mean $\mu_1+\mu_2$ and variance $\sigma_1^2 + \sigma_2^2$

Properties of Normal Distribution

We will be interested in the tail distribution of a basic normal variable with mean 0 and variance 1.

Define $\Phi(y)$ by the following

$$\Phi(y) = \int_y^\infty f(y)dy$$

In other words $\Phi(y)$ is the probability that a basic normal variable will take a value greater than equal to y. A simple lemma that we won't prove is the following:

Lemma: For every $x > 0$, $f(x)(1/x - 1/x^3) \leq \Phi(x) \leq f(x)/x$

Projections on to "random" vectors

Let v be a unit vector in $\mathcal{R}^n$. Given a random unit vector r in $\mathcal{R}^n$ we are interested in the distribution of the inner product $r \cdot v$ which is the projection of v in a random direction.

It turns out that the above projection is somewhat more messy to analyze. Instead we choose r to be a random vector $(X_1, X_2, \ldots, X_n)$ where the X_i are *independent* with a basic normal distribution (mean 0, variance 1). Recall that we used this with scaling to generate a random unit vector!

From here on, we will use "random" vector to denote a vector chosen as above

Projections on to "random" vectors

The advantage of the random vector that we defined is given by the following.

Lemma: For a unit vector v and a random vector r the variable $r \cdot v$ has a basic normal distribution

The proof is simple. Note that $r \cdot v = \sum_i v(i) X_i$ where $v(i)$ is the i'th coordinate of v and X_i is a basic normal variable. Sums of independent normal variables is normally distributed. The mean of $r \cdot v$ is 0 and variance is $\sum_i v(i)^2$. Since v is a unit vector the variance is 1

Projection based algorithm

Now we describe the algorithm to find a large independent set. Recall that the vector chromatic number is 3 implying that we have unit vectors $v_1, v_2, \ldots, v_n$ for the nodes of the graph such that $v_i \cdot v_j \leq -1/2$ for each edge (i,j) of the graph

We choose a threshold θ that we will fix later

Algorithm:

Choose a random vector r

$S = \{ i \mid r \cdot v_i \geq \theta \}$

Obtain independent set $S' \subseteq S$ by removing one end point (arbitrarily) of each edge (i,j) in G[S]

Analysis

We wish to lower bound the size of $Z = |S'|$ since that is the desired independent set size.

Let $X = |S|$ and $Y = |E[S]|$ be random variables for the number of nodes in S and the number of edges in the graph $G[S]$

We observe that $Z \geq \max\{0, X - Y\}$ since each edge in $G[S]$ can remove at most one node from S in the deletion phase (to obtain S')

Thus $\text{Expect}[Z] \geq \text{Expect}[X] - \text{Expect}[Y]$

Analysis

Expect[Z] $\geq$ Expect[X] - Expect[Y]

Note that Expect[X] = $\sum_{i \in V}$ Pr[r·$v_i \geq \theta$]

By symmetry, Pr[r· $v_i \geq \theta$] is the same for all v_i

Note that r· v_i is distributed normally (v_i is a unit vector and r is a random vector) and hence

Pr[r· $v_i \geq \theta$] = $\Phi(\theta)$

Thus Expect[X] = n $\Phi(\theta)$

Analysis

Now we upper bound Expect[Y]

It is straight forward to see that

Expect[Y] = $\sum_{(i,j) \in E}$ Pr[i, j both in S]

Pr[i, j both in S] = Pr [$r \cdot v_i \geq \theta$ and $r \cdot v_j \geq \theta$]

= Pr[$r \cdot (v_i + v_j) \geq 2\theta$]

We know that $r \cdot v$ is normally distributed if v is a unit vector. In the above (v_i+v_j) is not a unit vector. So let $u = (v_i+v_j)/||v_i+v_j||$ be a unit vector in the direction of v_i+v_j

Analysis

Let $u = (v_i+v_j)/||v_i+v_j||$

We have $||v_i+v_j||^2 = ||v_i||^2 + ||v_j||^2 + 2v_i \cdot v_j$

If (i,j) is an edge then $v_i \cdot v_j \leq -1/2$ which implies that $||v_i+v_j||^2 \leq 2-1 \leq 1$, hence $||v_i+v_j|| \leq 1$

Therefore,

$$\begin{aligned}
\Pr[i, j \text{ both in } S] &= \Pr[r \cdot v_i \geq \theta \text{ and } r \cdot v_j \geq \theta] \\
&= \Pr[r \cdot (v_i + v_j) \geq 2\theta] \\
&= \Pr[r \cdot u \geq 2\theta/||v_i+v_j||] \\
&\leq \Pr[r \cdot u \geq 2\theta] \\
&\leq \Phi(2\theta)
\end{aligned}$$

Analysis

Thus,

Expect[Y] = $\sum_{(i,j) \in E}$ Pr[i,j both in S]

$\leq m\ \Phi(2\theta) \leq n\ \Delta\ \Phi(2\theta)/2$

m is the number of edges in G which is at most $n\Delta/2$

Hence

Expect[Z] $\geq$ Expect[X] - Expect[Y] $\geq n\ \Phi(\theta) - n\Delta\ \Phi(2\theta)/2$

We can choose θ and use properties of Φ to maximize the above quantity

Analysis

$\text{Expect}[Z] \geq \text{Expect}[X] - \text{Expect}[Y] \geq n\,\Phi(\theta) - n\Delta\,\Phi(2\theta)/2$

We can choose θ and use properties of Φ to maximize the above quantity

We have earlier claimed that

$f(x)(1/x - 1/x^3) \leq \Phi(x) \leq f(x)/x$

where $f(x) = e^{-x^2/2}/(2\pi)^{1/2}$

Set $\theta = ((2/3)\log \Delta)^{1/2}$

Then using above inequalities on Φ and some simple algebra we can show that

$n\,\Phi(\theta) - n\Delta\,\Phi(2\theta)/2 = \Omega(n/(\Delta^{1/3}(\log \Delta)^{1/2})$

Summary

The projection based algorithm yields in expectation an independent set of size $\Omega(n/(\Delta^{1/3}(\log \Delta)^{1/2}))$ if the graph has vector chromatic number 3

We can iterate the algorithm to color the graph using $O(\Delta^{1/3} (\log \Delta)^{1/2} \log n)$ colors

Combining this with the greedy approach to removing high degree vertices gives a coloring using $O(n^{1/4} (\log n)^{1/2})$ colors

Gap of vector chromatic number

It is known that there are graphs for which chromatic number if $n^{0.05}$ while vector chromatic number is 3!

Thus using vector chromatic number alone, one cannot hope to obtain better than n^{ε} approximation.

It is also known that the analysis we showed is essentially tight as a function of Δ and k. There are graphs with vector chromatic number k and chromatic number at least $\Delta^{1-2/k-\varepsilon}$