Parameterized Algorithms
Prof. Neeldhara Misra, Saket Saurabh
Department of Computer Science and Engineering
Indian Institute of Technology-Gandhinagar

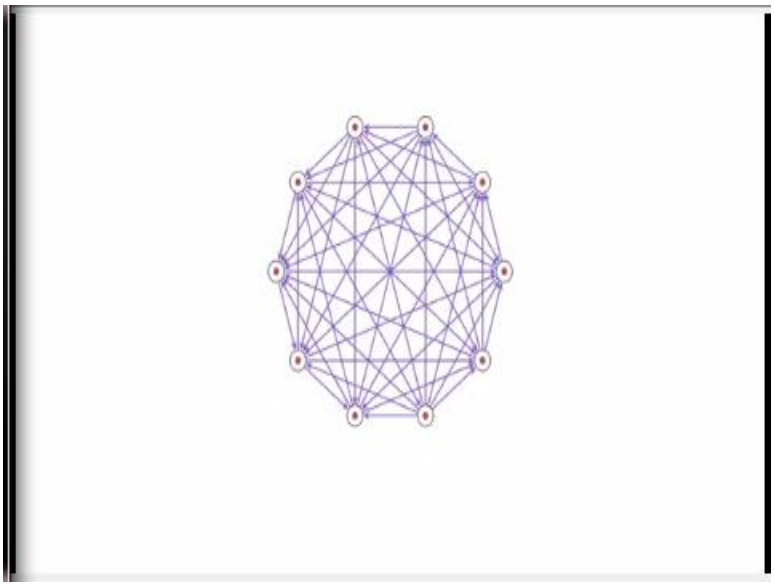
Lecture-18
Chromatic Coding for Feedback Arc Set on Tournaments

Welcome to the 3rd module in the 4th week of parameterized algorithms. This week we have been looking at randomized methods for coming up with FPT algorithms. And in just the previous module we explored a technique called colour coding where the high level idea was to partition the vertex subset of the input graph randomly in the hopes that the solution that we are looking for pops out and becomes easily identifiable.

Now this technique works great when the solution that you are looking for involves a small vertex subset. We are going to try and transfer the same philosophy for situations where the solution that we are looking for is a subset of edges rather than vertices. It turns out that in this situation not only does the same idea apply, it also turns out that the number of colours that are required to guarantee a decent probability is less than before and this actually is going to give us even better running times.

In particular we are going to see a sub exponential algorithm at the end of this discussion. So, the running problem that we are going to use is the problem of finding a feedback arc set in a tournament graph. This might remind you about the feedback vertex set problem that you saw when we were discussing iterative compression. It is a closely related problem, but yet it is different because we are looking at subsets of edges instead of subsets of vertices.

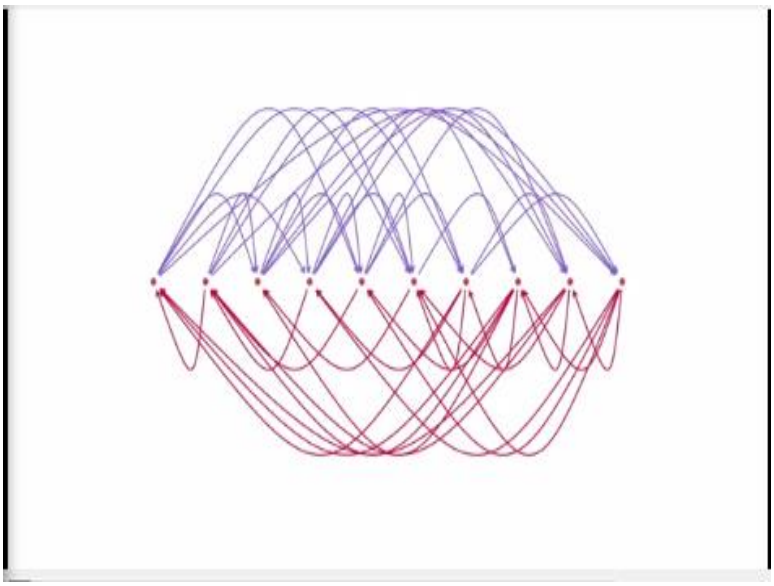
(Refer Slide Time: 01:46)



So, let us begin by recalling what a tournament is. A tournament is simply a complete directed graph. In particular if you have a graph on n vertices then you are going to have n choose 2 edges and each of these edges can be thought of as an ordered pair between vertices. So, between a pair of vertices u and v, you either have the edge from u to v or you have the edge from v to u. Now in general tournaments can have cycles, these are directed cycles and it is the usual definition in play this is going to be a sequence of vertices.

Such that every consecutive pair of vertices u v has an edge from u to v and it loops back so the last vertex of the sequence has an edge to the first vertex on the sequence, so that would be a cycle. And what we are typically interested in is the problem of getting rid of these cycles. Now in general if you have a tournament that does have at least 1 cycle.

(Refer Slide Time: 02:48)



Then no matter how you order it is vertices and lays them out on a sequence, you will have edges that go in both directions. So, the purple edges are what we will call the forward edges and the red edges are what we call the back edges. Now if you have at least one cycle, then no matter how you sequence the vertices in any permutation you are going to have at least one back edge. However, if you do not have any cycles then there is a way of ordering the vertices, so that there are no back edges at all.

And this is a useful view to keep in mind; you might remember the term topological sort from our discussion on feedback vertex set on tournaments. Where we said that a topological sort is exactly such a sequence where you do not have any back edges. And we said that in general if you are working with a directed acyclic graph, then you can always find a topological sort. So, it is just good to keep that at the back of your mind as we go along.

(Refer Slide Time: 03:51)

Now let us revisit the feedback vertex set problem. So, this is a problem that we have seen many times by now. And in particular applied to tournaments we have even learned how to solve it by using a reduction to hitting set and using iterative compression. So, let us just recap this very quickly. So, in general a feedback vertex set is a subset of vertices whose removal destroys all cycles. And applied to tournaments which simply means that what you are given as input is a tournament which again is a complete and directed graph.

Now let us define the analogous problem but in the context of edges rather than vertices. And notice that when you are dealing with edges there are 2 natural ways in which you might want to get rid of them.

(Refer Slide Time: 04:40)

The first way and the most natural one is to delete edges. So, the natural analog of the previous problem would be the question of whether you can remove at most k edges or we would call them arcs because that just reminds us that these edges are oriented or directed. So, the question here is can we remove at most k arcs to make the tournament acyclic? But there is another way that you can think of the process of removing an edge.

And especially in the context of directed edges, it makes sense to ask if we can get rid of edges by reversals. So, in particular can we reverse the orientation of at most k arcs? So, that the tournament becomes acyclic. You may have guessed this already, but this notation here T circle S is being used to denote the tournament, that is essentially the same as T with the only difference being that the orientations of all the arcs in S have been reversed.

Now these 2 problems feedback arc set by reversals and feedback arc set by deletions may seem like 2 problems that are both very similar and very different at the same time.

(Refer Slide Time: 05:53)

They are similar because they seem to have similar optimization objectives in both problems; we want to change the graph, so that the graph becomes acyclic. And we also want to minimize the amount of change that is involved in the process. However in the first version we are allowed to delete edges and in the second version we are allowed to reverse them and these seem like 2 inherently different operations.

Now from an algorithmic perspective which variant do you think is more manageable? Which do you think is easier to work with if you had to make a choice? This may seem like a slightly vague question at this point but let me still point out for instance that deletions can be a little bit nasty. For the reason that when you delete edges you actually damage the structure of the graph unlike when you reverse arcs.

So, if you reverse an arc in a tournament the resulting graph is still a tournament and you can take advantage of the tournament structure in any way that you want. On the other hand when you delete arcs, this is no longer true and it may be a little bit difficult to apply even an idea like branching on triangles for instance which worked out pretty well when we were dealing with the feedback vertex set version.

So, if you had the edges of a triangle and you deleted one of them, then well, the regressively generated instance would no longer be a tournament. And this fact that you might as well branch

on triangles will no longer be true. So, because you do not have the structural invariant you might find it just a little bit harder to work with this version of the problem. So, one hope is that you can work with reversals but still somehow solve the deletions problem.

And that would happen if we knew somehow that these problems were equivalent. Well, it turns out that surprisingly they are and I say surprisingly because at least intuitively it seems like deletions are more powerful operation, it seems like they can do more damage. I mean certainly whatever you can do with reversals, you can do with deletions. Because if you reverse the subset of arcs and the tournament becomes acyclic, then certainly deleting them would work just as well.

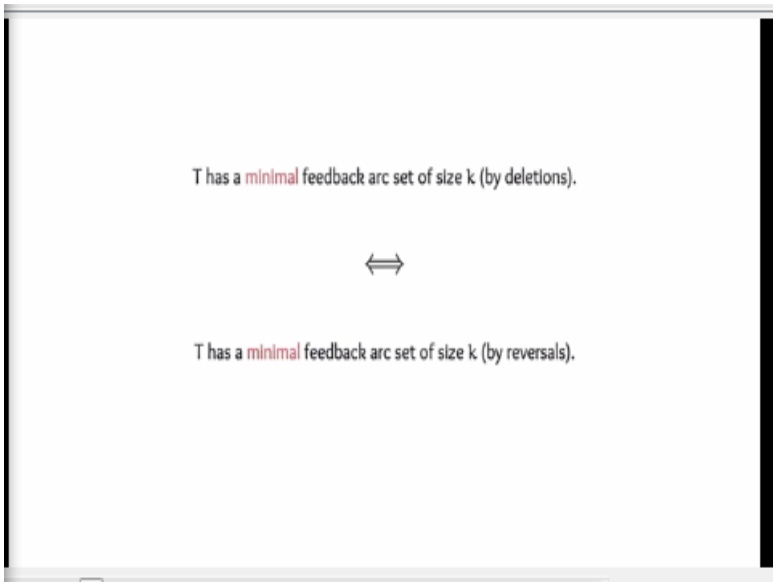
But it is not clear that if you delete a set of arcs then reversing them would actually have the same effect. In fact here is an extreme example, if you have a tournament from which you delete every single arc you delete everything that is there, then you left with an empty tournament which is trivially acyclic. But if you reverse all the arcs then you are essentially back to square 1 and if there was a cycle, it would still persist because it is just a cycle that is now going the other way.

So, certainly it seems like you can do more damage with deletions compared to reversals. In spite of this it turns out that we can make a sufficiently meaningful connection between the 2 variants of this problem. So, that we would be justified in focusing all of our attention on the reversals problem which I said is the more convenient variant to work with. And we would have still managed to solve the deletions question. So, let us take a look at how that works out?

(Refer Slide Time: 09:08)

Because of our discussion so far this statement is not really true, the problems are not equivalent on the face of it, but the following is true and is sufficient for our purposes.

(Refer Slide Time: 09:16)



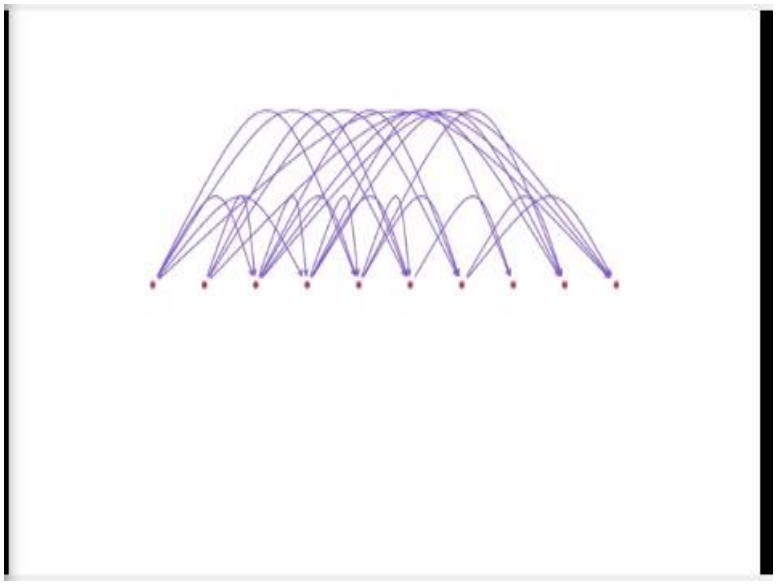
So, if you have a minimal solution for either variant then that can be turned into a minimal solution of the same size for the other variant. So, in particular if somebody gives you a minimal subset of arcs whose deletion makes the tournament acyclic, then it turns out that actually reversing those very arcs will also make the tournament acyclic. So, notice that our counter example here does not play out.

Because if I give you the subset of all edges in a tournament as a deletion set, that is not really going to be a minimal solution and you can verify that for yourself. So, let us see how this argument plays out?

(Refer Slide Time: 10:01)

So, let us begin with the easy case which is the reverse direction. So, suppose your tournament has a minimal feedback arc set by reversals of some size. So, let us imagine that you have reversed these arcs and you have obtained a topological sort. So, let us arrange the vertices according to this topological sort. And then reverse back the arcs, so that you get this sort of a visual. So, essentially all of these back edges are the edges of your solution.

(Refer Slide Time: 10:30)



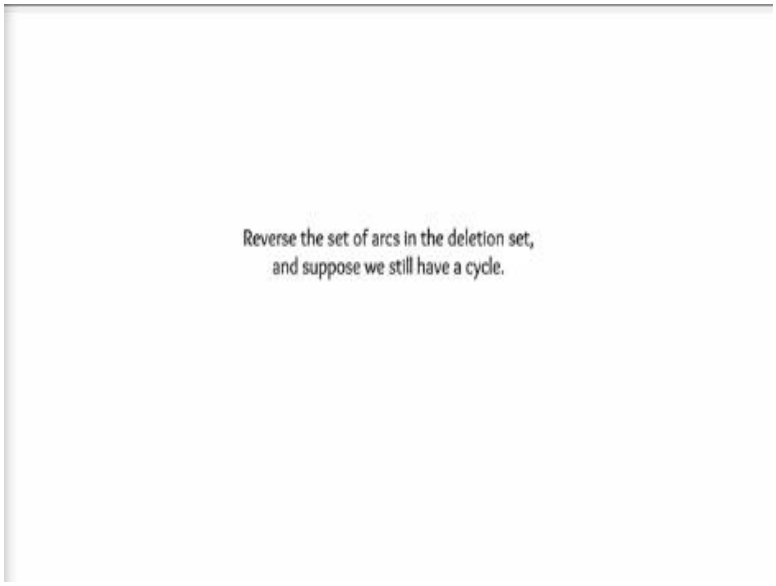
And notice that if you were to simply delete them then you are left with an acyclic tournament. So, certainly any subset of edges that can be reversed to make your tournament acyclic can also be deleted to make the tournament acyclic. Now there is a question of whether the solution is also a minimal deletion set starting with the assumption that it was a minimal reversal set. Well, think about it if this was not a minimal deletion set, then there would have been some edge that you can put back in it is original orientation without causing a cycle.

But if this was the case then you also would not have faced the need to reverse it, so that contradicts your assumption that you started with a minimal reversal set to begin with, so that is the reverse direction.

(Refer Slide Time: 11:15)

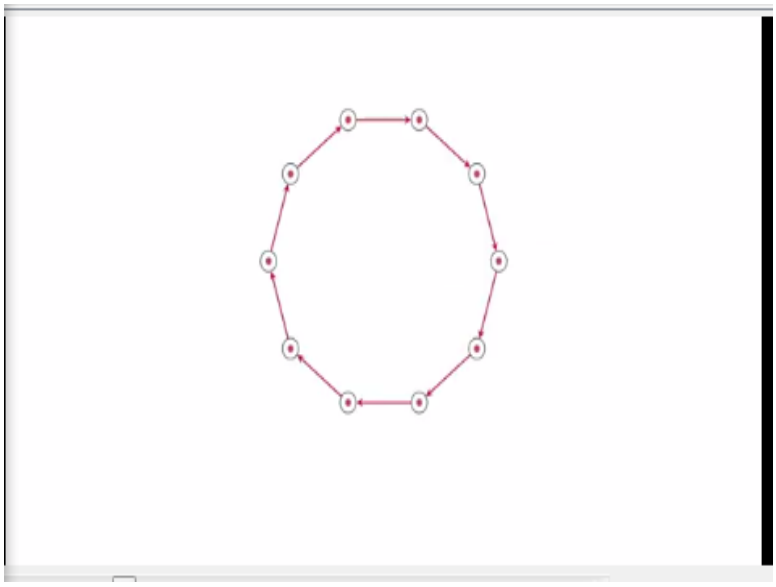
Now let us talk about the forward direction. So, suppose somebody gives you a feedback arc set of size k by deletions and remember that this is a minimal solution. So, we are going to use this fact quite crucially, so do keep that at the back of your mind.

(Refer Slide Time: 11:31)



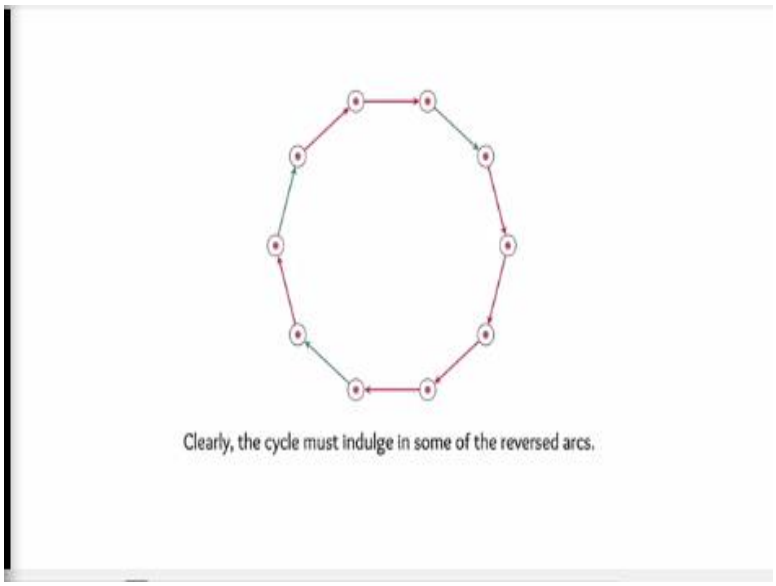
So, suppose we try reversing the set of arcs and if that actually turns your tournament into any cyclic one, then you are done, there is no more work to do and we can go home. But suppose this set of arcs does not work as a reversal set. So, in particular suppose we still have a cycle, then we want to derive some sort of a contradiction and then we will be done.

(Refer Slide Time: 12:00)



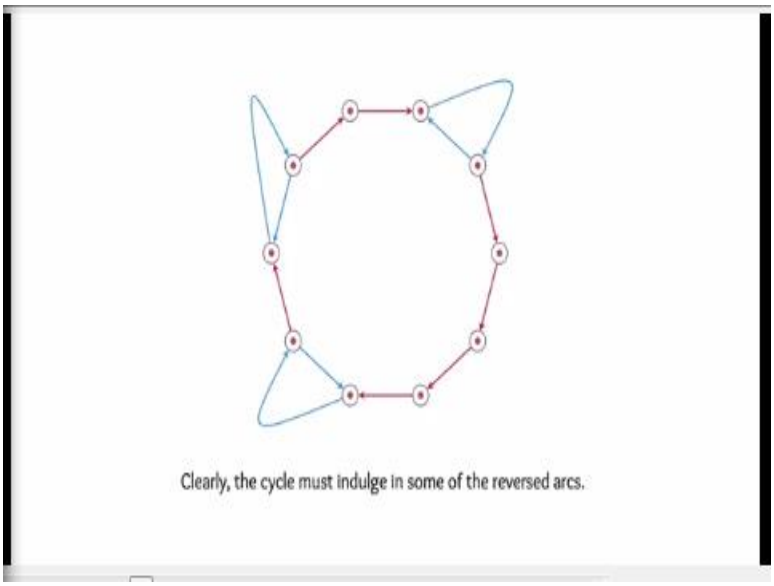
So, let us take a look at this cycle. So, the first observation is that this cycle must involve some of the reversed arcs. Because if it did not involve any of the arcs that we reversed, then this is a cycle that is also present in the tournament when all the reverse tasks are deleted but that will contradict our assumption that we were working with a valid deletion set in the first place.

(Refer Slide Time: 12:22)



So, let us mark these reversed arcs and denote them by these green edges out here. But now recall that these green edges belong to a minimal deletion set, which means that originally these arcs were facing this way, the other way. So, when the edges were green that is after the reversals. So, this was their original orientation and with respect to their original orientation each of these arcs did participate in a cycle. Because if these arcs did not participate in any cycles then they would not have had a license to belong to a minimal deletion set?

(Refer Slide Time: 13:05)



So, let us draw these cycles out. And notice that you can always choose these cycles to be such that they are not killed by any of the other edges in the deletion set. And once again you can always find such cycles because of the minimality of the solution. So, fix any of these base edges

on the red cycle if it was true that every cycle that they participated in was also being killed by the deletion of some other edge on that cycle.

Then again what is the use of this cycle in our solution? There is no need to delete it, because all the work that it is doing is also being done by somebody else. So, we might as well not delete it and we would still have a valid solution, so that would contradict the minimality of the deletion set. Therefore all this is to say that for every edge on the base cycle we can actually find a cycle where this is the only edge that is in the deletion set from that cycle.

Now what does that mean for us? What that means for us is that all of these blue paths that are jutting out of the base cycle, none of the arcs on these blue parts were reversed by us. Because none of these arcs belonged to the deletion set because of what we just argued. Now from here can you see a contradiction, just take a moment and pause and see if you can complete this argument yourself and come back when you are ready.

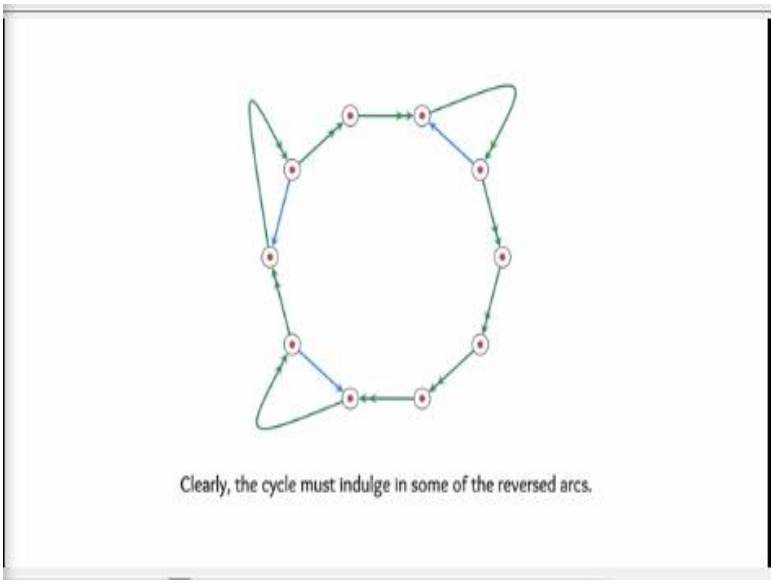
Alright, so hopefully you had a chance to think about this for a bit. Now what is the contradiction that we can hope to derive? Well, one way of getting a contradiction would be to identify a cycle that exists completely outside the set of deleted arcs. Because that would again contradict the assumption that the set of deleted arcs was a valid feedback arc set. So, is there such a cycle in this picture?

Well, notice that on the base cycle none of the red edges were deleted because all the edges that were marked for deletion. Well, we had marked them for reversal and we highlighted them, we unreversed them and we found these other cycles that witnessed their membership in a minimal solution. So, we have gone through that exercise for all the edges on the base cycle that were in fact marked for deletion.

So, the rest of them those are the red edges they were not marked for deletion and they were not reversed by us either, so these are all original edges. And similarly we know for all the paths that are jutting out of this base cycle, this is also true. That from our previous discussion we know that none of these edges were touched by the solution either. So, now let us try to go around the

base cycle but every time we encounter an issue because we have an edge facing the wrong way. Let us take the bypass via the cycle instead, so that cycle would look like this, the highlighted green cycle.

(Refer Slide Time: 16:00)



And notice that every edge on this green cycle actually does not belong to the proposed solution that you started with. But that contradicts the assumption that you were working with a solution in the first place. So, that pretty much concludes our argument but there is a small way in which this image here may be misleading, so let me take a minute to point that out. So, the 3 cycles that I have drawn on top of the 3 reversed arcs on the base cycle are conveniently disjoint here.

So, when you follow them along you get conveniently a cycle which you can use for contradiction. But in general there is no reason for these to be disjoint and I just want to point out that that is really not a problem. You can still follow them along pretty much the same way that we have described. It is just that what you will end up with is going to be a closed walk which is to say that you may have vertices repeating.

But it is fairly straightforward to extract a cycle from a walk and you will still get the same contradiction as before, so this is not really a problem. So, let us just step back and take stock a little bit.

(Refer Slide Time: 17:15)

So, in the previous lectures we have talked about the feedback vertex set problem on tournaments. And we saw that because of the observation that a tournament has a cycle if and only if it has a triangle you can reduce this to the 3 hitting set problem. And you can use algorithms for 3 hitting set to solve feedback vertex tournaments. And you can improve this running time using iterative compression.

Now on the other hand what can we say about the feedback arc set problem? Well, just like before we can do some branching, especially now that we know that we can work with the reversals version, it is easy to maintain the tournament structure as we go along. So, again using the observation that a tournament has a directed cycle if and only if it has a directed triangle we can once again branch on triangles for as long as we can find them.

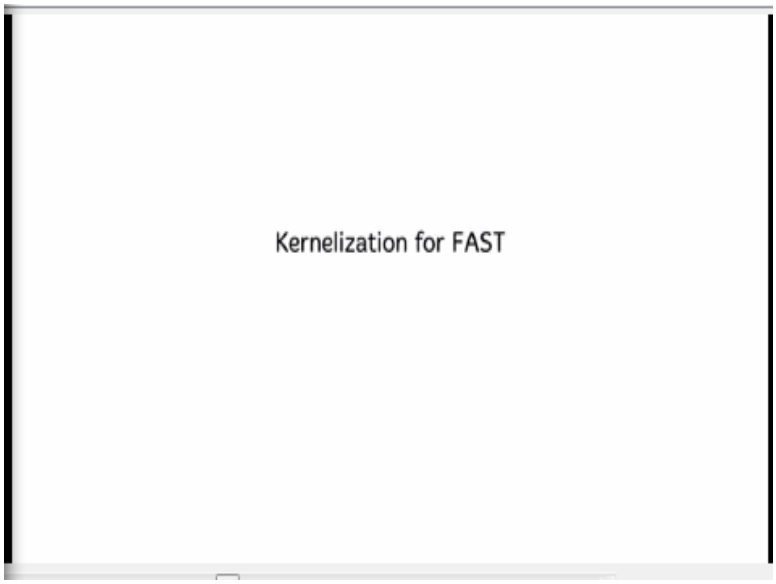
So, you find a triangle, you explore the 3 possibilities corresponding to the 3 arcs on the triangle, try reversing each one in turn, appropriately adjust the budget and move on. The only thing to be a little bit careful about is ensuring that you do not reverse an arc that you have already committed to reversing in some prior recursive step. This is something that you can ensure by just a little bit of extra bookkeeping, tracking the solution that you have built so far.

So, it is a good idea to work out the details for yourself just to be sure, but for now you could take my word for it there is a natural 3 to the k branching algorithm for the feedback arc set

problem on tournaments. However an analogous improvement to the feedback vertex set problem was open for quite some time. But when it did get improved, it got improved all the way to a sub-exponential running time and that is the randomized algorithm that we are going to see now.

Interestingly this randomized algorithm uses kernelization as an important ingredient apart from the usual drill of having a random experiment and a good event. So, we will get to that bit but as just a warm up or practically a bit of a prerequisite let us look at a kernelization algorithm for feedback arc set on tournaments.

(Refer Slide Time: 19:28)



So, this is again a fairly natural kernelization process, so it might remind you of some of the discussions that we had in the very first week, so let us kind of try to figure this out together.

(Refer Slide Time: 19:47)

So, first this is the analog of the high degree reduction rule for vertex cover, this idea is quite ubiquitous across problems. If there is one object that is doing a lot of work for you, there is a point where you cannot ignore it anymore. So, if there is an edge that participates in more than k triangles then it makes sense to mark it for reversals. So, reverse this edge in the tournament and reduce your budget by 1.

Now our next rule is kind of analogous to either the isolated vertex rule for vertex cover or the degree 1 rule for feedback vertex set. It is essentially asking how do we deal with vertices that do not participate in any triangles. Intuitively such vertices are irrelevant to our problem, so we should be able to delete them without changing the parameter and it turns out that this is indeed a safe reduction rule.

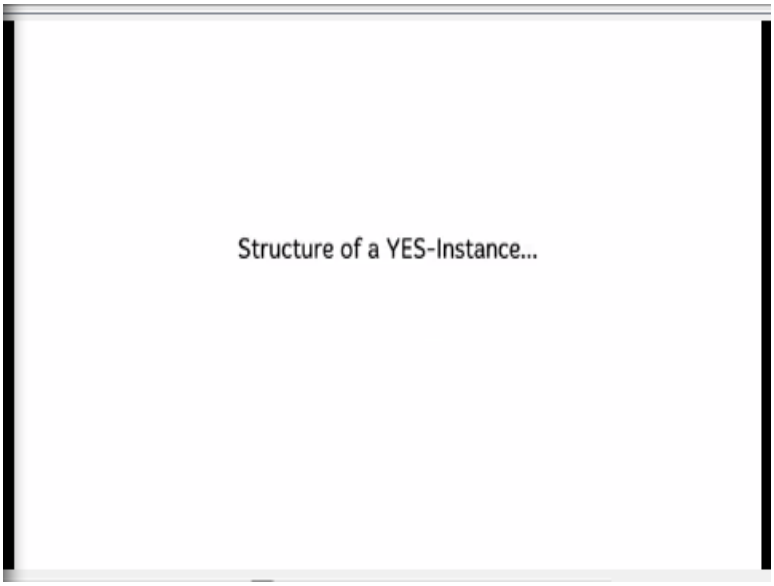
The forward direction is fairly straightforward because you are going into a sub tournament, so any solution will carry over quite easily. But what you would have to prove is that if you have a subset of arcs whose reversal makes the tournament T - V acyclic, where V is the vertex that you deleted. Then when you bring back V into this tournament with the arcs reversed, you do not suddenly get a cycle.

Well, the way to argue this would be to focus on the cycle that you might get for the sake of contradiction. And try and observe that well if there was such a cycle then it would not be the

case that we did not participate in any triangles in the original tournament. To complete this argument you might have to assume that the solution that you have is a minimal one and notice that you can always assume this without loss of generality.

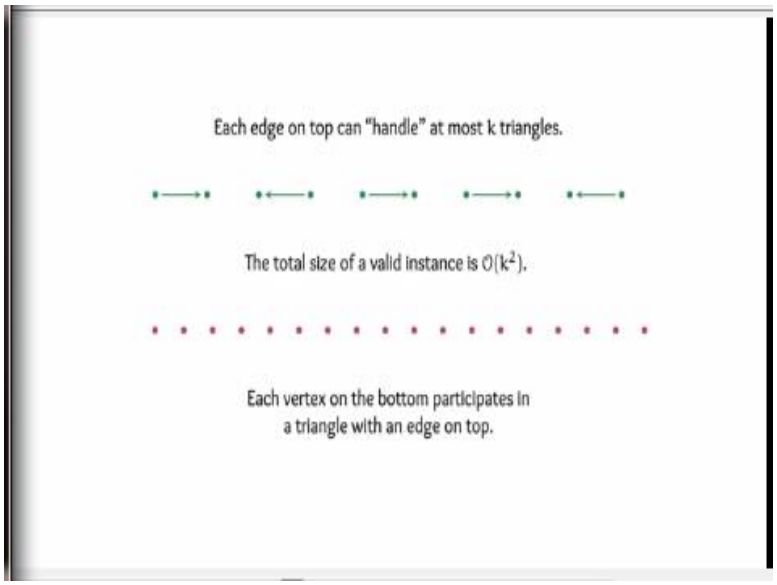
So, do work out the details once for yourself to be sure, at this point I am going to leave this as an exercise. And we will move on to actually talking about the size of the kernel, so in fact these 2 reduction rules are the only ones that you are going to need. Apart from another one which is more or less a formality which says that, well if the tournament is too big and the reduction rules do not apply, then you can say that it is a no instance.

(Refer Slide Time: 22:04)



To be able to say that let us try and understand the structure of a YES instance.

(Refer Slide Time: 22:11)



So, suppose you did have a subset of some k arcs or at most k arcs whose reversals indeed made the tournament acyclic? So, for the sake of analysis let us set these k arcs aside, of course they may not always be vertex disjoint as is shown in this picture. But this is just for clarity and we do know that the solution the arcs involved in the solution will span at most 2 k vertices. So, this is in some sense the maximum number of vertices that they can be involved then. Now what about the other vertices?

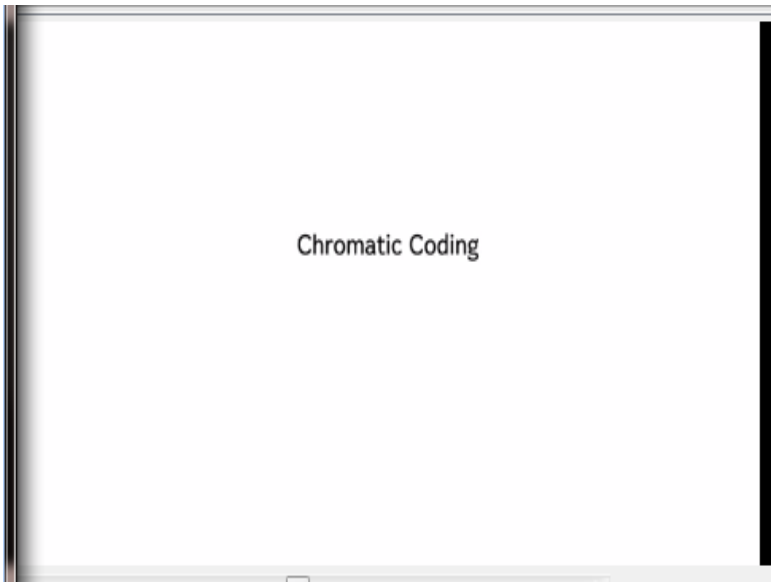
Well, there could be quite a few of them but notice that each of these vertices must participate in a triangle. The reason for this is the second reduction rule where we said we will eliminate any vertices that do not participate in triangles. And now notice that because this is a YES instance and we have set aside the arcs whose reversal guarantees that we are left with an acyclic tournament.

For each of the remaining vertices it must be the case that they participate in a triangle where one of the edges is in fact one of these reversed edges. Because if none of the edges of the triangle that a red vertex participates in is a reversed edge, then well this is not a valid reversal set. So, each of these vertices participates in a triangle with an edge from the set that is depicted on the top which is supposed to be a solution set in the setting of reversals.

But on the other hand from the first reduction rule we know that every edge in this reduced graph can only participate in at most k triangles. So, each of these edges on the top can only account for k vertices from the bottom. So, this tells us that the total number of red vertices could not have been more than k squared because there are k arcs on the top and each of them account for at most k vertices each.

Therefore if we were working with a YES instance then we could not afford to see more than k squared +2 k vertices. So, if the size of our instance exceeds this bound then we can say no that is our final reduction rule which leads to a kernel of size order k squared. Now this is something that will come into play in the final algorithm, so just keep this at the back of your mind.

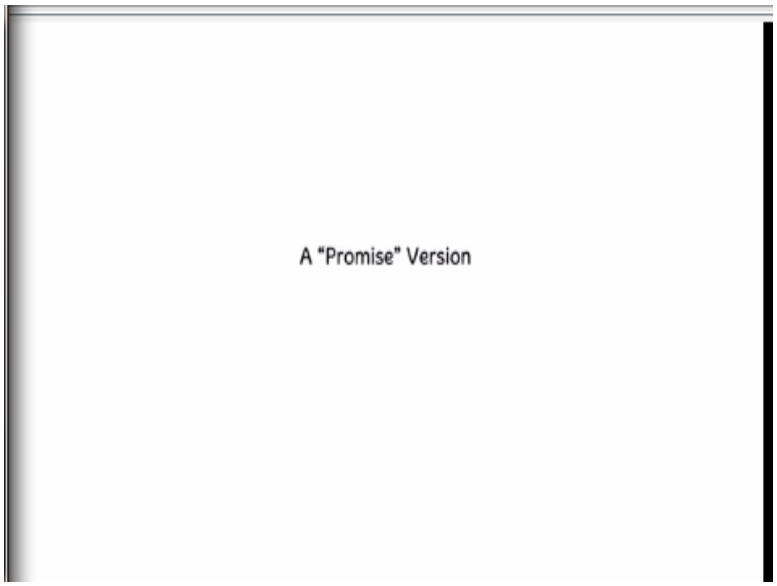
(Refer Slide Time: 24:37)



So, now let us turn to the randomized algorithm which uses a technique called chromatic coding and the reason for this name will become clear in just a bit. Now just like in colour coding where we wanted to randomly partition the vertex set of the graph. So, that the vertices of your solution fell into different parts. We want something similar to happen here, we want to randomly partition the vertices of the graph.

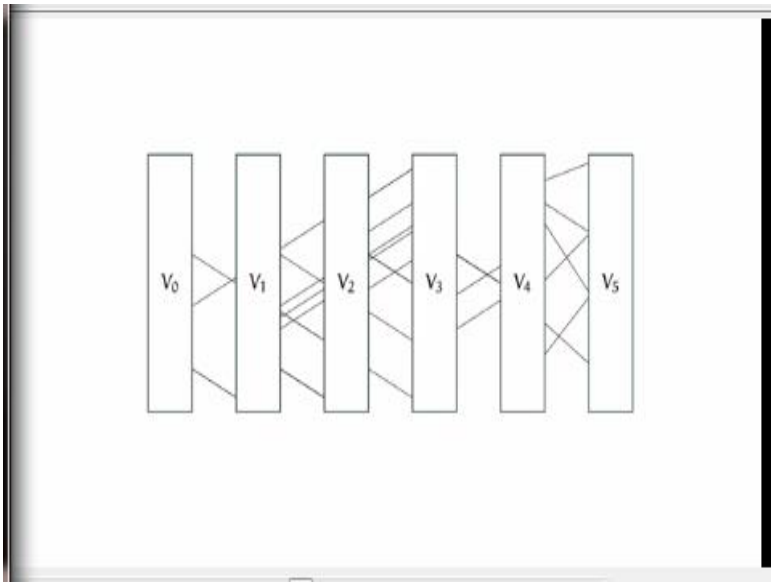
So, that your solution in some sense goes across these paths but now your solution is not a vertex subset, it is a subset of edges. So, what does it mean for a solution to be well caught by a partition?

(Refer Slide Time: 25:21)



So, let us talk about this by describing a promised version of the problem.

(Refer Slide Time: 25:23)



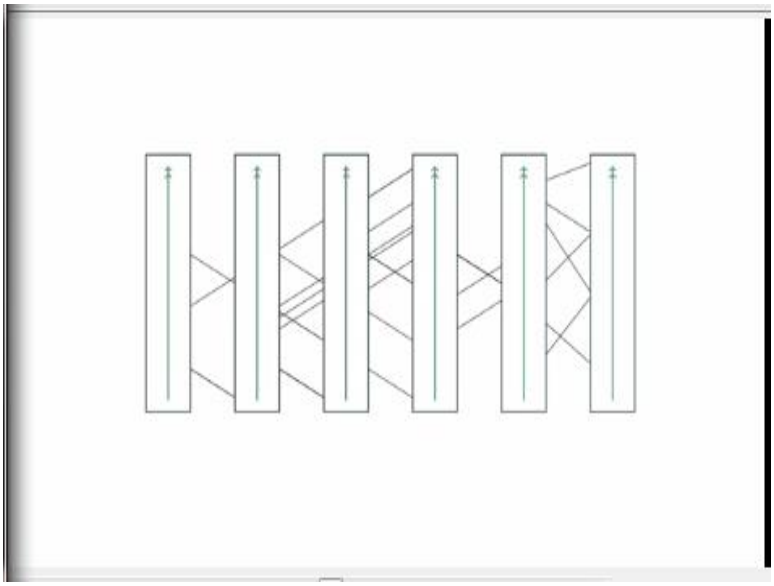
So, suppose I told you that you have been given a partition of the vertex set of the graph into some number of paths with the assurance that your solution only involves edges that go across these parts. So, there is never an edge from the solution which has both of it is endpoints in the same part of this partition. So, if you were to visualize this you could think of the edges as going across partitions, all the edges from your solution as going across partitions just as in this picture.

Once again this picture is slightly misleading because it is making it look like the parts have an ordering and the edges can only go between consecutive parts, it is really not like that. These edges can be all over the place but this is again just for clarity that it is shown in this way. Now once you do have this situation, is it an advantage does it help you identify a solution quickly? Well, let us think about this what leverage do we get if the edges never belonged inside any of the parts of this partition?

In particular does it impose some interesting structure on the individual paths? So, just visualize this for a minute, imagine that your graph has been partitioned into some number of paths and there is a subset of k arcs whose reversal does make the tournament acyclic. But all of these edges are going across the parts, no edge, no reversed edge has both of it is endpoints within a single part, within any single part.

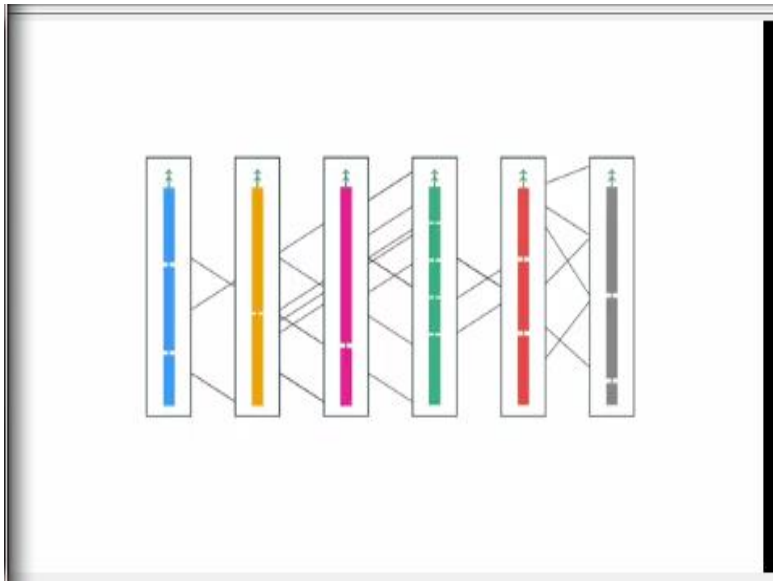
So, imagine that you go ahead and reverse these edges, the whole tournament becomes acyclic. With this in mind what can you say about the structure of the tournament when projected on any individual part? Think about this for a minute and come back when you are ready. Alright, hopefully you had a chance to think about this and perhaps you also came to the following realization which is that each part of this tournament must in fact induce an acyclic sub tournament, if the structure that I promised you was indeed true.

(Refer Slide Time: 27:36)



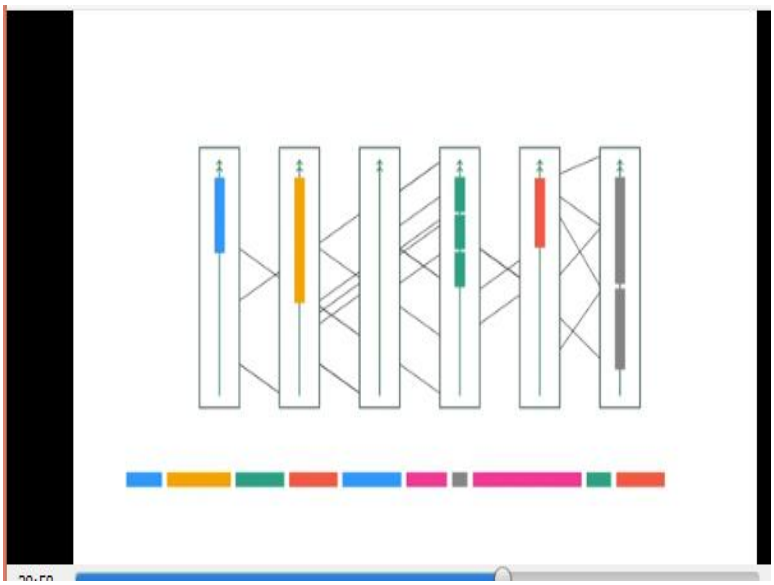
So, if you are only allowed to reverse arcs that go across the parts. Then notice that if you have for example a cycle which is completely contained in one of the parts, then it is impossible to fix this cycle by reversing arcs that go across parts. Therefore we conclude that any sub tournament induced on any one of these paths must itself have been acyclic. And therefore you can look at it in terms of it is unique topological sort, so that is the picture that we have here. And if you think about the final topological sort of the entire tournament, notice that it must essentially be a interleaving of these individual topological sorts.

(Refer Slide Time: 28:31)



Something that could look like this is just a again abstract example here but hopefully it just gives you a sense of what is going on. So, notice that you cannot mess around with the relative order of the internal topological sorts, you must maintain them.

(Refer Slide Time: 28:53)



But this is what a final topological sort could look like once the solution arcs have been reversed and they have done their job. So, let us just step back for a moment here and remember what we are doing? So, we are trying to understand why it is good for us if we are given the promise that all the solution arcs go across paths and never sit within. So, from this promise we have already derived some structure on the individual parts and now the question is can we leverage the structure to come up with an algorithm that actually quickly finds a solution.

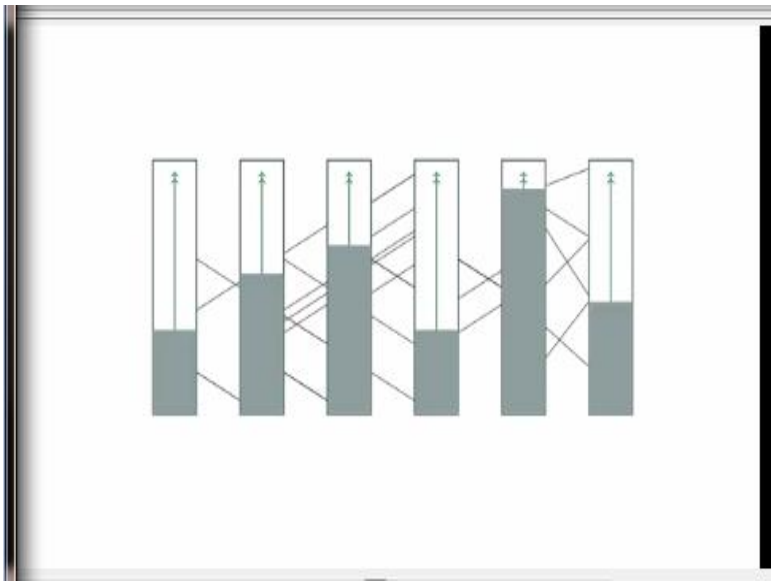
So, the analogous situation for instance in colour coding was when we were given a partition of the vertex set into k paths and we were promised the existence of a colourful path. So, back then if you remember we first did some brute force followed by a BFS and then we eventually improved it to a dynamic programming routine which actually did the job of finding the colourful path if it existed.

So, we need to do something similar here, we need to come up with an algorithm that can actually identify the arcs that need to be reversed. So, the picture is that you have been given a partition into some number of parts and there are all of these edges that are going across and some subset of them is going to do the trick for you and you need to identify them. So, once again what we are going to do here is use dynamic programming.

And just to establish some intuition for the states of the dynamic programming table. Let me say that what we are going to try and build up on as partial solutions is just to have an understanding of what is going on in the first eye positions of the final topological sort. So, we are really imagining the final solution after the arcs have been reversed as a topological ordering on the entire tournament.

And we want to kind of guess how much of which part appears in the first eye locations of this topological sort. We do not know exactly how much each part is contributing to the first eye locations.

(Refer Slide Time: 31:02)



So, what we are going to do is actually guess this. So, we are going to go over all possible ways in which you can get i vertices the first i vertices from all of these different parts. Notice that because you have to respect the relative order within each part. If I say that the first part contributes let us say 3 vertices, it has to be the first 3 vertices from the local topological sort induced by that part.

So, we make these guesses and we will come back and talk about the expense of making these guesses. You can probably already anticipate that let us say these parts have sizes n 1, n 2 and so on up to NP if there are P parts. Then the amount of information that we are trying to store is n 1

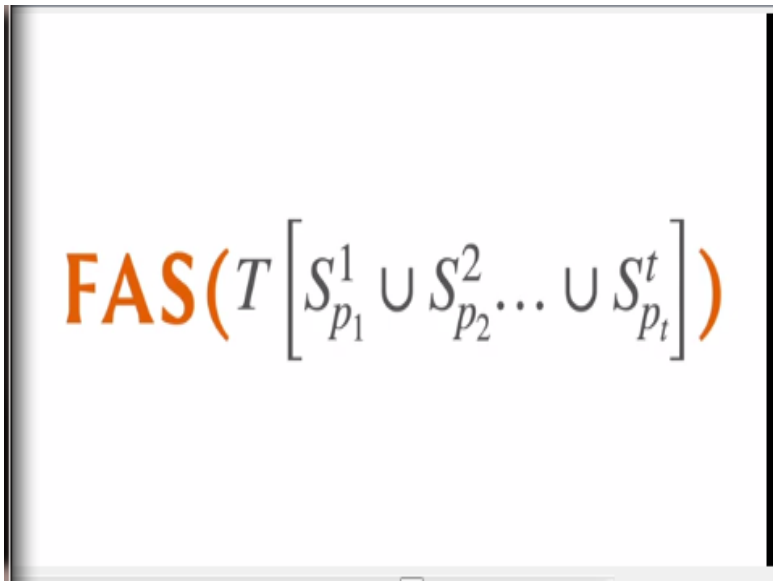
time, n 2 times n 3 so on up to times np. Because that is the number of ways in which you can cut across these parts for every choice of i.

So, that is a pretty big DP table and once again we will come back and talk about how we can control the size at this point? We do not even know how many parts there are, so we will kind of revisit the complexity. But for now let us focus on what we are trying to store and what the recurrence is going to be? So, well I mean for this particular partition, so we are saying that suppose this is what the first eye locations looked like it picked n1 vertices from the first part and 2 vertices from the second part and so on.

So, if it did that then what is the smallest cost of establishing a topological sort which had the pieces organized in this fashion? Notice that if you did compute this at every intermediate stage then when you are picking up all the n1 vertices from the first part, all the n2 vertices from the second part and so on, then at the very end you do get your solution. So, you have the minimum cost of arranging everything in a topological sort.

And it is easy to do some extra bookkeeping, so that you actually recover the arcs that need to be reversed as well and I will leave that part as an exercise. So, now that we know that the DP table contains the answer that we are looking for. We are hopefully motivated to come up with a recurrence for it. But just in case the states of the DP table were not completely clear or you found this picture a bit vague, let me also put it down notationally.

(Refer Slide Time: 33:45)



So, what we are interested in computing is the optimal feedback arc set or the value of the optimal solution for the following sub tournament. We are looking at the sub tournament that picks the first p 1 vertices from the first part, the first p 2 vertices from the second part and so on up to the first p t vertices from the tth part which is the last part. So, we are using t to denote the number of paths and notice I kept saying the first so many vertices from each part.

So, this first so many is with respect to the unique topological sort that we have imposed on each part just by virtue of the fact that we are working with this promise variant of the problem, where we have been told that we can assume that the edges of sum solution are going across parts. So, hopefully this notation makes it clear what exactly we want to store. And once again just to recap each p i will range from 0 to n i, where n i is the number of vertices in the part i.

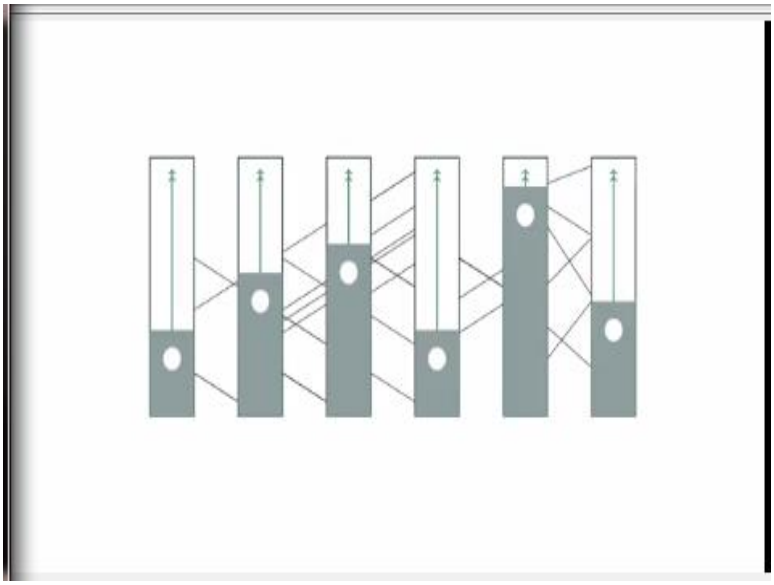
And therefore the total size of this DP table is the product of the sizes of all of the parts. And once again the solution is simply the DP table entry when each p i is equal to n i. So, now you might want to take a pause here and think about what would the recurrence look like? How are you going to actually compute this value? And one hint is to think about, well, we have this we have to compute the solution for this particular sub tournament.

A useful thing to think about might be what is the last vertex in an optimal situation which is to say that suppose we hypothetically had access to an optimal solution for this sub tournament and

suppose we reverse all of the arcs that correspond to that solution and write down the unique topological sort. Well, there is some vertex which sets at the end of this topological sort, what vertex can that be? Is it constrained in some way to be some special vertex from each of the parts? Can you say something about this?

And based on this can you come up with a recurrence? So, give that a thought and come back once you are ready. So, hopefully you had a chance to spend some time on this. I am going to highlight for you some special vertices from each of the parts that are under consideration.

(Refer Slide Time: 36:28)

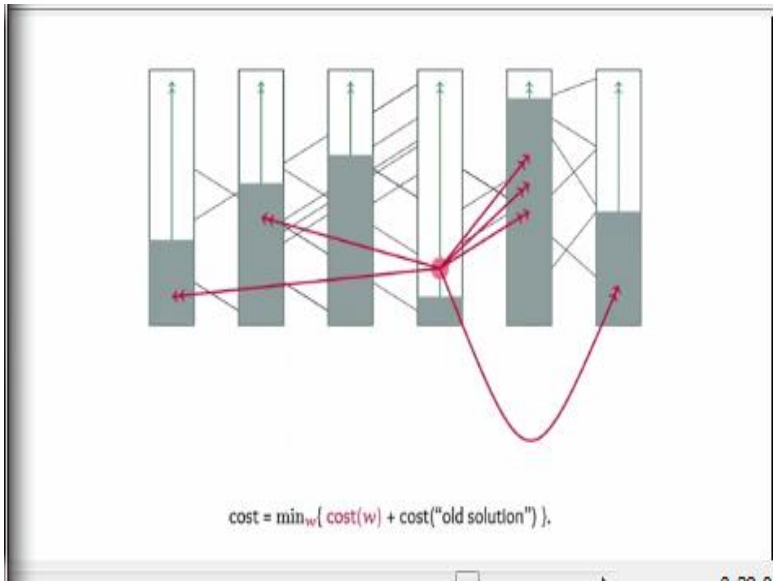


So, these are the last vertices from the chosen subset of vertices in each of these parts. Notice that in any optimal ordering which is to say that if you focus on the topological sort obtained after reversing the arcs of a solution that is local to this sub tournament, the sub tournament that has been highlighted. In this topological sort the last vertex must be one of these white vertices that have been pictured.

It cannot be anything else because if it was something else, then you would have destroyed the relative order between 2 vertices in the same part, which is not allowed. So, now that we know that these are all the candidates for being the last vertex on a hypothetical optimal order even if we do not know which one of these vertices is the last one. Given that there are only T of them to try out.

We can simply go over all of them, so this is what one would call guessing the vertex that comes at the end. For a specific guess what we need to figure out is, what is the cost of an optimal solution that places this particular vertex at the very end? This cost is going to have 2 components; it is going to be first the number of back edges that are generated by this very last vertex plus the cost of working with the rest of the graph which is essentially one DP table lookup.

(Refer Slide Time: 38:03)



So, let us fixate our attention on this one guess here. Notice that if we were to place this particular vertex at the end then a cost that we would incur is essentially the out degree of this vertex in the graph that we have seen so far. Because all of the out neighbours of this vertex would manifest as back edges in this ordering. So, these edges would have to necessarily be reversed. What happens beyond this is essentially something that we have already computed in the DP table where we looked at n1 vertices from the first part, n2 from the second and so on.

But ni - 1 vertices from the ith part assuming that we are working with the ith case and then the same number of vertices from all the remaining parts as well. So, essentially that is what I am capturing here is the cost of the old solution, you could write this down with proper notation and again that is just a fun exercise to work out. But this essentially captures the spirit of the DP recurrence.

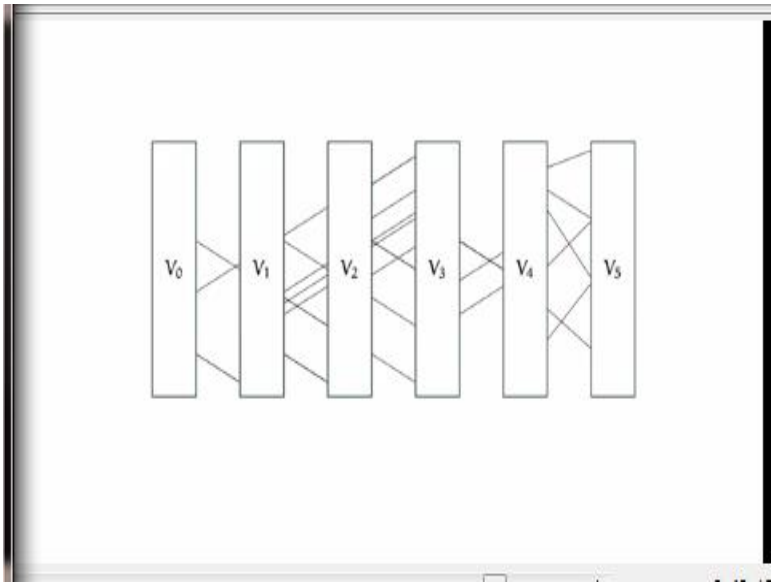
So, this min over w depicts the guessing that we are doing for the choices of the last vertex. But for a fixed choice of last vertex, it is going to essentially be the out degree of that vertex plus the remaining cost which can be pulled over from a DP table lookup. Now one thing that I am skipping in the description of this DP based algorithm is a description of the base cases but the base cases are fairly straightforward to fill out.

So, once again I would encourage you to fill in these little gaps and convince yourself that this dynamic programming algorithm works exactly as advertised. But now let us go back to the big picture and try to paste everything together. Remember that first of all we were working with this promise variant where we were told that the edges of your solution go across parts.

(Refer Slide Time: 39:58)

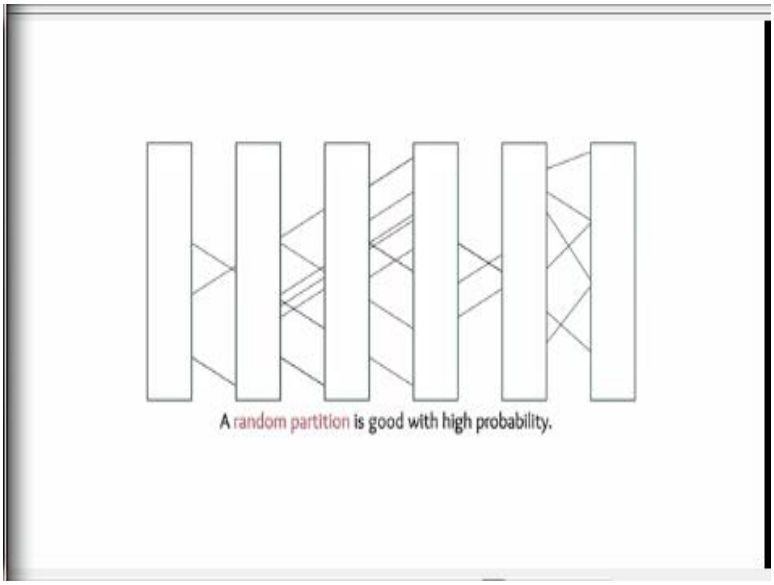
So, the first thing to do is to get rid of the promise and the way that we are going to do that is with randomization.

(Refer Slide Time: 40:05)



So, we are going to work with a random partition and hope that the edges of our solution actually have this nice behaviour.

(Refer Slide Time: 40:14)



So, what we want to say is just like we said with colour coding we want to establish here that a random partition is good with high probability. And notice that I have not even told you exactly how many parts we need to partition the vertex set into to ensure that we do have this quote unquote good probability.

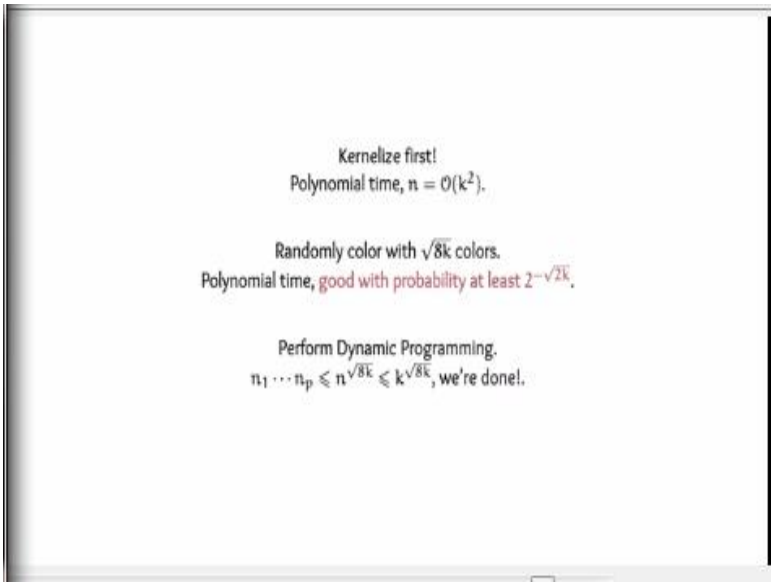
(Refer Slide Time: 40:37)

So, this is codified by the following lemma which I am again going to state here without proof in the interest of time. But if you are really curious you can again go and look up the notes on the website where we do have a bit of a sketch for why the statement is true. So, it turns out that if the vertices of a simple graph G on k edges are coloured independently and uniformly at random with just root 8k many colours.

Then the probability that this set of edges is properly coloured which is again as we said before a proper colouring is one that ensures that the end points of every edge get different colours which is exactly what we want. So, this probability is at least 1 over 2 to the root 2k. So, this is really helpful because notice that there is a sub graph on k edges and all we want is for that sub graph to be properly coloured.

So, this sub graph of course is the one that corresponds to the edges of a solution assuming that we have working with a YES instance. So, once again remember that we have working with these one sided error algorithms; we will never manifest a solution if one does not exist. But the only risk is that we may miss out on a solution if it was actually there. So, to ensure that happens with very low probability we are going to appeal to this lemma and of course do an appropriate amount of boosting. So, here is what the overall algorithm looks like.

(Refer Slide Time: 42:15)



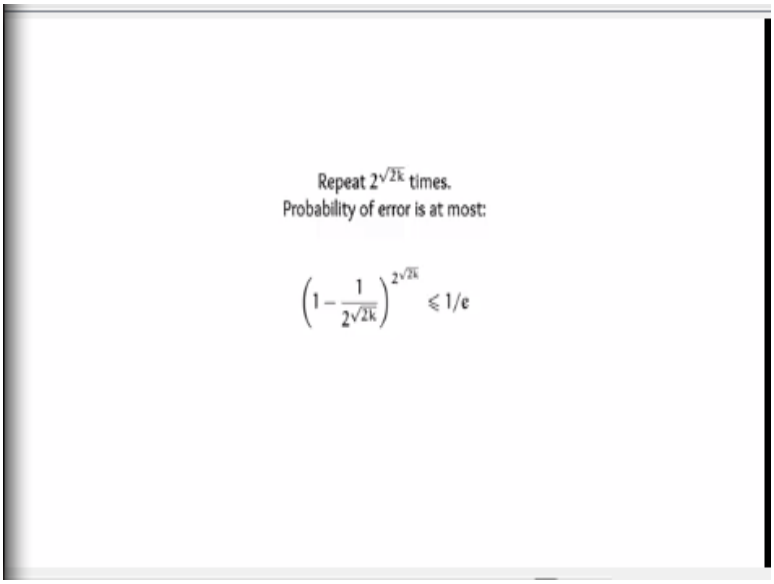
So, we are going to start with the kernelization which is a completely deterministic process and we will see in a minute why this is useful? What this ensures is that the number of vertices is brought down to order k squared and that is going to be our very first pre-processing step. After this we are going to randomly colour the vertices of the graph with root 8k colours, this is again a step that only takes a polynomial amount of time and is something that gives us a good colouring with probability at least 1 over 2 to the root 2k.

From here what we are going to do is apply a dynamic programming algorithm and we know that this is going to take time proportional to the product of the sizes of the parts. And we also know that the number of parts is exactly the number of colours that we use. So, p is square root of 8k and if we loosely upper bound each of the ni's with n, then we can say that this is at most n to the root 8k.

Now at this point this still does not look like a FPT running time and this is where the first step comes to our rescue. So, the fact that we have kernelized first means that n itself is bounded by order k square. So, this running time overall is bounded by k to the order root k and I think I have probably missed a constant in the exponent there but that is kind of the spirit of the running time.

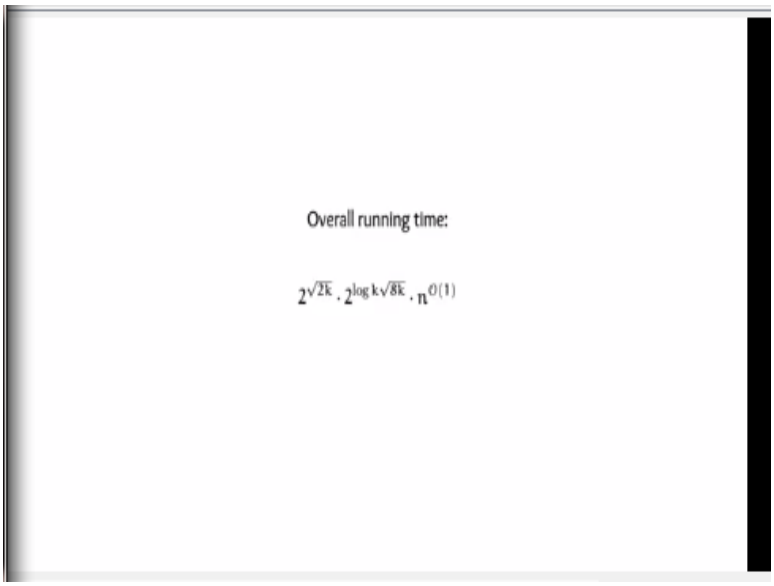
So, it is k to the order root k or in other words you could think of it as some constant to the root k log k. So, at this point we could say that we are done; of course we do our traditional exercise of boosting, so that we ensure a constant error probability.

(Refer Slide Time: 44:07)



So, notice that if you were to repeat this experiment 2 to the square root of 2k times then your error probability is going to be at most 1 over e.

(Refer Slide Time: 44:18)



So with that boosting in place this is going to be the overall running time of your algorithm. The first term comes from the repetition of the random experiment and the second term comes from the dynamic programming algorithm which has been applied to a kernelized instance. So,

remember that is important of course there is an additional polynomial time overhead for running the kernelization in the first place.

But this is going to be the dominant term in capturing the running time, so we are going to leave it at this. So, I think this is a very cool technique, it uses an interesting combinatorial fact about the way proper colourings work, the number of colours that you need to be able to properly colour a graph on k edges. Of course we did state this without proof and it is a really nice puzzle to try and figure out even if you do not get to the root 8k bound it is useful to think about coming up with some non-trivial bound.

And as I said if that is something that interests you please do check out the notes on the course website which have also been linked to from the description of this video. So, that brings us to the end of chromatic coding. And in the next video we are going to look at the problem of sub graph isomorphism which is something that we only briefly alluded to in the previous module when we were talking about colour coding.

We said that, well, there is this more general problem of trying to find small patterns in large graphs and we are going to tackle that problem head on. Again perhaps not in it is full generality for good reason but at least we will be able to come up with an interesting algorithm for a fairly large class of graphs. So, I will see you in that video, thanks so much for watching and bye for now.