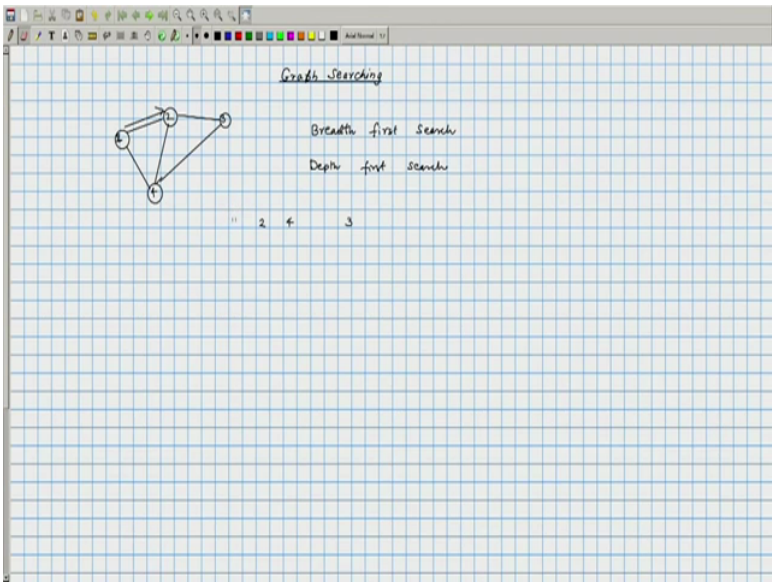
Discrete Mathematics
Professor Sajith Gopalan
Professor Benny George
Department of Computer Science & Engineering
Indian Institute of Technology, Guwahati
Lecture No. 16
Graph Searching; BFS and DFS

In this lecture we will learn about graph searching. So graph BFS is a collection of vertices and edges and we need to explore the vertices in a certain order. Let us take a simple example, if you think of any social networking site, you can look at your individual page as a certain node and you want to know all your connections. So you want to know who your friends are, their friends and so on.

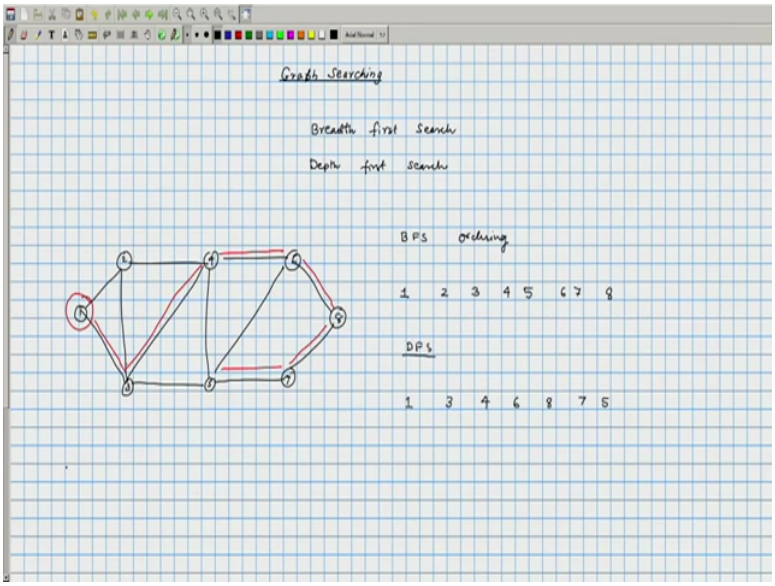
(Refer Slide Time: 1:11)



So you can think of this as a huge graph which this person is a node and there is an edge between two people if they are friends. So if given such graph so here if there is an example of 4 people; 1, 2, 3 and 4. And here 2 and 4 are friends with each other and they are friends with 1 as well. So you are given some arbitrary graph, how do you look at all the vertices, list all the vertices in a certain order, respected to let us say I mean if it is think about the case of social networking sites, you want to order all your connections in some order. You can either look at when you look at your friends, list them out and then friends of friends and then friends of friends of friends and so on and that would essentially be what we refer to as breadth first search.

You can also look at an alternate approach, you find one of your friends, look at another friend of his and keep on doing that sequence of friends then you will get what is called is DFS. The commonly used search and techniques they are we will just write them down breadth first search and depth first search. So here if you start your search at vertex 1, we may view vertices 2 and 4 are the vertices at depth 1 and the vertex 3 is at depth 2. So if you look out at the order 1 followed by 2 4 followed by 3 that would be the breadth first search, you have listed them in the order of their depths that is called as a depth first search, whereas if you list it as 1, 2, 3, 4, so you have gone from 1 to 2, from 2 you could go to 4, from 4 you could go to 3. But there are examples where these 2 searches return different orderings.

(Refer Slide Time: 4:01)



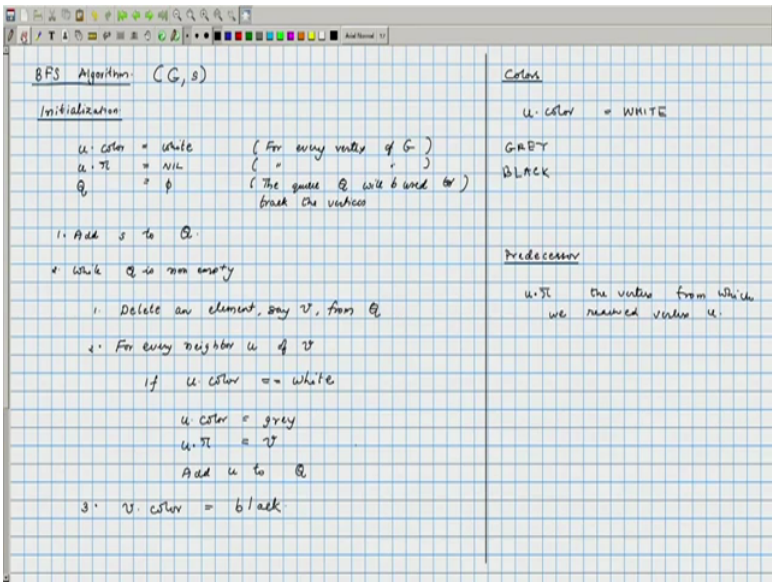
So let us look at this more formally, first look at the look at a little more complicated example so we have the following graph. Suppose we have this graph, what will be what will the BFS ordering be and what will the depth first search ordering be. So we will first the write down the BFS ordering, starting at vertex 1 so the start vertex let us colour this in red, this is just to indicate that is the starting vertex. So that will be the first vertex that is encountered in BFS or in DFS. Now look at all the neighbours of 1, there are precisely two neighbours 2 and 3, list them out and you will get 2 and 3. And the next stage would be to list out all the neighbours of 2 followed by all the neighbours of 3. To do that in this case the neighbours of 2 that is 3 so we will not list out the already listed neighbours.

So there are precisely two neighbours namely 4 and 5, so 4 is a neighbour of 2 as well as 3, 5 is a neighbour only of 3 and not of 2. And then we will list out the neighbours of 4 and 5, we will get 6, 7 and then we have the neighbour 8. So this would be the BFS ordering. Whereas

if you take that DFS ordering, so let us say so in BFS ordering it is not a means it is not fully determined in the sense the the vertex we set vertex 2 before 3 or we set 3 before 2 that is not really specified, you could be set it in any order and that has a chain of other choices that may not be fully determined., The same applies in DFS, which neighbour you pick there is no restriction in either BFS or DFS but whether you look at all neighbours of a node and then go to the next level or whether you keep on traversing the graph layer by layer from one vertex to its neighbour and then to its neighbour and so on so that is what differentiates BFS and DFS.

So DFS ordering so you could go from vertex 1 to vertex say 3, and from 3 you could go to 4, from 4 you could go either to 5 or 6, so the ones which we have visited we just draw in red, so 1 to 3 we went, from 3 to 4 we went and then 4 to 6 we could go, and from 6 we could go to 8 and from 8 we could go to 7, and from 7 we could go to 5, and after we have reached 5 we see that all its neighbours are essentially visited. So the neighbours of 5 are 3, 4 and 6, all of them are visited so you go back to the node from where you came to 5 namely 7, its neighbours are all being taken care of, same with 8, same with 6, same with 4, you back lag all the way up to vertex 4. 4 has a neighbour which is not yet visited so that is node number 2 so that will be the last node that is visited. So this will be the DFS ordering. Now we will see how we can algorithmically implement this, we will see an algorithm for (DF) doing these searches.

(Refer Slide Time: 8:32)



What we will do is a following, we will have colours associated with every vertex. So these colours is to indicate whether certain vertex has been visited whether all its neighbours have

been explored and so on. So we will have 3 colours, so colours will be one attribute, another attribute would be its predecessor, each vertex will have a predecessor so this will indicate the vertex from which we explored the given vertex. So if you have a vertex u, so u dot colour will initially be white for every vertex. So this is a start state, every vertex will be initially white, the other colours are grey and black, grey would mean we have partially explored that vertex and it is still under process, some of its neighbours have been explored some of them have not been.

And black basically means all it is means that vertex everything that we had to do with respect to that vertex is over, we have we have finished the processing of that particular vertex and there will be another parameter associated with each vertex that will be its predecessor, so denote it by u dot pi, so u dot pi would mean the predecessor of u and this is the vertex from which we reached vertex u so that will be the predecessor of u. Now an algorithm the way it works is it will have a queue and we will push these vertices into a queue as the algorithm progresses, and after (the proc) all processing is (com) is done, you would have visited all the vertices from a given (starti) starting vertex. We will assume that the initial graph that is given to us is an undirected graph which is a connected graph, of course BFS would work even if the graph was a directed graph and if it was a disconnected graph as well.

When the graph was disconnected, what we can do is we can start a BFS at some arbitrary node in each connected component. We might need to discover what are the connected components or we could just start BFS at every vertex of the graph, to start the BFS at every vertex of the graph we need to check whether a certain vertex is already being covered by any other previous BFS. So while doing the breadth first search, if we have already seen some vertex as part of searching certain other component we can ignore that BFS search so that modification can be made. And directed graphs these are the essentially the same algorithm would work, but for (simplify simply) just to (keeps) keep things simple we will just look at undirected example.

So in BFS algorithm we have to initialise the colours, so in initialisation phase where in for every vertex so u dot colour will be set to white that means at the start vertices are all unexplored so we will assume that the BFS algorithm which may we may call it as BFS, it has 2 parameters; the given graph and the starting vertex S. And for every vertex the parent u dot pi will be equal to nil, so initially we do not know what is the predecessor or parent of a

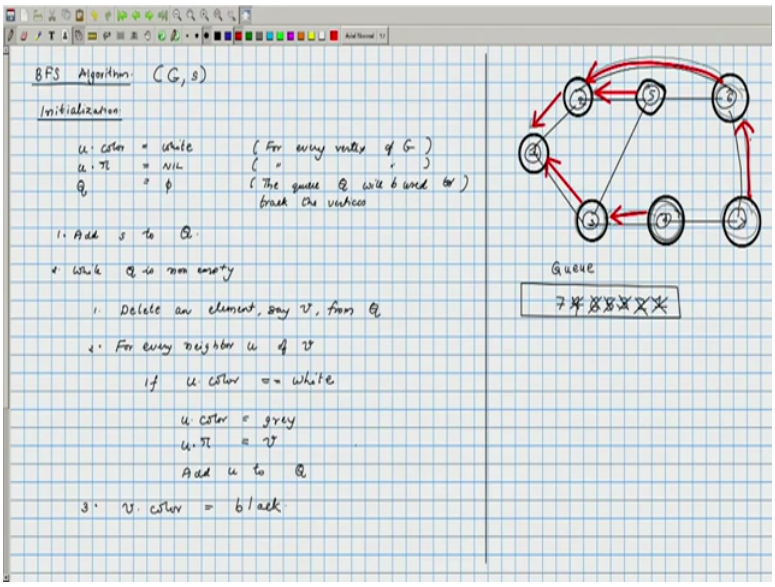
particular vertex so we will just set it to pi. As our algorithm progresses, when a node is discovered for very first time, the node the edge which cause the discovery of this particular node will be explored and that edge will tell us what is the predecessor of the particular node that we have just now found, so initially this is set to nil.

And then we can also initialise the queue, we will call it as with a queue we will name it as Q itself and this is set to empty so there is an empty queue, the queue Q will be used to track the vertices and it is initialised to empty. And then what the algorithm does is, it will repeatedly explore for whatever is there in the Q, so after the Q has been initialised we have to add one element so we will add S to Q. The starting element that we had the starting vertex is added to the Q, this is the first step of the algorithm and then while Q is non-empty, we will repeatedly do certain actions. So delete elements from the Q, so we will call this as deleted element as V from Q so Q is a first-in first-out data structure which means the element that you have added they will be removed in the same order.

So at the start we have added just one element so when you try to delete, the first element that will be deleted is going to be S, and then for every neighbour of V so we have deleted one particular element, for each of its neighbours we would we would start exploring those vertices. So for every neighbour say u of V we will do the following, we have one particular node that is deleted and that might have many neighbours and for each of the neighbours if it is an unexplored neighbour means no in this vertex vertex has not been explored before, we would want to do something with it, if it is already been explored we will just leave it as it is.

So if u dot colour is equal to white, in that case what we will have to do is we will have to push this into the Q and before adding it into the Q we will change its colour, so this means we need to process it so it will change its colour and the processing has begun, u dot colour is equal to grey and its parent u dot predecessor this we will set it to V because V was the vertex from which we visited u because u is a neighbour of V. So the parent is being set and then we can add u to the Q, add u to Q. And after this processing is done for every neighbour we can change the colour of the vertex V to black because V has been processed so V dot colour is equal to black, so that is the algorithm.

(Refer Slide Time: 17:35)



So now let us see this in action, so let us say we have this following graph, if you start at vertex 1 so this we will explain in a queue, so this is our initial queue and we will first add vertex 1 to the queue that is the only element that is there in the queue. And then we delete one, so initially the colour of all vertices are essentially white, once we have added the colour of that vertex changes it becomes grey so 1 is now a grey vertex. And we will look at all the neighbours of u and change its colour and we will set its parent to 1, so the vertex 1 has two neighbours namely 2 and 3, so 2 and 3 their colour is being changed to grey and the parent is being set to vertex 1 so there is going to be so by this arrow I will indicate the parent, so 2s is parent is 1, 3s parent is also 1.

And after this so this is the step that you do and 2 and 3 has been added, so we may assume that first 2 is added and after the processing is done 3 is added. So vertex 2 would be added and vertex 3 would be added and we would have taken off 1 after this, so that would that would have been the first step we have deleted that element and we have changed the colour of 1 to black. So now it is no longer grey but it has become black indicating that full processing is over. The queue is still not empty so we will pick out vertex 2, and if you look at vertex 2 that has been deleted, once vertex 2 has been deleted, we will look at all its neighbours.

Vertex 2 has namely 3 neighbours: 1, 3 and 5 sorry 1, 5 and 6 are the neighbours. When you look at these neighbours, 5 and 6 both are currently of colour white so we can add them, so let us say we first add 5 and then add 6 so that goes inside the queue and their colour as well

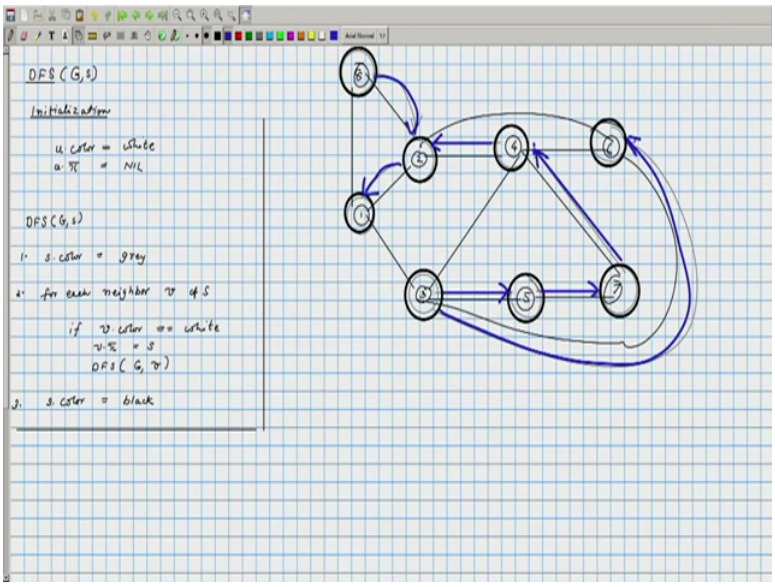
as parent is reset. So 5 becomes the first neighbour let us say it is 5. So the colour is changed and the parent is set 2 and 5 is added, and after 5 is added the next neighbour of 2 is going to be 6 that is also going to change its colour to grey and we have parent for 6 which is 2, and 6 is being added into the queue. So after 5 and 6 is being added, the next node that you process is 3. 3 has again three neighbours; 1, 5 and 4, out of them the only white neighbour is going to be 4 so that will be converted into grey and the parent would be set to 3.

And additionally we would have 4 coming into the queue and 3 by the stage is done with, we could have deleted 3. The next element that is going to be deleted is 5 but there is no further processing to be done with 5 because all the neighbours of 5 would have been already of colour other than white so 5 is also done. So when we are deleting and after the delete phase, after the neighbours have been processed we would have changed the colour to black so there we skipped couple of steps we would have changed 2 to black and 3 to black after that, and then 5 to black. The next node that is going to be processed is going to be 6, 6 has one neighbour namely 7 so we will add 7 into the next into the queue and its colours are changed and the parent is also set.

6 the other neighbours of 6 are 5 and 2 which surely cannot be added, and after 7 is being added we can say that the processing of 6 is over and its colour becomes black. The next known that would be processed is 4, 4 has two neighbours 3 and 7 but both of them have colours different from white and therefore there is nothing to be done with any other neighbours of 4, 4s colour will be changed to black and then the queue contains only 7 that is removed, all the neighbours of 7 namely 6 and 4 has already been processed, they are black in colour and therefore nothing has to be done with 7 other than change its colour to black. And at this stage the queue is again empty so that processing is over and we would have traversed all the nodes by that.

And note that if you look at just the predecessors of each node that will be an interesting diagram, this will be a tree then going to we know cycles it is a directed tree with every node having a unique path to the vertex 1 and that tree is called as a BFS tree and this entire algorithm can be implemented in linear time if you maintain the initial graph as an adjacency list. The next search algorithm that we will see is the DFS search.

(Refer Slide Time: 25:31)



So let us begin the DFS search or the depth first search. So again we will have two parameters associated with each node namely its colour and the parent. Instead of using a queue we could implement DFS algorithm using a stack, we will see a recursive version of DFS algorithm wherein we will not have to exquisitely maintain the stack, recursion would essentially keep track of this stack. So DFS algorithm this will require two inputs two parameters; one is the graph itself and the other is the starting vertex. We will have an initialisation phase wherein the colour of each vertex, so u dot colour is equal to white and u dot parent is equal to nil. So before you start the DFS algorithm on any particular graph, we need to do this initialisation.

And once this is initialised we can call our DFS routine, so write down the DFS routine two parameters G and S, we will assume that the initialisation has already been done and what we will do is the following. So we will make the colour of S to be grey, so S dot colour is equal to grey and then for every neighbour of S reach we will do the following, if V dot colour is equal to white so while the DFS is being run each vertex its colour could change. We want to ensure that the colours change from white to grey to black, so if you have a white coloured vertex that means it is not being explored yet and therefore we are ready to explore it now because it is a neighbour of vertex S and what we will do is we will just call DFS with this new vertex on the same graph.

So DFS G V and after all the vertices of S has been processed, we can set the colour of S to black, S dot colour is equal to black maybe at this point, just before calling DFS we will need

to set the parent V dot pi is equal to S. So let us see how this works, suppose we have this particular graph and we start DFS at 1. So initially every vertex is of colour white and the parents, there are no parents everything is nil. And when you call DFS on vertex 1 what happens is the colour of that vertex immediately changes to grey, and then what happens is the recursive call starts. So for every neighbour, vertex 1 has two neighbours namely 2 and 3, for each of them we are going to call the DFS.

Let us assume that the first DFS call will be for 2 and the next DFS call will be for 3, but before that we need to check whether those are white, indeed they are white and therefore we can make those calls. The very first call that will be for is DFS 2, when DFS 2 is being called what happens is, it will go into the recursive call, the recursive routine and inside that the colour of this would be made grey and we would have set the parent. The parent of 2 is going to be 1 and now we are going to be processed DFS G with 2. When we are running the DFS on 2, its colour would have been set to grey and then for each neighbour of 2 we will have to process this particular code, when we will have to process these code segment for every neighbour of 2.

2 has three neighbours namely 1, 4 and 6 but out of them only 4 and 6 are white and therefore those alone will be processed. So let us say vertex 4 is going to be processed first, so its parent is going to be set to 2 and while DFS G 4 is being processed this will turn into a grey node. Once this is a grey node we will have to explore all its neighbours, it has four neighbours; 2, 3, 6 and 7. The white neighbours are three of them; 3, 6 and 7 so presumably all of them will essentially be processed. Let us say DFS 4 when it is being called and neighbour that is picked first is vertex number 7 and therefore the parent would be set and 7 would turn grey and DFS 7 is going to be processed.

DFS 7 is processed, 7 has two neighbours namely 4 and 5, the only white neighbours is going to be 5 so that is going to be a DFS 5 and the parent is going to be set. And DFS 5 if you look at all its neighbours, its neighbours are 3 and 7, and 3 is still white in colour so that would be processed and that turns grey. And when you are processing 3, the only neighbour of 3 that is still in contention is 6 so it will directly go to 6. And let us look at the situation now, 6 would have turned to grey and all the neighbours of 6 are of colour other than white so the call will return back to 3, let us just make one more neighbour, if we had a neighbour 8 which is connected to both 1 and 2, the recursive calls would have happened in the same way.

If you look at 6, all its neighbours is being processed so we will go to 3, and 3 also every neighbour is being processed, there is no neighbour of 3 which is of colour white and therefore it goes back to 5, and from 5 it goes back to 7, from 7 it goes back to 4 because all the neighbours of these nodes were already processed. When you come back to 2, 2 had lots of neighbours namely 6, 4, 1 and 8 but the only neighbour which is still white in colour is 8 so that will be processed and the parent would essentially be pointing to 2 and this would become grey.

And at this point all the calls being serviced and 8 has no longer any other neighbour, and it does not have any neighbour which is white in colour and therefore the entire processing is over, we should have checked that while we are returning these calls the colour is changed from grey to black. So the place where we would have ended up search initially would have been 6 so at nodes when we were at node 6, all its neighbours were processed and at that point this would have turned black, 6 would have turned black and then we would have come back to 3 and all neighbours of 3 has been processed so this would have again been turned to black and then when the call returns to 5 this would have become black, and then that call returns to 7 and 7's call is also serviced, this becomes black, this would have become black and the call returns back to 2.

2 has one more neighbour to be processed 8, after 8's processing is done, this becomes black and then 2 becomes black and 1 becomes black. And at this point we have we have a tree which is obtained by looking at the predecessor notes and that is what we will call as the DFS tree. This entire DFS operation which we did via means of recursive calls we can implement it by means of a stack, so each recursive call instead of it being automatically serviced by means of stack we can explicitly construct our own stack and push the vertices into the stack and pop it out from the stack as and when those vertices are processed. So in either case we can implement the algorithm in linear time assuming that the graph is given in an adjacency list format. So this is about BFS and DFS, we will stop here for the time being, so that is the end of this lecture.