

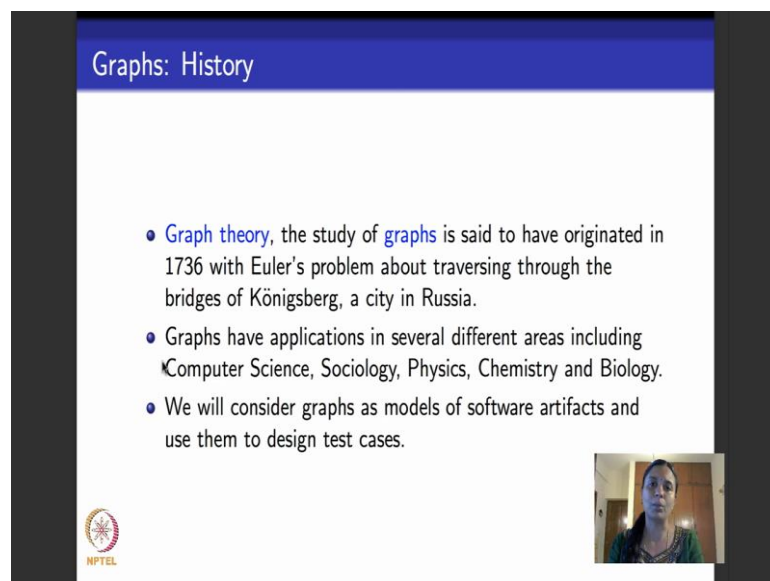
**Software Testing**  
**Prof. Meenakshi D'Souza**  
**Department of Computer Science and Engineering**  
**International Institute of Information Technology, Bangalore**

**Lecture – 05**  
**Basics of graphs: As used in testing**

Hello everyone. So, we begin looking at test case design algorithms in this module what I would be starting is to model software artifacts as graphs which will be one of the four structures that we would consider. If you remember, we said we would consider graphs, we would consider logic and expressions, we would consider sets that model inputs to software and finally, we would consider the underline grammar from which software programming language is written. So, we begin we are looking at test case algorithms that deal with graphs. So, why we look at test case algorithms that deal with graphs I would like to recap some basic terminologies related to graphs as we would be doing in this course in testing.

Graphs and graphs theory are vast areas, I will not be able to do just is to be able to cover even the basic minimum concepts that we deal with in graphs. So, we will restrict ourselves to just looking at terminologies that we need as far as test case design algorithms are concerned in this course.

(Refer Slide Time: 01:18)



The slide is titled "Graphs: History" in a blue header. It contains three bullet points: "Graph theory, the study of graphs is said to have originated in 1736 with Euler's problem about traversing through the bridges of Königsberg, a city in Russia.", "Graphs have applications in several different areas including Computer Science, Sociology, Physics, Chemistry and Biology.", and "We will consider graphs as models of software artifacts and use them to design test cases." In the bottom left corner is the NPTEL logo, and in the bottom right corner is a small video inset showing a woman speaking.

**Graphs: History**

- Graph theory, the study of graphs is said to have originated in 1736 with Euler's problem about traversing through the bridges of Königsberg, a city in Russia.
- Graphs have applications in several different areas including Computer Science, Sociology, Physics, Chemistry and Biology.
- We will consider graphs as models of software artifacts and use them to design test cases.

NPTEL

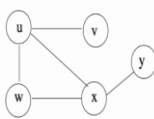
So, graph theory is a very old subject it is believed that study of graphs was initiated by Euler in the year 1736 when they were trying to look at this city called Konigsberg in Russia and then they were trying to model a problem of crossing the bridges in this city in a particular way. So, they considered modeling this problem as a graph and graph theory is supposed to have originated with Euler's theorem which is considered an old theorem. So, you can imagine how old graphs is. Today graphs enjoy applications not only in computer science, but in several different areas all sciences physics, chemistry, biology, and in fact, it finds exist extensive applications in sociology where people look at social networks and other entities is very large graph models.

So, what do we do with graphs? We will consider graphs as models of software artifacts and see how to use graphs to design test cases as we want them.

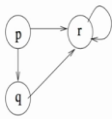
(Refer Slide Time: 02:19)

Graph

- A graph is a tuple  $G = (V, E)$  where
  - $V$  is a set of nodes/vertices.
  - $E \subseteq (V \times V)$  is a set of edges.
- Graphs can be directed or undirected.
- In an undirected graph, the pair of vertices constituting an edge is unordered, i.e., whenever  $(u, v) \in E$  then  $(v, u) \in E$  and vice versa.
- In a directed graph, the pair of vertices constituting an edge is ordered.



A simple undirected graph



A directed graph

So, we begin by introducing what a graph is say assume that you not seen it before. So, I will introduce you from the very basic concepts. If you seen it before feel free to, you know, sort of skip through these parts because they are talk about the basic terminologies related to graph. So, how does a graph look like? It looks like this. So, this is what is called an undirected graph which is simple in the sense that it does not have any self loops and here is what is called a directed graph, where the edges have directions.

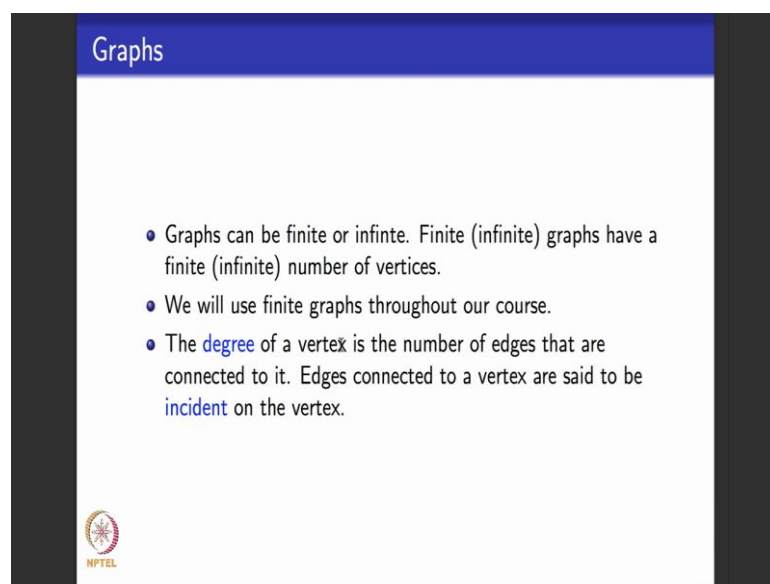
So, graphs have two parts to it there are nodes or vertices. Sometimes I will use these terms synonymously interchangeably. Some people also call it is points and different

books might call them differently. So, that is typically sets that is marked using circles like this each circle is given a number. So, there are 5 vertices or nodes in this graph which are labeled as  $u, v, w, x$  and  $y$ . Similarly there are three nodes vertices in this graph labeled as  $p, q$  and  $r$  and graphs also have what are called edges. Edges are also called arcs or lines in certain other books. So, what is an edge? Edge is basically a pair of two nodes or two vertices. So, this pair of nodes or vertices can be ordered or unordered.

So, if the pair is unordered that is, I do not really worry about whether I am looking at the pair  $(u, v)$  or  $(v, u)$ , then I say that the graph is an undirected graph and if the pair that I look at is ordered, like if I look at this figure of a directed graph on the right and looking at in ordered pair  $p$  comma  $r$ . So, there is an edge in this direction from  $p$  to  $r$  and in this graph there is no edge in the reverse direction right. So, such graphs are what are called directed graphs. Of course, it is to be noted that an edge can take vertex to itself there is no requirement that  $u$  should be different from  $v$ .


So, if you look at this directed graph the pair  $(r, r)$  constitutes this edge which is the self loop around the vertex  $r$ . So, when I look at a pair  $r$  comma  $r$  in an directed or in undirected graph because it is an reflects a pair I really do not worry about whether the pair is ordered or unordered right it does not matter. Otherwise, for each other pair of distinct vertices whether the pair is ordered or unordered defines the kind of graphs that we look at. Graphs could be directed as it is here or undirected as it is here.

(Refer Slide Time: 04:56)



### Graphs

- Graphs can be finite or infinite. Finite (infinite) graphs have a finite (infinite) number of vertices.
- We will use finite graphs throughout our course.
- The **degree** of a vertex is the number of edges that are connected to it. Edges connected to a vertex are said to be **incident** on the vertex.



So, graphs can be finite or infinite. So, a finite graph typically has a finite number of vertices and infinite graph has an infinite number of vertices. Because our use of graphs in this course is to be able to use them as artifacts that modeled various pieces of software we will not really consider infinite graphs, we do not really have the need to modeled any kind of software artifact as an infinite graph. So, we will look at finite graphs throughout this course.

A few other terminologies in this graph. What is the degree of a vertex? The degree of a vertex gives you the number of edges that are connected to the vertex another term for connected that people use in the graph theory is to say that an edge is incident on the vertex. So, if we go back to the graph examples that we have in the previous slide if you take this vertex  $u$  in this undirected graph three edges are connected to this vertex you write one coming from  $w$  one connecting it to  $x$  and one connecting it to  $v$ . So, all these three edges are supposed to be connected to  $u$  or incident on  $u$  and so the degree of the vertex  $u$  is 3.

So, similarly degree of the vertex  $y$  is just 1 because there is only one edge that is incident on  $y$ . So, if you go to this directed graph the degree of the vertex  $r$  would be 3. So, when we count the degree of a vertex for an undirected graph we count the in-degree of the vertex, in-degree is the number of edges that come into a particular vertex. So, here there are three edges that come into  $r$  - one from  $p$ , one from  $q$  and one from  $r$ . So, we say  $r$  has in-degree 3, right. Similarly we also talk about an out-degree of a vertex in an directed graph. So, if it look at the vertex  $p$ ,  $p$  has in-degree 0 because there is no edge that is coming into  $p$ , but  $p$  has out degree 2 because two edges go out of  $p$ .

If you look at the vertex  $q$ ,  $q$  has in-degree 1 because this edge from  $p$  to  $q$  comes into  $q$  and  $q$  has out degree 1 because this edge from  $q$  to  $r$  goes out of  $q$ .

(Refer Slide Time: 07:16)

**Initial and final nodes**

- For many graphs, there are designated special vertices like **initial** and **final** vertices.
- These vertices indicate beginning and end of a property that the graph is modeling.
- Typically there is only one initial vertex, but, there could be several final vertices.
- Initial vertex represents the beginning of a computation (of a piece of code) and the computation ends in one of the final vertices.

A simple undirected graph      A directed graph

So, moving on the graphs that we will look at will have several other things apart from just vertices and edges. So, one at a time we look at what are the add on or the additional features or annotations that we will consider to be a part of the graphs that we look at right. So, graphs could have special designated vertices called initial vertex and final vertex. So, what I have done here is I have taken the same graph that you saw two slides ago and marked the vertex u as an initial vertex u has an edge that is incoming line that is incoming into u, but it does not really have any vertex on the other side. So, such vertexes what is called an initial vertex or an initial node.

There could also be what are called final vertices. Final vertices are believed to be vertices that capture the end of some kind of computation in the models that we will look at when we will look at graph models corresponding to code and corresponding to design elements and become clear what is the purpose of final vertex, but from now you can understand it to be a final vertex is one in which computation is supposed to end in some way or the other and in our pictures that we will look at, final vertices will be marked by this concentric circle. So, if you see in this directed graph p is an initial vertex and r is a final vertex in this undirected graph u is an initial vertex and w is a final vertex.

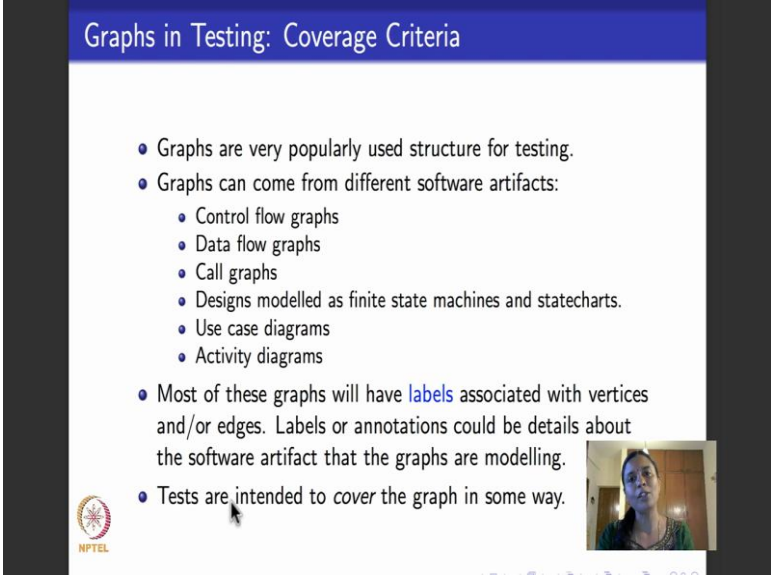
So, typically we believe that most of the software artifacts that we will consider like code mainly or design, is always supposed to be deterministic in the sense that its behaviour is definite and there is no non determinism in its behaviour. So, to be able to capture graph

as a model of a software that is meant to be deterministic, we all always say that the graph will typically always have only one initial state. If there were more than one initial states then it will be a bit confusing as far as determinism is concerned right because if there are more than one initial states let say there are three initial states where would you consider the computation as beginning from. You could interpret it as it is beginning from any one of the initial states, but then right there, the software is non deterministic right and we really do not typically look at nondeterministic software. Non deterministic implementations of software do not exist.

We always look at deterministic implementations of software and hence we will consider a graph models to always have only one initial state, but software as its executing could take one of the several different execution paths that it goes through and based on the path that it takes it could end in one of the many different states that it is in. So, typically graph models that represent software artifacts we will have more than one final state. In this example that I have shown in this slide it so happens that both these graphs have exactly one final state, but that need not be the case in general.

So, what is the summary of this slide? So, certain vertices in graphs which occurs models of software artifacts could be marked as special initial vertices from where computation is suppose to begin. We will identify them by this incoming line, which is not connected to any vertex on the other end and some vertices are marked as special final vertices which are marked by this double circles as you can see in these two figures and they are supposed to represent vertices in which computations end. Another point to note is that there could be graphs in which both initial vertex and one of the initial/final vertices is also an initial vertex it is nothing that is specified which says the set of the initial vertices and final vertices should be disjoint there is no requirement like that right.

(Refer Slide Time: 11:06)



The slide is titled "Graphs in Testing: Coverage Criteria" in a blue header. It contains a bulleted list of points. The first point states that graphs are a popular structure for testing. The second point lists various software artifacts that can be represented as graphs: control flow graphs, data flow graphs, call graphs, designs modeled as finite state machines and statecharts, use case diagrams, and activity diagrams. The third point notes that most graphs have labels associated with vertices and/or edges, which provide details about the software artifact being modeled. The fourth point states that tests are intended to cover the graph in some way. In the bottom left corner, there is a small NPTEL logo. In the bottom right corner, there is a small video inset showing a woman speaking.

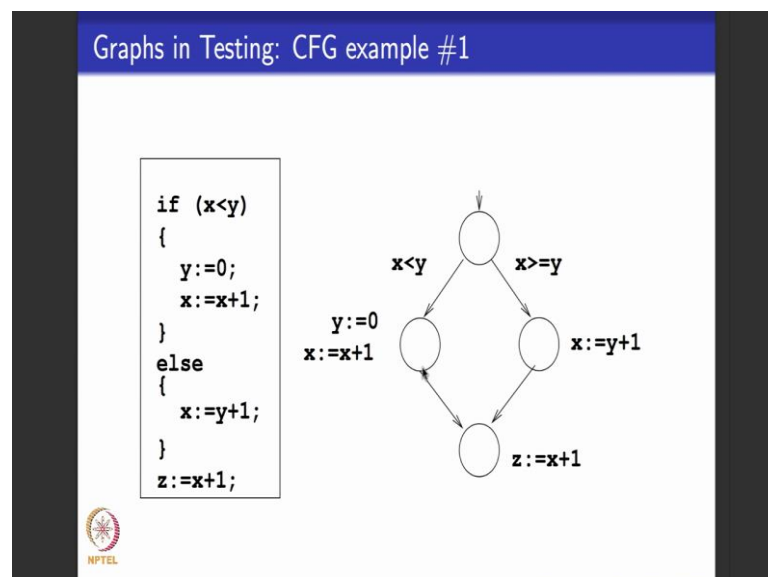
- Graphs are very popularly used structure for testing.
- Graphs can come from different software artifacts:
  - Control flow graphs
  - Data flow graphs
  - Call graphs
  - Designs modelled as finite state machines and statecharts.
  - Use case diagrams
  - Activity diagrams
- Most of these graphs will have labels associated with vertices and/or edges. Labels or annotations could be details about the software artifact that the graphs are modelling.
- Tests are intended to cover the graph in some way.

So, how are we going to use them as graphs? Why do we need look at graphs as far as software testing is concerned? Graphs, I believe, are next to logical predicates, may be a very popular structure used in testing right the several testing and static program analysis tools use graph models of software artifacts. So, where do these graphs come from? They could come from several different sources in software artifacts they could represent control flow graph of a particular program. Do not worry if you do not know these terms will introduce each of these terms as we move on in the course. They could represent what is called a data flow graph corresponding to a piece of code, they could represent what is called the call graph corresponding to a piece of code.

They could represent a software design element which is a modeled let say as an UML finite state machine or a UML state chart, they could represent a requirement which is given as a use case diagram or an activity diagram in the UML notation. All these are basically graph models that represent several different artifacts. Now you might ask a question, the kind of graphs that I define to you just a few minutes ago just had vertices edges and may be some vertices marked as initial vertices and some vertices marked as final vertices right. Obviously, the kind of graphs that I am talking about here through these different software artifacts and not going to be as simple as that. They we will typically have lot of extra annotations or labels as parameters right.

There could be labels associated with vertices there could be labels associated with edges and so on and so forth as and when needed we will look at a corresponding kind of graph, but no matter what the kind of graphs they are they will always have this underlying structure. We have vertices, some vertices marked as initial vertices, some vertices marked as final vertices and set of edges the edges could be directed or undirected. They will always have this structure and we will set of other things. And how are we going to use? Our goal is to be able to design test cases that we will cover this graph in some way or the other covering in the sense of coverage criteria that I defined to you in one of my earlier lectures.

(Refer Slide Time: 13:26)



So, here is an example of how typical graph occurs as a model of a software artifact. Another thing that I would like to reiterate at this point in my course is that when we look at examples like this I will never show you a complete piece of code that is fully implemented; I will always show you a small fragment of code. Like if you see in this example I have just shown a small fragment of code this does not mean that this is the entire program. It can never be a full complete program and it does not make sense if it is a full complete program because you do not know what its inputs are, what its outputs are where are the outputs being produced its clearly not an implementation ready code right.

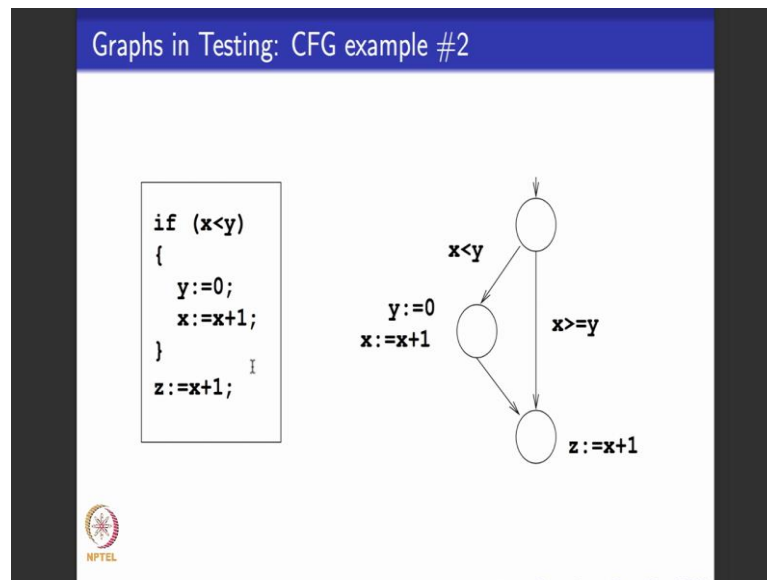


We will always look at fragments of code that is useful for us to understand a particular algorithm or a methodology for test case design. We will of course, see examples where we will see full pieces of code, but as I go through my lecture we will see a lot of code fragments. So, do not confuse them with the code that is ready for implementation, it will not be the case. So, here is a piece of code fragment which talks about an if statement. So, there is an if statement which says that if  $x$  is less than  $y$  which means if this predicate turns out to be true then you go ahead and execute these two statements what are these two statements one says assign 0 to  $y$  and the other says make  $x$  is equal to  $x$  plus 1 and with this predicate turns out to be false then you say you say  $x$  is  $y$  plus 1 and no matter what you do when you come out you make  $z$  as  $x$  plus 1.

So, here is a graph model corresponding to this particular code fragment. How does this graph model look like? So, corresponding to this first if statement there is an initial node in the graph which is marked here. This if statement basically tests for true or falsity of the predicate  $x$  less than  $y$ . So, if  $x$  is less than  $y$ , this code takes this branch. If  $x$  is not less than  $y$  which means  $x$  is greater than or equal to  $y$  then this code takes, but this branch. Suppose  $x$  is less than or equal to  $y$  then these two statements are to be executed. So, when it takes this branch, I model one collapsed control flow mode which basically represents the execution of two statements in order, in the order in which they occur. The two statements are as they come in the code--- assign 0 to  $y$ , assign  $x$  plus one to  $x$ .

Suppose  $x$  less than  $y$  was false, then the code takes the else branch and it comes here and it executes the statement  $x$  is equal to  $y$  plus 1. So, it does not matter whether it takes the then branch and next branch as per this example when it comes out of the if, it executes the statement. So, no matter whether it goes here or it here it always comes back and executes this statement. So, this is how we modeled controlled flow graph corresponding to a particular program statements. So, later I will show you for all other constructs, for loops and other things how those control flow graphs look like.

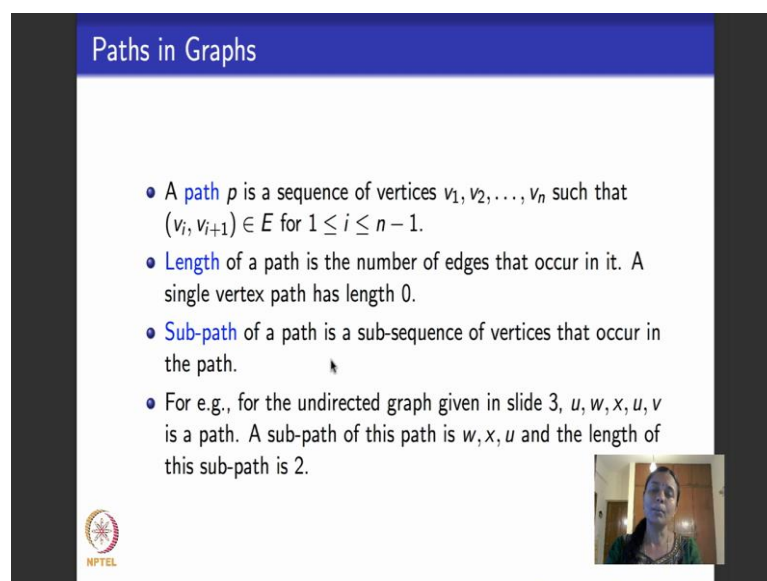
(Refer Slide Time: 16:33)



Here is another small example. Let say suppose you take the same if statement, but it in have the else clause right. So, there is nothing that specified in the code about what to do when this condition in this predicate  $x$  less than  $y$  is false. It does not matter, people can write code like that in that case what you do is if  $x$  is less than  $y$  from here this node which represents this if statement you do these two statements at this node which is assign 0 to  $y$  and assign  $x$  plus 1 to  $x$  and if this condition  $x$  less than  $y$  is false then you come directly and execute the statement  $z$  is equal to  $x$  plus 1.

So, if you see this kind of graph the graph that we saw in this slide or the graph that we saw in this slide they have vertices as we saw then they have a designated initial vertex. I have not marked any vertexes final vertex because I do not know whether the computation of this code fragment ends here as a part of the larger code or not, And, in addition to that if you see both vertices and edges have labels associated with them. These vertices, this vertex is labeled with two statements from the program this vertex is labeled with one statement from the program, these two edges are labeled with what are called guards or predicates that tell when this edge can be taken. So, like this typically all other models of graphs that we will look at, which is from this list, we will always look at some kind of extra annotations labels that comes with these kind of structures.

(Refer Slide Time: 18:03)



**Paths in Graphs**

- A **path**  $p$  is a sequence of vertices  $v_1, v_2, \dots, v_n$  such that  $(v_i, v_{i+1}) \in E$  for  $1 \leq i \leq n-1$ .
- **Length** of a path is the number of edges that occur in it. A single vertex path has length 0.
- **Sub-path** of a path is a sub-sequence of vertices that occur in the path.
- For e.g., for the undirected graph given in slide 3,  $u, w, x, u, v$  is a path. A sub-path of this path is  $w, x, u$  and the length of this sub-path is 2.

NPTEL

So, another important concepts that we need related to graphs as its used in testing is a notation of path in the graph. Think of a path as in the graph a sitting at some vertex or a node and just using the I just to walk through the graph. So, what is a path? Path is a sequence of vertices, just a finite sequence of vertices. I told you will consider finite graphs; finite graphs does not mean that they give rise to finite paths because a graph could have a cycle where in you can take it again. Cycle is a path that begins and ends at a same vertex and you could take it again and again to be able to get an infinite path, but we will look at finite paths for now. So, a path is a finite sequence of vertices let us say  $v_1, v_2, \dots, v_n$ , such that each pair of successive vertices in the path is connected by an edge.

So, if I go back here to the example graphs that we saw in the first slide right - here is a path and this graph it begins that  $u$ , from  $u$  I go to  $w$ , from  $w$  I go to  $x$ , from  $x$  I can go back to  $u$  and let say from  $u$  I go to  $v$  right. So, this is a path in the graph. So, what is the length of the path? When you talk about the length of the graph we count number of edges in the paths? So, the length of the path is a number of edges single vertex could be a paths and the length of such a path will be 0, right. So, what is a sub-path of a path? A sub-path of a path is just a subsequences of vertices that occur in the path. If we go back to that example graphs that we had in mind, I told you  $u, w, x, u, v$  is a path right.

(Refer Slide Time: 19:57)

Reachability in graphs

- A vertex  $v$  is **reachable** in a graph  $G$  if there is a path from one of the initial vertices of the graph to  $v$ .
- An edge  $e = (u, v)$  is **reachable** in a graph  $G$  if there is a path from one of the initial vertices to the vertex  $u$  and then to  $v$  through the edge  $e$ .
- A sub-graph  $G'$  of a graph  $G$  is **reachable** if one of the vertices of  $G'$  is reachable from an initial node in  $G$ .

NPTEL

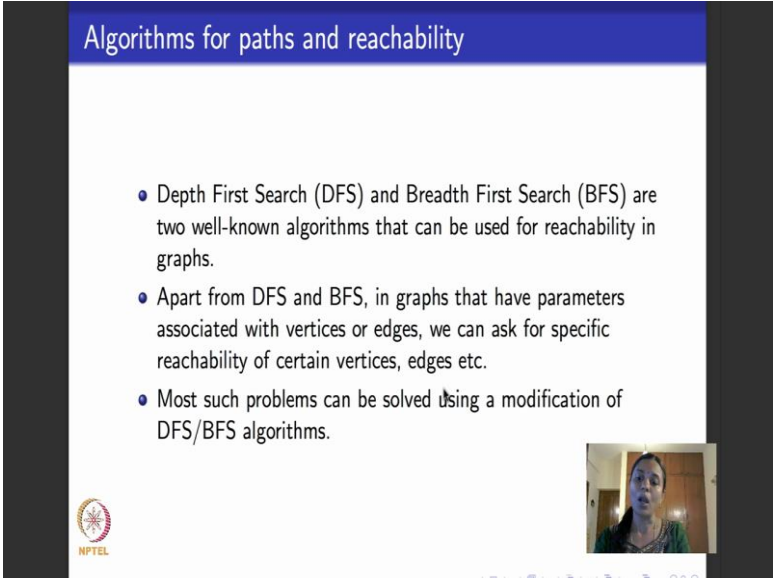
In this there is a sub path which is say  $w \rightarrow x \rightarrow u \rightarrow v$ . That is a sub path,  $u \rightarrow w \rightarrow x$  is another sub path. So, it is just sub sequence in the sequence of vertices that you encounter in the path. So, why are we looking at paths? We will look at paths because we have going to be able to design test cases that we can use these paths to reach a particular statement or a particular node in the graph corresponding to that software artifact. So, we say a particular vertex  $v$  is reachable in the graph if there is a path from one of the initial vertices of the graph to  $v$ . For the sake of reachability, we will consider those paths that begin at initial vertices only because we want to be able to satisfy the RIPR criteria if you remember. RIPR - the first R is reachability. So, reachability means from the inputs from the initial state of the corresponding graph, I should be able to reach a particular vertex and moving on I should be able to propagate the output to a particular vertex right. Those final vertex to which the output is propagated and visible to the user would be one of the final vertices.

So, we say a particular vertex  $v$  is reachable in the graph if there is a path from one of the initial vertices to that vertex  $v$  in the graph. Reachability is not restricted to just vertices you could talk about reachability for an edge also. So, when is an edge reachable in a graph? We say a particular edge is reachable in a graph if there is a path from one of the initial vertices to the beginning vertex of that edge which is  $u$  and moving on, it actually uses that edge to reach  $v$  right. So, there is a path from one of the initial vertices to the edge, to the vertex  $u$  and then it uses the edge  $u \rightarrow v$  to be able to reach the vertex  $v$

then you say that the edge  $e$  which is given by the pair  $u$  comma  $v$  is reachable in the graph.

I hope you know the notion of a sub graph of a given graph. So, what is a sub graph of the given graph? It has a subset of the set of vertices and then it has a subset of the set of edges restricted to only those vertices that occur in the graph. So, you should go back and look at the same example that we had. So, here is a graph, I can think of just this triangular entity right which consist of three vertices  $u$   $x$   $w$  and these three edges as a sub graph of this entire graph. Or, you could just considered just this stand alone vertex  $v$  and another stand alone vertex  $y$  as a sub graph just containing two vertices. So, we can talk about reachability for sub graphs also. So, we say a sub graph  $G$  prime of a graph  $G$  is reachable if any of the vertices in that sub graph is reachable from the initial vertex of  $G$ .

(Refer Slide Time: 22:41)



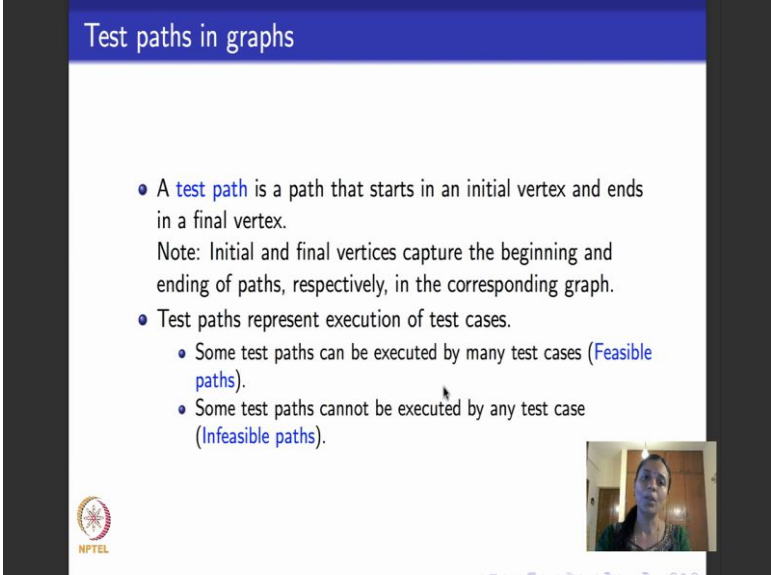
The slide is titled "Algorithms for paths and reachability" in a blue header. It contains three bullet points: "Depth First Search (DFS) and Breadth First Search (BFS) are two well-known algorithms that can be used for reachability in graphs.", "Apart from DFS and BFS, in graphs that have parameters associated with vertices or edges, we can ask for specific reachability of certain vertices, edges etc.", and "Most such problems can be solved using a modification of DFS/BFS algorithms." In the bottom right corner, there is a small video inset showing a person speaking. The NPTEL logo is in the bottom left corner.

- Depth First Search (DFS) and Breadth First Search (BFS) are two well-known algorithms that can be used for reachability in graphs.
- Apart from DFS and BFS, in graphs that have parameters associated with vertices or edges, we can ask for specific reachability of certain vertices, edges etc.
- Most such problems can be solved using a modification of DFS/BFS algorithms.

So, are there algorithms that deal with computing paths and computing reachability of a particular vertex? Of course, all I am assuming all of you know basic graph algorithms in case you do not know please feel free to look up NPTEL courses the deal with design and analysis of algorithms and get to know about graph algorithms. Two basic algorithms that you have to be familiar with what are called breadth first search and depth first search. Most of the test case design algorithms that we will deal with in this course will involve depth first search or breadth first search along with some

manipulations and add on to these algorithms. I will not be able to cover these algorithms because I want to be able to focus on test case design using graphs.

(Refer Slide Time: 23:29)



**Test paths in graphs**

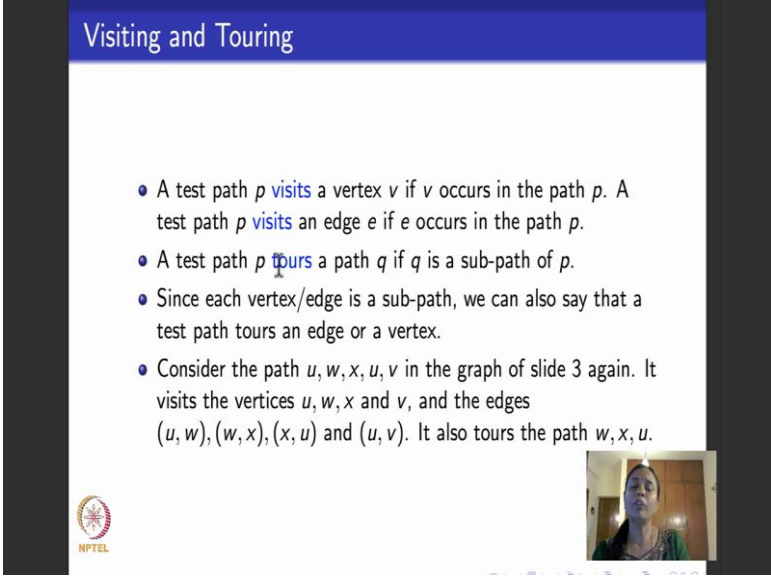
- A **test path** is a path that starts in an initial vertex and ends in a final vertex.  
Note: Initial and final vertices capture the beginning and ending of paths, respectively, in the corresponding graph.
- Test paths represent execution of test cases.
  - Some test paths can be executed by many test cases (**Feasible paths**).
  - Some test paths cannot be executed by any test case (**Infeasible paths**).

The slide also features the NPTEL logo in the bottom left corner and a small video inset in the bottom right corner showing a woman speaking.

So, now instead of looking at arbitrary paths and graphs we will see what are test paths in graph. So, what is a test path a test path as I told you has to begin in an initial vertex to be able to ensure reachability and it has to end in one of the final vertices. So, a test path in a graph is any path that begins in an initial vertex and ends in a final vertex. So, if I go back to the same example, if I see here path of the form  $u \rightarrow x \rightarrow w$  is a test path because it begins in an initial vertex  $u$  and ends in a final vertex  $w$ , whereas path of the form  $u \rightarrow w \rightarrow x \rightarrow y$  is not a test path because even though it begins in initial vertex  $u$ , the vertex that it ends which is  $y$ , is not one of the final vertices. So, for us, test paths will always begin at initial vertex and end at a final vertex.

So, test paths will result in some test cases being executable. Test paths, if some test paths can be executed by test cases then those of called feasible test paths. There could be test paths for which I cannot execute any test case, like for example, there could be a test path which says you somehow reach a piece of dead code or unreachable code. So, I will not be able to write a test case that you can reach the dead code with. So, such test paths will be called as infeasible test paths. We will make these terminologies clear as we move on.

(Refer Slide Time: 25:04)



The slide is titled "Visiting and Touring" in a blue header. It contains a bulleted list of definitions and a video inset in the bottom right corner showing a person speaking. The NPTEL logo is in the bottom left corner of the slide area.

- A test path  $p$  **visits** a vertex  $v$  if  $v$  occurs in the path  $p$ . A test path  $p$  **visits** an edge  $e$  if  $e$  occurs in the path  $p$ .
- A test path  $p$  **tours** a path  $q$  if  $q$  is a sub-path of  $p$ .
- Since each vertex/edge is a sub-path, we can also say that a test path tours an edge or a vertex.
- Consider the path  $u, w, x, u, v$  in the graph of slide 3 again. It visits the vertices  $u, w, x$  and  $v$ , and the edges  $(u, w), (w, x), (x, u)$  and  $(u, v)$ . It also tours the path  $w, x, u$ .

So, few other terminologies we need to be able start looking at algorithms for test case design. Those are the notions of visiting and touring. So, we say test path  $p$  visits a vertex  $v$ ,  $v$  occurs along the path  $p$  right. Similarly, a test path  $p$  visits an edge  $e$  if  $e$  occurs along the path  $p$  right.

A test path  $p$  tours path  $q$  if  $q$  happens to be a sub path of  $p$ . We will go back to the same graphs that we looked at, so here is a test path right  $u \rightarrow x \rightarrow w$ , right. So, this test path visits three vertices  $u \rightarrow x$  and  $w$  and it visits two edges the edge  $u \rightarrow x$  and the edge  $x \rightarrow w$  and it tours a sub path  $w \rightarrow x$ . You could consider another test path which looks like this it could be  $u \rightarrow w \rightarrow x \rightarrow u \rightarrow w$  right. So, this test path visits three vertices  $u \rightarrow x$  and  $w$  it happens to visit them again and again, but basically it visits only three vertices and it tours the sub path  $u \rightarrow w \rightarrow x$ . So, I hope visiting and touring a clear value we will look at what are tests and test paths.

(Refer Slide Time: 26:24)

### Tests and test paths

- When a test case  $t$  executes a path, we call it the **test path** executed by  $t$ , denoted by  $path(t)$ .
- Similarly, the set of test paths executed by a set of test cases  $T$  is denoted by  $path(T)$ .

Test case input:  $(a=0, b=1)$  Test path:  $u, v, x, x$   
Test case input:  $(a=1, b=1)$  Test path:  $u, w, x$   
Test case input:  $(a=3, b=1)$  Test path:  $u, x$

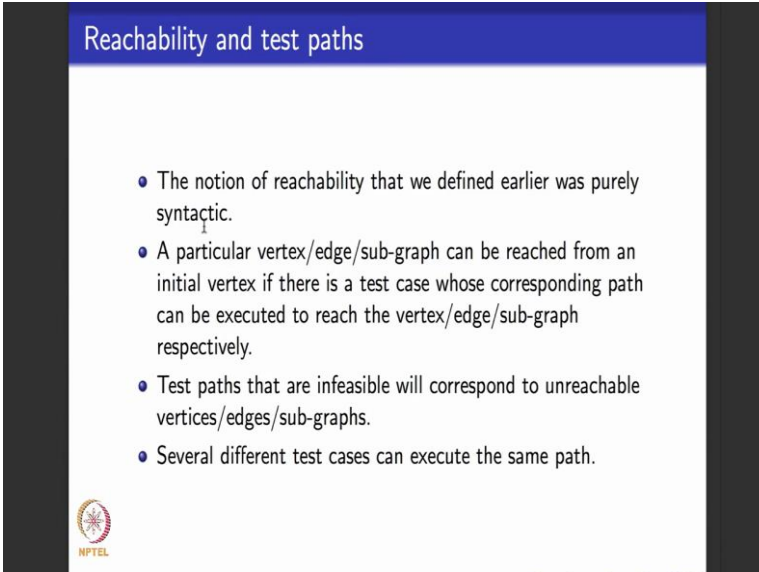
Let us take a small example look at this graph this graph is got four vertices,  $u$ ,  $v$ ,  $w$  and  $x$  and here is an initial vertex  $u$ . It models control flow corresponding to a simple switch statement switch case statement, the switch case statement is at the node  $u$ . There are three switch cases, if  $a$  is less than  $b$  you go to  $v$  execute may be some statements, if  $a$  is equal to  $b$  then you go to  $w$  and do something else if  $a$  is greater than  $b$  then you go to  $x$  and then you do something else and it so happens that computation terminates at  $x$ . So, what suppose I have a test case input  $a$  as 0 and  $b$  as 1 then which is the condition that it satisfies? It satisfies  $a$  less than  $b$  right.

So, which is the test path that this test case executes? This test case executes the path  $u$  to  $v$  and remember test path is one that has to end in a final state right.  $v$  is not one of the final states. Which is a final state in this graph? It happens to be the state  $x$  right. So, I go from  $u$  to  $v$  because my test case satisfies the predicate  $a$  less than  $b$  and I can freely go from  $v$  to  $w$  and then from  $w$  to  $x$  because these two edges and this graph do not have any guards or conditions labeling them. So, if my test case input is  $a$  0 and  $b$  1 then I say that the test path that it takes is  $u$  to  $v$  because it satisfies  $a$  less than  $b$  and then  $v$  to  $w$  and  $w$  to  $x$ . This is clear? So, similarly if my test input is less say  $a$  is 3 and  $b$  is 1 then in this switch case statement the predicate that it satisfies is this:  $a$  greater than  $b$  and then the test path that it takes is just the single edge  $u$  to  $x$  it just. So, happens that this path containing just this one edge already is a test path because it begins at the initial vertex  $u$  and ends at the final vertex  $x$ .



So, similarly when I have set of test cases I can talk about a set of test paths. For each test case in the set of test cases you consider the test paths that they execute and take the union of all the test paths to be able to get the set of test paths corresponding to a set of test cases. Please remember that one test case can execute many test paths. This example does not show that, but one test case can execute. Like for example, if a club these two conditions right and say  $a$  is less than or equal to  $b$  then I can write several test cases write that will execute this path.

(Refer Slide Time: 29:07)



Reachability and test paths

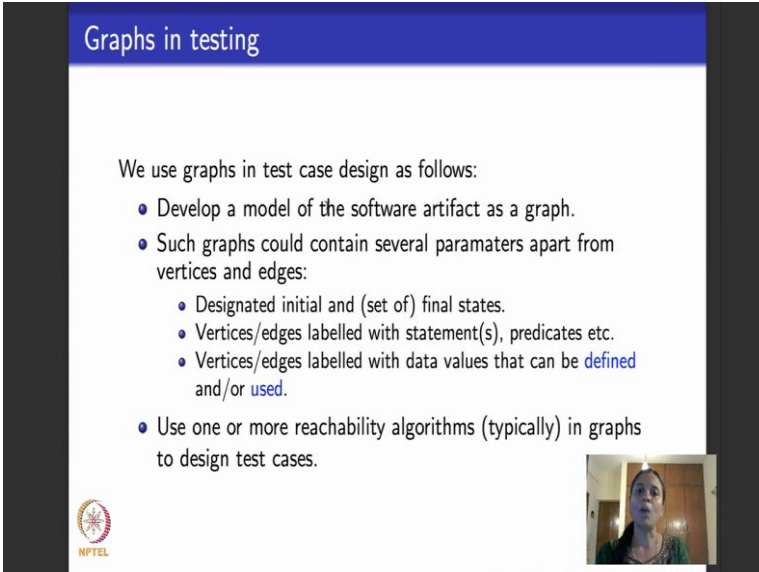
- The notion of reachability that we defined earlier was purely syntactic.
- A particular vertex/edge/sub-graph can be reached from an initial vertex if there is a test case whose corresponding path can be executed to reach the vertex/edge/sub-graph respectively.
- Test paths that are infeasible will correspond to unreachable vertices/edges/sub-graphs.
- Several different test cases can execute the same path.

NPTEL

Now, we will go back and see reachability in the context of test paths. When we define reachability for graphs we defined it in a purely syntactic nature you say a particular vertex  $b$  is reachable iff there is a path from the initial state to one of the vertices right. As you see in this example it need to be purely syntactic because there could be conditions associated with edges, there could be something associated with vertices. As you traverse along a path your implicitly allow to traverse only if that condition a guard is met right. So, in the real application of graphs we would be looking at reachability in the presence of these additional annotations and conditions. So, we will see what reachability means in the presence of these conditions. And it is to be noted that if I have a particular test path that is infeasible, like I told you right a test path that insist that you reach a piece of dead code than it corresponds to vertices or nodes or edges or graphs that are not reachable from the main graph.

Like for example, it might so happen that the control flow graph of a particular program has two disjoint components, in which case suppose I insist that all the initial vertices are in one component, I insist that you reach a vertex from the initial vertex to another vertex in the second component. Because these two are two disjoint components and may not be able to reach that vertex at all and I say that it represents an infeasible test requirement. So, we will see several examples of where infeasible test requirements could occur when we look at graph models. So, how are we going to use graphs for test case design?

(Refer Slide Time: 30:44)



**Graphs in testing**

We use graphs in test case design as follows:

- Develop a model of the software artifact as a graph.
- Such graphs could contain several parameters apart from vertices and edges:
  - Designated initial and (set of) final states.
  - Vertices/edges labelled with statement(s), predicates etc.
  - Vertices/edges labelled with data values that can be **defined** and/or **used**.
- Use one or more reachability algorithms (typically) in graphs to design test cases.

NPTEL

*(A small video inset in the bottom right corner shows a woman speaking.)*

We will develop a model of a software artifact as a graph, I already showed you to examples of how to take a code snippet and write a control flow graph corresponding to that. It is to be noted as we saw in that example that these graphs apart from vertices and edges could contain several different annotations labels. So, here are some examples some of the vertices could be initial and final vertices it could have a labels of statements predicates associated with its vertices and edges as we saw in those two examples. In addition to that, it could have data values data values for variables that are defined at particular statement, data values for variables that I used at a particular statement. Such graphs are called data flow graphs I have not shown you an example in this slide, but in later lectures we will see what data flow graphs look like.

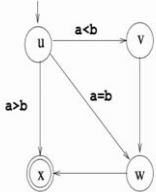
But what is to be remembered is that they will typically, graphs that model software artifacts will always have some kind of labels associated with their vertices or edges

right. So, we want to be able to use one or more reachability algorithms to be able to design test cases for these kind of graphs.

(Refer Slide Time: 31:57)

Graph coverage criteria

- **Test requirement** describes properties of test paths.
- **Test Criterion** are rules that define test requirements.
- **Satisfaction:** Given a set  $TR$  of test requirements for a criterion  $C$ , a set of tests  $T$  satisfies  $C$  on a graph iff for every test requirement in  $t \in TR$ , there is a test path in  $path(T)$  that meets the test requirement  $t$ .
- For example, the set of test cases below in the graph satisfy **branch coverage** at the node  $u$  in the graph.



Test case input: (a=0, b=1) Test path: u, v, x, x

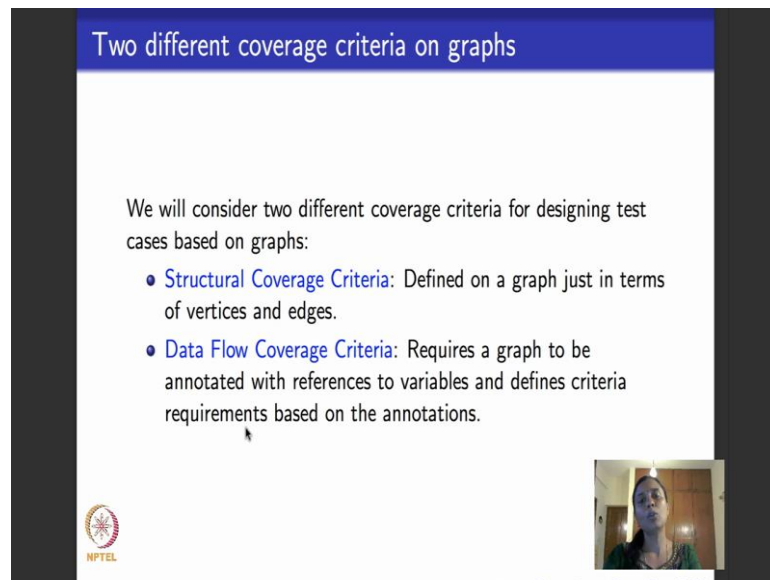
Test case input: (a=1, b=1) Test path: u, w, x

Test case input: (a=3, b=1) Test path: u, x

So, we will read a find what test requirement what test criteria are now specific to graphs. So, what is a test requirement? It just describes a property of a test path, test path as it corresponds to in a graph. What is a test criteria? Test criterion is set of rules that define the test requirement. Then, for example, if you take this graphs that we looked at earlier the test criterion that it describes is to say do branch coverage on the vertex  $u$  means the degree of vertex is the out degree of the vertex  $u$  is 3, there are three branches going out of  $u$ . Write test cases to cover all three branches that is what we did here we do not test cases to cover all three branches.

So, what is satisfaction? We see a particular set of test requirement satisfies a coverage criteria  $c$  if the test cases that are right for that test requirement satisfy all the test paths that need the requirement. Like for example, if my coverage criteria says branch coverage at mode  $u$  then, these three test cases right completely achieve branch coverage for this particular mode  $u$ .

(Refer Slide Time: 33:10)



Two different coverage criteria on graphs

We will consider two different coverage criteria for designing test cases based on graphs:

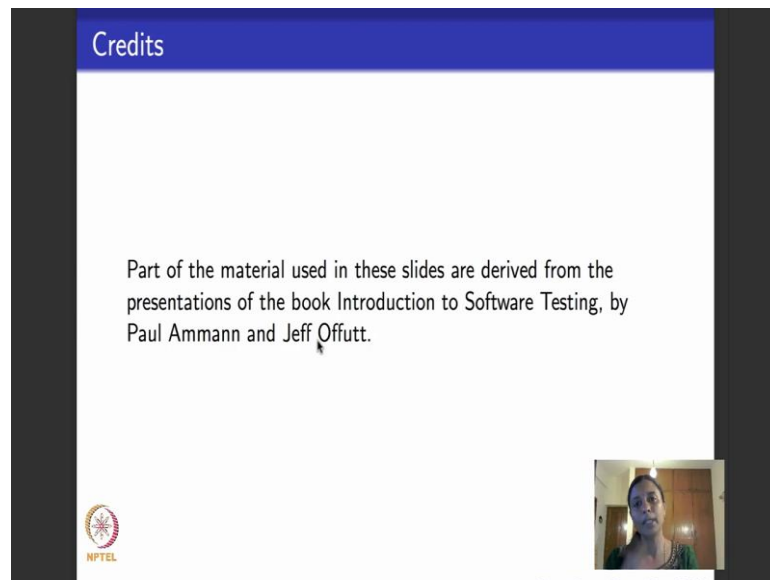
- **Structural Coverage Criteria:** Defined on a graph just in terms of vertices and edges.
- **Data Flow Coverage Criteria:** Requires a graph to be annotated with references to variables and defines criteria requirements based on the annotations.

NPTEL

So, moving on we will look at two kinds of coverage criteria related to graphs. The first coverage criteria that we will be looking at is what is called structural coverage criteria where we will define coverage criteria purely based on the vertices and edges in the graph.

They could be annotated with statements and so on, but we would not really use the what the statements are or the other annotations to be able to define the coverage criteria. We will purely define in terms of just vertices and edges. The next kind of coverage criteria would be data flow coverage criteria where we again look at graphs that are annotated with variables and so on and we will define coverage criteria based on the annotations, based on the variables and the values that they define.

(Refer Slide Time: 34:02)



So, in the next module I will begin with structural coverage criteria and walk you through algorithms and test case design for achieving various kinds of structural coverage criteria in graphs.

Thank you.