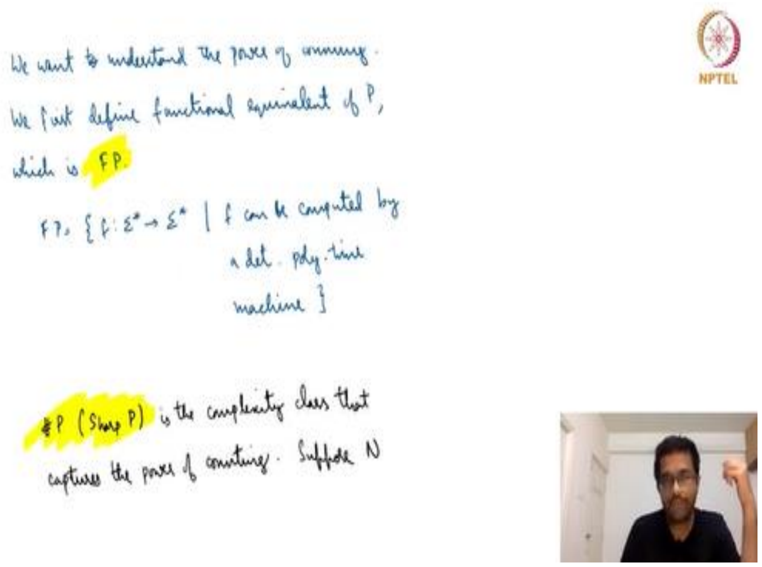
Computational Complexity

Prof. Subrahamanyam Kalyanasundaram
Department of Computer Science and Engineering
Indian Institute of Technology, Hyderabad

Lecture - 49
# P and the Complexity of Counting

(Refer Slide Time: 00:15)



We want to understand the power of counting.
We first define functional equivalent of P, which is FP.

FP = $\{ f : \Sigma^* \rightarrow \Sigma^* \mid f$ can be computed by a det. poly-time machine $\}$

#P (Sharp P) is the complexity class that captures the power of counting. Suppose N

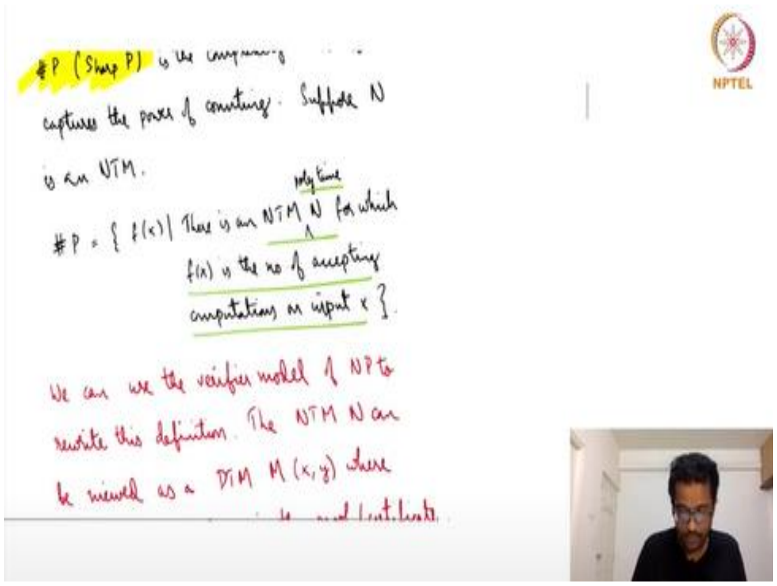
Hello and welcome to lecture 49 of the course computational complexity. In the previous lecture, we saw the complexity of unique SAT. We saw that even when restricted to the case when the given Boolean formula has 0 or 1 satisfying assignment. The problem of determining which of these is the case remains hard. So, now in this lecture, we will see a related problem. So, sorry given a Boolean formula.

Can we count how many satisfying assignments this Boolean formula has? So, it could be 1, it could be 10, it could be 100 or it would be 0. Now, if you could do that how powerful is this. So, counting in general or given a graph can we count how many cliques it has of a certain size. Given a graph, can we count how many cycles it has. So, counting the number of objects of a certain type is a problem that we will look at.

And we will try to understand the complexity of this counting. So, to begin we will first define the functional equivalent of P. So, till now everything that we have seen is our decision problems. Given a language is a given string is in the language or not in the language and similarly for promise problems given a yes class and no class is a given string in the yes class or the no class.

So, this is all yes, no questions. But the counting problems are not yes, no questions because, we have to the answer is the count. So, we first define what is called F P, which is a functional equivalent of P. F P consists of all the line, all the functions, that can be computed by a deterministic polynomial time Turing machine. That is F P. So, here, I have written the definition formally.

(Refer Slide Time: 02:24)



#P (Sharp P) is the complexity captures the power of counting. Suppose N is an NTM.

$\#P = \{ f(x) \mid$ There is an $\underset{\wedge}{\text{NTM } N}$ poly time for which $f(x)$ is the no of accepting computations on input $x \}$.

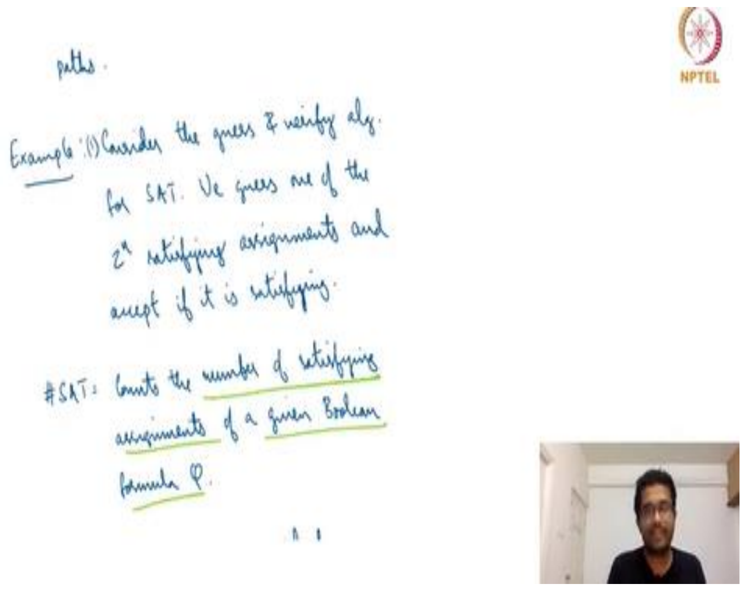
We can use the verifier model of NP to rewrite this definition. The NTM N can be viewed as a DTM $M(x, y)$ where [illegible]

And sharp P, is a complexity class that captures the power of counting. So, sharp P corresponds to the all the functions which can be captured by the number of accepting computations of a non-deterministic polynomial time Turing machine. So, f x so sharp P contains all the functions, such that f x is the number of accepting computations on an input x of a non-deterministic polynomial time Turing machine.

So, this may seem abstract, just to make things simpler, so consider sharp set. So, given a Boolean formula phi, how many satisfying assignments does this Boolean formula have. So, satisfiability is in NP. We have seen it as an NP. And, how did we show that it is an NP? We use the guess and verify model. So, we would guess an assignment and check whether it is satisfiable.

So, potentially we could guess all the two power N assignments, where N is the number of variables. And, so how many of these assignments will need to accept? Well, if it has 10 satisfying assignments, all these 10 satisfying assignments will lead to it getting accepted. If it had one satisfying assignment only, then it would only have once satisfying, one path that leads to acceptance.

(Refer Slide Time: 04:04)



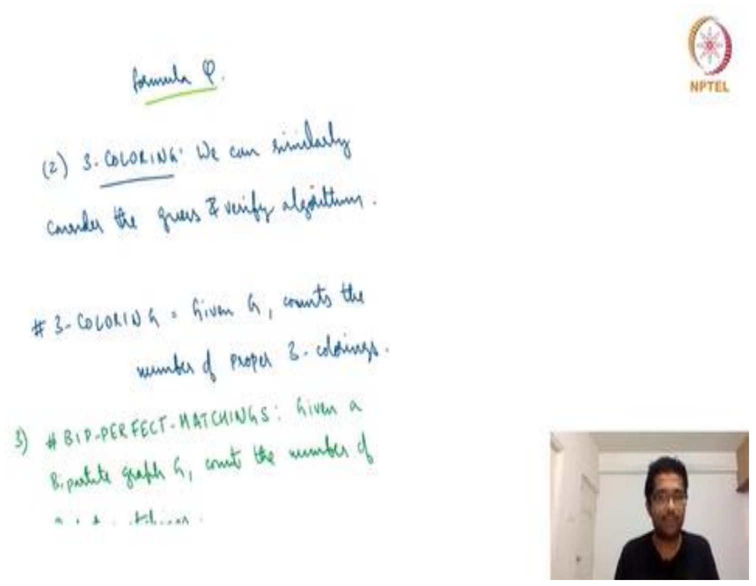
paths.

Example: (1) Consider the guess & verify alg. for SAT. We guess one of the 2^n satisfying assignments and accept if it is satisfying.

#SAT: Counts the number of satisfying assignments of a given Boolean formula φ.

So, one simple example of a language that is in sharp P is sharp SAT, or in other words which is, I have written here. It is the number of satisfying assignments of a given Boolean formula phi. Because SAT is clearly in NP and the number of accepting computations is merely the number of satisfying assignments.

(Refer Slide Time: 04:20)



formula φ.

(2) 3-COLORING: We can similarly consider the guess & verify algorithm.

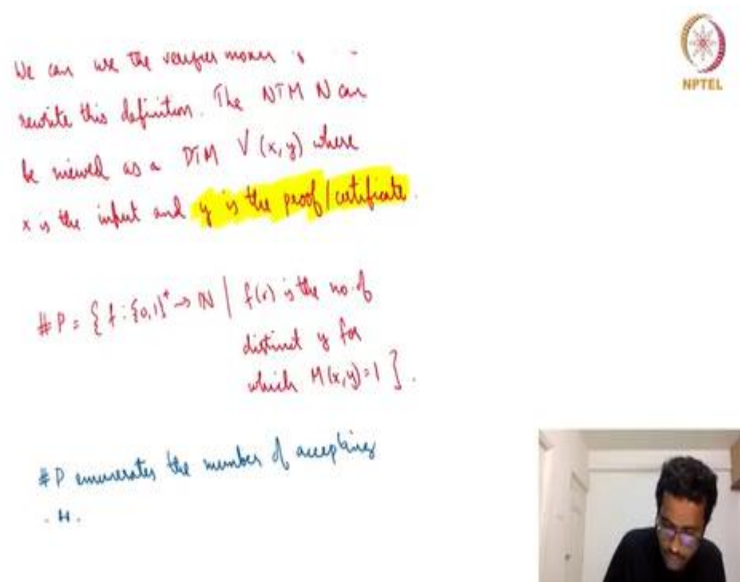
#3-COLORING = Given G, counts the number of proper 3-colorings.

3) #BIP-PERFECT-MATCHINGS: Given a Bipartite graph G, count the number of [illegible]

Another example is 3 Colouring. So, given a graph, how many 3 colouring does it have? So, 3 colouring is an NP, again we had a guess and verify model, to show that it is an NP. We would guess a 3 colour assignment, assignment that uses 3 colours. And, then we would check whether this assignment is a proper 3 colouring. So, the number of accepting number of accepting computations is merely the number of proper 3 colourings of the graph.

So, again 3 colouring is in sharp SAT, sorry sharp P. So, 3 colouring, so again these are all examples in sharp P. So, I am not explicitly writing these are all in sharp P. So, we have a problem in NP and we look at the counting version.

(Refer Slide Time: 05:15)



We can use the verifier model to rewrite this definition. The NTM N can be viewed as a DTM V(x, y) where x is the input and y is the proof/certificate.

$\#P = \{ f : \{0,1\}^* \to \mathbb{N} \mid f(x)$ is the no. of distinct y for which $M(x,y) = 1 \}$.

#P enumerates the number of accepting

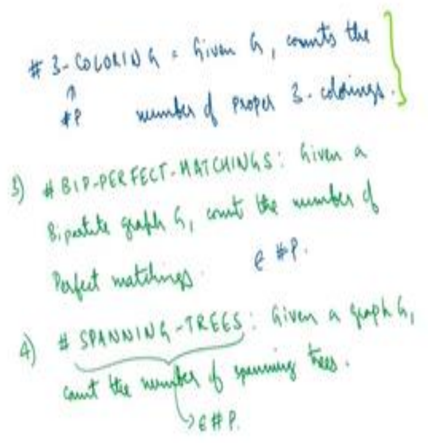
H.

So, just to give another definition we could use the verifier model of NP to write the definition. So, I first said that it is a number of accepting computations. So, in the verifier model instead of a non-deterministic Turing machine, we have a deterministic Turing machine the verifier. I think in the lecture, we use the symbol V for that and we defined the verifier model. So, maybe I will just stick to that. So, where x is the input and y is the proof of certificate.

So, now we are just counting we are asking because there is no non-determinism in the verifier. Verifier is deterministic. So, all the non-determinism gets pushed into the choice of this proof or the witness certificate or rather by that I mean the choice of the proof or certificate. So, how many y is are there that leads to the acceptance of x N V. So, that is another way to characterize sharp P. So, given so what is sharp P? It is a class of all functions which map x to the count.

So, it is x is from 0, 1 start to a count. A count is a natural number of course, and where the function corresponds to the number of y that leads to x being accepted. So, f x is the number of distinct y for which x is verified correctly. So, the number of proof or certificate strings for an x.

(Refer Slide Time: 06:47)



$\underset{\#P}{\overset{\uparrow}{\#}}$ 3-COLORING = Given G, counts the number of proper 3-colorings.

3) # BIP-PERFECT-MATCHINGS: Given a Bipartite graph G, count the number of Perfect matchings. $\in$ #P.

4) $\underbrace{\text{\# SPANNING-TREES}}$: Given a graph G, count the number of spanning trees. $\to \in$ #P.

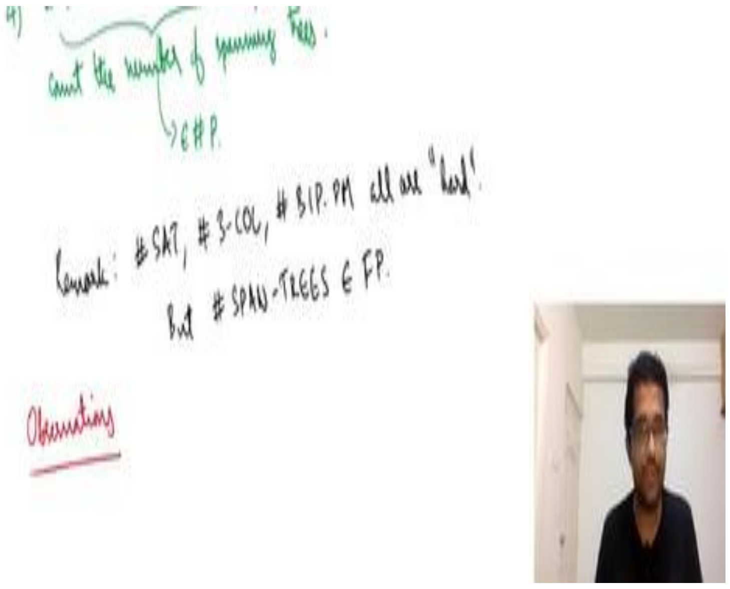
So, sharp P and you also enumerate the number of accepting parts again this is a repetition. So, we saw two examples. One is sharp SAT which is in sharp P and then, sharp 3 colouring is in sharp P. So, maybe I just write that here sharp SAT is in sharp P, and sharp 3 colouring also is in sharp P. So, one may ask is it the case that, what about other like both SAT and 3 colouring are in NP. But we could also have problems in P so for instance bipartite perfect matchings.

As I had mentioned before, given a bipartite graph G, to decide whether it has a perfect matching or not, it has a polynomial time algorithm. So, now I can ask, what is the number of perfect matchings in this graph. So, this is in sharp P sorry not P, but sharp P. And, I could ask another question, given a graph G. What is the number of spanning trees of, what is the number of spanning trees of G? Again, this is also in sharp P.

So, one thing to note is that, the first two problems, sharp SAT and sharp 3 colouring. These correspond to the 3 colouring and SAT are both NP complete. Even though they are in NP, they are NP complete. So, sharp the even the counting version is hard. Even the counting version sharp 3 colouring and sharp SAT are both in sharp P, the counting versions are considered hard. And, even for the case of bipartite perfect matchings, even though we think that bipartite perfect matching the decision version is easy.

There are efficient polynomial time algorithms, the counting version becomes hard. So, this is, so just to summarize, maybe I will just make some space here.

(Refer Slide Time: 09:17)



4) count the number of spanning trees.

$\in \#P$.

Remark: #SAT, #3-COL, #BIP. PM all are "hard".

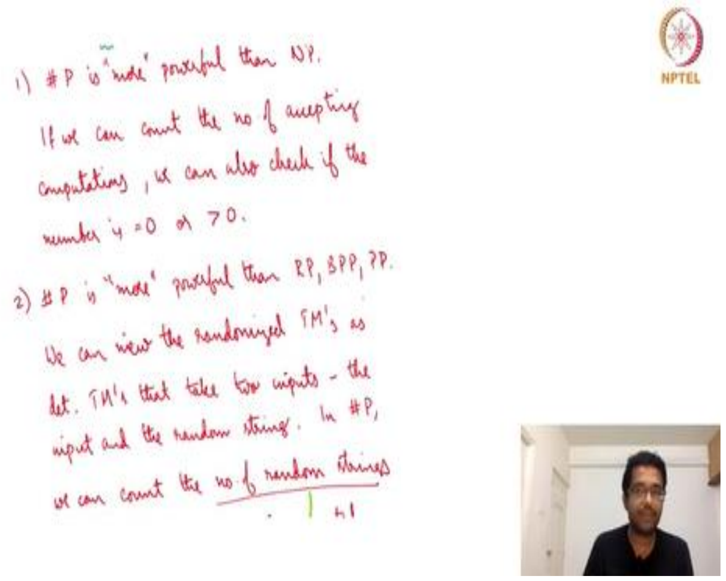
But #SPAN-TREES $\in$ FP.

Observations

Now to summarize maybe I will just say note, how to mark sharp SAT, sharp 3 colourings, sharp bipartite perfect matching's are all are hard. Meaning it is, we do not know of an efficient way of doing this and we believe it is these are all hard. Later, we will see sharp P completeness and in fact all three of them are sharp P complete. But a sharp spanning trees is actually in FP which means, it can be there is a deterministic polynomial time algorithm to compute this.

So, it basically boils down to a determinant computation. So, some problems even though the decision version is easy the counting version becomes hard. But not all problems need to be behaved like that. Like spanning trees and bipartite perfect matching, the decision versions are easy. But spanning trees the counting version remains to be easy but for bipartite perfect matching's the counting version becomes hard.

(Refer Slide Time: 10:37)



1) #P is "more" powerful than NP.
If we can count the no of accepting computations, we can also check if the number is =0 or >0.

2) #P is "more" powerful than RP, BPP, PP.
We can view the randomized TM's as det. TM's that take two inputs - the input and the random string. In #P, we can count the no of random strings

Some observations is that sharp P is seems to be more powerful than NP or at least as powerful as NP. Well, why in NP let us say satisfiability the goal is to determine whether this formula has a satisfying assignment. So, if you can count the number of satisfying assignments, we can check whether the number of satisfying assignments is 0 or more than 0. So, if we can count, we can certainly decide whether it is satisfiable or not.

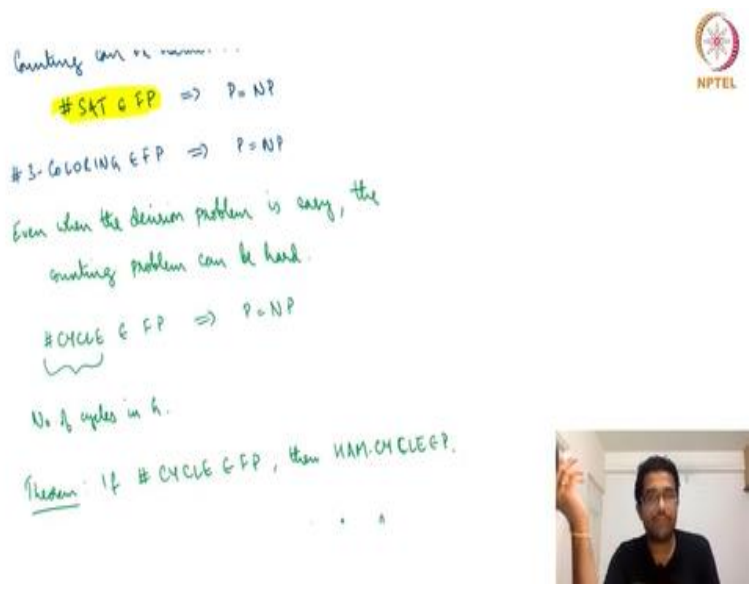
If you can count the number of proper 3 colourings, we can certainly decide whether the number of proper 3 colourings is 0, or something greater than 0. So, if you can, sharp P is at least as powerful as NP, and similarly sharp P is at least as powerful as the randomized classes RP, BPP etcetera. Because, we can again view the randomized Turing machines, as deterministic Turing machines which takes the random string as an extra input.

We can view it as a deterministic Turing machine that takes the input and the random string. And, we can view it like how many random strings allow an input x to get accepted or how many random strings lead to an input string x getting accepted. So, that we can that can be done using sharp P. And, once you do that, now you can count the fraction of random strings, random choices which lead to an input string x getting accepted.

So, if you want to check whether a given input string is in the language, we just check what is a fraction or what is the probability by which it gets accepted. So, for which we can estimate by the number of accepting or number of fractions of random strings that lead to it getting accepted. So, you can check, if it is supposed to, if it is a BPP machine then we can check whether the fraction of accepting strings is greater than two third or less than one third.

If it is RP, we can do the corresponding thing. If it is in PP then again, we can do the corresponding thing. So, if you can count the number of accepting computations, we can also decide the probabilistic polynomial time classes RP, BPP etcetera So, the point is P is at least as powerful as all these classes.

(Refer Slide Time: 13:20)



Counting can [illegible]

$\#SAT \in FP \Rightarrow P = NP$

$\#3\text{-}COLORING \in FP \Rightarrow P = NP$

Even when the decision problem is easy, the counting problem can be hard.

$\underbrace{\#CYCLE}$ $\in FP \Rightarrow P = NP$

No. of cycles in G.

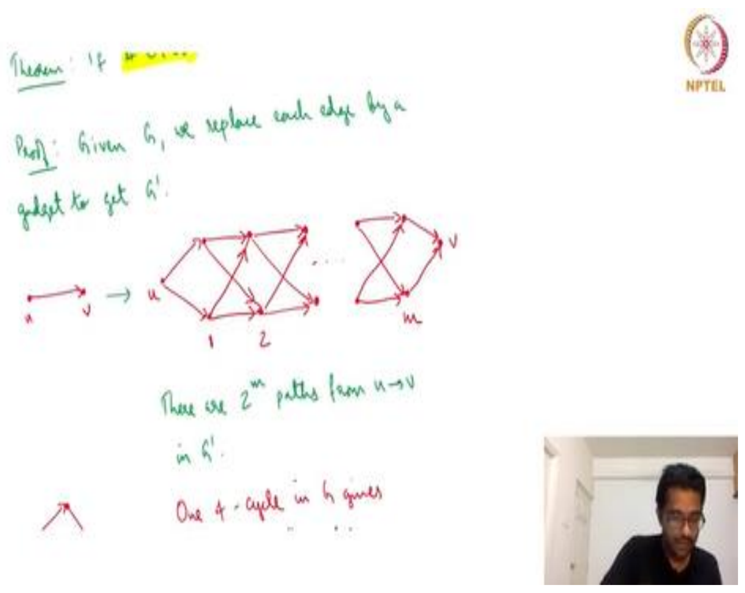
Theorem: If $\#CYCLE \in FP$, then $HAM\text{-}CYCLE \in P$.

So, this is another point, and the next point is something that I have already made, in the case of sharp SAT and sharp 3 colouring, these correspond to NP complete languages or the decision versions of which are NP complete. If we can count the number of accepting parts in polynomial time, if sharp SAT is in, if you can count the number of satisfying assignments in polynomial time right, if sharp SAT is in FP.

Then, it means that we can even decide whether it is satisfiable or not in polynomial time, which means P = NP. And same thing holds for 3 colouring as well, if you can count the number of proper 3 colourings, we can decide whether it is 3 colourable or not which means P, because 3 colouring is NP, complete this would imply P = NP. So, these two problems, we have the decision version is hard. So, the counting version remains to be hard.

And, as I mentioned earlier like, I already illustrated one bipartite perfect matching. Now we will see another problem, which is discounting the number of cycles. So, to decide whether the graph has a cycle is can be done in deterministic polynomial time. Now we are saying that, if we have an algorithm that counts the number of cycles. The in polynomial time, then P = NP. So, the counting the number of cycles implies P = NP.

(Refer Slide Time: 15:02)


Theorem: If [illegible]

Proof: Given G, we replace each edge by a gadget to get G'.

There are 2^m paths from $u \to v$ in G'.

One 4-cycle in G gives . . .

So, decision version is easy because it is easy to decide whether the graph has a cycle. But, counting version is as hard as NP complete. So, let us see, why this is the case? So, the formal statement is, if counting so I am using sharp cycle to denote the problem of counting the number of cycles in the graph. So, these are labelled cycles. So, each cycle, each label is counted separately.

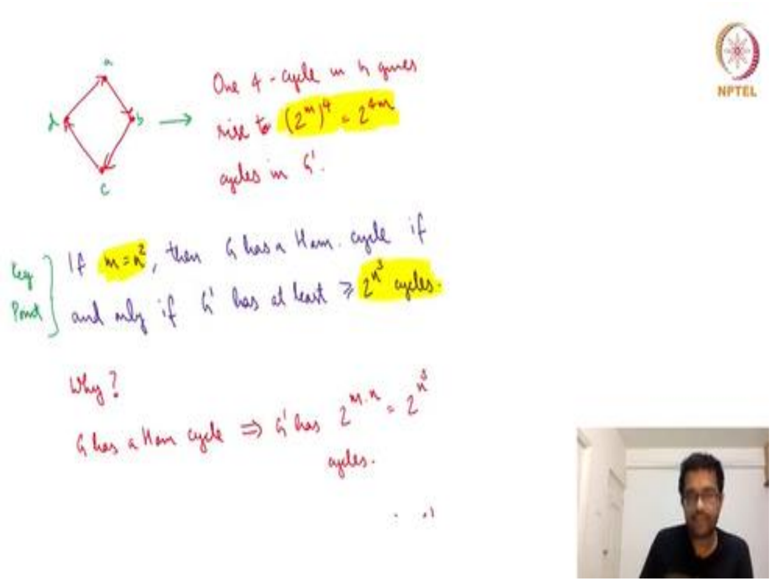
So, if the number of cycle is in FP, which means, if we can count the number of cycles in polynomial time FP is just a functional equivalent of P. Then, we will it will imply that Hamiltonian cycle is in P. So, as we have seen, when we saw NP completeness Hamiltonian cycle is an NP complete language. So, if Hamiltonian cycle is in P, that implies that P = NP. So, let us see why this is the case.

So, the main idea is we will use a gadget where we replace so given graph G. We want to count the number of cycles of G. So, what we will do is replace each edge of G with a gadget? Such that any cycle in G becomes a lot of cycles in G prime. So, we will transform G to G prime using this gadget. So, any cycle in G becomes a lot of cycles in G prime. And, the bigger the cycle the more cycles are produced in G prime.

So, Hamiltonian cycle is the biggest possible cycle that we could have in G. So, that results in a lot of cycles in G prime and in fact the only way that we will have so many cycles in G prime is if G had a Hamiltonian cycle. So, that will give us the equivalence. So, the gadget is really simple, the any edge is in the left-hand side here u v, is replaced by this crisscross network. So, you can see that u v is just one edge in G so this is G.

And in G prime corresponding to u v, we have two power m paths so from u we can go up, then go down then up, so every stage here 1, 2, 3 up to n, you can decide to go to whichever level you want, the top level or the bottom level. So, there are two power m choices to go from u to v. Two power m choices in G prime.

(Refer Slide Time: 17:48)



One 4-cycle in G gives rise to $(2^m)^4 = 2^{4m}$ cycles in G'.

Key Point: If $m = n^2$, then G has a Ham. cycle if and only if G' has at least $\geq 2^{n^3}$ cycles.

Why?

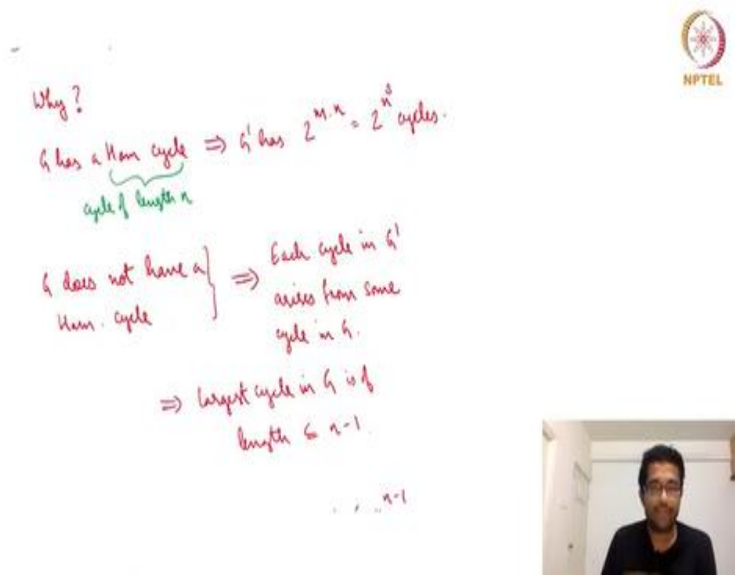
G has a Ham cycle $\Rightarrow$ G' has $2^{m \cdot n} = 2^{n^3}$ cycles.

And, now suppose let us say you have a cycle a, b, c, d in G, a four cycle. How many cycles does it correspond to in G prime? Well, the edge a b will be replaced by a similar crisscross network. So, there will be 2 power m paths from a to b 2 power m paths from b to c, 2 power m from c to d and 2 power m from d to a. So, there will be 2 power m paths for each edge and you could choose for each of these edges a b, b c, c d, d a.

There are 2 power m choices 2 power m whole power 4 is the number of cycles that we have in G prime corresponding to this cycle in G, these 4 cycles in G. 2 power m whole power 4, which is simply 2 power 4 m. So, in general, if we have a cycle of length K, this leads to 2 power K m cycles in G prime. So, a cycle of length K in G leads to a cycle of length 2 power K m in G prime. So, now we will come to the main statement or main key point.

So, we will say that, so again I have till now, I have not mentioned what should be m. I just so whatever m is we have some corresponding relations. So, if m is large enough let us say m is n squared, then the claim is that if G has a Hamiltonian cycle if and only if G prime has at least 2 power n cubed mini cycles. So, G has a Hamiltonian cycle, if and only if G prime has at least 2 power n cube cycles.

(Refer Slide Time: 19:48)



Why?

G has a $\underbrace{\text{Ham cycle}}_{\text{cycle of length } n}$ $\Rightarrow$ G' has $2^{m \cdot n} = 2^{n^3}$ cycles.

$\left.\begin{array}{l}\text{G does not have a} \\ \text{Ham. cycle}\end{array}\right\}$ $\Rightarrow$ Each cycle in G' arises from some cycle in G.

$\Rightarrow$ Largest cycle in G is of length $\leq n-1$.

$n-1$

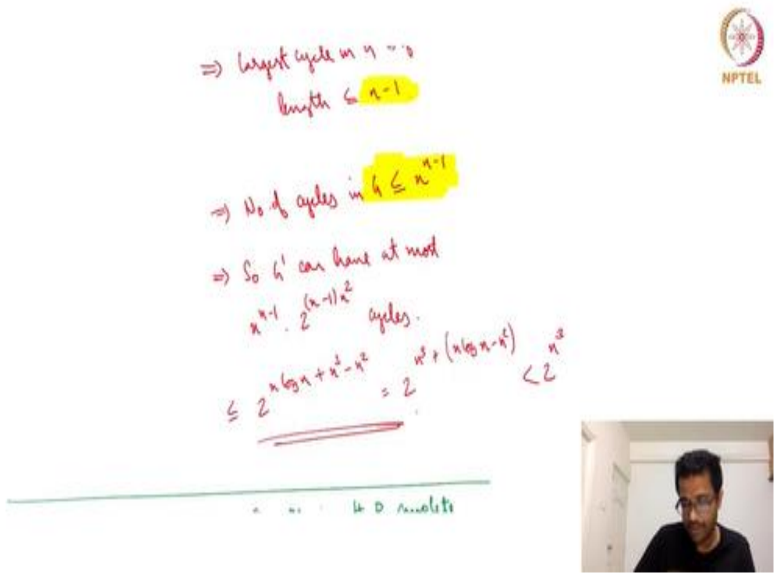
So, one direction is really easy, suppose G has a Hamiltonian cycle. That means, it is a cycle of length n so Hamiltonian cycle means cycle of length n. Now G prime has 2 power m multiplied by n cycles. So, two same thing 2 power 4 m that we did above, similarly 2 power m multiplied by n. Because, where n is the length of the Hamiltonian cycle, and by our choice of m, m is also n squared, so m times n is n cubed.

So, Hamiltonian cycle leads to 2 power n cubed corresponding cycles in G prime. A 2 power n cubed corresponding cycles. Now suppose G does not have a Hamiltonian cycle, now we want to say that, G prime does not have 2 power n cubed cycles. And, that is also not very difficult to see. Any cycle in G prime corresponds to some cycle in G. Because only thing we are doing is replacing an edge with some mesh network.

So, if there is a cycle in G prime, there is a cycle in G. So, if G does not have a Hamiltonian cycle, the largest cycle in so you look at G prime it, let us say you and you want to check the number of cycles. So, instead of counting the number of cycles in G prime, what we will do is to count the number of cycles in G and see how many corresponding cycles it gave rise to in G prime. So, what is the largest cycle in G? what is the length of the larger cycle in G?

We know that there is no Hamiltonian cycle. So, the length of the larger cycle in G is n - 1. So, the length of the larger cycle in G is n - 1. What, how many cycles could have been there in G? So, the first vertex can be chosen in n way, second vertex could be chosen in n ways, in fact you cannot repeat the vertex which, but anyway we are considering an upper bound. So, that, n power n - 1 base to, choose a cycle of length n - 1 or smaller in G. So, this is an upper bound n power n - 1.

(Refer Slide Time: 22:25)



$\Rightarrow$ largest cycle in G has length $\leq n-1$

$\Rightarrow$ No. of cycles in $G \leq n^{n-1}$

$\Rightarrow$ So G' can have at most $n^{n-1} \cdot 2^{(n-1)n^2}$ cycles.

$$\leq 2^{n\log n + n^3 - n^2} = 2^{n^3 + (n\log n - n^2)} < 2^{n^3}$$

This many ways are there to choose a cycle in G, at most these many ways. So, these are the number of cycles in G. There is an upper bound on the number of cycles in G, n power n - 1. And each cycle is of length at most n - 1, and we know that each cycle of length n - 1 gets blown up into 2 power n -1 into m cycles Because, each cycle of length m corresponds to 2 power m into, sorry, each cycle of length K corresponds to 2 power m K cycles in G prime.

So, the number of cycles in G prime is, the number of cycles in G which is n power n - 1 into or multiplied by the number of cycles that each cycle gives rise to at most, which is 2 power, the length of the cycle upper bound on the length of cycle multiplied by m and, our choice of m was n squared. So, I think there is a small typo here, so this is n squared, sorry, 2 power n - 1 into n squared. So, I may have to rewrite this a bit.

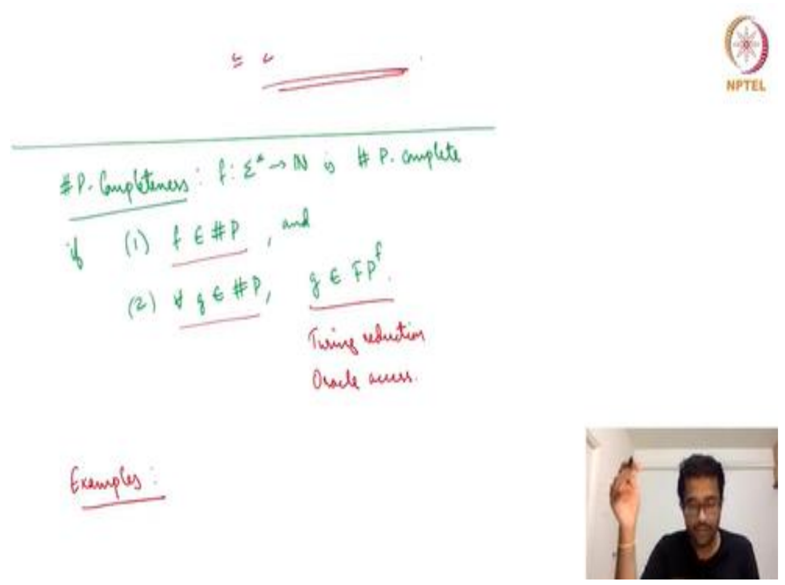
So, which just corresponds, so again n power n - 1 can be upper bounded by 2 power n log n. We could just write n power n, which is 2 power n log n. And, 2 power n - 1 times n squared is nothing but, n cubed - n squared. So, we have which is 2 power n cubed + n log n - n squared. So, you can notice that here we have n cubed, but then we have this n log n - n squared, n log n-n squared is a negative quantity. So, this number is certainly less than 2 power n cubed.

Because, n log n – n, n squared is a negative quantity. So, this means that the number of cycles of G prime is always less than 2 power n cubed. So, the number of cycles of G prime, is going to be at least 2 power n cubed, if and only if G has a Hamiltonian cycle. So, this means, if you are able to count the number of cycles in a graph in polynomial time, we could solve the Hamiltonian cycle problem by this reduction.

So, this means this problem, if we sharp cycle is in FP, then P = NP. So, this is what this we have shown. Again, the idea is very simple. Every edge is replaced by this network. So, if, so the networks give rise to a lot of options. So, every cycle of a certain length K, it gets replaced by 2 power m K cycles in G prime. So, if G has a Hamiltonian cycle, then we get 2 power n cube cycles in G prime, where we choose m to be n squared.

Whereas, if g does not have a Hamiltonian cycle, then the number of cycles that it gives rise to all the even the biggest cycle the n - 1 length cycles gives rise to even that will not reach the 2 power n cubed because there is this exponentiation that is happening. So, each additional length really takes it further from the previous length. So, even if you had a lot of cycles of length n into n, n - 1 even then we cannot reach 2 power n cubed. This is the key idea of the proof.

(Refer Slide Time: 26:47)



$\leq$ ⌊

#P-Completeness: $f : \Sigma^* \to \mathbb{N}$ is #P-complete

if (1) $\underline{f \in \#P}$, and

(2) $\underline{\forall g \in \#P}$, $\underline{g \in FP^f}$.

Turing reduction

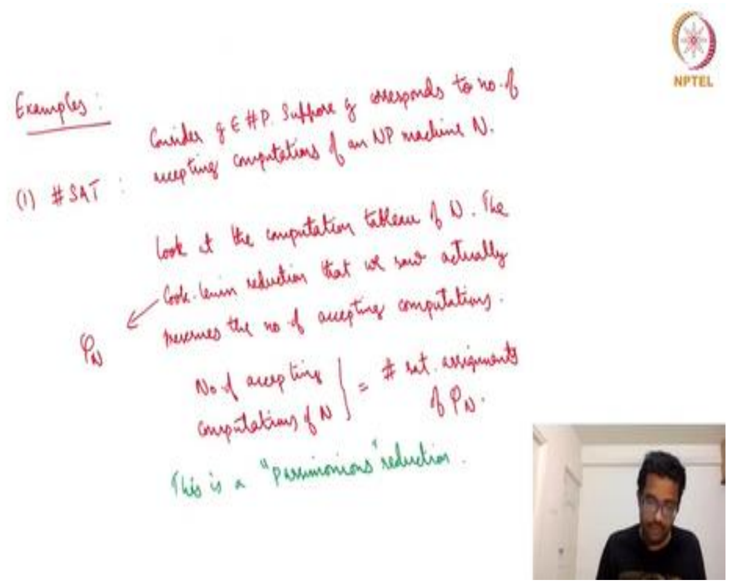
Oracle access.

Examples:

So, finally we will define what is sharp P completeness? And then conclude. So, sharp P completeness is defined very similar to NP completeness and other things. But there is one small difference. So, a function is sharp P complete, if that function is in sharp P, which is the first step, f is in sharp p. And, all the G in sharp P reduces to f. So, till now we have used the minimum reductions.

But here we will use the Turing reduction, which we saw while we defined the oracle model. We mentioned Turing reduction, especially while talking about Baker Gill Solovay theorem. So, if G has a function, so basically an FP machine with an oracle access to f should be able to compute g. So, if you can solve f, you can solve g, or if you can count f, you can count g as well.

(Refer Slide Time: 27:53)



Examples:

(1) #SAT : Consider $g \in \#P$. Suppose g corresponds to no. of accepting computations of an NP machine N.

Look at the computation tableau of N. The Cook-Levin reduction that we saw actually preserves the no. of accepting computations.

φ_N

$$\left.\begin{array}{l}\text{No. of accepting}\\ \text{computations of N}\end{array}\right\} = \#\text{ sat. assignments of } \varphi_N.$$

This is a "parsimonious" reduction.

So, some examples, so two of the examples we have already seen, one is sharp SAT. Basically the key idea is we have seen the reduction. Why is SAT NP complete? Because you can look at the computation table, or computation tableau, phi is very satisfiable. Sorry, I have to explain a bit more. So, you have to show that all the functions, all the NP machines reduce to satisfiability. So, we would we can use something very similar to Cook Levin theorem.

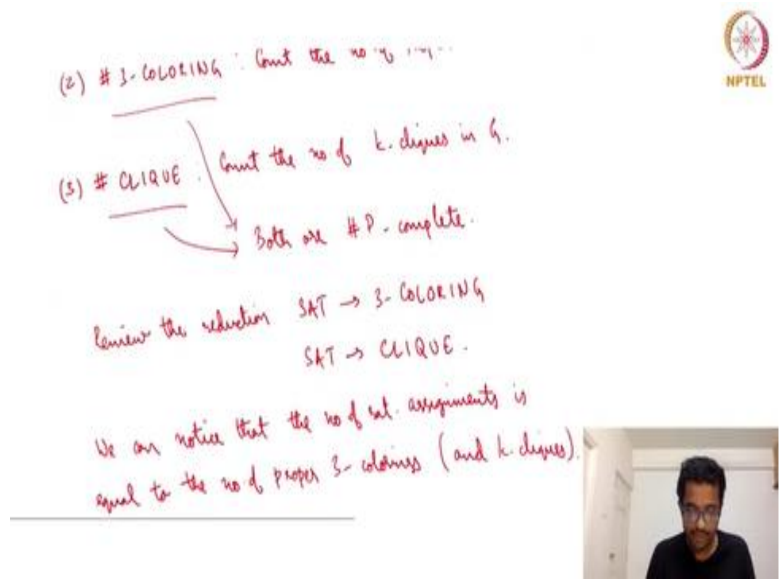
So, consider any sharp P language. So, consider, so the membership in sharp P we have already seen. So, consider g in sharp P. Suppose NP machine, suppose sorry, g corresponds to number of accepting computations of an NP machine n. Suppose g corresponds to the number of accepting computations of an NP machine n. So, now look at the computation tableau of n and the Cook Levin reduction the Cook Levin in reduction that we saw.

Actually, this Cook Levin reduction preserves the number of accepting computations. So, what I am saying is that suppose cook Levin theorem reduction gives us some formula called phi N. So, number of accepting computations of N is equal to number of satisfying assignments of phi N. So, this is called, so this property where the number is preserved is called parsimonious. So, this is a parsimonious reduction.

So, the number of accepted computations of N is equal to the number of satisfying assignments. So, if you just think about the reduction that we did. It is not like we are merging multiple accepting computations of N into the same satisfying assignment for the formula. So, this is a satisfying, this is parsimonious reduction. Parsimonious means, the number of accepting computations in the case of N is preserved when it translated to the Boolean formula.

So, because of which if we can count the number of satisfying assignments of a Boolean formula. We can count the number of status accepting computations of any non-deterministic Turing machine which means any g in sharp P can be computed. So, any g in sharp P can be computed by with access to a SAT oracle which is what we wanted to show.

(Refer Slide Time: 32:42)



(2) # 3-COLORING : Count the no of [illegible]

(3) # CLIQUE : Count the no of k-cliques in G.

Both are #P-complete.

Review the reduction SAT → 3-COLORING

SAT → CLIQUE.

We can notice that the no of sat. assignments is equal to the no of proper 3-colorings (and k-cliques).

Similarly other problems like 3 colouring clique, so 3 colouring asks for the number of proper 3 colourings of a graph. Clique asks for, so this is count the number of proper 3 colouring in g. So, this is asking for count the number of k cliques, where k is part of the input in G. So, both of these are also sharp P complete, both are sharp P complete. The easiest way to see that, is by review the, by reviewing the reduction, review the reduction from SAT to 3 colouring and SAT to clique.

And, we will notice that, we can notice that the number of satisfying assignments of a given Boolean formula and the number of 3 colourings, are either same or you can count you can estimate one from the other. I think it is actually exactly the same in both these cases. We can notice that the number of satisfying assignments is equal to the number of proper 3 colourings and k-cliques.

So, if you can estimate 3 colourings or the number of 3 colourings or k-cliques, you can also estimate the number of satisfying assignments. So, these are sharp P complete. Basically, if you count the number of 3 colourings, you can count the number of satisfying assignments and which is already shown to be sharp P complete. So, these are some examples of sharp P complete, I think most of the simple graph problems, even vertex cover, independent set they are all sharp P complete.

In fact, what I saw said earlier, the number of bipartite matching's, even that is sharp P complete. So, we will see the proof of that, this one there is a famous theorem. We will see the proof of that also in the next lecture. So, I will just summarize, so we defined what is sharp P which is an, it is a complexity class that captures the power of counting, which corresponds to the number of accepting computations of a non-deterministic polynomial time Turing machine.

We saw some examples like sharp SAT, sharp 3 colouring, sharp bipartite perfect matching's and sharp sinus spanning trees; these are all in sharp P. And, we saw that sharp P is at least as powerful as NP, and then the randomized complexity classes like RP, BPP etcetera. We saw that counting the number of cycles, if you can do that in polynomial time, deterministic polynomial time, then P = NP.

In other words, even though the decision version is easy, in polynomial time, deterministic polynomial time, the counting version can become hard. This was very simple reduction with, where we replace edge with a gadget. And, finally we define sharp P completeness which means all the functions in sharp P can reduce to the sharp P complete language or sharp P complete function.

And, we the main thing is that we use Turing reduction, so we use oracle access, so we could make multiple calls unlike the minimum reduction. And we saw that, sharp SAT, sharp 3 colouring and short clique are all sharp P complete. So, with that, I conclude today's lecture this is 49 and just to summarize what we have done in this week.

We first saw that, the given a bipartite graph outputting the number of bipartite perfect matching’s, is in RNC using isolation lemma and you have a parallel algorithm to output the number of bipartite perfect matching’s. This was the Mulmuley Vazirani Vazirani's paper. Then we define promise problems and Valiant and Vazirani theorem, which says that, even if we are given the promise that a given Boolean formula has either 0 or 1 satisfying assignments.

Even then with this promise also this the testing for satisfiability continues to be hard or is unlikely to be easy. And, finally we saw sharp P, we defined and we saw some examples and define sharp P completeness. In the next week, we will see two important theorems, one that shows that the problem of counting the number of bipartite perfect matching’s is sharp P complete. It is also called the permanent this problem is also called the permanent.

And, second, that sharp P is very powerful that in fact the entire polynomial hierarchy it is as powerful as at least the entire polynomial hierarchy. So, these two results we will see in the next week and till then thank you.