Lecture Transcript
Graph Traversal Algorithm (BFS)

Hi and welcome to this next lecture on Data Structures and Algorithms. In this session, we will introduce you to graph traversal algorithms. We have already discussed graphs and different data structure representations for graphs. How do you traverse such graphs? How do you enumerate the vertices and edges of these graphs in a non-redundant and useful manner? Today, we will discuss one such useful algorithm graph traversal which is BFS or Breadth First Search. So, there are two very well-known techniques for graph traversal and you begin at some source node 's'. What we'll discuss today breadth first search as a name says, it discovers all vertices at a particular distance 'd' from 's' before discovering any vertices at distance d + 1. So, the idea of breadth first search is to discover nodes in the graph slice by slice. So, this is node at depth 1 or depth 0, this is a node at depth 1, these are nodes at depth 2 and so on. On the other hand, breadth first search searches deeper into the graph wherever possible. So, what does this translate to? Well, this means I am going to first traverse this path as far as possible, then this path, then this path and then so on. So, let's do more of breadth first search. So the principle idea in breadth first search is to explore the graph and classify nodes and edges.

This is also the idea behind depth first search, but the classification and the order in which the nodes are explored is different. So in breadth first search you keep track of unexplored nodes and edges to begin with. So, UNEXP stands for any node or edge that has not been explored so far. So, the starting point in BFS is traversed through all the vertices of the graph and for each vertex label it as unexplored, do likewise for very edge in the graph, label every edge as unexplored. And then you start with an unexplored vertex, do a breadth first search and having completed breadth first search go back to a next vertex that is yet unexplored. Now, BFS the call to BFS on vertex 'V' might itself land up marking many nodes and edges through BFS in 'G' as explored. So, we have different kinds of explored nodes and only after coming back from this call of BFS when we look at another vertex and check for it being explored or unexplored and find that it's actually unexplored would we actually explore that again. So, the principle idea here is that your graph might have several disconnected components. So, this is component 1, this is component 2 and what you are doing in the process of calling BFS is calling this node, marking nodes and then you will find that this node is still unexplored and call BFS here.

Now, what does the call to BFS itself do? Well, it marks the nodes within this graph and the other net edges within this graph as visited or discovered edges. So, let's do some highlighting. Visiting of a node or the visiting of an edge within the call to BFS, you would expect to mark these nodes and these edges as visited and discovered respectively. It is also possible that there are some edges across this calls to BFS. So it is possible that this particular node here is connected

to some other node here, well that we would call a cross edge. This is an edge to a sibling or to a child shared with sibling. So, more about this in the next slide, okay. So, this is BFS as applied to a single connected component. Okay. Let's illustrate this with an example. So, when we begin the BFS on vertex 'V', let's call this the vertex 'V'. You start with the new empty sequence S0. Now, S0 is going to denote the set of nodes to expand, nodes or vertices to expand the next. So, initially, you just have 'V'. So, S0 is V and let 'i' be 0, so S0, S1 is two, all of them denotes set of nodes or vertices to expand next. So, I am going to more generally replace S0 with Si, 'i' is initialized to 0.

Now, we're going to pick an element from Si which is 'V' and parallelly start building up Si + 1 which is the set of nodes or vertices to expand next. So, for each element 'v', we have only one element to begin with, you look at the edges that are incident on 'v'. So, we are discussing undirected graphs to being with. Now, you check the label of each of these nodes, nodes that are labelled black here or black colour are all actually unexplored, all these are unexplored. So, you look at 'e' which is unexplored, get the other corresponding vertex for this edge 'e'. Let's say this is 'w', if 'w' is unexplored which is the case. We're going to set the label of 'e' to discovered and thus set the label of 'w' to discover. Let's do precisely that, 'e' 'w' and in fact you're going to add each of these to Si+1 which is now S1. S1 will now contain this node, let's just give them numbers 1, 2, 3, 4 and 5. So, S1 will contain the node number 2. We need to similarly check for other unexplored nodes and vertices. So, this edge is unexplored, so is this vertex and we're going to add 3 to the list. Is there any other edge incident on V? Well, none. So this completes our S1. Now, we're going to explore the nodes in S1 next.

So S0 is kind of ignored. If you want to S1, iterate over all nodes here, so we go to node number 2. What do we do with node number 2? Well, it turns out that node number 2 has one incident edge with a node 3 which is already labeled. So if getLabel 3 is not unexplored, then you set the label of the edge to cross and that's what we'll do now. We will basically set a label of edge to cross and proceed. We look at the other edge and the corresponding node is unexplored, so this will mean both of them are marked as discovered and visited respectively. What do we do next? Well, we look at the neighbours of 4, we find that 3 is already visited. So mark it read, we can then move on to 3 and find that its neighbour is visited through a discovered edge. This is what you'll get at the end of the execution of this algorithm. So note that we have slightly different names for visited nodes and discovered edges because edges that are not discovered, if they get discover they lead to visited nodes but they can also lead to nodes that were visited earlier. So the idea of a blue discovery edge is that which leads to a yet as yet undiscovered node, whereas the meaning of a red or cross edge is that it leads to an already discovered node. So, what does a cross mean here? Well, basically a cross means it's an edge to a sibling or descendant of a sibling and that's precisely why the vertex at the other end of the cross edge turns out to be already visited. So, of course, you can complete and enumerate your S(sub)2 and S(sub)3 and so on.

Now do you really need to instantiate so many lists. Is it possible to manage with a fewer list, in fact, can we actually do with a single queue. It turns out that we can. And what I do now is present a simpler version of BFS. This will be a BFS with less book keeping and I explain what less book keeping means. One big difference here is we are going to manage with a single queue. So S is initialized to be empty and of course we immediately add v to it. So summarily it's the same as what we've seen before. The next thing we do is set the label. So the label of V will be

basically visited because we know that we visited it, but we're going to distinguish between visited nodes and expanded nodes. So yes, indeed we have visited it, we'll mark it so but I'll introduce one more colour do denote nodes that are already expanded. v has not yet been expanded, it has been added to the queue. So continuing we're going to now iterate on queue so while not S. is empty() do. We're going to now iterate over all the nodes which are adjacent to v. So this can be done by iterating over all edges. This is same as before and like before I'm going to say w is e.getothervertex(v). I am going to check the label as before again if getLabel(w) is unexplored then I am going to set the label now. I won't bother much about edges. I'll just set the label of w to that of a visited node and what does that mean, it means I'm going to add this node to the queue.

So let's do that, but before I do that I going to also update some arrays. So, I'm going to keep track of the depth of node from v, so depth of v is zero and I'll set the depth of w as depth of v+1. I'll also keep track of where we got to v from so pi (v) is nil. Whereas pi(w) is v and finally, as promised I am going to add my node w to the queue. So after having incremented the depth I am going to say S.append(w) having appended w to the queue and having iterated on all the edges. What I'll do next is mark that v has been already explored all the neighbors of v have been added to the queue and their depths been updated and so has been their ancestor. So this we do by introducing a new label and stating the following setLabel (v) not to visited but to explored. This was a label that we have to explicitly introduce because we are managing now with one queue. So this explored is a new label and what does is explored saying? So let the explicit state what explode means. Export for any vertex v the label of v is explored. Means all the neighbours of v have been either visited or explored. So an explored node cannot have an unexplored label. So this is also form of breadth first search but here we manage with the single queue with one more additional book keeping. Some of the new statements we have introduced that include the statement about the depth or the statement about the parent pi.

These will be useful when we make use of BFS for purposes such as finding shortest path from v. And in fact finding the distances as well for the shortest path. Of course, you could have done all these book keepings even in the original BFS algorithm we presented here. So I will leave that note and proceed so optional steps here are as follows you can set the d(v) to be 0 and pi (v) to be nil. You could do the same thing here. d(w) is d(v) + 1, pi(w) is v or e, if you want to keep track of the edge. Let's lie breadth first search to slightly more complex graph and this node here labeled in blue is a node we want to begin with so it is already visited. So you discover its edge and visit node a, discover another edge visit node b, node e and then finally f. Now the next node that you get from the queue basically you get it from S(sub)1 so note that a, b, e and f all belong to S(sub)1. So you basically dequeue a from S(sub)1 and look at its neighbour the only neighbour that is unexplored is b well d is already explored or visited through the discovered edge here but whereas this edge is not yet discovered. However the node b is already visited. So that is why it was important to assign labels to edges so as to avoid enumerating this discovered or blue edge. And restrict our attention to this undiscovered or that edge that goes to b.

Now we've labeled this edge as cross. Because b is already visited by another path. Continuing on well there is one more cross edge from b to e, another cross edge a new discovered edge, another cross edge, another cross edge. So there are exactly five cross edges. Now these edges could be very useful in determining a tree for the graph, a spanning tree for this graph so we know by

observation that these discovery edges they help you span all the nodes in this graph so the blue discovery edges form spanning tree. This is a spanning tree they could be other spanning trees and if you also do the optional step of keeping track of distances we note that the distance of [d] is 0, distance of a is 1 and so on. And let's consider a slightly far of node here, the distance of this node say g. Now distance of g was obtained from the distance of e that's how it was traversed so edge distance of e + 1 now distance of e, in turn, is distance of d + 1 and we realized that distance of d is 0 so distance of e is 1. And therefore substituting we get the distance of g to be 2. Note that you could have approached g via b and e which would have given your path length of 3. However, that is not the path that BFS gives us because the cross edge is never a part of BFS. Thus we note how BFS helps you find the shortest path to any node in the graph from the source vertex that is d. And in fact, you can find the shortest path as well. So, what is the shortest path to g? Well, the shortest path to g, can be obtained by looking at the ancestor.

The pi of g is e, the pi of e is d and d doesn't have any parent. So, look at the parent of g and then e and then d and then walk backwards. So, that gets you to d, e, g and has length d[g]=2. And in case you want to be more explicit, you could actually write down this path pi[e], pi[g]. So, if Gv is a connected component of a graph G containing a vertex v then BFS (G,v) starting at v visits all vertices and edges of Gv. It labels some edges as discovered and the others as crossed, for each edge labelled discovered, you put them together, you get the spanning tree Tv of Gv. And in case your G happens to have several disconnected components V1, V2, then TV1 union TV2 will give you the spanning forest of G. So, the tree you get from each of these is basically a subset of corresponding graph. Now for each vertex v in S, particular Si, so the Si was the sequence that we populated at different levels. The path from v to u in Tv has 'i' edges. So, the Si is basically corresponded to different slices in the same graph. So, S1, S2, S3 these were the slices in the single graph and what we are saying here, the path from v to u any particular u in Tv has i edges if u belongs to Si and every path from u to v in G(sub)v has least 'i' edges. What this means? Is that the path from u to v is in T(sub)v is a shortest path. This is because path from v to u which belongs to S(sub)i is the shortest path. Analysis, fairly straight forward. For most graph algorithms, we'll basically resort to some kind of collective analysis.

So, the kind of analysis, collective analysis we will talk about is called aggregate analysis. And the aggregate analysis is by looking at the aggregate statistics of number of accesses. So, each vertex is labelled twice. How? Initially, it's labelled that as an unexplored and the moment you visit a label, you label it as visit. In the alternative algorithm that we proposed, each label is labelled thrice. One as unexplored, once as visited and the third time as explored but it's a constant factor. Coming back to the initial algorithm, each edge is also labelled twice once as unexplored the other as discovered or as crossed so and edge is never labelled as both discovered and crossed it's either of them. And then each vertex is inserted once into Si for some 'i'. It cannot be inserted into multiple Si's, it will be because a vertex is inserted only when it migrates from being unexplored to visited, it happens exactly once. And you look at all the incident edges for u exactly once for each edge. This will mean in aggregate analysis that you call this incident edges summed over all vertices u are times the number of edges incident on u and this is nothing but the number of edges itself. So the total time is basically order of V which comes from the first three steps and four which is order of E. So total time is order of (V+E).

Now, BFS has several interesting applications and all of these applications are run in order of (V+E) time. The first is computing a spanning forest of G. We've already seen that this is obtained by a union of all the discovered edges. So, all you need to do is run through all the edges once and in fact you can run through them in the order and in which the vertices were explored to compute a spanning forest of G. Given two vertices of G find a path if there exists one in G between then with the minimum number of edges and this basically means if you are looking at two vertices u and v, then you just need to call BFS on G rooted at u and d[v] is length of that shortest path and the cascade of pi v is, pi of pi of v , pi of v and then v is exactly the shortest path. To compute the connected components of G, so recall that we iterated upon all the vertices v and for each vertex v that was yet unexplored, we invoked BFS. So, number of connected components is basically all the nodes v for which BFS rooted at v was invoked. You can also determine if G is a forest and then if not a forest find a simple cycle in G if there exists one. This is not very difficult if G is undirected, becomes messy with Breadth First Search if G is directed and I leave this as a homework problem but there are other ways to find detect cycles and existence of cycles in a graph. This we will discuss in the context of DFS (Depth First Search) which is a natural choice for finding a cycle. BFS is also very classic strategy for determining a solution to the rubix cube problem where at any particular configuration, you explore all the possible next configurations before diving deep into any one specific alternative. So, BFS is natural choice for exploring solutions to the rubix cube problem.

Thank you.