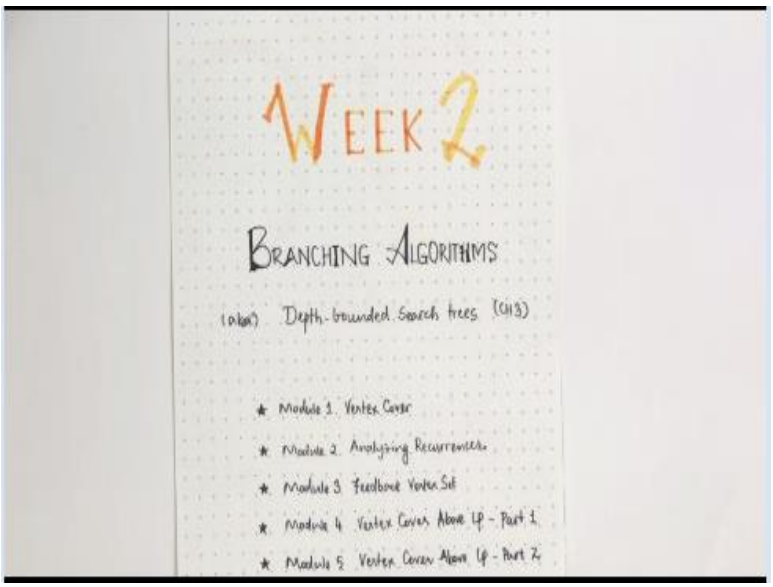
Parameterized Algorithms
Neeldhara Misra and Saket Saurabh
The Institute of Mathematical Sciences
Indian Institute of Technology – Gandhinagar

Lecture - 11
Applications of Vertex Cover above Matching

(Refer Slide Time: 00:11)



All right, so, welcome back to the final module of the second week on branching algorithms. So, this time, we will be looking at some applications of the algorithm that we developed in the previous lecture, where we were focused on a new above guarantee parameterization for vertex cover. And back then we promised that we will show you why this algorithm can also be used to solve some other problems that do not really look like vertex cover at all.

So, those are the problems that we will be meeting in this lecture, they will be 2 brand new problems that you have not seen so far. And this is going to be very exciting. In case, you have not seen the previous video, please do watch that one before coming here, at least to the point where you are comfortable with what we did show there, because we are going to just assume that as a black box for this video.

And before we get started, I want to introduce you to a slightly different above guarantee parameterization for vertex cover. **(Video Starts: 01:08)** And this is the problem of vertex cover above matching. So, here the parameter we want to look at is k – mu of G, where mu of

G is the size of a maximum matching. If you remember, we mentioned this parameterization in passing in the previous video.

And I said that if you parameterize above LP OPT that will be in some sense, stronger parameter, a stronger parameter and codes. Let us explore what that means. Right? So, suppose you have 2 parameters, p and q. And let us say you know that one of these parameters is always smaller than the other one or is bounded by the other one, right? So, let us consider these 2 possibilities here. I want to claim that one of these statements is true.

So, if a problem is FPT with respect to one of these parameters, then it implies that the problem is FPT with respect to the other parameters as well, but which one implies which, so take a moment here, pause and make sure that you have come to a conclusion about which option you want to bet on. So, the correct answer is that if a problem is FPT with respect to p, then it is FPT with respect to q.

And the reason for this is just consider the running time of your FPT algorithm right at some F of p poly n. And that is just automatically at most F of q poly n because of what we are given here. Intuitively also, it is as if somebody has given you just more wiggle room, somebody has given you a more generous parameter to work with. So, if you were able to come up with an algorithm when you were constrained and in tighten circumstances, then of course, you are going to do well when you have even more resources at your disposal.

On the other hand, if you had a FPT algorithm and q to begin with and then somebody asked you for a FPT algorithm and p instead, then it is not clear that you can automatically just rely on the non FPT algorithm in the larger parameter. It really may not have any implications for the smaller parameter p. In fact, the whole complexity status of the problem may even change as you go from q to p.

So, in general, going from q to p is sort of more challenging, uphill task and certainly not automatic. But if you are doing well with respect to p, then you are automatically doing well, with respect to q. Now, if you think about the 2 parameters that I am proposing here. So, in the one hand, we have k – mu of G and on the other hand, we have k – LP OPT from last time, you can work out that k – mu of G is actually the larger of the 2 parameters.

And the reason for that is mu of G is at most LP OPT. LP OPT is in general, at least the size of any maximum matching. So, in some sense, you are taking away less from k when you take away the size of maximum matching compared to when you take away the LP OPT. So, this just happens to be the larger of the 2 parameters. So, we already know that vertex cover parameterised by k – mu of G is FPT with the same running time, so 4 to the k – mu of G.

And why am I telling you all this? Why are we starting off by weakening a stronger result? Well, it just turns out that this formulation of the problem is more useful or more convenient to work with when we get to the reductions that we want to do. So, this is the problem that I want to keep at the back of our minds as we go through this lecture. So, I hope that is clear. And now we can transition to the first problem that we want to talk about, which happens to be the problem of finding an odd cycle transversal.

So, what is an odd cycle transversal or OCT for short, in a graph G? As always, we will be working with simple and undirected graphs for this discussion. And an OCT for such a graph is simply a subset of vertices whose removal makes the graph bipartite equivalently. You can think of OCT as a subset of vertices that interested in killing cycles, much like FPS a problem that we met recently. But unlike FPS and OCT does not want to kill all cycles.

It is only interested in targeting the cycles that have an odd number of vertices in them and hence the name or cycle transversal. So, notice that these 2 definitions that I have mentioned, deleting to get a bipartite graph and deleting, so that your intercept, all cycles of odd length are in fact equivalent. Because you might remember that a graph is bipartite if and only if it does not have any cycles of odd length.

As usual, let us just establish some familiarity with the problem simply by going through a small example. So, I am going to show you a graph and I am going to ask you to figure out what is the smallest OCT for this graph. So, take a moment here, pause the video if you need to and try to figure it out what would be a small OCT for this graph. Of course, small is relative just come up with the smallest that you can find.

So, hopefully, you had a chance to think about that. And a first observation that you might make is the following. If you decide not to remove the vertex w, which seems to be at the centre of this spider web thingy here, if you decide to not remove it, if you decide to not

include it in your OCT, then you are essentially left with these 4 triangles to worry about, which are disjoint, except for the vertex w.

And since you have, let us say, have committed to not picking w, each of these triangles is going to demand a special place of its own in the OCT. So, any OCT that chooses to ignore w must already pick at least 4 vertices. Now that sounds kind of expensive. So, maybe it is a good idea to start by picking the vertex w. Now, once you have picked the vertex w, notice that this is the graph that you are left with.

Maybe I should show you the graph that we started with as well. So, w is gone. And you are left with these sort of 4 matching edges and then you are left with a vertex z, which sort of connects to them in this particular way. So, notice that is still left with one cycle of odd length. So, one way of getting rid of that cycle is to, for example, get rid of the vertex z, although you could pick any one of the other witnesses on the cycle as well.

And I believe that should work. Although to be honest, I have not checked. But in any case, w from z is a valid OCT for this graph of size 2. And notice that you can also convince yourself that you do need at least 2 vertices. In that sense, this is optimal. It is not very hard to see that you have, for example, this odd cycle. And you could pick one of wab or wxy as another odd cycle, which is disjoint from this one.

So, just like we have been saying for problems like vertex cover and FPS. If you have multiple disjoint obstructions, then that is a lower bound for how many vertices you need in your solution. So, we have 2 disjoint cycles of odd length. So, any OCT here would have to have size at least 2. And so, we were fortunate that we found one that has size exactly 2 and therefore that is going to be optimal.

As always, the computational question associated with this definition is the question of finding an OCT that has the smallest possible size. And we will of course be working with the decision version of the problem where we are looking for an OCT of size at most k for a given budget k. So, that is the problem that we will be working with. So, if you like spend some time thinking about, you know, your own approaches to coming up with an FPT algorithm for this problem.

For instance, you might think about just generalising the algorithm that you learned for FPS or just trying to branch on short cycles of odd length. All of these are good places to start. And if you would like to take some time now, just to think about the problem, then please feel free to do that. As I promised earlier in this particular lecture, we are not going to be coming up with algorithms from scratch, but we will be trying to leverage the algorithms that we have already developed in the previous lecture.

So, once you are ready and once you come back, what we are going to be doing is we will be transforming instances of odd cycle transversal into instances of LP above matching. So, that is going to be an interesting experience. So, let us get started if you are ready. So, here is what we are going to do, let us say we are working with an instance of OCT. And this is just one concrete example, to make it easier to see what is going to happen.

So, suppose this is the instance of OCT that you are working with. We are going to convert this to an instance of vertex cover as follows. So, we are going to make 2 copies of the graph. So, you can see the 2 copies drawn out in black here. And on top of that, we are going to add a matching. And the way the matching is designed is that it is going to add an edge between 2 copies of any original vertex.

So, for instance, if you focus on the vertex a here, it is manifested twice and we label them a 1 and a 2 just to keep track of where it came from. And now, you have an edge that connects a 1 to a 2. And similarly for b and c and so on, all the remaining vertices. So, in general, if you started with a graph on n vertices, you are going to obtain a graph on 2n vertices and do m + n edges. So, all the original edges are going to be duplicated.

So, you have 2 copies of them. So that is the first 2m edges. But on top of those original edges, you are also going to add this matching. So, that is going to be an additional n edges. Why is this construction? Interesting? What are we going to get out of this? Well, it turns out, this is where all the magic happens. It turns out that if G is a graph on n vertices and if this graph that we just build out you call it as G tilde as we have done here, then the following is true.

G has an OCT of size at most k if and only if G tilde has a vertex cover of size n + k. This is really cool. I must say, if we can show this, then notice that we already have a 4 to the k

algorithm for OCT. And how is that? Well, notice that n is the size of a maximum matching in this graph. So, if you were to parameterize, this vertex cover instance by the above guarantee parameterization that we started with, which is k minus the size of a maximum matching.

Then notice that k minus the size of a maximum matching really amounts to the k that we have here for OCT, right. So, you have an algorithm whose running time is 4 to the k minus the size of a maximum matching, where k is the size of the vertex cover that you are looking for. And the size of a maximum matching is n. So, that is going to be n + k – n, which is case of the running time of your vertex cover algorithm.

The algorithm that we described in the previous lecture is going to be 4 to the k, where this k is now the k from the OCT instance. So, hopefully, you can see that once you show this equivalence, you have your OCT algorithm. And that is why we should be all motivated at this point to actually establish this equivalence. Once we do that, we are in fact, done. So, let us try to show this right. So, this is what we want to show.

G has an OCT of size at most k if and only if G tilde has a vertex cover of size at most k. By the way, you might just want to make a quick note of how the reduction works or just committed a little bit to memory because this is something that we will be working with you have G; you have G tilde. G tilde has 2 copies of G and a matching between the duplicate vertices. A matching that matches every vertex to its copy in G tilde.

So, with that out of the way, let us get started with proving the first of these 2 statements, which is to say that if G has an OCT of size at most k, let us show that G tilde has a vertex cover of size n + k. So, let us start with G. Here is G. And let us say, this is some OCT of size at most k. This is a very abstract version of G, I am no longer working with the specific graph that we started with.

So, at this point, the discussion is just going to be about some generic graph G, some generic instance of OCT. So, the assumption, remember was suppose G has OCT of size at most k. So, let us set the OCT aside. Let us keep it on top, let us call it x. So, this is the OCT of size at most k. And let us say it leaves behind a bipartite graph and it is deleted. So, this is G – x. And let us say G – x is bipartite with by partitions A and B.

Let us try and see what this picture looks like in G tilde. So, how do these subsets manifest in G tilde? Remember, we have 2 copies of everything. So, that is what this is going to look like. So, in G tilde, let us say these are the 2 copies of x. And we also have 2 copies of A and 2 copies of B. And remember that A B induced bipartite graph. So, even in these copies, you are going to have a bipartite graph here and a bipartite graph here, which is to say there are no edges inside A 1, B 1, A 2 or B 2.

So, by the way, I am using capital a 1 to denote the subset of vertices, which constitute the first copies of the vertices in A. A 2 to denote the second copies and so on and so forth. x 1, x 2, B 1 and B 2 that is the notation and hopefully that is clear. Now, what do we want to show here? What we want to show here is that okay, we had an OCT of size at most k here. So, we want to say that there is a vertex cover of size at most n + k here.

By just staring at this picture, can you spot this vertex cover of size n – k, it is a combination of some of these boxes. And there are only so many boxes. So, I suppose, there are 6 boxes. So, you have 64 options, but not really, I mean, at least you do not have to hopefully consider all of them. I think it is reasonably intuitive as to how you might want to construct the vertex cover.

By just observing where all the edges are and realising that those are the edges that you have to make sure to hit. So, just think about that for a minute. If you need to pause the video here, please do and come back when you are ready. So, notice that this is a subset of size 2 k, the 2 copies of x that seems like a small price to pay. So, let us just set those aside. Now, here we have these edges going across.

So, notice that if we pick either A 1 union B 2 or A 2 union B 1, then we would cover everything. It would not work to pick A 1 union B 1 for instance, because then these edges would not be uncovered. It also would not work to pick A 2 union B 2, because those edges would be uncovered. Similarly, it will not work to pick A 1 union A 2 because these matching edges would be left behind.

And it would not work to pick B 1 union B 2 for the same reason. Now, of course, you can pick any superset of the set that I described. But it turns out that because they said that I

described as enough that is what we will stick to doing. So, for example, here, I have the vertices from the 2 copies of x and A 2 union B 1. Now, what is the size of the subset that we have picked.

So, as we have already said, the 2 copies of x will cost us k + k that is 2k. And we are picking one copy of A from here and one copy of B from here. So, notice that that is used essentially the same number of vertices as we have in A union B. But A union B is essentially n – k vertices. So, the total number of vertices here k + k + n – k, which is essentially one of those case cancelled.

So, the total size of this highlighted subset, which is a vertex cover, is in fact n + k. But remember that is exactly what we were looking for. We wanted to prove that G tilde has a vertex cover of size n + k. So, we are done as far as the forward implication is concerned. So that is essentially half the battle. Let us now turn our attention to the reverse directions. So, so far, we have built a vertex cover based on an OCT.

And now, we want to somehow go the other way. Now, just to remind you, G tilde is this graph here. And we want to say that if this guy has a vertex cover of size n + k, then the original graph G that we were coming from, should have OCT of size at most k. So, let us try and show that now. So, as before, let us start with a picture of G tilde with the promised solution made explicit.

So, remember where we are coming from, we know that G tilde has a vertex cover of size n + k. So, let us make that vertex cover explicit here. Give it a name, let us call it y. And what we know is that G tilde minus y is an independent set. Now, let me also bifurcate this independent set into 2 paths and we will see why we are doing this. But it is quite natural for me to segregate the vertices that essentially are the first copies of the corresponding vertices in G versus those vertices, which are the second copies of the corresponding vertices in G.

So, for instance, if somebody gave you the independent sets C 1, A 2 in G tilde, then C 1 would go to A tilde and A 2 would come to B tilde. And notice that you will never have 2 copies of the same vertex sitting in G tilde minus y, because every time we had 2 copies of the same vertex, we explicitly drew an edge joining them. And this being an independent set, it is not going to bring in 2 copies of the same vertex.

So, for every vertex in G, we see, at most one of its copies in G tilde minus y that is a useful thing to keep in mind. Now, from here, we want to somehow pull back and OCT of size at most k. And it might be more useful to think about it in terms of wanting a large bipartite graph based on a large independent set, rather than a small OCT based on a small vertex cover.

Of course, these are both exactly equivalent, but it is just more convenient for me to think about it the other way. So, back in G, let us just bring up the vertices on which a tilde and B tilde are based, right. So, let us just drop the subscripts and go back to the original vertices corresponding to these subsets here. And this is going to tell us something about why; we are almost done.

So, let us ask ourselves these 3 questions. What can we say about the size of G tilde minus y? What can we say about the size of A union B? And what can we say about the graph induced on A union B? This is the original graph G. So, what does A union B look like in G? Let us try to answer these 3 questions. And you will see that we have essentially proved the reverse direction.

If you want to do this yourself, this is a perfect time to pause the video. So, what is the size of G tilde minus y? We know that G tilde are all together has 2n vertices and the size of y is at most n – k. So, when we subtract n – k vertices from 2n, we are left with n – k vertices. Did I just say we subtract n – k vertices? That is not quite right. We subtract n + k vertices at most n + k vertices from 2n. And we are left therefore with n – k.

So, the size of G tilde minus y is at least n – k. Also, notice that A union B, which is essentially the size of A tilde union B tilde is also at least n – k. Because we did not have 2 copies of the same vertex. As we noted earlier, the intersection of A and B is empty. So, the size of A union B is essentially the size of A tilde union B tilde. So, we have n – k vertices here. So, hopefully that is, so far.

And now, what can you say about G induced on A union B? It should be clear that that is going to be bipartite. So, notice that A and B are independent sets. Because if A, if there was an edge between 2 vertices of A, then that edge would have manifested in G tilde minus y as

well. But we have promised that that is an independent set. So that cannot happen. So, A is an independent set in G and B is an independent set in G and A union B is a set of size at least n – k.

Therefore, anything that is not an A union B, if you just look at that subset, so the vertex set of G – A union B. By definition that is going to be OCT. And by this calculation, that is going to be an OCT of size at most k. Therefore, we are also done with the reverse direction. And at this point, we have completed the argument for essentially why OCT admits an algorithm was running time is 4 to the k.

And we did this just by transforming an instance of OCT to an instance of vertex cover parameterised above the size of a maximum matching. So, hopefully, you have gotten the feel of the strategy that we are employing here. We are not coming up with a direct algorithm for the problem. But we are instead leveraging the algorithm that we already have for vertex cover parameterised by k – mu.

At this point, we are going to switch gears and talk about a different problem that we will solve using a very similar strategy and that we will reduce it to an instance of vertex cover above matching. So, if you want to take a break or just go back and make sure that whatever we have discussed about OCT is clear. This is a good intermission point. You can come back when you are ready.

So, the second problem I want to talk about is almost 2 SAT. But before we can talk about almost 2 SAT, we need to talk about 2 SAT first. So, hopefully, you are already familiar with the problem of SAT, it is perhaps the first NP complete problem that you encountered. And 2 SAT is a very, very special case of SAT, which actually happens to be polynomial time solvable.

So, let us recall what are 2 SAT formulas in CNF form. So, it is a Boolean formula, which has the following format. It is essentially the AND of M clauses where every clause is the OR have at most 2 literals. And a literal is simply a Boolean variable or its negation. So that is a 2 SAT formula. And so, in particular, if you gave me a truth assignment, truth values assigned to each Boolean variable that was involved in this formula.

Then we would say that the formula is satisfied by the truth assignment, if and only if in every clause, there is at least 1 literal that evaluates to true. So, either a true literal that was set to 1 or a false literal that was set to 0. Any one of these combinations would work. But every clause has to have at least 1 such literal that is true of any CNF SAT formula. The special thing about 2 SAT is that every clause only has 2 literals in it.

And it turns out that you can figure out if such a formula admits a truth assignment or not, which satisfies it in polynomial time. So, that is great. It is always nice to have a SAT variant that you can actually tackle in polynomial time. But now, suppose you were working with an actual instance of 2 SAT. And let us say, you run your algorithm and let us further assume that your algorithm comes back to tell you that this formula was not satisfiable.

So, of course, this is an objective truth, there is nothing you can do about it. And you could argue that there is no need to get emotional about this. You do not have to be happy or sad, just because your formula was satisfiable or not. And that is true if you were doing this SAT business for homework. But in the real world, typically, the reason people are solving instances of SAT in the first place, is because it models some real world problem. And the solutions correspond to actual solutions.

So, if things do not work out, you would typically be disappointed. Now, the question is, what is the next best thing you can do? That is always the question, right? So you could not satisfy the whole formula. But maybe you could you come close to satisfying, let us say, most constraints. If so, then, you know, maybe that is a good situation to be in once again. And that motivates the definition of the almost 2 SAT problem.

Now, there are 2 natural ways in which you can talk about almost satisfying a 2 SAT formula, you could say, let us try and satisfy, say, 90% of the constraints. Or you could also say that suppose I just eliminate a small number of variables from the picture, then can you satisfy the rest of the formula? So, both of these are valid ways of trying to talk about being close to being satisfiable. So, we are going to look at the variable version in this lecture.

So, the question is, given a 2 SAT formula and hopefully small budget k, can you remove at most k variables from the formula so that the rest of it is satisfiable? And what does it mean to delete at most k variables just means that you get the clauses that involve these variables

for free. So, you just eliminate all clauses that involve these variables in either positive or negative form.

So, just to get some practice with the definition of almost 2 SAT, let us just think about a specific example. So, here is a 2 SAT formula. And I would like you to take a pause here to think about whether the formula is satisfiable or not. And just in case, it turns out to be not satisfiable. Think about whether it is almost satisfiable with a budget of 1. So, feel free to take a pause here and come back once you have had a chance to think about this.

So, the answer to the first question, as you might have guessed, is no, because we probably do want the second question to be a legitimate one. But you can actually argue that this formula is not satisfiable. One way to do it is of course, to try all possible assignments to all the variables, but there is actually an easier way to do this. And that is why observing what happens if you try setting x to true and what happens if you try setting x to false. In both cases, you get essentially implication chains, which lead to contradictions.

Therefore, there is no satisfying assignment for the formula to begin with, because any such assignment would have to set x, one way or the other. But no matter what it did, it would end up in a fix and it is not going to be workable. On the other hand, if you remove this offending variable x from the picture, then a bunch of clauses go away and you are just left with these 3 right here in the middle.

And it turns out that those are satisfiable. In fact, I think in more ways than one, but it is certainly going to be a satisfiable formula. So, hopefully, the notion of an almost 2 SAT instance is clear by now. And we are going to solve this pretty much the same way that we solved OCT. We are going to build a graph based on this formula. And we are going to argue that this graph is a yes instance of vertex cover above matching if and only if the formula that we started with had a solution of size at most k.

So, I am going to describe the construction first in abstract terms, I would encourage you to actually try and work through the construction for a specific example. But then I will also work out a specific example with you. And then we will argue the equivalence. So, let us start with the actual construction. Recall that what we are working with is a 2 SAT formula phi. And that is what we need to convert into some sort of a graph.

So, that the almost 2 SATness of phi can be reflected in the size of the vertex cover of this graph that we construct. So, here is how we are going to build the graph. For every variable in the formula 5, we are going to introduce 2 vertices, one representing the positive literal and one representing the negative literal. So, just like we added a matching between the 2 copies that we created in the OCT construction.

We are also going to add edges here between the x and x bar vertices that represent a single variable x. So, at this point, your graph is just a matching. We have not added anything else just yet. So, this graph is capturing information about the variables. And we need to now also encode in this graph information about the clauses of the formula 5. So, remember that 5 was a 2 SAT formula and in 2 SAT formula, every clause has at most 2 literals.

So, we have clauses that have one literal and we have clauses that have 2 literals. So, let us make a distinction between them. So, if you have a clause that has just one literal in it, we are going to call these unary clauses. Then, if you have a clause with the literal x, then what we are going to do is add a degree 1 neighbour to the vertex that is representing the literal x and we have a unary clause with the literal x bar.

Then we will add a degree 1 neighbour to the vertex that is representing the literal x bar. Now, if you are thinking ahead a little bit already and you are wondering why we are doing this, then in some sense, the very high level intuition is that we somehow want to say that if you have a clause, which is just 1 literal, then that literal has to be set in the way that is dictated by that clause in any satisfying assignment.

And our vertex cover is somehow going to correspond to a satisfying assignment. And so, we are just whispering to the vertex cover that we are anticipating that look, just go ahead and pick this vertex x or pick this vertex x bar. So, these are going to be degree 1 vertices, which essentially force the choice of the neighbour. Remember that we have talked about degree 1 reduction rules before and we know that there is always going to be an optimal vertex cover that picks the neighbour of a degree 1 vertex.

So, that is kind of what we are trying to do here. I think this will become more explicit as we go along. So, if what I just said did not make sense, then feel free to just ignore it. It is

nothing to worry about. Now, for the clauses that have 2 literals, we are going to go ahead and add edges between the vertices that represent those literals and this is again trying to just tell the vertex cover something right.

We are trying to say, look, please go ahead and pick at least one of these vertices to ensure that this clause is satisfied. So, that is the entire construction. This is the graph that we want to build. And again, let me just bring back the example of the 2 SAT formula that we had a few moments ago. And what I would encourage you to do is to pause the video here and actually construct this graph for this small formula here, just to get a feel for what is going on.

And you might even want to start thinking about the main claim that we want to make here. So, just like we had with OCT, what we want to show here is that if you have a valid solution for 5, k, right, if you have a subset of at most k variables that which, when set aside, make the remaining formula satisfiable, then you also have a vertex cover of size n + k in the graph that we just described. So, that is what we want to show.

And once again, once you show this, you are done, you have the equivalence, you can then run the algorithm that we had from last time on this graph that we have just constructed. And because there is a matching of size and remember that is what we added in the very first step. Our above guarantee parameterization will ensure that the corresponding FPT algorithm has a running time of 4 to the k, where k is the standard parameter for the almost 2 SAT problems. So that is what we want to show.

So, you might want to take a moment here to actually start thinking even about how you will show this equivalence. So, pause the video and once again, come back once you are ready. So hopefully, you had a chance to absorb the construction. And hopefully, you have developed your own intuition for what the construction is trying to achieve. As I promised, we will go through an example together.

So, here is here is a different example, just for variety. And I think this is the same example that you will find also in the textbook. So, in case, you are following along with them, just wanted to alert you to that. So, here is the formula on 4 variables here. So, we have added

these 8 vertices with a matching. So, the matching is in the background and it is in black, we have 1 unary clause, so that is this degree 1 neighbour for the corresponding literal.

And the remaining edges in red correspond to the clauses that we have here. And if you like, you can take a moment here to verify that hopefully, there are no mistakes in this picture. So, once you have done that, let us I think try to actually get back to the equivalence that we wanted to show. So, we wanted to say that if you have a 2 SAT formula and you build this specific graph based on it, then almost 2 SAT solution corresponds to a vertex cover of a certain size.

So, if you have a solution of size at most k, then this graph has a vertex cover of size at most n + k. So, as usual, let us start with the forward implication first. So, let us begin by saying, suppose you do have a subset of variables whose removal makes the formula satisfiable. And suppose that the fact that this formula is satisfiable, as witnessed by tau that is the satisfying assignment for phi – x.

And of course, we are removing at most k variables that is a part of the deal that is a part of the assumption that we have. Now, what is the obligation? We want to show that G phi has a vertex cover of size n + K. And again, take a moment here to think about what this vertex cover should be. It is reasonably intuitive, just working backwards from the anticipated size. And knowing that for every matching edge, we do have to pick at least one of the endpoints.

So, it seems pretty natural to say that you know, whenever we have a vertex corresponding to a undeleted variable, let us try to pick that endpoint that tau sets to 1. And for everything that is chosen by x, maybe we can just write a bit both of the endpoints. So, that is the solution that we want to build. In fact, let us make that explicit here. So, if you have a variable that is picked in the solution, then we will include both of the vertices that are based half of that variable.

And for every variable that is not in the solution, we are going to pick the endpoint corresponding to the satisfying assignment down. Now, it should be clear that this set has size at most n + k. Essentially from each matching edge, we are definitely picking 1 vertex but there are k of these variables that were included in x. And for those, we are picking both endpoints of the matching edges.

So, we are picking k extra vertices on top of n. And, again, this is something that you should take a moment to verify if you are not completely convinced yet. And now, you also want to think about why does this form of valid vertex cover. And the reason for that is essentially going to be based on the fact that tau is a satisfying assignment for 5 – x. So, in particular, if you have an edge that is left over, after you delete this proposed subset here, notice that it cannot be a matching edge.

And the reason for that is, we definitely picked at least 1 endpoint from every matching edge. So, it is going to be either one of these edges that come out of a degree 1 neighbour or it is going to be one of the right edges. But you can see that if any one of these edges are left over, then there is a clause that was not satisfied in phi – x, right. So, again, I am going to leave this as an exercise, although this should be quite evident from the construction at this point.

So, hopefully, you can see that we have a vertex cover of size n + k, if somebody gives us a subset of k variables whose deletion makes the formula satisfiable. Now, let us talk about the reverse direction, which is that if you had a vertex cover of size n + k, then from that vertex cover, you can recover a subset of variables whose deletion makes phi satisfiable. And, again, here, you might have already developed the intuition that you need to prove this implication.

The idea is to observe that any vertex cover of size n + k is going to have the following format. It is of course going to be compelled to pick at least 1 vertex from every matching edge. But because of this budget being n + k, they are probably going to be about k edges from the matching, in fact, at most k edges from the matching, where you end up picking both of the endpoints.

And you might suspect that those matching edges where you are picking both endpoints correspond to somehow the variables that you should be including in your almost 2 SAT solution. Now, you might, of course, say, what about these degree one vertices that we added that do not correspond to any variable. But again, because of our discussion about the degree 1 reduction rule, we know that we can always assume that we are working with a vertex cover that you know, does not involve any of the degree 1 vertices.

So, that is what I wanted to point out here. So, your vertex cover is essentially involving only vertices that correspond to real literals. And that is because of the fact that you can always exchange degree 1 vertex that is in your vertex cover with its neighbour and you will still be left with a valid vertex cover of the same size. So, with that out of the way, let us describe the satisfying assignment based on the vertex cover that we have.

So, if both the literals belong to our vertex cover, then we include the corresponding variable in our solution. And otherwise, we are going to set the variables based on how the vertex cover has chosen from the corresponding matching edge. So, this in fact, right here is a witness for why phi – x admits a satisfying assignment. So, the reason, this is a satisfying assignment is that if it was not in, if there was a clause that was left over.

Then the leftover clause which did not get satisfied, would correspond to an edge in this graph that was not covered by our vertex cover. And that would be a contradiction. So, that is essentially the basis of the equivalence of these 2 instances. So, we have this main lemma here. And essentially, based on the algorithm that we already have seen, we get for free pretty much modular this reduction 4 to the k algorithm for the almost 2 SAT problem.

So, hopefully, these arguments made sense **(Video Ends: 44:29)**. Do reach out and let us know if you have any questions. You could either post your questions in the comment section of this YouTube video or you can reach out to us on the Google Groups mailing list that has been set up for this course if you are enrolled through swayam. So, thanks so much for watching and we will see you back next week.