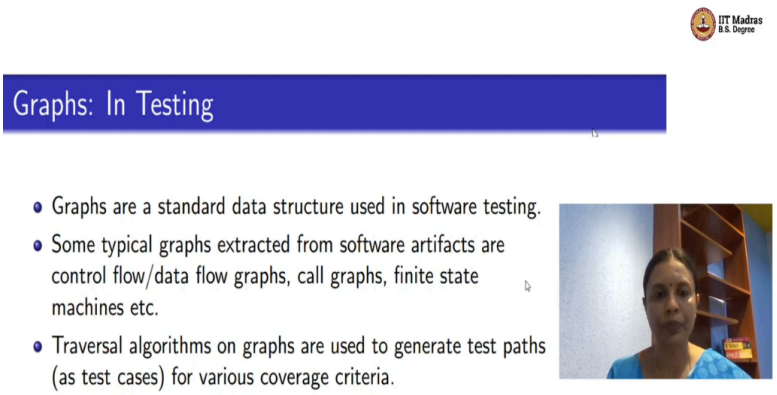
Software Testing
Professor Meenakshi D'Souza
Department of Computer Science and Engineering
Indian Institute of Information Technology, Bangalore
Week 2 Lecture 2
Elementry Graph Algorithms

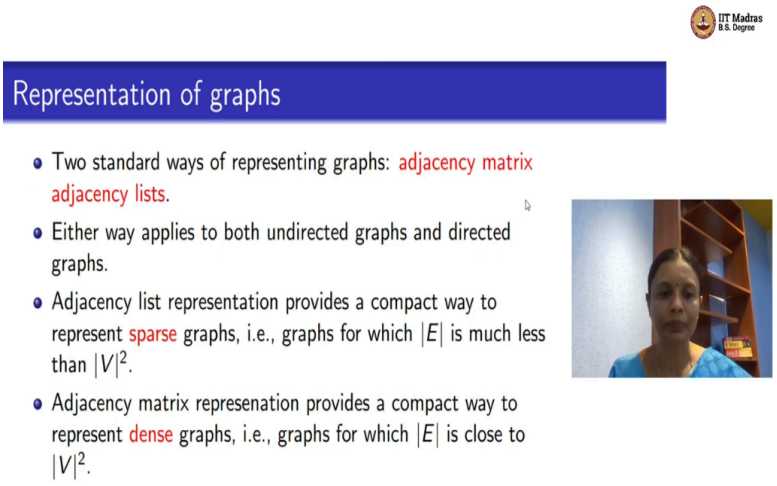
Welcome back to the second week. As I had told last time, we are going to use graphs as data structures to start with to learn some test case design algorithms. To implement those test case design algorithms you will need some fundamental graph algorithms to work with. And this lecture and the next one, is about teaching you a few of those Elementary Graph Algorithms. Most of you might be familiar with it but we will redo it once again to ensure uniformity as a part of this course. So, we are going to look at concepts in elementary graph algorithms.

(Refer Slide Time: 00:53)



The first one that we are going to see is breadth first search followed by depth first search and its applications in testing. So, after this I will tell you the kinds of graphs that we will use for testing- control flow graphs, data flow graphs, call graphs, and things like that. But the goal of this lecture and the next one is to purely learn these graph algorithms from an algorithm design point of view.

(Refer Slide Time: 01:21)



Before we look at the algorithms themselves, it helps to understand how graphs are represented as far as a program is concerned. There are several data structures that you can use to represent graphs. The two most simple data structures are matrices and lists. Graphs can be represented as a matrix called the adjacency matrix or as a list called the adjacency list.

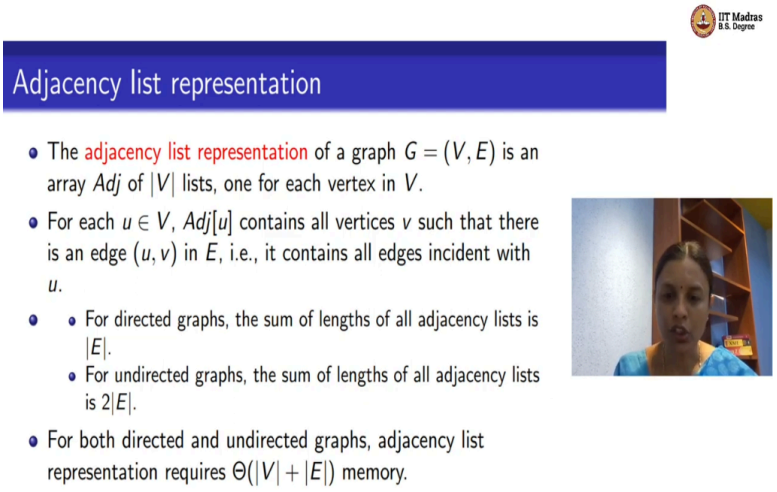
Both undirected directed graphs, the kind of graphs that we are going to use in testing, those that have initial vertices, final vertices, those that have annotations and edges, all of them can be represented using adjacency matrices or adjacency lists. Adjacency list, basically, talks about for every vertex take the list of all vertices that are adjacent to that vertex.

Adjacency matrix for unweighted graphs is basically a 0,1 matrix that assigns a value of 1 if there is an edge connecting two vertices in the graph or it assigns a 0 if there is no edge connecting two vertices in the placeholders for the vertices in the matrix. Adjacency list, each list in the adjacency list for every vertex grows or shrinks based on the number of vertices that are adjacent to that. So, if a graph is sparsed that is if it does not have too many edges then adjacency list is believed to be a compact representation.

On the other hand, adjacency matrix, the size of the matrix, the dimension of the matrix is purely dictated by the number of vertices. So, irrespective of whether the graph is parsed or densed, it is going to be a matrix of dimension n $\times$ n where n is the number of vertices. So,

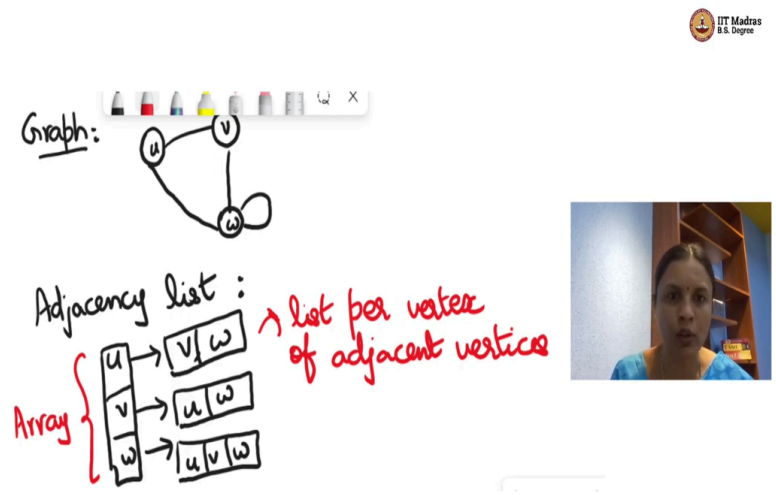
adjacency matrices are considered specifically good if your graph is densed because they have a fixed size.

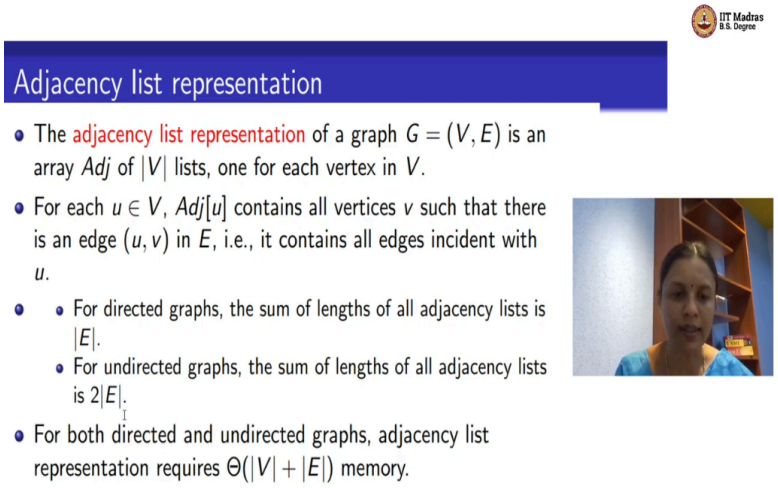
(Refer Slide Time: 03:11)



So, how does an adjacency list representation look like? So, as I told you take a graph. Let us say it has some vertices, it has some edges. An adjacency list is, basically, an array of lists, one for each vertex.

(Refer Slide Time: 03:28)





So, let me just shift to the writing board for a minute and tell you how an adjacency list looks like for a small graph. So, let us take this graph as an example. So, let me draw a small graph for illustrative purposes. So, let us say these are the vertices. Let us label them u, v and w. So, how does the adjacency list for this look like? So, it is an array of lists. So, there is going to be a list corresponding to u in this array. There is going to be a list corresponding to v. There is going to be a list corresponding to w. And what is that list going to contain? For u what are all the vertices that are adjacent to u.

So, for u, vertex v is adjacent to u, and vertex w is adjacent to u. So, this is the youlist that this is going to contain. And for v, what are all the vertices that are adjacent to v, u is adjacent to v it is again and w is adjacent to v. For w, what are all the vertices that are adjacent to w. u is adjacent to w, v is adjacent to w, w is adjacent to itself. So, each of these things is a list. One list per vertex of adjacent vertices and this is a array. So, you see the size of the list grows or shrinks based on the number of vertices that are adjacent to it. So, this is the simplest representation of a graph.

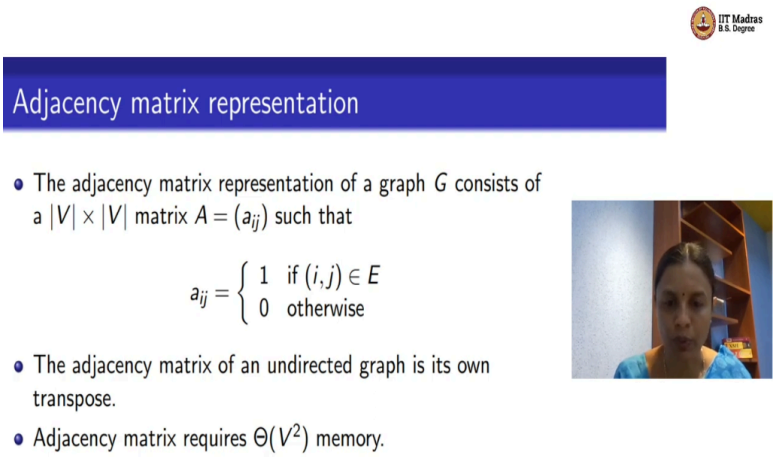
So, let us go back to the slides. So, this is what it says here. In the second bullet, for each vertex in the graph, adjacency list of that vertex contains all the vertices such that there is an edge from the vertex u to that vertex contains all the edges incident on the vertex u. For directed graphs, obviously direction does not matter, the example that we saw was for undirected graphs.

So, for directed graphs, if I take the sum of all the lengths of the adjacency lists then I will get the number of edges. Because it I put one element into a specific list if there is an edge. For

undirected graph I count edges in both directions like we did, right, u was adjacent to v, v was adjacent to u. So, the edge from u to v, we counted it in the adjacency list for u, we counted it in the adjacency list for v. So, the sum of the lengths of the adjacency list becomes 2 |E|.

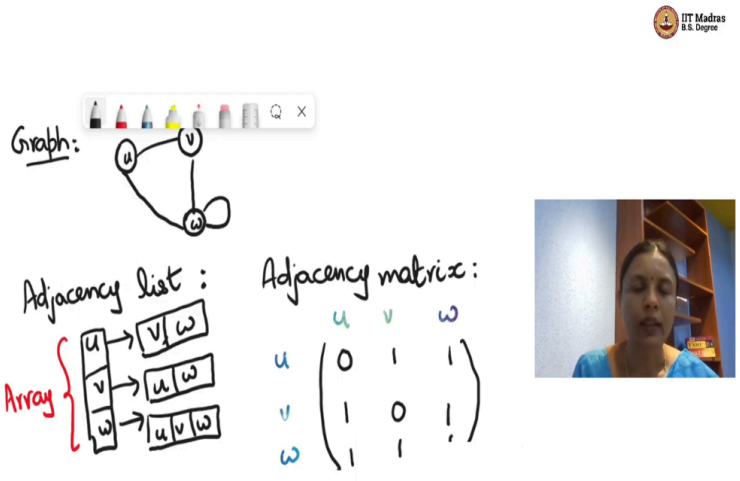
Adjacency list representation either way requires how much memory, how much space it requires $\theta(|V| + |E|)$. Because one an array running over vertices and then across all the lists that are mapped to each vertex I count order |E|, either |E| or 2 |E| edges. So, that is an adjacency list.

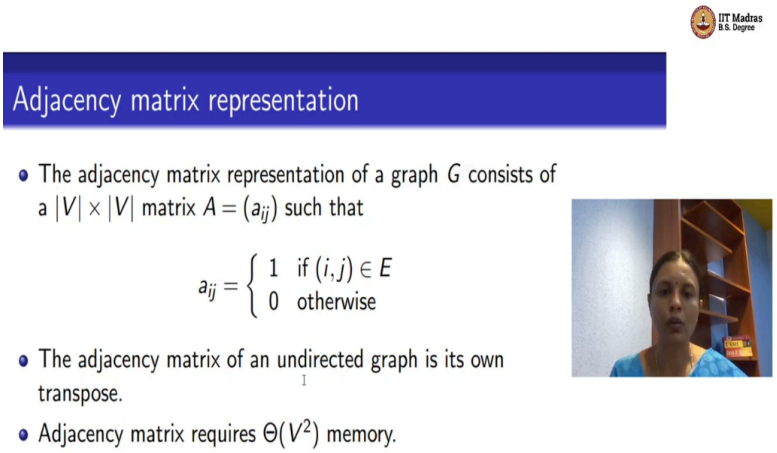
(Refer Slide Time: 06:49)



How does an adjacency matrix representation look like? This is very simple. If a graph is n vertices, take an n by n matrix and fill it up with 0 or 1 based on whether 2 vertices are adjacent or not. So, we will again look at a small example.

(Refer Slide Time: 07:04)





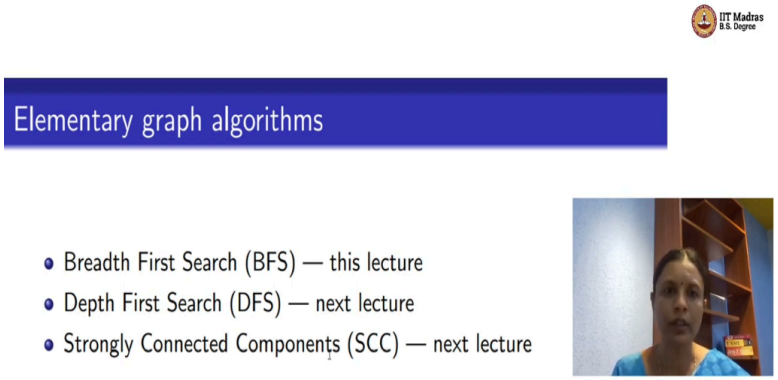
So, for the same graph, let me draw the adjacency matrix. So, I will just draw it here. Let me just erase this bit for you. So, this is the adjacency list that we saw. Let me now draw the adjacency matrix. So, how many vertices are there in this graph? Three vertices. So, I get a 3 by 3 matrix.

So, let me just assign placeholders. So, there is a place for vertex u, there is a place for vertex v, there is a place for vertex w. There is one more place for vertex u, there is one more place for vertex v, and there is one more place for vertex w. Now, we have to fill in the entries of the matrix. So, here I have to fill in a 0 or 1. Is there an edge from u to u in the graph?

There is no edge. So, I fill a 0. Is there an edge from u to v? Yes, there is. So, I fill a 1. Is there an edge from u to w? Yes, there is. So, I write a 1. I go on like this. There is an edge from v to u. So, I fill that corresponding entry with 1. There is no edge from v to v. So, I fill that corresponding entry with 0. There is an edge from v to w. So, I fill this entry with 1. w happens to have edges incident on all the vertices. So, all the rows of w will be filled with one. So, this is my adjacency matrix. Is this clear?

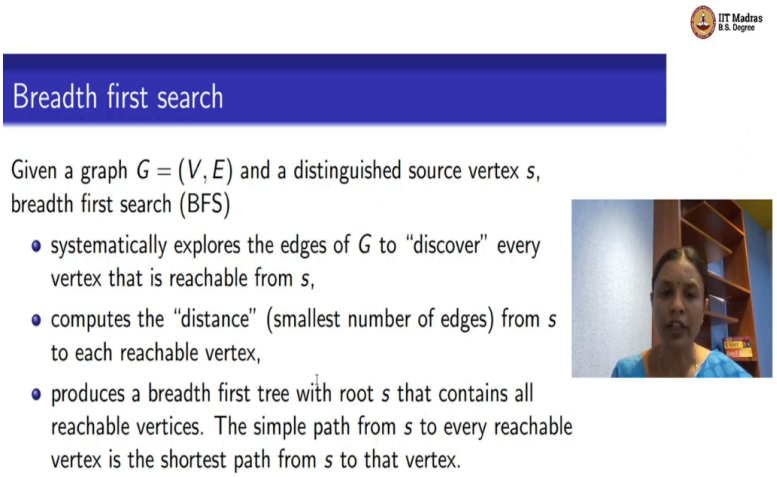
This is how adjacency matrix is looks like. So, an adjacency matrix is a mod V by mod V matrix. There it was a 3 by 3 matrix such that the ij'th entry corresponding to two pairs of vertices is one if that corresponding edge is there in E otherwise you fill it with 0. So, if a graph is undirected then if I take the transpose of its matrix, I will get back the same matrix. Because there is no difference. It is symmetric. Adjacency matrix obviously needs mod; I mean $\theta(|V|)$ memory.

(Refer Slide Time: 09:06)



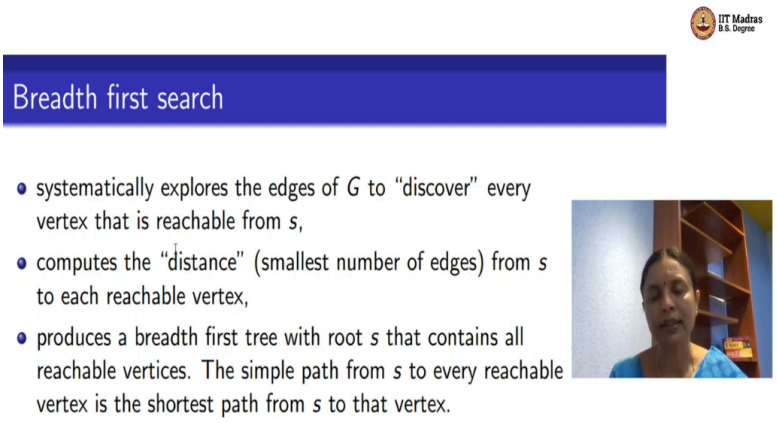
So, these are three basic graph traversal algorithms that you will use for the purposes of this course. Very basic algorithms, most of you might be familiar with it by now, but we will just recap it quickly so that you are not lost looking for other resources to learn about them. The two traversal algorithms that we will use are Breadth First Search and Depth First Search, equal in terms of complexity and enjoy their own properties. In addition, we might also use algorithms that determine strongly connected components in the graph.

(Refer Slide Time: 09:41)



So, we will start with breadth first search in this. So, let us look at the term breadth first search. So, it is, basically, an algorithm that searches through the graph in a breadth first way. So, it goes, it starts at some vertex then it looks at the adjacency list of that vertex fully and then goes to the first vertex in the adjacency list that it is populated and goes on from there. So, it goes breadth first. It does not go deep down a path; instead it goes via adjacency list of a vertex.

(Refer Slide Time: 10:12)

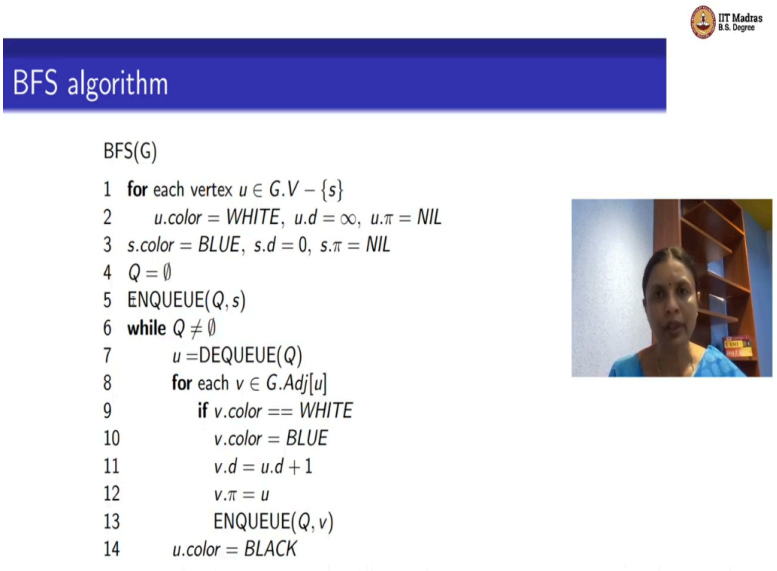


So, given a graph G, the inputs to your breadth first search algorithm are a graph and a place to start. So, there is a designated start vertex. What does the breadth first search BFS

algorithm do? It explores or walks through the graph by using its edges and finds vertex one at a time.

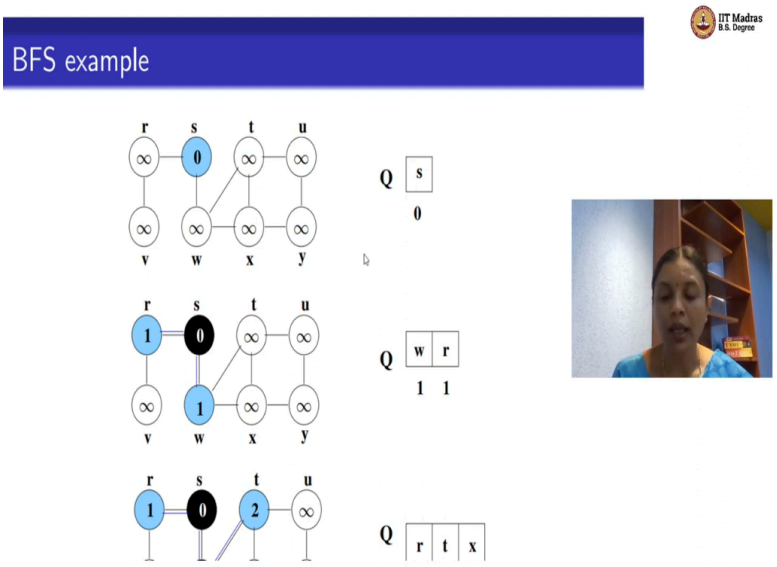
In that process, it computes the smallest path that is there between any two pair of vertices in the graph. By smallest I mean smallest in terms of the number of edges that you can encounter in that vertex. It also can produce a tree that you can use to traverse from one vertex to the other. We call that as a breadth first tree and it will contain all the vertices that are reachable from the vertex s.

(Refer Slide Time: 10:58)



So, this is the algorithm. The algorithm maintains a queue. Queue is a standard data structure. It comes with operations of 'enqueuing. Which is adding an element to the queue, 'dequeuing' which is removing the element from the queue. What I will do is I will first walk you through an example then I will come back to this algorithm.

(Refer Slide Time: 11:17)



So, let me just go to that example. I have it in the slides just for ease of use. So, on the left here, top left you see a graph. Study the graph a bit. How many vertices are there? r s t u v w x y, eight vertices are there. And the vertex named s is the starting vertex, is the source vertex from where you start.

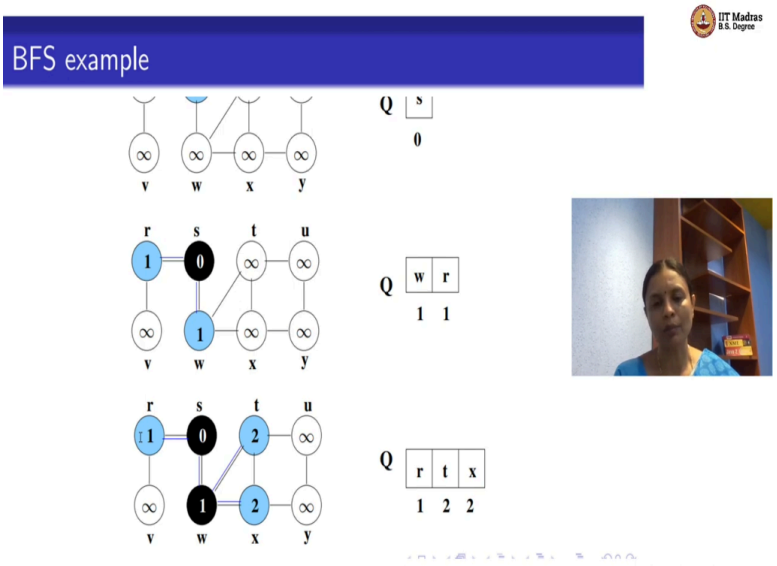
And your goal is to see compute, paths for all the vertices that are reachable from s. So, for example, how do I go from s to r? There is one edge connecting s to r. So, that is the path from s to r. How do I go from s to let us say u, the vertex u right here? So, I could do s to w, w to t or t to u and t to u.

Or another path would be s to w, w to x, x to y and y to u. Or I could use s to w, w to x, x to t, t to u. So, several different ways are there for going from s to u. Breadth first search explores all these different paths that connect the vertex, connects the vertex reachable from the source vertex s and gives you the path that is shortest, shortest in terms of the number of edges that you use along the path. So, how does the algorithm work? So, we are going to run breadth first search on this graph. What is the first thing that we do?

The first thing that we do is to assign numbers to each vertex. As you see here, there are eight vertices; the source vertex from where breadth first search starts is assigned the number 0 all other vertices are assigned the number infinity. This is the first step that breadth first search algorithm does. You always start breadth first search from a source, assign the number 0 to that source, for all other vertices in the graph assign the number infinity.

Alongside that you keep a queue data structure. Let us call that queue as Q. To start with you put the source vertex from which you are going to start the breadth first search into that queue. So, to start with the queue is initialized with the source vertex s. So, this is the first step of the breadth first search algorithm. Then what does the algorithm do?

(Refer Slide Time: 13:45)



It starts examining the adjacency list of the source vertex s. So, if you look at this graph, I do not have the adjacency list but I can get it from the graph. Which are the two vertices that are adjacent to this source vertex s? There is an edge from s to r. So, r is adjacent. There is an edge from s to w.

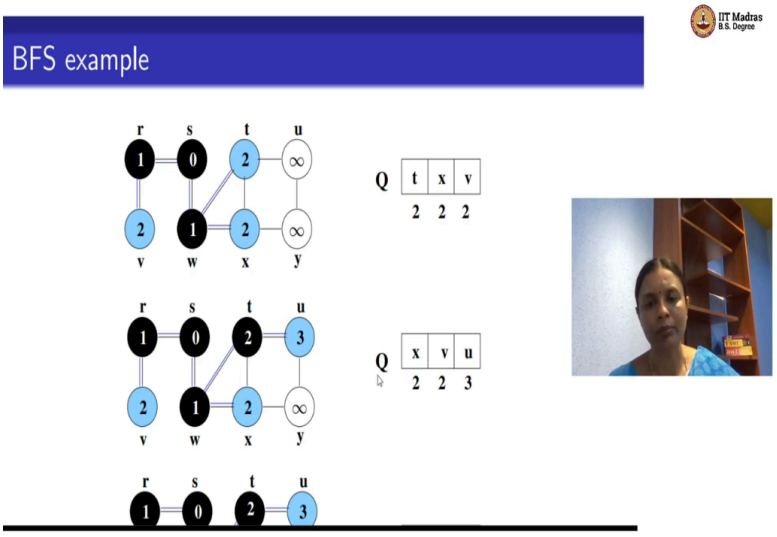
So, w is also adjacent to s. So, what you do is you take out s from the Q and you put all the vertices that are adjacent to s into the Q. So, that is this Q in the second step for you. So, s goes out of the Q, w and r which are adjacent to s get into the Q. There is no more vertices out of s to be explored.

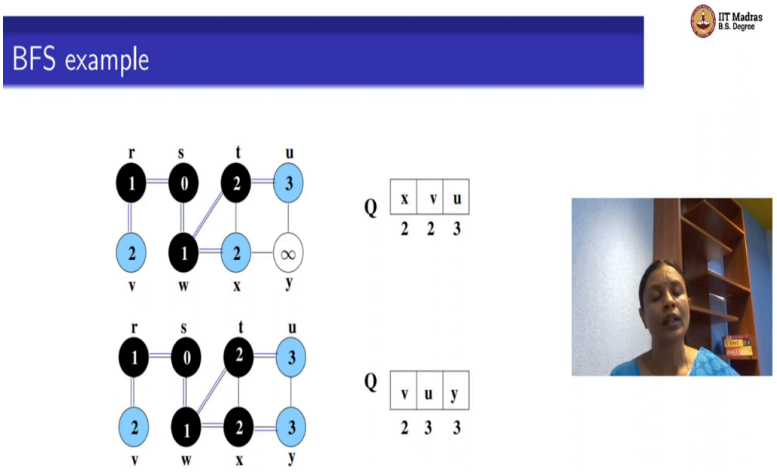
So, the color that I gave to s which was originally blue, now, changes to black indicating that I have explored all the outgoing edges from the vertex set s. r w and in fact all other vertices were colored white in the first step. Now, r and w are into the queue. So, they got colored blue, w and r are in the queue. So, you might ask the question between these two vertices w and r, which do I put first in the queue? Which do I put second in the queue?

The answer it does not matter for breadth first search algorithm. You can put them in either way. Now, you repeat the same process. Take the top most vertex in the queue which happens

to be w, explore its adjacency list. Let us look at the adjacency list of w. w's adjacency list has t; it has x. Throw in them into the queue and get out w from the queue that is the step. So, w becomes colored black t and x which were new vertex that were added into the queue gets blue, and r stays on the queue, so, it stays in the blue. Now, again repeat the same step.

(Refer Slide Time: 15:35)



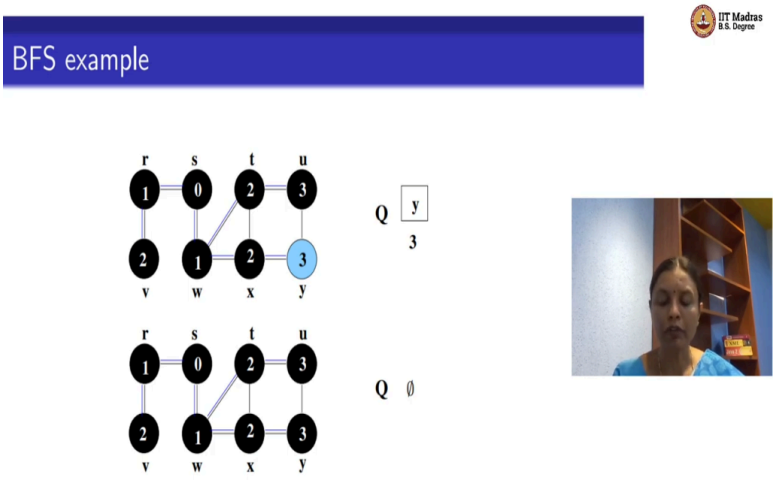


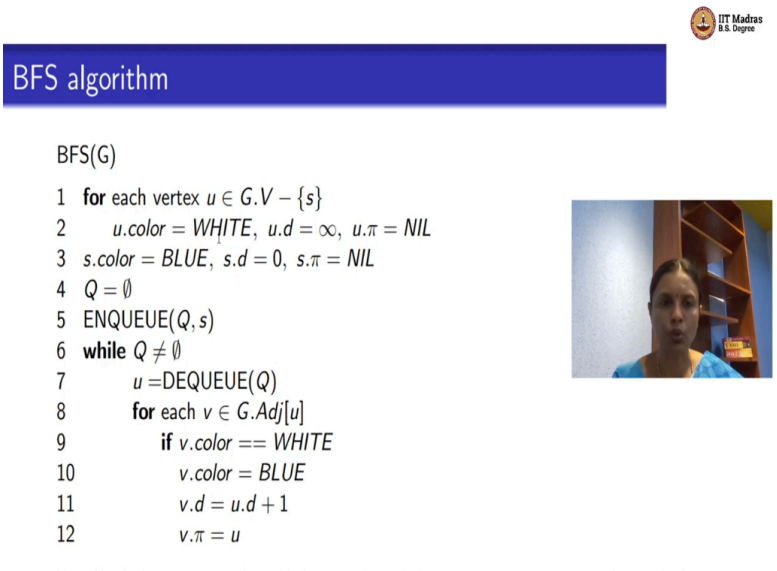
Look at the top of the queue. What is there at the top of the queue? r is there at the top of the queue. Look at the adjacency list for r. r has two vertices that are adjacent s and v. So, s has already been explored add v to the end of the queue. Remove r from the queue, and color it black. Move on like that. After this which is the next vertex, t is the top of the queue, explore

the adjacency list of t. t has what? w, u and x. So, w is already been explored, u is colored blue, x is colored blue gets into the queue.

Like this you move on and on and on and on, keep coloring vertices from white to blue to black. The vertices that are white have not yet been explored by breadth first search algorithm. The vertices that are blue you are currently wanting to explore parts out of those vertices by looking at the edges that are adjacent to them. The vertices that are colored black are those vertices that are out of the queue and the paths that go out of them or the edges that go out to them have been fully explored, no more to explore.

(Refer Slide Time: 16:48)





So, this algorithm continues by adding and removing vertices from the queue in a FIFO fashion till all the vertices that are reachable from s are explored and Q is empty. Q is empty means what? There are no more blue colored vertices; all the vertices are colored black. This is when the breadth first search algorithm stops.

In this process it also outputs the tree of shortest paths. I will come back to that in a minute. If you see as we have moved along, we have double lined some edges. There is a blue color, there is a black, there is a blue line and a black line. Those are the vertices that get into the tree.

But for now, let me go back to the pseudo code of the algorithm and show you what it is. So, this is the main BFS algorithm. So, it says for each vertex u in the graph, apart from the source vertex s, s is the source vertex, set three attributes to that vertex. It set its color to white that is u dot color. Set its distance from the source u dot d to infinity. Set its pi parent, parent for pi in the tree to be nil.

For s set its color to be blue. Set its distance from itself to be 0. Because it is the same vertex. And set its parent is parent to be null because that is going to be the root of the BFS tree, the place from where you begin BFS. Initialize a Q to be empty. First you enqueue s into the Q. enqueue Q s. that is what we did.

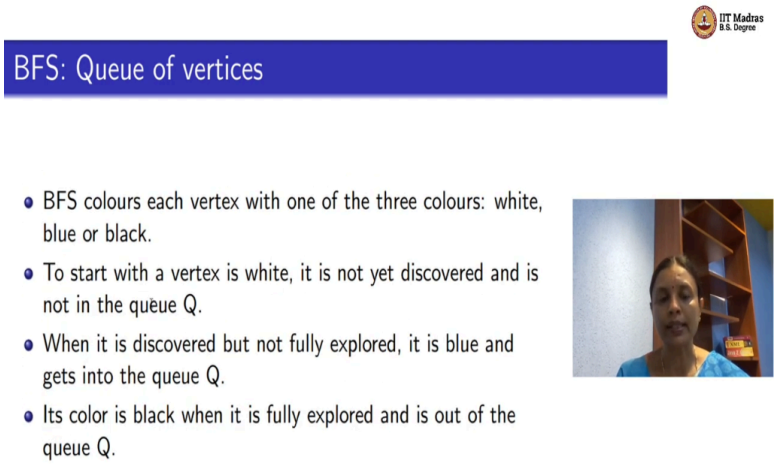
Now, as long as the Q is non-empty, repeat the following. Take the vertex at the top of the queue, dequeue that vertex. Start exploring its adjacency list. Let us say u is that vertex, for each vertex in the adjacency list of u, consider each vertex in the adjacency list of u, if it is

colored white, color it blue. Set its distance from s the d attribute by adding 1 to it, set its parent to be u and put it in the end of the queue.

Repeat this process. And when you are out of the for loop, you have explored the adjacency list of this vertex u fully. So, you check the color of u to be black and get back into this while loop. The while loop, basically, repeats as long as the queue is non-empty and inside the while loop, for the vertex that is there at the top of the queue, you explore its adjacency list.

Consider all the edges that were adjacent to that vertex at the top of the queue. And if it was colored white that is if you found a previously unvisited vertex, color it blue, put it under the queue, set its distance by, increase its distance by 1, set its parent to u and keep moving from there. This is, basically, the algorithm that we saw through this example.

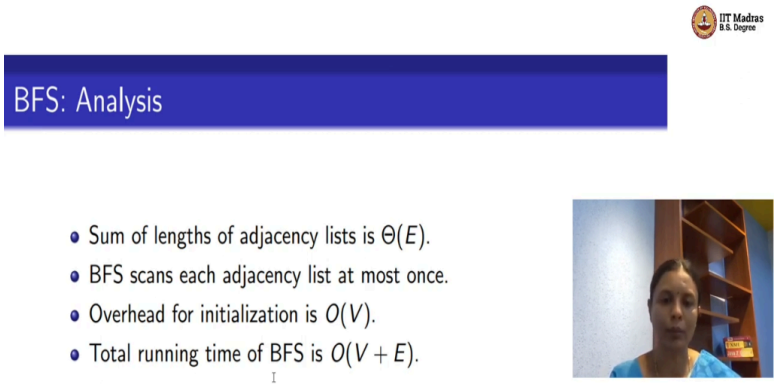
(Refer Slide Time: 19:49)



So, here is a summary of what happens in the queue. So, BFS maintains a queue of vertices. It basically colors each vertex with one of the three colors. To start with they are all white. Along the way they can get blue color and when I am done with exploring a vertex I color it black.

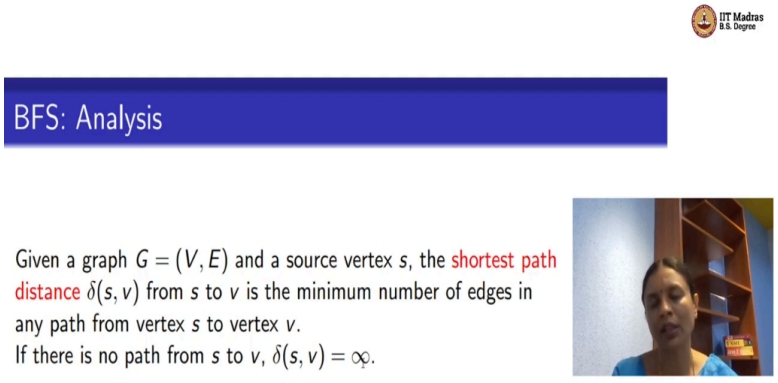
So, if a vertex is white, it is yet to be discovered by breadth first search algorithm and it is not yet put into the queue. When it is discovered but it is not yet colored black, it is colored blue and put into the queue. Eventually, when its adjacency list is explored, its color will become black and it is removed from the queue. So, this is the queue that we maintained through this example. So, this is the example that I am trying to illustrate for you. I hope this is clear.

(Refer Slide Time: 20:38)



So, now, some analysis, you can ignore this if it is not needed because it is not a course on algorithms, we may not need this fully. But the basically this slide tells you that the running time of breadth first search is linear. Linear in the size of the graph which is the count of the number of vertices and the number of edges.

(Refer Slide Time: 20:58)

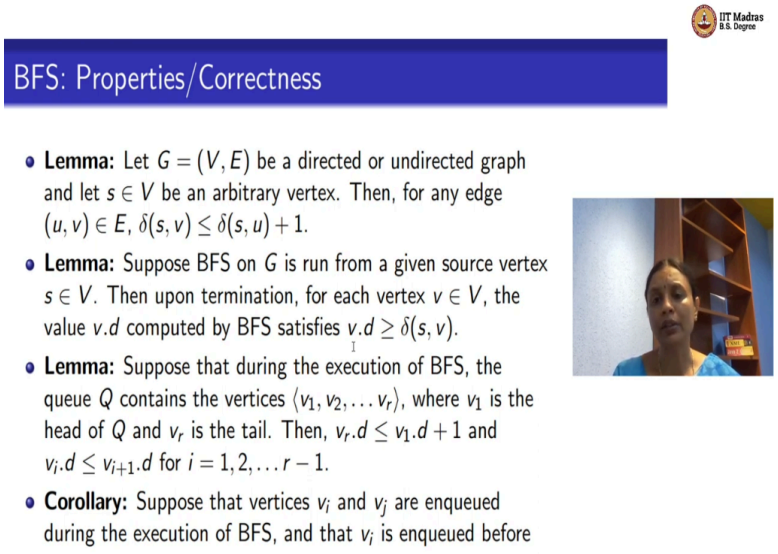


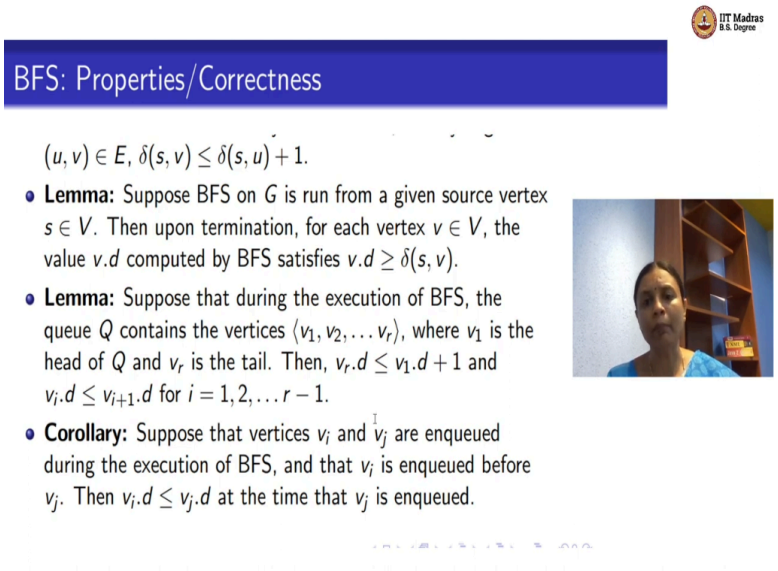
The other nice thing is BFS also produces the shortest paths. So, let us understand what is shortest path. Given a graph G which has vertices and edges and a source vertex s. The shortest path distance delta of s v, from s to a particular vertex v is basically the path that has

as minimum and number of edges as possible. These graphs do not have weights in the simplest additions.

If we have weights then we say it is the least weighted path. But in the absence of weights on the edges of graphs, we just count the number of edges and say that it is a minimum number of edges. So, if there is no path from s to v, remember BFS algorithm originally initialized the vertex to be infinity. So, the shortest path stays at infinity. We indicating that there is no path.

(Refer Slide Time: 21:47)

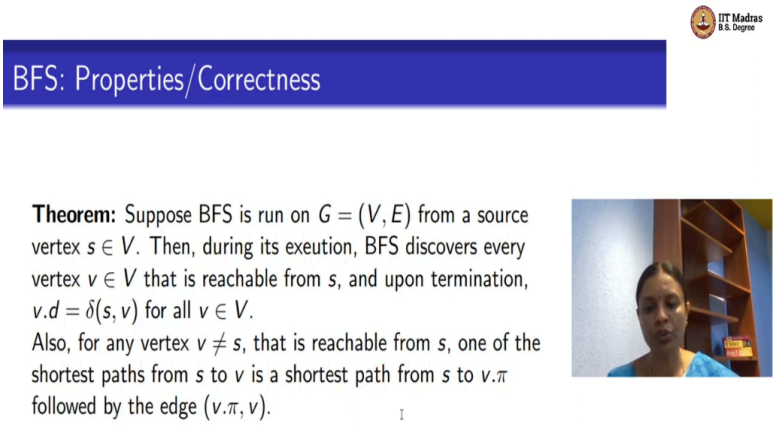




There are some properties that tell you why BFS returns, correctly explores all vertices that are reachable from s, and returns the shortest paths. I have put them in this slide for the sake of completeness. If you would like to learn more about why BFS correctly explores all

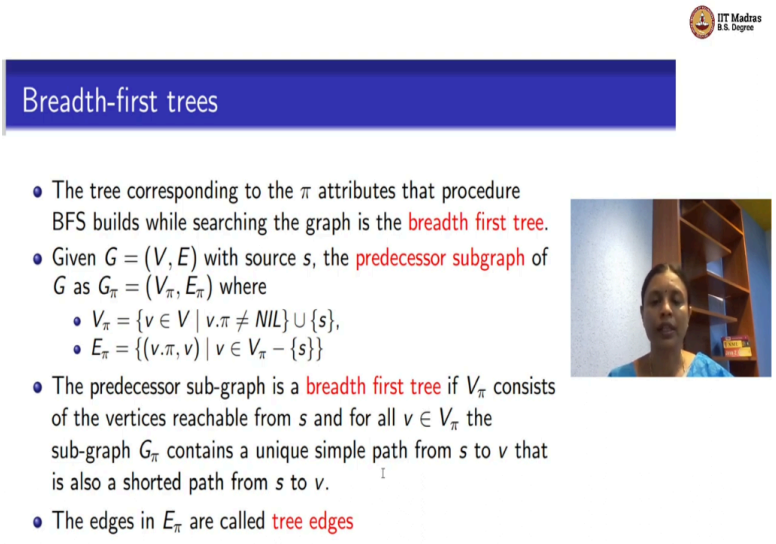
vertices that are reachable from s and returns the shortest paths, you could contact me offline and I will be able to tell you. Otherwise these lemmas need basically put together show that BFS works correctly as a procedure. For it to work correctly, what does it mean?

(Refer Slide Time: 22:22)



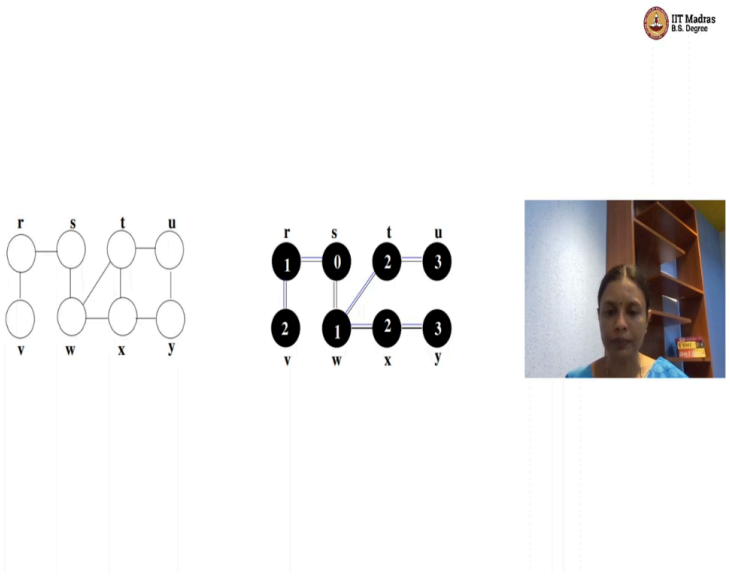
That is what this theorem says. It says suppose, I take a graph and run BFS from a distinct source vertex s then as it is executing, BFS finds every vertex that is reachable from s. And when it is terminating, it will return the shortest path to each vertex from the source. If a vertex is not reachable from s then you do not get the shortest paths.

(Refer Slide Time: 22:47)



So, if I take the tree of shortest paths, it will correspond to the breadth-first tree and those edges are what we call tree edges.

(Refer Slide Time: 22:58)



So, if you take this example, the same example that we had earlier I have now removed the distance matrix that we started out with but the graph is the same. s was the source vertex that we started out with and this is how the final output of BFS looks like. All vertices are colored black.

Initially, all vertices were colored white, s was color blue and then subsequently vertices became colored blue and then black. I am omitting all the steps that were shown in the illustration. On the right, you have this final vertex. All the vertices are colored black and the vertices have some numbers associated to them. What do they count?

They count the length of the shortest path. So, if you see shortest path from s to itself is 0, length of the shortest path, the vertex r is reachable from s by just this one edge. So, the length of the shortest path is 1. Then I can go from s to r to v that the length of the shortest path from s to v is 2 that is s to r to v. So, that is given here. And all the other edges go away.

If you see for example this edge from t to x did not correspond to the shortest path. So, it is not there as a part of the best breadth first tree. Similarly, the edge from u to y is not there as a part of the breadth first tree, these double colored edges, blue and black are basically the original edges of the graph that are a part of the shortest path tree.

I keep these attributes as a part of my algorithm. I keep a color attribute to help my algorithm work through coloring a vertex, putting it into the queue and removing it from the queue which basically indicates whose adjacency list I am exploring. Then I also keep a d attribute which populates this number to be infinity for all other vertices and 0 for s to start with. And then the d gets added and finally becomes this number.

Basically, a count of edges from the source, that edges that come along the shortest path. So, this is the final BFS output. You can get the BFS tree of shortest paths. You can get how many edges far away is a particular vertex from the source. And in this graph, all the vertices happen to be reachable from s. So, there are no infinite d attribute vertices otherwise it is all the same. So, that was a short module on BFS. In the next lecture you will learn about the other traversal algorithm Depth First Search. I will stop here for now.