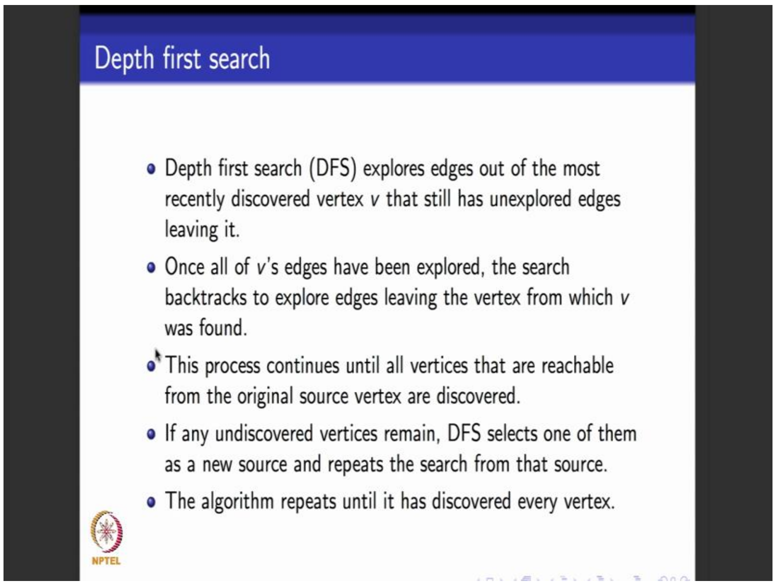
Software Testing
Prof. Meenakshi D’Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

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)



Depth first search

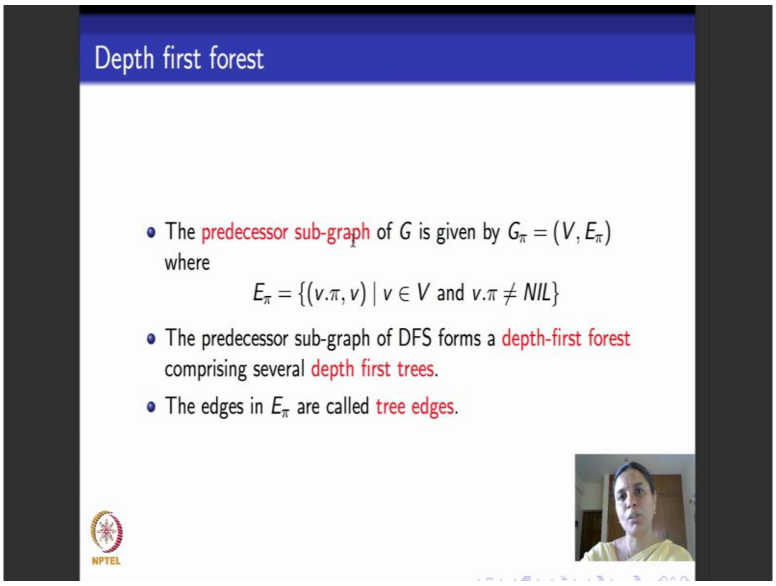
- Depth first search (DFS) explores edges out of the most recently discovered vertex *v* that still has unexplored edges leaving it.
- Once all of *v*’s edges have been explored, the search backtracks to explore edges leaving the vertex from which *v* was found.
- This process continues until all vertices that are reachable from the original source vertex are discovered.
- If any undiscovered vertices remain, DFS selects one of them as a new source and repeats the search from that source.
- The algorithm repeats until it has discovered every vertex.

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)

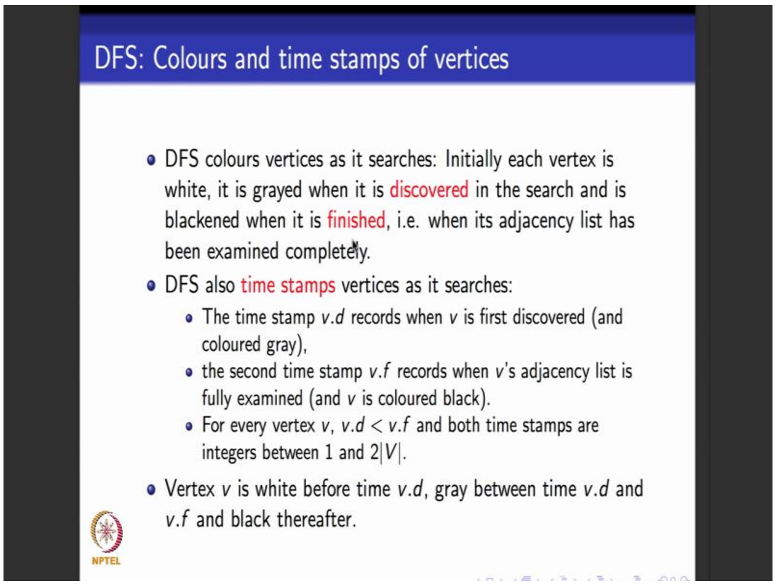


Depth first forest

- The predecessor sub-graph of G is given by $G_\pi = (V, E_\pi)$ where
$$E_\pi = \{(v.\pi, v) \mid v \in V \text{ and } v.\pi \neq NIL\}$$
- The predecessor sub-graph of DFS forms a depth-first forest comprising several depth first trees.
- The edges in E_π are called tree edges.

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)



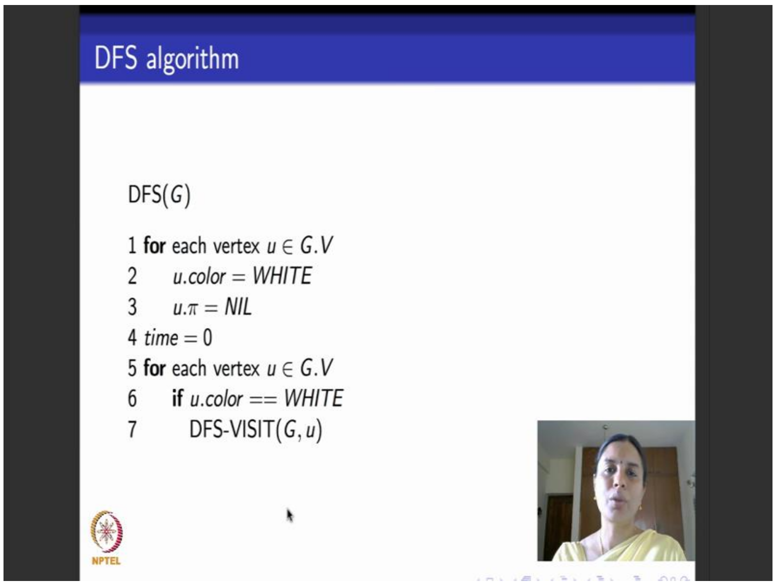
DFS: Colours and time stamps of vertices

- DFS colours vertices as it searches: Initially each vertex is white, it is grayed when it is discovered in the search and is blackened when it is finished, i.e. when its adjacency list has been examined completely.
- DFS also time stamps vertices as it searches:
  - The time stamp $v.d$ records when v is first discovered (and coloured gray),
  - the second time stamp $v.f$ records when v's adjacency list is fully examined (and v is coloured black).
  - For every vertex v, $v.d < v.f$ and both time stamps are integers between 1 and $2|V|$.
- Vertex v is white before time $v.d$, gray between time $v.d$ and $v.f$ and black thereafter.

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)



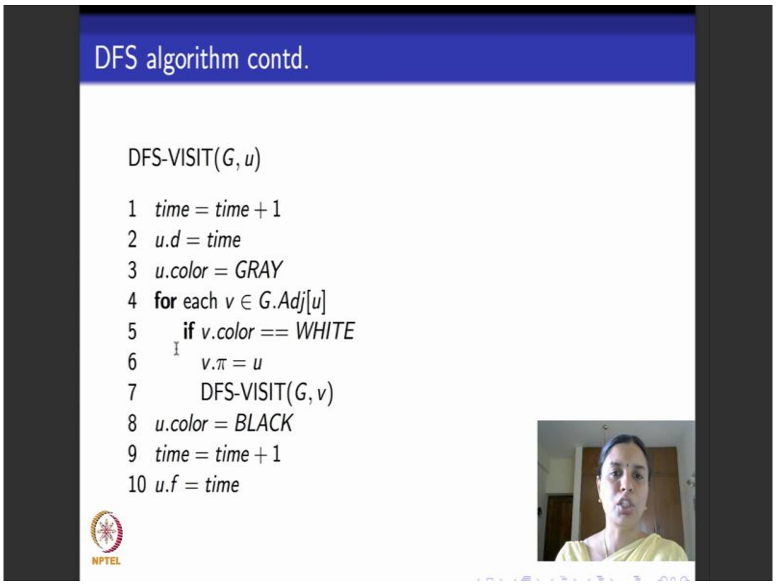
DFS algorithm

```
DFS(G)

1 for each vertex $u \in G.V$
2     $u.color = WHITE$
3     $u.\pi = NIL$
4 $time = 0$
5 for each vertex $u \in G.V$
6     if $u.color == WHITE$
7         DFS-VISIT$(G, u)$
```

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)



DFS algorithm contd.

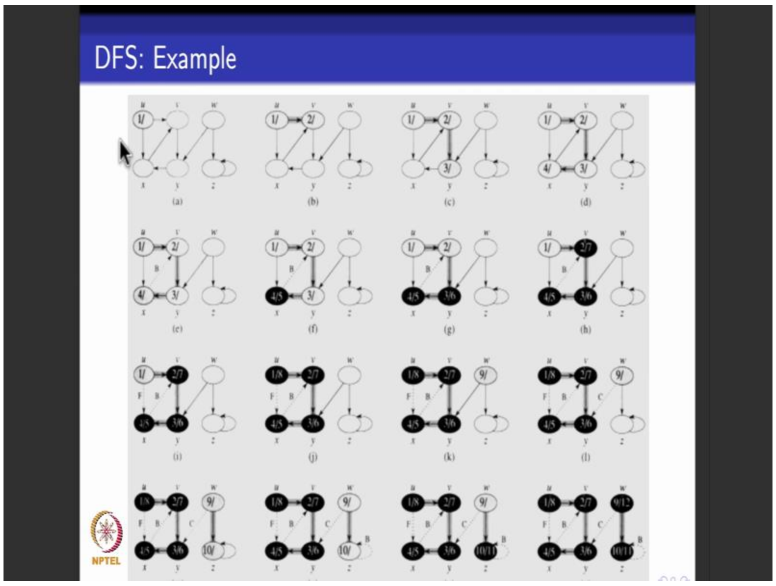
```
DFS-VISIT(G, u)

1  $time = time + 1$
2  $u.d = time$
3  $u.color = GRAY$
4  for each $v \in G.Adj[u]$
5      if $v.color == WHITE$
6          $v.\pi = u$
7          DFS-VISIT(G, v)
8  $u.color = BLACK$
9  $time = time + 1$
10 $u.f = time$
```

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)



DFS: Example

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.

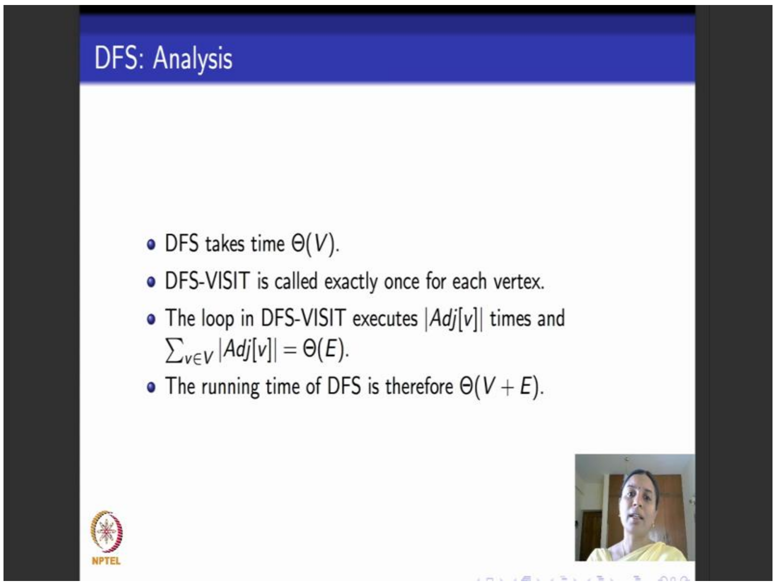
So, v which was originally colored white now becomes grey color. Again I have not depicted like in BFS, I have not depicted the color of the vertices here, the grey color is not when depicted just the black color is when depicted for clarity sake. So, I have discovered v, my timestamp is got incremented 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)



DFS: Analysis

- DFS takes time $\Theta(V)$.
- DFS-VISIT is called exactly once for each vertex.
- The loop in DFS-VISIT executes $|Adj[v]|$ times and $\sum_{v \in V} |Adj[v]| = \Theta(E)$.
- The running time of DFS is therefore $\Theta(V + E)$.

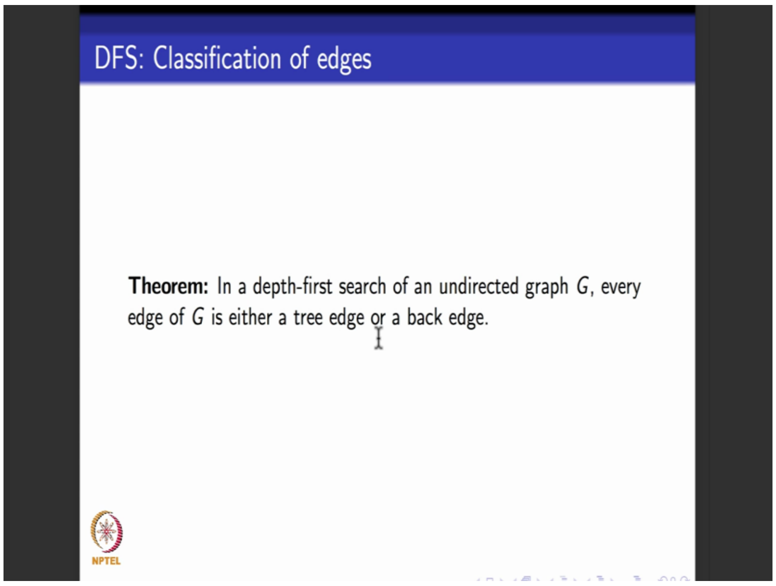
What is the running time of DFS? Let us go back and look at the code this for loop which does the initialization takes mod v time, and then the procedure DFS visit, how many times does it run? It runs at most once for every vertex that is reachable through an edge in the graph from the source. So, the 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)

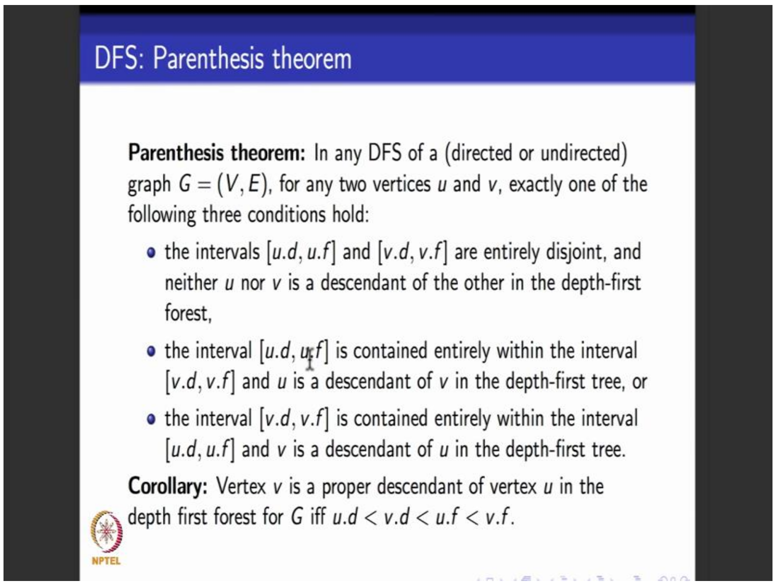


DFS: Classification of edges

Theorem: In a depth-first search of an undirected graph G, every edge of G is either a tree edge or a back edge.

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)



DFS: Parenthesis theorem

Parenthesis theorem: In any DFS of a (directed or undirected) graph $G = (V, E)$, for any two vertices u and v, exactly one of the following three conditions hold:

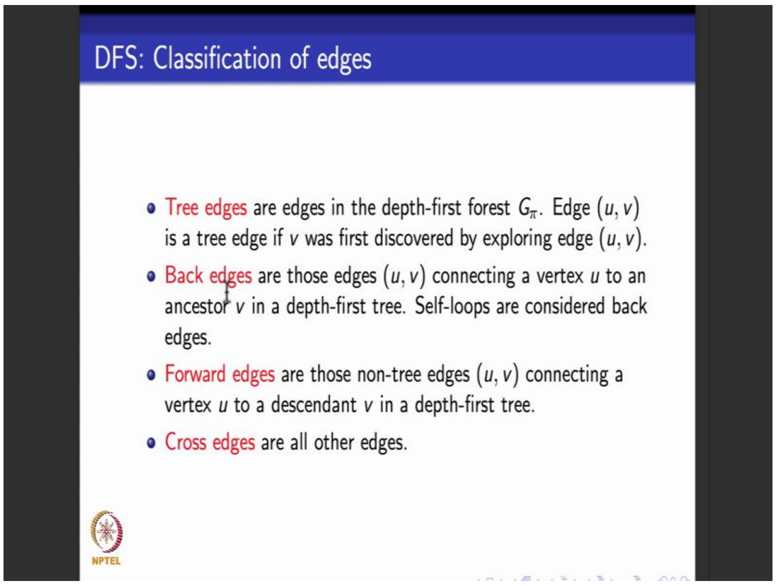
- the intervals $[u.d, u.f]$ and $[v.d, v.f]$ are entirely disjoint, and neither u nor v is a descendant of the other in the depth-first forest,
- the interval $[u.d, u.f]$ is contained entirely within the interval $[v.d, v.f]$ and u is a descendant of v in the depth-first tree, or
- the interval $[v.d, v.f]$ is contained entirely within the interval $[u.d, u.f]$ and v is a descendant of u in the depth-first tree.

Corollary: Vertex v is a proper descendant of vertex u in the depth first forest for G iff $u.d < v.d < u.f < v.f$.

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)



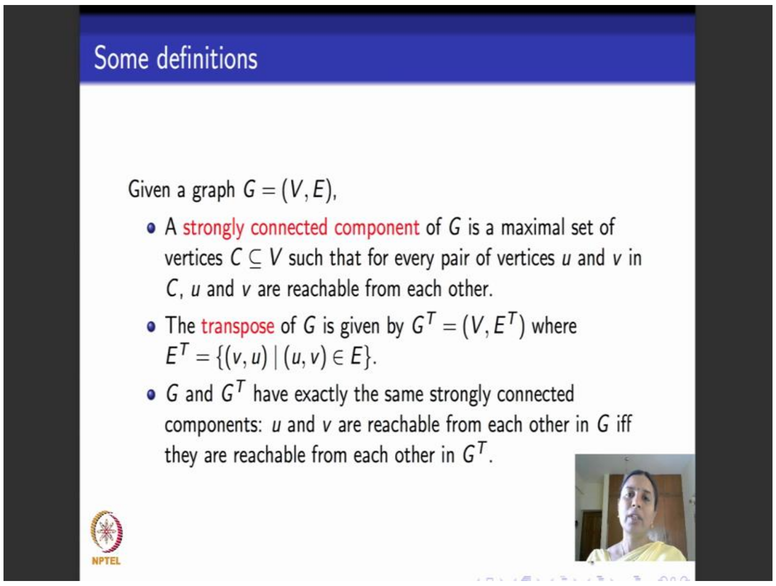
DFS: Classification of edges

- Tree edges are edges in the depth-first forest G_π. Edge (u, v) is a tree edge if v was first discovered by exploring edge (u, v).
- Back edges are those edges (u, v) connecting a vertex u to an ancestor v in a depth-first tree. Self-loops are considered back edges.
- Forward edges are those non-tree edges (u, v) connecting a vertex u to a descendant v in a depth-first tree.
- Cross edges are all other edges.

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)



Some definitions

Given a graph $G = (V, E)$,

- A strongly connected component of G is a maximal set of vertices $C \subseteq V$ such that for every pair of vertices u and v in C, u and v are reachable from each other.
- The transpose of G is given by $G^T = (V, E^T)$ where $E^T = \{(v,u) \mid (u,v) \in E\}$.
- G and G^T have exactly the same strongly connected components: u and v are reachable from each other in G iff they are reachable from each other in G^T.

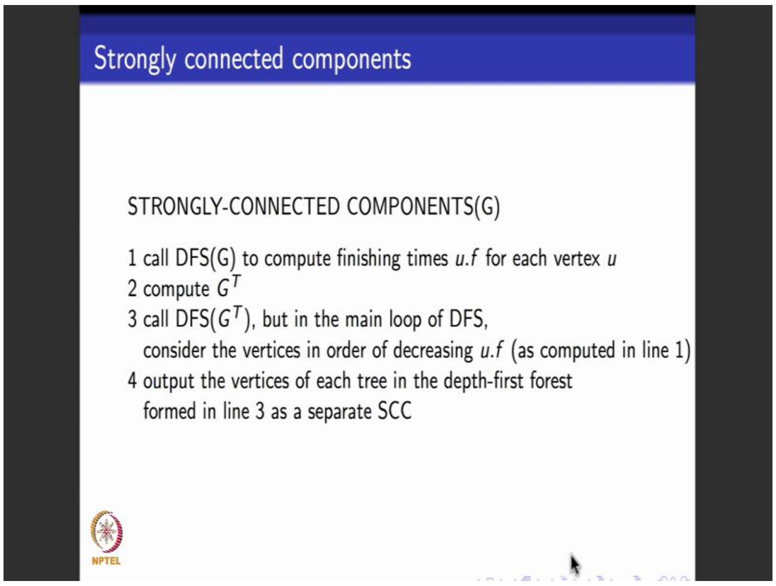
Now, the other thing that I wanted to do is to tell you how to use DFS to output what are called strongly connected components in a graph.

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)



Strongly connected components

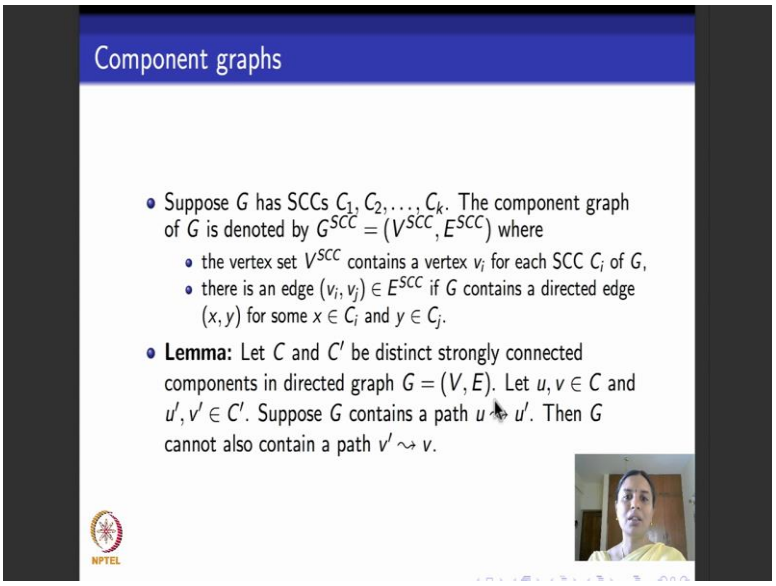
```
STRONGLY-CONNECTED COMPONENTS(G)

1 call DFS(G) to compute finishing times $u.f$ for each vertex $u$
2 compute $G^T$
3 call DFS($G^T$), but in the main loop of DFS,
    consider the vertices in order of decreasing $u.f$ (as computed in line 1)
4 output the vertices of each tree in the depth-first forest
    formed in line 3 as a separate SCC
```

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)



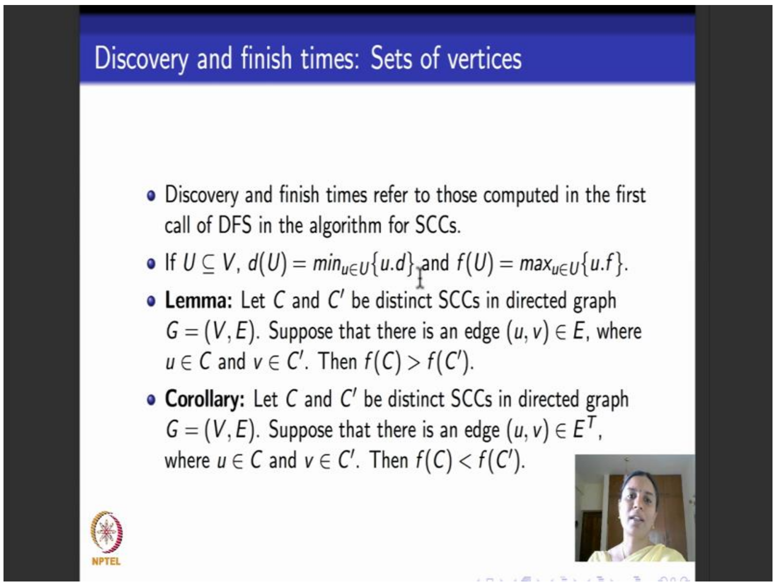
Component graphs

- Suppose G has SCCs $C_1, C_2, \ldots, C_k$. The component graph of G is denoted by $G^{SCC} = (V^{SCC}, E^{SCC})$ where
  - the vertex set V^{SCC} contains a vertex v_i for each SCC C_i of G,
  - there is an edge $(v_i, v_j) \in E^{SCC}$ if G contains a directed edge (x, y) for some $x \in C_i$ and $y \in C_j$.
- **Lemma:** Let C and C' be distinct strongly connected components in directed graph $G = (V, E)$. Let $u, v \in C$ and $u', v' \in C'$. Suppose G contains a path $u \leadsto u'$. Then G cannot also contain a path $v' \leadsto v$.

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)



Discovery and finish times: Sets of vertices

- Discovery and finish times refer to those computed in the first call of DFS in the algorithm for SCCs.
- If $U \subseteq V$, $d(U) = min_{u \in U}\{u.d\}$ and $f(U) = max_{u \in U}\{u.f\}$.
- **Lemma:** Let C and C' be distinct SCCs in directed graph $G = (V, E)$. Suppose that there is an edge $(u, v) \in E$, where $u \in C$ and $v \in C'$. Then $f(C) > f(C')$.
- **Corollary:** Let C and C' be distinct SCCs in directed graph $G = (V, E)$. Suppose that there is an edge $(u, v) \in E^T$, where $u \in C$ and $v \in C'$. Then $f(C) < f(C')$.

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.