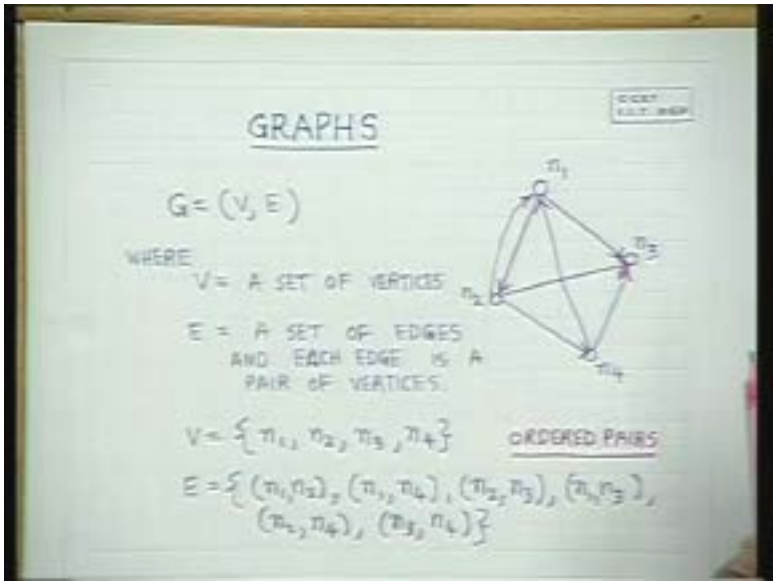
Programming & Data Structure
Dr. P. P. Chakraborty
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture # 30
Graphs – II

We continue our study of graphs. We have seen in the previous class that graphs are useful to represent a large number of problems and we saw several problems in the previous class and we came to the conclusion that graphs is an important data structure and it is a data structure which can be used to solve a number of actual real life problems. So what did we look at in graphs?

(Refer Slide Time: 4:55)

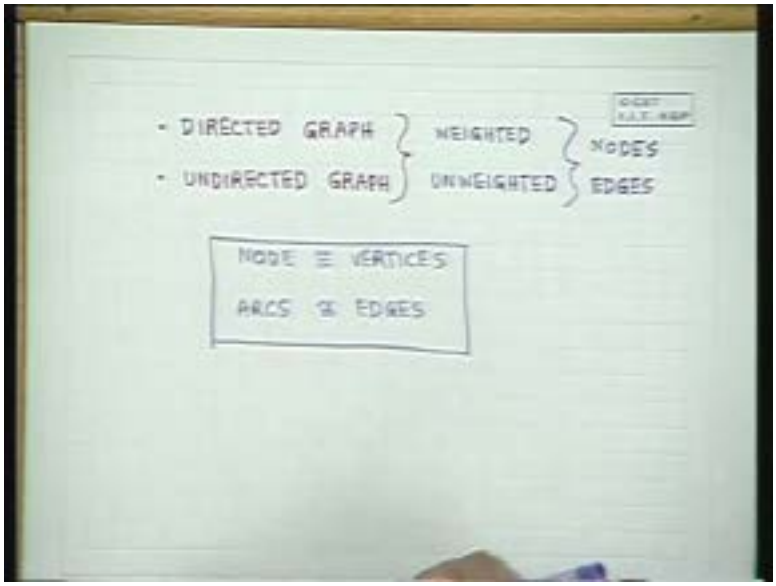


We define a graph g to be 2 items. Let us take an example first. A graph consisted of nodes and edges, say this is n_1. A graph consist of a set of nodes or a set of vertices V and a set of edges E where V is a set of vertices and E is a set of edges and each edge is a pair of vertices. For example here in this graph V is equal to n_1, n_2, n_3, n_4 and E is equal to (n_1, n_2), (n_1, n_4), (n_2, n_3), (n_1, n_3) and (n_2, n_4) and (n_3, n_4), 1 2 3 4 5 6 edges are there, so there are 6 edges here. Now, here this pair is an unordered pair in the sense that n_1 n_2 is the same as n_2 n_1 but because this is an undirected graph, graphs can also be directed and directed graphs are represented by arrows, every edge will have an arrow. Now if every edge has an arrow then the pairs here will become ordered pairs and if these arrows are there then the edges will be n_1 n_2, n_2 n_3, n_2 n_4, n_4 n_3, n_1 n_3 and n_4 n_1 instead of the way it is written here. Sometimes unordered graphs or undirected graphs are represented in a directed fashion by writing both the arrows. If you have only one edge, you represent them by putting in both the arrows.

So very or one convention of representing graphs is by representing directed graphs so that for an indirect undirected graph you represent it by two directed arrows, if there is an edge between n_1 n_2 you represent it by two directed arrows. So our graph maybe a directed graph or an undirected graph. That is you can have a direct un a directed graph, undirected graph and for both directed and undirected graphs as we said before you can have weights on the vertices as well as on the edges. That is in some cases you can have weights on the edges, in some cases you can have weights on the vertices and in some cases you can have weights on both.

So directed as well as undirected graphs maybe weighted or unweighted and weights can be on both nodes as well as edges. We use the terminology nodes and arcs. So node is the terminology we use with vertices and arcs is the terminology which we use with edges. Now graph theory has got a lot of applications because graph theory is the most, one of the most well developed and well-built theories.

(Refer Slide Time: 00:06:47)

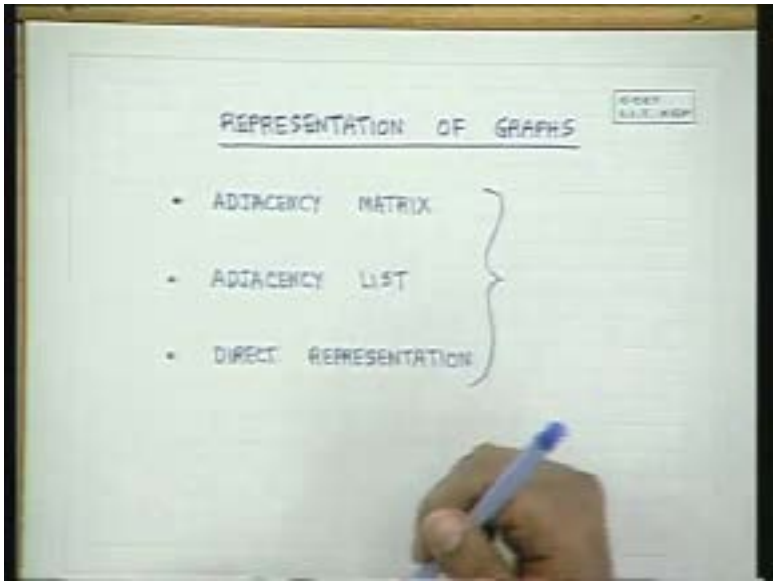


Now in graph theory you have got several results related to graphs, planar graphs, non planar graphs and several other issues are involved. Here we shall not go in deeply into those issues of graph theory and graph mathematics. What we are interested in here is graphs as a data structure and we shall see some algorithm or queries on graphs and try to solve those queries on graphs. So in terms of a data structure, we need to know how to represent a graph as a data structure and how to have answer some queries, some simple queries on graphs.

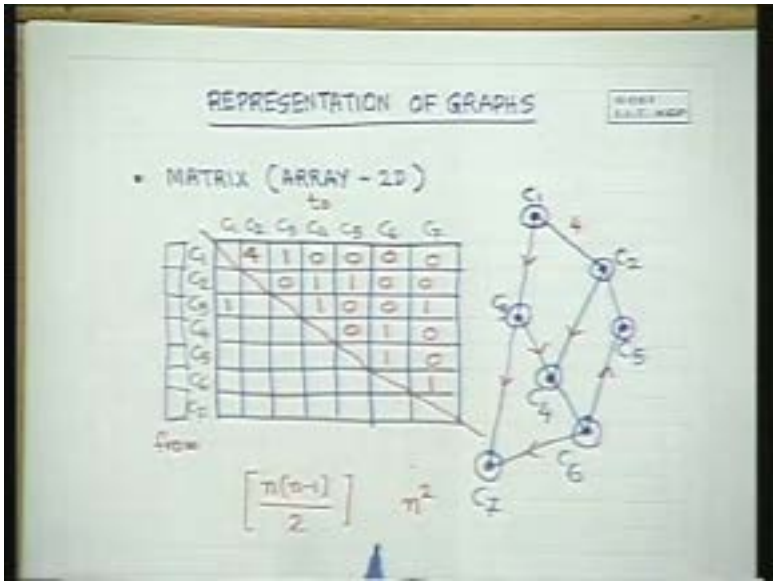
There are several complex queries and complex problem that can be solved on graphs but we shall try to take examples of the basic simple data structure queries on graphs and then see how our algorithm design techniques that we have developed can be used effectively to design these functions efficiently.

In graphs we have seen how to represent graphs. We saw that a graph can be represented by an adjacency matrix. In an adjacency matrix which we saw in the previous class, a graph is represented by an array, an array of cities in this example, so an array of nodes actually and an edge will be 1 or 0 depending, a cell will be 1 or 0 depending on whether the directed edge c_1, from c_1 to c_2 is there or not there.

(Refer Slide Time: 00:09:31)



(Refer Slide Time: 00:11:56)

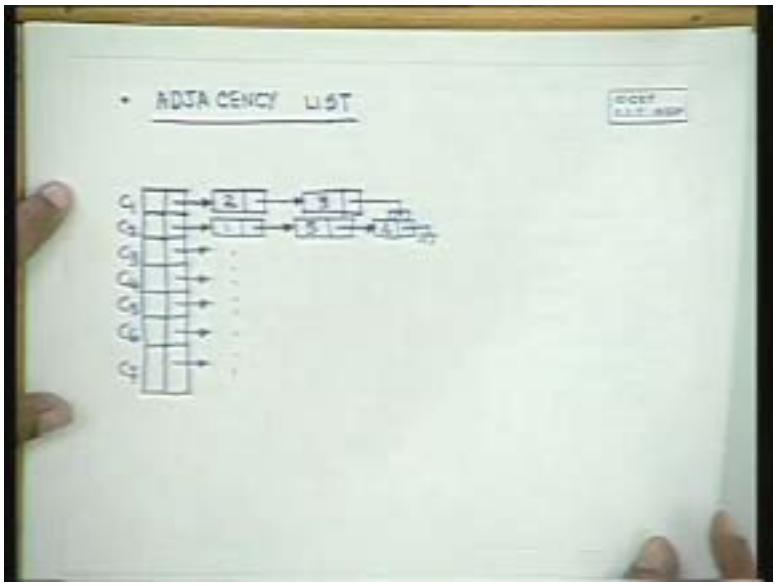


And as we mentioned before undirected graphs will be represented by marking out both of them or by mentioning that it is symmetric about one axis and directed graphs can be easily represented.

Weights on graphs that is weights on the edges can be implemented by putting the weight values on the cells and weights at a node can be implemented by maintaining a separate information with every node and marking the weight of the node, only in one place this can. So a separate array to mark the node weights and this actually mark the edge weights, this is the edge weight matrix. So this is the adjacency matrix representation of a graph.

The next is the adjacency list. In the adjacency list representation, we represent it on set of nodes where the node numbers, node names as well as their weights we put here and pointers which mark the edges and c_1 that is node c_1 is connect to, node one is connected to... say if this is node 1, it is connected to node 2 and node 3.

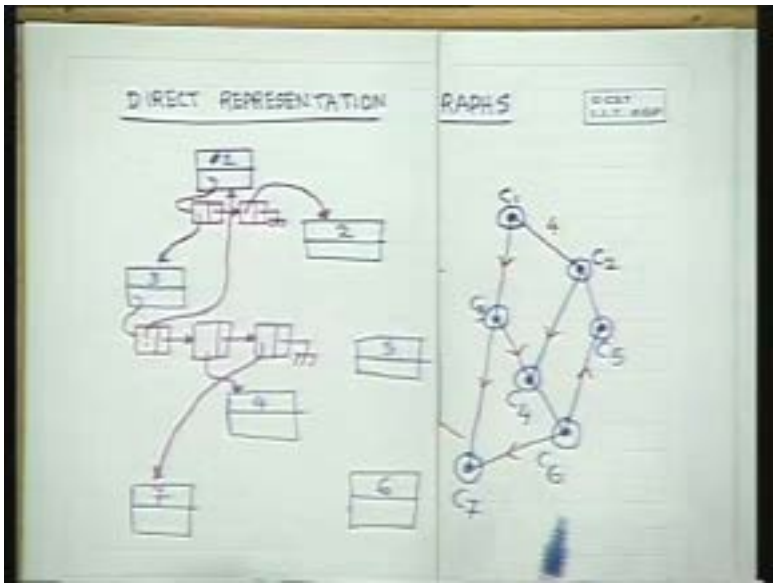
(Refer Slide Time: 00:10:00)



Node 2 is connected to node 1 and node 5. As we mentioned in an indirected graph, if we want to represent will represent both the edges. So in an adjacency list, we represent all the edges and therefore for a sparse symmetric, this is a good way of representing the information because we need only the edge information and here most of it would have been 0 but if it is a dense matrix then it is a very good way of doing it because access to an edge is direct whereas here to access whether an edge exists is indirect.

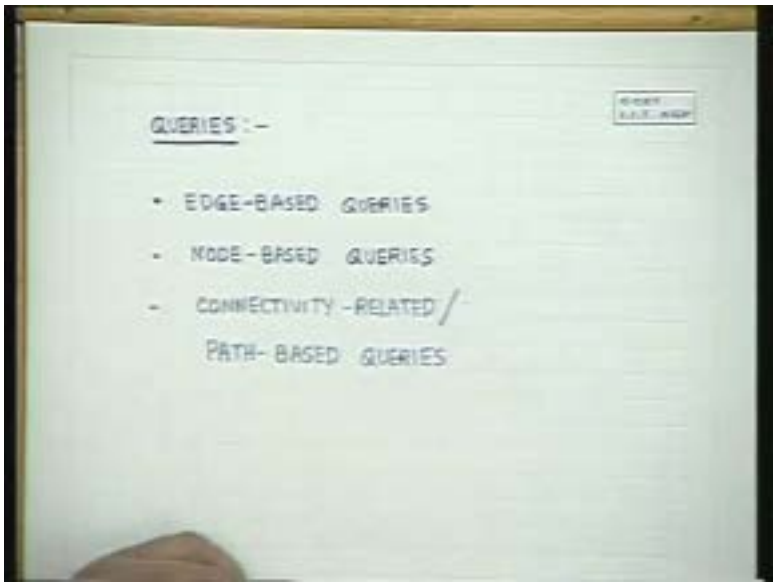
The third is the direct representation. In the direct representation, we represented the graph as it exists like this. That is if you recall what our original graph was then in the direct representation from c_1 we put in a set of linkage nodes and link it up. This is connected from this to this and this to this, so we linked it up. Similarly this is connected to 3, 1 2 and 3, so we linked it up. So this way we got a direct representation on graphs.

(Refer Slide Time: 00:11:31)



So we discussed mainly these three representations on graphs and these can be represented in many ways. Next we try and answer things like certain queries on graphs and what are the queries on this data structure graph. Edge based queries, node based queries, connectivity related queries or these are also called path based queries.

(Refer Slide Time: 00:13:06)



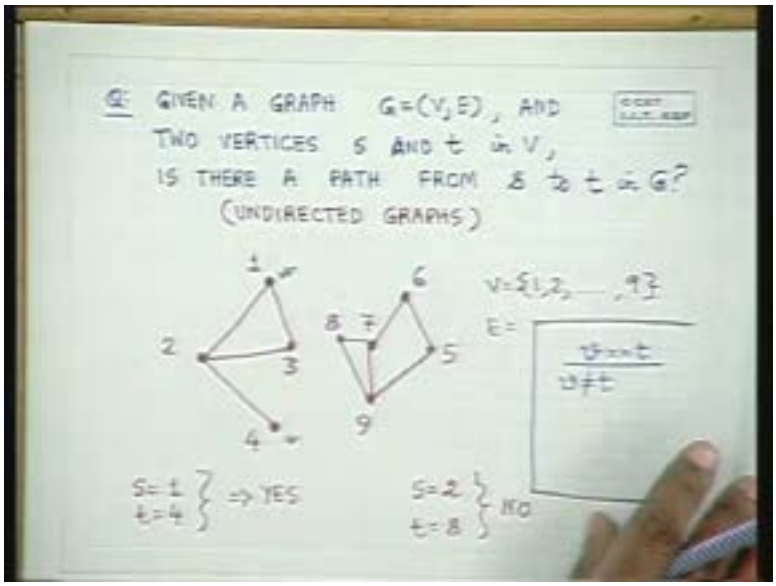
So a query maybe of various types, it can be a node based query where you ask questions relating to nodes only, what is the weight of a node, is this node connected to any other node etc. Edge based queries does this edge exists, if it exists what is the weight of the node, weight of the edge. These are simple queries. On the other hand you can have path

based queries which are more complex and it says that if the, is the graph connected in the sense that is there a path from every node to every other node or in a simple question is there a path from one node to another node, given a node number say node number 5, is there a path from node number 5 to node number 7, etc.

So we will see what sort of queries we can answer in such situations. Edge based queries can be simply answered and let us see which data structure is most efficient. Suppose I asked whether there is an edge or not an edge? In an adjacency matrix, it is simply checked by checking whether that element exists and we can extract that weight out whereas in an adjacency list, we have to go through. In an adjacency matrix answering that query is direct, you just access it at in a constant time you can answer a single edge query.

On the other hand in an adjacency list, this would be, you would have to go through it one by one. Similarly for a direct representation, you would have to access it, if you ask it from a node again you would have to access it and find out whether that edge exists. Therefore if you have pure edge based queries either you will implement it by a matrix representation or you will implement a data structure which is more related to edges. Node based queries can be easily answered by asking out questions related to any node. Therefore node and edge based queries are simple and they can be answered easily. We come to the path based queries and the first query we ask in a path and is that is there a path from vertex x to vertex y then we will ask other questions. So let us first try and see how we will solve the problem of answering the query on the graph data structure of is there a path from vertex, given a vertex s to a vertex g in a graph capital g, is there a path from s_1 to s_2 say. And we will consider undirected graph.

(Refer Slide Time: 00: 20:16)



So our query is given a graph G equal to V, E and two vertices s and t in V is there a path from s to t in G. For example, let us say we have got a graph and we are considering here

only undirected graph. So let us say we consider only undirected graphs, let the numbers be 1 2 3 4 5 6 7 8 and 9. Now this is an undirected graph, you can write it out with V is equal to 1 2 up to 9 and E you can list out all the edges. Now is there, suppose I give you 1, s is equal to 1 and t is equal to 4, the answer is yes because there is a path but if I give you s is equal to 2 and t is equal to 8, the answer is no. So how do we find out whether there is such a solution to our graph?

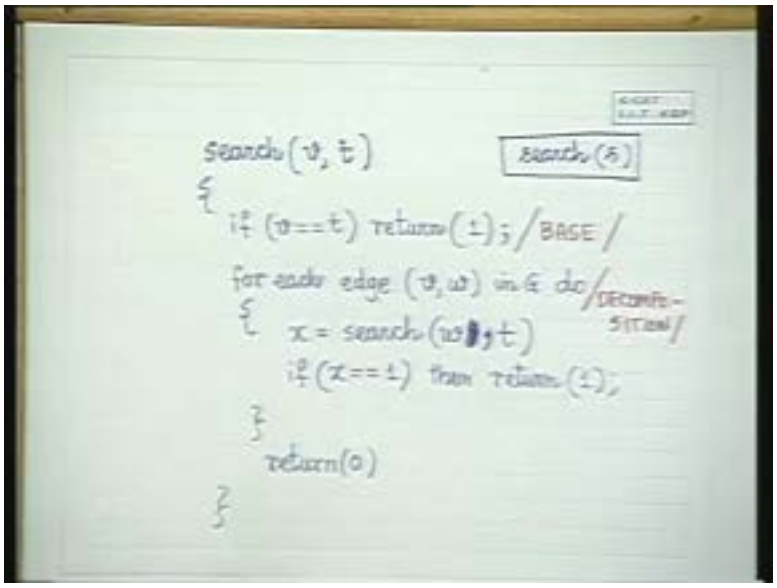
So now we start using our algorithm design techniques to solve this problem and what is the basic idea in our algorithm design techniques. The first idea is get a recursive definition to the problem and try and analyze that, then try and analyze that recursive definition and after you analyze that recursive definition try and use the techniques of algorithm design and data structure to solve the problem and get the final result. So let us see how we would solve the problem and how we would decompose it in a recursive fashion. Suppose we have to start from one of particular vertex. A recursive definition of this would look like this.

When you are at a node say v, if v is equal to t then you can terminate because if you have started from v and v is equal to t that is somebody is given you 1 1, the answer to 1 1, s is equal to 1 and t equal to 1 is obvious. Now if this is not so, then what do you say? If there is a path from v, so v is not so, case 1 is this, case 2 v is not equal to t. If v is not equal to t then look at the successors of v. For example suppose this is v and our goal is this. So 1 is not equal to 4.

Now if there is a path from 1 to 4 then there must be a path from either 2 to 4 or 3 to 4. That is if there is a path from 1 to 4 then and 1 is not equal to 4 then there must be a path from anyone of the successors of 1 to 4 and if there is no path from anyone of the successors to 4 then there is no path. So from this idea we get our basic recursive definition idea and the concept is search in a vertex v you are trying to search whether you have reached t and you will start with s, you will call it with search s. This is how we will call it. I am not writing out the definition, the data structure and the declarations because we are now trying to just do the decomposition. We are trying to say what is the recursive decomposition.

What is the basis condition if v equal to equal to t, return 1 that is we have found. So now v is not equal to t, so this is the base condition. We said that we will have the base condition and the inductive condition that is the recursive conditions. That is what we did in our recursive definition. If this is not the case then for each edge v w in G do, x is equal to search, so, we try to search w and see what is the result.

(Refer Slide Time: 00:23:38)

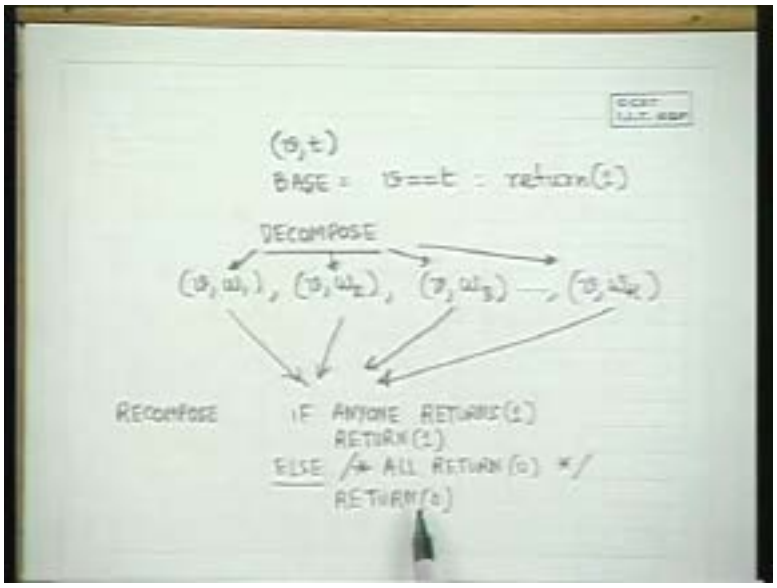


If x is equal to equal to 1 that is if there is a path sorry, here also t should be there because t is an argument, t is an argument so from if v is not equal to t then try to search from the edge w to t and if you return with x then return with 1 because if you have found the path from w to t then you will get it. And you continue to do with all the edges and if you come out here that is you have not returned anywhere then obviously you have not found an edge from any of the successors of v neither is v true, so you return 0. So this is the basic recursive definition and here is the decomposition.

So what have we done, what is our algorithm design technique? Our algorithm design technique as we mentioned before consist of decomposite base condition, decomposition and recomposition. So what is the idea that we have followed here. The idea is that either that node from that node we have got, we have reached our goal or from that node we will try and find out successors and for all the successors we will see if the successors can reach the goal. If the successors can reach the goal then we have found the goal, if the successors cannot reach the goal then there is no goal from this node itself.

So the problem decomposition and recomposition structure would look like this from (v, t) we have got the base condition. Base condition is v equal to equal to t, return 1 otherwise decompose. If there are (v, w_1) if these are the edges (v, w_k) if there are k edges then you call it recursively on all of them and if anyone of them returns 1 the recomposition… what is the recomposition?

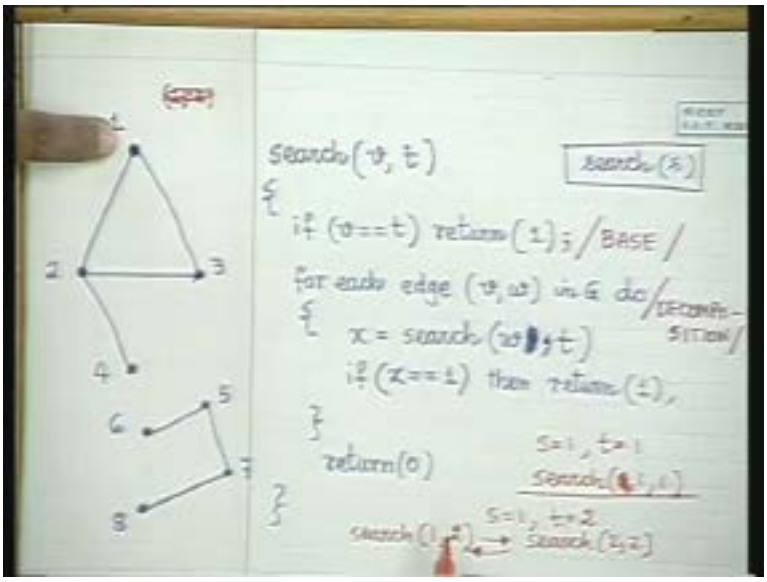
(Refer Slide Time: 00:26:01)



If anyone returns 1, return 1 else that is else means what? Else means that all return 0 then return 0, so what does this mean? This means that either t is the goal or from t anyone of the successors you must reach the goal. Therefore this is a correct decomposition; a proof of this is obvious from the inductive definition. So from the inductive definition, we can easily prove that this algorithm is correct but once we go to implement this algorithm that is this definition is correct, this problem decomposition style is correct but once we go to implement this algorithm, we shall see what are the issues that will crop up.

So let us next try and implement the algorithm. To implement the algorithm let us take in an example and let us take back the example that we started previously 1 2 3 4 5 6 7 8 and let us see what our algorithm is. Is it visible? So let us try and work it out on this example and let us say the first query that we give is try to find out whether there is a path from 1 to 1. So let the first query be s is equal to 1, t is equal to 1. So this algorithm will be called with search s 1 1, search 1 1. So here you will come to 1, you will get this you will return true, so this problem is easily solved.

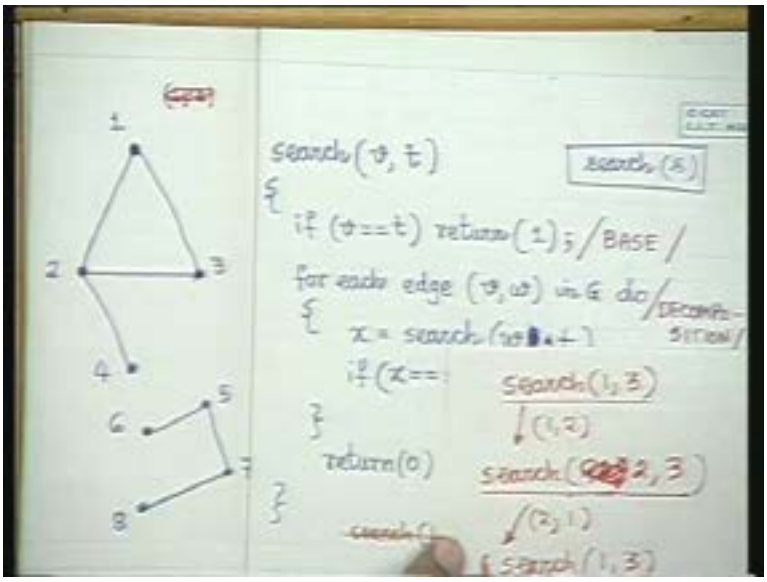
(Refer Slide Time: 00:29:47)



Suppose I give s is equal to 1 and t is equal to 2 then suppose we have the situation that s is 1 and t is 2. So you will call it with search 1 2, you will come here search 1 2, this is not equal to 1 is not equal to 2. So for each edge, now suppose we start with this edge then you will call it here, suppose we start with this edge 2 first and 3 second, so let us start here you will call it here. Once you call it here, you will come and you will get, you will call search 2 2 so that recursion will come in, it will call 2 equal to 2, it will return 1.

So once it returns 1, it will return 1 back into search 1 2, so this will return 1 back and the problem will be solved but if this does not mean the algorithm is perfectly correct because here if in search 1 2 for each edge, now if we did this edge before this edge because we will be doing it in a particular order and suppose I do it in the order 2 and then 3 then suppose the problem which is defined is this. Let us consider a definition which tries to solve this problem. Search 1 3, here it will come with search 1 3, it will come here this is not true for each search, so it will first call search 1 2. To search 1 2 it will come here, it will come here once it comes here sorry it will not search 1 2 it will search 2 3 because (w, t), w is 2 for the edge 1 2, w is 2 so it will call search 2 3.

(Refer Slide Time: 00:31:34)



Now once it call search 2 3, it will come into this, 2 is not equal to 3 so then again it will call for the edge. Now 2 has an edge 2 1, so it will again call since there is an edge 2 1 because an undirected edge is represented by 2 directed edges, it will call search t, w is 1 and 3. Now see you have ended up in a, you have got the same sub problem back and this will again call 2 3, this will call 1 3 and you will go into an infinite loop.

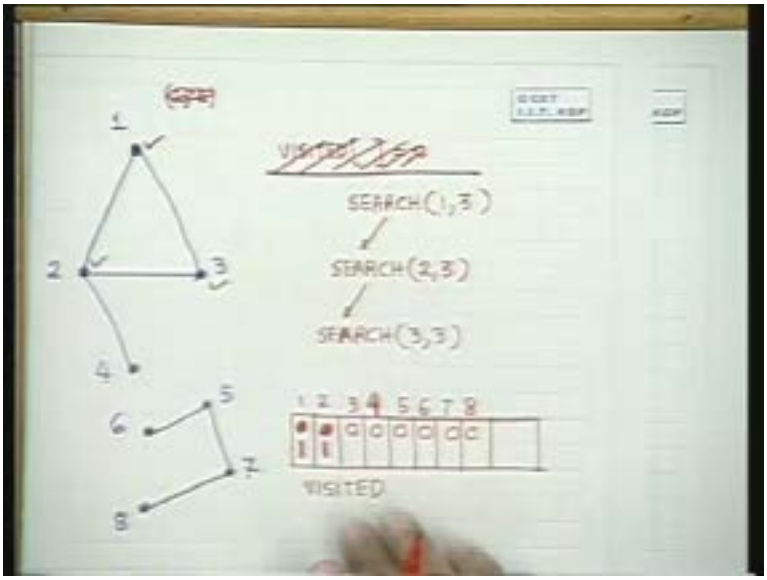
So the problem with this if you notice is the problem possibility of going into an infinite loop. I repeat, to search 1 3 you will start here then you will come here then from here again you will go back here then you will come here and you will continue in this loop though, so the algorithm will not terminate. That is the algorithm will not finish off properly. What do we do in such a situation? What is the problem that has cropped up? We have reached an identical solution and for this identical sub problem, the algorithm will continue recurring. In order to handle this identical sub problem, what do we have to do? If you remember our dynamic programming strategy, all that we have to do is remember that we have come to this.

Now let us look at this problem. If we have already come to a node that is we have already visited a node that what does it mean. It means that if you come to it again you will be trying, from here you will be trying to solve the same sub problem again, a sub problem that we have solved before. In the Fibonacci problem it was just repetition of work that is the algorithm did not become incorrect. Here it is not only mere repetition of work, it is repetition in such a way that your algorithm will go into an infinite loop. Therefore dynamic programming not only makes it efficient, here is an example where it becomes absolutely essential. So how do we use dynamic programming?

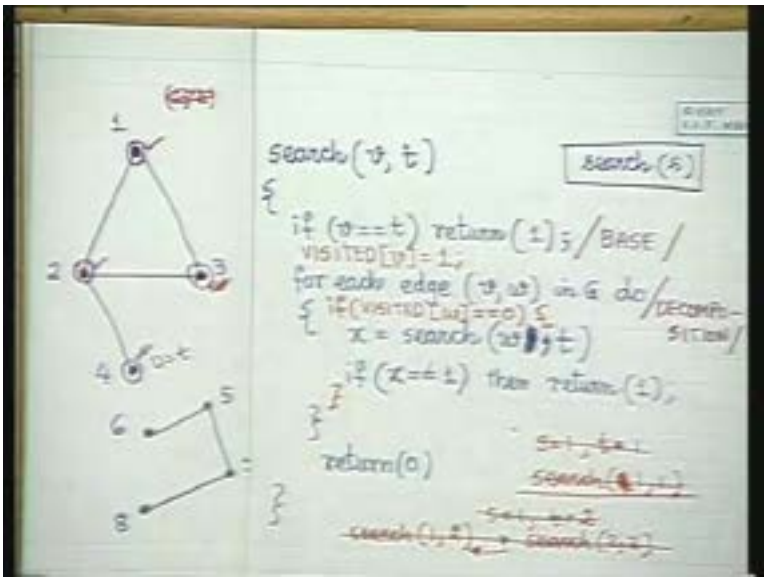
So let us see. We will have to main remember as we said before done array, we will have to remember that we have done this thing before. So here we will call it a visited array. We will maintain an array visited of vertices. Here this will be an array of vertices,

initialize all of them to 0. Now whenever you come to a node, you say it has been visited so let us come here. This part is ok, for each edge in we g do, before this you say visited v it's a global array is equal to 1 that is it is visited. So once you come and mention that it is visited, now for the edge all that we have got to do is mention here if it is not visited only then you will search.

(Refer Slide Time: 00:38:17)



(Refer Slide Time: 00:40:00)



So if visited w is equal to equal to 0 only then will you search and do anything. That is if it is visited only then will you try and do it, if it is not visited you will not do it. So by putting in the concept of this visited node, we are now able to remember that we have

come to this node before and let us see what effect it has on our algorithm. Let us try and resolve our problem as we were trying to solve that is we were trying to solve search 1 3. Let us see what our visited array is. We have got 8 vertices, initialize all of them to 0, this is our visited array.

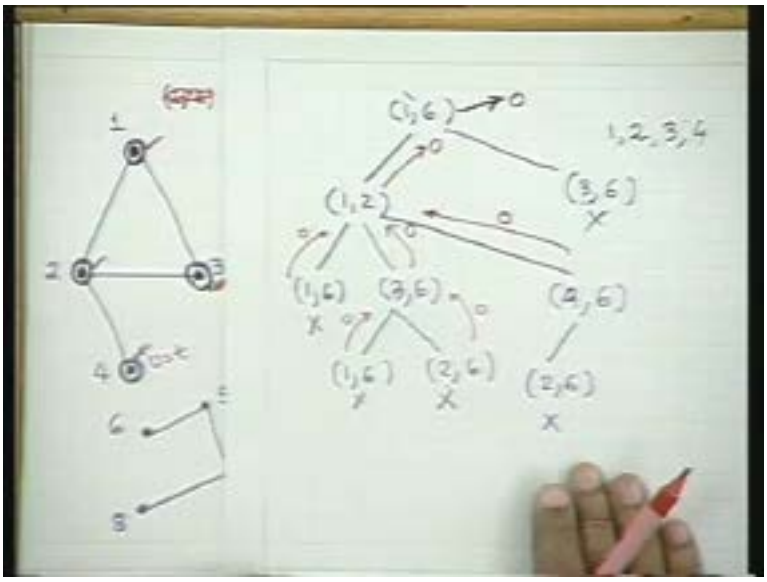
Now when I come here, the first thing that I do is v equal to t, no, because it is 1 3. Then I make visited v 1, v is 1 so visited 1 is made 1. Then for each edge v w, so 1 to 2 in v do, if visited w is equal to 0, so is visited 2 equal to 0, yes. So if visited 2 is equal to 0 then I do this part as we were doing, so I visit this. So I call it recursively search, so I call search 2 3. Now in 2 3 what do I do? In 2 3, what I do is I come here, again I see is v equal to t, 2 equal to 3. No, then I make visited 2 equal to 1, so I make visited 2 equal to 1. Next for each edge v w, so one edge I have to see, so I come here and see this edge. So what is the edge 2 1, if visited w equal to equal to 0 is visited 1 0, no, visited 1 is 1. So I do not make this call and once I do not make this call, why because I have remembered that I have already made this visit.

Since I have made this visit, I do not make this call and I come out and the next edge is 2 3, so I have made this visit. So, now I check for 2 3, is visited 3 equal to 0, visited 3 is 0. So I call search 3, once I call search 3 3 and then I come here and I finish, I have obtained my goal. So using this technique, we are able to avoid the problem of a loop. So you now see that using the technique of dynamic programming, we have been able to make our algorithm correct.

Now let us see how this algorithm works and let us see whether this algorithm is efficient. Now let us see if I give 1 4 then it will come here, from here it will come here, here. So now we will work it out without this algorithm, we will see very quickly. Whenever I visit a node I circle it, so 1 4 I visit this then will come here, I will visit this then I will come back here but since I have come here I have already visited it so I will come back, I will come here, I will visit this, from here I will come here, I will come back because this is visited I will not do it, from here I will come here, this is visited I will not do it (Refer Slide Time:. So I will return 0 because nothing will work here, so I will return 0.

So this return 0 here so this continues, this has failed here, this has come here gone in and failed, this will come here and worked this out. Here it will find v equal to t and find goal. So this is how from any node to any node, if there is a connectivity this search algorithm will do it correctly. So now suppose I give you 1 6 then what would have happened? Let us do it a different color and start, this would be visited first then this would be visited, this would say it is cannot be done, it would come here, this would be visited it would go back. This would come here, so this would return 0, this would come here, this would come here, it would visit this, come back return 0, so everything would return 0. So let us drop the search tree, once we drop the search tree here we will see how search 1 6 fails.

(Refer Slide Time: 00:43:02)



Suppose we are given search 1 6. To search 1 6 we will start searching with 1 2, so 1 is visited, 1 is visited, we will come here, 2 will be visited from 2 we will try to see 1 6 but since this is visited, we will come out, we will come and see here and we will see 2 2 3 6. 3 6 will mark 3 visited and then 3 6 will try 1 6 but once it comes here 1 is visited, so it will not do. 3 6 will try 2 6, 2 6 2 is also visited so it will not do this. So this one, this one will return 0, so here 0, x is still 0 nothing happens to x. This will return 0 because if you notice that if no edge it becomes 1, it returns 0.

So this will come to here, 1 will come back and try 1 3 sorry 2 will continue, 2 has tried 1 2 has tried 2, 2 will continue and try 2 and try 4 6, 4 6. 4 6 on the other hand will try 2 6 and fail and it has got no other edges, so this will fail and return 0. Once this returns 0, this 2 will return 0, once 2 returns 0, 1 will try 3 6 but again 3, so here 4 had become visited. Again 3 is visited, so since 3 is visited this will fail and therefore this will return 0. So if there is, see you will see that you are visiting only those nodes which are connected and we are not visiting any other node and also we are visiting a node which is an every edge is visited only once.

So, we will see that how this technique of using dynamic programming has helped us to solve the problem of graph connectivity. This type of search using a recursive style, using the dynamic programming of called making it visited is called a depth first search of a graph and of an undirected graph. In a directed graph also you will have depth first search. In an undirected graph, you will have this concept of depth first search and this depth first search as we shall see in the next class is efficient and also is useful in solving a large number of problems. So we shall see that in the next class.