Software Testing
Professor Meenakshi D'Souza
Department of Computer Science and Engineering
Indian Institute of Technology, Bangalore
Lecture 59
Basics of Graphs

(Refer Slide Time: 0:18)

Basics of Graphs

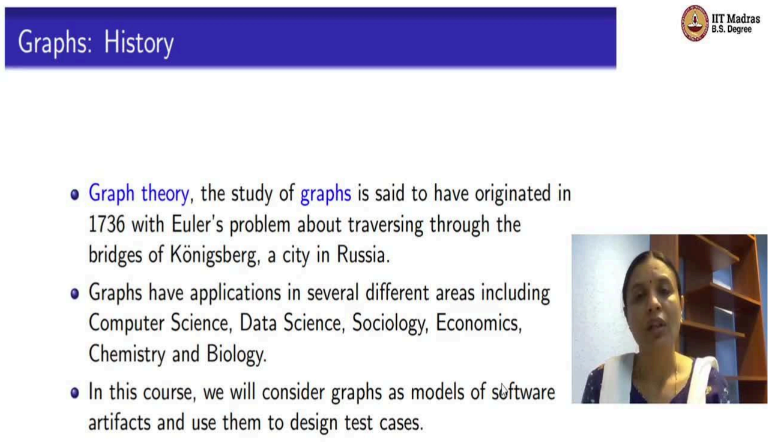
Meenakshi D'Souza

International Institute of Information Technology
Bangalore.

Hello, everyone. Welcome to week 2. This is the first lecture of week 2. As I told you last week, we are going to use a lot of standard data structures and other entities from computer science and elementary mathematics for designing test cases. One such data structure that we would use extensively throughout this course is that of graphs, we are going to learn several algorithms that will design test cases based on graphs.

So, to make this course complete, what I will do is introduce you to those entities and data structures in a brief way, as it is relevant for software testing, as and when we move on in the course. So, in this module, we are going to learn about the basics of graphs as it is used in software testing. Later this week, we will learn some graph algorithms. If you have already done them, there will be a good recap for you, and then see how we can get started with using graphs for software testing.

(Refer Slide Time: 1:25)



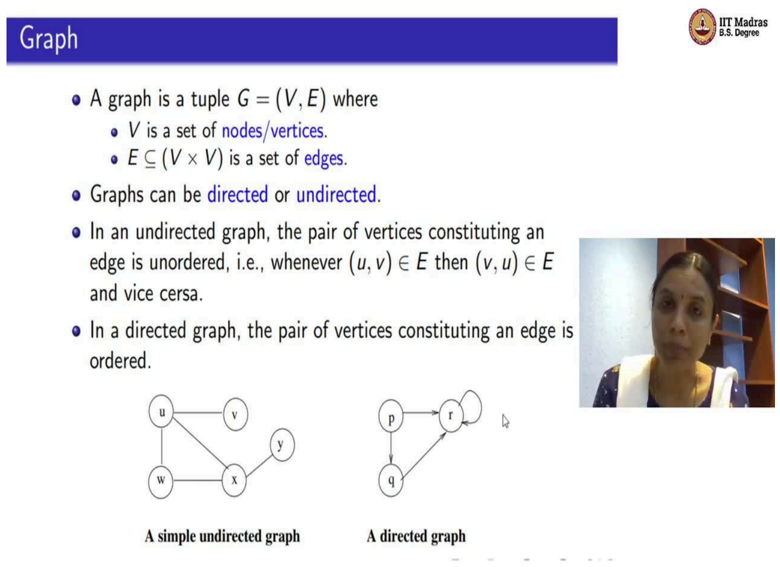
Graphs: History

- 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, Data Science, Sociology, Economics, Chemistry and Biology.
- In this course, we will consider graphs as models of software artifacts and use them to design test cases.

Graphs are a very ancient data structure. Graph theory is said to have originated in the year 1976. With this mathematician called Leonard Euler, who is believed to be the father of graph theory. It so happened that there was the city in Königsberg, which is modern day Russia, which had 7 bridges. And the problem was about can somebody traverse through all the 7 bridges in such a way that they visit each bridge exactly once. Euler later showed that this is not possible.

And this is how graphs are believed to have been invented. Now, graphs are an important entity not only in computer science and data science, but they are also extensively used in several other fields, including sociology, economics sciences, including chemistry and biology. In this course, we will not learn a lot about graphs, but we will learn the basics as we would need for software testing.

(Refer Slide Time: 2:29)



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 cersa.
- In a directed graph, the pair of vertices constituting an edge is ordered.

A simple undirected graph

A directed graph

So, what is a graph, it should not be confused with what we write on a graph paper with an x-axis and y-axis. Those are plots. A graph is an entity that looks like this picture that you see here on the bottom left or an entity that looks like this picture that you see on the bottom right. So, a graph has 2 kinds of components.

One represented by the set v is the set of nodes or vertices. Those are these circular entities that you find in these 2 figures here, and the other one, represented by the set E, which is basically a subset of $V \times V$, that is the Cartesian product of the set v, and v, the set of vertices is the set of edges.

So, if you see in this graph, on the left, there are 5 vertices, u, v, w, x, and y, we have just named them, you can also name them as 1, 2, 3, 4, 5 or A, B, C, and so on, just give some name to the set of vertices, just so that we know which is which vertex. In the graph that you see on the right, there are 3 vertices p, q, and r.

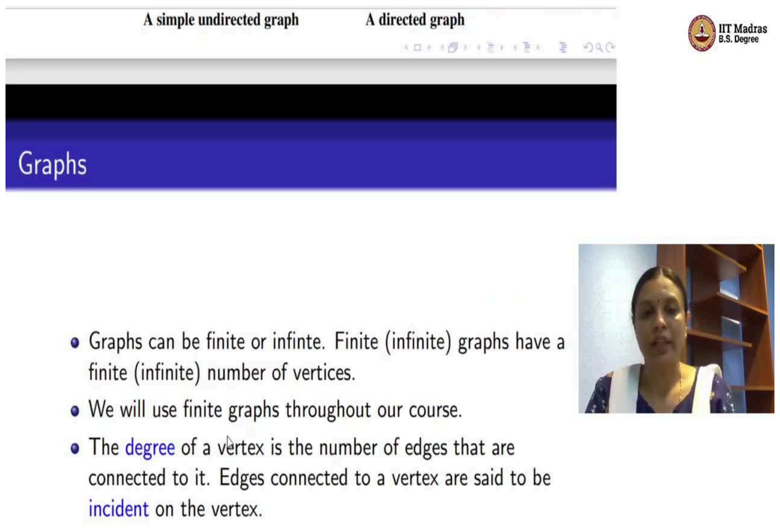
Again, we have named them as p, q, and r. What are edges? In the graph that you see on the left-hand side, let us count the number of edges, there is one edge from vertex u to vertex v represented by this line, horizontal line that you see going from u to v, there is one edge from vertex u down to vertex w in this vertical line, there is one more edge going horizontally from w to v, there is one edge from u to v, there is one more edge from x to y.

If you see this graph has no edge from vertex v to vertex y, no edge from u to y, no edge from w to x. So, the set of edges e is only a subset of the set of vertices v, is this clear? So, based on whether we attach a direction to an edge or not, a graph can be directed or undirected. The one graph that is there down here on the left is an undirected graph. Why? Because there are no arrows or directions attached to edges. Whereas if you see this graph on the right, it is a directed graph which means how do I interpret it as there is an edge from vertex p to vertex r, and it has a direction it is in the direction from p to r. So, there is no edge from r back to p, is that clear?

Similarly, there is an edge from p 2 q or there is a directed edge from p to q, but there is no edge from q back to p this kind of edge that you see, that is sticking itself to the vertex r is a special kind of edge, we call them as self-loops or loops. So, it is an edge from r to itself. Is this clear? So, directed graph edges have directions in there, which means edges are an ordered pair of vertices. In an undirected graph edges represent an unordered pair of vertices.

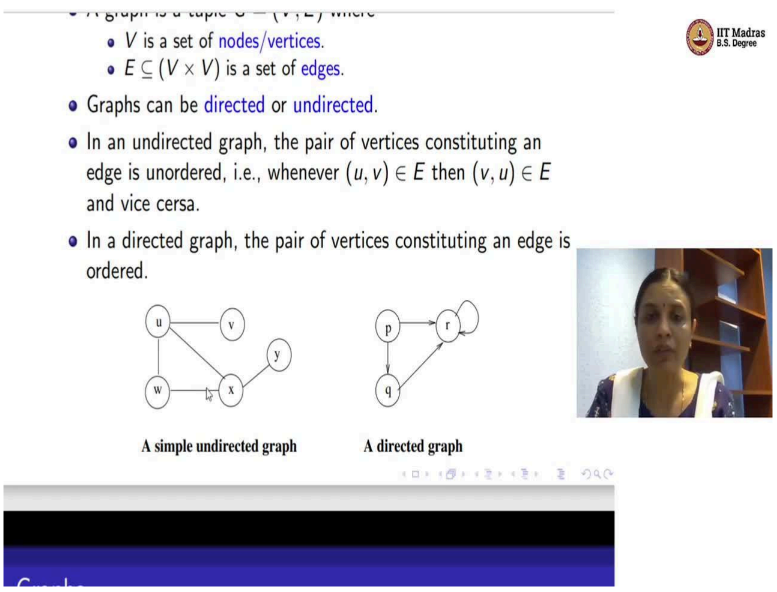
So, for example, if I take this graph on the left, there is an edge from u to v, it could be interpreted as the edge from u to v or the edge from v back to u, it does not matter. Whereas here on the directed graph, if I say there is an edge from p to r, it is unidirectional that direction is from p to r, is this clear? So, the edge is ordered in a directed graph has a direction, edges is unordered in an undirected graph, has no direction.

(Refer Slide Time: 6:28)


A simple undirected graph A directed graph

Graphs

- Graphs can be finite or infinte. 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.



- 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 cersa.
- In a directed graph, the pair of vertices constituting an edge is ordered.

A simple undirected graph

A directed graph

Graphs can also be finite or infinite, the graphs that we saw in the previous slide, these ones here are finite, because for finite graphs, the number of vertices should be finite number, both have finite number of vertices. Infinite graph having finite number of vertices, theoretically, they are useful not for the purposes of software testing.

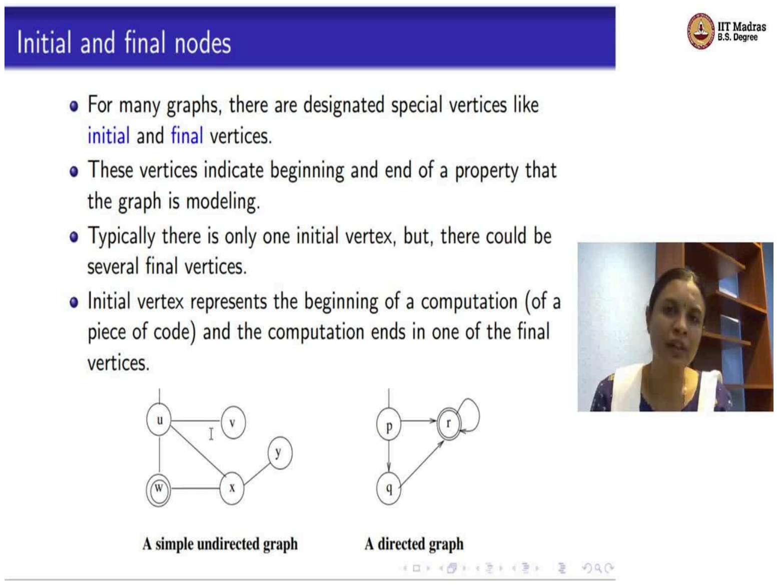
So, we will use finite graphs in this course. One more concept, simple elementary concept that you have to remember in graph theory is that of a degree of a vertex, degree of a vertex basically counts the number of edges that are connected to it. So, let us go back to the previous slide.

Let us take this graph, let us take the vertex u, how many edges come out from u, there is one from u to v, one from u to x and one from u to w. So, the degree of the vertex u is 3, is this clear? The degree of vertex y similarly is 1 degree of vertex v is 1 degree of vertex x is 2, and so on.

For directed graphs, the degree gets split as in degree and out degree, like for example, the out degree of the vertex p, which counts the number of edges that go out of the vertex p is 2, because there is 1 directed edge from p to r, 1 directed edge from p to q, the in degree of vertex p is 0, the in degree of vertex r is how much? 3 because there is 1 edge coming from p to r, 1 edge coming from q to r and one edge from r into itself, the out degree of the vertex r is 1. So that is the notion of a degree, we say an edge from u to v is incident on the vertex u and is also incident on the vertex v.

Degree of a vertex is the number of edges incident to a vertex. For a directed graph degree gets split as in degree versus out degree. That is what this slide has it says the degree of a vertex is the number of edges that are connected to it, it is connected to a vertex are set to be incident on the vertex.

(Refer Slide Time: 8:45)



Initial and final nodes

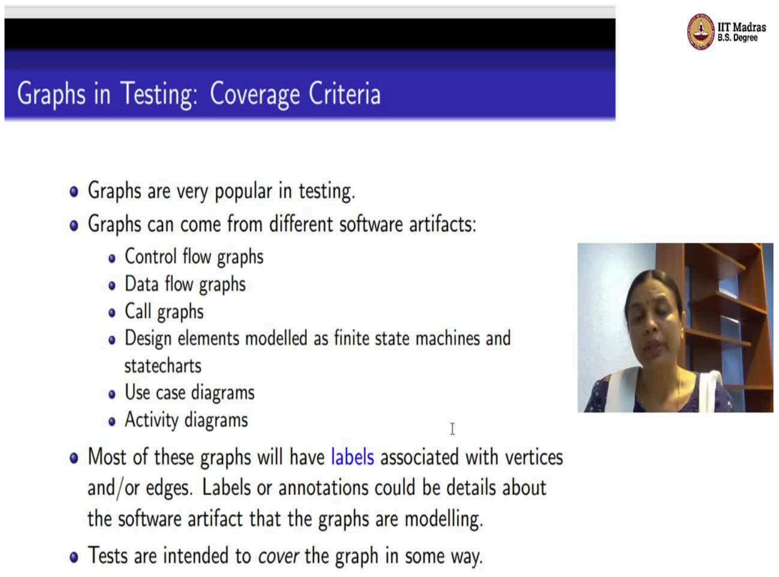
- For many graphs, there are designated special vertices like initial and final vertices.
- These vertices indicate beginning and end of a property that the graph is modeling.
- Typically there is only one initial vertex, but, there could be several final vertices.
- Initial vertex represents the beginning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

Then, one more notion this is very specific to areas like testing, in general graphs need not have designated nodes that have special properties. But typically, when we consider graphs in testing, we consider certain nodes as initial nodes and certain other nodes or vertices as final nodes or vertices. Basically, initial vertices or initial nodes indicate that something begins from the maybe the computation of a method or a program begins from that. So, it is marked like this by an incoming arrow. Like here in this graph, u is an initial vertex, and in the graph on the right, p is an initial vertex.

For a final node, it represents something ending there. So, for example, for this graph, w is a final node is marked by this double circle. In the graph on the right, r is a final node, it is marked by this double circle bottom. So, it means some computation or something is ending their initial vertices and final vertices are designated vertices, where typically from an initial vertex a computation begins and from a final vertex a computation ends.

Typically, there is only 1 initial vertex because many software is so called deterministic so we do not have 2-3 places from which a software can start, it always starts from 1 statement, that could be an initial vertex. And it could end in many of the several places that are possible for it to end based on the execution that it takes. So, there could be more than 1 final vertex. Our graph models for software artifacts that we use for testing will typically have initial and final vertices.

(Refer Slide Time: 10:30)



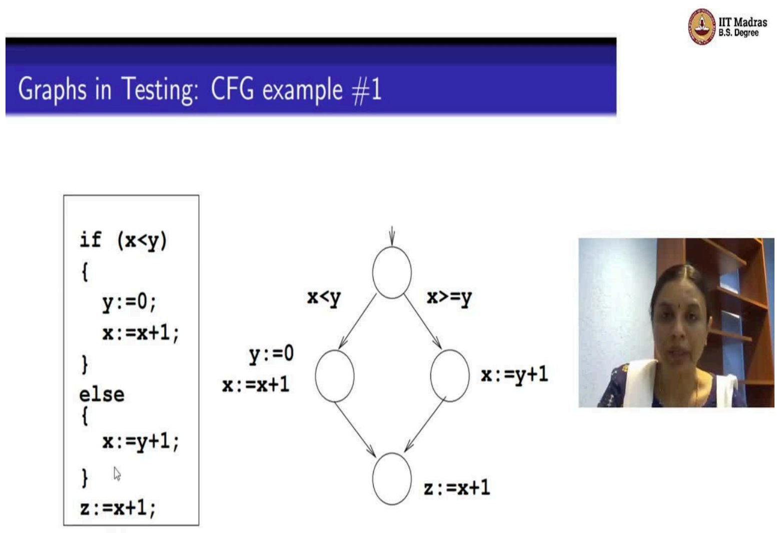
Graphs in Testing: Coverage Criteria

- Graphs are very popular in testing.
- Graphs can come from different software artifacts:
  - Control flow graphs
  - Data flow graphs
  - Call graphs
  - Design elements 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 why do we need? Why are we studying about graphs? Why do we need them in testing? As I told you graphs are one of the most popular data structures used in testing, you will learn how to use them and we will learn several different kinds of graphs for testing, we learn what we call control flow graphs, data flow graphs, graphs that represent call interfaces between methods, call graphs, graphs corresponding to UML diagrams, like state machines, state charts, use cases, activity diagrams, and so on and so forth.

Of course, these graphs are not going to look as simple as this with some abstract numbers or letters, labels and edges having nothing in them, they will always have several different annotations that represent various properties of software artifacts, these annotations will come as labels that are associated with vertices or edges, we will learn as we move on. And we are going to design test cases that are meant to cover some property of the graph or the other, which in turn means covering some property of the software module under test.

(Refer Slide Time: 11:39)



Graphs in Testing: CFG example #1

```
if (x<y)
{
  y:=0;
  x:=x+1;
}
else
{
  x:=y+1;
}
z:=x+1;
```

So, here is one such graph. Focus on this code snippet on the left, it is just a fragment of code does not represent full executable code, represents some code of an if statement. So, it says if x is less than y, then you execute these 2 statements y equal to 0 x is x plus 1 else, you execute the single statement, x is y plus 1. And then you say after this if is executed, z is x plus 1. So, it is just a fragment of code. So, for this fragment of code, suppose I have to draw the control flow graph, which is abbreviated as CFG in this title, how will it look like?

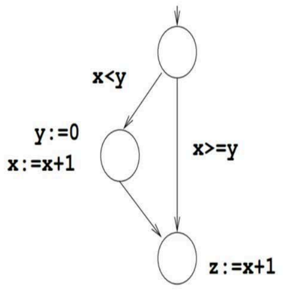
It looks somewhat like this graph that you see on the right, so this point here that has this incoming arrow represents this if statement, this if statement is represented by this node, this node. And then there is a decision point, it checks for the truth value of x being less than y, if x is less than y, it takes the left edge and goes to a node, which represents the 2 statements inside the then part y is assigned 0 x is assigned x plus 1.

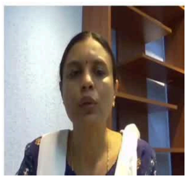
If x is not less than y, that is, if x is greater or equal to y, or the condition of if is false, then it takes the else part and it goes to this statement, where x is equal to y plus 1. So, that is this node on the right, whatever it is, it is 1 of this will be executed. And after this, the last statement z is equal to x plus 1 will be executed. We will learn this more thoroughly properly. This is just meant to be an illustrative example for you.

(Refer Slide Time: 13:21)

Graphs in Testing: CFG example #2

```
if (x<y)
{
  y:=0;
  x:=x+1;
}
z:=x+1;
```

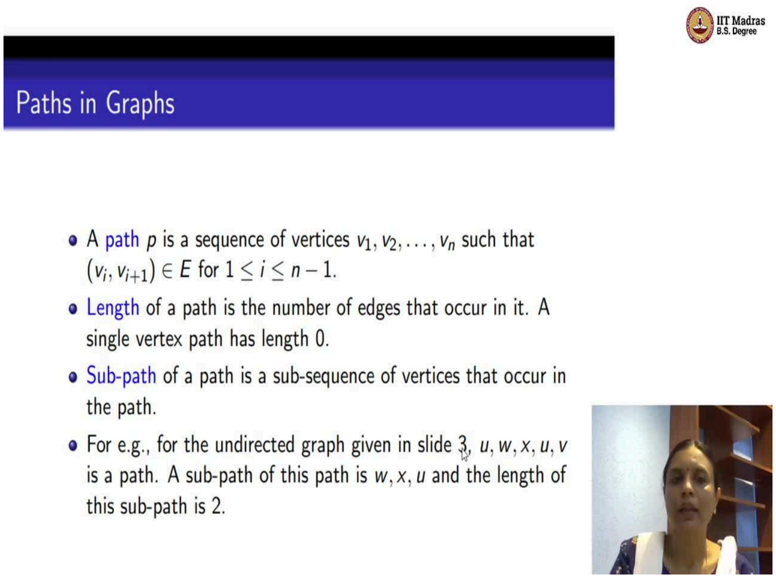




Here is the second example, suppose we had an if statement without an else part, so if x is less than y, then you do this y equal to 0 x is equal to x plus 1, get out of the if and then execute the statement z is equal to x plus 1. How does the control flow graph for that look like? It looks like this graph on the right. So, there is an entry point, there is only one part of the if statement, which checks if x is less than y, this condition here, which corresponds to this condition here. If it is true, then it goes ahead and executes these 2 statements.

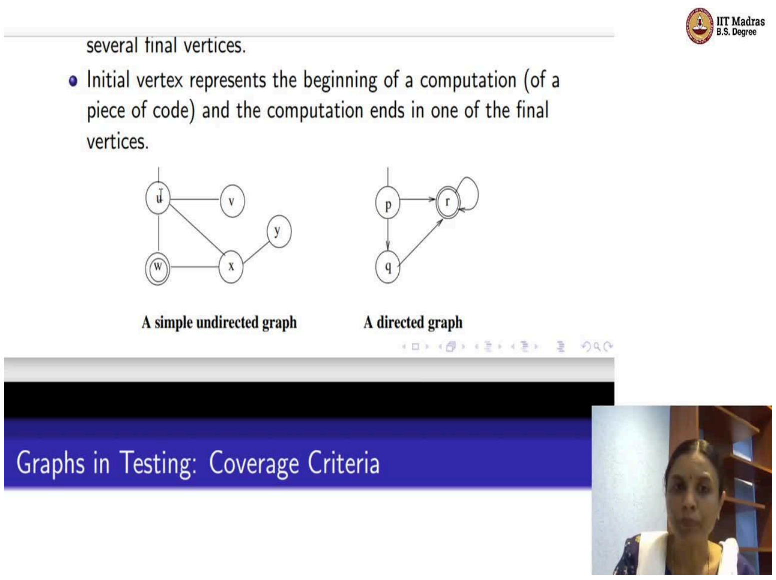
They are clubbed and put together in this 1 node. And after it finishes executing, or even if the if statement is false, it goes to this last statement, which is z is equal to x plus 1 that is represented by this node here below. Is this clear? So, this is just an example. As I told you, do not worry, we will see the same thing in detail a little later.

(Refer Slide Time: 14:16)



Paths in Graphs

- A path p is a sequence of vertices $v_1, v_2, \ldots, v_n$ 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.



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

Graphs in Testing: Coverage Criteria

So, a few more concepts from graphs that you would need for the purposes of software testing. So, 1 notion is that of a path so how do you define a path, take a graph, by now you know what it is, start from some vertex, put yourself in as standing in some vertex or keep a pebble there in one vertex, then take an edge that goes out of that vertex, where will you reach?

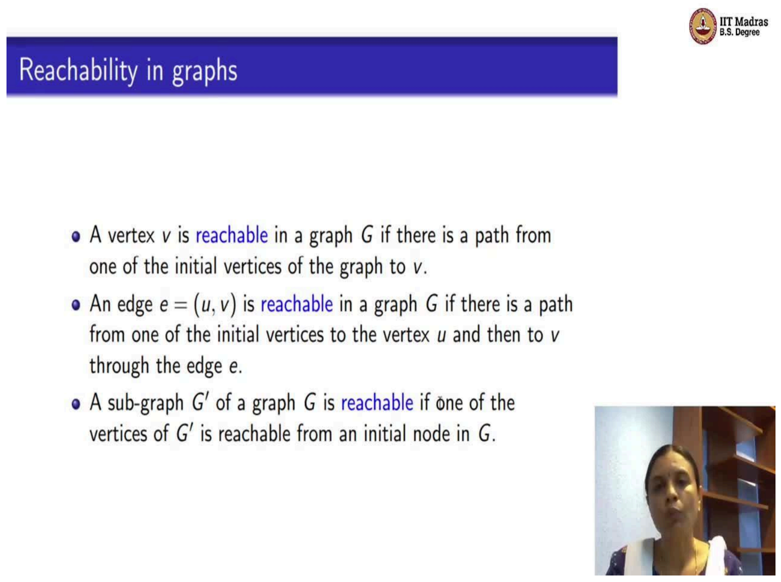
You will reach the second vertex, some vertex that the edge is going into, then you are in that vertex, then take another outgoing edge from that vertex, you will reach the third vertex and then take another outgoing edge from that vertex, you will reach the 4th vertex, that is all you do. So a path is basically a sequence of vertices. Let us say v1 v2 and so on up to vn, such that 2 consecutive vertices at any point in time, that is v1 and v2, v2 and v3 in general vi and vi plus 1 are related by an edge. So, let us go back a little top.

And maybe we will look at this graph. So, can I trace out a path in this graph, so I can start at vertex u take the edge from u to x, go to x, then take the edge from x to y, go to y, and stop, because there is nothing going out of y I have to stop at y the path ends there. Alternately, I can start at let us say v, vertex v take the edge only outgoing edge from v to u reach u go to w, reach w, then take this edge w to x, go to x, and then maybe take this edge x to y, go to y and stop. So, that is another path. So, I hope you understand what a path is.

So, a path is a sequence of vertices such that I can hop from one vertex to the other in the sequence in order by using edges that connect those vertices in the graph. I also need the notion of a length of a path, which basically counts the number of edges in the path for length of a path, we do not count the number of vertices, we count the number of edges, you have to remember this.

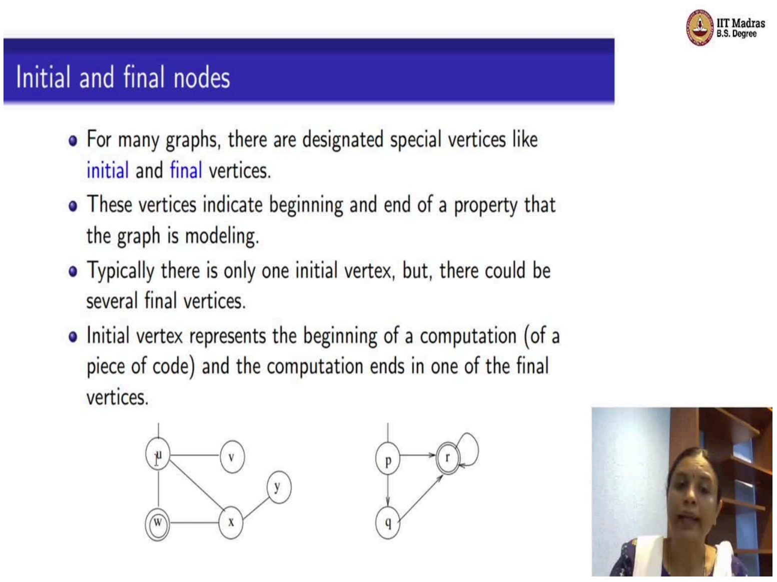
A path can have smaller paths inside them set smaller path are called sub paths, like you have a set and a sub set you can have a sub path of a path, which is basically a sub sequence of vertices, which is a part of that path which occurs is a contiguous sequence. So, in that same example, there is a path which is u, w, x, u, v. Vertices can repeat in the path, no problem, and it gives a sub path, so this is another notion that we will need.

(Refer Slide Time: 17:05)



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.



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.

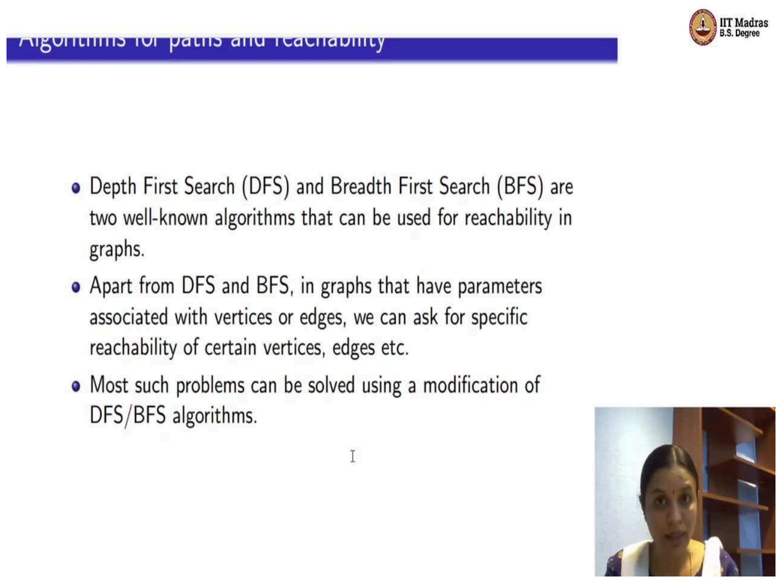
Then there is a notion of reachability. So how do I say some vertex is reachable from some other vertex? I say one vertex is reachable from the other vertex, if there is a path connecting both of them. Simple enough, right? So, let us go back and look at the same example. So, moving back and forth in slides.

So here, I can say, vertex y is reachable from vertex u. Why? Because there are several paths that connect u to y, I can go from u to w, w to x, x to y, I can go from u to x, x to y, that is fine. But in the graph on the right, suppose I asked you the question is vertex q reachable from vertex r? I will repeat is vertex q reachable from vertex r?

The answer will be no, because if you see there is no path that connects r to vertex q. But suppose I asked some other questions. Suppose I say is vertex r reachable from vertex p? What is your answer? The answer is yes, because I can find 2 parts one is the direct edge from p to r, which is also a path, the next is the edge from p to q and q to r, is this clear? So, that is the notion of reachability. So, let me go back to the slide on reachability. So, we say a vertex v is reachable in a graph, if there is a path from one of the initial vertices of the graph to v.

This initial vertex is just an extra condition, you can say that a vertex v is reachable from a vertex u, that is also fine. So, just as we talk about reachability of vertices, we can also talk about reachability of edges. So, we say an edge is reachable in a graph, if there is a path from one of the initial vertices to the vertex u and then to v, the edge connects u to v fine. Then the notion of a sub graph, sub graph is just a smaller graph of a graph. So, you can talk about an entire sub graph being reachable from an initial node. If one of the vertices in the sub graph is reachable, we may not use reachability of sub graphs, but we will use reachability of vertices and reachability of edges, fine.

(Refer Slide Time: 19:32)



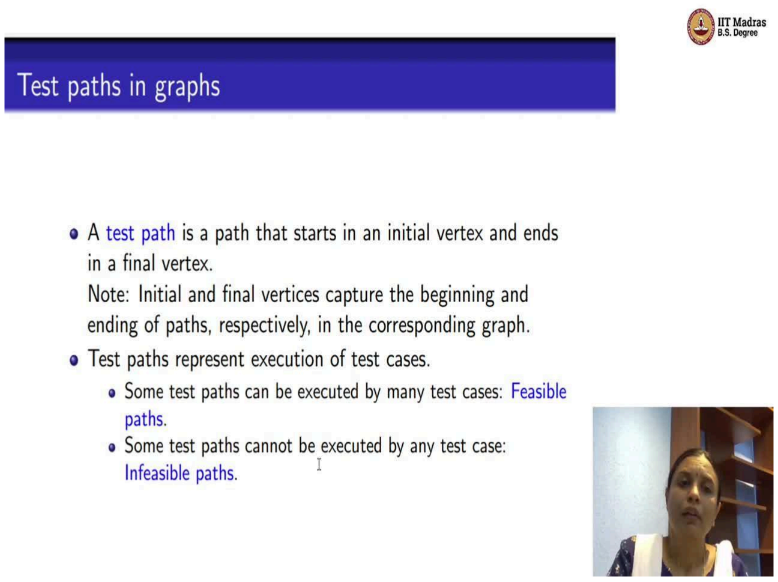
Algorithms for paths and reachability

- 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, we now understand what is reachability? How do I know whether a particular vertex is reachable or not? Are there algorithms to solve for reachability? Yes, there are. There are 2 popular basic algorithms that serve as a backbone for several reachability problems related to graph theory. One is Depth First Search the other is Breadth First Search, Depth First Search abbreviated as DFS, Breadth First Search abbreviated as BFS.

So, we will use BFS and DFS as well-known algorithms that can be used to talk about reachability properties in graphs, so we will use them for testing also, because we are going to use them for testing, irrespective of whether you have done it in the past or not. In the next module, I will reteach Depth First Search and Breadth First Search from a standard textbook. Most reachability problems can be solved using variants of Depth First Search or Breadth First Search. So, it is useful to know these 2 algorithms fresh as a part of this course. We will do it in the next module.

(Refer Slide Time: 20:30)



Test paths in graphs

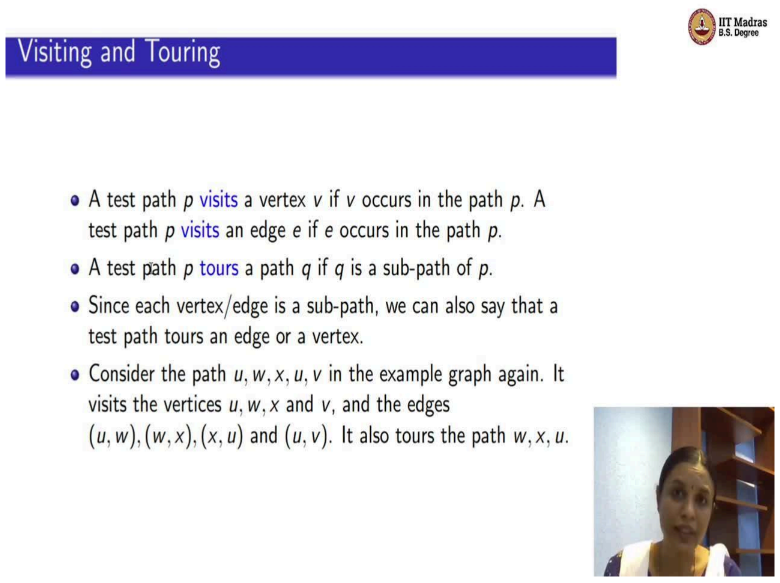
- A test path is a path that starts in an initial vertex and ends in a final vertex.
  Note: Initial and final vertices capture the beginning and ending of paths, respectively, in the corresponding graph.
- Test paths represent execution of test cases.
  - Some test paths can be executed by many test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

A few more concepts in graphs before we wind up for today. For this module, the next one is notion of a test path, we saw what a path is? Path can start from any vertex and end at any vertex, it can only be of length 1, in which case the path is just 1 edge. A test path always starts from one of the designated initial vertices and ends at the final vertex.

So, a test path is a path that starts in an initial vertex and ends in a final vertex. Intuitively, it makes sense to start in an initial vertex, because that is where computation of your software module under test is going to begin, it also makes sense to end in a final vertex, because that is where the computation of your software module under test is going to end, is this clear?

Test path always will come, because you are executing some test case, we will make all these things clear. So, if I get the test path as a result of executing a test case, then the test path becomes a feasible test path. Sometimes somebody might have a goal of, can you illustrate this test path for me? But you will realize that you will not be able to find any test case that will result in such a test path. In that case, we will say that the test path is infeasible. We will see examples of all of these as we move on in the course.

(Refer Slide Time: 22:03)



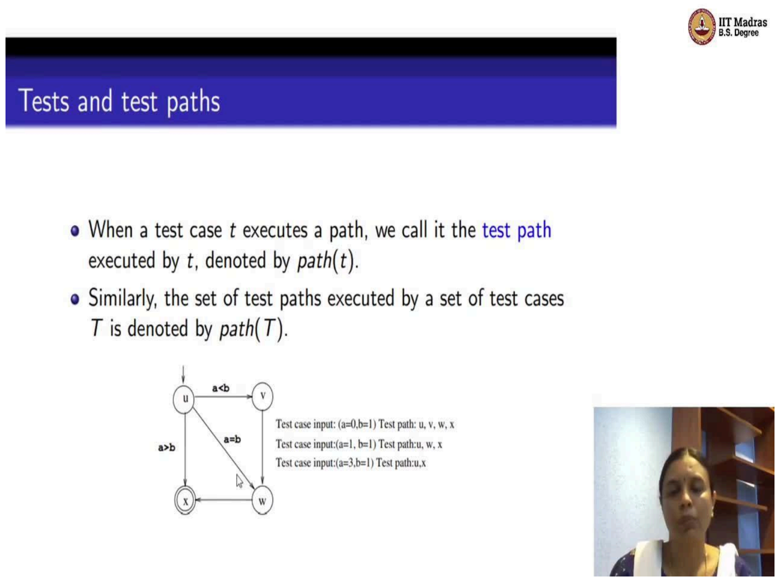
Visiting and Touring

- 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 example graph 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.

Then there is notion of when it comes to test path, in software testing, we distinguish subtle notions about visiting and touring. In English, it might be the same thing, but in testing, as far as graphs are concerned, we distinguish them in some way. So, here is what the definition says. So, we say a test path p visits a vertex along the way, if v is one of the vertices that occur in the path, a test path p visits an edge e, if e is an edge that occurs in the path, it just comes as a part of the path, a test path p tours a sub path q, if q is a sub path of p, is this clear?

Strictly speaking, every vertex is sub path, every vertex that occurs in a path is sub path, trivial sub path of that path, every edge that occurs in a path is also a trivial sub path of that path. So, touring and visiting are one and the same for vertices and edges. Is this clear? So, this example, just talks about the same thing, you take the graph that we saw earlier, we saw that there was a path u, w, x, u, v. So, if you see these visits, how many vertices what are the vertices that occur in the path vertex u, w, x and v. It also visits the edges u w, w x, x u and u v and it tours a sub path for example, w x u it could also tour the sub path x u v and so, is this clear?

(Refer Slide Time: 23:50)



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 $path(T)$.

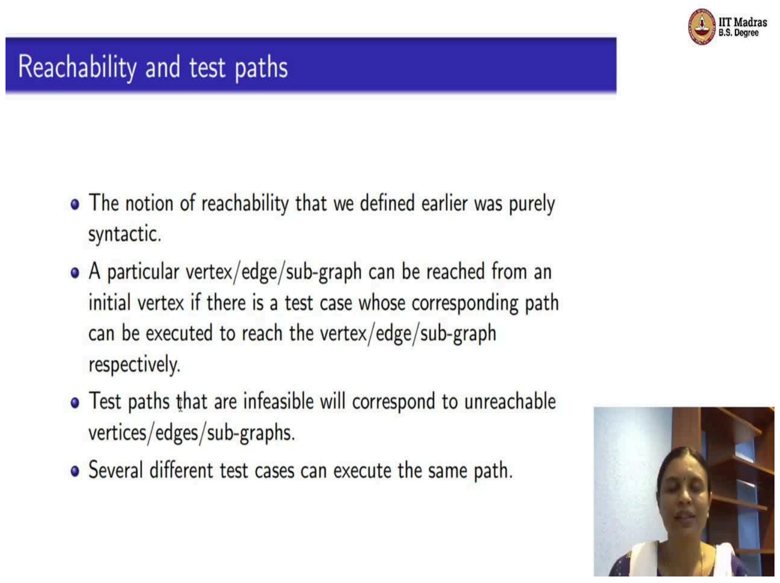
$$\begin{tikzcd}
\arrow[d] & \\
u \arrow[r, "a<b"] \arrow[d, "a>b"'] \arrow[dr, "a=b"] & v \arrow[d] \\
x & w \arrow[l]
\end{tikzcd}$$

Test case input: (a=0,b=1) Test path: u, v, w, 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, the last notion, so, when a test case that we give executes a test path, we call the test path executed by path of t. This is another fictitious example, let us say you have a graph that looks like this and node one represents some… I mean node u represent some kind of branching, this is when a is less than b, this is when u to w is when a equal to b, u to x is when a is greater than b.

So, for example, you could give test case values for a and b, so suppose you give some value where a less than b is true, then it might take the path u, v, w, x which is the first one a0 and b1 suppose you give some value where a is equal to b say both are 1 then it will take the path u to w to x. Test path always begin at an initial vertex and end at a final vertex please remember that. Suppose you give a test case value where a is greater than b. Then it will take the test path u to x, it is just a trivial path with just 1 edge in it.

(Refer Slide Time: 24:59)



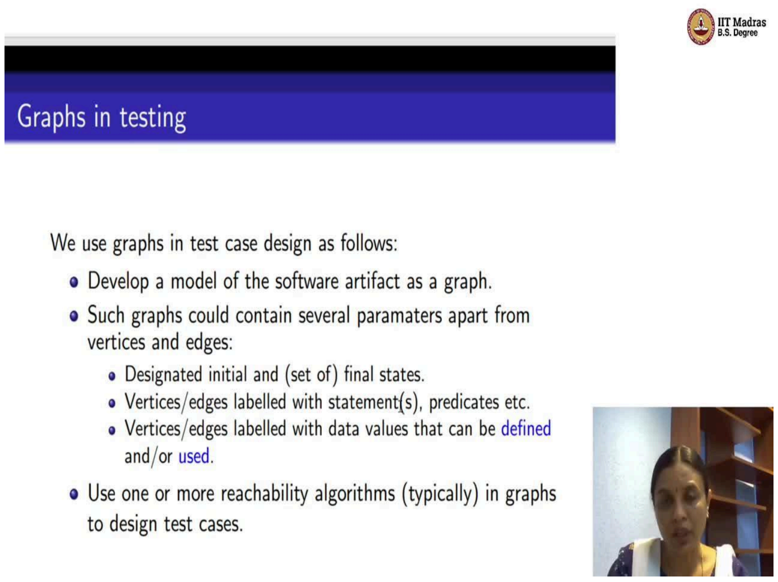
Reachability and test paths

- 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.

So, what we defined reachability for which we will see algorithms in the next lecture was purely syntactic in the sense that we do not know what the underlying graph was we just take the graph, we do not know where it came from. Then we talk about reachability. When it comes to testing, we will worry about reliability by using test cases as I told you please remember the notions of observability and controllability that I told you.

Observability means coming from the initial vertex to a particular vertex, controllability is from that particular vertex going to a final vertex because initial vertices represent places in the code where you can give inputs, final vertices represent places in the code where you can observe the outputs that are produced. So, we will use reachability from the point of view of test cases executing as test parts and consider feasible test parts. The ones that are infeasible will be unreachable for us. All these things we will see for specific graphs as we move on.

(Refer Slide Time: 26:03)



Graphs in testing

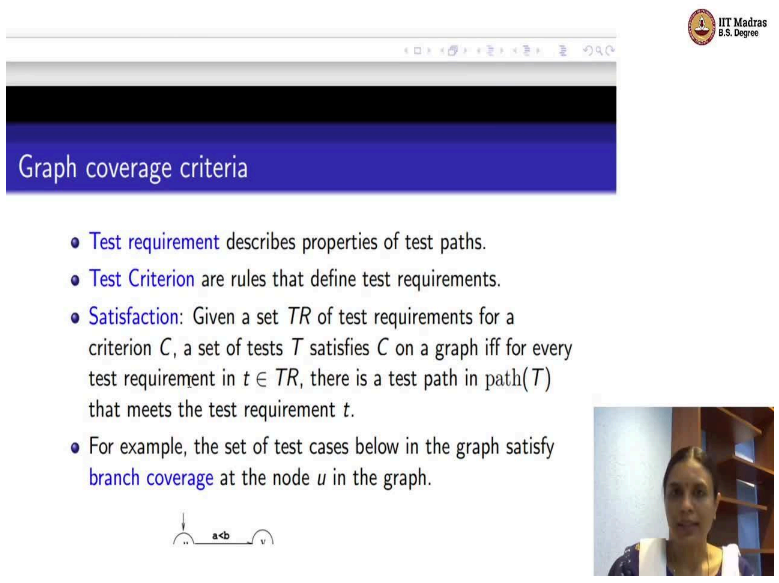
We use graphs in test case design as follows:

- Develop a model of the software artifact as a graph.
- Such graphs could contain several paramaters 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.

So, as I told you, we will use several different kinds of graphs, which we are going to define as we move on. So, what we are going to do is we will learn how to develop a model of a software artifact, it could be a piece of code, it could be a requirements, document anything. So, we learn how to take that and write a graph out of that, if you use an IDE, you can help use the IDE to get the graph many IDE, IntelliJ, Eclipse, Visual Studio, they all have features to generate control flow graphs for you.

So, we will develop a model of a particular software artifact as a graph. Those graphs can have several parameters, they will have initial final vertices, they will have annotations, as I told you, then they will sometimes when you look at data flow graphs, you can annotate specific vertices or edges with values of variables, with conditions on variables and so on. And then we will use reachability algorithms to work with testing of these graphs. So, that is what we are going to do.

(Refer Slide Time: 27:04)


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 $\text{path}(T)$ 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.

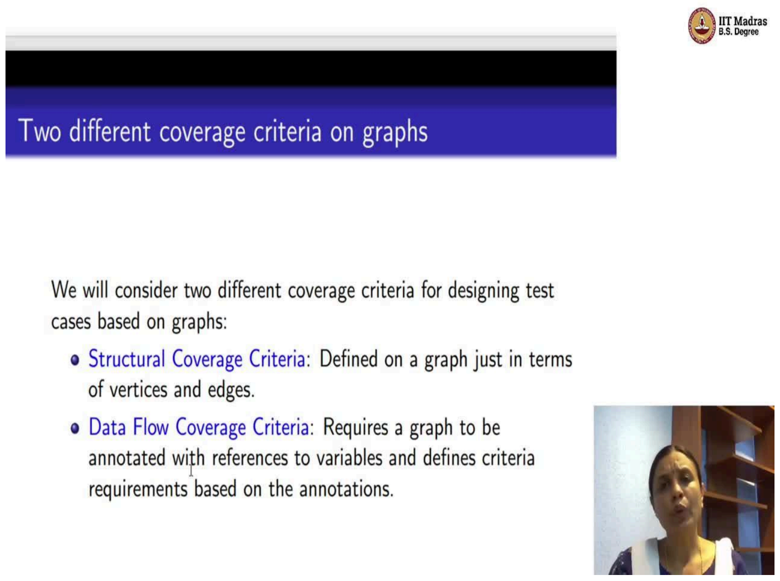
One last bit before we go on just as an example, we will see this more formally and then subsequent modules. So, we saw these things earlier. So, a test requirement is for example, you could say a cover every if statement in a particular piece of code. So, that is your test requirements, somebody tells you what to test for. Somebody might say, cover every loop, test every loop in your method that could be a test requirement. So, it describes some properties to be tested some properties of test paths, then a test criteria.

A test criterion is basically rules that define test requirements, somebody telling you this, then we say given a set of test requirements for a Criterion C, let us say cover every loop, that is the criterion and there is a set of test requirements for that. Our goal as a tester is to write a set of test cases that satisfy this criterion C, I can cover every loop and I say such set of test cases satisfies this criteria C if and only if for every test requirement T and TR, there is a test path that meets the test requirement T.

For example, you take this graph that I showed you earlier, this has 4 vertices u, v, x and w. So, many edges, some of them have labels like for example, you can take the implicitly read this as u is the initial vertex x is the final vertex, you can take the edge from u to v provided a is less than b, you can take the edge from u to w provided a is greater than b or a is equal to b. You can take the edge from u to x provided a is greater than p. So, based on the test case, I give whichever of these conditions is satisfied, that edge will be taken.

Suppose I give 3 test cases as given here, one for each kind of condition, one for a less than b, one for a equal to b and one for a greater than b, then what have I basically covered, I have basically covered all the 3 edges that are coming out of u, I have defined test cases that will execute all the 3 parts that are coming out of u, this case I mean it as to say I have covered all the branches that go out of the vortex u or in other words, I have done branch coverage for u. This is just an example, I will formalize this for you.

(Refer Slide Time: 29:34)



Two different coverage criteria on graphs

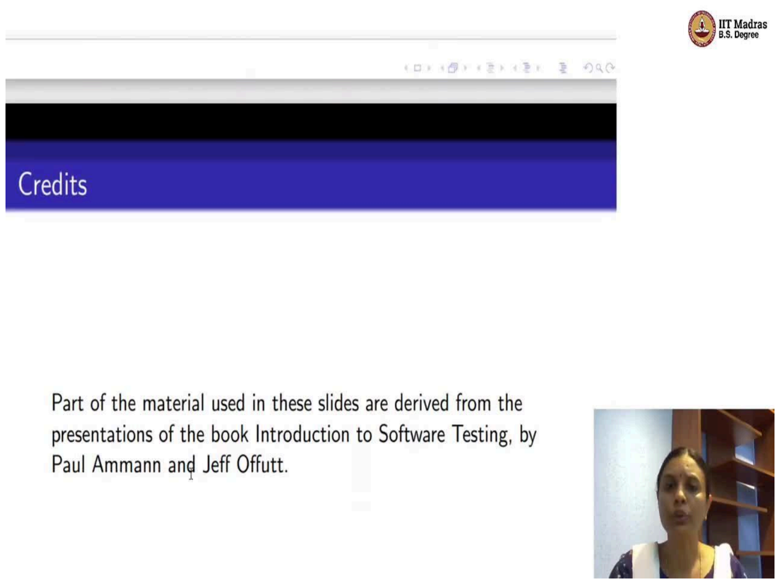
We will consider two different coverage criteria for designing test cases based on graphs:

- 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.

So, we will look at coverage criteria that are purely structural. Without any worring about any variable that types where do they come from? Nothing. We will just define graphs in terms of vertices and edges and define coverage criteria first to start with. That is this week. We call that a structural coverage criteria.

Next week, we will start worrying about variables, annotating the vertices or edges with those values with conditions and worry about flow of a variable from its definition to its use. So, we call such coverage criteria as data flow coverage criteria. We will see them in the next week. So, these would be coverage criteria that we would see.

(Refer Slide Time: 30:17)



Credits

Part of the material used in these slides are derived from the presentations of the book Introduction to Software Testing, by Paul Ammann and Jeff Offutt.

If you would like to know more about it, you can refer to this textbook which I have given us a reference. So I will stop here for today. In the next module, we will do Depth First Search and Breadth First Search. Thank you.