Graph Traversal and Search Techniques

By: Ashok Basnet

Lecture Outline

- Graphs
- Binary Tree Traversal Technique and Search
- Graph Traversal Technique and Search: BFS and DFS
- Connected Components and Spanning Trees
- Bi-connected Components and DFS

Binary Tree Traversal Technique

- Search means finding a path or traversal between a start node and one of a set of goal nodes.
- Search is a study of states and their transitions. Search involves visiting nodes in a graph in a systematic manner, and may or may not result into a visit to all nodes.
- When the search necessarily involved the examination of every vertex in the tree, it is called the traversal.
- There are three common ways to traverse a binary tree:
  - **Preorder**
  - **Inorder**
  - **postorder**

Binary Tree Traversal Technique

- In all the three traversal methods, the left subtree of a node is traversed before the right subtree.
- The difference among the three orders comes from the difference in the time at which a node is visited.
- ***Inorder Traversal:***
- In the case of inorder traversal, the root of each subtree is visited after its left subtree has been traversed but before the traversal of its right subtree begins. The steps for traversing a binary tree in inorder traversal are:
  - Visit the left subtree, using inorder.
  - Visit the root.
  - Visit the right subtree, using inorder.

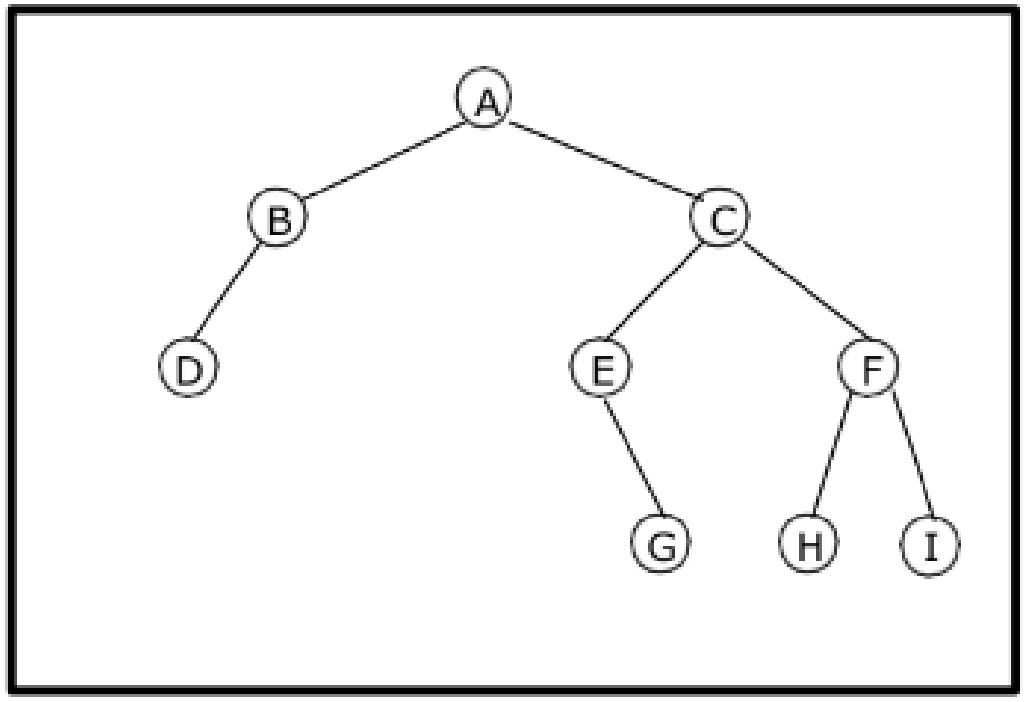
Binary Tree Traversal Technique

- ***Preorder Traversal:***
- In a preorder traversal, each node is visited before its left and right subtrees are traversed. Preorder search is also called backtracking. The steps for traversing a binary tree in preorder traversal are:
  - Visit the root.
  - Visit the left subtree, using preorder.
  - Visit the right subtree, using preorder.
- ***Postorder Traversal:***
- In a postorder traversal, each root is visited after its left and right subtrees have been traversed. The steps for traversing a binary tree in postorder traversal are:
  - Visit the left subtree, using postorder.
  - Visit the right subtree, using postorder
  - Visit the root.

Binary Tree Traversal Technique

Example 1:

Traverse the following binary tree in pre, post and in-order.



Bin a ry T ree

Preordering of the vertices:
A, B, D, C, E, G, F, H, I.

Postordering of the vertices:
D, B, G, E, H, I, F, C, A.

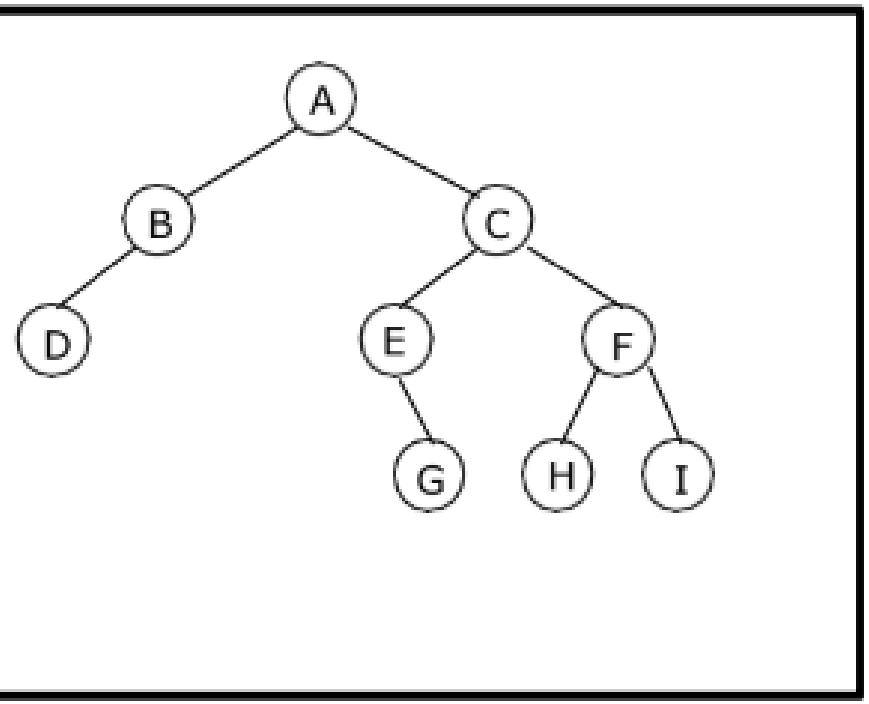
Inordering of the vertices:
D, B, A, E, G, C, H, F, I

Pre, Post and In-order Traversing

Binary Tree Traversal Technique

Example 2:

Traverse the following binary tree in pre, post, inorder and level order.



Bin a ry T re e

- Preo rder tra versal yields:
  A, B, D, C , E, G , F , H, I
- Posto rder tra versal yields:
  D, B, G , E, H, I, F , C , A
- Ino rder tra versal yields:
  D, B, A, E, G , C , H, F , I
- Level o rder tra versal yields:
  A, B, C , D, E, F , G , H, I

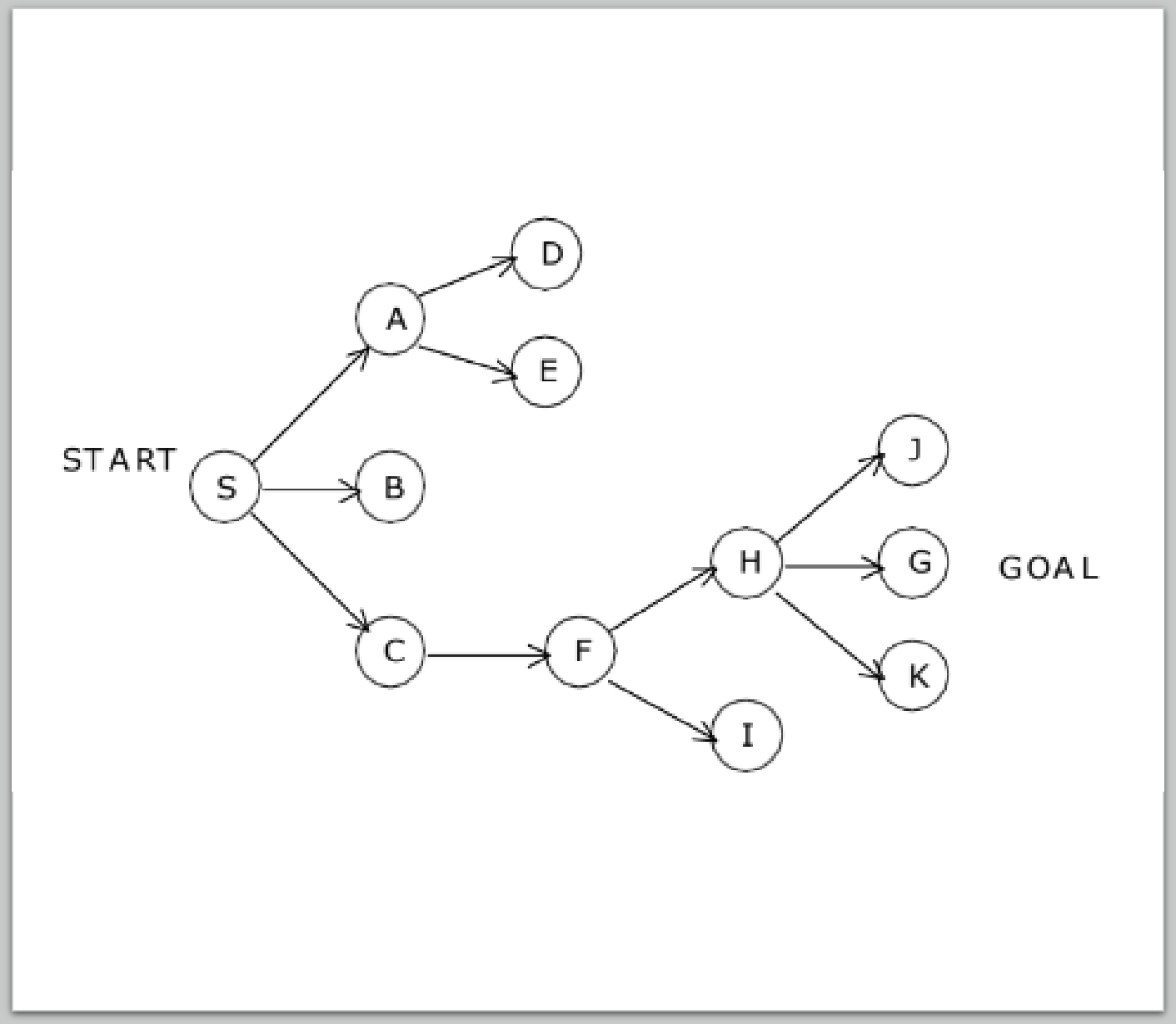
Pre, Post, Inorder and level order Traversing

BFS and DFS

- Given a graph G = (V, E) and a vertex V in V (G) traversing can be done in two ways.
  - ***Depth first search***
  - ***Breadth first search***

Depth First Search

- With depth first search, the start state is chosen to begin, then some successor of the start state, then some successor of that state, then some successor of that and so on, trying to reach a goal state.
- If depth first search reaches a state S without successors, or if all the successors of a state S have been chosen (visited) and a goal state has not get been found, then it “backs up” that means it goes to the immediately previous state or predecessor formally, the state whose successor was ‘S’ originally.



Depth First Search

- For example consider the figure. The circled letters are state and arrows are branches.
- Suppose S is the start and G is the only goal state. Depth first search will first visit S, then A then D. But D has no successors, so we must back up to A and try its second successor, E. But this doesn't have any successors either, so we back up to A again.
- But now we have tried all the successors of A and haven't found the goal state G so we must back to 'S'.
- Now 'S' has a second successor, B. But B has no successors, so we back up to S again and choose its third successor, C.
- C has one successor, F. The first successor of F is H, and the first of H is J. J doesn't have any successors, so we back up to H and try its second successor. And that's G, the only goal state. So the solution path to the goal is S, C, F, H and G and the states considered were in order S, A, D, E, B, C, F, H, J, G.

Depth First Search

Disadvantages:

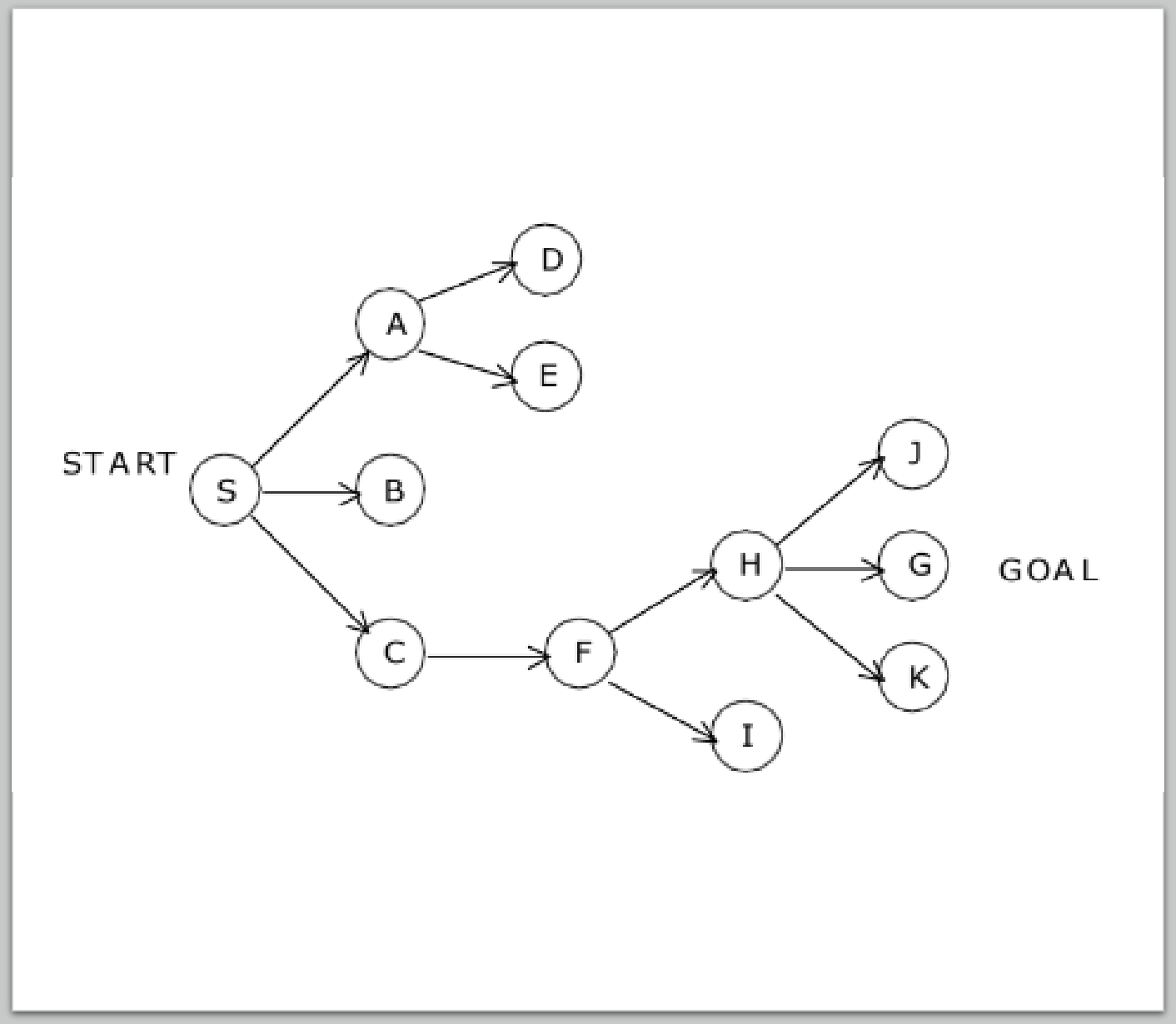
- It works very fine when search graphs are trees or lattices, but can get struck in an infinite loop on graphs. This is because depth first search can travel around a cycle in the graph forever. **To eliminate this keep a list of states previously visited, and never permit search to return to any of them.**
- One more problem is that, the state space tree may be of infinite depth, to prevent consideration of paths that are too long, a maximum is often placed on the depth of nodes to be expanded, and any node at that depth is treated as if it had no successors.
- We cannot come up with shortest solution to the problem.

Breadth First Search

- Given an graph **G = (V, E),** breadth-first search starts at some source vertex S and “discovers" which vertices are reachable from S. Define the distance between a vertex V and S to be the minimum number of edges on a path from S to V.
- Breadth-first search discovers vertices in increasing order of distance, and hence can be used as an algorithm for computing shortest paths (where the length of a path = number of edges on the path).
- Breadth-first search is named because it visits vertices across the entire breadth.
- Breadth first search does not have the danger of infinite loops as we consider states in order of increasing number of branches (level) from the start state.
- To illustrate this let us consider the following tree:

Breadth First Search

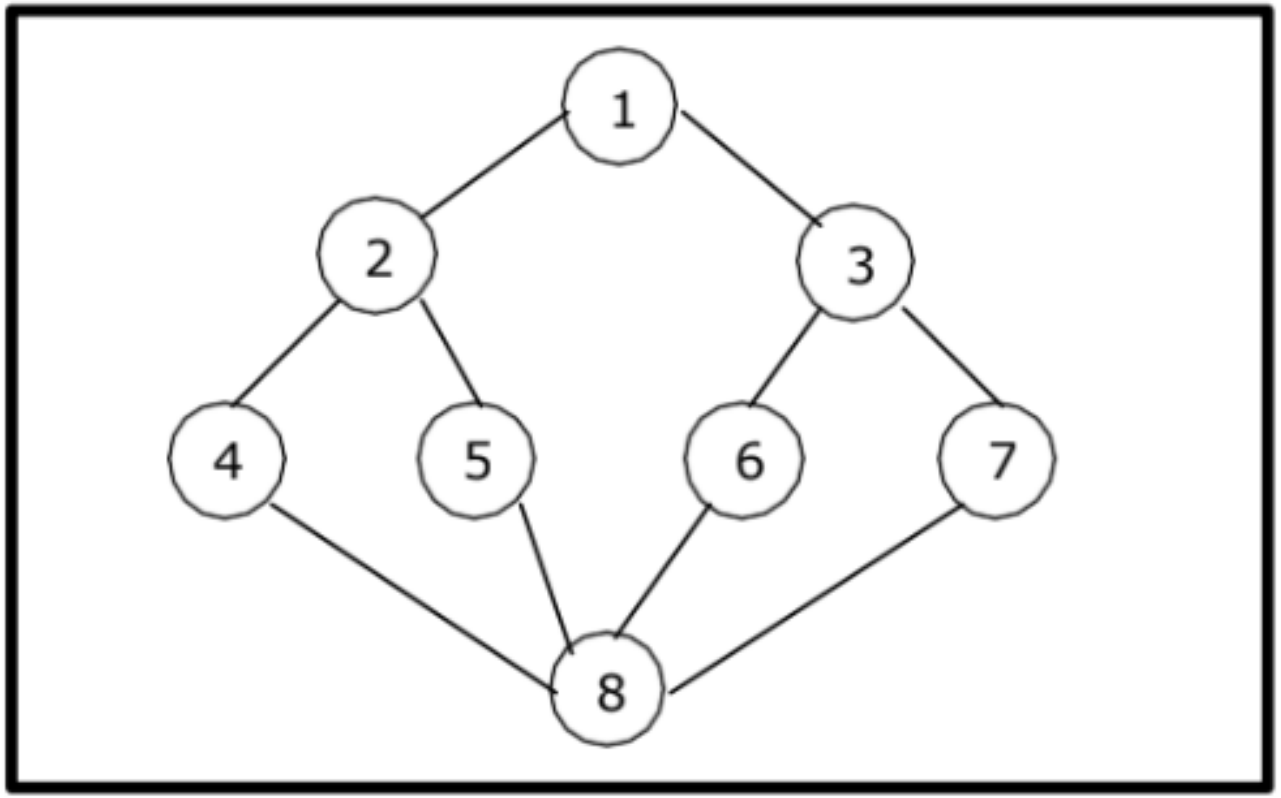
- Breadth first search finds states level by level. Here we first check all the immediate successors of the start state. Then all the immediate successors of these, then all the immediate successors of these, and so on until we find a goal node. Suppose S is the start state and G is the goal state. In the figure, start state S is at level 0; A, B and C are at level 1; D, e and F at level 2; H and I at level 3; and J, G and K at level 4. So breadth first search, will consider in order S, A, B, C, D, E, F, H, I, J and G and then stop because it has reached the goal node.



Depth First and Breadth Spanning Trees

- BFS and DFS impose a tree (the BFS/DFS tree) along with some auxiliary edges (cross edges) on the structure of graph. So, we can compute a spanning tree in a graph.
- The computed spanning tree is not a minimum spanning tree. Trees are much more structured objects than graphs. For example, trees break up nicely into subtrees, upon which subproblems can be solved recursively.
- For directed graphs the other edges of the graph can be classified as follows:
- **Back edges:** (u, v) where v is a (not necessarily proper) ancestor of u in the tree. (Thus, a self-loop is considered to be a back edge).
- **Forward edges:** (u, v) where v is a proper descendent of u in the tree.
- **Cross edges:** (u, v) where u and v are not ancestors or descendants of one another (in fact, the edge may go between different trees of the forest).

BF and DF Spanning Tree



Gra ph

Articulation Points

- An **articulation point** (or cut vertex) in a graph is a vertex whose removal disconnects the graph.
- In other words, if you were to remove an articulation point, the graph would become disconnected or have more components than before.
- Articulation points are critical for maintaining the connectivity of a graph.
- The presence of articulation points is often associated with vulnerabilities in network communication.

Biconnected Components

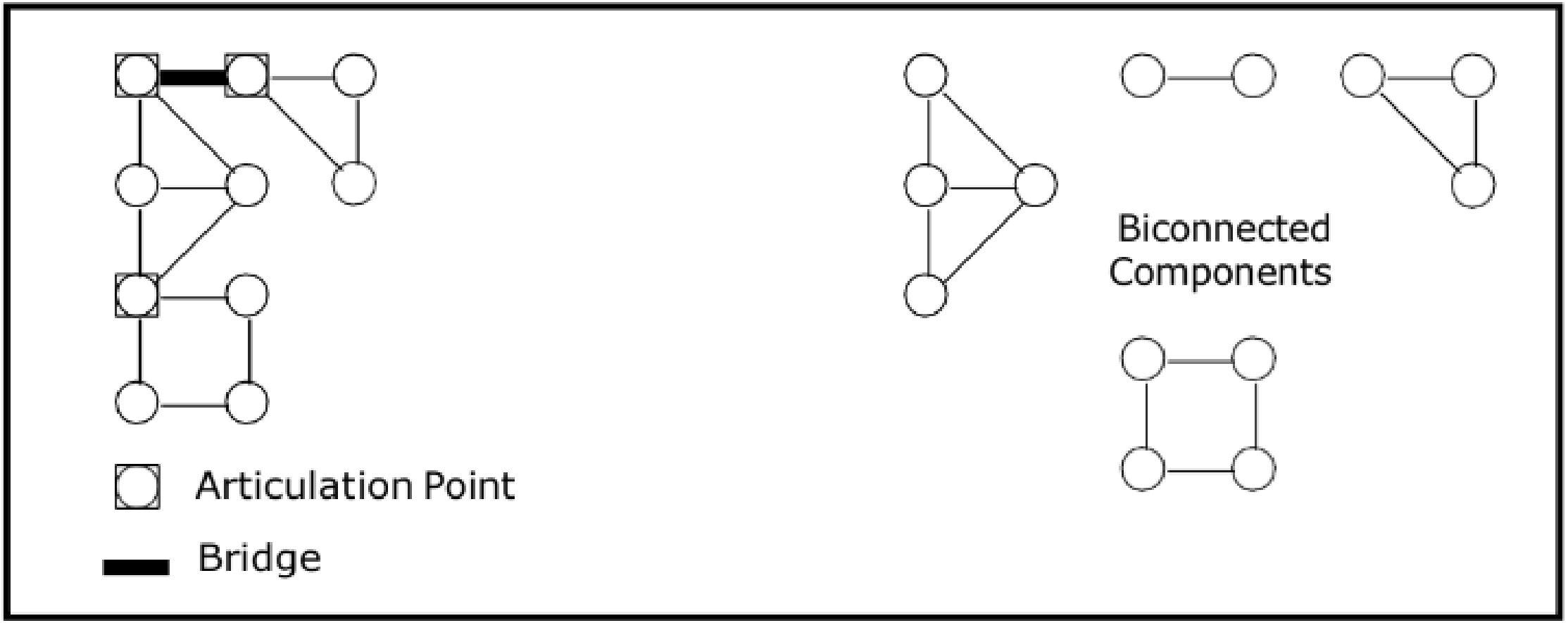
- A biconnected component in a graph is a maximal subgraph in which any two vertices are connected by at least two disjoint simple paths.
- In simpler terms, a biconnected component is a portion of the graph that is highly connected; even if you remove any single vertex (and its incident edges), the remaining graph will still be connected.
- Biconnected components are important for understanding the robustness of a graph and identifying parts of the graph that can withstand the removal of certain nodes.

Articulation Points and Biconnected Components

- Articulation points and biconnected components are concepts related to the analysis of connectivity in a graph.
- Articulation points play a crucial role in the identification of biconnected components.
- Removing an articulation point often results in breaking the graph into multiple biconnected components.
- The graph itself can be viewed as a union of its biconnected components, and articulation points are the vertices that connect these components.
- In summary, articulation points are key vertices whose removal can separate a graph into multiple connected components, and biconnected components are maximal subgraphs that remain connected even after the removal of any single vertex.
- The analysis of these concepts helps in understanding the connectivity and robustness of a graph

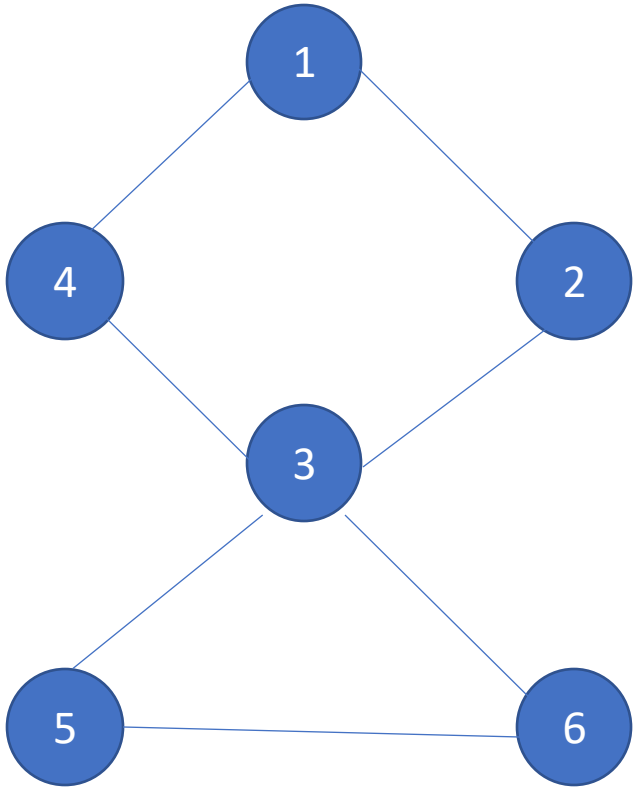
Articulation Points and Biconnected Components

- Let G = (V, E) be a connected undirected graph. Consider the following definitions:
- **Articulation Point (or Cut Vertex):** An articulation point in a connected graph is a vertex (together with the removal of any incident edges) that, if deleted, would break the graph into two or more pieces.
- **Bridge**: Is an edge whose removal results in a disconnected graph.
- **Biconnected:** A graph is biconnected if it contains no articulation points. In a biconnected graph, two distinct paths connect each pair of vertices. A graph that is not biconnected divides into biconnected components.
- Biconnected graphs and articulation points are of great interest in the design of network algorithms, because these are the “critical" points, whose failure will result in the network becoming disconnected.
- This is illustrated in the following figure:



Articulation Points and Bridges

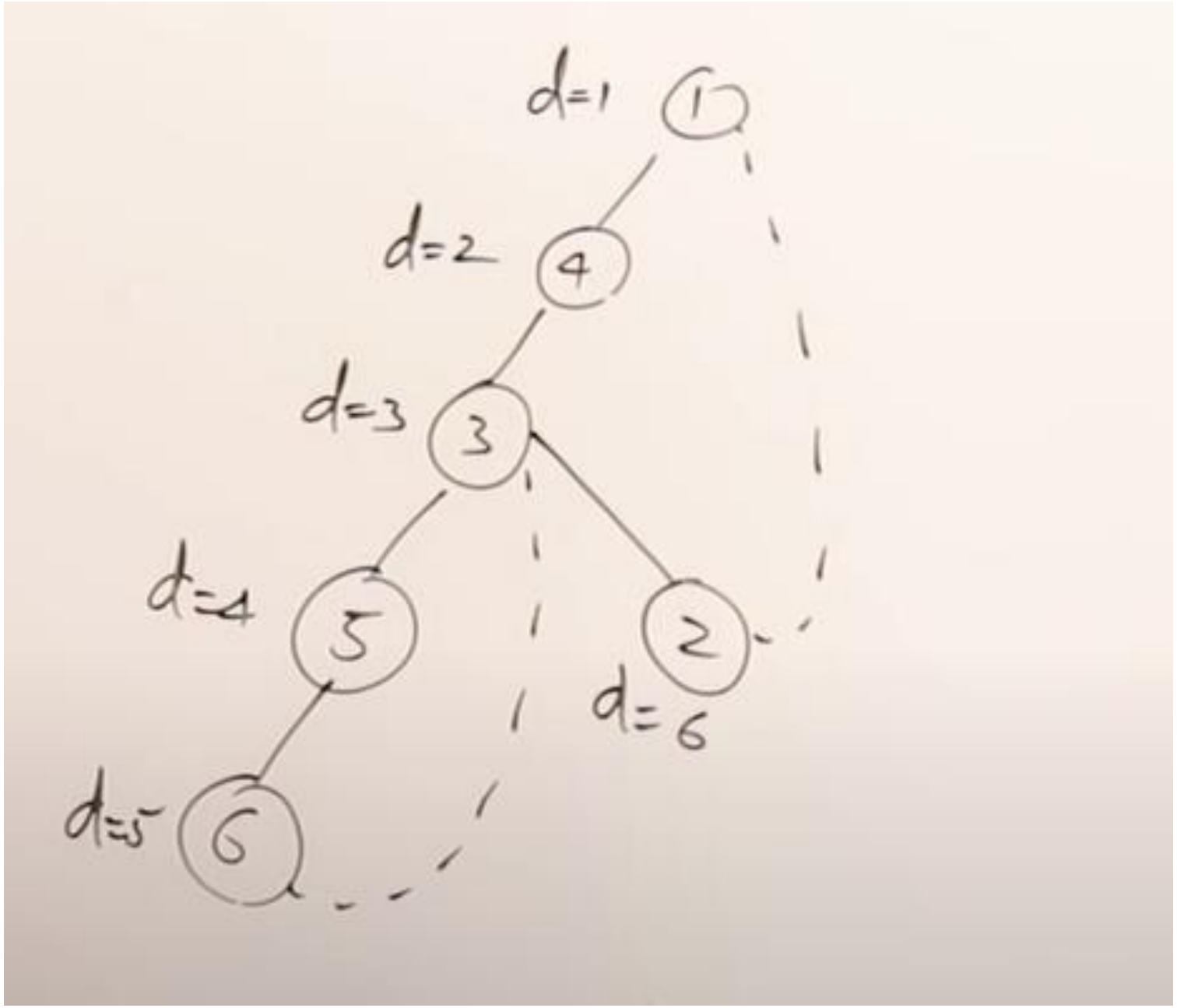
Find Articulation Point



Articulation Points and Biconnected Components

- Let G = (V, E) be a connected undirected graph. Consider the following definitions:
- **Articulation Point (or Cut Vertex):** An articulation point in a connected graph is a vertex (together with the removal of any incident edges) that, if deleted, would break the graph into two or more pieces.
- **Bridge**: Is an edge whose removal results in a disconnected graph.
- **Biconnected:** A graph is biconnected if it contains no articulation points. In a biconnected graph, two distinct paths connect each pair of vertices. A graph that is not biconnected divides into biconnected components.
- This is illustrated in the following figure:

Compute discovery time



Make table for identifying articulation point

node	1	2	3	4	5	6
Discovery time(d)	1	6	3	2	4	5
L	1	1	1	1	3	3

Compute articulation point

- (u, v) = (parent, child)
- If L[v] >= d[u] then articulation point exist at u except for root, otherwise not articulation point.
- Different condition will be applied for root vertex.
- If root is having multiple child then root is an articulation point.

Compute articulation point

- **Initialization:**
  - Perform a DFS traversal of the graph starting from any vertex (usually the root).
  - Initialize variables to keep track of the discovery time and low-link values for each vertex.
  - Use a stack to keep track of the vertices in the current DFS traversal.
- **DFS Traversal:**
  - During the DFS traversal, assign a unique discovery time to each vertex as you visit it for the first time.
  - Keep track of the lowest discovery time (low-link value) reachable from the current vertex, considering both forward and backward edges.

Compute articulation point

- **Identifying Articulation Points:**
  - An articulation point is identified based on the following conditions:
    - If the current vertex is the root of the DFS tree and has more than one child, it is an articulation point.
    - For other vertices, if there exists a child **`v`** such that **`low[v] >= discovery_time[current]`**, then the current vertex is an articulation point.
- **Backtracking:**
  - During the backtracking phase of DFS, update the low-link values of the current vertex based on the low-link values of its children.

Thank You