Course: Introduction to Graph Algorithms

Professor: C Pandu Rangan

Department: Computer Science and Engineering

Institute: IISc

Week:07

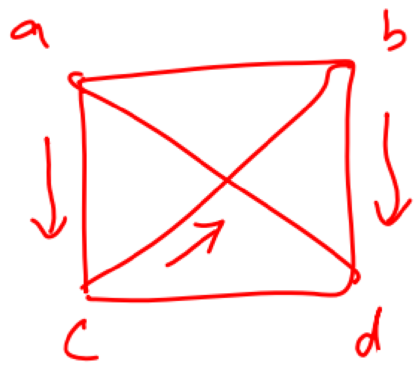
Lecture 28 DFS

Namaskara, in this session we are going to discuss about a systematic way of exploring graphs. The method that we are going to discuss is called depth first search or DFS, depth first search is a way in which we are going to go in the structure as deep as possible. We start from a vertex and then go to its neighbor and from there choose one of its neighbors and go over there and so on. So we will be always extending a path. That is why we say it is going deeper and deeper into the structure.

We go along a path and try to extend the path. Now, when we are not able to extend we will backtrack and then see the other options available, okay. So a graph being a very complex structure, systematic exploration of that is non-trivial. If you look into the array or linked list or doubly linked list, these are all very simple structures and traversing them, visiting all values and all nodes processing, all of them are very simple.

However a relation is a very complex structure, both directed and undirected graphs are so complex that we need a method, even to process the structure, processing the input itself if you want to solve problems on graphs, the graph must be processed, you have to do computation with all nodes edges and so on. So how do we explore the graph? How do we systematically account for each vertex and each edge, therefore this is a systematic exploration of a graph.

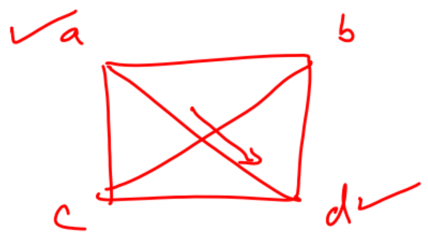
We begin with undirected graphs. Then we move on to look into the details related to directed graphs. They vary significantly in the way in which the exploration gives insight about the structure okay. So first we look at undirected graph then we look at directed graphs, okay. You can imagine as if you are starting from a vertex and walking along the edge and reach the other end of that edge, from there you proceed further and reach the other end of another edge. So in this way you are going to visit the vertices and edges, for example if this is a graph. Let us say a, b, c, d you can imagine that you are starting from a then visited along c along ac and then you have reached the c. After reaching c you have options you can go to b or you can go to d choose any option let us say you have chosen the option of going via cb. So you can go like this and reach b and from b you have several options. Of course, a is already visited, c is already visited, so you are going to take the option of using the edge bd and then you would come here and so on.



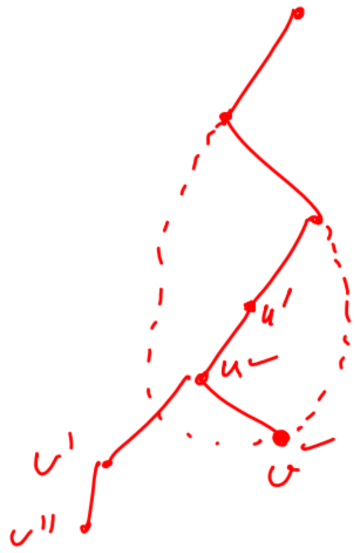
Therefore, the way in which we are going to explore the graph can be intuitively viewed as walking along the edges and reaching the vertices okay. Here we need to avoid visiting the same vertex again and again or using the same edge repeatedly and so on. In order to have a systematic way we would like to make sure that the visit is organized in a proper manner, otherwise we may get lost in a complicated structure. So we will have two categories of vertices, unvisited vertices, visited vertices, initially all vertices are unvisited, the graph is not explored, you are going to choose a vertex from which the exploration begins. In our case technically, it is called the depth first search begins at a vertex so you choose a vertex and you are starting there, so that is visited, in this picture I put the tick mark in implementation we can set appropriate Boolean flag, visited 0 is not visited, visited equal to 1 okay, marks that it is visited and so on. So we have several ways of recognizing whether a vertex is visited or not. Pictorially I will probably draw some tick mark but it is easy to implement the idea that a vertex is visited or not. From an unvisited vertex you begin from there you go to an unvisited vertex.

So mark a vertex as visited and from there you go to unvisited vertex. If there are several choices you can choose any one of them and visit there. Okay, there is no specific rule as to which one you should choose any unvisited vertex. For example when I was in a I could have gone to b or c or d you have 3 edges from a I choose to go to c that is all. So you I could have gone from a to any one of the unvisited vertices if you choose some other vertex that would define another way of exploring the graph.

For example, for the same picture a, b, c, d, I start from here. So I visited that. I have three options. Let us say I use this edge. Suppose I walk along this edge and reach here. Now d is visited. After coming to d, I am going to go to a node that is not yet visited and from there I apply the same logic. So go to a node and then from there choose an unvisited vertex and go over there along that edge alright and after going there I am doing the same thing. Therefore, there is a very simple recursive way to describe our plan. It is just a high level description of the plan.



We will see lot more details added to this. Our plan is start from somewhere, choose an edge and go to an unvisited vertex. From there, choose an edge, choose an unvisited vertex and go over there. Therefore, our initial few stages will look something like this. I start from a vertex, go to an unvisited vertex and from there go to another unvisited vertex, yet another unvisited vertex, another unvisited and suppose this process you keep going like this and you have come to a vertex that has got several neighbors, but all of them are visited okay. You cannot extend this any further, from there if there is an unvisited vertex, I could have gone there but there is no unvisited neighbor, in other words all the neighbors are already visited. Let me draw that pictorially like this it has one neighbor like this but that is already and another neighbor that is also already visited. So all neighbors are visited we cannot proceed any further from there. What do we do now? Now we go back this is called a backtracking right.



You have, let us say this vertex is v this vertex is u, you have come to v from u. At u there were several auctions you have chosen the option of v and you have come to v. At v you are not able to proceed further, therefore this auction is closed. go back to u and see whether there are any other options available. Suppose there is another option available one more unvisited neighbor, so you can take that one let us say v dash, from v dash you can go to v double dash and so on the journey continues exploration continues.

If all neighbors are visited then you backtrack, you go to the node from where you have come to this node okay. Suppose all neighbors of u are accounted for, now you go back to this node, you call this as let us say u dash you will go to u dash okay. This is the simple reason why a natural way to represent or describe our process is using recursion because there are two reasons. One is you start from a vertex, you go to another vertex and do similar thing, therefore recursion is a natural way to describe the process. The second thing is, after finishing everything about it, you are backtracking to the node from which you have come here, that is also handled by the recursive calls and recursive functions.

A recursive function may call the same function recursively and once that recursion is done, the control comes back to the point from where it is called that is the way recursion works. So for example, at u you have called recursively an action to work on v, at v the things are done, no further exploration is possible. Once that is done the control automatically comes to u and that is what we wanted, what we wanted is we want to come back to u and explore other possibilities. Therefore there is a very natural way to describe whatever we had in our mind as a recursive program okay.

Here is a vanilla version of recursive program of our plan, okay you call that as depth first search, you have given a graph G and there is a vertex u you have chosen from which you want to start the process okay.

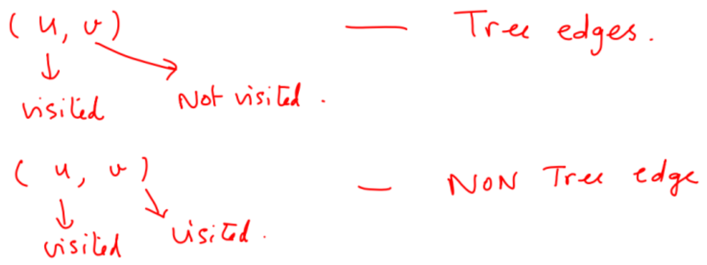
For each v which is in the adjacency list of u, that is for each neighbor you do the following. If v is unvisited or if u is not visited okay why unvisited that is bad we will write if v is not visited, DFS Gv recursively apply the same process there. So when does a particular recursive call terminate? A recursive call terminates when all neighbors are examined, so if it is unvisited you will go there and continue the visit. If it is visited skip that and go to another neighbor but there are only finitely many neighbors the adjacency list is finite, so this process is going to stop after exploring all neighbors then the recursive call ends, okay very simple way to describe our plan.

So we already introduced the notion of visited and not visited or unvisited kind of terminology. So to begin with, we are going to assume that all vertices are not visitor or unvisited nodes because nothing has started, as and when you visit you mark it, okay. If v is not visited DFS so in the process of DFS when you begin you are going to first mark it right. So a statement that should be written here is mark u visited.

How do you start the process, you are starting at u, so that is ticked which is equivalent to marking u visited and then for each adjacent vertex if v is not visited then DFS. So how the visiting is marked and how do you know whether a vertex is visited or not it can be recognized by a simple Boolean flag in our implementation. all right. Therefore this is a very high level plan we are going to refine it. We are going to refine it by adding some more formal steps and not only that as done here, you can perhaps execute it and you will understand that you are going to various nodes, however there is no other information about what you have done is kept in this version, that is why it is called the vanilla version, it is only a plan okay, in order to make the exploration more useful as and when we explore we are going to generate some useful information and store them and build them as we go along. This additional bookkeeping information are very helpful okay.

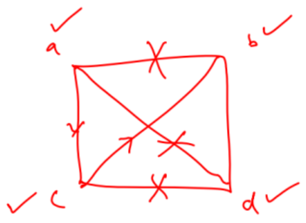
You are saying that you are going from one visited node to an unvisited node. Sometimes from one visited node to another visited node there is an edge, you will try to go there but it is visited so you are ignoring it. So certain edges do not cause trigger, certain edges do cause a trigger, for example let us go back to the previous picture. Let us look at this picture, from a you had gone to c, from c you had gone to b, from b you had gone to d. at d, you are looking at a neighbor db well that is already visited b is visited you are looking at dc that is already visited, c is already visited here okay.

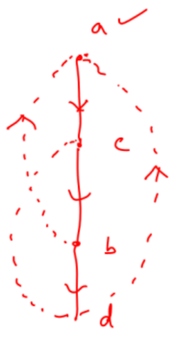
So you are not doing anything, no further recursive calls are triggered alright, so certain edges lead to further recursive calls, certain edges because the other end is also visited, they are not triggering any further recursive call. We would like to distinguish this, okay ,so we will have the following added to our plan we are refining our search strategy while exploring we will recognize the following two kinds of edges, okay. I will give the name now we will justify it little later. The edge that has taken you from edge of the form uv, u is visited, v is not visited or unvisited, from a visited node to an unvisited node, these edges are called tree edges. From a visited node to another visited node, an edge may take you there, if uv this is visited and this is also visited, it is already visited. If there is an edge like this, this is called non-tree edge okay. So we are going to classify the edges as tree edges and non-tree edges.



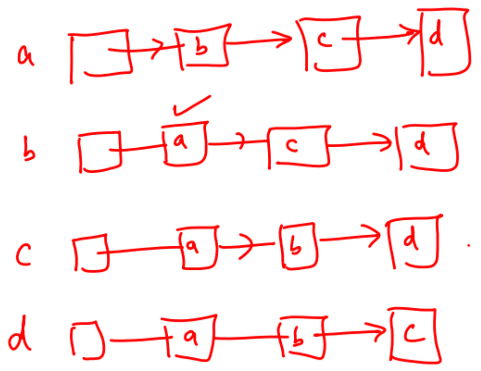
They have distinct property. So, this allows us to understand the structure and the nature of the nodes that are visited, the property of an edge and so on. The first thing that we are going to do is classify the edges as tree edges and non-tree edges. Let me demonstrate again with the same example. Let us take a, b, c, d okay, in the flow in which I have visited the vertices I am going to draw the same graph in the following way, I start from a I have marked it a is visited, initially all are unvisited now I have visited a because I started from there I mark it as visited. I choose an edge there are three options from a I can go to any unvisited vertex there is no specific order nothing is specified.

So let us say I have gone to c, so I have used to this edge, from a I had gone to c, now c is visited. From c let us say I go to b, then b is visited from c I had gone b via this edge, had gone via this edge. After b if you look at ba because all adjacent ones are to be examined, ba, a is already visited. So I am looking at the edge ba, b is visited the other end a is also visited. Therefore ba is a non-tree edge, this is not a part of the tree okay this is not the part of the tree, let me tell you why I call this tree in a minute but first of all, so in my picture I am going to draw like this with a dotted line. It is not a part of the tree. I explored that edge but that edge is taking me to a visited node so I am not going to do any further with that edge. I come back to b and then I look at bc. bc is also c is also visited. okay c is also visited, it is an undirected graph actually cb and bc are same edge it is an undirected graph right. So it is it is it is already visited, so I am not going to use that edge also , I in fact this edge has been already used so bc is another version of the same edge.





That is because the way in which the adjacency list it will represent easier twice. For instance, in the list for a I will have b, I will have c and I will have d. In the adjacency related to b who are all neighbors, a is a neighbor, c is a neighbor and d is a neighbor, for c again a is a neighbor, b is a neighbor and d is a neighbor, so adjacency list, these are all vertices adjacent to this and d also has got a, b and c as neighbors. It is a graph, adjacency list representation. Every edge I am looking at it because I am going to go through the entire adjacency list.

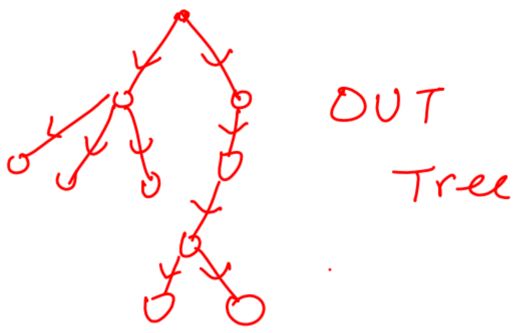


So when I look at b, what about a, ba? that is non-tree edge, b is visited, a is also, bc well that is another copy okay, it is already entered as a tree edge, so I am not going to look at that again, bd, d is not visited, so from b I would go to d, d is visited. you have a as a neighbor so da this edge is there but that is not a tree edge, non tree edge, da sorry da is not a tree edge, dc is also a non tree edge, db of course is another version of bd, it is already written okay. So you can see that at d I have looked at all neighbors all of them are visited. So at d I have, I am not going to extend any further because I would extend only if I have an unvisited neighbor.

From visited I go to unvisited, you always go from visited to unvisited, but I have no unvisited neighbor and so everything related to d is done, there is a recursive call you have triggered for d, that is done, the control comes back to where that call was made. The recursive function, recursive call at b has triggered d, that is done, so the control comes back here. So at b again now you look at all neighbors they are all bc is done, bd is done, ba is also done, so the control comes back to c. At c again all neighbors are done, therefore in this way you see that the control comes back and when the control comes back to the first node where you have started and if it all neighbors are accounted for okay, then you see that the process is complete okay.

So we have to distinguish between tree edges and non-tree edges. How is the tree edge defined, here I have drawn the tree edge in solid lines, you always go from one visited vertex to another unvisited vertex, it is always a fresh unvisited vertex. So these set of edges, they are never going to form a cycle because every time the edge is taking to an unvisited one, there is no repetition here. Therefore the set of edges which I called as a tree edges indeed they form a tree, first reason there is no cycle, are they connected, of course they are connected, you keep extending them right you know you are not jumping over here and there and picking edges from different parts it is a continuous growth. right and backtrack and then extend the other part and backtrack and extend the other part.

So it remains connected, connected acyclic so it is a tree, that is the reason why this set of edges which are of the type visited to unvisited, these kind of edges they are called tree edges. Other kinds of edges are non-tree edges okay. If you look at the non-tree edges, of course in this picture it is not , that the property is not really visible, we can work out another example where that property is visible. okay the non-tree edges we will see the reason soon they are called back edges for undirected graph okay, non-tree edges are called back edges, we will see why, we have seen the reason, the edges that are going from a visited vertex to unvisited vertex, they are called the tree edges. All other edges are non-tree edges. Therefore, the entire graph is viewed in a different way. It is viewed as tree edges and non-tree edges. In order to help the intuition, we will orient the tree edges in the way in which it was flowing away from the root, you start from a vertex and then keep going down and down, so it is going away from the root. If I have a tree in which all edges are going away from the root towards the leaf that is called out tree. okay, so this is the root all edges are oriented like this, this is an example of out tree, in tree means all of them will be oriented towards the root, out tree means from the root they go towards the leaf, that is how in our depth first search tree edges are built, you start from a node, go to the next node, from there you go to another unvisited.



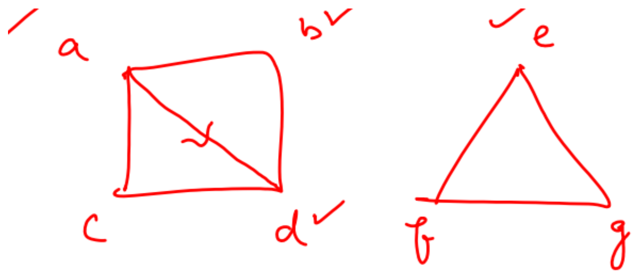
Therefore I have a natural direction flow in which tree edges can be visualized, this is a tree so there is a root which is a starting point and then the tree edges are drawn in an outgoing fashion okay. So you can see that in this depth first search a to c, c to b, b to d they are tree edges, they are going towards the leaf okay. This tree is called DFS tree, depth first search tree. When you perform the systematic exploration in a depth first fashion you get set of edges, they form a tree and that tree is called the DFS tree.

DFS tree is oriented from the root towards the leaf. What about back edges? Back edges are going to be oriented in the opposite direction. They are taking you back. It is visualized as going from a visited one to an already visited one. So it is taking you back. Therefore, back edges will be oriented in the opposite direction.

This is purely a view that will help our intuition in understanding the structure. Notice that my original graph is an undirected graph, I am doing depth first search in a particular order, in that order, in that flow I orient the tree edges, I give the direction to the tree edges in that order and the non-tree edges are oriented in the opposite direction. direction that is why they are called back edges they are taking you back to a higher level in the tree. The tree goes from top to bottom and the back edges will take you from bottom to top, from below to above. This understanding this view is very helpful in solving lot of problems on graphs okay.

Our plan is you are given a undirected graph, perform the search, identify few edges as tree edges, recognize all other edges as non-tree edges, imagine the tree edges are all flowing down and back edges are taking you back. This view is very important. Okay, not only there as we are building the tree we can recognize some important properties of the graph that has been explored so far, you have done exploration of certain things, whatever you have done up to that part some consolidation of information is possible and that is what we are going to use later, okay therefore as and when we perform the exploration we are going to build, we are going to obtain some additional information and that also we will maintain in a very comfortable bookkeeping fashion. So I am going to introduce some bookkeeping mechanisms or certain variables associated with each vertex and they are going to carry certain values.

Those values are all going to provide valuable information. First of all, the input graph itself could be a disconnected graph, for example, if the input graph is like this a, b, c, d and e, f, g.



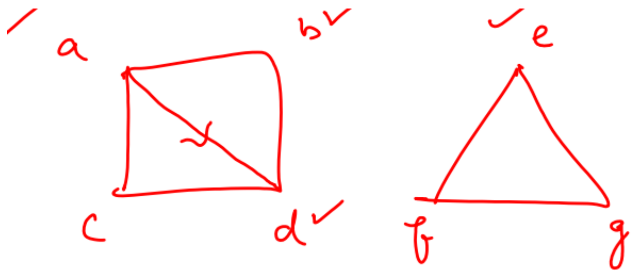
You see that this is a whole graph it has got 2 connected components and if I start from any vertex let us say a I take from a I go to let us say d, this edge ad is used, from d let us say I go to b. I go to b, b has two neighbors a, b has one neighbor a that is already visited, b has another neighbor d but that is from where you had come here alright. So db is visited, therefore everything with respect to b is done. No more exploration further exploration is possible, the control backtrack to from where the call for b was triggered.

The call for b was triggered at d, so when you come back to d, da is visited, db is visited, dc so you can see that this c is visited. Now c has got a neighbor d that is from where it had come c has another neighbor a, but that is already visited, that is already visited. Everything related to c is done, the control comes back to d, everything related to d is done the control comes back to a, at a also all neighbors have been explored therefore the whole process stops here but you see that e, f, g are not even explored, that is also a part of a graph except that it is in another component. Therefore you have to trigger a fresh search from one of the unvisited vertices, therefore you, let us say start from f suppose you have gone from f to e, then from e to g and g has two neighbors e and f they are already visited, g has one neighbor e and g has another neighbor f, gf is an entry edge and control comes back to e, control comes back to f and you stop, you can see that the depth first search has resulted in two trees, one tree for each component okay.

So if the graph is connected, the depth first search because it is connected, you start from a vertex, everything reachable from that vertex will be in the tree, can eventually go there. If the graph is connected depth first search will give single tree. If the graph has several components it will give you as many trees as possible therefore depth first search results in a single tree in which case it is called depth first search tree if G is connected, it results in several trees, this is called depth first search forest, why it is called the forest? It is a collection of trees, you have several trees. For example, in this case there are two trees, it is called depth first search forest, if G is disconnected and has several components.

If it has several components for each component there will be one tree because a component means everything is mutually connected. So you start from a vertex all vertices, in that component are reachable, so your search will reach them, a tree will be found, non-tree edges will be there in that component and that completes the exploration of that component therefore you see that if G is disconnected, For each component there will be a tree, therefore the whole process will result in depth first search forest and non forest edges or non tree edges in each component. So how do we generalize? Here is the first way in which we are going to refine our code. First step mark all v belonging to V as unvisited none of them are visited okay.

2. for each v belonging to V that is unvisited perform DFS Gv. Now you see that this extension of our vanilla, will not stop with the first tree that is constructed, still there are unmarked vertices in this picture.



After marking all of them you are not going to simply stop there, there are still unmarked vertices, you started from f you made a fresh start. from one of the unmarked vertices you could have started at e, you could have started at g, you could have started at f, any unmarked vertex you can choose and you can start your fresh search and that search will explore all the connected component where that vertex is located okay. this extension runs through the entire graph and each DFS that is triggered is going to construct a tree with that vertex as a root. So a DFS forest may result okay if the input graph is connected then The first trigger will cover all the vertices and you will have only one DFS tree but if the original graph is disconnected a trigger of recursive call will be done on behalf of every component and a tree will be constructed for every component by this procedure.

So this is an extension for arbitrary graph. My earlier discussion was only for a connected graph and a vertex in that connected component. Now this takes care of exploration of the entire graph okay when you are exploring. It is not just that a vertex is visited or unvisited there is another state for a vertex exists during our exploration that is the backtracking. When you backtrack you are exploring the other option you are not completely done with it because there are still some other neighbors unvisited or available. For example, if I start like this, suppose I am here, I had gone here, this is done.

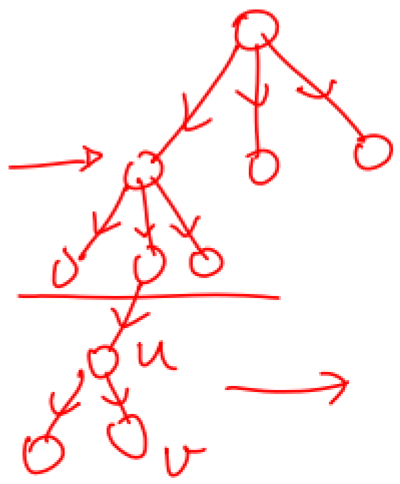
Suppose I have explored like this, this is done with. Then you backtrack and you take another option and then you do something and this is also this. explore another neighbor , unvisited neighbor there you explore further like that you can see that in this you come back and take another option and again once that option is done you come back and then take another option because all neighbors must be explored. okay all neighbors must be explored all edges must be explored so all neighbors must be explored. So you will come back again and again until all the neighbors are explored and after all the neighbors are explored the control goes back to its parent there again its neighbors are explored and so on. Therefore we see that this particular node is visited in the backtracking phase and again other unvisited neighbors are continued okay.

Therefore for every node we would like to have three status okay not visited. visited and live, what we mean by live is you are not done with it, you have taken one option, you have finished with that and then you have come to this, again you are taking another option, after finishing that again you come back, so whenever you come back the node is still live. okay visited and live, then not visited completed the visit. All neighbors are explored everything related to that node is done you backtrack you go back and then you are here in that case the whole thing is completed with respect to a particular node.

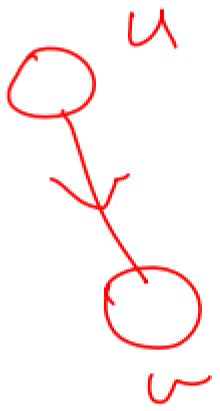
Therefore we have to recognize three different states. In our more refined version we will do that you can do that with three numbers 0, 1 and 2 but we will do that with color. Color is an attribute that will tell you in which state a node is okay. Different books will use different ways of recognizing the states. I am following the notation which is there in Korman which is one of the prescribed text for you okay. The Korman is using objects notation for every object you have attributes you have fields so use the dot notation.

So for example for a node v, v dot color that tells you the status of v, so when it is white it is not visited, when it is gray it is visited and live, live in the sense you are not done with it. If you are done with it, it is a black, completed the visit and left for its parent. left for its parent, why I use the term parent, this is going to give me the next important bookkeeping information.

I have an out tree, my depth first search tree is going to be an out tree. Like this, every node has got several children, for example these nodes are all children for this node. So node to children association is one to many, one node may have several children but a node to a parent is unique every node has got only one parent node.



Therefore one way of representing this tree is by representing the parent relation, for example if this vertex is u if this vertex is v, v is a child of u and u is the parent of v okay, if this is the case v is a child of u, u notice that u has several children v is a child of u but u is the parent of v there is only one because of this I am going to specify the tree I am going to represent the tree using what is known as parent array, parent of v equal to u. So, for every vertex I maintain the parent information this is the array notation. Coming to the typical notation of objects this you call it as v dot p, this is a field it is a parent field like you had the color field, you have the parent v dot p equal to u. v dot p equal to u means parent of v is u, in this picture you have u and v specified like this. So let me write down this is u and this is v and it is like this parent of v equal to u is denoted by v dot p equal to u. Then what is the tree edge, tree edge is v dot p, v this is the tree edge, this is the tree edge ending at v, this is the tree edge ending at v, so you have v dot p to v okay, so to summarize we have two attributes of a vertex v.color, v.p defined.



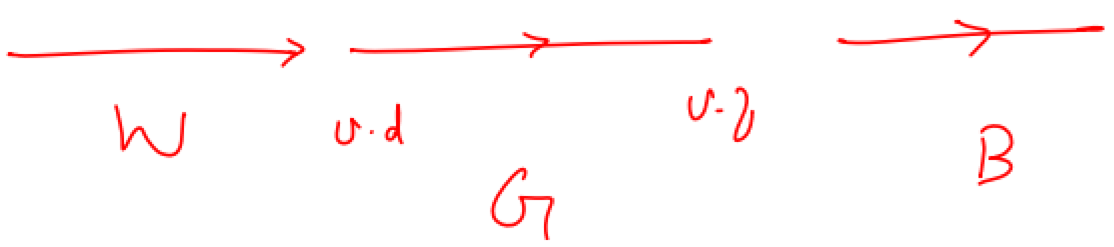
$$v.p = u$$

$$p(v) = u$$

$$\underline{\underline{\pi}}$$

We are going to do one more important bookkeeping information this an integer variable and this is going to us explicitly the order in which we have explored, the order in which the graph was explored by the depth first search. So when you are visiting a vertex v it is important to know in what order or when you have visited that vertex. Which vertex was visited first, which vertex was visited next and so on, so I am going to associate an integer okay and that integer will give the rank, the time at which it was visited first. So I am going to maintain a global variable called time, that time will keep ticking every time you visit a particular node and every time when you leave the node. That will tell you when you have visited and when you have finished and the time at which you have entered a node will be stored in v.d, this is called discovery time, v.f is finish time. When you are finished with that and then you had gone back. So between the discovery time and finish time the vertex is said to be active or live, before discovery time it will be not visited, it is not discovered it has your search has not come to v at all. So we will remain white first time when you enter. that is the discovery time and that is when it is going to become gray okay. And then after some time you have finished the exploration and you will exit v, you will backtrack from v that is when it is finished then it becomes black because that is an indicator that you have done with.

So there is a window of time up to which it will be white and between the discovery time and finish time it will be gray and after that it will be black.



So every vertex will have a color and from the color of the vertex you will understand whether it has been done with or you have not even started or currently it is active, in the sense its exploration is still going on, okay because it has taken one of the options and you had gone there you might have come back and you have might have taken another option still some more options might be available it is not done yet. So that is indicated by the fact that the color value is gray, therefore the status of your exploration is understood by the color. I will explain the discovery time and finish time with an example and that should give an idea of the values that we are computing during depth first search, okay. So we will take a closer look at an example in which all these values are explicitly computed. We will see the details in the next session. Thank you.