Indian Institute of Information Technology Design and Manufacturing,
Kancheepuram, Chennai 600 127, India
An Autonomous Institute under MHRD, Govt of India
http://www.iiitdm.ac.in
COM 209T Design and Analysis of Algorithms -Lecture Notes

Instructor
N.Sadagopan
Scribe:
Dhanalakshmi S

Graph Algorithms

Objective: In this module, we shall introduce graphs which are a powerful model for modelling combinatorial problems in computing. We also discuss graph traversals, namely, Breadth First Search (BFS) and Depth First Search (DFS) and their applications.

1 Preliminaries

Definition 1 (Graph) *A **graph** G is a pair $G = (V(G), E(G))$ consisting of a finite set $V(G) \neq \emptyset$ and a set $E(G)$ of two-element subsets of $V(G)$. The elements of $V(G)$ are called vertices of G. An element $e = \{a, b\}$ of $E(G)$ is called an edge of G with end vertices a and b.*

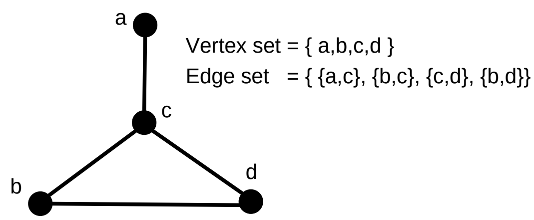


Figure 1: An example for a simple graph

Note that, graph is a non-linear data structure. A graph is said to be simple if it has no multiple edges between any two vertices and no self loops (an edge from a vertex to itself). Throughout this lecture, we will look at only finite and simple graphs.

Definition 2 (Subgraph) *Let G be a finite simple graph. A graph H is said to be a **subgraph** of G if $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. The graph H is said to be an **induced subgraph** of G if $V(H) \subseteq V(G)$ and for every pair of vertices u and v, $\{u, v\} \in E(H)$ if and only if $\{u, v\} \in E(G)$.*

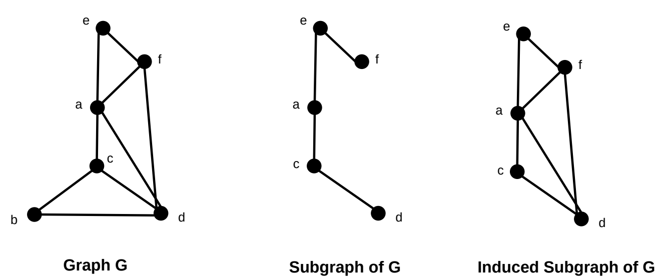


Figure 2: A graph G, a sub graph of G on the vertex set $\{a, c, d, e, f\}$ and an induced graph on the vertex set $\{a, c, d, e, f\}$

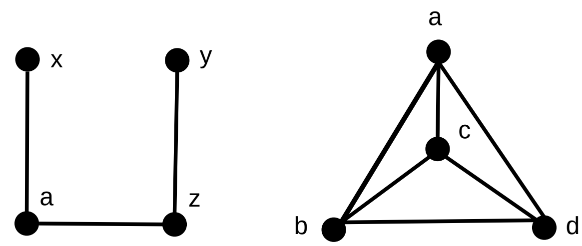


Figure 3: An example for a connected graph

Definition 3 (Connected Graph) *Two vertices u and v of a graph G are said to be connected if there exists a path from u to v in G. A graph G is said to be* ***connected*** *if every pair of its vertices are connected.*

Definition 4 (Disconnected Graph) *A graph G which is not connected is said to be* ***disconnected****.*

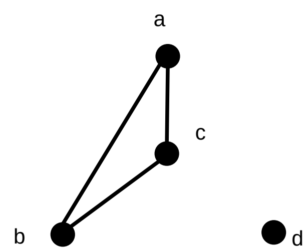


Figure 4: An example for a disconnected graph. The graph has two connected components, one is a graph induced on the vertex set $\{a, b, c\}$ and the other is $\{d\}$

Definition 5 (Neighborhood of a vertex) *The* ***Neighborhood of a vertex*** *v in a graph G is the set $N(u)$ consisting of all vertices u which are adjacent to v. For example, the neighborhood of the vertex c in Fig. 3. is $N(c) = \{a, b\}$.*

Definition 6 (Acyclic Graph) *A graph that contains no cycles is called* ***acyclic graph****.*

Definition 7 (Tree) *A connected acyclic graph is called a* ***tree****. It is a* ***spanning tree*** *of a graph G if it spans G (that is, it includes every vertex of G) and is a sub-graph of G (every edge in the tree belongs to G).*

- *In a tree, we can find only one path for every pair of its vertices.*
- *Tree is a non-linear data structure.*

Definition 8 (Bipartite Graph) *A graph G is called a* ***bigraph*** *or* ***bipartite graph*** *if V can be partitioned into two disjoint subsets V_1 and V_2 such that every line of G joins a point of V_1 to a point of V_2. (V_1, V_2) is called a bipartition of G.*

1.1 Breadth First Search(BFS) Algorithm

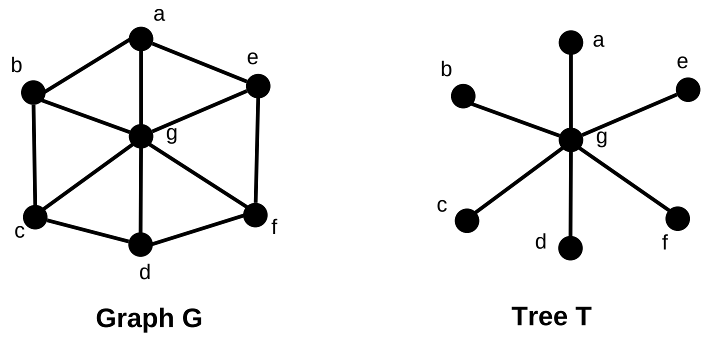


Figure 5: An example for a graph and its corresponding spanning tree

---

Algorithm 1 BFS Spanning tree algorithm(G)

Input: A Graph G=(V,E)
Output: Spanning Tree T of a graph G
Step 1: Let i=0.
Step 2: Start with any vertex v in G. Add v in level i of a tree T; $i = i + 1$.
Step 3: Find the neighbors of v and add it in level i of a tree T.
Step 4: Find the neighbors (only the unvisited neighbors) for every vertex in level i and add it in level $i + 1$.
Step 5: Repeat step 4 until there are no neighbors to visit.

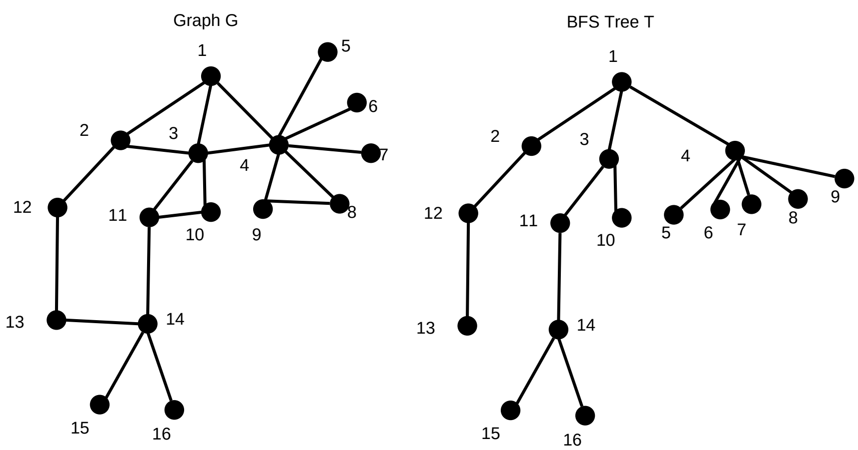


Figure 6: An example for the construction of BFS tree

Time complexity: $O(n)$ effort is spent in initializing the boolean array to keep track of which node is visited/unvisited. As part of BFS procedure, each edge is visited at most once, therefore, the total time is $O(n + m)$, where $n \rightarrow$ vertices and $m \rightarrow$ edges of a graph G.

$E(G)$ denotes the edges in a graph G, $E(T)$ denotes the edges in the BFS tree T of G. The set $E_n = E(G) \backslash E(T)$ denotes the set of non-tree edges i.e., the edges which are in the graph G but not in the tree T.

Definition 9 *A* ***non-tree (missing) edge****,* $\{u, v\} \in E(G) \backslash E(T)$ *is said to be a:*

- *Cross edge if $u \in L_i$ and $v \in L_i$ (i.e., both the vertices are in same level). Let E_c denotes the set of all cross edges in T.*
- *Slanting edge if $u \in L_i$ and $v \in L_j$, $j = i+1$ or $j = i-1$ (i.e., both the vertices are in adjacent levels). Let E_s denotes the set of all cross edges in T.*

Remark: Note that, j can not be greater than $i+1$ or less than $i-1$.

1.1.1 Applications of BFS

- **Test for Connectedness**:
  Problem: Given a graph G, find whether the given graph is connected or not ?
  Solution using BFS: Call BFS algorithm once, if $\mid V(G) \mid = \mid V(T) \mid$, then G is connected and if $\mid V(G) \mid \neq \mid V(T) \mid$, then G is disconnected, where T is the BFS tree constructed in the first call to BFS algorithm. i.e., if number of calls to BFS is greater than one, then G is disconnected and the number of calls to BFS gives the number of disconnected components.

- **Test for cyclicity**:
  Problem 1: Given a connected graph G, find whether G contains a cycle or not?
  Solution using BFS: Run BFS(G). If $E_n = \emptyset$, then G is acyclic. Otherwise G contains at least one cycle.

  Problem 2: Given a graph G, find whether G contains a cycle or not?
  Solution using BFS: Run BFS for each connected component of G and check if $E_n = \emptyset$ for all such components, if so, then G is acyclic. Otherwise G contains at least one cycle.

  Problem 3: Given a graph G, find whether G is a tree or not?
  Solution using BFS: Do test for connectedness and test for acyclicity. If G is connected and acyclic, then G is a tree.

- **Test for existence of C_3**:
  Problem: Given a graph G, find whether G contains a C_3 or not?
  Solution using BFS: Run BFS(G) and collect all non-tree edges, E_n. For all, $e = \{u, v\} \in E_n$, check whether the shortest path between u and v in $G' = G \backslash e$ is two (i.e., check for a common neighbor in the same level of u or in the adjacent levels of u), if so, G contains a C_3. This check can be done in $O(m) \times O(n+m)$ time, as for every non-tree edge ($O(m)$), construction of BFS for shortest path computation takes $O(n+m)$ time.

  Approach 2: For every edge $\{u, v\} \in E(G)$, check whether $N_G(u) \cap N_G(v) \neq \emptyset$ or not. If it is empty, then G does not contain a C_3. Otherwise, G has at least one C_3. Time complexity: In a adjacency matrix of G, do Boolean AND for the two rows corresponding to the end vertices of a cross edge and scan for the existence of 1 in that, this takes $O(n)$ time and we do this for all edges in G, $O(m)$. Thus, this algorithm takes $O(nm)$ time.

- **Test for existence of odd cycles**:
  Problem: Given a graph G, find whether G contains an odd cycle, a cycle of length $2k+1, k \geq 1$, or not?
  Solution using BFS: The existence of cross edges in T ($E_c \neq \emptyset$) implies the existence of odd cycles in G. Let $e = \{u, v\}$ be an cross edge in T and let x be the common parent of u and v. It is clear that, the length of P_{xu} (Path from x to u in T) is equal to the length of P_{xv}. Thus, P_{xu} and P_{xv} forms an odd cycle together with e. The converse of this statement: The existence of odd cycles in G implies the existence of cross edges in T ($E_c \neq \emptyset$) is also true.

- **Test for existence of C_4**:
Problem: Given a graph G, find whether G contains a C_4 or not?
Solution using BFS: Run BFS(G) and collect all non-tree edges, E_n. For all, $e = \{u, v\} \in E_n$: collect $A = N_{G'}(u) \backslash N_G(v)$ and $B = N_{G'}(v) \backslash N_G(u)$, where $G' = G \backslash e$, and for every element in A, check whether it has a neighbor in B or not. If it has a neighbor then there exists a C_4. Time complexity: Number of non-tree edges is $O(m)$, number of elements in A is $O(n)$ and in B is $O(n)$. Thus, this approach takes $O(mn^2)$ time.

- **Test for existence of even cycles**:
Problem: Given a graph G, find whether G contains an even cycle, a cycle of length $2k+2, k \geq 1$, or not?
Solution using BFS: The existence of slanting edges in T ($E_s \neq \emptyset$) implies the existence of even cycles in G. Let $e = \{u, v\}$ be a slanting edge in T and let x be the common parent of u and v. It is clear that, the length of P_{xu} (Path from x to u in T) is equal to the length of $P_{xv} + 1$ or the length of P_{xv} (Path from x to u in T) is equal to the length of $P_{xu} + 1$. Thus, P_{xu} and P_{xv} forms an even cycle together with e. The converse of this statement if false. i.e., consider a complete graph on four vertices, consider the BFS tree with respect to vertex 1, in the tree there is an even cycle using four cross edges.

- **Test for Bipartiteness**:
Problem: Given a graph G, find whether G is a bipartite graph or not?
<u>**Trivial Algorithm:**</u> Partition the vertex set V of G into two sets with different combinations and check for the bipartiteness for each combination. Time complexity of this algorithm $= n + nC_2 + \ldots + nC_{n/2} = O(2^n)$.
<u>**Algorithm using BFS:**</u>
  - Mark the non-tree (missing) edges using dotted lines in the BFS tree T.
  - Decompose E_n into E_c and E_s.
  - We know that, a graph is bipartite if and only if it is odd cycle free. By using the fact in test for odd cycles: we can conclude that, if $E_c = \emptyset$, then G is bipartite.

- **Test for 2-colorability**:
Problem: Given a graph G, check whether can we color the vertices of a graph G using two colors such that no two adjacent vertices have the same color.
Solution using BFS: Testing whether a graph is 2-colorable or not is equivalent to testing whether a graph is bipartite or not.

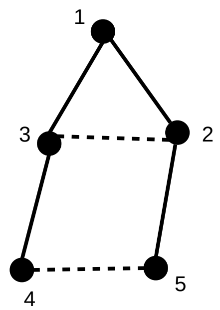
- **Shortest path computation**:
Problem: Given a graph G and two vertices u and v, find the shortest path between u and v.
Solution using BFS: Run BFS(G) by having starting vertex as u. Since, T=BFS(G) is a tree, there exist only one path from u to v and that path is the shortest path from u to v. This takes $O(n+m)$ time.

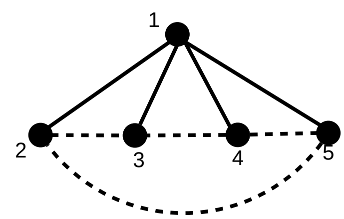
- **All pairs Shortest path problem**:
Problem: Given a graph G, find the shortest path between all pairs of vertices in G.
Solution using BFS: For every vertex $v \in V(G)$, Run BFS(G) by having starting vertex as v. For each BFS tree T, print the path from v to x for all x in T. This takes $O(n(n+m))$ time.

Remarks:

- If $|E_c| = \emptyset$ then G is bipartite.

- If $|E_c| \neq \emptyset$ then G is not bipartite.
- If $|E_c| = \emptyset$ and $|E_s| = \emptyset$ then the given graph G is a tree (bipartite).
- Even cycles can be formed by only cross edges. i.e., even cycle may exist even in case of $E_s = \emptyset$. Examples are as follows:





1.2 Depth First Search(DFS) Algorithm

Algorithm 2 DFS Spanning tree algorithm(G)

Input: A Graph G=(V,E)
Output: Spanning Tree T of a graph G
Step 1: Let i=0.
Step 2: Start with any vertex v in G. Add v in level i of a tree T; $i = i + 1$.
Step 3: Find any one neighbor of v and add it in level i of a tree T.
Step 4: Find any one neighbor (only the unvisited neighbor) for the vertex in level i and add it in level $i + 1$.
Step 5: When there is no neighbor to visit, backtrack from the last level. Otherwise, Repeat step 4.

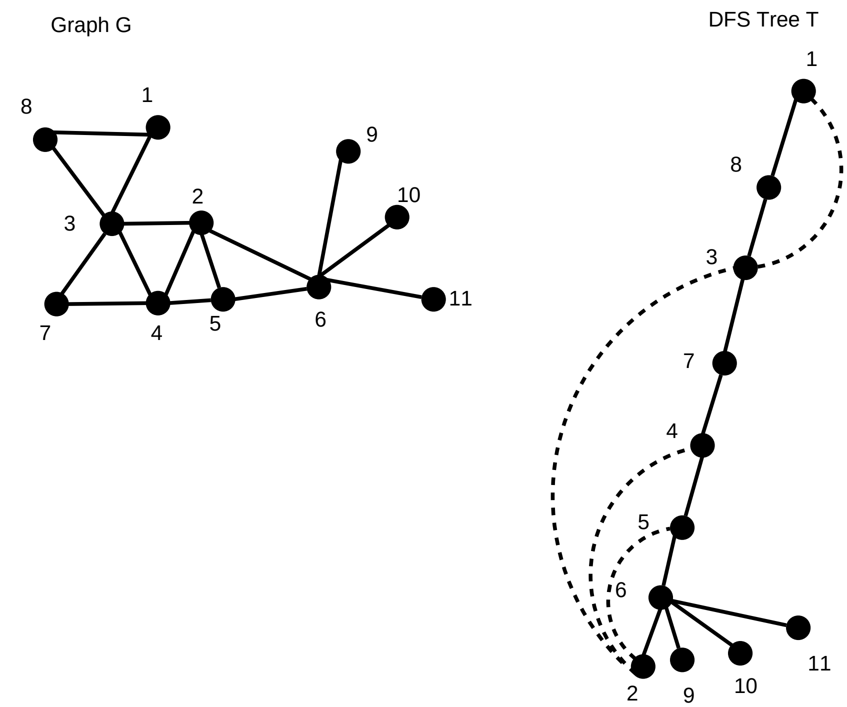


Figure 7: An example for the construction of DFS

Time complexity: Since each edge is visited at most twice: one during DFS call and the other visit is during back tracking, the effort is $O(n + m)$, where n $\rightarrow$ vertices and m $\rightarrow$ edges of a graph G.
Note: Here the non-tree edges are called as back edges.

1.2.1 Applications of DFS

- **Test for Connectedness**:
  Problem: Given a graph G, find whether the given graph is connected or not ?
  Solution using DFS: Call DFS algorithm once, if $\mid V(G) \mid = \mid V(T) \mid$, then G is connected and if $\mid V(G) \mid \neq \mid V(T) \mid$, then G is disconnected, where T is the DFS tree constructed in the first call for DFS algorithm. i.e., if number of calls to DFS is greater than one, then G is disconnected.

- **Test for cyclicity**:
  Problem 1: Given a connected graph G, find whether G contains a cycle or not?
  Solution using DFS: Run DFS(G). If there is no back edge, then G is acyclic. Otherwise G contains at least one cycle.

  Problem 2: Given a graph G, find whether G contains a cycle or not?
  Solution using DFS: Run DFS for each connected component of G and check if the number of back edges is equal to zero for all such components, if so, then G is acyclic. Otherwise G contains at least one cycle.

  Problem 3: Given a graph G, find whether G is a tree or not?
  Solution using DFS: Do test for connectedness and test for acyclicity. If G is connected and acyclic, then G is a tree.

- **Determine the number of connected components**:
  Problem: Given a graph G, find the number of connected components in G.
  Solution using DFS: Run DFS until $V(G) = V(T)$. The number of calls to DFS determines the number of connected components in G.

Definition 10 (Articulation Point/Critical node) *Let G be a connected graph. A vertex $v \in V(G)$ is said to be an **articulation point** if the removal of the vertex v from G results in a disconnected graph.*

Definition 11 (Bridge/Critical link) *Let G be a connected graph. A edge $e \in E(G)$ is said to be a **bridge** if removal of the edge e from G results in a disconnected graph.*

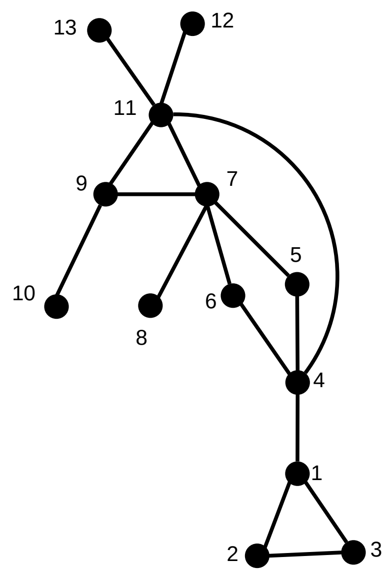


Figure 8: Articulation Points: vertex 1, vertex 4, vertex 7, vertex 9 and vertex 11; Bridges: {1,4}, {7,8}, {9,10}, {11,12} and {11,13}.

Note:

- If a network doesn't contains a bridge and an articulation point then it is a good network.
- If G is 2-connected then it can tolerate single node failures but not 2-node failures
- It is not necessary that existence of articulation point implies the existence of bridges.
- But if there is a bridge in a graph G then there exist atleast one articulation point.
- The upper bound for number of articulation points in a connected graph G with n vertices is $n - 2$.
- The upper bound for number of bridges in a connected graph G with n vertices is $n - 1$.

- **Test for existence of an articulation point**:
  Problem: Given a graph G, find the articulation points in G.
  Approach 1: For every vertex $v \in V(G)$, run $DFS(G' = G\backslash\{v\})$, if the number of connected components is greater than one, then the vertex v is an articulation point. This approach takes $O(n(n + m)) = O(nm)$ time.

Approach 2: The vertex w in a DFS tree T is said to be an articulation point if there is no back edge from the descendant vertices of w to the ancestor vertices of w. The root node of T is an articulation point if degree of the root node in DFS tree is greater than or equal to two. This can be done using the following algorithm:

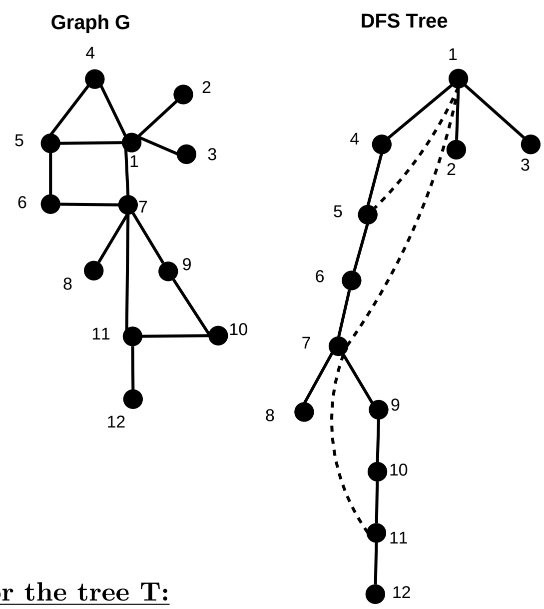
Algorithm 3 To compute Articulation point

Input: DFS tree T of a Graph G=(V,E) and a vertex u.
Output: Whether the vertex u is an articulation point or not.
Step 1: W.r.t. T,
Compute $L(u) = min\{dfn(u), min\{L(w)|w \text{ is a child of } u\}, min\{dfn(w)|(u,w) \text{ is a back edge}\}\}$
Step 2: If u is a root in T with degree ≥ 2 then u is an articulation point.
Step 3: If u is not a root in T then u is an articulation point iff u has a child w such that $L(w) \geq dfn(u)$

Remark: What does for a vertex v, for all child w_i, $L(w_i) \geq dfn(v)$ mean ? It means that, there does not exist a back edge from any descendant of v to any antecedent of v. Moreover, to check whether a vertex v is an articulation point or not, it is enough to check whether there exist a child for v whose descendants do not have a back edge to any antecedent of v.

Time Complexity to list all APs: $O(n+m)$ [$O(n+m)$ for DFS Tree + $O(n)$ for post order traversal + $O(n)$ for checking whether it is A.P]

Example 1:



Vertex	dfn
1	1
2	11
3	12
4	2
5	3
6	4
7	5
8	6
9	7
10	8
11	9
12	10

<u>**L - Values for the tree T:**</u>

$L(8) = min\{6, \infty, \infty\} = 6$

$L(12) = min\{10, \infty, \infty\} = 10$

$L(11) = min\{9, 10, 5\} = 5$

$L(10) = min\{8, 5, \infty\} = 5$

$L(9) = min\{7, 5, \infty\} = 5$

$L(7) = min\{5, min\{6, 5\}, 1\} = 1$

$L(6) = min\{4, 1, \infty\} = 1$

$L(5) = min\{3, 1, 1\} = 1$

$L(4) = min\{2, 1, \infty\} = 1$

$L(3) = min\{12, \infty, \infty\} = 12$

$L(2) = min\{11, \infty, \infty\} = 11$

$L(1) = min\{1, min\{11, 12, 1\}, \infty\} = 1$

By Algorithm 3, Articulation points are 1,7,11.

Query 1: Why not, v is an articulation point if there exist at least one child of v, say w, such that $L(v) \leq L(w)$?
Counter example: In the example, given above, for a vertex 10, there exist a child 12 such that $L(12) = 10 > L(10) = 5$, but 10 is not an articulation point.

Query 2: Why not, v is an articulation point if there exist at least one child of v, say w, such that $dfn(v) \geq L(w)$?
Counter example: In the example, given above, for a vertex 9, there exist a child 10 such that $dfn(9) = 7 > L(10) = 5$, but 9 is not an articulation point.

- **Test for the existence of a bridge**:
  Problem: Given a graph G, find the bridges in G.
  Approach 1: For every edge $e \in E(G)$, run $DFS(G' = G \backslash e)$, if the number of connected components is greater than one, then the edge e is a bridge. This approach takes $O(m(n+m)) = O(m^2)$ time.

  Approach 2: The edge $\{u, v\}$ in a DFS tree is said to be a bridge if there is no back edge from the descendant vertices of v to u or to the ancestor vertices of u. This can be done using the following algorithm: By Algorithm 4, the bridges for *Example* 1 are {1,2}, {1,3}, {11,12}, {7,8}.

Algorithm 4 To compute Bridges

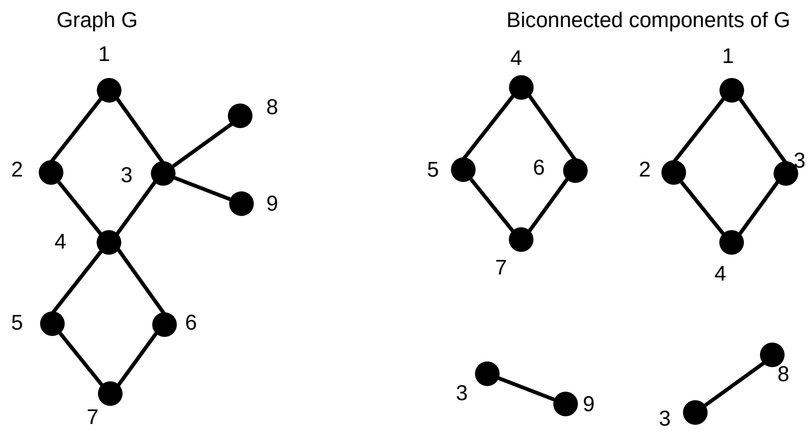
Input: DFS tree T of a Graph G=(V,E) and an edge (u, v).
Output: Whether the edge (u, v) is bridge or not.
Step 1: W.r.t. T,
Compute $L(u) = min\{dfn(u), min\{L(w) | w\, is\, a\, child\, of\, u\}, min\{dfn(w) | (u, w) is\, a\, back\, edge\}\}$
Step 2: (u, v) is a bridge if $dfn(u) < dfn(v)$ and $L(v) > dfn(u)$.

Query 1: Why not, the edge $\{u, v\}$ is a bridge if (i) $\{u, v\}$ is a tree edge (ii) $dfn(u) < dfn(v)$ and (iii) $L(v) = dfn(v)$?

Since the above condition respects the definition of bridge, and in particular, condition (iii) ensures there is no back edge from any descendant of v to u or the ancestor of u, the above check indeed works.

Query 2: Why not, the edge $\{u, v\}$ is a bridge if there exist a child w for v such that $L(w) \geq L(v)$?
Counter example: Consider the Example 1: Take an edge $\{10, 11\}$, there exist a child 12 for 11 such that $L(12) = 10 > L(11) = 5$, but $\{10, 11\}$ is not a bridge.

Definition 12 (Biconnected Components) *A maximal connected components of a graph G without any articulation point.*



- **Determine the biconnected components**:
  Problem: Given a graph G, find all the biconnected components of the graph G.

Algorithm 5 Biconnected Components

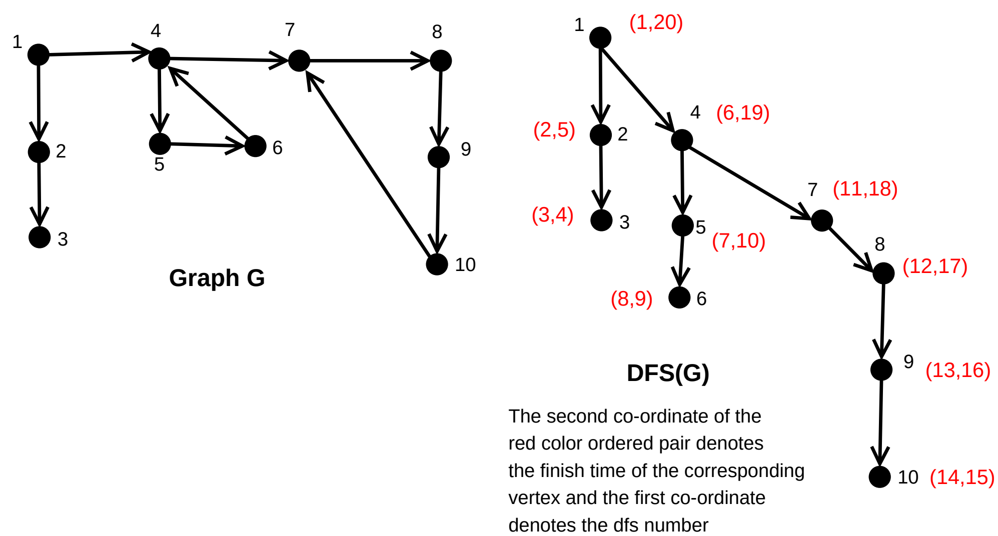
Input: Graph G=(V,E)
Output: Biconnected Components.
Step 1: Identify any one articulation point.
Step 2: Remove that point from the graph. We will get collection of connected graphs G'
Step 3: Add back the articulation point to all the connected components.
Step 4: The connected components which has no articulation point are biconnected. For the components which has articulation point, repeat the above process.

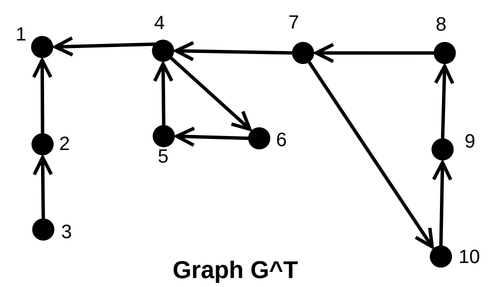
Strongly Connected Components in a directed graph G:

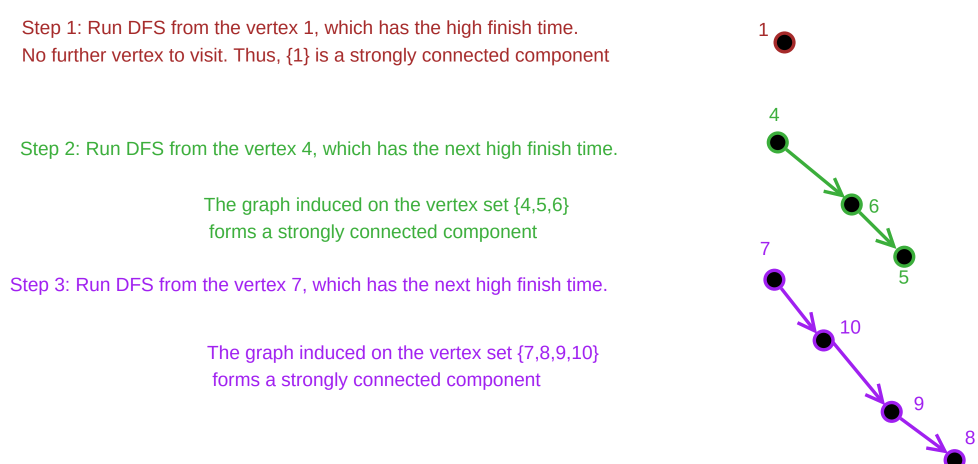
Problem: Given a directed graph G, find all of its strongly connected components.
An example which traces this algorithm is as follows:

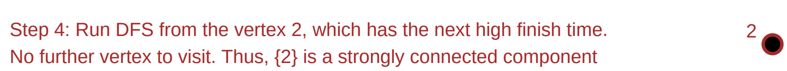
Algorithm 6 Strongly Connected Components (SCC)

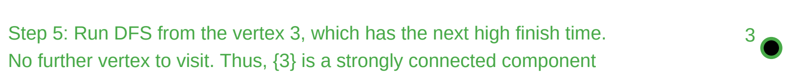
Input: Directed Graph G
Output: Strongly Connected Components of G.
Step 1: Run DFS(G) and compute the finish time for all vertices.
Step 2: Find G^T and sort the vertex set of G in decreasing order based on its finish time.
Step 3: Run $DFS(G^T)$ from the vertex which has maximum finish time. Do this step repeatedly until all the vertices in G^T are visited at least once (this gives you the collection of SCC).
Step 4: Each forest in $DFS(G^T)$ is a SCC.







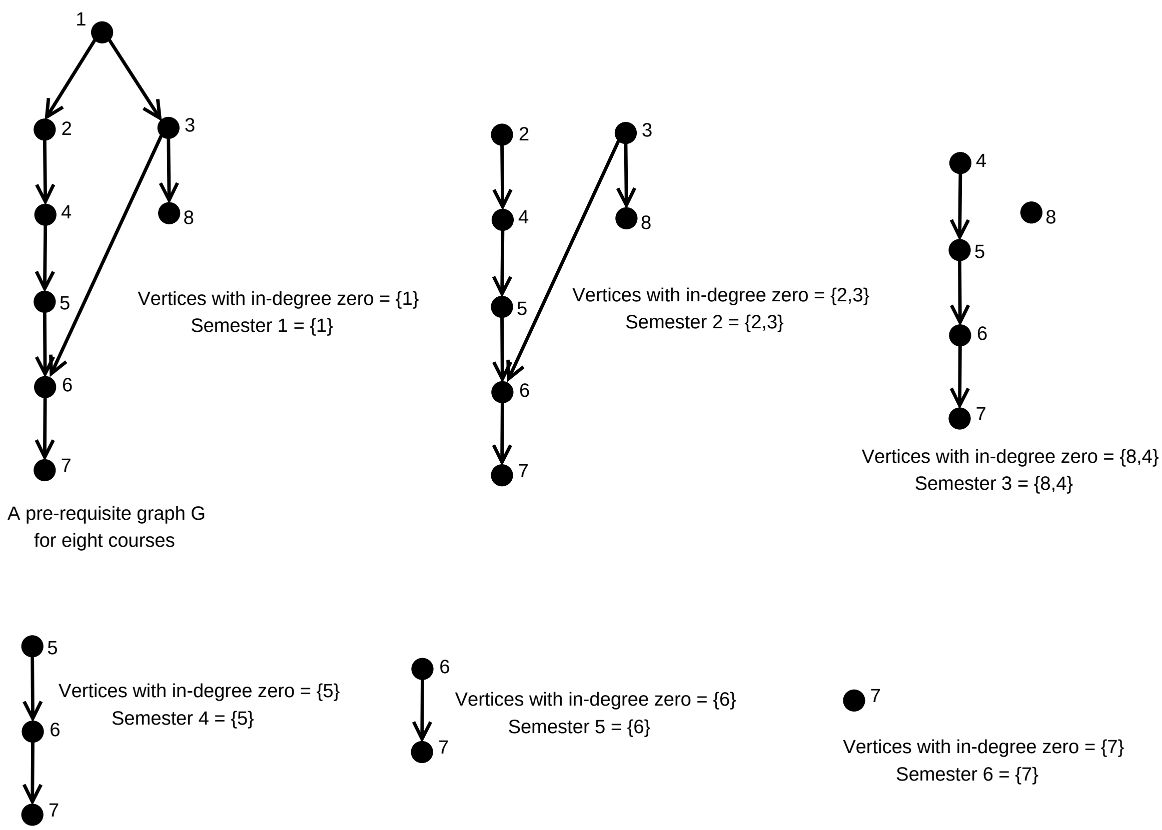




How long will it take to complete B.Tech programme ?

Problem: Every B.Tech student in IIITD&M has to complete the set of 55 courses in the curriculum to get their degree certificate. If you are given a chance to do as many courses as possible in a semester with a constraint: some courses has a pre-requisite course, which has to be completed in the previous semesters, what is the minimum number of semesters to complete all 55 courses ? (The question maximum is invalid because one can do one course in a semester to reach the maximum number)

Strategy 1: Construct a pre-requisite graph on 55 courses. i.e., construct a graph with 55 vertices (each vertex corresponds to a course) and an directed edge $(u, v) \in E(G)$ if the course u is the pre-requisite for the course v. Now, remove the vertices of in-degree zero and add the corresponding courses in semester 1. Repeat this process until you have visited all the vertices in the graph G, to get the minimum number of semesters. An example is illustrated below:



The minimum number of required semesters is 6

Strategy 2: Run DFS and compute the height for each forest in the DFS. Pick the maximum height. This strategy fails because of the following counter example:

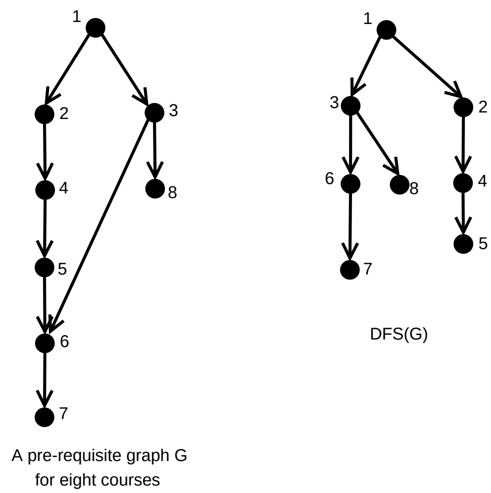


Figure 9: Course 6 has to be done before the course 5, which is a contradiction as course 5 is a pre-requisite course for course 6

Acknowledgements: Lecture contents presented in this module and subsequent modules are based on the following text books and most importantly, author has greatly learnt from lectures by algorithm exponents affiliated to IIT Madras/IMSc; Prof C. Pandu Rangan, Prof N.S.Narayanaswamy, Prof Venkatesh Raman, and Prof Anurag Mittal. Author sincerely acknowledges all of them. Special thanks to Teaching Assistants Mr.Renjith.P and Ms.Dhanalakshmi.S for their sincere and dedicated effort and making this scribe possible. Author has benefited a lot by teaching this course to senior undergraduate students and junior undergraduate students who have also contributed to this scribe in many ways. Author sincerely thanks all of them.

References:

1. E.Horowitz, S.Sahni, S.Rajasekaran, Fundamentals of Computer Algorithms, Galgotia Publications.
2. T.H. Cormen, C.E. Leiserson, R.L.Rivest, C.Stein, Introduction to Algorithms, PHI.
3. Sara Baase, A.V.Gelder, Computer Algorithms, Pearson.