Assignment II

Course Code: CSE214

Course Name: Algorithm

Submitted To

Subroto Nag Pinku
Department of CSE
Daffodil International University

Submitted By

Name: S.M. Khasrul Alam Shakil
Id: 191-15-12180
Sec: 0-14

1)Full Tree Traversal

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree −

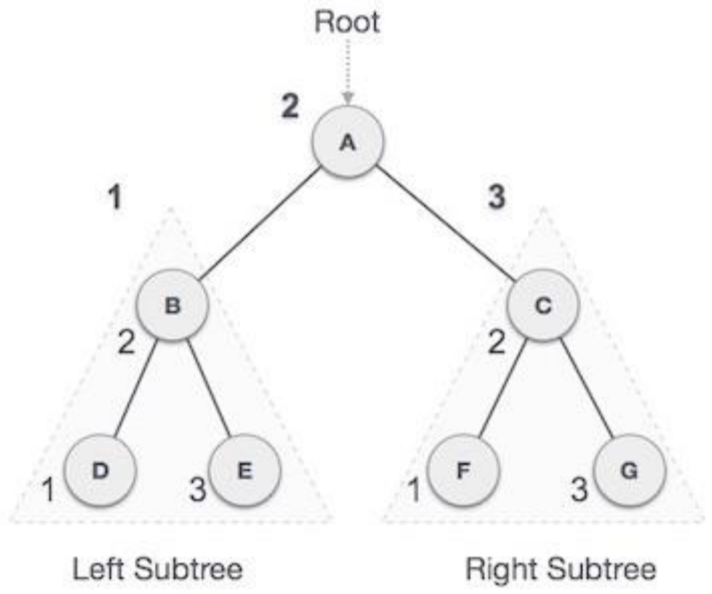
- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

In-order Traversal

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.

If a binary tree is traversed in-order, the output will produce sorted key values in an ascending order.



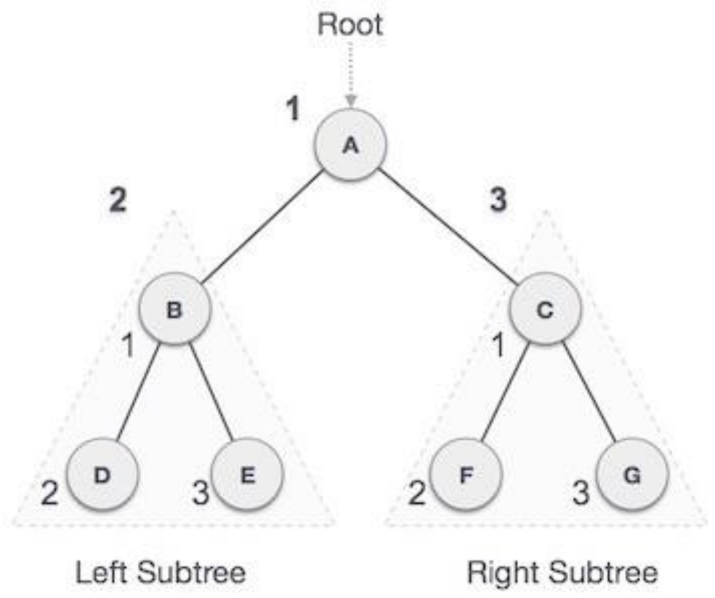
We start from A, and following in-order traversal, we move to its left subtree B. B is also traversed in-order. The process goes on until all the nodes are visited. The output of inorder traversal of this tree will be −

D → B → E → A → F → C → G

Algorithm
Until all nodes are traversed −
Step 1 − Recursively traverse left subtree.
Step 2 − Visit root node.
Step 3 − Recursively traverse right subtree.

Pre-order Traversal

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.



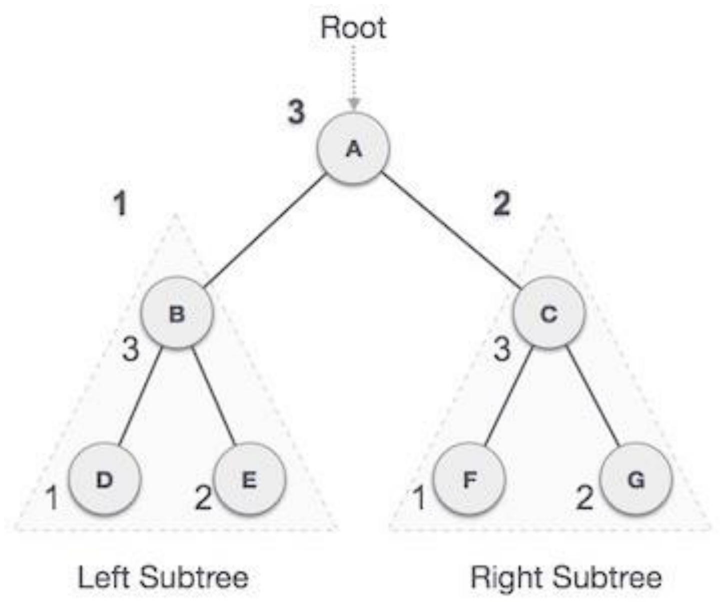
We start from A, and following pre-order traversal, we first visit A itself and then move to its left subtree B. B is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be −

A → B → D → E → C → F → G

Algorithm
Until all nodes are traversed −
Step 1 − Visit root node.
Step 2 − Recursively traverse left subtree.
Step 3 − Recursively traverse right subtree.

Post-order Traversal

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.



We start from A, and following Post-order traversal, we first visit the left subtree B. B is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be −

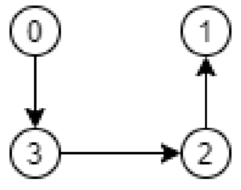
D → E → B → F → G → C → A

Algorithm
Until all nodes are traversed −
Step 1 − Recursively traverse left subtree.
Step 2 − Recursively traverse right subtree.
Step 3 − Visit root node.

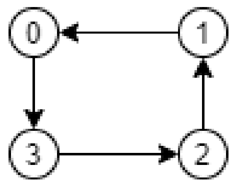
2) Cycle Finding

A graph cycle is when there is a "loop" or circular reference. This can be a series of edges that connect back to an origin vertex. If a graph has a cycle it is a cyclic graph.

To determine if a graph has a cycle, we can traverse the graph and look for a back edge. A back edge is one that connects a vertex to an already visited ancestor.



Acyclic graph (no back edeges)



Cyclic graph (introducing back edeges)

To detect a cycle in a directed graph (i.e to find a back edge), we can use depth-first search (with some introduction of local state to tell we if a back edge occurs):

[We will maintain 3 buckets of vertices: white, grey, & black buckets. (We can also colour vertices instead)]

- The white bucket will contain all of the *unvisited* vertices. At the start of our traversal, this means every vertex is in the white bucket.
- Before visiting a vertex, we will move it from the white bucket into the grey bucket.
- After fully visiting a vertex, it will get moved from the grey bucket into the black bucket.
- We can skip over vertices already in the black bucket, if we happen to try and visit them again.
- When visiting the children/descendants of a vertex, if we come to a descendant vertex that is *already* in the grey bucket - that means we have found a back edge/cycle.
- This means the current vertex has a back edge to it's ancestor - as we only arrived at the current vertex via it's ancestor. So we have just determined that there is more than one path between the two (a cycle).

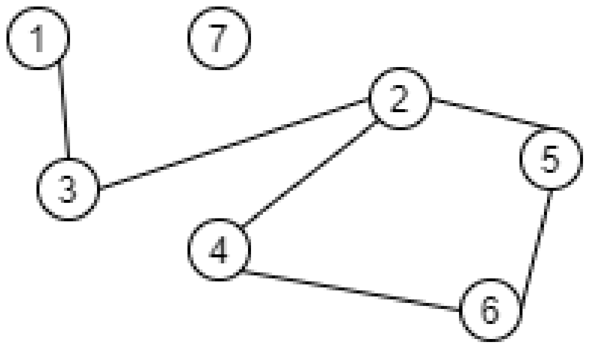
To detect a cycle in an undirected graph, it is very similar to the approach for a directed graph. However, there are some key differences:

- We no longer colour vertices/maintain buckets.
- We have to make sure to account for the fact that edges are bidirectional - so an edge to an ancestor is allowed, if that ancestor is the parent vertex.
- We only keep track of visited vertices (similar to the grey bucket).
- When exploring/visiting all descendants of a vertex, if we come across a vertex that has already been visited then we have detected a cycle.

Time complexity is O(v + e) for an adjacency list. Space complexity is O(v). For an adjacency matrix, the time & space complexity would be O(v^2).

3) Component Finding

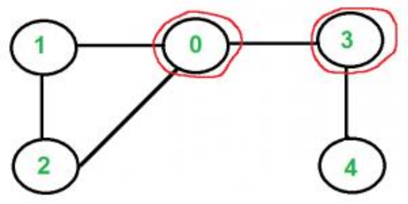
We have to find that, which node are connected together. Two node is called connected when it is possible to go one to another node. Component finding can be done by using both DFS and BFS. We will run a loop outside BFS and DFS and will see for every node it is visited or not, otherwise will do BFS or DFS on that node. So how many time we do BFS or DFS is the result. In the bellow image if we start BFS or DFS from 1 then 1-6 all will be visited, so we don’t need to do BFS or DFS for 2,3,…6. But 7 is not visited yet. So again have to do BFS or DFS for 7 , after doing that all node will be visited. Therefore we needed to call BFS or DFS for 2 times in outside loop. So, there is 2 component.



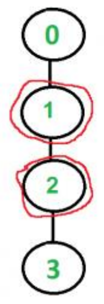
4) Articulation Points in a Graph

A vertex in an undirected connected graph is an articulation point (or cut vertex) iff removing it (and edges through it) disconnects the graph. Articulation points represent vulnerabilities in a connected network – single points whose failure would split the network into 2 or more components. They are useful for designing reliable networks.
For a disconnected undirected graph, an articulation point is a vertex removing which increases number of connected components.

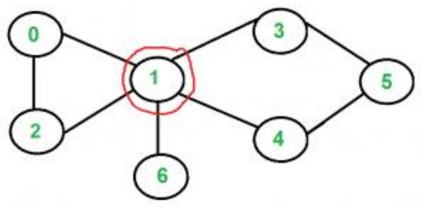
Following are some example graphs with articulation points encircled with red color.



Articulation points are 0 and 3



Artiulation Points are 1 & 2



Articulation Point is 1

Find all articulation points in a given graph:

A simple approach is to one by one remove all vertices and see if removal of a vertex causes disconnected graph. Following are steps of simple approach for connected graph.

1) For every vertex v, do following

   a) Remove v from graph
   b) See if the graph remains connected (We can either use BFS or DFS)

   c) Add v back to the graph

Time complexity of above method is O(V*(V+E)) for a graph represented using adjacency list.

A O(V+E) algorithm to find all Articulation Points (APs):
The idea is to use DFS (Depth First Search). In DFS, we follow vertices in tree form called DFS tree. In DFS tree, a vertex u is parent of another vertex v, if v is discovered by u (obviously v is an adjacent of u in graph). In DFS tree, a vertex u is articulation point if one of the following two conditions is true.
1) u is root of DFS tree and it has at least two children.
2) u is not root of DFS tree and it has a child v such that no vertex in subtree rooted with v has a back edge to one of the ancestors (in DFS tree) of u.

We do DFS traversal of given graph with additional code to find out Articulation Points (APs). In DFS traversal, we maintain a parent[] array where parent[u] stores parent of vertex u. Among the above mentioned two cases, the first case is simple to detect. For every vertex, count children. If currently visited vertex u is root (parent[u] is NIL) and has more than two children, print it.
How to handle second case? The second case is trickier. We maintain an array disc[] to store discovery time of vertices. For every node u, we need to find out the earliest visited vertex (the vertex with minimum discovery time) that can be reached from subtree rooted with u. So we maintain an additional array low[] which is defined as follows.

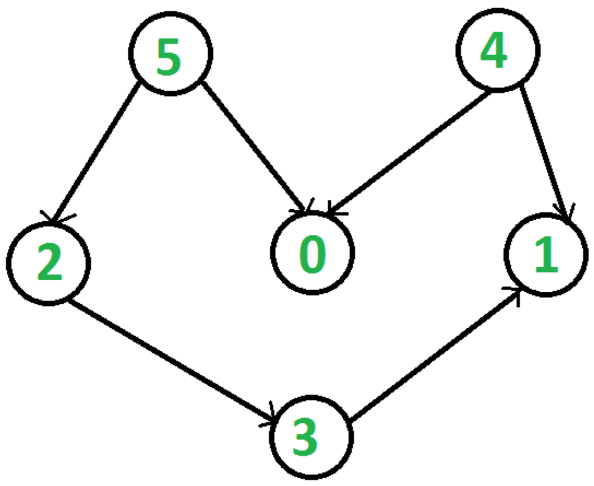
low[u] = min(disc[u], disc[w])
where w is an ancestor of u and there is a back edge from
some descendant of u to w.

5) Topological Sort

The nodes must be arranged in the directed graph in such a way that if there is any directed arm from u to v, then the sorted array is preceded by u, v. This is possible only when there is no directed cycle in the corner of that graph. If there is a cycle, we can't arrange the nodes of that cycle in any way, right? We can also use this algorithm to find out if there is a cycle in a corner graph.

We can do topological sort in two ways. One with BFS and the other with DFS. Seems relatively easy with BFS, also doesn't bother with stack overflow here. But DFS tends to be a little comparatively short. First let's see how we can solve with BFS. First we need an array of in-degree where the in-degree of each node is written. Now a queue should be placed for those nodes whose index is zero. Now take the node one by one from the queue. We will take a node one by one and look at the arms that come out of it and reduce the node's index on the other end. If the index on the other end is zero, we will insert it into Queue. This will continue until the Queue is empty. The order in which the node is inserted in the queue is the order of the topological sort. But one thing, if we see that even after the Queue is empty, the node's degree is now zero, which means there is a cycle in the graph. This is a very intuitive algorithm. It is easy to understand why it is working. For example, at the beginning, we inserted node nodes with queue in Queue because these nodes do not depend on anyone. Now when we actually take these nodes, it means that the dependence of those who depend on it will decrease one by one. If their dependency becomes zero, it means we can take this node now if we want, so we put it in Queue

Let's see how to solve this with DFS. Suppose T is an array of our results, and visited another array whose initial value is 0. Now we will look at each node one by one and if the value of the visited of that node is not now 2 then we will call its DFS. The first thing we will do with DFS is to set the value of visited to 1. After that we will see the directed arm that has come out from here. If the value of the node visited on the other end is 1, it means that we have got a cycle, so there is no topological order in the corner of the node. If the value of visited is 2 we have nothing to do. And if 0, then we call DFS for that node thus, when the processing is completed for each node of Adjacent, we will make Visited 0 of that node and insert it in T and return it from DFS. Thus, when the DFS of all nodes is finished, we get the topological sort in T in reverse order. Why? Note that we call DFS for all its descendants before inserting the node in the corner. That is, I put them in T first and then myself. So this method will definitely work if there is no cycle in the corner in our graph. Now, how do we know if we have a cycle? It's also easy to do DFS if we come to the previous visited node then there is a cycle. But that is not entirely true. Let's think about when the mind is not. Suppose we have an arm in the graph from A to B. Now if we do DFS of B first, then for A then but we are coming to a visited node with A - B arm. But there is no cycle here. Let's look at another example, a graph of three arms A– B. A - C, C – B. Suppose first A's DFS is called here, then from here we go to B, return it to visit and when we come to C and go to sit B again but we will get a visited node, but there is no cycle here. Then. What is the way? By the way, currently DFS is there, there is a cycle if we can go to a corner where we have not come back yet. As in the previous two examples we have already done DFS. Back to B this return is being written in the visited array with 2. And 1 means DFS is still running. For example, if the arms of our graph are folded, A- B, B - C, C - A. Now we start DFS from A, then we start B, then C, then A again. Come on. Here we see that A's DFS is not over yet but we have come to A! We will understand that there is a cycle here.

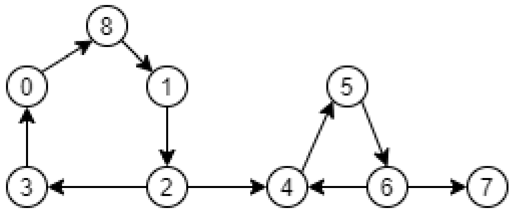


6) Strongly Connected Components

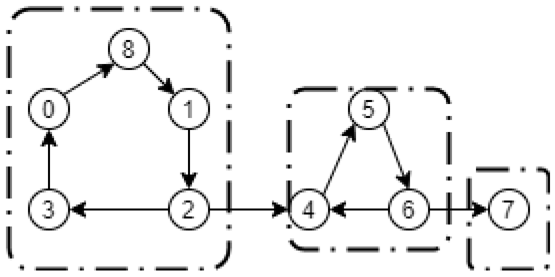
In a directed graph when we can go from node u to v and also v to u then both are called strongly connected node (SSC). Also A strongly connected component is the portion of a directed graph in which there is a path from each vertex to another vertex. It is applicable only on a directed graph.

For example:

Let us take the graph below.



The strongly connected components of the above graph are:



We can observe that in the first strongly connected component, every vertex can reach the other vertex through the directed path.

These components can be found using Kosaraju's Algorithm.

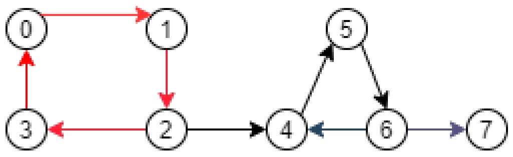
Kosaraju's Algorithm:
1)Kosaraju's Algorithm is based on the depth-first search algorithm implemented twice.

Three steps are involved.

Perform a depth first search on the whole graph.

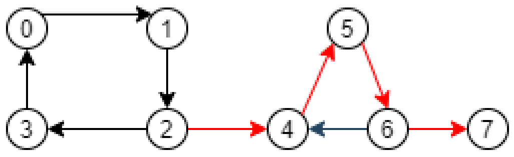
Let us start from vertex-0, visit all of its child vertices, and mark the visited vertices as done. If a vertex leads to an already visited vertex, then push this vertex to the stack.

For example: Starting from vertex-0, go to vertex-1, vertex-2, and then to vertex-3. Vertex-3 leads to already visited vertex-0, so push the source vertex (ie. vertex-3) into the stack.



Visited	0	1	2	3				
Stack	3							

Go to the previous vertex (vertex-2) and visit its child vertices i.e. vertex-4, vertex-5, vertex-6 and vertex-7 sequentially. Since there is nowhere to go from vertex-7, push it into the stack.



Visited	0	1	2	3	4	5	6	7
Stack	3	7						

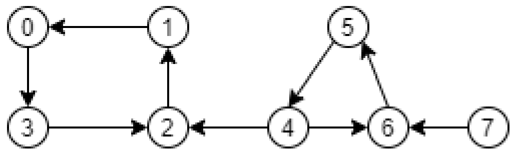
Go to the previous vertex (vertex-6) and visit its child vertices. But, all of its child vertices are visited, so push it into the stack.

Visited	0	1	2	3	4	5	6	7
Stack	3	7	6					

Similarly, a final stack is created.

Visited	0	1	2	3	4	5	6	7
Stack	3	7	6	5	4	2	1	0

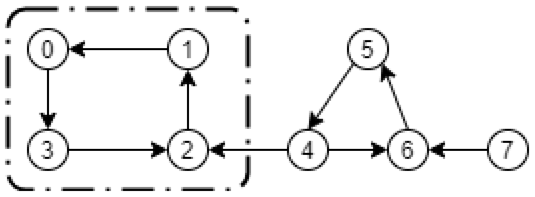
2) Reverse the original graph.



3) Perform depth-first search on the reversed graph.

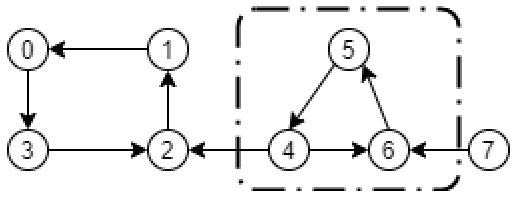
Start from the top vertex of the stack. Traverse through all of its child vertices. Once the already visited vertex is reached, one strongly connected component is formed.

For example: Pop vertex-0 from the stack. Starting from vertex-0, traverse through its child vertices (vertex-0, vertex-1, vertex-2, vertex-3 in sequence) and mark them as visited. The child of vertex-3 is already visited, so these visited vertices form one strongly connected component.

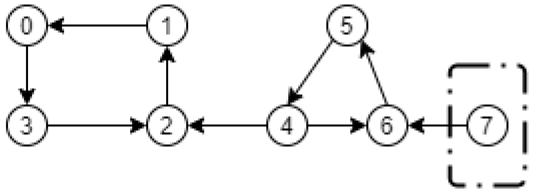


Visited	0	1	2	3				
Stack	3	7	6	5	4	2	1	
SCC	0	1	2	3				

Go to the stack and pop the top vertex if already visited. Otherwise, choose the top vertex from the stack and traverse through its child vertices as presented above.



Visited	0	1	2	3	4	5	6	
Stack	3	7	6	5				
SCC	4	5	6					



visited	0	1	2	3	4	5	6	
Stack								
SCC	7							

4) Thus, the strongly connected components are: