CS787: Advanced Algorithms

Scribe: Amanda Burton, Leah Kluegel **Lecturer:** Shuchi Chawla
Topic: Max-Cut via SDP ctd., Streaming Algorithms **Date:** October 24, 2007

17.1 Last Time

Last time we discussed Semi-Definite Programming. We saw that Semi-Definite Programs are a generalization of Linear Programs subject to the nonlinear constraint that a matrix of variables in the program is Positive Semi-Definite. Semi-Definite Programs are also equivalent to Vector Programs. Vector Programs are programs over vectors instead of real uni-dimensional variables. The constraints and objective function of these programs are linear over dot products of vectors. We know how to solve these programs to within a $(1+\epsilon)$-factor for any constant $\epsilon > 0$ in time that is polynomial in the size of the program and $1/\epsilon$. So we can assume that we can get the optimal solution, or at least a near optimal solution, to these programs. To design approximation algorithms, instead of expressing programs as Integer Programs and then relaxing them to Linear Programs, we can express programs as Integer Programs or Quadratic Integer Programs, relax them to Vector Programs, solve the Vector Program using Semi-Definite Programming, and then somehow round the solutions to integer solutions.

17.2 Max-Cut

Recall that for the Max-Cut problem we want to find a non-trivial cut of a given graph such that the edges crossing the cut are maximized, i.e.

Given: $G = (V, E)$ with c_e =cost on edge e.
Goal: Find a partition (V_1, V_2), $V_1, V_2 \neq \phi$, $\max \sum_{e \in (V_1 \times V_2) \cap E} c_e$.

17.2.1 Representations

How can we convert Max-Cut into a Vector Program?

17.2.1.1 Quadratic Programs

First write Max-Cut as a Quadratic Program.

Let $x_u = 0$ if vertex u is on the left side of the cut and $x_u = 1$ if vertex u is on the right side of the cut. Then we have

Program 1:

$$\begin{array}{lll} \text{maximize} & \sum_{(u,v)\in E} (x_u(1-x_v) + x_v(1-x_u))c_{uv} & \text{s.t.} \\ & x_u \in \{0,1\} & \forall u \in V \end{array}$$

Alternatively, let $x_u = -1$ if vertex u is on the left side of the cut and $x_u = 1$ if vertex u is on the right side of the cut. Then we have

Program 2:

$$\begin{array}{lll} \text{maximize} & \sum_{(u,v)\in E} \frac{(1-x_u x_v)}{2} c_{uv} & \text{s.t.} \\ & x_u \in \{-1, 1\} & \forall u \in V \end{array}$$

Note that $x_u x_v = -1$ exactly when the edge (u, v) crosses the cut.

We can express the integrality constraint as a quadratic constraint yielding

Program 3:

$$\begin{array}{lll} \text{maximize} & \sum_{(u,v)\in E} \frac{(1-x_u x_v)}{2} c_{uv} & \text{s.t.} \\ & x_u^2 = 1 & \forall u \in V \end{array}$$

In these programs an edge contributes c_{uv} exactly when the edge crosses the cut. So in any solution to these quadratic programs the value of the objective function exactly equals the total cost of the cut. We now have an exact representation of the Max-Cut Problem. If we can solve Program 3 exactly we can solve the Max-Cut Problem exactly.

17.2.1.2 Vector Program

Now we want to relax Program 3 into a Vector Program.

Recall:
A Vector Program is a Linear Program over dot products.

So relax Program 3 by thinking of every product as a dot product of two n-dimensional vectors, x_u.

Program 4:

$$\begin{array}{lll} \text{maximize} & \sum_{uv\in E} \frac{(1-x_u \cdot x_v)}{2} c_{uv} & \text{s.t.} \\ & x_u \cdot x_u = 1 & \forall u \in V \end{array}$$

This is something that we know how to solve.

17.2.1.3 Semi-Definite Program

How would we write this as a Semi-Definite Program?

Recall:

Definition 17.2.1 *A Semi-Definite Program has a linear objective function subject to linear constraints over x_{ij} together with the semi-definite constraint $[x_{ij}] \succeq 0$.*

Theorem 17.2.2 *For any matrix A, A is a Positive Semi-Definite Matrix if the following holds.*

$$\begin{aligned} A \succeq 0 &\Rightarrow v^T A v \geq 0 \quad \forall v \in V \\ &\Leftrightarrow A = C^T C, \text{ where } C \in \mathbb{R}^{m \times n} \\ &\Leftrightarrow A_{ij} = C_i \cdot C_j \end{aligned}$$

To convert a Vector Program into a Semi-Definite Program replace dot products with variables and add the constraint that the matrix of dot products is Positive Semi-Definite.

So our Vector Program becomes

Program 5:

$$\begin{array}{lll} \text{maximize} & \sum_{(u,v)\in E} \frac{(1-y_{uv})}{2} c_{uv} & \text{s.t.} \\ & y_{uu} = 1 & \forall u \in V \\ & [y_{ij}] \succeq 0 & \end{array}$$

If we have any feasible solution to this Semi-Definite Program (Program 5), then by the above theorem there are vectors $\{x_u\}$ such that $y_{uv} = x_u \cdot x_v \forall u, v \in V$. The vectors $\{x_u\}$ are a feasible solution to the Vector Program (Program 4). Thus the Semi-Definite Program (Program 5) is exactly equivalent to the Vector Program (Program 4).

17.2.2 Solving Vector Programs

We know how to get arbitrarily close to the exact solution to a Vector Program. So we get some set of vectors for which each vertex is mapped to some n-dimensional vector around the origin. These vectors are all unit vectors by the constraint that $x_u \cdot x_u = 1$, so the vectors all lie in some unit ball around the origin.

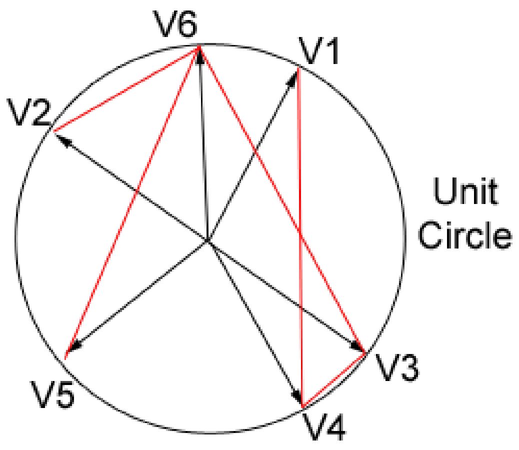

Figure 17.2.1: The unit vectors in the Vector Program solution.

Now we have some optimal solution to the Vector Program. Just as in the case when we had an exact representation as an Integer Program and relaxed it to a Linear Program to get an optimal solution that was even better than the optimal integral solution to the Integer Program, here we have some exact representation of the problem and we are relaxing it to some program that solves the program over a larger space. The set of integral solutions to Program 3 form a subset of feasible solutions to Program 4, i.e., Program 4 has a larger set of feasible solutions. Thus the optimal solution for the Vector Program is going to be no worse than the optimal solution to the original problem.

Fact: $OPT_{VP} \geq OPT_{Max-Cut}$.

Goal: Round the vector solution obtained by solving the VP to an integral solution (V_1, V_2) such that the total value of our integral solution $\geq \alpha OPT_{VP} \geq \alpha OPT_{Max-Cut}$.

Our solution will put some of the vectors on one side of the cut and the rest of the vectors on the other side of the cut. Our solution is benefitting from the edges going across the cut that is produced. Long edges crossing the cut will contribute more to the solution than short edges crossing the cut because the dot product is smaller for the longer edges than the shorter edges. We want to come up with some way of partitioning these vertices so that we are more likely to cut long edges.

17.2.3 Algorithm

In two dimensions good cut would be some plane through the origin such that vectors on one side of the plane go on one side of the cut and vectors that go on the other side of the plane go on the other side of the cut. In this way we divide contiguous portions of space rather than separating the vectors piecemeal.

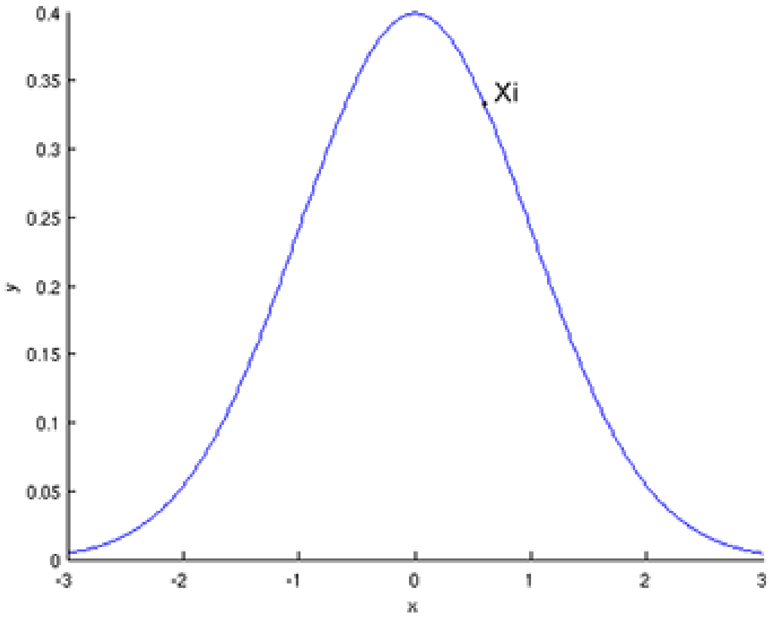

Figure 17.2.2: A sample Gaussian variable, x.

1. Pick a “random” direction - unit vector $\hat{n}$.

2. Define $V_1 = \{v | x_v \cdot \hat{n} \geq 0\}$, $V_2 = \{v | x_v \cdot \hat{n} < 0\}$.

3. Output (V_1, V_2).

To pick a "random" direction we want to pick a vector uniformly at random from the set of all unit vectors. Suppose we know how to pick a normal, or gaussian, variable in one dimension. Then pick from this normal distribution for every dimension.

This gives us a point that is spherically symmetric in the distribution. The density of any point x in the distribution is proportional to $exp^{-\frac{1}{2}x^2}$. So if each of the components, x_i is picked using this density, the density of the vector is proportional to $\prod_i exp^{-\frac{1}{2}x_i^2} = exp^{(-\frac{1}{2}\sum_i x_i^2)}$, which depends only on the length of the vector and not the direction. Now normalize the vector to make it a unit vector.

We now want to show that the probability that an edge is cut is proportional to it's length. In this way we are more likely to cut the longer edges that contribute more to the value of the cut. So what is the probability that we will cut any particular edge?

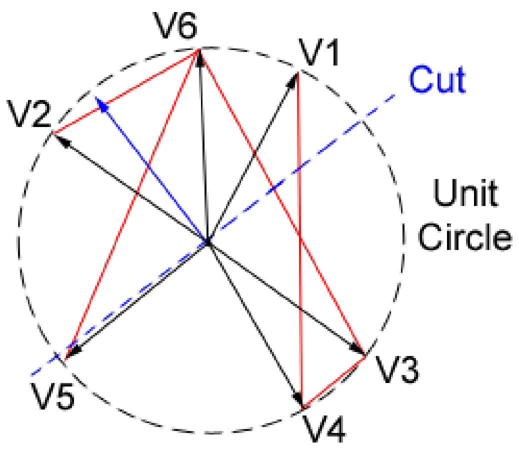

Figure 17.2.3: An example cut across the unit circle.

From Figure 17.2.3 we can see that

$Pr[$ our algorithm cuts edge $(u,v)] = \frac{\theta_{uv}}{\pi}$.

This is for two dimensions, what about n dimensions? If the cutting plane is defined by some random direction in n dimensions, then we can project down to two dimensions and it is still uniformly at random over the n dimensions. So we have the same probability for n dimensions. So the expected value of our solution is

$E[$ our solution $] = \sum_{(u,v)\in E} \frac{\theta_{uv}}{\pi} c_{uv}$

In terms of θ_{uv} the value of our Vector Program is

$Val_{VP} = \sum_{(u,v)\in E} (\frac{1-cos(\theta_{uv})}{2})c_{uv}$

Now we want to say that $E[$ our solution $]$ is not much smaller than Val_{VP}. So we look at the ratio of $E[$ our solution $]$ to Val_{VP} is not small.

Claim 17.2.3 *For all θ, $\frac{2\theta}{\pi(1-cos\theta)} \geq 0.878$*

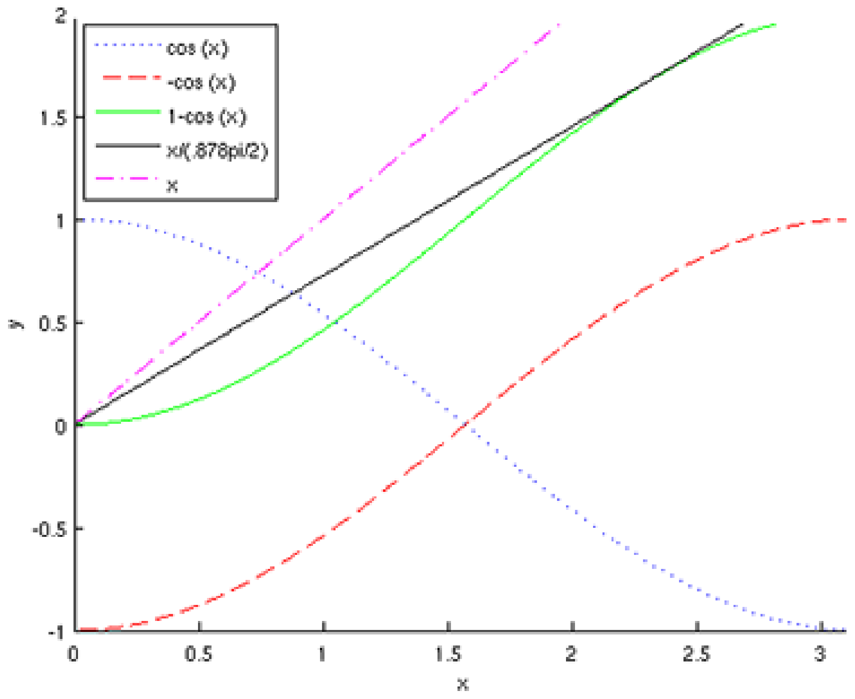

Figure 17.2.4: A visual representation of the .878-approximation to the solution.

As a corollary we have the following theorem. (Note: $1.12 \approx 1/0.878$.)

Theorem 17.2.4 *We get a* 1.12*-approximation.*

17.2.4 Wrap-up

This algorithm is certainly better than the 2-approximation that we saw before. Semi-Definate programming is a powerful technique, and for a number of problems gives us stronger approximations than just Linear Programming relaxations. As it turns out, the Semi-Definate solution for Max-cut is the currently best-known approximation for the problem.

There is reason to believe that unless $P = NP$, you cannot do better than this approximation. This result is highly surprising, given that we derived this number from a geometric argument, which seems like it should have nothing to do with the $P = NP$ problem. The conjecture is that you cannot get a better approximation unless a certain problem can be solved in P, and that certain problem is strongly suspected to be NP-hard. In comes down to a combinatorial problem that in the end has little to do with the geometric argument we used to get an approximation.

17.3 Streaming Algorithms

Sreaming Algorithms are useful under very extreme conditions, where we do not have very much space or very much time to work with at all. A good example to keep in mind for these kinds of algorithms is analyzing packets of information sent across some network, such as the internet. Say we want to be able to compute statistics about the packets being sent or detect attacks or determine customer usage costs, we need to be able to read the packets as they're being sent over the network and compute statistics. The problem under these conditions is that there is a huge amount of data going by and we don't have much time to read the packet or do any sophisticated analysis on that data.

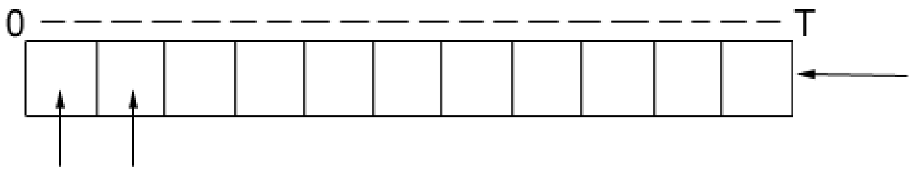

Figure 17.3.5: An example input stream.

The model is that we have a stream of input data, of some very large length T. Ideally, the amount of storage spacewe would like to use would be some constant, or barring that, logarithmic in terms of T. Also, at every point in time, we read some new data from the stream, do some computation, and update our data structures accordingly. We would also like this update time to be sufficiently fast, ideally some constant or amoritized constant time, or some time logarithmic in terms of T.

Storage Space: $\Theta(1)$ or $O(polylog(T))$

Update Time: $\Theta(1)$ or $O(polylog(T))$

Under such strong constraints, there are a lot of problems that we cannot solve. There are fairly simply lower bound arguments that tell us there are certain things you cannot hope to do at all. In these cases, we resort to some kind of approximation. The typical techniques that one would use are some amount of randomization, some amount of sampling, and so on. We will see more as we further explore Streaming Algorithms in this class.

We allow ourselves some randomization, and we allow ourselves some error, but we really want to respect the time and space bounds.

17.3.1 Examples

So what kind of problems might we want to solve here? Ususally one wants to solve problems such as computing statistics on the data, or compressing the data to do calculations on later, and so on. So let's take some specific examples, and see how algorithms can develop for those examples.

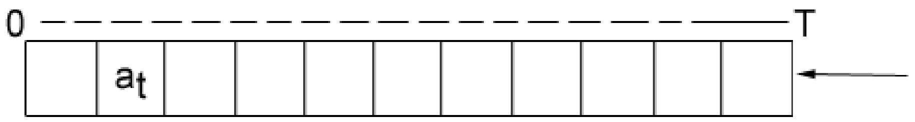

Figure 17.3.6: An example $a_t \in [n]$.

Let us use a more specific model of streaming: At time t you get some element $a_t \in [n] \;\; \forall t \in [T]$. As the stream goes by, we get n different kinds of objects, and we see one element at a time. We are going to think of n as very large, where the different kinds of items we can see are also very large, such as distinct source IP addresses on the internet. There are a lot of source IP addresses, and we cannot use a data structure of size n to store all the data we want about the packets. So again, we would like to use some storage space and some update time logarithmic not only in time of T but also in time of n.

Space: $O(polylog(n))$

Time: $O(polylog(n))$

Of course, one amount of information we get from the stream for all these objects is the frequency of each object. So let m_i denote the frequency of the i^th element of $[n]$, $\forall i \in [n]$ let $m_i = |\{j|a_j = i\}|$.

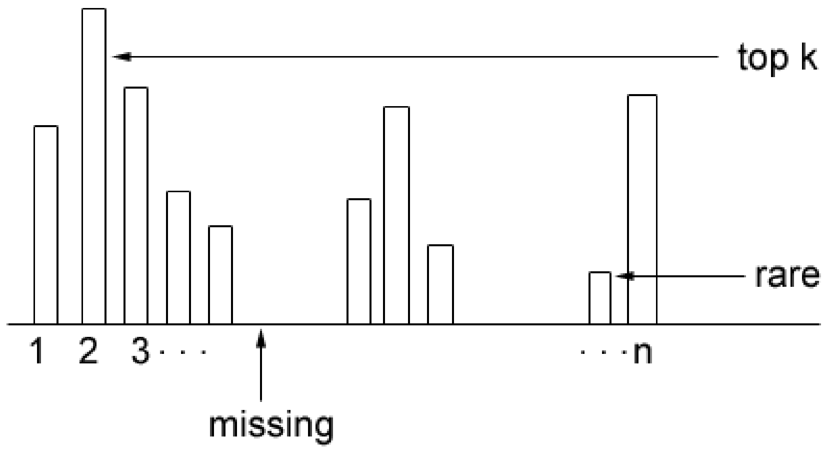

Figure 17.3.7: A sample histogram of frequencies.

So you have these elements with their frequencies, and there are various different kinds of statistics we might want to compute on this data. What are the top k elements, in terms of frequencies? What are the missing elements? What is the number of distinct elements? How many rare elements are there, where rare is defined as occuring at least once, but having a frequency below some number.

We might want to measure how even the distribution is, as measured by expected value and standard deviation, and these are called frequency moments. The k^{th} freqency moment of the distribution $= \sum_i m_i^k$. The zeroth frequency moment, where m_i^0, is equal to the number of distinct elements. The first frequency moment, where m_i^1, is the sum of the m_i's, and is equal to the number of packets. The second frequency moment, where m_i^2, represents the variance. If everything had equal frequencies, than the sum of squares would be small. If the frequency was uneven, where one element occured much more than the rest, than the sum of squares would be large. Higher moments give us some idea of the skews in the distribution. These kinds of things can be used to determine useful information about the stream, such as frequency of outliers, how typical is a particular stream, etc.

What we are concerned with in streaming algorithms are determining good ways of either exactly computing or estimating these quantities. We will see this as we work through some examples.

17.3.2 Number of Distinct Elements

How would you compute the number of distinct elements exactly? Let us momentarily forget about the polylog requirements on space and time, and think about what is the least amount of space we could use to keep track of these elements. We could keep an array of n bits, to keep track of every element. But what if n is really large? What if it is even larger than T, the length of the stream? For example, if the stream were all of the words in a webpage, and you wanted to keep track of the number of distinct words. The number of distinct words could be much larger than the number of words in the webpage.

So if n is larger than T, can we do something on the order of T rather than n? We can use a hash table of size T, and map every element to its position in the hash table, with at most T distinct elements. We can pick a hash table large enough that we never see a collision, by picking a hash table of $O(T)$ or $O(T^2)$ if you want there to be no collisions. Let's define a hash function, $h : [n] \rightarrow T^2$. Whenever we see a_t, compute $h(a_t)$ and update our array, A, at $A[h(a_t)]$. The number of distinct elements in the stream is equal to the number of nonzero elements in A. We can then calculate the exact number of distinct elements in $O(T^2)$ or the approximate number of distinct elements in $O(T)$.

Can we do any better if we want to solve this problem exactly? It turns out that we cannot do any better than $T\ log\ n$.

Theorem 17.3.1 *we cannot computer the number of distinct elements exactly in $o(T\ log\ n)$ space.*

Proof:

Let's assume that the set of elements that we saw was some $X \subseteq [n]$, where X is a uniformly at random subset of $[n]$ of size T. Then, let's assume that we have some algorithm that lets us compute the number of distinct elements in X with storage $o(T\ log\ n)$, then we give one more element from $[n]$ to this particular algorithm, and if that element was not previously in X, then it increments its counter, and if that element was previously in X, then it does not increment its counter. This algorithm lets us solve the question of memebership into X. Then the amount of storage the algorithm uses, exactly represents the set x. But how many bits does it need to represent set X? There are $\binom{n}{T}$ possible different X's that we could have, and for any exact representation of these, we need to have at least log of that many bits. So we need exactly $\Omega(T\ log\ n)$ space. ■

17.4 Next Time

Next time we are going to look at approximate ways of getting very close to the number of distinct elements. As long as we allow ourselves a little bit of randomness, and a little bit of error, we can improve these bounds.