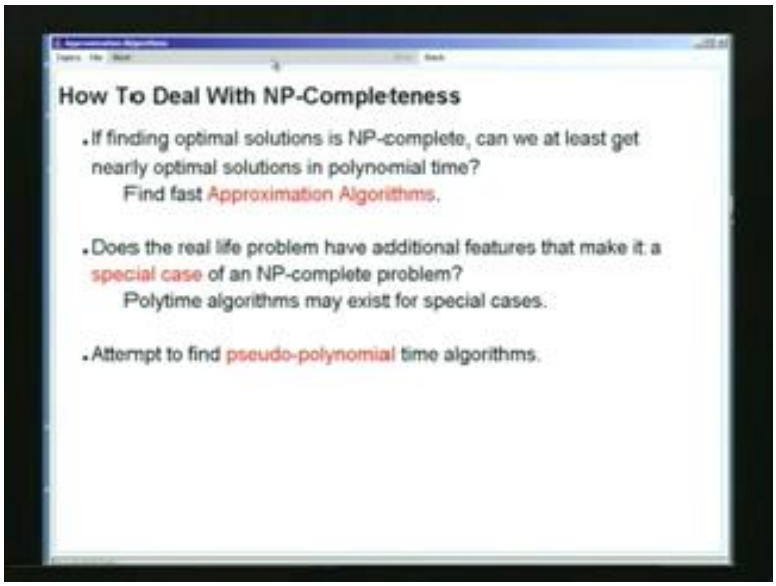
Design and Analysis of Algorithms
Prof. Abhiram Ranade
Department of Computer Science & Engineering
Indian Institute of Technology, Bombay

Lecture - 32
Approximation Algorithms for NP- Complete Problems – I

Welcome to the course on Design and Analysis of Algorithms. The topic for today is Approximation Algorithms for NP Complete Problems. So, let me start with a question, suppose we have an NP complete problem, which we need to solve. So, you wanted to solve a certain problem, which arose in some real life situation. And it turned out, that it was NP complete. What you do? This is going to be the subject of the next two three lectures. How do we cope with the problem, which is known to be NP complete? Usually, NP complete problems arise, when we talk about optimization problems.

(Refer Slide Time: 01:36)

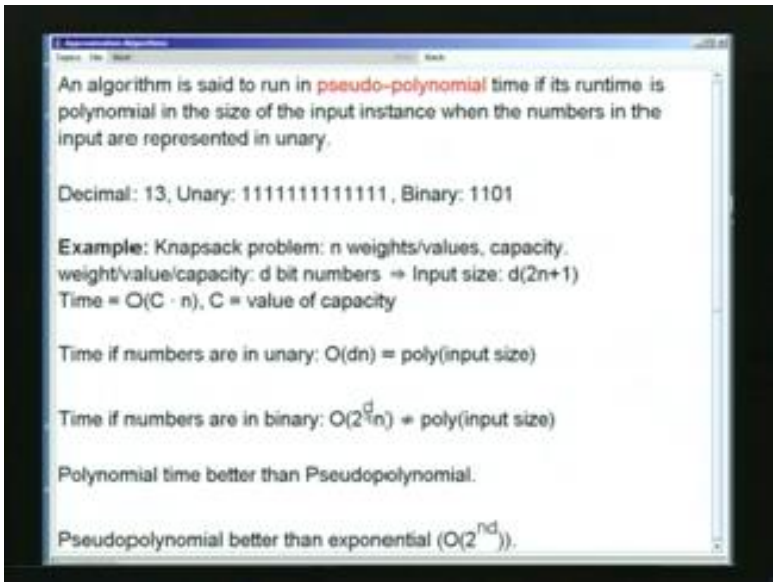


So, if finding an optimal solution is NP complete. One wonders whether, we can at least get nearly optimal solution in polynomial time. This approach is actually, quite promising. And it is the approach of finding fast approximation algorithms. It will not be interested in this two three lectures in finding the optimal solution. But, we will be interested in finding an approximately optimal solution. And therefore, we will be devising algorithms, which are called approximation algorithms.

And our hope is that for the real life application that we are worrying about, the approximately optimal solution that we find is also fairly useful. Another possibility is to examine, whether the real life problem that we want to solve has additional features that make it a special case of an NP complete problem. If this is true, then sometimes the special cases can have efficient algorithms, can have polynomial time algorithms.

So, for example, vertex cover is NP complete. But, if you are finding about X cover on a tree or on the bipartite graph. We do have fast algorithms, polynomial time algorithms for solving such problems. So, it is useful to think about, whether the real life problem that we are solving has any special features. Sometimes those special features can be exploited to get fast algorithms. Another possibility is to find what are called pseudo polynomial time algorithms. So, let me explain this a little bit more.

(Refer Slide Time: 03:31)



An algorithm is said to run in pseudo polynomial time. If it is run time is polynomial in the size of the input instance so far so good. So, far like the usual definition. Here is a difference. The run time is polynomial in the size of the input instance. When, the numbers in the input are represented in unary. In the normal definition or in the definition of polynomial time, we require that the numbers be represented in binary or in some radix, which is larger than 1. What happens, if you represent them in unary?

So, just to clarify, if we have a number 13 that we represent, that we want to represent. In the unary number in the unary representation system. It will be represented by a string of

13 ones. So, note for one thing, that this representation is going to be much longer, than this representation. If you look at binary, this is going to be the representation. But, this is still substantially smaller over here, over than over here. In fact, the number of bits needed over here is log of this.

So, there is going to be a bit difference between the length of your input instance, when measure it in unary or in binary. So, naturally if you are only interested in devising algorithms, which run in time polynomial in the length of the unary representation. You have got a lot more freedom to work with your algorithms can take somewhat longer. Then if they were to be running in time polynomial, in by in their binary representation, when the numbers are represented in binary.

We have in fact see in pseudo polynomial time algorithm in this course. So, this was the napes the dynamic programming algorithm, that we saw for the knapsack problem. Let me remind you, what the problem was you are given n items specified by their weights and values. And you are given an integer capacity for a knapsack. All these where integers, this description applies only 20 ((Refer Time: 05:51)). Now, it was assumed implicitly, that the weight value and capacity are in d bit numbers.

So, since there are 2 n weights and values and one capacity the input size is d times 2 n plus 1. And if you remember, we showed that the time taken for the knapsack problem was O of C times n, where C is the value of the capacity. So, this is crucial it is not necessarily the length of the bit string needed to represent the capacity. But, it is actually the value of the capacity. Now, if these numbers are represented in unary, then the time taken is O of d n. Because, then C would be d bits long.

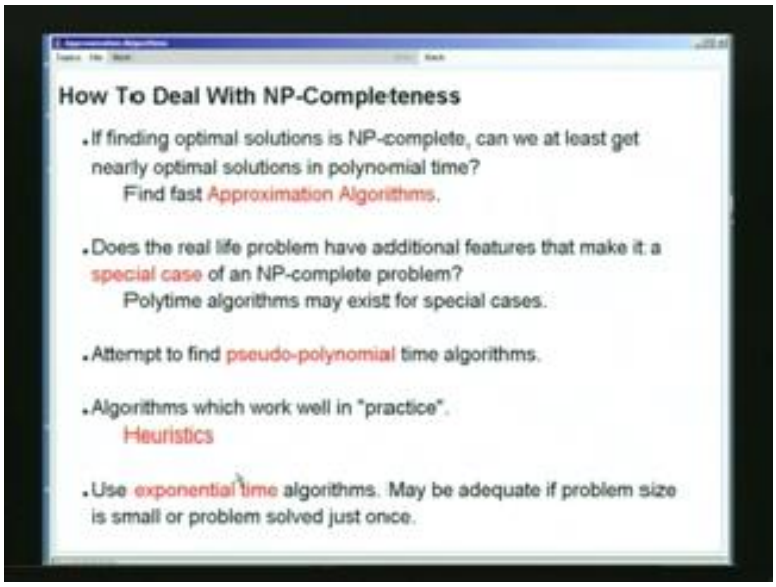
So, this C itself would be d as would be smaller than d. And therefore, there is no problem, the time taken would be C times n. But C times n is also d times n is at most d times n as well. And this is certainly polynomial in the input size, because the input size is just this. In fact, it is linear in the input size. However, if the numbers are represented in binary what happens. Well, if the capacity is represented as a d bit binary number. Then C can be as large as 2 to the d.

So, then this time O of C times n really could be as large as O of 2 to the d times n. Now, notice that this expression 2 to the d times n is not polynomial in this expression. Whatever, power you take of this, whatever constant power you take of this you cannot

beat this and therefore, this is not polynomial. So, clearly polynomial time, if you can get polynomial time it is better than pseudo polynomial time. So, pseudo polynomial time is not the best possible.

Or it is really different from our notion of good algorithms, which are polynomial time algorithms. However, pseudo polynomial time is better than exponential time. So, that is also worth noting, because the length is O of n d. And exponential would be something like 2 to the power n d. So, here we are getting d in the exponent, but we are getting n not in the exponent n just as a multiplier. So, this is certainly, still much better than this. So, as a compromise between polynomial time and exponential time. It is useful to think about whether, there are pseudo polynomial time algorithms, possible for the problem that you want to solve.

(Refer Slide Time: 08:52)



Then people do look at algorithms, which are difficult to analyze, but instead of analyzing them. They try out lots of instances and check whether the algorithms run fast enough. This is what I mean by saying we try to discover algorithms, which work well in practice, such algorithms are called heuristics. And they do tend to be useful, when solving problems which are known to be very hard. So, often it may turn out that you may have a good heuristic, which you really cannot analyze.

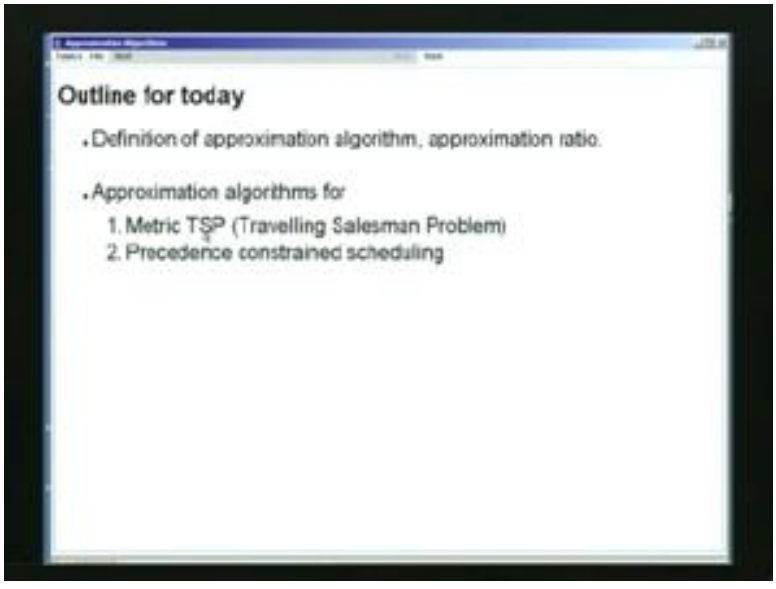
But, it seems to do the job, if it seems to do the job why not yourself. The last idea, which is also used is to use the exponential time algorithm. So, if a problem is NP

complete, we know that it can be solved by an exponential time algorithm. So, we use that exponential time algorithm. If the problem size is small or small enough, then the time taken may be acceptable. Or, if the problem is just to be solved once, then even if the problem takes a day does not matter.

We will run a computer for a day and get a solution. So, this also works sometimes, sometimes for solving real life problems. The real life problems tend to be reasonably small and today computers are getting really fast. So, exponential time algorithms can work, it is not that they are entirely useless. Our focus at these lectures however, is going to be on approximation algorithms.

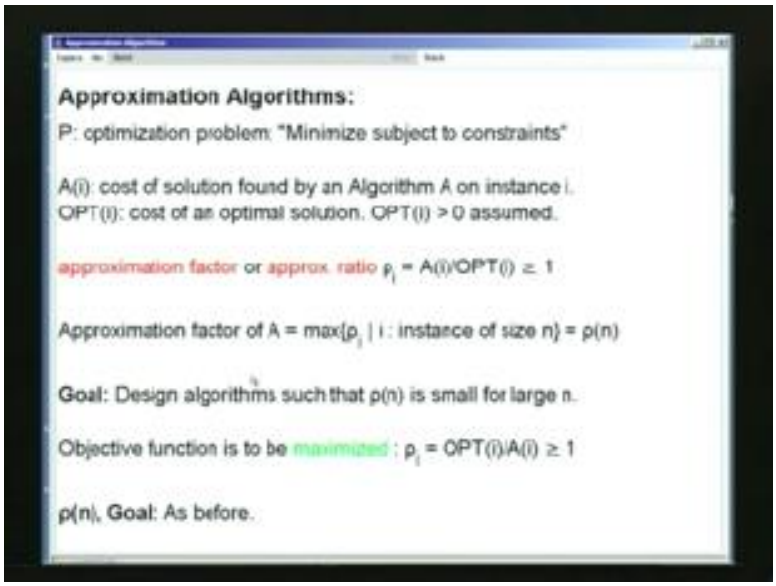
We would like to device algorithms, which are provably fast which are running in polynomial time, that is all that we mean in this, in these two three lectures when we say provably fast, that there are in polynomial time, and while they may not give the optimal solutions. We will prove that they will give somewhat close to optimal solutions.

(Refer Slide Time: 10:49)



So, here is the outline for today. So, I am going to define the notion of approximation algorithms. I will also define a term called the approximation ratio of an algorithm or an approximation factor of an algorithm. And then, I will describe approximation algorithms for two problems. One is the metric traveling salesman problem. And another is the precedence constrained scheduling problem. So, let us begin with the definition of approximation algorithms.

(Refer Slide Time: 11:22)



So, let us say P is denotes an optimization problem. P is an optimization problem and it look something like minimize this objective function subject to these constraints. Of course, it could be maximize, but for definiteness let us consider minimization first. Let A of i denote the cost of the solution found by an algorithm A on instance i. So, we are not worrying about the time right now. We are worrying about the objective function cost. So, we want this objective function cost to be as small as possible.

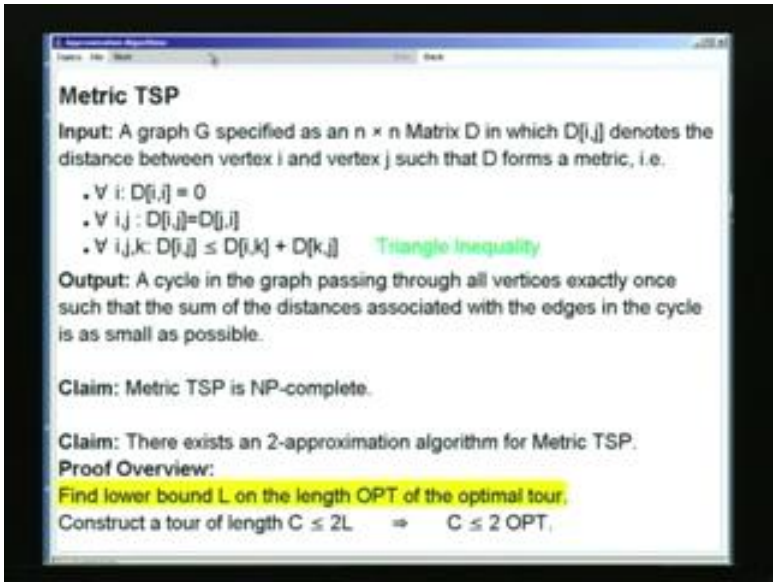
But, this algorithm A, when run on instance i produces this objective function value. Suppose, OPT of i denotes the cost of an optimal solution to this instance i. For technical reasons will assume that OPT of i is greater than 0, we will see why in a minute. Now, we define the approximation factor or the approximation ratio rho on instance i as A of i upon OPT i. What is the factor by which A is worse than OPT i. That is what this approximation ratio is all about. So, it is a natural definition.

Clearly, A of i the cost found by the algorithm can at best be as small as the optimal cost. In general, it could be larger and therefore, this rho sub i is going to be larger. And we would like it to be as close to 1 as possible. In general, the approximation factor of this algorithm is just the maximum value of rho sub i over all possible instances of size n. So, it is customary to use the worst case by enlarge. And so here, as well we are going to look at the worst case ratio.

And of course, it is going to be parameterized by the size n. So, we will write this as rho of n. So, for different n we will have a different ratio. So in fact, we are looking for, looking to evaluate this and we are looking to keep this small. The goal clearly, is to design approximation algorithms or algorithms such that, rho of n is small as close to 1 as possible for large n. And of course, the time for this algorithm is polynomial. The algorithm must run in polynomial time.

Sometimes, you want to maximize the objective function. In which case, we will define rho sub i, the approximation factor as the reciprocal. So, now we know, that A of i can at most be as large as this. And therefore, it will turn out. But, this is still going to be bigger than 1. So, again our goal is going to be similar. So, rho of n is going to be the same. And the goal is also going to be similar. We want algorithms, which keep rho of n as close to 1 as possible, which get rho of n as small as small as possible.

(Refer Slide Time: 15:09)

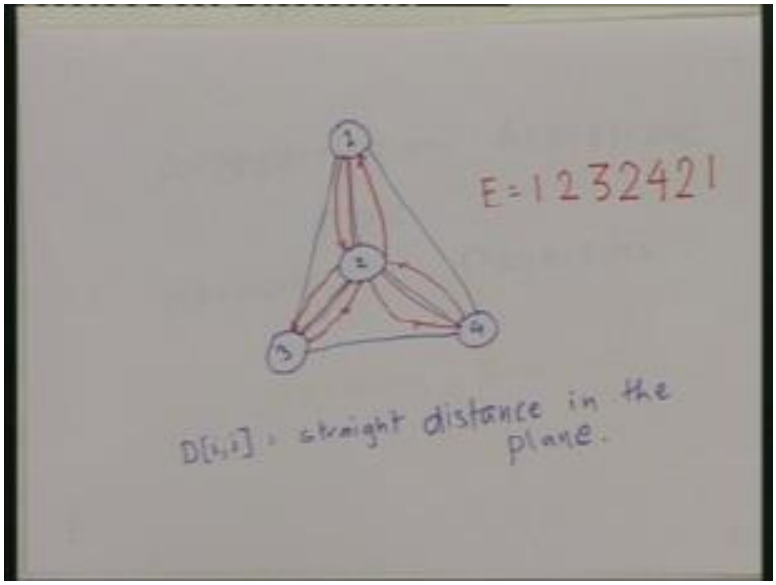


So, now we will use these ideas, to device an approximation algorithm for the metric TSP problem. So, let me define this problem first. The input to this problem is a graph G. And this is going to be specified as an n by n metrics D, in which D of i, j denotes the distance between vertex i and vertex j in this graph. Now, the metric in the title, says that D has to have certain additional structure. Specifically, D has to form a metric and by that we mean, first of all for all i the distance of a node to itself is has to be 0.

The distances have to be symmetric, the distance from i to j has to be the same as the distance from j to i. And the final thing is that for all i, j, k the distance of going from i to j directly has to be no larger than the distance of going from i to k first and from k to j next. This is often called a triangle inequality constraint. So, imagine that i, j, k are the vertices of a triangle. And this just says, that the straight distance going from i to j is smaller than the indirect distance.

So, let me first take an example of what a metric problem is going to be. So, I am not going to draw the metric, the metrics D, but I am just going to take the problem. And I am going to draw the graph corresponding to the problem. So, one way to use such a graph is to imagine that the vertices are embedded in the Euclidean plane.

(Refer Slide Time: 17:10)



So, for example, here is vertex 1, here is vertex 2, here says vertex 3, here says vertex 4. I could draw out all the edges, but even without drawing all the edges. Let me tell you, that the distance from i to j is simply the straight line distance in the plane. So, D of i to j is straight distance, straight Euclidean distance in the plane. Now, all of us know that the distance from here to here, plus the distance from here to here can never be smaller than the shorter distances from the straight distance.

And so clearly, our third constraint the triangle inequality constraint is obviously, applicable over here. So, for completeness I could write down this is the graph that here looking at. For example, and if you do the arithmetic you could say for example, that you

could calculate the distances. So, this is the graph. And if you look at D i, j to be the Euclidean distance. Then clearly, it will satisfy all these metric property, the properties mentioned over here.

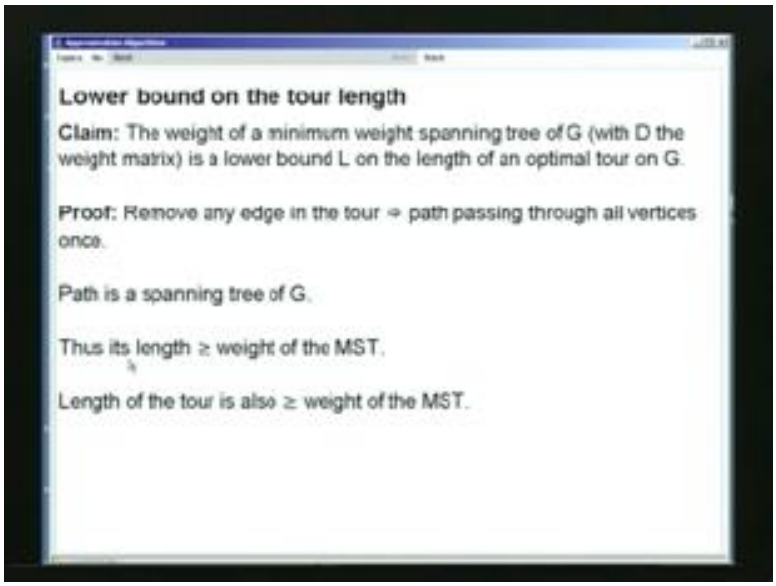
So, this would be a traveling salesman problem instance. What is suppose to be output ((Refer Time: 18:41)). What is suppose to be output is a cycle in the graph, passing through all vertices exactly once. Such that, the sum of the distances associated with the edges in the cycle is small as, is as small as possible. So, this is the same thing as in the TSP problem. You want to tour in the graph passing through every vertex. Such that, the tour length is as small as possible.

The first claim is that metric TSP is NP complete. Well, I think we have studied earlier, the TSP is NP complete. But, it turns out that even with these restrictions. So, this is the special instance of a TSP. But, even with these special restrictions TSP remains NP complete. Here is the claim, that we are interested in and which we are going to prove. So, the claim is there exist a two approximation algorithm for metric TSP. Here is a quick overview of the proof.

In fact, the proof is actually quite simple. The idea is actually quite interesting and but short. So, the general idea is this. And this scheme appears in other places as well. So, first we are going to find the lower bound L, on the length OPT of the optimal tour. So, whatever graph we are given, it has some optimal tour, will try to figure out a lower bound on it. Then, we will construct a tour of length C, which is at most twice this. So, notice that, it is very hard to figure out the length of the optimal tour.

So, we really want a tour, which has say twice the length of the optimal tour. But, rather than that we will find a lower bound, which will be easily computable. And we will show that we can construct a tour, which is at most twice the length. But since, this is a lower bound. We know that this is that C must also be less than twice OPT. Because, L is less than OPT. Therefore, C is less than twice OPT. So, this is going to be what we are going to do. So, we will look at each step in turn. So, this is the first step. So, we want to find a lower bound on the length of the optimal tour.

(Refer Slide Time: 21:00)



So, here is the main claim. The claim says that the weight of a minimum weight spanning tree of G, with D as the weight matrix is a lower bound L on the length of an optimal tour of G. So, this is the lower bound that we wanted. So, I just have to prove this. So, let us imagine that we are given any optimal tour. We take that optimal tour and we remove an edge in it, edge from it. What do we get? Well, we will get a path, which passes through all the vertices of the graph once.

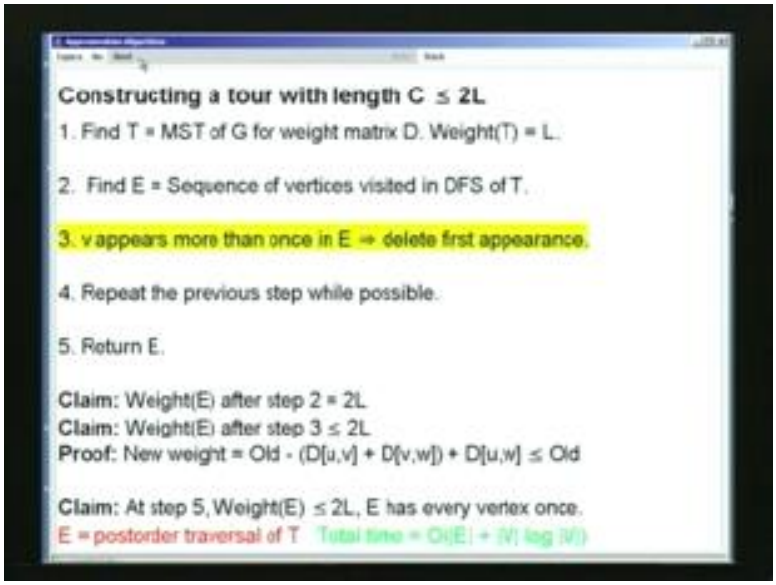
It starts at some vertex and it passes through all the other vertex and returns to some other vertex. But, is there anything interesting that we can say about this path. Well, this path is also a special case of a spanning tree. This is a spanning path, it passes through every vertex. And therefore, this also is a spanning tree of G. So, it is length, it is total length is certainly no smaller than the weight of the minimum spanning tree. Because, the minimum spanning tree is by definition, that spanning tree whose weight is the least.

And therefore, the length of this, which is the weight of the corresponding the spanning tree, by the way, in this case weight and length are to be use synonymously. Weight is the terminology used in connection with minimum spanning trees. And length in connection with tours. So, I am sticking to those, stick into that, but really length and weight are the same. So, the length of the path has to be greater than or equal to the weight of the minimum spanning tree.

It will be equal, if the path itself happens to be the minimum spanning tree, a minimum spanning tree. But, the length of the tour is bigger than the length of the path. Because, the tour in fact, contained an extra edge. And therefore, the length of the tour is also bigger than the weight of the minimum spanning tree. But, this minimum spanning tree has beat L. And therefore, we are done. So, the length of the optimal tour is bigger than L.

So, ((Refer Time: 23:32)) we have proved this. We have established a lower bound L on the length OPT of the optimal tour. Now, we want to argue, we want to construct a tour and argue that it is length is at most 2 times L. And once we are done, we will have proved our result.

(Refer Slide Time: 23:56)



So, here is have you construct a tour with length less than 2 times L. So, I am going to give you the algorithm. So, first we find a minimum spanning tree, which allows us to determine the L. So, the weight of the tree is L. We can actually, write this down. We can find the minimum spanning tree. And we can find it is weight and that is going to be L, the lower bound. Next, we do a DFS, a DFS traversal depth first traversal of T or do a depth first search of T. And we look at, the sequence of vertices that get visited.

And that sequences return out as E. So, let us take our graph and let us look at that sequence. So, here is our graph ((Refer Time: 24:49)). Now, we start at 1 and if we are doing the depth first search well there could be many ways, in which we do the depth

first search. So, first of all I have to identify, what this tree T is going to be. So, clearly this is going to be the tree T. So, this is going to be the minimum spanning tree in this graph the red edges.

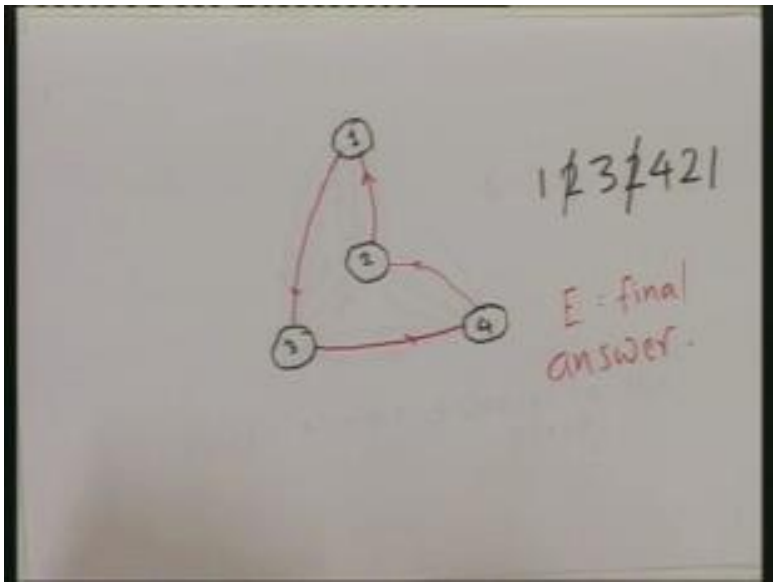
Now, if I want to do a depth first traversal of this tree, say starting at vertex 1, what would I get? So, from here, let me just use red again. So, from here I would visit 2. Then from here I would visit 3. Then I would go back, then I would go forward. Then I would go back, then I would go forward. And so the E that I get is going to be something like this. So, I start with 1, then I go to 2, then I go to 3, then go to 2, then I go to 4, then I go to 2 and then I go to 1. So, this is going to be my sequence E, so 1, 2, 3, 2, 4, 2, 1.

So, this is how I have constructed E. Now, the idea is that if D appears, more than once in E. So, there are several vertices, which appear more than once. We are going to delete it is first appearance. So, 1 appears more than once, but this 1 is really the ((Refer Time: 26:28)) same as this 1, because the tour is just closing. So, we do not worry about this. So, the first that appears more than once is this 2. So, now we are going to delete it. And we are going to replace it by the direct edge.

So, we are going to delete 1 to 2 and 2 to 3. And we are going to replace it by a direct edge. Or maybe I will use black this fine, this will be perfectly understandable. And then, we are going to repeat the previous step while possible. So, if a vertex appears several times, we are going to short circuit it, we are going short cut it. So, our current E now is going to be this, we have just removed this. ((Refer Time: 27:14)) So, the next vertex that appears twice is 2.

It already appeared, but it is going to be 2. It could be another vertex, but in this case it just happens to be 2. So, we are again going to delete it is first occurrence. So, if we delete it is first occurrence what does it mean, instead of going from 3 to 2 and 2 to 4. We are going to remove these edges and we are going to replace it with this direct edge. So, we are going to keep on doing this step as many times as needed. And at the end we are going to return E. So, let me draw another picture to show you what this E, that was that is to be return is…

(Refer Slide Time: 28:02)

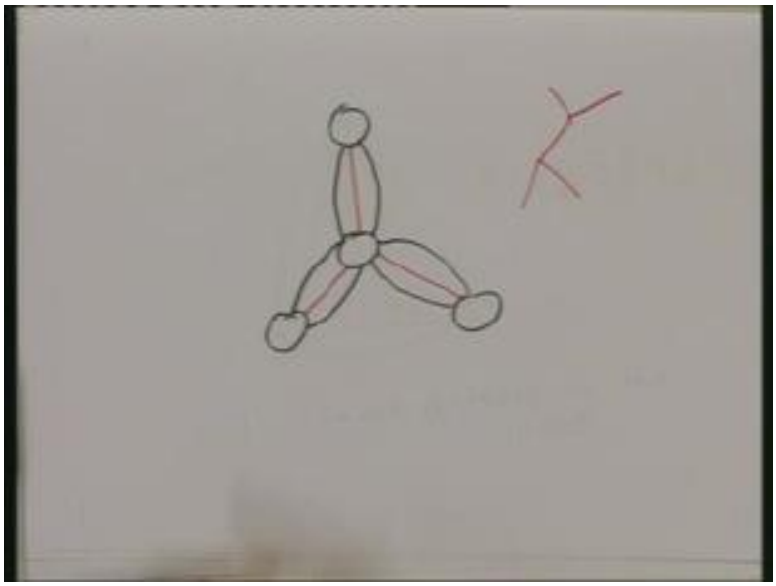


So, this is our vertex 1, vertex 2, vertex 3, vertex 4. So, in our so original tour was 1, 2, 3, 2, 4, 2, 1. So, we removed this 2 and then we removed this 2 and we were left with this. So, our E that remains at the end is going to be this So, going directly from 1 to 3, then going directly from 3 to 4 then 4 to 2 and 2 to 1. So, this is the E that we would be returning. This is the claimed final answer. So, let us go over each step. And we will very about how exactly it is done.

So, ((Refer Time: 29:10)) the first step is done as is finding the minimum spanning tree. So, how long does it take? Well, if we use prims algorithm it will take something like E plus V log V, so clearly polynomial time. How long does this take, this is just depth first search. So, it takes time linear. In fact, so this time is less than this time. So, now let us worry about. So, these steps are again going to be fairly straightforward. So, let us not worry about the time. But, let us worry about the correctness.

So, once we find this E, we eventually we modify it and eventually return E. So, let us try to figure out some properties of E. So, my first claim is that weight of E after the step 2 is going to be twice L. So, that actually, is obvious from this picture. But I will just draw it again.

(Refer Slide Time: 30:19)



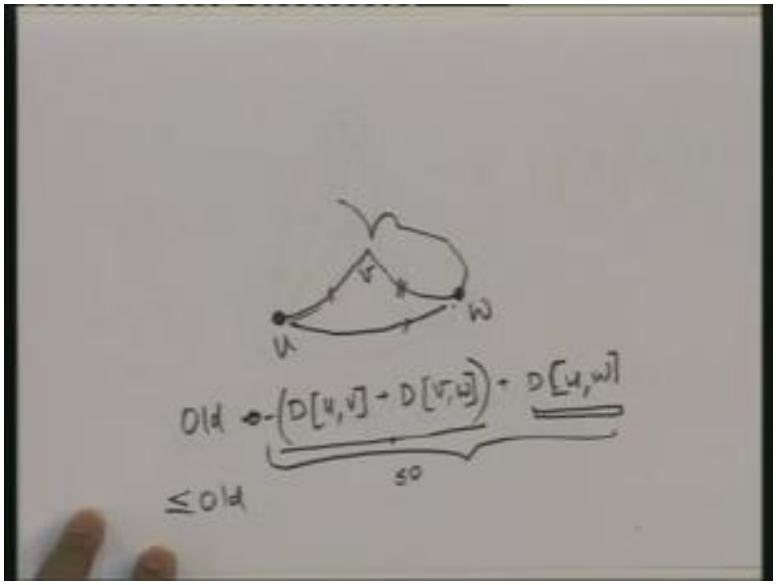
So, our graph was this. And our tour was this. So, notice that our tour used every edge our tour E, the original value of tour E used every edge twice. This is going to work in general, yes. On every tree it will work, because no matter what do you have. When you do the tour starting from any edge, you go down an edge and then eventually, you come back up. And you have to do it exactly once. So, clearly every edge will appear twice ((Refer Time: 31:01)) in this E.

And therefore, the weight is going to be twice the length of the tree, twice the total weight of the tree. But, the weight of the tree itself is L. And so the weight of E after the step 2 is going to be twice L. The other property about E that is important is that E must contain all the vertices in T. E has to contain all the vertices in T. So, it is a tour the only problem is that it contains some vertices more than once. And that is why, it cannot be it is not a good tour for us.

So, if v appears more than once in E, we remove the first appearance. So, this is good. Because, we are going towards making sure that every vertex appears only once. But, what is this due to E in particular does it do anything bad to the weight of E. So, here is the important claim. The claim says, that after step 3 weight of E is at most twice L. It can only decrease. Now, this is the main part of the argument. And the proof is actually, quite simple.

So, what is the new weight? The new weight is the old weight. And suppose v was the edge we deleted. So, let me take a picture to explain this.

(Refer Slide Time: 32:25)



So, this was a portion of E and this is the vertex V, which we are deleting. How do we deleted, we take the previous vertex, which we call u. Let us call it u, we take the next vertex, we call it w. And we removed these two edges. And we put down this edge. So, what happens to the total weight? Well, the total weight now becomes the old weight plus what we put in or minus what we removed. So, minus what we removed is so minus of D of u, v.

And we also removed this plus D of v, w. And we put in u, w. So, this is the new weight. That is what we have written down over here. But, notice that these are two sides of a triangle. And this is the third side essentially. So, this is the straight path and this is the cross path. So, which of these two is bigger? So, clearly this one is going to be bigger, if at all. And therefore, we know that this entire thing has to be less than or equal to 0 or therefore, the whole thing is at most old.

So, we have proved that the new weight is at most the ((Refer Time: 34:01)) old weight. The old weight was twice L. And so the new weight is also twice L. So, if we keep on repeating this step as many times as we can what happens, the weight keeps on reducing. So, it will always be bigger than, it will always be smaller than twice L. So, we have

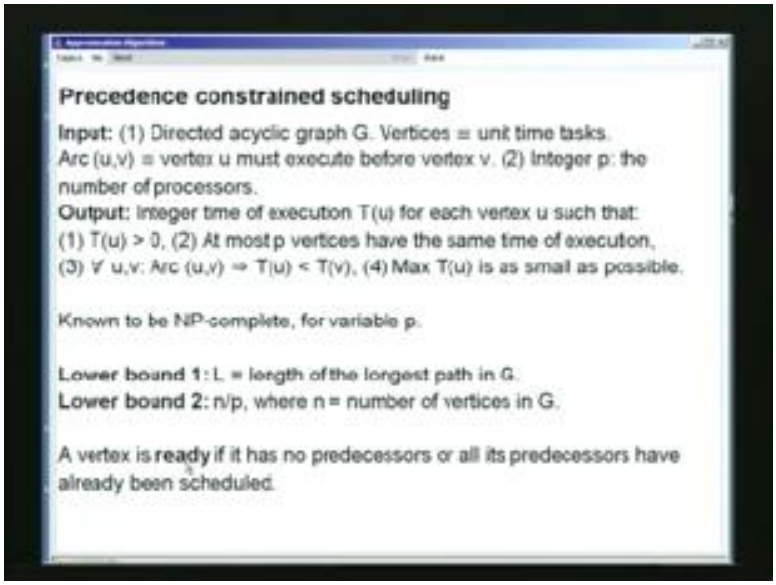
proved that the weight of E is always going to be at most twice L. So, the final claim is that just before we return of course, the weight is going to be at most twice L.

But, E is also going to have every vertex exactly once. Why is that? Well, we repeated until no vertex appeared more than once. So, clearly no vertex appears more than once. But, initially every vertex did appear at least once. And therefore, finally, every vertex appears exactly once. And so therefore, E is a tour every vertex appears once. And it is weight is twice L. L is lower bound and therefore, we are done. The final issue is there might be some question about the time required for this part.

So, here is a very nice simple observation, which says that this entire thing can be done in linear time. What does this loop do? So, it says, if v appears more than once, delete the first appearance. But, what if v appears several times, then we will delete all, but the last appearance. And this is going to be true for every vertex. So, we are going to keep only the last appearance of every vertex in this traversal E.

But, what is that, we know that. When you do graph search, we should do a post order traversal that is exactly what this is. And therefore, the time for this steps 3 and 4 together is at most the time for a breadth for a depth first search. And therefore, it is just O of the number of edges plus the number of vertices. So, the total time is just simply is dominated by the time for finding the minimum spanning tree. And therefore, it is E plus v log v, say using prims algorithm.

(Refer Slide Time: 36:41)



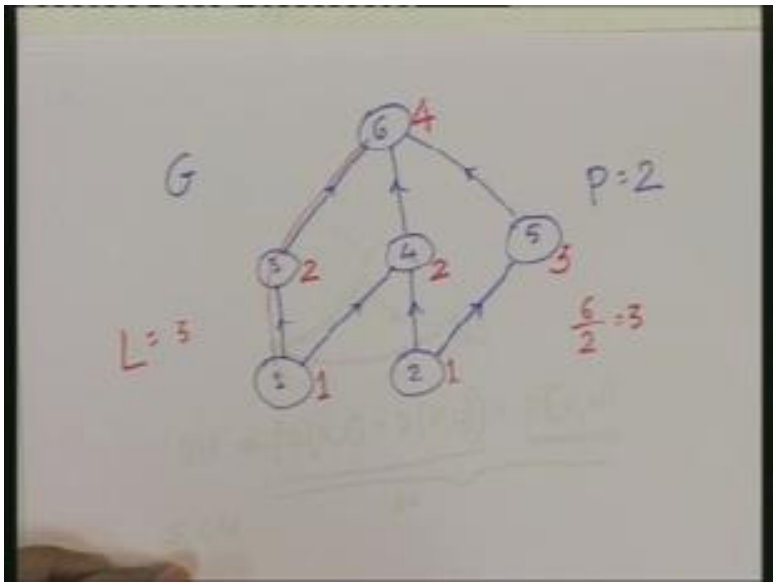
Let us now consider the next problem. The next problem is precedence constraint scheduling, which is also an NP complete problem. And we are going to find a polynomial time approximation algorithm for this. The input to this problem has two parts. The first part is a directed acyclic graph G. Vertices in this graph, represent unit time tasks. And there is an arc directed edge going from u to v corresponding to the restriction that vertex u must execute before vertex v.

So, there is a precedence constraint from u to v. And therefore the name of this problem. You are also given as a part of the input, and integer p, where p denotes the number of available processors. So, p is the number of tasks that you can perform at each step. You may not be able to find that many task. But certainly, you cannot perform more than p tasks at each step. For the output we require to specify an integer time of execution T of u for each vertex u.

Such that, first of all T of u is greater than 0. And at most p vertices has the same time of execution. Furthermore, if there is an arc from u to v. Then the time of u is must be strictly less than the time of v. Remember that, these are integers, so this really means less than or equal to so there is a difference of at least one. And finally, we want to minimize the length of the schedule. So, the maximum over all times is as small as possible. This problem is known to be NP complete for variable p.

So, if p, p changes p is allowed to change as a part of the input. Then this is known to be NP complete. Here is one lower bound. I claim that the length of the longest path in this graph is a lower bound. So, let us do that band. Let us see that and for that let us take an example as well. So, let us take a simple graph.

(Refer Slide Time: 39:32)



So, say the graph G looks something like this. So, here is vertex 1, which is one task. Here is vertex 2, which is another task. Then maybe there is vertex 3 over here. And there is an edge from 1 to 3, there is vertex 4, there is an edge from 1 to 4 as well, may be there is an edge from 2 to 4 also. May be there is a vertex 5. And there is an edge and say there is an vertex 6 with these edges. So, this for example is G. So, this is one part of the input. And let us say P is equal to 2. So we want to find a schedule.

So, I claim the first lower bound, which and that is claimed in the first lower bound. That no matter what you do, the length of the longest path is lower bound at the time required. ((Refer Time: 40:35)) So, the idea is actually fairly simple. So, let us identify a longest path over here. So, in this case, the longest path is quite simple. So, say for example, this is the longest path. There are several longest paths, but this is the longest path.

What is this length? Well, we are suppose to measure the length, in terms of the vertices, the vertex length. So, this has length 3. And the claim is that the length of the schedule must be at least 3. Why is that? Well. the precedence constraints says that, if this is executed at step 1. Whatever step it is executed, at this cannot be executed at the same step. So, it has to be executed one step later. This has to be executed one step further than that and so on.

So, whatever the length of the graph is that many steps are needed for this execution. So, that is the first lower bound. ((Refer Time: 41:36)) Second lower bound is based on how

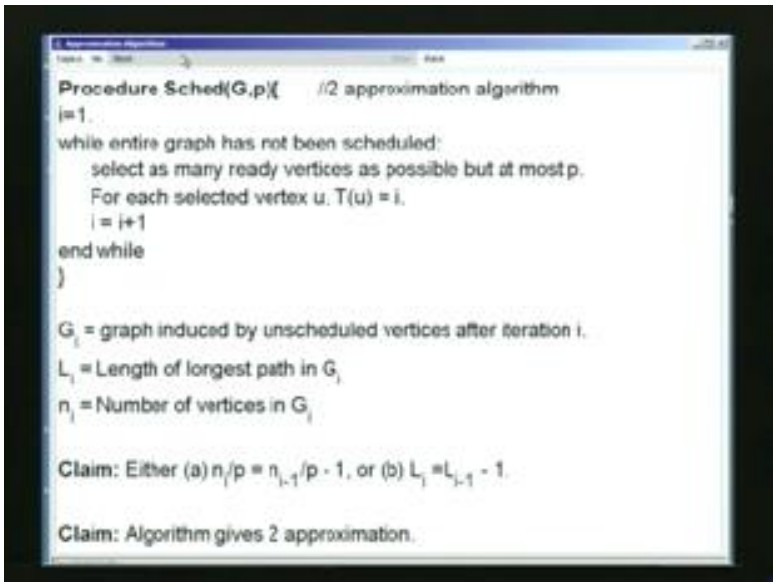
much load can be consumed at each step. So, if n is the number of vertices in G. How many time steps, how many how many vertices can be consumed can be worked on by the p processors at each step. Well, at most p and therefore, n over p steps are at least needed.

So, that lower bound in this case is 6 upon 2 which is also equal to 3. So, the first lower bound L is equal to 3 the second lower bound is also equal to 3. So, let us now consider, let us now examine whether in fact, the upper bound for this matches. So, is it match? Well, here is one possible schedule. So, we will schedule this at step 1. We will schedule this also at step 1. And in fact, that is our only choice. Next, we have these 3. So, we can pick say this we will schedule at step 2. This we will schedule at step 2.

This is ready to be scheduled, but we cannot schedule it, because we only have two processors. So, this has to be schedule at step 3. This we have a processor available. But, this cannot be scheduled at step 3, because this has to be scheduled only after this. So, this has to be scheduled at step 4. So, T of 1 and T of 2 are both 1's. T of 3 and T of 4 are both 2's. T of 5 is 3 and T of 6 is 4. So, in this case the upper bound in fact, is 4 and it is bigger than the lower bounds.

So, now I am going to describe the algorithm, which will get ((Refer Time: 43:36)) within twice the best possible schedule. And it will use these lower bounds. And it will also use the notion of a ready vertex. So, vertex is set to be ready or ready to be scheduled, if it has no predecessors. Or all it is predecessors have already be in scheduled. So, now I will describe the scheduling algorithm.

(Refer Slide Time: 44:04)



So, this is a procedure sched, which takes G and p and it is a 2 approximation algorithm. It produces a schedule, whose length is twice the optimal schedule as we will prove in a minute. So, here is the algorithm actually, it is quite simple, while the entire graph has not been scheduled. We select as many ready vertices as possible, but at most p. For each selected vertex u, we will set T of u equal to i. So, we will schedule it at step i. And then, we will increment the time and then we will repeat.

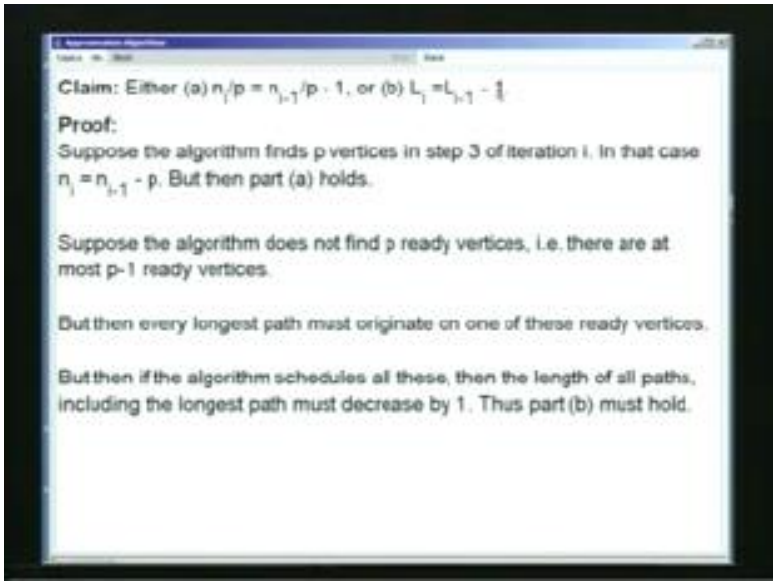
How long does this whole thing take? Well, the algorithm will take time the time required will be the time to identify this ready vertices. So, the ready vertices will be found by looking at by looking at vertices, which have already been scheduled. I will just say that this can be done very efficiently by doing a topological sort and in fact, you can do the whole thing in time linear in the size of the graph. So, this in fact, will run in polynomial time. So, it is easily shown that a topological sort will suffice.

Let us now, consider whether this is correct. So, is this correct well we are following the restriction about the number of processors, because we are only picking at most p vertices. We are following the restriction about precedence, we are because we are picking only ready vertices. So, this is going to produce a correct schedule a valid schedule. And it is going to run in polynomial time, the only thing that we need to prove that it is a two approximation algorithm.

So, let G sub i denote the graph induced by the unscheduled vertices after iteration i. L sub i is the length of the longest path in G sub i. So, remember that that is a lower bound on G sub i. Let n sub i denote the number of vertices in G sub i. The first claim is either n sub i upon p, which is the lower bound on the ith graph is equal to n sub i minus p, n sub i minus 1 upon p minus 1. So, either this lower bound decreases or L sub i is equal to L sub i minus 1 minus 1.

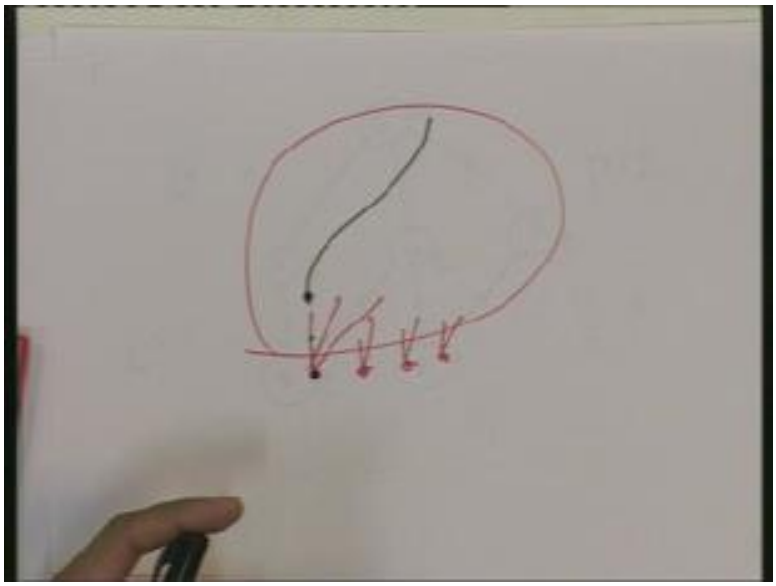
So, either this lower bound decreases or this lower bound decreases. So, after first iteration we have L sub 1 then we have L sub 2. So, L sub 2 will be either 1 less or this lower bound for the second iteration will be 1 less. And this will be enough to prove the 2 approximation. So, let us prove this.

(Refer Slide Time: 47:00)



The proof is actually quite simple. So, the basic step in the algorithm is to find p vertices in that step3 of iteration i. So, suppose it does find those p vertices in iteration i. So, what happens? So, if it finds p vertices, then the number of vertices that remain is going to be p less. So, n sub i is going to be equal to n sub i minus 1 upon p. But now, if you simply divide by p then we will get part a. So, this happens then part a will hold. The other case is suppose that the algorithm does not find p vertices. If the algorithm does not find p vertices, then there are at most p minus 1 ready vertices. So, what are the ready vertices, the ready vertices are the vertices in the graph.

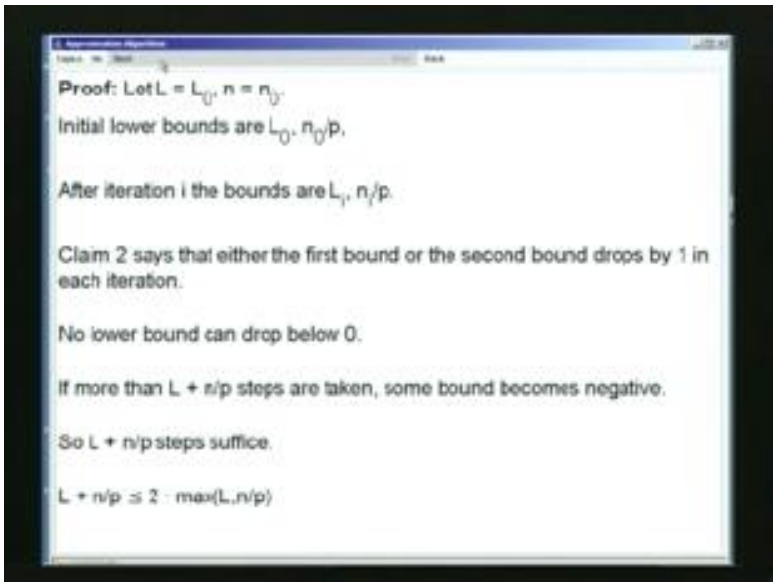
(Refer Slide Time: 48:06)



Such that their predecessors have already being scheduled or they do not have any predecessors whatsoever. What do we know about such vertices? Well, what do we know about paths. Here is the key idea every longest path ((Refer Time: 48:22)) must originate on one of these ready vertices. Suppose, it does not, suppose here is a longest path. Well, we go back this is not ready vertex. So, there must be vertex behind it. If there is a vertex behind it, then we are getting a path even longer.

Therefore, by contradiction the longest path must originate over here. So, ((Refer Time: 48:47)) the algorithm on the other hand schedules all these ready vertices. But, if it does schedule all these ready vertices, then the lengths of all the paths starting at these ready vertices, including the longest path must decrease by 1. But, that is essentially saying that L sub i equal to L sub i ((Refer Time: 49:04)) minus 1 minus 1. Thus ((Refer Time: 49:07)) we have proved this, either this holds or this holds. The next claim is that this algorithm gives a 2 approximation. So, here is a proof.

(Refer Slide Time: 49:17)



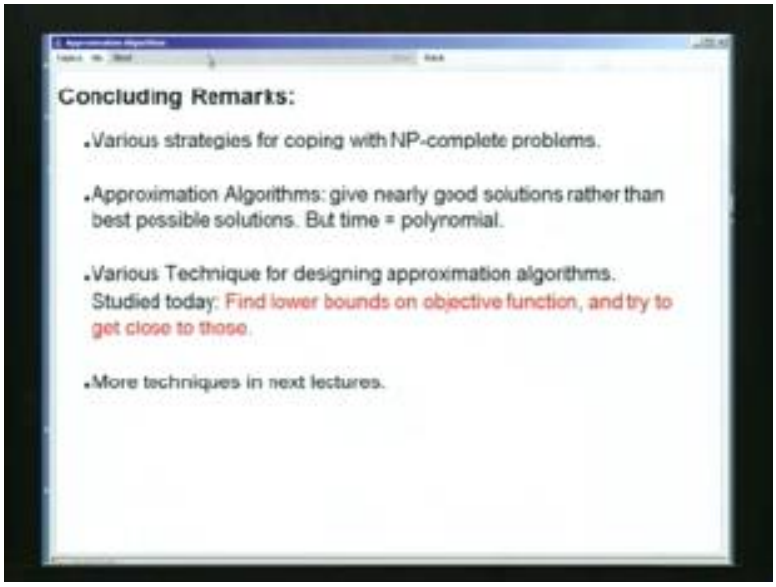
So, remember L was a lower bound the length of the longest path in the entire graph. So, I am going to call it L sub 0. n was the number of vertices in the entire graph I am going to call it n sub 0. The initial lower bounds thus are L sub 0 and n sub 0 upon p. After iteration i the bounds are L sub i and n sub i upon p, and what else to be known. Claim 2, which we just proved says that either the first bound or the second bound drops by 1 in each iteration.

So, starting from L 0 and this n 0 upon p, we go to L 1 and n 1 upon p. L 2 and n 2 upon p and so on. Claim 2 says that either the first one drops or the second one drops. Eventually, until we get to the last iteration. No bound can drop below 0 .Because it does not make sense to say that the length of the path is negative. Or that the number of vertices is negative. So, which means that if more than L plus n over p steps are taken. Then one of these bounds must become negative starting from here.

Because, this L 0 can only drop by L, this can only drop by n over p. So, one of these has to go below 0. But, that is not possible. And therefore, it means that L plus n over p steps must suffice. Our schedule must have length L plus n over p at most. But, what do we know about L plus n over p. Well, this is certainly less than 2 times max of L and n over p. So, we will just replace the smaller of the 2 with the max. So, this is going to be less than 2 times max of L and n over p.

But, what is max of L and n over p. So, this is a lower bound, this is a lower bound. So, the larger of the 2 is also lower bound. So, but if it is a lower bound, then OPT is even bigger than this. So, this is less than twice OPT. So, this max is a lower bound. So, OPT the length of the optimal schedule cannot be smaller than the max. And therefore, we have that L plus n over p is less than 2 times OPT. But, this is the length of the schedule, which we produced. And this length is less than 2 times OPT, so L done. So, we have proved that this algorithm gives a 2 approximation.

(Refer Slide Time: 51:58)



So, now I am going to conclude. So, today we discussed various strategies for coping with NP complete problems. The strategy which we are going to study is the strategy of devising approximation algorithms. So, these are defined as giving nearly good solutions rather than the best possible solutions. But, the good thing about them is that the time is polynomial. There are various techniques for designing approximation algorithms. And the techniques that we studied today can be summarized as follows.

So, basically we try to find lower bounds on the objective function, which we want to minimize. And then we try to get close to this. Of course, if the objective function had to be maximize and we will try to find upper bounds and we will try to get close to those. So, device we will algorithms in this case, which will get close to this lower bounds. So, the lower bounds are easily should be easily identifiable. And therefore, we can actually compute them.

And then, we can may be try to target an algorithm, which tries to meet them. But, of course, it will not succeed in meeting them. But, it will try to it will succeed in hopefully getting close to them. We will see more such techniques in the next lectures.

Thank you.