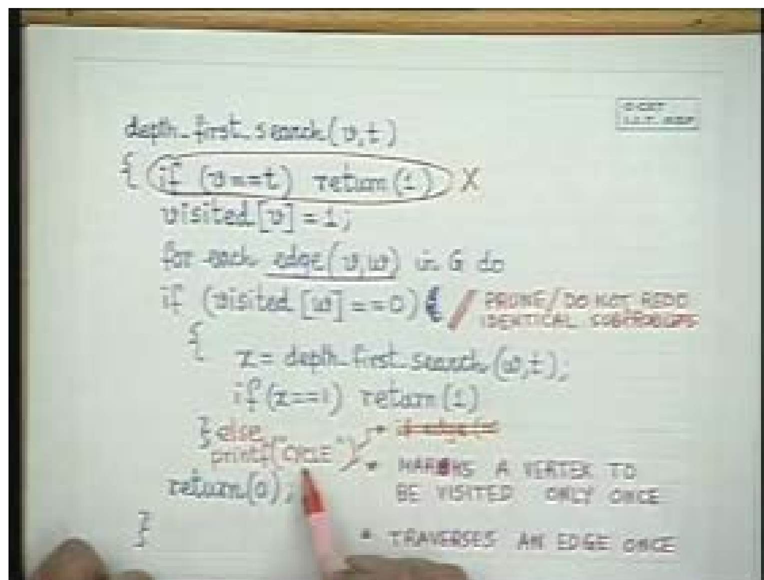
Programming and Data Structure
Dr. P. P. Chakraborty
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture # 31
Graphs – III

We continue our study of depth first search in graphs. You will remember that the problem that we were tackling was that of finding out whether there is a path from a vertex s to a vertex t in a graph g and in that problem we first got an initial definition. The initial definition was a recursive one with the base condition saying that if s is equal to t then you will return true otherwise then for each successor w or each edge v w in the graph, if w to t if there is a path from w to t then obviously there is a path from v to t because there is an edge from v to w. So v recursively searched w to t and if anyone of them gives a solution, we stopped and said yes there is a path. But while implementing it we saw that there is a problem of identical, say there is a problem of falling into an infinite loop due to identical sub problems.

Now there we used our concept of dynamic programming to get out of that issue and we used an array visited which was like the done array of Fibonacci which indicated that that vertex has already been seen or that sub problem has already been solved and if that was solved we did not solve it again. So by that we obtain the following algorithm. The algorithm was if v is equal to t return 1 and then mark it visited because this is the vertex we have visited. Otherwise that is v is not equal to, for each edge v w in t here we used the dynamic programming idea, here prune or do not redo identical sub problems.

(Refer Slide Time: 00:19:52)

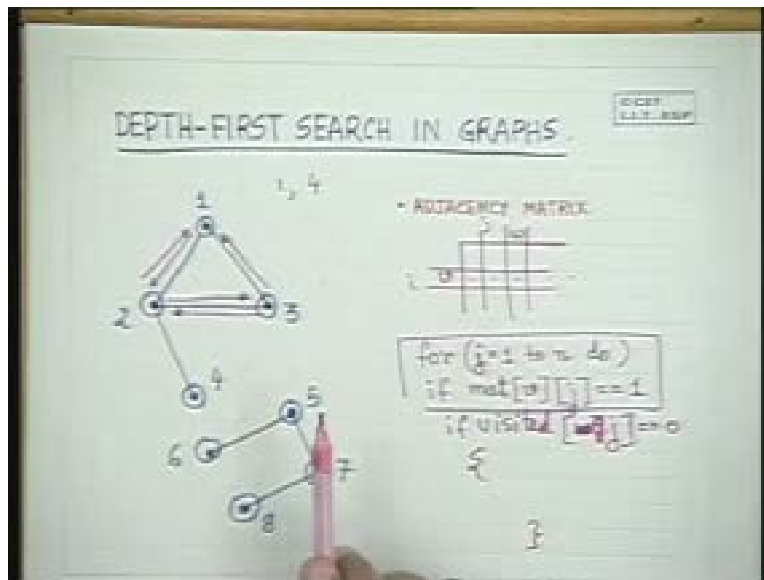


Now if w is visited already and there was a path from w to t then the algorithm would not have come here, it would have stopped whenever we had solved this w earlier, alright.

Therefore we did not take a solution from there. So if it is not visited only then you would recursively visit it, so that was the algorithm that we obtain now. Let us see how useful or what is the efficiency of that algorithm or how we will implement it in a particular data structure. That is will we use an adjacency list, will we use an adjacency matrix or will we use a direct representation which is the representation that we will use. So let us take an example. So, let us take our running example which we had, let us redraw it.

Now in this example let us see how, which of the algorithms which we are going to implement, which of them will be efficient. So let us see what if we use an adjacency matrix. If v is equal to t is directly checked, visited this is an array so this will take constant time. For each edge v w in g, this is very crucial, this part. Now in an adjacency matrix as you will notice, this is a matrix with v and w. If you want to find out whether the edge v w exists, it is ok but if you want to find out all the edges in t although w's which are connected you will have to go through all of them and here in the adjacency matrix representation for this case, you will have to write it out this way for if there are n such vertices i equal to for j, suppose this is i and this is j for j equal to 1 to n do.

(Refer Slide Time: 00:07:52)

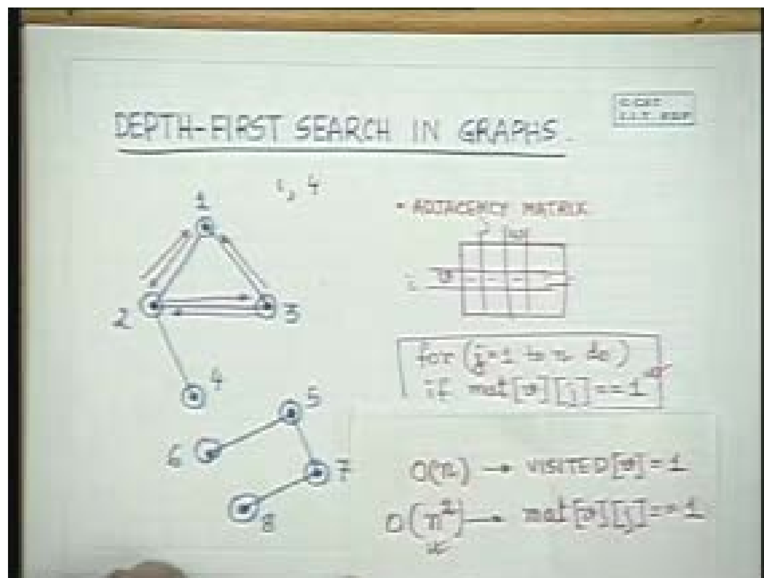


If matrix mat v to j equal to 1 that is what is going to be is, this is the path and then you will write if visited w or here it will be j is equal to equal to 0 etc. So this part will replace this, ok the rest will be identical. So what is the complexity? Each node, each edge you will come, from this edge once and this edge back once. Once you come here, a node will be visited, will be marked visited only once. Therefore let us see if I give you 1 to say 4 then you will come from here to here once, from here ok let we did not put this edge. From here to here once and once this is marked visited, you will not come back. From here to here once, from here to here once but this is marked visited, so you will not continue you will come back then from here to here you will try once and then so for every edge you will try it only once.

So the complexity of this algorithm it visits, it marks a vertex visited once, marks a vertex to be visited only once and traverses an edge at most twice that is v w once and w v. So if you put both the edges then an edge is also visited once. Therefore each edge v w will be checked only once either v w or w v, v w will be checked once or w v will be checked once and no more because either once if it is checked once and if it is not visited w will become visited, so next time you will not required to come across this edge.

Now in an adjacency matrix representation, this will be checked by checking all the elements here. So if there are n vertices ok, so let us see what is the complexity of the adjacency matrix representation. If we use an adjacency matrix representation, what is the complexity?

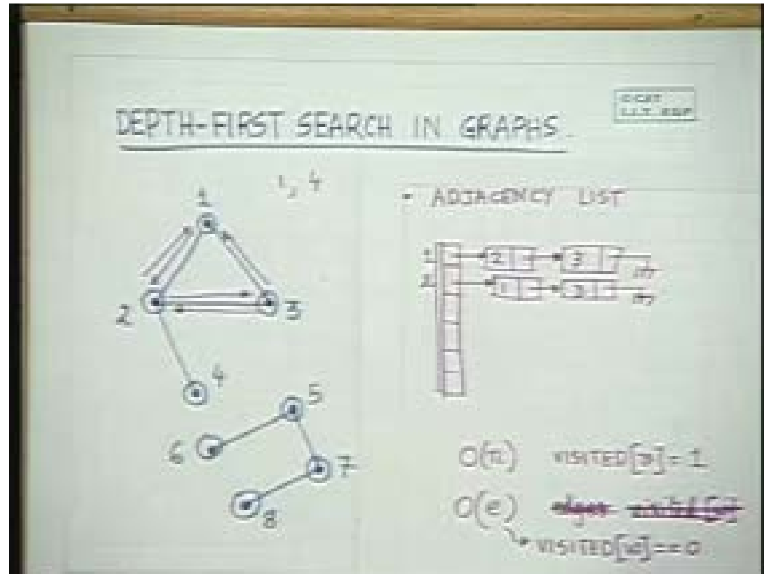
(Refer Slide Time: 00:10:41)



Visited is marked n so that is order n, there are n vertices. On the other hand here for each edge to check all the edges of this, you will have to go n time. So for each edge you will check this, this, this, this, this so for each vertex you will check all the edges. So you will be checking so many rows, for each vertex you will check all the rows. So the total number of rows is n into n, so n square comparisons here, this comparison will be done n square number of times.

So here this is for visited v equal to 1 and this will be for checking mat v j equal to equal to 1, so that complexity using the adjacency matrix will be order n square. Now let us see what will be the complexity using an adjacency list.

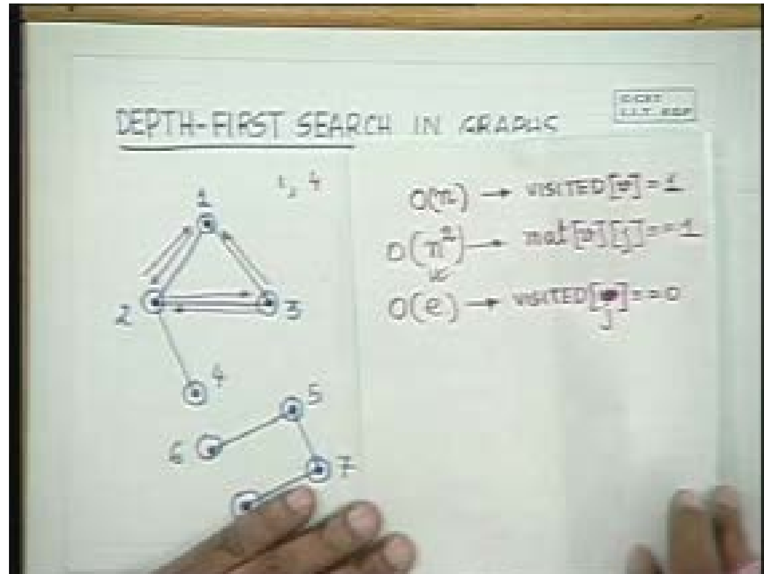
(Refer Slide Time: 00:12:51)



So suppose we use an adjacency list. You will recall in an adjacency list for each vertex say 1 it is kept like this, 2 3. For 2 this is 1, this is 2, for 2 I will have 1, 3 and so on. Therefore in the algorithm for the each edge v w in g, it will just proceed along this linked list. So each edge will be checked only once, so the vertices visited will be marked here. So to mark the visited again here will be order n, visited v equal to 1 but this check if visited w will be checked for each edge and this each edge will be implemented like this by moving along this.

So unlike in the adjacency matrix where you need to check whether this edge exists or does not exist, here you only need to check on the edges. So if e are the number of edges then here it will do, edges will be checked. So here this visited w, the num, if e are the number of edges then this check visited w equal to equal to 0 which comes here, this check will be done only order e times. In the matrix adjacency matrix also this check will be done e times but so in the adjacency matrix the check that mat i j is equal to equal to 1 that will be checked order n times.

(Refer Slide Time: 00:13:35)

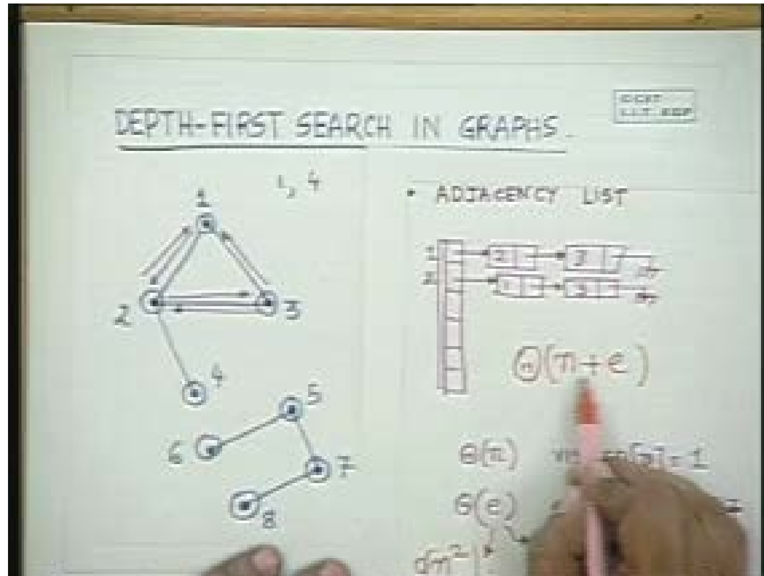


So in the adjacency matrix this will be the order e times where e is the number of edges you will check visited w equal to equal to 0 this or visited j equal to equal to 0, this will be checked e times but this will be checked n square time but this does not exist, this checked is not required in the adjacency list. Therefore in the adjacency it will be order n and order e and these are exact orders you can easily show. So this is the complexity, now order e the number of edges, how many edges can you have in a graph? In a graph of n nodes you can have n square edges.

So in the worst case e is approximately n square is of order n square, so this will be order n square but if e is less that if e is less then it will still be order e. Therefore this adjacency list representation is more useful because it checks only the number of edges, though in the worst case the number of edges are order n square but in the better cases it will order e. Therefore adjacency list representation is more useful then the adjacency matrix representation for solving this problem, right.

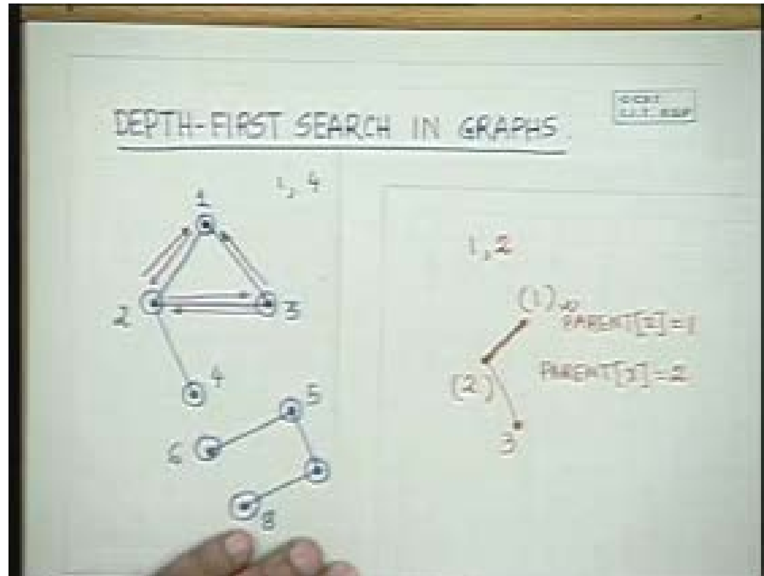
Now take the direct representation. The direct representation will move along the graph just as it is and in the direct representation you can put in that place the concept of visited. So the direct representation will also traverse which will be just like this and it will have traverse every edge only once. So the direct representation or the adjacency list representation both of them will work out this algorithm to find out whether there exists an edge or vertex in order maximum of n plus e time which is linear because the size of the graph is n plus e, we have to give number of edges plus number of vertices. So we get using depth first search, we get an optimal linear time algorithm to solve the graph connectivity problem. So we have how we approach the problem by a recursive definition and then we did use dynamic programming by marking the node visited and by such a simple idea, we were able to prove or we were able to obtain the feature that this algorithm is now an optimal linear time algorithm.

(Refer Slide Time: 00:15:16)



This depth first search is useful to solve a number of problems. Let us see the next problem and let us try and see what problems we can solve using this sort of depth first search. Now suppose I come back to a node and the node is visited. Can I say that I have got a cycle because suppose from 1 to 2, 2 to 3, 3 to 1 I have reached from 1 to 2, 2 to 3 and I have reached back 3 to 1 and 1 is visited therefore obviously this is a cycle. So does this algorithm that we have written, at this point can we write down if visited this is fine else that if this visited that if it is already visited, can we say that this, we have reached a cycle. That is can we just modify the algorithm like this and write down here that is here we write down else printf cycle. That is if this is true we do it else we say there is a cycle that is we have already reached a node which we visited before that was motivated from this situation but it is not so easy to understand that this is a cycle. Let us go back to our original example and see.

(Refer Slide Time: 00:20:29)



Let us say we start from 1. So 1 is visited and we start from 1 and we have no terminal vertices, we just continue searching and because there is no t, therefore this part, this part is not there. But then will the algorithm terminate? This is not there, in our depth first search t is not required because if all of them are visited obviously we have reached the solution. Now suppose we start from 1, from 1 we will go to 2, 2 will be visited but from 2 we will go to 1 and 1 is visited.

So in an undirected graph we will say there is a cycle when there is an edge, right. So the solution here that if visited v equal to 0, do search otherwise print cycle this will not work. Why because this is a, this is just an edge. Now in this edge we will mark this edge to be also a cycle. So how do we get out of this issue? That is how do we indicate that an edge if you mark an edge then that is not a cycle. What we can do is we can quickly checkup whether the edge 2 1 exists.

Now in order to quickly checkup whether the edge 2 1 exist that is from 1 to 2 we have reached and we have found that this is like this then what we need to do is we need to checkup whether this edge is a, is existing or not. If this edge is existing then obviously this cannot be a cycle. So if we do that then we modify this part instead of writing printf cycle. Here we check if edge w, see this means that from v to w we have reached but edge v w exists. Now what is this cycle from? This cycle must have been from somewhere.

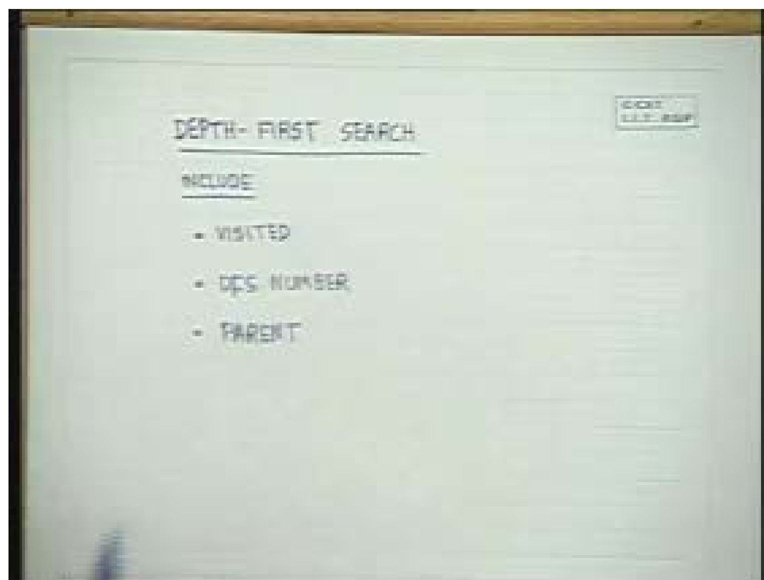
For example from 1 to 2, 2 to 3 I will get 1 2 3 and then here this will be a cycle. So how do we avoid this issue? We avoid this issue by denoting in the depth first search, who is the parent of 1. For example 1 will be the parent of 2, so parent of 2 is equal to 1, parent of 3 is equal to 2. So this way if we mark the parents and we come to a node here which is already visited, we have to check that it is not the parent because if it is the parent then obviously there is a cycle, there is no cycle but if it is not the parent then obviously we

can reach a cycle. So based on this idea, we now try and develop the algorithm as to solve the query is there a cycle.

So let us have a look at the algorithm which does is there a cycle. During this algorithm we also do a numbering on the graph that is we start numbering the nodes in a particular way. How do we number. We instead of marking it visited, we give a number to them that is we say this is visited first then this second then this third then this four, if it is in this order, the number will be 1 2 3 4 it depends on the order in which we are going to visit them.

So we make a modification of this depth first search algorithm by incorporating one something called the depth first number. So we are going to reach our final depth first search algorithm. Include we have already included visited depth first number and we can put instead of marking it 1 or 0 we can put the depth first number in the visited array and we also include the parent.

(Refer Slide Time: 00:22:20)



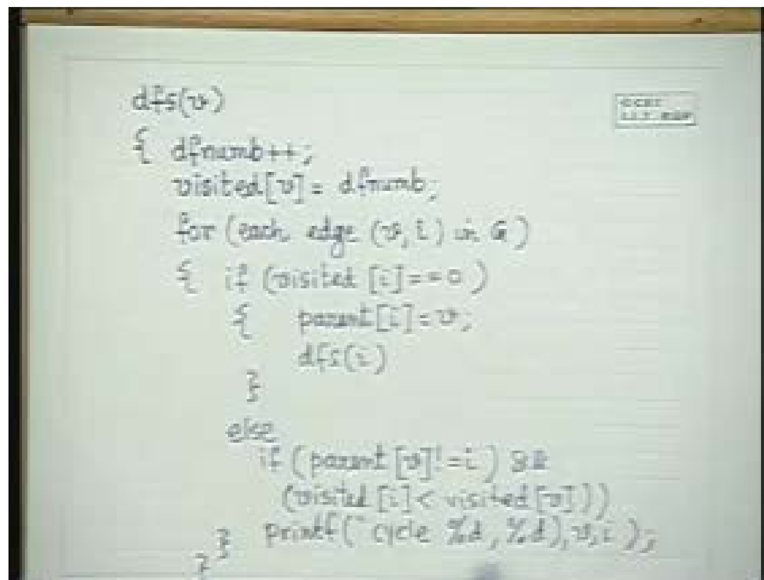
By marking including all these three we are now ready to find out whether there is a cycle. So note the dynamic programming on the concept of remembering something or trying to do data structuring where you remember, first you remember whether you visited it before. Now you are trying to remember also who is the parent. So you are trying to keep more data while doing the search. So data structuring during of the search is as we mentioned, we are done some simple examples earlier now we come to some concrete cases where we will require it. The algorithm let us write it down and develop it slowly then we will get. The algorithm is called depth first search dfs v.

What we do is we have a depth first number this is a global array df numb, this is incremented initially it was 0, instead of marking this node visited now we don't have any goal t, we are only at a node and we are trying to search everything. For each edge,

so instead of visited equal to 1 or 0 we put the depth first number. For each edge v, i in g if obviously visited this part remains visited i is equal to equal to 0. we search it but before we do dfs (i) that is we are going to do dfs (i) but before we do dfs (i) we will remember that the parent of i is equal to v.

So if it is not visited we do this else we come to the cycle detection part. And in the cycle detection part what do we have? If parent v that is, that means we have already reached visited. Now if v, parent of v is not equal to i that means there is no direct edge like that.

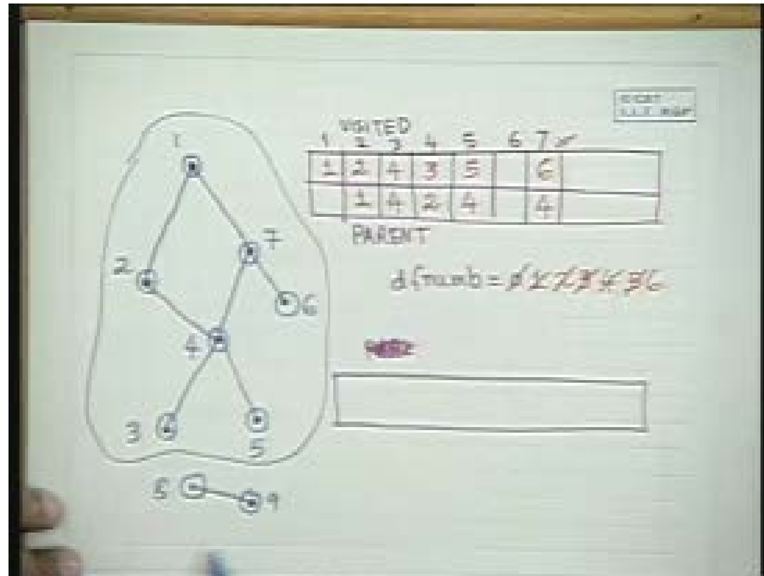
(Refer Slide Time: 00:26:03)



If this is true and i was visited before v obviously, if i must, if i is visited then i must have been visited before v, previously 1 or 0 was the case. Now since v is visited already and i is also visited then i must have been visited before v and visited i less than visited v then printf cycle between this and this, between v and i and you close your brackets. So this was our cycle detection part. The earlier part here we have just put in a number to indicate which is visited before what, alright and this part remains the same, only we have put in the parent concept here and this detects whether there is a cycle. Here just it should not be the parent and visited i is equal to visited is less than visited v.

So let us work it out on an example. So that is now we have started incorporating our ideas of dynamic programming. An incorporated, we are now moving more deeply into the problem. Once you move more deeply into the problem you start getting into the group of graph algorithms. You talk in terms of depth first search only, you forget about that we are using dynamic programming etc because once you get used to these ideas these will come naturally. So let us see what happens when we use this algorithm.

(Refer Slide Time: 00:34:30)



Suppose we take 1 2 3 4, I am purposely giving some other numbers so that the depth first number, so the visited array will be marked in pink that is the visited array number will be marked here. So visited this is an array of 7 nodes, initially all are 0 I am not writing it down, 1 2 3 4 5 6 7 initially all are 0. df numb is initially 0 and there is the parent array so let the down part of it be the parent array. There is the parent array which is also 0, all of them are 0, this is vertex 1 2 3 4 5 6.

So let us start with vertex 1. df numb will become 1. See what is the first idea? df numb becomes 1 then visited and then loop inside and go on. So the algorithm if you have noted down, I think the base thing would be to note down this algorithm by hand and then keep it at your hand and once we keep it with you, you can follow what we are doing. So first please note down this algorithm. Then, so first df numb is made 1 then visited 1 is made df numb, so this is made 1. Then for each edge these two, vi 2 and 7 start is if visited 2 is 0 yes, visited 2 is 0. So you move in and make parent of 2 equal to 1 and call it recursively. So you have called this then here again you go into the loop, you increment df numb to 2, visited this make it 2, come here.

Now this has got this edge and this edge. For this edge visited i is equal to 0. So you come to the else part that is in the algorithm, here for each edge this visited i is already done, it is not 0 because visited 1 is 1. So you will come here and check whether it is a cycle. Parent of v which is 2 is not equal to i, here parent of 2 is 1. So this condition is falsified therefore cycle detection will not be done, this is not a cycle so you go back into the loop and you forget about this edge. So let us see how we have forgotten about this edge.

Now once we have forgotten about this side, we come to 4. This is the other edge so we come to vertex number 4 and before you come to vertex number 4, the rent of 4 is made 2. Now here I knew recursively go to dfs 4, once you go into dfs 4 dfs numb increases,

this number becomes 3. Once this number becomes 3 you start doing dfs here. So you will do 2. This one again this is visited and you will see the parent of 4 is 2, therefore this edge cannot detect a cycle. So you will come here, so you will come to 3.

Now in 3 you will see, you will mark 4 to be the parent of 3, you will increment df numb to 4, you will come here mark this visited 4 equal to df numb, search from 3, from 3 there is only 4 but the parent of 3 is 4. Therefore cycle cannot be detected here. Now 3, from 3 there is no where else to go so we will come up, come back to 4 come to 5, 5 is not visited so you will mark 4 to be the parent of 5, df numb will become 5, visited 5 will become 5.

Here you will check with 4, this is already visited but parent of 5 is 4 so this cannot be a cycle. So you will come back out of the recursion and go back to call of 4. Once you go back to the call of 4 then you have already checked this side, all these 3 done so now you will come here. Now once you come to 7, 7 is not visited, so you will mark 4 to be the parent and start searching 7. So number will be 6, this number will be 6.

Now here you will start searching for 7. For 7 the first child is 1. The first child is 1 is visited. Now once 1 is visited you come here. Once you come here, you will able to see that parent of 7. Is it 1? No, it is not 1 and visited 1 is less than visited 7, visited 1 is 1 and visited 7 is 6 so this is true. Therefore there is a cycle between 7 and 6. So there is a cycle between 7 and 1. So now you have detected the cycle at this point. So this is how you see that we are able to detect the cycle at a node. So using the concepts of dynamic programming, we are slowly moving on to a very important algorithm of depth first search in graphs. In graphs this concept of depth first searching is very important and the data structures that are maintained, the parent, the visited, the depth first number etc etc are very useful.

Now let us see if this depth first search, can you find out whether, so we have answered two queries. One is whether there is a path from s to v, s to t. We have also found out whether there is any cycle both we have found out in order n plus e time. I am not working out in details but you can find out that we have found in order n plus e time. Now let us see what is the concept of a connected graph or graph is said to be connected if there is the path from every node to every other node.

Now if you apply depth first search here, you will move to every node to every other node. But suppose there was a vertex 8 and 9, you start depth first search from anyone, anywhere here you will never come here or you start anywhere here, you will never come here. So what will happen is at the end of depth first search, if you start from this cluster you start anywhere from this cluster, after the completion of depth first search, the depth 8 and 9 will have visited equal to 0. And if you start from this cluster, after the completion you will have for all of them visited will be 0.

Therefore to check whether a graph is connected, all you have to do is to perform depth first search. And once you perform depth first search and then check for all the vertices if anyone is not yet visited, if anyone is not yet visited you can say that the graph is not

connected. Therefore we have solved several problems, one is whether a graph has got a cycle, whether there is a path, whether there is a connected path. Now you can try and solve these problems for directed graphs as well. The algorithm will slightly change but the depth first variation will be similar.

The next question is shortest path, short length path how will we solve that problem. So we have to look at that problem. So we will look at that problem in the next class but let us remember that there are other issues which we can solve in here. Suppose is marking cities and path between cities or telephone lines and we would like to ensure that the graph is such that even if one link fails, there is still it is connected between every other node. How will we solve that problem and how will we do that efficiently.

We will see the depth first search and other types of search like breadth first search etc will be very useful in solving search graph type of problems. So I would suggest that you go deeply into depth first search and try and see these algorithms and graph. In graphs we have got a number of algorithms to solve where these techniques of dynamic programming of recursive definition, even of balancing will come into the picture, techniques of branch and bound will also come into the picture.

In fact, when you are trying to solve the shortest path length, shortest length path in a graph you will see how you use dynamic programming to solve the problem. So today we will conclude the class by saying that we have tried to see graph algorithms that is the data structures of graphs can be represented by adjacency matrix or adjacency list or direct representation and depth first search is a very useful way of answering a number of connectivity especially path related problems.

For edge and node related problems, it is obvious and here in such problem the adjacency list representation is often useful. There are other problems where the adjacency matrix representation is also useful. So this is now we conclude our initial study of graphs. Graphs is a very detailed subject, lot of issues remain in graph algorithms but our idea was to study how algorithm design and data structures and programming methodology goes hand in hand to solve a number of application areas. And graphs is one area when all these will go on together. So we do not have the scope or the time to discuss graphs in absolute details therefore we conclude here. I would refer to you to any standard book on algorithm design because the end of this course really moves you more deeply into algorithm design. We have just done a bit of data structuring, deep analysis algorithm analysis of algorithms, design of algorithms which have to go hand in hand will come up. So we conclude our initial discussion on graph algorithms today.