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

Lecture - 07

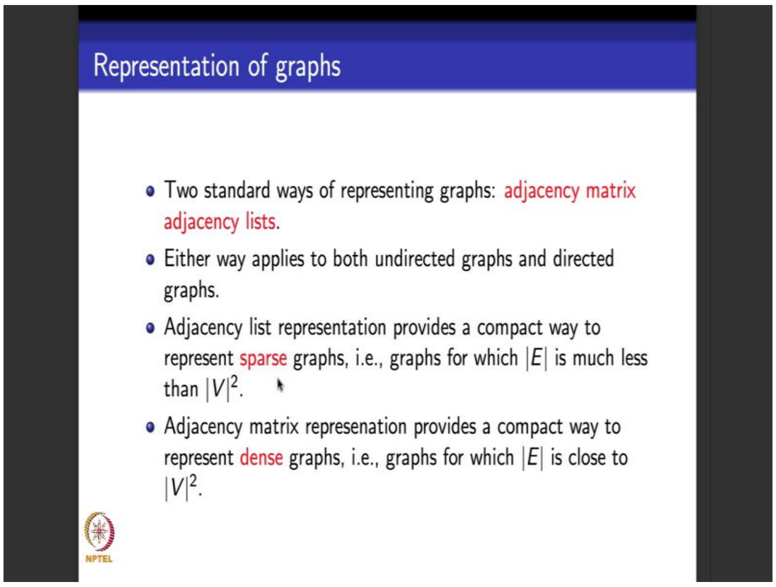
Elementary Graph Algorithms

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 note coverage, h 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 for 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 deals 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)



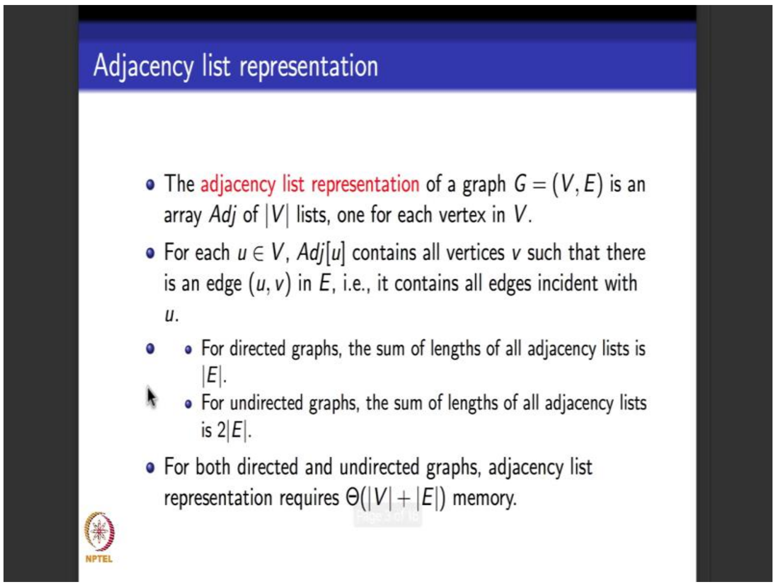
Representation of graphs

- Two standard ways of representing graphs: adjacency matrix adjacency lists.
- Either way applies to both undirected graphs and directed graphs.
- Adjacency list representation provides a compact way to represent sparse graphs, i.e., graphs for which $|E|$ is much less than $|V|^2$.
- Adjacency matrix represenation provides a compact way to represent dense graphs, i.e., graphs for which $|E|$ is close to $|V|^2$.

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 lists and adjacency matrix. Both undirected and directed graphs can be represented both as an adjacency lists 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 from 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)



Adjacency list representation

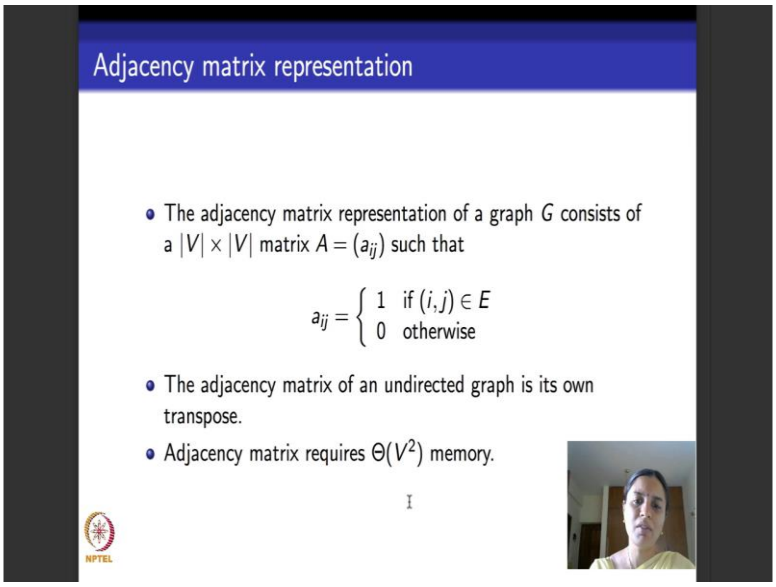
- The adjacency list representation of a graph $G = (V, E)$ is an array Adj of $|V|$ lists, one for each vertex in V.
- For each $u \in V$, $Adj[u]$ contains all vertices v such that there is an edge (u, v) in E, i.e., it contains all edges incident with u.
-
  - For directed graphs, the sum of lengths of all adjacency lists is $|E|$.
  - For undirected graphs, the sum of lengths of all adjacency lists is $2|E|$.
- For both directed and undirected graphs, adjacency list representation requires $\Theta(|V| + |E|)$ memory.

So, given a graph with vertex at V and at set E, what is an adjacency lists. 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 adjunct 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 lists? 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)



Adjacency matrix representation

- The adjacency matrix representation of a graph G consists of a $|V| \times |V|$ matrix $A = (a_{ij})$ such that

$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

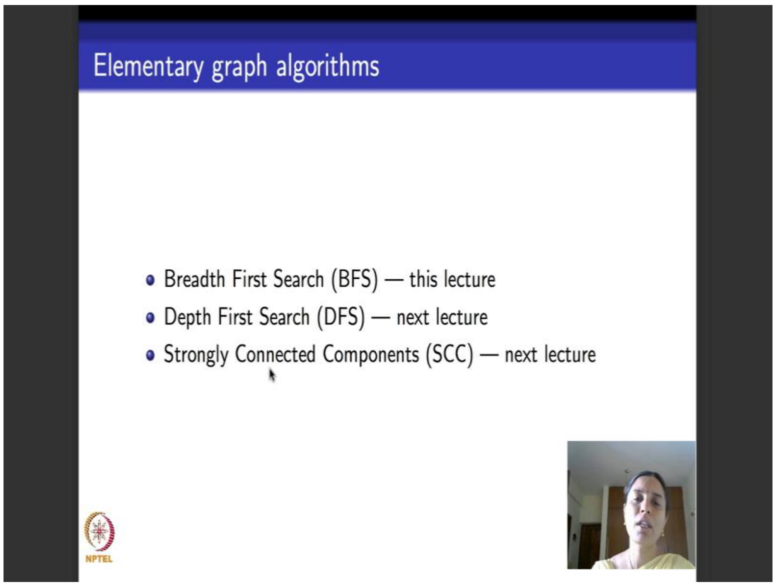
- The adjacency matrix of an undirected graph is its own transpose.
- Adjacency matrix requires $\Theta(V^2)$ memory.

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 certain edges of the graph otherwise it contains a 0. So, if you see adjacency matrix will go directed and undirected graphs will need V square theta V square 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 adjacent 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 square. 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)

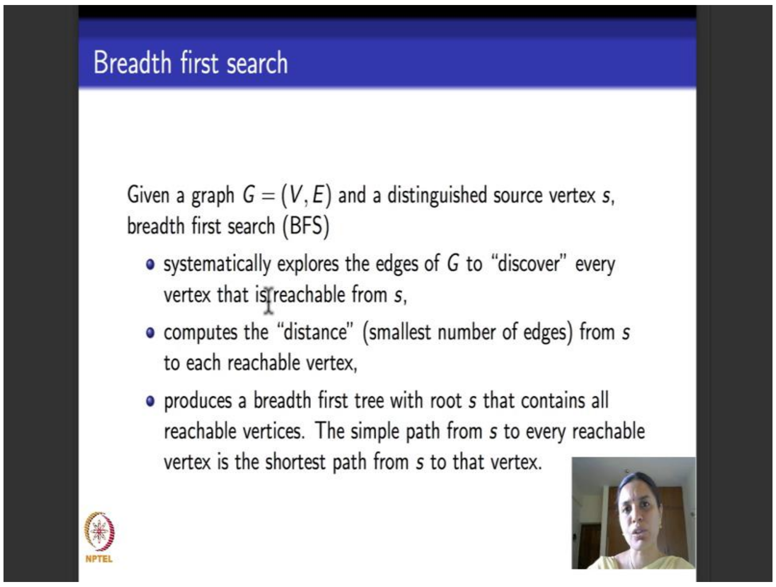


Elementary graph algorithms

- Breadth First Search (BFS) — this lecture
- Depth First Search (DFS) — next lecture
- Strongly Connected Components (SCC) — next lecture

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)



Breadth first search

Given a graph $G = (V, E)$ and a distinguished source vertex s, breadth first search (BFS)

- systematically explores the edges of G to "discover" every vertex that is reachable from s,
- computes the "distance" (smallest number of edges) from s to each reachable vertex,
- produces a breadth first tree with root s that contains all reachable vertices. The simple path from s to every reachable vertex is the shortest path from s to that vertex.

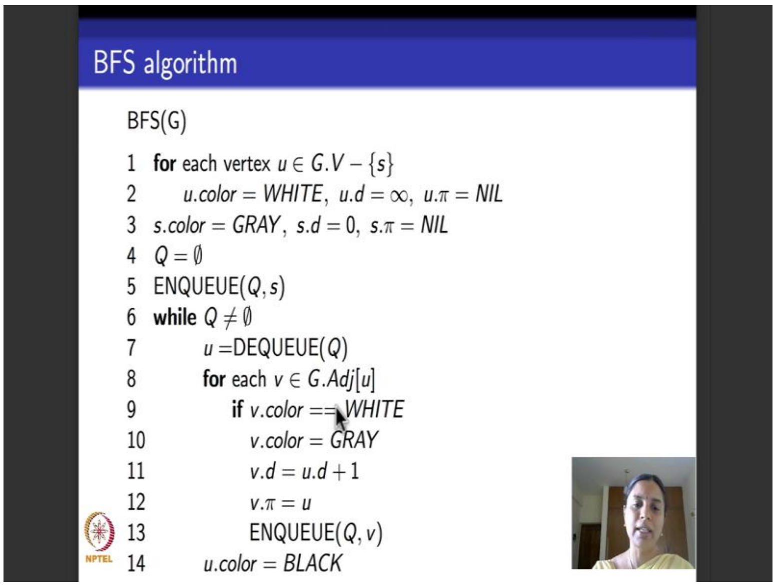
So, we will move on and start with breadth first search. What is breadth first search do? As the name says it searches through a graph, and how does it search through a graph? It searches through a graph in a breadth first way. That is it starts it searches from a particular vertex, let us say call it 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 finish 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 and 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)

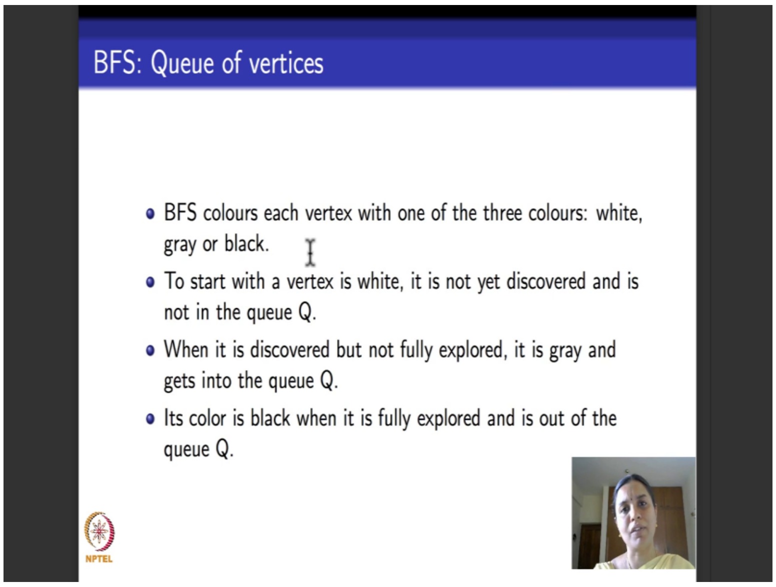


BFS algorithm

```
BFS(G)
1  for each vertex $u \in G.V - \{s\}$
2      $u.color = WHITE$, $u.d = \infty$, $u.\pi = NIL$
3  $s.color = GRAY$, $s.d = 0$, $s.\pi = NIL$
4  $Q = \emptyset$
5  ENQUEUE($Q, s$)
6  while $Q \neq \emptyset$
7      $u$ =DEQUEUE($Q$)
8      for each $v \in G.Adj[u]$
9          if $v.color == WHITE$
10             $v.color = GRAY$
11             $v.d = u.d + 1$
12             $v.\pi = u$
13             ENQUEUE($Q, v$)
14     $u.color = BLACK$
```

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)



BFS: Queue of vertices

- BFS colours each vertex with one of the three colours: white, gray or black.
- To start with a vertex is white, it is not yet discovered and is not in the queue Q.
- When it is discovered but not fully explored, it is gray and gets into the queue Q.
- Its color is black when it is fully explored and is out of the queue Q.

So, it takes the graph and the main job that breadth first search does is to be able to cover 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 of 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 vertices 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 vertices 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 if the pseudo code for the breadth first search. So for each vertex that is not the source remembers s is the source for which breadth first search algorithm start. So, for each vertex (Refer Time: 11:26) 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.

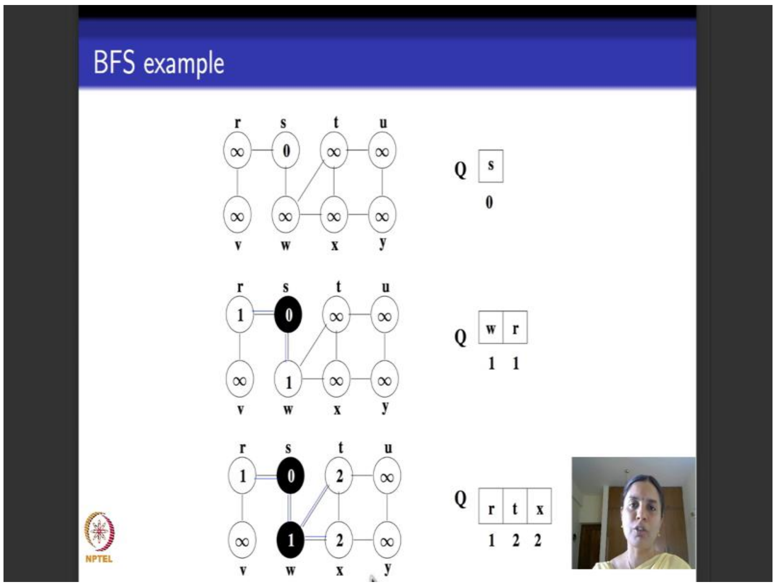
So, to start with breadth first search algorithm colors each vertex white such its distance to be infinity, because we the distance will reduce right to start with I do not know how far it is from the source. And if the tree is not yet developed so the breadth first search algorithm is just began, so its pi attribute is NIL. Now it begins its search from the source vertex s. So, the first thing that gets into the Q; Q is the Q (Refer Time: 12:52) it maintains. The first vertex that gets into the Q is s. So, if you see this enqueue Q s puts s inside the Q 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 Q.

So, as long as there are vertices in the Q 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 (Refer Time: 13:29) line number a 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 in 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 get here. 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; this is 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 Q color with 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 do 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)


BFS example

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 whenever the graph is directed or undirected.

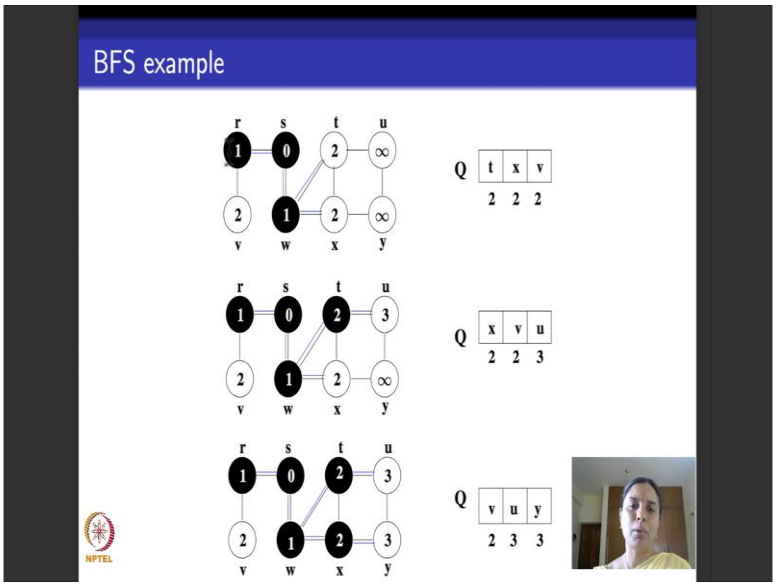
So, there are eight vertices in the graph; the vertex s is source vertex. So, what are these various things? The other vertices are r s t u v w x y. So, I have also put these parameters inside the vertex, what they correspond to is the d attributes. You see the distance from s the source is itself 0 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 Q. 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 nod s with its successor as r and a other 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 in pi attributes. So, I color this as blue indicating that the predecessor of t is w and then I color this h 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)

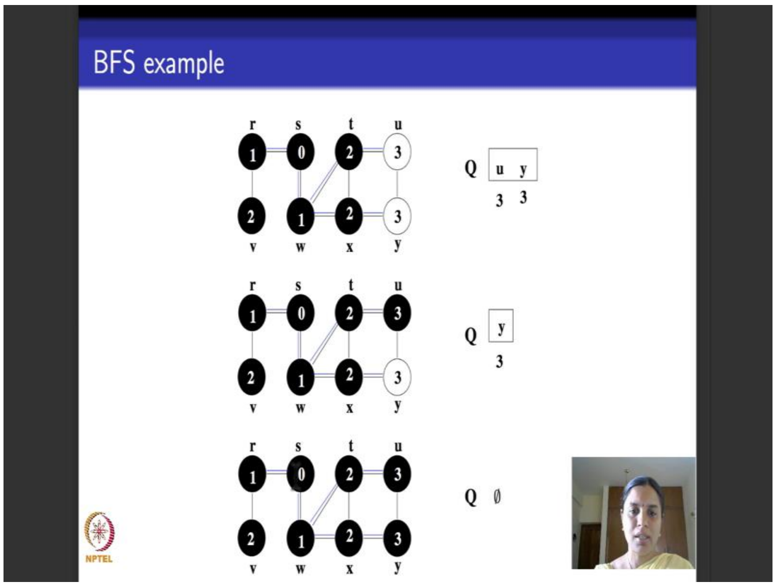


BFS example

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 s 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 c 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)

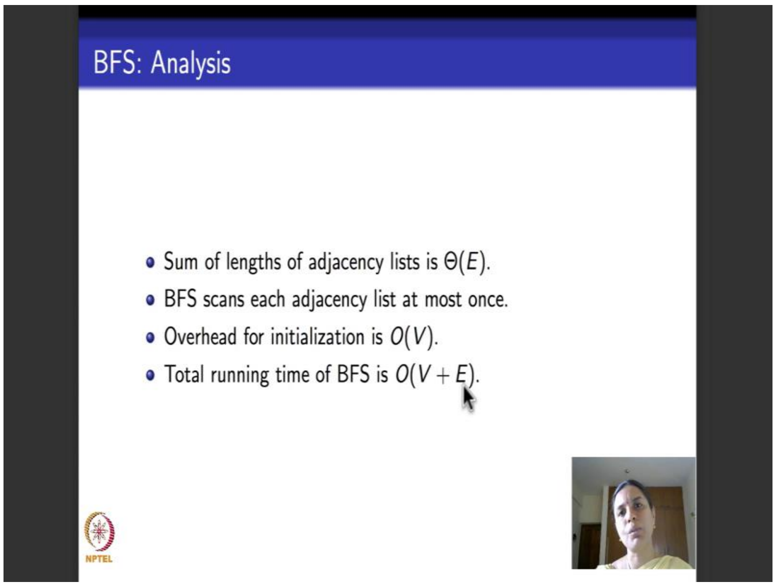

BFS example

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 x 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)



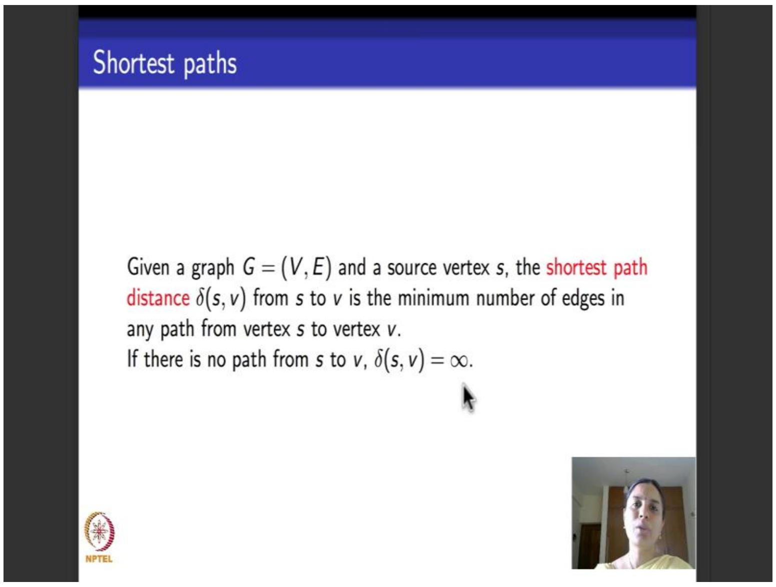
BFS: Analysis

- Sum of lengths of adjacency lists is $\Theta(E)$.
- BFS scans each adjacency list at most once.
- Overhead for initialization is $O(V)$.
- Total running time of BFS is $O(V + E)$.

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 long 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)



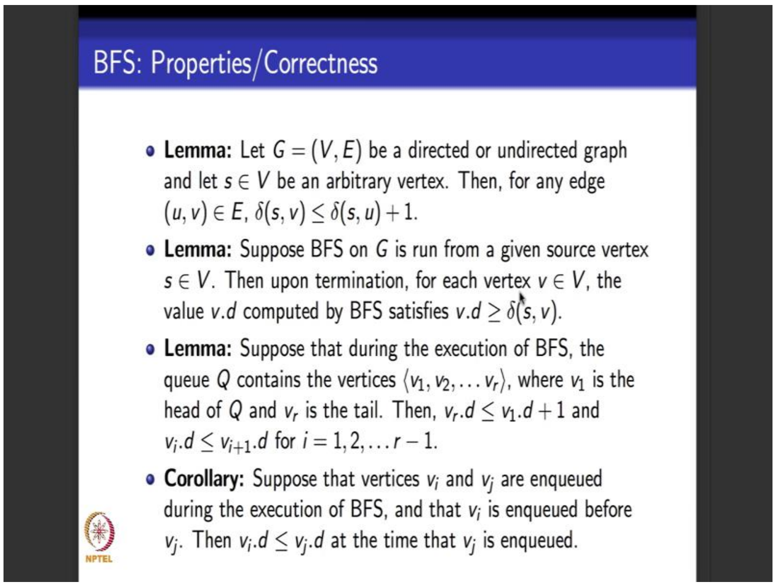
Shortest paths

Given a graph $G = (V, E)$ and a source vertex s, the shortest path distance $\delta(s, v)$ from s to v is the minimum number of edges in any path from vertex s to vertex v.
If there is no path from s to v, $\delta(s, v) = \infty$.

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; right 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 like book called Introduction to Algorithms by Cormen Leiserson; CNRS Cormen Leiserson (Refer Time: 22:03) outputs is 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 encounter.

(Refer Slide Time: 22:26)

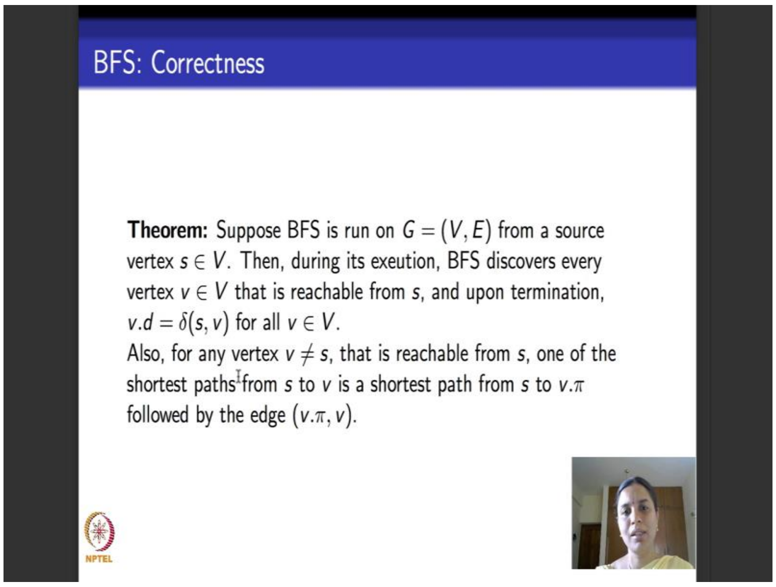


BFS: Properties/Correctness

- **Lemma:** Let $G = (V, E)$ be a directed or undirected graph and let $s \in V$ be an arbitrary vertex. Then, for any edge $(u, v) \in E$, $\delta(s, v) \leq \delta(s, u) + 1$.
- **Lemma:** Suppose BFS on G is run from a given source vertex $s \in V$. Then upon termination, for each vertex $v \in V$, the value $v.d$ computed by BFS satisfies $v.d \geq \delta(s, v)$.
- **Lemma:** Suppose that during the execution of BFS, the queue Q contains the vertices $\langle v_1, v_2, \dots v_r \rangle$, where v_1 is the head of Q and v_r is the tail. Then, $v_r.d \leq v_1.d + 1$ and $v_i.d \leq v_{i+1}.d$ for $i = 1, 2, \dots r - 1$.
- **Corollary:** Suppose that vertices v_i and v_j are enqueued during the execution of BFS, and that v_i is enqueued before v_j. Then $v_i.d \leq v_j.d$ at the time that v_j is enqueued.

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

(Refer Slide Time: 22:32)



BFS: Correctness

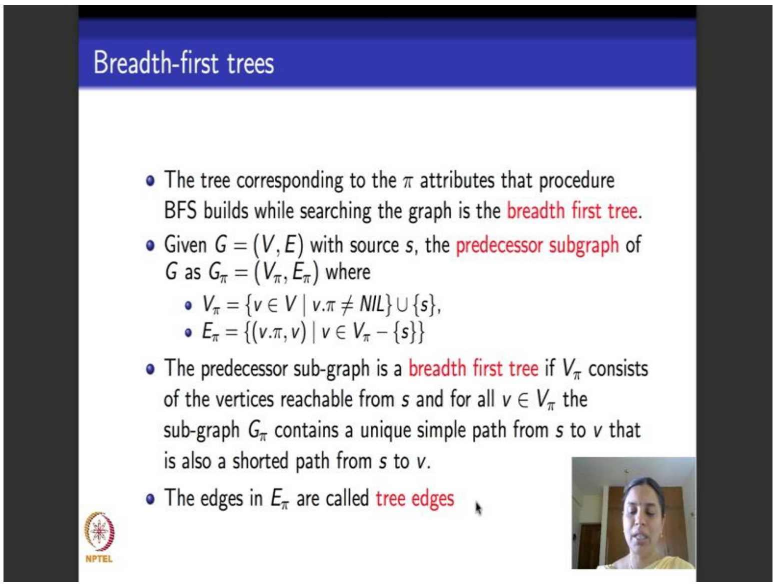
Theorem: Suppose BFS is run on $G = (V, E)$ from a source vertex $s \in V$. Then, during its exeution, BFS discovers every vertex $v \in V$ that is reachable from s, and upon termination, $v.d = \delta(s, v)$ for all $v \in V$.
Also, for any vertex $v \neq s$, that is reachable from s, one of the shortest paths from s to v is a shortest path from s to $v.\pi$ followed by the edge $(v.\pi, v)$.

So, we will state a unproven 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 correct. 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 nod s then you can explore the graph once again from other source and reach all the vertices that are 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 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)



Breadth-first trees

- The tree corresponding to the π attributes that procedure BFS builds while searching the graph is the breadth first tree.
- Given $G = (V, E)$ with source s, the predecessor subgraph of G as $G_\pi = (V_\pi, E_\pi)$ where
  - $V_\pi = \{v \in V \mid v.\pi \neq NIL\} \cup \{s\}$,
  - $E_\pi = \{(v.\pi, v) \mid v \in V_\pi - \{s\}\}$
- The predecessor sub-graph is a breadth first tree if V_π consists of the vertices reachable from s and for all $v \in V_\pi$ the sub-graph G_π contains a unique simple path from s to v that is also a shorted path from s to v.
- The edges in E_π are called tree edges

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 there 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 I adjacent the tree look like.

So, you can prove what theorem which says that such a predecessor of graph is actually a tree; this helps 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 the algorithm it is the edges stated the lemmas and not proved their correctness. Feel free to refer to a book like CNRS 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 about 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 h 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 course right in the beginning when I look at the source x 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 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 say. So, if the Q respects the distance enqueue and dequeue respects the distances of the vertices.

So, you send these lemmas and 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.