Parameterized Algorithms
Neeldhara Misra and Saket Saurabh
The Institute of Mathematical Sciences
Indian Institute of Technology – Gandhinagar

Lecture – 10
Vertex Cover above LP

(Video Starts: 00:11) Welcome to the fourth module of the second week on branching algorithms. As you can probably tell already this is a topic that we will be covering in 2 parts, this lecture and the next and the final one. And that is partly because we just have a lot to say about this. This is a problem called vertex cover above LP. So, this is what I was talking about earlier when I said this is going to be old wine in a new bottle.

So, in fact this lecture is the first time that we will talk about a new concept called above guarantee parameterization which is a very natural and a very exciting thing to talk about. So, let us get started here. So, let us begin by bringing back our favorite problem which of course is vertex cover. So, you might remember that a vertex cover of a graph is a subset of vertices whose removal leaves us with no edges.

In other words, it is a subset of vertices that intercepts every single edge in the graph. Computationally, we are typically interested in finding this smallest vertex cover in a graph or typically in the parameterized setting we are given a budget k and we are trying to find a vertex cover that has at most k vertices. Now, if you will recall from the introductory discussions that we had about what makes a good parameter.

Then you might remember that a good parameter is of course a parameter that gives you some workable algorithmic insight into the problem and that allows for hopefully FPT type running times. But a good parameter is also one that you expect to be small in most situations. Now, if I ask you what do you think about the standard parameter for vertex cover which is the solution size that is been what we have worked with all along so far.

Now, without context that is a question that is slightly hard to make sense of what do you mean if it is a good parameter. It seems like a matter of opinion. It is hard to be objective about it. Of course, it is worked out great for us. So, we should not be complaining. It is a

parameter that is given us linear sized kernels. It is a parameter that is given us these really nice branching algorithms that we discussed just earlier this week.

So, what is not to love, we should not be complaining. But, one question to ask ourselves is you know is this a really small parameter in most contexts. So, as a concrete example, let us think about graphs that have perfect matching. So, if a graph has a perfect matching, what can you say about the size of any smallest vertex cover in the graph? Not asking for the exact size because that is going to depend on the graph.

The graph could range from being just a perfect matching to being a complete graph and you know. So, there is going to be a lot of variety. But, can you give me a guaranteed lower bound on the size of the vertex cover for such graphs? Remember that a perfect matching is simply a collection of disjoint edges such that every vertex in the graph participates in one of these edges. So, it is a matching that covers every vertex in the graph.

So, for graphs that do have these perfect matching, what can you say about the size of the smallest vertex cover? While you are thinking about this let me just point out that you can assume for G to have an even number of vertices. And the reason for this is that if you want G to have a perfect matching in the way that we just described it, then G would have to have an even number of vertices.

So, when G has an odd number of vertices people do still talk about these so-called near-perfect matching which are matching that cover every vertex except for one. But that is the sort of detail that is not really relevant to this question. So, do not worry about it. And just assume that the number of vertices in G is even. In this case, you can see that whenever G has a perfect matching its vertex covers are going to have size at least n by 2 where n is the number of vertices in G and n over 2 is the number of edges in any perfect matching.

The logic that drives this inequality is probably exactly what you would expect which is that in general if a graph has say a matching on q edges then any vertex cover in G must have at least q vertices. Because notice that these matching correspond to a collection of disjoint edges and we know that a vertex cover has to cover every edge. And since these edges are disjoint, each one of them is going to demand their own slice of **(()) (05:13)** in the vertex cover.

So, the vertex cover has to have q distinct vertices to even just accommodate for the matching edges. And it probably needs some more to accommodate for the remaining edges. There are of course classes of graphs that are really nice with respect to this relationship between the vertex cover and the matching size. So, you might recall that if you were working with a bipartite graph, then these quantities are in fact the same. But, in general, the vertex cover could be larger than the size of a maximum matching.

And whenever the size of a maximum matching is large, we know that the vertex cover is also bound to be large. Now, the word large in the context of a parameter does not really sound so nice. In fact for all of these graphs that have perfect matching, it looks like our vertex cover parameter which we know and love is beginning to look like it has a nasty dependence on n which means that all of these FPT algorithms that we made just this week begin to look like regular exponential algorithms in n.

Even the kernel sizes, if you go back to the best kernel you had which was a kernel of size 2k, it is just meaningless on this class of graphs. It does not give you any information at all. Now, of course, to all this, you might say, wait. Look, what are we complaining about? Every parameter is going to have a bad day once in a while. There are going to be instances for which the parameter is going to be large relative to n.

We do know this. We expect this. So, what is the big deal? We have just you know made explicit a class of graphs on which this does happen. Is this something to be unhappy about? And to that I would say you are actually right. Maybe, we should not be so upset. But, at the same time, there is here a sort of motivation to pause and ponder about. You know, can we come up with a better parameter. Something that is smaller than this standard parameter.

And hopefully gives us faster running times even on these classes of graphs where the original parameter just happens to be large. Well, one way to think about it is the following. So, this lower bound tells us something. The lower bound tells us that look the vertex cover size is already going to be at least something. It is at least for example in this case we talked about the size of a maximum matching. So, it is already going to be at least that much.

So, the non-trivial work really is to figure out how much more do you need on top of this lower bound. And that is kind of the question that we are going to tackle here. So, in fact, we are going to work with a different lower bound for vertex cover which is again something that you know you have seen in the context of **(()) (08:19)**. So, we have the optimal vertex cover which I am going to denote by VC.

And you also have the optimal value of the linear program that we wrote for vertex cover which is VC star. Now, between the two, can you pin down an inequality which one of these is always guaranteed to be at least as large as the other? Let me clean this up a bit. So, here are the 2 competing entities. And I would like to know which one is the smaller one. So, does this go this way? Does this go this way? Actually that is upside down.

If you remember from last time the value of the LP OPT of course is going to be at most the value of the real OPT the actual value of the vertex cover. And the reason for that is that you are optimizing over a more liberal space. You are allowing for more values than what you would be allowed if you were working in the discrete setting. So, we know that any vertex cover is bound to be at least a VC star of G for sure.

The question that we are interested in posing is how much more? And that how much more is what we want to manifest as our parameters. So, we are still looking at the same problem as before. We want to know, does G have a vertex cover of size at most k? But, this time we want to parameterize by the difference between k and the LP OPT. So, this is no longer the standard parameter. This is what we call an above guarantee parameterization.

And hopefully that name makes sense. Because the parameter here is really focused on how much more do you need to form a vertex cover above what you know you are going to need for sure. So, it is above a certain guarantee. Now, you could pick your favorite guarantee. You could have multiple lower bounds in the size of a minimum vertex cover. We have already seen that the size of a maximum matching is also a lower bound and so is the size of or the value of the LP OPT.

It turns out that between these 2 and you can try to check this for yourself between these 2 the value of the LP OPT is a stronger lower bound. In that it is always at least the size of a maximum matching. So, if you are parameterized above a higher lower bound your parameter

is going to be smaller. So, that is the one that we are going to be working with. And we want to see if this problem is FPT.

It is not obvious that vertex covers FPT when you look at it through this very new lens. So, that is going to be the subject of this lecture. And this is the main result that we want to prove. We want to say that there is in fact an algorithm for vertex cover above LP in that runs **(()) (11:34)** 4 to the k minus LP OPT with a polynomial overhead as usual. So, it turns out that I think this is a very interesting result on its own right.

But what makes this even more crazy is that there are a lot of other problems that look nothing like vertex cover. Forget vertex cover. These are problems that are not even problems about graphs. But they happen to reduce to the problem of vertex cover parameterized by k minus VC OPT. So, that is very exciting. And those are applications that I promise we will get into in the next video.

But for those applications to work, we need this interesting algorithm here. So, hopefully you are all motivated to talk about this FPT algorithm. So, that is what we are going to do now. Before getting into the algorithm though, let us just recall the LP formulation for vertex cover. Just to fix up some notation, you have already seen this in the last lecture of the first week. So, this is just going to be a quick recap.

So, remember that we introduced a variable for every vertex in the graph. And we have a constraint for every edge. And the variables take values between 0 and 1. And the constraints ensure that if you have a pair of variables that represent the endpoints of some edge in the graph then the values of the corresponding variables must add up to at least 1. So, that is the LP for vertex cover.

Now, suppose somebody gives us a solution we can think of this as a vector X with n coordinates. And for convenience in fact you can think of this vector as being indexed by the vertex set of the graph. So, you could be talking about the value x sub v to denote the value of the variable that is representing the vertex p from the graph G. Also, we will use W of x to denote the sum of all of these values.

And we will refer to this as the cost of the solution x or the weight of the solution x. And we will tend to use these 2 terms interchangeably. Let me also point out that the all half solution is always going to be feasible for the LP. So, if you set every variable to half all the constraints are satisfied. Of course, this may not be an optimal solution. In fact, it may be quite a bit bigger than the optimal solution.

But, it certainly serves as a valid upper bound for the value of LP OPT. In other words, LP OPT never needs to be more than n by 2. Finally, let us also recall that we claimed that there is always a half integral solution to the vertex cover LP. In other words, we have that there is always an optimal solution where the value of every variable is either 0, half, or 1. Such a solution is called a half integral solution.

And it can be found in m root n time where m is the number of edges in the graph and n is the number of vertices. So, let us just keep all this at the back of our minds. And now, we are ready to start thinking about the algorithm. We are still solving vertex cover but with respect to this brave new parameter which is trying to be much smaller than the previously understood standard parameter.

Having said that a good place to start generally speaking is familiar territory. So, we have recently seen a branching algorithm for vertex cover. So, let us just quickly recall what we did back then. The strategy was essentially to branch on high degree vertices whereby high degree I just mean vertices **(()) (15:01)** degree is at least 3. Now, what we did was the following. You pick a vertex of maximum degree.

If there are multiple vertices that have the highest degree just pick any one of them. And then you branch on that vertex which is to say that you explore the following 2 exhaustive scenarios. In the first, you say, you are going to pick this vertex in the solution. So, you generate the graph G minus v. And there you look for a vertex cover of size k minus 1. In this branch, the measure which back then was just the size of the solution drops by 1.

Now, in the other branch, we said that we will not pick this vertex in our solution. But that had the immediate implication that we had to pick all of its neighbors in the solution. So, in this case, the measure again, the size of the vertex cover dropped by at least 3. And in

particular, it dropped by the degree of the vertex that we were branching on. So, we then went on to analyze this algorithm. Look at some base cases and so on and so forth.

And those details are not going to be as relevant immediately. But what we want to do is really think about whether we can actually leverage the algorithm that we already know. So, let us analyze this algorithm with respect to our brand new parameter. And in particular, to keep life simple, let us just focus on the left branch where we are just getting rid of one vertex. So, what do we want to do here?

What does it mean to analyze this algorithm with respect to this parameter? Well, we want to see, how this parameter evolves as we make this change in this graph? So, let us actually take a look at this. Little bit visually. So, here we have k and the value of LP OPT. And notice that the gap between them this highlighted region right here. That is in fact our parameter. So, if you want the running time of our branching algorithm to be bounded as a function of this parameter then we need to show that as the instances evolve in the branching algorithm.

This parameter this gap actually shrinks. So, we already know that k is going to reduce by 1. So, to understand, what happens to this gap? How does it shift? We need to understand, how the value of LP OPT evolves as we modify the graph? So, let us take a look at a couple of possible scenarios. First, let us say that k reduces by 1 as we said that it would. But, now, let us say LP OPT happens to remain the same.

For whatever reason, after deleting a vertex LP OPT does not change. In this case, notice that what happened was that k in fact came closer to this fixed value of LP OPT. So, the gap in fact shrunk which of course is very good for us. We do want this parameter to reduce and that is happening. So, this is a good case for us. Now, just for contrast, let us take a look at a slightly different scenario.

So, here as before k drops by 1 that is just how the branching algorithm works. But, after deleting this one vertex v, let us say that LP OPT dropped rather drastically. So, let us say in particular that it dropped by 2. Now, I am not sure if this picture is to scale. But, you can double check whatever we are saying with the arithmetic if you are finding the scales misleading in the picture. So, in any case, if this happens, is this good for branching?

So, let us think about this for a second. By the way, you might already be alarmed by the prospect of LP OPT decreasing by 2. You might be thinking, can this even happen? And we will address that in a moment. But, this is just a hypothetical. Suppose, it happened then would this be good for us or would this be bad for us. Well, hopefully, you can see that whenever this happens the gap actually increases.

So, this is not good at all. You have gone further down in your branching. But, your measure has gone up somehow. Of course, you may not have a measure which is exactly the parameter. It may be some sophisticated function of the parameter. But, in general, intuitively if your parameter is increasing as you are making progress in your algorithm that sounds somehow like not a good thing to happen.

So, I think we can agree at least at an intuitive level that we want to avoid this kind of scenario. Now, stepping back even further it should be hopefully becoming evident that to understand how the branching impacts our parameter of interest which is k minus LP OPT. We really need to understand how LP OPT itself evolves as the graph changes. So, let us try and get the hang of that.

And then we will try and see if we can always remain within some good scenario. And hopefully get to the branching algorithm that we are looking for. So, let us try to get the hang of this by just asking ourselves some questions. Remember that we are trying to compare the LP OPT of the graph G that we started with the LP OPT of the same graph G but with a single vertex removed because that is what is going on in our branching algorithm.

So, the first question that I want to pose is whether the value of LP OPT can increase as we go from G to G minus v. Actually, if the value of the LP OPT increased that would be a very good thing for us. Because we have that k is decreasing and if LP OPT also went up notice that the gap would shrink even faster which seems like a very desirable thing. So, can this lucky situation actually manifest?

Take a moment to think about this and come back when you are ready. The sad answer to this question is no. The LP OPT cannot increase after you delete a vertex. And the reason is pretty intuitive. So, suppose you have an optimal solution for the original graph G. Now, let us look

at the vertex v that was removed. It takes some value here. Without loss of generality, we could be considering a half integral solution.

But, it does not really matter. Because, in any case, we know that x v takes a value that is between 0 and 1. Now, let us just block out x v from this vector. And notice that what remains is actually a feasible solution for G minus v. You can check this for yourself. It is fairly easy to verify. But, now, if this is a feasible solution then we know that the value of VC star of G minus v is bounded above by the cost of this solution because if nothing else we can always fall back on this.

So, the value of LP OPT for G minus v will definitely not be more than this. And now, how large can this solution be? Well, we started with VC star of G the LP OPT of G. And we actually took something away from it. So, if anything this should be smaller than VC star of G. But, the largest that it can be is if x sub v happen to be 0 then we did not subtract anything at all. So, this solution can be as large as VC star of G. But, it certainly will not be any larger.

And therefore, the value of LP OPT can certainly not increase as you transition from G to G minus v. The next question we want to ask ourselves is the opposite of what we just asked before. Can the value of LP OPT decrease after you delete a vertex? Let us just again take a moment to think about this. Unlike before the answer to this question is in fact yes. The value of LP OPT can decrease after deleting a vertex.

And here is a simple example where this can happen. So, consider a graph which is just a star you can check that the LP OPT for such a graph is actually 1. There is at least 1 edge that needs to be covered. So, there is at least 1 constraint that forces a couple of variables to add up to at least 1. As a result, LP OPT is 1. But, imagine deleting the center of the star, then you get an empty graph. And there clearly LP OPT is 0.

So, the value of LP OPT can indeed decrease after deleting a vertex. However, let us think about whether it can decrease a lot. So, for instance, the bad situation that we were describing a few moments ago was when LP OPT decreased by rather a lot. Now, with the example of the star we have already seen that the LP OPT can decrease by as much as 1 but let us ask ourselves if it can decrease anymore.

So, in that example, we said it goes down by 2. Let us ask ourselves if that can really happen. So, specifically the question here is, can the value of LP OPT decrease by more than 1 after you delete a vertex? Once again, take a moment to think about this and come back to this when you are ready. So, it turns out spoiler alert that the answer to this question is no. So, the value of LP OPT can decrease by 1 but not more than 1. And again let us see, why?

So, let us take a look at the LP OPT for G minus v. So, this is a bunch of values for all the vertices in G except for v. And now, let us append to this solution a value for the vertex v. And let us say that value is 1. Again you can check that this solution is in fact a feasible solution for the LP OPT of G. The reason is essentially that you only have to worry about constraints that the vertex v is involved in.

But, by setting v to 1, you automatically take care of those constraints. And notice that the cost of this solution is essentially 1 plus whatever we had before which was essentially VC star of G minus v or the value of the LP OPT for G minus v. So, now, suppose that the LP OPT did drop by more than 1, then that means that we in fact have a solution whose value is less than VC star of G. But is in fact a valid solution for the LP with respect to G.

So, if the LP OPT drops by more than 1, we can use a solution that witnesses this magical drop. And we can extend it to a solution that in fact beats the LP OPT in the original graph which just to be absolutely clear is a contradiction in plain sight. So, the LP OPT cannot drop by more than 1. So, let us summarize everything that we have learned so far. So, the LP OPT certainly cannot increase and it can drop but not by more than 1.

So, the way LP OPT evolves as you go from G to G minus v is that it may reduce by anything between 0 and 1 inclusive. And if you appeal to the half integrality of the solutions then in fact you know that there is going to be a decrease of either nothing at all or half or 1. So, let us just keep this in mind and ask ourselves what is it that we need to worry about knowing what we know now. What is the bad situation for our branching algorithm?

So, we know that LP OPT can decrease by at most 1. If it decreases by anything that is less than 1, then we still manage to actually shrink the gap. So, if it does not decrease at all then there is a nice reduction in the gap. But, if it decreases by anything that is less than 1, then you still maintain something of a reduction in the gap. So, what is the dangerous situation?

Hopefully, you have also come to this conclusion that the dangerous situation is when LP OPT drops by exactly 1.

When LP OPT drops by exactly 1, what happens is that both k and LP OPT move back the same amount. And therefore, the gap remains exactly as it was before. And our branching algorithm is seemingly not making any progress. So, this is the scenario that we have to worry about. And somehow deal with. At this point, I would say that we really need some new idea to make progress.

I guess it is fair to say that we have more or less exhausted the scope of what we can get out of looking at the algorithm that we already know. Analyzing it from the point of view of this new parameter and perhaps secretly hoping that it would just work. Although, I have to say that I think it was a pretty good start. I think almost everything looked like it was working out great except for this one really annoying corner case which was when LP OPT drops by exactly 1.

And of course, this is annoying for the first branch where we delete the vertex v. But, you can imagine that there is an analogous problem in the other branch where you delete the neighborhood of v and k drops by the degree of v. There you would be annoyed if LP OPT also dropped exactly by the degree of v. So, these are really the 2 edge cases that we have to worry about.

I think if we could somehow tackle these then we are very much back in business. So, let us think about what happens when LP OPT manages to drop by exactly 1 which notice by the way is the maximum drop that LP OPT can experience. So, it seems like maybe this is some sort of an extremely situation and maybe instances where this actually happens. They are kind of special.

And perhaps we can tackle them with some sort of a pre-processing rule and just get rid of the case entirely. So, let us actually explore this thought. So, what can we say about graphs where when you delete a vertex the LP OPT drops exactly by 1? To make things concrete let us say that we are deleting a specific vertex v. And in G minus v, let us say this is some optimal solution for LPVC of G minus v.

Now, from our previous discussion, we know that we can extend this to a solution for LPVC of G by bringing back the vertex v and setting the variable x v to 1. We already know that this is a feasible solution. But, in fact, in this situation, this is also an optimal solution. And the reason for that is that the cost for this part of the solution is LPVC of G minus 1. Because, remember we are in the case when LP dropped by 1.

So, when we add 1 back to that the total cost of the solution goes back to just being LP OPT. So, this tells us something about the graph G. Specifically, what it tells us is that in the space of optimal solutions for LPVC of G, there is at least 1 which has at least one variable set to 1. Now, that was a lot of ones in that sentence. But hopefully, the statement is clear. Basically, what we are saying is that if there is a vertex whose removal causes the LP OPT to drop fully by 1.

Then there is some optimal solution for the original graph where some variable is set to 1. So, this is the situation that is problematic for us. So, what if we could create a situation where the original graphs LP had no optimal solution where some variable was set to 1. So, what is the intuition here? If you go back to the LP based kernel for vertex cover, you will remember that we somehow had a forcing strategy for variables that took the value 1.

So, it seems like as long as we have optimal solutions with variables being set to 1, there is still some remaining opportunity to clean up the graph and force some variables. So, this is exactly the idea that we want to formalize now. And now, you should be getting some flashbacks to the LP based kernel for vertex cover. So, if you remember what we did back then was that we worked with an LP OPT solution to classify the vertices into 3 major groups.

Those vertices which took on the value of exactly half and in that lecture we talked about variables whose value was more than half and less than half. However, notice that we can afford to always start with a half integral solution to begin with. So, in my picture every variable that is strictly greater than half is in fact set to 1. And every variable that is less than half takes on the value 0. And we can say this without loss of generality.

So, with that out of the way, let us recall the reduction rule itself. Remember what we said was that we can delete the vertices of v naught and include the vertices of v 1 in our solution

leaving us just with the graph on the vertices of v half. So, the reduced instance looks like the graph induced on v half with a reduced parameter k minus the size of v 1. So, this seems promising in general.

Maybe as long as there is one optimal solution with at least one variable set to 1. We just try to get to the breakup that you can see in the picture on the left and actually make progress. Now, there are 2 issues with this that we need to be careful about. First of all, it is possible that there is some solution which has at least one variable set to 1. But, when you solve the LP, you get the all half solution.

So, at this point, your reduction rule will actually get stuck. And it will show no signs of progress. But, in fact, you are still in trouble. And you need to do something about it. Now, the second issue is a little more subtle. And we will come back to it later. But, I just want to flag it now. So, you are aware. This issue is about what happens to the parameter when you apply this reduction rule.

So, remember when we are branching, we needed to be careful about ensuring that our parameter actually strictly reduces. That is why we are having all this discussion in the first place. But, now, if you think about these reduction rules they are being applied recursively at every step of the branching program. Now, at some point if the application of a reduction rule increases your parameter, then you have again undone all the work that you have done so far.

And it will become very difficult to control the depth of the branching algorithm. So, just keep this in mind but first things first. Let us try and understand how will we ensure that we are making progress as long as there exists some solution in which some variable is being set to 1? The answer to that question is going to be this stronger claim. So, what we can show is the following.

For a graph G, we can either always find a optimum half integral solution that is different from the all half solution or we can correctly conclude that the all half solution is in fact the unique optimum solution. So, notice why this is useful. As long as you are in this situation here, you can apply the reduction rule from before. Of course, this is assuming that we can prove that the parameter does not increase when we apply this rule.

As I said earlier, we will come back to that. But, if you are not in this situation if you are stuck with respect to this lemma then well you are not really stuck because if you can correctly conclude that all halves is the unique optimal solution then essentially you never get into this danger zone. So, this was the danger zone for the branching algorithm when the LP OPT dropped exactly by 1.

But, now, notice that if all halves is the unique optimum, LP OPT cannot drop by 1 because that is what we just argued a couple of minutes ago. So, this is something that is useful to prove. So, let us go ahead and try to prove this. The high level idea is going to be the following. We will try to come up with some LP OPT. And if that LP OPT is already of the sort that we want which is that it is different from all half then we are pretty much done already.

But, if it is the all half solution, then we need to probe it more. So, the way we probe it more is by essentially trying to set variables to 1. Forcing variables to 1 and seeing if things work out. If after forcing a particular variable to 1, we can get an a solution for the rest which actually adds up all together to LP OPT. Then, we are again in business. But, if this does not work out for any variable then we can actually conclude that all halves is the only optimal solution.

So, let us take a closer look at the strategy. You might want to make a note of the statement that we want to prove. Because that is something that I am going to push out of the screen just to make some room. So, the first step is to actually solve LPVC of G and get some optimal solution. And if this optimal solution is already different from the all half solution then we are immediately done.

But, if it is not, then we are still left with the question of whether there is a solution that is different from the all half solution. So, let us think about this a little bit. Suppose we try setting every variable to 1 in turn. So, in particular, what we are going to do separately for each vertex in G we are going to eliminate the vertex v from G and solve for the rest. And let us come back and consider the solution x which is an extension of the LP OPT for the smaller graph by setting x v to 1.

So, that is what we are going to do. And as we have discussed a few times by now, we already know that this is an feasible solution for the original graph for sure. But, the question is, is it optimal? So, the cost of this solution is 1 plus the cost of the optimal solution for the smaller graph. So, you know if this is optimal then what can you say. Well, this is optimal means that we are again done because we have an optimal solution that is different from the all half solution.

And that is again because at least if nothing else, x v is being set to 1. So, again we are good if this solution for G minus v extended with x v set to 1 happens to be optimal for any v. So, then that is a good situation. So, let me just summarize that in words. Forcing a variable to 1 and you know extending the solution for G minus v that turns out to be optimal it is anyway feasible then we are good.

So, suppose this did not work out for every variable we forced it to 1. And what happened was that when you combined it with the LP OPT for the smaller graph you got back something that was bigger than VC star of G. So, we are in this situation apparently stuck. So, this is what is happening for every vertex v, VC star of G minus v plus 1 is greater than VC star of G. So, in this case, we are also done.

Because, it turns out that in this situation all halves is actually the unique optimal solution for LPVC of G. And that is probably not completely obvious although it may be intuitive. But let us just take a look at the formal argument. So, recall again that this is the premise for every vertex v we know that the LP OPT of G minus v was too big. It was certainly bigger than the LP OPT of G minus 1.

Now, notice that because these are LPs which have half integral optimums. If you double these LP OPTs you get 2 integers. And because of that this inequality can be tightened a little bit further. So, VC star of G minus v being strictly greater than VC star of G minus 1. Actually means that it also has to be at least VC star of G minus half. Not just VC star of G minus 1. You can imagine that this quantity here is half of an integer.

So, you know you can only fall back so far. You have to be strictly ahead of this. So, you have to be still at least you know this quantity plus half which is VC star of G minus half. So, now, let us prove what we want to prove by contradiction. So, we want to show that there is

you know no optimal solution different from the all half solution. All halves is the unique OPT that is what we want to say.

So, suppose not. So, suppose there is something that is different from all halves then such a solution must have at least one variable whose value is strictly greater than half and that is just a counting argument. Clearly, if you are different from all halves and knowing that all halves is optimal it cannot be that it is different just by some variables being set to 0. Because then you have a solution that is smaller than the all half solution which is optimal.

So, that is not possible. So, there must be something that is greater than a half to balance out anything that might be less than half. But, now, suppose you pick this variable or the vertex corresponding to this variable for deletion. So, you have an optimal solution where something was greater than half. Let us actually delete that vertex and restrict this solution to just G minus v.

So, notice that because you have taken away more than half the solution restricted to G minus v. Just the original presumed solution that is different from the all half solution and is optimal. So, that restricted to G minus v will be less than the LP OPT for G minus half. Because you started with LP OPT and you took away something that was more than half. But, now, this contradicts this inequality here. And that is why you are done.

So, what we have shown with this argument is the fact that you know we can always assume that we have landed at a situation where all halves is the unique optimal solution. Once again we need to make sure that this reduction rule does not increase the parameters. So, let us quickly take a look at that argument as well. So, remember that this is what we are working with. So, we have this partition of the graph into v naught v half and v 1.

And you have deleted v naught plus v 1 vertices. And from our argument from before remember we said that when you delete one vertex LP OPT can drop by at most 1. So, let us say we have p and q vertices here. So, when you delete these p plus q vertices the LP OPT can drop by at most p plus q. But notice, what is happening to k? So, k becomes k minus the size of v 1 but we have LP OPT could potentially drop by as much as v 1 plus v naught.

Because that is how many vertices we are deleting out here. So, that is based on our simple argument from before that when you delete one vertex you know you never drop by more than 1. But now, this is not really good enough for us to know because you know we have k minus VC star of G. And now, we have k minus v 1 minus VC star of let us say G prime which is essentially the original graph G projected on the vertices that take on the value half.

So, now, we really need for LP OPT to drop by at most v 1. If we were to ensure that the parameter does not increase because if the LP drops by anything more, then the gap between k and LP OPT in these respective graphs would actually go up. So, now, what is the argument for the fact that LP OPT does not drop by too much? So, let us make this claim that LP OPT drops by at most v 1 as you transition from G to G prime.

Well, suppose LP OPT dropped by more than v 1. So, hopefully with all the arguments that we have made so far you can see where this is going to go. So, feel free to pause this video at this point to think about the solution. But I am gonna actually try and eliminate literally v naught and v 1 and talk about what happens. So, we are left with the graph v half here. And let us say this has a really small LP OPT.

So, it drops by even more than v 1. So, using such an LP OPT which where the value has dropped dramatically we can try and construct a solution for the original graph G which is even better than VC star of G which of course would be a contradiction. So, let us use the the optimal solution. So, suppose for the sake of contradiction that there is a x dash whose value is strictly less than k minus v 1.

So, that is an x x prime which is a valid solution for v half. It satisfies all the constraints here. Now, let us bring back what we deleted which is v 1 and v naught. And let us set these variables to 0 and these variables to 1. So, when you obtain a solution x like that its weight is going to be strictly less than k minus v 1 plus v 1 because we have only said size of v 1 many variables to 1. And, is this a feasible solution? Well, yes.

Because notice that again the setting of variables in x prime satisfy all the constraints that involve vertices which are both and v half and the setting of vertices in v 1 to 1 ensures that you meet all the constraints which involve vertices of v 1. And notice that vertices of v

naught to begin with when never involved in any constraints involving other vertices from v v naught or v half. So, as a result, this is a feasible extension.

But it is an extension whose cost is even better than the original LP OPT. But that means that we have actually a contradiction. So, the value of LP OPT cannot drop by more than the size of v 1 as you make this transition from G to G prime. So, this is a perfectly valid reduction rule to apply for as long as you can. And the number of times that you apply this reduction rule is at most the number of vertices that are there in the graph.

Because we need to just keep trying to force each one of these variables corresponding to each vertex to 1 to make sure that we have eliminated the possibility that there is even a single optimal solution which sets a variable to 1. So, once we have done all of this we can be confident that LP OPT does not drop by as much as 1. And therefore, our branching is going to in fact shrink the parameter. So, we have argued all this for the case when the branching gets into the instance G minus v.

In the other case, the branching gets into G minus the closed neighborhood of v and the argument there is very similar. So, I encourage you to think about it and work out why the measure strictly shrinks essentially in the right branch as well. Now, before summarizing everything that we have learned so far, I just want to quickly talk about the most important part of any recursive algorithm which is the base case the thing that ensures that our algorithm actually terminates.

Now, usually a good place to stop for any branching algorithm is when the measure that you have defined for it hits 0 because that is what will allow you to bound the depth of the branching as a function of the measure. Now, we have not explicitly defined the measure for the algorithm that we have right now. But you have probably guessed that the measure should simply be the parameter. Because this is what we have been focused on all along.

This is the thing that we argued will shrink when you get into the 2 branches. This is the thing that we said will not increase whenever we apply the reduction rules. So, it is a very natural choice for the measure. So, we in fact see that as you go down the branching the measure drops by at least half in both of these branches because we know that of course k reduces by 1. And in this case, k reduces by the degree of the vertex that is being deleted.

These were the 2 branches that we had if you remember. In the left branch, we remove a vertex v. And in the right branch, we remove the closed neighborhood of the vertex v. So, that is what we have. So, that is what k drops by. And we have ensured that the LP OPT on the other hand in this branch never drops by 1. And in this branch, never drops by the degree. But because we are dealing with LPs that are half integral if it does not drop by 1 then it drops at most by half.

And here also you can work out that the LP OPT drops by at most d of v minus half which means that overall the measure mu shrinks by at least half. And that happens in both branches. So, if you are able to show that when the measure hits 0 then the algorithm terminates. So, we can figure out a way for the algorithm to terminate. Then we are essentially done because in each branch we have that the measure drops by at least half.

So, the number of steps that this algorithm can last on a longest route to leaf path in this branching tree is going to be at most 2 times the measure that you had to begin with. Just to make this concrete with some silly numbers, let us say that the measure was 10 to begin with. And you know that it reduces by at least a half in every step. Maybe in some steps you get lucky. You might have a bigger drop.

But you definitely guaranteed that even in the worst case every time you spawn another layer of the branching tree. Every time you go one step deeper into the branching the measure has dropped by at least half. This means that the branching can last for at most 20 steps in any path from root to leaf. So, hopefully that is clear. And in fact that is what leads you to the theorem that we set out to prove.

So, this running time comes essentially from saying that you have 2 branches. So, you have 2 raised to the depth of this branching. And the depth of this branching is bounded by 2 times the measure. So, it is really 2 to the 2 times k minus LP OPT of G which works out to 4 to the parameter. So, that is our theorem. And we are pretty much done if we can figure out this stopping criteria. So, we do want the algorithm to stop when the measure hits 0.

Otherwise, this analysis is not complete. So, what can we say when the measure hits 0. What does this mean? This means that the value of k actually matches with the value of the LP

OPT. So, k equals LP OPT. Now, let us think about after applying the reduction rule that we had in mind. Can we really go down another step in the branching algorithm? Does it make sense? So, we have k and VC star of G being in the same place.

And if we were to branch in either of these directions let us say on the left. So, I mean it is possible that unlike in the previous case when k equals to 0 you could actually just look at the graph and say well if the graph is not empty say no. And if the graph is empty then say yes. Because then our measure was just k it was the standard parameter. But right now, you could be in a situation where the measure is 0 but the graph is still non-empty.

And there is something going on here. So, we need a slightly different argument. So, again let us think about what happens when let us say we try to branch further after having reduced the graph. So, let us say we are trying to remove a vertex. Let us say there is some vertex that is available whose degree is at least 3 and so on. So, if we do that then k certainly becomes k minus 1.

But LP OPT because of our reduction rule we are assured that LP OPT is not going to drop all the way by 1. So, this can only drop by a little bit less than 1. But notice that something very funny has happened right now. So, if you look at this picture you should see a red flag already. So, this means that in the graph G minus v we are hoping to get a vertex cover of size k minus 1. But we have a LP OPT which is larger than k minus 1.

But we do know that any vertex cover must be at least the size of any optimal vertex cover must be at least the LP OPT. And this should be G prime. Sorry about that. So, G prime being G minus v. So, since we know that the optimal vertex cover size must be larger than LP OPT, we know that we have run out of budget at this point. We do not have enough budget to actually create an optimal vertex cover.

So, after the reduction rules have done their job if the graph is not already fully resolved then you can actually say no. It is going to be the same argument in the other branch. So, there is really no point in getting into either of these branches. Because you know for sure that you are headed into no instances. So, you can pretty much roll up your sleeves and say no if you are stuck with a graph that is not fully resolved even after the application of your reduction rules.

If this was a yes instance then it would be completely tackled by the reduction rules that you have in place. So, that essentially is a summary of the whole algorithm let us actually recap everything once because there was quite a bit that was going on here. So, remember, we were working with vertex cover above LP which is essentially the same old vertex cover problem but with a different measure.

And the measure was how much do we need beyond the LP OPT which was a very natural lower bound for the size of the vertex cover. And our whole discussion really was focused on this idea that we reuse a branching algorithm that we have already seen before. So, this was the branching algorithm that just branched on high degree vertices. But our main focus was really on ensuring that this measure actually dropped in both branches.

And so, it was a tension between how k is changing and how the LP OPT is changing. So, we made a few observations about the way this gap parameter evolves. And to understand how the gap changes, all we needed to do is to understand how LP OPT changes. So, we saw that after deleting a vertex LP OPT cannot increase and it cannot decrease by more than 1. So, the dangerous situation for us was when it actually dropped by exactly 1 because in that case the gap remained the same.

So, we did know from before that we can always work with half integral solutions that can be found in polynomial time. And we also had a reduction rule that could get rid of anything that did not look like half. So, we had a way of pruning out vertices which took values 0 or 1 in any optimal half integral solution. But we realized that this was not enough to ensure that we can make progress in the branching.

So, we proved a stronger version of this claim which was to say that you can either find something that is different from the all half solution or you can actually conclude that all halves is in fact the unique optimal solution. And after that we were back in business. So, we could just keep applying a reduction rule for as long as we could. And it was not just a one-shot application.

So, this lemma had an algorithmic proof and that is what actually is the reduction rule as well. So, you have to try forcing every vertex to 1. Pretend that it takes the value 1 and then

solve the rest of the instance. And see if you can get to a feasible solution for the entire graph where some variable is set to 1. It is only when this does not work out for every single vertex that you can conclude that in fact all halves is the unique optimal solution.

So, the time that it takes to apply this reduction rule is essentially n times m root n. Because you just have to keep finding feasible solutions for G minus v. But you have to try this for every single v. So, that is just a overhead of n on top of this running time. So, it is a polynomial time reduction rule which we argued does not cause the parameter to increase. It is again important that you show this explicitly especially when you are dealing with an above guarantee parameter.

It is not obvious often that these reduction rules are safe in this sense. So, you do that and once you are assured that the LP OPT is not going to drop dangerously. You can now get into the branching. The branching is the same as before. And then we finally had fact that your measure decreases by at least half in each branch. So, the depth of your branching is going to be at most 2 times the measure. We also argued that you can stop once the measure hits 0.

And that gave us finally the result that we were looking for that vertex cover above LP OPT is FPT. And it has an algorithm whose running time is 4 to the k minus LP OPT. So, next time, we will see how we can make use of this algorithm to solve some other problems which are interesting on their own right. But we will not solve them directly. We will simply reduce or transform those problems into instances of vertex cover parameterized above LP OPT.

That is going to be a lot of fun. And you will hopefully be convinced that all of this hard work was worth it. Not just for solving vertex cover above LP OPT on its own right but also for all of these cool applications that we more or less get for free. So, I hope you enjoyed being introduced to this important concept of an above guarantee parameterization. And seeing how it leads to even faster and even better algorithms for our favorite problem vertex cover.

So, this is a good place to wind down. So, I would like to thank you for watching. And, I will see you in the next video. **(Video Ends: 59:57)**