Course: Introduction to Graph Algorithms

Professor: C Pandu Rangan

Department: Computer Science and Engineering

Institute: IISc

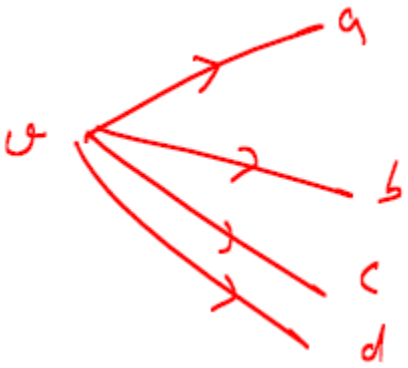
Week: 07

Lecture 31 Iterative DFS

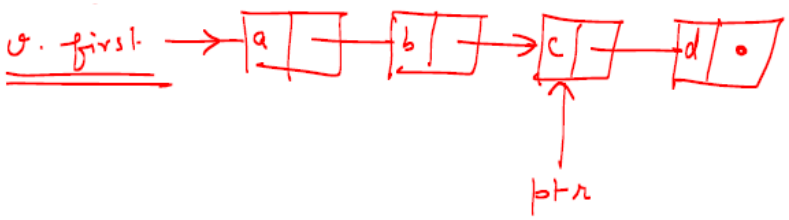
Namaskara, today we will discuss an iterative method to carry out the depth first search. Depth first search done in a recursive form is simple, direct and even intuitive, that is because if we have DFS G u what we simply do is that for each v belong to adjacency of u ,if v is not visited. then we perform DFS Gv. We go to an unvisited neighbor and do the same thing. Therefore it is a handled through recursion, vanilla version the basic version and essentially this is what we do but we do several other bookkeeping information also and As we perform the depth first search we compute the related information. For the sake of efficiency in certain context we need an iterative version for the depth first search. The iterative version is a bit tricky.

This is because we have to maintain and manage explicitly the transfer of control return of control after finishing a job choosing the next task to be done all of them are to be done explicitly by us. Since simulation of any recursion requires a stack to prioritize the to order the sequence in which we perform the computation. We maintain explicitly a stack and then produce an iterative version for this. However, returning to the point from where we have digressed and continuing with the next item to be explored is to be handled in a subtle way. So we are going to see the details related to that now okay.

Iterative DFS or non recursive DFS. So, we assume that we have G represented as an adjacency list, each vertex has got all its neighbors given in an adjacency list. We denote in our pseudocode the adjacency list for v in the following way, v dot first is a pointer, pointing to the first element in the adjacency list.



So it is a name of the list is called v dot first for the vertex v the first neighbor is indicated by the pointer v dot first for example if v has several neighbors a b c d this may be a and this is connected to b and this is to c and this is to d and is a null pointer, null pointer is denoted by a dot, sometimes you see a notation something like that these are all used to denote the last box will not have any further link.



So these are all neighbors of v, out neighbors from v you have the edges going towards a. If the graph is undirected each edge will give rise to 2 boxes because there is no direction, an edge will create an entry a in the adjacency list of v and then entry v in the adjacency list of a. So each edge will give rise to two boxes one box in adjacency list of v another box in adjacency list of a however for a directed graph for each edge there is only one outgoing vertex and that will give rise to the box as indicated here.

So if we have several neighbors all the neighbors are put in a list the order really does not matter whatever is the order in which the list is created is the order in which we would consider their neighbors okay. A specific position in this list can be denoted by an independent pointer. Let us say we use ptr as an independent pointer to indicate a specific position in the adjacency list. For example, I may have ptr, ptr is an independent pointer, the reason we may require one such pointer is we are visiting various edges and vertices and when we are done with one neighbor then we have to consider the next neighbor.

Therefore we have to indicate the position of the next item to be processed for that we use a pointer in the adjacency list, so each adjacency list will have a running pointer that will indicate a position in that particular list. So, v ptr is the notation for a position indicated by the pointer ptr in the adjacency list of v. In the adjacency list of v we use an

independent pointer, so this pointer can be used to represent the current neighbor that is being looked at, so ptr is pointing to vertex c okay.

A box in the linked list will have the following structure, a vertex and next is a pointer to the next box, so this is a pointer, this is a vertex of the graph.

So the linked list is made up of such boxes they are all chained up and you have a linked list of adjacent vertices that is called the adjacency list data structure, okay. These two notations are good enough for us to describe our iterative scheme.

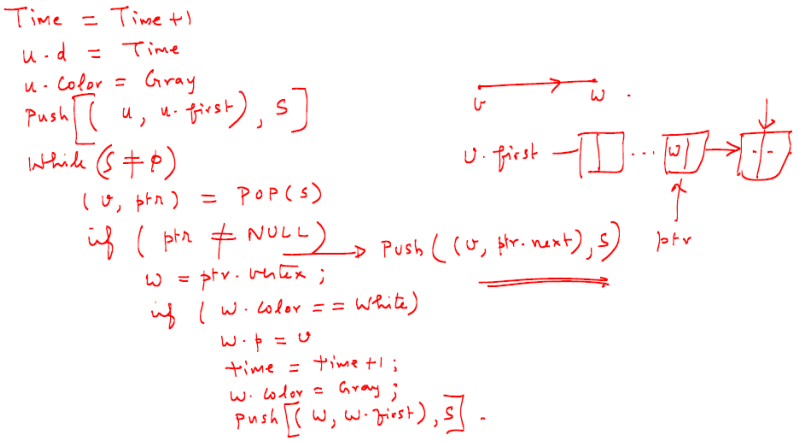
Note that in iterative scheme we explicitly maintain a stack and in the stack the things to be done are stored, like the recursion, when the recursion is executed the system will maintain a system stack and there the system stores all the references related to future computations to be done. The references for the future computation to be done are going to be maintained by us now explicitly using a stack defined by us okay. So here is a code, non recursive DFS, this starting from a vertex u, works as follows S is empty, what is S? S is a stack and this stack contains this stack containing pairs of the form v ptr that means in the adjacency list of v a particular position indicated by ptr will have a reference. and that is what is stored in the stack.

So stack will have items of this kind, okay that correspond we put that, corresponds to the position of the neighbor of v to be explored, which neighbor you are going to take up next, that will be available in the stack, which vertex and for that vertex which neighbor would you take it up for the next step of the exploration, that is available in the stack. So a stack will have pairs of this type and we take the top element and whatever is the top element we process, we continue our exploration with the item indicated by the top element of the stack. Initially the stack is empty, time equals time plus 1, this is the time stamping book keeping, u.d discovery time is the moment you enter the first statement that you execute is update the time and then store the time stamp. So this is called the discovery time and u.color is gray, this indicates the fact that u has become active now initially u will be white, the moment a call is made u becomes active from this point onwards u is active and its neighbors will be examined one after another. So u dot color is grey and push u, u dot first, push this item in the stack S, okay first you put this in the

stack. So notice that stack has items which are pairs, first component is a vertex second component is a pointer u.first is a pointer, it is a pointing to the first box in the adjacency list of u, you are going to push that item there, now while S is non-empty, as long as it is non-empty you are going to do the following, you are going to take up the task as indicated in the stack okay. Let v, ptr equal to POP S. So what is push and pop they are the stack operation when you are trying to include in the stack you get it done by the operation called the push operation and you throw an element when you delete an element from the stack you get it done using the operation called the POP. So stack has got a lot of pairs POP means the top item from the stack is deleted and let v ptr is the item that has been popped up okay.

If ptr is not equal to null. okay this is a non-null one. Now consider w equal to ptr dot vertex, ptr is pointing to a particular box and that box has got a vertex and that vertex is w. So this is v, this is a vertex v and you are considering several neighbors and now you are considering the neighbor w, this is indicated by v dot first, that is showing the adjacency list, at some part in the adjacency list you have ptr and that is pointing to w, ptr is pointing to w.

So w so this is the edge I am considering. I am considering the edge vw from v I am considering the edge w vw so I am going to w. If w dot color equal to white that means it is an unexplored vertex. I am going to, if w dot color equal to white then I will begin my exploration, w dot p equal to v, time equal to time plus 1, w dot color equal to gray and push w comma w dot first in the stack S, because you have to continue your exploration with this therefore this you are going to push.

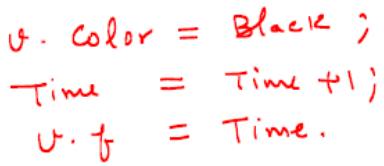


When the pointer is not equal to nil, you have already accessed it, so after finishing this where would you continue ,suppose you finish with this vertex, you are going to continue with the next vertex. Let us say, this is currently you are exploring this and this is the process of exploring and after this exploration you have to continue and where would you continue, you have to continue with this one, so that is what you have to push that into

the stack, so if w.ptr, if if ptr is not equal to null okay. So first thing that we do is we can insert the following statement here if ptr not equal to null push v comma ptr.next in the stack S. Notice that ptr.next you can consider only when ptr is not equal to null. When ptr is not null, you are pushing this and this tells you after finishing with the v.ptr you are going to take up this one. So you go to w, else, this else means vw is a non-tree edge, vw is a non-tree edge because color of W is not white. Since the color of w is not white it has come to the else clause if it were white if w dot color is white you would have explored it if it is not white you are going to do whatever you are planning to do for non-tree edge. So do computations else, in this else part do computations related to non-tree edge vw, okay this is what you will do in the else part.

There is another if statement we have considered if ptr is not equal to null. If ptr equal to null so that I write else in this case ptr equal to null, if ptr is null what does it mean? V dot ptr and ptr is null that means the last edge has been considered and then there is no further edge is available exploration related to v is complete. Exploration related to v is complete, everything related to that is complete.

Therefore what we have to do is that v dot color equal to black and time equal to time plus 1 and v dot f equal to time because for that vertex the processing is done.



v dot ptr equal to null means all neighbors are processed, right now it has got v and null, ptr is null means we have if ptr is not null, it will be like this, it will be pointing to a physical box okay and that vertex is to be processed, but if ptr is null that means v is completely processed. You have started from v all neighbors of v has been examined now v comma ptr which is equal to v comma null so let me write that v comma ptr which is equal to v, null indicates that all neighbors of v visited hence visit at v maybe finished. You may finish the visit at v and you can go back, right. So the fact that v is finished is done by these 3 steps set the color of v is black until then the color of v will be grey okay. So when all neighbors are finished you set that as black okay, so what you do is keep pushing the exploration still to be done, the positions at which explorations to be done into the stack. So what we are doing is that, keep pushing the positions for which explorations are still to be done into the stack. Keep pushing into the stack all these postponed task, I have to do this, I have to explore this and I have to explore all of them, I have to explore their children, whatever the position, whatever the things that you have to do, all the postponed one things to be done in the stack.

Then POP from stack and keep doing one after another alright, therefore stack now defines the order in which we are carrying out the task. So when we are currently done with the position, the next position at which it has to continue is going to be put in the stack and that is what is done by this step. So if you look into this step it is doing the pushing, it is doing the, it is saving the position at which I am going to continue after I am done with the current position right. So after finishing the visit corresponding to the position v ptr we must continue with v comma ptr dot next that is why we puch v, ptr.next into the stack and this completes our discussions on iterative DFS. The complexity is same , we have not done anything we have just maintained a stack explicitly. And in that stack all the positions at which we have to still to explore are stored, so all the postponed jobs are stored in stack. Stack defines an order in which the postponed tasks are to be done, one after another you have to do, one after another vertex and edge you have to visit, but in which order, stack determines that order. Earlier a recursion was determining the order and now the stack explicitly maintained by us determines the order because anything that is postponed is all put in the stack, then from the stack we retrieve one position after another and then carry out the visit at the retrieved position. Therefore this works in a way quite similar to the recursive one except that we are maintaining and managing the positions at which we have to continue using a stack.

The complexity is same, alright it is linear, the complexity is same as the recursive one and the complexity is order n plus m,

$$O(n+m)$$

n is the number of vertices, m is number of edges. okay this completes our discussions related to non-recursive version of DFS thank you.