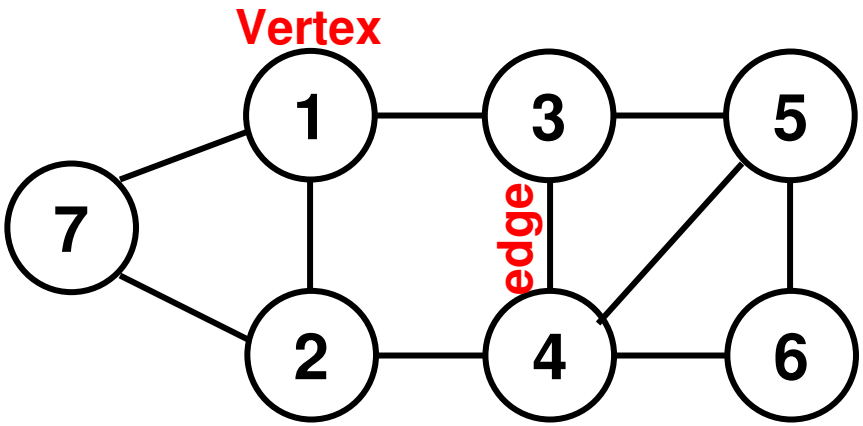
Module 9:
Graphs

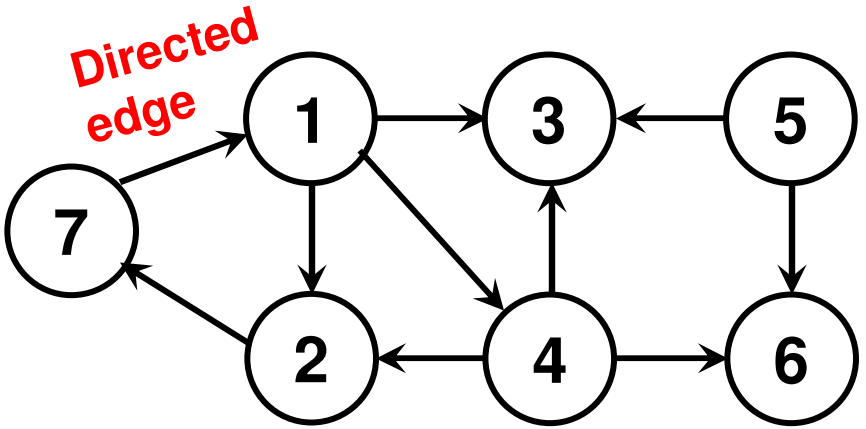
Dr. Natarajan Meghanathan
Professor of Computer Science
Jackson State University
Jackson, MS 39217
E-mail: natarajan.meghanathan@jsums.edu

Graph

- Graph is a data structure that is a collection of nodes (vertices) and links (edges).
- A graph could be an undirected or directed.
- A graph could be used to model complex real-world networks
  - E.g., a network of cities, a network of airports, social networks, communication networks (like Internet), etc.



Undirected Graph



Directed Graph

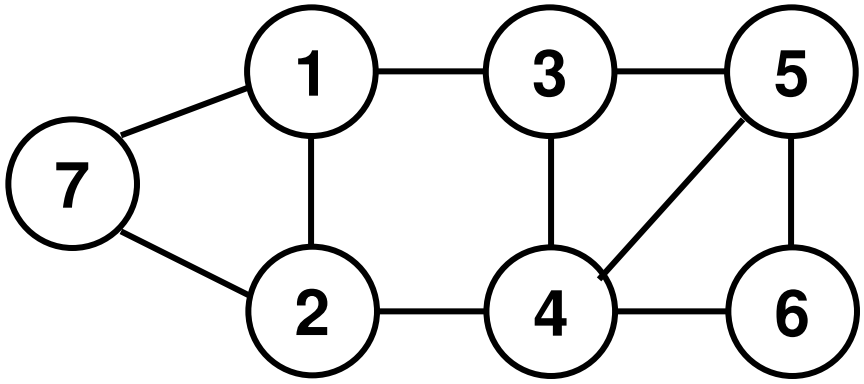
Adjacency List and Adjacency Matrix

- Information about the nodes and edges could be stored in the form of an adjacency list or an adjacency matrix
- **Undirected Graphs**
- **Adjacency List**: is an array of lists (like Linked List) that store the neighbors (edges incident) for each vertex
- **Adjacency Matrix**:

  A[i, j] = 1 if there is an edge between i and j

  0; otherwise

Note that Adjacency matrix for undirected graphs is a symmetric matrix i.e., A[i, j] = A[j, i]



Undirected Graph

Adjacency List

1: 2, 3, 7
2: 1, 4, 7
3: 1, 4, 5
4: 2, 3, 5, 6
5: 3, 4, 6
6: 4, 5
7: 1, 2

Adjacency Matrix

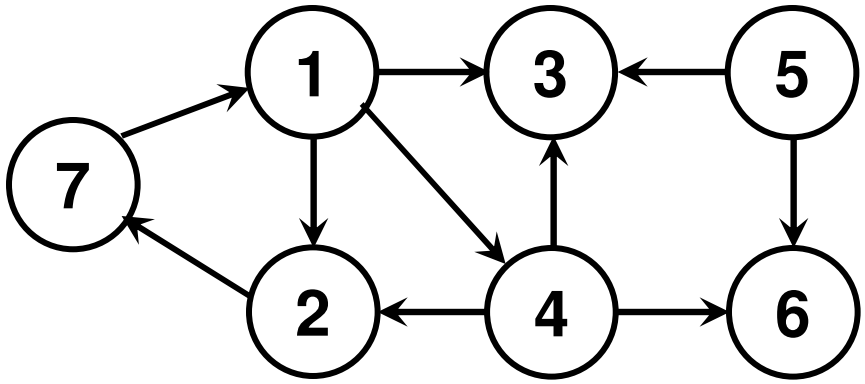
	1	2	3	4	5	6	7
1	0	1	1	0	0	0	1
2	1	0	0	1	0	0	1
3	1	0	0	1	1	0	0
4	0	1	1	0	1	1	0
5	0	0	1	1	0	1	0
6	0	0	0	1	1	0	0
7	1	1	0	0	0	0	0

Adjacency List and Adjacency Matrix

- Information about the nodes and edges could be stored in the form of an adjacency list or an adjacency matrix
- **Directed Graphs**
- **Adjacency List**: is an array of lists (like Linked List) that store the outgoing edges (outgoing neighbors) for each vertex
- **Adjacency Matrix**:

  A[i, j] = 1 if there is an edge from i to j

  0; otherwise



Directed Graph

Adjacency List

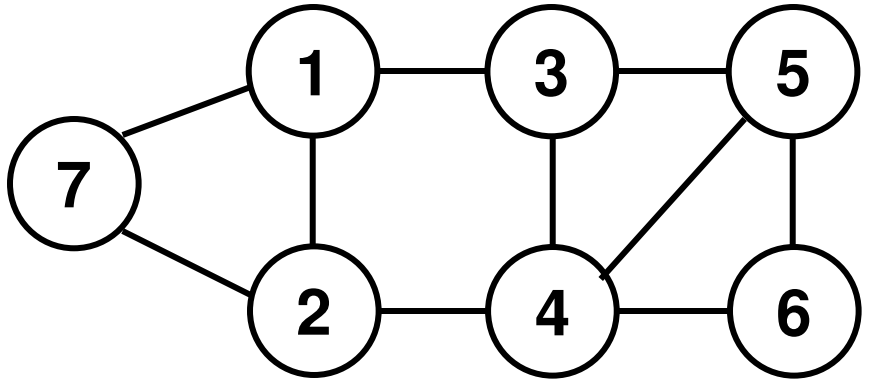
1: 2, 3, 4
2: 7
3:
4: 2, 3, 6
5: 3, 6
6:
7: 1

Adjacency Matrix

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

Vertex Degree (Undirected Graphs)

- The degree for a vertex is the number of neighbors incident on the vertex
- The average degree for a graph is the average of the vertex degrees.
- The probability for finding a vertex with a certain degree k is the ratio of the number of vertices having that degree and the total number of vertices in the graph.



Adjacency List	Degree
1: 2, 3, 7	3
2: 1, 4, 7	3
3: 1, 4, 5	3
4: 2, 3, 5, 6	4
5: 3, 4, 6	3
6: 4, 5	2
7: 1, 2	2

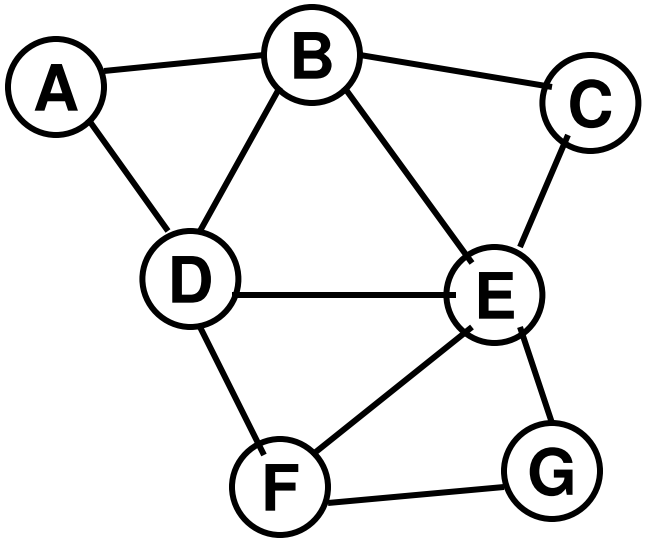
Degree K	Count # Vertices	Prob(Degree) P(K)
2	2	2/7 = 0.286
3	4	4/7 = 0.571
4	1	1/7 = 0.143

Average Degree = $\sum K*P(K)$
= (0.286 * 2 + 0.571 * 3 + 0.143 * 4)
= 2.86

Edges in the Graph = Sum of Degrees / 2
= (3 + 3 + 3 + 4 + 3 + 2 + 2) / 2 = 10

Average Degree = Sum of Degrees / # Vertices = (3 + 3 + 3 + 4 + 3 + 2 + 2) / 7 = 2.86

Example 2: Vertex Degree



Adjacency List	Degree
A: B, D	2
B: A, C, D, E	4
C: B, E	2
D: A, B, E, F	4
E: B, C, D, F, G	5
F: D, E, G	3
G: E, F	2

Degree K	Count # Vertices	Prob(Degree) P(K)
2	3	3/7 = 0.429
3	1	1/7 = 0.143
4	2	2/7 = 0.286
5	1	1/7 = 0.143

Average Degree = $\sum K*P(K)$
= (0.429 * 2 + 0.143 * 3 + 0.286 * 4 + 0.143 * 5)
= 3.14

Average Degree = Sum of Degrees / # Vertices
= (2 + 4 + 2 + 4 + 5 + 3 + 2) / 7 = 3.14

Edges in the Graph = Sum of Degrees / 2
= (2 + 4 + 2 + 4 + 5 + 3 + 2) / 2 = 11

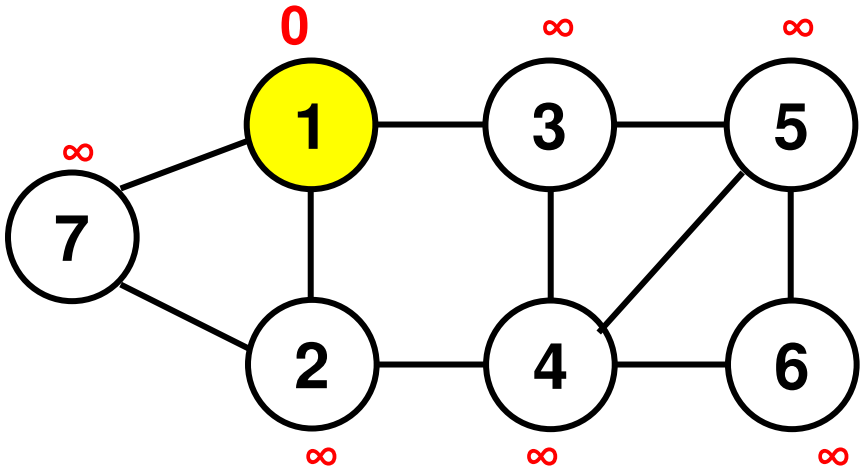
Breadth First Search (BFS)

```
BFS(G, s)
    Queue queue
    queue.enqueue(s)        // 's' is the starting vertex
    Level[s] = 0
    Level[v] = ∞;  for all vertices v other than 's'
                    // The level # is also the estimated number of edges
                    // on the minimum edge path (shortest path) from 's'
    Visited[v] = false; for all vertices v other than 's'
    while (!queue.isEmpty()) do
      u = queue.dequeue();
      for every vertex v that is a neighbor of u
          if (Visited[v] = false) then
                    Level[v] = Level[u] + 1
                    Visited[v] = true
                    Queue.enqueue(v)
                    Edge u-v is a tree edge
          end if
          else
                    Edge u-v is a cross edge
       end for
    end while
End BFS
```

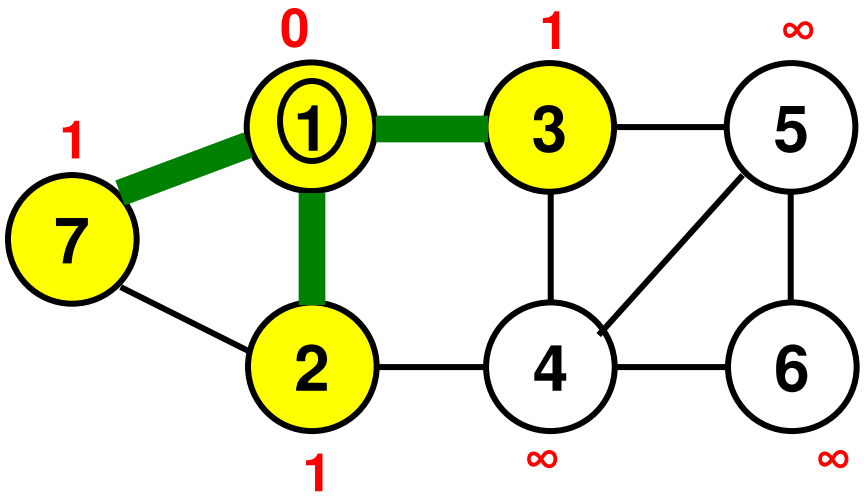
Time Complexity:

If there are 'V' vertices and 'E' edges, we traverse each edge exactly once as well as enqueue and dequeue each vertex exactly once. Hence, the time complexity of BFS is Θ(V+E) when implemented using an Adjacency list and $\Theta(V^2)$ when implemented using an Adjacency matrix.

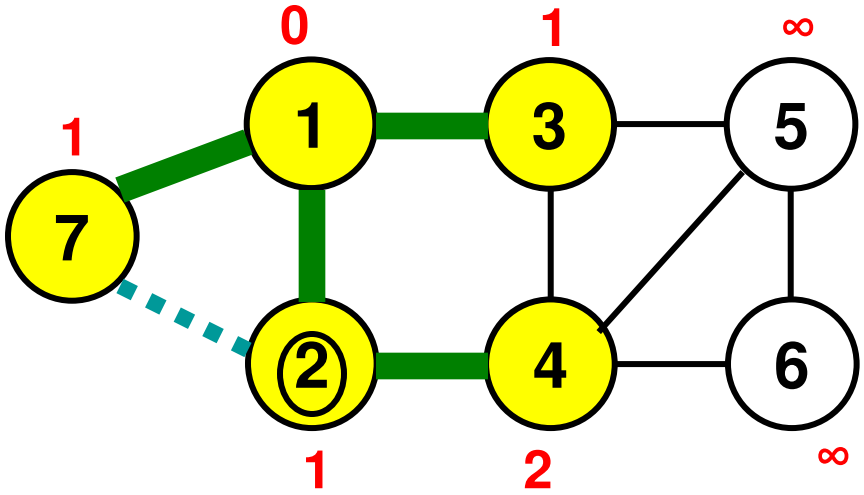
BFS: Example 1



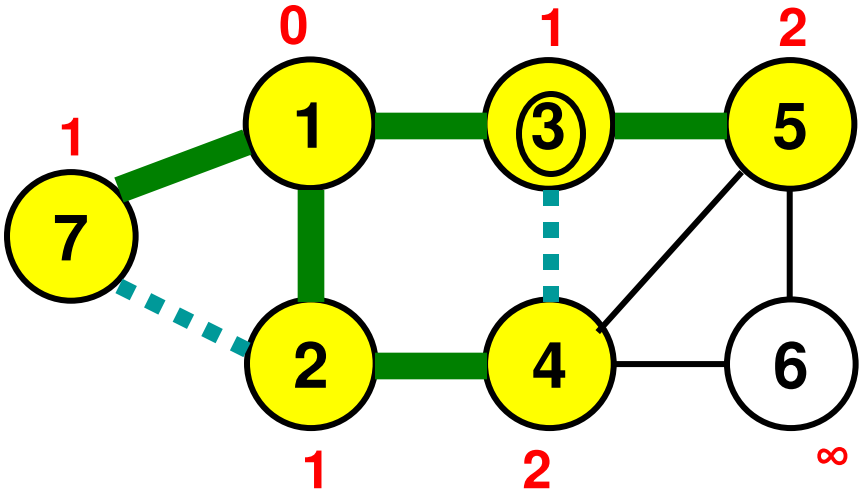
Initialization: Queue = {1}



Iteration 1: Queue = {2, 3, 7}

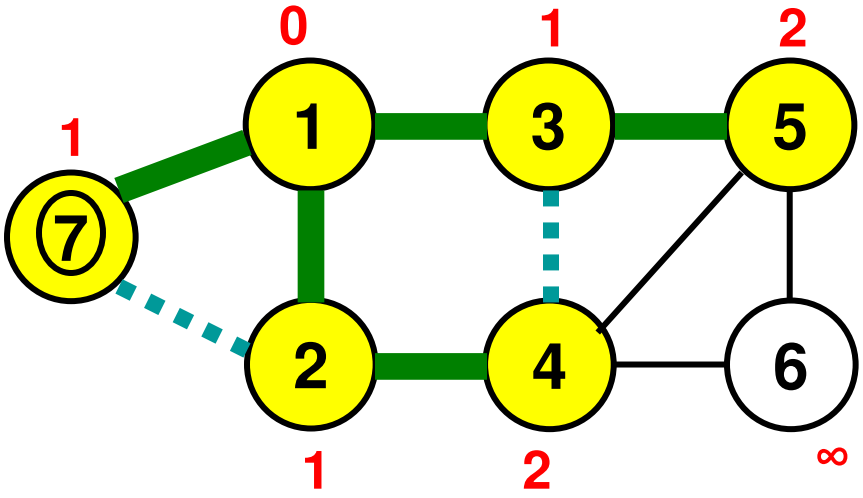


Iteration 2: Queue = {3, 7, 4}

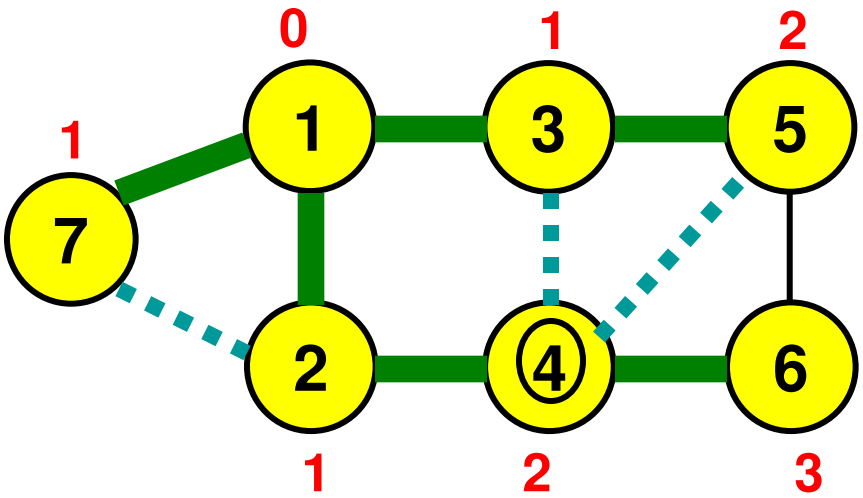


Iteration 3: Queue = {7, 4, 5}

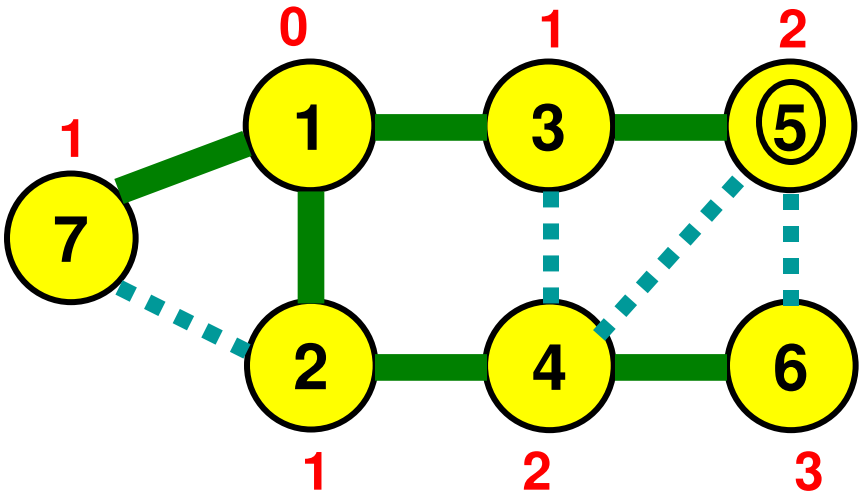
BFS: Example 1



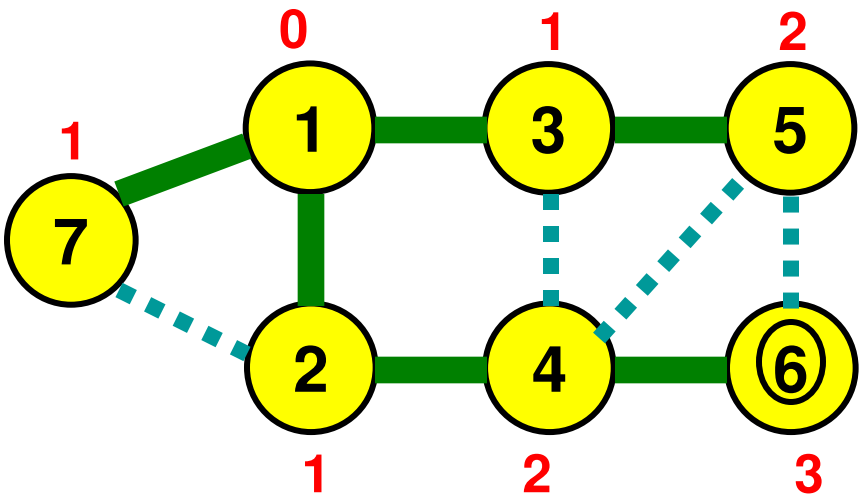
Iteration 4: Queue = {4, 5}



Iteration 5: Queue = {5, 6}



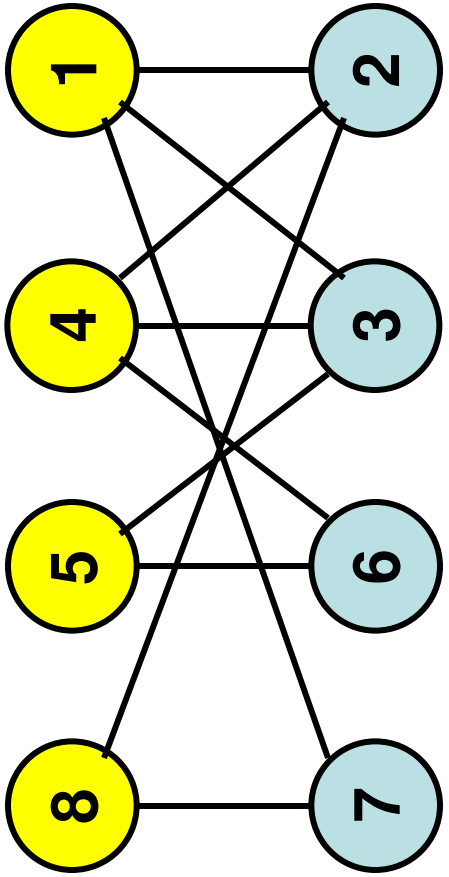
Iteration 6: Queue = {6}



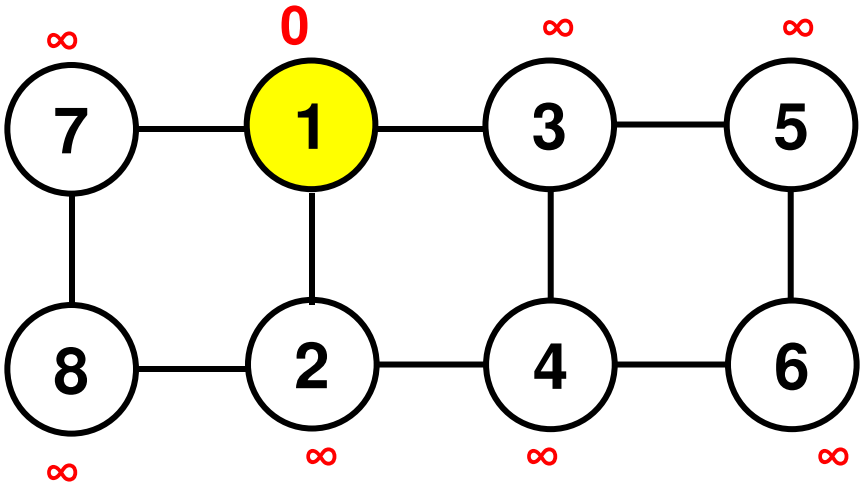
Iteration 7: Queue = {}

Bipartite Graph

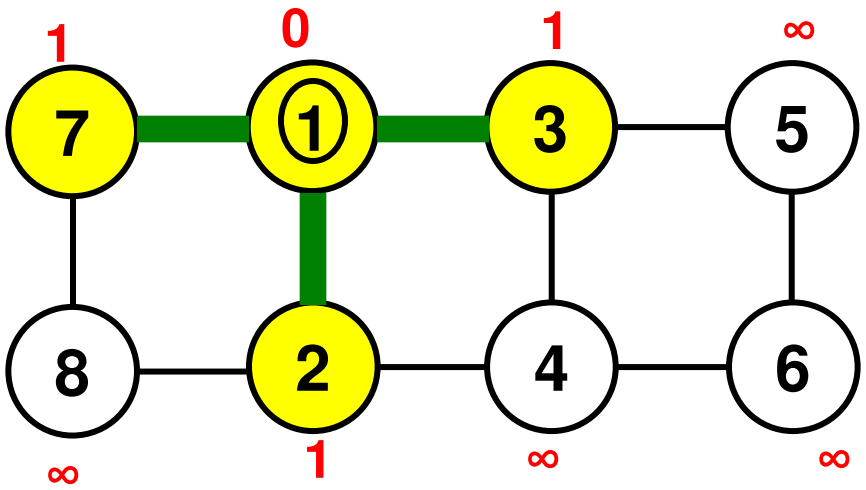
- A graph in which the vertices could be partitioned to two disjoint sets such that all the edges in the graph are between the vertices in the two sets and there are no edges between vertices in the same set.
- An undirected graph is bipartite if there are no cross edges between vertices at the same level while doing BFS.
  - Note there could be cross edges between vertices at different levels. This is fine.
- A bipartite graph is also said to be “2-colorable”.
  - There are only two colors to color the vertices
  - For each edge u – v, the end vertices should be of different color.



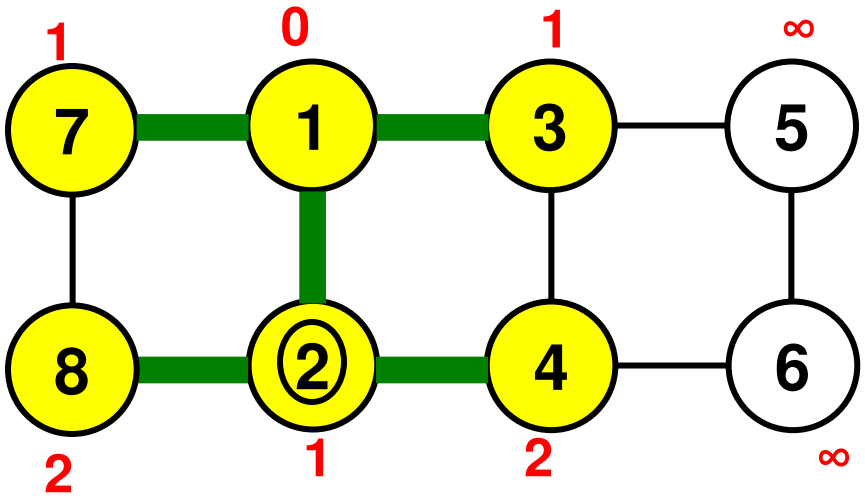
BFS: Example 2



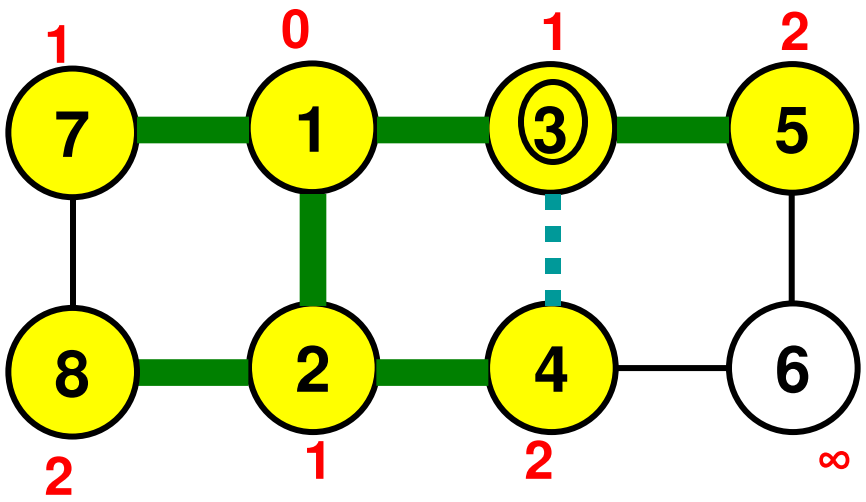
Initialization: Queue = {1}



Iteration 1: Queue = {2, 3, 7}

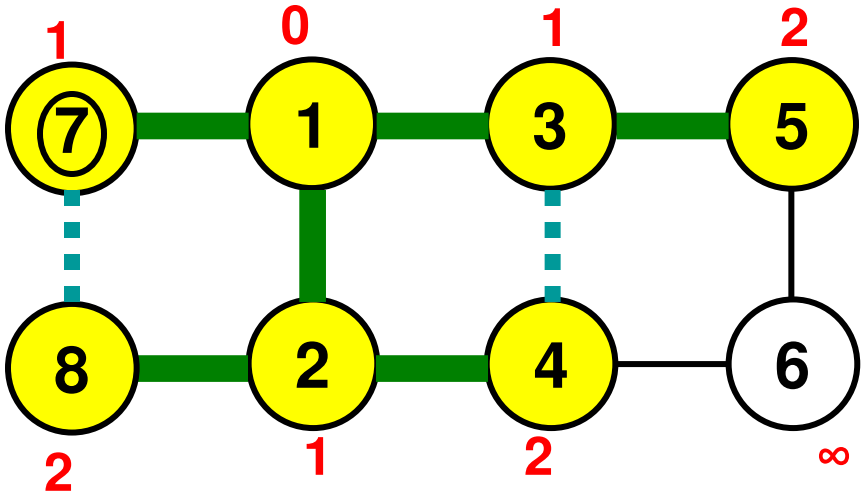


Iteration 2: Queue = {3, 7, 4, 8}

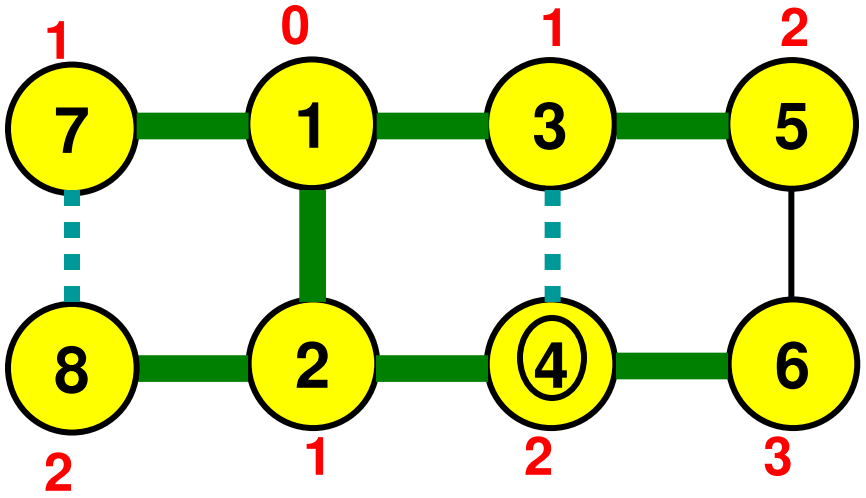


Iteration 3: Queue = {7, 4, 8, 5}

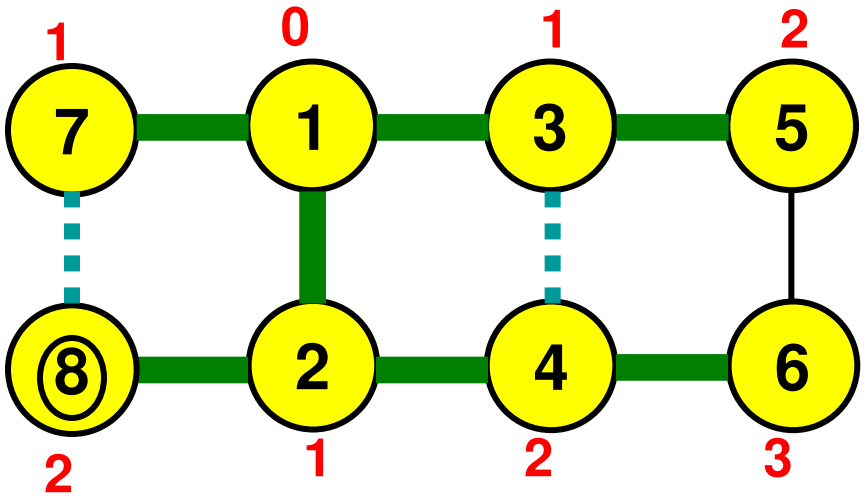
BFS: Example 2



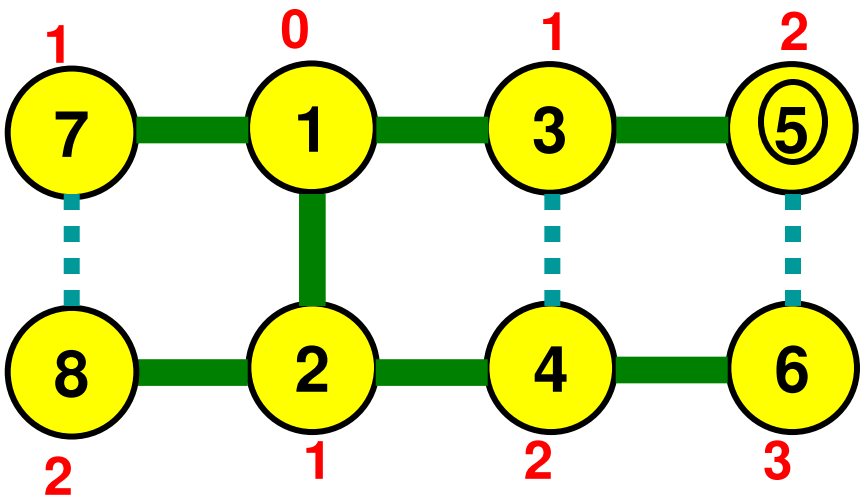
Iteration 4: Queue = {4, 8, 5}



Iteration 5: Queue = {8, 5, 6}

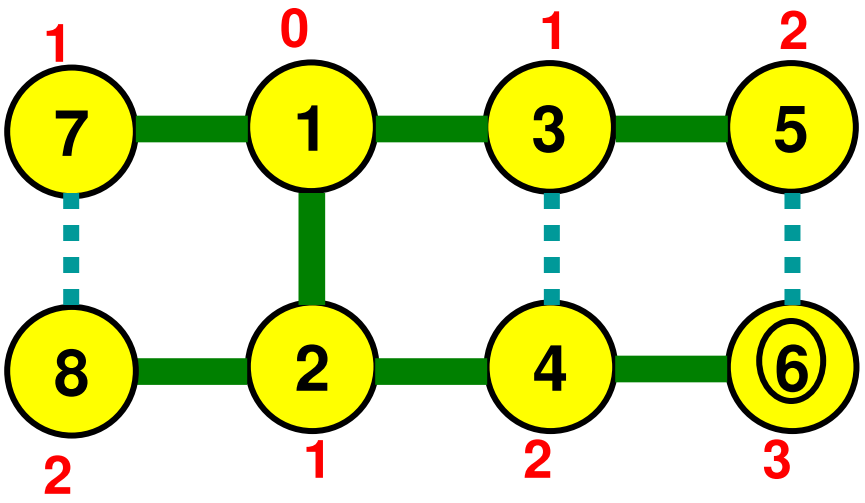


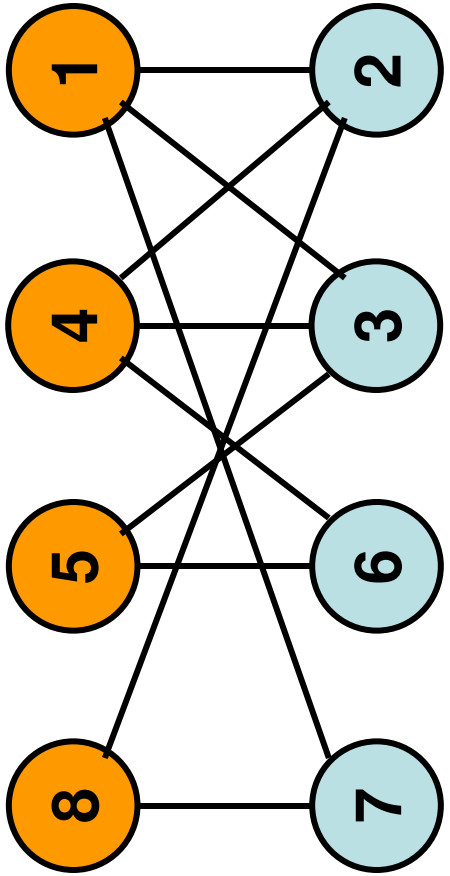
Iteration 6: Queue = {5, 6}



Iteration 7: Queue = {6}

BFS: Example 2

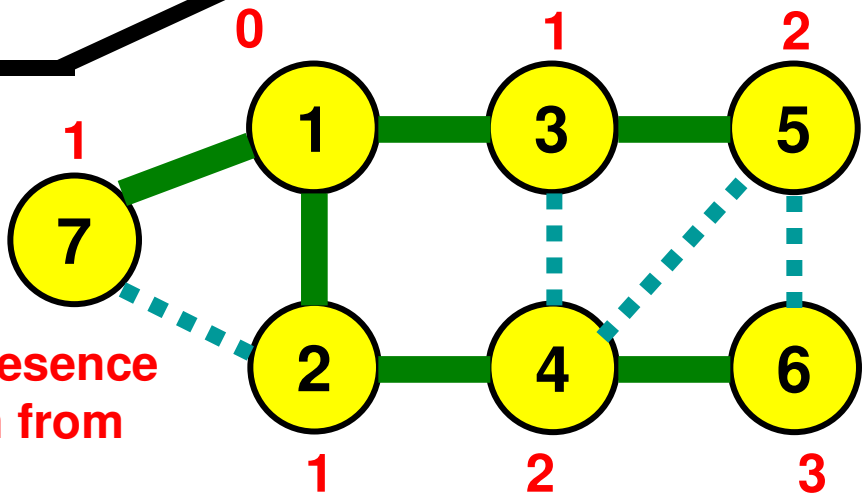




Iteration 8: Queue = {}

The vertices in the odd level are in one color and the vertices in the even level are in the other color

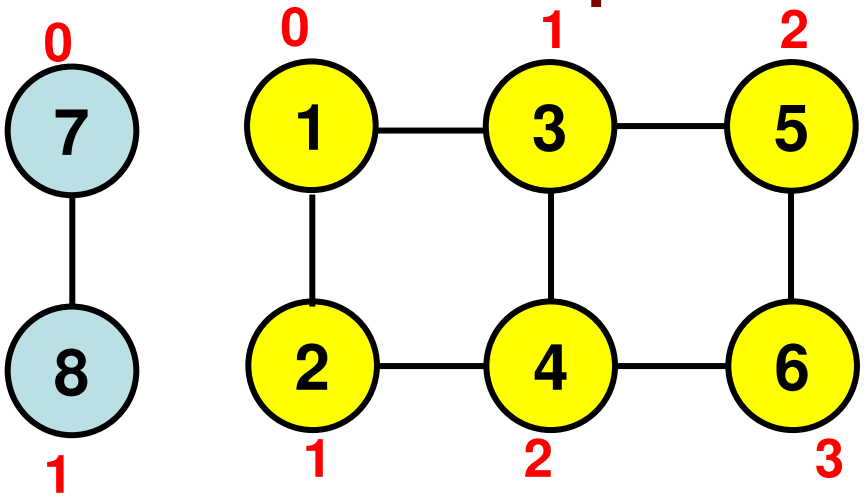
BFS: Example 1



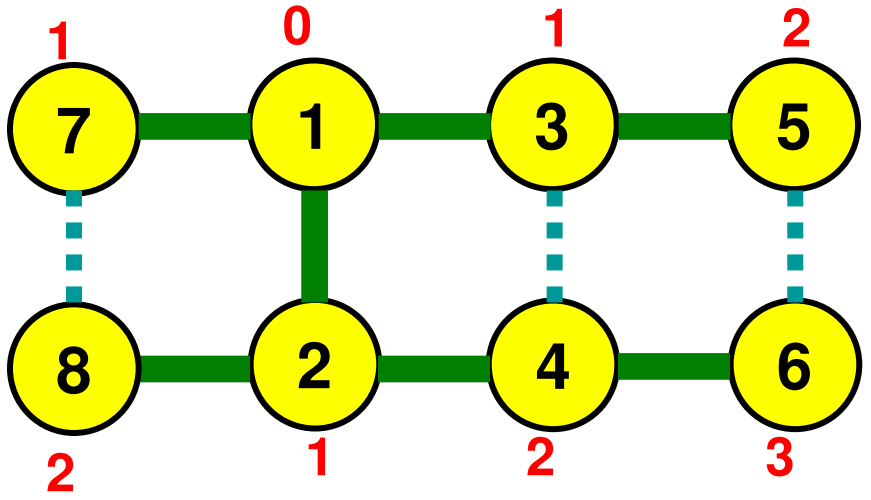
The graph is not bipartite as there are edges 2 – 7 and 2 – 4 that are cross edges between vertices at the same level. The presence of even one such edge rules out the graph from being bipartite.

Connected Undirected Graph

- An undirected graph is said to be “connected” if we could start BFS from any arbitrary vertex in the graph and be able to visit the rest of the vertices in the graph.
  - All the vertices in a connected undirected graph are said to be in “one component”.
- If even on vertex is not reachable from the starting vertex of BFS, the graph is considered to be “not connected” and will be composed of two or more components.
  - A component is the largest subset of the vertices in the graph such that all the vertices within the subset (component) are reachable from each other.



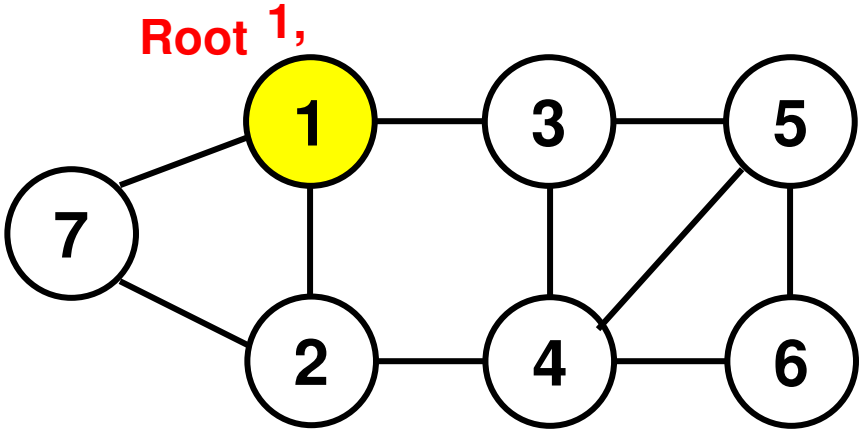
The graph above comprises of “two” components; the graph below has only “one” component



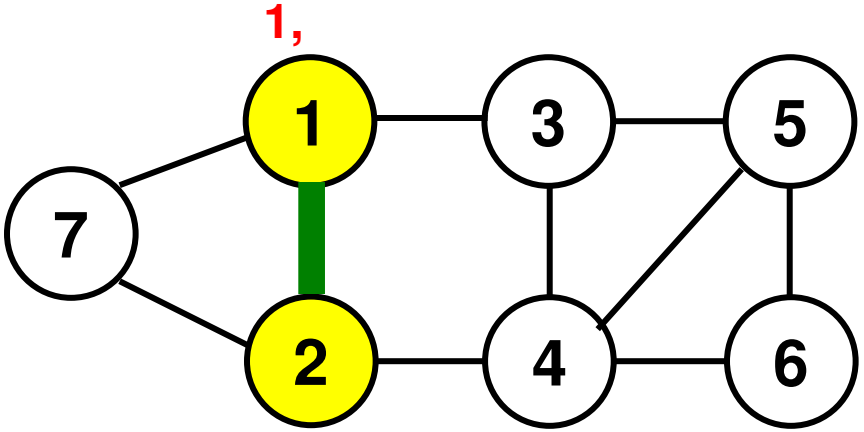
Depth First Search (DFS)

- With DFS, we start with a vertex (root) and visit one of its neighbors and go further deep until we reach a dead end, and then back track.
- We use a Stack to keep track of the vertices that are visited for the first time (pushed to the stack) and back track after we reach a dead end (popped from the stack).
- In the case of an undirected graph, the edge that leads to a vertex to be pushed to the stack is called a **tree edge**, and the edge that leads to a vertex that has been already pushed to the stack is called a **back edge**.
- If a back edge is encountered during a DFS, the graph is said to have a cycle.
- Time complexity of DFS is the same as that of BFS. Each vertex and edge is visited exactly once.
  - Adjacency List: $\Theta(V+E)$ Adjacency Matrix: $\Theta(V^2)$

DFS: Example 1

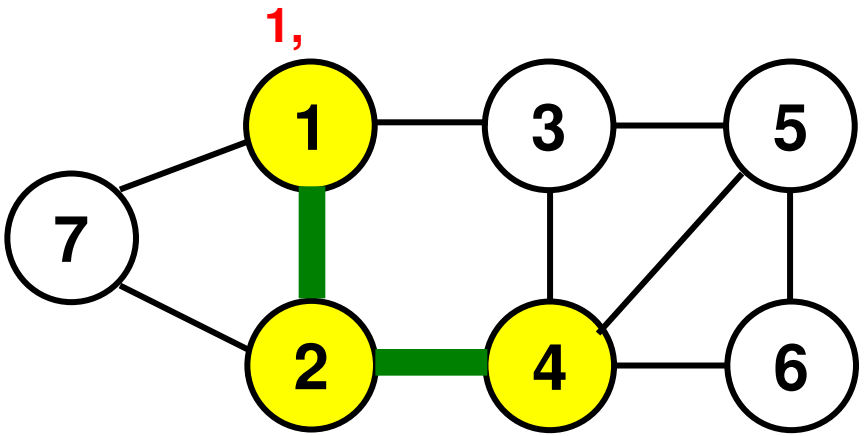


Initialization: Stack = {1}

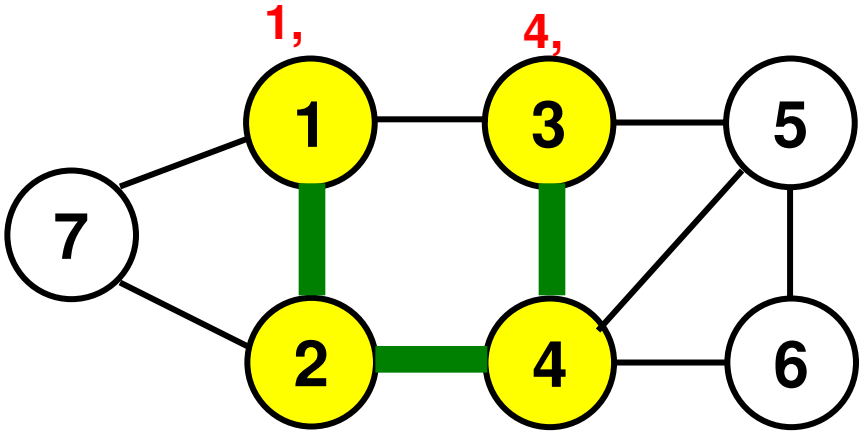


2,

Stack = {2, 1}

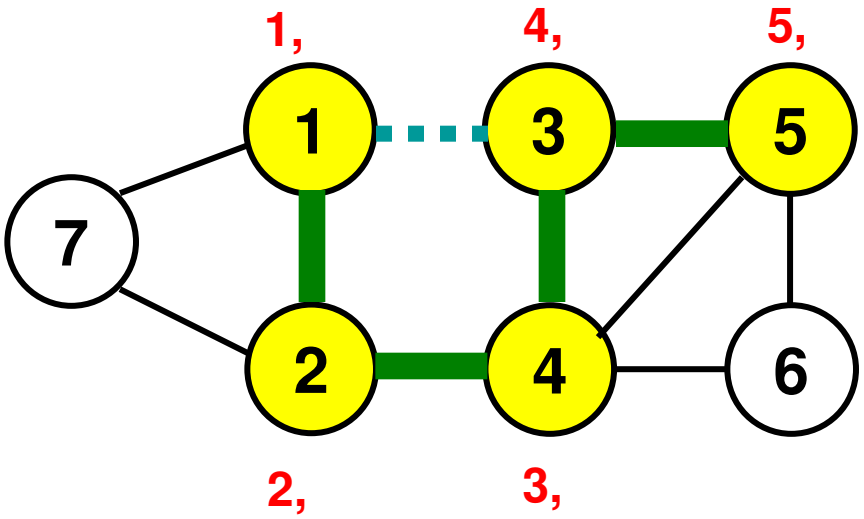


Stack = {4, 2, 1}

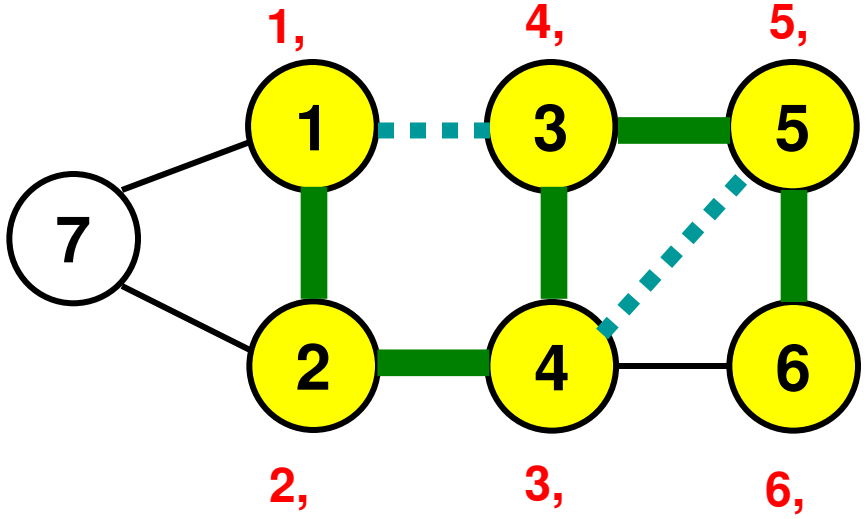


Stack = {3, 4, 2, 1}

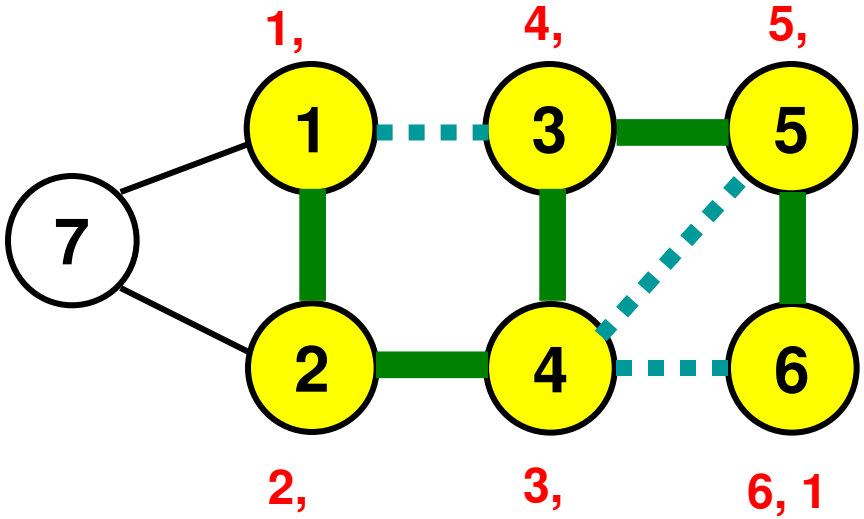
DFS: Example 1



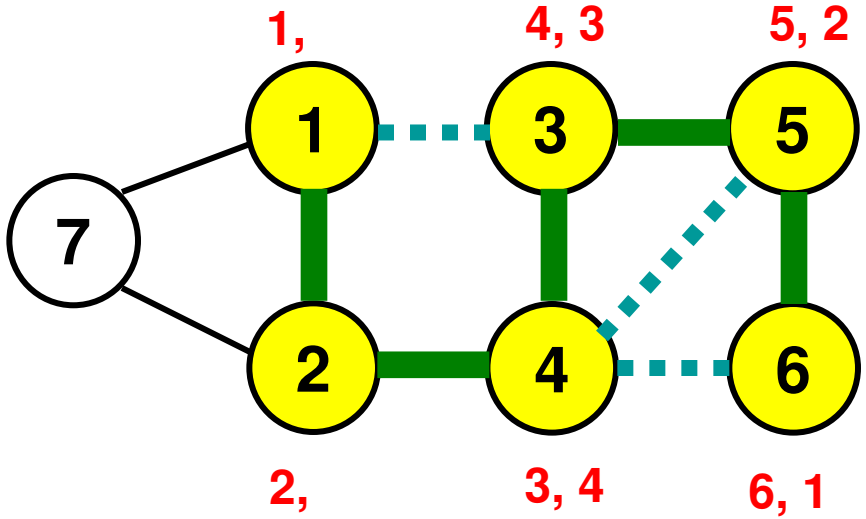
Stack = {5, 3, 4, 2, 1}



Stack = {6, 5, 3, 4, 2, 1}

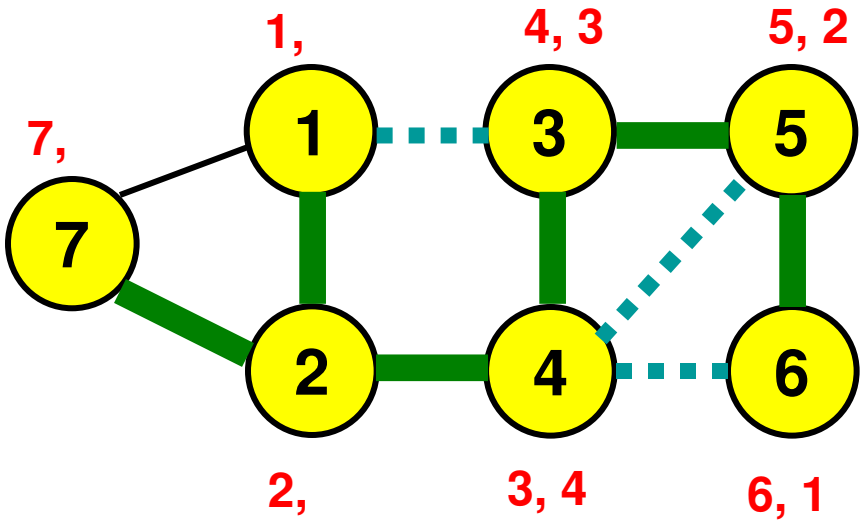


Stack = {5, 3, 4, 2, 1}

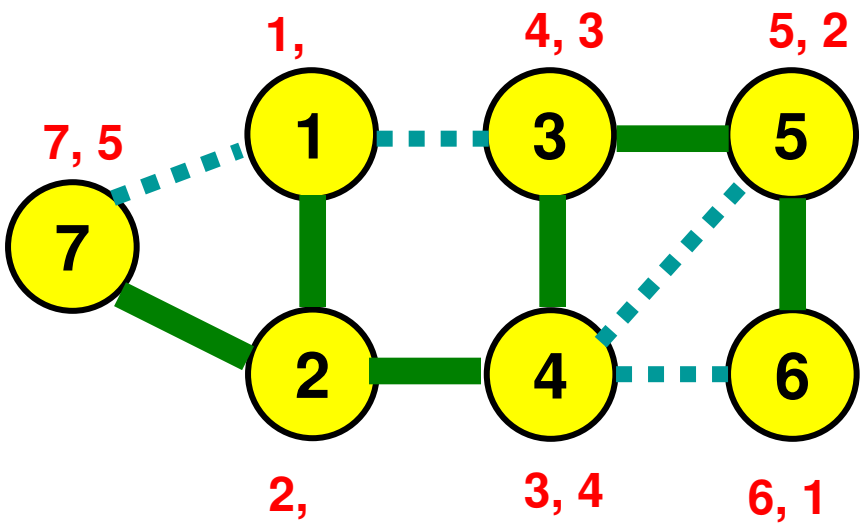


Stack = {2, 1}

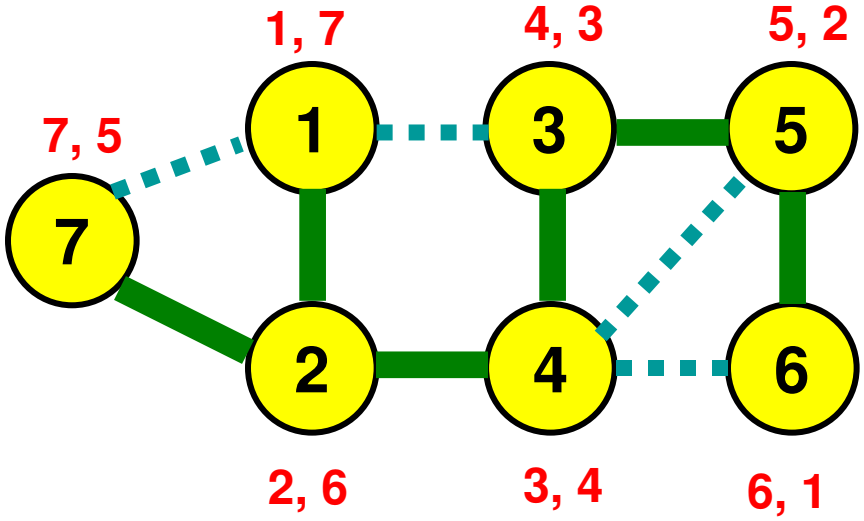
DFS: Example 1



Stack = {7, 2, 1}

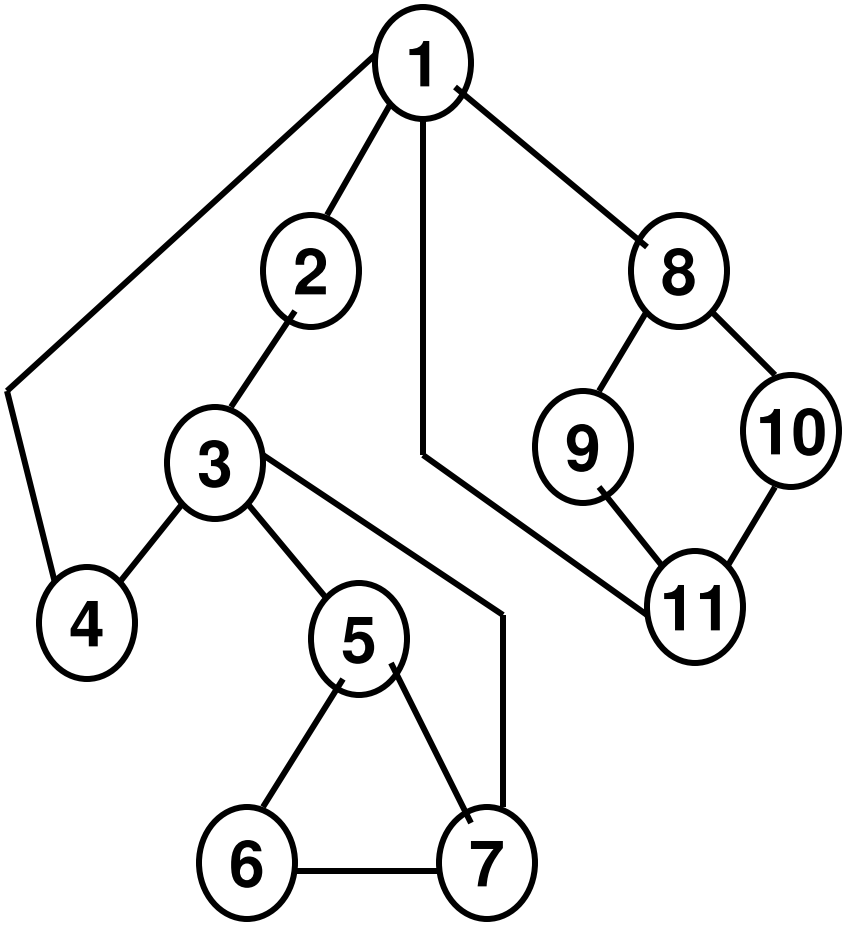


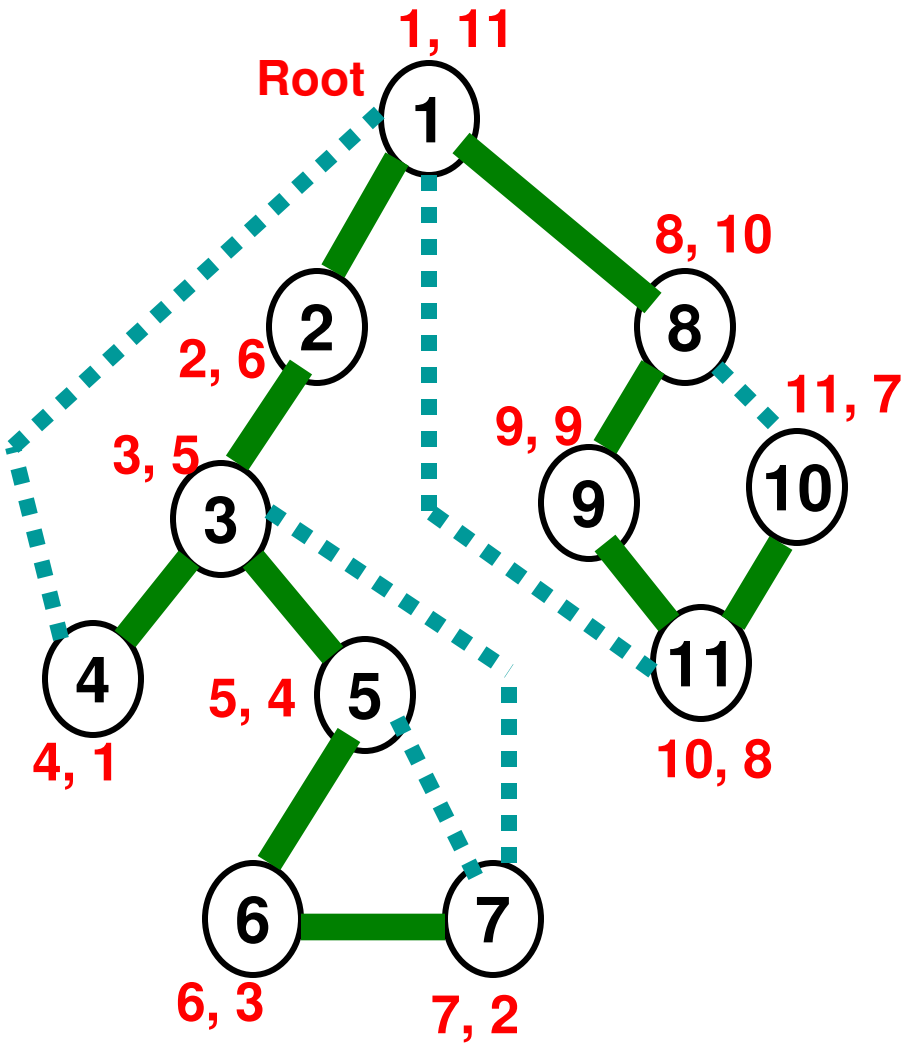
Stack = {2, 1}



Stack = {}

DFS: Example 2





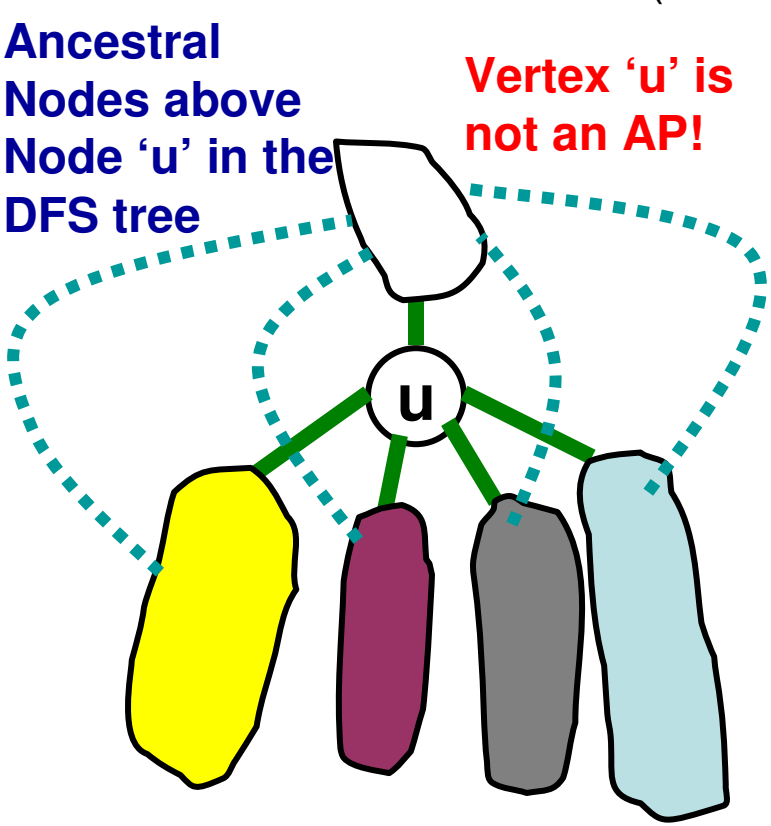
Articulation Point (AP)

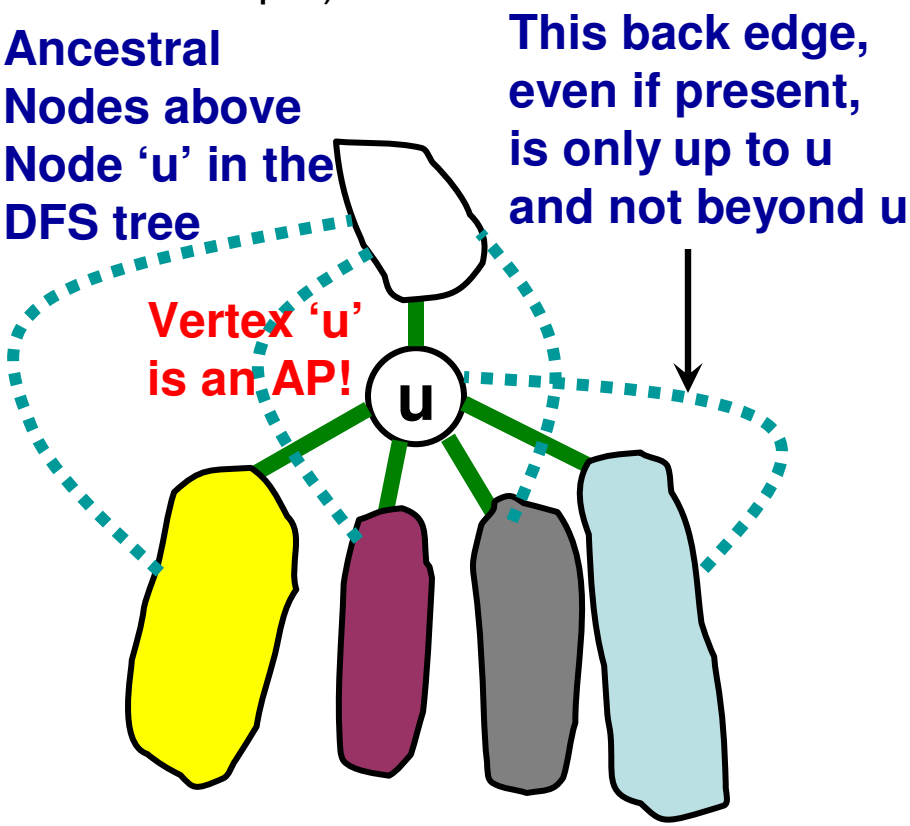
- A vertex is called an articulation point (AP) if its removal would disconnect a graph into two or more components.
- We could use the results of DFS to identify the APs of a graph.

- Criteria for deciding whether a node is an AP or not
- The root of a DFS tree is an articulation point if it has more than one child connected through a tree edge. (In the above DFS tree, the root node ‘a’ is an articulation point)
- The leaf nodes of a DFS tree are not articulation points.
- Any other internal vertex *v* in the DFS tree, if it has one or more sub trees rooted at a child (at least one child node) of *v* that does NOT have an edge which climbs ’higher ’ than *v* (through a back edge), then *v* is an articulation point.

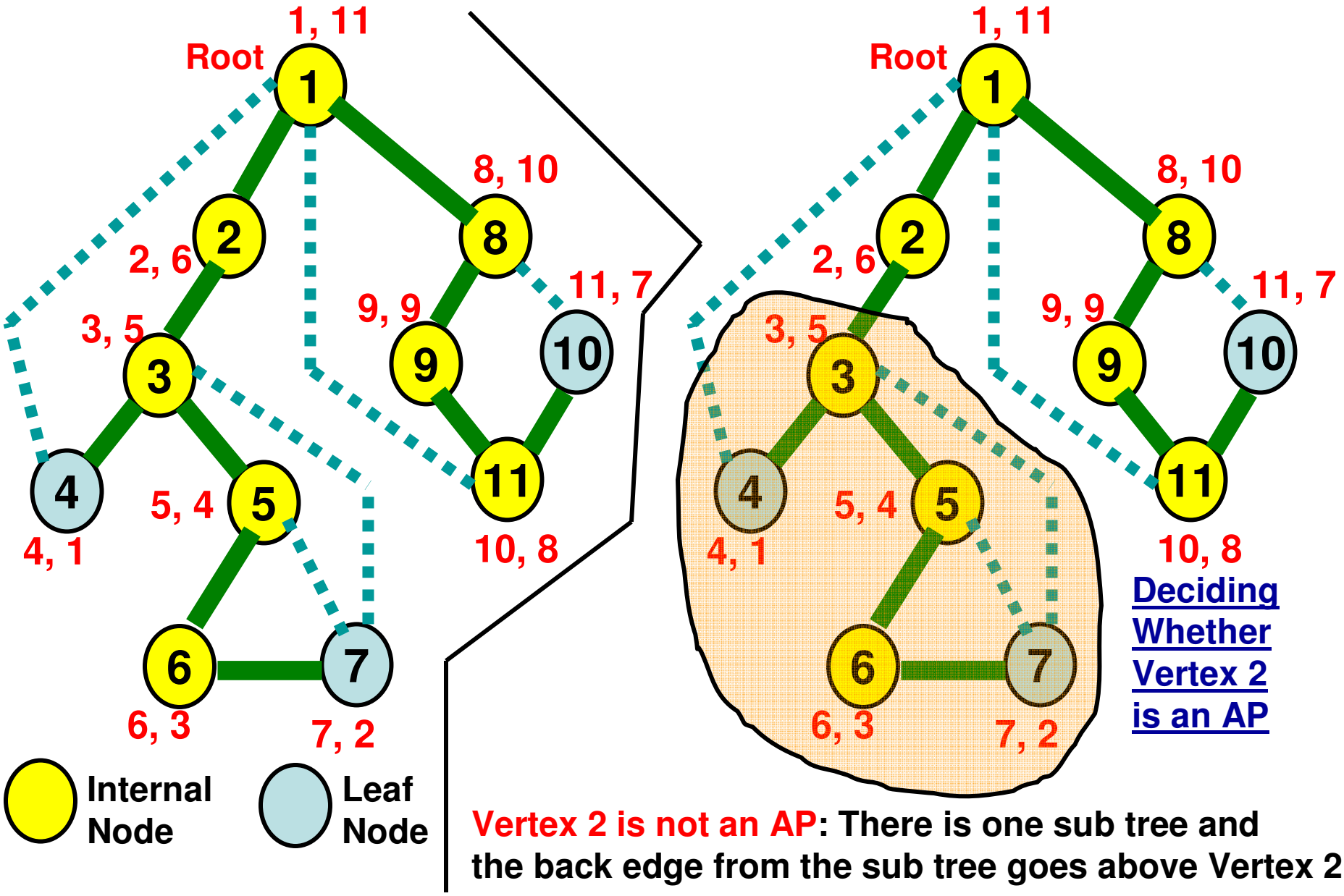
Deciding whether an Internal Node is an Articulation Point or not

- An internal node is NOT an articulation point if there exist one or more back edges from each of the sub trees of the node to one or more ancestral nodes in the DFS tree (like the left side example)
  - In other words, an internal node is an articulation point if there exists at least one sub tree from which there is no back edge that goes above the node in the DFS tree (like the right side example).

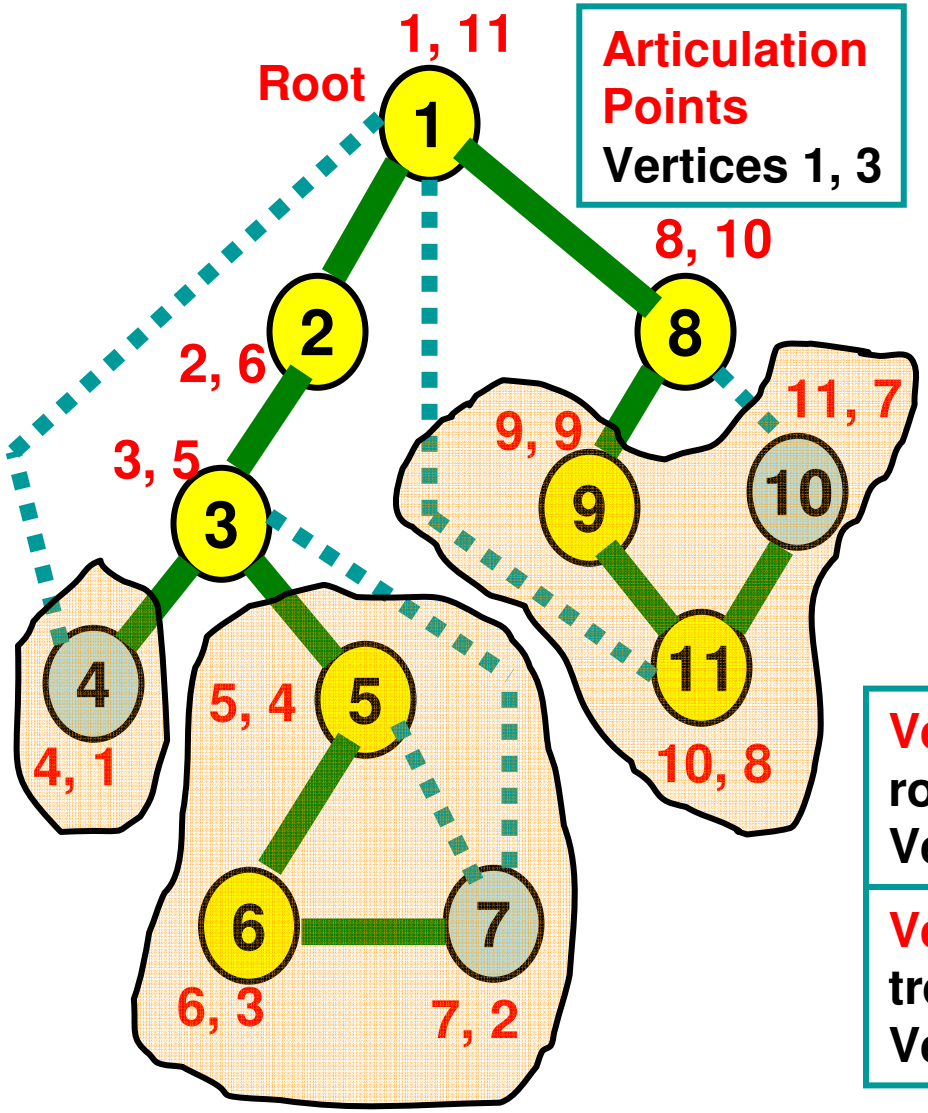




Articulation Points: Example 1



Vertex 2 is not an AP: There is one sub tree and the back edge from the sub tree goes above Vertex 2.



Articulation Points
Vertices 1, 3

Vertex 5 is not an AP: There is one Sub tree rooted at vertex 6 and it has a back edge that goes above vertex 5.

Vertex 6 is not an AP: There is one Sub tree rooted at vertex 7 and it has two back edges that go above vertex 6.

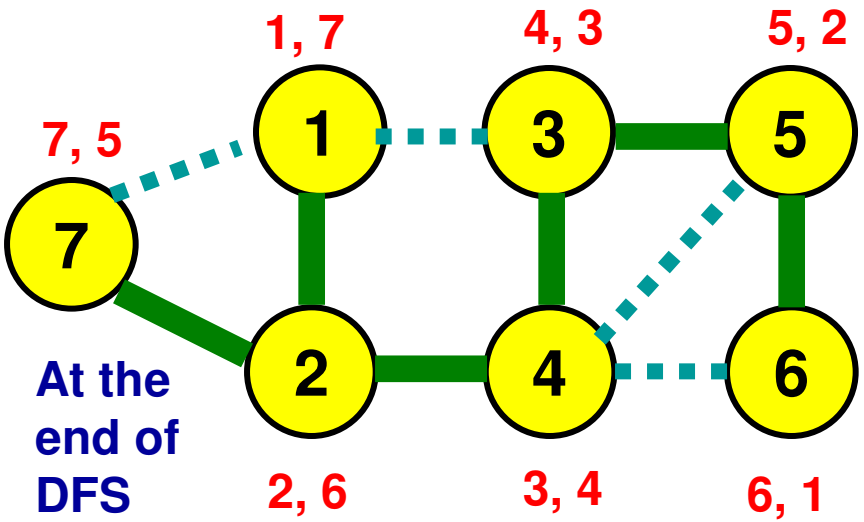
Vertex 8 is not an AP: There is one Sub tree rooted at vertex 9 and it has a back edge that goes above vertex 8.

Vertex 9 is not an AP: It has one sub tree rooted at vertex 11 that goes above Vertex 9.

Vertex 11 is not an AP: It has one sub tree rooted at vertex 10 that goes above Vertex 9.

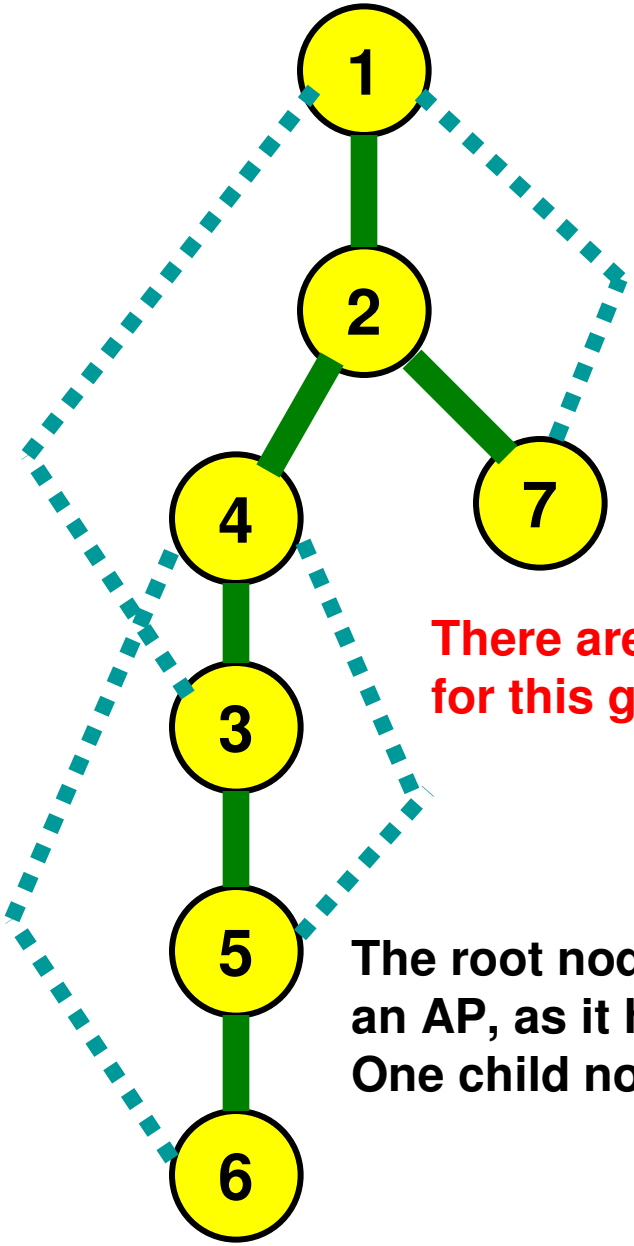
Vertex 3 is an AP!!: There are two sub trees for Vertex 3. The sub tree rooted at vertex 4 has a back edge that goes above vertex 3. However, the sub tree rooted at vertex '5' has a back edge that goes only up to vertex 3. Hence, vertex 3 is an AP. If it is removed from the graph, the sub tree rooted at vertex 5 will get disconnected from the rest of the graph.

Articulation Points: Example 2



Vertex 2 has two sub trees. But both the sub trees have a back edge that go above vertex 2.

Vertex 4 has a sub tree. Though two of the three back edges do not go above Vertex 4, there is a back edge from Vertex 3 to 1 that goes above vertex 4.

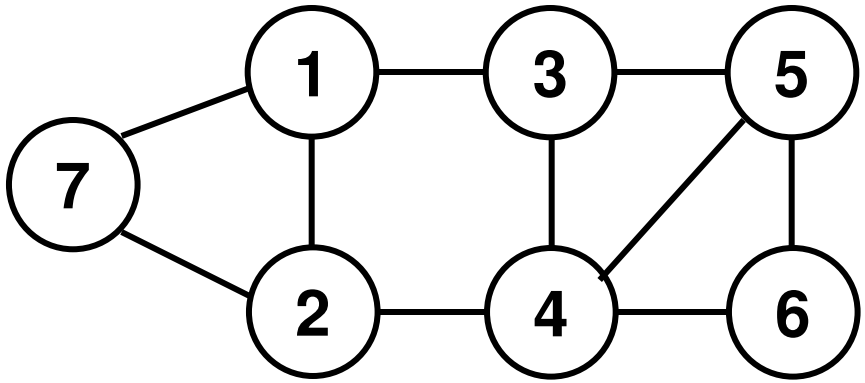


There are no APs for this graph

The root node is not an AP, as it has only One child node.

Biconnected Graph

- An undirected graph is biconnected if there are **at least two vertex disjoint paths** between any two vertices.
- An undirected graph is biconnected if the following two conditions are met:
  - The graph is connected
  - There are no articulation points in the graph
- Both of the above could be decided by running DFS starting from any arbitrary vertex.
  - A graph is connected if we are able to visit all the vertices in the graph as part of DFS initiated from an arbitrary vertex.

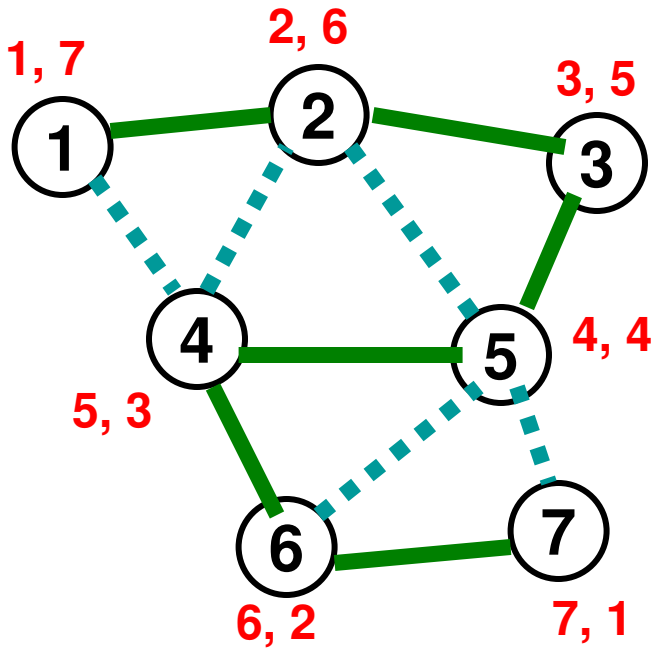


The graph is connected and has no APs

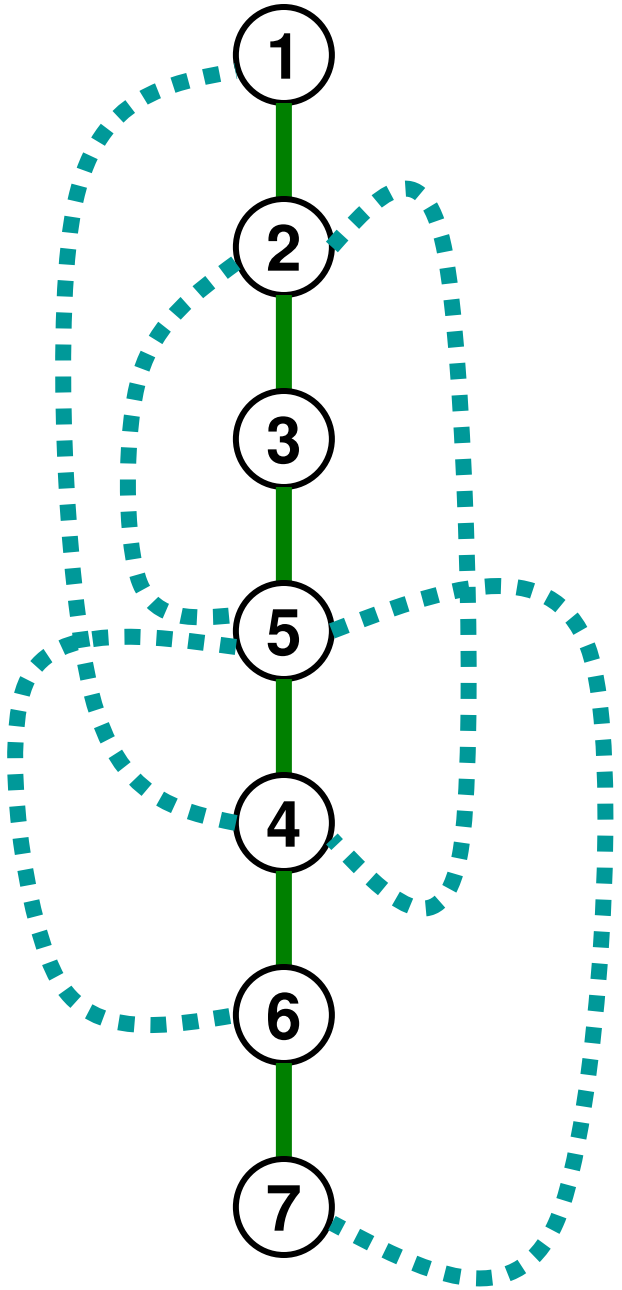
Examples of vertex disjoint paths

1, 2: 1 – 2; 1 – 7 – 2
1, 3: 1 – 3; 1 – 2 – 4 – 3
1, 4: 1 – 2 – 4; 1 – 3 – 4
1, 5: 1 – 3 – 5; 1 – 2 – 4 – 6 – 5
1, 6: 1 – 2 – 4 – 6; 1 – 3 – 5 – 6
2, 3: 2 – 1 – 3; 2 – 4 – 3
2, 4: 2 – 4; 2 – 1 – 3 – 4
2, 5: 2 – 1 – 3 – 5; 2 – 4 – 6 – 5

Articulation Points: Example-3

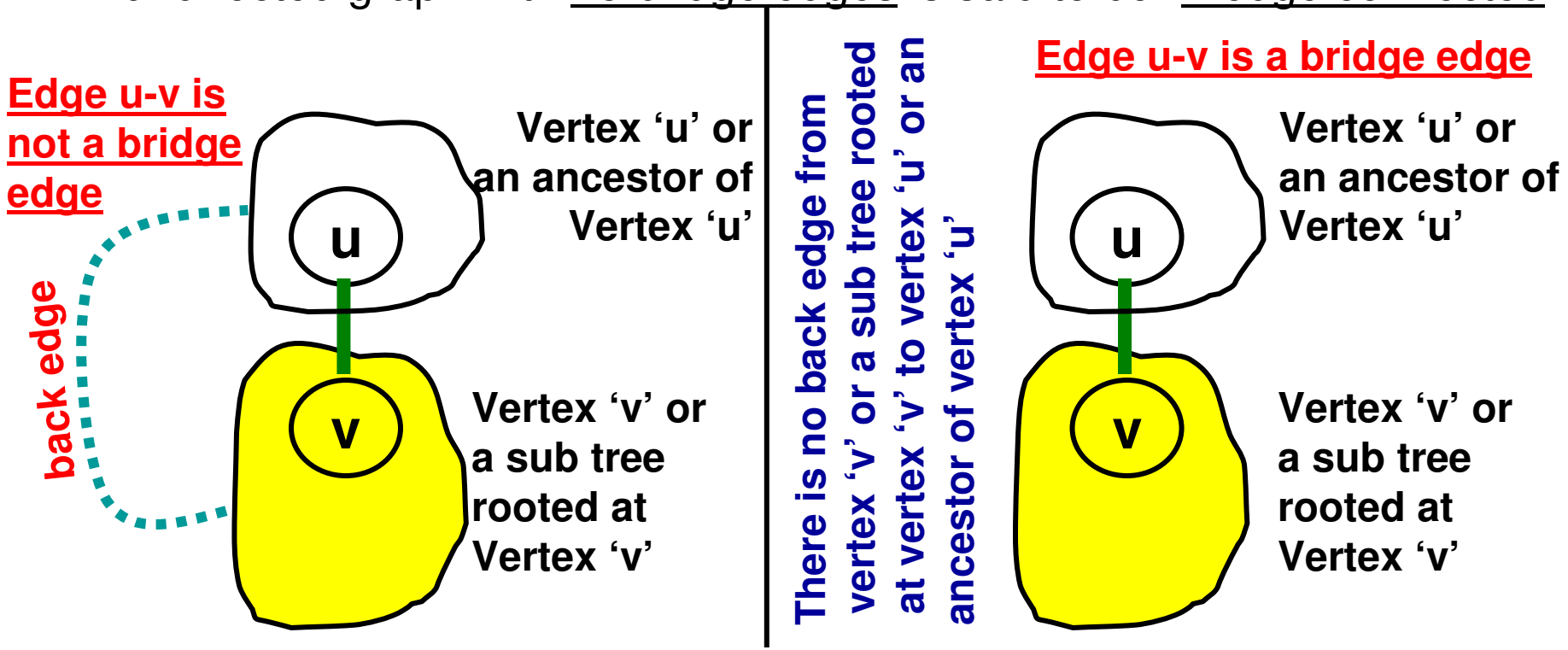


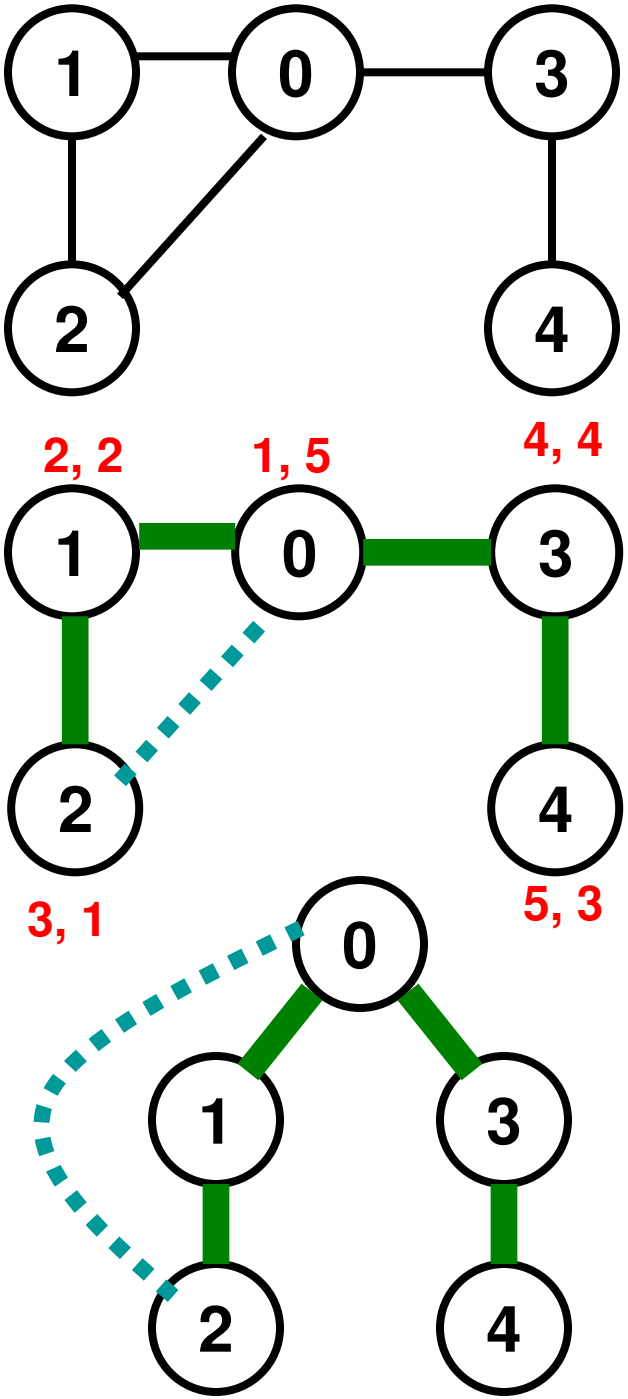
No vertex is an AP.
The graph is connected.
Hence, it is a biconnected graph



Bridge Edges in a Graph

- An edge is a “bridge” edge in a connected graph if its removal would disconnect the vertices in the graph.
- We can identify the bridge edges of a graph using DFS
  - The “back” edges are not bridge edges
  - A tree edge u – v is a bridge if there do not exist any back edge to reach u or an ancestor of u in the sub tree rooted at v.
- An undirected graph with no bridge edges is said to be “2-edge connected”





Bridge Edges in a Graph: Example 1

Edge 0 – 1 is not a bridge edge as there is a back edge from the sub tree rooted at vertex 1 to vertex 0

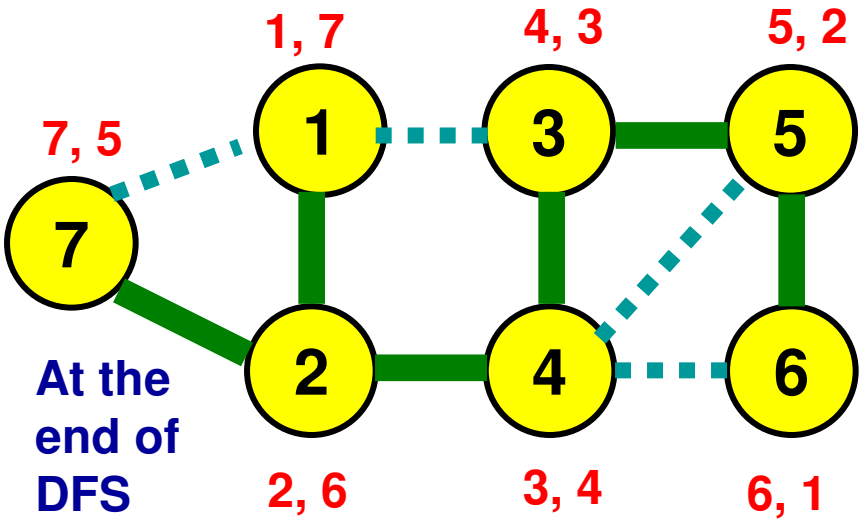
Edge 1 – 2 is not a bridge edge as there is a back edge from the sub tree rooted at vertex 2 to an ancestor of vertex 1

Edge 0 – 2 is not a bridge edge as it is a back edge

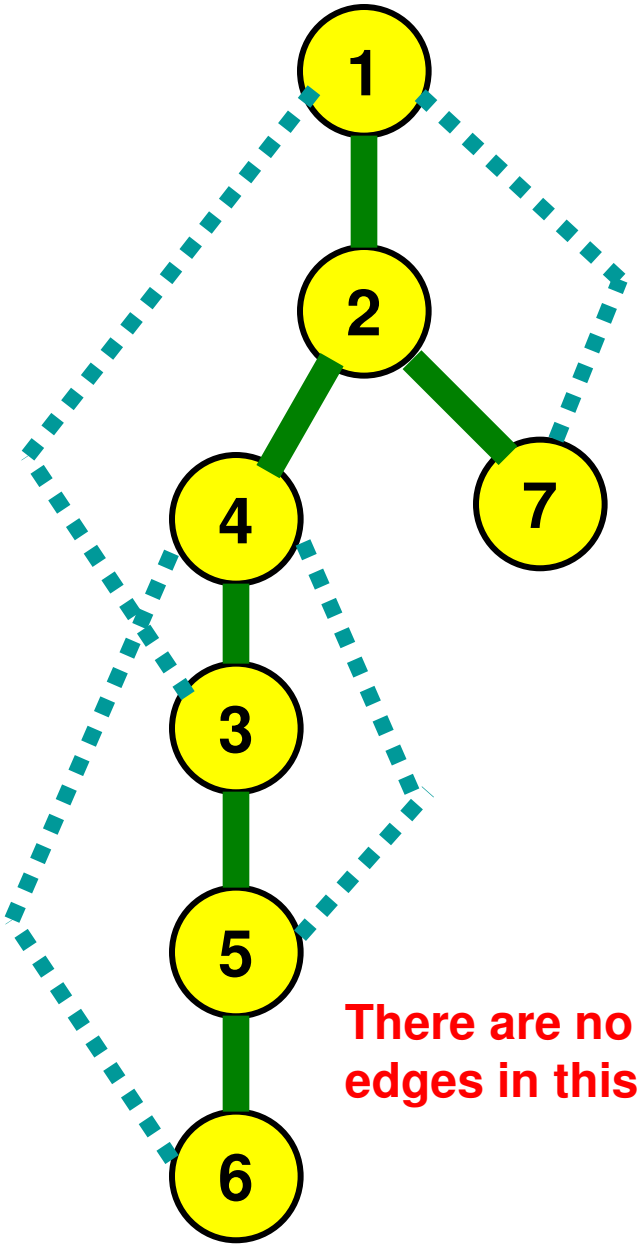
Edge 0 – 3 is a bridge edge as there is no back edge from the sub tree rooted at vertex 3 to either vertex 0 or an ancestor of vertex 0.

Edge 3 – 4 is a bridge edge as there is no back edge from the sub tree rooted at vertex 4 to either vertex 3 or an ancestor of vertex 3.

Bridge Edges in a Graph: Example 2

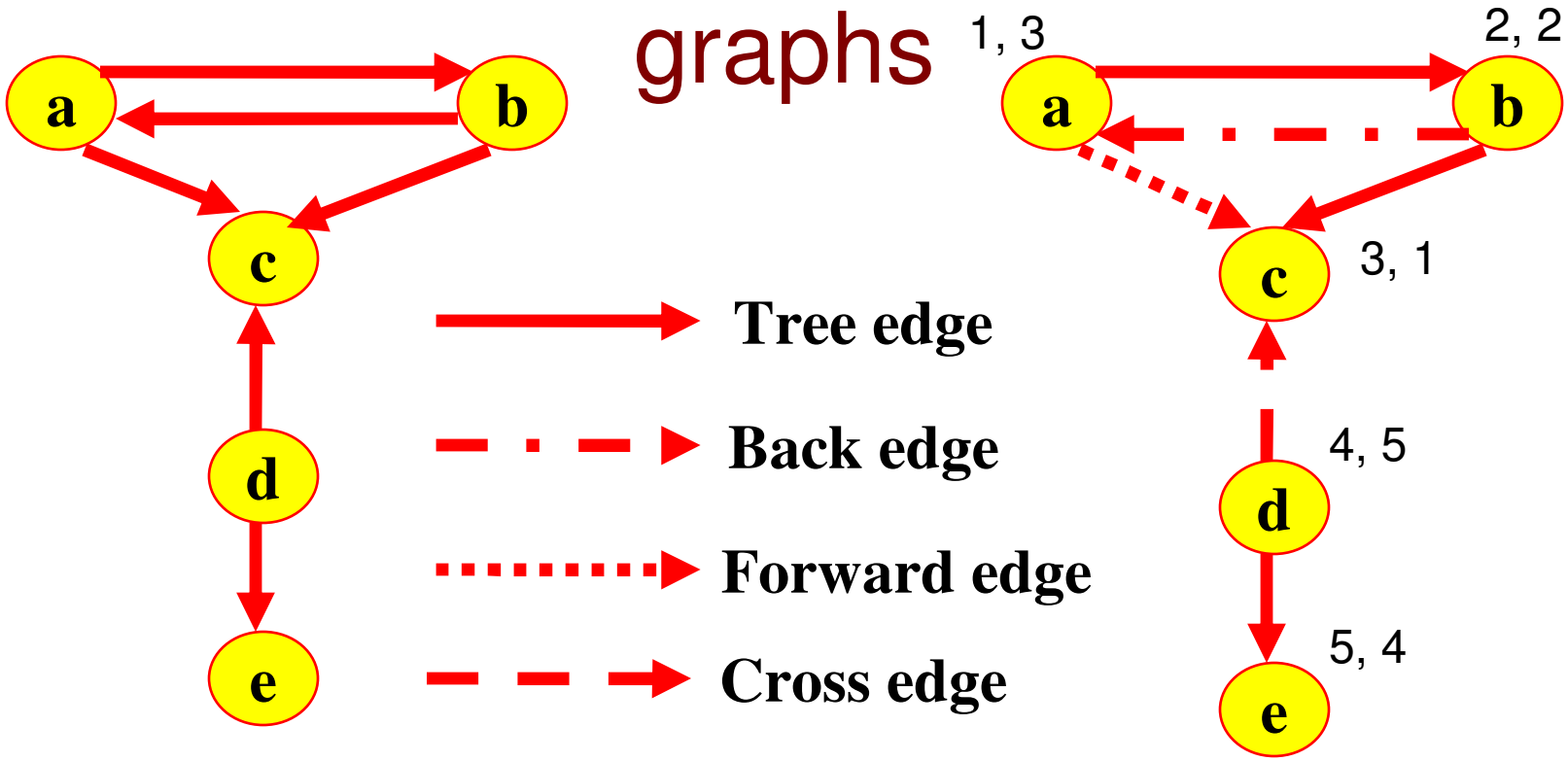


Candidate Edge	Back Edge(s) making it not a bridge edge
1 – 2	3 – 1
2 – 4	3 – 1
4 – 3	6 – 4, 5 – 4
3 – 5	6 – 4, 5 – 4
5 – 6	6 – 4
2 – 7	7 – 1



There are no bridge edges in this graph

DFS: Edge Terminology for directed graphs

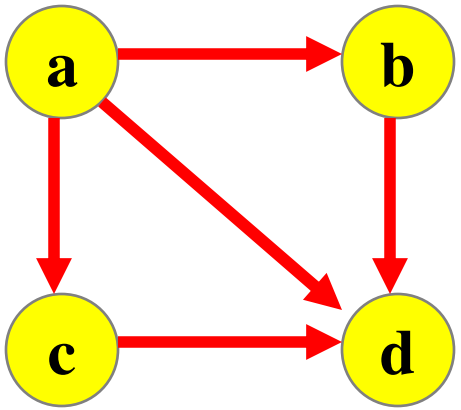


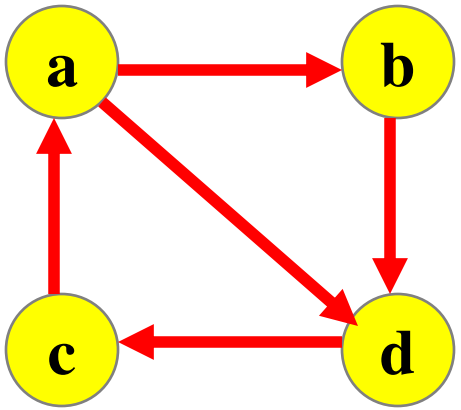
Tree edge – an edge from a parent node to a child node in the tree
Back edge – an edge from a vertex to its ancestor node in the tree
Forward edge – an edge from an ancestor node to its descendant node in the tree. The two nodes do not have a parent-child relationship. The back and forward edges are in a single component (the DFS tree).
Cross edge – an edge between two different components of the DFS Forest. So, basically an edge other than a tree edge, back edge and forward edge

Directed Acyclic Graphs (DAG)

- A directed graph is a graph with directed edges between its vertices (e.g., u → v).
- A DAG is a directed graph (digraph) without cycles.
  - A DAG is encountered for many applications that involve pre-requisite restricted tasks (e.g., course scheduling)

a DAG

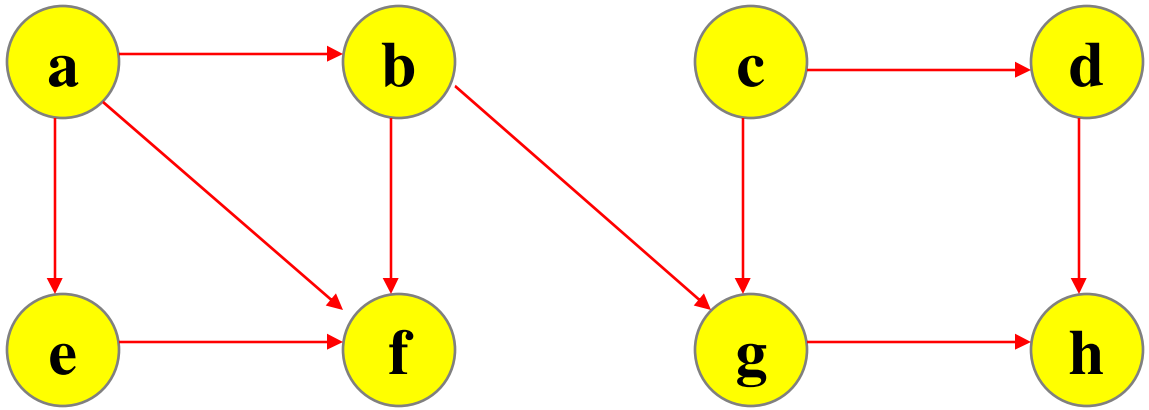




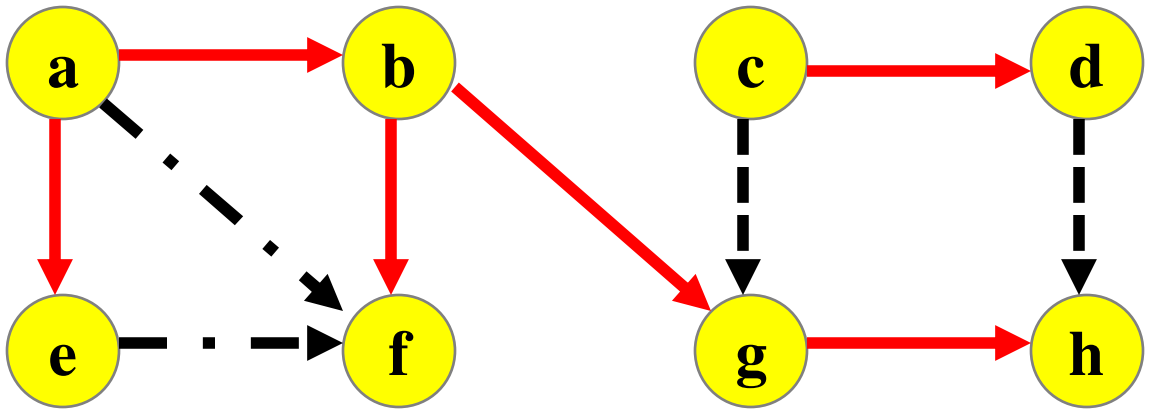
not a DAG

To test whether a directed graph is a DAG, run DFS on the directed graph. If a back edge is not encountered, then the directed graph is a DAG.

DFS on a DAG: Example 1



$h_{5,2}$
$g_{4,3}$
$f_{3,1}$
$b_{2,4}$ $e_{6,5}$ $d_{8,7}$
$a_{1,6}$ $c_{7,8}$



Forward edge

Cross edge

Order in which the Vertices are popped of from the stack

f h g b e a d c

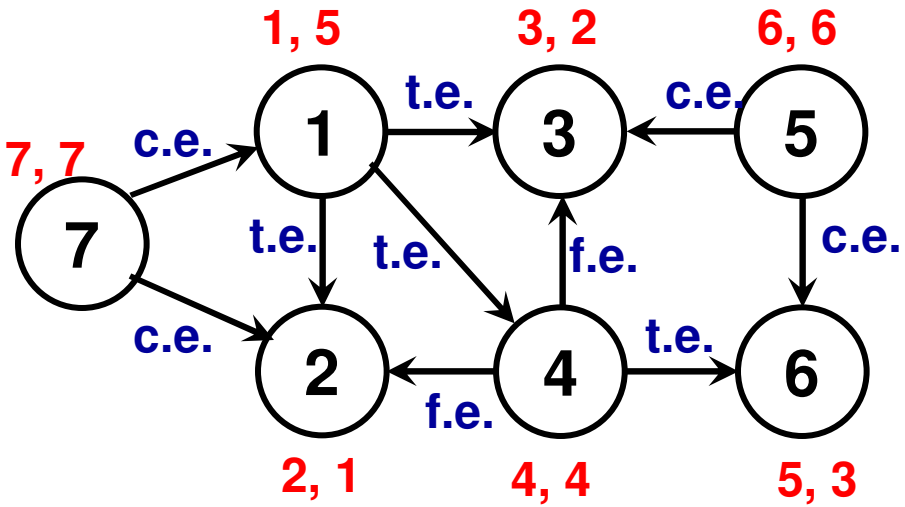
Reverse the order

Topological Sort

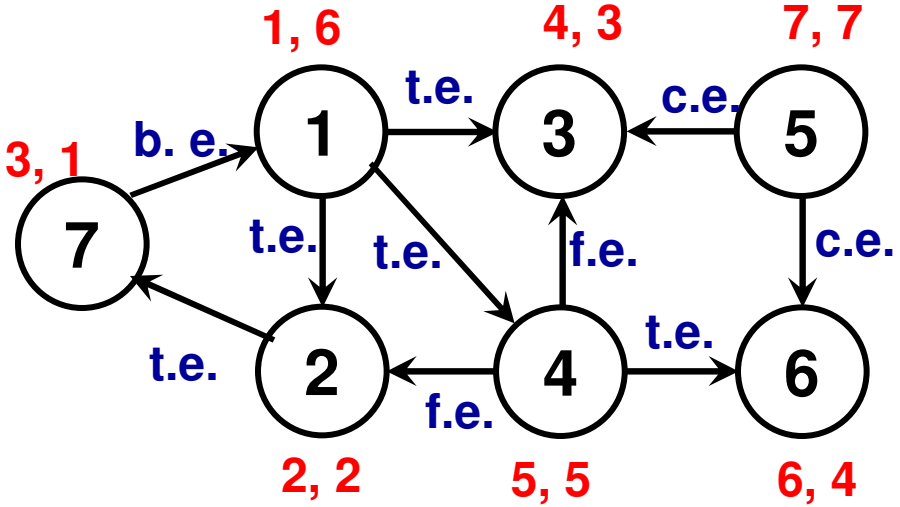
c d a e b g h f

Topological Sort of a DAG

- Topological sort of a DAG is a listing of the vertices (each vertex appears exactly once) in such a way that for any edge u → v, u appears somewhere before v (u …. v) in the topological sort.
- A topological sort can be written for a directed graph if only if it is a DAG.
  - Directed graph is a DAG ➔ Topological sort exists



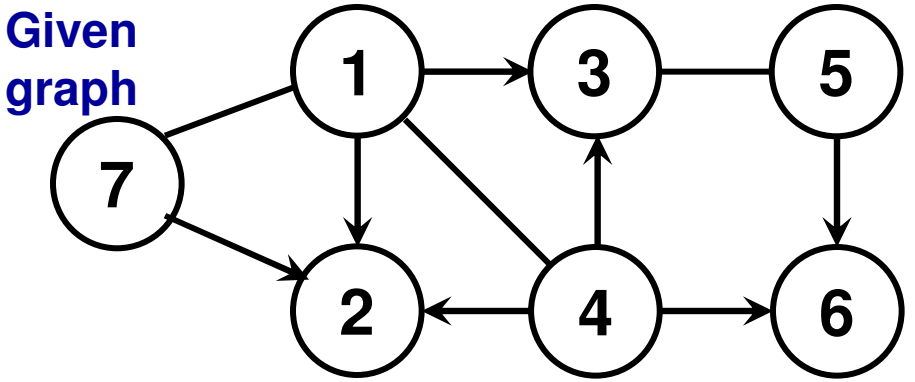
Topological Sort: 7, 5, 1, 4, 6, 3, 2

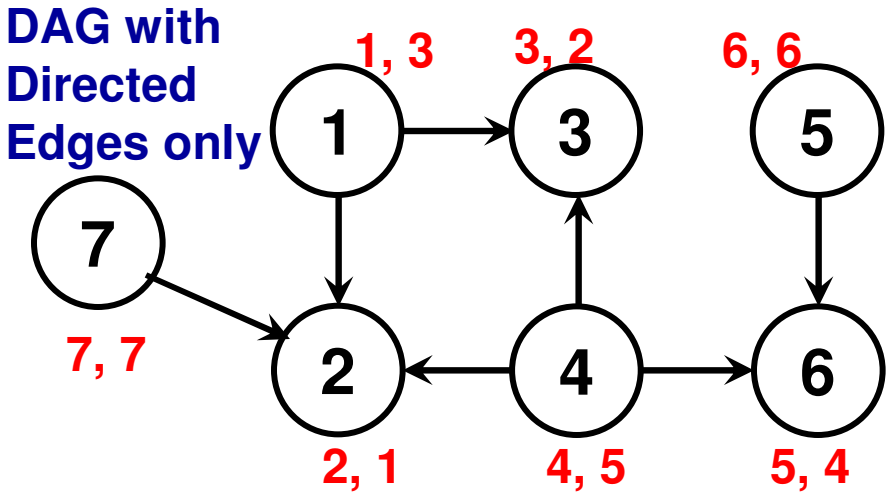


Topological Sort does not exists!

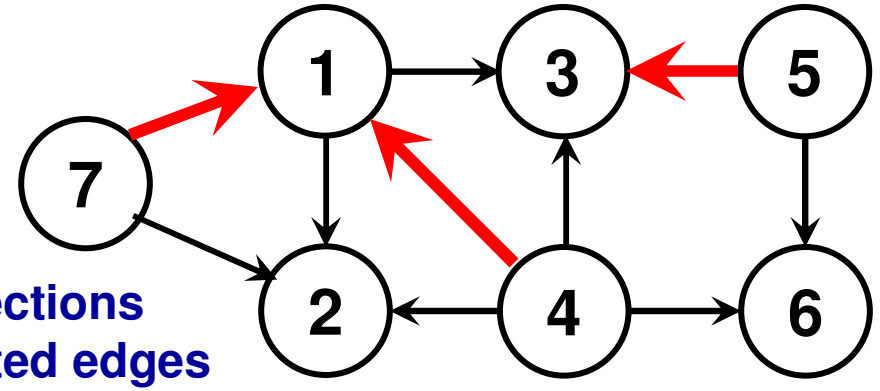
Assigning Directions to Undirected Edges in a Graph (DAG)

- Consider a directed graph that has a mix of undirected and directed edges. The directed edges of the graph do not create a cycle. We want to assign directions to the undirected edges so that the directed graph remains acyclic.
- Solution:
- Run DFS on the directed edges alone and determine a topological sort of the directed edges (a DAG)
- Consider an undirected edge u – v. Assign the direction u → v if u appears before v in the topological sort; otherwise, assign v → u



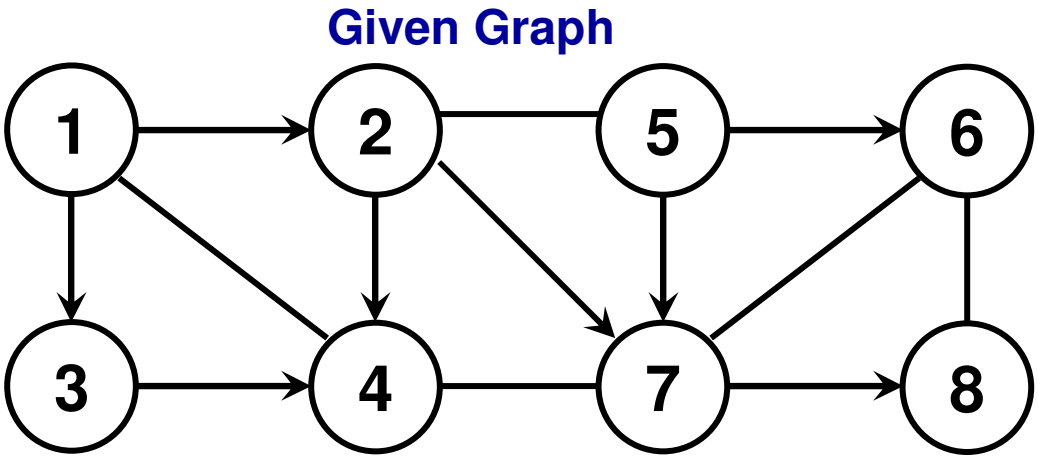


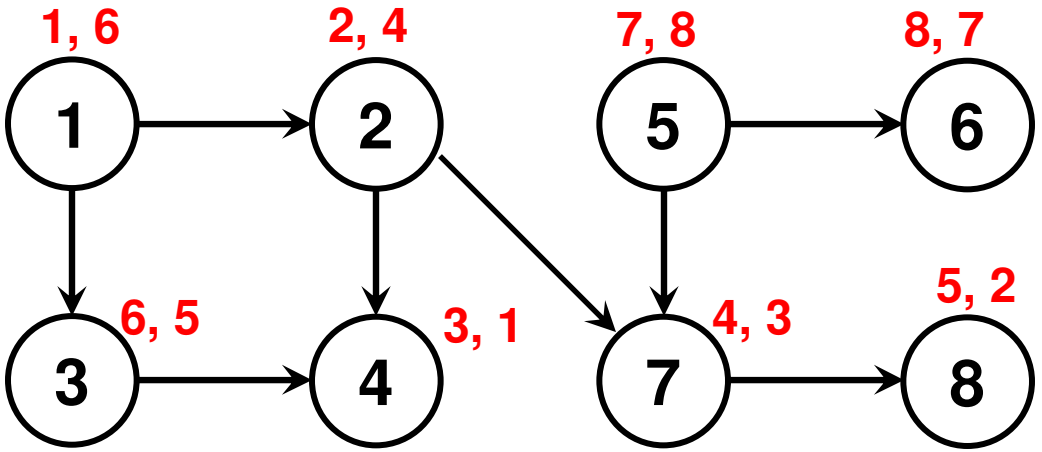
Topological Sort (DAG): 7, 5, 4, 6, 1, 3, 2

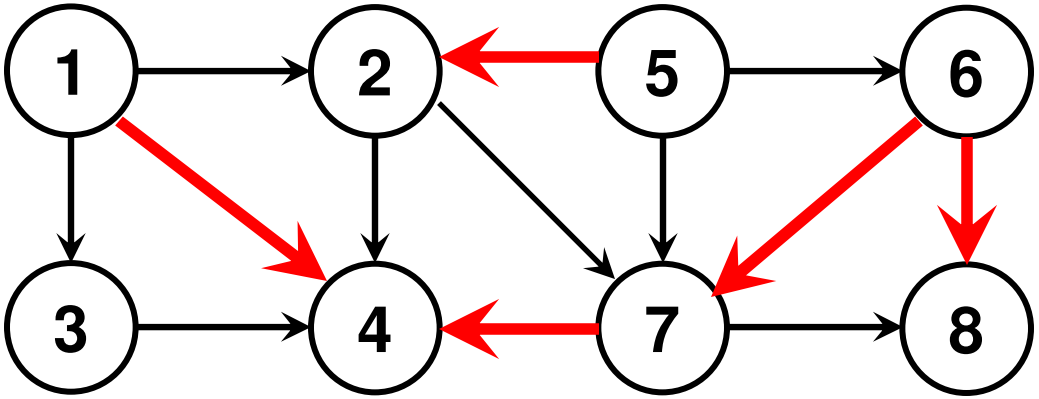


Assigning directions to the undirected edges

Assigning Directions to Undirected Edges in a Graph (DAG): Example 2







Graph of Directed Edges (a DAG)

Topological Sort

5, 6, 1, 3, 2, 7, 8, 4

Strongly and Weakly Connected Components of a Directed Graph

- Recall, a component is the largest subset of the vertices that satisfy a particular property and if we add any other vertex to that component, the property will no longer be satisfied.

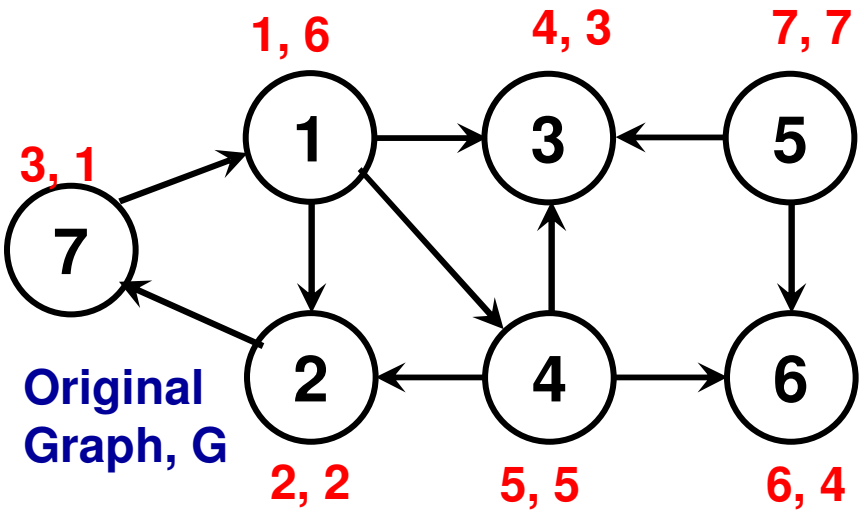
- A strongly connected component of a directed graph is the subset of the vertices that are reachable from each other, either directly or through a multi-hop path.
  - Note that if a directed graph is a DAG, there are no strongly connected components of size more than 1.

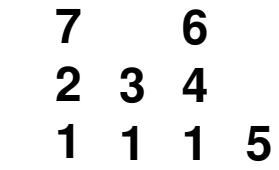
- A weakly connected component of a directed graph is the subset of the vertices that are reachable from each other in the undirected version of the graph.

Strongly Connected Components (SCC) of a Directed Graph

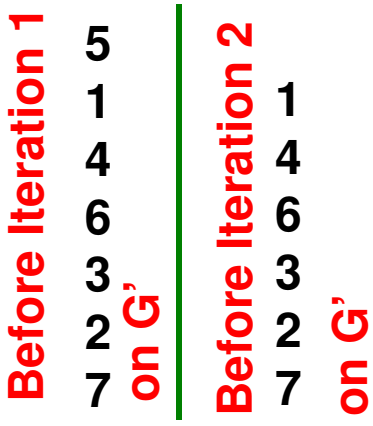
- We maintain two stacks: Regular-Stack and SCC-Stack
- We conduct DFS on the given directed graph (G) and use the Regular-Stack to push and pop vertices as part of the DFS.
  - Whenever a vertex is popped out of the Regular-Stack, push it to the SCC-Stack.
  - Run DFS until all vertices are visited.
- Reverse the directions of the edges in the directed graph (call it G').
- Pop the topmost vertex from SCC-Stack and run DFS on G' starting from that vertex. All the vertices visited as part of this DFS are said to form a strongly connected component.
  - Remove all the vertices visited from G' and the SCC-Stack
  - Run DFS on the remaining version of G' starting from the topmost vertex in the SCC-Stack.
  - Continue DFS until are vertices in G' are visited.

Strongly Connected Components for a Directed Graph: Example 1

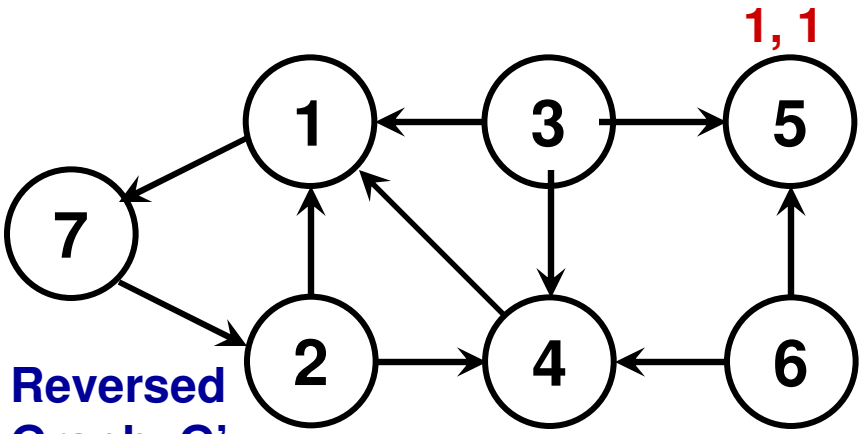


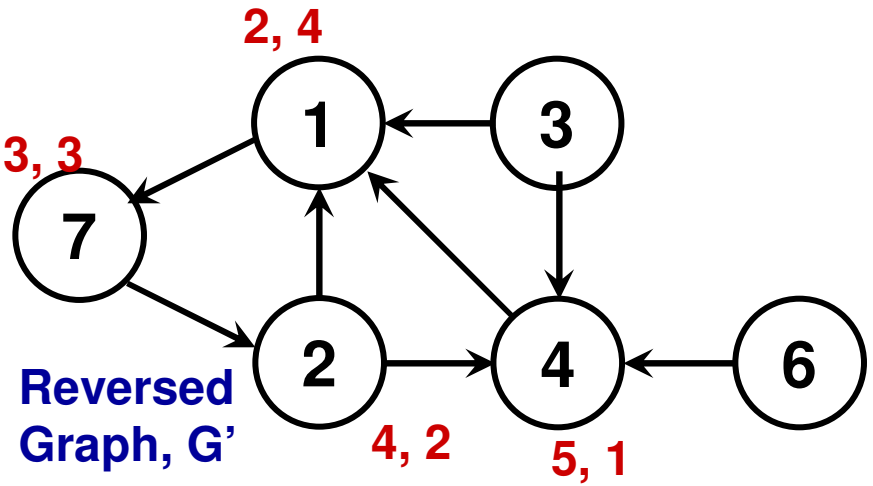


Regular Stack

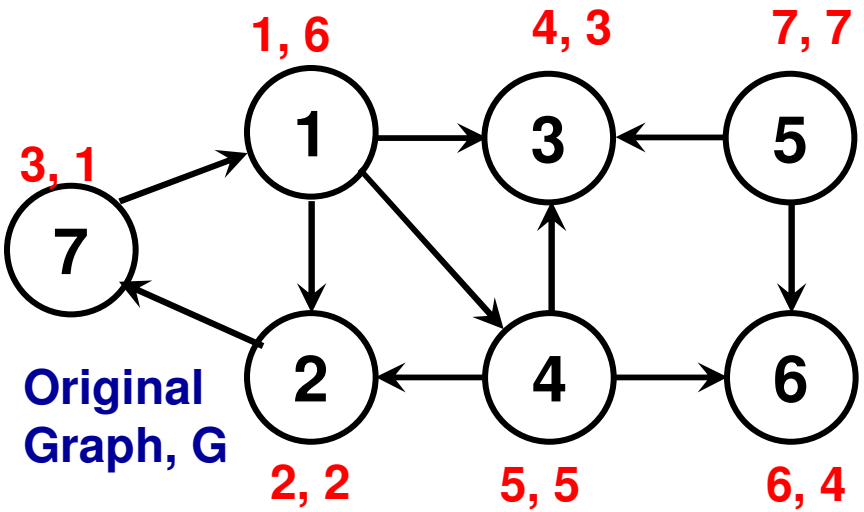


SCC Stack





Strongly Connected Components for a Directed Graph: Example 1

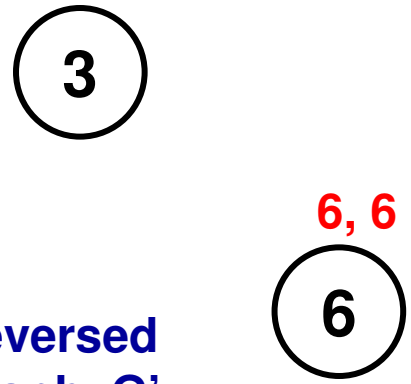


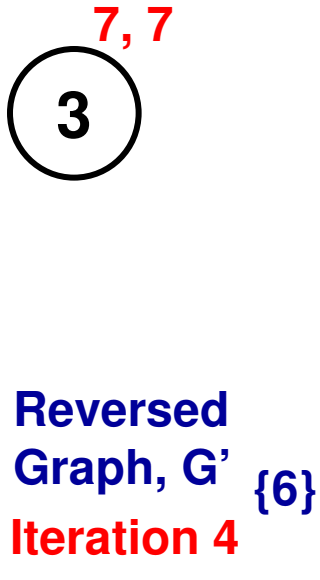
Regular Stack

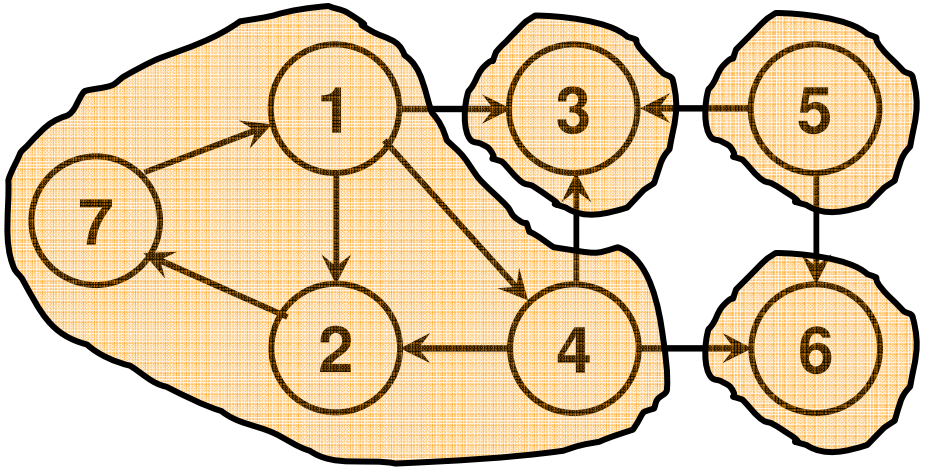
7		6	
2	3	4	
1	1	1	5

SCC Stack

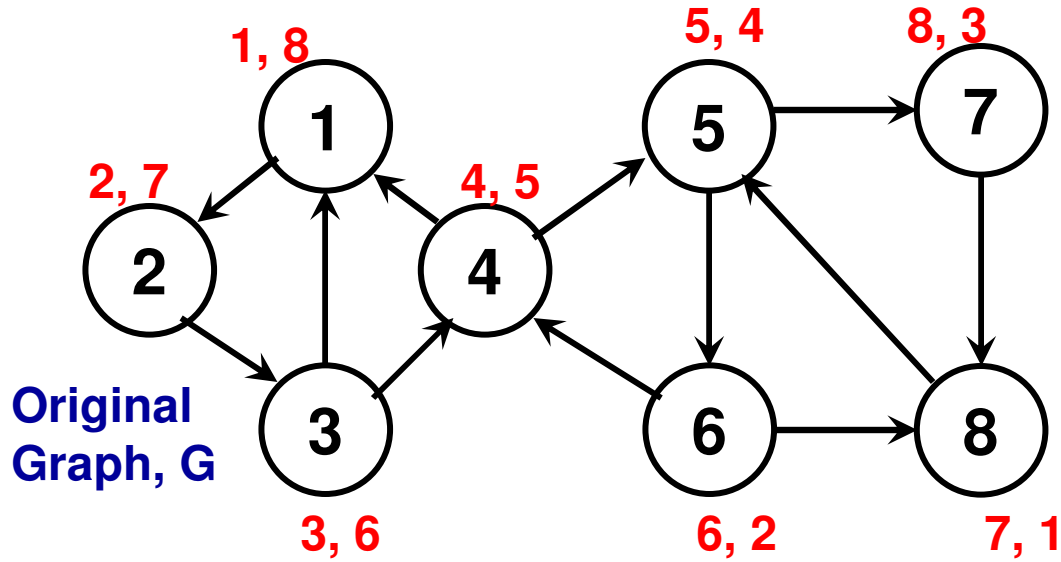
Before Iteration 1 on G'	Before Iteration 2 on G'
5	1
1	4
4	6
6	3
3	2
2	7
7	







Strongly Connected Components of a Directed Graph: Example 2



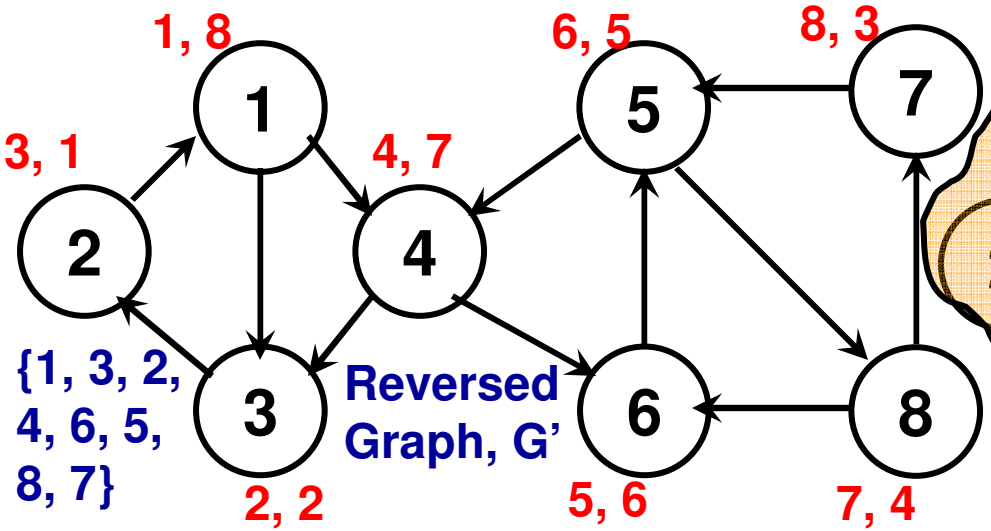
Regular Stack

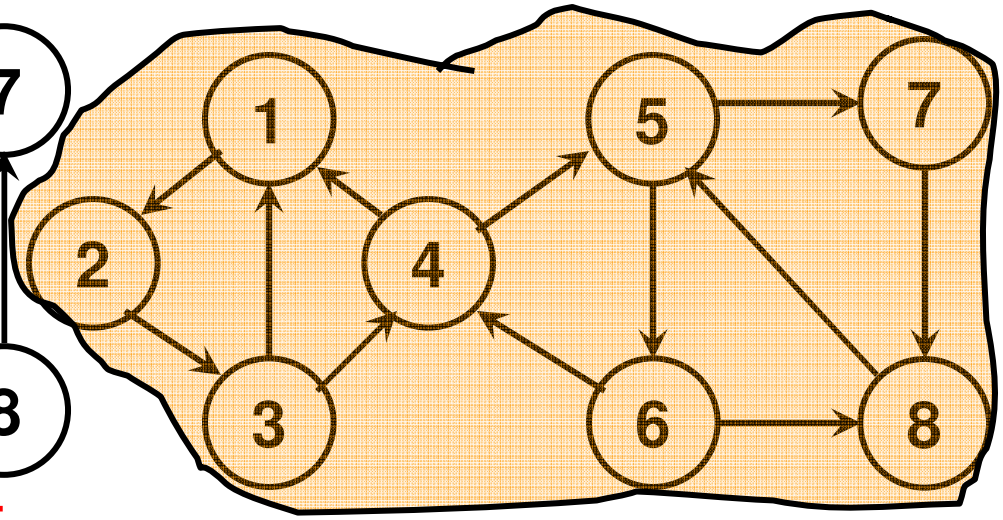
8
6 7
5 5
4 4
3 3
2 2
1 1

Before Iteration 1 on G'

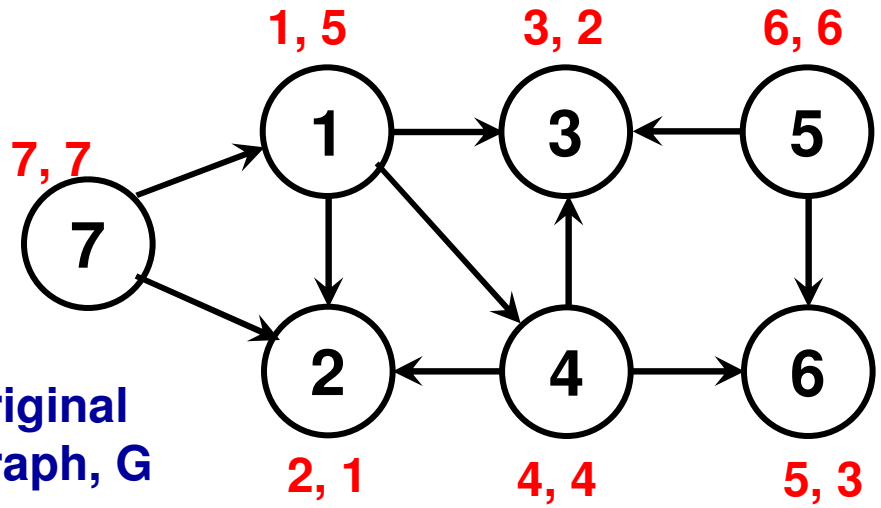
SCC Stack

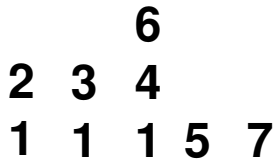
1
2
3
4
5
7
6
8

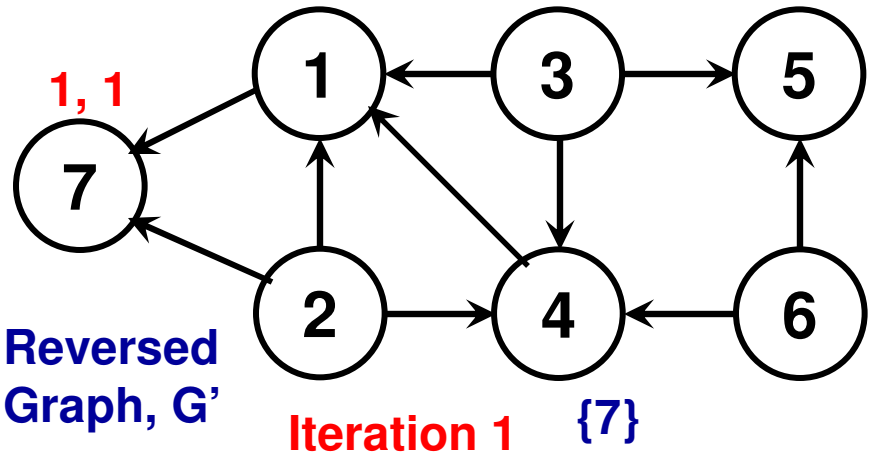


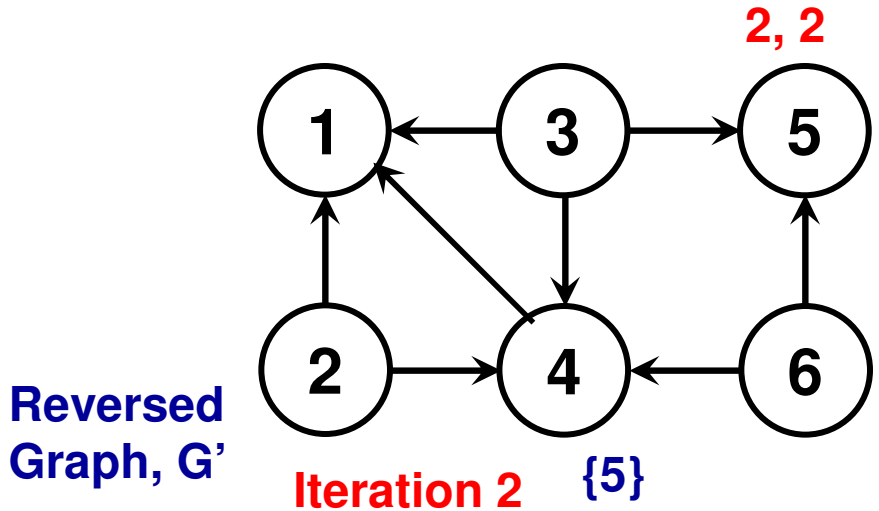


Strongly Connected Components of a Directed Graph: Example 3

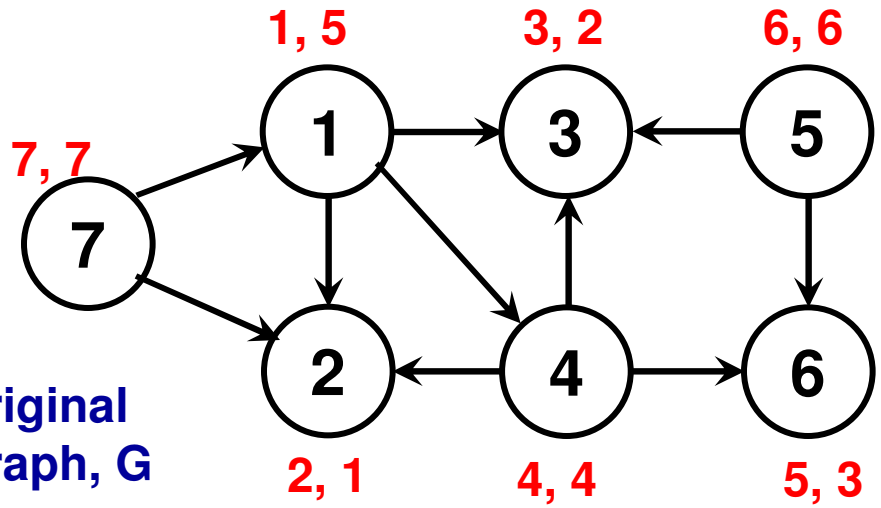




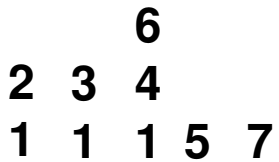




Strongly Connected Components of a Directed Graph: Example 3

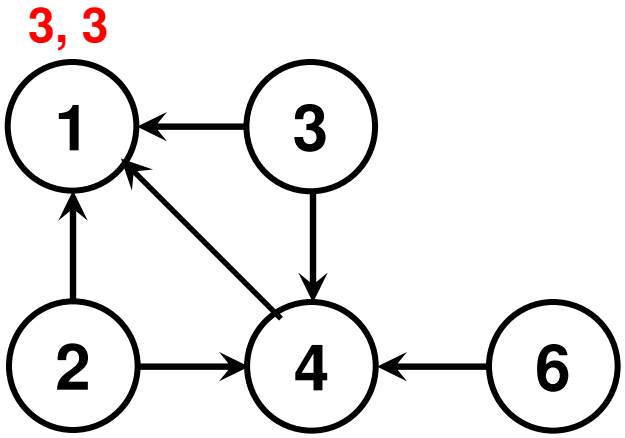


Original Graph, G



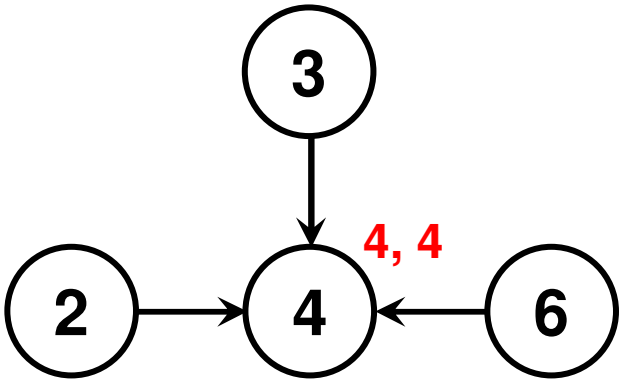
Regular Stack

SCC Stack



Reversed Graph, G'

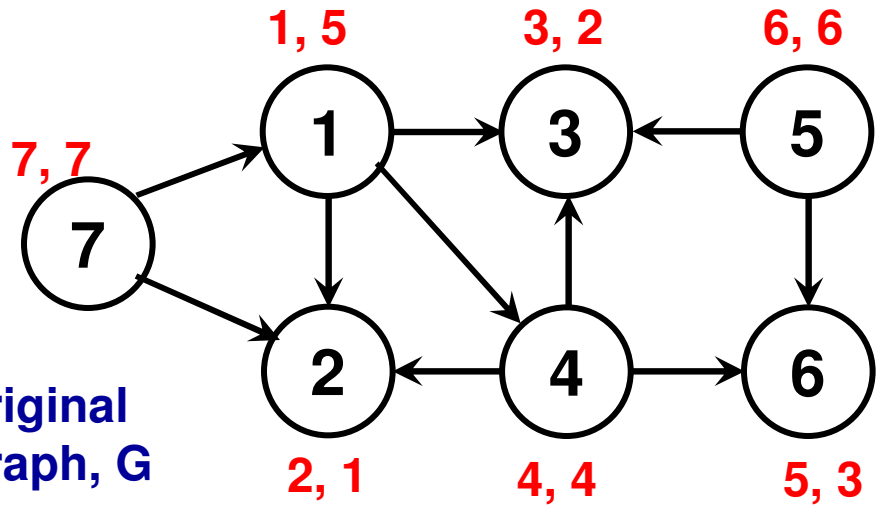
Iteration 3 {1}

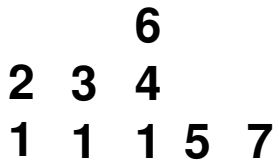


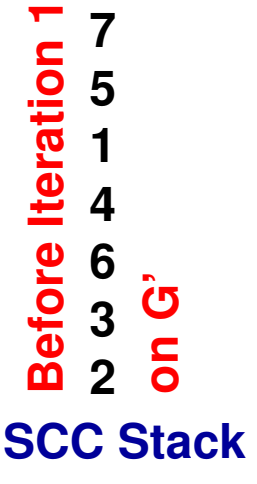
Reversed Graph, G'

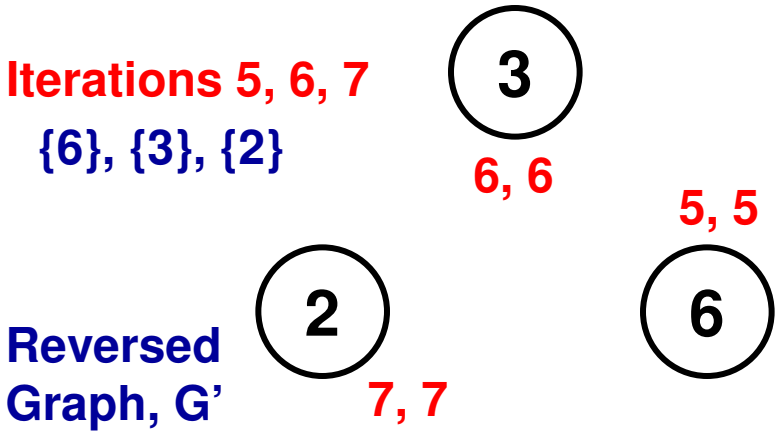
Iteration 4 {4}

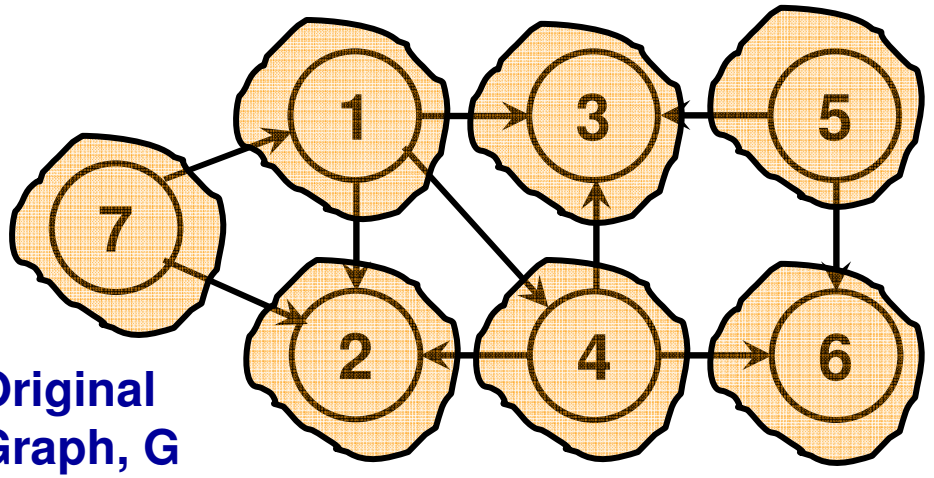
Strongly Connected Components of a Directed Graph: Example 3







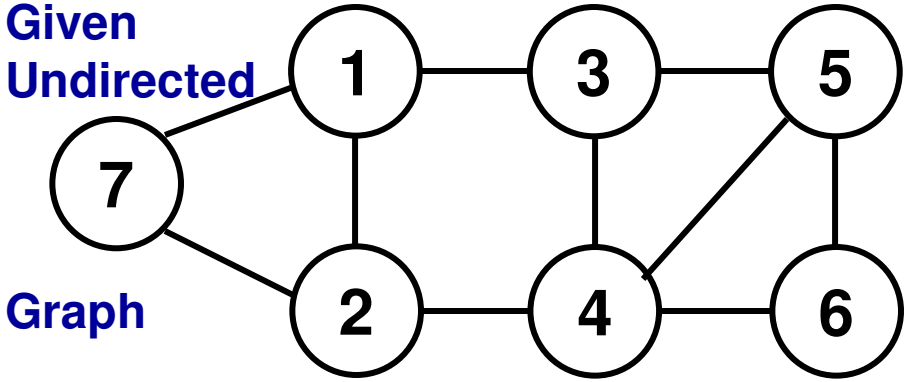


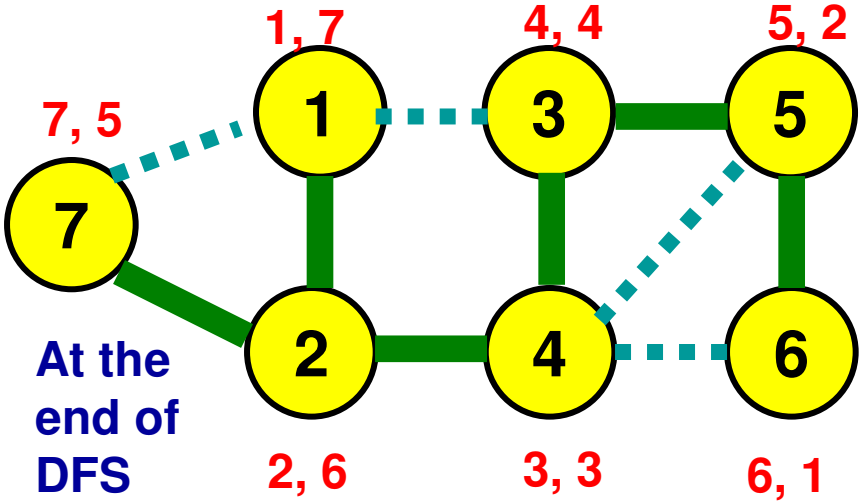


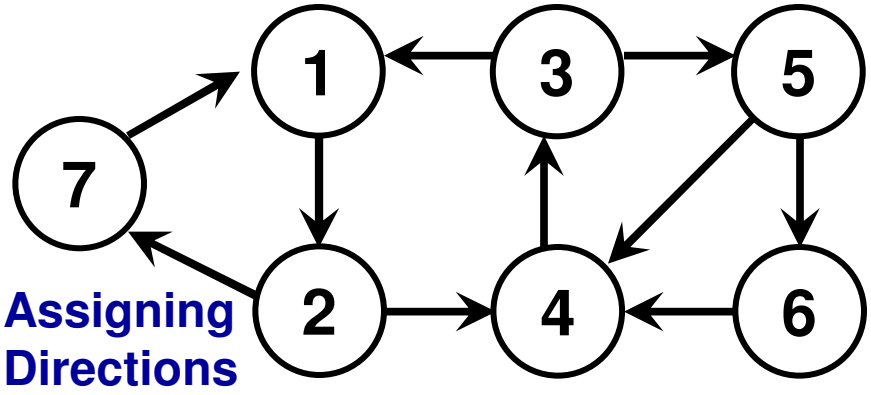
Assigning Directions to Edges in an Undirected Graph (Strongly Connected Directed Graph)

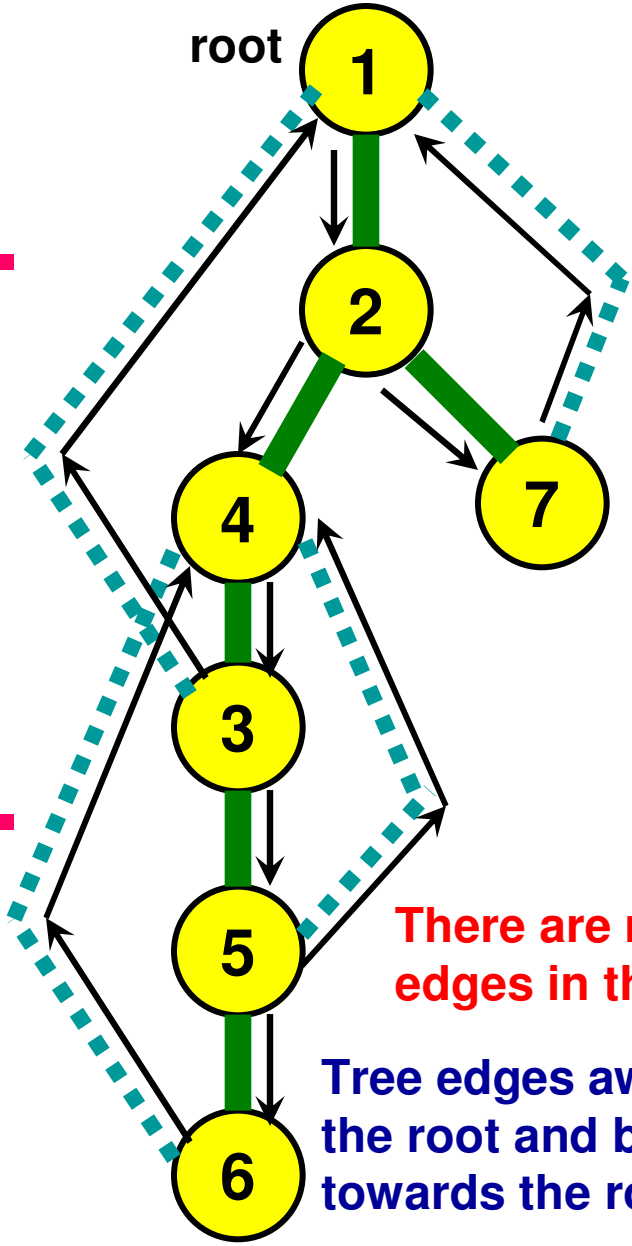
- We are given an undirected graph that is connected and has no bridges (i.e., the graph is 2-edge connected).
- We want to assign directions to the edges of this graph so that the resulting directed graph has all the vertices in a single strongly connected component (i.e., the resulting directed graph is strongly connected).
- **Solution:**
- Run DFS on the given undirected graph and make sure it is connected and has no bridges (use the structure of the DFS tree)
- If the undirected graph is “2-edge connected” use the structure of the DFS tree and orient all the tree edges to be away from the root and all the back edges to be towards the root.

Example 1



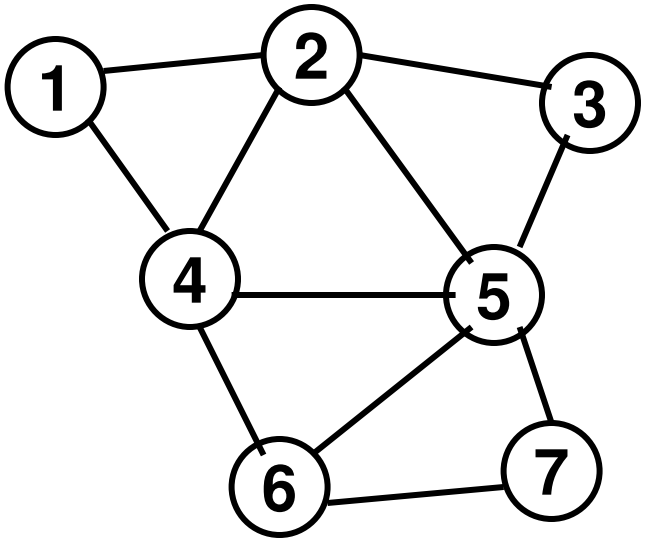


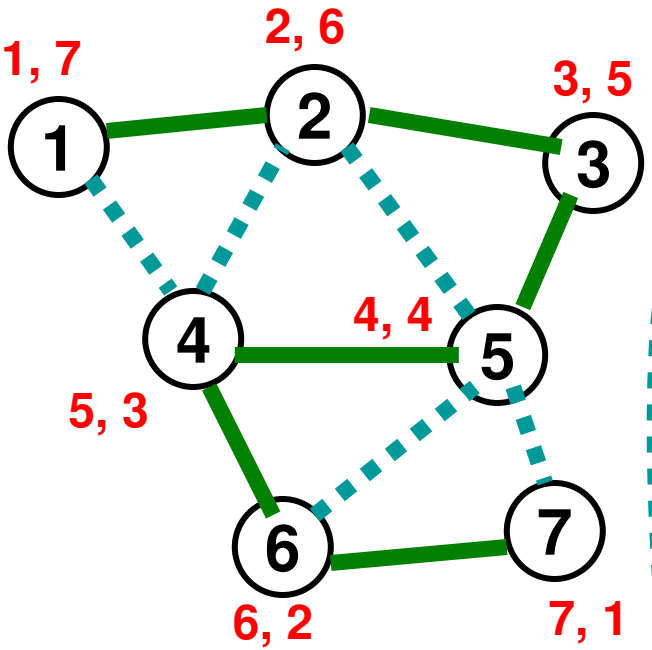


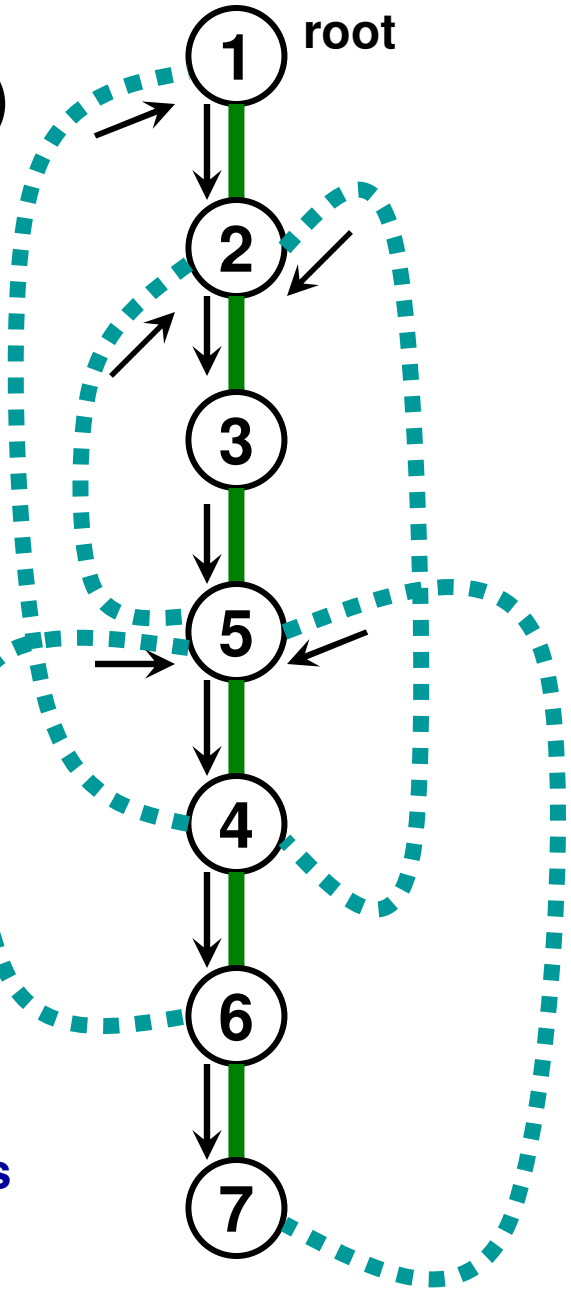


There are no bridge edges in this graph

Tree edges away from the root and back edges towards the root

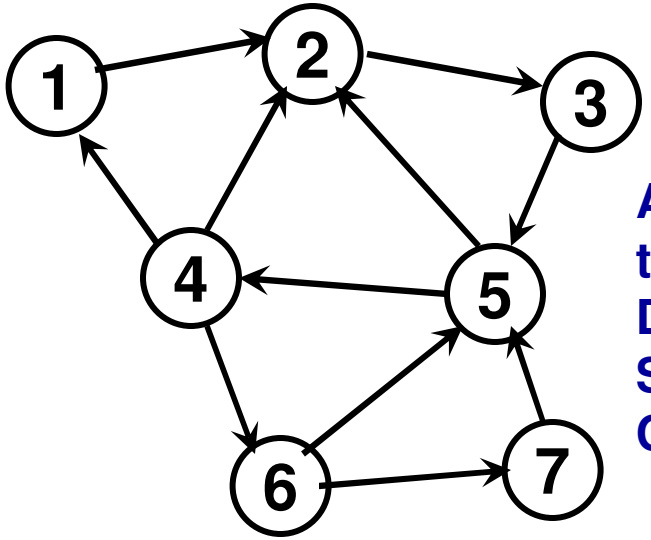






Ex-2

There are no
Bridge edges

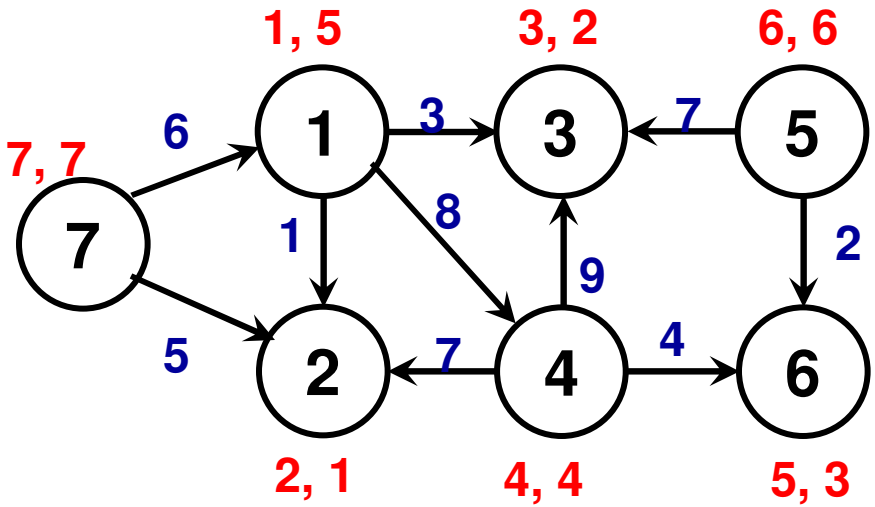


Assigning directions to
the edges so that the
Directed graph is a
Strongly connected
Component of all its vertices

Longest Paths from a (Source) Vertex in a DAG

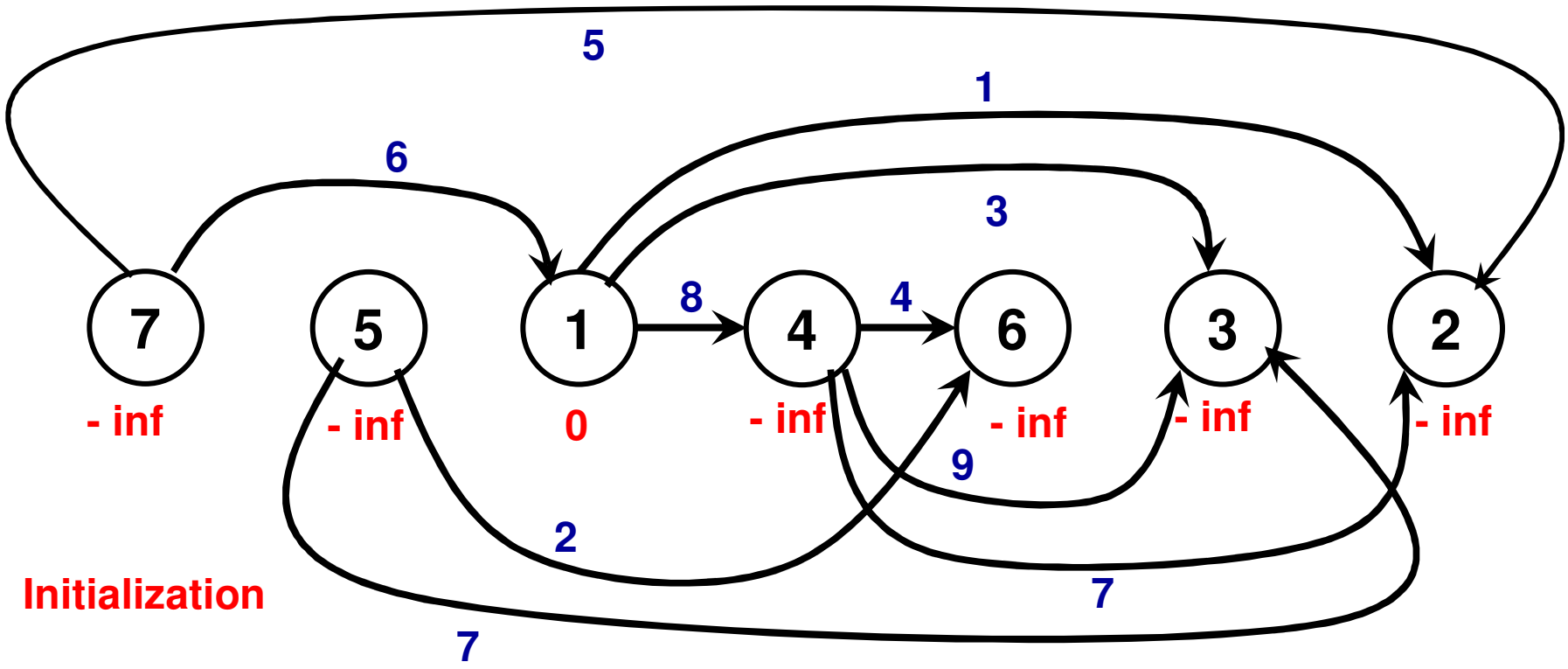
- Given a DAG with edge weights, run DFS on the DAG without considering the edge weights and get a topological sort of the vertices
- Initialize the distance from the source to itself as 0 and the distance from the source to every other vertex as -∞.
- Consider the vertices in the order in which they appear in the topological sort.
  - For a vertex u, consider its outgoing neighbors v.
    - if distance[s ~ v] < distance[s ~ u] + w(u-v) then
      - distance[s ~ v] = distance[s ~ u] + w(u-v)
      - Predecessor[v] = u

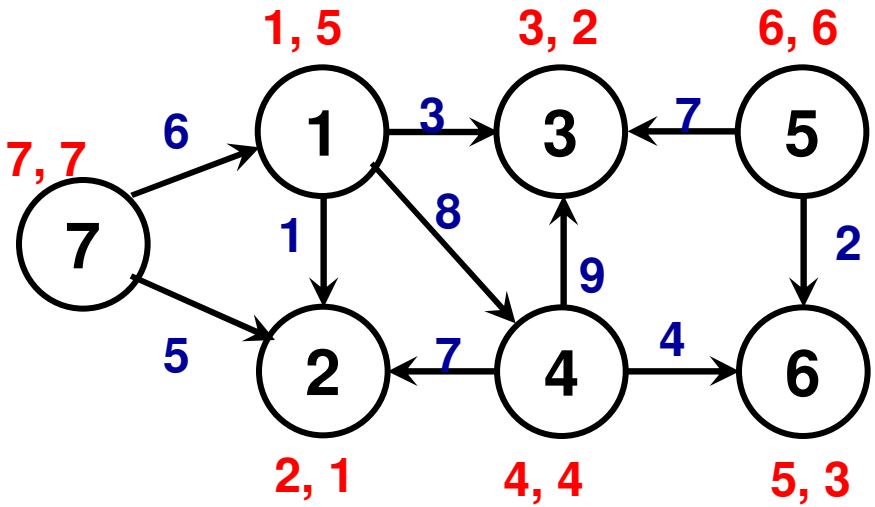
Longest Paths in a DAG: Example 1



Topological Sort
7, 5, 1, 4, 6, 3, 2

Let vertex '1' be the source

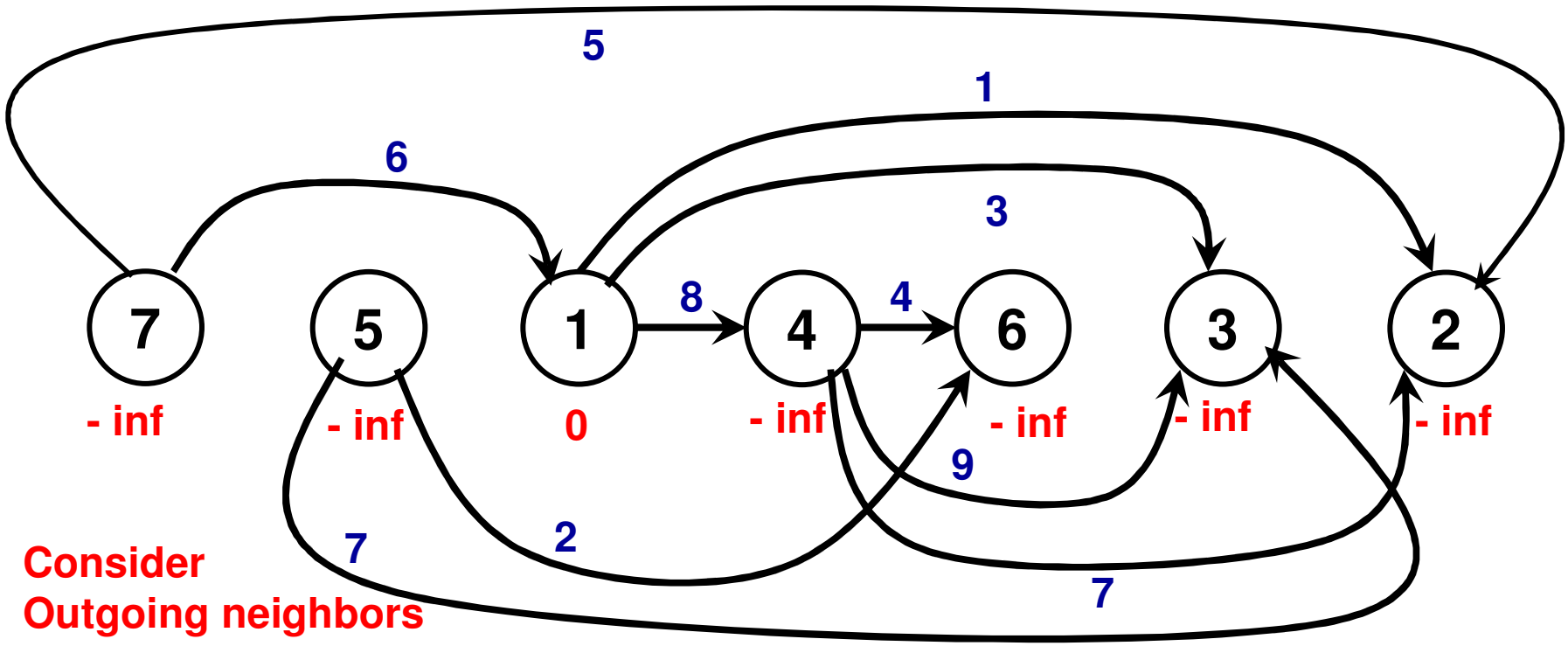




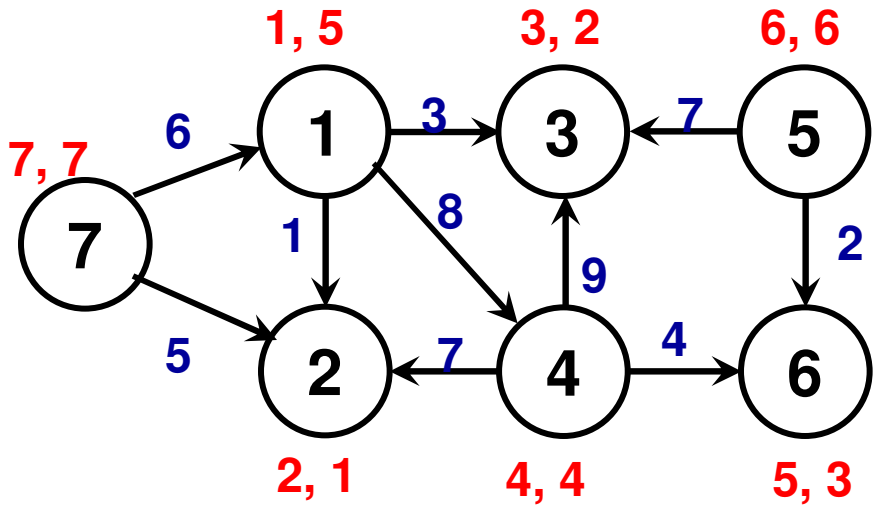
Longest Paths in a DAG: Example 1

Topological Sort
7, 5, 1, 4, 6, 3, 2

Let vertex '1' be the source



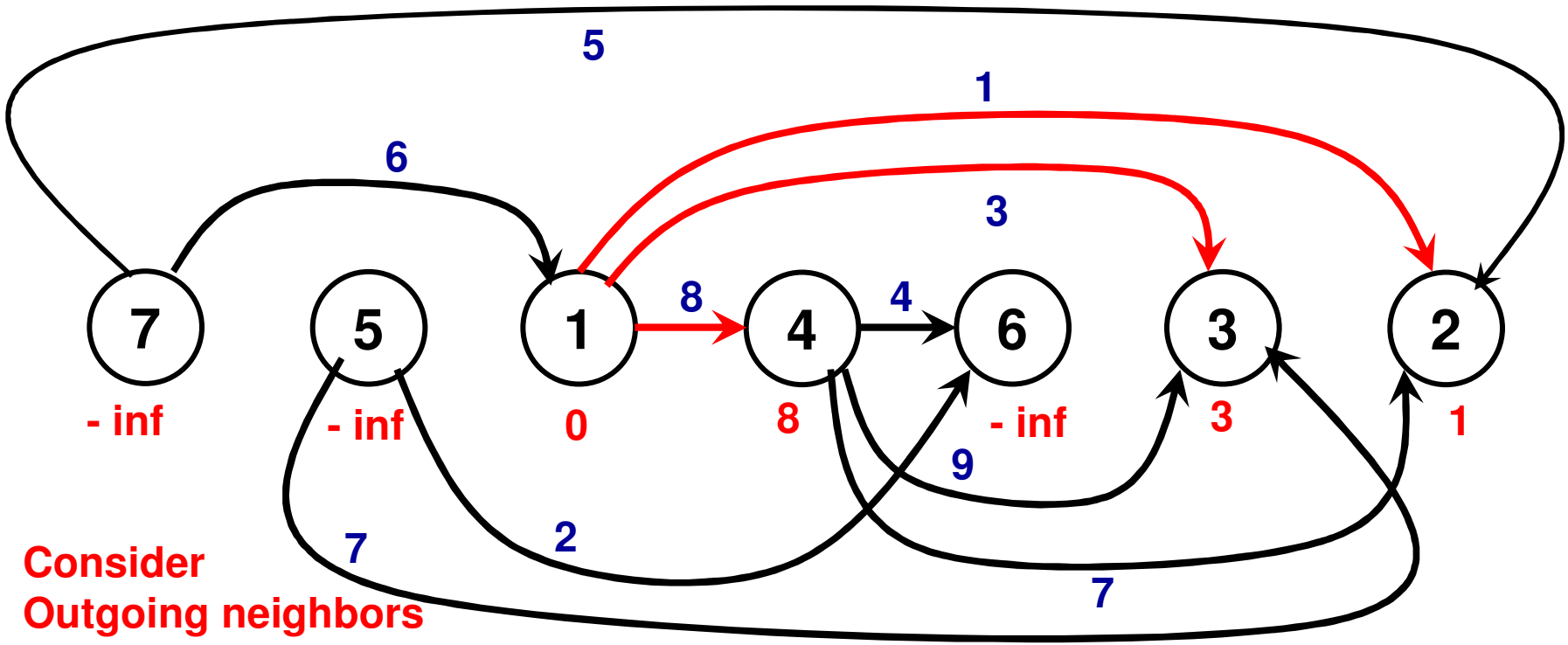
Consider
Outgoing neighbors
of Vertex '7' and '5' (no change)



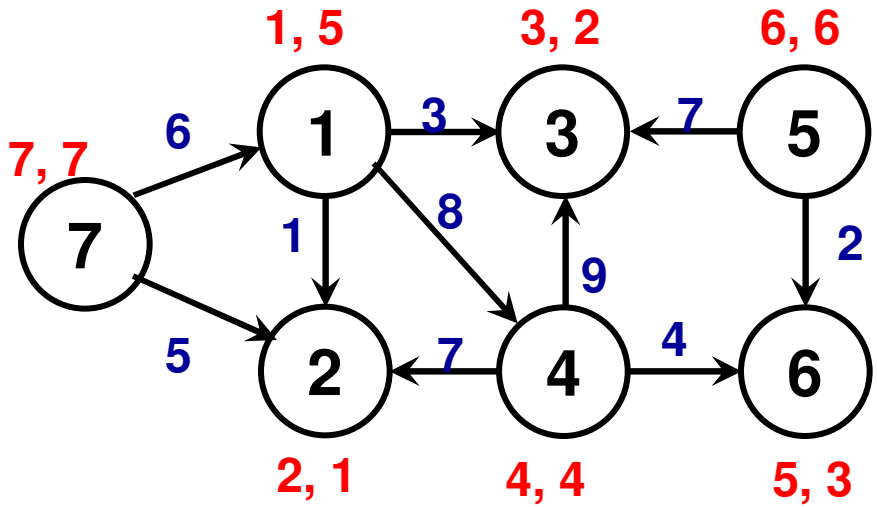
Longest Paths in a DAG: Example 1

Topological Sort
7, 5, 1, 4, 6, 3, 2

Let vertex '1' be the source



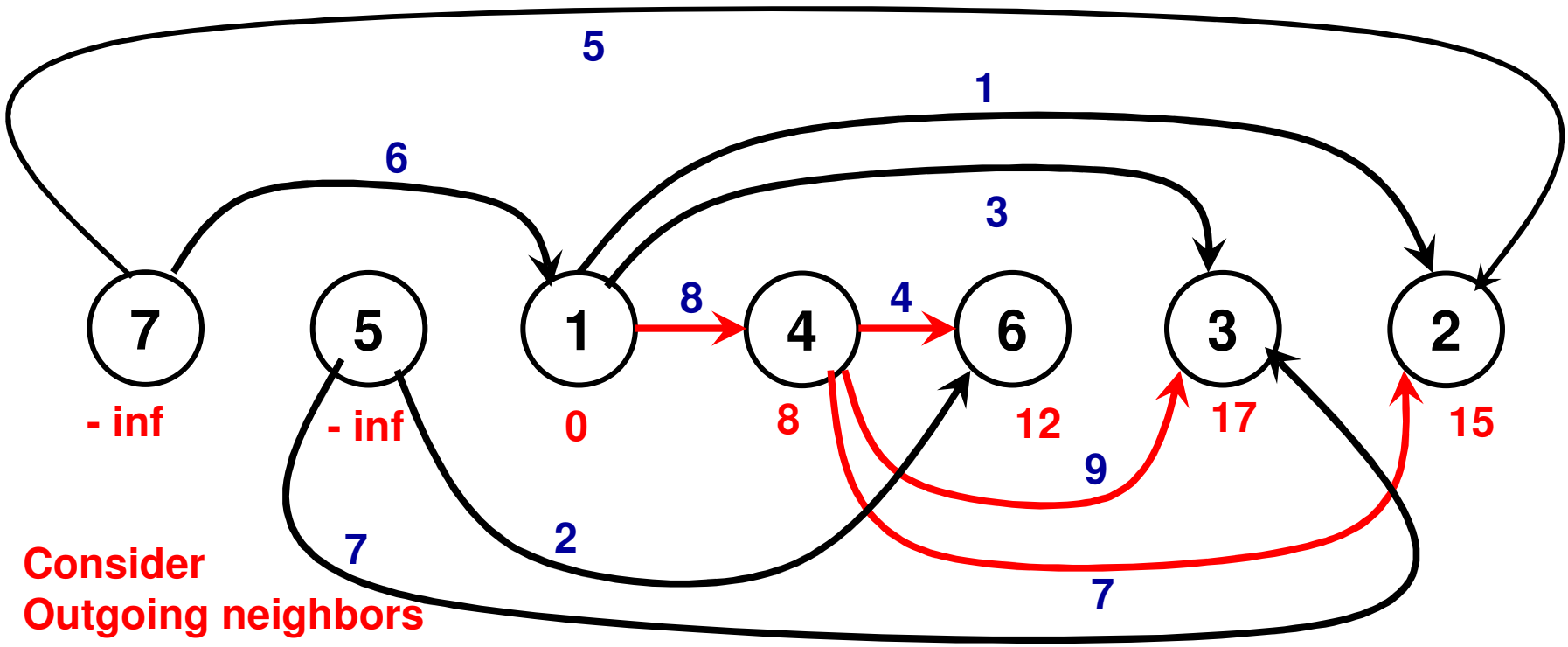
Consider Outgoing neighbors of Vertex '1'



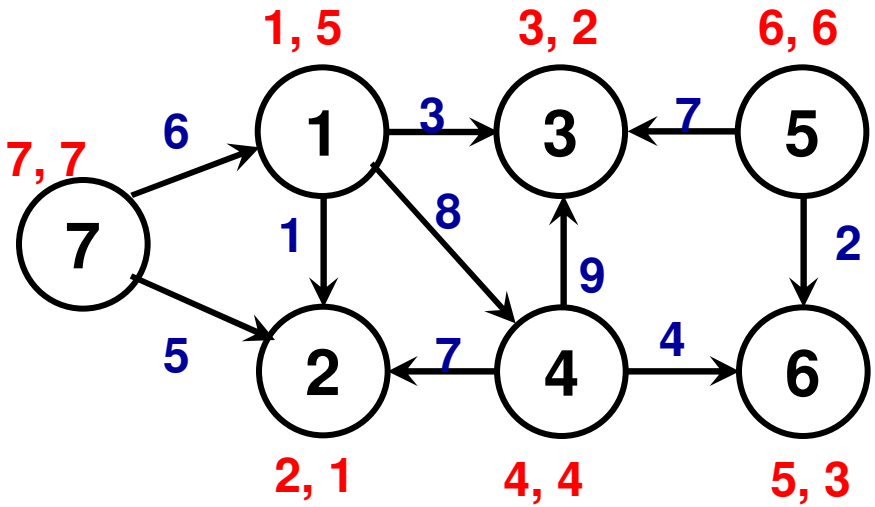
Longest Paths in a DAG: Example 1

Topological Sort
7, 5, 1, 4, 6, 3, 2

Let vertex '1' be the source



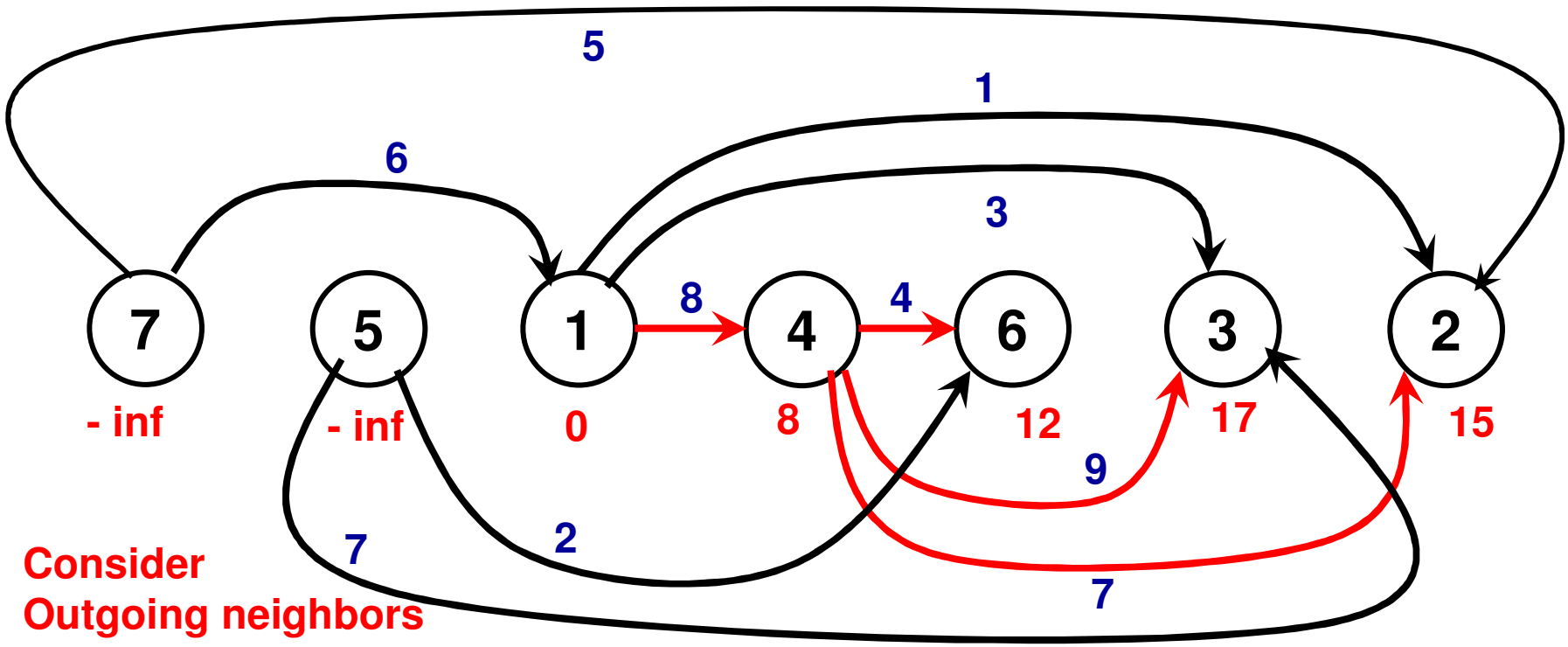
Consider Outgoing neighbors of Vertex '4'



Longest Paths in a DAG: Example 1

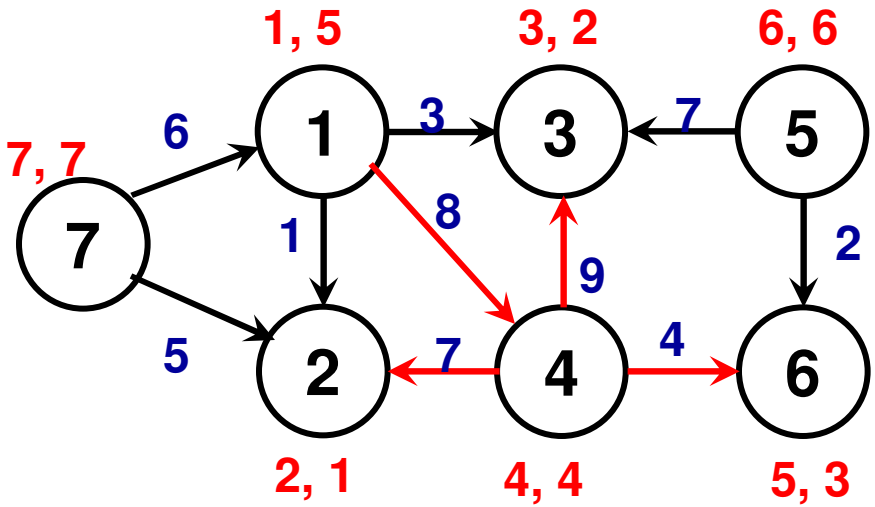
Topological Sort
7, 5, 1, 4, 6, 3, 2

Let vertex '1' be the source



Consider
Outgoing neighbors
of Vertices '6', '3', '2' (no change)

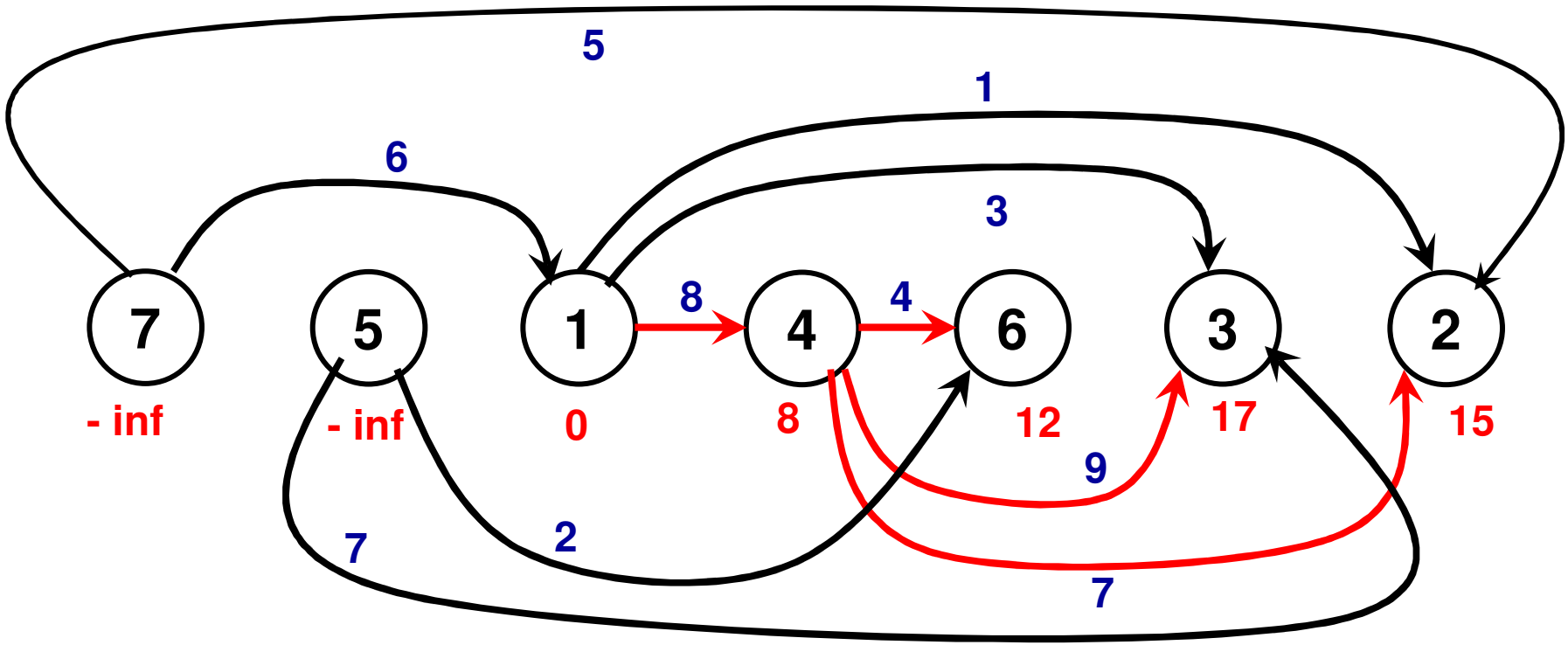
Longest Paths in a DAG: Example 1

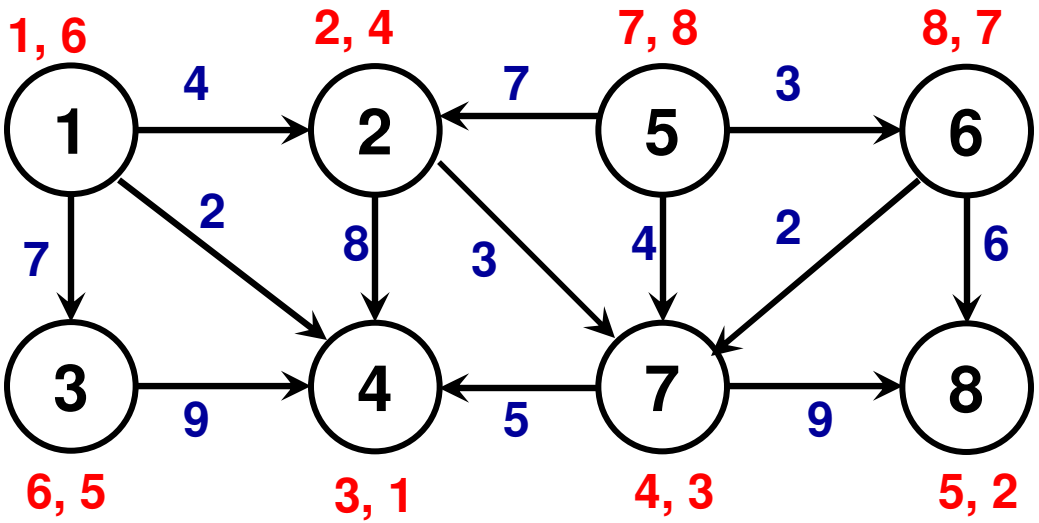


Final Longest Paths

Topological Sort
7, 5, 1, 4, 6, 3, 2

Let vertex '1' be the source



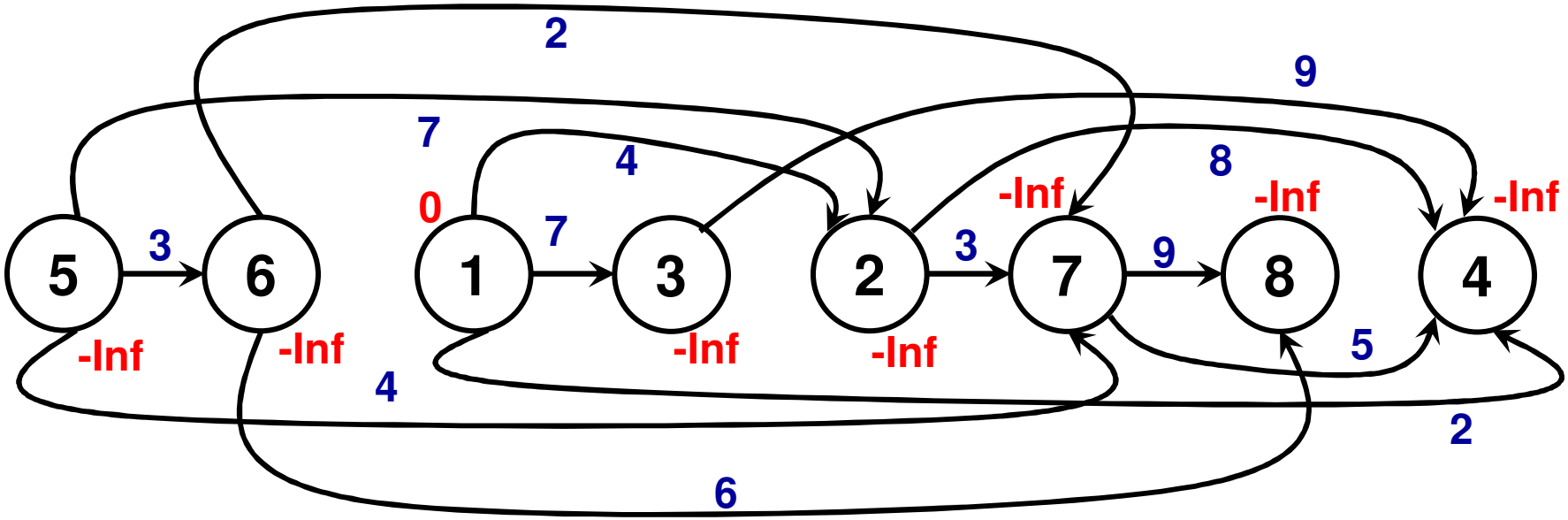


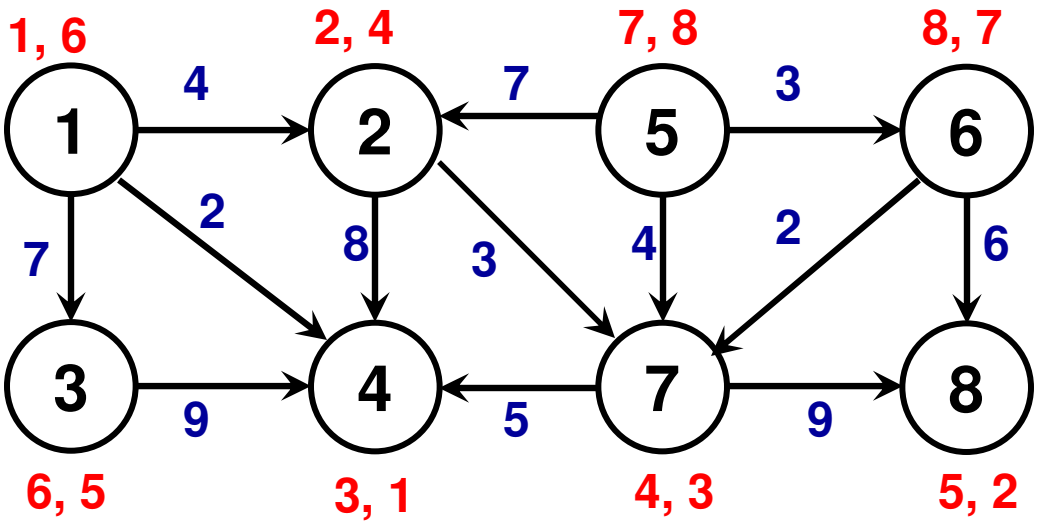
Longest Paths in a DAG: Example 2

Topological Sort
5, 6, 1, 3, 2, 7, 8, 4

Let vertex '1' be the source

Initialization



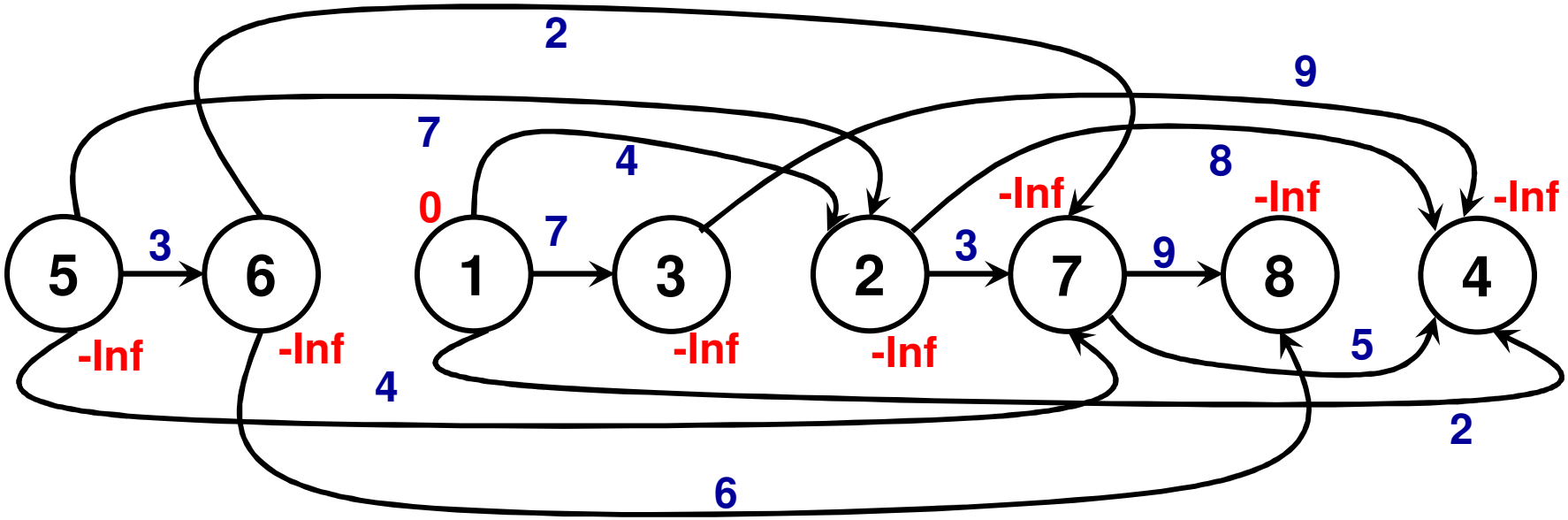


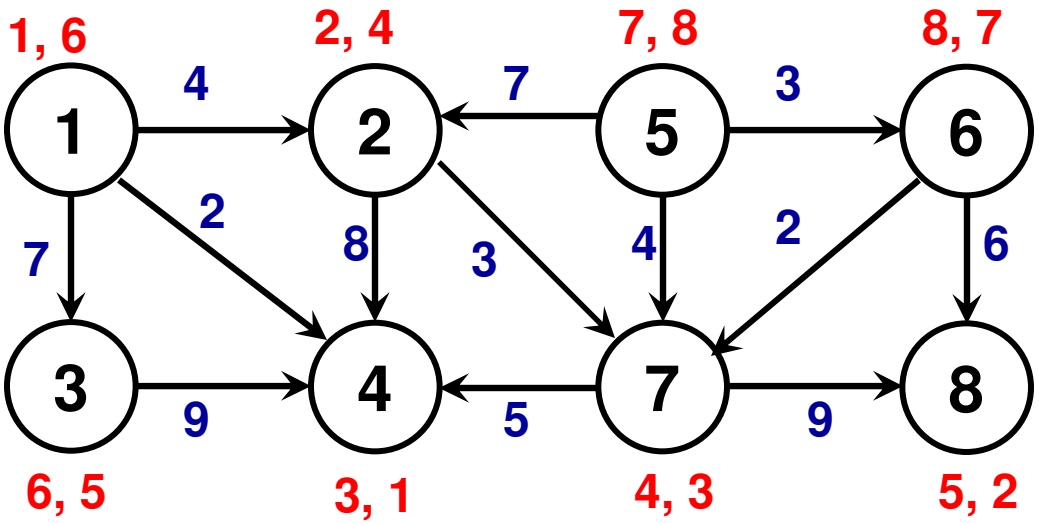
Longest Paths in a DAG: Example 2

Topological Sort
5, 6, 1, 3, 2, 7, 8, 4

Let vertex '1' be the source

Considering vertices '5' and '6' (no changes)



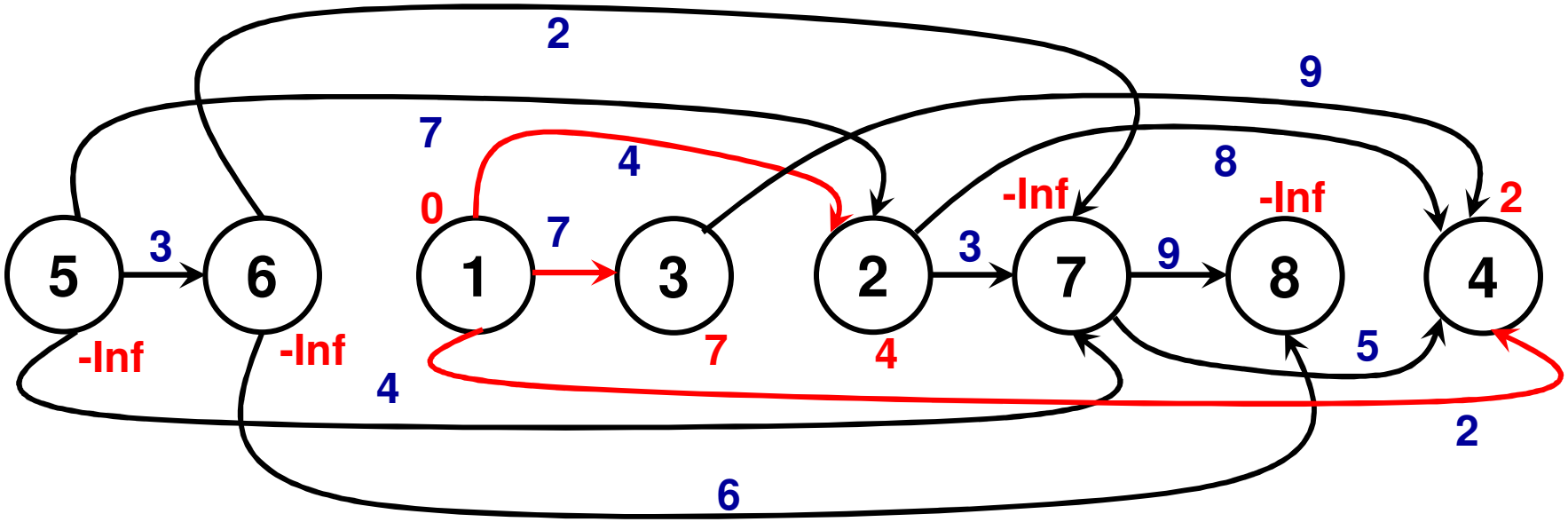


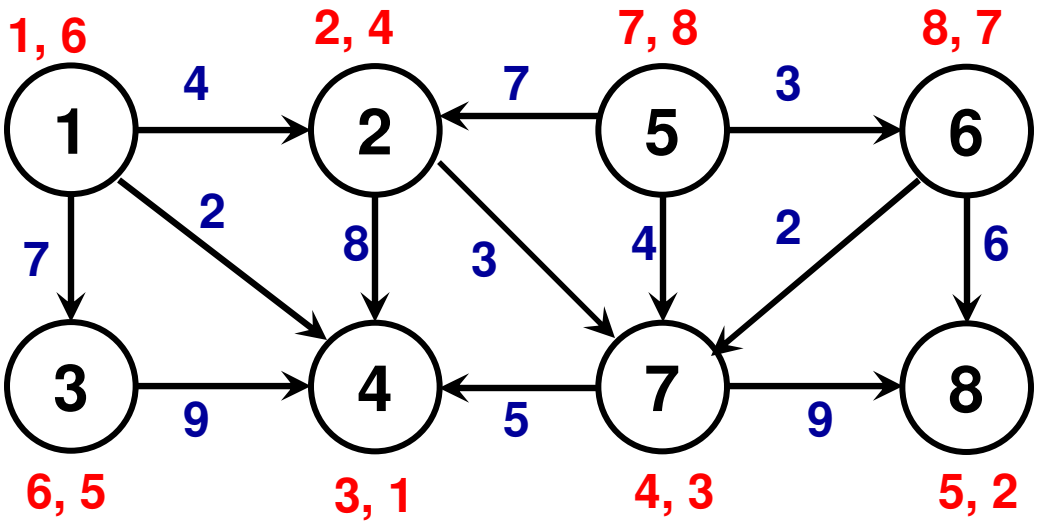
Longest Paths in a DAG: Example 2

Topological Sort
5, 6, 1, 3, 2, 7, 8, 4

Let vertex '1' be the source

Considering vertex '1'



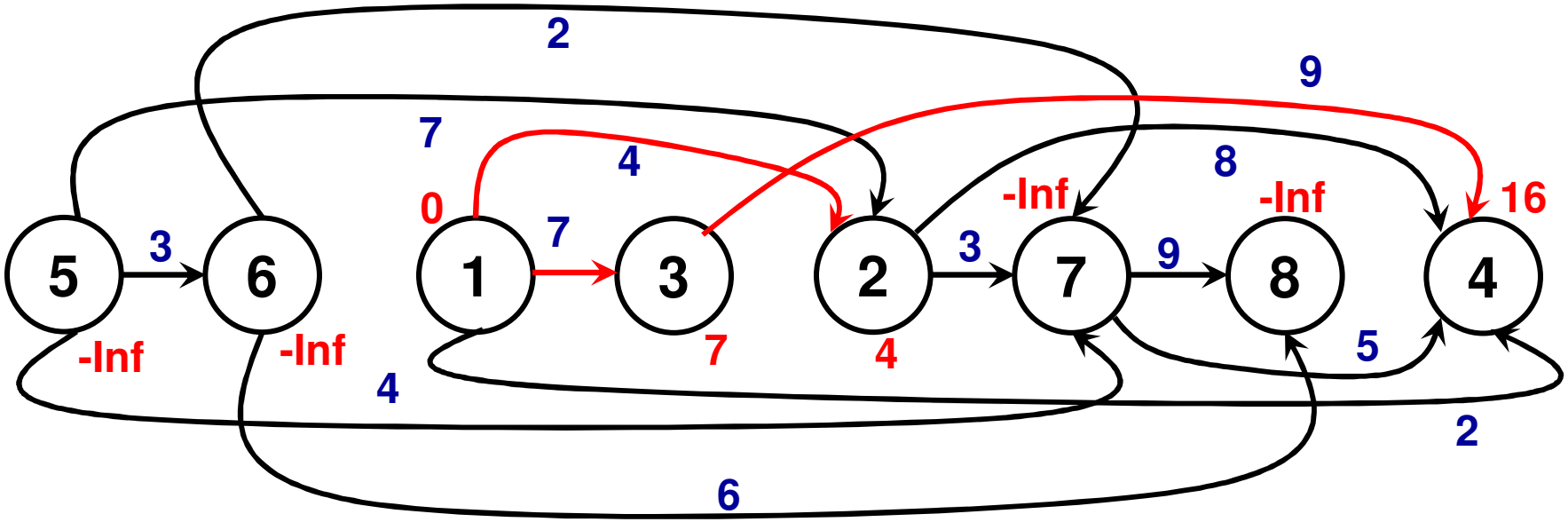


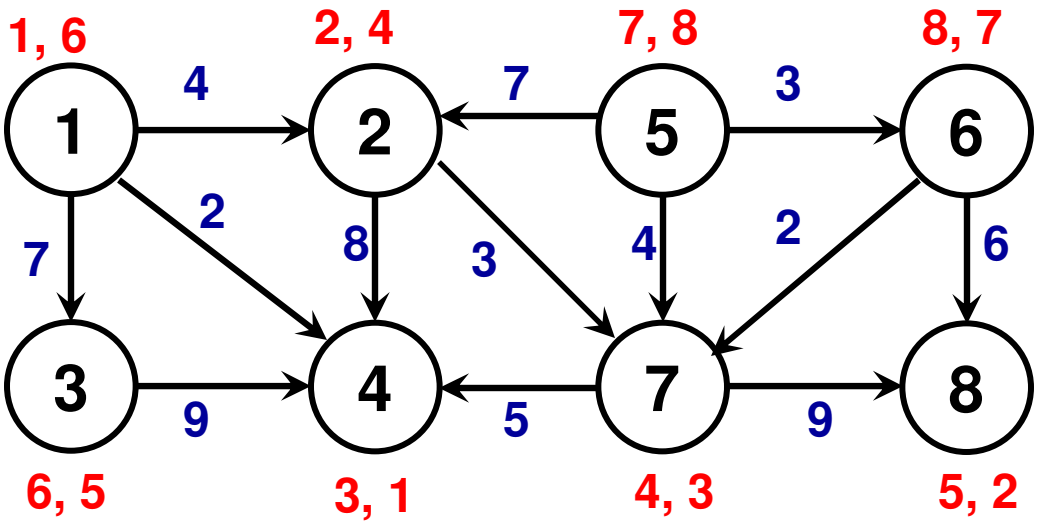
Longest Paths in a DAG: Example 2

Topological Sort
5, 6, 1, 3, 2, 7, 8, 4

Let vertex '1' be the source

Considering vertex '3'



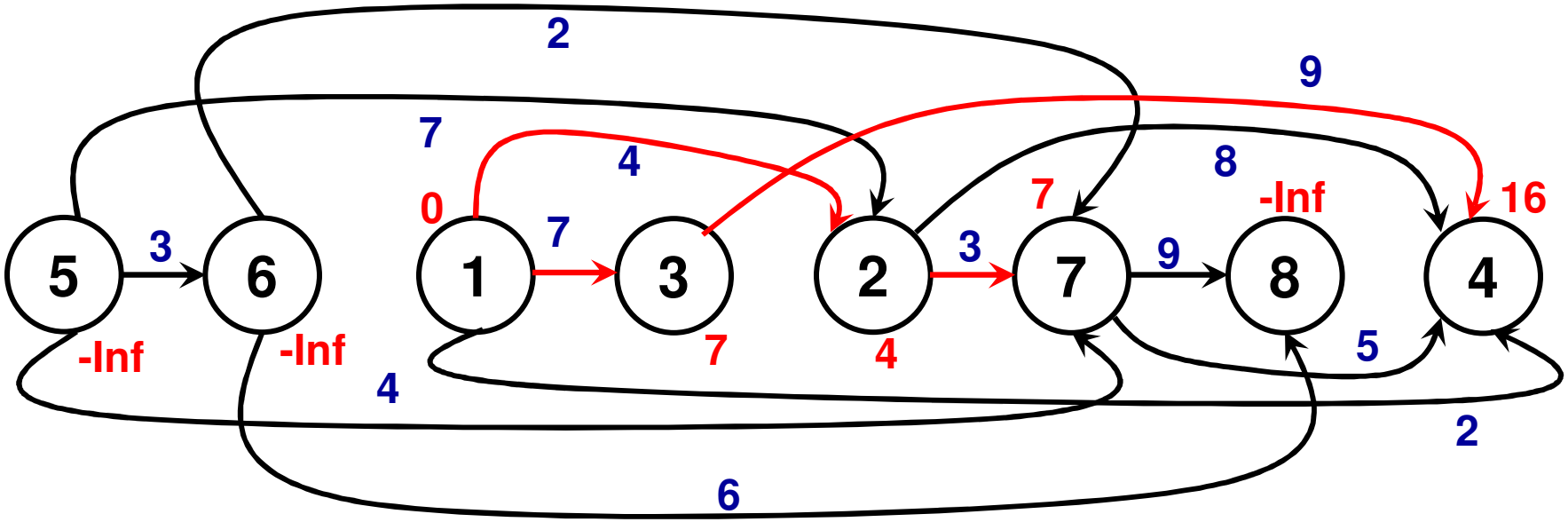


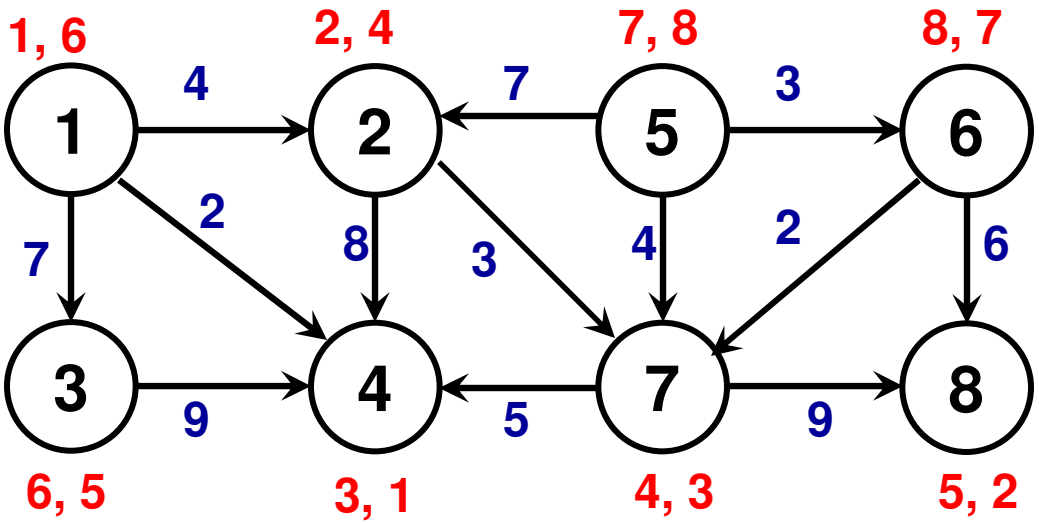
Longest Paths in a DAG: Example 2

Topological Sort
5, 6, 1, 3, 2, 7, 8, 4

Let vertex '1' be the source

Considering vertex '2'



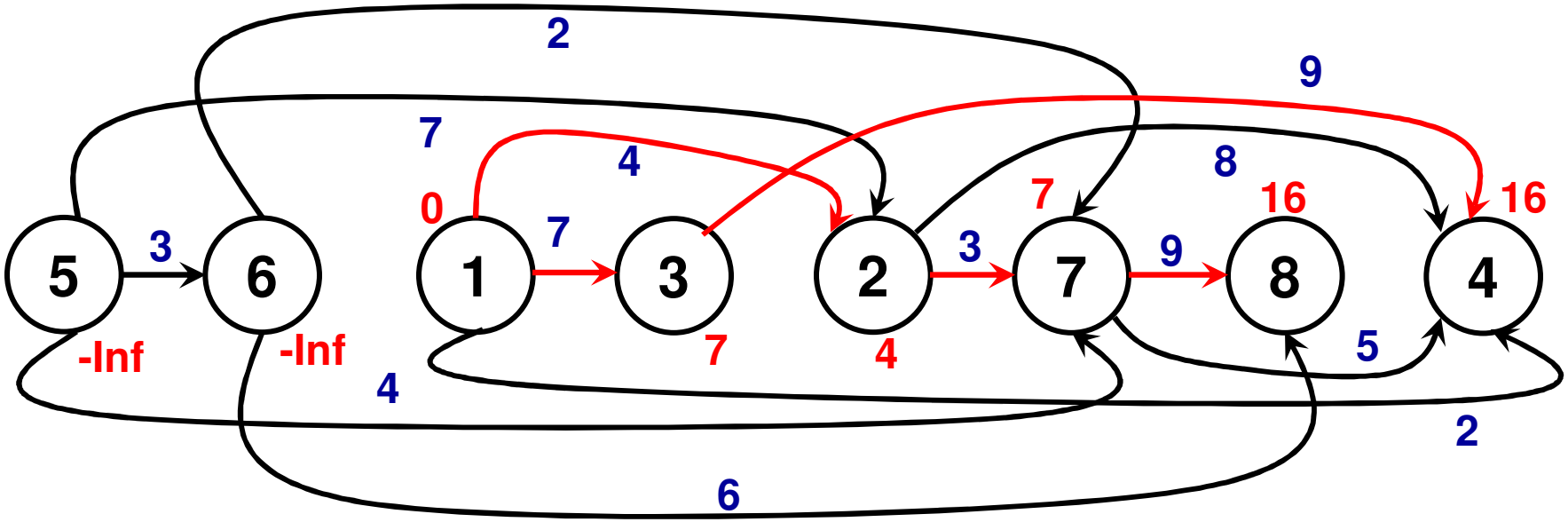


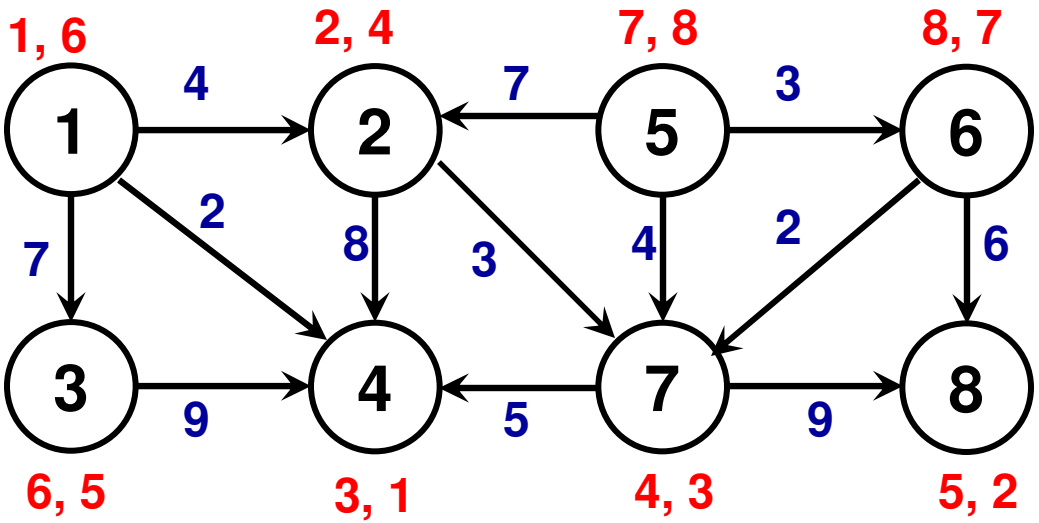
Longest Paths in a DAG: Example 2

Topological Sort
5, 6, 1, 3, 2, 7, 8, 4

Let vertex '1' be the source

Considering vertex '7'



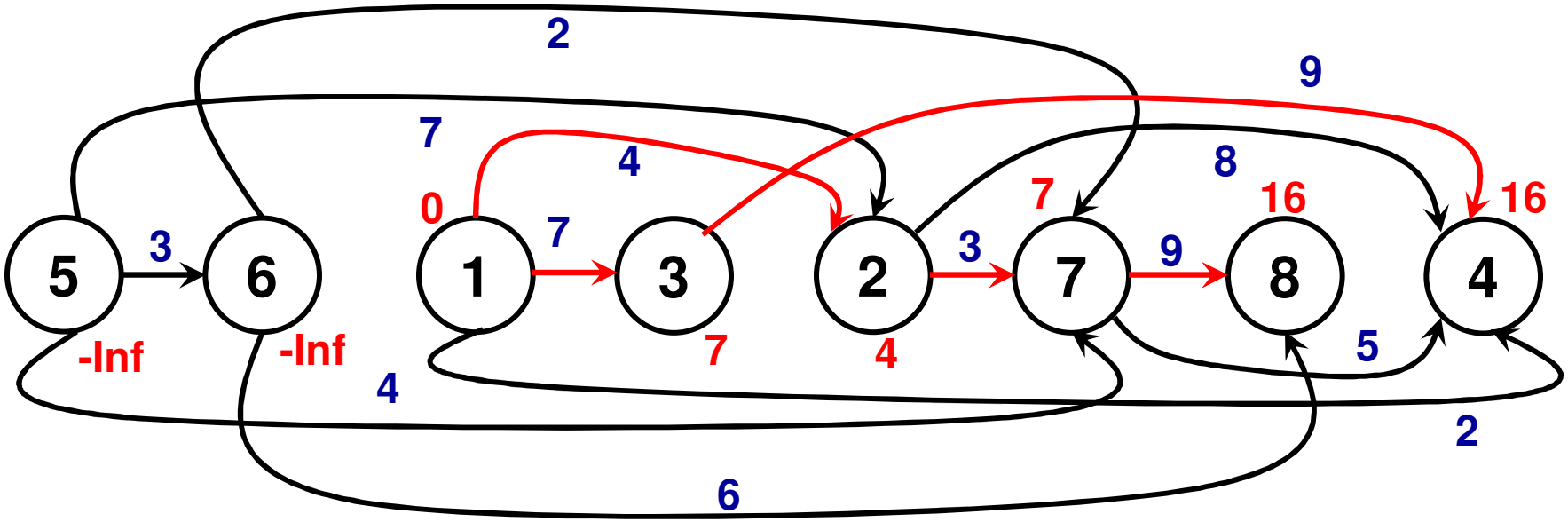


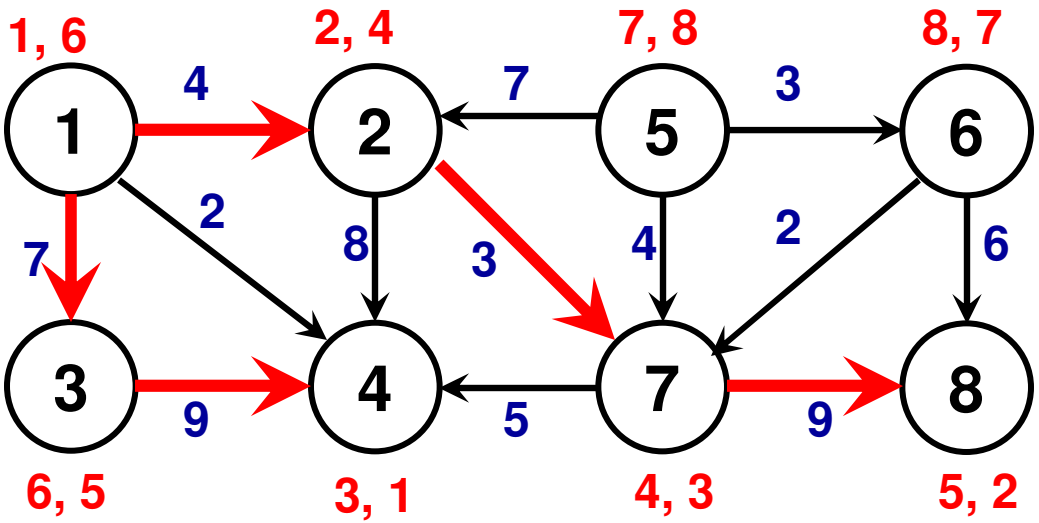
Longest Paths in a DAG: Example 2

Topological Sort
5, 6, 1, 3, 2, 7, 8, 4

Let vertex '1' be the source

Considering vertices '8' and '4' (no changes)





Longest Paths in a DAG: Example 2

Topological Sort
5, 6, 1, 3, 2, 7, 8, 4

Let vertex '1' be the source

Final Longest Paths