CS640 **Computational Complexity**

Derandomization: The Deathly Hallows

Instructor: Manindra Agrawal *Scribe: Ramprasad Saptharishi*

Last class we revisited MA protocols. And by putting certain restrictions on the size of the proof, number of random bits used by verifier, and the number of probes into the proof, we came up with the definition of MIP. We stated the following theorem:

Theorem 1. $\mathsf{MIP} = \mathsf{NEXP}$

In this case we had exponential sized proofs and polynomial many probes and random bits. We wanted to see if a scale-down can tell us anything about NP. Then we stated the PCP theorem:

Theorem 2. $\mathsf{NP} = \mathsf{PCP}(O(\log n), O(1))$.

Infact the constant number of probes can be made to 3. The philosophical interpretation of this is amazing: some gives you a polynomial sized proof of the statement that $x \in L$ and all you need to do is read 3 bits of this proof to verify whether the proof is right or not with good probability.

This came as a *huge* surprise to complexity theorists and is considered one of the crowning achievements in this field. If one needs a picture of why this could be true, look at the proof as an error-correcting code. So the idea is that, if the proof was incorrect, then it differs from a right proof (if any) at some places. Encode it with an error correcting code. Then, even if there was a single place where it differed from the actual proof, it would now propogate to a lot of places in the encoded word. So finding an error there is a lot easier.

The original proof is quite complex and involves a builds up on a lot of earlier results. A much simpler proof of the PCP theorem was given by Irit Dinur in 2005. We shall be doing her proof in this course.

1 The Deathly Hallows

We shall make a digression and look at randomization again and certain combinatorial objects that can help reduce (or remove) randomness from algorithms.

We shall discuss 3 combinatorial objects:

1. Something that squeezes out "nearly uniform" bits from "partially random" source.

2. Something that reduces our dependence on random bits used in algorithms

3. Something that eliminates randomness all together

One may wonder why we need the second object if we can eliminate the randomness all together. The fact is that we know how to make create the second object but not the third.

1.1 Extractors

This is our first object - to extract "nearly uniform" bits from a "partially random" source. We first need to formally define what we mean by partially random or nearly random distributions.

1.1.1 Some Parameters on Distributions

Let X be a probabilistic distribution over $\{0,1\}^n$. If X were the uniform distribution, then for every string $s \in \{0,1\}^n$, $\Pr[X = s] = 2^{-n}$. But if the distribution was skewed then this need not be true.

We need some way of characterizing how skew distributions are. Suppose the distribution was such that some string like $00\cdots0$ appeared with probability $1/2$, then the distribution is really skewed; most of the time you would sample this string.

Hence, one way to characterize the skewness is to look at the item that gets sampled with highest probability; this is certainly atleast 2^{-n}. If this wasn't too large, then we intuitively see that it's not too skewed. This is captured by the definition of *min-entropy.*

Definition 1. *A distribution X over $\{0,1\}^n$ is said to have min-entropy k if*

$$\max_{s\in\{0,1\}n} \Pr[X = s] \leq \frac{1}{2^k}$$

Note that min-entropy of any distribution over n-bit strings is between 0 and n. If the min-entropy was n then we already have a uniform distribution.

Intuitively, if min-entropy was k, then the distribution has essentially only k truly random bits. Interpret this over the following distribution over

n bit strings. Define the probability as 2^{-k} for all strings that have the last $n-k$ bits as zeroes. In essence, this is just a uniform distribution over the first k bits and padding them up with zeroes.

Thus, any distribution with min-entropy has k truly random bits inside. We are interested in extracting them somehow.

Next is the notion of "nearly uniform." What does it mean to say a distribution is almost random? That is captured by the following distance measure between two distributions.

Definition 2. *For any two distributions X, Y over $\{0,1\}^n$, define the distance between them as*

$$\delta(X,Y) = \frac{1}{2} \sum_{s \in \{0,1\}^n} |\Pr[X=s] - \Pr[Y=s]|$$

If X was indeed equal to Y then the distance is zero. And since

$$\begin{aligned} \frac{1}{2}\sum_s |\Pr[X=s] - \Pr[Y=s]| &\leq \frac{1}{2}\sum_s \Pr[X=s] + \frac{1}{2}\sum_s Pr[Y=s] \\ &= \frac{1}{2} + \frac{1}{2} = 1 \end{aligned}$$

the distance is always a number between 0 and 1. The distance is equal to 1 if the two distributions are orthogonal[1].

And with this definition we can formally define the notion of "nearly random".

Definition 3. *A distribution X is said to be ϵ close to uniform if $\delta(X, U_n) \leq \epsilon$ where U_n is the uniform distribution over n-bit strings.*

Now if you look back at all the randomized algorithms we have done, they assume truly random bits. It can be seen that instead of that, if we were to provide them with random bits that are ϵ close to uniform, then the error probability just goes from $1/3$ to $1/3 + \epsilon$.

With these, we can now define our extractors formally.

Definition 4. *A (k, ϵ) extractor is a polynomial time computable function $E : \{0,1\}^n \times \{0,1\}^t \longrightarrow \{0,1\}^m$ such that for any distribution X over $\{0,1\}^n$ of min-entropy at least k,*

$$\delta\left(E(X, U_t), U_m\right) \leq \epsilon$$

[1] for any string s, either $\Pr[X=s] = 0$ or $\Pr[Y=s] = 0$

So E is an extractor that takes a distribution of reasonable min-entropy, and a random *seed* of t purely random bits and *extracts* out the randomness from X. This is meaningless if t was large. For instance if t was as large as m, then we can just ignore the first parameter of E and just output the uniform bits.

However, the power of this is seen when t is much much smaller than n or m. We shall prove the following theorem later.

Theorem 3. *There exists an (n^δ, ϵ) extractor E where*

$$E : \{0,1\}^n \times \{0,1\}^{O(\log n)} \longrightarrow \{0,1\}^{n^{\delta'}} \qquad \delta' \leq \delta$$

This in essence means that you need your random seed as small as just logarithmic in the m and n. With just that many seeds, you can extract almost all the random bits from X.

It can be shown that the random seed is essential and that to do any reasonable extraction, you need $O(\log n)$ random bits as a seed. Of course, in the next iteration one could use the almost uniform random bits as a seed. But we need $O(\log n)$ random bits to start with.

In a sense, the seed is a catalyst to the reaction, you need it to start the process; without that we won't be able to extract any randomness.

1.2 Expanders

This is the object that will help us reduce the number of random bits used in randomized algorithms. This has a totally different definition compared to extractors; these are graphs.

Definition 5. *A d-regular graph G on n vertices is called an (n, d, ϵ)-vertex expander if for every subset S of size less than $n/2$, the number of edges going out of S is atleast $\epsilon d|S|$. That is, $E(S, V \setminus S) \geq \epsilon d|S|$.*

Since G is a d-regular graph, the number of vertices incident on S is $d|S|$. The definition of expander says that an ϵ fraction of those edges actually go out of S.

1.2.1 Boosting Success Probability using Expanders

Let us take some randomized algorithm we did in class. To see the power of expanders better, let us take an algorithm whose error is one-sided. We saw numerous algorithms where if x was in the language, then the algorithm accepts with probability 1. The error was only when $x \notin L$, we could

incorrectly accept x with some small probability. Let us say this error was $1/3$ to begin with. We want to boost this error.

Our approach was to repeat this algorithm k times and therefore the error probability would come down to $(1/3)^k$. What is the number of random bits used in the process? Suppose a single round used m purely random bits. Then this method of boosting would use km random bits totally.

We shall see how this can be reduced substantially using expander graphs. Look at all the possible random strings you could get. The algorithm uses m random bits, the number of possible random strings is $n = 2^m$. Of these strings, some of them lead the algorithm to the right answer, and some of them don't. Let S be the set of bad strings, those that lead the algorithm to the wrong answer. Since the error is bounded by $1/3$, it follows that $|S| < n/3 < (1/2)n$.

Consider a (n, d, ϵ) expander where each vertex encodes a sequence of m bits. Note that d, ϵ are constants. Now S, the set of bad vertices, is a subset whose size is less than $n/2$ and therefore has the expansion property. Pick a random vertex v. If this vertex was outside S, then we already have a witness. The bad case when we get a vertex inside S. At this point, pick one of the d neighbours of S at random and move to vertex. Note that, by the expansion properties of the graph, an ϵ fraction of the edges go out. Therefore, with prob at most $(1 - \epsilon)$ you will remain in S. Now repeat this picking a random neighbour and moving to that for say k times. Then the probability that this entire walk stayed inside S is about $(1 - \epsilon)^k$. In this way, you boost the error probability to again inverse exponential just like the naive independent trials method.

But what is the total number of random bits used? The first vertex to be picked needed m random bits. After that, to pick a neighbour at random, we just need $\log d$ random bits. And since d is a constant, this is just contant number of random bits. Therefore, the total number of random bits used is $m + O(k)$ which is way better than mk.

Notice that there is a small catch here. How do we construct this expander? The problem is that the size is *huge*, 2^m vertices. How do we even construct such graphs? Notice that you really don't need to store the entire graph (since that would be exponential space and would destroy all hopes of efficient computation). All we need is, given a vertex v, what are the neighbours of v?

This is accomplished by maps called *rotation maps*. These are maps that take u and an integer $i \leq d$ as input and returns the i-th neighbour v of u and a j such that u is the j-th neighbour of i. And fortunately, expanders

can be constructed such that this map is polynomial time computable; we shall see this soon.

And thus, we can reduce the number of random bits used in the boosting.

1.3 Pseudo-random Generators

This is the third object; the one that completely eliminates randomness. Let us first the formal definition of it and then look at the intuition.

Definition 6. *Let $f : \{0,1\}^\star \longrightarrow \{0,1\}^\star$ be a function that satisfies the following properties:*

- *$f(x)$ is computable in $2^{O(|X|)}$ time.*
- *There exists a constant c such that on any string x of size $c \log n$, $f(x)$ is a string of size less than n. In other words, f stretches strings of $O(\log n)$ bits to n bits.*
- *For every circuit C of size n with n inputs,*

$$\left| \Pr_{x \in \{0,1\}^n} [C(x) = 1] - \Pr_{y \in \{0,1\}^{c \log n}} [C(f(y)) = 1] \right| \leq \frac{1}{n}$$

Then f is called an optimal pseudo-random generator against linear sized circuits.

First let us explain the adjectives used here. It is optimal this pseudo-random generator is stretchings of logarithmic size to n bits. It can be shown that there does not exist any pseudo-random generator that stretches some function less than $\log n$ to n bits. And hence, in that sense it is optimal.

It is called a pseudo-random because the output of f is strings of $\{0,1\}^n$ but they aren't really random but appear random to the circuit C. To understand this better, think of C as a randomized algorithm. Suppose its input was fixed and C is just look for the random coin tosses so that it can give the answer.

What we know is that if x is a string in the language, then $\Pr_{x \in \{0,1\}^n}[C(x) = 1] > 2/3$ is atleast and if it was not in the language this is less than $1/3$. Suppose instead of x, we pick up a random string y of size $c \log n$ and give $f(y)$ to C. Then by the third property, we know that $\Pr_{y \in \{0,1\}^{c \log n}}[C(f(y)) = 1]$ is just about $1/n$ away from $1/3$ or $2/3$ and therefore is still good enough. In essense, instead of giving C really random bits, f provides it with bits

that aren't really random but still manages to make C work properly. That is what is meant by pseudo-random generators *against* linear sized circuits.

Another way to look at it is think if C as a distinguisher circuit - tries to distinguish between random and pseudo-random bits provided as input. The third property tells us that C cannot really distinguish between them since the output of C on real random bits and outputs of f isn't too far away from each other. Therefore, the function f is able to *fool* C into thinking that it is actually providing it with truly random bits.

Now notice that f is stretching strings as small as $c \log n$ bits to n bits. Therefore, we can just run over every string $y \in \{0,1\}^{c \log n}$ and find the fraction of points where $C(f(y)) = 1$. We are guarenteed that this is very close to $1/3$ or $2/3$ and therefore we will know for sure how C would behave if provided with truly random bits.

Thus, by running over every possible y, which can be done in polynomial time since f is computable in $2^{O(|x|)}$ time, we can completely remove randomness from the algorithm C. Summarizing this as a theorem:

Theorem 4. *If there exists an optimal, pseudo-random generator against linear sized circuits, then* $\mathsf{BPP} = \mathsf{P}$. □

Unfortunately, we do not know of any way to find such PRGs at the moment. But there are strong evidences to believe that such PRGs exist. And hence, people believe that $\mathsf{BPP} = \mathsf{P}$. However finding a PRG of this kind doesn't mean we can derandomize classes like AM. To remove randomness from higher classes, we need more powerful PRGs.

We shall get back to this after our discussion on expanders and the PCP theorem. We will see that PRGs are strongly connected to extractors.

CS640	**Computational Complexity**
Expanders: Vertex and Spectral Expansion	
Instructor: Manindra Agrawal	*Scribe: Ramprasad Saptharishi*

Last class we saw three very powerful and useful combinatorial objects: extractors, expanders and pseudo-random generators. With the promise that we shall get back to extractors and PRGs in a while, let us go into expander graphs.

2 Two Notions of Expansion

2.1 Vertex Expansion

Last class we defined a notion of expansion. The definition we saw last time is called *vertex expansion*.

A d-regular graph G on n vertices is called an (n, d, ϵ)-vertex expander if for every subset S of size less than $n/2$, the number of edges going out of S is atleast $\epsilon d|S|$. That is, $E(S, V \setminus S) \geq \epsilon d|S|$.

Another definition is what is known as the *spectral expansion*.

2.2 Spectral Expansion

Let G be a n-vertex d-regular graph. Let A be the normalized adjacency matrix, which is $A_{ij} = \frac{1}{d}$ if (i, j) is an edge and 0 otherwise; the usual adjacency matrix divided by d.

The reason we do this is that since G is d-regular, each row has exactly d non-zero entries of value $\frac{1}{d}$ and so does each column. Therefore, each row and column sum up to 1. And further, this is a symmetric matrix.

There is a theorem in linear algebra that states the following:

Theorem 5. *Let A be a symmetric matrix with real entries. Then there exists a basis $\{v_1, v_2, \cdots, v_n\}$ such that each v_i is an eigenvector with eigenvalue λ_i which is real. And further, the basis is orthonormal, that is $v_i \cdot v_j = 0$ if $i \neq j$ and $v_i \cdot v_i = 1$.*

And therefore, our normalized adjacency matrix has a set of orthonormal eigenvectors as a basis. Without loss of generality, we can assume that the eigenvalues are decreasing : $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n$.

Claim 6. $\lambda_1 = 1$.

Proof. Firstly, we need to show that 1 is infact an eigenvalue. That is pretty straightforward to see. Consider the vector of all 1s, which is sometimes referred to as $\bar{1}$.

Note that $A\bar{1} = \bar{1}$ since each row of A adds up to 1. Therefore $\bar{1}$ is an eigenvector corresponding to the eigenvalue 1. We now need to show that all eigenvalues have to be at most 1.

Let λ be any eigenvalue and x be the eigenvector corresponding to λ. Look at the product Ax. Without loss of generality, assume that the first coordinate of x is the largest entry in x. Then we have

$$\begin{aligned} Ax &= \lambda x \\ \implies (Ax)_1 &= \lambda x_1 \end{aligned}$$

Now note that the LHS will merely be the average of some d coordinates of x and the average of d coordinates can never be larger than the maximum value. And hence, $\lambda \leq 1$. Infact one can slightly tweak the argument to show that $|\lambda| \leq 1$.

Therefore all eigenvalues lie between 1 and -1; and 1 is the largest eigenvalue. □

Definition 7. *The second largest eigenvalue of a graph G is the second largest eigenvalue in magnitude of the normalized adjacency matrix A of G.*

$$\lambda_2(G) = \max\{|\lambda_2|, |\lambda_n|\}$$

With this, we can define our spectral expansion.

Definition 8. *A n-vertex d-regular graph G is called a (n, d, λ)-spectral expander if $\lambda = \lambda_2(G)$.*

The use of this notion of expansion is that it allows us to use linear algebra to analyse such graphs and their properties. We shall soon see that the two notions of expansion are essentially equivalent. To easy the notation used there, we shall used *Dirac's Notations* for vectors; it is more intuitive to understand in that setting.

3 Dirac's Notation

Dirac's notation of representing vectors, inner products, outer products is widely used in Physics and expecially in the setting of quantum mechanics

where equations start looking messy with ordinary vectors and transpose etc. Dirac's notation cleans them up and it makes it a lot easier to understand.

- A normal vector v is written $|v\rangle$, pronounced as "ket v", which is the column vector. And the row vector v^T is written as $\langle v|$ and pronounced as "bra v."
- In this setting, note that the usual inner product is just the vector product $v^T v$ which translates to $\langle v| |v\rangle$ which is "bra-ket (bracket) v."
- The outer-product vv^t which gives an $n \times n$ matrix is written as $|v\rangle \langle v|$. This is used less often than the inner product but is very useful. For example, the adjacency matrix E can be expressed as

$$E = \sum_{(u,v) \in \text{Edges}} |u\rangle \langle v|$$

  where $\langle v|$ is the characteristic vector of v that has 1 in the v-th position and 0 everywhere else.

This notation really cleans up a whole mess of arrow, or transposes or dot-products etc. We shall see the power of this notation by using it to prove the connection between vertex and spectral expansion.

4 Vertex Expansion $\Leftrightarrow$ Spectral Expansion

The following theorem shows that the two notions of expansion are essentially the same.

Theorem 7. *Vertex expansion $\Leftrightarrow$ Spectral Expansion*

- *If G was a (n, d, ϵ) vertex expander, then G is also a (n, d, λ) spectral expander where $\lambda \leq 1 - (\epsilon^2/4)$.*
- *If G was a (n, d, λ) spectral expander, then G is also a (n, d, ϵ) vertex expander where $\epsilon \geq (1 - \lambda)/2$.*

If ϵ was large, then λ will be small and vice-versa

We shall prove just the second part of the theorem.

Proof. Let G be a (n, d, λ) expander. Let S be any subset of vertices such that $|S| \leq n/2$. We want to estimate the number of edges going out of S. Let T be the set of vertices that are not in S. We need to estimate $E(S, T)$.

The first step is to write $E(S, T)$ as a product of vectors/matrices. Let u_S be the characteristic vector of S, the vector which has a 1 on exactly those coordinates that belong to S. In Dirac's notation, $|u_S\rangle = \sum_{i \in S} |i\rangle$ and similarly $|u_T\rangle$. Now look at the term $u_S^T A u_T$:

$$\begin{aligned} u_S^T A u_T = \langle u_S | A | u_T \rangle &= \left(\sum_{i \in S} \langle i | \right) A \left(\sum_{j \in T} |j\rangle \right) \\ &= \frac{1}{d} \left(\sum_{i \in S} \langle i | \right) \left(\sum_{(u,v) \in E} |u\rangle \langle v| \right) \left(\sum_{j \in T} |j\rangle \right) \\ &= \frac{1}{d} \sum_{(i,j) \in E(S,T)} \langle i | i \rangle \langle j | j \rangle \qquad \text{(all other terms become } 0\text{)} \\ &= \frac{1}{d} |E(S, T)| \end{aligned}$$

Suppose $\{v_1, v_2, \cdots, v_k\}$ is the orthonormal eigenbasis of A, let

$$\begin{aligned} |u_S\rangle &= \sum_{i=1}^{n} \alpha_i |v_i\rangle \\ |u_T\rangle &= \sum_{i=1}^{n} \beta_i |v_i\rangle \end{aligned}$$

And therefore, we can calculate $\langle u_T | A | u_S \rangle$ as

$$\begin{aligned} \langle u_T | A | u_S \rangle &= \left(\sum_{i=1}^{n} \alpha_i \langle v_i | \right) \left(A(\sum_{j=1}^{n} \beta_j |v_j\rangle) \right) \\ &= \left(\sum_{i=1}^{n} \alpha_i \langle v_i | \right) \left(\sum_{j=1}^{n} \lambda_j \beta_j |v_j\rangle \right) \\ &= \sum_{i=1}^{n} \alpha_i \beta_i \lambda_i \langle v_i | v_i \rangle \qquad \text{(since } i \neq j \implies \langle v_i | v_j \rangle = 0\text{)} \end{aligned}$$

We know that $\lambda_1 = 1$ and that $v_1 = \frac{1}{\sqrt{n}}\bar{1} = \frac{1}{\sqrt{n}}\sum_{i=1}^{n}|i\rangle$. And further,

$$\begin{aligned}
\langle u_S|\,|v_1\rangle &= \sum_{i=1}^{n}\alpha_i\,\langle v_i|\,|v_1\rangle \\
&= \alpha_1 \\
\implies \frac{1}{\sqrt{n}}\sum_{i=1}^{n}\langle u_S|\,|i\rangle &= \frac{|S|}{\sqrt{n}} = \alpha_1 \\
\text{Similarly, } \frac{|T|}{\sqrt{n}} &= \beta_1
\end{aligned}$$

Therefore

$$\begin{aligned}
\langle u_S|\,A\,|u_T\rangle - \frac{|S||T|}{n} &= \sum_{i=2}^{n}\alpha_i\beta_i\lambda_i \\
\left|\langle u_S|\,A\,|u_T\rangle - \frac{|S||T|}{n}\right| &\leq \left|\sum_{i=2}^{n}\alpha_i\beta_i\lambda_i\right| \\
&\leq \sum_{i=2}^{n}|\alpha_i||\beta_i||\lambda_i| \\
&\leq \lambda\sum_{i=2}^{n}|\alpha_i||\beta_i| \\
&\leq \lambda\sqrt{\sum_{i=2}^{n}\alpha_i^2}\sqrt{\sum_{i=2}^{n}\beta_i^2}
\end{aligned}$$

Now let us go back to the definition of $|u_S\rangle$. We know that $\|u_S\|^2 = |S| = \langle u_S|\,|u_S\rangle$. Evaluating this as a linear combination of eigenvectors, we have

$$\begin{aligned}
|S| = \langle u_S|\,|u_S\rangle &= \left(\sum_{i=1}^{n}\alpha_i\,\langle v_i|\right)\left(\sum_{i=1}^{n}\alpha_i\,|v_i\rangle\right) = \sum_{i=1}^{n}\alpha_i^2 \\
\implies |S| - \alpha_1^2 &= \sum_{i=2}^{n}\alpha_i^2 \\
\implies \sum_{i=2}^{n}\alpha_i^2 &= |S| - \frac{|S|^2}{n} = |S|\left(\frac{n-|S|}{n}\right) = \frac{|S||T|}{n} \\
\text{Similarly } \sum_{i=2}^{n}\beta_i^2 &= \frac{|S||T|}{n}
\end{aligned}$$

Hence the above equation becomes

$$\begin{aligned} \left| \langle u_S | A | u_T \rangle - \frac{|S||T|}{n} \right| &\leq \lambda \frac{|S||T|}{n} \\ \implies \left| E(S,T) - \frac{d|S||T|}{n} \right| &\leq d\lambda \frac{|S||T|}{n} \\ \implies E(S,T) &\geq \frac{d|S||T|}{n} - d\lambda \frac{|S||T|}{n} \end{aligned}$$

From the definition of vertex expansion, we have

$$\begin{aligned} \frac{E(S,T)}{d|S|} = \epsilon &\geq \frac{|T|}{n}(1-\lambda) \\ &\geq \frac{1}{2}(1-\lambda) \qquad (\text{ since } |S| \leq n/2 \text{ and therefore } |T| \geq n/2) \end{aligned}$$

Therefore, any graph with spectral expansion λ has vertex expansion of $\frac{1}{2}(1-\lambda)$. □

In essence, both notions of expansion are equivalent. It is easier to work with the spectral definition since it allows us to use a lot of linear algebra techniques to analyze.

CS640	**Computational Complexity**
Random Walks on Expanders	
Instructor: Manindra Agrawal	*Scribe: Ramprasad Saptharishi*

Last class we saw how linear algebra can be used; we saw that spectral expansion and vertex expansion are essentially equivalent.

To illustrate yet another example of how we can use linear algebra to prove theorems on expanders, let us prove the theorem we informally stated in the earlier lecture - amplifying success probability in an RP algorithm.

We stated that if we consider a small subset and took a random walk starting from a vertex in this subset, then the probability that we don't leave this set is exponentially small. We shall prove this formally this class.

5 Random Walks on Expanders Graphs

Theorem 8. *Let S be an arbitrary subset of vertices of a (n, d, λ) expander of size at most δn. Pick a vertex X_1 at random and start a walk of length k from that vertex. Let the walk be $X_1, X_2, \cdots, X_{k+1}$. Then*

$$\Pr[X_1, X_2, \cdots X_{k+1} \in S] \leq ((1-\lambda)\delta + \lambda)^k$$

Proof. We shall express all probabilities, etc in terms of matrix/vector products and analyze them. We first pick a vertex at random and therefore any vertex of the graph can be picked with probability $1/n$. Let us capture this by using a vector $|p_1\rangle = \frac{1}{n}\sum_{i=1}^{n}|i\rangle$ that has a value $1/n$ in every coordinate.

We want the entire walk in the set S and therefore we want the first vertex X_1 also to belong to the set S. In order to capture this, we need to rule out the cases when the vertex does not belong to S. This can be captured by the diagonal matrix, say B, that has a 1s exactly at all (i,i)-th positions when $i \in S$ and the rest of the entries 0. Multiplying any vector with this matrix would preserve just the coordinates corresponding to S and kill all other coordinates to 0. In Dirac's notation

$$B = \sum_{i \in S} |i\rangle \langle i|$$

Hence, if we want X_1 to belong to S, we need to sum up all the coordinates of $B\left|p_1\right\rangle$ which is just the L_1 norm of $B\left|p_1\right\rangle$. Then

$$\Pr[X_1 \in S] = \left|B\left|p_1\right\rangle\right|_1$$

What about the next vertex? Notice that if you start at some probability vector v which represents the current distribution over vertices, the new distribution after choosing a neighbour at random is just Av. And now we need to check if this new vertex is in S and therefore

$$\Pr[X_1, X_2 \in S] = \left|BAB\left|p_1\right\rangle\right|_1$$

And similarly, for a walk of k steps, the final probability would just be

$$\Pr[X_1, X_2, \cdots, X_{k+1} \in S] = \left|(BA)^k B\left|p_1\right\rangle\right|_1$$

It is always useful to convert it into L_2 norms since it is easier to analyze than compared to the L_1 norm. Let $\left|\hat{1}\right\rangle$ be the vector with every coordinate as 1. Then $\left|v\right|_1 = \left\langle\hat{1}\right|\left|v\right\rangle$ and therefore by applying Cauchy Schwarz, we get $\left|v\right|_1 = \left\langle\hat{1}\right|\left|v\right\rangle \leq \sqrt{n}\left\|v\right\|$.

Here is one possible approach: we have a notion of a norm of a vector. What about the norm of a matrix? Thinking of a matrix as a transformation taking one vector to another, can amount by which the matrix "stretches" the vector be used to define the norm of a matrix?

Definition 9. *For an $n \times n$ matrix C, the norm of C is said to be the least $\alpha \geq 0$ such that for every vector $\left|v\right\rangle$, $\left\|C\left|v\right\rangle\right\| \leq \alpha \left\|\left|v\right\rangle\right\|$.*

Thus we could use this definition to get an upper bound on $\left\|(BA)^k B\left|p_1\right\rangle\right\|$ by eliminating one matrix at a time. Let us try and see what the norm of B and A are.

The norm of B is 1 since if you give it a vector that has non-zero entries at exactly those coordinates corresponding to S, then B would just preserve the vector. Therefore, the norm of B is 1.

What about A? Unfortunately, the norm of A is 1 as well. Thinking of vectors expressed as linear combinations of the eigenbasis, notice that except the direction of $\left|v_1\right\rangle$, A shrinks every other direction since $\left|\lambda_i\right| \leq 1$. But since A preserves $\left|v_1\right\rangle$, the norm of A is 1 as well.

But then this would give us something as trivial as $Pr[X_1, X_2, \cdots, X_k \in S] \leq 1$. But the point is that we learn something important from this attempt:

- The matrix B kills all components of the vector in directions outside S.
- The matrix A shrinks all but the component along $\left|\hat{1}\right\rangle$ by at least λ.

So instead, let us look at the norm of the matrix BA. At this point we shall use a nice trick. Write the matrix $A = (1-\lambda)J + \lambda C$, where J the $n \times n$ matrix with every entry being $1/n$. One could express J as $\frac{1}{n}\left\langle\hat{1}\right|\left|\hat{1}\right\rangle$ in the dirac notation. We would be using the following two important properties of the norm.

Observation 9. $\|P+Q\| \leq \|P\| + \|Q\|$ *and* $\|PQ\| \leq \|P\|\,\|Q\|$

Proof. Suppose $\|P\| = \alpha$ and $\|Q\| = \beta$. Then:

$$\begin{aligned} \|(P+Q)\left|v\right\rangle\| &= \|P\left|v\right\rangle + Q\left|v\right\rangle\| \\ &\leq \|P\left|v\right\rangle\| + \|Q\left|v\right\rangle\| \\ &\leq \alpha\,\||v\rangle\| + \beta\,\||v\rangle\| \\ &\leq (\alpha+\beta)\,\||v\rangle\| \end{aligned}$$

The other property is pretty straight forward as well. □

Using this, we have $\|BA\| \leq (1-\lambda)\,\|BJ\| + \lambda\,\|BC\|$. Note that for any vector $|v\rangle$, $J\left|v\right\rangle$ will just have every coordinate as the average of all entries of v. Let us say the average of all entries of v was t. Then $BJv = B(t\left|\hat{1}\right\rangle)$. Now B would kill all but the coordinates that belong to S and therefore $\|B\left|v\right\rangle\| = t\sqrt{|S|}$. Therefore

$$\|BJ\left|v\right\rangle\| = t\sqrt{S} = t\sqrt{\delta n} = \frac{||v\rangle|_1}{n}\sqrt{\delta n} \leq \frac{\sqrt{n}\,\||v\rangle\|}{n}\sqrt{\delta n} = \sqrt{\delta}\,\||v\rangle\|$$

We need to get a bound on $\|BC\|$ and we would be done.

Claim 10. $\|C\| \leq 1$.

Proof. By our definition, $C = \frac{1}{\lambda}\left(A - (1-\lambda)J\right)$. Let $|v\rangle$ be any vector. Write $|v\rangle = |v'\rangle + |w\rangle$ where $|v'\rangle$ is parallel to $\left|\hat{1}\right\rangle$ and $|w\rangle$ is orthogonal to it.

$$C\left|v'\right\rangle = \frac{1}{\lambda}A\left|v'\right\rangle - \frac{1-\lambda}{\lambda}J\left|v'\right\rangle = \frac{1}{\lambda}\left|v'\right\rangle - \frac{1-\lambda}{\lambda}\left|v'\right\rangle = \left|v'\right\rangle$$

As for the orthogonal component, notice that $J\left|w\right\rangle = 0$ since w is orthogonal to $\left|\hat{1}\right\rangle$. Therefore,

$$C\left|w\right\rangle = \frac{1}{\lambda}\left(A\left|w\right\rangle - (1-\lambda)J\left|w\right\rangle\right) = \frac{1}{\lambda}A\left|w\right\rangle$$

Taking norms,

$$\|C\,|w\rangle\| = \frac{1}{\lambda}\,\|A\,|w\rangle\| \leq \||w\rangle\|$$

since $|w\rangle$ is orthogonal to $\left|\hat{1}\right\rangle$. Also note that the vector $C\,|w\rangle$ will be a vector in the orthogonal complement of $\left|\hat{1}\right\rangle$.

$$\begin{aligned}\|C\,|v\rangle\|^2 &= \left\|C\left|v'\right\rangle\right\|^2 + \|C\,|w\rangle\|^2 \\ &\leq \left\|\left|v'\right\rangle\right\|^2 + \||w\rangle\|^2\end{aligned}$$

And therefore, $\|C\| \leq 1$. It is infact equal to 1 since the second eigenvector $|v_2\rangle$ is the candidate. □

And therefore,

$$\begin{aligned}\|BA\| &\leq (1-\lambda)\,\|BJ\| + \lambda\,\|BC\| \\ &\leq (1-\lambda)\sqrt{\delta} + \lambda \\ \implies \left\|(BA)^k B\,|p_1\rangle\right\| &\leq \left((1-\lambda)\sqrt{\delta} + \lambda\right)^k \|B\,|p_1\rangle\| \\ &= \left((1-\lambda)\sqrt{\delta} + \lambda\right)^k \sqrt{\frac{|S|}{n^2}} \\ &= \left((1-\lambda)\sqrt{\delta} + \lambda\right)^k \sqrt{\frac{\delta}{n}} \\ \implies \left|(BA)^k B\,|p_1\rangle\right|_1 &\leq \sqrt{n}\cdot\left((1-\lambda)\sqrt{\delta} + \lambda\right)^k \sqrt{\frac{\delta}{n}} \\ &= \left((1-\lambda)\sqrt{\delta} + \lambda\right)^k \sqrt{\delta} \\ &\leq \left((1-\lambda)\delta + \lambda\right)^k\end{aligned}$$

And that proves the theorem. □

6 Amplifying Success Probability in Randomized Algorithms

We did make a passing remark on how expanders can be used to reduce the number of random bits used in the naive boosting technique to boost the success probability.

6.1 Expanders in RP algorithms

Let us look at any RP algorithm; probabilistic algorithm with error only on one side. When x was in the language, then your algorithm always answers yes on all random strings. The only case when you could be fooled is when x is not in the language but your random choice also says yes. The naive approach is to repeat this say k times and hope that you get at least one random choice that says no.

Suppose the algorithm had error probability of say δ initially and used m random bits. Then after k tries, the error probability as δ^k and the total number of random bits would be mk. We want to make this more efficient.

The idea is to interpret each random m-bit string as a vertex in an expander graph. Consider an $(2^m, d, \lambda)$ expander and each vertex encodes a random string. You pick a vertex at random in the graph. That corresponds to some string. Run that algorithm on this string. Then pick a random neighbour of this vertex and move to this vertex. Use the string corresponding to this vertex as your new random string and do the algorithm. Repeat this k times by making a walk of length k on the expander.

Let S be set of random strings on which your algorithm errs. Then since the error probability was δ to start with, $|S| = \delta n$ and we can apply the above theorem. Then we get the error down to exponential in k by a k-step random walk.

The total number of random bits is m to get the first vertex, and then $\log d$ random bits to pick a random neighbour everytime. Therefore, the total number of random bits is just $O(m+k \log d)$ and d is usually a constant and therefore $O(m + k)$ which is way better than $O(mk)$.

6.2 Expanders in BPP algorithm

In order to reduce the number of random bits in a BPP algorithm we need a stronger theorem on random walks to apply for BPP. We won't be doing the proof of this; it is far more complex than the one we proved above.

Theorem 11. *Let S be a subset of vertices of an (n, d, λ) expander graph such that $|S| \leq \delta n$. Let $X_1, X_2, \cdots, X_{k+1}$ be a random walk on this expander graph. Then*

$$\Pr\left[\left|\frac{\#\{i \ : \ X_i \in S\}}{k} - \delta\right| > \epsilon\right] \leq 2e^{-\frac{(1-\lambda)\epsilon^2\delta}{60}}$$

This essentially means that if you start with say an error probability of $1/4$ initially, then essentially after a k step random walk, the number of

times you revisit the set of bad vertices is very close to δk. Therefore, if you started with $1/4$, your final fractino will also be very close to $1/4$ and therefore, the majority vote will will give you the right answer with prob $2^{-O(k)}$.

Thus, with $O(m+k)$ random bits, you can amplify the success probability in a BPP algorithm as well.

CS640	**Computational Complexity**
Constructing Expanders: The Zig-Zag Product	
Instructor: Manindra Agrawal	*Scribe: Ramprasad Saptharishi*

Over the last few classes we were introduced to expander graphs and we saw how they could be used to reduce the number of random bits required for amplification.

The crucial part in that is that, given a vertex v of the expander, we should be able to pick a random neighbour of that quickly. Though the graph is of huge size (2^m vertices), we want our neighbour computation to be fast. This is captured by *rotation maps.*

7 Rotation Maps

Definition 10. *Let G be a d-regular n vertex graph. The rotation map, denoted by Rot_G, is a map $Rot_G : V \times \{1, 2, \cdots, d\} \longrightarrow V \times \{1, 2, \cdots d\}$ such that $Rot_G(u, i) = (v, j)$ if the i-th neighbour of u is vertex v and the j-th neighbour of v is u.*

From the definition it is clear that this is a permutation on $V \times \{1, 2, \cdots, d\}$ and it is also an involution; that is Rot_G applied twice successively is the identity. Therefore, this map can be represented as a symmetric permutation matrix of dimension $nd \times nd$.

We would want our expander graphs to have rotation maps that are computable in time $\mathsf{poly}(\log n, \log d)$.

8 Constructing Expander Graphs

Expander graphs are available in plenty. Infact, the following result states that almost all graphs are expanders

Theorem 12. *A random d-regular graph on n vertices has its second largest eigenvalue less than $9/10$ with high probability.*

Therefore, the existence of such graphs is clearly not an issue. But we would want to explicitly construct them.

A few such constructions were discovered earlier and all of them had a similar flavour - simple to visualize but really hard to show that they are indeed good expanders or even find efficient rotation maps. To illustrate that, consider the following graph. Let p be a prime and consider a graph on p vertices. For each i, connect it to $i-1, i+1$ and $i^{-1} \bmod p$ (put a self loop on 0). This actually forms an expander graph!

People were looking for explicit constructions of a family of constant degree expander graphs for a long time. The general idea was to take a small expander graph, and somehow blow it up or enlarge it to give a larger expander graph. We shall now look at certain *graph products.*

9 Graph Products

We shall look at some ways by which we can take two graphs G and H and try and get a bigger graph by taking some product between them.

9.1 Powering

Suppose G and H are (n, d, λ) and (n, d', λ') expanders. Define the adjacency matrix of the product graph GH as the product of the adjacency matrices. This may not be a $0, 1$ matrix but think of this as a multigraph where $A_{ij} = k$ means that there are k edges between vertex i and j.

The resulting graph will also be an n vertex graph and have degree dd'. What about the eigenvalue?

Notice that this new normalized adjacency matrix also has $\left|\hat{1}\right\rangle$ as an eigenvector with eigenvalue 1. Take any other eigenvector orthogonal to this. Then the adjacency matrix of H shrinks it by λ' and then G shrinks it by λ. Therefore, the second largest eigenvalue is $\lambda\lambda'$.

Therefore the resulting graph GH is an $(n, dd', \lambda\lambda')$ expander. We shall denote a product of G with itself k times as G^k.

9.2 Tensor Product

Let G be an (n, d, λ) expander and H be a (n', d', λ') expander. The tensor product $G \otimes H$ is defined as follows:

- The vertex set of $G \otimes H$ is $V_G \times V_H$. Vertices can be thought of as (u, u') such that $u \in G$ and $u' \in H$.
- Vertex (u, u') is connected to (v, v') if and only if $(u, v) \in G$ and $(u', v') \in H$.

It is clear from the definition that the number of vertices is nn' and the degree is dd'. And it is not hard to check that the adjacency matrix of this new graph will now be the tensor product of the adjacency matrices of the old graphs. And therefore, the related eigenvalues and eigenvectors are also corresponding products. Therefore, the second largest eigenvalue of $G \otimes H$ is $\max\{\lambda, \lambda'\}$.

Therefore, the resulting graph $G \otimes H$ is an $(nn', dd', \max\{\lambda, \lambda'\})$ expander.

Let us now look at what the two products give us. The powering is good in the sense that it reduces λ, which is good for us. The number of vertices however remain the same. The tensor product on the other hand increases the number of vertices and doesn't change the eigenvalue much since it remains the max of the 2 graphs and is therefore under control.

The problem with both the operations is that the degree of the graph keeps increasing. And with just these two products there is no hope of getting a family of constant degree expanders.

The degree blows up because of the freedom that both the products allows. In case of powering, if you look at GH, it is equivalent to taking one edge in H and then one in G. So in essence you have complete freedom of choosing any edge in G as long as it is incident in the vertex you are on after the H-edge. In case of tensoring, they are essentially two parallel moves, one on G and one on H and therefore allows tremendous freedom.

What we need to do is reduce this freedom; somehow make the edge in one of the graphs "influence" the other edge. This is exactly what happens in the *zig zag* product which finally solves the problem of degree blow-up.

10 The Zig-Zag Product

Suppose G is an (N, D, λ_1) expander and H is an (D, d, λ_2) expander. We can then define what the zig-zag product of the two graphs are. It is denoted by $G ⓩ H$.

We shall first define it informally so that the readers get a good picture of what it is. Take the graph G. Each vertex u has degree D and therefore has D edges going out. Now replace each vertex u by a *cloud* of D vertices such that each new vertex u_i represents the i-th edge going out of u. Therefore, the vertex set of $G ⓩ H$ is of size ND since each vertex in G is now blown up by a D vertex cloud. To make the understanding, we shall think of the new vertices as pairs (u, i) which corresponds to the i-th vertex in the

cloud of u. We hence have ND vertices, identified as clouds, present with absolutely no edges between them. We need to now define the edges.

Any edge in the zig-zag product will correspond to a 3-step walk which is as follows. You start at vertex (u, i) which is the i-th vertex in the cloud of u. Now think of the vertices in this cloud as vertices of H. Take one edge in H to go from (u, i) to (u, j) where j is a neighbour of i in H. Now you are at vertex (u, j). The vertex j corresponds to the j-th edge going out of u; take that edge in G to go to vertex v in G and thereby going from a vertex in cloud u to a vertex in cloud v. Now we would go from u to v through some edge of v, say the k-th edge. This corresponds to (v, k) in the cloud. So we end up there. Now think of the cloud of v as the graph H and go to some neighbour of k, say l. Therefore, you finally end up in (v, l). This three step walk defines a single edge in $G(\mathbb{Z})H$; you then connect (u, i) and (v, l) by an edge.

OK, let us first find out what the degree of the new graph is. How many neighbours does (u, i) have? Let us see what the freedom is. From (u, i) we can take any edge in H for the first step of the walk; that gives us d choices. Then the new vertex defines the intercloud edges so there is no choice there. After that, we take one more edge in H in the new cloud which again gives us d choices. Therefore, the degree of this new graph is d^2.

Infact we can say that (u, i)'s (a, b)-th neighbour is (v, k) if your first walk took the a-th neighbour of a and then in the last step too the b-th neighbour to k. Therefore, the edge labels can also be thought of as tuples (a, b) where $1 \leq a, b \leq d$.

Let us now formally define the edges. Recall that the rotation map $\text{Rot}_{G(\mathbb{Z})H}$ is a map that takes a vertex and edge number and returns the destination vertex and its edge number that we took to get there.

To compute $\text{Rot}_{G(\mathbb{Z})H}((u, i), (a, b))$, let $\text{Rot}_H(i, a) = (j, a')$ and let $\text{Rot}_G(u, j) = (v, j')$ and then let $\text{Rot}_G(j, b) = (k, b')$. Then, define $\text{Rot}_{G(\mathbb{Z})H}((u, i), (a, b))$ as $((v, k), (b', a'))$.

This rotation map therefore defines all the edges of $G(\mathbb{Z})H$. Now for the bounds on eigenvalues.

10.1 Eigenvalue Bounds

We now need to argue that he second largest eigenvalue isn't too large. Note that we don't need the eigenvalue to actually decrease. The graph powering reduces the eigenvalue a lot. Therefore if we reduce it enough

through powering and then use the zig-zag product to get the degree down and a slight increase in eigenvalue will not cause too much of trouble.

The following theorem gives a bound on the eigenvalue of the new graph.

Theorem 13. *The graph $G(z)H$ is an (ND, d^2, λ) expander where $\lambda = 1 - (1 - \lambda_1)(1 - \lambda_2)^2$.*

Proof. The adjacency matrix of $G(z)H$ isn't as difficult as it seems. Every edge in the new graph corresponds to a 3 step walk. You start at (u, i) and go to a neighbour j of i in cloud u and therefore to (u, j). This is like the adjacency matrix of H acting on the second coordinate and keeping the first coordinate fixed; this is what is captured by the matrix $I \otimes H$ where H is the normalized adjacency matrix of the graph H.

From (u, j), you go the j-th neighbour, say v, of u. You would be taking v's k-th edge into v and hence you end up at (v, k). This means that u's j-th neighbour is v and as a k-th neighbour of v. This is exactly the rotation map of G, takes (u, j) and returns (v, k).. Let us call the matrix representing the rotation map as R_G. Then this matrix captures the second step.

The third step is again keeping the first coordinate fixed and the second coordinate changing based on H. Therefore, the third step is $I \otimes H$ as well. Hence, the normalized adjacency of $G(z)H$ is $(I \otimes H)R_G(I \otimes H)$.

We need to compute the 2nd largest eigenvalue of $Z = (I \otimes H)R_G(I \otimes H)$. Write $H = (1 - \lambda_2)J + \lambda_2 C$ where J is the vector with all $1/D$s.

$$\begin{aligned} Z &= (I \otimes H)R_G(I \otimes H) \\ &= (I \otimes ((1 - \lambda_2)J + \lambda_2 C))R_G(I \otimes ((1 - \lambda_2)J + \lambda_2 C)) \\ &= (1 - \lambda_2)^2(I \otimes J)R_G(I \otimes J) + (1 - \lambda_2)\lambda_2(I \otimes C)R_G(I \otimes J) \\ &\quad +(1 - \lambda_2)\lambda_2(I \otimes J)R_G(I \otimes C) + \lambda_2(I \otimes C)R_G(I \otimes C) \\ &= (1 - \lambda_2)^2(I \otimes J)R_G(I \otimes J) + (1 - (1 - \lambda_2))^2 E \end{aligned}$$

where E is the rest of the matrices. After this, it's just some verification using the eigenvectors. It can be shown that $\|E\| \leq 1$. We leave it to the reader to think about it.

As for the other term, look at the matrix $(I \otimes J)R_G(I \otimes J)$. Thinking of it graphically, the first $(I \otimes J)$ preserves the first coordinate and sends the second coordinate to "any" vertex in H. Then R_G then goes to "any" neighbour of G and then the final $I \otimes J$ fixes the first component again. Hence, it can be formally shown as well, $(I \otimes J)R_G(I \otimes J) = G \otimes J$.

Now the only eigenvalues of J are 1 and 0 (0 has multiplicity $D-1$) and therefore the second largest eigenvalue of J is 0. Therefore, the second largest eigenvalue of $G \otimes J$ is λ_1.

Therefore,

$$\lambda \leq (1-\lambda_2)^2\lambda_1 + 1 - (1-\lambda_2)^2 = 1 - (1-\lambda_1)(1-\lambda_2)^2$$

□

10.2 Revisiting Probability Amplification

Recall the part when we discussed amplifying the success probability in an RP algorithm. We had an algorithm that used m random bits and whose error was less than say δ. We used $m + k\log D$ bits to get the error down to $2^{-\Omega k}$. Think of this as a first block of m bits to figure out the first vertex, and then k chunks of $\log D$ bits. In this k chunks of $\log D$ bits, you can recursively apply the procedure. Think of $\log D = m'$ and this looks like k repeated applications of the naive algorithm that uses m' random bits. Therefore, you can think of an expander on $2^{m'} = D$ vertices say of degree d and do a random walk there to reduce the random bits again.

Therefore we would now be using $\log m + \log D + \log d + \log d + \cdots$. One could group the first two chunks to get a chunk of $\log mD$ random bits and then followed by a lot of $\log d$ chunks. This is like a random walk on an expander on mD vertices and degree $2^{O(\log d)}$ which say is d^2. And this is essentially what we get in the zig-zag product!

Thus, we can think of the probability amplification in two steps: start with an expander on 2^m vertices but not necessarily of constant degree D. Now use the smaller expander on D vertices to reduce the random bits even further. This corresponds to using the expander which is the zigzag product of the two expanders.

11 Efficient Computable Family of Constant Degree Expanders

Using the 3 graph products that we discussed, we can now go ahead and construct a family of efficiently computable expander graphs with constant degree. To start with, let us pick a small constant sized graph with $\lambda < 1/2$. This can be picked by bruteforce.

Let H be a $(d^8, d, 1/2)$ expander. Define $G_1 = HH$. Therefore, G_1 will be an d^{16} vertex d^2 regular graph. Now interatively, define G_k for $k \geq 2$ as

$$G_k = (G_{k-1} \otimes G_{k-1})^2 ⓩ H$$

The following claim is very straightforward to check.

Claim 14. *The family of graphs $\{G_i\}$ form a family of expander graphs each of degree d^2. And G_k is a $(8(2^k - 1), d^2, 1/4)$ expander. And further, the rotation maps of G_k can be computed in time $O(k)$ which is logarithmic in the size of the vertex set of the graph and is hence efficient.*

CS640	**Computational Complexity**
Towards the Proof of the PCP Theorem	
Instructor: Manindra Agrawal	*Scribe: Ramprasad Saptharishi*

Last class we saw completed our discussion on expander graphs. We shall now go to the proof of the PCP theorem. We need to prove that $\mathsf{PCP}(O(\log n), O(1)) = \mathsf{NP}$.

One direction, $\mathsf{PCP}(O(\log n), O(1)) \subseteq \mathsf{NP}$, is trivial. If there exists such a PCP protocol, then the nondeterministic machine can just guess the proof of the prover. And since the number of random bits used is just $O(\log n)$, he can then run over all the random choices. And further, since the number of probes is constantly many, the total number of probes is $n^{O(1)}$ which is polynomially many again. Therefore, $\mathsf{PCP}(O(\log n), O(1)) \subseteq \mathsf{NP}$.

It is the other direction that uses a lot of weapons.

12 Problems with Naive Approaches

Let us take $3-\mathsf{SAT}$ for example. We are given a formula φ say on n variables and has m clauses, and the prover should convince the verifier that the formula is indeed satisfiable. Here is one possible PCP protocol:

1. The prover provides the satisfying assignment for the formula.
2. The verifier tosses his random coins to pick one clause of the formula and probes the values of the three literals in that clause from the "satisfying assignment" provided by the prover.
3. Using the values of those three literals, the verifier checks if that particular clause is true by the assignment. If it is true, he accepts. Else, he rejects.

Thus, the verifier just probes 3 places of the proof and uses $O(\log m)$ random bits. One thing is clear that if the formula was indeed satisfiable, then there does exist an honest prover who would provide a correct satisfying assignment and thus the verifier would accept with probability 1. The other direction, however, does not hold.

Let us say that the formula φ was not satisfiable. How can the prover cheat? Suppose the formula was in such a way that there exists a assignment that satisfies all but just 1 clause. Then the prover could provide just

that assignment. So unless the verifier is lucky and picks that unsatisfied clause, the verifier would end up accepting. Thus, we can only say that the probability that verifier rejects is bounded above by $1 - \frac{1}{m}$. This isn't good enough, we want to bound this probability by a constant.

Can't we amplify this probability by more tries? Do we have more problems? Let us say instead of choosing 1 clause at random, the verifier choose k clauses. Then that would reduce the bound to roughly $(1 - \frac{1}{m})^k$. Thus, if we choose $k = m$, we would have bounded the probability by $1/e$ which is what we want. What are the issues?

Total Random Bits Used: We need to repeat this event of picking a clause m times and each of the times we would be needing $O(\log m)$ random bits. Therefore, the total number of random bits used is $O(m \log m)$ which is too much. Can we do better?

One possible solution is to use expander random walks as we discussed earlier. Since we are repeating an experiment for a lot of times, we could a constant degree expander to do the trick. But that again would reduce the total number of random bits to $O(m + \log m)$ which is still bad. In fact, on analysing carefully, the expander can't even be used here. Remember that the random walks would be good only if you start with your bad set being relatively small. Here, you are starting with your bad clauses being a $1 - \frac{1}{m}$ and thus expander random walks won't help in locating the unsatisfied clause.

Total Number of Probes: This is a bigger issue, repeating experiments m times will force us to have $O(m)$ probes. This is certainly not acceptable, as this is as good as reading the entire proof.

Thus what we need is essentially that the formula we get should satisfy the property that either it is satisfiable or that any assignment makes a constant fraction of the clauses unsatisfied. This notion is formalized by looking at the *gap* in a problem.

13 The Notion of Gap and Dinur's Theorem

We shall look at two different problems and the notion of gap associated with it. Let us take 3SAT first. A 3SAT formula φ is said to have gap δ if any assignment for the formula leaves at least a δ fraction of the clauses unsatisfied. We shall denote this by $gap(\varphi)$.

An immediate consequence is that if we started of with a formula with

constant gap, then the verifier would be able to catch the prover even if he tries to cheat since any assignment leaves a constant fraction of clauses unsatisfied.

Instead of looking at 3SAT we would be looking at a different NP-complete problem called the constraint graph problem.

13.1 Constraint Graphs

Definition 11. *An instance to G_k, is a tuple of the form (V, E, Φ) where V is the set of vertices, E the edges and Φ is a set of constraints. One could think of $\Phi = \{\phi_1, \phi_2, \cdots, \phi_{|E|}\}$, one for each edge. And a constraint $\phi_e : [1..k] \times [1..k] \longrightarrow \{0, 1\}$. Essentially, it takes two possible colours for the end points of that edge and says if that pair satisfies that constraint or not.*

An instance (V, E, Φ) is the language if there exists a colour assignment $\pi : V \longrightarrow [1..k]$ such that it satisfies all the edge constraints.

A simple observation is that the constraint graph problem is NP-complete for $k \geq 3$.

Lemma 15. *The problem G_k is NP-complete for $k \geq 3$.*

Proof. Clearly it suffices to show that it is NP-complete for $k = 3$. It is obvious that this is infact in NP since the machine can just guess the satisfying colouring and check.

And showing it is NP-hard is simple as well as graph 3-colourability directly reduces to G_3 where each constraint on the edges is just inequality. □

In the constraint graph problem, we could have a similar notion of gap as well. An instance of G_k is said to have gap δ if any assignment leaves at least a δ fraction of the constraints unsatisfied. Again, if the instance to the PCP protocol was indeed an instance with constant gap, the verifier can make reject the prover if he tries to cheat with good probability. He just picks an edge at random ($O(\log m)$ random bits), looks at the colour of the two end points (therefore makes $2 \log k = O(1)$) probes and then answers according to whether the assignment satisfied that constraint or not.

Dinur's theorem was the following:

Theorem 16 (Irit Dinur)**.** *There is a polynomial time reduction f from G_{16} to G_{16} such that*

- *If the instance G was satisfiable, then so is $f(G)$.*
- *If the instance G was not satisfiable, then $f(G)$ has gap which is a constant less than 1.*

And a simple observation:

Observation 17. *Dinur's Theorem* $\implies$ PCP *Theorem*

Our goal is to prove Dinur's Theorem and we shall do it through 2 lemmas. Before that, we need a notion of a reduction of amplifying the gap. We shall first define what that reduction is and then discuss why we need those properties.

Definition 12. *A polynomial time function f is said to be a (k, δ) reduction from G_k to $G_{k'}$ if*

- *$|f(x)| \leq c|x|$ for some constant c.*
- *If $gap(x) = 0$, then $gap(f(x)) = 0$.*
- *If $gap(x) \neq 0$, then $gap(f(x)) \geq \min\{k \cdot gap(x), \delta\}$.*

This essentially is means the following, the reduction doesn't increase the size of the instance too much, but increases the gap by a factor of k, unless the gap was already larger than δ. Why do we want the size to be linearly bounded? This is because we would be doing this reduction for logarithmically many steps and therefore unless the size increase was linearly bounded, the final formula size could be huge.

The following two lemmas directly imply Dinur's Theorem.

Lemma 18 (Amplification)**.** *There exists a constant δ such that for every t, there exists a k and a (t, δ) reduction from G_{16} to G_k.*

But this could (and infact will) increase the alphabet size from 16 to some arbitrary number k which could be like 16^t, and thus repeating the process might blow up the alphabet size too much. Thus, we need a lemma to reduce back the alphabet size.

Lemma 19 (Alphabet Reduction)**.** *There exists a constant ϵ such that there exists an $(\epsilon, 1)$ reduction from G_k to G_{16}.*

Thus what we would do is first apply the amplification lemma to amplify gap by a factor of t. Then we apply the alphabet reduction lemma to reduce the alphabet size back to 16 but in the process we would lose a factor

of ϵ but we would choose our t in such a way that the net process doubles the gap. Thus, $t = 2/\epsilon$ so that at the end we have a linear blow up in the instance with the gap doubled. And clearly, by repeating this for logarithmically many steps, we get to a point where the gap is constant. And that would prove Dinur's Theorem and hence the PCPTheorem.

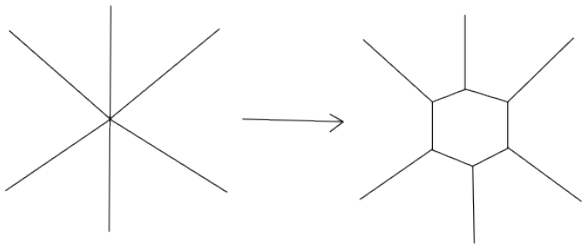
14 Steps for the Amplification Lemma

We shall take an instance of G_k and amplify the gap in 3 steps. We would lose a constant factor on the gap in the first two steps but we can make up for the loses through the third step.

1. Make the graph a constant degree graph.
2. *Expanderize* the graph.
3. Do some clever walks on the expanderized graph to amplify the gap, and increase the alphabet size.

We shall look at one possible approach for step 1. Firstly, we can assume that every vertex of the graph has degree at least 3. Otherwise we can just add some redundent vertices and edges with trivial constraints that are always satisfied no matter what the colouring is.

Now we take every vertex v of this graph and blow it up into $deg(v)$ many vertices just like in the zig-zag product construction. And we connect these $deg(v)$ many vertices by a cycle and put equality constraints on them. The other constraints get naturally transformed here.



Does this give us just a constant factor reduction on the gap? One way to argue this is the following. Take any colouring in this new graph. We want to map this to a colouring of the old graph. What we do is, in every cloud of $deg(v)$ many vertices, take the most popular colour there and give that as a colour to the vertex v in the old graph. If we can somehow argue that since the old graph has a reasonable graph, the new graph cannot have

a very small gap, we are done.

But unfortunately this isn't the case since lots of the equality constraints on the cycle we added could be violated to satisfy the edge constraints of the original graph. The reader is recommended to try to work out a bound and see why putting a cycle isn't a good idea.

The solution is something we have seen earlier as well: do not put a cycle; instead put an expander graph there.

14.1 Closing the Cloud with an Expander

Instead of connected the vertices of a cloud with a cycle, we shall connect them with an expander. The intuition is that since expanders are well connected, voilated constraints will see its effect in a lot of edges.

To argue that the gap just reduces by a constant factor, we shall take any colouring of the new graph G'. In the new graph, each cloud correponding to a vertex v in G would give different colours to the vertices. We shall give v the most popular colour. Lets say that by giving the most popular colour, some k intercloud edges that were satisfied in G' now get violated. For each such edge, one of the two end points of the edge in G' do not get the most popular colour (for if they did, then the edge would remain satisfied). Let us pick one vertex for each new violated edge and call this set of vertices as T. Look at a vertex $v \in G$ and the cloud of vertices in G'. Pick a colour that is not the most popular colour and look at the subset S of all vertices in T present in this cloud with that colour. Since we picked a colour that was not the most popular, we are guaranteed that $|S| \leq m/2$ where $m = \deg(v)$. And hence, by the property of the expander, the total number of edges going out of S is at least $d\epsilon|S|$. And since we put equality constraints over all of them, all of these edges will be violated since they are going between different colours.

Summing over all non-popular colours and all clouds, we get the total number of edges to be at least $d\epsilon|T|$. But there is a slight trouble here since we may be counting edges twice. Nevertheless we can still claim that the number of edges violated within the clouds is at least $d\epsilon|T|/2 = d\epsilon k/2$

Thus, the total number of edges violated in G' should be at least

$$\text{gap}(G)|E| - k + d\epsilon k/2 = \text{gap}(G)|E| + (\epsilon d - 2)k/2$$

and in the construction of the expander graphs, we could make $\epsilon d - 2 > 0$ (this is where the cycle fails but the expander succeeds). Therefore, the number of unsatisfied edges in G' is at least $\text{gap}(G)|E|$

Hence, dividing by the number of edges in G', we get

$$\text{gap}(G') \geq \frac{\text{gap}(G)}{2d+1}$$

14.2 Expanderising the Graph

The next step is to make it an expander. This is done by a very nice and simple trick: just super impose an expander on this graph. All we do is think of these n vertices as vertices of an expander and just add the new edges. As for the constraints on the new edges, we just make them trivial constraints: always satisfied.

The increase in the number of edges brings down the gap slightly. To be precise, if we started with a d-regular graph and superimposed an (n, d', ϵ) expander on it, the new number of edges is $n(d+d')$ and therefore, the new gap is now at least

$$\frac{d}{d'+d}\text{gap}(G)$$

The final step is to amplify the gap. We shall discuss that in the next few classes.

Index

Dirac's notation
  bra,ket vectors, 10
  inner and outer products, 10

expanders, 4
  $\lambda_2(G)$, 9
  boosting BPP success, 18
  boosting RP success, 4, 18, 25
  family of constant size expanders, 25
  graph products, 21
    powering, 21
    tensor product, 21
    zig-zag product, 22
  random graphs, 20
  random walks, 14
  rotation maps, 5, 20
  spectral expansion, 9
  vertex expansion, 4, 8
extractors, 3

linear algebra
  norm of a matrix, 15
  spectral theorem, 8

PCP Theorem, 1
  (k, δ) reduction, 30
  alphabet reduction lemma, 30
  amplification lemma, 30
  constraint graphs, 29
  Dinur's theorem, 29
  gap in constraint graph, 29
  gap in SAT, 28
PRG, 6
probability distributions, 2
  ϵ-close to uniform, 3
  distance, 3
  min-entropy, 2