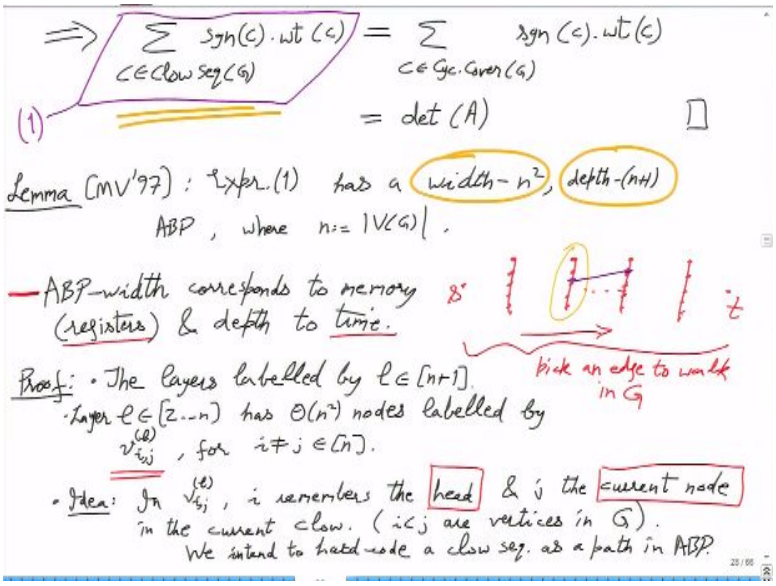
Arithmetic Circuit Complexity
Prof. Nitin Saxena
Department of Computer Science and Engineering
Indian Institute of Technology-Kanpur

Lecture - 06

In the last class, we showed that determinant is equal to this expression 1.

(Refer Slide Time: 00:17)



Let us say expression 1 here is this big sum of signed weights of all clow sequences with length exactly equal to n, which is the number of vertices. So this in particular contains cycle covers. And cycle covers is another way to look at permutations. So we are actually computing a bigger sum, but then the extra things we showed cancel out. So now what we will do is we will show that for some weird reason this big sum is easier to compute than the previous sum. Okay, any questions?

Okay, so we will prove this lemma by Mahajan, Vinay. So we will actually show that there is an ABP of width n^2 and depth n + 1. Okay, so there is an $n^2 \times n + 1$ ABP where every path from s to t will be implementing a clow sequence with the signed weight, exactly as in the expression. So the sum over all the parts will give you determinant, okay.

So before that, let me just point out a physical interpretation of ABP because that is what we will be using here. So ABP width you can think of as the number of registers

in your machine, okay. So width corresponds to memory. So registers and depth corresponds to time. So basically, if you have width-many registers, then you can implement any ABP of that much width in time equal to the length or depth of the ABP.

So in other words what we will be doing is, we will show that determinant can be computed using n^2 registers in time n okay. That is an alternative way to look at this implementation. Now note that width n^2 or n^2 registers is not too much because your matrix already had n^2 entries.

So you can think of the matrix entries being stored one per register and within those many registers, you can actually do computation and in n time you will have the determinant. And when you convert this via matrix multiplication iterated matrix multiplication into circuit then you get a different complex or different interpretation. So there you will get actually $\log n$ time. So the things are a bit different there.

So this ABP is actually a stronger result, because you can derive the circuit result from here, right. So the way we will do this is we will have layers labeled by l wearing from 1 to n + 1. So one l = 1, l = n + 1 are the source and target vertices. But for l,2 to n layer l,2 to n has each of them have around n^2 nodes which will be labeled by a pair. So i, j. And the layer label will be in the superscript for different i, j between 1 to n, okay.

So the vertices that you see in a layer are these $v_{i,j}^{(l)}$. So these around n^2, many things you can imagine them to be stored in the registers. At time l, they are stored in the register and what they represent is well, i will represent the head whose clow we are currently building or the walk we are doing and j will be the current vertex. So nothing else will be remembered okay.

So at time l all we remember is what is the head of the close walk that we are currently trying to do and where are we in the closed walk because of these two things

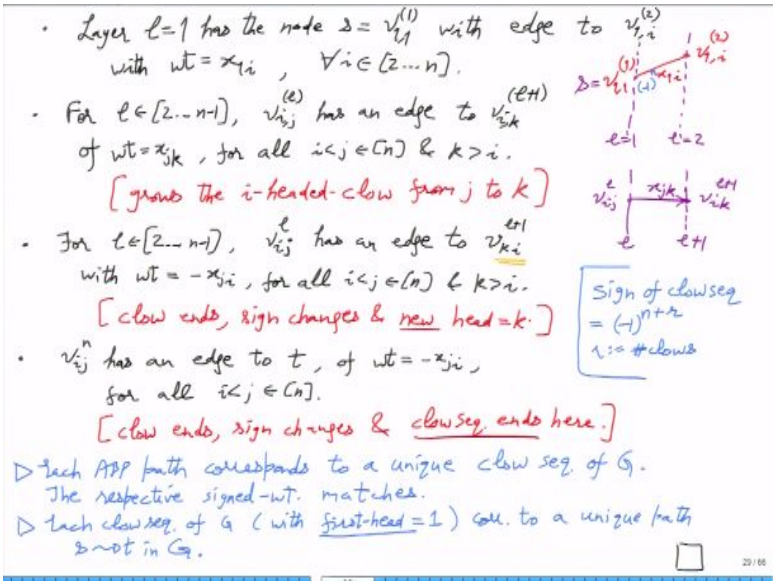
and then based on this you have to take the next step. So that next step will go to l + 1. You will pick an edge and you will go to some $v`_{i,j}^{(l+1)}$.

So this will be basically in the closed walk. What you are doing is that you are trying to build a clow with the head equal to i and from j^{th} vertex you are moving to j` and the only thing you have to ensure is that both j and j` are greater than i. Okay this is the only constraint. So using this nearly oblivious walk, you will be able to complete a clow.

So when you come back to i then the interpretation is that you have completed the walk, closed the walk and at that point you should introduce a sign because every clow introduces a sign of -1 and these -1 then get multiplied out to get the actual sign. So let us look at the details of the implementation. The idea is as I just mentioned that in $v_{i,j}^{(l)}$, i remembers or i in fact is the head and j the current node in the current clow being constructed.

It is not yet a closed walk, but it is just a walk which will ultimately close back to i. So in particular i will be less than j. So here actually the subscripts i, j denote vertices. They are exactly the vertices, so i has to be smaller than j vertices in the graph G. So let me write that down in fact. And so we intend to hard code a clow sequence as a path in the ABP. This is the overall idea. So let us look at the layers in more detail.

(Refer Slide Time: 10:00)



So layer $l = 1$ has the node s with edge of weight $v_{i,1}{}^{(1)}$ to $v_{1,i}{}^{(2)}$, so there is so we are drawing an edge from first layer to second layer with the weight $x_{1,i}$ and this you do for all i. So this is just the first layer. So this is basically the s vertex. That is the first layer going to the second layer and we are just describing how the edges will be from s to neighboring vertices.

So neighboring vertices are n - 1 and when you are looking at a path that takes this $x_{1,i}$, it is like you are building a clow with head one and moving to vertex i. $x_{1,i}$ is I mean you should think of it as the symbolic variable in the $n \times n$ matrix, but you can also think of it as the weight of the (1,i) edge in the graph. If you are given something in the input then you can use then you should think of $x_{1,i}$ as that.

And okay so now let us look at a general layer l. So for $l \in [2, \cdots, n-1]$, so intermediate layers and for all $i < j$. In fact, let me suppress this. So we are at layer l and $v_{i,j}{}^{(l)}$ is the vertex. So this has an edge to $v_{i,k}{}^{(l)}$ of weight equal to, so the weight should be? If you are at $v_{i,j}{}^{(l)}$ and you move want to move to $v_{i,k}{}^{(l+1)}$, what is the weight that you should put on this edge in the ABP; x_{jk} right. So that is the thing.

Do this for all $i<j \in [n]$ and k should be the only condition is it should be more than i. Okay, so do it for all these k's that are bigger than i and between 1 to n. And

for all these (i, j)s also. So this describes the edges that will cross layer l to go to layer l + 1. So that describes all the intermediate layers and the first layer. So then what is the meaning of interpretation of this? This grows the i headed clow from j to k.

Okay, this is what we are doing. When we pick $x_{j,k}$ in a path in the ABP in the graph what we are doing is we are actually walking in a clow. Where? That is what we have in the picture. So we are in a clow whose head is i and then currently we are at j. We do not know what happened between i and j. We also do not care. We just want to move to something bigger than i which is in this case k.

Okay, so it is an oblivious, nearly oblivious, walk. So oblivious to what happened before. But we cannot just continue walking, at some point we have to fall back to i to the head. So that needs special treatment because till now we have not introduced any sign. So how do you introduce the sign. So that we will do when we are folding back to i. So that is here.

So for $l \in [2, \cdots, n-1]$, $v_{i,j}{}^{l}$ so let me ignore the brackets and the comma. So $v_{i,j}{}^{l}$ has an edge to $v_{k,i}{}^{l+1}$. Yeah, so now i has gone, so if you look at the subscript i has gone to the second coordinate. So what this is meant to signify is that i headed clow we were at j, we go back to i and then we create a new head called k. Okay, so this is the transition point from current clow to the next clow.

So this is the place where we should introduce a minus sign. Every clow development introduces a minus sign. Lets first talk about the weight. Yes. No so okay. Let me write it down here. Sign of a clow sequence is equal to $(-1)^{n+r}$ where r is the number of clows. So let us ignore the $(-1)^{n}$ because n is common across all the clows. The thing that is different is this r.

How many clows are there in your clow sequence? So whenever you are developing a new clow, just introduce a minus sign. So that is the formula we are using. $(-1)^{n}$ we will assume to be just one. So we just ignore that. We could have put that in the

beginning when we looked at $l = 1$ we could have put it here. It is not very important. Okay, so here we are jumping to the next clow.

So what should be the conditions? Has an edge to $v_{k,i}^{(l+1)}$ with weight $-x_{j,i}$. So usually it would have been just this j to i edge weight, but here we are also introducing a sign. And just introduce this for all $i < j \in [n]$ and $k > i$. You want the next head to be bigger than i. So that is all you have to remember. So what happened is clow ends, sign changes and the new head is k, right.

This is the interpretation in the walk in the graph. So this describes everything from layer 1 to layer n – 1. So the last layer what do you do? So $v_{i,j}^{n}$ has an edge to t of weight equal to, well so in the last the layer just before t so that has to close the current walk, right. It cannot leave it open and it cannot create a new clow. So if i, j was the, so you are in $v_{i,j}^{n}$. So then you should go to from j we should just go back to i.

Okay, there is no option. And you have to flip the sign then. So this will be $-x_{j,i}$ and do this for all $i < j$. So what happened here is clow ends, sign changes and clow sequence ends. So not just the clow but also the clow sequence ends here. Do you get the feeling that every path from s to t will give you a clow sequence with the correct signed weight if you take the product of the edge weights?

So then ABP by definition computes sum over all the paths, which will be determinant. So those are the observations. So the observations are each ABP path corresponds by design to a unique clow sequence of G. Moreover, the signed weight the respective signed weight matches. So every part defines a unique clow sequence and when you look at the signed weights of these two they are the same.

And second observation is that each clow sequence okay so when you look at a path s to t in this ABP you will get a clow sequence, but you also have to show the converse, that every clow sequence in the graph has been implemented as a path. Is that true?

Right. But we fix the first head to be 1. Is that kosher? Could the head have been 2? No. So note that the clow sequence has length exactly n.

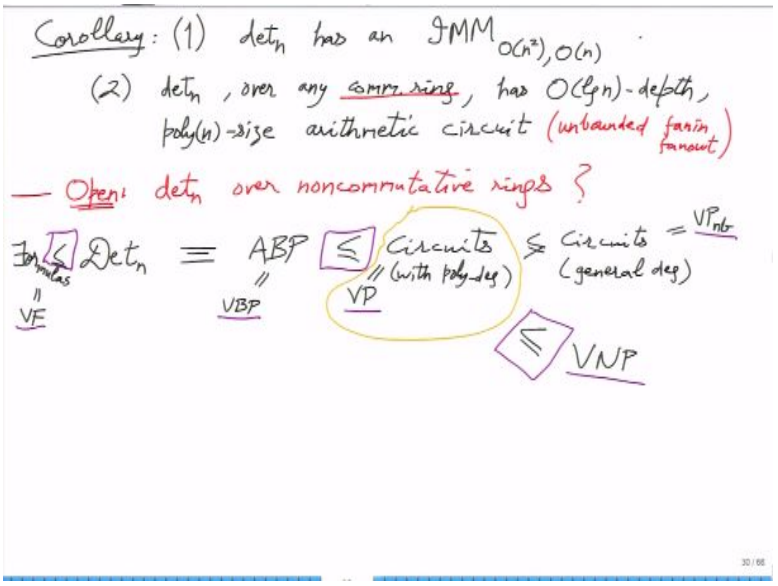
So I think the heads are already fixed. So the first clow we could start with 2 and okay I see! So basically these clow sequences that start with 2 are not being produced here. But will it matter? It would not matter because it does not contribute to determinant. So that is the only small point that we do not really care about all the clow sequences. We just want those clow sequences that have a potential of contributing to determinant.

So those are being produced. So each clow sequence of G with first head equal to 1 corresponds to a unique path in G. So the only thing we have fixed here is the first head. After that you have all the freedom. You can move wherever you want to, but that is fine. Okay, so these the sum of over all the paths of the signed weights will give you determinant. So that finishes the proof.

"Professor - student conversation starts" I mean when we look at any monomial or the determinant we can just start of with sum $x_{1,i}$. So then that is why you are putting sign. But to make such an argument we have to make sure everything cancels right? But that was shown last time, because if anything that is not covered it is bound to get canceled. **"Professor - student conversation ends".** For that you have to observe that a clow sequence that starts with head 2 head equal to 2 under join or break will give a clow sequence of head equal to first head equal to 2.

So in the partition, we are actually removing elements together with their transformation. So on the subset also our map is actually an involution. Yeah, so those are minor points. This basically is done. So what did we learn from this?

(Refer Slide Time: 27:32)



So this is actually a big result, it shows that determinant as a polynomial has an iterated matrix multiplication expression where the matrices are $n^2 \times n^2$ and they are n of them. Determinant can be kind of factored into matrices. So it was an determinant of an $n \times n$ matrix. But these factor matrices that you are multiplying are slightly bigger. They are $n^2 \times n$ square.

So that may be something to optimize, but we do not know whether it can be optimized. It is an open question. This is the best representation for determinant and then second thing when once you convert this IMM into circuit, so since you have n matrices to multiply, right, you can do it by you can do it recursively by having and so that gives you a circuit and what is the depth?

Same as the number of recursive calls right, which will be $\log n$. So you actually get a $\log n$ depth arithmetic circuit that will compute the same thing and its size will obviously be polynomial. So that gives you a very fast fastest possible parallel algorithm. So determinant and there is no division happening. So over any commutative ring has $\log n$ depth, poly n size. If you analyze the size is also not too bad.

It will be, I do not think it will be more than n^5. So you have $n^2 \times n^2$ matrix and you have n of them. So n^5 I think is a trivial bound on the size. It is not too big a circuit,

but the time that it gives you is log n. So this is an improvement even over the first result you saw, where we showed determinant is in VP. There we got lock square n depth. Now this is log n depth.

Slight difference is that here this is unbounded fanin fanout. Unbounded as in it is not bounded by a constant it is not 2. So fanin and fanout are bounded only by the size because when you look at the recursive implementation of iterated matrix multiplication. So when you want depth to be log n, then you actually are forced to add and multiply many things. So addition multiplication gates are actually multiplying many things. Okay.

Maybe multiplication can be bounded the fanin but addition certainly is unbounded. You have to add many, many things. **"Professor - student conversation starts"** Is there anything similar known for permanent? What do you mean? I mean do you have, can I be able to compute permanent like this, permanent. Permanent has a poly time algorithm? No what is the barrier? **"Professor - student conversation ends".**

The barrier is the cancellation. In the proof we had this involution, which was canceling things. So that was happening because of the sign. So that kind of a result we do not have for permanent. We do not have a mathematical result for permanent.
[Student] If we ignore the signs we probably will overcount the clow sequence.

No so then you would just yeah, the more clow sequences you look at, more things will get added here.

It never converges to something meaningful. **"Professor - student conversation starts"** So this was unbounded fanin like it should not be a problem like if we consider as if the bounded as well because if we like bound the fanin then it should not like depth would increase but. Depth will increase. So then you will get the previous result. **"Professor - student conversation ends".**

Depth will become $\log^2 n$. Because if you have if there are n^5 things going inside an addition gate, again by divide and conquer, you can implement this in log n depth, $\log n^5$ depth. So then the depth just gets multiplied by $\log n^5$, which is $O(\log^2 n)$. So then you go back to the previous result, but the previous result does not give this result.

So this is really a stronger result and except the constants in the exponent of size this is really optimal. You cannot improve this. Neither in theory nor in practice. So yeah, but there is a meaningful question for determinant that it has not answered, which is the assumption of commutativity. So what can you say about determinant over non-commutative rings.

So for all practical purposes, this result is good because well you only look at determinant, usually over fields or integral domains. So those are all covered. But if one day you choose to pick your entries in the matrix as being matrices themselves so you are looking at determinant of a matrix whose entries are matrices. So then actually, you are looking at determinant over a non-commutative ring because matrix algebra is non-commutative.

So we do not know results for that except just brute force computation. So it will really depend on the order the way you are multiplying these entries. Yes, so that is a question which is not answered or even considered here in this analysis. So this was so I think, currently it is known that for general non-commutative rings, determinant is as hard as the usual permanent.

So there is I think, no hope for doing this efficiently. But you can still ask for special rings. So I do not know what the results are there. So this completes the discussion about basic models and what they can compute. So this shows that determinant and ABP are the same. And ABPs are contained in circuits.

ABPs are, I mean circuits are at least as strong as ABP, believed to be stronger and where do you think are formulas? On the left yes but that we have not shown. So some time we will show that formulas are on the left. So formulas are weaker than ABP which is equal to determinant and these are weaker than circuits. Circuits also we have two kinds, circuits with bounded, with poly degree.

So that is VP and we also can look at circuits with unbounded degree. I mean general degree okay. So the circuits at poly degree is VP. That is the VP class. And circuits with no assumption on the degree just on size, this is the class VP subscript nb which just stands for not bounded not bounded degree. Yes, so this is currently the summary of what we have learned in terms of complexity classes.

So formulas you can call as defining complexity class VF, okay. Let us say Valiant's formula. ABP you can use it to define a complexity class called VBP, so Valiant's Branching Programs. So these are basically polynomial families for which there is a polynomial families with I mean index by the number of variables n for which there are a ABPs of size poly n, right.

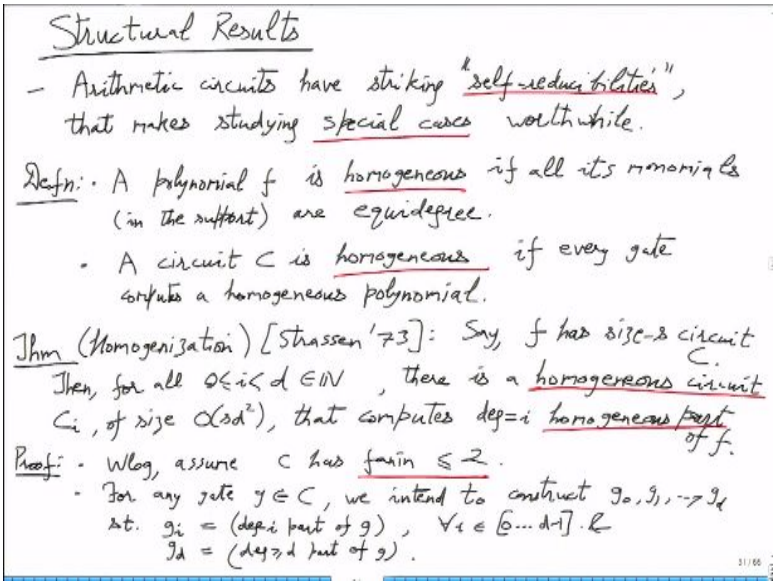
$$VF \leq Det_n \equiv ABP \leq VP \leq VP_{nb}$$

So these are classes of polynomial families, VF, VBP, VP, and VNP. Right, so we have some understanding of how they compare and how powerful they are. But we do not know anything else here. So we do not know whether VF is strictly weaker than VBP, VBP is strictly weaker than VP. Well we know that VP is strictly weaker than VP nb just because of degree, right.

So that we already know by definition. But this is an open question and strictness of these two, the first two inequalities it is an open question. So what we will do now is we will look at some very powerful results, structural results here, okay. So circuits with poly degree, which is VP. So we will now look prove some structural theorems about VP okay which will really be some of them will be very unexpected, okay.

You will not conjecture them looking at the definition just like you will not conjecture a determinant equal to VBP from the definition, with equally unexpected results.

(Refer Slide Time: 40:36)



In the previous thing, there is another branch which is by adding non determinism. So that gives you VNP, right which obviously is our central open question. So these are your complexity classes in this course. You can define more but those will be straightforward definitions. Okay. So one thing about arithmetic circuits that we will see over the next few weeks or months is that they have some striking self-reducibility.

So they have striking self-reducibilities that makes studying very special cases worthwhile. So the term self-reducibility means that in general arithmetic circuits you can convert them into something much simpler: a much simpler arithmetic circuit, a special case of arithmetic circuits. And so then studying these special cases becomes an interesting question. So when you can understand these special cases good enough, then you actually will have results about the general circuit model.

Okay, so these self reducibilities are actually very special to the algebraic model. Their Boolean analogs do not exist. If you try to do the same thing for Turing machines or Boolean circuits, those things are false. So this will really be a

consequence of algebra. So the first thing, first special property that we want to show is, well, so a circuit computes a polynomial, let us say n variate degree d.

So you can talk about the degree d homogeneous part. So homogeneous parts means that you only want those monomials and respective coefficients where the degree is the same, let us say d. So you want to extract those monomials whose degree is d. There may be monomials of degree d -1, d - 2. You want to drop them.

So the question is, if the original circuit has a small size or the original polynomial has small circuit complexity, what can you say about the homogeneous part? Is this also easy to compute? Right, so this is called homogenization. So a polynomial f is homogenous if all its monomials well, when I say all its monomials, I mean those that are in the support. I do not care for those monomials whose coefficient is zero.

So monomials that appear in f with coefficient nonzero they have the same degree, f has equidegree monomials. So then we call it homogenous and a circuit is homogeneous if so how do you want to define this homogeneous circuit? Right so the definition will be syntactic so it should hold for every gate, not at the root. But every gate should be computing homogeneous polynomials.

If every gate computes a homogeneous polynomial. The theorem that we will show is; we will I will call it homogenization property of circuits shown by Strassen. So say f has size as circuit. So then for all i's where d you can think of as the degree of f. So for all for any i you can compute the homogeneous part of f using a homogeneous circuit. So there is a homogenous circuit C_i of size sd square that computes degree i homogeneous part of f.

So for every homogeneous part there is a homogeneous circuit of size slightly bigger, $O(sd^2)$. And in fact, the result is slightly stronger. So I do not need to assume d to be the degree of f, d is just any number. So basically, the circuit may have a very large degree. But if you are interested only in low degree parts they can be computed

efficiently. So the thing becomes expensive only when you start asking for very high degree homogeneous parts.

Anything that is low that can be extracted out in a size sd^2 which is small. How do you show this? It has a simple proof. Since you want a homogenous circuit already, this demands that from the very beginning, so from the leaves you should be computing the homogeneous parts separately. Okay, so at any point, any layer you can assume that you have currently the homogeneous parts available of an intermediate polynomial.

And so when you want to add two such things, you do it using homogeneous parts, I mean, you add the corresponding homogeneous parts to get another homogeneous parts for the sum. And when you want to multiply, then using the definition of polynomial multiplication, you can again do this in a homogeneous way to get the homogeneous parts directly.

No, so it is a, so you have two polynomials g1 and g2 let us say in the original circuit, you had two polynomials g_1 and g_2 which were non homogeneous. So the multiplication gate that is multiplying these two this you have to implement by a homogeneous circuit. So you actually have to break g_1, g_2 into homogenous parts and part by part you have to multiply and output those things by different multiplication and addition gates.

So this is what the simple proof idea is and you implement it just by, formally by induction. So without loss of generality assume C has fanin just 2, if it the fanin is more then convert it into fanin 2.

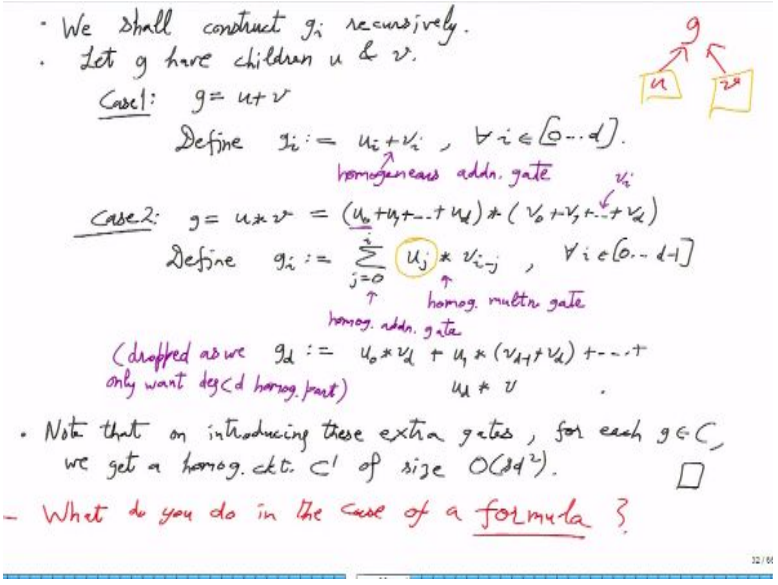
So this will increase the size slightly but not by much. So we will be using this fact. So for any gate g in the circuit we intend to construct g_0, g_1, g_d such that g_i computes degree i part of g. So I want to give a more general proof right. So I have to

say okay let me say that this was strictly less than d. I want things from 0 to d – 1. And what I will do here is up to d -1.

So g_0 to g_{d-1} are indeed the degree i homogeneous parts; g_d is the rest. So degree greater than equal to d part of g. So $[0, 1, \cdots, d-1]$ these homogeneous parts plus the rest which is then the parts of degree d or more. So this is what we want to maintain. So we are actually building from the leaves inductively we are building another circuit C` such that for every gate g in C, there will be an analogous computation happening inside $C$` in a completely homogeneous way.

So think of another C` being built inductively. That is our goal. That is our strategy.

(Refer Slide Time: 54:38)



We shall construct g_i recursively or inductively. So let g have children u and v. So g has fanin is 2 so u and v are the only 2 gates which are feeding into g. Now g itself may be an addition gate or multiplication gate so we have to handle it separately. So case 1 is easy case which is the sum, g is an addition gate. Well so for u you have u_0 to u_d available. For v you have v_0 to v_0 available. You just do the obvious thing. So define $g_i = u_i + v_i$ for $i \in [0, \cdots, d]$.

This is, the property of this is that it is a homogeneous addition. So using addition gates you can compute g_i's now inductively. And how much has the size grown? Not

much, by $O(d)$. Just an additive growth. So this is under control. So next is when you are multiplying, this will grow slightly more because here what will be g_i ?.

These are available to you, u_i 's and v_i 's . So now when you want to compute the ith homogeneous part, what do you do? So u_0 should be multiplied with some v_i. And u_1 with the one behind v_i and u_2 with the one behind and so on. **"Professor - student conversation starts"** And I think we can handle quite a bit of fanin right? Because there this is this x + y = d they may have 3 variables x + y + z = d I mean like that.

No, but they are not just u v, there is u, v, w. Yeah, so once you take something from u and something from v what you take. No, but the number of products exponentially starts blowing up. **"Professor - student conversation ends".** If you are multiplying r things, then you have d to the r many products. So we do not want that actually, we do not want to go there. Let us just do with the fanin 2.

So define g_i as now this convolution

$$g_i = \sum_{j=0}^{i} u_j * v_{i-j} \quad , \quad \forall i \in [0, \cdots, d]$$

This product * is homogeneous. Homogeneous multiplication gate and also the sum is homogeneous. So we have computed all the g_i. The price we pay is for every g_i, we have around $O(d)$ many multiplication gates that we have introduced. And so there are $O(d^2)$ many gates being introduced.

Yes, so as you can see the multiplication fanin is not growing. This remains 2 in the end. But the addition for fanin is d, okay. Yeah actually one has to be careful here also. In the boundary case this should only go up you should only go up to d - 1. Yeah. Because if you start introducing the dth parts yeah there it is inhomogeneous. So but then you can define it easily.

$$g_d = u_0 * v_d + u_1 * (v_{d-1} + v_d) + \cdots + u_d * v_d$$

So the properties of homogeneity are there only for these g_0 up to g_{d-1}; g_d we can compute, but we do not care about homogeneity. Yeah, so I should have rephrased it maybe in a different way. So g_d part is really not homogeneous. Will that be a problem?

Well, okay, so the it is still fair to say that the circuit resulting circuit is homogeneous circuit, because if you are only interested in parts up to d - 1, then at no point of time in your intermediate computation, will you need degree d, or more homogenous part. So actually, g_d could be forgotten. If in the end you do not want to compute, you only want to compute up to d – 1 homogeneous part then at every intermediate computation g_d could be forgotten.

So this part is actually not needed. This can be dropped. If you include g_d I mean g_d also can be computed, but if you include then there is the slight non homogeneity that is being introduced. So note that on introducing these extra gates for each gate in C, we get a circuit C` whose size grows per gate by d^2. So its size is now $O(sd^2)$.

Yeah, so we are computing all the homogeneous parts up to d – 1 using a homogeneous circuit. That was not too difficult. So what do you do with a formula? So you are given a formula which means fanout is one and you are interested in computing all the homogeneous parts as formula. So fanout should remain 1. So you observe that here in our proof, we are using this u_j.

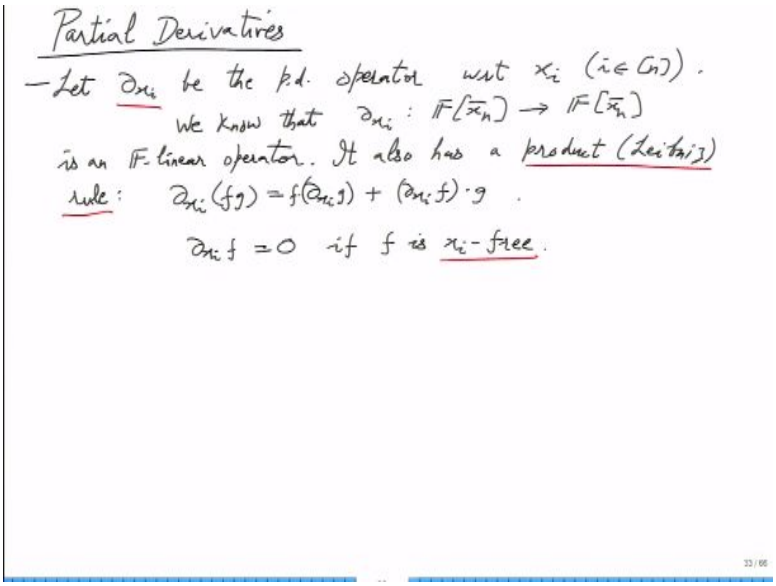
We are assuming that u_j has been computed once here below or well, Here u_j and v_j have been computed and then we are using them again and again. The fanin of u_j is more than one. So that is not allowed in the case of formula. So this proof will fail. Does it mean that you cannot homogenize a formula? So it can be done. Think about that.

It is a proof even simpler than what we did. Okay, so for that I should clarify. You know if your demand is to get a homogeneous formula, then that I think is an open

question. But if you just want a formula for the homogenous part, formula complexity of homogeneous part that you can show is low. But if you also want the resulting formula to be homogeneous then I think it is not clear.

Yeah, so this is a fair question. Do this for formula. Do some things but not everything.

(Refer Slide Time: 1:06:30)



Okay, so we will move to the next property which is partial derivatives, okay. So since we are working with polynomials after you have understood the homogeneous parts you want to understand other operations on polynomials. So the first natural operation is differentiation. So you differentiate let us say with respect to variable x_1.

So what is the complexity of the derivative polynomial and you can differentiate now you can look at higher order derivatives. So like x_1^2 or x_1x_2 or $x_1x_2x_3$. So what is the complexity of these partial derivatives? So let ∂_{x_i} be the partial derivative operator with respect to x_i. So these are the first order derivative, the simplest ones. So we know that ∂_{x_i} it takes n variate polynomials and send them to itself with the same ring.

And this is a linear operator. Why is it linear? Simply because f + g derivative is equal to derivative of f plus derivative of g, that is all. Well, if you take the base field to be

the function field having variables x to say i is 1. So ∂_{x_i} has the property that if you look at the function field with variables $x_2 \cdots x_n$. So over this function field, you are looking at univariates in x_1.

So ∂_{x_1} is actually also $F[x_2, \cdots, x_n]$ linear. So you can also push the other variables inside in that function field also it is actually a linear operator. But it has the same proof. Moreover, it has this really unique property that well so the it preserves essentially the addition operation, but it does not preserve the multiplication operation. There it has a unique, it follows a unique identity which is called Leibniz rule or the product rule.

Leibniz rule is

$$\partial_{x_i}(fg) = f(\partial_{x_i} g) + (\partial_{x_i} f)g$$

This is one nice property of derivatives. And it goes without saying that the third property is if you differentiate a polynomial

$$\partial_{x_i} f = 0 \text{ if } f \text{ is } x_i \text{ - free.}$$

So this also we will be using all the time. Derivative vanishes unless its polynomial depends on x_i.

So what we will do next time is show that these first order derivatives, they can all be computed simultaneously by a circuit whose size is the same size and the same depth as before? Okay, which should be a pretty surprising thing because there are n derivatives first order derivatives, all of them can be computed essentially the same size and depth. Okay. So that is not at all clear from the definition of a circuit. Any questions? Okay.