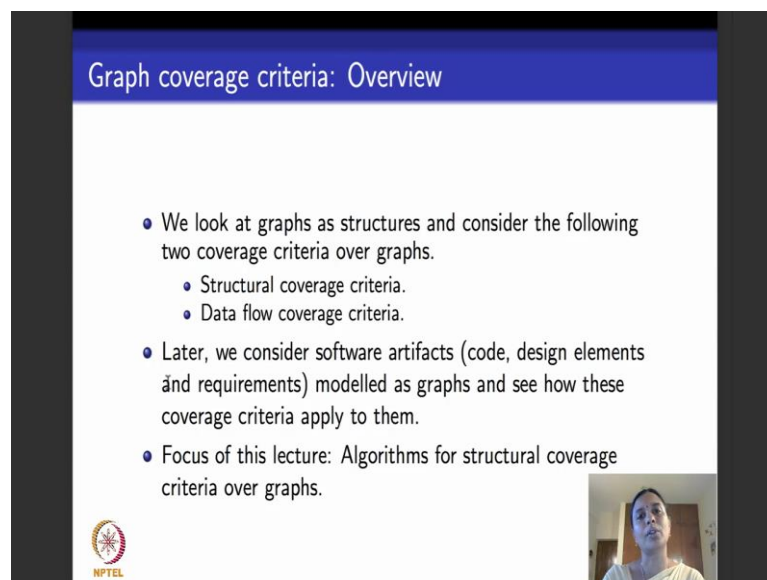


Software Testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture – 09
Algorithms: Structural Graph Coverage Criteria

Hello everyone. In this module our main focus would be to get back to structural coverage criteria on graphs that we saw two modules ago, and look at algorithms that will help us to write down the test requirements for each of the coverage criteria we saw. And also look at algorithms that will help us to generate test paths that will meet the test requirements for these algorithms.

(Refer Slide Time: 00:37)



The slide is titled "Graph coverage criteria: Overview" and contains the following bulleted text:

- We look at graphs as structures and consider the following two coverage criteria over graphs.
 - Structural coverage criteria.
 - Data flow coverage criteria.
- Later, we consider software artifacts (code, design elements and requirements) modelled as graphs and see how these coverage criteria apply to them.
- Focus of this lecture: Algorithms for structural coverage criteria over graphs.

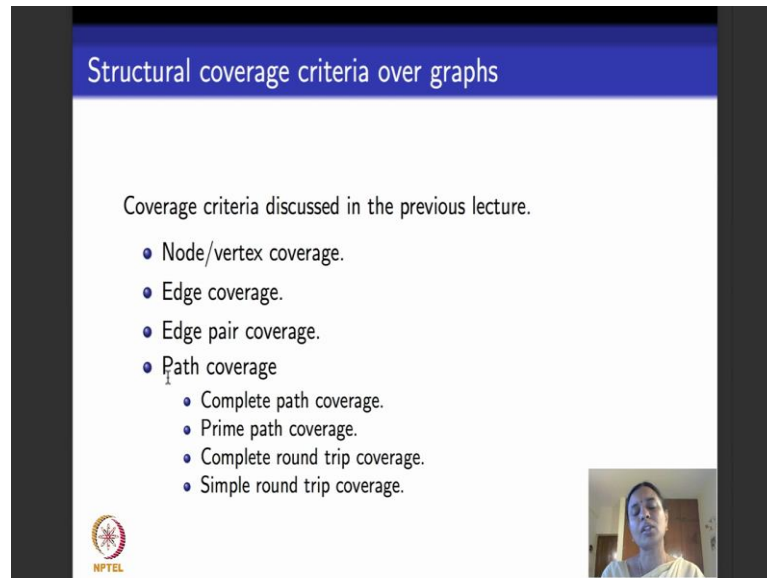
In the bottom right corner of the slide, there is a small video inset showing a woman, presumably Prof. Meenakshi D'Souza, speaking. In the bottom left corner, there is a logo for IITEL.

So, just to recap the overall module that we are looking at currently; we are now looking at designing test cases where software artifacts are modeled as graphs. So, in the graph models we consider two kinds of coverage criteria: coverage criteria based on the structures of the graph which we saw two modules ago. And in the next week we will look at coverage criteria on graphs augmented with variables, talking about variables and seeing data flow coverage criteria.

Moving on from there we will see how various software artifacts can be modeled as these graphs, and how these coverage criteria can be used for designing test cases. Now

what will be the focus of this lecture? We look at algorithms for structural coverage criteria in this lecture.

(Refer Slide Time: 01:24)



The slide is titled "Structural coverage criteria over graphs" and lists the following coverage criteria discussed in the previous lecture:

- Node/vertex coverage.
- Edge coverage.
- Edge pair coverage.
- Path coverage
 - Complete path coverage.
 - Prime path coverage.
 - Complete round trip coverage.
 - Simple round trip coverage.

The slide also features the NPTEL logo in the bottom left corner and a small video inset of the presenter in the bottom right corner.

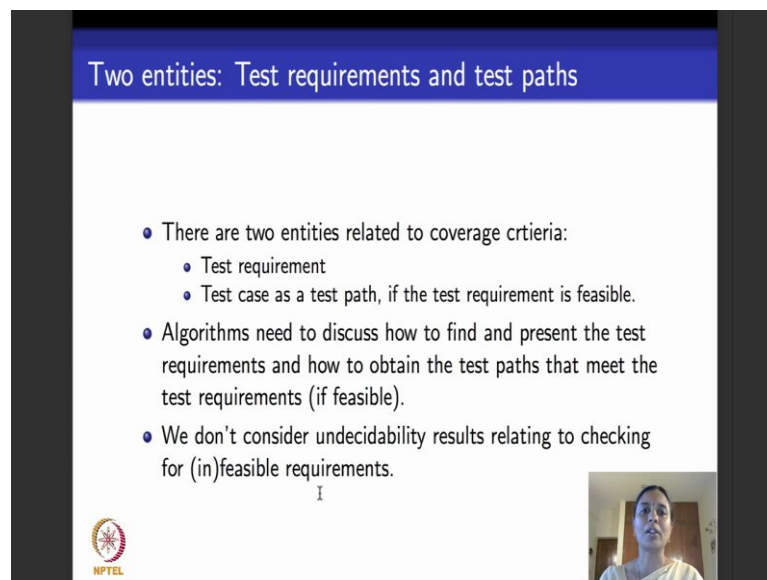
So, to recap what was the structural coverage criteria over graphs that we saw what two lectures ago. If you remember we saw all these coverage criteria we began with node or vertex coverage, then we moved on to edge coverage, we looked at edge-pair coverage, and then we looked at path coverage graphs. We began with complete path coverage which I told you was infeasible if a graph has a single loop or a self loop, because you can go round and round the loop several times and get infinite number of paths. So, this no notion of the finite path set that I can completely cover.

So, a work around would be to do specified path coverage. Specified path coverage is very user gifts or the paths. Another interesting path coverage criteria that has existed in the testing literature is that are prime paths coverage. These specially help to cover graphs with loops by helping us to execute the loop once, execute the loop many times, and also to skip the loop. And then prime paths that begin and end with the same node what are called round trip coverage.

So, we ended the structural coverage criteria lecture by looking at two round trip coverage criteria. One was complete round trip coverage, which insisted that you cover all the round trips. The next one was simple round trip coverage which insisted that you cover one of the round trips.

So, what we will be looking at now is fine we have these coverage criteria each coverage criteria has its set of test requirements as per the criteria that is under consideration, and then I go as a tester is to be able to define test paths for those coverage criteria. How does one go about doing it? So, what are the algorithms that will help us to do this?

(Refer Slide Time: 03:14)



Two entities: Test requirements and test paths

- There are two entities related to coverage criteria:
 - Test requirement
 - Test case as a test path, if the test requirement is feasible.
- Algorithms need to discuss how to find and present the test requirements and how to obtain the test paths that meet the test requirements (if feasible).
- We don't consider undecidability results relating to checking for (in)feasible requirements.

NPTEL

i

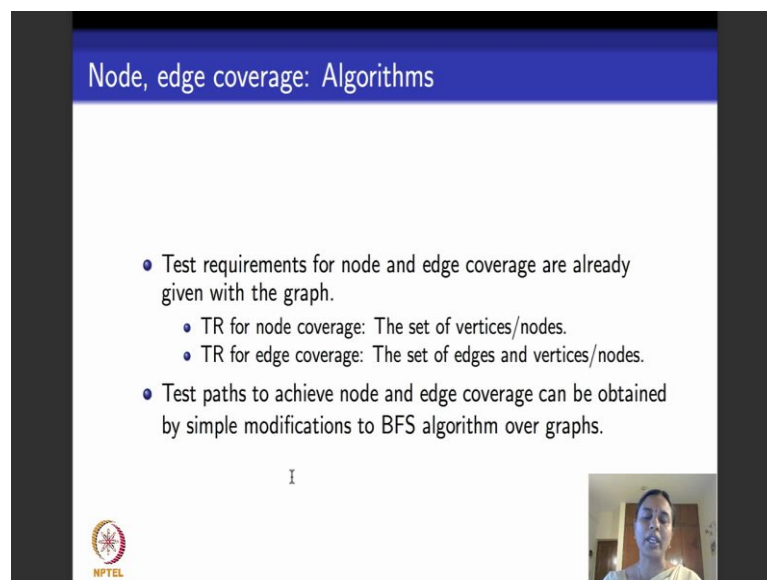
So, we will saw DFS and BFS, now we will see how to use these algorithms to be able to define algorithms for these coverage criteria. When I talk about coverage criteria that I discussed in this slide, for each of those coverage criteria we basically saw two entities. So, one entity was what is called a test requirement. So, when I say you achieve edge-pair coverage the test requirement is achieving edge-pair coverage. Corresponding to this test requirement I will look at the graph and generate a set of test paths in the graph that will achieve this test requirement for edge-pair coverage.

So, when I talk about coverage criteria, I am talking about two entities that are related to the coverage criteria: one is the criteria specified as a test requirement of deviated as TR, and the other is the set of test paths which occur a test case that are designed to satisfy the criteria. Now it could very well be the case the criteria that I define is infeasible. Like I told you right if there is a graph that has a loop and my criteria say do complete path coverage it is directly infeasible, because there are an infinite number of paths. So, only for feasible coverage criteria as test requirements do we talk about designing test paths as test cases for them. For infeasible coverage criteria we do not even consider test paths.

So, another interesting problem if you step back and ask you could ask yourself- am I considering the problem of deciding whether a given coverage criteria is feasible or not? Yes this is a very important problem, but that will not be the focus of this course, because as I told you it is an undecidable problem to check whether several different coverage criteria are feasible or not. And because this course is intended to be an application oriented course I really do not want to introduce how you get those undecidability results and what are the reduction tools and all that.

So, what we will focus is; we will assume that the test criteria that we have working with are feasible and then we will see how to right test requirements for them, and how to design test cases as test paths for them.

(Refer Slide Time: 05:39)



The slide is titled "Node, edge coverage: Algorithms" in a blue header. It contains the following text:

- Test requirements for node and edge coverage are already given with the graph.
 - TR for node coverage: The set of vertices/nodes.
 - TR for edge coverage: The set of edges and vertices/nodes.
- Test paths to achieve node and edge coverage can be obtained by simple modifications to BFS algorithm over graphs.

At the bottom left is the NPTEL logo, and at the bottom right is a small video inset of a man speaking.

We will begin with this listing one at a time we look at. So, we begin with node and vertex coverage and then we will move on rounds list. So, we begin with node and edge coverage. As I told you for each of this coverage criteria two things to look at: what is the test requirement or TR and what is the test case that will satisfy those TRs. What are test cases? Test cases are test paths in the graph that will satisfy the test requirement.

Just to recap a test path always has to begin at a designated initial node and end in one of the designated final nodes that is one of the requirements that we impose is the part of the definition of test paths in this course. So, for node and edge coverage the test requirements are already given along with the graph. What is the test requirement for

node coverage which is basically the set of all nodes of the graphs, the set of all vertices of the graph? What is the test requirement for edge coverage which is basically the set up of edges of the graph? There is nothing much to do.

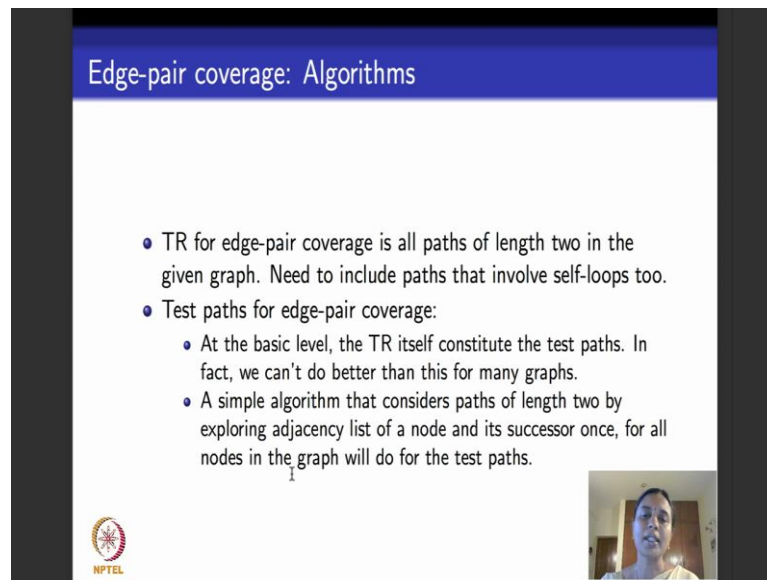
Now, these are the TRs, they are already given to you, we do not need any algorithms to list them or define them specifically. Now what about test cases or test paths that will satisfy node coverage or edge coverage. If you see for node coverage very simple application of breadth first search that we saw about a lecture one lecture ago will directly give us node coverage or even depth first search. You take a graph fix a (Refer Time: 06:58) node and do BFS or DFS on that node it will result in a breadth first tree or a depth first tree of the graph.

Suppose this tree spans out all the graphs; they are it is the paths of the tree will give you the test path is test cases for node coverage. Suppose such the first breadth first search of the first DFS does not finish exploring all the vertices of the graph; there are vertices that were not reachable from this designated source. Then you fix a new source and start DFS or BFS again from that source and you get another tree. So, this way you run DFS or BFS on the complete graph till you get forest of trees and the resulting forest of trees will give you the set off test paths that will achieve node coverage.

We can do a very similar thing for edge coverage. I can do a simple modification of the BFS algorithm wherein by after running BFS algorithm I checked if I have indeed covered all the edges of the graphs. And if I have not covered all the edges of the graph I consider the remaining edges which could be cross edges tree edges or back edges and see how I can include them in the test paths to be able to obtain test paths that acts as test cases for node and edge coverage.

Just one additional point here it would be useful to begin your search from the designated initial node in the graph, because that is where I really a test paths will originate. And if you begin your BFS or DFS from the initial node you directly have a test path that will end in one of the final nodes. In case the path does not end in one of the final nodes we could see how to extend it to the final nodes in case the final node is reachable. So, final node is not reachable from the initial node then you have to begin somewhere in another fresh source for DFS or BFS.

(Refer Slide Time: 08:46)



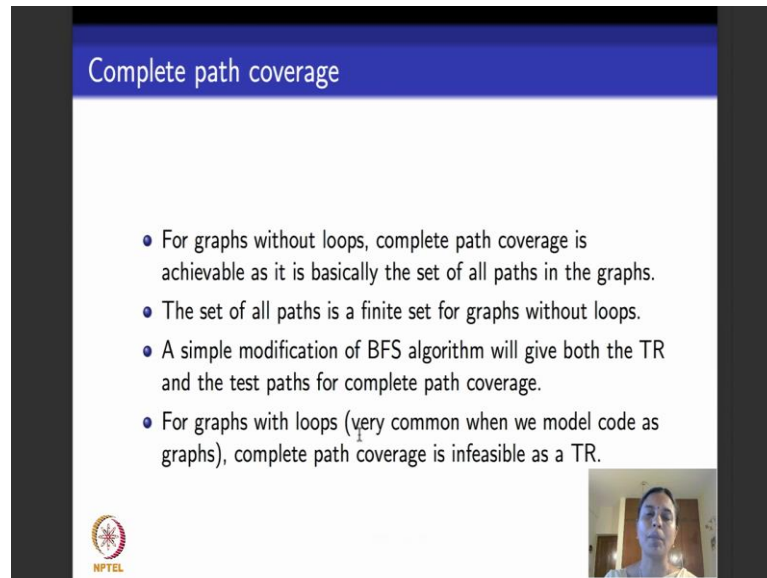
The slide has a blue header with the text "Edge-pair coverage: Algorithms". Below the header, there is a list of bullet points. In the bottom left corner, there is a small circular logo with the text "NPTEL" below it. In the bottom right corner, there is a small video inset showing a person's face.

- TR for edge-pair coverage is all paths of length two in the given graph. Need to include paths that involve self-loops too.
- Test paths for edge-pair coverage:
 - At the basic level, the TR itself constitute the test paths. In fact, we can't do better than this for many graphs.
 - A simple algorithm that considers paths of length two by exploring adjacency list of a node and its successor once, for all nodes in the graph will do for the test paths.

We move on: now the next coverage criteria analyst is what is called edge-pair coverage. The test requirement for edge-pair coverage is all paths of length 2. So, here when I say all path of length 2 I need to include paths that involve self loops also. How do I compute all paths of length 2? I can compute all paths of lengths 2 by exploring the adjacent list of a node and its successor and stopping at that. Exploring the adjacency list of one node will give me the edges that are adjacent to that node that are incident on that node. And exploring the adjacency list of each of these success would get me the address that are incident each of these successors. This should give me a path of length 2 and that is what is needed for a edge-pair coverage.

Once I have the TR for edge-pair coverage the test requirements itself could act as a test path. We saw an example of a graph there in fact we cannot do better than that, the TR itself becomes a test path. Otherwise what I do is I can again run BFS or DFS and get as set of test path that will satisfy edge-pair coverage as a test requirement.

(Refer Slide Time: 09:58)



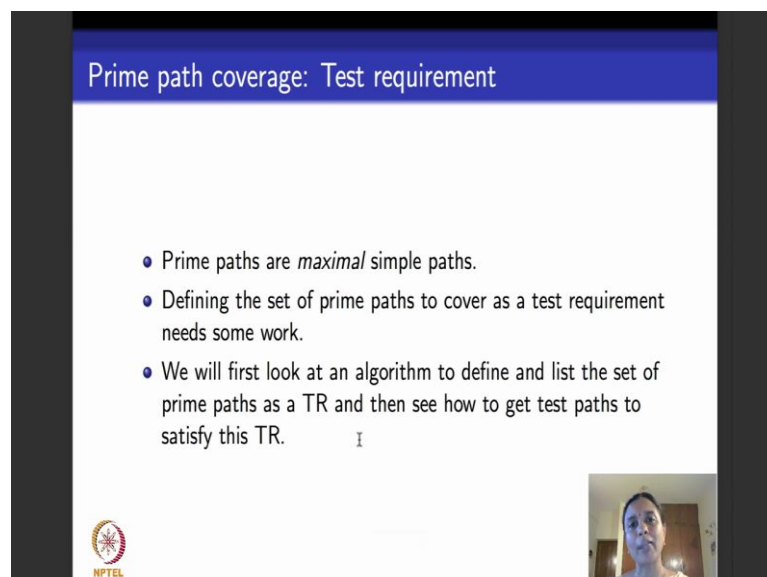
Complete path coverage

- For graphs without loops, complete path coverage is achievable as it is basically the set of all paths in the graphs.
- The set of all paths is a finite set for graphs without loops.
- A simple modification of BFS algorithm will give both the TR and the test paths for complete path coverage.
- For graphs with loops (very common when we model code as graphs), complete path coverage is infeasible as a TR.

NPTEL

Now, the next coverage criterion in our list is what is called complete path coverage. As I told you complete path coverage is many times not feasible, it is not feasible in graph that have loops. And graphs that model software artifacts do have loops typically control flow graphs do have loops. So, there is no point and looking at complete path coverage because it will be infeasible, should be directly move on and look at prime path coverage.

(Refer Slide Time: 10:22)



Prime path coverage: Test requirement

- Prime paths are *maximal* simple paths.
- Defining the set of prime paths to cover as a test requirement needs some work.
- We will first look at an algorithm to define and list the set of prime paths as a TR and then see how to get test paths to satisfy this TR.

NPTEL

So, just to recap what is the prime path prime; path are maximal simple path that is there a simple they do not come as sub path of any other simple path. Now, I want to be able to do the same old two things for prime path, I want to be able to define TR test requirement for prime path, after I finish doing that I want to be able to define test cases that will need this test requirement for this prime path.

To begin with we look at test requirements for prime path and what are the algorithms to see that. So, instead of giving you the pseudo code for the algorithm like we did for BFS and DFS, what I thought we could do is we could take an example graph and I will work you through how the algorithm will run to compute the list of prime path. Once the algorithm computes the list of prime paths what I have with me is basically still my test requirements. With this list of prime paths as my test requirements I now have to go ahead and generate test paths that will cover or satisfy all these prime paths. So, I will work you through an example and explain the algorithm by using that example.

(Refer Slide Time: 11:41)

Computing prime paths: An example

Consider the graph below:

```
graph TD; Start(( )) --> 0((0)); 0 --> 1((1)); 1 --> 2((2)); 2 --> 3((3)); 3 --> 1; 1 --> 5((5)); 5 --> 6(((6))); 0 --> 4((4)); 4 --> 4; 4 --> 6;
```

- Our algorithm will enumerate all simple paths, in order of increasing length.
- We will then choose the prime paths from this list, as and when we enumerate the simple paths.

NPTEL

So, here is a simple graph, here it has seven nodes beginning from 0 and ending at 6; 0 is an initial node, 6 is a final nodes as it marks with a double circle. This graph has branching, it has branching at node 1, it branches into 5 or 2. This graph also has self loop here at node 4 and this graph has a cycle here 1, 2, 3, 1; this is a cycle so it will be interesting to look at prime path. As I told you what is a single big use of prime paths? Assuming that this cycle represents a loop prime path give you a need coverage criteria

that will help you to cover this loop, it will help you to skip the loop, it will help you to execute the loop and its normal operation.

So, you can here you can assume there are two loops: oneself loop here and one cycle here. So, we will see how to compute prime paths for this graph by as a test requirement first and then we will see test cases that will help us to achieve this test requirement. So, what are prime paths? Primes paths are maximal simple paths. So, what is this strategy that my algorithm will follow is the following.

So, what it will do is that it will take this graph and it will enumerate all simple paths one after the other. So, is there a systematic way of enumerating simple paths? Yes, the following the systematic way of enumerating simple paths that we will follow. We will enumerate simple paths in order of increasing length; that is we will enumerate simple paths of lengths is 0, then will enumerate simple path of length 1 will enumerate simple path of length 2 and so on. And then when do we end what is the criteria to end please note that we are enumerating simple paths. So, what is the maximum length that a simple path in a given graph can have? Because simple path cannot contain cycle if the maximum length of path in the given graph can have is the number of vertices.

So, our algorithm is guaranteed to stop. So, what we do to begin with we enumerate simple paths of increasing length, and as and when we enumerate simple paths of increasing length we will mark out some of those paths has been prime paths and pull them out. As and when we mark and pull them out and finish enumerating all the simple paths the final list of mark and pulled out simple paths will be the prime paths that we will obtain as a test requirements.

So, I will show you how that works for this example graph. So, I begin with enumerating simple paths of length 0.

(Refer Slide Time: 14:21)

Computing prime paths: Example

Simple paths of length 0 (7 paths).

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

Exclamation mark (!) after path [6] implies [6] cannot be extended further. Note that 6 is a final node and has no out-going edges.

So, what is a simple path of length 0? Simple path of length 0 is just the vertex, because it has one single vertex. How many vertices are there in this graph? There are seven vertices starting from 0, 1, 2, 3 and so on up to 6.

So, I enumerate each of these vertices as a path of length 0. There are seven such paths of length 0. I just enumerated them. Another things to be noted is at the end of this 0 length simple path which contains the single vertex 6 I have put an exclamation mark. What is the role of the exclamation mark? The exclamation mark tells us that this path this simple path which contains the single vertex 6 cannot be extended anymore as far as this graph is concerned. Why is that so? Because 6 is a final vertex and there are no edges that are going out of 6.

So, as far as this graph is concerned simple path 6 cannot be extended anymore. So, I remember that by putting an exclamation mark. Now, after enumerating all paths of length 0 and go ahead and enumerate paths of length 1.

(Refer Slide Time: 15:34)

Computing prime paths: Example

Simple paths of length 1 (9 paths).

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

The asterisk (*) implies that the path cannot be extended further, it is already a simple cycle.

NPTEL

So, here is the same graph and here are the paths of length 1. How have a enumerated paths of length 1? I go back to the previous slide, so I say here is a path of length 0 which begins at vertex 0. So, you go back and look at the graph vertex 0; vertex 0 what are the two paths emerging from vertex 0? A path of length 1 from 0 to 4, a path of length 1 from 0 to 1; that is what I have written here vertex 0 this path of length 0 can be extended to two paths of length 1, namely the edge 0, 1 and the edge 0, 4.

Similarly, vertex one which was a simple path of length 0 can be extended to two paths of length 1 paths 1, 2 and path 1, 5. If we move on vertex 2 can be extended to path 2, 3; vertex 3 can be extended to path 3, 1; vertex 4 can be extended to two simple paths of length 1 one leading to 6 and 1 using this self loop from 4 to itself and vertex 5 can be extended to one simple path of length 1 which is the edge 5, 6.

Now let us look at what are these exclamation marks that we have put like last slides. So, again in this listing of paths of length 1; nine simple paths I have put two paths with exclamation mark, which means both the paths end in the final nodes 6 and they cannot be extended further. So, exclamation mark for us is to mean that this path cannot be extended further. Why am I interested in paths that cannot be extended further? Path that cannot be extended further could mean that they are heading towards being a maximal simple path. And that is my interest right, maximal simple paths of prime path is what I want to look at.

So, in addition to exclamation mark I have also gone ahead and put an asterisk. What is that mean? That means the following; this means that this path which is this edge 4 to 4, it cannot be extended further not because there are no outgoing edges, but because it already forms a simple cycle. Remember if the only way to extended further is to use this edge once again 4, 4 and 4; in which case it will not turn out to be a simple path which is not interest to me. And the other way to do it is to do 4, 4 or 6 in which case also it will not be a simple path so it is not of interest to me. So, I mark it with a asterisks which says that this path also cannot be extended further.

Now I look at all the other remaining paths which are this list on top, and see how to extend them to get simple path of length 2.

(Refer Slide Time: 18:27)

Computing prime paths: Example

Simple paths of length 2 (8 paths).

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

NPTEL

So, you look at the paths 0, 1; I extend the path 0, 1 into two possible paths of length 2 path going from 0 to 1 and then from 1 to 2 and path going from 0 to 1 and then from 1 to 5; that is what is listed here.

Similarly, the path 0, 4 can be extended to 0, 4, 6. Please note that I can also extend 0, 4 to 0, 4 and 4, but that is not as simple path so I do not list it here. And I go on doing this path 1, 2 can be extended to 1, 2, 3 path 1, 5 can be extended to 1, 5, 6 which comes with an exclamation mark because that is where it stops cannot be extended further. Similarly path 2, 3 can be extended to 2, 3, 1. 3, 1 can be extended to 3, 1, 2 and 3, 1, 5. So, paths

of length 2 how many paths are there and two of them cannot be extended further.

(Refer Slide Time: 19:24)

Computing prime paths: Example

Simple paths of length 3 (7 paths).

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

Many paths of length 3 cannot be extended.

NPTEL

Now, I start with paths of length 2 and consider path of length 3. It so happens that I get seven different paths. So, path 0, 1, 2 can be extended to 0, 1, 2, 3 path 0, 1, 5 can be extended to 0, 1, 5, 6 right path 1, 2, 3 can be extended to 1, 2 there one I can go back and so on.

If you see in this listing almost every paths of length 3 is marked with an exclamation mark or an asterisk. In fact, there is exactly one path that is not marked with either of this. What do all these mark path mean? They mean that that is it we cannot extend them anymore. We cannot extend them anymore for two reasons: one is the end in the vertex from which there are no outgoing edges or if we extend them further I violate the criteria of they being a simple path.

So, I am pretty much closed here as for as paths of length 3 are concern there is only one path here 2, 3, 1, 5 that is not marked so that is the only path that has go for extension. So, I go ahead take that path and extend it by adding the edge 5, 6 you see 2, 3, 1, 5 that is a path of length 3 I can extend it to length 4 by adding this edge 5, 6.

(Refer Slide Time: 20:36)

Computing prime paths: Example

- Only one path of length 4 exists: [2,3,1,5,6]!
- The above process is continued:
Every simple path without a ! or a * can be extended.
- The process is guaranteed to terminate as the length of the longest prime path is the number of nodes.
- There are totally 32 simple paths in the example graph, only 8 of them are prime paths.

NPTEL

That is what I have done here. And the moment I did that I am force to put an exclamation mark here for the same reason, because I have ended in the node 6 which is a final node and it does not have any outgoing edges.

(Refer Slide Time: 21:05)

Computing prime paths: Algorithm

I

- To enumerate paths of length 2, consider each path of length 1 that is not a cycle (marked with a *).
- Extend the path of length 1 with every node that can be reached from the final node in the path, unless that node is already in the path and not the first node.
- Repeat the above till we reach paths of length $|V|$, where V is the set of vertices of the given graph.

NPTEL

So, what do I do, here is the algorithm in English. I consider paths of length 2 by extending paths of length 1 with every node that can be reach from the final node in the path unless that node is already in the path and is not the first node. And I keep doing this for paths of length n increase to path of length n plus 1 till I reach path of length mod

V. Why do I stop at mod V; because that gives me the maximum length simple paths. Any other path of length greater than mod V, but (Refer Time: 21:37) principle one vertex as to repeat and the path will not be simple any more.

So, I hope the algorithm is clear. What is the algorithm do? Starts with paths of length 0 extends them to obtain paths of length 1, extends them to obtain paths of length 2, extends them to obtain paths of length 3, keeps repeating this still it can get paths of maximum length which is the length as the number of vertices. While doing this extension it marks out certain paths. There are two kinds of marking that we do; we mark at with an exclamation mark or we mark with an asterisk. When do we mark with an exclamation mark? When I know that that particular path cannot be extended because there may not be any outgoing edges; and mark it with an asterisk when I know that if I extended I will violate the criteria that this path needs to be a simple path. All the other paths are every step that are not marked with one of the special markers can be extended and I keep repeating this process.

So, for this particular graph it so happens that there are 32 simple path for this particular graph, but the only eight of them are prime paths. And those are the ones that have been obtained by these markings. In general it is not a easy problem to count the number of simple path, I have just given this for the specific graph. And why is this algorithm guarantee to terminate, because it is to stop when I get a path of length mod V as we discussed.

(Refer Slide Time: 23:07)

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]!

NPTEL

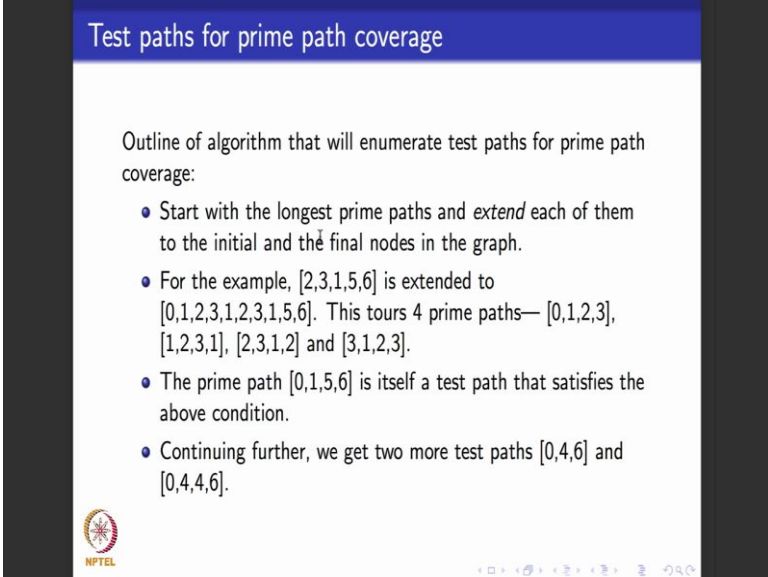
So, now I will just present this graph and all the prime paths for the graph. So, this is the graph that we be using to understand an algorithm, and here are all the prime paths that I worked my algorithm out and I will listed through that. So, let us look at them the path 4, 4 is this which was listed as a path of length 1 if you remember and write them marked as with the star because it could not be extended it was already a simple cycle. Then there was a path 0, 4, 6. Then there is a path 0, 1, 2, 3; 0, 1, 5, 6; then 1, 2, 3, 1; and then 2, 3, 1, 2; 3 1, 2, 3.

If you see these three paths no 1, 2, 3, 1 they basically corresponded this cycle in the graph and between the three paths they tell you various ways in which you could diverse this lop. And this path 4, 4 is a prime path that covers this loop. The rest of the paths traverse the rest of the graph. Like for the example the path 0, 1, 5, 6 skip this loop, the path 0, 4, 6 skips this loop. And then the other path that is left is 2, 3, 1, 5, 6 that is like existing this loop 1, 2, 3 and coming out to a final node.

So, for this graph our algorithm that begins by enumerating simple path of increasing length and identifying prime path as and when we enumerate them in by listing all these eight paths as prime path in the graph. And if you sort of play them back in the graph like we did just now you can see how they neatly cover the two loops and how they skip the loop in the graph.

Now, what have we done? What we have done so far just gives us the test requirement or TR for prime path coverage. We say now you write a set of test cases for covering these paths. This elaborate exercise that we did was just to obtain the test requirement for prime path coverage. We still have to go ahead and design test cases to be able to get text path to meet the test requirement for prime path coverage.

(Refer Slide Time: 25:28)



The slide is titled "Test paths for prime path coverage" and contains the following text:

Outline of algorithm that will enumerate test paths for prime path coverage:

- Start with the longest prime paths and *extend* each of them to the initial and the final nodes in the graph.
- For the example, [2,3,1,5,6] is extended to [0,1,2,3,1,2,3,1,5,6]. This tours 4 prime paths— [0,1,2,3], [1,2,3,1], [2,3,1,2] and [3,1,2,3].
- The prime path [0,1,5,6] is itself a test path that satisfies the above condition.
- Continuing further, we get two more test paths [0,4,6] and [0,4,4,6].

The slide also features the NPTEL logo in the bottom left corner and navigation icons in the bottom right corner.

How do we go ahead and get the test paths for prime path coverage? Again there are several algorithms that have been used in literature. I will give you one algorithm that are illustrate on the same example because that turns out to list the prime paths it is a heuristic that lists is reasonably faster than other non algorithms. So, what we do is in this list of prime path which are given as my test requirement you start with the longest path. Which is the longest path in this list? That is the last path 2, 3, 1, 5, 6. What you do is that this you extend this path to the left to see if you can find a path from the initial node to the beginning node of this path. And then from the right assuming that this is not a final node you see if you can extend it to the right to make it end in a final node.

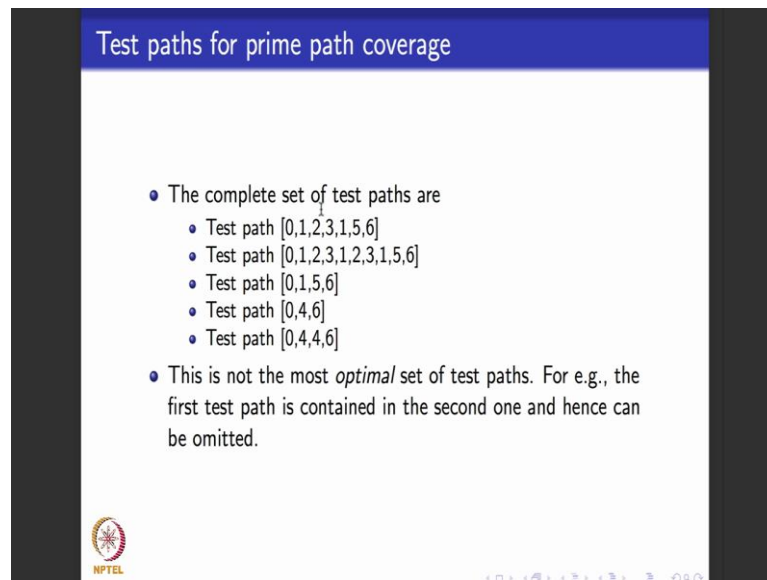
In this case 6 is already a final node, so I have already ended in the final node, but 2 is not an initial node. So, I take the longest prime path and see if I can go to the left and extend this path to make it as if it was beginning from an initial node. So, if I do that for 2, 3, 1, 5, 6 I get such a path; I get 0, 1, 2, 3, 1, 2, 3, 1, 5, 6. So, I will trace it out here 0, 1, 2, 3 right; 1, 2, 3, 1, 5, 6. Now you might ask why did I repeat this 1, 2, 3 twice; I

repeat that with the purpose of I could have not done that, I could have done 0, 1, 2, 3 1, 5, 6 in which case I would not have covered these paths here. If you remember I had three prime paths which tell you how to traverse the loop beginning from three different vertices of the loop. I could begin the loop at 1, I could begin the loop at 2, I could begin the loop at 2; then you basically going round the loop once, but the vertices in which they were beginning where different. So, that I want to be able to cover all of them here so that is the reason why I repeat this loop here.

Please note that this is a test path, so that need not satisfy the notion of a simple path or a prime path, because it is just an ordinary test path. The only condition that it has to satisfy is it has to begin at an initial node which is 0 for our example and ended at a final node which is 6 for an example. In between nodes can be repeated, because it is not a test requirement for prime path coverage it is only a test path for prime path coverage. So, test paths need not be simple cycles, they need not be maximal simple paths. So, if I do that then right there I have toward four prime paths in this list: I have done this, I have done this loop, so I have done I entered the loop from the initial state and I have gone through the loop.

So, what I do is the remaining; what is the remaining amongst the remaining prime path which is the longest prime path that is left out that is this 0, 1, 5, 6. It so happens that that already begins in an initial node and ends in final node. So, does have to be extended anymore, so I leave it. What is the other path that is left out? Those are these two. In this path 0, 4, 6 again begins in the initial nodes and end in the final node, but I need to be able to give scope to cover this loop so extended it to 0, 4, 4, 6.

(Refer Slide Time: 29:13)



The slide is titled "Test paths for prime path coverage" in a blue header. Below the header, there is a bulleted list of test paths. The first bullet point states "The complete set of test paths are" and is followed by five sub-bullets: "Test path [0,1,2,3,1,5,6]", "Test path [0,1,2,3,1,2,3,1,5,6]", "Test path [0,1,5,6]", "Test path [0,4,6]", and "Test path [0,4,4,6]". A second main bullet point states "This is not the most optimal set of test paths. For e.g., the first test path is contained in the second one and hence can be omitted." In the bottom left corner, there is a small NPTEL logo. In the bottom right corner, there are small navigation icons.

So, putting it all together the complete set of test paths for prime path coverage as my test requirement are this list that I obtain. If you remember I took this which is the longest prime path I extended it to cover the loop over the vertices 1, 2 and 3, then I took the path that skip the loop, then there was one more loop which these two test paths cover. If you notice and if you worry about is this the best set of test paths that I can get for prime path coverage for this graph I would say no. Why because, if you see the simple reason the 0, 1, 2, 3, 1, 5, 6 is completely embedded within this second path, so I could remove one of them. So, I am not got the least number of test path that I could cover.

Similarly 0, 4, 6 is completely embedded in 0, 4, 4, 6, so I could remove 0, 4, 6, but that is alright. The focus for us is not to get the most optimal set off test paths, the focus for us is to be able to correctly defined the test requirement and get a set of test path that satisfy the test requirements; that is what we have done here.

(Refer Slide Time: 30:22)

The slide has a blue header with the text "Test requirements and their test paths". Below the header is a white area containing a bulleted list of four points. In the bottom right corner of the slide, there is a small video inset showing a person speaking. The NPTEL logo is visible in the bottom left corner of the slide.

- 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 is.
- Many algorithms that come up with optimal test paths for a given TR are NP-complete.

In general the problem of what is optimality, optimal test path corresponding to a test requirement. There are several notions of optimality available in the literature. We will not really focus on that because I want to be able to move on, but I point you to good references where you could look up for this kind of information. In general many algorithms that come up with optimal test paths, or decide what is an optimal notion and typically intractable problems and lot of them are NP-complete problem.

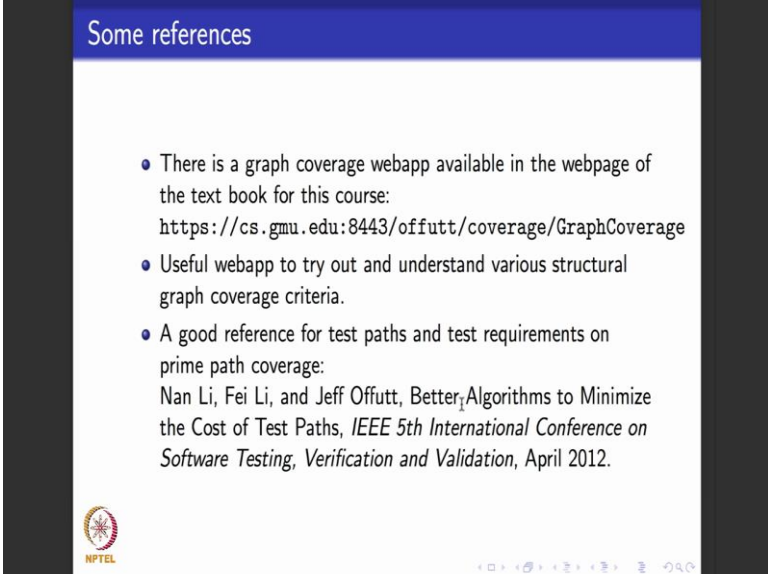
(Refer Slide Time: 30:53)

The slide has a blue header with the text "Symbolic execution based algorithms". Below the header is a white area containing a bulleted list of two points. In the bottom right corner of the slide, there is a small video inset showing a person speaking. The NPTEL logo is visible in the bottom left corner of the slide.

- There are several *symbolic execution* based algorithms for obtaining test paths to achieve coverage criteria.
- We will see symbolic execution later in the course and use it for path coverage in programs.

In fact, what we saw was explicit state algorithms- algorithms that use breath for search or depth for search you could leave them aside and do what is called symbolic execution based algorithms. I will do symbolic execution a little later in the course after a few weeks; they also can be used to obtain specified path coverage criteria.

(Refer Slide Time: 31:15)



Some references

- There is a graph coverage webapp available in the webpage of the text book for this course:
<https://cs.gmu.edu:8443/offutt/coverage/GraphCoverage>
- Useful webapp to try out and understand various structural graph coverage criteria.
- A good reference for test paths and test requirements on prime path coverage:
Nan Li, Fei Li, and Jeff Offutt, Better Algorithms to Minimize the Cost of Test Paths, *IEEE 5th International Conference on Software Testing, Verification and Validation*, April 2012.

NPTEL

I would like to end this module by encouraging you to use these web apps. So, the text book that have been using for this course is the book by a (Refer Time: 31:23) software testing in the web page of the text book they have a very nice web app for several different coverage criteria that they introduce in the textbook. For now you begin by using this particular web app the app that is available in this URL. You can try out how the various structural coverage criteria work by input in your own graph. And then they have algorithms which are basically Java programs they have written with will output the test requirement and the test paths that satisfy the test requirement.

If you want go ahead and read further on this paper is a very good reference to talk about prime path coverage, which is the most difficult structural coverage criteria that we saw till now.

Thank you.