Lecture Transcript
Graph Traversal Algorithm (DFS)

Hello and welcome to this next lecture on 'Data Structures and Algorithms'. We'll continue our discussion on graph traversal algorithms. We'll consider the next one, the Depth First Search. Contrasting Depth First Search with Breadth First Search the main difference is that a node and the subtree corresponding to that node is completely explored before other siblings of the node are explored. So like before, we will adapt a legend for nodes different labels that will be assigned to nodes by default every node is unexplored and in fact, every edge is also unexplored. So that is initialization. Every node and every edge is labelled and unexplored and then for each corrected component in the graph, you find a vertex and perform a DFS. So, DFS for a 'v' belonging to each connected component. Now, what is this DFS itself going to do? Like before, DFS is going to assign labels then labels node labels we are looking at now are visit. The edge labels like before are Discovery. There is a slight difference now, BACK edges are labelled 'red' as against cross edges that we encountered in the case of Breadth First Search. The back edge is basically an edge to an ancestor in the tree as against to a node which is a sibling or a descendent of a sibling.

Now this back edges become very handy when it comes to detecting cycles in the graph. So the DFS, the DFS iterates over every node in the graph and in fact, to begin with all the edges incident on a node. I am going to illustrate this now. So here we begin with some nodes 1, 2, 3, 4, 5. We begin with 'v' that is a first node. Look at all the edges incident on that first node, you get a 'w' that is 2. Look at the label of 'w' it is all unexplored, in fact, all the node in black are now unexplored. If so then set the label of 'e' this particular edge to Discovery edge and also perform a DFS on the node labelled 2. What does this DFS in turn do? Well, its going to look at all the neighbours of 2, it's just 1, start with the first one. In fact, before starting of with that edge you would label 2 itself as discovered. So node is labelled as discovered before any subsequent DFS activities performed. So you will do likewise for 4 perform a DFS and you come across 3. You try and perform a DFS but you find that this w, 1 at least in first node that 3 is connected to happens to be already explored, so that is when you decide to set the label of 'e' to a 'BACK', this is a back edge. What it means is that we have come across a cycle. Of course, now there is no more DFS to perform on 1, so you iterate over all the other edges of 3, you have one more edge which you can mark as 'DISCOVERY' edge 'DISCOVERED' edge and then 5. What happens when you invoke DFS of 5 is that you don't have any edges incident on 5. So, now you start going backward to 3 find 3 has no other edges, 4 has no other edges nor this 2 have and so on.

Now, this is a very simple version of DFS which makes use of a recursive call. So, we are going to make this DFS procedure somewhat smarter. The first thing we will do towards that is eliminate recursive call. And we will basically do that by making use of the stack, auxiliary data structure.

How would you do that? Well, we're going to move the part of DFS, simulated by having w inserted into a stack. So, we're going to defer the DFS call here, in fact, shift the DFS call to the main DFS itself. I might just do well by putting a cross here and for this particular w, I am going to insert a push w into a stack S. We'll assume that S was initialized to be empty. And, you are going to do this for all the 'e' edges incident on v. What does this mean? We are saying that if this was v, it had multiple edges and e1, w1, e2, w2 and so on. We're going to push each of w1, w2 and so on into the stack and once we have exhausted all the incident edges for v, which is at this point of time, we're going to say pop elements from the stack. So, what we need to do is as follows. We'll need to initially insert the element v into this stack and add a while, a while block, while(S.notempty()) i.e., while S has elements to pop, we're going to pop the latest element v, we might want to call the start node v prime and this will be a specific v.

So, v is S.pop(), popped out the element, perform all these activities on the incident edges for v, get all the 'w's, push all those 'w's into S and continue this whole exercise till S becomes emptied. So, this is going to give you the same effect as the original DFS called without needing to invoke recursion. We can also make some other interesting modifications to this procedure by doing some more book-keeping. So, recall that for BFS, the book-keeping we introduced was that of depth an ancestor. So, what we'll do is like in BFS, introduce some additional book-keeping. So, what book-keeping are we talking about? Well, first of all, we can keep track of the vertex that lead to an expansion of the current vertex. So, what I mean by the keeping track for expansion is as follows. So, as I push w into S, I'll also set the pi(w), the ancestor of w to be v. Initially, of course, the ancestor of v prime is nil, it's a root. But, I will set the pi(w) to be v. I can also keep track not just of depth, but of the time that a particular node started getting expanded, and the time that we stopped expanding a particular node. So, we start expanding a particular node, moment we pop that node.

We can also do some additional book-keeping in terms of when a particular node starts getting expanded, when the subtree for that node v gets expanded, beginning of that and the beginning of that basically b[v], we can set it to some time, this time, is initially 0 and it increases the moment I start expanding a node, time++. However, it's a little tricky to keep track of when expansion on v concludes because the expansion on v concludes when the expansion of all its descendants is also concluded. So, this expansion becomes very natural when you have the DFS call. So, what I am going to do is state the following. e[v] is time++, if recursion is used, and what this means is that the call DFS(G, w) were used instead of the stack. e[v] could also be set in the case of the stack being used instead of the recursive call, but I am going to leave that as homework. Your homework will be how would you set e[v], the end time of expansion of subtree for node v, if stack were used instead of recursion.

So, some of this book-keeping might become handy in subsequent discussions. Let's now look at the complexity of DFS. We are going to illustrate DFS through a slightly more interesting example. You have node 'd' where you begin with, and you explore this edge to visit 'a', explore the next edge from 'a' to visit 'b', from 'b' you visit 'c', 'e'. And now when you start looking at all the incident edges on 'e', well you will find a back edge that goes to a visited node 'b', well the other edge is also a back edge that goes to node 'd'. But there is a new unexplored edge, discover it and find 'f', visit 'f'. Well, from 'f' if you go to 'd', this is what depth first search is about, you visit 'd' from 'f' and not 'g' from 'e' and that gives you another back edge, a forward edge from 'f' to 'g' and again back edge from 'g' to 'e'. There is one more traversal left as you go trace back that is from 'b' to 'd', that again gives you a back edge. So we have marked a different discovery and back edges here. The properties of DFS, well it visits all the vertices and edges in the connected component Gv containing v. And as I pointed out earlier in the case of BFS, you are going to run your DFS on V1 or V2 for every connected component Gv1 or Gv2. And the edges labelled discovery by DFS in every such connected component forms a spanning tree Tv of Gv. So this gives you a spanning tree Tv1, spanning tree Tv2 and so on.

Some analysis of DFS. Each vertex is labelled twice that gives you order of V complexity. One is when you make it unexplored and the other is when you visit it and mark it as visited. We never visit a visited node, in fact, what we do is look at all the descendants of a visited node until they are completely explored. Each edge is also labelled twice, the first time mark it as unexplored, the other is either marked it as discovered edge or as a back edge. So this gives you basically order of V+E complexity. The G.incidentEdges is called once for each vertex u over all the incident edges that also gives you order E, so overall V+E. Applications, well, making use of all the book keeping and the implementation using a stack that we discussed. You could solve certain problems in order of V+E complexity. And these happened to be problems that are more natural with DFS than with BFS. So, of course, you could find the path from vertex 'v' to 'u' by invoking DFS(G,v). This you could keep track of by looking at pi[u] and then pi[pi(u)] and so on. Look at all the parents and thereafter all the ancestors. And you could also detect cycles, that's a very important contribution of DFS. What you need to do is invoke DFS(G,v) for any v and keep looking for a back edge.

So, existence of a back edge means there is a cycle. Where you find all the cycles? Well, if you look at the example that we showed, there are multiple DFS instances where back edges were found. So indeed you can enumerate several back edges by doing a DFS on the entire graph rooted at v instead of stopping at the first instance of a cycle. I leave it as a homework problem for you to determine if you can find all the cycles. DFS is also the classic strategy for exploring a maze, where basically you are doing backtracking search. So it make sense to do DFS instead of BFS because you are not interested in all the path that lead to the goal, but you are interested in one path and exit through that, and DFS is made for that.

Thank You.