Daffodil International University

Algorithms, CSE 214

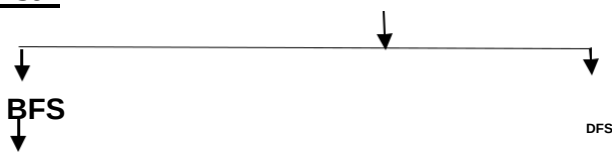
Submitted to
Subroto Nag Pinku,
Lecturer, FSIT.
Daffodil Int. University.

Submitted by
Ahasan Kabir
191-15-12930
CSE, 0-14, DIU

e 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(link). We always start from the root(head) node.

Tree Traversal



Level order Traversal

Inorder Traversal

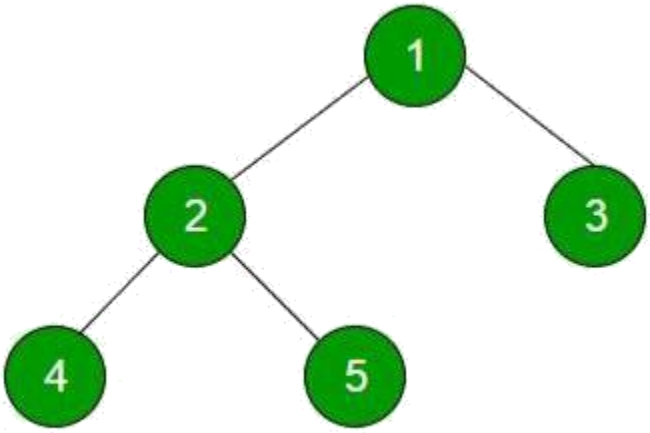
Preorder Traversal

Postorder Traversal

#Component finding:

Breath First Search:

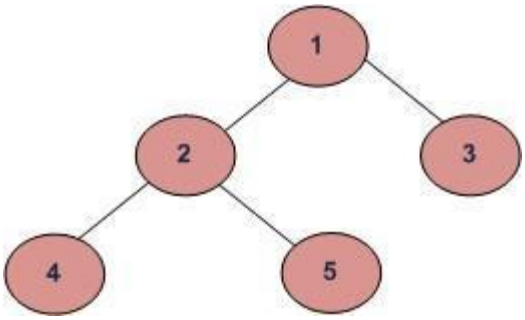
BFS stands for Breath first search is a vertex based technique for finding a shortest path in graph. It uses a Queue data structure which follows first in first out. In BFS, one vertex is selected at a time when it is visited and marked then its adjacent are visited and stored in the queue. It is slower than DFS.



OUTPUT is: 1 2 3 4 5

Depth First Search:

DFS stands for Depth first search is a edge based technique. It uses the Stack data structure, performs two stages, first visited vertices are pushed into stack and second if there is no vertices then visited vertices are popped.



Depth First Traversals:
(a) Inorder (Left, Root, Right): 4 2 5 1 3
(b) Preorder (Root, Left, Right): 1 2 4 5 3
(c) Postorder (Left, Right, Root): 4 5 2 3 1

Algorithm Inorder(tree):

1. Traverse the left subtree.
2. Visit the root.
3. Traverse the right subtree.

Algorithm Preorder(tree):

1. Visit the root.
2. Traverse the left subtree.
3. Traverse the right subtree.

Algorithm Postorder(tree):

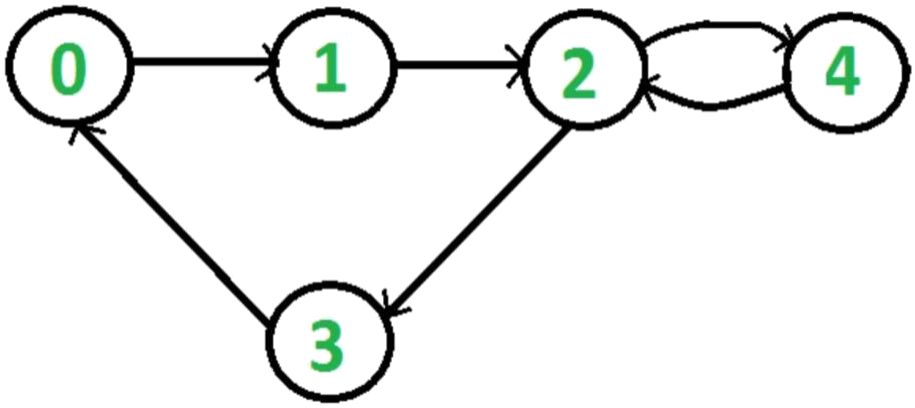
1. Traverse the left subtree.
2. Traverse the right subtree.
3. Visit the root.

#Cycle Finding:

Like directed graphs, we can use DFS to detect cycle in an undirected graph in
O(V+E) time. ... We do BFS traversal of the given graph. For every visited vertex 'v', if there is an adjacent 'u' such that u is already visited and u is not parent of v, then there is a cycle in graph.

Detect Cycle in a Directed Graph using BFS:

1. Increment count of visited nodes by 1.
2. Decrease in-degree by 1 for all its neighboring nodes.
3. If in-degree of a neighboring nodes is reduced to zero, then add it to the queue.



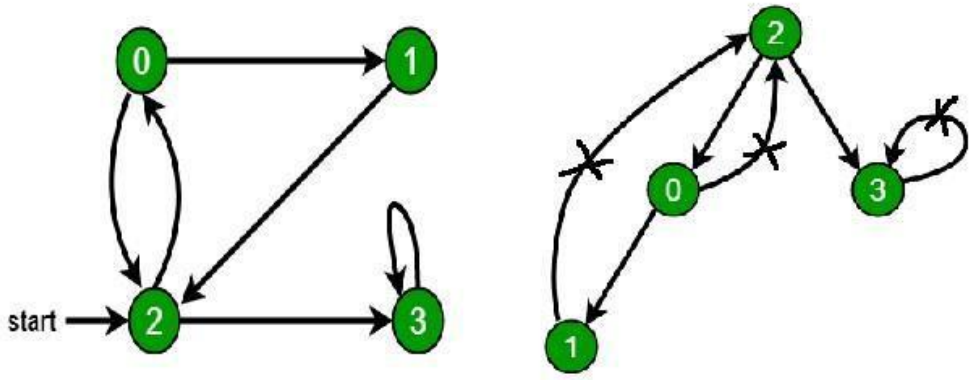
We do a BFS traversal of the given graph. For every visited vertex ‘v’, if there is an adjacent ‘u’ such that u is already visited and u is not parent of v, then there is a cycle in graph. If we don’t find such an adjacent for any vertex, we say that there is no cycle.

Time complexity:

The outer for loop will be executed V number of times and the inner for loop will be executed E number of times, Thus overall time complexity is O(V+E).

Detect Cycle in a Directed Graph using DFS:

Depth First Traversal can be used to detect a cycle in a Graph. DFS for a connected graph produces a tree. There is a cycle in a graph only if there is a back edge present in the graph. A back edge is an edge that is from a node to itself (self-loop) or one of its ancestors in the tree produced by DFS. In the following graph, there are 3 back edges, marked with a cross sign. We can observe that these 3 back edges indicate 3 cycles present in the graph.

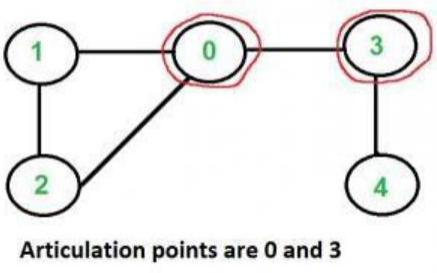


Time complexity:

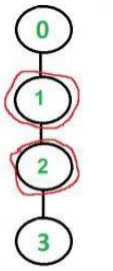
A graph without cycles has at most |V| - 1 edges (it's a forest). Therefore, if the DFS discovers |V| edges or more then it already found a cycle and terminates. The runtime is accordingly bounded by O(|V|).

#Articulation Point Finding:

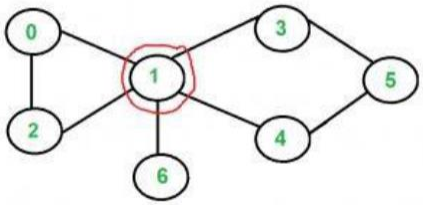
A vertex in an undirected connected graph is an articulation point (or cut vertex) iff removing it (and edges through it) disconnects the graph. Articulation points represent vulnerabilities in a connected network – single points whose failure would split the network into 2 or more components. They are 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

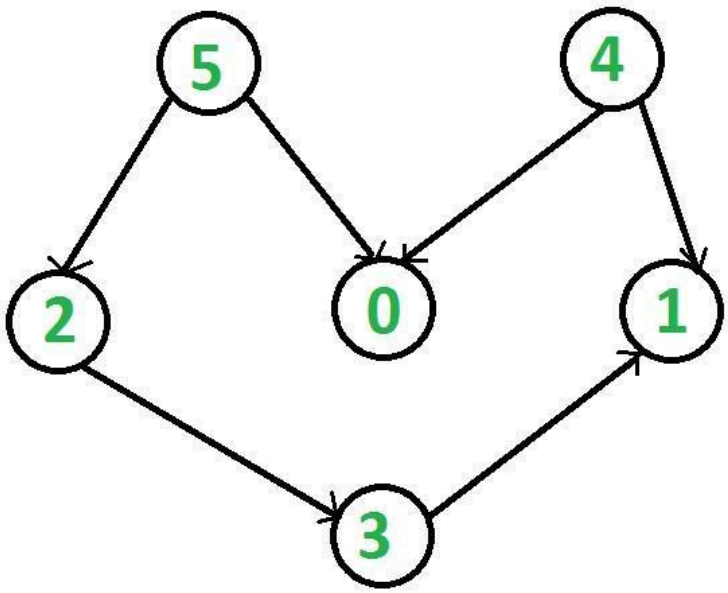
A simple approach is to one by one remove all vertices and see if removal of a vertex causes disconnected graph. Following are steps of simple approach for connected graph.

a)For every vertex V;

1. Remove v from graph.
2. See if the graph remains connected (We can either use BFS or DFS).
3. Add v back to the graph.

#Topological Sort:

Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge uv, vertex u comes before v in the ordering. Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge uv, vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG. For example, a topological sorting of the following graph is “5 4 2 3 1 0”. There can be more than one topological sorting for a graph. For example, another topological sorting of the following graph is “4 5 2 3 1 0”. The first vertex in topological sorting is always a vertex with in-degree as 0 (a vertex with no incoming edges).



Topological Sorting vs Depth First Traversal (DFS)*:*

In DFS, we print a vertex and then recursively call DFS for its adjacent vertices. In topological sorting, we need to print a vertex before its adjacent vertices. For example, in the given graph, the vertex ‘5’ should be printed before vertex ‘0’, but unlike DFS, the vertex ‘4’ should also be printed before vertex ‘0’. So Topological sorting is different from DFS. For example, a DFS of the shown graph is “5 2 3 1 0 4”, but it is not a topological sorting.

Algorithm to find Topological Sorting:

We recommend to first see implementation of DFS here. We can modify DFS to find Topological Sorting of a graph. IN DFS, we start from a vertex, we first print it and then recursively call DFS for its adjacent vertices. In topological sorting, we use a temporary stack. We don't print the vertex immediately, we first recursively call topological sorting for all its adjacent vertices, then push it to a stack. Finally, print contents of stack. Note that a vertex is pushed to stack only when all of its adjacent vertices (and their adjacent vertices and so on) are already in stack.

Time Complexity: O(V+E).

The above algorithm is simply DFS with an extra stack. So time complexity is the same as DFS which is.

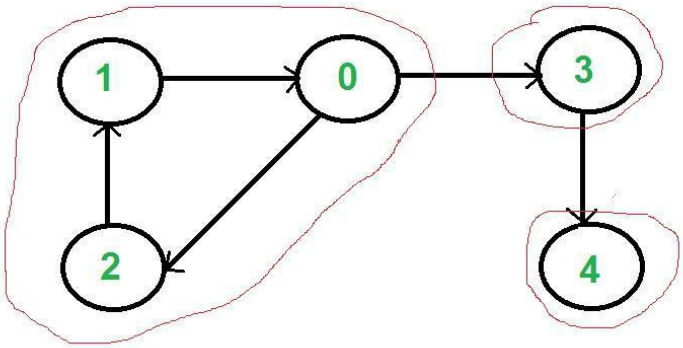
Note:

Here, we can also use vector instead of stack. If the vector is used then print the elements in reverse order to get the topological sorting.

#Strongly Connected Components:

In the mathematical theory of directed graphs, a graph is said to be strongly connected if every vertex is reachable from every other vertex. The strongly connected component of an arbitrary directed graph form a partition into subgraphs that are themselves strongly connected.

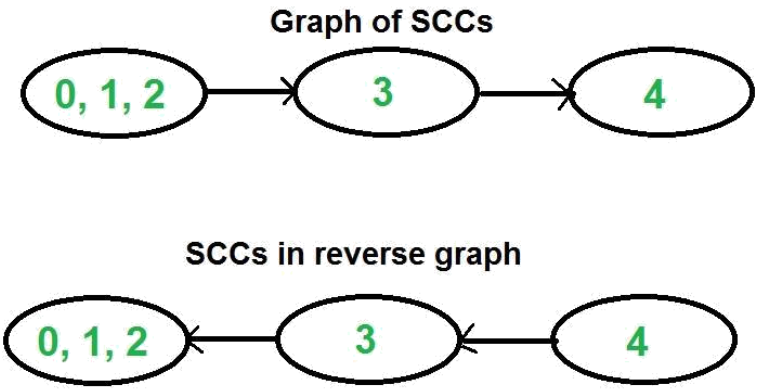
A directed graph is strongly connected if there is a path between all pairs of vertices. A strongly connected component (ssc) of a directed graph is a maximal strongly connected subgraph. For example, there are 3 SCCs in the following graph.



We can find all strongly connected components in O(V+E) time using Kosaraju's algorithm. Following is detailed Kosaraju's algorithm.

1. Create an empty stack 'S' and do DFS traversal of a graph. In DFS traversal, after calling recursive DFS for adjacent vertices of a vertex, push the vertex to stack. In the above graph, if we start DFS from vertex 0, we get vertices in stack as 1, 2, 4, 3, 0.
2. Reverse directions of all arcs to obtain the transpose graph.
3. One by one pop a vertex from S while S is not empty. Let the popped vertex be 'v'. Take v as source and do DFS. The DFS starting from v prints strongly connected component of v. In the above example, we process vertices in order 0, 3, 4, 2, 1 (One by one popped from stack).

Procedure:



The above algorithm is DFS based. It does DFS two times. DFS of a graph produces a single tree if all vertices are reachable from the DFS starting point. Otherwise DFS produces a forest. So DFS of a graph with only one SCC always produces a tree. The important point to note is DFS may produce a tree or a forest when there are more than one SCCs depending upon the chosen starting point. For example, in the above diagram, if we start DFS from vertices 0 or 1 or 2, we get a tree as output. And if we start from 3 or 4, we get a forest. To find and print all SCCs, we would want to start DFS from vertex 4 (which is a sink vertex), then move to 3 which is sink in the remaining set (set excluding 4) and finally any of the remaining vertices (0, 1, 2). So how do we find this sequence of picking vertices as starting points of DFS? Unfortunately, there is no direct way for getting this sequence. However, if we do a DFS of graph and store vertices according to their finish times, we make sure that the finish time of a vertex that connects to other SCCs (other that its own SCC), will always be greater than finish time of vertices in the other SCC (See this for proof). For example, in DFS of above example graph, finish time of 0 is always greater than 3 and 4 (irrespective of the sequence of vertices considered for DFS). And finish time of 3 is always greater than 4. DFS doesn't guarantee about other vertices, for example finish times of 1 and 2 may be smaller or greater than 3 and 4 depending upon the sequence of vertices considered for DFS. So to use this property, we do DFS traversal of complete graph and push every finished vertex to a stack. In stack, 3 always appears after 4, and 0 appear after both 3 and 4. In the next step, we reverse the graph. Consider the graph of SCCs. In the reversed graph, the edges that connect two components are reversed. So the SCC {0, 1, 2} becomes sink and the SCC {4} becomes source. As discussed above, in stack, we always have 0 before 3 and 4. So if we do a DFS

of the reversed graph using sequence of vertices in stack, we process vertices from sink to source (in reversed graph). That is what we wanted to achieve and that is all needed to print SCCs one by one.

Time complexity:

The above algorithm calls DFS, finds reverse of the graph and again calls DFS. DFS takes O(V+E) for a graph represented using adjacency list. Reversing a graph also takes O(V+E) time. For reversing the graph, we simple traverse all adjacency lists.