SOFTWARE TESTING

Prof. Meenakshi D'souza
Computer Science andEngineering
IIIT Bangalore-IISc

INDEX

WEEK – 9

WEEK – 10

WEEK – 11

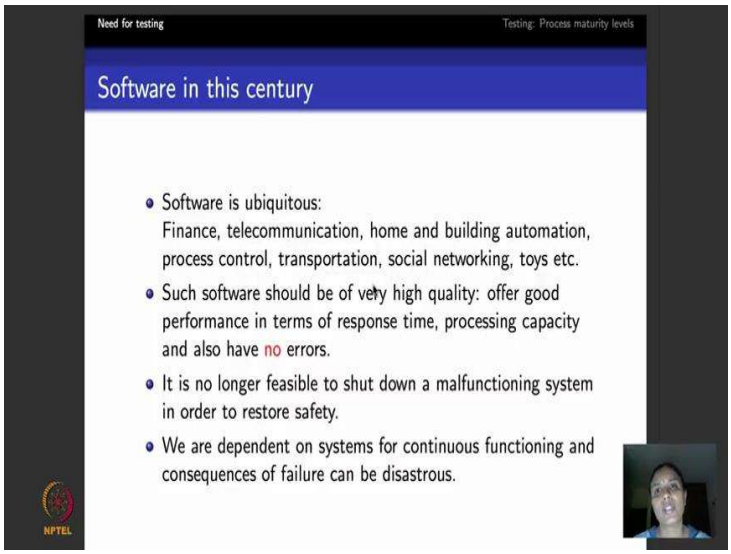
WEEK – 12

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

Lecture – 01
Software Testing: Motivation

Hello everyone my name is Meenakshi. And this is a first lecture that I will be doing as a part of the course on software testing that I am currently offering for NPTEL. So, what are we going to do today? Basically I will talk to you about the kinds of software that we encounter in the present world, and what do expensive errors look like, and how to avoid these expensive errors by using testing. So, it is basically motivation module where we deal with how important testing is.

(Refer Slide Time: 00:45)



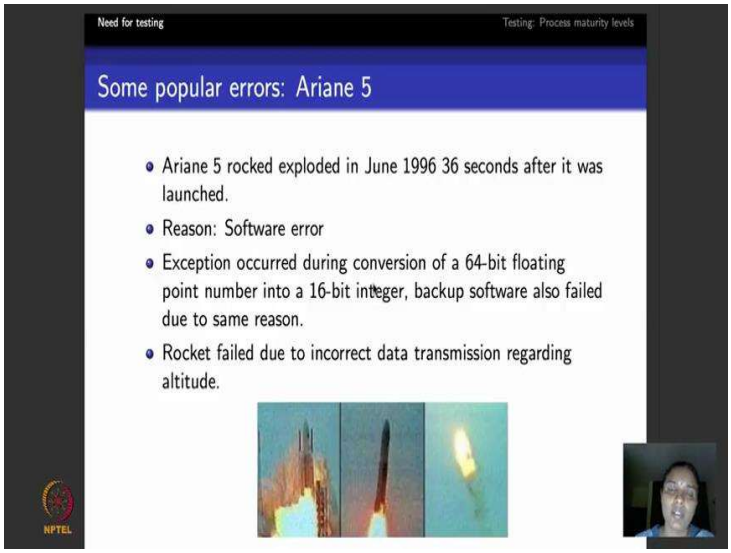
So, what a software in the current century look like? If you look all around you, it is not too difficult to understand that there is software.

So, we use Paytm, we use banks online, we go to an ATM machine there is software, I make a call to my friend or my parents using my mobile phones, there is software. Software that controls the heating, ventilation then air conditioning in our homes and offices, there is software that tells you how electricity flows through a grids, how water flows through a network. Software that run in manufacturing industries, the software that help us to drive a car, autopilot and other kinds of software that help you to fly our

planes, when the software that controls real network, the software even in toys that children use right.

So, what is our expectation about such software? We all expect the software to be error free, not only that we want the software to respond fast right. The second I insert my card into the ATM machine and what the welcome screen the next second before I had link my id, I cannot be it long enough. And the other important thing to notice that if you consider a software like the autopilot of a plane, let us say there is an error in the software you cannot say that I will shut it down mid flight, I will rectify the bug and I will restart to be able to run right. It is no longer feasible to be able to shut down system because there is an error in the software. We want the system to be able to continue to done in the autopilot case it needs to be able to continue to fly the aircraft safely and help us to land safely.

(Refer Slide Time: 02:22)

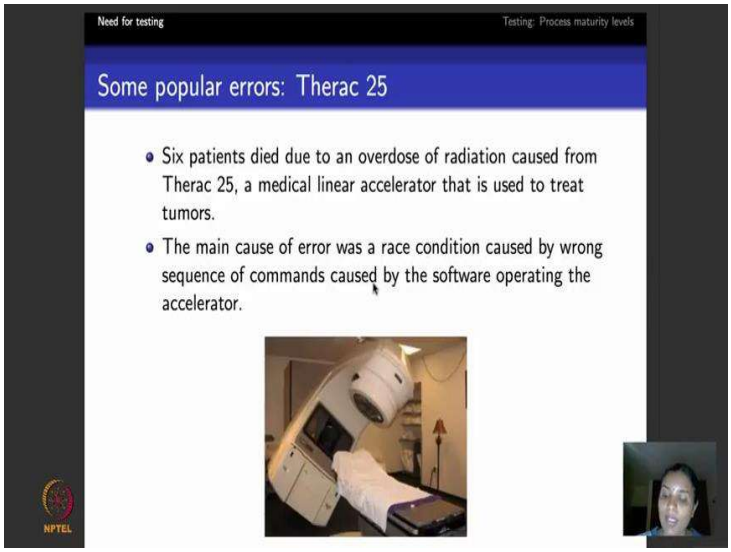


So, I would like to talk to you about some popular errors that have occurred in the past. This was the error is the error is the error in European space agency rocket called Ariane 5, it occurred exactly 21 years ago and the important thing to note that this error is because of a software bug. So, there was this rocket called Ariane 5 which was being controlled by software that is launched by European space agency.

So, this software had the following error. So, it was trying to squeeze in data corresponding to a 64-bit floating point number into a memory space that is allotted for a

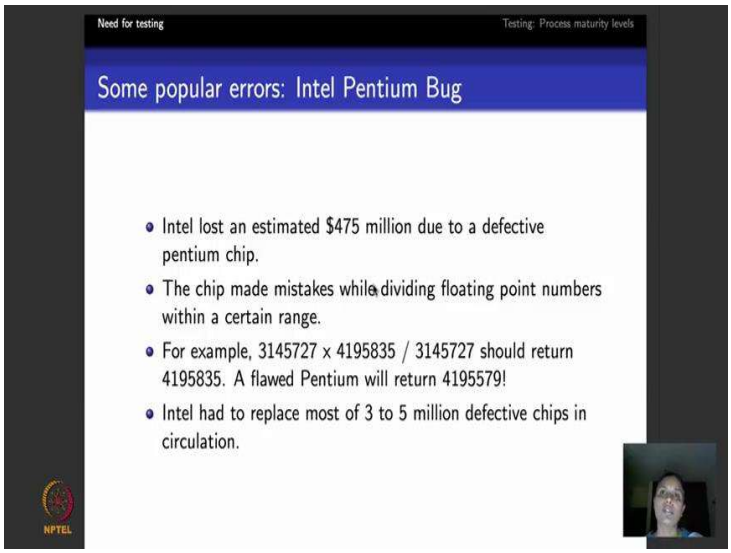
16-bit integer. So obviously it is not going to be able to succeed in doing that. And because such rockets have safety critical systems they always have backup software, but in this case the problem was the backup software also had exactly the same error. So, this resulted in transmission of incorrect altitude data to the aircraft and this rocket Ariane 5, went and plunged into the Atlantic Ocean within 36 seconds after it was launched. So, that is about 15 years of total effort and you can imagine that millions of Euros that would have been lost.

(Refer Slide Time: 03:33)



So, the next example that I would like to discuss about is another unfortunate example that happened, again because of a software error. So, in this case 6 patients lost their life due to buggy software in a machine that gives radiation therapy to cancer patients. So, there was a race condition error in this software. So, this software ended up calculating more dosage of radiation then what was needed and unfortunately these patients lost their life because of an overdose of radiation.

(Refer Slide Time: 04:05)



So, the next kind of example that I would like to discuss with you it is software that is expensive that costssomebody a lot. You all would have heard of this Intel Pentium processors right. So, there was this particular P4 Pentium processor, the mathematician in the US, when he was trying to do research towards its prime numbers, he found that this Pentium 4 processor was doing floating point division wrongly. So, when it was announced an Intel investigator did realize that all the P4 processors that it had released to the market had the same error. So, the only solution left was to be able to recall all these processors and that cost Intel a lot of money.

So, this does not bring us to the end of expensive errors. So, if you Google you know you can talk, you can find Toyota breaks, crashes all kinds of crashes happened.

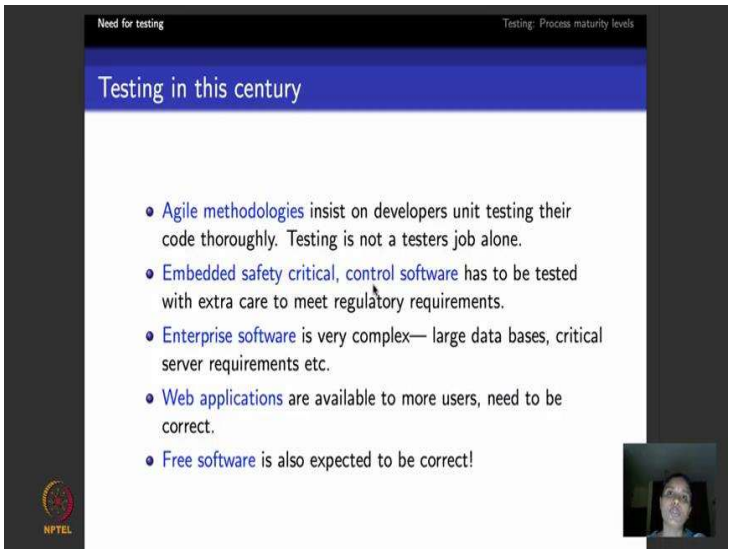
(Refer Slide Time: 04:56)

So, here are some few reports that talk about how expensive errors can be and what is the cost of inadequate testing. National institute standards and technology in the year 2002 released a report which basically discussed the impact of software testing in US.

So, it says that inadequate software testing causes the US economy anywhere between 22 and 59 billion every year. Not only that the report goes ahead and says that if there are better approaches that can be found to do software testing, then you could bring down the amount of these losses to almost half of what it is. The other popular categories of losses are due to web applications which we use all the time. So, one particular thing that I would like to discuss with you now happened almost 10 years ago. Amazon had this online discount sale, and because of a software error what happened was that it ended up giving double the amount of discount and by the time Amazon realized it, it is too late that are already lost lot of money.

The next fact that I would like to drought your attention is a report that Symantec, the security organization published in 2007. So, it says that security related vulnerabilities that occur in a financial transactions, in our online wallets and so on mainly occur no because of cryptographic errors, they occur because of software errors.

(Refer Slide Time: 06:21)



So, we know expensive errors and software can be. So, let us try to look at the various kinds of software and typically how they are developed. You might have heard this buzzword called agile methodologies right. So, agile methodologies basically insist that people who develop the code, that is the developers, also have to unit test a code. Typically, developers do not have any great knowledge on testing, but agile methodology believes that the developer himself or herself the best person to identify the error in the code. So, that puts a lot of pressure on developers to be able to know testing well.

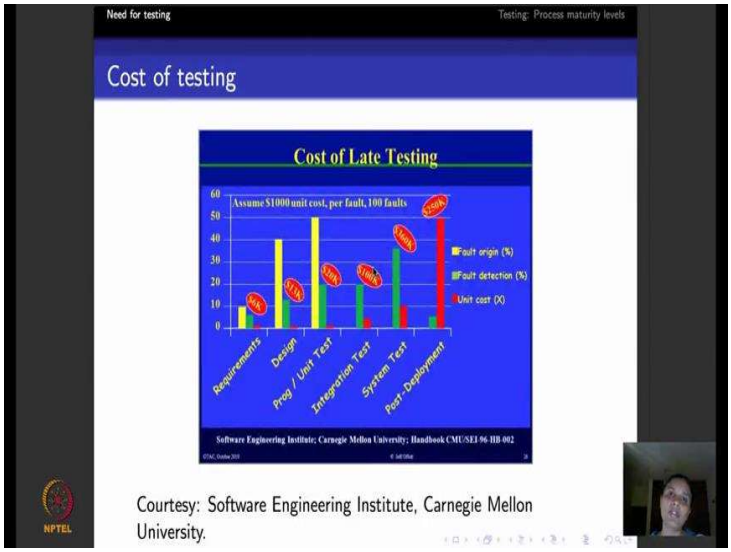
Now, let us look at the various kinds of software that we will deal with. If you consider a software that typically runs in your cars that helps you to do breaking automatically, or does a cruise control or a software that runs in your aeroplanes, such software is what is called embedded control software. So obviously, such software runs in what are called safety critical systems where a failure can be catastrophic. So, safety critical software have to meet regulatory standards where, by which regulatory authorities decide if a software has been tested enough and it is fit for release. So, this just means that almost double the effort goes into testing the software.

Other kind of software is what is called enterprise software. So, what would be a typical example of an enterprise software let us see a software that run city bank right or a software that managers are Indian railways ticketing system right. What do such software

have to deal with, the basically the first complex thing is that they deal with fairly huge data basis that have a lot of data and the software is critically dependent on the server and backup server running all the time right.

The third popular category of software is what is called web applications. Things like social networking sites, amazon online and so on and so forth. We all know how important it is for these software to be correct. Finally, I would like to end this slide with the point what free software suppose you pick up a piece of software from the internet for free. Just because it is available for free you are not willing to accept the fact this software could have errors. So, paradoxically we expect free software also to be error free.

(Refer Slide Time: 08:41)

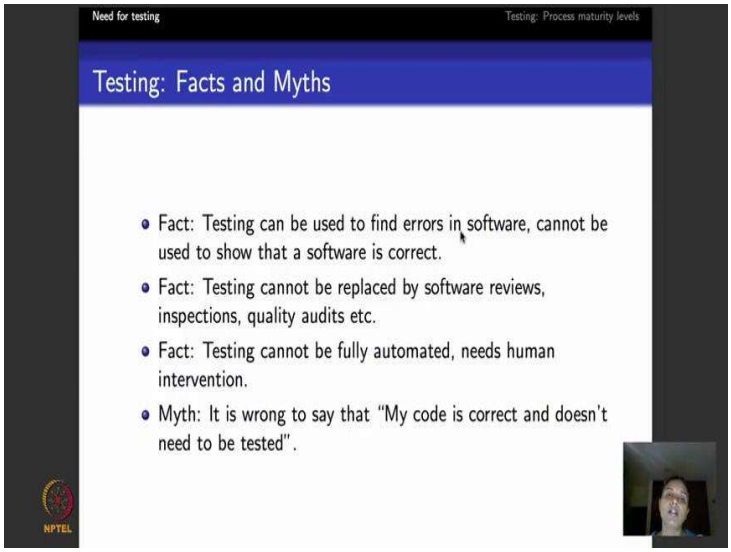


This is a very important slide. So, this is a part of study that was done at Carnegie Mellon university a few years ago. So, this slide discusses about how expensive testing can get as we go down software development.

So, if you see a typical software development initially you write requirements and then you do design right. And then your unit test your software and then put together the modules, that you have tested and do what is called integration testing then you put the software with the system or hardware that it is supposed to run on and in your system testing and finally, release the software.

So, suppose there was an error in the requirements, and it was found that and there then what the slide tells you is that gives you the cost - the cost is fairly low right, but suppose there was an error that was found in requirements or design, and it went all the way through testing, integration testing, system testing, did not get detected at all. The software called released and the error was found there. Then what this slide tells you is that the cost fixing that can be very high, it is not only the cost fixing the error the consequences of that error as we saw through the examples can also be really bad right.

(Refer Slide Time: 09:49)

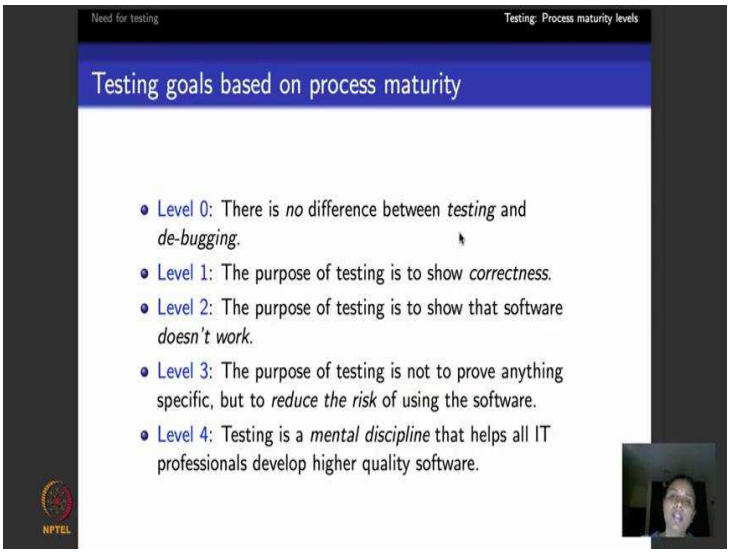


So, what are the facts and myths about testing? There is a popular saying by Edgar Dijkstra who said that never think that you can use testing and say that I have proved my software to be correct. It is wrong to say the testing prove software to be correct. The main goal of testing is to be able to find errors in the software right. Using testing I can never say that I have tested my software and it is fully correct, it is a wrong statement to make. The other wrong statement to make is that a developer typically thinks that you know " have written the code, I have debugged it well and my quality auditor my SQA in my company has inspected the code. So, there is no need to test it". That typically will not work; it still needs to be tested.

Another popular thought that people have especially these people who sell software testing tools is to be able to say that you just download and install my tool it can do magic; it will do every kind of testing you can think of. That is not true.

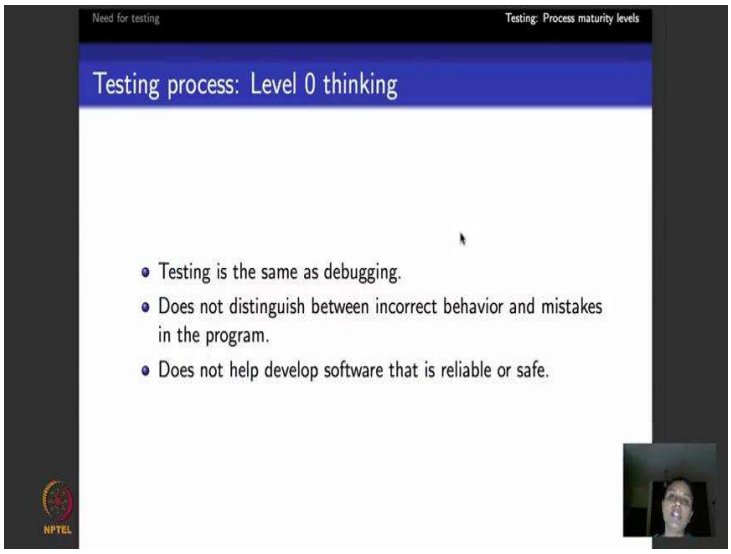
The main goal of testing tools is to be able to help you execute the test cases and record the result. Now to be able to design test cases, you will have to be able to design cases to find errors effectively. Typical Pareto principle applies here 80 percent of the errors come in 20 percent of the code or design. So, test case design which is what the course is about needs a human to be able to do it right.

(Refer Slide Time: 11:18)



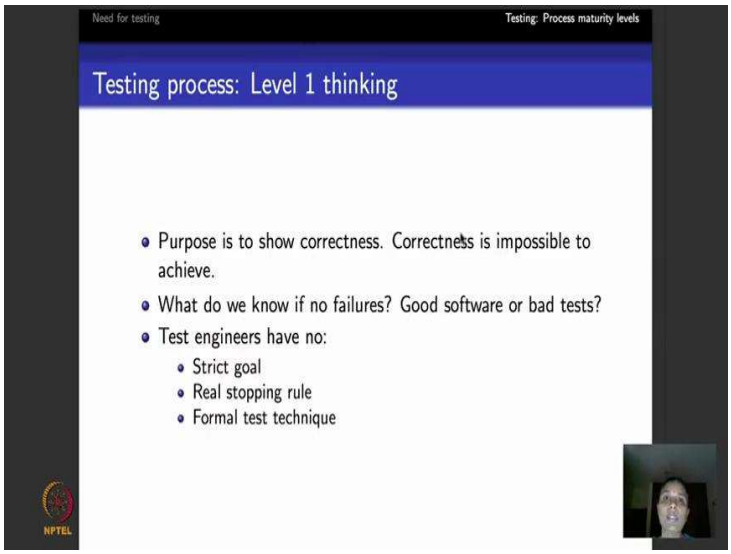
So, I would like to end this module by discussing about certain process maturity levels in testing. Why is it important to look at process maturity levels? It is important because it tells you what role testing plays in the software development that you do. Broadly there are 5 levels beginning from 0 and going all the way till 4.

(Refer Slide Time: 11:42)



So, what does level 0 tell you? Level 0 tells you that there is basically nothing called testing. Developers write their code, they debug their code they ensure that it is correct and that is it, they go ahead and release it right. So, it is clear that this does not really help to develop software that is considered to be fully safe and reliable.

(Refer Slide Time: 12:01)

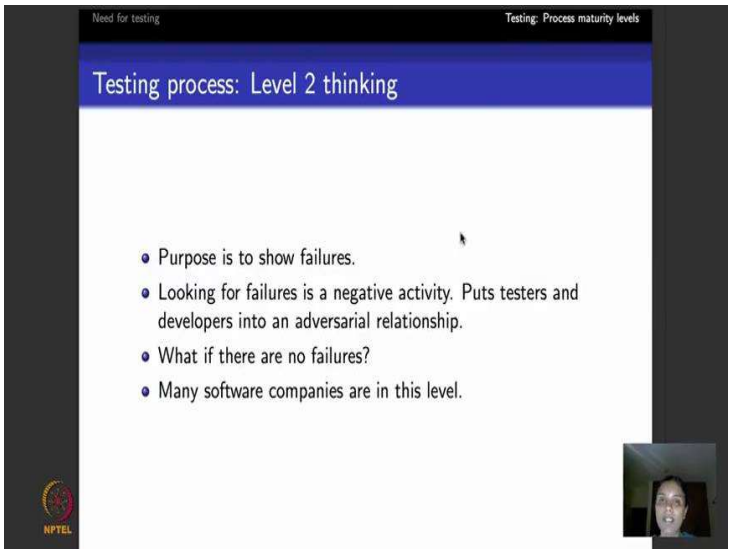


The next comes level one thinking. Level one thinking a developer thinks that he or she has written a piece of code and their goal to test is to be able to show that the code is correct. As we saw little while ago I clearly cannot use testing to show that a piece of

software it is correct right. So, let us see a particular piece of code has an integer variable. To be able to exhaustively tested on the 32 bit processor I need to be able to give every value which is 2 power minus 32 to 2 power 32 plus or minus 1, and even when an extra fast PC this is going to be able to take several years to do. So, unless I test it for every value I cannot say that the testing is correct.

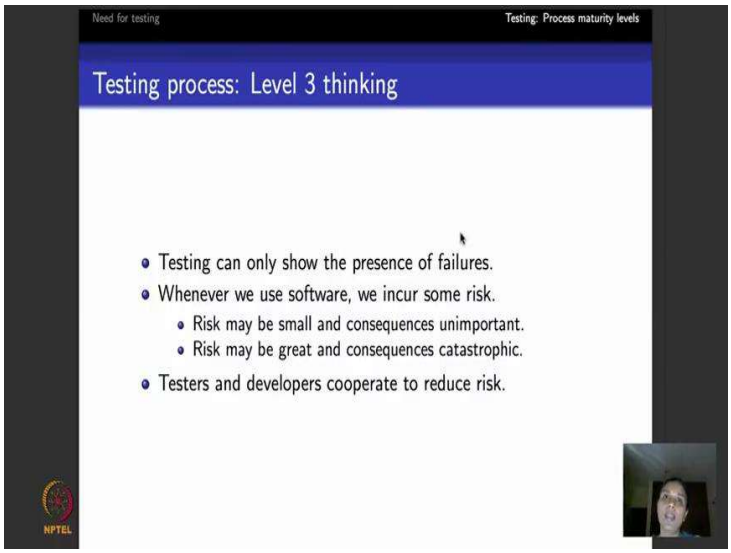
So, here there is nothing like test engineers and they even if there are they do not have goals and they just show that the software is not failed, but the underlined listen it is not failed because it is correct or it is not failed because you have designed the test cases wrongly, that is never clear.

(Refer Slide Time: 13:00)



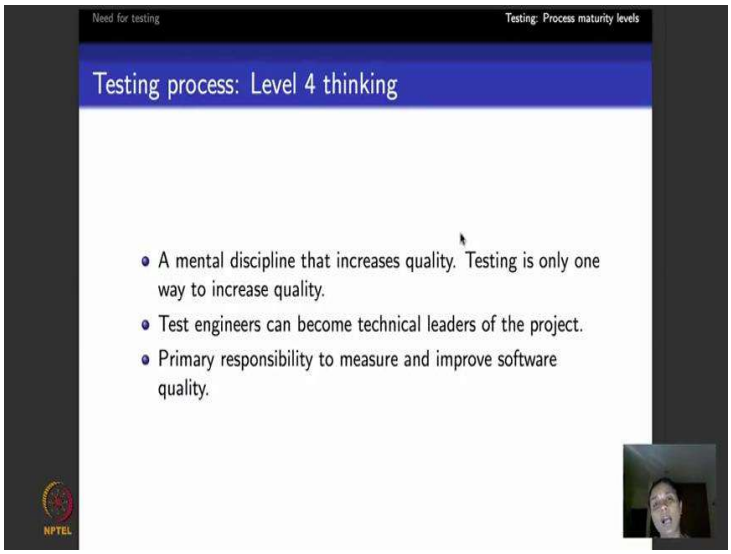
The next level is level 2 thinking when you begin to believe that the goal of testing is to be able to identify failures or errors in the software. This is the beginning of positively using testing, but it organizations that are typically at this level there is lot of tiff between developers and testers, they belong to different teams and then one does not want to help the other, and there is confusion even though the goal of testing is fully realized. We move on to level 3 where they not only realize that the goal of testing is to find errors, but they also work together and say that we will not only find errors, we will make sure that we reduce errors to the extent possible in the software.

(Refer Slide Time: 13:28)



And that is also conscious understanding that when I release the software there is a bit of risk involved. The risk could be high or the risk could be low and my goal is to make the risk as low as possible.

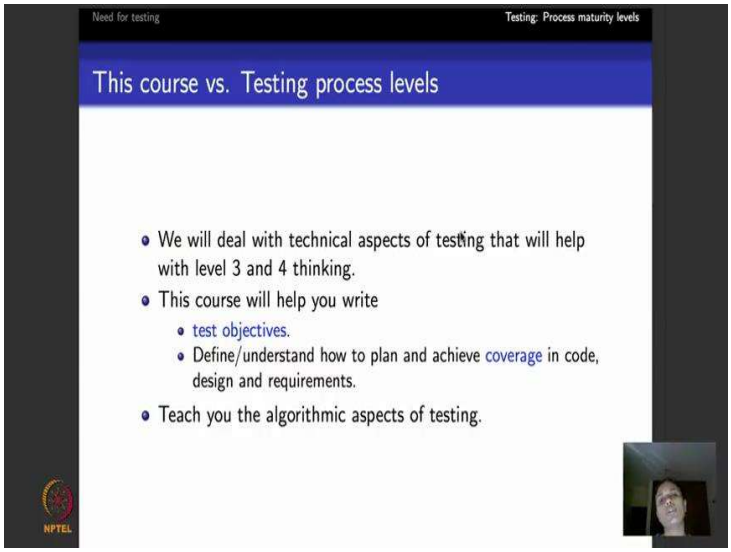
(Refer Slide Time: 13:55)



Level 4 thinking is where large organizations strive to be in, here testing becomes a mental discipline. So, there is positive thought and efforts in the organization level to make sure that testing teams' efforts are taken into consideration to develop software that is as error free as possible. An organization like Microsoft which try to achieve

excellence in level 4 thinking and this is what is the desired level that we would like to achieve in testing.

(Refer Slide Time: 14:28)



Now, what is this course got to do with all the levels and terminologies that we saw till now? So, this course will help you to think that I want to be able to do testing at levels 2 3 or 4, which means what that the goal of my testing is to be able to find errors. So, to find errors, I have to first define what are my testing objectives in technical terms, and I have to be able to figure out how to effectively design test cases to be able meet or cover these test objectives.

So, in the course we will look at algorithms and techniques that will help you to formulate test objectives and design test cases that will help you to meet these objectives. So, the next module that we will be seeing, we will introduce the various terminologies that exist in testing and also clarify about what we would use is a part of this course.

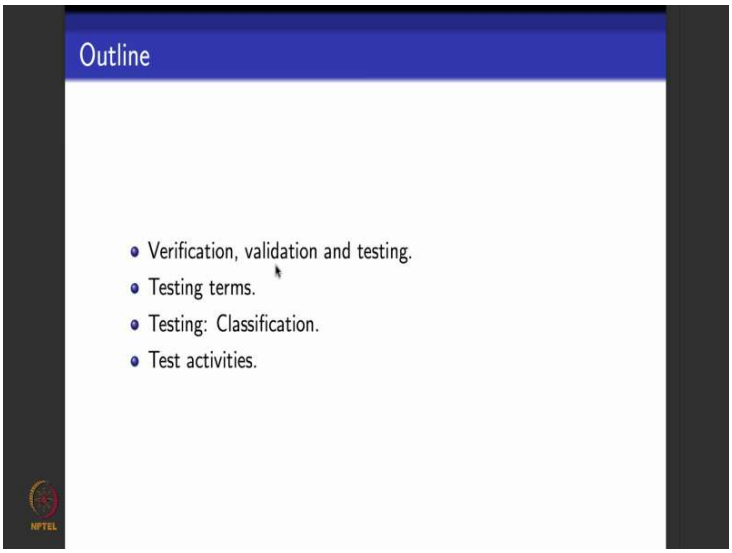
Thank you.

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

Lecture – 02
Software Testing: Terminologies

Hello everyone. So, this is a first module of the first week. So, in this module what I will be doing is to introduce you to the various terminologies that exist in the area of software testing. What we would be using in this course - clarify the basic terms and tell you what are the various types of testing, the various methods in testing. We will also look at various activities in testing and see what this course will deal with. Here is an outline of the course.

(Refer Slide Time: 00:39)

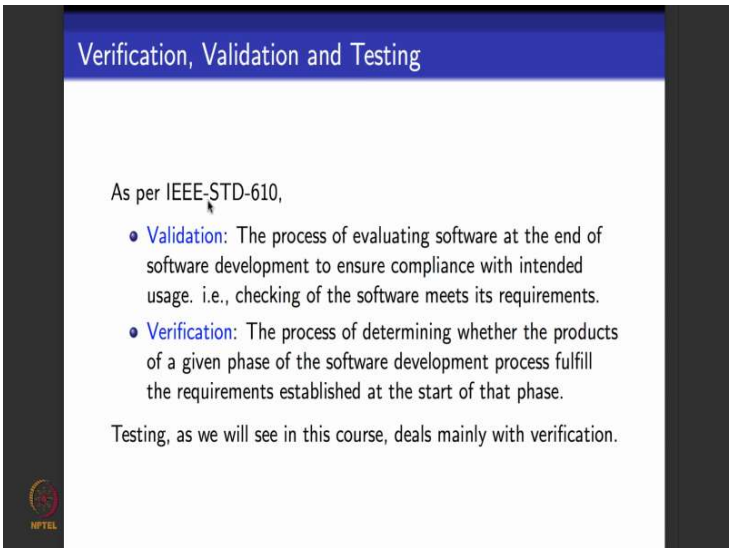


So, to begin with we clarify on these terms which you might have heard about verification, validation and testing and I will tell you what we will deal with in this course. We will also look at a whole set of related areas briefly and clarify what this course will cover. There are several terms and terminologies related to testing. What is the test case? What is an error? What is a fault? What is a failure? So, I will introduce you to all these testing terms and we will see what they mean.

And then in testing there are several methods of testing, several types of testing, several phases of testing. You might have heard terms like white box, black box, functional

testing, usability testing, performance testing. So, we will see what these various terms are in the classification of testing. I will end this module by introducing you to various testing activities and tell you what our course is going to focus on.

(Refer Slide Time: 01:36)



IEEE maintains a standard glossary of software engineering terms in this particular standard called STD-610. So, as per that standard what is verification and what is validation? Validation is something that you do at the end of software development or system development. You may have your own software or system developed ready in place and you want to be able to check if the software of system meets all its requirements when you do that at the end of the development then that is what is called validation. Verification deals with what you test or verify while developing software. So, while developing software you might go through various phases, phase where you define requirements, phase where you do design and architecture. Phase where write code and phase where you test. At each stage you are dealing with a set of software artifacts.

So, the kind of verification that you do where in you check whether a particular software artifact needs the requirements that were established for that particular set of artifacts then you do what is called verification. Testing as we will see in this course mainly deals with verification. There are several other related areas that you might have heard about.

(Refer Slide Time: 02:46)

One of the most popular or my favorite ones is that of formal methods, formal verification. There are 3 broad areas that people work on within formal verification namely model checking, theorem proving, program analysis. Testing as we will see in this course, we will not deal with model checking or theorem proving or program analysis. Towards the end of the course we will see symbolic execution. A lot of symbolic execution is used in program analysis, but we will see it from the point of view of testing.

In fact, there are several NPTEL courses available in each of these areas and if interested you are more than welcome to go and see them.

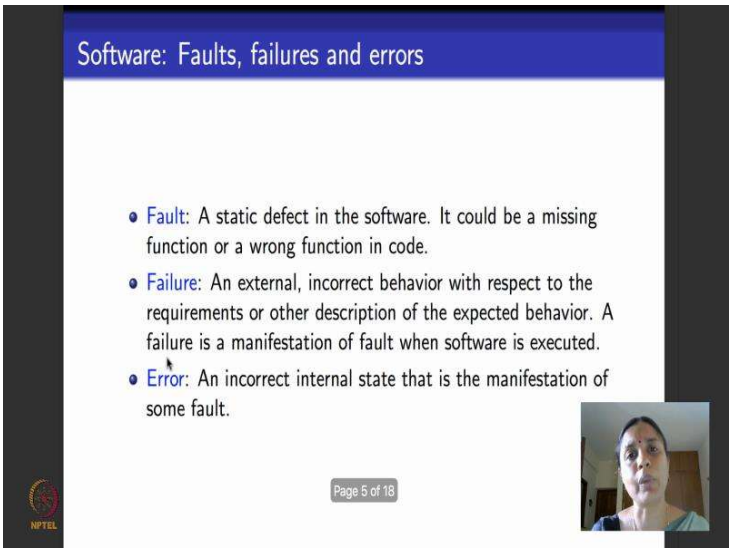
A related area where people also do verification is what is called modeling and simulation. Let us see if you have a piece of hardware design or a system design or a software design. For hardware design popular languages that you could use a VHDL very law, so you have your design and then you model your design in this particular language and you run simulations to see if the design model that you have done correctly does the design as per its requirements. So, you have a modeling language, which is usually proprietary or open source you modeled using that modeling language and run several different kinds of simulations to see if the design model needs its requirements.

So, that area is modeling in simulation and the testing that we would see in this course we will not deal with that also. Another related area is what is called accreditation. Now

what is accreditation? Accreditation deals with software that is safety critical and needs to be certified. Take for example, software that runs a plane, right, like typical autopilot software, any company you cannot say that I have written in autopilot software as per the normal requirements of autopilot software. So, here it is, take and run right. So, these autopilot softwares being safety critical because they cannot afford to fail they have to be audited and accredited by responsible authorities like FAA in the US, DGCA in India and they demand lot of additional testing there is a separate process of accreditation for which there is testing needs to be done.

But accreditation as it is, we won't look at, in this course what we will look at is the kind of testing that people do towards accreditation and the algorithm behind that kind of testing.

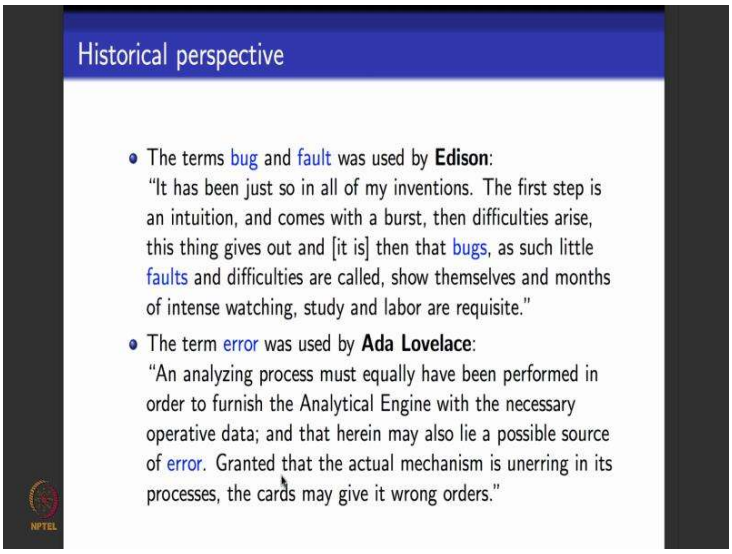
(Refer Slide Time: 05:02)



We will move on to looking at various terms in testing, you might have heard of lot of these terms faults, failures, errors … what do they mean? So, what is a fault? Fault is considered to be a static defect that occurs in software, a static defect in the sense the defect that occurs as a part of the software, not a defect that occurs when the software is executed. So, this defect could be a function that was wrongly written, function that was not written at all, it could be any of these, right and then when software with the static defect or fault is executed, there is a misbehavior or a wrong behavior on software that is observed, right. This wrong, observable behavior is what is called failure. When the

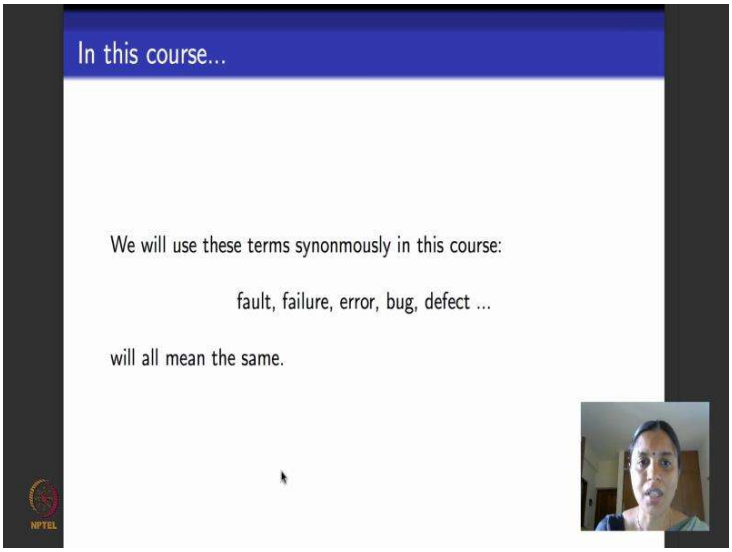
software has a fault to execute around the software you observe a failure, then the software is supposed to enter an incorrect state. The incorrect state in which software enters, the software that has a fault enters and a failure is manifested is what is called the error in the software.

(Refer Slide Time: 06:07)



This is a small piece of history thanks to the book by Ammann and Offutt software testing. So, they connected the term bug and fault to Edison. So, apparently Edison use these terms first. So, he talks about his inventions and he says there is a lot of excitement when you initially invent something and then comes a bug or a fault, which he calls difficulties and he says they show themselves and it takes sometimes lot of effort to fix them. The term error is credited for its first use, to the first programmer who could have heard of namely the lady Ada Lovelace. So, when she was programming using Charles Babbage's analytical engine and punch cards, she says that, there could be an error. So, this is considered the first use of error and she says error could make the cards give wrong orders.

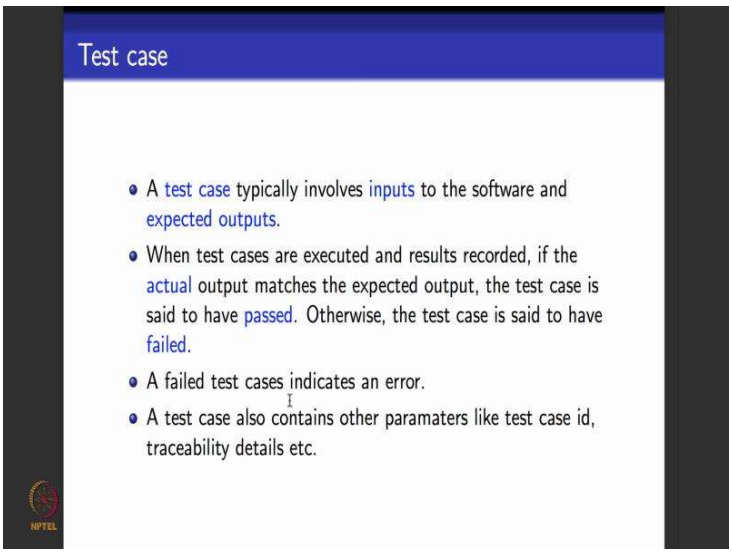
(Refer Slide Time: 07:00)



As far as this course is concerned, we will use all these terms synonymously--- fault, failure, error, bug, defect they will all be used interchangeably and synonymously, they will all mean that they something wrong in the software artifact that I am testing.

Another important term that you need to know and get a clarified right now that we will use throughout the course in testing is when we say testing the first thing that we have to do is to write test case. What is a test case?

(Refer Slide Time: 07:26)

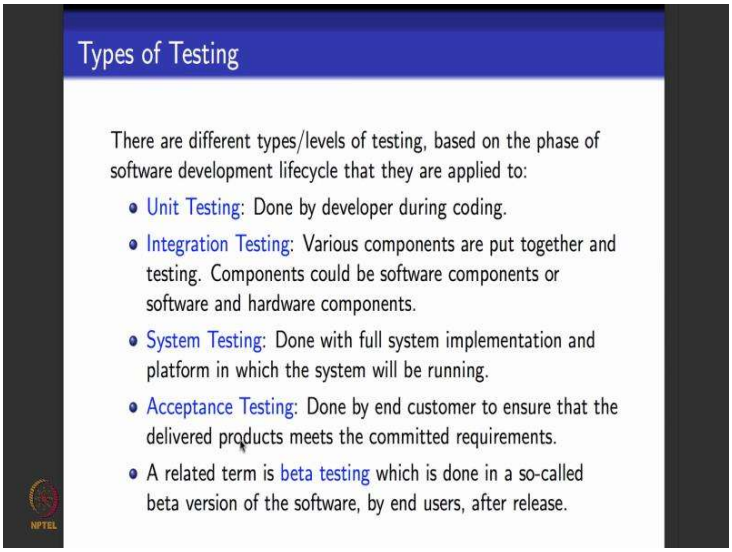


Test case typically has 2 main parts, it gives inputs to the software artifact that is concerned and it also says when I give this input to the software and execute the software on that, what is my expected output. So, test case has an input and what is the expected output, it is to be noted that if a test software artifact like code I always give only inputs to the codes and the part of the test case and be typically do not give values to internal variables to output variables and so on.

So, now I give this test case give inputs to the software and run the software, the software produces the output right. So, the output produced by software is what is called actual output. So, now, what we do typically, we check the actual output, compare the actual output to expected output and say whether they match or not. If the actual output matches the expected output, then you say the test cases said to have passed. If the actual output defers from the expected output then typically in testing this says that the test case has failed. A failed test case indicates that there is a fault in the software and the fault has manifested itself by resulting in an error, as per the terminologies that we saw.

Typically test case also contains things like an ID because you need to track and record the test case and people also maintain traceability information. What is traceability for a test case? Traceability tells you what are you testing this particular software artifact for. You might be testing a particular functionality in a piece of code. Who tells you to test for such functionality? Where did that functionality come from? That functionality could have come from a set of requirements. So, if it came from a set of requirements, which are the requirements that it come comes from? So, all these information is maintained in what is called traceability matrix, traceability data which is also a part of the test case.

(Refer Slide Time: 09:34)



Now, we will move on to looking at the various terms that are popular in testing and trying to gain clarity on what is what. So, when I do testing, I do testing throughout software development and at each stage of software development I could do a different kind of testing. So, there are various levels of testing based on which phase of software development that I am testing on. The right first testing that I will do is called unit testing, unit testing is typically done by a developer. When he or she writes the code, they debug the code, they test the code then and there to see if the code is working fine as for as its requirements of concern.

These days because of the stress on agile methodologies developers are expected to do a lot of unit testing themselves, they do not really have the luxury to pass things to a tester and say that you do it, right?

After unit testing we typically do what is called integration testing. So, what is integration testing? A developer again can do integration testing. So, let us see a developer has written several modules of code and then they are trying to put together the code, what do you mean by putting together the code? Maybe the code is in different methods in particular method could call another method, a procedure could call another procedure. So, when you test these focusing on these calls, these function calls, procedure calls when you test the interfaces that occur between the various modules in code then you do what is called software integration testing.

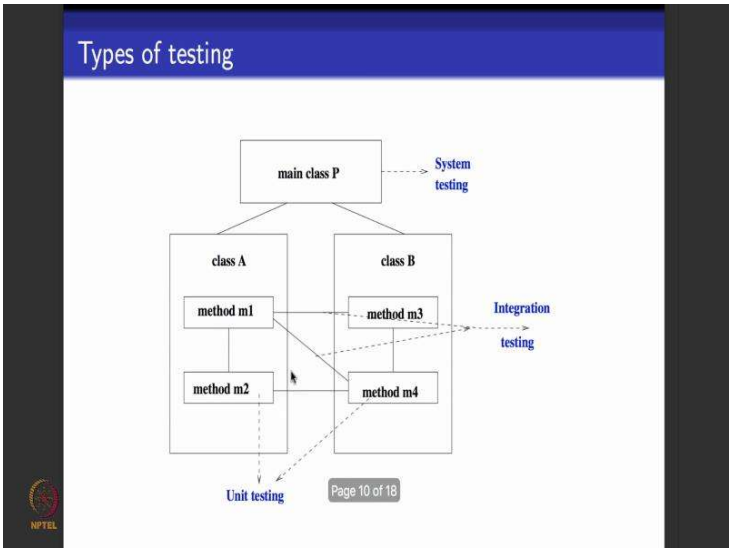
There is also another terminate integration testing, where people take the software as it is written and put it on the hardware that is supposed to work on. Let us say you have a fairly large server right you put it on the server that is supposed to work on you supposed to have integrated the hardware within the software. So, when you integrate the hardware with the software and test the integrated piece of hardware and software that is also called integration testing. So, post integration testing, the next step that people do is what is called system testing. So, in system testing the entire system is put together. Let us say take the case of an enterprise software, you will put together the server, the database, the web interface, the application server, the clients and all the interfaces between them and then you will test the end to end system from where the inputs come, what happens in the central system, what are the kind of main decisions, that are taken and how do they result in the output being produced.

In embedded software typically like a car or a plane you put the software as a part of the main system, which could be within the car or within the plane, the inputs will come from sensors the main control algorithms will run, from by picking up the sensor and puts from the bus and then they will produce actuator output. So, this end to end software as a part of the system with the inputs and outputs integrated from their original sources, the entire system when we tested we call it system testing. After system testing we believed that the system is more or less ready for reduce.

The last phase of testing the people do is what is called acceptance testing. Here you give the system to an end user and check if the system is working fine as per the end user committed requirements that the software or the system was supposed to meet. You might have also heard of a term called beta testing. What is beta testing? In beta testing people specifically release what is called beta version of software. When they release a beta version of software they roughly mean that this software is working fine, but I may not have tested it for mitigating all the involved risks.

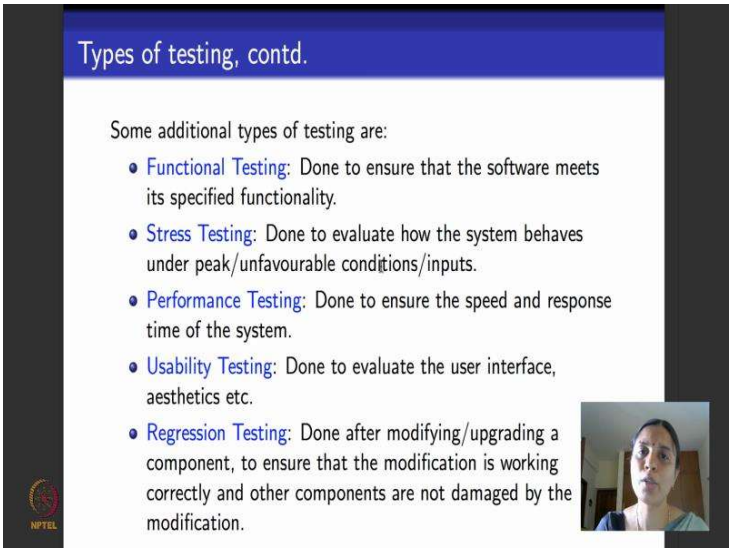
So, they are asking the users to start using the software and let them know if it has any bugs or defects. So, when you do this kind of testing then you are doing what is called beta testing right.

(Refer Slide Time: 13:27)



So, let us take an example to understand is a little better. So, let us say there is a main class called class P, which in turn has 2 classes A and B. Let us say class A has 2 methods m1 and m2, class B also has 2 methods m3 and m4. Let us see a particular developer is working on writing code for method m2 or m1 or m3--- just that method. So, when the developer finished his writing code or while writing code the kind of testing that the developer would do to test the functionality of just that method is what is called unit testing. So, now, if you look at this picture method m1 happens to call methods m3 and m4, m3 intern calls method m4. So, when a developer puts together these 2 classes all their methods and tests features specific to these calls, then the developer is doing what is called software integration testing.

(Refer Slide Time: 14:41)



Let us say this after this integration testing, the developer puts together the entire system and tests at the level of the main class B then that phase of testing is what is called system testing. You might have heard of all these additional terms in testing: functional testing, stress testing or load testing, performance testing, usability testing. What are they we will see them now. So, what is functional testing? Functional testing is typically done to ensure that the software meets its specified functionality. What do we mean by this? Let us take a software that runs as a part of an ATM machine of a bank right. What do you think is the core functionality of an ATM software? Core functionality of the ATM software could be the following. Once the user enters the password and the pin, if the credentials match and if there is sufficient balance left in the users account then, the amount that he or she has requested to be withdrawn should be provided to the user correctly.

So, ATM is meant to do just this and when I test a software to check if it needs it is main intended functionality then I do what is called functional testing. Another kind of testing that is popularly done is what is called stress testing. So, here what happens is that the system is meant to be up and running in available lot of times and sometimes the system experiences what is called peak input conditions. For example, recently the class 12 marks where released right? So, they were released on the internet through a web application software. So, this system will experience what is called stress levels of input within 3-4 hours of the time in the date of release when all the students would want to

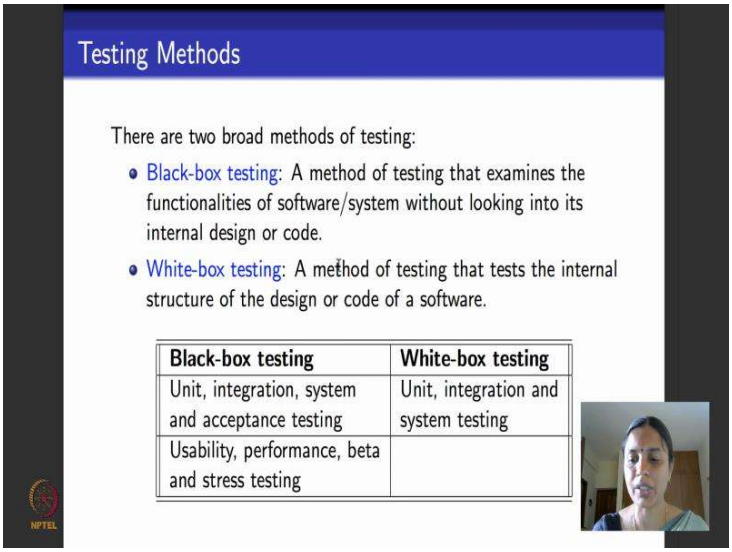
login and check what their marks are. So, when you test a system under this peak load conditions or input conditions then you do what is called stress testing.

The next kind of testing that you might have heard about is what is called performance testing. So, what will happens here is that here, people do testing to ensure that the system meets it is desired response time. It is fast enough, like for example, when we insert the card into an ATM machine, we want the ATM machine to be able to respond with the welcome screen almost immediately right, we cannot afford to wait for 10 seconds or minute or so. Performance testing specifies performance requirements on the system and checks whether all these performance requirements in terms of latency, speed, response, time, etcetera are met by the system.

Next popular kind of testing is what is called usability testing. It is not only useful to have a great software system, but the great software system is meant to be usable by a user which means the software should have a good user interface. It could be graphical or not, but the software should have a good user interface that is friendly. It should also be accessible, accessible in the sense by someone who is visually impaired, hearing impaired. It should be accessible. So, testing that is done to ensure that a software is usable, has a good interface, meets all the accessibility guidelines, is aesthetic enough--- the kind of testing that is done to do all this is what is called usability testing.

Finally, one important term in testing before we move on is what is called regression testing. So, when does regression testing happen? So, let us see you have developed piece of software and usually released it. So, post release there is either change to the software that you do or you add a new functionality to the software. Now it is obvious that you might want it first check if the change or the new functionality that you have added is working fine and the second thing that you might want to do is that this change or the new functionality does not affect any of the other features that they are already present in the software before this was done. The kind of testing that is done to ensure that the software is working fine after modifying or upgrading it is what is called regression testing.

(Refer Slide Time: 18:45)



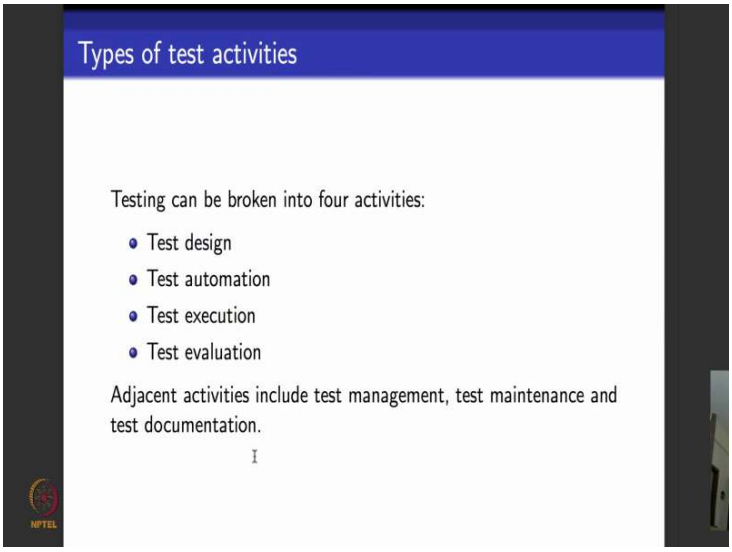
So, when all these methods of testing, all the various types of testing basically, 2 different methods in testing, you might have heard of these terms Black box testing and White box testing. So, what happens in black box testing? When you do black box testing you consider the software artifact that is being tested as a black box, which means you do not look inside it like, when you test the code you consider the entire code as a black box give inputs to the code execute the code and observe the outputs to check if the expected outputs match the actual outputs.

White box testing is the exact opposite. When I am testing a piece of code, with reference to white box testing I look into the code and I test for requirements like, suppose there is an if statement in the code can you write a test case that will cover the if statement. What is it mean to cover in the if statement? It means that can you write a test case which will execute the then part of the if statement ones which will execute the else part of the if statement once. So, similarly, when a white box test code I could insist that you cover loops in the code. So, let us say there is a while loop in the code, covering the loop would mean that you write test cases to skip the loop, to execute the loop in normal operations and execute the loop on boundary conditions.

So, in summary what does white box testing do? White box testing actually looks at the code, looks at the design at the corresponding software artifacts, looks at structure what it what is it have, what are the kind of statement, what are the kind of design elements it

has and then it tells you how to test it by looking at its structure. So, black box testing applies to almost all phases of testing, all types of testing it applies throughout the development life cycle, it applies for doing usability testing, performance testing, stress testing and so on. White box testing is typically done only during software development Once software is ready put together the system is tested. Later on when we test for other non functional requirements like performance, stress, response time, good user interface, being accessible, we typically do not test it, as a white box testing then we consider the system as a black box and then test for all these quality features in the software right.

(Refer Slide Time: 21:12)

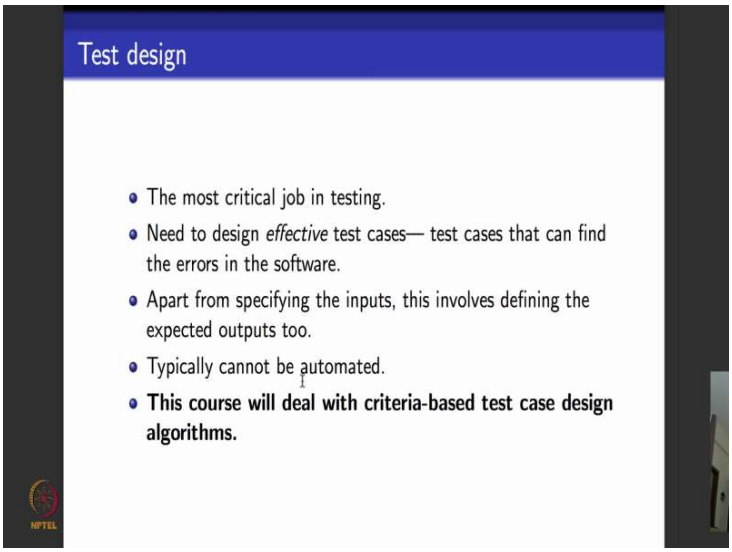


Now, we will move on to the last module, where we look at the various activities in testing. There are typically 4 raw activities in testing. So, we begin with design in cases because, if you do not have a test case then you cannot execute it and you cannot find errors in the software .The first thing is to be able to define or design a test case, once we design a test case I have to make it have to make it execution ready, that process is what is called test automation. After I make it execution ready, I can use a tool to actually executed and record the results. After I record the results then, I do what is called evaluation or analysis to check whether the test case is passed or failed and if it is failed and if there is an error where the error is and so on. These are the core technical activities and testing.

Apart from this there are several other umbrella activities. Like you have project management in software development, project management specific to testing is what is called test management. So, there are test managers to design test teams try to organize them.

Another important activity that happens is a part of testing is what is called test maintenance and test documentation. A lot of testing especially in enterprise software domain relies on reusing test , extensive number of test cases are reused again and again to be able to text software. So, how do I reuse a test case I should be able to document it and store it well to be able to make it amenable for use.

(Refer Slide Time: 22:59)



So, the kind of activities people do to ensure the test cases are well documented and well maintained and available for reuse as an when they are needed broadly constitute test maintenance and documentation.

So, now moving on to test case design this is considered the most critical job in testing. Why is it considered the most critical job in testing? Because, the Pareto principle applies to testing which means roughly 80 percent of the errors, a lot of the errors is focused on a very small percentage of the code. So, if I do not design my test cases effective what do you mean by effective in turn, if my test cases are effective then they will find errors faster, they will find all the errors or most of the errors areas that are

present in software, there is no point in saying that I designed so many test cases and I tested them for days and days and your software is doing fine right.

So, the effectiveness of test case design is in finding errors right and this needs domain knowledge. This needs knowledge about the system, about how it is developed and typically cannot be alternative. A lot of this course with deal with algorithms and testing techniques, that we will use to design test cases. I will give you more details about the precise kind of algorithms and artifacts that we will use in the next module.

After you designed your test case the next comes the converting this test cases in to executable script. So, you might have a test execution framework you could use JUnit, you could use Selenium. These are open source tools that will let you do test execution. So, you have to be able to make your test case ready for execution please remember, I told you that test case always talks about Inputs. So, suppose there is a requirement white box testing requirement, which says that you go to a while loop which is somewhere deep inside my code and you test for coverage of that while loop. It might so happen that the while loop has got no input variables, it directly deals with internal variables. Now it is up to the tester to be able to design test cases and give test cases input values such that this while statement is reached; not only it is this while statement reached this while statement is also covered under various coverage requirements.

So, how does a tester go about doing this? Software has to meet 2 important criteria called observability and controllability. We will look at these 2 concepts in the next module in this week. So, after your design your test case and it is ready for execution then comes the job of actually executing it and recording the results, this typically is almost always fully automated. There are several open source and proprietary tools that can do this really well for you.

After you have executed the test case, again, you need human intervention to be able to evaluate the result of the test case. So, let us say the test case is passed everything is fine, but if the test cases failed that indicates that the software is in an error condition. So, where exactly is the error? which is the erroneous components? Especially when you are doing system testing or integration testing, your software or system might be fairly large. It takes some amount of domain knowledge experience and human effort to be able to

isolate the fault and facilitate the development team to be able to debug and test it once again.

So, we would deal mainly with test case design algorithms in this course. So, in the next module I will tell you what are the algorithms are and what are the mathematical models modelling software artifacts that we would deal with in this course.

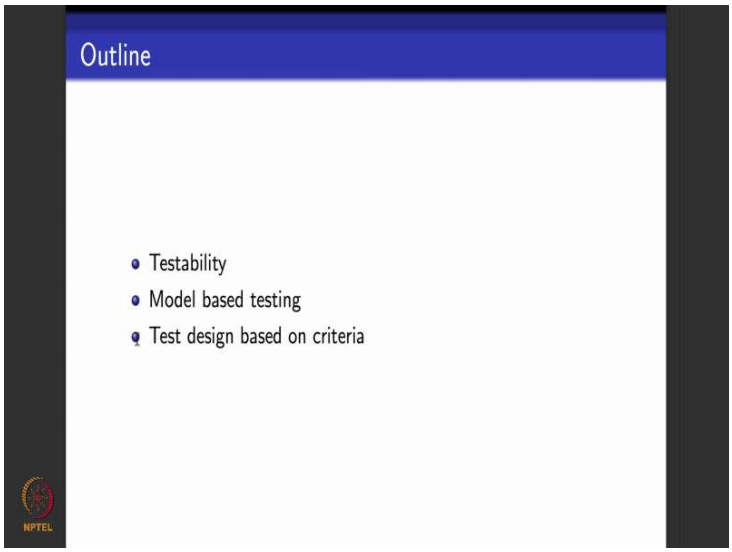
Thank you.

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

Lecture - 03
Software Testing: Testing based on models and criteria

Hello everyone. This is the lecture on the second module of first week. So, what will be seeing in today's lecture? We will look at it what it means to test the software.

(Refer Slide Time: 00:22)

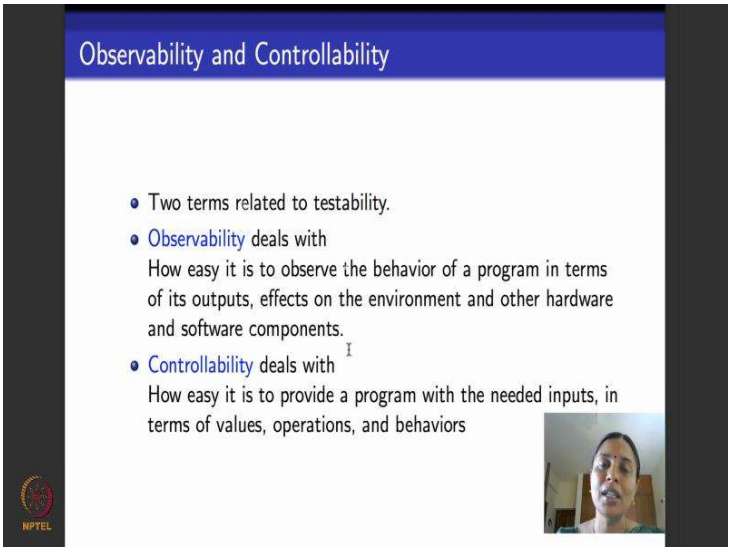


What is testability; we will also look at what is model based testing as its commonly perceive in the testing world, and how we will use model based testing when we design algorithms for test cases. As I told you lot of this course is going to be on algorithms and methodologies for test case design. And in fact we will be looking at what is call criteria based test case design. So, we will formally define what is criteria, and look at the various artifacts on which we will be defining these criteria to design test cases for.

So, I begin with introducing what is testability. So, what is testability? If you look at it in simple English terms it simply answers the question--- is the software testable or not. So now, if I ask the same question again in term what is it mean for the software to be testable, right? So, we say can I give inputs to the software; inputs a test cases to software? After giving the test cases to the software can I execute the software an observe the outputs. You might think it is a very obvious question--- when I write a piece

of core what is the difficulty about giving inputs to the software, executing the software an observing its outputs.

(Refer Slide Time: 01:37)



So, we will see what are the particular hiccups that can arise while doing this. Two related notions of testability are what are called observability and controllability. So, what is observability mean? Observability it tries to answer the following question. So, it says the suppose you give a software certain kinds of inputs and execute it, how observable is its behaviour,? How observable are the outputs that the software produces?

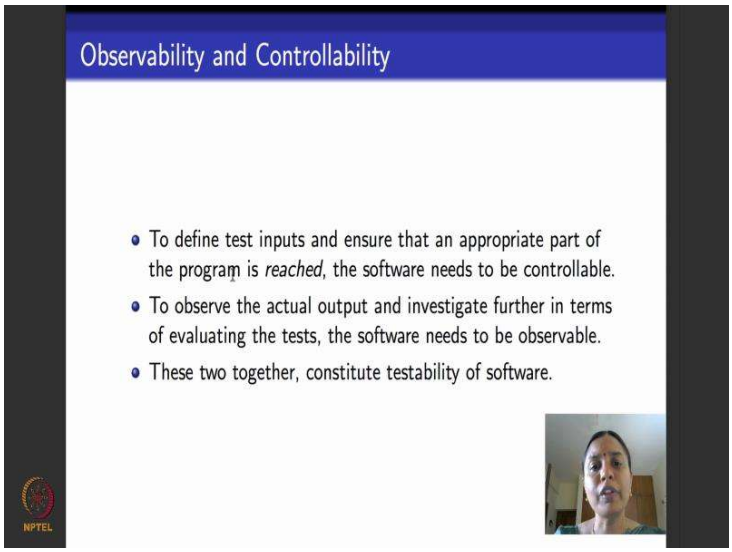
It could do we doing a lot of things silently and producing only the final output, whereas as a tester you might be interested in some intermediate outputs that the software produces. So, observability tries to answer this question about how observable the outputs produced by the software are and how much we can tweak the outputs that we want the software to produce towards testing it.

So, when I test the software I might want to observe a little more about its outputs, not actually the actual outputs of software. But I might want to know the values of specific internal variables, specific call and return values. Observability deals with all this. Another related term for testability is what is called controllability. What is controllability? Controllability says if a software controllable by giving it in some input and actually executing the software.

Again this might seem like a trivial question you might ask why is it so difficult to give inputs to a software. Let us take a particular case for example, where piece of software runs for some time and then at particular point and code it calls a particular function. So, I want to be able to let us say "integrate test" this function call. So, what I am interested in controllability is the fact that I should be able to give inputs to the softwares such that I can ensure that the particular function of procedure that is embedded deep inside the code is actually called for the values of the inputs. And not only is the function called, I now want to be able to observe what the function returns in turn.

So, controllability and observability help you to answer these questions and software. And, they are one of the several different quality attributes broadly call "ilities", that the people worry about while testing and writing software.

(Refer Slide Time: 03:51)

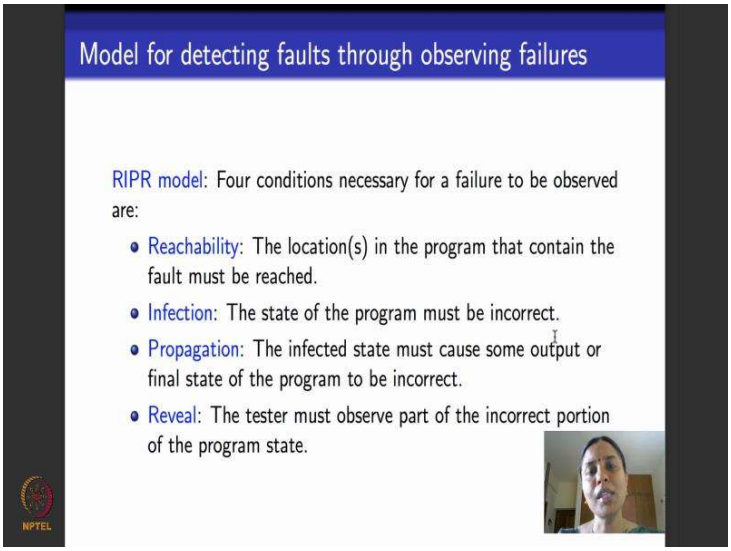


So, to define test inputs and to ensure that in appropriate part of the code is reached, like I told you an appropriate function is called and I will be able to in turn observe the value return by the function I say that the software needs to be controllable. When I want to observe the value that is return by the function and see how it gets passed on to the main callee program then I say that the software needs to be observable. Observability and controllability put together constitute testability of a software.

Obviously, for a software to be testabile, its testability quotient must be very high; it should be observable, it should be controllable. So, we will little more in detail

understand what this notion are; when people say a software is testable they usually talk about four parameters for observability and controllability put together.

(Refer Slide Time: 04:38)



So, these four parameters are what are called RIPR model. So, what is R? R stands for reachability. Reachability mean suppose I am testing, white box testing, a piece of software and I want to check whether a particular statement is reached, whether a particular function call is reached, whether a particular code segment is reached; I should be able to give test inputs that guarantee the execution of software in such a way that this particular target place in the software of goal is reached.

Now the next important thing is I: I stands for infection, what do we mean by infection? Not only do I want to reach that place I want to be able to execute those statements in the software. And suppose there is an error that is observed I want the error to be exercised or I want the error to be reached; I want the software to reach its faulty state. And I want to be able to give inputs in such a way that I can infect that particular statement in the software and actually pull out the error or the faulty states the software was in. Suppose I manage to do that, but the software still manages to run file and execute such that it acts normally, then even though the software is erroneous I, as a tester have not really observed the error.

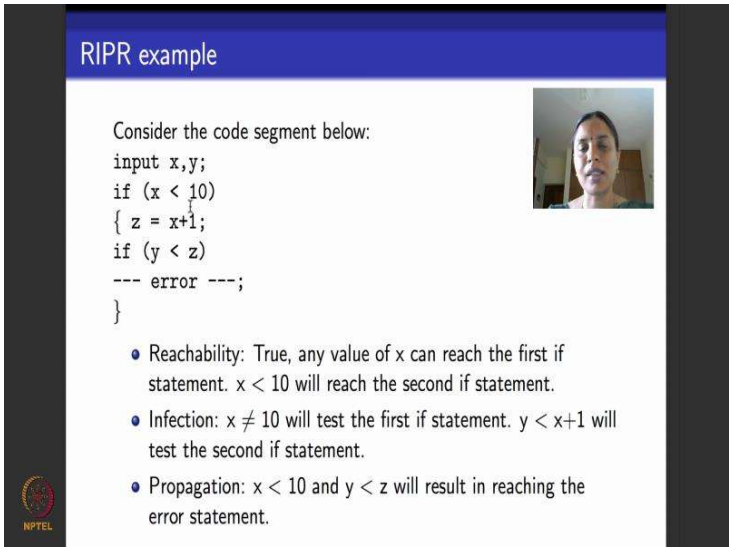
So, the next parameter that I am looking for is P which stands for propagation. So, I say not only much particular statement be reached and the error be revealed as infected; the

error should also be propagated to output, observable output, such that I know that there is an error that has occurred in the software. And it could be propagated and for some simple reason like you might just forgotten to write a print statement or forgotten to track this particular values the propagated error might not actually get revealed to the tester. So, at the last parameter in this RIPR model that I am looking for is for the error to be revealed; this tester will be able to observe the incorrect part in the program and correctly isolate and identify which part of the code was the error in.

So, to summarise we say that if I am targeting a particular a piece of code that I suspect to be erroneous or that I want coverage to be achieved; somebody told me you please test this piece the code thoroughly. You please test this function and the function called thoroughly. So, I am looking for four properties the software should satisfy for it to be observable and controllable.

The first one is that the particular target code fragment or segment should be reachable. The particular target code fragment or segment, if it has an error should infect and the particular test case that I give should make sure that the error state has actually occurred. This occurred error, in turn, should propagated to the software producing a different kind of output that reveals the error. And the revealed error should, in turn, help the tester to be able to identify and isolate the error as having occurred in that piece of software. So, software that needs this RIPR model property is supposed to be highly testable.

(Refer Slide Time: 07:48)



So, here is an example of an RIPR model: you consider the small code fragment that is given here I am not given the full piece of code here, I have just given a small fragment of the code. In this small fragment of the code there are two inputs x and y, and then they could be other statements lot of other commands and the software. But let us focus on this particular thing fragment of the code which says that for there are two nested if statements here. The first statement checks if x is less than 10 that it sets the value of z to be x plus 1. And then there is one more if inside that which says that if y is less than z then there is an error in the software. I haven't actually written what the error could be, I have just written it like a stub which marked it out as error.

So, suppose I want to actually; my goal is to be able to be actually reach this error statement and reveal the fact that this error as occurred. So, what does RIPR parameters do for this? So, let us look at reachability first. So, what is it mean for the first if statement to be reachable; any value of x will reach that statement assuming that there are no code fragments that change the value of x, any value of x that is given here will reach the first statement which is x less than 10. So, reachability can be thought of as being true, always true, there is nothing specific that I need to do.

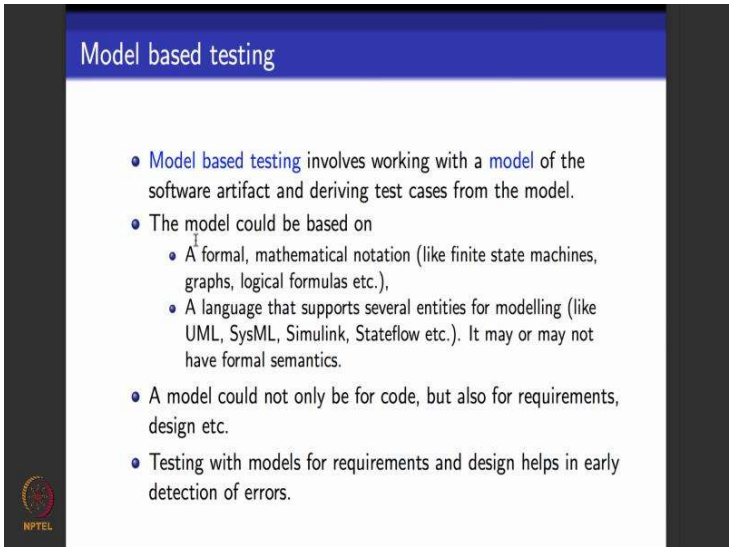
But to be able to reach the second nested if statement; suppose the first if statement test negative in the sense suppose I give a value of x that is not less than 10. So, this if statement, this predicate will written false and the code will exist this if statement it will never enter this if statement. If it does not enter this if statement it is not going to be able to test the second if make it positive and then reach the error statement. So, reachability for the second if statement will happen only if the first if statement returns true and the first if statement will return true if I give a value of x that is less than 10.

So, the predicate x less than 10 should be true for me to be able to reach the second if statement. So, what is infection? Infection means I not only reach the statement, I manage to execute that statement and make it true or false. So, again an infection for the first if statement is any value of x less than 10 like for example, or any value of x greater than 10 or even x is equal to 10 will manage to infect this first statement. For the second statement remember z is set to x plus 1. So, you must give a value of y that is less then x plus 1 to be able to test the second if statement. Now moving on, when it comes to propagation I should be able to go past the first if statement makes it true, go past the second if statement make that also true, and then only I will reach the error statement.

So, both these predicates x less than 10 and y less than z will have to be true for me to be able to reach the error statement.

I hope what reachability infection and propagation is very clear. So, in this piece of the example it is quite small, so it easy to be able to do what are the conditions for reachability, infection and propagation and so on. But for large fragments of code, where I have thousands of lines of code it is not an easy problem to be able to come up with less list of predicate for reachability for infection and propagation and to be able to solve them. In later modules when we look at testing criteria we will see how what it means to solve the predicate or a particular piece of code for reachability, for infection, and for propagation in detail.

(Refer Slide Time: 11:20)



I will now move on to looking at model based testing. What is model based testing mean? In the English sense of the term, model based testing means that you have a model of the software artefact; the artifact could be requirements it could be design, it could be architecture, it could be code, and then you design test cases by looking at the model. There are of course, several other uses of the models for software, well done models could be used to auto generate codes they could be used to early validate the system. There are several other uses for model based design and model based testing.

There are broadly two kinds of models: the model could be based on a formal, mathematical notation. Here are some popular modelling languages: finite state

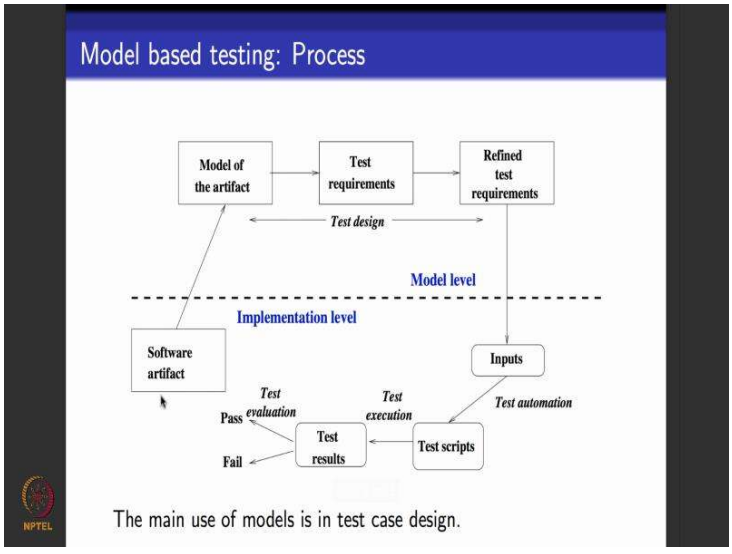
machines, graphs, logical formulae. As I told you in the last module there are several different verification techniques, like model checking theorem proving, they all use mathematical models of software artifacts to be able to verify a software.

Models it could also mean a particular domain language or specifically designed language that specifies all the artifacts that particular software is concerned. You might have heard popular modelling language like UML which stands for unified modelling language, SysML which is systems modelling language, Simulink, Stateflow which are modelling languages that are proprietary to a company called Mathworks which makes a popular tool called MATLAB.

So, these modelling languages support several diagrammatic notations for me to be able to module various software arifacts like use cases requirements design, control laws and so on. They may or may not have formal semantics. So, it is to be noted as I told you that the model need not be only for code; it could be for requirements, it could be for design, and so on. Testing with models for requirements and design: suppose I manage to have good, well defined models for requirements and design and I am able to design test cases for them, then remember that I am doing this kind of testing right at the requirements level or design level, before I write code.

So, if I find an error I find it early on in the life cycle and that is considered to have lots of benefits. So, what is the process of model based testing, how does it work?

(Refer Slide Time: 13:32)



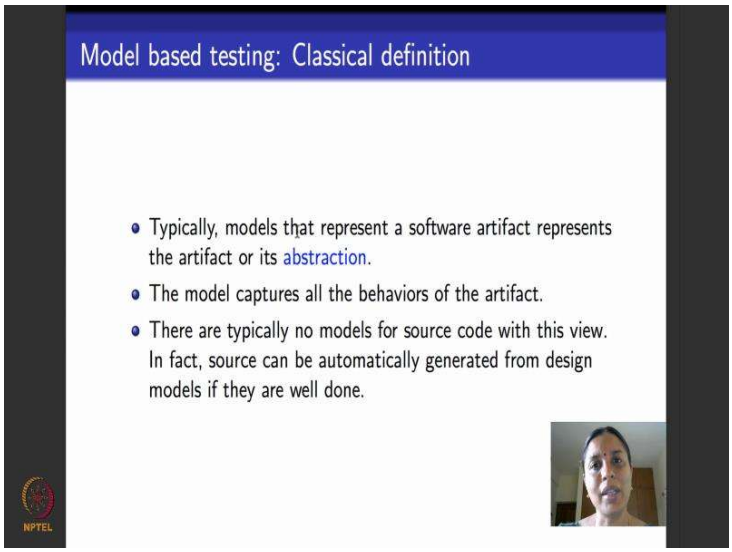
Let us begin here with this part: I have a software artifact with me under consideration that would be code that could be a requirements document that could be a design document. Code is typically executable, but many software companies have requirements of design document written in English. So, English documents cannot act as models. So, I take the software artifact and come up with the model of the artifact. The model of an artifacts is designed in a language that I have predisposal decided as suitable for the class of software that I am working with.

Once I have the model of this software artifacts I do what is called model based design and I can subject this model to several different uses, but what we will focus on today, is in this particular slide, is the use of the model when it comes to designing test cases. So, I take this model of the artifacts and then I also have a set of test requirements that have been given to me. Test requirements could say something like--- there are some highly critical requirements you please design test cases to exhaustively test for all the requirements. Test requirement could say something, like you cover this particular fragment of the model or a piece of code.

So, I take these test requirements and refine them with reference to the model that I have in mind. After I refine the test requirements I have done my test case design and I am ready to give my test case as inputs. Remember once I have a test case that has to be actually executed in code. So, I assume that I have an implementation ready and I move on to the implementation of the software. Once I have given my test case design I pass these inputs, make it ready for execution; make it ready for execution you remember in the last module we saw this step called test automation. So, I do test automation and get that test scripts which are ready for execution then I use the tool to execute this test scripts and observe the results. And then I conclude the test cases have passed or failed.

So, this is the normal process of testing that we saw the only difference here is that while doing designing of test cases I have a particular model of the software and hand and I design test cases for a set of test requirements on that specific model of the software. Here my main use of models is to be able to do test case design.

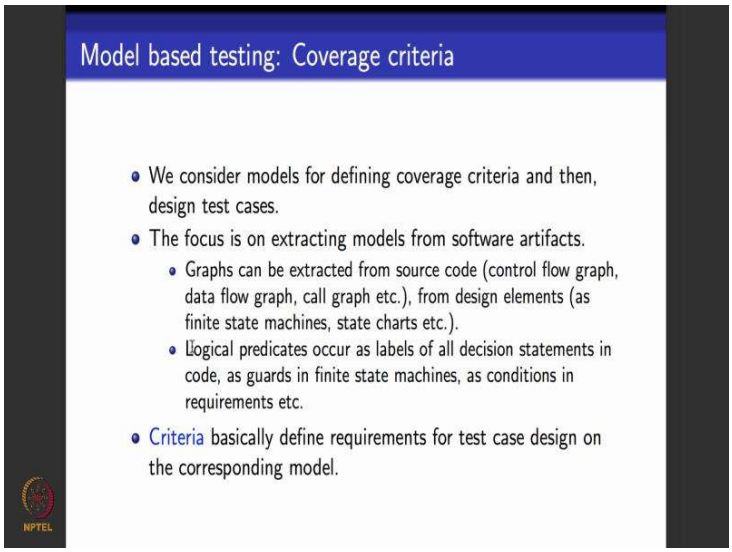
(Refer Slide Time: 16:03)



So, the classical view of model based testing or model based design is to say that models could represent a very high level abstraction of the software, it need not represent the actual code. In fact, code is almost always never represented as a model. Code is represented in the programming language of your choice and is always an executable entity. Models may or may not be executed, they could be static artifacts that represent the code. Typically a model is supposed to capture all the behaviours of the particular software artifacts and it is for the user to ensure that the model of the software is accurate and good enough to meet the required expectations of the model. If the model that you do of the software artifacts is itself wrong, then obviously the test case and everything else that you do with the model is also going to be wrong.

So, is up to the user to ensure that he or she knows the modelling notation well and is modelled software well enough to be able to design all the test cases. So, we do not view the classical view of model checking in this particular course, we will view model based testing for doing coverage criteria based design in this course.

(Refer Slide Time: 17:13)



So, what do we do? We first focus and represent the software artifacts as models, we typically work with several different models. So, where do these models come from? So, graphs are one kind of models that I want to work with. Where do I get graphs from? I can get graphs from source code. You might have heard of control flow graph or a flowchart corresponding to a piece of code. In fact, there is something called data flow graph which represents the control flow along with tracking of the variables that are involved at a particular piece of statement in the code.

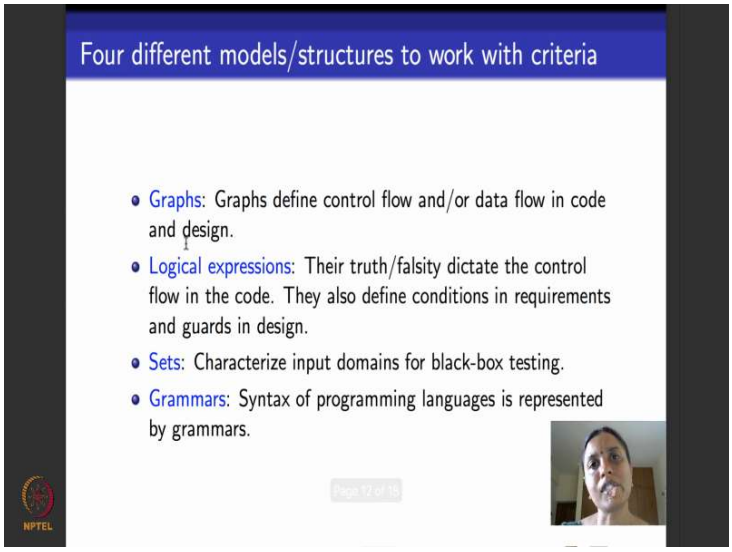
And then there is an notion of call graph in an inter procedural call graph when the code is modularized into several functions or procedures or method. I can also get graph from design; lots of designs these days are represented using UML notations which are basically finite state machines, state charts, all these are some specific kinds of graph. They have several different specific parameters to each of them, but they are all basically graphs.

Another class of models that we will be working with what are called logical predicates. Where do logical predicates come? If you see a typical code or an implementation is studded with them, at every decision point in the code. What is the decision point in the code? It could be an if statement, a while statement, a for statement which says that there is going to be some kind of the branching in the execution of a code, some kind of the choice in the execution of code. At every decision point in the code there is a predicate.

Based on whether the predicate is true or false the code execution takes one path or the other.

So, what we will try to do is we will try to work with graphs, logical predicates and a few other things is a models of software and design test cases based on these models, based on certain criteria.

(Refer Slide Time: 19:09)



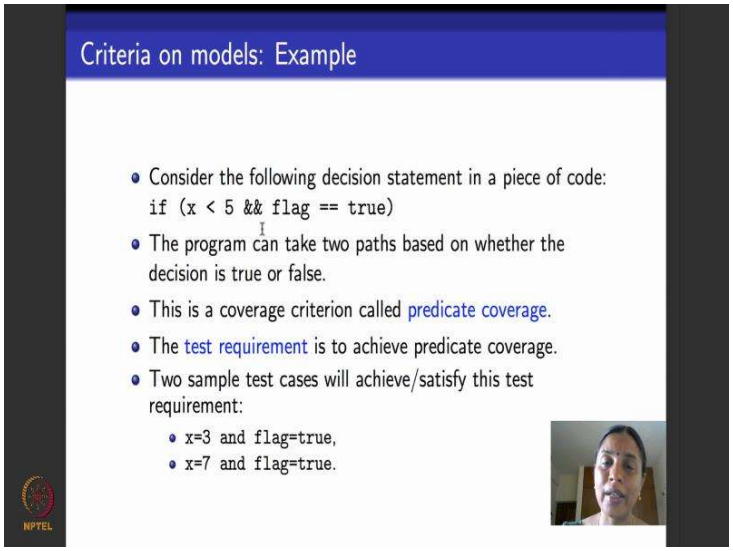
So, what are the four different model structures is that we will be working with as a part of this course. We will first begin with graphs next week. So, graphs typically in software define control flow, they also define data flow and as I told you little earlier they also could be graphs that depict calls of procedures, one procedure calling the other. So, we look at graphs as models representing such entities about a piece of software and then we will define criteria based on that and understand how to design test cases based on criteria.

Then we will look at logical expressions as they occur software, as they occur code, as they occur guards in design that represented as finite state machines, as they occur in requirements. And we will design test cases that check when a logical predicate could be true or when a logical predicate could be false. We will also, for the point of view of black box testing, look at sets of inputs that are given to a software and design test cases based on partitioning of those sets. So, sets would be another structures that we would be working with in this course.

And finally, when we look at coverage criteria we will also look at the underline syntax of programming language. So, every programming language as you know it could be C, it could be Java, it could be Python as an underlining grammar which tells you what is allowed in terms of writing in that programming language, how to write programs in the programming language. And when I compile a piece of software in the process of compiling I also check at the particular program adheres to the underline syntax or not. So, you can do mutation testing which exploits the syntax of a particular piece of programming language and designs test cases by manipulating inputs that adhere to the underline syntax or grammar of that particular language.

So, to summarise this slide, what we will do is we will look at software artifacts as four different structures or models: we will first consider them as graphs, and then will consider the logical predicates that occur in the code, then we will look at sets of inputs and outputs, and finally we will look at the grammar or the syntax of the programming languages. For each of these, we will design what are called testing criteria and also see algorithms on how to design test cases to test for these criteria.

(Refer Slide Time: 21:39)



So, here is a simple example to understand what criteria means. Let us say somewhere in your piece of code there is this following decision statement. How does this read? It says if x is less than 5 and a Boolean variable called flag is true then you do something. We are not interested in what we do, I just want to focus on this particular if statement.

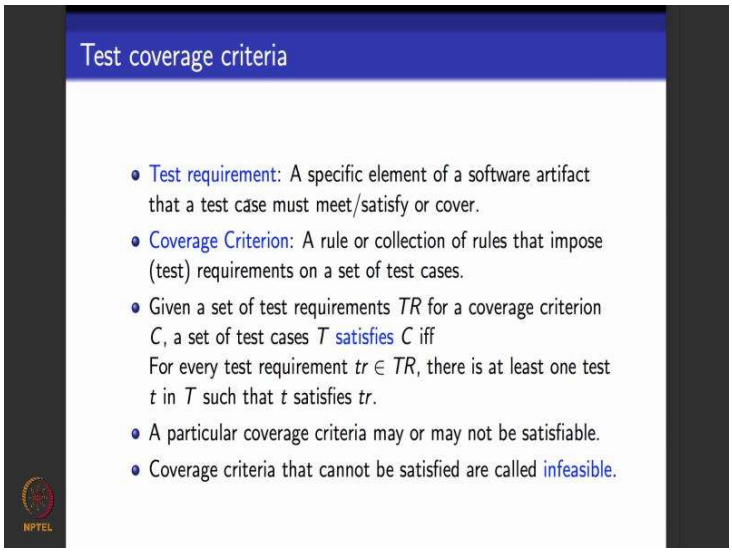
Now, what is the label of this if statement? The label of the statement is this particular thing right x less than 5 and flag is equal to true. It involves two kinds of variables: it involves, maybe, an integer variable x whose values compared to 5, and it involves a Boolean variables flag which is tested to be a equivalent to true. So, this whole thing is what we call a predicate. So, let say suppose this predicate turns out to be true that is x is indeed less than 5 and flag is also equal to true then you say that the if statement takes the then branch of the code. And suppose this predicate is false then you say that the if statement takes what is called the else branch of the code.

So, how will I make this predicate true? I say, you write two kinds of test cases one that makes this predicate true and one that makes this predicate false; that way I would have exercise this if statement for predicate coverage or branch coverage. When I cover a predicate I also cover then branch of the if statement and predicate is true and I cover the else branch of the if statement when the predicate is false.

So, for this particular if statement: suppose I set x to be equal to 3 and the value of the variable flag to be equal to true then the whole predicate with this and operator evaluate to be true. So, this would exercise the then part of the if statement. And suppose I give a value of x to be 7 and let say I set the Boolean variable flag to be true, then this predicate would evaluate to be false, because this first condition will evaluate to be false. So, then this will mean that this whole if statement will have to exercise the else part of the code that is present after the if statement.

So, I say this is my test requirement; my test requirement is to be able to achieve predicate coverage on this if statement and these are the two test cases that achieve this test requirement. We will see how to define such coverage criteria based on graph, based on predicates, based on sets of inputs and how to automatically design test cases that will tell you whether the particular coverage criteria is achieved or not, and if it is achieved how is it achieved.

(Refer Slide Time: 24:21)



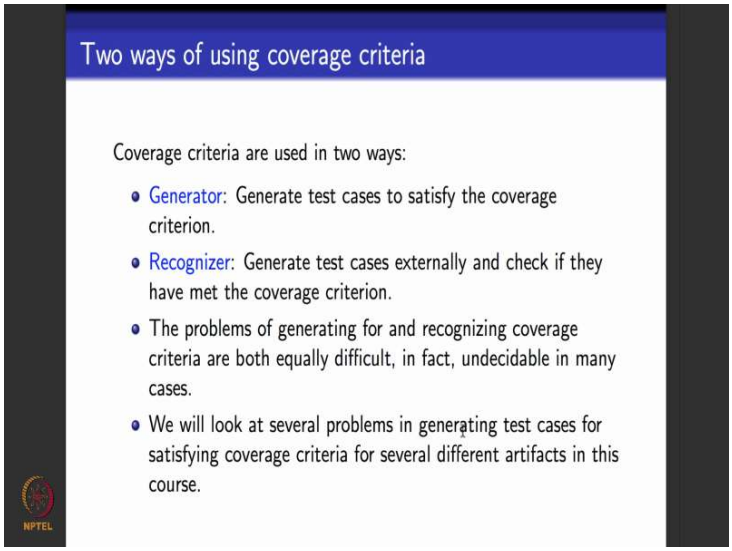
So, here are some definitions. What is the test requirement? A test requirement is basically a requirement that you say about a test case. You gave a thing like please design a test case to cover this if statement, to execute this if statement once for the then part once for the else part. You might say, please design a test case that will execute this while statement at least three times. You know these are all test requirements. They basically give you requirements on how to design test cases.

What is a criterion? A criterion is a rule or a set of rules that impose certain requirements on a set of test cases. Like in the previous slide if you say my test requirements is to test this if statement to be true once and to be false once then the criteria is what we call as predicate coverage. So, given a set of test requirements for a coverage criteria C, we say that a set of test cases satisfies the coverage criteria C if every test case in that set of test requirements satisfies the coverage criteria C; there is nothing more to it.

So, it is to be noted that a particular coverage criteria may or may not be feasible. Like for example: you take an if statement that occurs in something like this, let us say this if statement is labelled by a predicate that can never be made false. For example right, it could be labelled by a predicate that is always true or a tautology. In which case I say that predicate coverage criteria on this if statement becomes infeasible, because predicate coverage says that you make this predicate that labels an if statement true once and false once.

The predicate is such that it can never be made false. If it can never be made false then I cannot achieve predicate coverage. When I cannot achieve predicate coverage then I say that the particular coverage criteria that I cannot achieve are what is called infeasible. So, it is undecidable problem to check or an arbitrary coverage criterion whether it is feasible or not. What we will see several heuristic or techniques that will let you decide implicitly whether particular coverage criteria is feasible or not; and if it is feasible to be able to design test cases that will satisfy the coverage criteria.

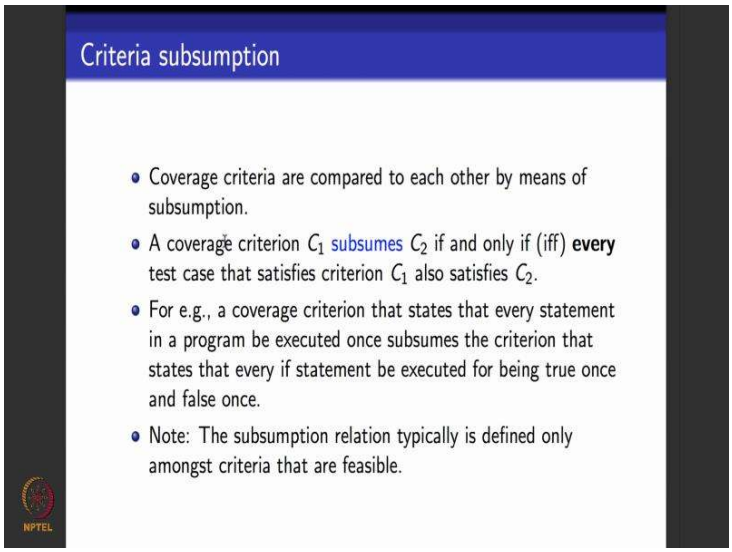
(Refer Slide Time: 26:39)



So, how do we design test cases that satisfy the coverage criteria? There are two base to do it: one is to automatically generate test cases that satisfy the coverage criteria, and the next one is to generate test cases externally; externally means you generated arbitrarily and then somebody gives you the coverage criteria. Now you take this set of generate test cases and the coverage criteria and you check if these test cases meet those coverage criteria so that is done by what is called a recognizer. And directly given a coverage criteria generating test cases for that coverage criteria is done by what is called generator.

Both generator and recognizer for testing based on coverage criteria are in their most generality, undecidable problems. In fact several times the algorithms for decidable fragments also have high complexity, but we will look at algorithms, nonetheless, without worrying about what the complexities and see how to use them to be able to generate test cases.

(Refer Slide Time: 27:47)



Now, suppose I have a particular thing and I define several different coverage criteria on a particular test requirement. So, I should know how do each of these coverage criteria compare to each other. Like for example, I in a piece of code I could have two different coverage criteria: one coverage criterion says that- you design a set of test cases that will execute every statement once. Another coverage criterion says that- you design a set of test cases which will execute every branch for the then part and for the else part once.

So, how do I know which of this is better and the work that I do for achieving one coverage criteria, implicitly also meets another coverage criteria. So, the notion of subsumption comes to our rescue here. So, we say that a particular coverage criteria C 1 subsumes another coverage criteria C 2 if every test case that satisfies criterion C 1 also satisfies criterion C 2. Like for example, suppose I told you there was a criteria with says executive every statement once right, which means executive every statement in the program. And let us say there was another other criteria which says that is executed in every branch that you encounter, execute the true part executive the false part.

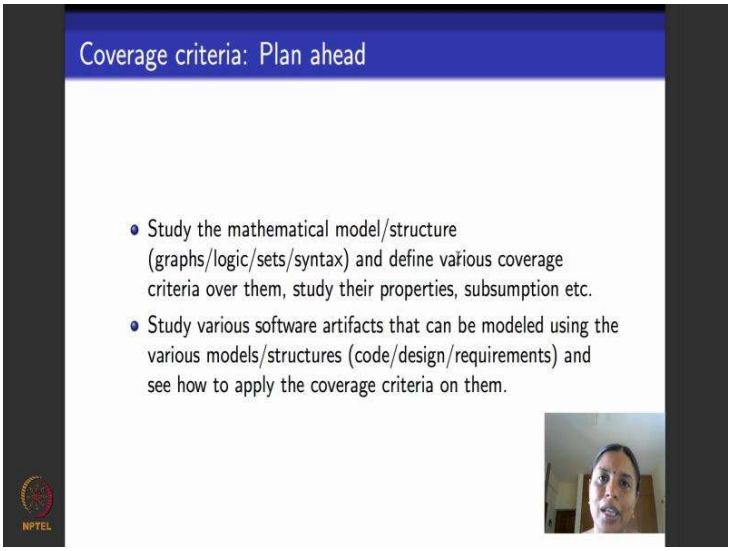
It is obvious that the first coverage criteria with says executive every statement one subsumes the second coverage criteria, because if I execute every statement once I would also be executing the statements of the then branch and the statements of the else branch.

So, this is how I use the notion of subsumption to be able to compare coverage criteria. It is important to be able to compare coverage criteria to reduce the burden on you

repeatedly generating test cases for different coverage criteria that are actually subsumed by the other coverage criteria. So, it is important to know if I generate test cases for one kind of coverage criteria what other parts of other different coverage criteria I have already achieved because of this. So, the nation of subsumption will help you to answer these kinds of questions.

One important note to observe that is that suppose a particular coverage criteria is infeasible then I really do not look at what subsumption for that coverage criteria, I define subsumption only for feasible coverage criteria.

(Refer Slide Time: 30:14)



So, what is the plan ahead for the next weeks? What we will do is that we will consider software as various mathematical models or structures. So, what are the four structures that I said we will look at? We will look at graphs as model software artifacts, we look at logical predicate that occur in software artifacts, we will look at sets of inputs and outputs that are given to the software, and finally we will look at the underlining grammar or the syntax of software.

So, each of these models as structures we will define coverage criteria and discuss algorithms that will let us you design test cases to achieve coverage criteria. Of course, we look at subsumptions of coverage criteria and so on and so forth. So, what we will do first is we look at the model at an abstract level. Just as a graph, as a predicate, and then define algorithms for coverage criteria. Post that we will look at various software

artifacts; so we will see how to take code and model it is a graph, and what are the coverage criteria that we look at the graphs and how do those help to test various aspects of the code. Then we will take design and model it as a graph, and then we will see what are the coverage criteria that we looked at for graphs that will be relevant to design test cases for these designs.

So, similarly when we go to logical predicates, we look at logical predicates as their occur in code, look at coverage criteria and then see how these coverage criteria means testing which parts of the code. So, we look at these mathematical structures look at coverage criteria separately, then we look at how to model software artifacts using these structures and what do the various define coverage criteria mean for these structures.

So, next week when I begin my first module we will look at graph as models of software artifacts and define coverage criteria based on graphs so that will be the next lecture.

Thank you.

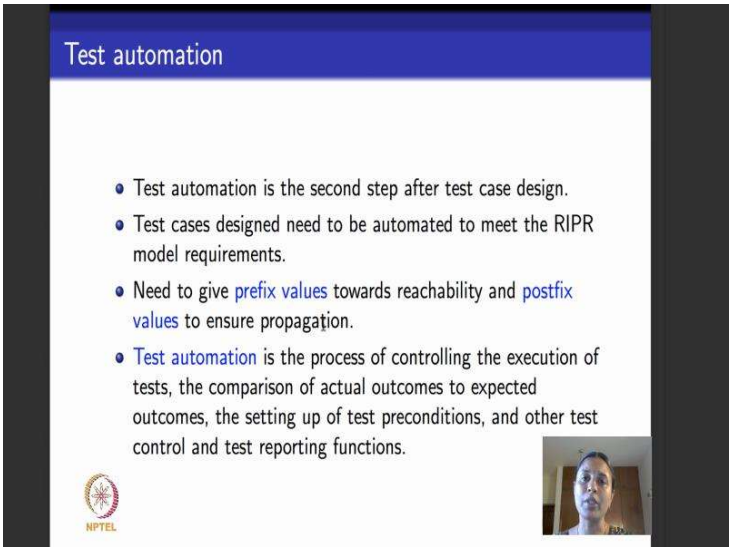
Software Testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture – 04
Software Test Automation: JUnit as an example

Hello everyone. We will do the last module of the first week. In today's module I would like to concentrate on the second step of testing. If you remember in the previous modules we saw that there were four steps involved in software testing: your design test cases, then you automate test case design- that is you make the test design test case ready for execution, followed by the third step which is the test execution process, and finally when the test execution results are recorded you evaluate the test.

So, the second step in these four processes is that of test case automation. And the focus of this lecture is to understand what is test case automation and what exactly goes into it.

(Refer Slide Time: 00:54)

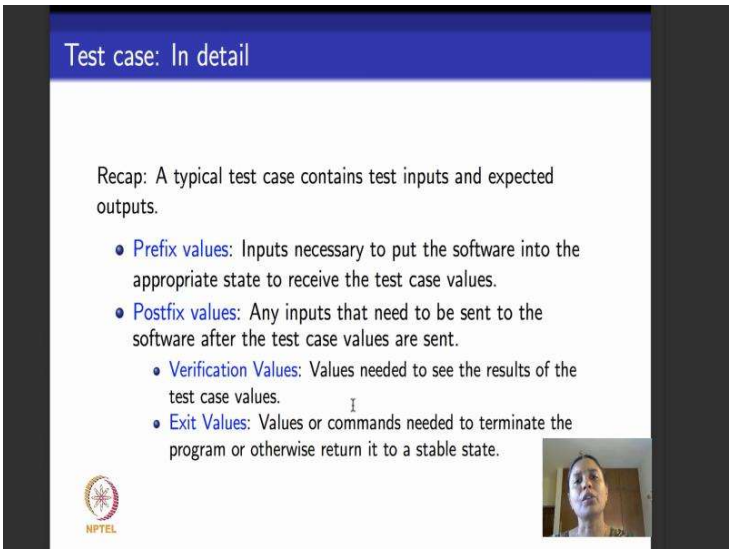


In the previous module we saw that testability of a software, in turn, translates into observability and controllability, and these are measured using the RIPR model; R for reachability, I for infection, P for propagation and the final for R for revealing. So, how do I ensure that reachability propagation and revealing are done? I ensure that these three are done in the step of test case automation which is what we will look at in this module. So, how do we do that? We have to give what are called prefix values to ensure

reachability of a particular piece of code, and then once that test case design exercises that piece of code we have to give postfix values to the test case design to ensure that if there is an error that gets propagated outside.

So, all these things are what are called test automation. To formally define test automation it is the process of controlling the execution of tests, and actually ensuring the reachability is done by giving prefix values of preconditions. Then, you execute the test and compare the actual out come to the expected outcome and then you know report the results of your test case execution.

(Refer Slide Time: 02:11)



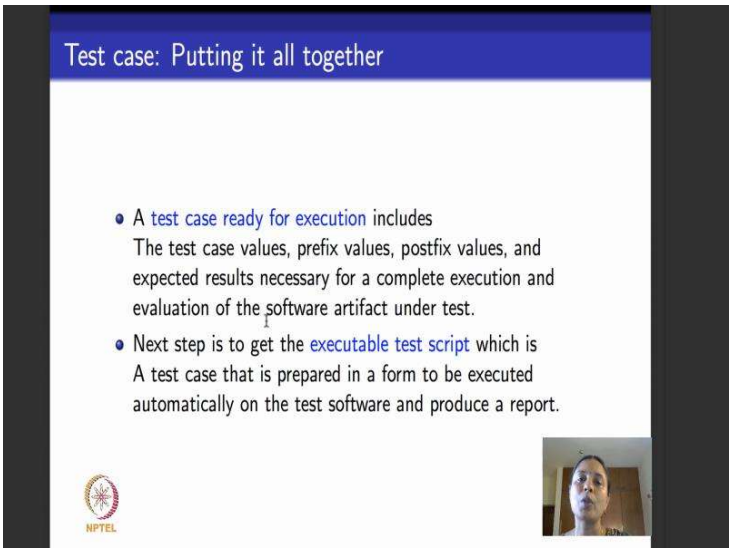
So, we will recap what a test cases and then see what are the add ones that we have to do to a test case to make it ready as an executable test script. If you remember from the first lecture what was the test case; test case basically contains test inputs and expected out puts. And if the expected outputs match the actual output after execution then you say that the test cases passed otherwise it is failed. This is a raw test case that has been designed.

Now, to make it into an executable test script the process of test automation has to add prefix values and postfix values to this test case. So, what are prefix values? They are basically inputs that are necessary to put the software or the software artifact into an appropriate state so has to be able to receive the actual test case for execution. And after the test case is executed, postfix values come into picture. What a postfix values? They

have values that are necessary so that the results of execution are sent to the software as an observable value by an external user.

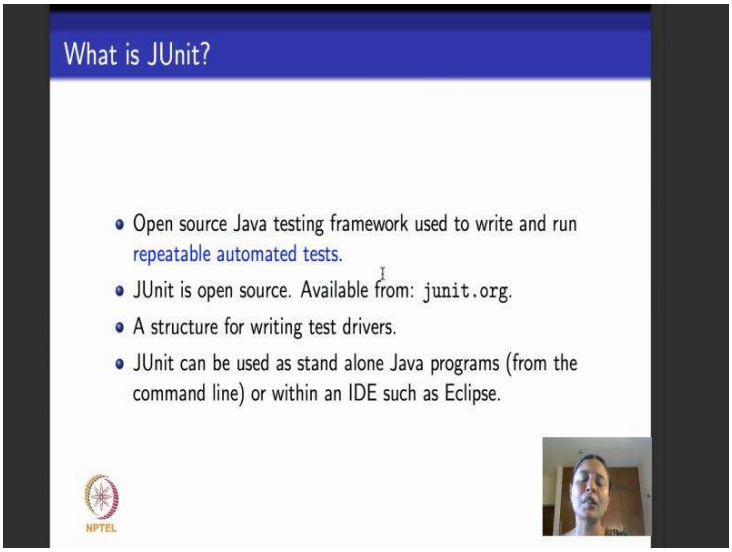
Postfix values intern bifurcated into two categories: verification values and exit values. What are verification values? Verification values basically tell you that an exception has occurred this test case has passed or it has failed and it is a value that clearly tells you what is the result of test case execution. And what are exit values? Exit values are basically values or pieces of code that are needed to make sure that after the test case is executive the code appropriately finishes its execution fully and exits so, as to retain and reveal the error state, if there was any present during execution.

(Refer Slide Time: 04:04)



So, now if you take the raw test case as it was designed and make it ready as a test script to be executed what are the summary of things that it has. It has the actual test case values, the prefix values, the postfix values, and as I told you it also has the expected outputs. So, when we put it all together and get it ready, a final result at the end of the test automation step should be to be able to get an executable test script, which is a test script that contains all these values and can be executed directly on the piece of code that it to being tested on.

(Refer Slide Time: 04:45)

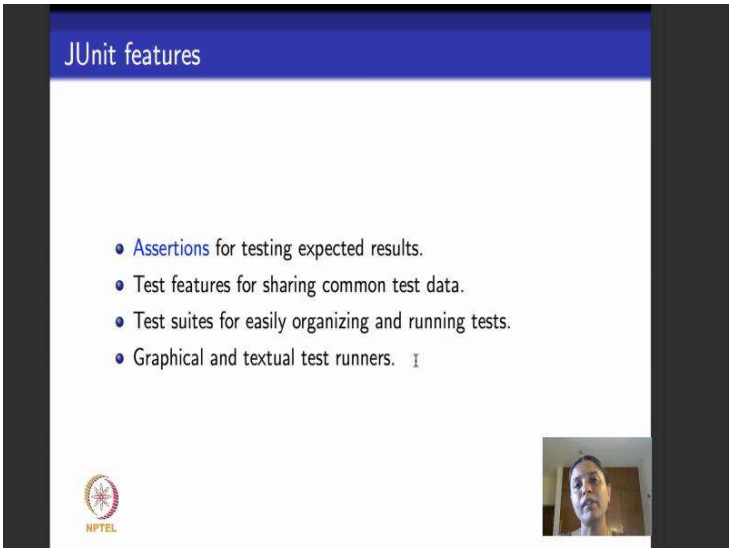


So, I will explain in detail how to give prefix values, how to give postfix values, and what happens in the process of test automation by using this open source tool; very good open source tool called JUnit as an example. Later in the part of the course when we see several examples of Java programs or C programs you could use the JUnit framework to be able to experiment with the test cases that you have designed as a part of assignments or other lectures that we will see in the course.

This module is not meant to be an exhaustive introduction of JUnit, but it is more meant to be used to interpret JUnit as a test automation tool, as a framework to understand how test automation works. In that process you will also learn some of the commands of JUnit which will be useful when you run your own experiments with JUnit at later points in the course. So, what is JUnit? It is an open source testing tool very popularly used in the industry. You can download it from Junit.org. And what are the things that it supports? It is very good for writing and executing test scripts.

So, once you design a test case you can automate it using JUnit and you can execute it and evaluate the results also using JUnit. It can be used as a standalone entity on Java programs or it can be used within an IDE like Eclipse.

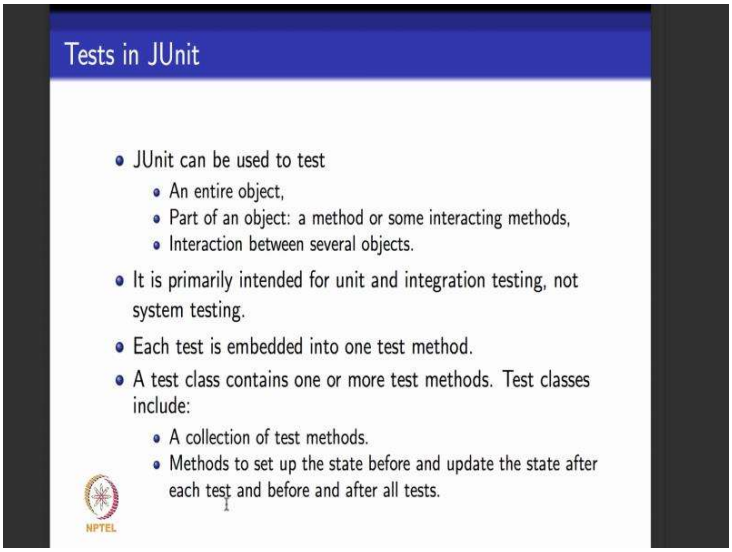
(Refer Slide Time: 06:13)



So, what are the high level features of JUnit? JUnit supports what are called assertions. How are assertions used? Assertions are basically statements that will always return true or false. You can view an assertion as if it returns true then everything is fine, the test case is passed. If an assertion returns false then there is an error that has been found and you can use this assertion to be able to reveal the error to the tester. So, we will assertions, how to use assertions through examples in JUnit.

JUnit also has test features for sharing common test data. Let us say two people are writing a common piece of code and they want to be able to share the test cases that they are writing then you can use JUnit to be able to do that. And JUnit also has suites for easily organizing, running, executing and observing tests. And of course, like many other tools it has a textual interface and it also has graphical interface.

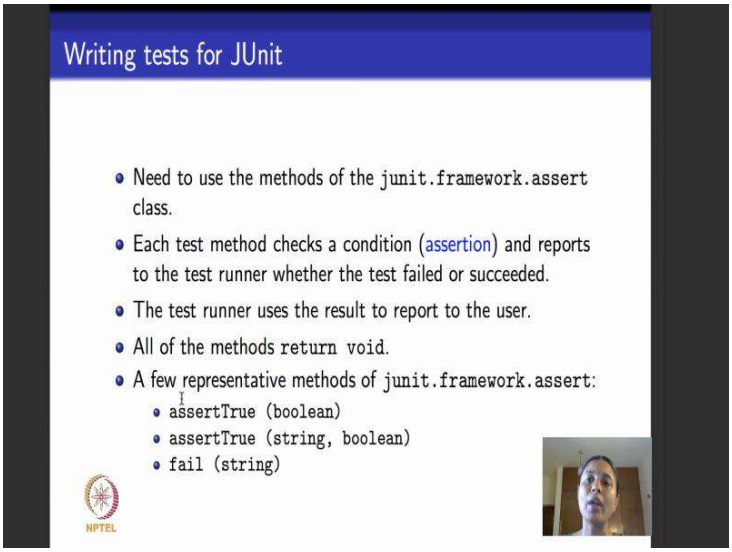
(Refer Slide Time: 07:11)



So, what can be JUnit be used for testing? The main thing is JUnit used for testing Java programs. So, it can be used as a very good unit testing tool or it can be used as a very good integration testing tool. It obviously cannot be used as a system testing tool. In system testing if you remember what I had told you, we take the inputs put the system is a part of the input server, connected to the database and let the outputs be observed as commands and so on.

JUnit being a testing tool for Java cannot be used for system testing, but can very well be used for unit testing and integration testing phases. So, it can be used to test an entire object, it can be used to choose just one method a certain interacting methods within an object it is up to you. So, JUnit has what are called test methods and each test is embedded into one test method, then it has a test class that contains one or more test methods. Test class, apart from this, can also have method that I used to setup the software to the state before and update the state after each test. So, they are these methods can also be used to do the prefix and postfix values that I had telling you about. And then there are code test methods which actually contain the test cases that have to be executed.

(Refer Slide Time: 08:34)



So, how do I write tests for JUnit? The first thing that we will understand while writing test using JUnit is to be able to use assertions. As I told you a little while ago what is assertion used for; assert is like a debug but in the context of testing. So, I say assert something if this test case passes, assert something if this test case fails. Assertion is always a Boolean expression. The value inside an assert always evaluate to true or false.

The implicit understanding while writing assertions is that if it evaluates to true then everything is fine your test cases passed so you can move on. But if an assertion evaluates to false then it indicates or it might indicate an error state. So, you will typically put a print statement there, saying what is gone wrong and you throw an exception or you print an appropriate warning and so on. So each test method that is used inside JUnit basically checks for a condition which is nothing but the assertion. And reports to the main test runner method about whether the test is passed or failed.

The test runner method uses the result of this assertion failing of passing to be able to report the result to the end user. Here are some examples of how asserts run. So, it is present in JUnit or framework or assert. So, you can do something like

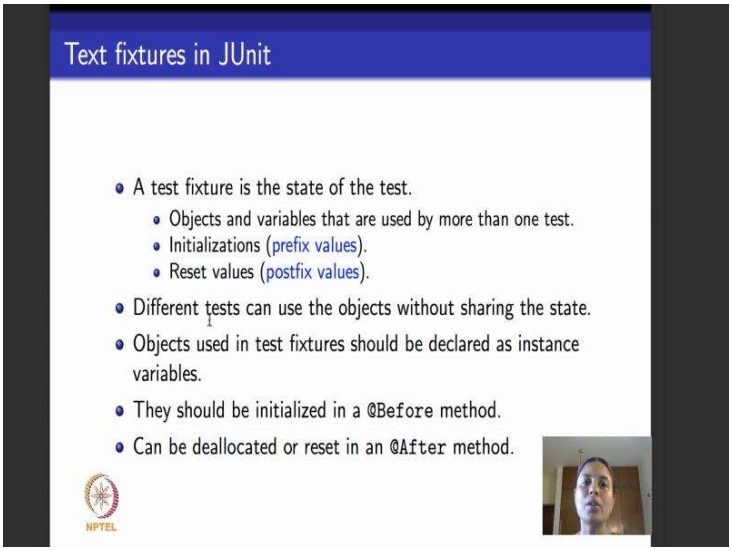
assert true Boolean.

So, this just says that if this Boolean predicate inside this asserts true evaluates to true then you just quietly say that it is true. If it evaluates to false maybe he will give a warning and terminate if it require.

The second asserts true test two arguments it takes a string and it takes a Boolean predicate. The idea here is that if the Boolean predicate evaluates to be true then you do not do anything. And if the Boolean predicate evaluates to be false then you print the string. So, it can be used as a way of warning the user.

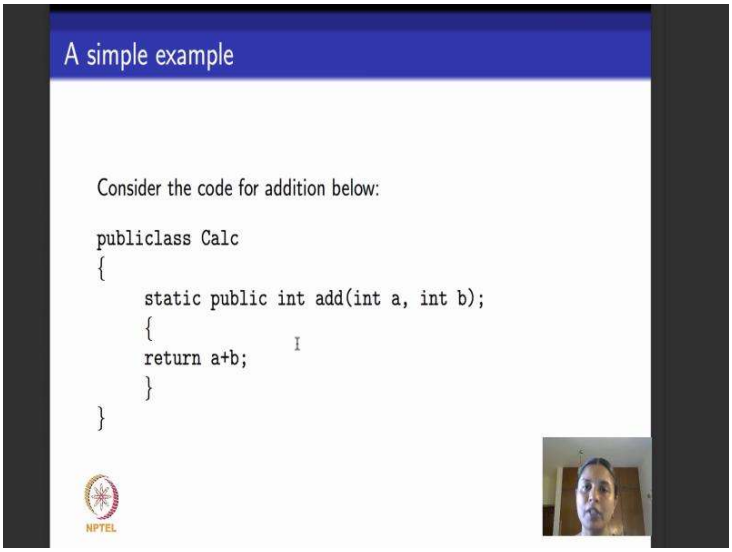
A third assertion is what is called fail assertion which basically says only if it fails, if a particular thing fails then you output string as a warning to the user. So, when we see examples of code that we write with Junit, I will show you examples of how to use all these assertions.

(Refer Slide Time: 10:58)



So, now how to write a test fixture? Assertion is a core component which tells you when a test cases passed or failed. Now we have to write still prefix values, postfix values, how do you do that. So, how do I do that? Prefix values are initialized in what is called before method; 'at before' method and postfix values are given in a method called 'at after' method- the names are very intuitive and very easy to remember.

(Refer Slide Time: 11:27)

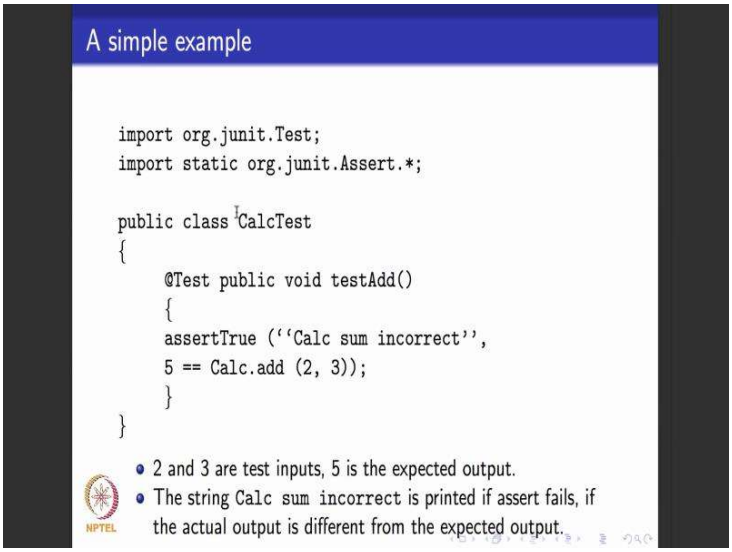


So, I will first walk you through a couple of examples. We will start with the simple code for addition as an example, then I will look at an another example which returns the minimum element in the list, and tell you how to write prefix values, how to write postfix values, how to give the test cases, and how to write assertions that will output the result of the test case passing or failing.

So, we will start with an example of a code that does addition. So, here is an example of a code that as does addition there is nothing complicated to it, it takes two integers a and b and it returns the sum of a and b which is a plus b. Now suppose you are given the task of testing this code. So, let us recap and understand what a test case is.

So, test case should give inputs and it should give expected outputs. What are inputs to this piece of code? Inputs are one value for a and one value for b, and expected output is the actual sum of a and b. Now what you have to do is a part of the automation you say- that you please take these inputs and compare it to the expected output. If the actual output matches the expected output fine, everything is working fine, but if the actual output defers from the expected output then you use assertions to be able to flag the fact that the test case execution has failed.

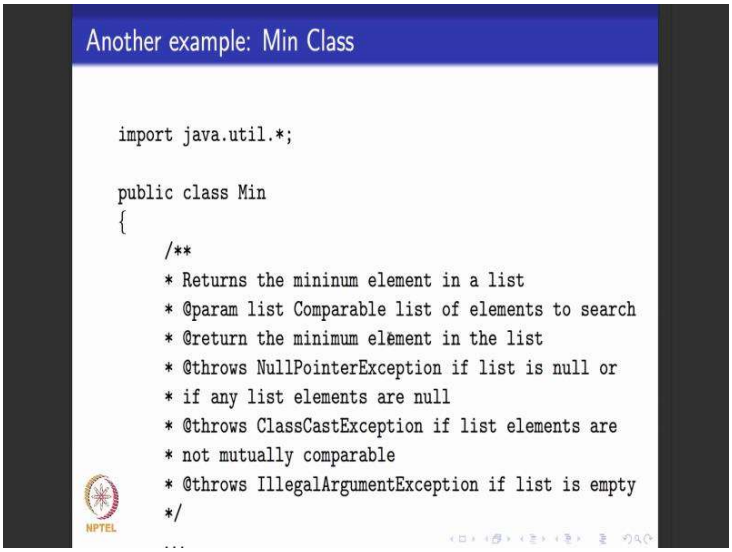
(Refer Slide Time: 12:50)



So, how will you do it in JUnit? This is how you do it in JUnit. So, these has standard things that you have to import, every time you will write any program any test method using JUnit for test case automation and execution. So, you import what is called JUnit or test and you import this library of assertions. So, I am writing something called the test for the calc method. How do I do? I write these two statements the first statement says that- you assert true and it gives a string, and the second statement actually runs the addition code that we saw in this slide with two actual test inputs--- 2 and 3. And if 2 and 3 are the test inputs assuming that addition works correct what is the expected output? The expected output should be 5. So, it compares it to 5.

So, what it says is that you run this add program on inputs 2 and 3 and check if the result that is outputted by the add program on these inputs is actually equal to 5. If it is actually equal to 5 then you exit, test cases passed. But if it is not equal to 5, if the calc program actually has an error; in this case it does not have an error because it just a simple program that returns a plus b. But assuming that it does have an error you write it for some other program, then this assert true will take over and it will output this string. So, it will say that calc sum is incorrect. So, it will say that there is a error somewhere in the code.

(Refer Slide Time: 14:22)



So, now we look at another example, slightly longer than that simple toy addition example. So, this is an example of a piece of code, Java code, that written the minimal element in a list. So, this code is very well debugged; in the sense that it takes care of all exception conditions, it takes care of what if the list is empty there is nothing to written, what if the list has entities that cannot be compared at all. For example, suppose the list has a string and a number, the list has a string and a Boolean value--- there of different types they cannot be compared.

So, in all these cases this code is meant through several different kinds of exceptions. So, these are what are called a part of debugging. Whether developer himself takes care of all the corner cases like wrong inputs, wrong data types etcetera which need not be taken care of by a tester. So, this code is well written to be able to take care of all the exceptional cases. Of course, the tester's job would be able to test for all these features also as we will see through examples, but we will first see what the code does and what are the various exceptions that it takes care of.

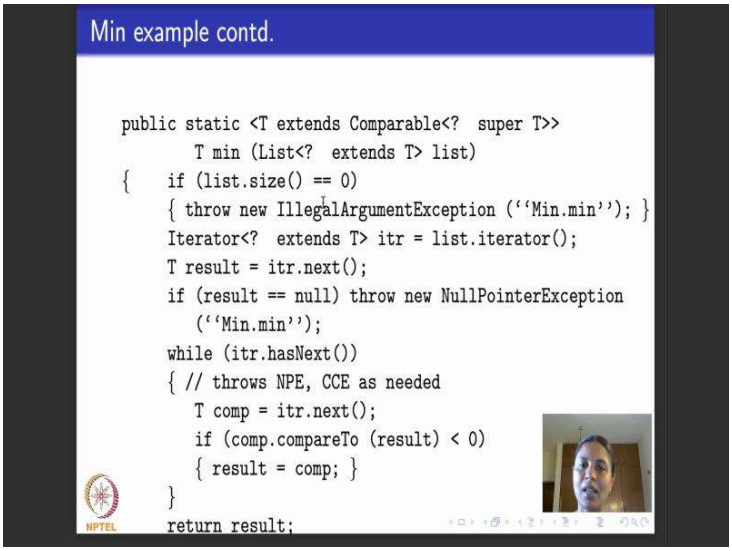
So, this class is called min and as I told you it returns the minimum element in a list and it takes as input comparable list of elements. What I said is that all of them should be comparable, we should all be a list of numbers or they should be a list of strings that I can compare them with respect to the lexicographic ordering. They should not be

incomparable types. And what is the main functionality of this class min? It supposed to return the minimum element in the list.

As I told you this class min is also supposed to take care of exceptions, like wrong inputs, incomparable inputs and so on. So, it has several exceptions. The first exception that it has is what is called a null pointer exception. It throws the null pointer exception if the list is empty or if any of the elements of the list are empty. And the second kind of exception that it throws is what is called as ClassClastException which it throws if the list elements are not comparable to each other. As I told you one is a string and the other is a Boolean constant how do you compare them, so you have to throw an exception. It also throws an illegal argument exception if the list that is passed to it is empty, there are no elements to compare.

So, I have split this code across two slides because otherwise it would not be readable. So, we will move on and look at the rest of the code in the next slide.

(Refer Slide Time: 16:54)

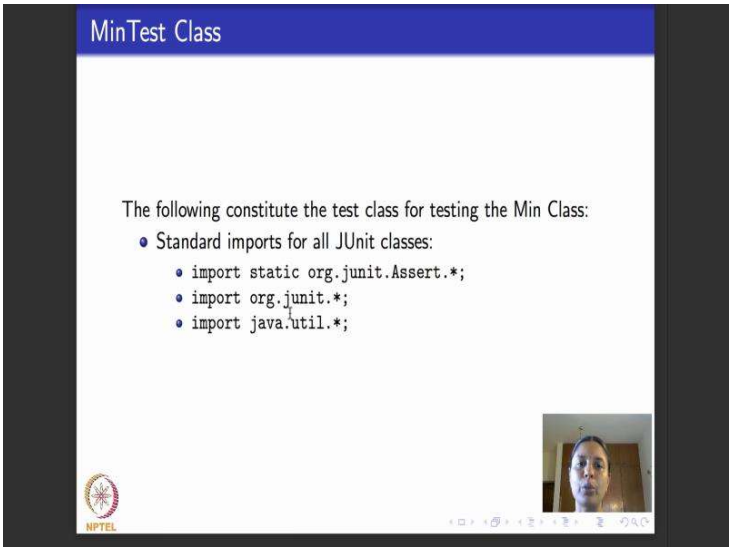


So, what is the main code look like? This is how the main code looks like, I just glanced through it because I do not want to read the code line by line, is just a standard written Java program that takes a list called T. And then if the list is empty then it throws an illegal argument exception otherwise it repeatedly compares it to the next element in the list and returns the result in this value called result; it returns the minimum element. So, this is a piece of Java code. So, just to recap what its main functionalities are, it takes a

list of compare table elements as its argument and then returns the minimum element in the list.

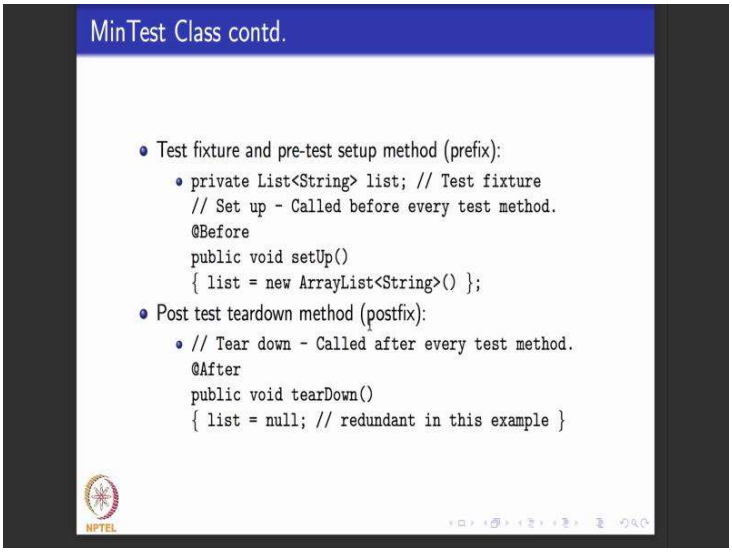
Suppose the list does not have comparable elements, the list is empty; the list has other kinds of problems this code does exception handling very well. So, now our job is to be able to design test cases for this minimum program and see how you can use the second step which is the test automation step that is the focus of this lecture to be able to test this program.

(Refer Slide Time: 18:02)



So, how do I do? The first step as I told you is because we want to be able to use the tool JUnit we have to import all the standard classes; we have to import JUnit assert, we have to import JUnit you have to import the util library. After this the next job is to be able to give prefix value and postfix values to the code.

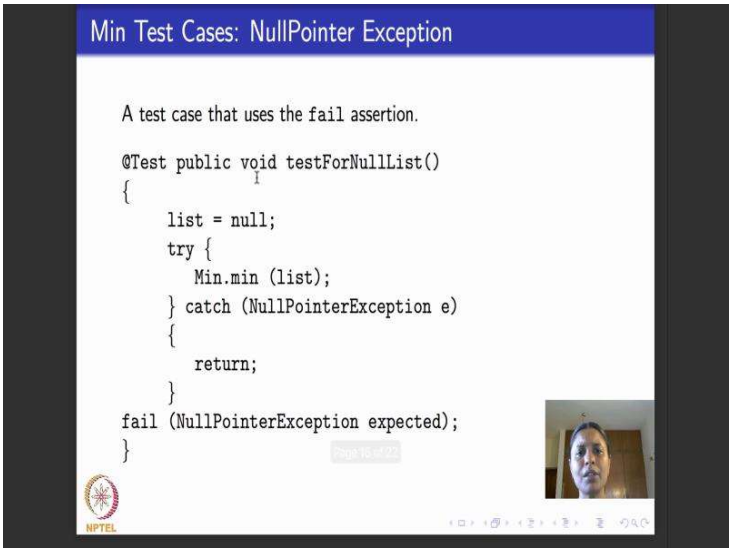
(Refer Slide Time: 18:25)



How is that done? Prefix values is given by this kind of, as I told you right, by this act before thing. So, what you do is that you give a test fixture and a pre test setup method. So, you have it like this--- you give it a list which is a test fixture and then you set it up which is called before the main test method. The main test method actually has the test cases for execution.

So, you say that this is the thing and I pass a new array for this. And after this in the main test case the main test method what we called containing the test cases, and post this I have what is called teardown method which gives the postfix values which basically in this case is not relevant because there is nothing much to do, but assuming that you had a fairly large piece of code postfix teardown method will actually do the rest of the execution to be able to see the output as being produced by the code. In this case because it is a small example we will directly see the output. So, there is not much postfix activity to be done here.

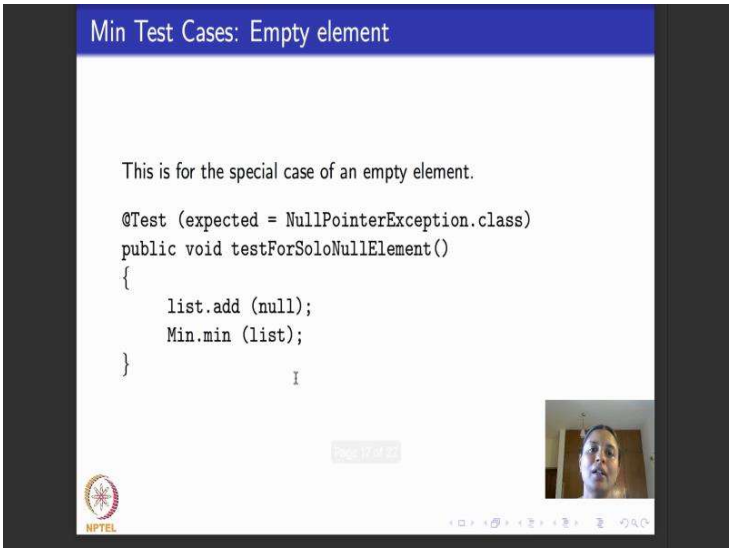
(Refer Slide Time: 19:29)



So, here is an example of a test case that uses the fail assertion. If you remember I had told you there are three kinds of assertions. So, if you give me a few seconds I will go back to that slide. Remember there were three kinds of assertions I used took different kinds of assert true which basically return a warning if this Boolean string that is passed to it. If this Boolean predicate that is passed it is returns false or it returns string if the Boolean predicate that is passed to it is return false. And then third kind of assertion will returned this string if it fails.

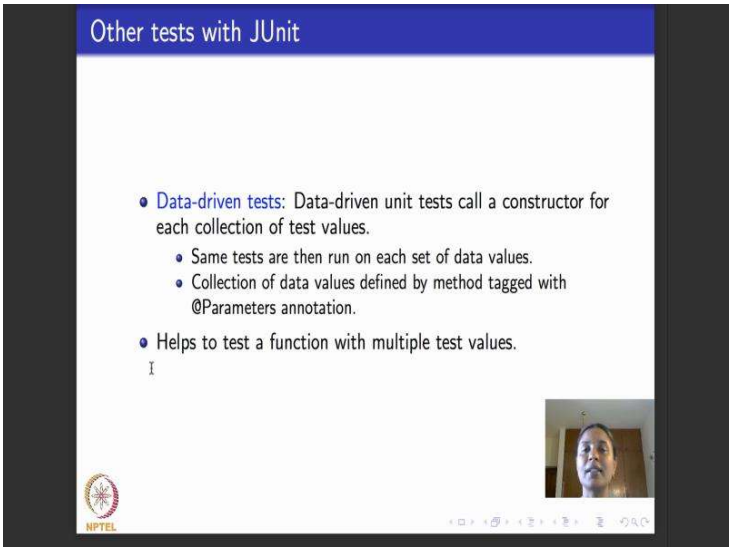
So, here for this min example we will use the fail assertion and here is a test case that uses the fail assertion. What is this test case method called it is called test for null list, it is a void method and it passes an empty list and then its tries to see if the code actually throws a null pointer exception. And if it does not throw a null point exception using this try and catch, it will use the fail assert to say that it had actually expected a null pointer exception which did not happen. So, there is an error in the code. In our code this will not come because this error is handled.

(Refer Slide Time: 20:49)



So, here is another example of a test case that tests our min code for the special case of an empty element. So, what it says is that I tried to add an empty element in the list and I tried to compute its minimum. And in this case because the code is well written this is also taken care of, it will return find and there is no error in this code even for this kind of test case.

(Refer Slide Time: 21:20)

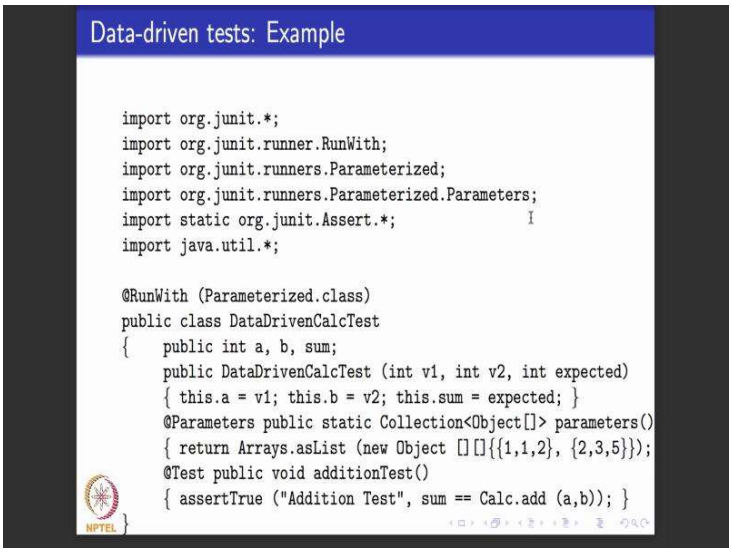


Now, what are the other things that we can do with JUnit. We saw two examples this is with testing for two exception cases. Of course, you can go on testing this minimum code

for several other exception cases, because the code handles several other an exception handling mechanism. But suppose I have to test the main functionality of the code. What is the main functionality of the code? The main functionality of the codes to be able to give a list of elements that are comparable and check if it actually returns the minimum value from the list; so for that I need to be able to give data, I need to be able to pass a list of compatible elements. How does that happen? How does one do it JUnit?

For giving data to test with JUnit we have a constructor for them. So, that constructor what it does is that you can passed several different data to it and the same tests are run for each set of data values and the collection of these data values are defined by a method tag with a @parameter. So, it basically helps to test a function with multiple text value. So, I can test a minimum function with several different lists and check for each of these lists does not return the minimum. I can check the add function that other toy example that we looked at with several different arguments a and b and check in each of these case does it actually return the sum of a and b.

(Refer Slide Time: 22:53)



So, I will go back to the add example and show you how to use this @parameters with the constructor to be able to provide data along with prefix, postfix and assert. So, here is the complete JUnit program for the same. So, import as I told you all these various classes, now what I do is I have to pass data. Now I am going back to the add examples. So, I have to always first two integers a and b to it and check if the sum of a and b is

actually returned by the addition code. So, I use this constructor and then I write this particular class that does the main testing.

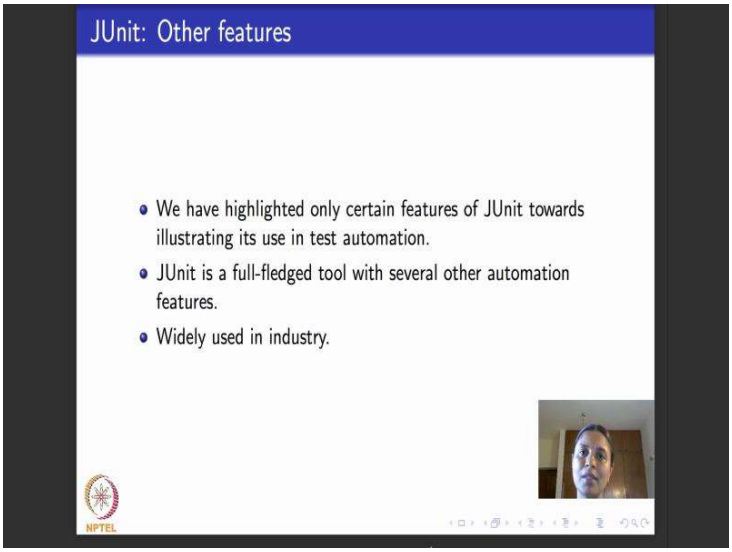
So, how does this work? If you see this dot a is equal to v 1 and this dot b is equal to v 2 these are the variables that are used to pass the actual parameters. And then the expected result is stored in this dot sum. And if you go here there is an array which the first value set of test case values that it passes are a is 1, b is 1, and expected result is 2. The second set of test cases that it passes are a is 2 b is 3 and the expected result is 5. In this example I have just given two so that I can explain it you, but you can pass an array of as many test cases as you want along with their expected outcome.

So, what it will do is; it will go back and execute this particular method. It will execute this particular method, and in any case for whichever test case if this assertion fails then it will output that this sum is incorrect. How it does is it will do it for each of these test case values and the expected output. And it will finally exit without giving any assertion violations if all of them pass. So, for out particular example the calculator addition was correct code so it will pass for all these examples.

So, similarly for the minimum element in a list also you can put it all together, right, check it for various null pointer, empty list and other kinds of exceptions, and after passing all these exceptions actually use this constructors class to be able to pass several different list of values and their minimum element and check whether the code actually returns the minimum element from this list; because that piece of test code in JUnit is fairly long to write it would not fit into even 2-3 slides, I have not given that as an example. What I can do is I will be putting it along with my notes feel free to look at it.

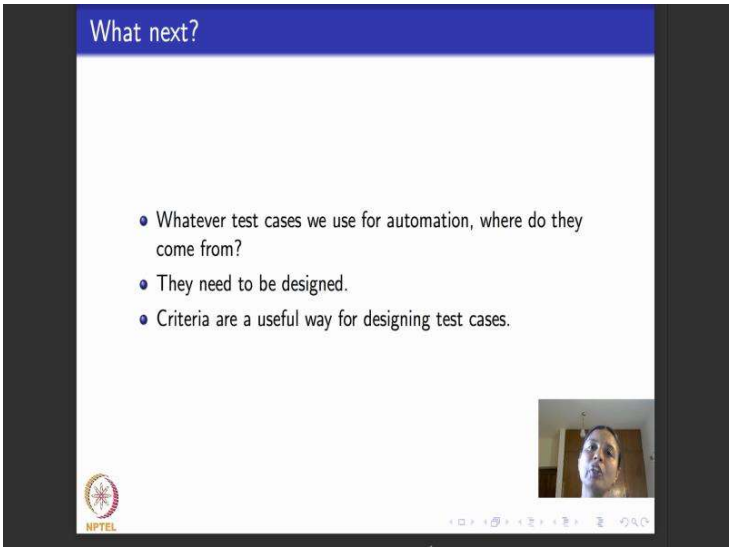
So, hopefully at the end of this exercise you would have understood how to give prefix values to a test case and how to actually give the test case values write asserts that will indicate whether the test case is passed or failed. And if needed, how to make postfix values in the codes such that the assert failure or assert pass actually reflexes output in the code.

(Refer Slide Time: 26:01)



To understand these we used JUnit. JUnit is a fairly extensive tool as I told you in the beginning; the purpose of this module was not to be able to teach you exhaustive features of the JUnit, but to be able to teach you how to use JUnit for test automation. So, feel free to go download JUnit and explore all its other features and try it out on your own Java programs to be able to see if you can test them or not.

(Refer Slide Time: 26:27)



So, where are we going on from this? So now, we saw how to automate and once you automate and execute JUnit also has an execution framework. As I told you it is fully

command driven, fully automated. So, there is nothing much to discuss about it, I will not be discussing about that in detail. After that you actually observed by using assertions the result of failures.

Now if you go back right to the first step; that is test case design it actually tells you what are the test cases that you pass on to a tool like JUnit for automating. How do you design test cases, how does one go about giving effective test cases without giving blind test cases and not hoping to find any errors. So, for this we go back to the problem of test case design. So, we will look at criteria based test case design, we model software using different mathematical model structure as I told you graphs, logical expressions, sets and so on. And teach you how to design test cases based on each of these. So, that will be the focus of my lectures beginning next week onwards.

Thank you.

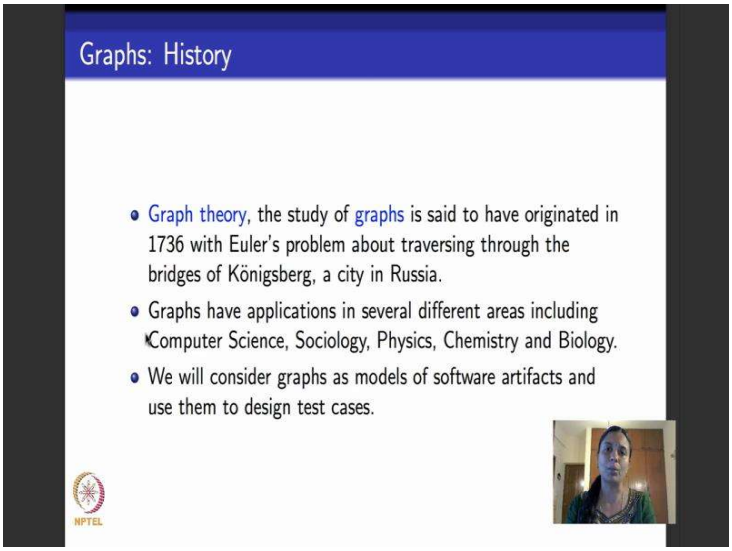
Software Testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

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.

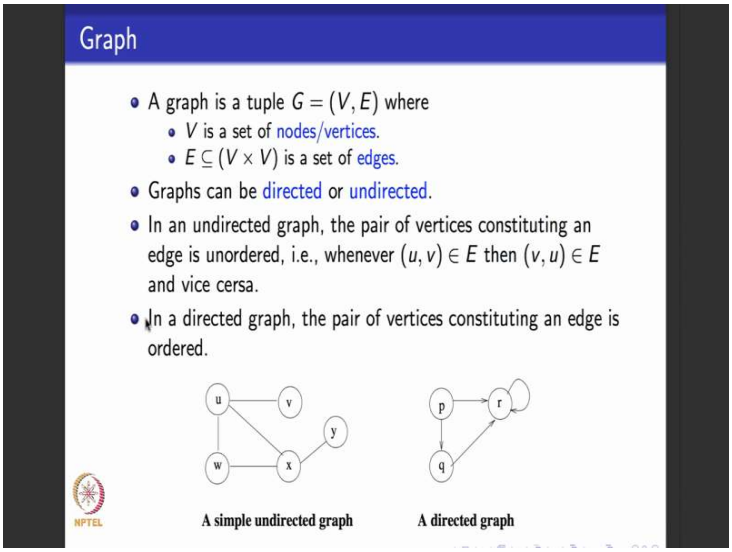
(Refer Slide Time: 01:18)



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)



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.

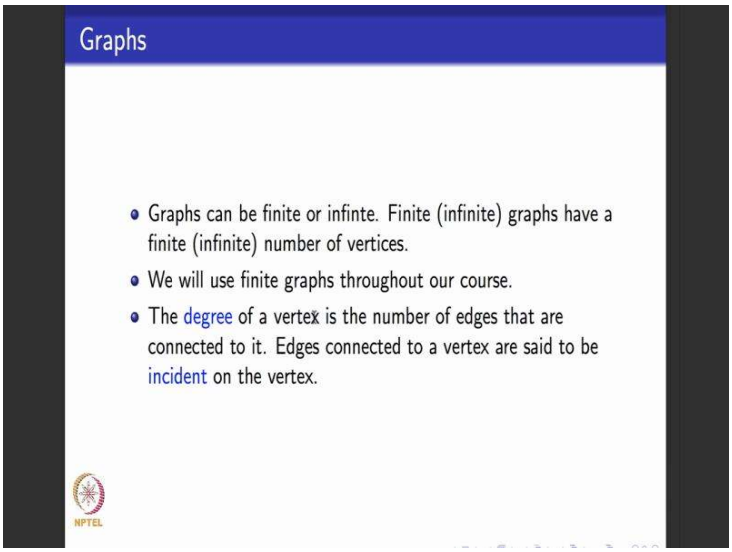
So, graphs have two parts to it there are nodes or vertices. Sometimes I will use these terms synonymously interchangeably. Some people also call it is points and different

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 modes 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 notes or two vertices. So, this pair of nodes of 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)



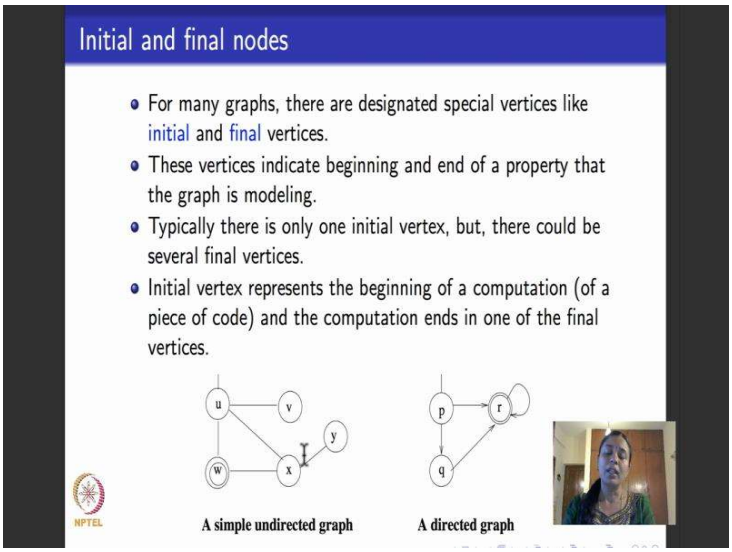
So, graphs can be finite or infinite. So, a finite graph typically has a finite number of vertices and infinite graph has an infinite number of vertices. Because our use of graphs in this course is to be able to use them as artifacts that modeled various pieces of software we will not really consider 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)



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.

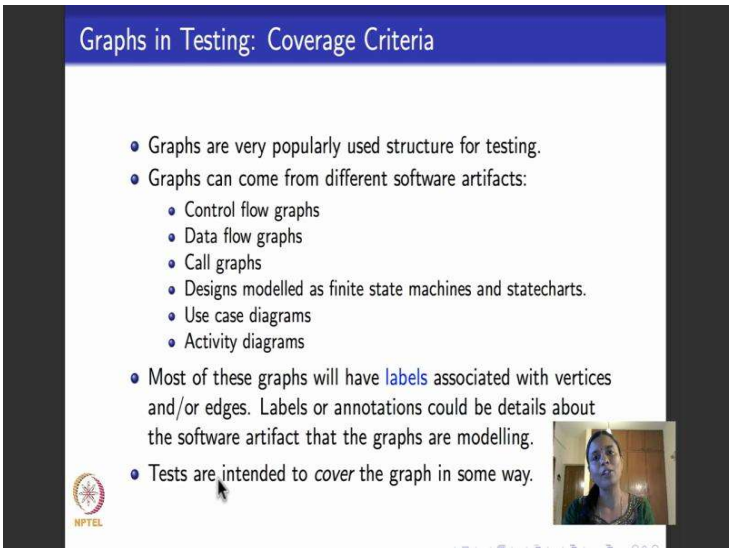
So, typically we believe that most of the software artifacts that we will consider like code mainly or design, is always supposed to be deterministic in the sense that its behaviour is definite and there is no non determinism in its behaviour. So, to be able to capture graph

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 u 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 has 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 a 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)

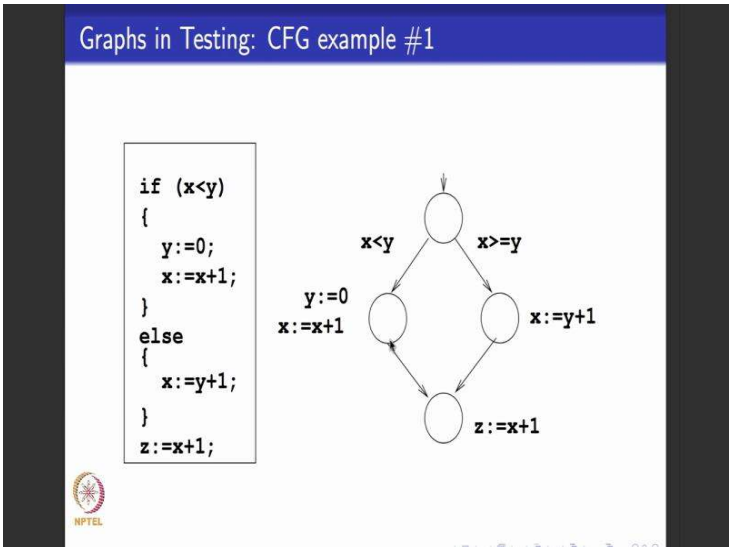


So, how are we going to use them as graphs? Why do we need look at graphs as far as software testing is concerned? Graphs, I believe, are next to logical predicates, may be a very popular structure used in testing right the several testing and static program analysis tools use graph models of software 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 finial 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)



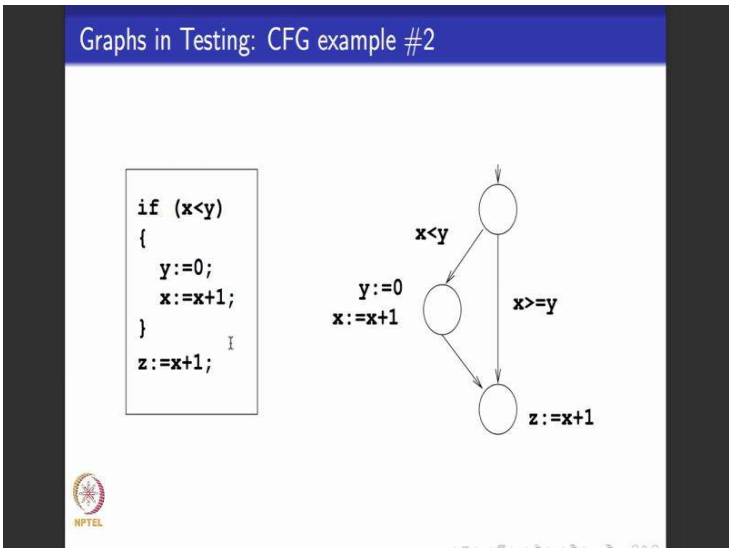
So, here is an example of how typical graph occurs as a model of a software artifact. Another thing that I would like to reiterate at this point in my course is that when we look at examples like this I will never show you a complete piece of code that is fully implemented; I will always show you a small 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 falsiity 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.

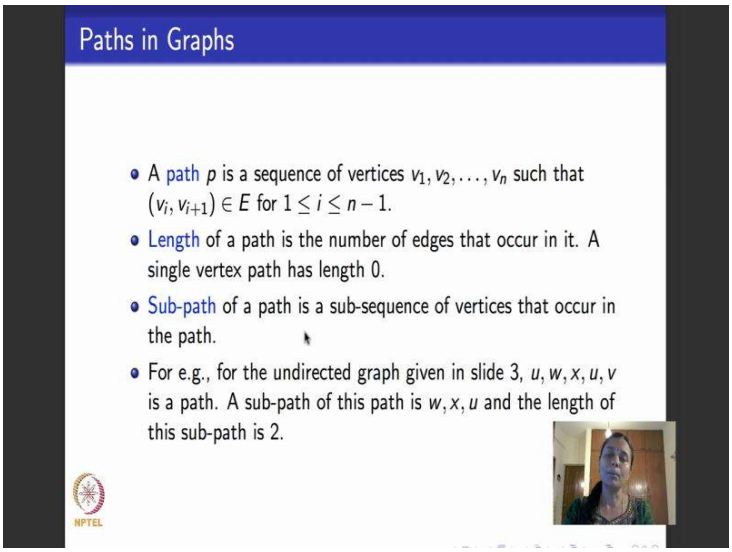
(Refer Slide Time: 16:33)



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 then 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.

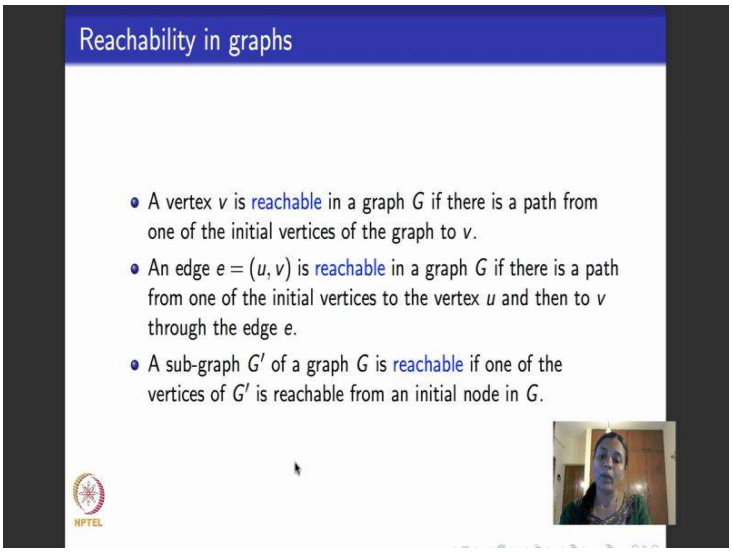
(Refer Slide Time: 18:03)



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 one v two and so, on up to 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)



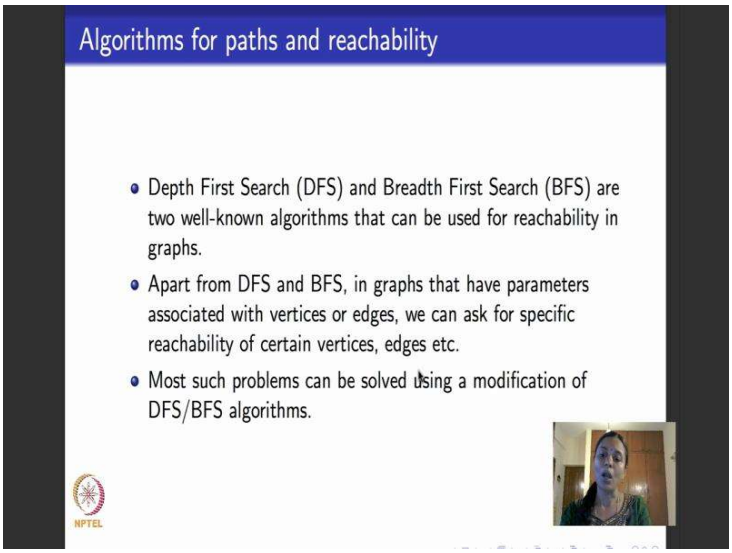
In this there is a sub path which is say w x u v. That is a sub path, u w x is another sub path. So, it is just sub sequence in the sequence of vertices that you encounter in the path. So, why are we looking at paths? We will looking at paths because we have going to be able to design test cases that we can use these paths to reach a particular statement or a particular mode 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 v to be 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.

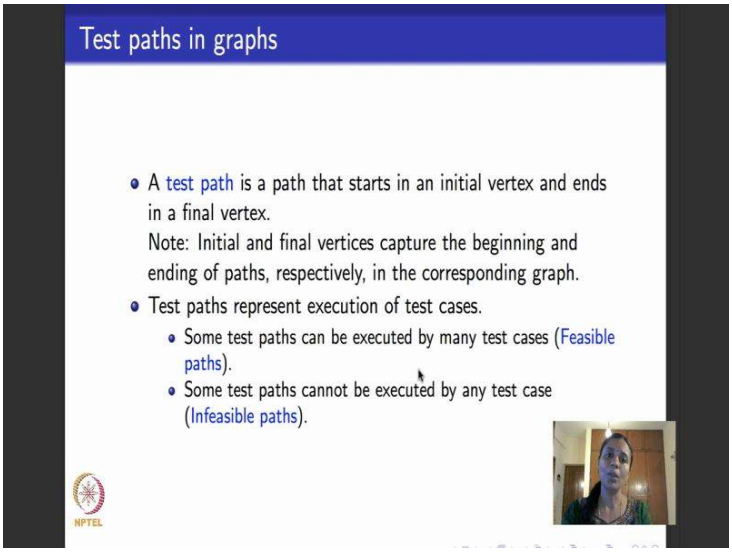
(Refer Slide Time: 22:41)



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.

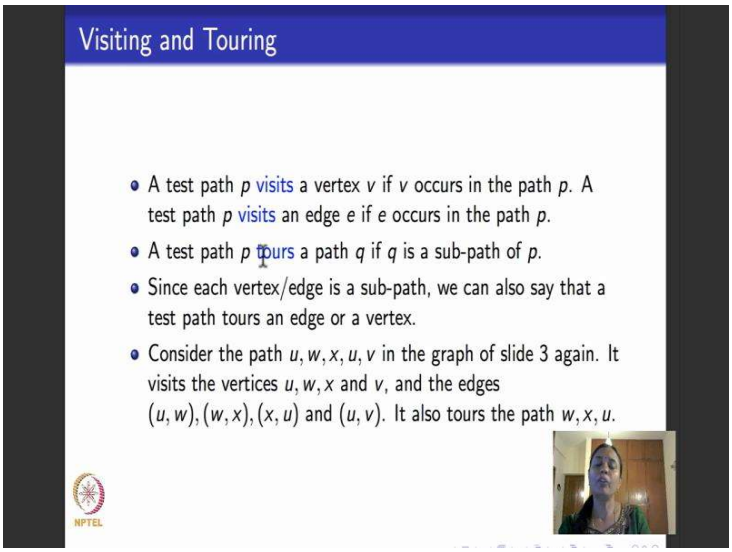
(Refer Slide Time: 23:29)



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 x 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 w x 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.

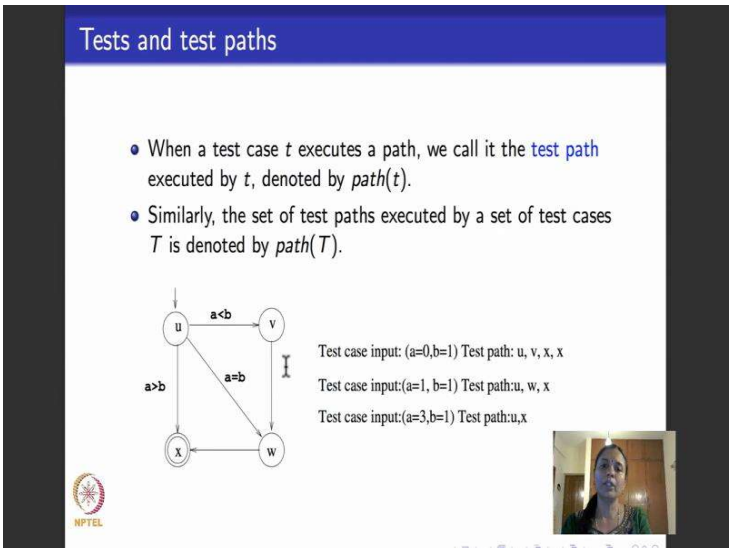
(Refer Slide Time: 25:04)



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 x w, right. So, this test path visits three vertices u x and w and it visits two edges the edge u x and the edge x w and it tours a sub path w x. You could consider another test path which looks like this it could be u w x u w right. So, this test path visits three vertices u 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 w 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)

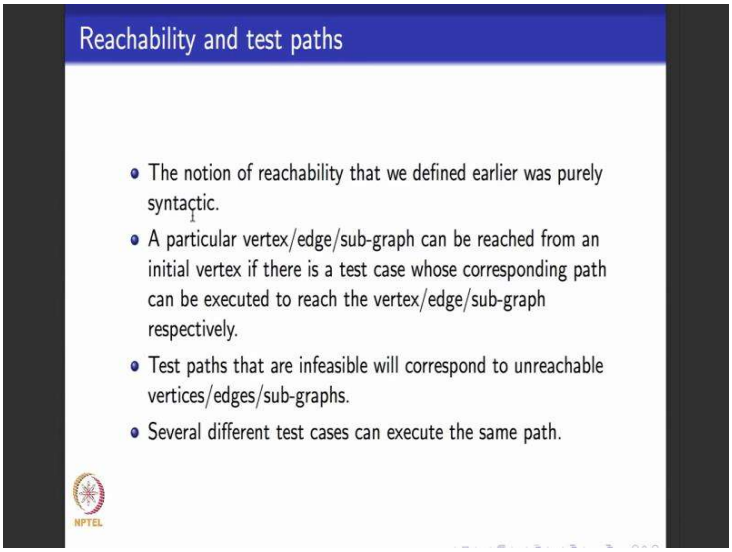


Let us take a small example look at this graph this graphs is got four vertices, u v w and x and here is an initial vertex u. It models control flow corresponding to a simple switch statement switch case statement, the switch case statement is at the node u. There are three switch cases, if a is less than b you go to b execute may be some statements, if a is equal to b than 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 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 be because my test case satisfies the predicate a less than v and I can freely go from v to w and then from w to x because these to 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 p b and than the test path that it takes is just the single edge u 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.

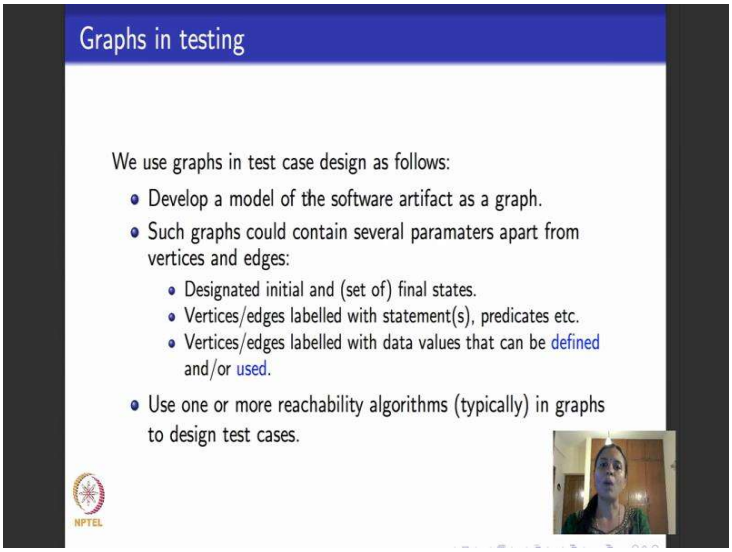
(Refer Slide Time: 29:07)



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 migh so happen that the control flow graph particular program has two disjoint components, in which case suppose I insist that and 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 in feasible 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 go into use graphs for test case design?

(Refer Slide Time: 30:44)

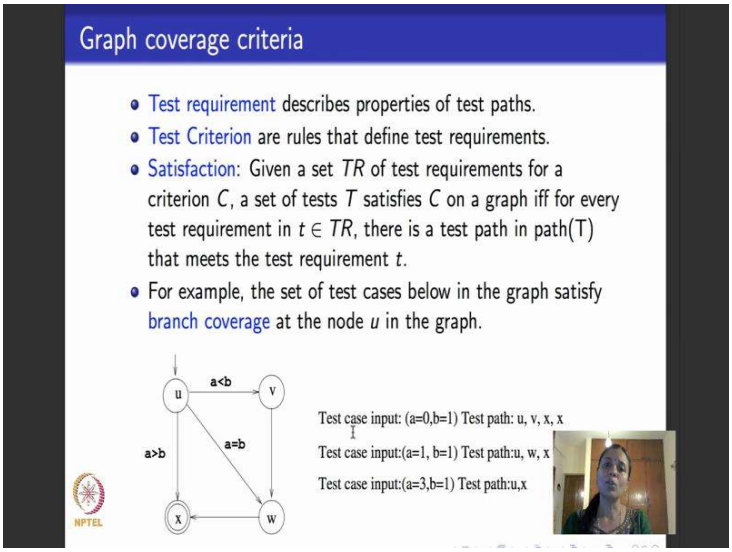


We will develop a model of a software artifact as a graph, I already showed you to examples of how to take a code snippet and write a control flow graph corresponding to that. It is to be noted as we saw in that example that these graphs apart from vertices and edges could contain several different annotations 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.

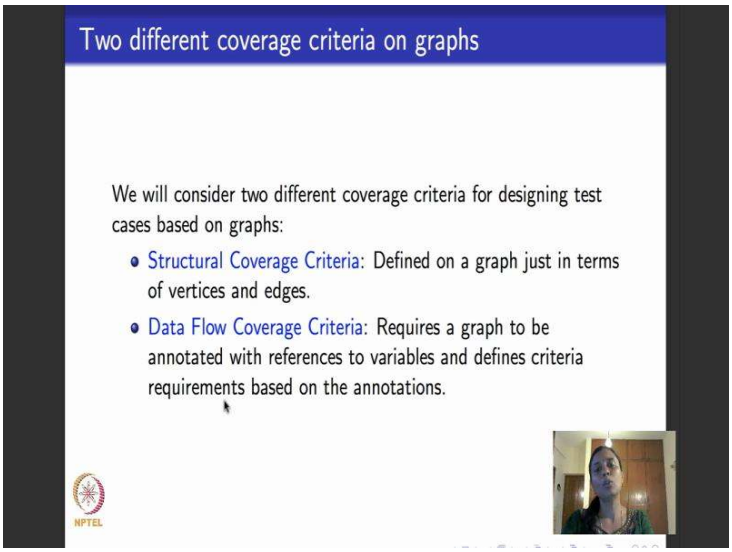
(Refer Slide Time: 31:57)



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.

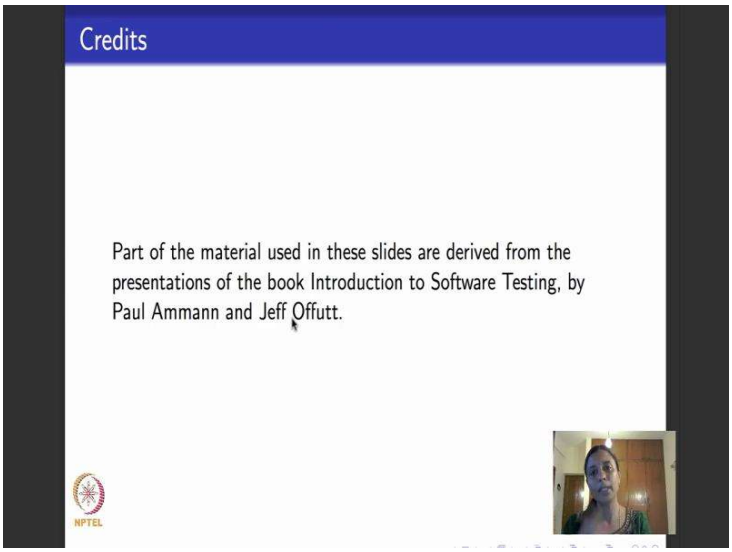
(Refer Slide Time: 33:10)



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.

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

Lecture - 06
Structural Graph Coverage Criteria

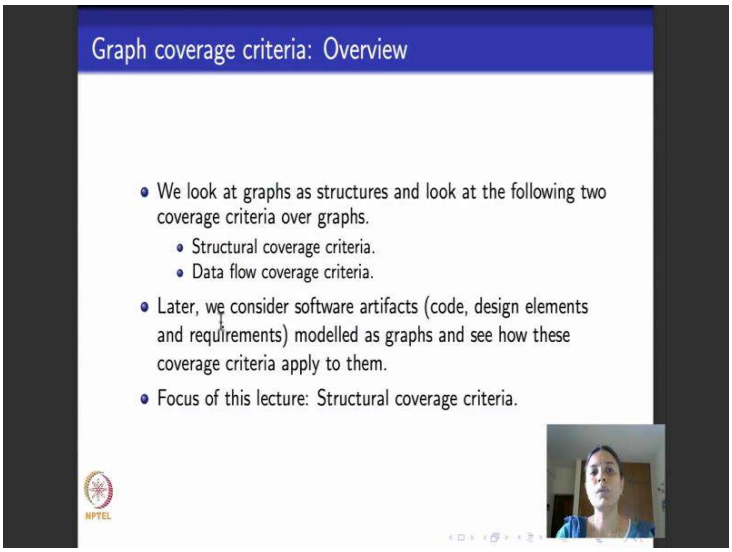
Hello everyone. Now we are in the second week. What we looking at this week is basically you look at graphs and to look at various algorithms that we can use for test case design using graphs. In the last module I gave you a brief background of graphs as it was necessary for this course. So, to recap what we saw in the last module: we saw what graphs where and in the basic concepts like degree of vertex, what is the notion of paths, and what are trips, what is visiting a note and few are the details. So, the plan is to be able to use graphs to design test cases. So, we take software artifacts like code, requirement, design, model them as graphs and see how to design test cases, how to define coverage criteria and then to design test cases using graphs.

So, in the first two modules that we will see as a part of graph based testing what I will do is we look at coverage criteria purely based on graphs. I will not really show you too many examples of how software artifacts are modeled as graphs. Instead we directly deal with graphs as data structure it is and define coverage criteria. In today's lecture we will define coverage criteria that are based on the structure of the graphs. That are based on nodes, vertices, edges, paths and so on.

The next lecture I will look at some algorithms and deal with coverage criteria that are again based on the structure of the graph. Moving on what we will do is when annotate the graph the vertices and the edges of the graph with statements and other entities. And we will define coverage criteria based on data that deals it with in the graphs. After we do these three kinds of coverage criteria, look at how they are related to each other, and how they can be use to design test cases based on graphs we will consider a later module where we will take various kinds of software artifacts; we will begin with code, then to go on to design, then move on to requirements.

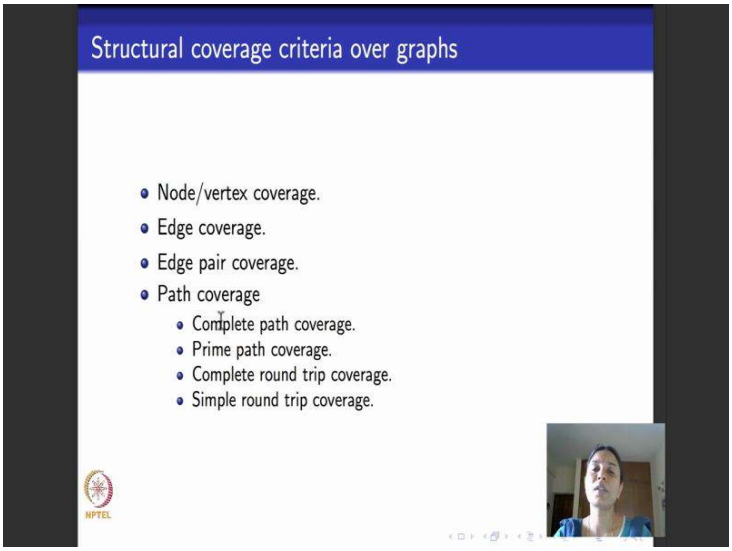
Model each of them as graphs and then see how the coverage criteria that we will be learning throughout these modules can be used to actually design test cases for covering the software artifacts.

(Refer Slide Time: 02:29)



So, what are we be doing at. So, first I will today's module I will deal with structural coverage criteria, in the next module also we look at structural coverage criteria but we will focus on algorithms, and then we will look at what is called data flow coverage criteria which consider graphs with data, variables and they values and then define coverage based on them. As I told you post this, we will take it has several software artifacts one at a time module them as a graphs and see how to use these coverage criteria to design test cases. So, we begin with coverage criteria in this course.

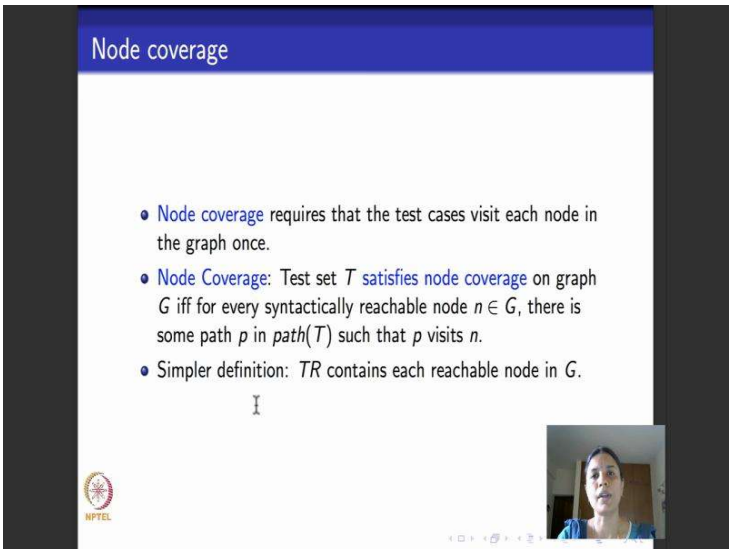
(Refer Slide Time: 03:03)



So, what are the various coverage criteria that we are going to see? Here is the listing. So, we will begin with what is call node coverage or vertex coverage. We will define a test requirement which says you have to cover/visit every node. And then we will define test coverage requirements for edge coverage where we say we visit every edge. Then we look at edge pair coverage which insists on visiting every consecutive pairs of edges which are paths of lengths two. Then we look at path coverage in the graph. In path coverage these are the various things that we look at. We look at complete path coverage which may not be feasible all the time.

And we will move on and look at very popular coverage criteria called prime path coverage. And then to make prime path coverage feasible we look at these round trip coverage criteria. So, we will see each of these one at a time.

(Refer Slide Time: 03:59)



So, we begin with node coverage, what is node coverage? It simply insists that you have a set of test cases that visit a every node in your graph. So, it could be the case that the graph corresponding to a particular software artifact can be connected or disconnected. If it is disconnected then the graph has several disjoint components. What do we mean by disjoint components? There are no edges between vertices of these components.

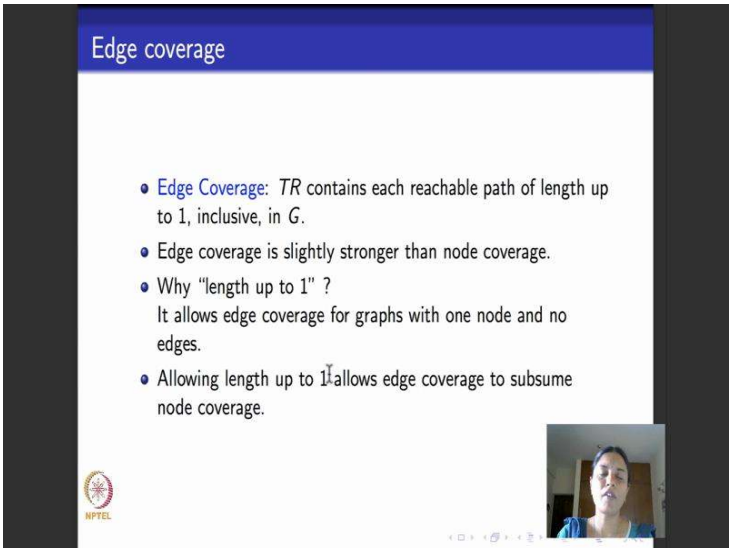
So, in which case if I see node coverage needs to visit every node, you might ask how do I go and visit other nodes then the graphs. So, what we insist is the node coverage visits every reachable node; every reachable node means every node that can be reached from

the designated initial node. So, what we mean is that per component per reachable component that is connected, you visit every node.

So, what is node coverage? Node coverage says, the test requirement for node coverage abbreviated as TR, says you basically right a set of test cases that will visit each node in the graph. Now how will you write a set of test cases that will visit each node in the graph? What will a test cases look like? How can you go about visiting nodes in a graphs. You have to start from a initial state and you have to walk along paths in the graphs. As you walk along paths you visit various different nodes.

So, set of test cases that will meet the test requirement on node coverage would be a set of test paths that begin at an initial state and visit every reachable node in the graph. I will show you an example after a couple of slides.

(Refer Slide Time: 05:40)



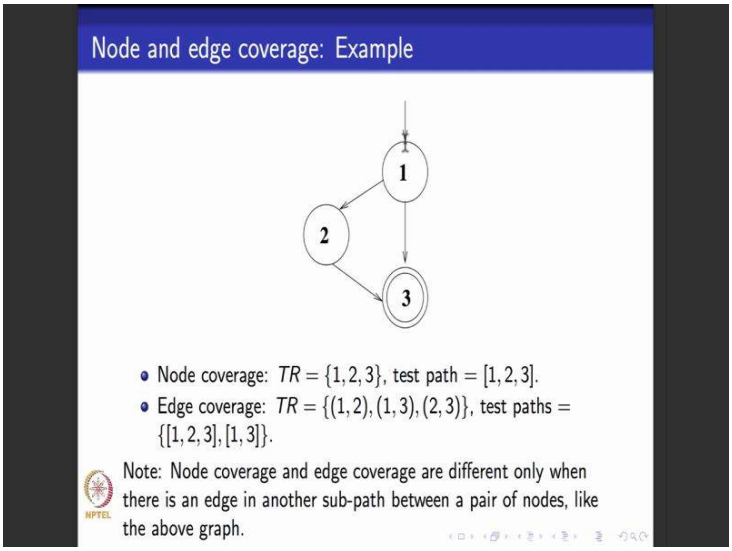
And then the next coverage criteria that we will be looking at; is what is called edge coverage. So, light node coverage edge coverage basically insists that your visit every edge in the graph. So, your test requirement says you visit each requirement path of length up to 1. So, you might ask when I say visit every edge why am I writing it as visit each reachable path of length up to 1? Now look at an edge in a graph what is an edge look like an edge can be thought of as a path of length 1, because which is connects two vertices. But instead of saying visit every path of length 1 which basically means visit every edge, we are slightly modifying the conditions to say visit every path of length up

to 1, which means you visit paths of length is 0 and paths of length 1. What do paths of length 0 correspond too? Paths of length 0 correspond to paths the just contains single vertex.

We want to be able to say that edge coverage subsumes node coverage in the sense that if I have a set of test paths that satisfy edge coverage then those set of test paths will also satisfy node coverage. Because we want to be able to say so, we modify edge coverage criteria definition to include that it visits every path of length up to 1.

So, it visits every single vertex and it visits every single edge. So, by definition edge coverage will subsume node coverage.

(Refer Slide Time: 07:17)



Here is an example to illustrate how edge and node coverage work. So, here is a small three vertex graph: vertex one is the initial vertex. As you can see,its mark to design coming arrow. Vertex three is the final vertex it is marked with its double circle and it is a small graph. If you remember in the previous example we had a new statement and I showed you how to model the control flow graph corresponding to that if statement remember, which did not have a else part just had the then part. This was the graph that correspond to the control flow graph of a that if statement.

I of course, remove the labels in the edges all that stuff, because we are looking at purely structural coverage criteria which defines coverage criteria based on the basic structure

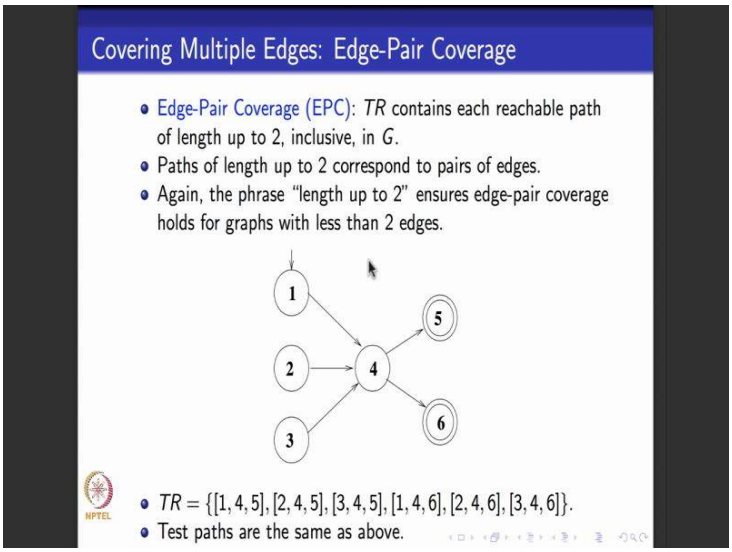
of the graph; does not really look at labels of edges of vertices. So, in this graph what would be node and edge coverage. Node coverage: the test requirement for node coverage says that there are three nodes so you please visit all three reachable nodes, all three of them are reachable here. So, how many paths can visit all three reachable nodes? So, I could take this path I start from 1 and node 2 and then to 3. In this path I have visited all the three nodes 1, 2 and 3. There is another possible path in the graph I could start from 1 and then I go to 3. Suppose I take this as the test path corresponding to the test requirement then I have only partially achieved node coverage, I have visited the nodes 1 and 3 I have not visited the nodes 2.

There is no harm in choosing that, suppose you choose that then you have to add this path also 1, 2 and 3. But instead, I could directly choose this path 1, 2 and 3 as my test path. And in that case I would directly met the test requirement of node coverage which is one path.

Now, let us look at edge coverage how many edges are there in the graph? There are three edges in the graph: one edge from 1 to 2, one edge from 1 to 3 and another edge from 2 to 3. What is edge coverage mean? The test requirement or TR, for edge coverage says you visit all the three edges. The edge 1, 2 the edge 1, 3 and the edge 2, 3. Here if you see suppose I take this path 1, 2 and 3. How many edges do I visit in this path? I visit 1, 2 and 2, 3. So, I visit two edges so that is this path here. But I have to be able to visit the edge 1, 3 so I have to take this path.

So, to meet the test requirement for edge coverage and this graph I need two test paths; the test path 1, 2, 3 and then the test path 1, 3. This is another small observation. Suppose the graph is a sort of a straight line right, there are no branching and it just to corresponds to some core the corresponds to sequence statements. It just like one linear order or a straight line or a chain. In that case node and edge coverage are same, because when I visit every edge there is only one path in the graph I visited the path and in the process I visited every node also. So, node and edge coverage become different in terms of the test paths that satisfy it only if there is a branching in the graph.

(Refer Slide Time: 10:24)



So, we move on. What is the next coverage criteria that were there in my list? Go back to that slide. So, we did node coverage which basically says visit every node. So, you write test path that visit every node edge coverage says visit every edge once, so you write test paths that visit every edge. Now will move on to edge pair coverage and then look at path coverage.

Before we move on to edge pair coverage I would like you to spend a minute thinking about what would node coverage, how would node coverage and edge coverage be useful. Assuming that such a graph is a control flow graph corresponding to a program what can you think of node coverage and edge coverage put together they basically mean you execute every statement in the program.

Because they insist that you visit every node and then they insist that you visit every edge. If you write a set off test path that satisfy this then you are writing a test path that basically execute every statement in the program. So, you are looking for a set of test cases that will exercise for test every statement in the program. Node coverage and edge coverage might be quite difficult to achieve, because for large pieces of program where the control flow graph is fairly large, writing a set of test cases that will achieve execution of every statement, every node or every edge, can be a large set of test cases and sometimes infeasible also.

So, we look at other state of test cases. The next set of structural coverage criteria that I would like to talked about is what is called edge pair coverage. So, as a name says, you your test requirement here is the actually consider pairs of edges; not pairs of edges that far away from each other pairs of edges that are consecutive to each other. In other words you consider paths of length up to 2. Paths of length up to 2 includes paths of lengths 0 which include nodes, paths of length 1 which are edges and paths of length 2 which are actually sets of edges that occur one after the other. So, if you take this example graph, how many paths of lengths two there? All these six paths a paths of lengths 2. So, I start from 1 which is an initial vertex 5 and 6 of final vertices. So, I go from 1 to 4, 4 to 5 that is a path of length 2 pair of edges then I go from 1 to 4, 4 to 6 another pair of edges that are consecutive; 2, 4, 5, another pair of edges 2, 4, 6; 3, 4, 5; 3, 4, 6. So, my test requirement says- you visit all these paths of length to exactly once. So, here they have written length up to 2, but I have listed here paths of lengths 2. I implicitly assume that it includes paths of length 1 paths of length 0.

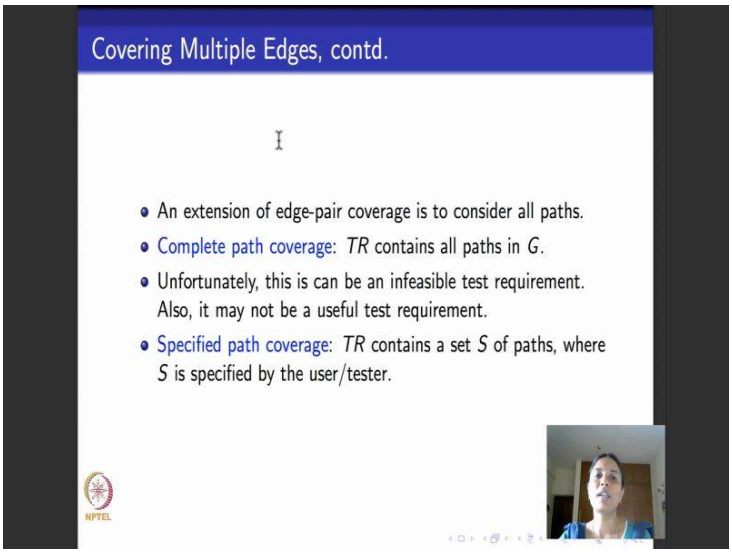
So, how many text requirements can I write? If you look at this structure of the graph it is so happens that I need one path for meeting each of these Trs, I cannot do better. So, test paths or basically all these six things, I cannot write anything shorter. Now why do I need paths of lengths up to 2, the same reason we said when we do edge coverage even though edge is a path of length 1 we changed it as path of length up to 1, because we wanted edge coverage to subsume node coverage. So, for the same reason we want edge pair coverage to subsume node coverage and edge coverage. So we insist that the TR contains all paths of length up to 2.

Suppose you do not you are not very particular about this requirement there is no harm in saying that edge pair coverage means, you visit all paths of length exactly 2. So now, before we move on what you think edge pair coverage would be useful for? One good use of edge pair coverage it is to cover all branches in the program. So, if you think of this as the control flow graph representing some piece of code, lets not worry about what that piece of code is, but let us assume that this is the CFG for some code.

What could node four be? Node four could be corresponding to an if statement right, which says it has two branches: if then is true may be you go to 5 if else true may be you go to 6 right due to something here you do something here. So, when I do edge pair coverage what I cover is from 1, 2 and 3, what are the various ways and which I can visit

4, and from 4, what are the various ways and which I can cover the two branches that go out of 4? So, edge pair coverage is useful. Suppose I think about it what did I do I did paths length 0 which was vertices no coverage, then I did paths of length 1 which were edges edge coverage. Now I say paths of length up to 2, I did edge pair coverage you can move on right you can say paths of the length 3, paths of length 4, paths of length 5. Then if we go on like this, what we have is what is called complete path coverage.

(Refer Slide Time: 15:17)



You have a test requirement TR which says that you cover all paths in the graph. Now if you think about it what would be the use of this? Will it be useful? What would all paths mean? If I say cover all paths on the graph let us assume a case where the graph has a loop. In the two examples that we saw in the previous two slides, the graph did not have a loop, but let us assume the graph has a loop. How many paths do you think will be there in the graph? They will be infinite number of paths in the graph, because I visit the loop once I get one path, and I visit the loop once again I will get another path, I visit a loop once again I get another path. So, I can go round the loop again and again and again and get more and more paths, longer and longer paths.

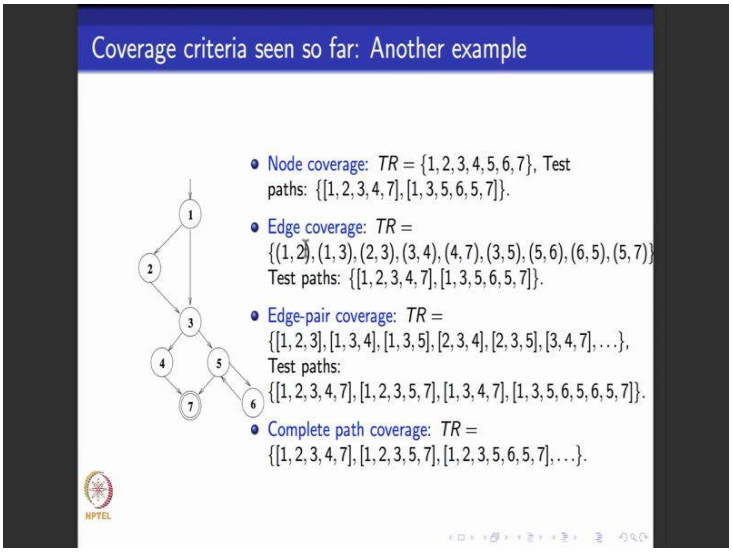
So, there will be infinite number of paths. And given these infinite numbers of paths how do I achieve complete path coverage? I will have to be able to go on writing test paths. So, for this reason complete path coverage is believe to be an infeasible test requirement. Infeasible in the sense that I will never be able to stop testing- if I want to achieve

complete path coverage intuitively. Also if you think about it not only is it infeasible, what is a use of it? It may not be very useful for us. What is the point you going on repeatedly executing a loop once, twice, thrice, four times and so on? It is not interesting, infeasible and not very useful.

So, what we say is that we will list complete path coverage as a test requirement, but it is not practically usable. What is practically usable is what is specify path coverage. What does specified path coverage say? It says that the use as a test engineer will give you the set of paths to cover. The TR will be a set of paths that is specified by a test engineer or a user which says you cover these paths. So, paths could be some special things, we do not know what they are, but the test is basically gives a set of paths and says you this is your test requirement. Please write a set of test cases that will cover this path.

So, we see specify path coverage, your test requirements contained a set of paths where that is set is specified by the user.

(Refer Slide Time: 17:44)



So, will go and now move on and see what is one specific, useful specified path coverage. Before that I will give you another example just to recap all the coverage criteria that we have sees so far. So, what are the folk structural coverage criteria that we saw so far? We saw node coverage, edge coverage, edge pair coverage and complete path coverage. So, let us take a small graph. Here is a graph on the left hand side, it has a about 7 nodes, the node 1 is the initial node, and there is one final node which is node 7.

If you see this could constitute a reasonably interesting control flow graph. So, if you try to map it to assume that it is a control flow graph corresponding to some code there is branching at statement one. So, maybe there was an if statement here. There is another branching at statement 3. There is a another branching statement 5. One branching ends in the final state 7, one branching goes into a loop between 5 and 6. So, maybe there was a while statement here, this means skipping the loop and this means executing the loop.

So, what will node coverage on such a graph be? Node coverage on such a graph--- the test requirements says there are 7 nodes, please visit all 7 nodes. How many test path will visit all the 7 nodes? There could be one test path like this I do 1, 2, 3 4 7. This is the path, this test path from the initial node 1 to the final node 7.

So, in this test path I have visited how many nodes? I have visited 5 of my 7 nodes I visited nodes 1, 2, 3 4 and 7. What are the node 7 left behind; 5 and 6. So, I have to write another test path which includes that. So, again because it is a test path I have to begin from an initial state and end in a final state. So, I begin at node 1, then I go to 3, I do not want to go to 4 because I have already considered that in my test path; from 3 I go to 5; and then I do not want to go to 7 because if I do that then I will miss visiting node 6, so form 5 I visit node 6. And remember it is a test path so I have to be able to end in a final state. So, I come back to 5 and then I do 7, so that is the test path.

So, just to summarize node coverage test requirements says you visit all the 7 nodes which is the set. And what are the test paths that will visit all the 7 nodes there are two test paths one which goes like this 1, 2, 3, 4, 7 which leads of nodes 5 and 6. So, I put another test path which says 1, 3, 5, 6, 5, 7. So, these two test paths put together satisfy this test requirement of node coverage. Similarly on this graph how do I do edge coverage? How many edges are there? So many edges are there right; (1, 2); (1, 3); (2, 3); (3, 4); (4, 7) and so on.

So, test requirement says you visit all this edges which is the set of all edges in the graph. What are the two test paths a very similar to node coverage, because I told you right edge coverage also meant to subsume node coverage. So, if I do this test path 1, 2, 3 4 7 I visited all the edges that occur on this set. Now, I write another test path that lead covers a missing edges which is this (1, 3); (3, 5); (5, 6); (6, 5) and so on. So, between these I have done edge coverage the next test requirement is edge pair coverage. Edge pair

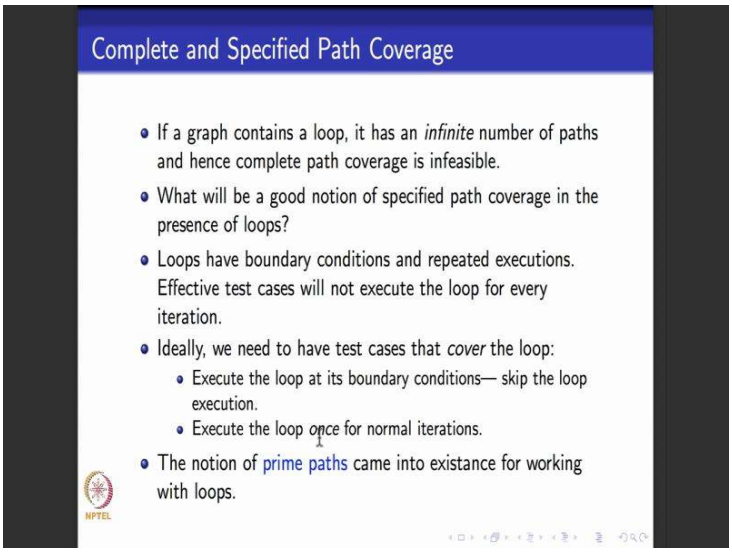
coverage lists all paths of length 2 which is 1, 2, 3; 1, 3, 4; 1, 3, 5 and so on that put this dot dot dot because I ran out of space to list all the edge pairs, but you can finish this its only the finite sets slightly larger than edge set that is my TR.

So, please write path then visit all this path of lengths 2 that would means it will visit all these consecutive pairs of edges. So, test paths for these would be if you see 1, 2, 3, 4, 7 it covers edge pairs that occurs a longest path and then I do 1, 2, 3, 4. I forgot this pair so 2 to a 3, 3, 5, so I do a 3, 5 7 which covers all test paths along these lines and then I do 1, 3, 4, 7 which includes this edge pair. Now I have to include 1, 3, 5, 6 5, 7 which completes all the edge pairs that I had.

So, with this four test paths I can achieve the test requirement for edge pair coverage. Now complete path coverage for this graph- please remember this graph has a loop here from 5 to 6. So, complete path coverage for this graph test requirement will not stop, it will go on listing paths, because I have this path it skips the loop, I have this path which also skips the loop for then I have take this path which enters the loop. Once I enter the loop I have a path that looks like this which visits the loop once. Then I can do the same thing again 1, 3, 5, 6, 5, 6, 5, 6, 7 and then I can do 1, 3, 5, 6, 5, 6, 5, 6, 5, 6, 7 then I can go on writing.

So, test requirement for complete path coverage for this graph with a loop will be an infinite set. So obviously, I am not going to be able to make it feasible or write a test path, set of test paths that will execute complete path coverage. Its only for completeness we really do not consider it is a useful coverage criteria by any means.

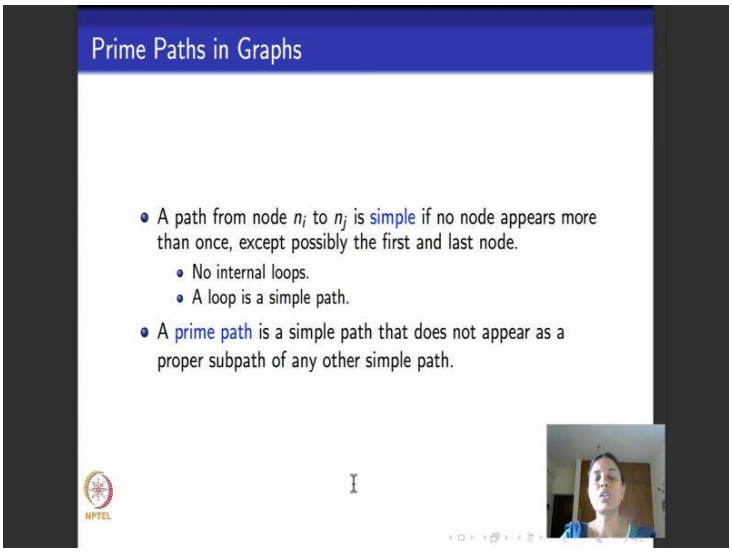
(Refer Slide Time: 23:02)



Now we will see how to overcome this problem about complete path coverage. I have this graph with loops; the only way I have touched upon this loop is to be able to do complete path coverage. But then that is not very useful because it gives rights to an infinite number of paths. So, is there a mid way solution? So, the question that we want to ask is what will be a good notion of specified path coverage in the presence of loops. Now let us look at loops. Assuming that this kind of control flow comes from a loop, when we want to test a loop what do we ideally want to test? We say that loops have boundary conditions and then they have normal operations. So, when I test a loop maybe I want to test around its boundary condition. And let us say the loop is meant to execute 100 times, it is a far loop for I is equal to 1 to 100. So, I do not want to really execute 100 normal executions of the loop.

So, I want to be able to test the loop for its normal execution maybe once. And then I want to be able to test the loop around its boundary conditions. What happens when the loop begins, what happens when the loop ends? So, when we say we need a set off test cases that cover a loop we are looking for the following kind of test cases. When we execute the loop at its boundary conditions which will involves skipping the loop also and then we execute the loop maybe once or few number of times for its normal operations.

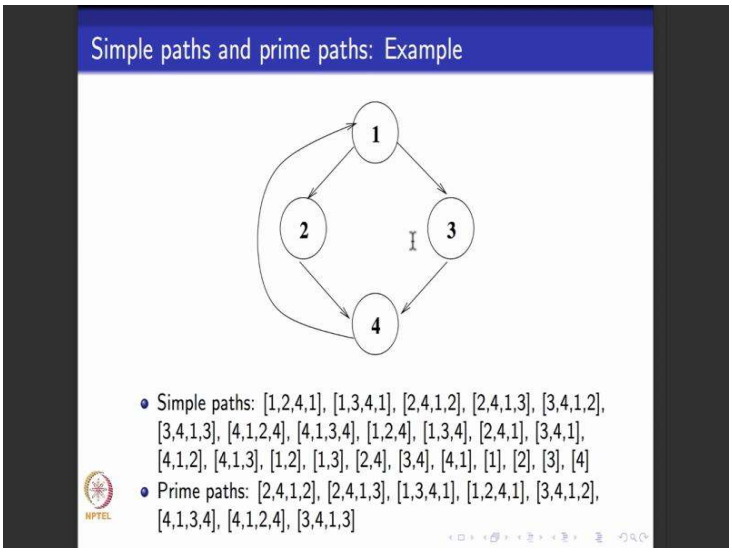
(Refer Slide Time: 24:36)



So, the notion of prime paths and prime path coverage help you to achieve this kind of execution for loops. So, what are prime paths? Before we look at prime paths I need to tell you what a simple path is. So, what is the simple path? A path from a particular node n i to another node n j is said to be simple, if no node appears more than once except possibly the first and the last node. So, simple path could be cycles, they could begin and end in the same node n i and n j could be the same vertex, but in between the path nothing appears more than once; which means there are no internal loops in the graph and every loop is believed to be a simple path.

Now, moving on ,what is a prime path? A prime path is a simple path that does not appear as a sub path of any other simple path. Just to repeat; what is a prime path? It is a simple path; and what is a second condition, second condition says that it is a simple path that does not come as a sub path of any other path.

(Refer Slide Time: 25:36)



So, I will show you an example to make this definition clear. Let us look at this graph, it has four nodes: 1, 2, 3 4. How many simple paths are there? If you go and see this listing what I have done is I have listed all the simple paths, I have listed them in a particular order in fact maybe will begin that this end. This 1, 2, 3 4 if you see right at the end of the listed simple paths they are simple paths of length 1.

Later we will move ahead and listed simple paths of lengths 2 which are basically; sorry 1, 2, 3 4 as simple paths of length 0 they just contains single vertices. Then I have gone ahead and listed simple paths of length 1 which as just edges. Then here I have listed simple paths of length 2 which are edge pairs like way, and the then I have listed simple paths of length 3.

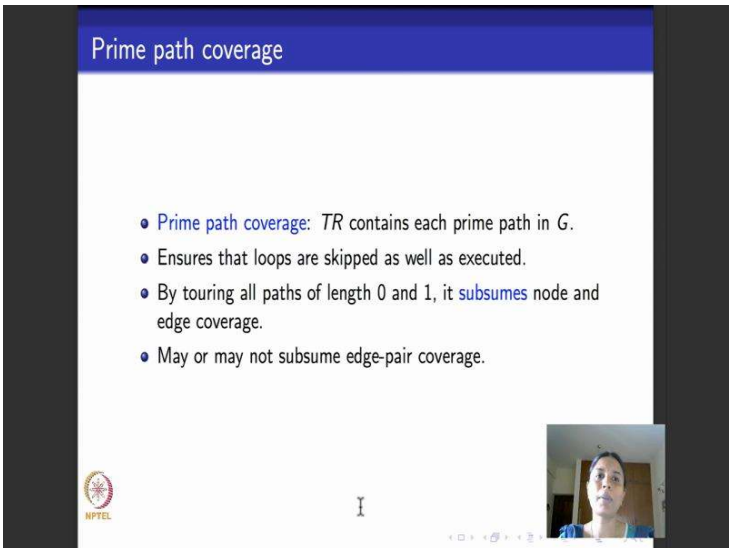
The other thing to note that this is the graph with four vertices. So, simple path means no vertex should be repeated. So, if I have a condition then what is the maximal lengths simple path that I can have with four vertices? The maximal length simple path that I can have with four vertices should be of length 3, because the moment I increase one more edge I am repeating one vertex.

So, the maximum length that I can get without repeating a vertex with four vertices is of length 3. Now if you see, if you look at all these simple paths, there so many of them, I say out of all these simple paths only these a prime paths. Why do I say these are prime paths? Go back to that definition of prime- path prime path should be simple, prime path

should not be a sub path of any other path. So, if you take a path that looks like this let us take 4, 1, 2 this path; 4, 1, 2. If I take a path like that this 4, 1, 2 path comes as a sub path of this path here 2, 4, 1, 2 it also comes as a sub path of this path here 3, 4, 1, 2. So, 4, 1, 2 does not qualified to be a prime path.

Similarly, if I take something like 2, 4; 2, 4 occurs a sub path here 1, 2, 4, it occurs a sub path here 2, 4, 1 it occurs as a sub path here, here, here, several places. So, path like 2, 4 does not qualified to be a prime path. If I go on looking like this, I basically eliminate all paths of length 0, 1, 2 and 3. And only paths of length four in this case happened to be prime paths, because none of these paths of length four occur as sub paths of each other. So, this graph has several simple paths, but it has only about eight prime paths.

(Refer Slide Time: 28:09)

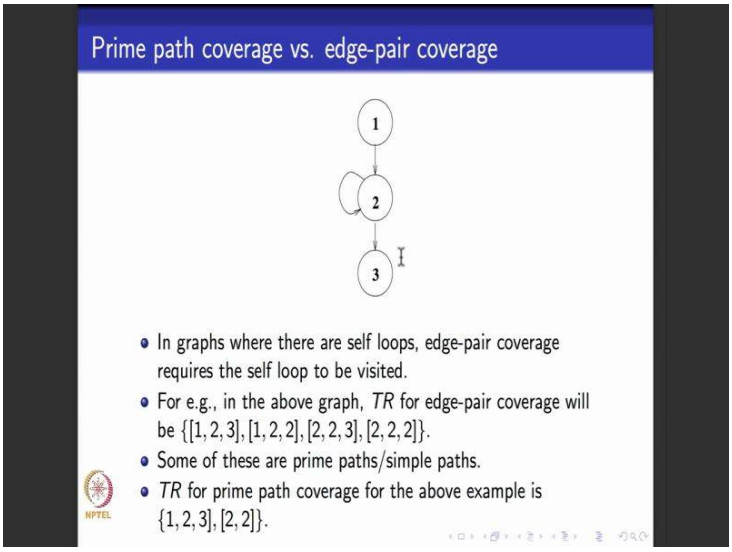


Now, what is prime path coverage? Prime path coverage very simple; the test requirement for prime path coverage says- please cover all the prime paths in G. Now, to be able to meet this test requirement you need to write a set of test paths that first identify the prime paths in C and then cover them. So, in the next lecture I will tell you how to do this, what are the algorithms that will do this, but for now we will go ahead and see a little bit more about prime paths coverage without worrying about how to do it.

So, the first thing that I want to tell you is that prime path coverage actually meets this loop coverage criteria that I told you about; this one test a loop at its boundary and test a loop for its normal operations. So, we will understand how that happens. Prime path

coverage ensures that loops are skipped and executed. And towards all paths of length 0 and 1 we saw that for that example at least. So, by default it subsumes node and edge coverage, because if you see here it includes node coverage and it includes edge coverage.

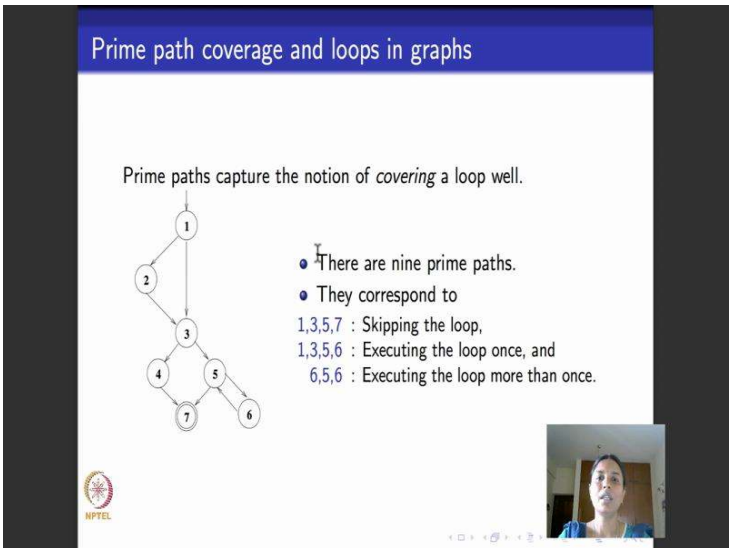
(Refer Slide Time: 29:23)



This node says that it may or may not subsume edge pair coverage. Why is that so? I will show you an example: considered a graph that looks like this, it has three vertices and then it has got a self loop here. So, edge pair coverage for this graph requires that the self loop needs to be visited. So, edge pair coverage for this graph we will say you visit this edge 1, 2, 2; you visit this path of lengths 2 which is 1 to 2 and 2 to itself. But if you look at this path 1 to 2; if you see the vertex 2 repeats here. And that violates the condition of it being a prime path. Remember prime path a simple paths no intermediate vertices as supposed to repeat then non supposed to have loops.

So, except for this problem prime path coverage does cover edge pair coverage, but not for graphs like this; if a graph looks like this right prime path coverage does cover edge pair coverage. But if a graph has a self loop then prime path coverage does not cover edge pair coverage.

(Refer Slide Time: 30:19)

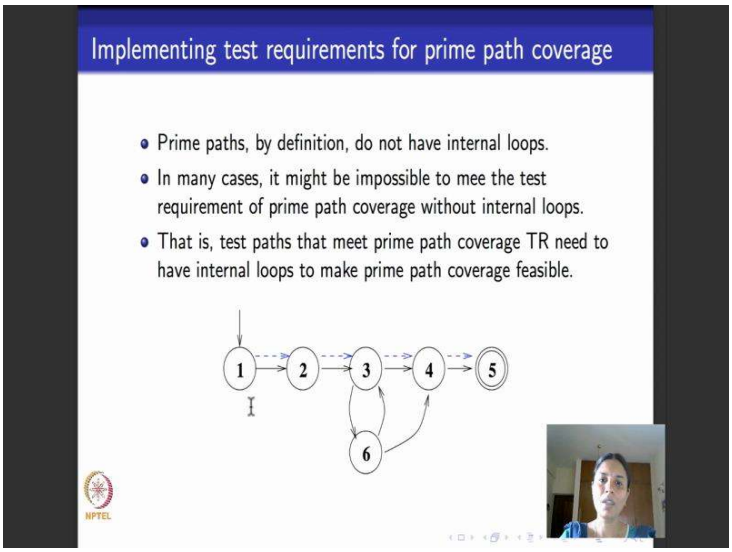


So, now we will move on and understand how prime paths cover the notation of a loop. So, we will go back to this example graph that we looked at a few slides earlier. In this example there is this loop here between nodes 5 and 6. And I am not listing the prime paths for this graph, but you can take it on faith that this graph has nine prime paths. In the next module I will tell you how to list all those nine prime paths.

So, then if we see this is one prime path; 1, 3, 5, 7 because it is a simple path and it does not occur as a sub path of any other path. So, this prime path for this particular example corresponds to skipping this loop, because a completely avoids this loop and if you see here is another prime path 1, 3, 5, 6. This prime path gets into the loop, it corresponds to executing the loop once.

And then this is another prime path for this example 6, 5, 6. Why is this prime path? Remember in prime paths which have simple path the beginning and ending nodes are allowed to repeat, only the intermediate nodes are not allowed to repeat. So, 6, 5, 6 is a prime path. And, what is the main job then 6, 5, 6 is doing for this graph? It is executing the loop more than once. So, if we see these three prime paths 1, 3, 5, 7 skips the loop 1, 3, 5, 6 enters the loop and then 6, 5, 6 helps you to execute the loop for its normal operations. So, it is intuitively in this sense that prime paths exactly capture loop coverage the way we want them to do in test case design.

(Refer Slide Time: 32:04)

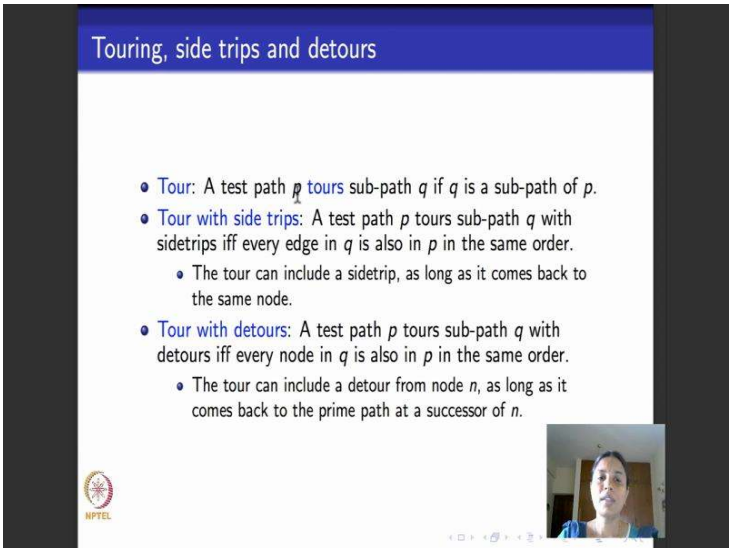


So, one important thing is that, I told you right, in the next module I tell you how to come up with an algorithm that will help us to design test cases that will need the test requirement of prime path coverage. But before we do that I will tell you we could have some problems. Like for example, you take a graph like this. Prime path coverage for this graph, this is a prime path 1, 2, 3, 4, 5, this is a prime path. It might so happened this graph if this graph corresponds to the control flow graphs is some piece of code here is a loop. It might so happen that this code from which this control flow graph is derived, the loop present at statement number 3 and 6 is such that you can never write you have to execute the loop at least once.

Like for example, if I take a C program. There are two kinds of loops: there is a do-while and a while-do loop right. While-do loop first checks the condition and then executes the loop, but if I have a do-while loop I have to execute the loop at least once before I move on. So, it might be an infeasible test requirement for that kind of a code when I say that you achieve this prime path without executing the loop. Why should I not execute the loop? Because if I execute the loop then the vertex three which is an intermediate vertex in this path will occur more than once. So, it will not be a prime path. So, 1, 2, 3, 6, 3, 4, 5 is not a prime path because 3 occurs more than once; 1, 2, 3, 4, 5 is a prime path. But to be able to write a test path and execute it in the code to achieve 1, 2, 3, 4, 5 the code might be such that I might have to go through this loop once. Like I told you write the code might have a do while statement.

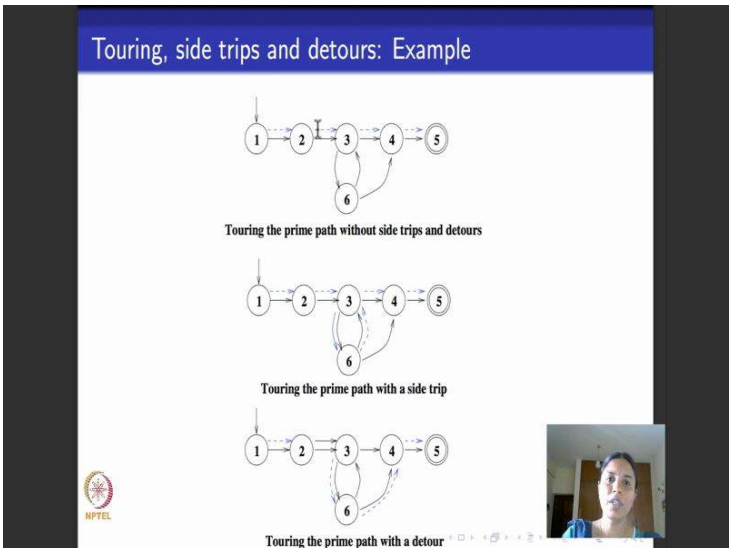
So, how do we get over this? We get over this by using a notion of a side trip and a detour. So, we will see what they are.

(Refer Slide Time: 34:09)



So, what is a tour? Tour I introduced you in the last module when we looked at graphs tour is a test path that tours is a sub path if q is a sub path of the overall test path. So, what is a tour with a side trip? A tour with a side trip is the following--- test path p towards the sub path q with side trips, if every edge of q is also in p in the same order. It is like taking a side trip as a part of a main holiday. And what is a tour with the detour? A tour with a detour is the test path p towards a sub path q with the detour if edges do not come in the same order, but vertices come in the same order.

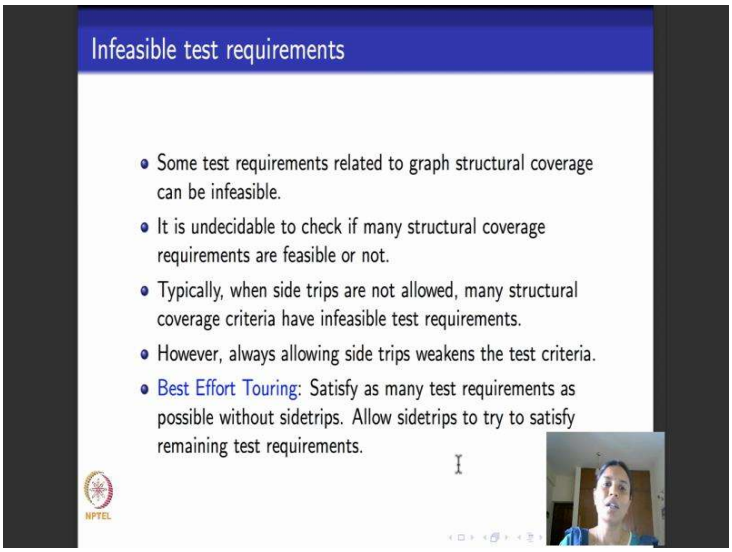
(Refer Slide Time: 34:52)



So, I will show you an example. So, here is my test path 1, 2, 3, 4, 5. I take the same test path to meet the prime path coverage of 1, 2, 3, 4, 5 with the side trip. So, what do I do in the side trip? Side trip says that it is a same path 1, 2, 3, 4, 5, but I do 1, 2, 3 take a side trip around 6 come back to 3 and do 4, 5. So, I retain the vertices that come in the main path. See, side trips says that you retain vertices that come in the same path in the same order; retain the vertices 1, 2, 3, 6, 3, 4, 5, but in the main path which is this blue dotted line the vertices occur in the same module.

Now what is a detour? Detour says you retaine edges in the same order, may or may not retained vertices in the same order. So, a detour looks like this 1, 2, 3, 6, 4, 5. It has left this edge it does not retained edges in the same order, but it retains this path. So, why do I need this? I need this because to be able to achieve this prime path coverage I might have to go through the loop. And if I go through the loop the prime paths ceases to be at prime path. So, I do a small work around then say I actually cover this prime path, but with the help of a side trip. Sometimes I might say I cover this prime path, but with the help of a detour. They have just extra additions that we add so as to not to alter the notion of prime paths being simple paths, but I still want to be able to achieve these test requirements.

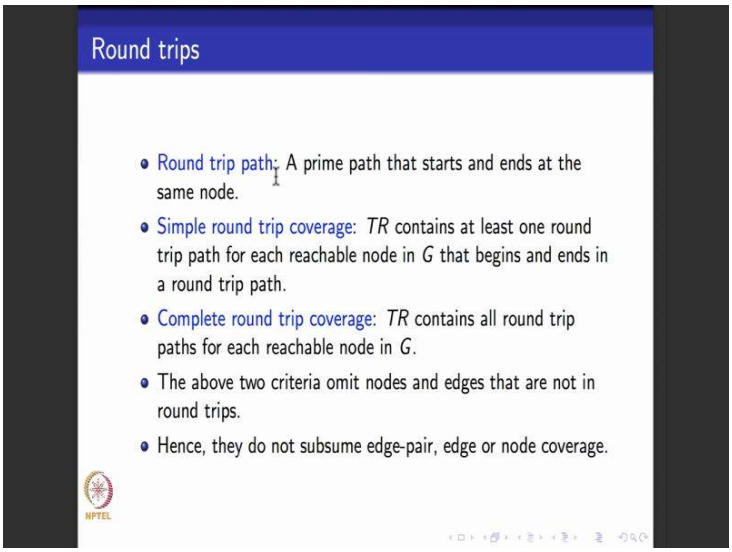
(Refer Slide Time: 36:30)



So, this is what I was telling you about. Some test requirements related to graph coverage criteria may be infeasible. In fact, it is undecidable to check whether even test requirement is feasible or not, we won't really go onto its details. But typically when I allow side trips, then I might be able to achieve test requirements with reference to a particular code.

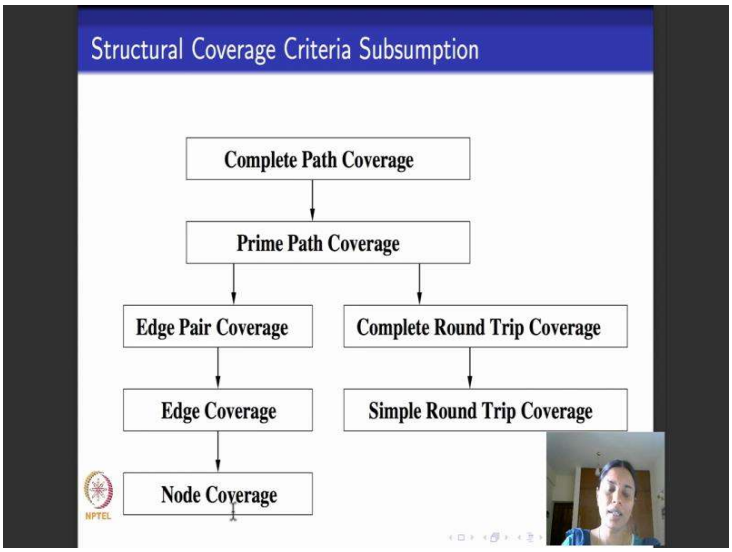
So, what is called a best effort touring is that you tried to satisfy as many test requirements is possible without side trips or detours, and then if you still have test requirements that are unachievable, consider using side trips and detours to be able to satisfy them.

(Refer Slide Time: 37:09)



So, what is a round trip? A round trip is a prime path that starts and ends in the same node, we have seen these, these suggest specific kinds of prime path. What is a simple round trip coverage? Test requirement contains one round trip path for each reachable node that begins and ends in the same vertex. What is complete round trip coverage? Test requirement contains all the round trip paths. So, they subsume edge and node; I mean they do not subsume edge pair edge or node coverage.

(Refer Slide Time: 37:42)



Now to summarize what are the various coverage criteria we saw; we saw node coverage, visit every node; edge coverage, visit every edge; edge pair coverage visit every path of length at most two, visit prime paths means compute on the prime paths right test paths that exactly visit every prime paths. We will see how to do this in the next module. Then, complete path coverage which means visit every path in the graph which is, I believe, a useless test requirement. And then we had these two which talk about round trips. They are basically prime paths, but begin and end with the same node.

And why have I put them in the structure? This structure captures how each of them subsume the other. We discussed this right, edge coverage subsumes node coverage. Edge pair coverage subsumes both edge coverage and node coverage. Prime path coverage, except for graphs with self loops, subsume edge pair coverage, edge coverage and node coverage. Complete path coverage subsumes everything, but is infeasible and useless.

So, what we will see in the next module is how do we look at algorithms that given each of these structural coverage test requirements, how do I come up with the test cases? Edge, edge pair and node coverage a easy to do, but prime path coverage is a nice algorithm. So, we will spend most of the next module looking at these algorithms.

Thank you.

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

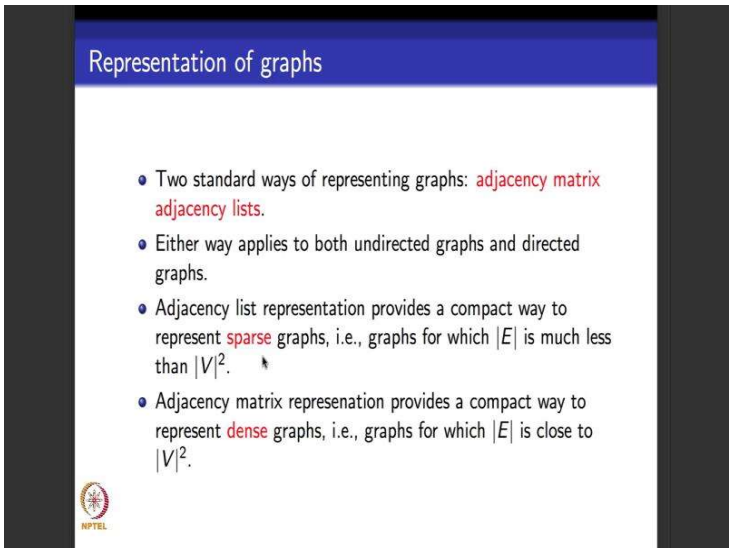
Lecture - 07
Elementary Graph Algorithms

Hello everyone. Welcome to the next module. In this module what I would like to spend some time on is relating to elementary graph algorithms. The last module we saw structure in coverage criteria related to graphs, we saw how to rectify test requirements for various structural coverage criteria; we began with node coverage, edge coverage, went on till we saw prime path coverage.

The next thing that I would like to focus on as far as testing is concerned is how to define algorithms that will help us to write test requirements and test paths that we achieve various graph coverage criteria. It so turns out the algorithms that will help us to write test paths and test requirements for graph coverage criteria use elementary graph search algorithms; most of them use in fact BFS- breadth first search. Some of them can also use things like finding strongly correcting components, finding connected components etcetera. So, once you know these algorithms very well it is a breeze to be able to work with algorithms that deal with structural coverage criteria and graphs.

So, what I am trying to spend time on in this module and in the next module is to help you to revise elementary graph search algorithms. We will look at breadth search first search in this module, in the next module we will recap depth for search and also look at algorithms based on DFS that will help us to output strongly connected components. Once you know these algorithms you will move on come back and look at the algorithms to write test requirements in test paths for structural coverage criteria based on BFS and DFS.

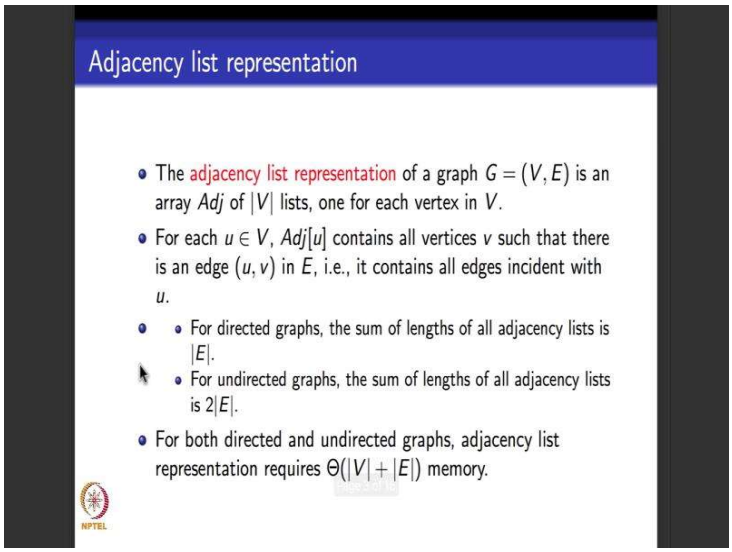
(Refer Slide Time: 01:59)



Let us begin looking at what are the standard representations of graphs that algorithms that manipulate graphs will look. Like graphs can be represented in two standard ways as you might know; as an adjacency list and adjacency matrix. Both undirected and directed graphs can be represented both as an adjacency list and as adjacency matrix. It so turns out that in testing when we look at graphs as data structures modeling various software artifacts, we will never look at undirected graphs they are not very useful to us maybe except rarely when we look at let us say class graph or things like that otherwise most of the graphs that we will deal with will be directed graphs.

But why I am saying this that the algorithms that we will look in this module and next module work both well for directed graphs and for undirected graphs. We will use then for directed graphs. So, we will go ahead and see what an adjacency lists looks like.

(Refer Slide Time: 03:01)

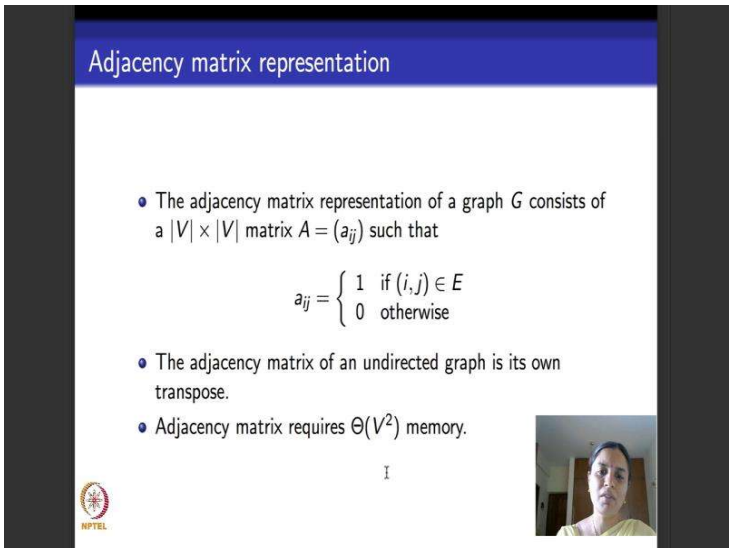


So, given a graph with vertex at V and at set E, what is an adjacency list? An adjacency lists keep an array of lists. How many lists are there in the array? There is one list for every vertex, so totally there are mod V lists. What do each of this lists contain each of this lists contain all the vertices that are adjacent to the given vertex v; that is it contains each vertex u such that (u, v) is an edge in the graph. So, what does and adjacency lists contain? It contains an array of mod V list one list for every vertex where the list for every vertex contains all the other vertices that are connected to this vertex through an edge.

What is the size of an adjacency list? If you see for directed graphs the directions of the edge do not matter right. So, when I have an edge (u, v) I practically have two edges (u, v) and (v, u). So, for undirected graphs the size of the adjacency lists is two times model: one edge two edges for every edge present in the graph in the adjacency list. And for directed graphs there is one for every edge there is one vertex somewhere in the adjacency list of some vertex, so the size of adjacency list is mod E.

So, for both directed and undirected graphs this total size of the adjacency list will be this. This first mod V in the theta component represents the number of lists and each list can be at most mod E size. So, the total size is theta of mod V plus mod E.

(Refer Slide Time: 04:40)



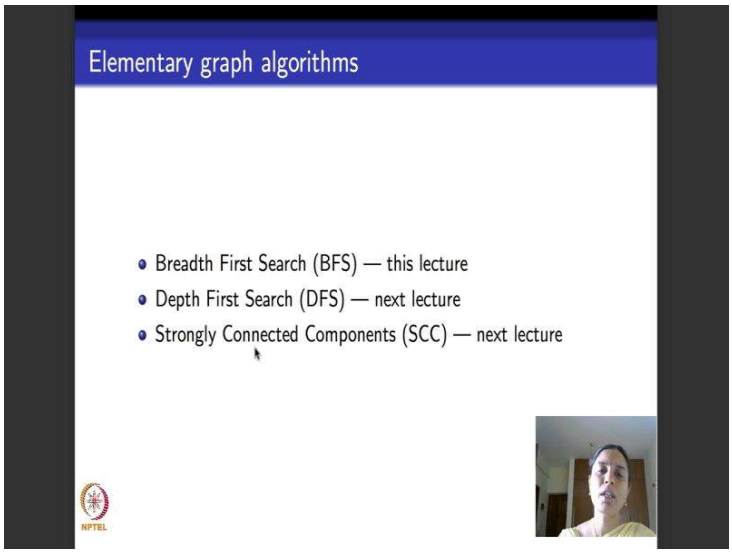
Moving on what is an adjacency matrix representation of a graph look like? Adjacency matrix of a graph is a mod V by mod V matrix. So, it has as many entries is that of number of vertices, it is a square matrix, and its filled with 0 and 1 entries. It contains a 1 at vertex v I and v j if the edge (v I, v j) is in the set of edges of the graph otherwise it contains a 0. So, if you see adjacency matrix will go directed and undirected graphs will need V squared, theta V squared memory, because they are of matrices of size mod V by mod V. In addition for undirected graph because the edges are there on both sides, the transpose of its adjacency matrix will be same as the given adjacency matrix.

So, which is good for what kind of graphs? Graphs could be dense or they could be sparse. If a graph sparse means it has very few edges; it has lot of vertices which has very few edges. Trees are examples of graph that has sparse. So, graphs are sparse then adjacency list is considered to be a good representation, because the size of each list for every vertex will be small. On the other hand if a graph is dense, which means it has lot of edges the number of edges is close to mod V squared, then an adjacency matrix is considered good, because the size of the adjacency matrix is fixed to be V by V matrix.

So, if there are many edges it is just means more one entry lesser 0 entries, whereas if it is an adjacency list for a dense graph then the size of the each list would be really long. So, for sparse graph adjacency lists are considered to be good, for dense graph adjacency matrices are considered to be good; otherwise there is no really big trade of about one

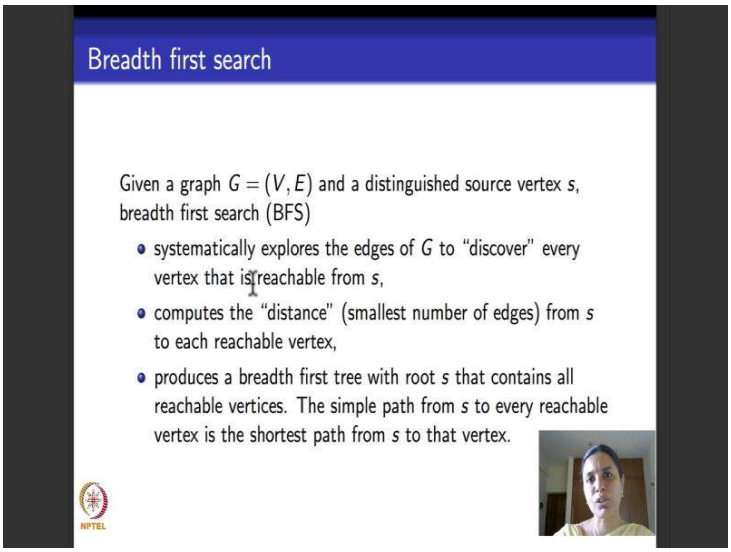
representation or the other it is just for the convenience of our this thing. In our algorithms we will mainly use adjacency lists. The algorithms that we will deal with in this two modules we will use adjacency lists.

(Refer Slide Time: 06:41)



So, what are the elementary graphs such algorithms that we will be looking at? We look at three algorithms, because these three are the main ones that we will need for test case generation. So, we will look at breadth first search which we will do in this module. The next module I will help you recap depth first search and we will also depth first search and breadth for search have several applications. So, we look at one particular algorithm which deals with how to compute strongly connected components or connected components using depth first search. So, those two we will look at it in the next lecture.

(Refer Slide Time: 07:12)



So, we will move on and start with breadth first search. What is breadth first search do? As the name says it searches through a graph, and how does it search through a graph? It searches through a graph in a breadth first way. That is it starts it search from a particular vertex, let us say call it source vertex and what does is that it first explores the adjacency list of the source vertex; that is, it explores the span of the breadth of the graph at the vertex x. Once it is finished exploring the adjacency list of the vertex s which is the source, what is it mean to explore, it adds it to a particular queue that it gives, it goes ahead takes a one vertex at a time and explores that vertex adjacency lists and it goes on like this.

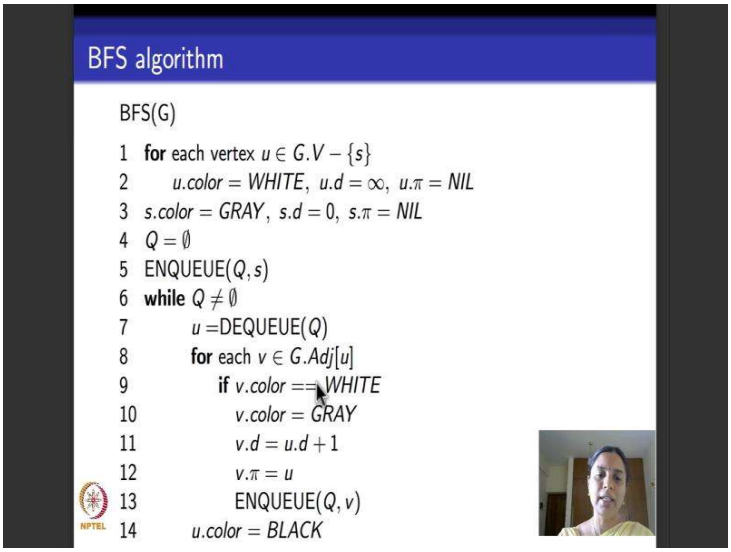
So, this way of searching or traversing through a graph by exploring the adjacency list of each vertex fully is called breadth first search, because it explores the graph in a breadth first way; does not go deep unlike depth first search. As it explores it also computes a few things which are useful for us to keep. What it says is as it explores it also keeps track of given a fixed source s what is the distance of any other vertex in the graph from the source s.

If we look at the graphs that do not have any attributes like weights rather things attached to its edges. So, for us distinct plainly means the number of edges. So, how far is the particular vertex in the graph from the designated source vertex? How far mean how

many edges are there between the particular vertex in the graph that is reachable from the source vertex. That is what is called the distance of a vertex from its source.

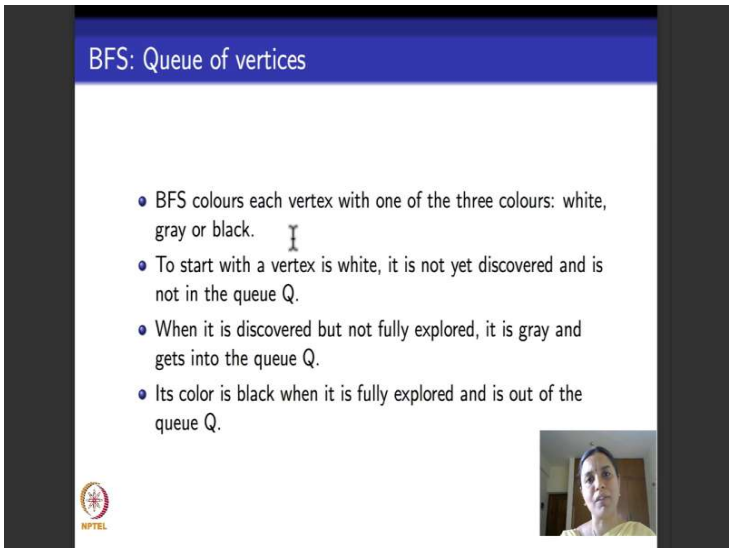
Breadth search first also computes the distance. It so happens breadth first search computes this smallest distance or the shortest distance we will see what is that in a meanwhile. And it once you explore a graph by using breadth first search then the result of that exploration is a tree containing the path from the source to call other vertices that are reachable from the source. Such a tree is called breadth first tree. Breadth first search algorithm can be used to output the shortest distance of every vertex from the source. It can also be used to output the breadth first tree which contains the shortest path from the source to every other vertex.

(Refer Slide Time: 09:32)



So, here is how the algorithm looks like; before I show you the pseudo code of the algorithm I will tell you what it does.

(Refer Slide Time: 09:40)



So, it takes the graph and the main job that breadth first search does is to be able to color each vertex in the graph; keeps the queue of vertices and that queue contains the queue of vertices that are colored grey. So, what is breadth first search do? It keep it color codes every vertex and there are three kinds of colors that it keeps. To start with every vertex is colored white. And then vertex that is colored white further changes to color grey and after its colored grey it changes to color black.

Once it changes to color black we say we are done with exploring that vertex fully. So, to start with all vertices are white, and when does the white vertex become grey? It becomes grey, when it enters the breadth first search maintains a queue of vertices and when it enters queue it becomes grey. So, it becomes grey when it is first discovered as a part of the adjacency list of some vertex. And then it is put into the queue.

When it is put into the queue the aim is grey colored it is discovered now, but I am yet explore this vertex's adjacency list. So, I have put it into queue to remember that I have to do that. And I move on and explore the adjacency list of all other siblings of this particular vertex. And then when I come back and explore this particular vertex's adjacency list and finish exploring that then I color it black, and when I color it black I will move it form the queue.

So, how does the breadth first search work? So, here is what is the pseudo code for breadth first search. So for each vertex that is not the source, remember, s is the source

for which breadth first search algorithm start. So, for each vertex read G dot V s for each vertex in the graph that is not the source which is not s set its color. So, we keep three attributes with each vertex; we keep a color attribute which tells you what the color of the vertex is: it could be white, grey or black. We keep a d attribute, d for distance which tells you what is the distance in terms of the number of edges of this vertex from the source vertex s. And we keep a pi attribute: pi representing predecessor of parent which tells you who is the parent of a particular vertex in the breadth first tree that is algorithm is going to output.

So, what are the three attributes that we keep with every vertex? A color attribute, a distance attribute which tells its distance in terms of a number of edges from the source, and a predecessor or a pi attribute which tells you who is the predecessor of this particular vertex in the breadth first tree.

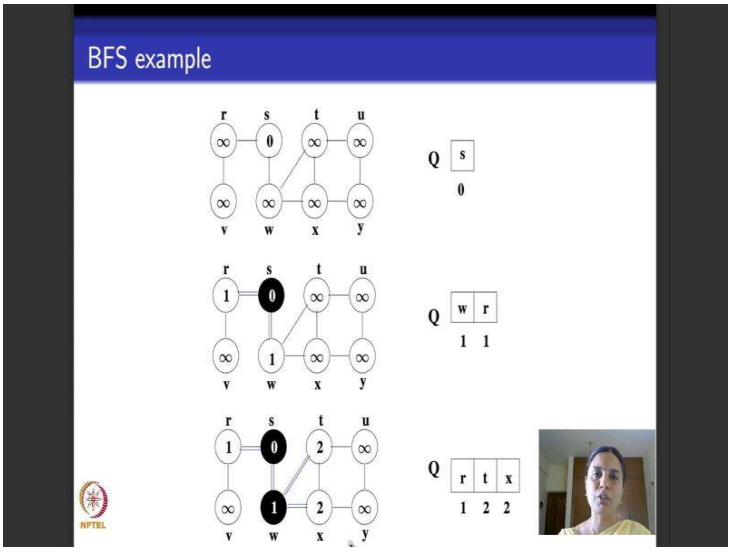
So, to start with breadth first search algorithm colors each vertex white, sets its distance to be infinity, because the distance will reduce right, to start with I do not know how far it is from the source. And if the tree is not yet developed so the breadth first search algorithm is just began, so its pi attribute is NIL. Now it begins its search from the source vertex s. So, the first thing that gets into the queue; Q is the Q queue it maintains. The first vertex that gets into the queue is s. So, if you see this enqueue Q s puts s inside the queue Q. So, when puts inside the s Q the color of the vertex s is grey; s the distance from s itself is 0, s is its own parent so it does not have predecessor, its predecessor is set to NIL. So, Q is initialized and s is put into the queue.

So, as long as there are vertices in the queue which means as long as there are grey color in the vertices what do you do? You pick up the first vertex available from the Q, you dequeue it right and you start exploring adjacency list that is what this one for loop line number 8 does. So, it says for each other vertex in the adjacency list of the vertex u in the graph. If its color is white then you make it grey, which means you have discovered that vertex. Now you set its distance from the source s to be whatever the distance from the source s of u was plus 1. And then you say its parent in the breadth first search tree is going to be u, because I found it as the part of the adjacency list of u so it is natural that its parent is u.

Once you do this, you add V to the Q that you maintain. And you keep doing this, keep doing this, till you finish exploring the adjacency list of u. Once you fully finish exploring the adjacency list of u, you come out and color u as black. So is it clear what the algorithm does? Just to quickly repeat. To start with the color each vertex white sets the distance and bi attributes, it begins its search at the source vertex s, add this to the queue, colors it grey, sets it g and pi attributes, and then for each vertex in the Q it takes it out from the Q explores the adjacency list of the vertex that was just taken out fully.

What does it means to explore the adjacency list of the vertex? It looks at each vertex in the adjacency list of the vertex u, if it was colored white, which means it was not yet discovered it says I am discovering it now marks it grey; sets its distance from s as the distance of u from s which was already set plus 1; sets its parent in the breadth first search tree to be u and adds it to the Q. When its finish doing this for all the three adjacency list of u, it removes u from the Q here and colors it black.

(Refer Slide Time: 15:12)



So, here is an illustration of how breadth first search works on a small example. In this particular case I have taken an undirected graph as a example, but as I told you in the beginning directed or undirected graph does not matter. The algorithm will work fine because it just takes an adjacency list as an input and does not really worry itself about whether the graph is directed or undirected.

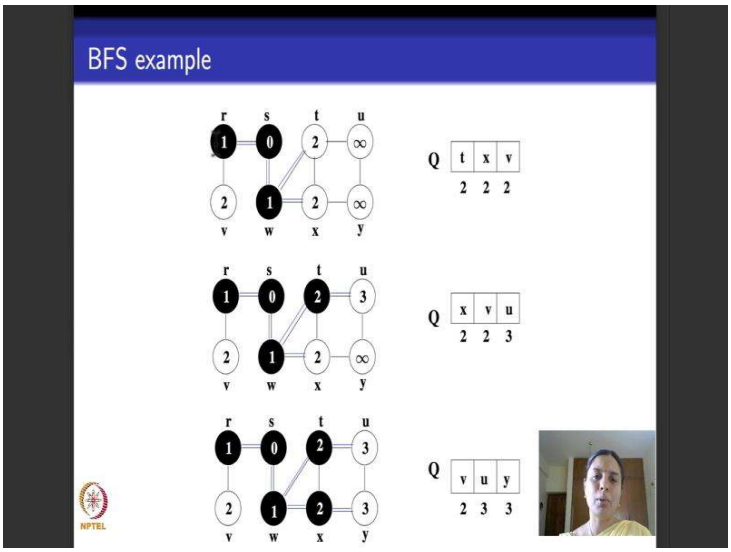
So, there are eight vertices in the graph; the vertex s is source vertex. So, what are these various things? The other vertices are r, s, t, u, v, w, x, y. So, I have also put these parameters inside the vertex, what they correspond to is the d attributes. You see the distance from s, the source to itself, is 0 to start with every other vertex is this thing infinity from the source s. The Q initially has s its distance from itself is 0.

Now I start exploring the adjacency list of s because that is what is there in my queue. So, what are the vertices that are adjacent to s? That is r and w. So, this blue color arrow, I hope you can see them means that I have set the predecessor attribute. So, I have set the predecessor of r as s and I have set the predecessor of w also as s. So, so far my breadth first tree that I output which I read out from the predecessor attributes contains this sub graph; it contains the node s with its successor as r and another successor as w. And because I have discovered these tow vertices for the first time I have put them in the Q and set their distances from the set s to be one because they are reachable from s through one edge.

Now, you can put them in any order. It just so happens in this case that I put w first and then r, nothing will go wrong in algorithm if you put r first and then w. Now because I have put w first I start exploring the adjacency list of w. What are the two vertices that are adjacent to w? If you look at the third figure here, t and x. So, I add them to the breadth first tree which is again setting its pi attributes. So, I color this as blue indicating that the predecessor of t is w and then I color this edge also blue indicating that the predecessor of x is also w and I add t and x to the Q. And how far are they from the source s? They are at distance two from the source s right because they there are two edges you take two edges to reach t.

Please remember that I had not colored vertices grey in this example, because it became a little cumbersome to do it. But what all the vertices are there in the Q are all colored grey, but it is not depicted in the figure here.

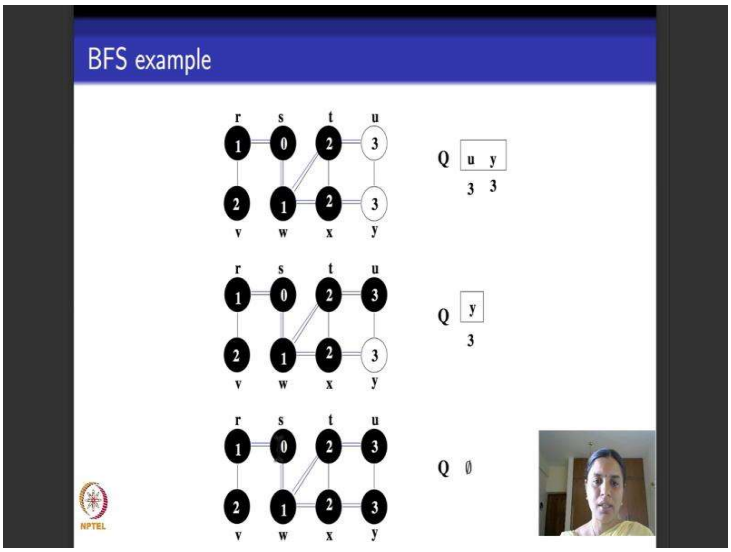
(Refer Slide Time: 18:00)



And move on like this: now I start exploring the adjacency list of vertex r because that is what is there at the head of the Q. So, V is adjacent to r, so add V to the Q here, color r black and come out.

Now, the next element whose adjacency list needs to be explored is that of t. So, I take t; there are two vertices at adjacent to t that is u and w; w is already been explored its color black so no need to explore it once again. So, add u to the Q set its distance from s to be 3 and color t black. And I go on like this the next vertex to be explored is x. So, the if I take x the only other vertex unexplored vertex that is incident to x is that of y, then remaining two vertices t and w are already colored black. So, I add y to the Q set its distance attribute to be 3 and color x black.

(Refer Slide Time: 19:00)

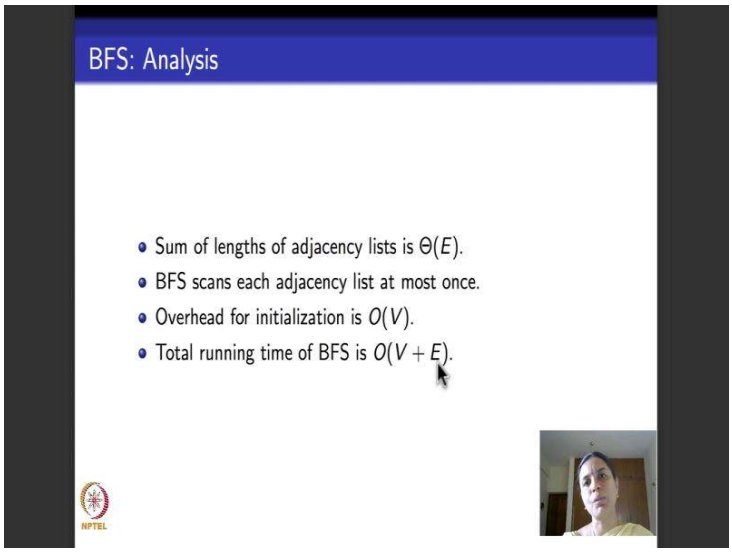


And move on like this and keep adding till I can. At some point the Q will start shrinking and the Q will become empty. So, when will the Q become empty? When I have finished all the vertices and colored all of them black.

So, if you look at this final tree how to read this tree? It says s is the source and you just read out the blue arrow edges as the tree. So, the tree has an edge from s to r and the tree goes like this s to w, w to t, w to x, x to y, t to u. So, breadth first search as an algorithm outputs this tree by saying that I have explored all the vertices and it also outputs the distance of each vertex from the tree. So, I hope the algorithm is clear to you. So, what do I do to start with I take a graph, I start with the source I explore the adjacency list of each graph that is what it means to say that I go in a breadth first way and I keep doing till I finish exploring.

And the output of breadth first search is basically a tree that I read out from the pi or the predecessor attributes, and the distance of each vertex as it occurs in the tree from the source s.

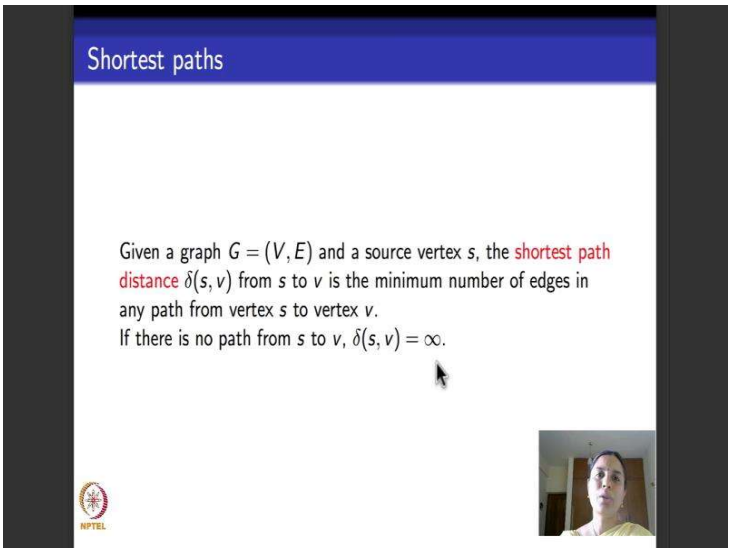
(Refer Slide Time: 20:14)



So, what is the running time of breadth first search? If you go back and see the pseudo code of the breadth first search, so there is this initial for loop that runs once for each vertex, so this takes order big O of V time. And then this white loop it runs, once it enqueues a vertex it does not really go and put is back again; once it enqueues and dequeues in each vertex gets in and gets out of the Q exactly once. And for each vertex it explores for loop along the length at the adjacency list of that vertex.

So, BFS scans each adjacency list at most once and sum of the lengths of adjacency lists that we saw is theta E. And I told you the initial for loop takes big O of V. So, the total running time of BFS is big O of V plus E. So, it is a linear time algorithm that is linear in the size of the graph. So, when I talk about the size of the graph which typically consider the vertices and the edges; the number of vertices and the number of edges we do not really say only the vertices it is a wrong thing to see you take the loop size of the edges also as very much the part of the graph. So, BFS runs in time linear of the size of the graph.

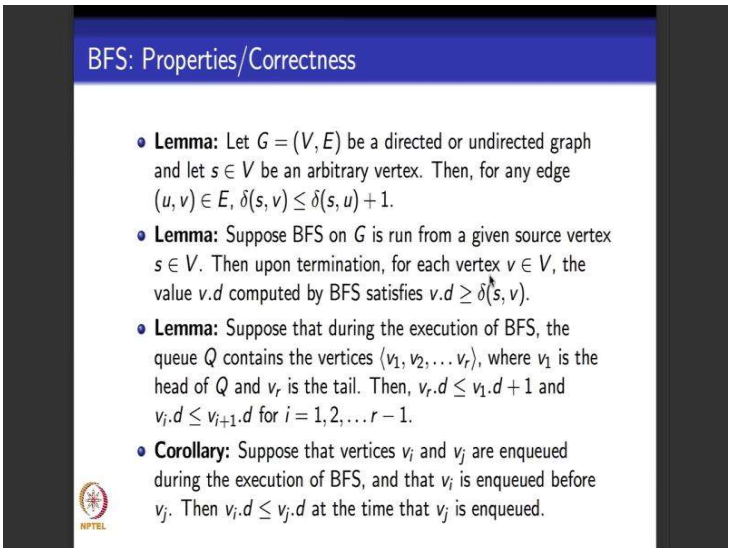
(Refer Slide Time: 21:27)



So, I told you along with breadth first search it also outputs a few other things that will be useful. It outputs what is called the shortest paths distance from the source s. So, BFS runs from a fixed source and shortest path distance is the distance in the BFS tree in terms of the number of edges, it so happens that BFS outputs the shortest paths distance of each vertex from the source. Standard terminology that a book called Introduction to Algorithms by Cormen Leiserson; CLRS (Cormen Leiserson, Rivest and Stein) outputs the shortest paths distance and the notation that is used to this is delta. I have taken the pseudo code these examples of that book it is a pretty standard book.

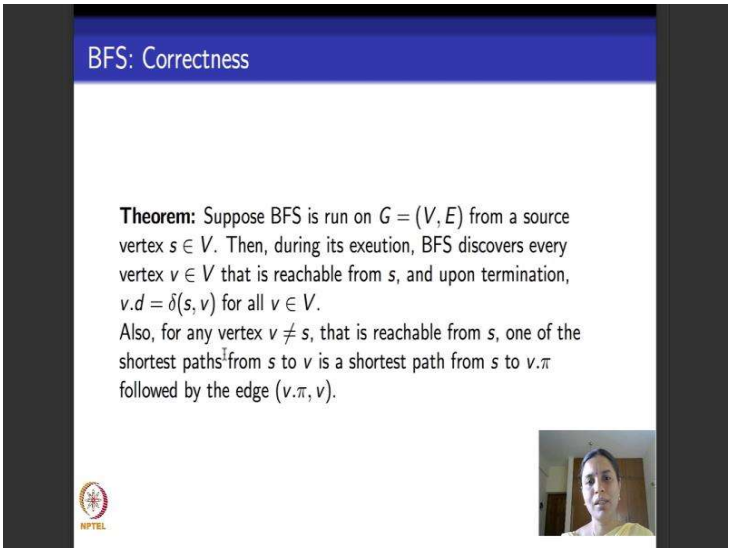
So, it outputs the shortest path distance written as delta from the source s to each vertex V which is the shortest path in terms of the number of edges that is encounters.

(Refer Slide Time: 22:26)



There are several properties that you need to show that actually BFS is correct; what is it means to say BFS is correct?

(Refer Slide Time: 22:32)



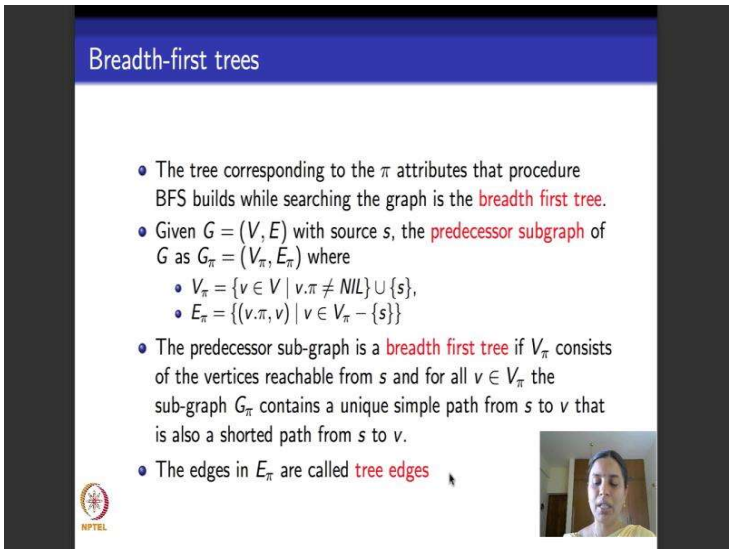
So, we will state and prove a theorem like this, let us read out this theorem. It says suppose breadth first search is run on a particular graph from a fixed source s then during its execution BFS manages to find or discover every vertex that is reachable from s in the given graph g and when it terminates it outputs the shortest path distance from s to each vertex in the graph. For each vertex that is reachable from s; one of the shortest paths

will be the path that the breadth first tree takes. So, for vertices that are not reachable from s; this particular example graph that we saw did not have anything like that. For vertices that are not reachable from s breadth first search will not be able to output anything because it will not explore it.

So, what you basically do is you begin breadth first search from a fresh source; that is not s then you can explore the graph once again from other source and reach all the vertices that were not reachable from the source s. And you can repeat this process to get a forest of breadth first trees till we complete. The way I have presented this pseudo code, it is presented in such a way that we present it as we are exploring from a fix source s and then we stop.

So, those vertices that are not reachable from s, if there are any in the graph will not be explored, but there is no harm in running the breadth first search algorithm again from another source to repeat the process. Please remember that. So, you can explore all the vertices in the graph instead of getting one breadth first tree you will end up getting a breadth first forest.

(Refer Slide Time: 24:11)



So, now let us go ahead and discuss about how to use the pi attributes to be able to get the breadth first tree. If you see this example what I have done is that a blue lined arrows are the pi attributes. How do we use that to be able to output the tree? So, what do I do? I do this. Given a graph G is equal to V comma E with the source s which is basically

input to the breadth first such algorithm, I generate what is called the predecessor subgraph by using the pi attributes.

So, what are the vertices of the predecessor of graph? The vertices of the predecessor of graph are all the vertices of the given graph such that they are reachable from s. Once they are reachable from s then pi attribute will be non NIL right, it will not be a NIL thing because BFS will reset it to the parent vertex and then you take this source s. What are the edge set of the edges of the predecessor of the sub-graph? It will be the vertex and its predecessor, because that is how edges in the tree look like.

So, you can prove what theorem which says that such a predecessor of graph is actually a tree; in the sense that it is connected and it has no cycles. So, what breadth first search outputs is a tree or a forest of trees containing shortest path from the source to each reachable vertex. The edges of this tree are called tree edges. Here are the lemmas that we need to use to show the correctness of breadth first search, because the main focus of this course is not on algorithms, I have just stated the lemmas and not proved their correctness. Feel free to refer to a book like CLRS to check for correctness or proofs of all these lemmas.

So, initially you need a lemma which says that the shortest path distances increases as I explore the graph. So, it says when we start from the source then you reach a vertex u from the source having delta of s u has now to be, and suppose u comma v is an edge in the graph then what is the shortest path distance from s to v. It is at most the shortest path distance from s to u plus 1. Because I need to be able to consider the paths to reach from s to u and then consider the edge u v, it cannot be more than that this is popularly called triangular inequality.

The second lemma says that suppose you run BFS on a graph from the given source vertex then when this algorithm terminates, for each vertex v,, the value the distance v dot d the BFS algorithm computes will be at most at least the shortest path distance; sorry it will be at least the shortest path distance, it could be more n fact it will be equal to the shortest path distance for this kind of breadth first search algorithm. Third lemma says that as I go on exploring the graph, the shortest path distances increases. It monotonically increases that is what the third lemma says.

Fourth lemma says that the Q respects the order in which, the Q strongly influences the order of exploring the vertices. For example, if you go back and see in this code, right in the beginning, when I look at the source s right it has two vertices in adjacency list r and w; I put w first and r next. And I told you at that itself that there is no sanctity about it I could put r first and w next. I would have got a slightly different breadth first tree, but nonetheless there would still be the shortest path distance from the source. That is what this lemma says. It says that the queue Q respects the distance enqueue and dequeue respects the distances of the vertices.

So, using these lemmas, I will be able to show correctness in BFS. Correctness in BFS basically says that I do BFS transverse it will finish exploring every vertex that is reachable from the source, and it will output the shortest path distance in terms of the number of matrix from that source to every other vertex. So, this is all I had to tell you about BFS. In the next module we will look at depth first search and also we will look at algorithms for connected components in the graph.

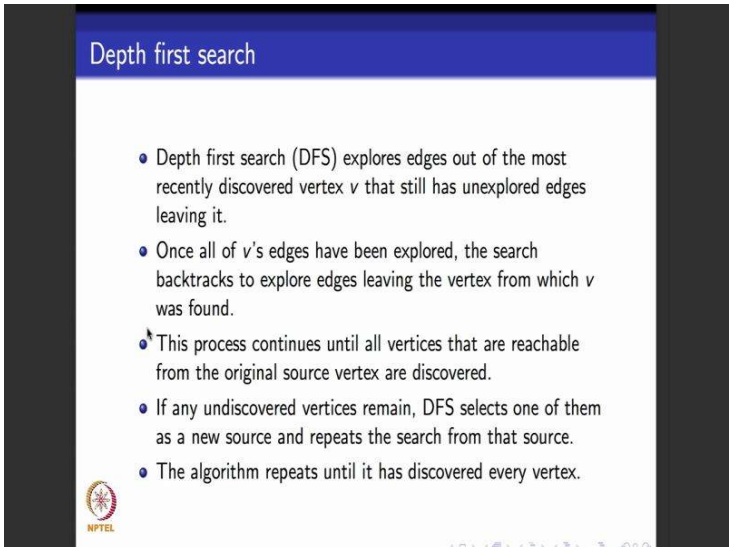
Thank you.

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

Lecture - 08
Elementary Graph Algorithms

Hello again. The goal of today's lecture is to be able to do depth first search algorithm which is another popular graph algorithm followed by strongly connected components how to use depth first search to output what are called strongly connected components in a graph.

(Refer Slide Time: 00:30)

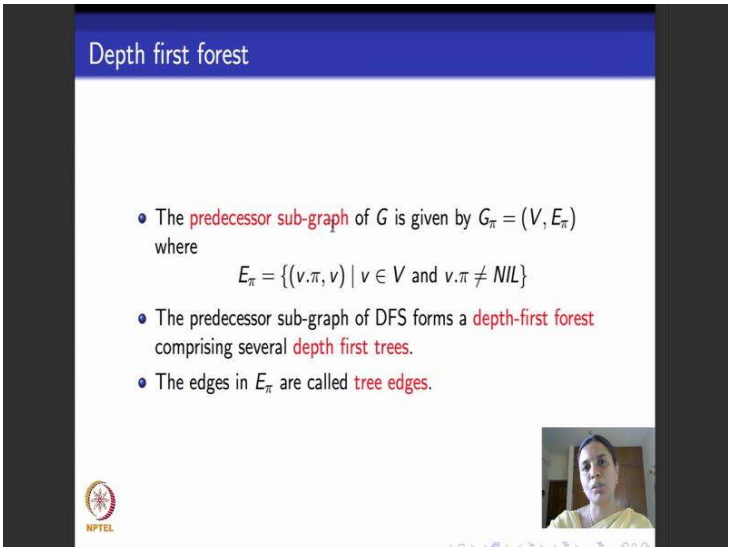


So, we will begin with depth first search. Unlike breadth first search that we saw in the last module, depth first search is also meant to traverse or explore the vertices of the graph, but in a depth first way. So, it goes deep down a graph as much as possible. It begins at a particular designated source like BFS and then instead of exploring the adjacency list of that source fully it takes one successor from that source goes to that successor, then it picks up one successor of the successor goes to that successor and so on. So, it goes deeper in the graph and first finishes exploring the graph to the deepest possible path that it can trace from the source.

Then it backtracks comes back and picks up the next vertex in the adjacency list of the source, and goes deep down that vertex. And when it finishes going down for that vertex

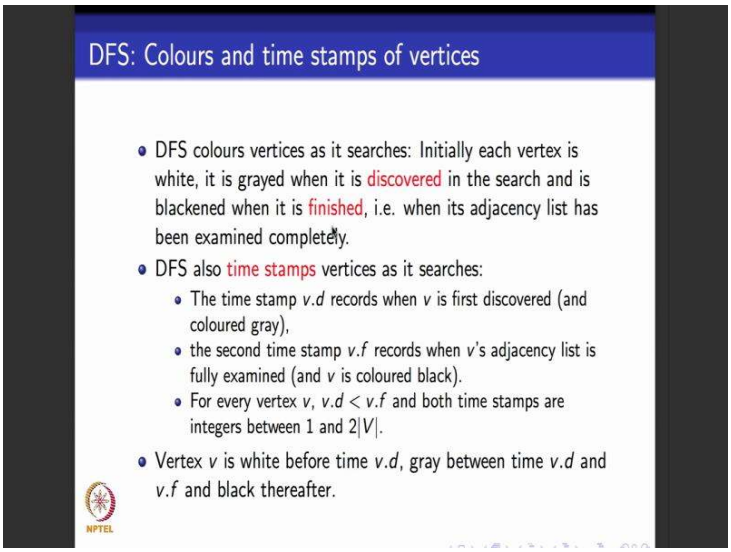
It comes back, picks up the next adjacency vertex adjacent to s, goes deep down and when it finishes exploring the adjacency list of each vertex deep down it finally, comes back and colors s black right. So, the goal of depth first search is also to traverse a graph and produce a tree, but unlike depth first search it goes deeper down in the traversal. The breadth first search grows breadth first in the traversal.

(Refer Slide Time: 01:57)



Like BFS, DFS also keeps several attributes it keeps the pi predecessor attribute associated with each vertex, it keeps a color associated with each vertex it keeps 2 kinds of time stamps unlike breadth first search I will tell you what they are very soon, but what does the pi attribute look like? Pi attribute basically is useful to produce or output the depth first tree right, set of depth first tree from different sources we will constituted depth first forest and the edges in this tree are called tree edges. So, as and when I explore the graph deep wise, I said the pi attribute of each vertex that I encounter to be it is parent, and when I consider all the pi attributes this way I get the full predecessor sub graph which happens to be a tree or a forest of trees based on whether the graph has one connected component reachable from the source several connected components reachable from the source.

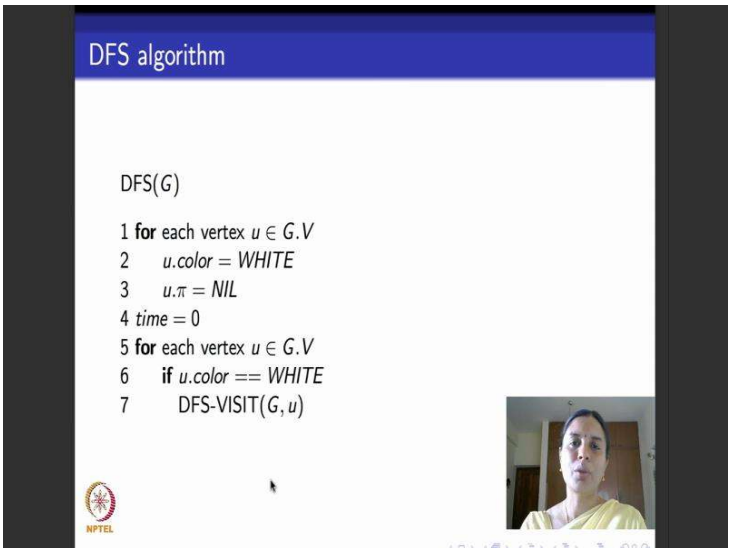
(Refer Slide Time: 02:54)



So, like BFS, DFS also goes ahead at first discovers a vertex and then it also finishes the vertex. Now what we do is unlike DFS we keep 2 different kinds of time stamps here, a time stamp call dot d which is given when a vertex is first discovered and which means a vertex which was originally colored white now becomes colored grey. And then another time stamp called dot f, f for finish time stamp which is given when the vertex is colored black right. When the adjacency list that vertex is fully examined. So obviously, a vertex first needs to be discovered before its adjacency lists is fully explode at the vertex is finished.

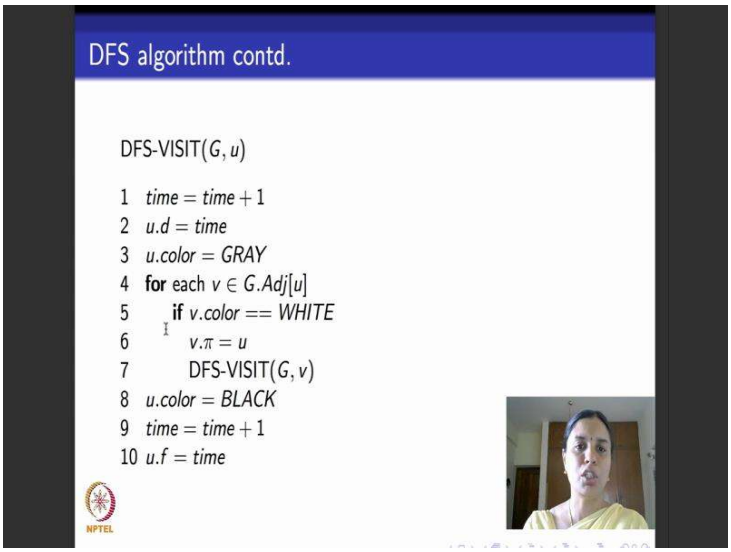
So, the d timestamp that is given to a vertex is always strictly less than the f timestamp that is given to the vertex. And the d and the f timestamps cannot be more than the number of vertices in the graphs, because that that many paths could be there assuming that the whole graph is connected one for d timestamp and one for f timestamp. So, they are basically values between 1 and 2 mod V.. So, before it is given the d timestamp the color of a vertex is white. Between when it is given the d timestamp and the f timestamp, it is color is grey and when it is colored black we give the finish timestamp to a vertex. So, here is how the algorithm looks like. I have split this algorithm across 2 slides because I could not fit in to one slide.

(Refer Slide Time: 04:24)



So, this is the initialization part as done for breadth first search. You start for every vertex you color at white you set it is pi attribute to nil. If you remember in the BFS code, we had set it is distance attribute from s to 0. Here we don't do the distance attribute we do the discovery and finish time stamps. For that I need a generic variable called time which I will use to set the dot d and the dot f timestamp. So, I initialize the generic variable time here to be 0. So, after I have done this, what I do is for every white colored vertex in the graph I begin this procedure called DFS visit from that vertex. What does the procedure DFS visit to? DFS visit first thing it does is increments the timestamp because it is beginning to search from a new vertex and it says this new vertex from where I am beginning my search which is the vertex u is discovered now.

(Refer Slide Time: 05:17)

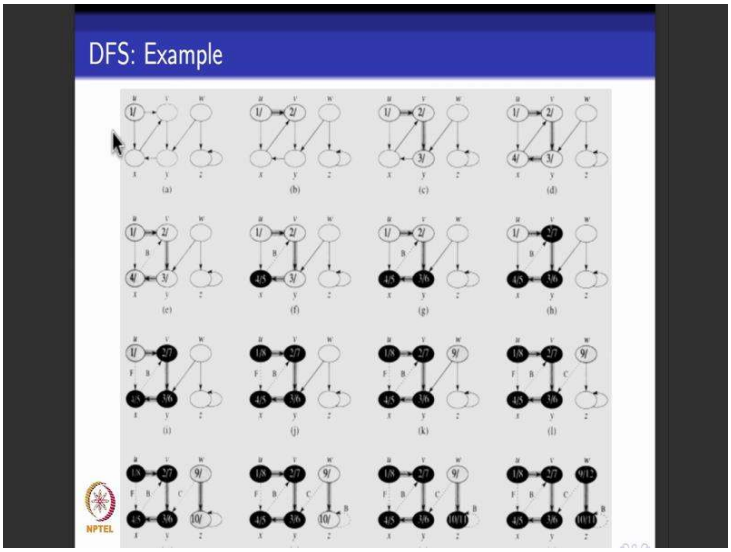


So, it sets the timestamp that it is just incremented as the discovery timestamp for the vertex u, and because I am going to begin exploring from the vertex u which was white, I set the color attribute of u to be grey. Now I start exploring the adjacency list of u. So, if there is a white colored vertex in the adjacency list u, I have to color that vertex grey and then start exploring deep down that vertex v. So, I pick up the white colored vertex call it v that is present in the adjacency list of u, and then the first thing that I do is I say u is the predecessor of v in my depth first tree, because I found v as I was exploring u. Now I have to go down and deeply explode the path that come out of v right. So, at v I recursively call the same procedure DFS visit. So, assuming that this code gets instantiated here, what would you do, you increment the timestamp you say that the vertex v is discovered, set its timestamp, set color of v to gray and go inside the adjacency list of v.

If you find another white vertex their you call the same procedure again for that vertex in the adjacency list. So, this way DFS let us you go down the path corresponding to a particular vertex. And when you finish exploring all the paths deep down from a particular vertex u, which means you finished exploring this recursive call when you come out of this recursive call for every vertex in the adjacency list of u, then you say I have finished exploring the adjacency list of u fully. Then you color u black, increment the time right, time variable that you kept as a global variable, at say that u is finished

because it is colored black and set is finished timestamp to be the current value of time right.

(Refer Slide Time: 07:39)



So, we will see an example to make it clear. What I have done this I have squeezed in the graph all in one slide. I hope you can read this which is not what I did for breadth for search here I have squeezed the whole thing into one slide to make it better understandable.

So, here is this graph you look at the top left corner that is the given graph. How many vertices does it have? It has 6 vertices u, v, w, x, y and z right. Unlike breadth for search here just for illustrative purposes, I have followed CLRS and taken an example which is a directed graph. This is the example from the same book by CLRS. So, what I do is u is the source from where I begin my depth first search. Now if you look at this graph a bit before you start doing depth first search of the graph, you realize that the vertices v, x and y are reachable from u, but the vertices w and z are not reachable from u. If you notice there is no path, no edge that connects either of these vertices to w and to z. So, when I explore starting from u, I will be able to explore only these 4 vertices. And then I have to start my DFS search again fresh from w to be able to explore the remaining two vertices w and z. And what will be the output of the algorithm? It will be a forest containing 2 trees. One with u as the root or the source of exploration, and one with w as the source of the root of the next depth first tree.

So, will begin with u. So, what is this label inside u? Read it has one slash nothing. So, there are 2 parts to a label there is a numerator apart thing of this label as a number having like a fraction having a numerator and the denominator. The numerator part, the part on the top, the left side of the fraction indicates the discovery time the dot d time of a vertex. The denominator part indicates the finish time or the dot f time the vertex. Right now to begin with because I am exploring DFS from u as the source vertex, I say discovery time of u is one, because I am exploring. Not yet finished exploring the adjacency list of u fully. So, no finish time is assigned to u. So, the right hand side of the bottom part which contains the finish time is left black. Now I explore the adjacency list of u as per what this pseudo call says. How many vertices are there if you see in this graph? There is a v in the adjacency list of u there is x in the adjacency list of u. I can choose either of them this particular example let say be go to v.

So, v which was originally colored white now becomes grey color. Again I have not depicted like in BFS, I have not depicted the color of the vertices here, the grey color is not when depicted just the black color is when depicted for clarity sake. So, I have discovered v, my timestamp is got incremented to two. So, I have sign discovery timestamp of v to 2 and then I say this shade of this arrow indicates that in the depth first tree u is the predecessor of v. The pi attribute assigned to v assigns the value as u. Now what do I do? I have to recursively call the procedure DFS visit from the vertex v which means I have to explore the adjacency list of the vertex v. In this example it just so happens that there is only one vertex adjacent to v in the adjacency list which happens to be y.

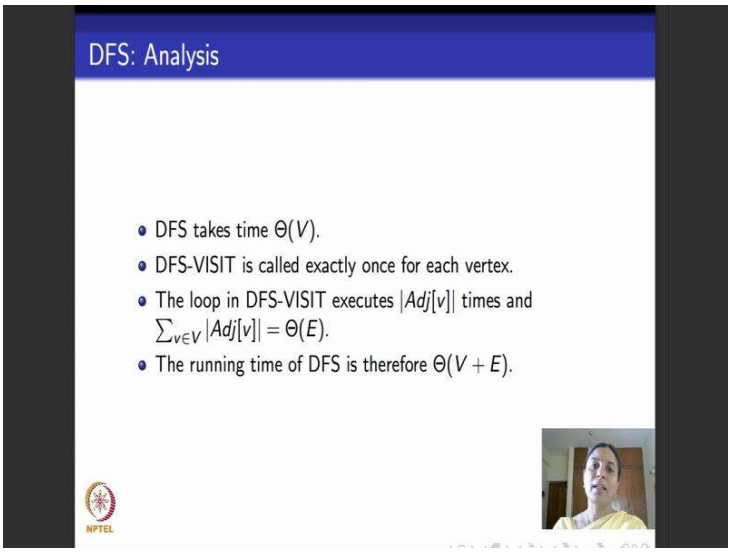
So, I say I go their y has been discovered I assign it is discovery timestamp to be 3, and then I said the predecessor y to be v in the DFS tree. Now I start exploring recursively the adjacency list of y. If you notice, I am going deeper in the graph right. I did not bother after doing v to come to x, because I am not doing breadth first search. After doing v I went to it is successor y because I am doing depth first search. Now I explore the adjacency list of y. What is there in the adjacency list of y in this example? It is just x. So, I discover x, assign it is discovery timestamp to be 4, and set y as the predecessor of x by putting the shade. So, so far the depth first tree that I have generated looks like this. It has these 4 vertices u, v, y and x and the edges of the tree are these grey shadow edges right. Now I repeat the procedure, I have to look at the adjacency list of x. If you

look at the adjacency list of x what is the only vertex that is present in the adjacency list of x, that is v.

But remember v is already being discovered. So, v is not colored white right. So, I do not go back and read discover v right. So, I let it be there is no other vertex of the adjacency list of this vertex x. So, which means I have finished my exploring my vertex, my search from the vertex x. So, I assign a finish stamp of 5 which is one more than the discovery stamp of 4, and then I color x black. I move on, I repeat this and then I say I have come back to the same thing whatever was holding for x holds for y, there is no more to explore. So, I finish y color it black, assign a finished time. And then I move back to v same thing I finish v, color it black assigned a finished time move back to u, finish u, color it black assigned a finish time. No more to explore at this stage if you look at this graph that is labeled with j, I finished exploring the full thing, the remaining 2 vertices were not reachable from my source u. Those 2 vertices were w and x.

So, I start fresh ones more depth first search from the vertex w. The last finish timestamp I had was 8. So, assigned for discovery timestamp of 9 to w, start and repeat the same exploration from w. In this case I finish first enough through all these edges because there are only 2 vertices that nothing more to it, and this path again leads to all already black colored vertex. So, I repeat the same procedure for w and get this. So, the final output of my DFS algorithm we will 2 depth first trees. One that has its root at u and has all these vertices, the grey colored entity is the tree that you draw out, and the next depth first tree in the forest has it is root as w and I just has these 2 vertices and one edge. So, this is how DFS works.

(Refer Slide Time: 14:22)



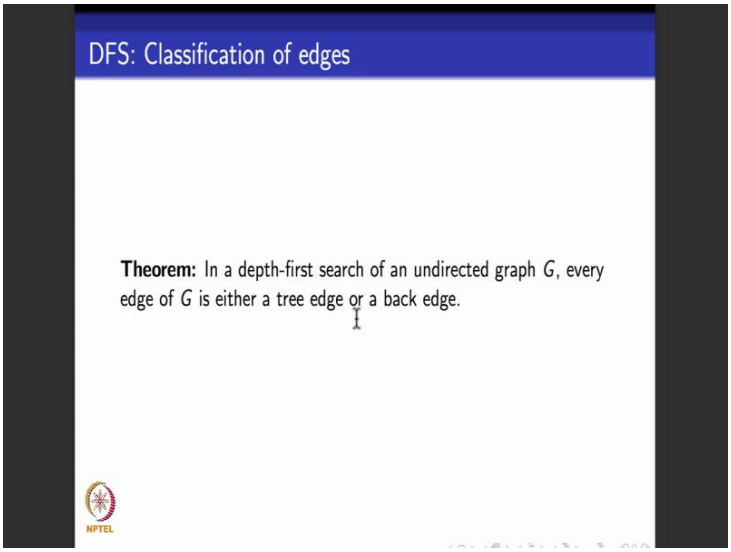
What is the running time of DFS? Let us go back and look at the code this for loop which does the initialization takes mod v time, and then the procedure DFS visit, how many times does it run? It runs at most once for every vertex that is reachable through an edge in the graph from the source. So, the initial thing takes order V time. DFS visit is called exactly once for each vertex it does not call a vertex that is already colored black.

So, the loop in DFS visit if you see there is one more loop here this loop in this recursive procedure DFS visit, runs at most this time right, at most the cardinality of the adjacency list of a particular vertex. Sum of all the adjacency list is no more than the number of adjust as we saw. So, the total running time of depth first search is also V plus E. So, depth first search also runs in time linear in the size of the graph. So, pretty much it is the same as breadth first search they have no difference except in terms of the convenience of what you want to use. I would ideally say that if you are not sure about what is the kind of graph it is you go for breadth first search because it is a safer way to explore. If we go for depth for search you might entire a loop in a graph that corresponds to the control flow graph and the loop could be infinite and you may not be able to come out of the loop.

So, breadth first search is slightly better to explore a graph when you are not sure about the kind of graph that you are looking at. So, towards showing correctness of depth first search again I will not be able to spend time proving the correctness of this algorithm.

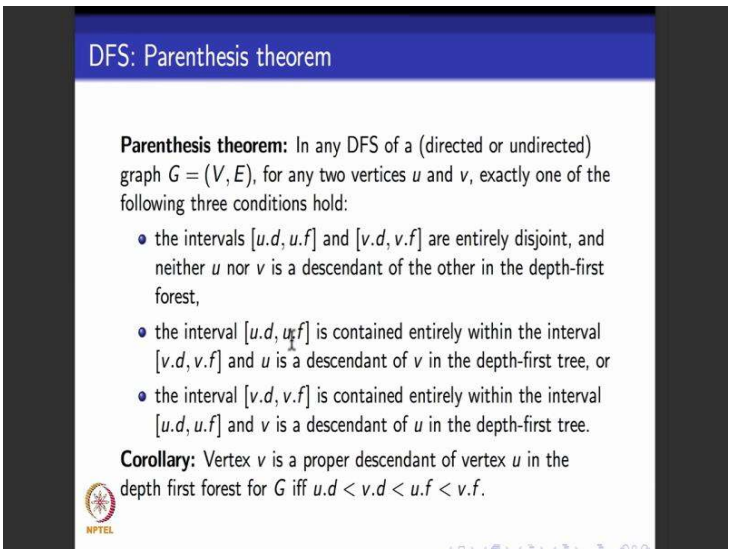
What I will do is walk you through the results that finally, show that depth first search is correct.

(Refer Slide Time: 16:02)



So, what is the main correctness result that I want to show about depth first search, that is this. So, when I take a graph and run depth first search on the graph then, depth first search explores the graph and it produces a forest of depth first trees containing tree edges or back edges.

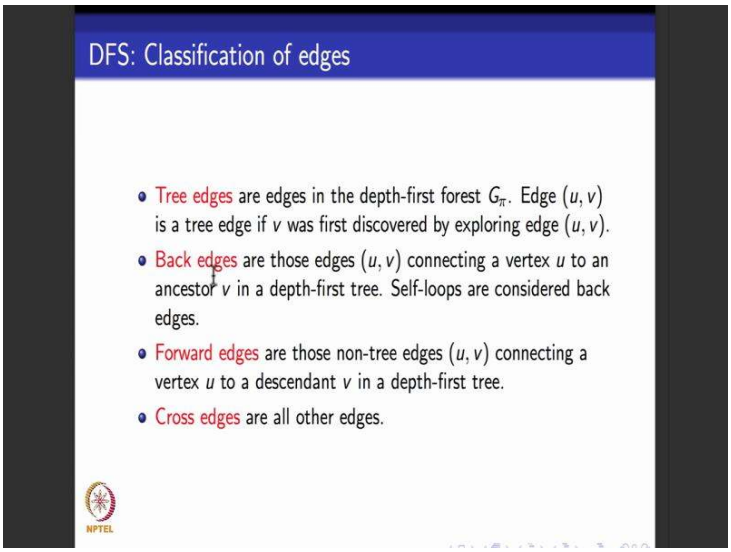
(Refer Slide Time: 16:22)



So, towards that I have all these theorems. So, this parenthesis theorem basically says that the discovery and finish times of all the vertices are in proper intervals. If you see this example right I continuously increase, I first discovered this time is one I next discover this timestamp is 2 then I next discovered this time stamp is 3, and when I finish your timestamp is 5, right. And if you see if I go back and finish the vertex u, its timestamp is 8 which is greater than the timestamp 5.

So, this lemma says that v is a proper descendant of a vertex u in the depth first forest then the timestamp that is assigned, the discovery, u assigned discovery timestamp first and then it is successor descendant v is discovered - u is finished and then v is finished right.

(Refer Slide Time: 17:17)

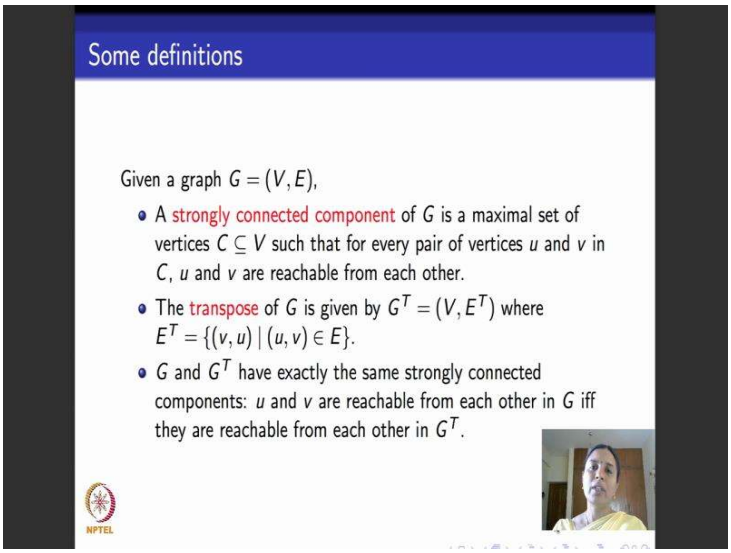


So, there are 4 kinds of edges that depth first search returns. One is what is called tree edges, the edges that belong to one of the depth first trees. The next kind is what is called back, edges that connect back, that connect a descendant back to it is ancestor what are called back edges. Forward edges are edges that follow the same direction as that of tree edges, but they sort of cut across several descendants and directly connect an ancestor to a descendant.

All other kinds of edges what are called cross edges. If you go back and take this example this is the final output, right, look at the last graph that my curser is in the final depth first tree is, looking at it here. So, that edges that colored grey what are called tree

edges. This edge from u to x marked f is a forward edge because it connects u to one of it is descendants x. This edge from x to v marked v is what is called a back edge because it connects a descendant back to one of it is ancestors. Similarly, this self-loop is also a back edge and this kind of an edge which is not a forward edge, not a tree edge, not a back edge is what is called a cross edge right. There should be a first categorizes edges into 4 parts.

(Refer Slide Time: 18:51)



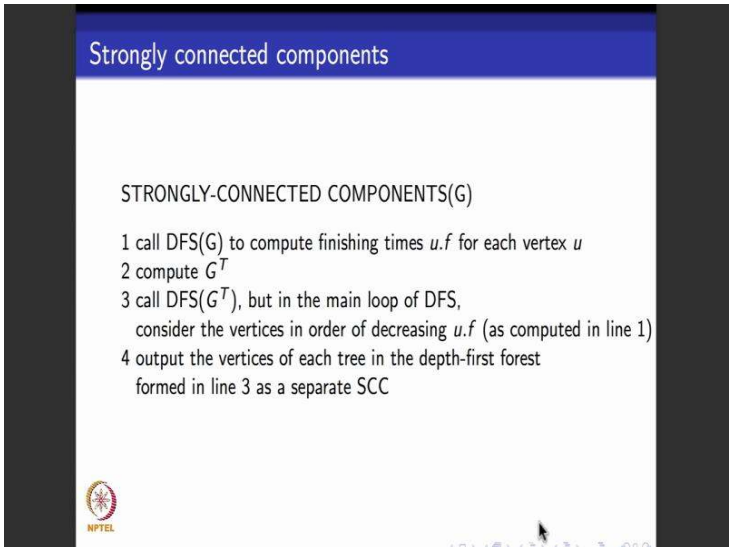
Now, the other thing that I wanted to do is to tell you how to use DFS to output what are called strongly connected components in a graph.

So, you need a directed graph to have strongly connected components and if you have an undirected graph you output what are called connected components. So, what is strongly connected component? A strongly connected component is a sort of a cycle in a directed graph. So, it says a subset of vertices is a strongly connected component if for every pair of vertices u, v in that component u is reachable from v and v is reachable from u. So, if you go back and look at this graph that we had in the slide here, if you see this graph, this is a strongly connected component, v, y, x. They, all 3 of them are reachable from each other through this cycle that my and tracing out now, through the cursor. Similarly, just this is z is another strongly connected component, single vertex strongly connected component. If you see w cannot belong to this strongly connected component because w can be y can be reached from w, but not vice versa.

So, this graph has to strongly connected components one that has this triangle and one that has this self-loop. So, I want to be able to know how to use DFS to be able to output the strongly connected components in the graph because I will use them to be able to look at prime paths in other entities for test case generation. So, to do that I look at the graph and I also look at its transpose. What is a transpose of a graph? You take the same graph and you reverse the directions of the edges assuming that it is a directed graph? So, a transpose of this graph in this example would be the same graph in terms of the vertices, but every edge will be presented with it is direction reversed. So, I look at the transpose at the graph. One thing to note is that G, the graph and its transpose have the same strongly connected component right. Because if one vertex was reachable from the other in the original graph then the same property would hold in the reversed graph. So, I just go in the reverse way.

They were reachable from each other in both directions right. So, to be able to output the strongly connected components, a standard technique to do is to be able to run DFS.

(Refer Slide Time: 21:02)

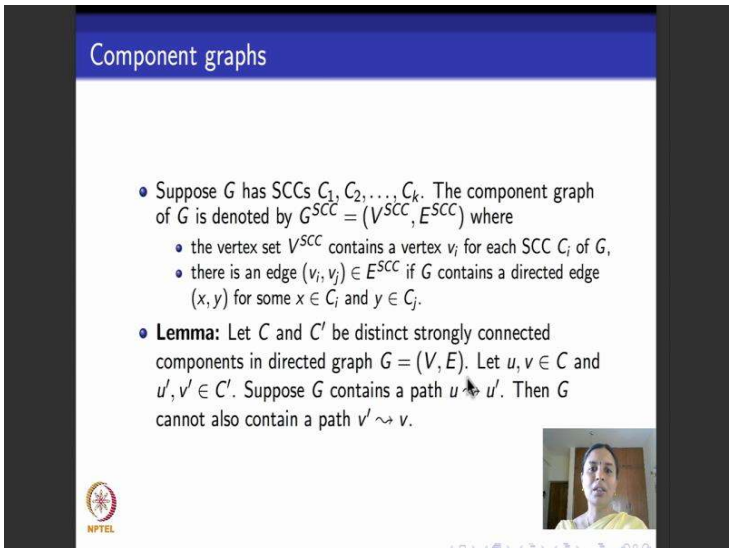


once on the graph take it is transpose and run DFS once on the transpose, but in the reverse direction of the finish times right. So, that is what this pseudo code does. You first run DFS compute finish times for each vertex u right. So, that you give you a forest of DFS trees. Then you compute the transpose of the given graph. Now you run DFS on

the transpose, but in the main loop of the DFS algorithm you consider vertices in the order of decreasing finished time. So, what it is intuitively saying is if you go back to this example after running DFS on this graph I get something like this right. Now what I do is I take the same graph take it is transpose. So, I reverse the direction of every edge.

So, I get the sort of a reversed graph. So, I run DFS once again, but in the reverse direction from the highest finished time vertex. So, in some sense this tree would have traced this, if I take this strongly connected component, what I am trying to do is I am trying to traverse one half of the strongly connected component through one DFS and I am trying to traverse the other part of the strongly connected component by running DFS once again on the transpose. That is what this code is trying to do. So, how do you find strongly connected components? You run DFS on the given graph, record the finished times, take the transpose of the graph, run DFS again on the transpose graph, but in decreasing order of the finish times that you recorded in step number one. And then what you do is that you output each vertex, because you would have gone through it once this way once that way both the ways is the strongly connected components. So, each DFS tree in the DFS forest would be a separate strongly connected component that you can output right.

(Refer Slide Time: 22:51)

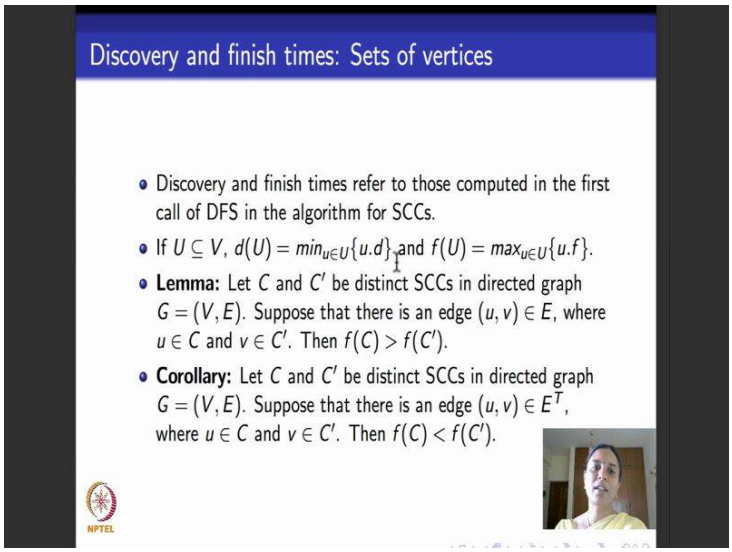


So, these are lemmas that tell you that the algorithm for running DFS is basically correct. So, what they tell you is that you take the given graph, take all its strongly connected

components. Lets say it has k is strongly connected components. You collapse each strongly connected component to create a meta vertex right. So, in this example if I see, I told you right, this is one strongly connected component, this is one strongly connected component, these 2 standalone separate strongly connected components, single vertex they want have any significant, but they are like that. So, what I do is I take this and I collapse and create one vertex with this, one vertex for this entire strongly connected component, one vertex for this and one vertex for this. So, I have 4 vertices. These edges get absorbed, these edges that connect these meta connected strongly connected components get retained the component graph.

So, that is how I create the component graph. So, I say vertex is are the set of vertices for each strongly connected components there is one meta vertex. And I say if that is an edge that connects one strongly connected component to the other, then you put an edge in the new graph right. This one says that if there are 2 distinct strongly connected components in the given graph, and I take a vertex path from one strongly connected component in the meta graph to the other strongly connected component in meta graph then, they cannot be a path in the reverse direction. Basically what it says is this component graph corresponding to a given graph where I collapse this strongly connected components into a single vertex is an acyclic graph.

(Refer Slide Time: 24:37)



So, using the fact that it is a cyclic graph now what I relate is I relate the discovery and finish times right. So, what I do is in this particular lemma one thing to be noted is that when I talk about discovery and finish times, I discuss I talk about the times that were recorded by the first DFS that runs in this algorithm, where the first DFS not by the second right. So, for an entire component I says discovery time is the least of discovery times of all the vertices. For an entire component, finish time is the highest of the finish times of all the vertices. So, what I do is suppose I have distinct strongly connected component C and C prime, then if there is an edge in the direction u to v where u is an C and v is the C prime then, I say that C would have been finished C prime would have been finished before C right.

So, corollary is the reverse. So, it says that if c and c prime are distinct strongly connected components in a given graph, and suppose there is an edge from u to v in the transpose graph then the reverse of that wholes right. So, what it basically says is that if I run DFS once here and if I run DFS once in the transpose, but taking it in the decreasing order of finished times, then I will be able to distinctly identify strongly connected components individually in this graph. Why does that hold true that holds true? Because if I compose the meta graph considered the meta graph where I compose and collapse each strongly connected component into one vertex then an meta graph is acyclic, so if I run DFS on that acyclic graph I would have correctly done both the DFS right. So, the algorithms were strongly connected components that we saw here is basically correct.

So, to summarize what I wanted to recap through these 2 modules was to teach you basic graph there was an algorithms depth first search breadth first search and they have several different applications strongly connected components is one application. Similarly, you can do topological sort elementary graph such algorithms where you keep extra parameters extra tax can all be done using basic manipulations of breadth first search and depth forts search, and both these algorithms have linear running time. What we will do in the next module is to see how to use this algorithm to be able to define algorithms for test requirements and test path generation to satisfy the test requirements.

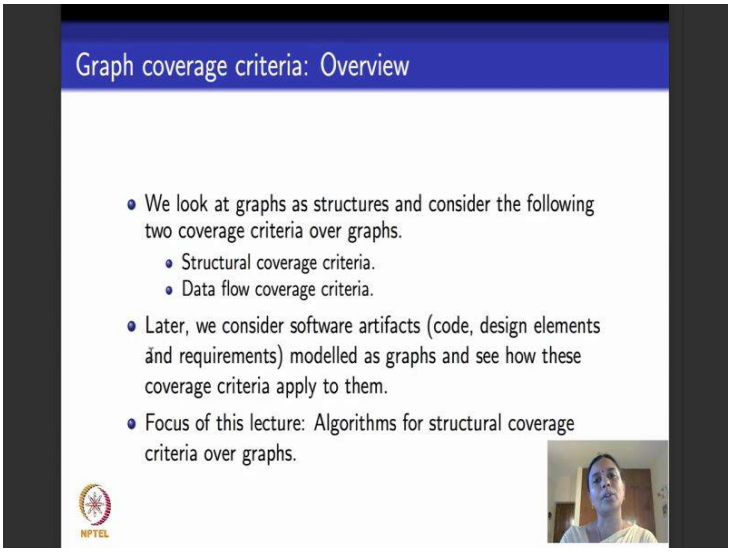
Thank you.

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 backs 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)

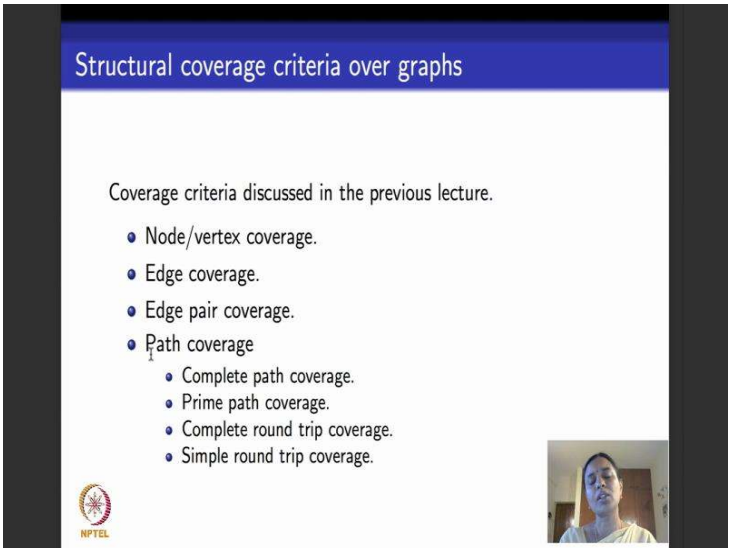


So, just to recap the overall module that we are looking at currently. We are now looking at designing test cases were software artifacts some modeled as graphs. So, in the graph models we consider two kinds of coverage criteria: coverage criteria based on the structures of the graph which we saw two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



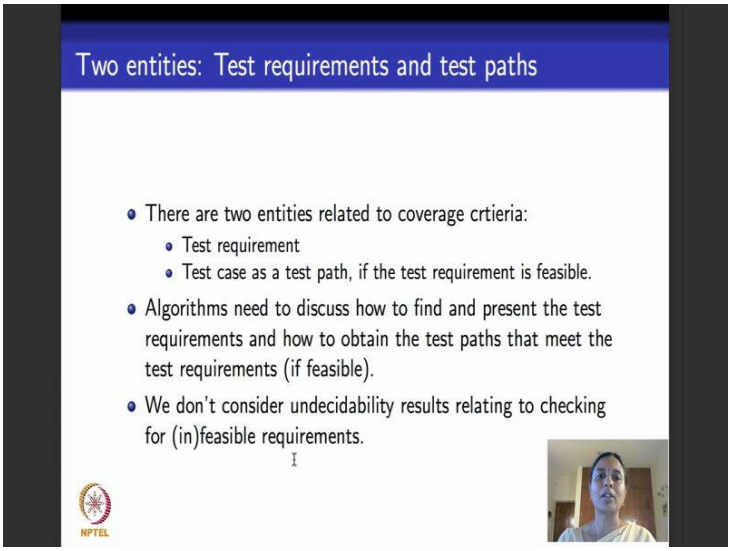
So, to recap what was the structural coverage criteria over graphs that we saw about two lectures ago. If you remember we saw all these coverage criteria we began with node or vertex coverage, then we moved on to edge coverage, we looked at edge-pair coverage, and then we looked at path coverage in graphs. 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, there is 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 where user gives 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 our goal 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)



So, we 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 look at the graph and generate a set of test paths in the graph that will achieve this test requirement for edge-pair coverage.

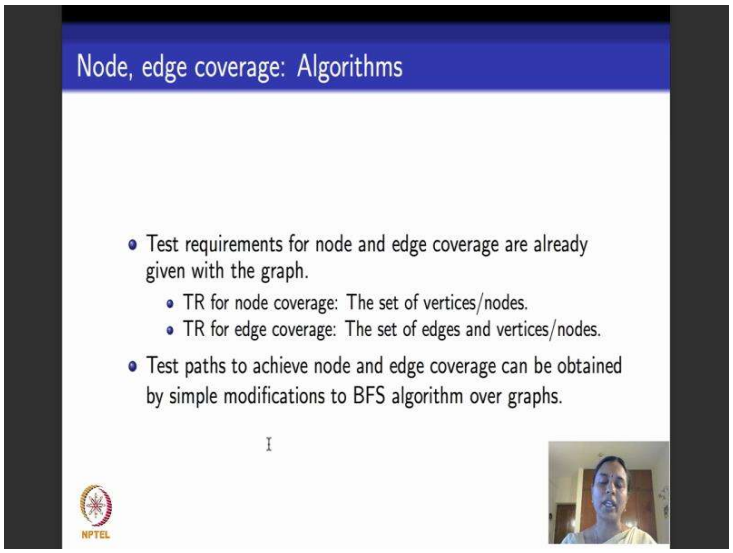
So, when I talk about coverage criteria, I am talking about two entities that are related to the coverage criteria: one is the criteria specified as a test requirement, abbreviated as TR, and the other is the set of test paths which occur a test case that are designed to satisfy the criteria. Now it 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 proofs 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 write test requirements for them, and how to design test cases as test paths for them.

(Refer Slide Time: 05:39)



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 around this 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? It 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 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 source 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.

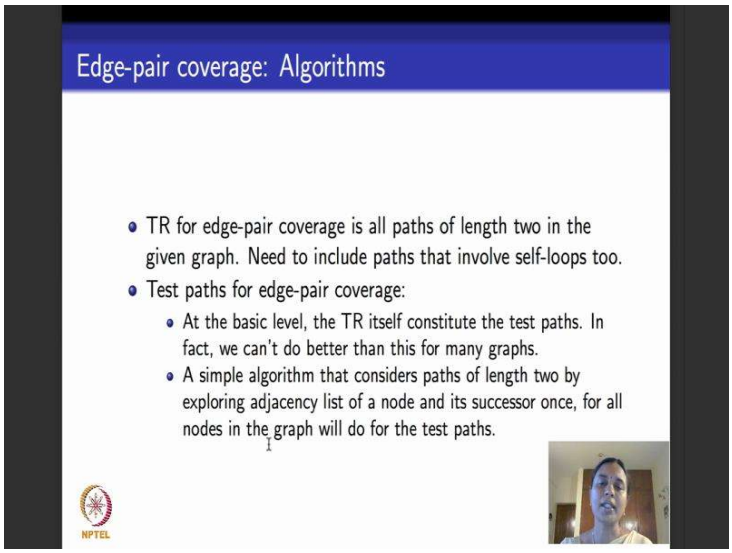
Suppose this tree spans out all the graphs; there it is, the paths of the tree will give you the test path is test cases for node coverage. Suppose such the first breadth first search of the first DFS does not finish exploring all the vertices of the graph; there are vertices that were not reachable 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 of 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 check 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 act 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 ideally a test path 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.

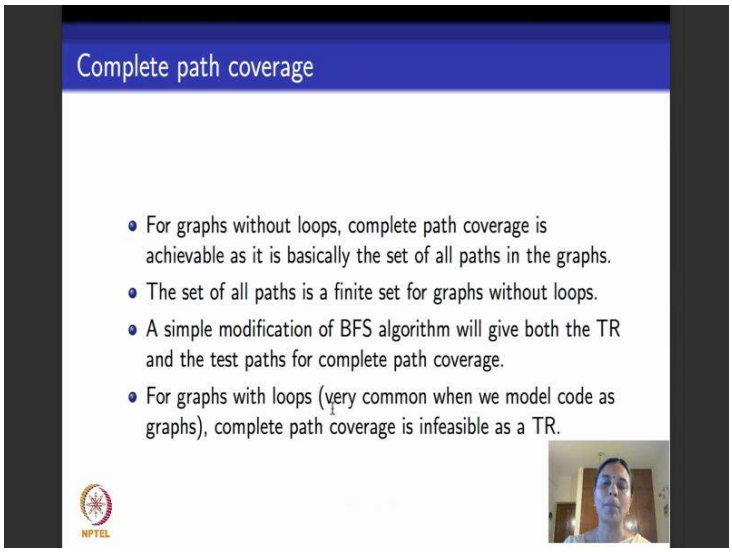
(Refer Slide Time: 08:46)



We move on: now the next coverage criteria in our list 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 edges 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.

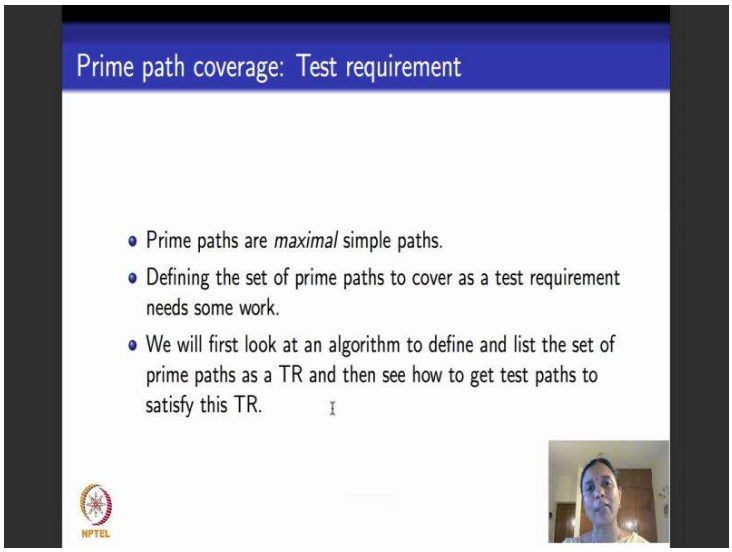
Once I have the TR for edge-pair coverage, the test requirement itself could act as a test path. We saw an example of a graph there in fact we cannot do better than that, the TR itself becomes a test path. Otherwise what I do is, I can again run BFS or DFS and get as set of test path that will satisfy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

(Refer Slide Time: 09:58)



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, so we directly move on and look at prime path coverage.

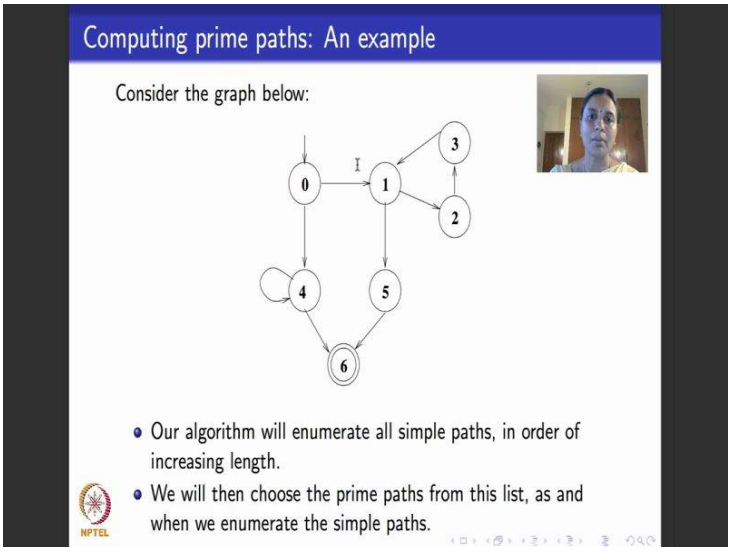
(Refer Slide Time: 10:22)



So, just to recap what is prime path? Prime paths 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 walk you through how the algorithm will run to compute the list of prime paths. 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 requirement, 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)



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 node 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 paths. As I told you what is a single big use of prime paths? Assuming that this cycle represents a loop prime paths 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 operations.

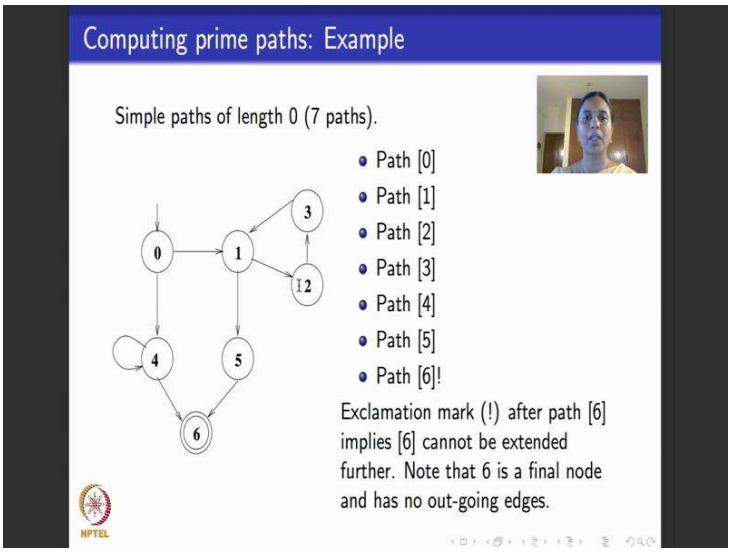
So, you can here you can assume there are two loops: one self loop here and one cycle here. So, we will see how to compute prime paths for this graph by, as a test requirement first and then we will see test cases that will help us to achieve this test requirement. So, what are prime paths? Primes paths 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 is 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 of 0, then will enumerate simple path of length 1, we 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 as being 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 marked 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)

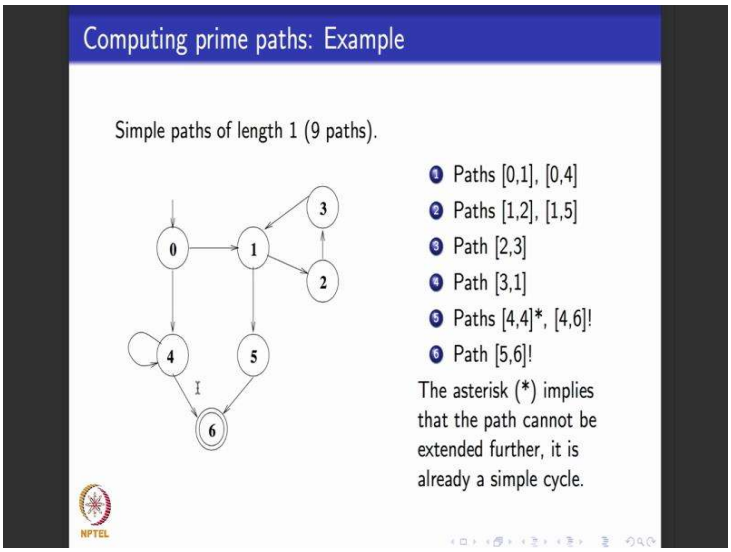


So, what is a simple path of length 0? Simple path of length 0 is just the vertex, because it has one single vertex. How many vertices are there in this graph? There are seven vertices starting from 0, 1, 2, 3 and so on up to 6.

So, I enumerate each of these vertices as a path of length 0. There are seven such paths of length 0. I have just enumerated them. Another thing 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, I go ahead and enumerate paths of length 1.

(Refer Slide Time: 15:34)



So, here is the same graph and here are the paths of length 1. How have I 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).

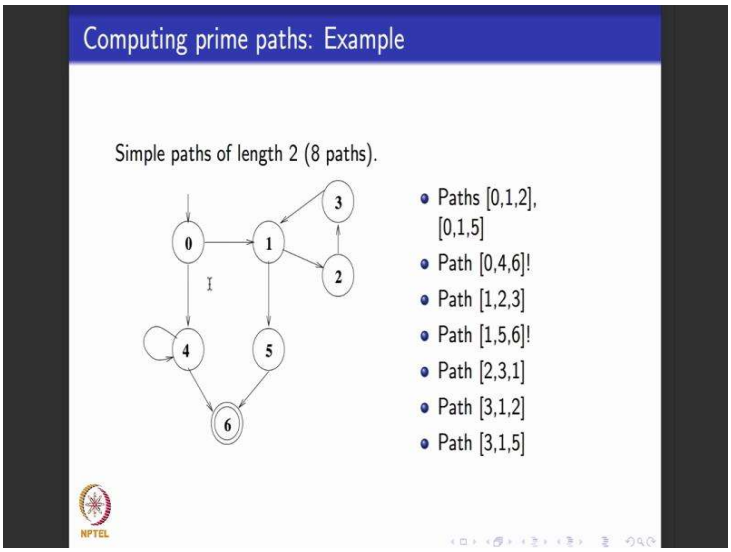
Similarly, vertex one which was a simple path of length 0 can be extended to two paths of length 1 paths 1, 2 and path 1, 5. If we move on vertex 2 can be extended to path 2, 3; vertex 3 can be extended to path 3, 1; vertex 4 can be extended to two simple paths of length 1, one leading to 6 and 1 using 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 slide. 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 nodes6 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? Paths 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 or prime paths is what I want to look at.

So, in addition to exclamation mark I have also gone ahead and put an asterisk here. 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 asterisk which says that this path also cannot be extended further.

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

(Refer Slide Time: 18:27)

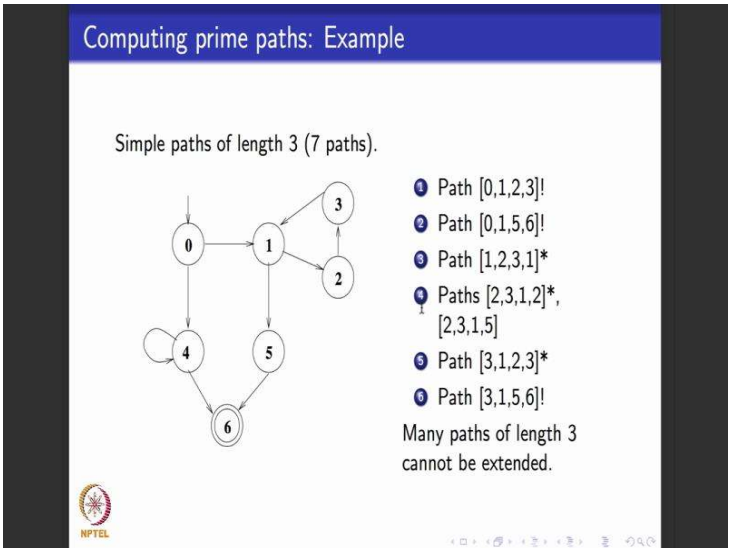


So, you look at the paths 0, 1; I extend the path 0, 1 into two possible paths of length 2 path going from 0 to 1 and then from 1 to 2 and path going from 0 to 1 and then from 1 to 5; that is what is listed here.

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? Eight path are there and two of them cannot be extended further.

(Refer Slide Time: 19:24)

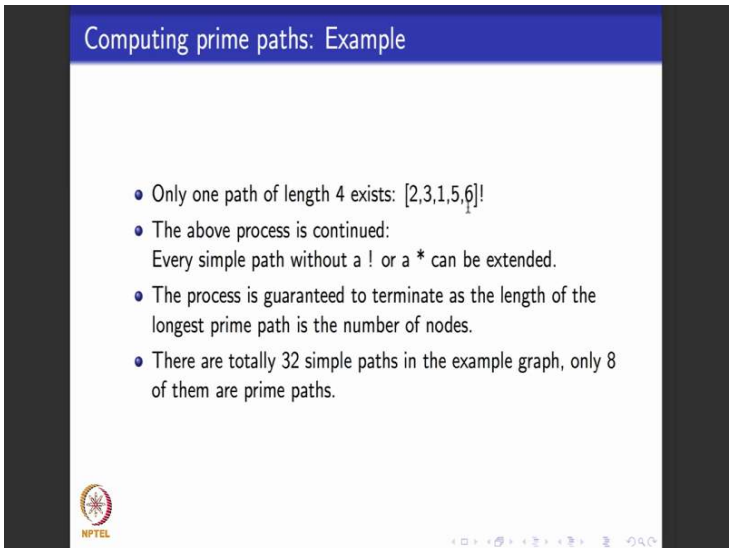


Now, I start with paths of length 2 and consider path of length 3. It so happens that I get seven different paths. So, path 0, 1, 2 can be extended to 0, 1, 2, 3. Path 0, 1, 5 can be extended to 0, 1, 5, 6 right, path 1, 2, 3 can be extended to 1, 2, 3, 1. I can go back and so on.

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 marked paths mean? They mean that that is it, we cannot extend them anymore. We cannot extend them anymore for two reasons: one is they 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.

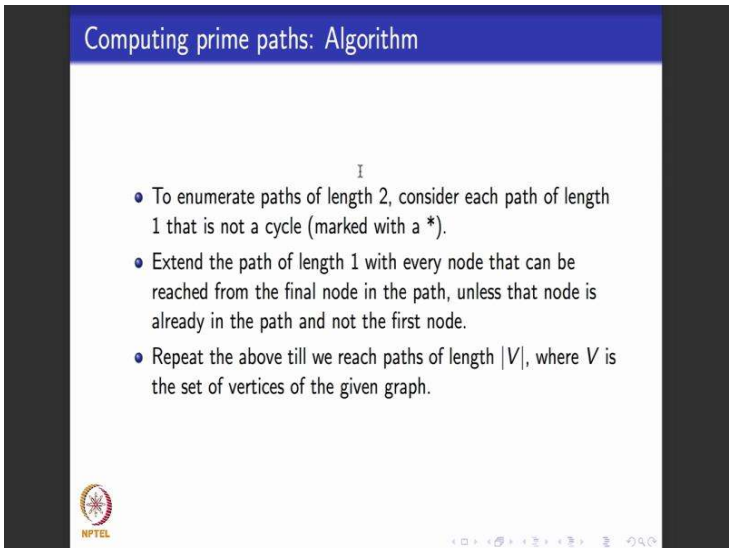
So, I am pretty much closed here as for as paths of length 3 are concerned. There is only one path here 2, 3, 1, 5 that is not marked, so that is the only path that has go for extension. So, I go ahead take that path and extend it by adding the edge (5, 6). You see 2, 3, 1, 5 that is a path of length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

(Refer Slide Time: 20:36)



That is what I have done here. And the moment I did that I am forced 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)



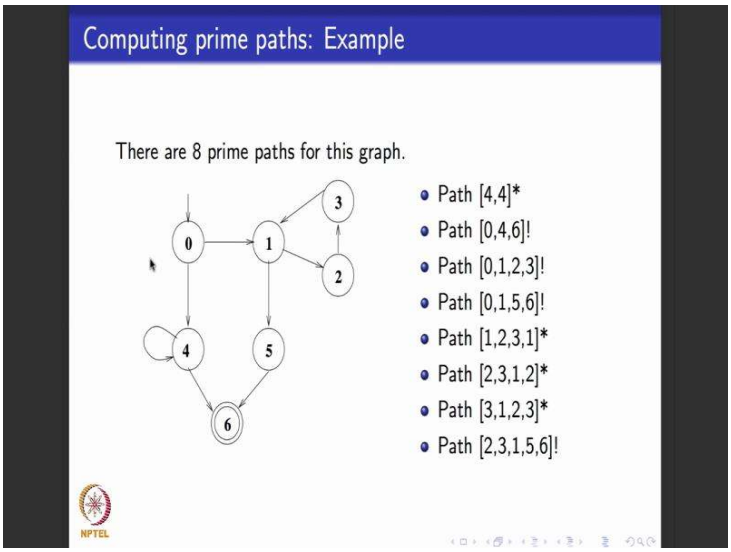
So, what do I do? Here is the algorithm in English. I consider paths of length 2 by extending paths of length 1 with every node that can be reach from the final node in the path unless that node is already in the path and is not the first node. And I keep doing this for paths of length n, increase to 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, by pigeon hole 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 markings 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. I 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 at 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 only eight of them are prime paths. And those are the ones that have been obtained by these markings. In general it is not an easy problem to count the number of simple path, I have just given this for the specific graph. And why is this algorithm guaranteed to terminate, because it is to stop when I get a path of length mod V as we discussed.

(Refer Slide Time: 23:07)



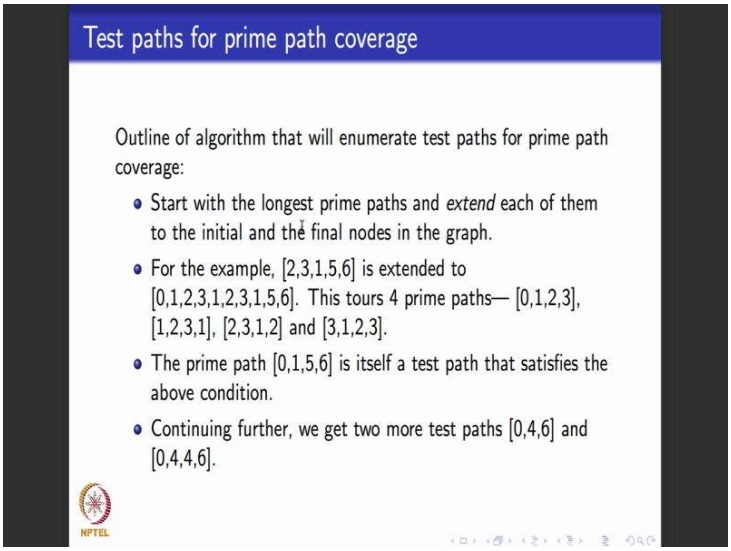
So, now I will just present this graph and all the prime paths for the graph. So, this is the graph that we have been using to understand an algorithm, and here are all the prime paths that I worked my algorithm out and I will listed through that. So, let us look at them :the path 4, 4 is this which was listed as a path of length 1 if you remember and right then marked 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 traverse this loop. 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 exiting 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, ends 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 test paths to meet the test requirement for prime path coverage.

(Refer Slide Time: 25:28)



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 I will 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 known 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.

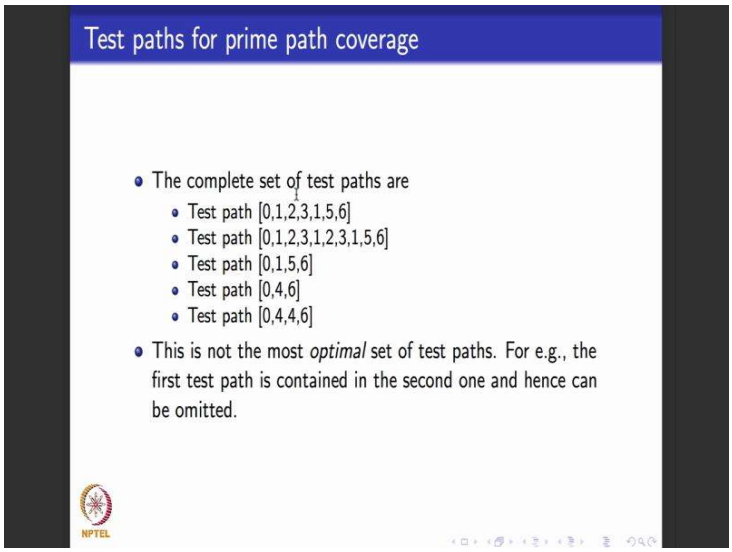
In this case 6 is already a final node, so I have already ended in the final node, but 2 is not an initial node. So, I take this longest prime path and see if I can go to the left and extend this path to make it as if it was beginning from an initial node. So, if I do that for 2, 3, 1, 5, 6, I get such 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

repeated 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 3; 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 if just an ordinary test path. The only condition that it has to satisfy is that it has to begin it an initial node which is 0 for our example and ended it a final node which is 6 for a 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 toured 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 node and end in the final node, but I need to be able to give scope to cover this loop so I extended it to 0, 4, 4, 6.

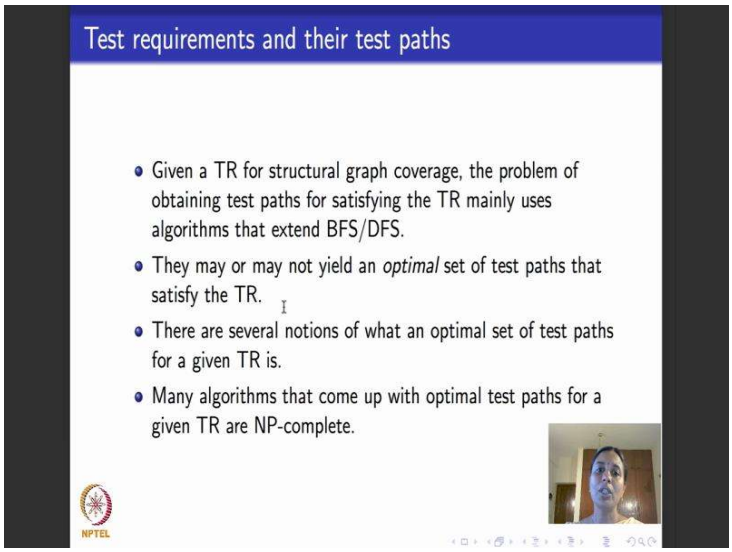
(Refer Slide Time: 29:13)



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 skiped 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 have not got the least number of test path that I could cover.

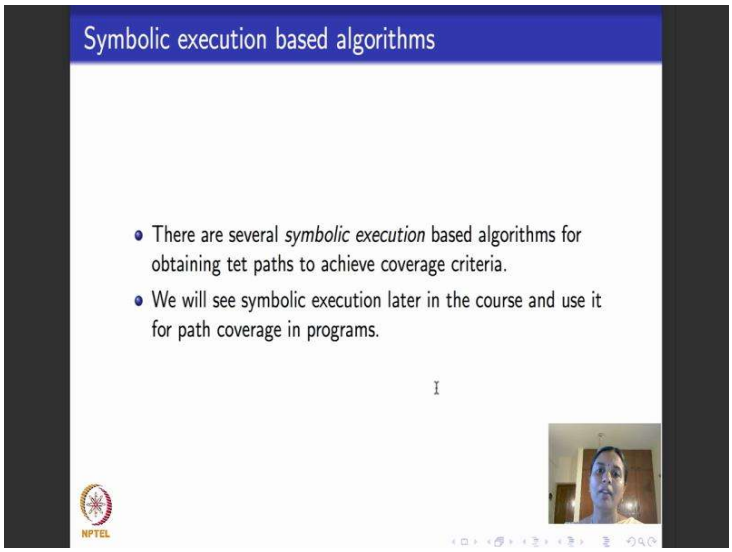
Similarly 0, 4, 6 is completely embedded in 0, 4, 4, 6, so I could remove 0, 4, 6, but that is alright. The focus for us is not to get the most optimal set of test paths, the focus for us is to be able to correctly define the test requirement and get a set of test paths that satisfy the test requirements;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

(Refer Slide Time: 30:22)



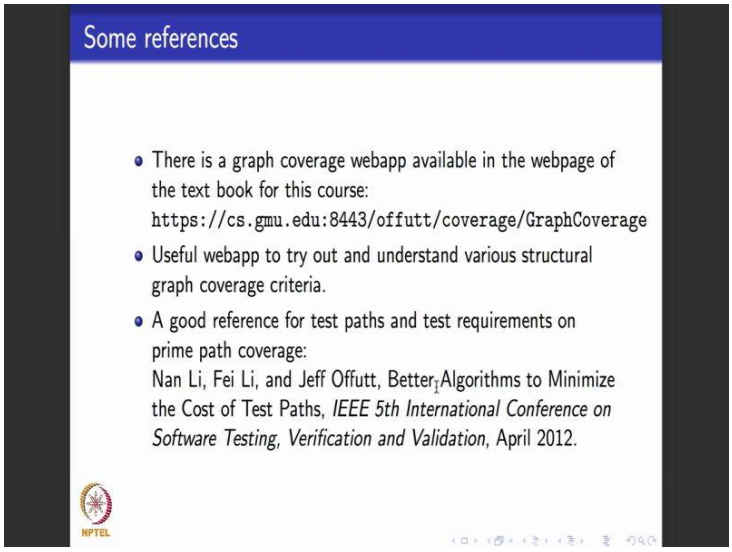
In general the problem of what is optimality, optimal test path corresponding to a test requirement. There are several notions of optimality available in the literature. We will not really focus on that because I want to be able to move on, but I point you to good references were you could look up for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

(Refer Slide Time: 30:53)



In fact, what we saw was explicit state algorithms- algorithms that use breath first search or depth first 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)



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 Ammann and Offutt called 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 which 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.

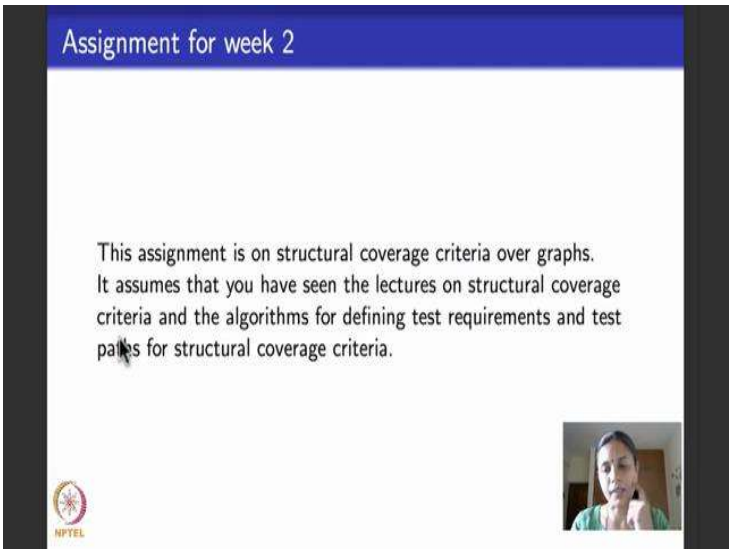
Thank you.

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

Lecture – 10
Assignment 2: Structural Coverage Criteria

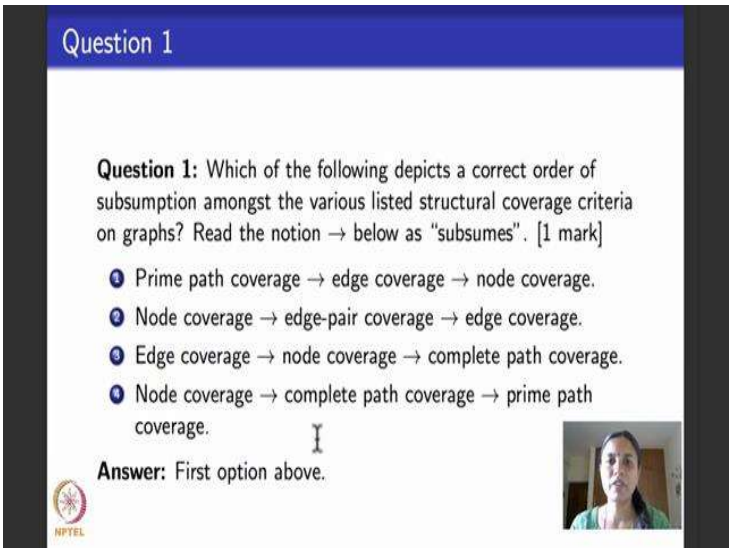
Hello everyone. So, this is the first lecture of third week. What I wanted to do today that is not really a lecture, but more to help you walk through the second week's assignment, assuming that you have looked at it and you made an attempt to solve it and you have also uploaded solutions. We will see how we will go about solving it, what are the questions, let us discuss our understanding of structure and coverage criteria over graphs to see if we could solve this assignment fully, correctly.

(Refer Slide Time: 00:43)



This lecture assumes that you have gone through all the videos of the graph structural coverage criteria, the algorithms for graph structural coverage criteria, and you have also looked at the assignment for the second week and made an attempt to solve it. So, what I will do now is I will walk you through one question after the other, tell you what the a correct answer is and tell you how to get the correct answer.

(Refer Slide Time: 01:06)

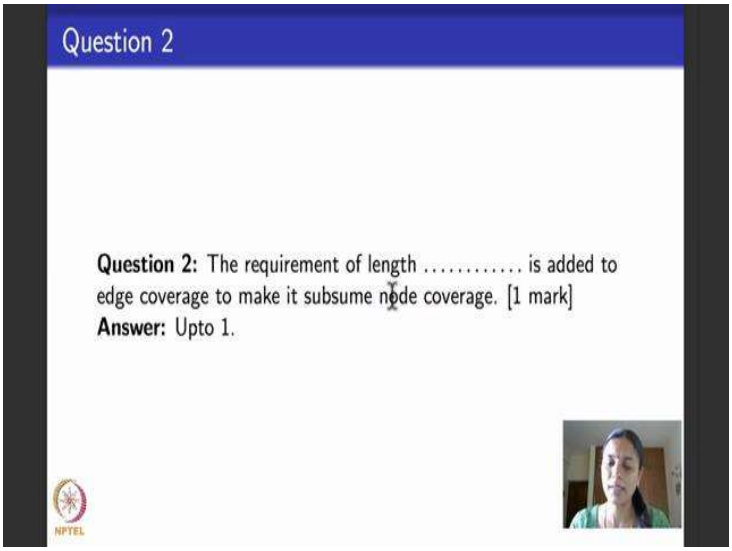


So, which was the first question? The assignment had 8 questions, the first question talked about coverage criterion subsumption; to read out the question it asks which of the following depicts a correct order of subsumption amongst the various listed coverage criteria below. So, this notion, this symbol right arrow you should read it as subsumes. So, how do I read the first option? Your answer is to choose one of these options, the correct option. How do I read the first option? You read it as prime path coverage subsumes edge coverage which in turn subsumes node coverage. Similarly second one would be node coverage subsumes edge pair coverage which in turn subsumes edge coverage and so on? So, the correct choice for this answer for this question happens to be the first answer, that is this one. Prime path coverage that subsume edge coverage which in turn subsumes node coverage.

Now, why are the rest of them not the correct option? If you see the second option it says node coverage subsumes edge pair coverage which in terms subsumes edge coverage. So, it as to subsumption relation like all other options this the second one edge pairs coverage subsuming edge coverage is correct, but node coverage does not subsume edge pair coverage. So, this is not the correct option similarly for the third one, edge coverage does subsume node coverage, but node coverage clearly does not subsume complete path coverage. In fact, complete path coverage could even be infeasible as we discussed several times.

So, third one also cannot be the right option. What is the fourth one? It says node coverage subsumes complete path coverage which in terms of subsumes prime path coverage. So, this part is correct, complete path coverage that subsumes guide path coverage because if I do complete paths I always include prime paths in my TR, but this is not correct as I told you in the option for the previous answer node coverage does not subsume complete path coverage. So, except for 1 the other options 2, 3 and 4, one of the listed subsumption is not correct. So, none of them can be a correct answer. In the first one both the listed subsumption criteria happened to be correct. So, the first one is the correct answer.

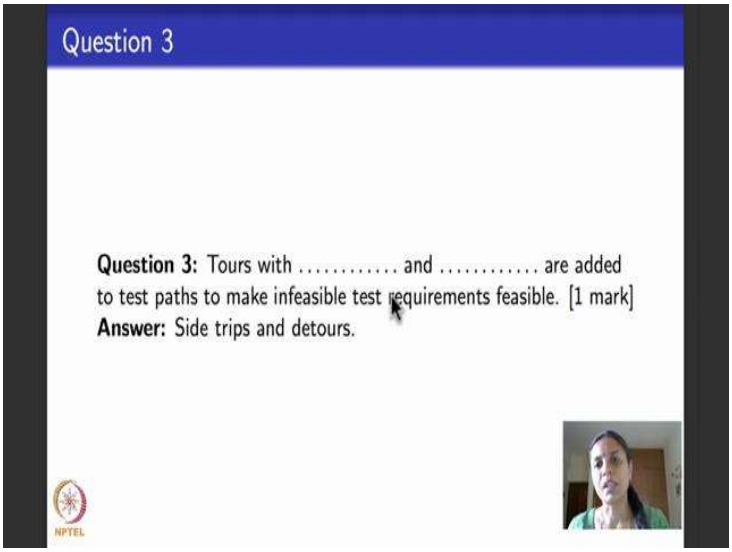
(Refer Slide Time: 03:28)



Now, moving on to the second question. Second question was a simple fill in the blank question, it asks the following it says the requirement of length dash should be added to edge coverage to make it subsume node coverage. The answer is requirement of length up to 1, if you remember what edge coverage is, the test requirement or t r for edge coverage said you cover all the edges of the graph. So, all the edges of the graph means all paths of length exactly 1. So, I do not keep it as exactly one I make it as a length up to 1 because I want edge coverage to subsume node coverage. Node coverage TR says nodes the path of length 0. So, if I include the requirement length up to 1 it includes paths of length 0 and it also includes paths of length what. So, because I want edge coverage to subsume node coverage, I change the requirement for edge coverage to be all

parts of length up to 1 as it is test requirement. So, the answer for this would be all paths of length up to 1.

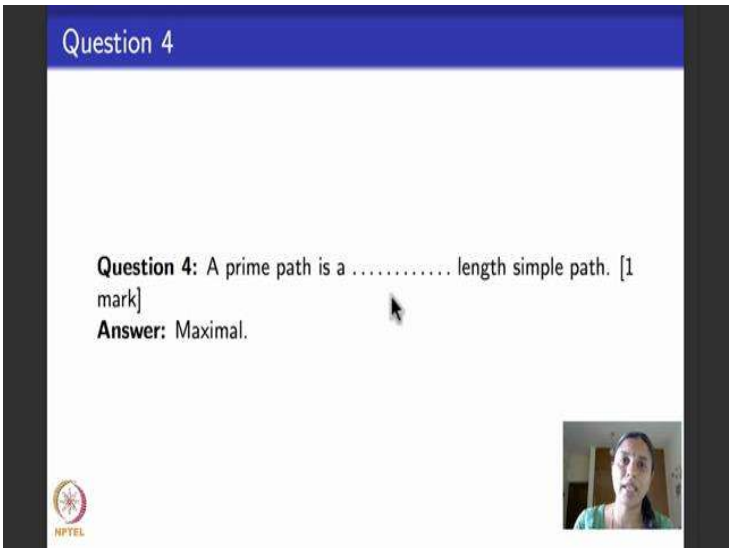
(Refer Slide Time: 04:31)



The next question asks, again a fill in the blank question, it says tours with the dash and dash are added to test parts to make infeasible requirements feasible. If you remember the lectures we have discussed about side trips and detours. Side trip, basically you take the test paths and then at some point in a vertices test path you decide to move out take another small side trip come back and join the vertex right. What would be a detour? Detour is very similar to a side trip, but it might skip some edges also on the side trip. Why would I need this? If you remember test path especially those dealing with prime path coverage could sometimes become infeasible, the test requirements will become infeasible.

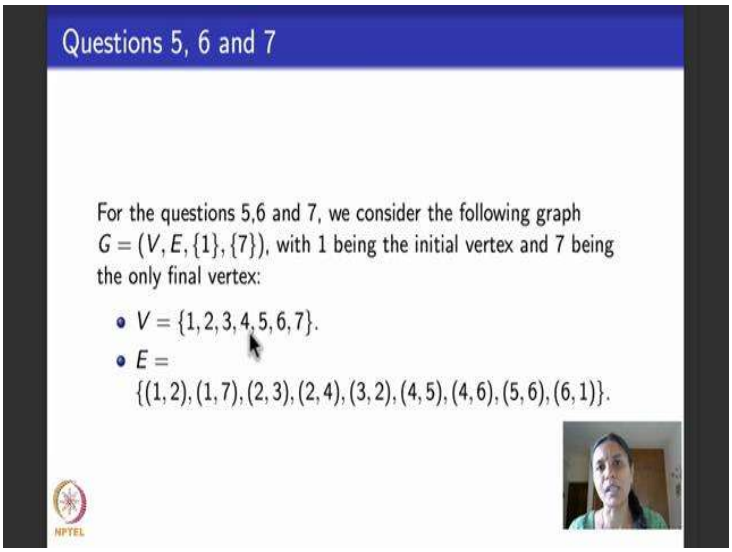
Especially if I have a loop like do while loop and I look for prime path coverage on that do while loop and may not be able to skip the loop at all right because do while says you first do one execution of the loop and then you check for the condition. Whereas prime path coverage will insist on skipping the loop. So, I say I meet that test requirement for prime path coverage by using a test path that will skip the loop, but actually execute the loop as a side trip right. So, they are very useful to make infeasible test paths as feasible.

(Refer Slide Time: 05:56)



So, the next question was again fill in the blank question it asks what sort of path a prime path is. The prime path, if you remember the definition is a simple path there does not come as a sub path of any other path. So, prime path are what are called maximal simple path, maximal length simple path. Why do I put maximal and not maximum? I hope you know the difference maximal says that they could be more than one maximum length simple path right. There could be you see the examples if you remember that we looked at while dealing with algorithms for structuring coverage criteria, there is prime path of length 2 there were prime paths of length 3 and there were prime paths of length 4. For each of these lengths that happens to be the maximum length simple path. So, that is why we say maximal. So, the answer to this in a prime path is a maximal length simple path.

(Refer Slide Time: 06:48)

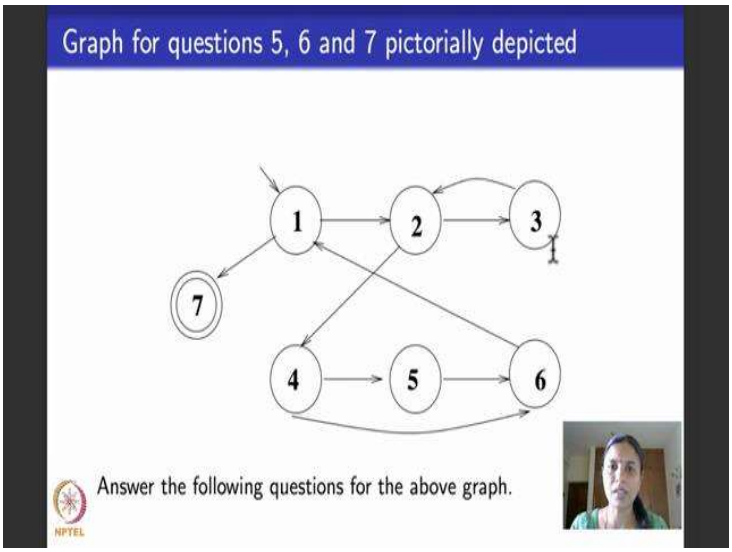


The last 3 questions in the assignment where to deal with coverage criteria that is based on an example graph that was given as a part of the assignment to you. So, here is a graph. So, the graph had 7 vertexes. Numbered 1 to 7, the vertex 1 was the initial vertex or the start vertex the vertex 7 was the only final vertex and these were all the edges in the graph. And then you had 3 questions 5, 6 and 7 which dealt with answering questions about coverage criteria on this graph.

So, before we go ahead and look at how to answer the questions 5, 6 and 7, whenever I am asked a question like this the first wise thing to do is to be able to draw the graphs to be able to visualize the graph for yourself. What are the benefit is of drawing the graph, one is the notion of how the edges are placed what the edges are and how to look for paths in the graph becomes visually and it becomes a little more clear right? Then instead of staring at the set repeatedly to figure out what the notion of path is and things like how to achieve node coverage how to achieve edge coverage even to the extent of how to achieve prime paths coverage can be looked at the visual graph and answered very easily. Intuitively, you do not even have to use the algorithms that we discussed to be able to come up with test requirements and test paths for these criteria.

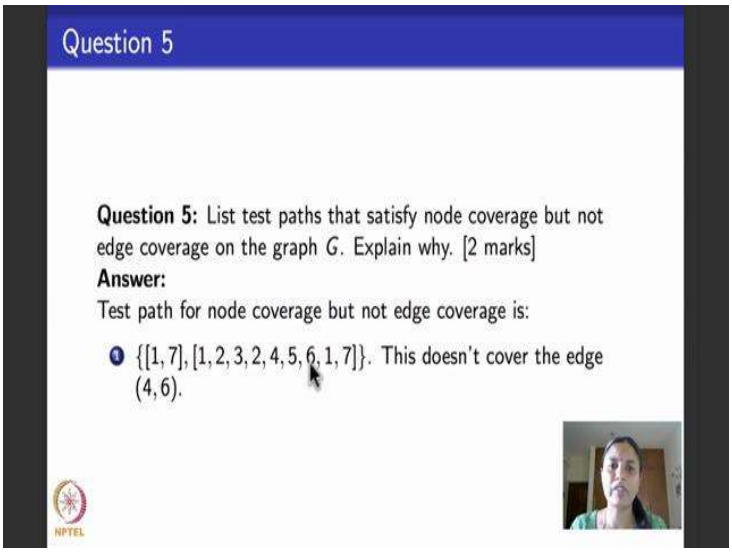
So, that is what I have done for you. Now I have drawn this graph. So, this graph has 7 vertices 1, 2, 3 and so on up to 7.

(Refer Slide Time: 08:17)



So, I have put this 7 vertices. Another thing to note is that while drawing the graph you may not get the layout of the graph. So, perfectly fine do not worry about that if you have to draw long edges, short edges, queued edges you still it is still helps to have a visual graph. And then a mark this initial vertex one mark the final vertex 7 using double circles, and then I have marked the edges if you see there were so many edges about 9 edges. So, there were 2 edges going out of 1, 1 to 2 and 1 to 7, I have marked those edges 1 to 2, 1 to 7, 2 edges is going out of 2, 2 to 3, 2 to 4, marked here 2 to 3, 2 to 4 only 1 edge going out 3 which is from 3 to 2, 3 to 2. Two edges going out of 4, one to 5, one to 6, 4 to 5, 4 to 6; one edge going out of 5 to 6 and 1 edge going from 6 to 1. So, here is one edge going from 5 to 6 one edge going from 6 to 1. So, I think we captured all the edges here. So, this is the graph that is the same as this, but drawn.

(Refer Slide Time: 09:31)



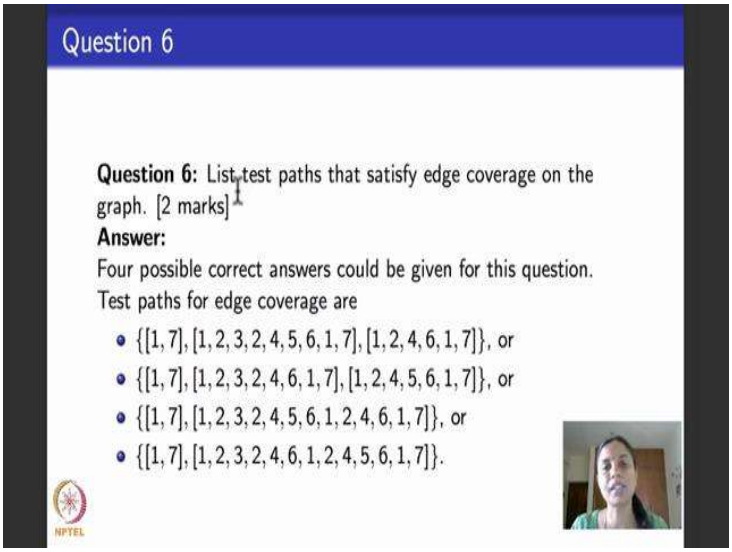
So, now let us go ahead and look at the questions that were asked about this graphs. So, question number 5 says to list all the test paths that satisfy node coverage, but it also says that some of list tests path in such a way that that they satisfy a node coverage, but do not meet edge coverage. If you remember node coverage does not subsume edge coverage in all cases. So, in this particular graph node coverage need not subsume edge coverage. So, it might be possible to come up with a set of test path that satisfy node coverage, but not edge coverage. So, what is node coverage test requirement for this graph? It is basically the set consisting of all the 7 vertices. What are the test paths for node coverage? If you see in this path from 1 to 7, is like a loner path right, it is on it is own. So, you have to be able to take this this is the only way to go to 7. So, to do node coverage for the vertex 7, I need to be able to include this as a standalone test path. It is a test path because it satisfies all the requirements of a test path it begins at an initial vertex which is one it ends in the final vertex which is 7.

So, I have to be able to include this test path to node cover 7. So, the remaining nodes are 2, 3, 4, 5 and 6. So, how do I cover them I can include any test path remember test paths always have to begin in in initial vertex and end in a final vertex. So, I begin at 1, I do not have much of a choice, here I go 2 from 2; let us say I go to 4, 5, 6; from 6 I go to 1. So, which are the nodes I have covered if I had taken such a path ? I have covered 1, 2, 4, 5, 6, earlier I had covered 1 and 7 and left on 3; right, but I have not still not left out. So, I can always go back and then I will continue this path then take it from 1 to 2, I visit 2

once again that is all right there is no harm in this because test paths can have cycles, and then I go to 3. So, I have covered now all the vertices, but I cannot end my test path here because the test path has to end in a final vertex. So, I take this vertex back, and now I still cannot go to 7 the only way to go to 7 is to go to 4 and then may be do this 4 to 6, 6 to 1 and 1 receive.

So, basically what I am trying to say is that if my test requirement is node coverage then there has to be a test path that consists of this segment in the graph which is just this edge 1 to 7. And for all the remaining vertices you need to go through this whole zigzag cycle once and come back to 7. It could include any test path that does that. Here, what I have done is I have included this test path I have this, 1, 2, 3, 2, 4, 5, 6, 1, 7. So, to trace it out 1, 2, 3, 2, 4, 5, 6, 1, 7, it does need note coverage if you see all the nodes that are included in the test path, but I had one more condition in my question that was do not have do not make it subsume edge coverage. So, I am chosing test path in such a way that I have left out one edge which is this edge I have left out the edge 4, 6. So, if you see this test path achieves node coverage, but it almost achieves edge coverage it visits all the edges except for this edge 4, 6. So, this test path is good enough an answer for this question.

(Refer Slide Time: 12:53)



So, the next questions is list all the test paths that can satisfy edge coverage criteria on the graph. So, here they could be several possible correct answers. In fact, 4 different

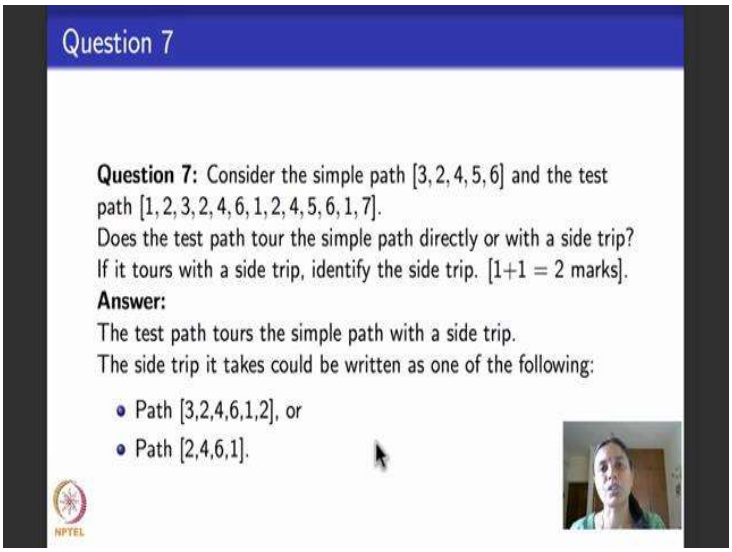
possible correct answers are there. Let us go back and look at the graph. As I told you right, this path is the standalone path it always has to be taken whether you do more coverage edge coverage prime paths coverage. So, 1, 7 will always be there. If you see in all the 4 options that I have listed here 1, 7 is always there andd then there is a question of this covering the rest to the edges. So, if you see when I execute the only thing to remember is that all these are pretty much serial, it is only at this place do I have a branching or a choice of course, 2 also I have a branching, but it is this this, but once I do that again I have another choice here at 4.

So, suppose I take this branch 4 to 5 and 5 to 6. Then I have left out this one this edge 4 to 6. So, I need to be able to come back how do I come back only way to come back is to go from 6 to 1, 1 to 2, 2 to 3, 3 back to, 2 to 4 and then I have to take 4 to 6. So, the 4 options basically exploit this. So, you have to do 1, 7 separately, and what are these 4 options? They let you do the let you traverse from 4 to 6 either using the 2 edges 4 to 5, 5 to 6 or using this single edge 4 to 6. So, it is 1, 2, 3, 2; use the edge 4, 5, 6, go back to 1 and then to 7, finish. And then you begin at 1 go to 2 use the edge 4 to 6, the direct edge do 1 and 7 again. This one is the same thing, but it says, so, I will just trace it out here you do 1, 2, 3, 4, 5, 6, 1, 7 for you do 1, 2, 3, 4, 6, 1, 7.

The second one what I have done is, I have done, 1, 2 4, 6, 1, 7 and then I have done, 1, 2, 3, 2, 4, 5, 6, 1, 7. So, those are the 2 options here. And the last 2 options I have basically joined them both. I have avoided the repetition of going back to 7 and beginning again at 1, I just go back to 1 and resume from 2 again. So, what I have done just to trace the path here is I have done, 1, 2, 4, 5, 6, 1, 2, 3, 2, 4, 6, 1, 7 that is this one long test path that is here.

The second one reverses this order of visiting 4, 5, 6 and 4, 6. It does 4, 6 first and then 4, 5, 6. So, four answers for edge coverage do 1, 7 separately and then the rest of these parts visit them in any order that you like, but you have to be able to do this large strongly connected component twice, once going through this path and once going through this path and there is a choice here. Similarly to once if you exercise this then the next time you have to be able to exercise this that is the only way I can cover all the edges in the graph. So, that is the test path for edge coverage.

(Refer Slide Time: 16:02)



The last question of this assignment asks you to go back to the graph and consider the simple path 3, 2, 4, 5, 6. So, let us go and see what that simple path looks like. It is this: 3, 2, 4, 5, 6. Please remember simple path is not a test path. So, it can begin an end at any node the only thing to note is that it is a simple path does not have any cycles.

So, suppose you had a specified path coverage requirement of having to write a test path that covers this simple path. So, they have made an attempt, given you some test path. So, the test path very long it has 1, 2, 3, 2, then that is the 4, 6 edge comes back to 1 and that is the 4, 5, 6 edge it comes back to 1 and then to 7. It is this long path the just walks through the entire graph once round and round. So, the question that was asked were two parts does this test path toward the simple path directly or with a side trip? The test path does not toward simply path directly right, if you see the simple path is 3 to 4, 5, 6 it is split into 2 parts a 3 and 2 come here, 4 and 5 and 6 come here.

So, it does not toward simple path directly. If it was the simple path directly the simple path will be a contiguous sub path of the test path. Because it is not there it does not do other simply path directly. So, the test path in fact, tours the simple path with a side trip. What is the side trip? Side trip is this bit that is left in between. So, 4, 6, 1, 2, so, you could write it like this 2, 4, 6, 1 or you could include 3 and these 2 and make it 3, 4, 6, 1, 2. So, this test path tours this is given simply path with the side trip and the side trip looks like this if you are written either of these it would be acceptable.

So, I hope this small exercise helped you to understand how to solve simple assignments that we would be giving every week. I will try to upload a couple of more videos as we move along for other assignments that will help you to solve these assignments.

(Refer Slide Time: 18:08)

For now, to get to know more about the graph structural coverage criteria I encourage you to look at the web app that comes as a part of this course book. It is a very nice app. It will help you to learn how to represent graphs as adjacency lists, and how to feed them and how to do various coverage criteria. Please do try it out hope it be useful to you.

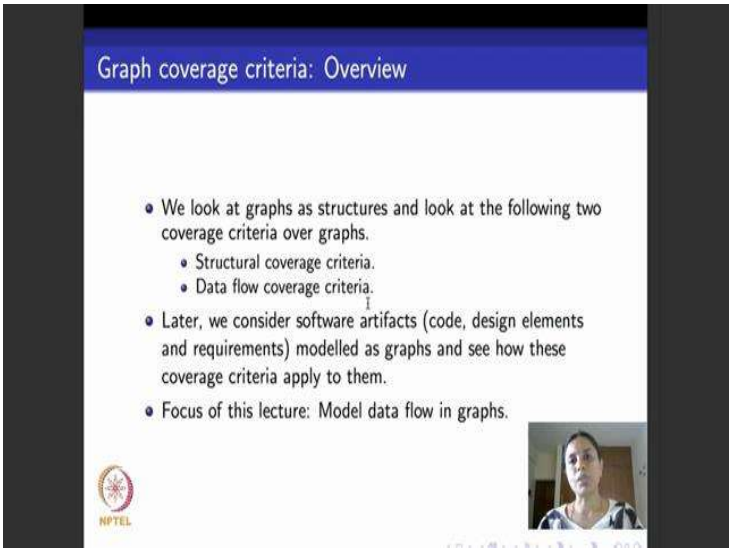
Thank you.

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

Lecture – 11
Data flow in graphs

Hello again, from today's lecture onward well continue with graphs, but we will begin to look at not only control flow and also about discuss about modeling data and how to write a test cases and define test paths that deal with handling about how data flows.

(Refer Slide Time: 00:30)

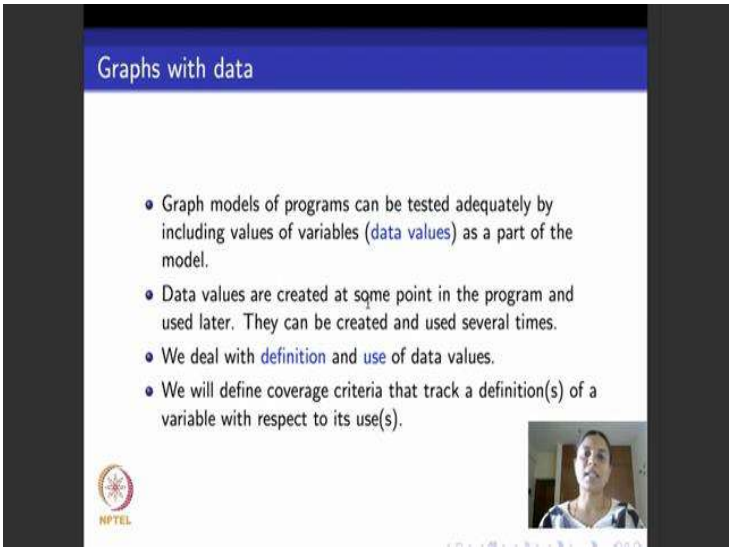


Just to recap where we are in the course. We are looking at a test case generation algorithms and test requirement definitions, based on graph coverage criteria. In the previous week, we saw structural graph coverage criteria which is just coverage criteria based on the structure of the graph, vertices, edges, paths and so on. This week I will begin with dealing with a second part which is data flow coverage criteria and then after we do these two, very soon we will look at various software artifacts core design requirements, see how to model them as graph and recap all these structural coverage criteria and data coverage criteria that we saw and see how to use them to actually test these software artifacts.

So, the focus of today's lecture and for a couple of lectures more to go would be related to data flow coverage criteria. What we will be seeing today is what is data in a software

artifact mainly programs as far as this is concerned and how to model data in graphs. In the next module I will tell you about once we have models augmented with data, how to define coverage criteria and what is the state of the art when it comes to defining test paths that achieve these coverage criteria.

(Refer Slide Time: 01:51)



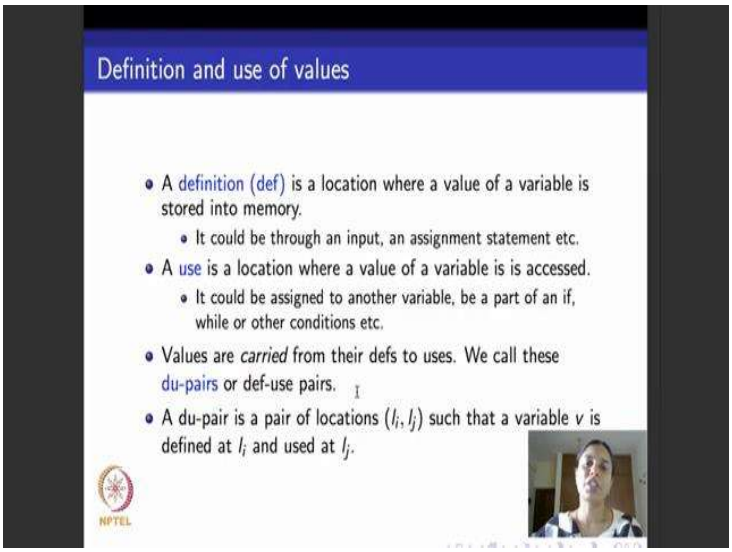
So, what is data? As far as programs are concerned, as far as many software artifacts are concerned there is only one kind of data available. Data is available as variables. Variables could have types: variables could have simple types variables could be complex types like arrays several other user defined types and so on, but whatever it is all the data that typical program deals with is basically represented as variables. What are these variables used for in the program? They are used because they represents some kind of information or data, but there are basically 2 kinds of basic ways in which we subject variables to the program. One is to say that a available gets created at some point in the program typically when it is declared, when it is declared and initialized. And at some point later in the program a variable gets used. So, it could be the case that certain declarations get missed or certain use is get missed.

Suppose the variable that is not declared suddenly gets used in a program then; obviously, the program will not compile. It will be a compiler error, compiler will say that there is a variable that is not well declared in the program, but think of the other way round right a variable is declared in the program, but never used at all. Maybe the

program was large and this programmer meant to use this variable. So, he declared it, but did not really use it because it was wrongly declared. We really do not want such things to happen.

So, we want to be able to track a variable from the place where it is defined to all the places where it is used. And we would not check whether it is used at all or not. So, what we will deal with throughout this lecture would be how to define data places where variables are used, and what are the places where they are actually put to use. And in the next lecture we look at coverage criteria that will track the definition of a variable with reference to it is use.

(Refer Slide Time: 03:53)



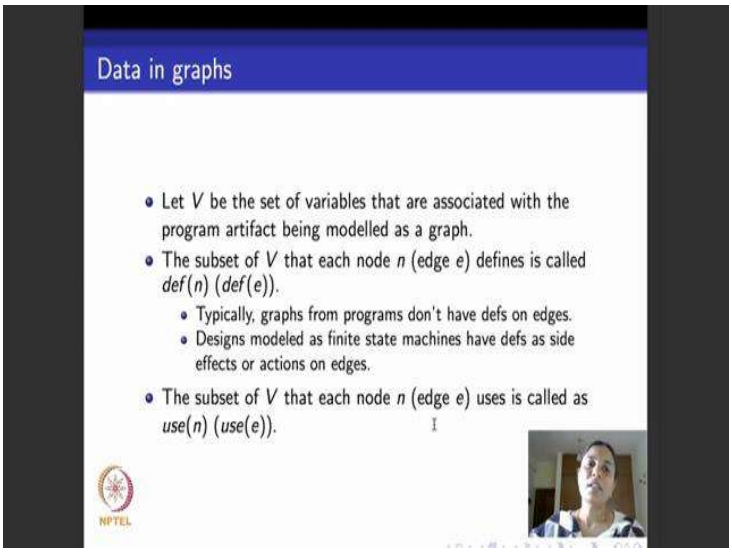
So, what is the definition of a variable? A definition of a variable is a location in the program where the value of the variable is stored into memory. That location could be an input statement right, it could be an assignment statement, it could be parameter passing from one procedure to another, could be any kind of statement. We will call statement abstractly as locations so that we can smoothly switch between using statement is a location and when we look at graphs nodes for as will become location. So, just to recap what is definition of a variable with use the word def for short; it is any location in the program where a variable is stored into memory it is value is stored into memory, assignment statements, input statements, parameter passing, procedure calls through parameter passing all these could be places where a variables value is defined.

Now, a defined variable has to be put to use. So, what is a use? A use is a location in a program where a variables value that is store is accessed is accessed by the statement of a program, by an assignment statement where and it could be assigned to another variable, it is accessed because it is being checked as a part of a condition that comes along with an if statement or it is checked, it is used as a part of a condition that comes along with the loop like for loop or while loop. So, whenever a value of a variable is accessed you say that that is a use corresponding to a variable. So, variable is defined and then in one or more ways and then a variable is used in one or more ways and typically a program carries the value of each variable from it is definition to it is use right. So, these pairs of definitions and uses of a variable are what are known as du-pairs or def use pair.

So, what is a du-pair a du-pair is a pair of locations say (l i, l j) such that a particular variable v is defined that l i and used at l j. Please note that even though ive not mentioned here v is an explicit parameter toward du-pair per variable they could be several du-pairs right, per pair of location there could be several variables that are defined and used at any point of time given a variable you are looking at what it is d u pair and given a pair of location, you are looking at all the set of variables that are involved in that location definition. And use of that location we may or may not mention it explicitly in this lecture, but there is always a parameter or an attribute to every definition or a use or a d u pair which is always a variable, because if variables are not there what do we define and what do we use.

So, what did we learn till now? We learned what a definition of a variable is. It is basically a place where if variable values first time written into memory: could be an input statement could be a declaration could be an initialization could be parameter passing several things what is a use a use of a variable is a a statement or a location where a value of the variable is accessed it is use could be because it comes on the right hand side of an assignment statement or it is used is a part of a predicate that comes for checking for an if or for a loop condition. What is a d u pair a du-pair is a pair of locations (l i, l j) such as the variable is defined an l i and used at l j?

(Refer Slide Time: 07:15)

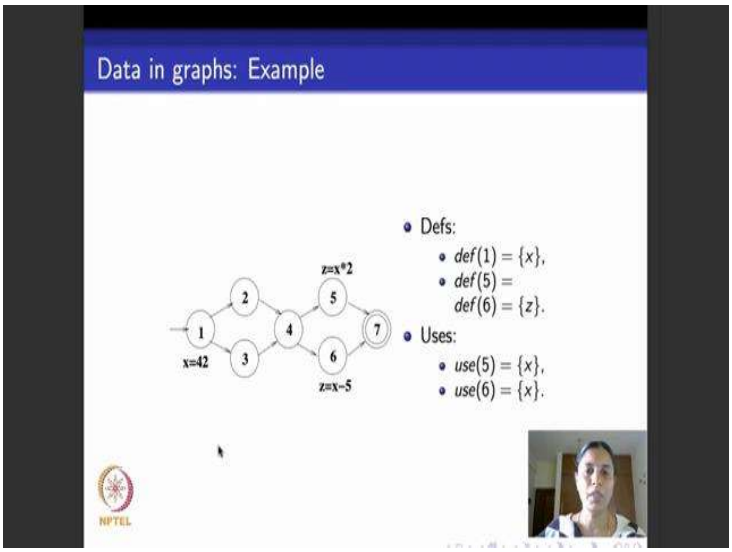


So, consider program that are let us say modeled as a graph. We will look at graph or data flow testing with reference to graphs in this module. So, we consider a program that is modeled as a graph, and let v be the set of variables that are associated in the program right. I want to know how these variables now come inside this graph right. So, what I do is I take the graph then graph has nodes and edges and per node and per edge, wherever it is applicable, I will say which are the variables that are defined in that node called as def of n for a node n. And for an edge e I will say which are the variables that are defined in the edge e.

Typically graphs that correspond to programs will not have definitions on edges. I mean will not have definitions on edges, as I told you. When is a definition, the definition is when a variable is written into memory. So, it is like having an assignment statement is a part of an edge, it is rare, it is rare or almost impossible in a graph that models a program. But for graphs that come as design models let us say graphs that come as finite state machines modeling designs, it is possible to have definitions on edges. Definitions could be thought of as actions that change the value of a variable or side effects related to an edge right. The subset of the set of variables that each node or edge uses is called use of n or use of e.

So, per node and per edge in the graph, we talk about what are the subsets of variables that it defines and what are the subsets of variables that it uses.

(Refer Slide Time: 08:47)

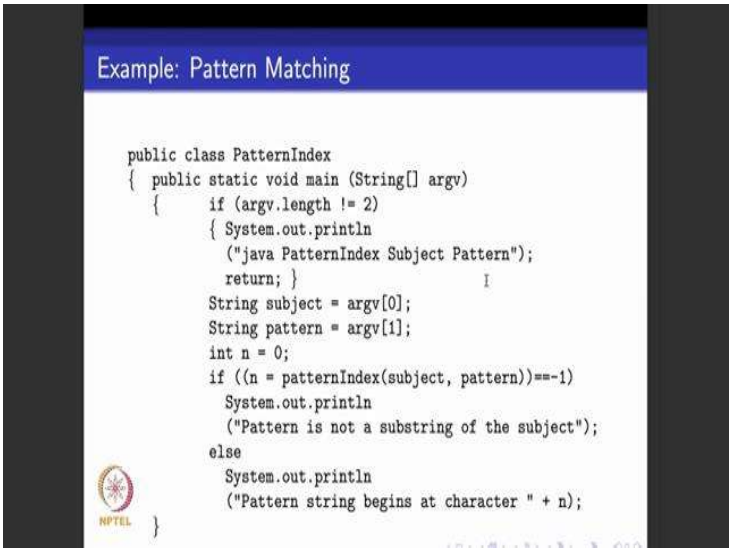


So, here is a small example. Here is a graph that has 7 nodes, initial node is one final node is 7. Not all nodes have defs and uses only few nodes have. What I have a done here? I said at the node 1 there is a statement which says x is equal to 42, at node 5 there is another statement which says z is equal to x into 2, at node 6 there is another statement which says z is equal to x minus 5. So, you can think of this statement x is equal to 42 that comes at the initial node 1, as defining the value of x at node one right. So, we say definition one def of one is this singleton set {x}. 2 and 3 and 4 nothing happens, I move on, I look at 5. For statements at nodes 5 and 6 have expressions that calculate the value of z based on the value of x right. So, at 5 we have z is equal to x into 2, at 6 we have z is equal to x into 5.

So, nodes, these 2 nodes can be thought of as having statements that define z. So, we say definition of 5 and definition of 6 both are the singleton set {z}. Now how is z defined at nodes 5 and 6 z is defined using an expression that involves a constant and it involves the variable x. Similarly, here z is defined using another expression which involves this constant 5 and a variable x. So, we said x is used at nodes 5 and 6. So, is it clear? So, the first occurrence where the value of x is initialized or set to some value or declared or read from input is it is definition. In this case the def, x is defined at node one and whenever there is a statement that puts this defined value into use right like here at nodes 5 and 6, we say x is used in those places. In addition to that at nodes 5 and 6 the value z

is assigned an expression involving a x, and so we say x is defined at nodes 5 and 6. We look at a more detailed example to understand this clearly.

(Refer Slide Time: 11:05)



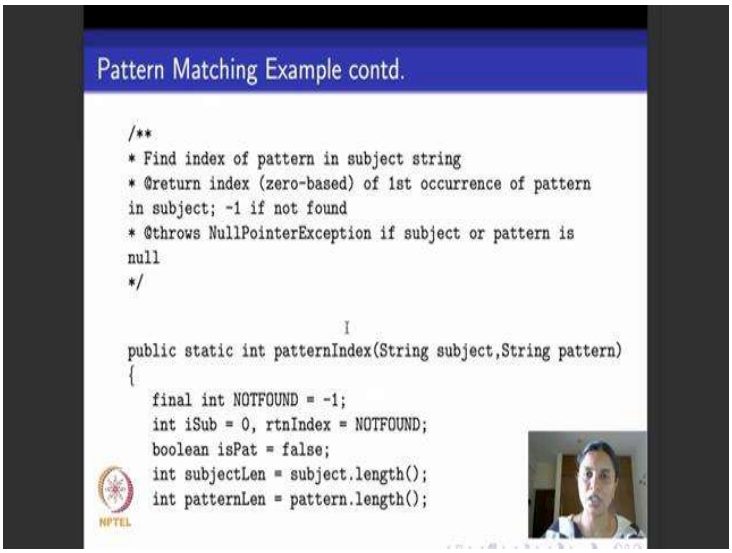
So, here is an example that deals with the Java code corresponding to a pattern matching example. Some version of pattern matching or the other if you look at you will find it in several papers and books related to testing. It is one of the popular examples that we always deal with in testing because mainly because it is got lot of rich control flow structure. So, what I have done here is I have put a Java code for pattern matching that was taken from this text book, Introduction to Software Testing by Ammann and Offutt. What I have done is to make it readable, I have spread across this Java code over 3 slides. So, we first go through the program line by line, well understand what this program does, then well go ahead and draw the control flow graph of this program, and look at how the data or the variables that come in this program can decorate the control flow graph, where is it defined where is it used right. We use this example to understand the defs and uses of all the variables on the control flow graph corresponding to this code.

So, this is probably the first full piece of code that you are looking at and we will reuse this example let a few other places as we go down in our lectures. So, what does this code do? This code takes two arguments: a subject and a pattern, both of them are strings. And then it looks searches for a pattern in that subject. Like for example, I could

say subject is a string which says institute, pattern could be something like ins right. If ins as a pattern comes in the subject, then you say that this pattern is found in the subject and you return index at which this pattern begins in the subject. If the pattern is not found in the subject then you return a string called -1. So, that is what it says. So, it says that you return -1. If the code returns -1 then your output saying pattern is not a substring of the subject. If the code returns any other character, then you say pattern does occur in the subject as a substring. And it begins at this particular index, yeah one more thing to note is this we look at a pattern as a contiguous substring we do not look at it as bits and pieces.

So, this is the first slide containing the first half of the code, which does on the initializations in the main prints .

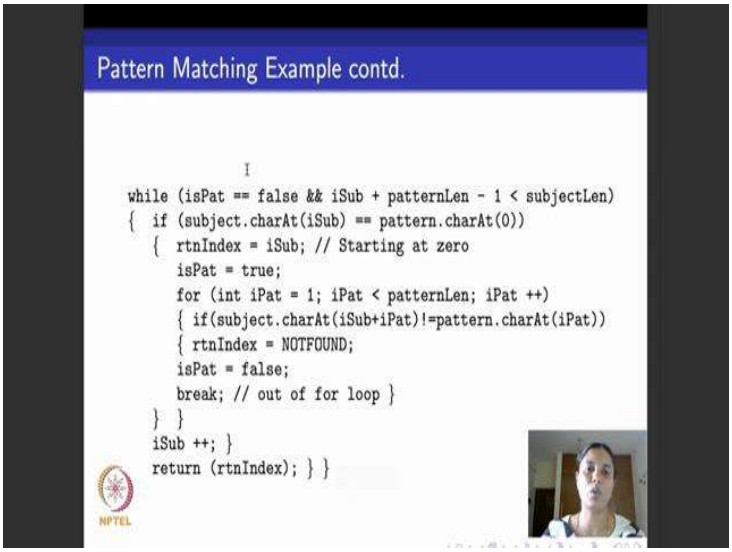
(Refer Slide Time: 13:30)



The actual code is given in continued across 2 more slides. So, here is the code that is continued from the first slide into the second slide. So, as I told you what is the goal of this code ? It is to find the index of the pattern, if it occurs in the subject and it will return the index 0 based in the sense that we count indices starting from 0, the first character of subject is index is 0, right. If the first occurrence of pattern in the subject we return -1 if it is not found if subject or pattern is empty then we threw a null pointer exception right. How does this pattern index work ? As I told you takes 2 arguments a subject and a pattern, and it does all these kinds of initialization?

So, it uses these variables not found, it initializes it to -1 in integer variable. It uses 2 other integer variables, isub you can read it as index that moves across the subject it initializes the index of the subject to 0. And then it sets the return index to be -1 which is not found. Remember when the return index continues to be -1 if you go back to the previous slide I return saying pattern is not a substring of the subject. And then it uses a few Boolean variables, ispat standing for 'is pattern', initializes it to false and then it calculates the length of the subject and the length of the pattern. Why do we need the length of the subject and length of the pattern ? Because we want to use a for loop to move across the length of the subject from left to right and another for loop to move across the length of the pattern from left to right?

(Refer Slide Time: 15:08)



So, here is the main part of the code. There is a while loop inside this there is a condition there is a for loop, and then there is an if loop. Why am I talking about this while if for if, is to tell you that this code has very rich control flow structure. The control flow graph of this code will involve some amount of branching which is a very which makes it a very interesting case for testing and here there are lots of nested loops one inside the other. So, the control flow structure it becomes even richer and so it is an interesting example to look at for testing. Now going back and understanding what the code is. So, it says as long as it 'is pattern' turns out to be false and pattern index is within the subject of the length of the pattern is within the subject index then what do I do in this if loop, I start matching character by character.

So, I say wherever I am in the subject which is this index of subject isub, does it match the first character of the pattern? First character of the pattern is present at index 0. If it is then, you say that the return index could be this value. Let us say if you take the earlier example that we had right, let us say a subject was the string institute and the pattern was ins, so, right up front at the index 0 itself, the subject and the pattern match, i and i match right. So, you set the index where it matches to be the return index and then you set this Boolean flag is pattern found which is ispat to be true right.

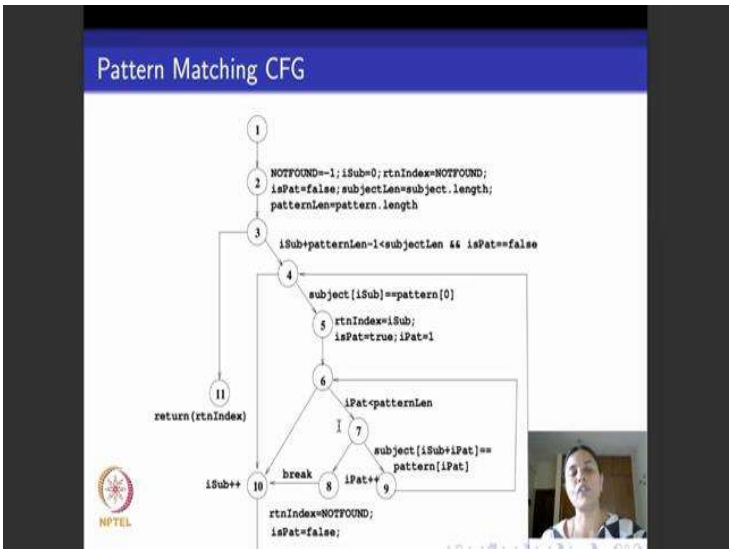
Now what you do you enter a for loop, keep incrementing the position that you look at in the pattern as long as you go till the end of the pattern and see if every character of the pattern, one at a time, matches the corresponding character in the subject that is what this if statement does. If there is a mismatch at some point in time it says he sets the return index to not found and makes is pattern false and comes out of the for loop right. If every character of the pattern continues to match the subject it keeps incrementing the index in the subject successfully finishes the for loop and it will return this value here that it is set as the return index. So, is it clear what the code does?

So, I just to quickly recap we are looking at this pattern matching code. It takes 2 arguments, a subject and a pattern and it uses a few Boolean variables to search for the pattern in the subject. What it does is initially it assumes that the pattern is not found in the subject. So, it says 'is pattern' is false, and then here it checks with the length of the pattern is well within the length of the subject and then it says now let us look at the first character of the pattern which is at index 0 along with wherever I am in the subject which is given by this index isub, if they match if this return statements gives true, then you say I may have found the pattern beginning at this index.

So, you set that to be the return index. And then you say I have found the pattern set you to true, then you enter a for loop where you continuously walk down the pattern and check if every character in the pattern that you encounter, in sequence, matches the subject from the index in the same sequence. If there is a mismatch at any point in time then you reset a return index to not found say pattern is not found come out of the for loop. But if we successfully finish executing this for loop then you increment the subject and then you can return this value of the index where the pattern was found in the subject.

So, now our first goal is to draw the control flow graph corresponding to this program right. So, as I told you for control flow graph this program is spread across these 3 slides, but I am ignoring this part of the code, and this part of the code which is just commented out. I begin to draw control flow graph mainly from here, from this main pattern index method. So, we see this pattern index method it has a whole series of assignment statements, it has a loop, a branch, another loop another branch.

(Refer Slide Time: 19:15)



So, here is how the control flow graph of that main method looks like. This node one is a dummy node which represents this statement. It says subject and pattern are passed to this node. There is no concrete statement representing this so I have left it blank. Node 2 stands for all these initializations that you do in the beginning of the program right all these statements these 6 statements that are there. You could put it in different nodes, but typically when we draw control flow graph, when we have a sequence of assignment statements without any control statement in between just assignment statements one after the other, it is a standard practice to collapse them all into one node and put it as one node and then augment that node with all these statements. That is what I have done because there is not much point in calling each of them as separate node, we really do not find any use for it when it comes to testing.

So, we collapse it all and put it into one node. So, this single node represents all these assignment statements. After that what do we do with that code we get into this while

loop right. So, that is this while loop at node 3, I say this is the condition that I check in the while loop you can see that is the condition I check. This condition could be true this condition could be false. Just for increasing the readability I have written it only as a label for this node the reverse of this condition the negation of this predicate is true here. I have not mentioned it because the graph was becoming the figure was becoming too cluttered. So, I have just left it at that right. So, read the label of this edge as negation of this.

So, I am entering my while loop here. As per the code the next is an if statement which check for another condition. So, that I am entering my if statement here, if statement returns true, this is if statement returning false. After my if statement returns 2, I do two assignments I set, what does is it mean for the if statement to return true I have found the first match. First character with the pattern has found a match in the subject. So, I do these 2 assignments if you remember right I set the index and say I have found the first match by setting the flag to be true. So, that is this node 5.

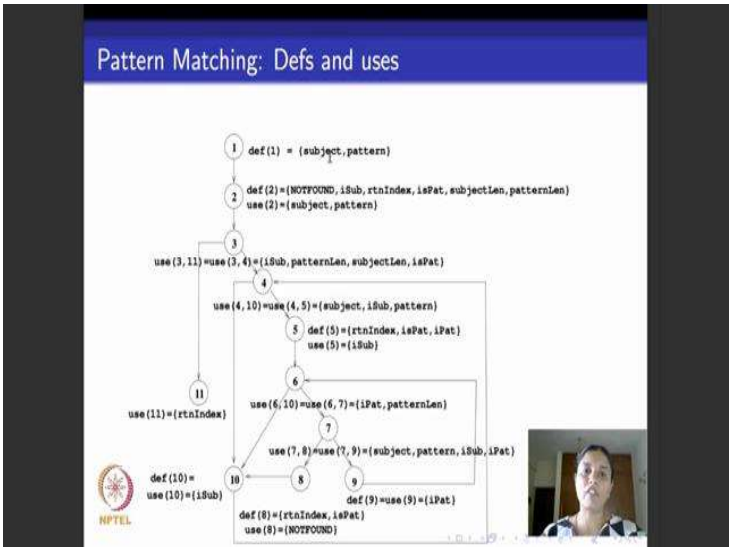
After that I enter, sorry, after that I enter this for loop, that is this. This is the statement corresponding to the for loop being true. And inside the loop I keep checking if the pattern index matches the subject index. At node 9, I increment the pattern index go back and repeat this for loop here from 6 to 9 and when the condition fails I come out with a for loop. I break out of the for loop and I am here that is what this represents and then this is the thing that they use to go back for the main while loop. This is when I exit the main while loop, I do this final statement which is return index. Is it clear? I am sorry I have to go back and forth across the slides because there is no way I could have put the whole a program and the control flow graph into one slide.

So, just to summarize we have looked at a pattern matching program that I had presented to you across 3 slides which were this, this and this. It is a program that has rich control flow structure looks if pattern given as a string comes with a substring of a subject. What we did here was to take the main method in the program and draw it is control flow graph.

So, I will repeat this exercise of how control flow graph will look like for various program constructs, but hopefully this example will help you to understand it also. Now we go back to the main goal of this lecture which is to understand data right, data

definition and data use. What is data for this particular program? Data is all these variables: NOTFOUND, isub, rtnIndex, subject, pattern, ispat. So, many variables are there right. So, what do I do, I take the same control flow graph instead of annotating it with statements I say which are the variables that are defined and which are the variables that are used at various nodes in the graph and various edges in the graph.

(Refer Slide Time: 23:46)



So, that is this graph that I have drawn here. It is a same control flow graph instead of depicting statements it now depicts definitions and uses. So, if you look at node number one, it says the definition of one that is at node number one, which are the two variables that are defined? The Two variables that are defined at one are subject and pattern. You might wonder this here it is blank and while have I suddenly put def(1) as subject and pattern as I told you when you explain the CFG read this node one as these two patterns subject and pattern being passed on as parameters to this method. Because this passed as parameters I am not depicted here, but because it is passed as parameters it is defined here.

Now, what is the def (2)? If you see there are, so many assignment statements here which is not found as -1, isub is 0, rtnIndex is NOTFOUND, ispat is false and so on right. So, I say definition of 2 is all the variables that have been initialized here, which is NOTFOUND, isub, rtnIndex, ispat, subject length, pattern length. Now 2 also has two uses which is subject and pattern. Why is it use of subject and pattern I use the variable

subject and pattern that was passed as parameters at node 1 and compute their length here subject length and pattern length and assign it to subjectLen and patternLen. So, I have used these 2 variables subjectLen and patternLen. So, that is why I have put use of 2 as subject and pattern right. Is one more settle point do not worry about it. If you do not understand it if you see I have initialized not found to be minus 1. So, I have to def(2) as not found and I have also initialize a return index to be not found.

So, strictly speaking I should put use(2) as not found. So, you could go ahead and do that, but at the little later well see that we typically do not consider what are call local uses of a variable. Because I have collapsed all these 5, 6 assignment statements into one node in the control flow graph, I say this variable is defined and used within one node. So, it is like a local use of a variable. We typically do not capture local uses of variables, but if you are not comfortable with this if you want to see use(2) it is subject pattern and not found you can go ahead and do it there is no harm in it right like this I move on.

Now, I can define defs and uses for edges also remember well saw definition and uses for vertices and definition and uses for edges. So, which is this, if a predicate that labels this edge. So, it uses thus a variables isub, patternLen, subjectLen and ispat. So, that is what I have written here. The use of this edge is this predicate i is this set isub pattern length subject length ispat. If you remember I told you that this corresponds to a branch, this is the branch where I enter the while loop. So, this predicate is true. This is the branch where I exit the while loop where the predicate becomes false.

Just for readability I didnt write the condition here, but the same set of variables with this condition negated comes here also. So, I say use of this edge (3, 11) and this edge (3, 4) is the same set right. Because the both the predicates one the positive version and one the negated version both will use the same set of variables. Similarly, this is the if statement. So, use of this edge (4, 10) and this edge (4, 5) is this set subject, isub, pattern and then at statement 5, I have these 2 assignment statements. So, I say rtnIndex, ispat and ipat are defined at the statement. Please note that ipat is defined once here also sorry, ispat is defined once here also at 2 and it is defined at 5.

So, if you go back and look at the code it is initialized at 2 and its value is reset to another value at 5. So, both we call them as definitions because it is again written, rewritten in memory. So, definition of node 5 is this set. Similarly use on node 5 is this