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.

1 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.

2 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.

2.1 Constraint Graphs

Definition 1. *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 1. *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 2 (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 3. *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 2. *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 4 (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 5 (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.

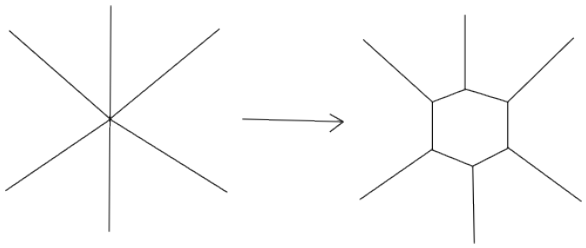
3 Steps for the Amplification Lemma

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.

3.1 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

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

3.2 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.