Course: Introduction to Graph Algorithms

Professor: C Pandu Rangan

Department: Computer Science and Engineering

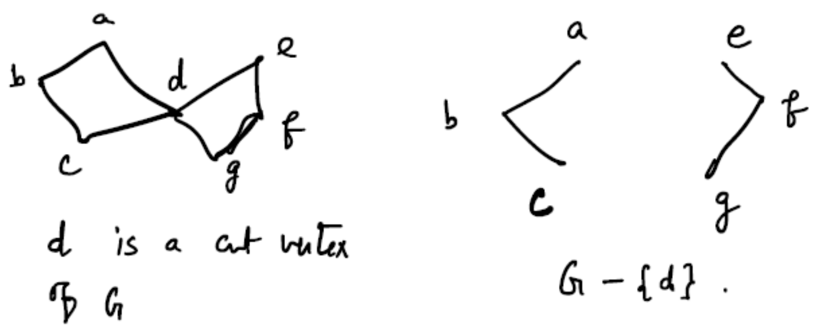
Institute: IISc

Week: 07

Lecture 30 Algorithm for Cut Vertex

Namaskara we have seen death first search and a skeletal form of the death first search. Death first search in its vanilla form will look like a simple recursive program but it has got several control points at which we can meaningfully insert additional computational steps. What kind of computational steps to be inserted where is what we have seen in depth first search skeleton. Here is a canonical example where that skeleton is going to be filled with meaningful computations. We are going to solve the problem of finding the cut vertices of a graph. We are working on undirected graph for this case.

So let G be a connected graph a vertex v is a cut vertex if it is removal disconnects the graph for example in the graph shown in the picture vertex d is a cut vertex you remove it you can see that several pieces are created abc, efg these are all different pieces it is a disconnect. so d is a cut vertex of a graph. or f is not a cut vertex you remove f still you have a connected piece remaining there okay.

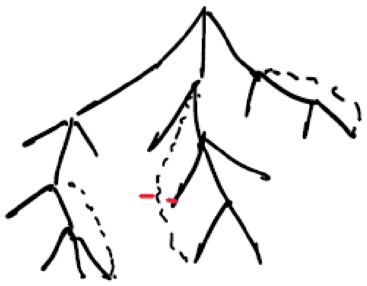


So we have seen that problem solving begins with a mathematical thinking and then we see how to proceed with computational thinking finally, we translate that into algorithmic thinking. Our problem is to solve the cut vertex problem, that is you are given a graph you have to say which vertex is a cut vertex and which vertex is not a cut vertex. So every cut vertex is going to be labeled ,this is a cut vertex, this is not a cut vertex and so on. So you can have a attribute cv so that v dot cv equal to 0, if v is not a cut vertex v dot cv equal to 1, if v is a cut vertex right. If it is not a cut vertex this is when v is indeed a cut vertex very simple a Boolean, we have to determine this.

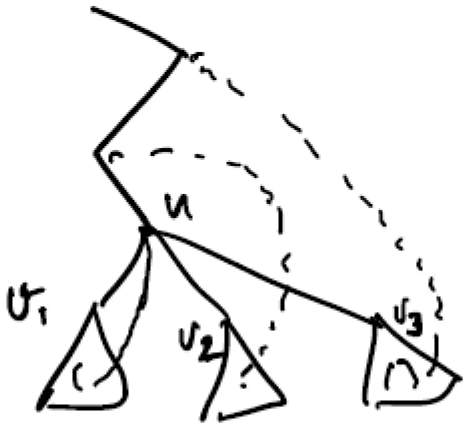
So mathematicians have a way to say which vertex is a cut vertex, which vertex is not a cut vertex if and only if characterizing condition. A vertex v is a cut vertex if and only if there is a pair of vertices x, y such that every path between x and y contains v. Look at the way in which the cut vertex is characterized it is a beautiful theorem and nice proof of this is also available but is it possible for us to arrive at a computational process based on this characterization, it is an if and only if. So think for a moment, you have a vertex v you must find out a pair of vertices such that all paths between that pair of vertices must contain v. So you take a pair of vertices, enumerate all paths and check V is in all paths.

If v is not in all paths you have to work for another pair, maybe this is not a pair it is another pair, again for that pair you have to enumerate all the paths and in each path whether v is there you have to find and number of paths can be exponential and there is no convenient way to enumerate all paths very complicated task. Therefore while this is an interesting mathematical characterization it does not lead to a convenient computational process. So we are going to think in a different way and then arrive at a new characterization but with this new characterization it is easy to device a computational method.

So let us view G in terms of tree edges and non-tree edges, so you perform a DFS and with respect to a DFS you will have tree edges and non-tree edges. All non-tree edges are called the back edges because they take you from a descendant to an ancestor, so it will be like this.



There would not be a non-tree edge like this there would not be a non-tree edge like I have marked in red the reason is simple if that edge were there while exploring this node you would have considered that non-tree edge and then you would have included it. Therefore there would not be any non-tree edge going to some other part of the tree. All the non-tree edges will go only to an ancestor, that is why they are called the back edges. We believe we imagine the tree is growing downwards, the non-tree edges will take you to an ancestor. So it will be like this.



So consider a vertex u and during the exploration it will explore a child and after finishing it will come back to u and after finishing with v2 again you will come back to u, the exploration goes recursively. Suppose you have a child. that does not have child or it is a descendant does not have a back edge taking you beyond you like in this picture v1. But for v2 you can see there is a back edge from a descendant of v2 going all the way up it will be going over and above you similarly v3. If this is the case removal of u is going to disconnect v1 because v1 and its descendants are maximum connected with u.

If u is removed then these nodes have no other connection to anywhere else the graph gets disconnected that is the reason why you have to see whether a node has a child which may become an orphan if that node is deleted if u is deleted v1 will become an orphan, v1 gets completely disconnected right. So motivated by this we have a definition low u is minimum over v dot d where v is a vertex reachable from you by using 0 or more tree edges and 0 or 1 back edge okay in the most general form from u, you go through some tree edge and one back edge, one single back edge. So what is the earliest visited vertex you can reach by this process from the node walk along certain tree edges and then use one back edge and then find out where you can go through that back edge. Whenever there is a back edge found the other end of the back edge and its discovery time is going to be recorded and it is a minimum of discovery time. So low u is going to be computed in the following way, so first observe that minimum of v.d where v is reachable from a descendant of u by a back edge and this quantity is nothing but minimum of low of v because v dot d minimum is nothing but low. So also note that all values needed to compute low u are not available in one place so that you can compute easily the minimum you have to compute low value by looking into the relevant values as and when they generate.

So during the exploration at different points the relevant values will be discovered and you use them to update the minimum. So low u is nothing but minimum of low u and v dot d where uv is a back edge, it is going from u because from u with no tree edge and one back edge that means let u be the vertex and with no tree edge. And one back edge means from u you may have several back edges each going to some vertex and we are interested in the one that has got the minimum discovery time.

Therefore low u in this case captures 0 tree edge followed by one back edge no tree edges involved here, low v for a child can be recursively computed. And low U is updated as minimum of low u and low v because if you go one step down from u get a child v from here whatever is the low that you obtain is done by one tree edge followed by 0 or more tree edges followed by one back edge. Therefore low v gives the when you recursively compute it gives the vertex that can be reached from v or it is a descendants okay. Therefore in order to compute the low, all we have to do is the following.

We add u dot low equal to u dot d at control point 1, control point 1 is when it is discovered. So u dot low is u dot d, this step is to be added at control point 1. Now when you backtrack from a vertex v you have to update. low of u, so look at the control point 4. So in control point 4 we have to include the computation to be done while backing from v to u, while backing from v to u you are updating the low of u by finding the minimum that is what is done here.

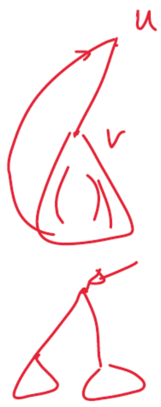
Therefore this must be added at control point 4 and when there is a back edge we have seen u.low is minimum of u.low cum v.d and this must be inserted at control point 5 because control point 5 is where the computation related to back edge are to be done. So this is the step that you have to add at control point 5. So when you fit in these steps into the skeleton it becomes an algorithm finding the low of a vertex. Low of all vertices can be computed, you just do the DFS during DFS various points you do various computation that would solve the problem of finding the low for every vertex okay.

But finding the low of every vertex, you can see here is the DFS low of G for each vertex u dot color is white, u dot p is nil, u dot d is 0, u dot low is 0. Now time is 0, DFS low Gu, DFS low Gu is found like this time equal to time plus 1, u dot d equal to time this is the depth of search. What is that I have added? u dot low equal to u dot d this is at the time of entry. Therefore at control point 1 you can see a step is added. What is added at control point 4? This is control point 4, this is control point 1, this is control point 4, this is control point 5, at control point 5 whatever you have to do for the back edge is added at control point 4 whatever you are supposed to do for backtracking is done.

Therefore this completes the specification of the algorithm, see how elegantly during DFS, in the skeleton of DFS by adding certain steps you are getting a solution for a problem. Cannot yet solve the cut vertex problem, we have solved a problem of generating a very useful information and that is called the low of a vertex. The low of a vertex is useful in coming up with a new characterization of a cut vertex. You can see that a vertex is a cut vertex, if and only if, A vertex is a cut vertex if and only if there is a child v of u in the DFS tree such that low of v is greater than or equal to u of d okay. If this is the case low of v is greater than or equal to u of d.

If this is the case u is a cut vertex why u is a cut vertex u is like this v is like this low of v is from v you can go down and then go to this and back edges from v also back edges wherever maximum it goes is only up to this. is not going above u because low of v is greater than or equal to u dot d all back edges, tree edges are clustered here. So when you remove u, v will be orphaned. In other words u is a cut vertex so u is a cut vertex if and only if there is a child v of u such that low of v is greater than or equal to u dot d. But this characterization is true for root right because discovery time of the root is 1 any other vertex will therefore this characterization should not be applied for the root.

The root will be a cut vertex, if it has got degree more than 1 because this is the root of the DFS tree, if the root has got degree more than 1 naturally its removal will disconnect the graph therefore root is a cut vertex if and only if it has more than one child



Therefore your algorithm becomes much simpler how do you find a cut vertex u dot cv equal to 1 if u is a cut vertex otherwise it is 0, all we have to do is that if u is not the root and v dot low u is greater than or equal to ud u dot cv equal to 1 only this has to be added at control point 4, that is it. When this is done you have solved the problem of finding the cut vertices the whole solution emerges as a magic okay, initially things were complicated but with a new characterization we are able to finish the task of finding the cut vertices in a very easy manner. Okay, this is the power of algorithmic thinking. You start with mathematical characterization but if that mathematical characterization is not satisfactory, find new characterization, depth first search allows you to find some values which are helpful in arriving at a new characterization. Use that characterization to determine whether a vertex is cut vertex or not and this can be done on the go.

Therefore we have a linear time algorithm for finding all cut vertices of the graph, extremely elegant and simple one. This shows the power of DFS how at various control points of DFS by adding additional meaningful computational steps we can design an algorithm solving a graph problem okay. I conclude at this point. Thank you.