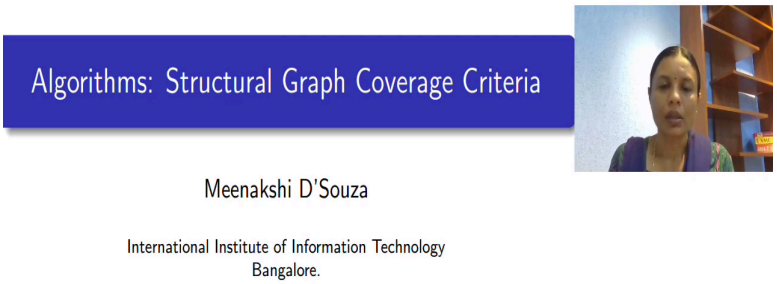
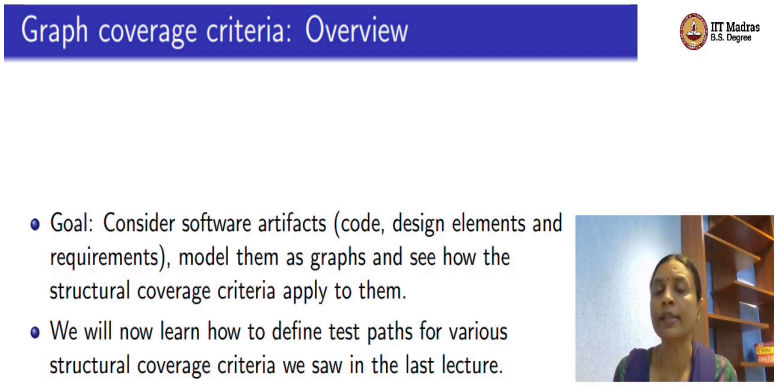
Software Testing
Professor Meenakshi D'Souza
Department of Computer Science and Engineering
Indian Institute of Technology, Bangalore
Week 2 Lecture 5
Algorithms_ Structural Graph Coverage Criteria

(Refer Slide Time: 00:14)



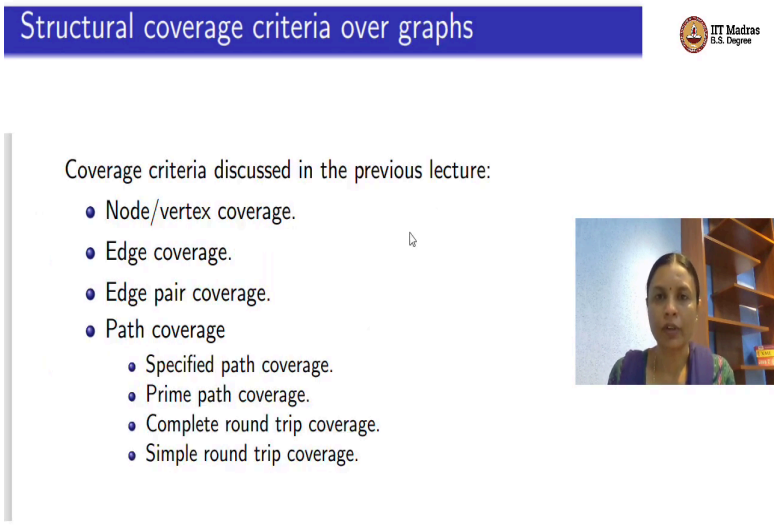
Welcome back. Last lecture of week 2. In the earlier lecture, we saw algorithms for structural graph coverage criteria. Now, what I am going to tell you is how to derive the test requirements if you have a graph from a software artifact. And if you would like to achieve structural coverage criteria, let us say node coverage, or covering loops, which is prime paths, how to actually get the test requirements? are there algorithms to do it, we learn the fundamentals of these algorithms based on the elementary graph algorithms that we saw. And towards the end, I will point you to a nice web application that you can use to automate it.

(Refer Slide Time: 01:01)



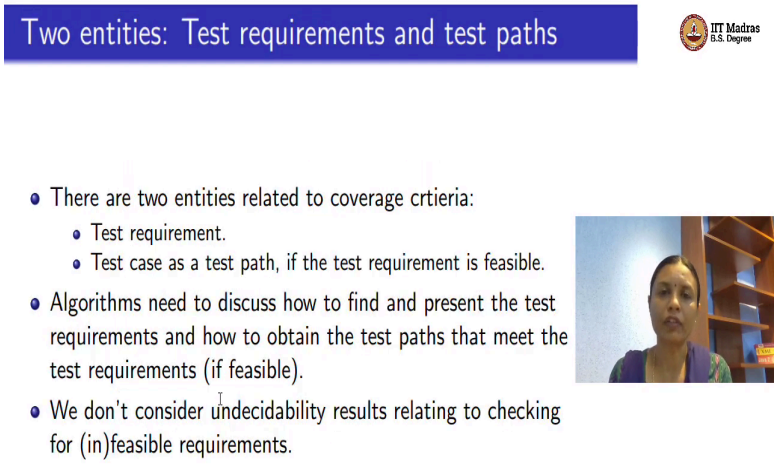
So, as I told you think of our broad goal as I have software artifacts with me, procedures, code, methods. And I also can have design elements, requirements, and I have generated graphs from them. And I would like to apply structure and coverage criteria on these generated graphs to test the software artifact for execution of every node, which corresponds to every statement, for execution of every edge met might correspond to branching, for execution of loops, which might correspond to prime paths, and so on. So, now we are going to learn how to get the test paths for these various test requirements and the case of prime paths how to get the test requirements itself.

(Refer Slide Time: 01:47)



So, these are all the structural graph coverage criteria that we saw in the previous lecture, node coverage, edge coverage, covering pairs of edges, covering paths, complete path coverage which we said is not useful if the graph has loops, specified path coverage, very specifically, prime path coverage for loops, and then round trips, simple round trip and complete round trip. So, for each of these coverage criteria, we will see how to write the test requirements, and how to get test paths that will achieve the test requirements.

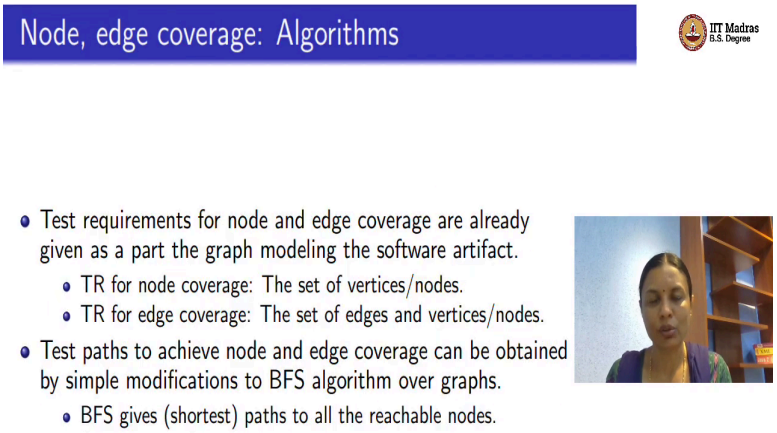
(Refer Slide Time: 02:22)



So, as I told you, per coverage criteria, we have two entities. One is the test requirements itself, which is easy, like for node coverage, you already have it as a part of the graph, and the other is the test path to achieve the test requirement. So, how are we going to get these two entities? So, we are going to see algorithms that will tell you how to find present the test requirements, and how to get the test paths for these test requirements, assuming that the requirement is feasible.

We do not really have procedures to check if the requirement is feasible or unfeasible, we will not really see that part, we leave it to the tester based on the application to see whether the requirement can be achieved or not. If it can be met, we say it is feasible, if it cannot be met, then we say it is infeasible.

(Refer Slide Time: 03:12)



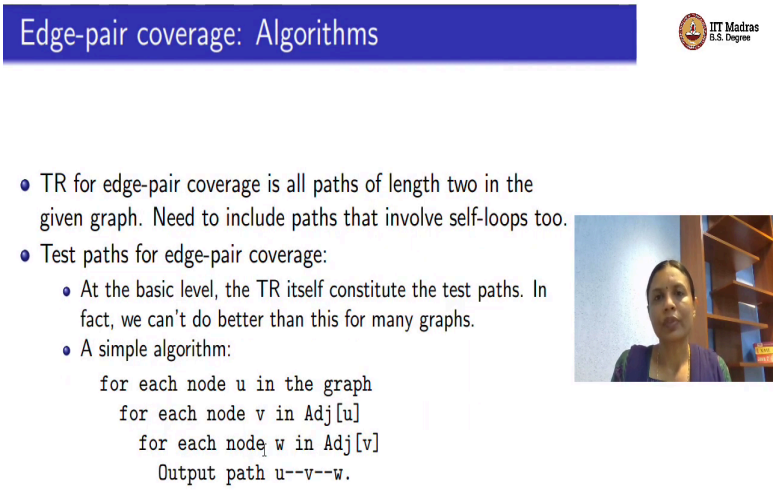
So, node and edge coverage were two simple coverage criteria that we saw, there is nothing much in terms of algorithms for writing the test requirements for them, they come as part of the graphs themselves. Test requirements for node coverage is given as the set of nodes along with the graph. Test requirement for edge coverage has given us the set of edges along with the graph itself.

So, the test paths to achieve node and edge coverage, how do I get them, we just saw BFS, breadth-first search. Remember what did breadth-first search do? It gives you a systematic procedure to visit every node that is reachable from one of the initial nodes. So, in graphs for testing, we already have marked initial nodes.

So, run breadth-first search from that initial node, all the nodes that are reachable from that node will give you test paths that achieve node coverage, you just have to ensure that the test path ends in a final vertex. Same thing for edge coverage also, there is no big deal. In fact, breadth-first search in addition gives you shortest paths, which is a useful property which you may or may not use in the context of testing.

So, it is fairly easy to write test requirements and test paths for both node and edge coverage. If you happen to have a node that is unreachable from an initial node, then that particular node coverage criteria becomes infeasible. So, you leave it at that. As long as all the nodes are reachable from the initial node, I can always use an algorithm like breadth first search to be able to get test paths for both node coverage and edge coverage.

(Refer Slide Time: 04:46)

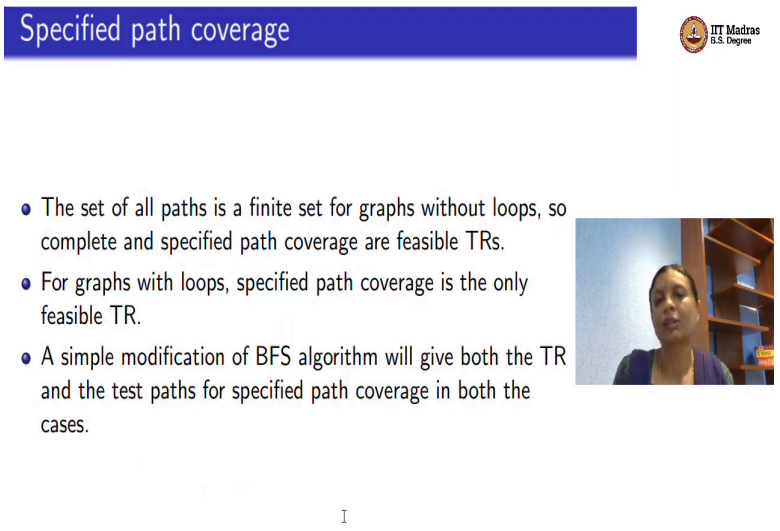


The next is edge pair coverage. This is we saw, pairs of edges. Test requirements for edge pair coverage is all paths of length 2 in the given graph, so we need to include paths that involve self-loops also, we saw in that example, comparing prime paths to edge pair coverage that self-loops come for edge pairs. So, it is again quite easy to enumerate test requirements for each pair coverage.

They could themselves be the test paths as long as they start in an initial node and end in a final node. So, I have just written the same thing as a simple algorithm. So, you take nodes in the graph, let us say take a node u in the graph, for each search node u in the graph, consider the adjacency list of u. Let us say there is a node v in the adjacency list u.

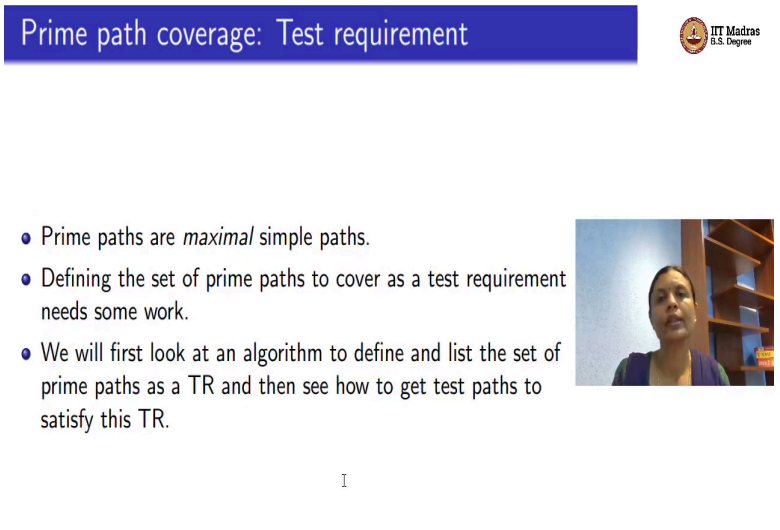
Now, you look at the adjacency list of v and look at the node w in the adjacency list of v and the edge pair coverage test requirement that you want is u to v to w pairs of edges, which is paths of length 2. If u and w happen to be initial and final vertices, you are done then and there, otherwise expand u to the left by augmenting it with a test path that starts in an initial vertex. And similarly, if w is not a final vertex this add some extra vertices to the end to get it as a final vertex. So, that was edge pair coverage.

(Refer Slide Time: 06:12)



Specified path coverage, some user tester gives you a set of paths if the graph is without loops, then specified path coverage can be easily achieved, because you can cover all the paths you can do complete path coverage. For graph and any algorithm depth-first search or breadth-first search will let you do. For graphs with loops we saw complete path coverage is infeasible, specified path coverage is the only feasible test requirement. Again, for the set of paths that I need, I can always modify breadth-first search algorithm to get specified path coverage.

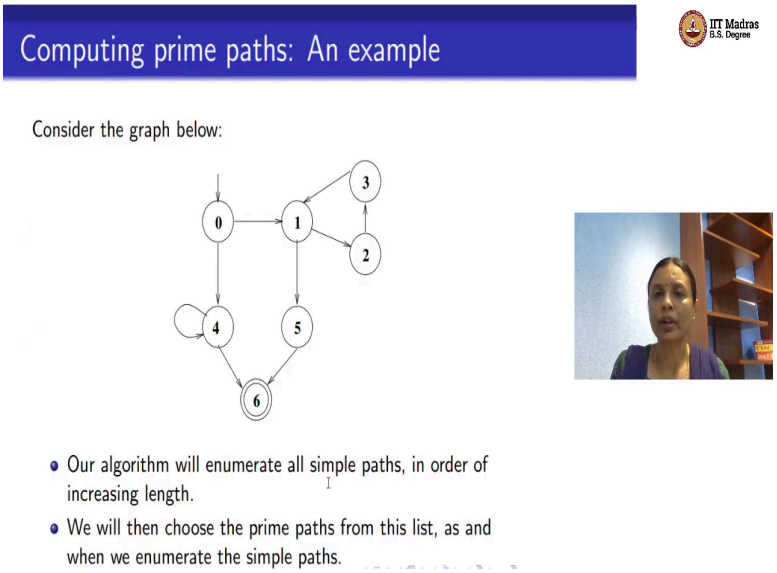
(Refer Slide Time: 06:46)



The rest of this lecture, let us spend it on its most interesting structure and coverage criteria, which is prime path coverage as the test requirement. So, what did we see, to recap, prime

paths are maximal simple paths. So, when I define prime paths coverage as my test requirement, I need to first get the prime path from the graph. So, how do I get all the prime paths from the graph, that is what we are going to see today.

(Refer Slide Time: 07:17)



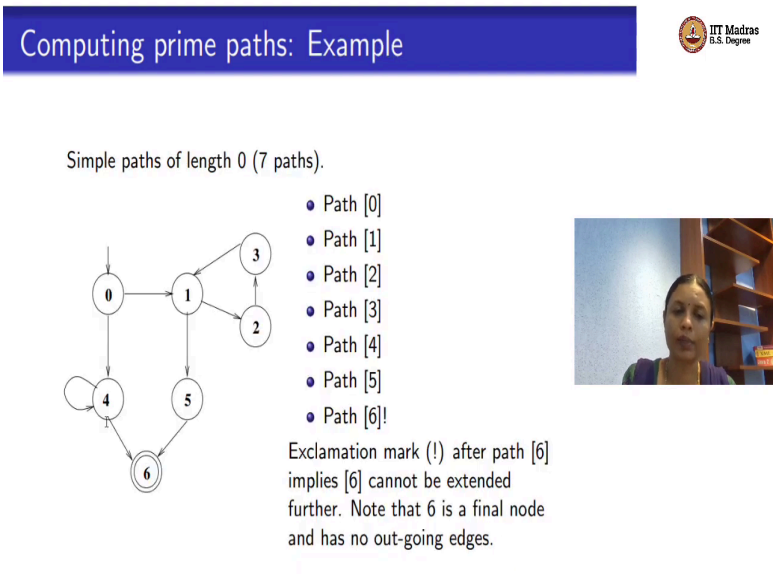
So, here is one algorithm, I will instead of giving the pseudocode of the algorithm, I am going to illustrate the algorithm on a running example. In fact, you all can take it up as a small exercise to see if you can write the pseudocode of this algorithm. So, consider the graph that is given here, it has seven nodes numbered from 0, 1, and so on up to 6, node 0 is the initial node, node 6 is the final node.

This graph, if you study it has tends to have two loops. There is one self-loop at node 4. And then there is one loop here from 1 to 2, 2 to 3, 3 back to 1. So, assume that this is a graph that you have derived from some software artifact, graph corresponding to a method or something like that. Computation starts at node 0, initial node, computation ends at node 6, final node. And along the way, there are two loops, one self loop at 4 and one more at 1, 2, and 3. Is it clear?

So, now for this graph, we are going to see my test requirement is, look for all prime paths, enumerate all the prime paths, achieve prime path coverage. So, how are we going to enumerate all the prime paths, that is the algorithm that we are going to see. So, our algorithm will is a brute force algorithm not efficient, but it works for all practical purposes. And it is a terminating algorithm. That is all we need, we are not bothered about efficiency and all that in software testing.

I will point you to a reference where you can get very efficient algorithms for enumerating prime paths. For now, we will just see a simple algorithm to enumerate prime paths is test required. So, our algorithm is just going to do this, it is going to enumerate all simple paths in increasing length in order of increasing length. And then amongst the simple paths, it is going to choose prime paths from them.

(Refer Slide Time: 09:04)



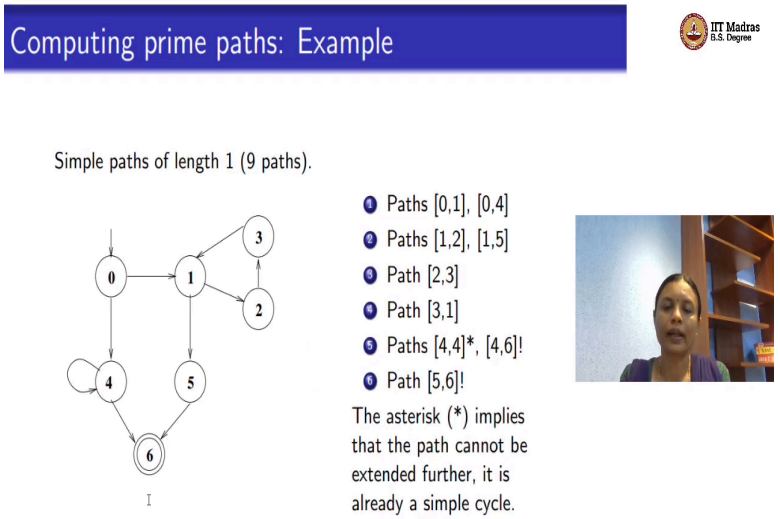
So, I have taken the same example, put the same example that you saw in the earlier slide here on the left. And remember what the algorithm is, keep it in your mind, enumerate all simple paths in order of increasing length, and pick and choose prime paths from them. That is all I am going to do. So, this is the first step of the algorithm, where I am enumerating simple paths of length 0, I am enumerating simple paths of length 0.

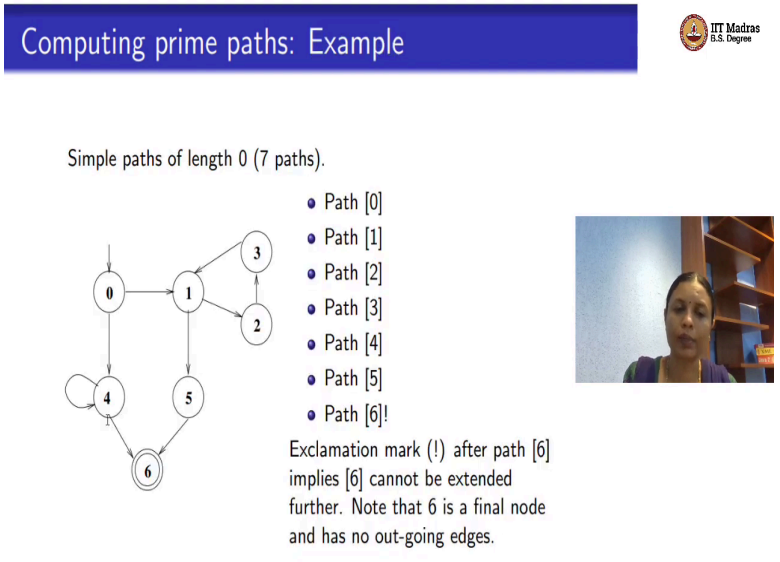
What are simple paths of length 0? They are nothing but vertices or nodes in the graph. So, path, I have just written it like this to emphasize that I am enumerating paths of length 0, so that is path corresponding to the node 0, path corresponding to the node 1 and so on, path corresponding to the node 6. So, there are seven simple paths of length 0, which I have put aside and the right-hand side for the graph on the left-hand side.

Out of these seven simple paths that I have listed in the last simple path, I have put an exclamation mark, if you notice here. So, why that exclamation mark is present? The exclamation mark after path 6 implies that path 6 cannot be extended further. That is also obvious, because if you see there are no outgoing edges from this final vertex 6.

So, I make a note of that by putting an exclamation mark there that this, I have enumerated all paths of length 0, which are basically vertices. And amongst all paths of length 0 that I have enumerated, I mark, especially the parts that cannot be extended further, let us say with an exclamation mark. Now, my next step is enumerate all paths of length 1. So, how many paths of length 1 are there? There are basically the number of edges.

(Refer Slide Time: 10:56)





So, I enumerate all paths of length 1, and I have grouped them together in this list. So, the first bullet if you see here, path 0 1, and 0 4, are two paths of length 1 or two edges that originate from node 0, 0 to 1, 0 to 4. I have taken the vertex 1, and enumerated paths that come out of 1, 1 to 2, 1 to 5. So, there are two paths of length 1 or two edges, that correspond

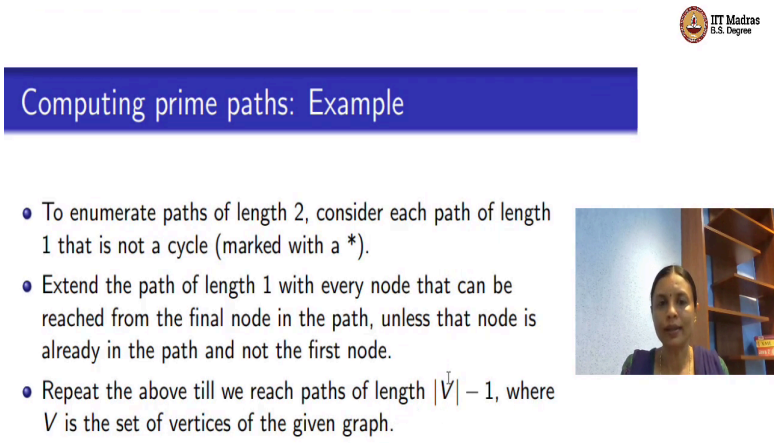
to paths of length 1 coming out of the vertex 1, then paths of length 1 coming out of the vertex 2, which is this path 2 3, the third item in the list.

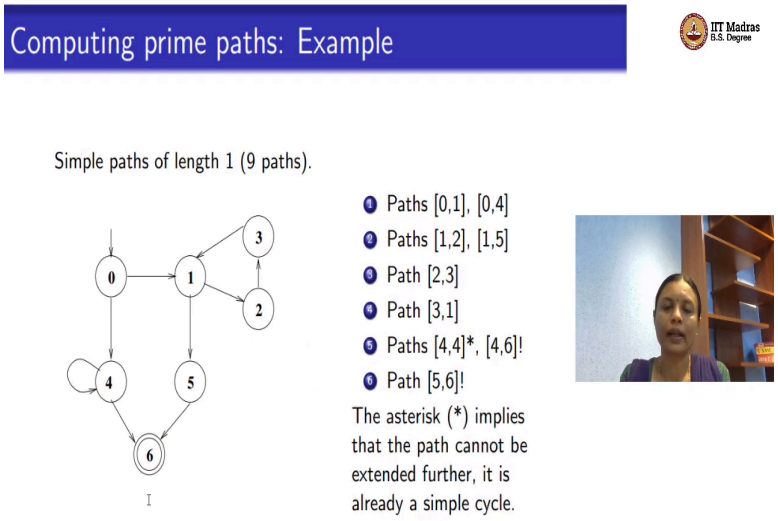
Similarly, path of length 1 coming out of the vertex 3, the fourth item in the list, paths of length 1 coming out of the vertex 4, 4 4 the self-loop, and 4 to 6 the edge. And similarly, paths of length 5, coming out, I mean paths of length 1 coming out of the vertex 5, that is 5 6. And in the earlier slide, I had 0 1 2 3, I had said path of length 0 starting from 6 cannot be extended, I had already put an exclamation mark.

So, in this paths of length 1, path starting from 6 do not exist. So, this is all paths of length 1, first I did paths of length 0, now I am doing paths of length 1. And now again, I am doing my marking. So, in this last path here, path ending at 6, that is path 5 to 6. And similarly, this path here, path 4 to 6, I have put exclamation marks, for the same reason, they are paths that end at node 6, and they cannot be extend further.

I have also put a star or an asterix in the path 4 4, this is different from an exclamation mark. Here, structurally in the graph, that path can be extended, I can go 4 4 6. But I do not want to extend it. Because remember what my goal is, my goal is to get prime paths, prime paths are maximal simple paths, this path is already maximally simple, 4 to 4, it is already a simple cycle. So, because of that, if I extend it, it will cease to become a prime path. My goal is to get prime paths. So, I said this path is itself a prime path. And I leave it at that. Is that clear? Now, the next step of the algorithm, I enumerate paths of length 2.

(Refer Slide Time: 13:30)



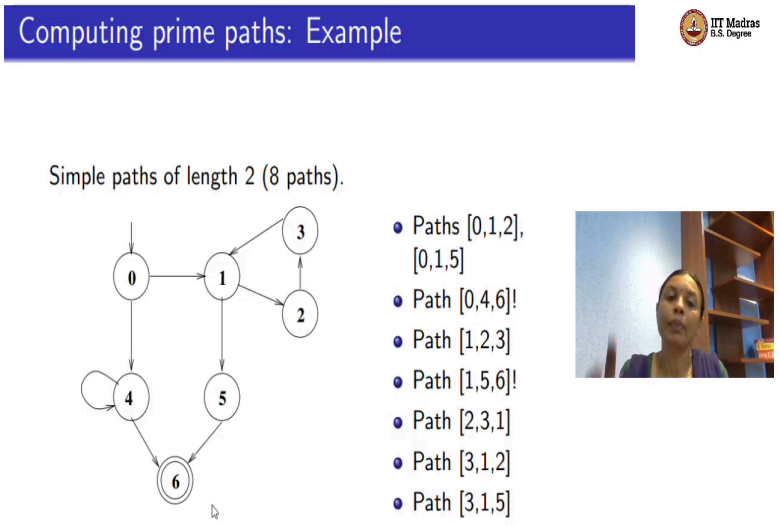


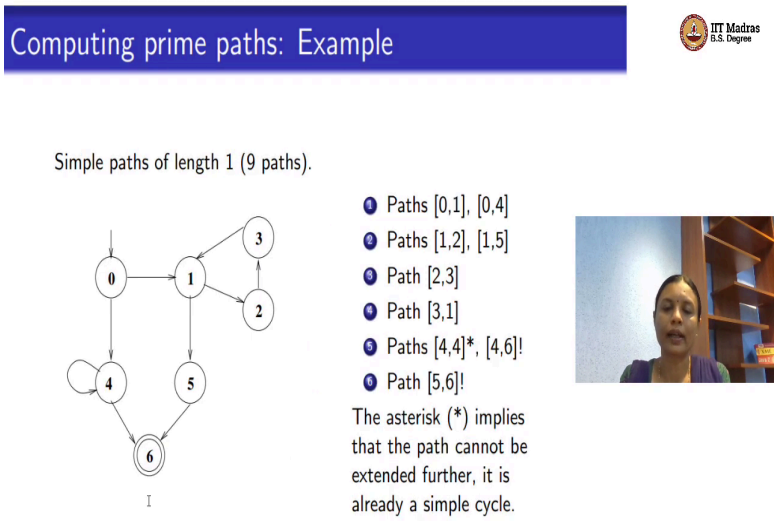
So, I consider each path of length 1, that is not a cycle, I have already marked cycles like this 4 4 with a star, I am not going to consider them the remaining paths of length 1, which are all these first one 0 1, 0 4, 1 2, 1 5, 2 3, 3 1, and so on, all these paths of length 2, I am going to take, and each part of this length, 1, I am going to extend it such that it final node can be reached from that path.

And I repeat this process by extending paths. So, at any point in time, I have paths of length k, I take some of them I mark out as the stars or exclamation marks. So, what is the convention? Convention for exclamation mark is this is it, I cannot extend the path anymore. Convention for marking a path of length k with a star is that the path is already a simple cycle.

The rest of the paths I tried to extend it and mark them once again and this algorithm repeats, for how long this algorithm repeats, till we reach paths of length |V| - 1, where |V| is the number of vertices in the graph, because remember, as I told you, in the last lecture, we are looking at simple paths, paths that do not contain internal loops. Maximum length of such a simple path can only be at most the number of vertices minus 1, so this algorithm is guaranteed to stop. So, let us go on.

(Refer Slide Time: 14:57)

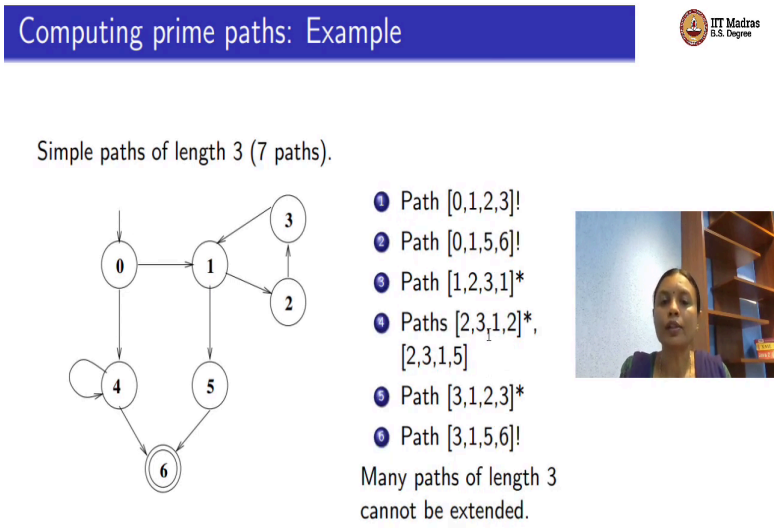


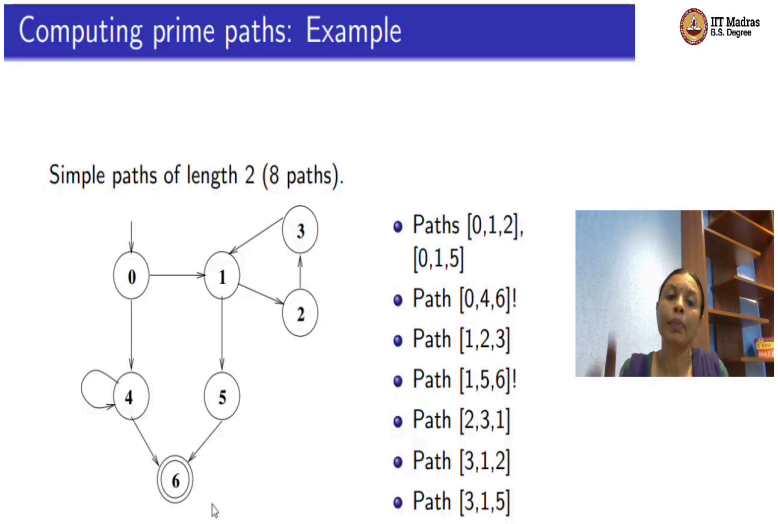


I had paths of length 1 in the earlier slide. I am using those parts and writing out paths of length 2, so 0 1 2 is a path of length 2, and 0 1 5 is a path of length 2, they both were obtained by extending the path of length 1, which was the edge 0 1. Similarly, 0 4 6 is a path of length 2, and so on, 1 2 3, 1 5 6, 2 3 1, 3 1 2, 3 1 5.

All these were obtained by extending these parts in this slide that were not marked with a star or exclamation mark. So, those are these bots. Now, I again, do the exercise of is there a path that cannot be extended further, put an exclamation mark, keep it aside, is there a path that is already a simple cycle, put a star keep it aside by marking it as a prime path. So, here, it is so happens that to have paths of length 2 cannot be extended further, that is 0 4 6, and 1 5 6. So, I keep them aside, I do not need them.

(Refer Slide Time: 16:03)



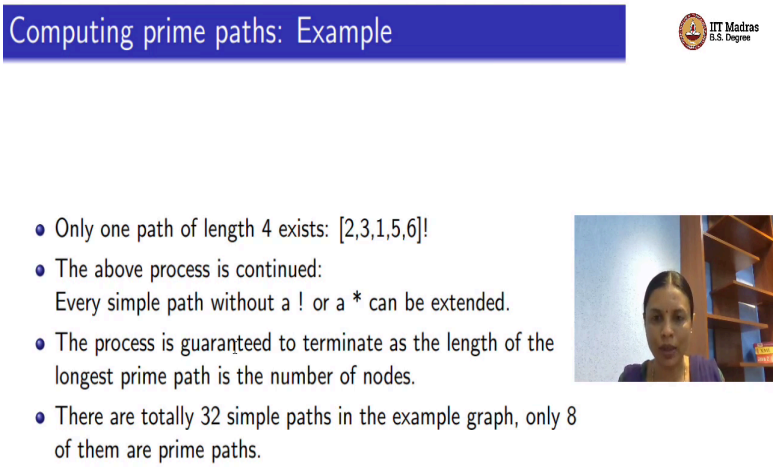


So, now simple paths of length 3, I repeat it. So, these are all the simple paths of length 3, 0 1 2 3, 0 1 5 6, 1 2 3 1, 2 3 1 2, 2 3 1 5, 3 1 2 3, 3 1 5 6, I have just extended the parts that come here that were not marked with an exclamation mark to make paths of length 3. Again, I do the exercise of the paths that further cannot be extended mark them with an exclamation mark, the parts that are already simple cycle mark them with a star. So, if you see quite a few of marked with an exclamation mark, paths that ended 6, 0 1 5 6 and 3 1 5 6, they both are marked with an exclamation mark.

And paths that are simple cycles, like 2 3 1 2, 1 2 3 1, 3 1 2 3, they are already prime paths. So, I keep them aside. In fact, in this list of paths of length 3, I just have one path here that I can extend 2 3 1 5. Sorry, I just have one path here that I can extend 2 3 1 5, that is it. So, I

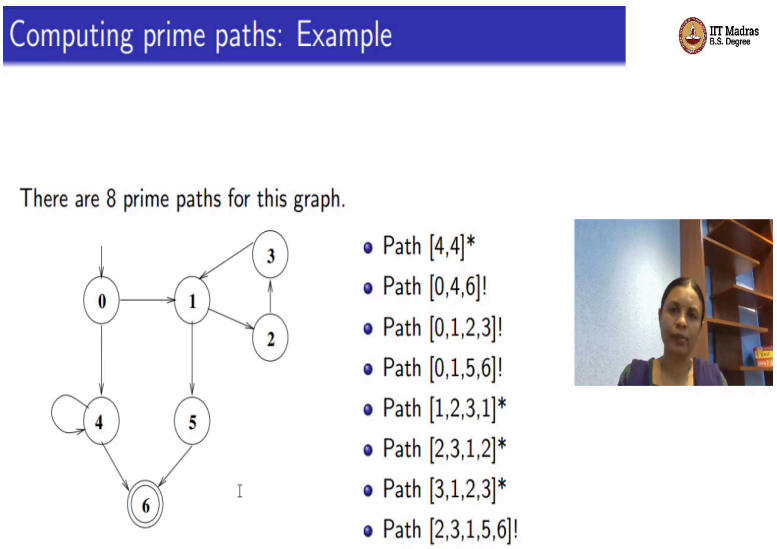
take that one path and I extend it, what can I extend it to 2 3 1 5 6. Let us go back and look at the graph 2 3 1 5 and 6, that is it, it ends at 6. So, this cannot be extended.

(Refer Slide Time: 17:24)



And you can continue this till every path till you reach paths of length |V| - 1. And that is it, we cannot do any more and keep marking it with exclamation marks and stars. And this algorithm is guaranteed to terminate because I told you the length of the longest prime path is basically the number of nodes minus 1. So, for this example, it so happens that there are 32 simple paths out of which there are eight prime paths.

(Refer Slide Time: 17:53)



Obviously, I am not listing the 32 simple paths. But here are the eight prime paths. So, the ones that we had pulled out as stars, and some of the ones that we pulled out with exclamation marks, they will turn out to be the prime paths. For our convenience, this graph on the left, which was our running example, here are the prime paths that are given on the right, totally eight of them.

So, path 4 4 corresponds to visiting this simple loop, self-loop once 0 4 6. The second path in this list, by now you can easily guess what does it mean, it is a path that corresponds to skipping the loop at 4, path 0 1 5 6, the fourth path here 0 1 5 6, this path, sorry, this path corresponds to skipping this loop 1 2 3 loop, 0 to 1, do not enter the loop, 0 to 5, 5 to 6. So, it corresponds to skipping the loop.

Paths 0 1 2 3, the third one in this list, lets you enter the loop 0 to 1, 1 to 2, 2 to 3. And as many times as you want, you can visit this loop. Which are the paths that will help you to do? These three prime paths 1 2 3 1, 2 3 1 2 and 3 1 2 3 will help you go round and round the loop as many times as you want, you can use any of them. And finally, when you want to exit the loop, use this path 2 3 1 5 6, this one 2 3 1 5 6, the last one.

So, what we have done just to summarize, our goal was we had a graph like this that we got from a software artifact that graph had some loops. My test requirement was to get the prime paths out of the graph because I would like to skip every loop that is present and execute every loop that is present.

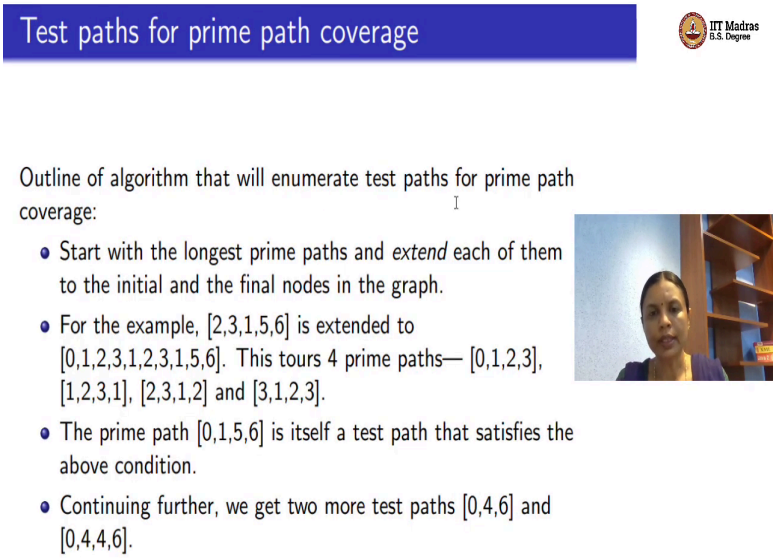
So, the algorithm to enumerate prime paths that we saw, will start and numerating simple paths starting from length 0, length 1, length 2, and so on, and pick and choose prime paths as it enumerates the simple paths. So, for this algorithm, it so happens that there are, I mean for this example, so happens that there are 32 simple paths out of which eight of them happen to be prime paths that are listed in this slide.

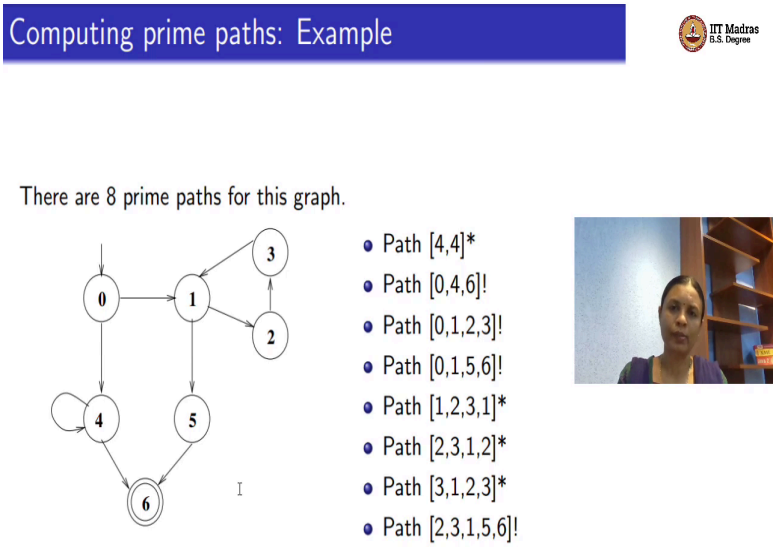
Once you have prime paths, magically, it so happens that how many ever loops are there in the graph, there could be nested loops, they could be separate loops, as you see in this example, every loop will be skipped and covered once or more than once. That is the correctness about the prime paths example, which I am not going to do the correctness. So, this gives me what, this gives me all the prime paths in the graph.

So, this gives me the test requirement for prime path coverage. Next comes test path. So, once I have the test requirement, if you see these prime paths, some of them if they already

begin in an initial node and end in the final node, then that itself becomes a test path. Like for example, this one, path 0 1 5 6 already begins in an initial node and ends in a final node, so there is no problem, this by itself is a test path, you are done. Otherwise, what we do is the following.

(Refer Slide Time: 21:18)





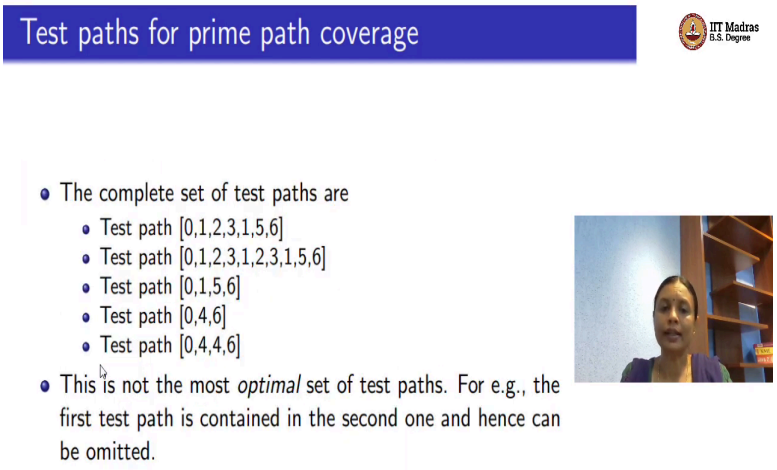
So, here is an outline of the algorithm that will enumerate test paths for prime path coverage, you start with the longest prime path, and basically extend each such prime path to the initial vertex on the left and to the final vertex on the right. For example, you start with the prime path 2 3 1 5 6 for this example, it was this last prime path that you saw here. Why do we start with the longest one?

Because this will subsume some other, some parts will come as sub paths of these. So, 2 3 1 5 6 is what I start with. And I can extend it to by appending with 0 and 1 because of the structure of this graph on the left. And then I have to do one more iteration just to complete it. So, this will tour 4 prime paths 0 1 2 3, 1 2 3 1, 2 3 1 2, 3 1 2 3, basically, the path that goes round and round the loop, you can use them up to your convenience to dictate the number of iterations in the loop.

This test path, for example, achieves two iterations of that loop. As I told you earlier, the prime path 0 1 5 6 is itself a test path that satisfies the condition. And for the other loop 0 4 the self-loop at 4, we have 0 4 6, which is itself a test path, and then I can do 0 4 4 6, which would cover that path.

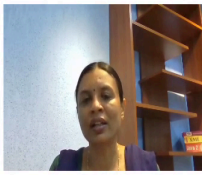
So, if you see between the two test paths 0 4 6 0 4 4 6 for this loop, and 0 1 2 3 1 2 3 1 5 6 for these three prime paths, and 0 1 5 6 for the prime path that skips the loop, I am done with prime path coverage. So, this is a rough outline of the algorithm. Basically, what it says is start with the longest prime path, extend it to the left, extend it to the right. Use any search traversal algorithm to extend to the left and extend to the right, that will give you the test path.

(Refer Slide Time: 23:18)



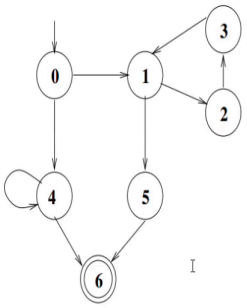
Test requirements and their test paths

- Given a TR for structural graph coverage, the problem of obtaining test paths for satisfying the TR mainly uses algorithms that extend BFS/DFS.
- They may or may not yield an *optimal* set of test paths that satisfy the TR.
- There are several notions of what an optimal set of test paths for a given TR_I is.
- Many algorithms that come up with optimal test paths for a given TR are NP-complete.

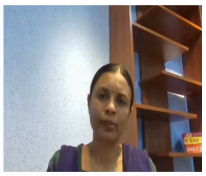


Computing prime paths: Example

There are 8 prime paths for this graph.



- Path [4,4]*
- Path [0,4,6]!
- Path [0,1,2,3]!
- Path [0,1,5,6]!
- Path [1,2,3,1]*
- Path [2,3,1,2]*
- Path [3,1,2,3]*
- Path [2,3,1,5,6]!



So, for that example, we had five test paths for prime path coverage, if you see the first test path visits the loop exactly once, second test path visits the loop twice, you could repeat it by making the loop, this segment 2 3 1 2, 1 2 3 1, you can repeat any of them to make the loop visit more than once. The third test path skips that 1 2 3 loop.

This fourth and fifth test paths cater to the other self-loop at 4, 0 4 6 skips the self-loop, 0 4 4 6 visits the loop any number of times here once but you could repeat it to visit any number of times. So, these are test paths, they need not be prime paths. Test requirements have to be prime paths remember that, that difference should be clear.

So, these are prime paths. They have maximal simple paths. Test paths meets the test requirements of prime paths, they always begin an end with, begin at an initial vertex end at a

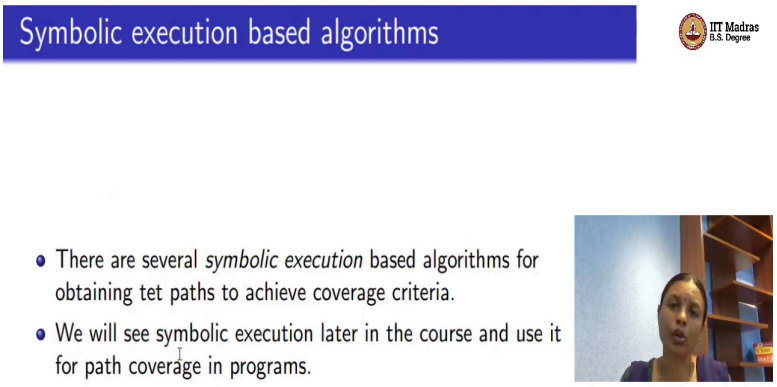
final vertex, they can have repetitions because they are just test paths that satisfy prime paths. Now, you can ask for this example. We had so many prime paths. How many?

We had eight of them eight prime paths, and we came up with five test paths in this for this example, that will achieve the test requirement of satisfying the eight prime paths. Can we come up with a shorter set of test paths? Is this the most optimal in terms of the number of test paths?

You could ask questions like this because graphs sometimes can be very large the ones that correspond to is real software, and there could be several loops in the code, there could be several different prime paths. So, things can get very large, you do not want too many test cases as test paths. So, can I come up with an optimal number, either in terms of the number of test paths or in terms of shorter test paths.

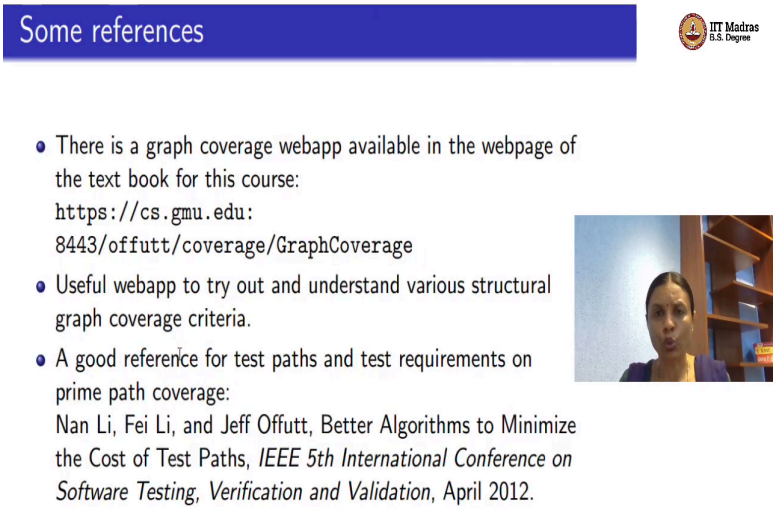
The bad news is that such a problem is in general intractable, while we know of some algorithms that might compute optimality. We do not know of good polynomial time algorithms that will compute optimality. So, we usually do not try to get optimal test paths.

(Refer Slide Time: 25:39)



There are some very, there is an important technique and testing called symbolic execution, which you can use to get better test paths, we will do symbolic execution later in the course.

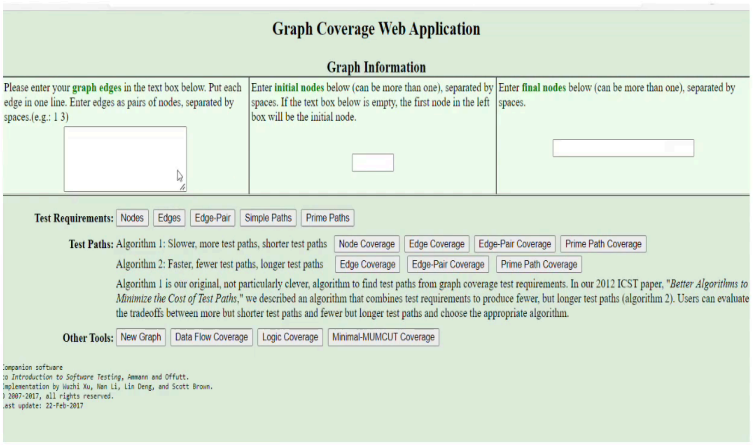
(Refer Slide Time: 25:53)



So, as I told you, suppose you want to do it on your own, with the help of instructor, in the next week, I will teach you how to generate graphs from methods. And once you have the graph, suppose you want to get prime paths as test requirements and want to get test paths that achieve prime paths as test requirements. Obviously, you could deploy the algorithm that was taught in this lecture.

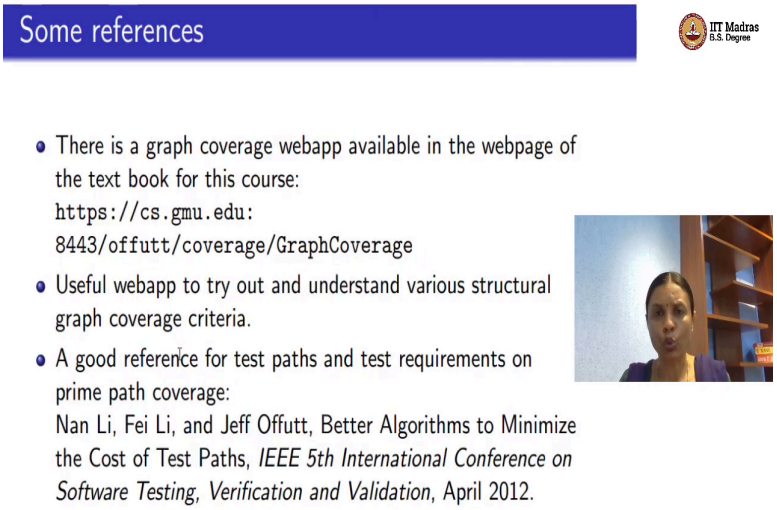
But if you want a shortcut, go to this page. This is the course page of the textbook by George Mason University, they have this app web app where you can enter the graph through the interface they have. And that will give you the graph coverage criteria. So, let me just show it to you once.

(Refer Slide Time: 26:33)



So, this is how the web app looks like. So, you can enter the graph edges here, you can enter the initial node here, you can enter the final node here. And then this is a list of test requirements, do you want node coverage, edge coverage, edge pair coverage, covering simple paths, covering prime paths, they give 2, 3 different algorithms. We saw algorithm one, which is the slower one, they have a faster algorithm. And then you can get the test paths for these. So, this is a very useful web application, try and use it.

(Refer Slide Time: 27:08)



So, if you want to learn more, this is the paper that you can refer too for getting test paths and test requirements for prime path coverage. So, I will stop here. In the next lecture, we will take examples of code and derive graphs out of them and apply structural graph coverage criteria that we saw in these two lectures. Thank you.