

Module 5

Graph Algorithms

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

5.1 Graph Traversal Algorithms

Depth First Search (DFS)

- Visits graph's vertices (also called nodes) by always moving away from last visited vertex to unvisited one, backtracks if there is no adjacent unvisited vertex.
- Break any tie to visit an adjacent vertex, by visiting the vertex with the lowest ID or the lowest alphabet (label).
- Uses a stack
 - a vertex is pushed onto the stack when it's visited for the first time
 - a vertex is popped off the stack when it becomes a dead end, i.e., when there is no adjacent unvisited vertex
- “Redraws” graph in tree-like fashion (with tree edges and back edges for undirected graph):
 - Whenever a new unvisited vertex is reached for the first time, it is attached as a child to the vertex from which it is being reached. Such an edge is called a tree edge.
 - While exploring the neighbors of a vertex, if the algorithm encounters an edge leading to a previously visited vertex other than its immediate predecessor (i.e., its parent in the tree), such an edge is called a back edge.
 - The leaf nodes have no children; the root node and other intermediate nodes have one more child.

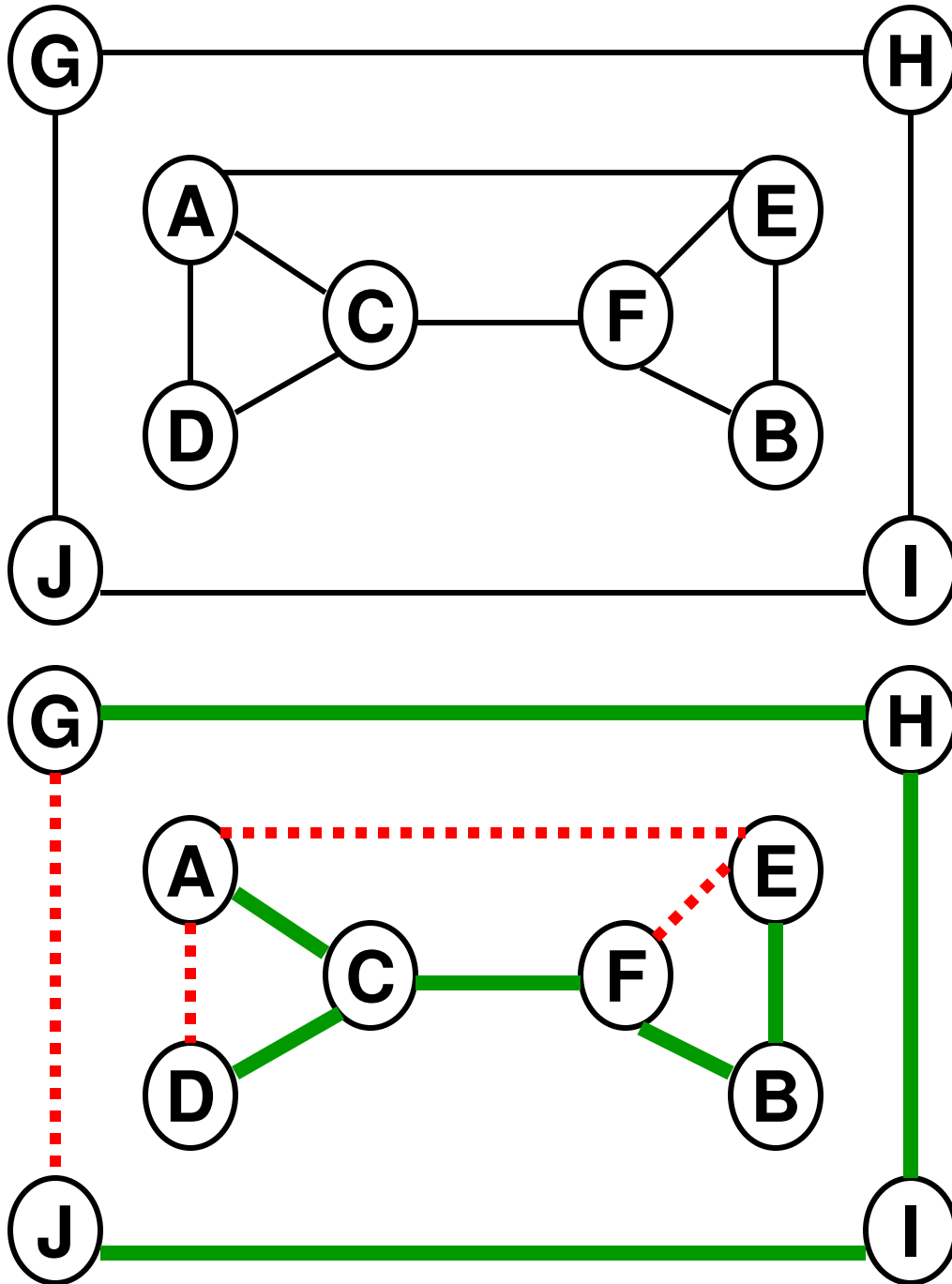
Pseudo Code of DFS

ALGORITHM *DFS*(*G*)

```
//Implements a depth-first search traversal of a given graph
//Input: Graph  $G = \langle V, E \rangle$ 
//Output: Graph  $G$  with its vertices marked with consecutive integers
//      in the order they are first encountered by the DFS traversal
mark each vertex in  $V$  with 0 as a mark of being “unvisited”
count  $\leftarrow 0$ 
for each vertex  $v$  in  $V$  do
    if  $v$  is marked with 0
        dfs( $v$ )
```

dfs(v)

```
//visits recursively all the unvisited vertices connected to vertex  $v$ 
//by a path and numbers them in the order they are encountered
//via global variable count
count  $\leftarrow$  count + 1; mark  $v$  with count
for each vertex  $w$  in  $V$  adjacent to  $v$  do
    if  $w$  is marked with 0
        dfs( $w$ )
```



Example 1: DFS

$J_{3, 1}$
 $I_{2, 5}$
 $H_{1, 6}$
 $G_{7, 10}$
 $E_{6, 2}$
 $B_{5, 3}$
 $F_{4, 4}$
 $D_{3, 1}$
 $C_{2, 5}$
 $A_{1, 6}$

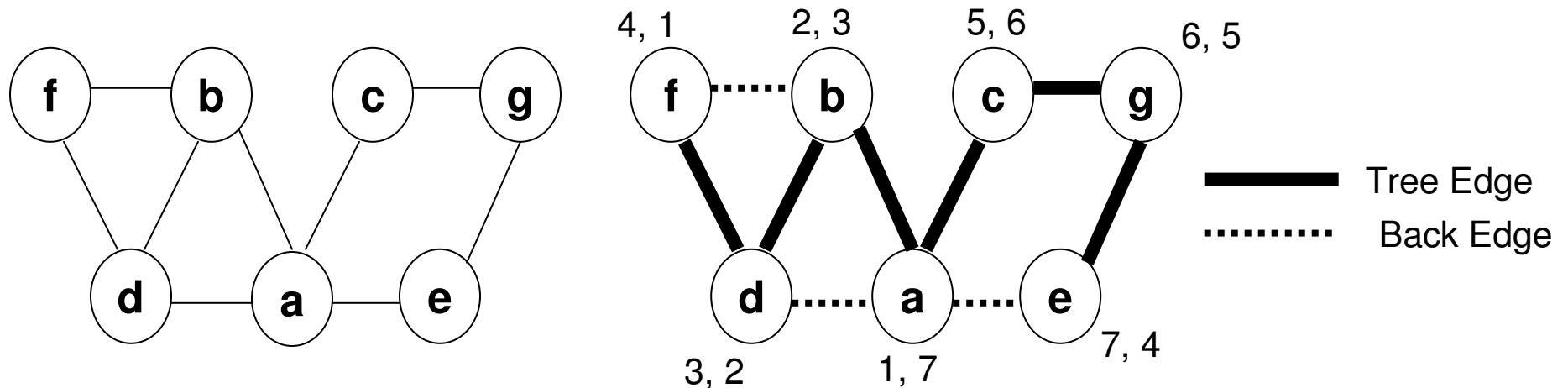
 Tree Edge
 Back Edge

Source: Figure 3.10: Levitin, 3rd Edition: Introduction to the Design and Analysis of Algorithms, 2012.

DFS

- DFS can be implemented with graphs represented as:
 - adjacency matrices: $\Theta(V^2)$; adjacency lists: $\Theta(|V|+|E|)$
- Yields two distinct ordering of vertices:
 - order in which vertices are first encountered (pushed onto stack)
 - order in which vertices become dead-ends (popped off stack)
- Applications:
 - checking connectivity, finding connected components
 - The set of vertices that we can visit through DFS, starting from a particular vertex in the set constitute a connected component.
 - If a graph has only one connected component, we need to run DFS only once and it returns a tree; otherwise, the graph has more than one connected component and we determine a forest – comprising of trees for each component.
 - checking for cycles (a DFS run on an undirected graph returns a back edge)
 - finding articulation points and bi-connected components
 - An articulation point of a connected component is a vertex that when removed disconnects the component.
 - A graph is said to have bi-connected components if none of its components have an articulation point.

Example 2: DFS

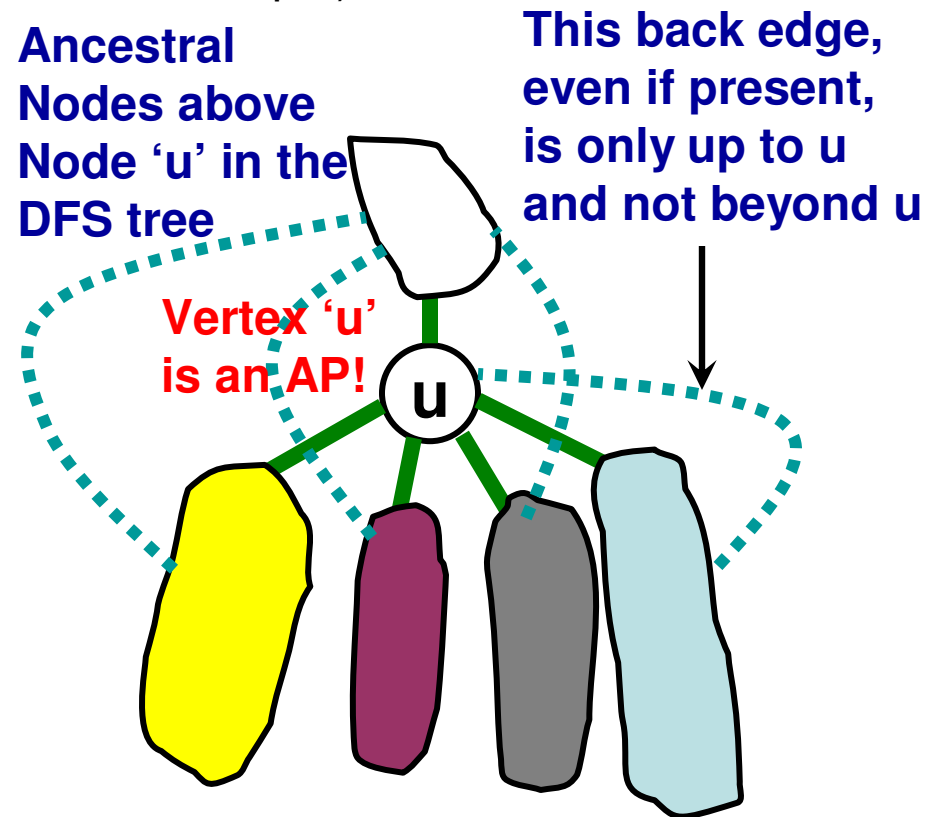
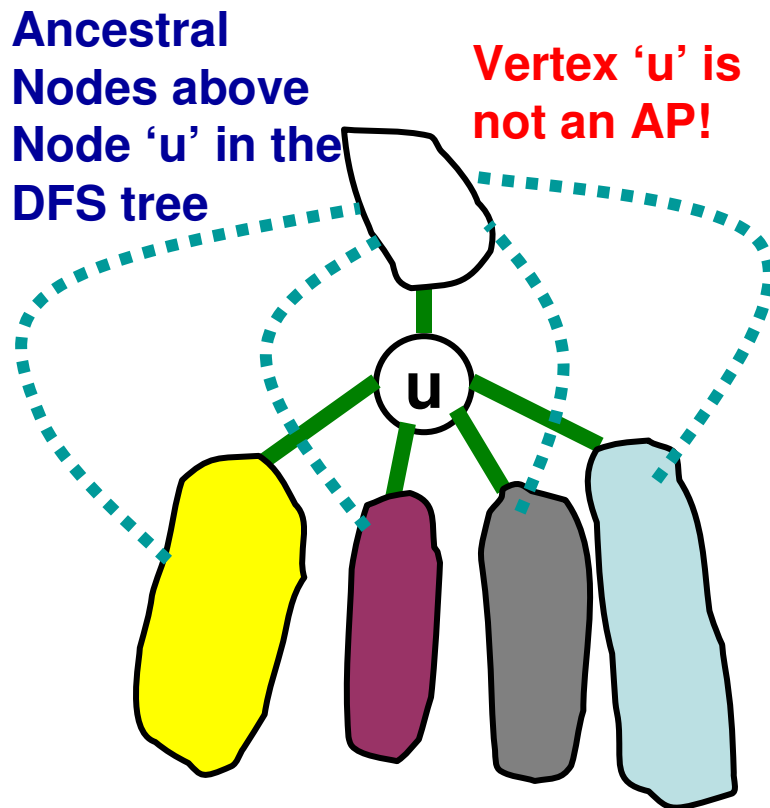


- Notes on Articulation Point

- 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.

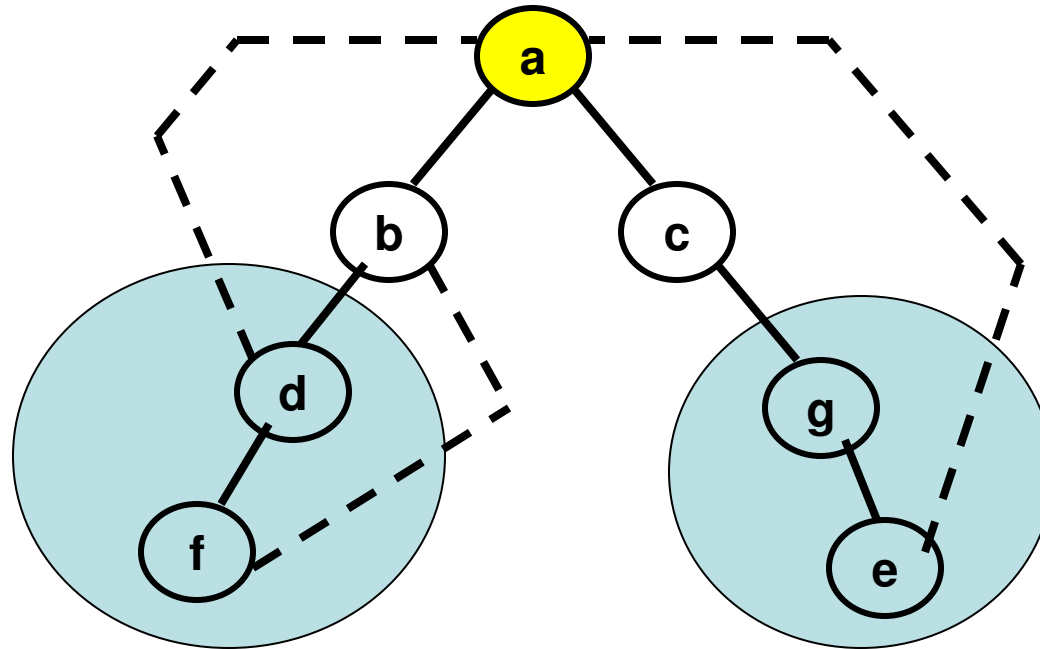
Deciding whether an Internal Node is an Articulation Point or not

- 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).



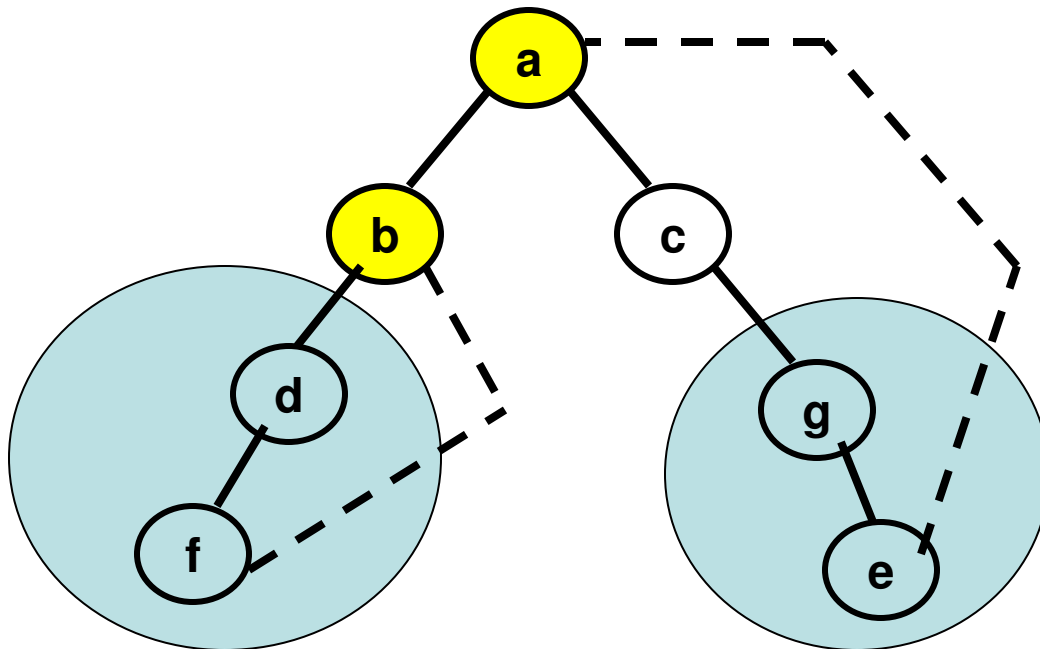
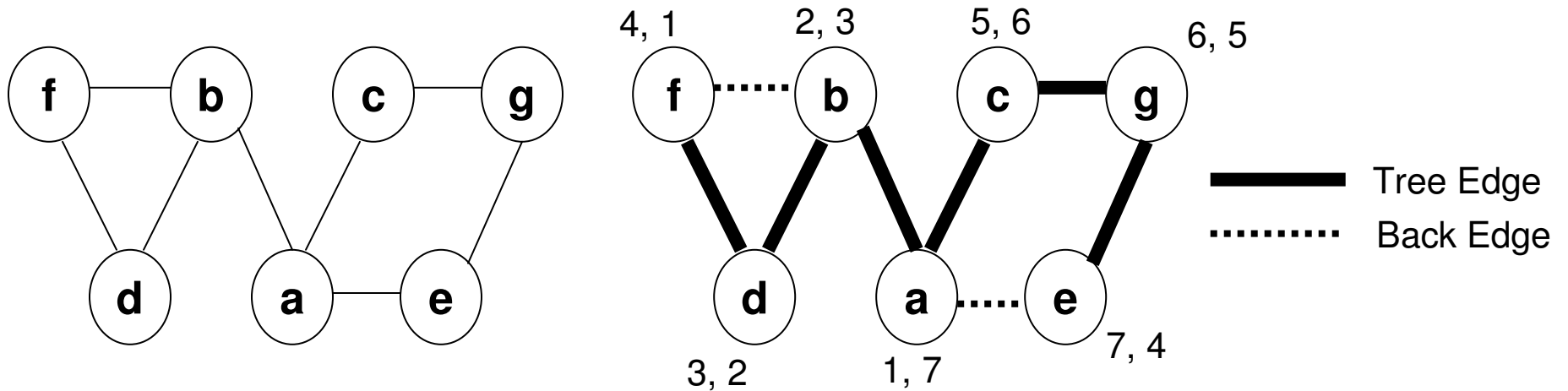
DFS: Articulation Points

Based on
Example 2



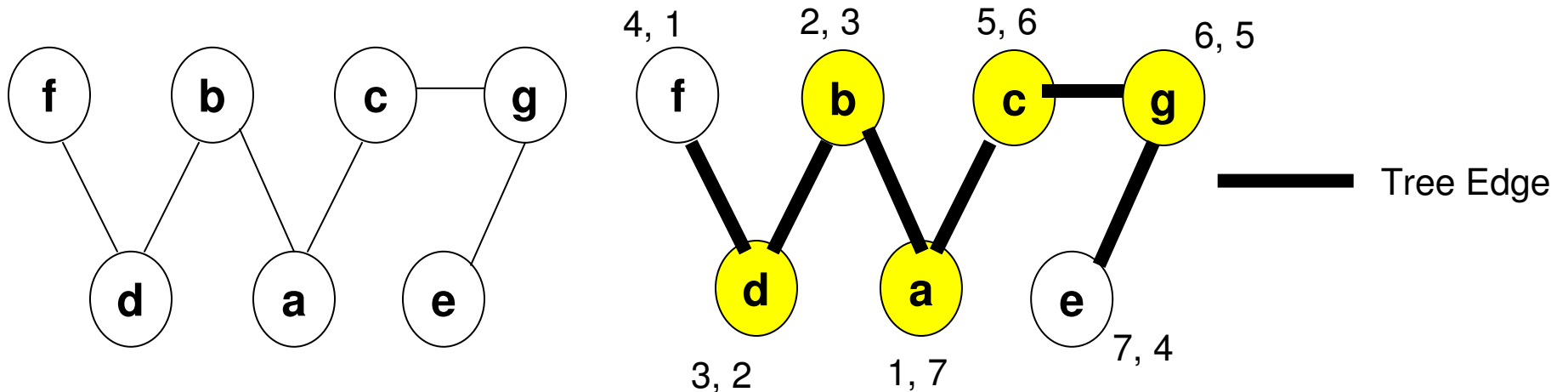
- In the above graph, vertex 'a' is the only articulation point.
- Vertices 'e' and 'f' are leaf nodes.
- Vertices 'b' and 'c' are candidates for articulation points. But, they cannot become articulation point, because there is a back edge from the only subtree rooted at their child nodes ('d' and 'g' respectively) that have a back edge to 'a'.
- By the same argument, vertices 'd' and 'g' are not articulation points, because they have only child node (f and e respectively); each of these child nodes are connected to a higher level vertex (b and a respectively) through a back edge.

Example 3: DFS and Articulation Points



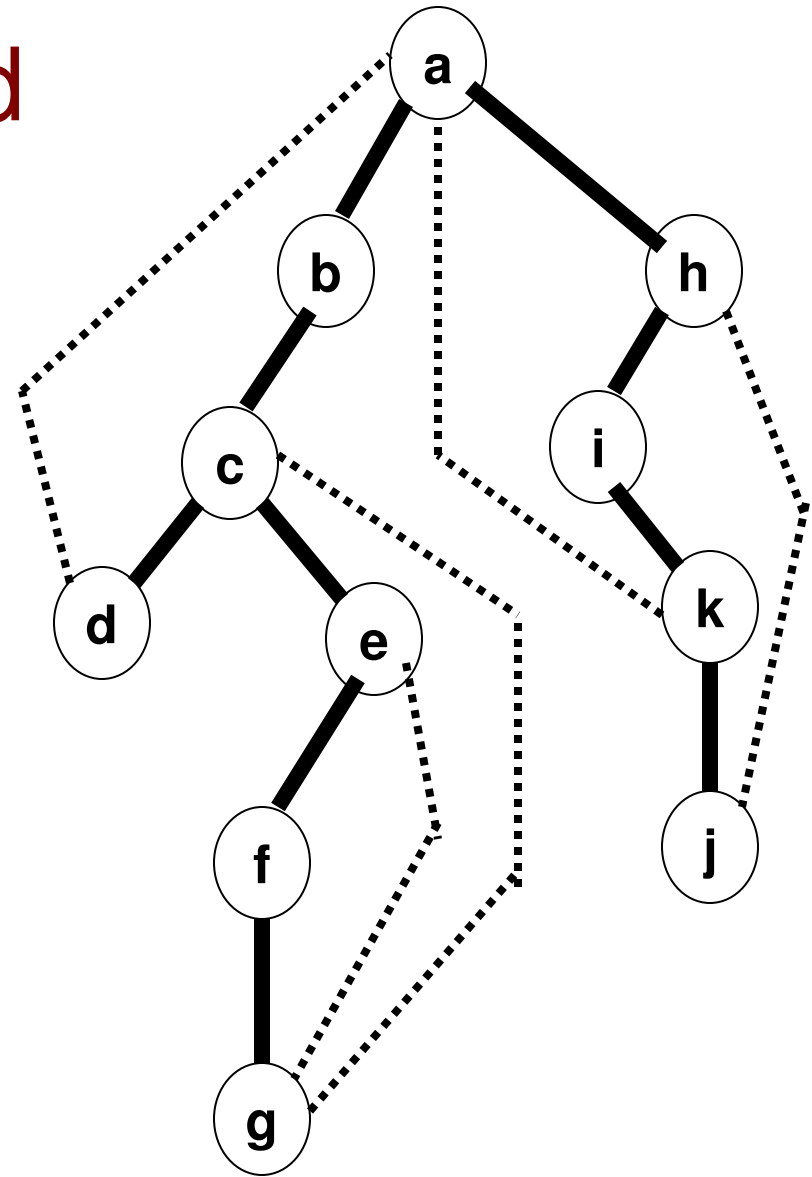
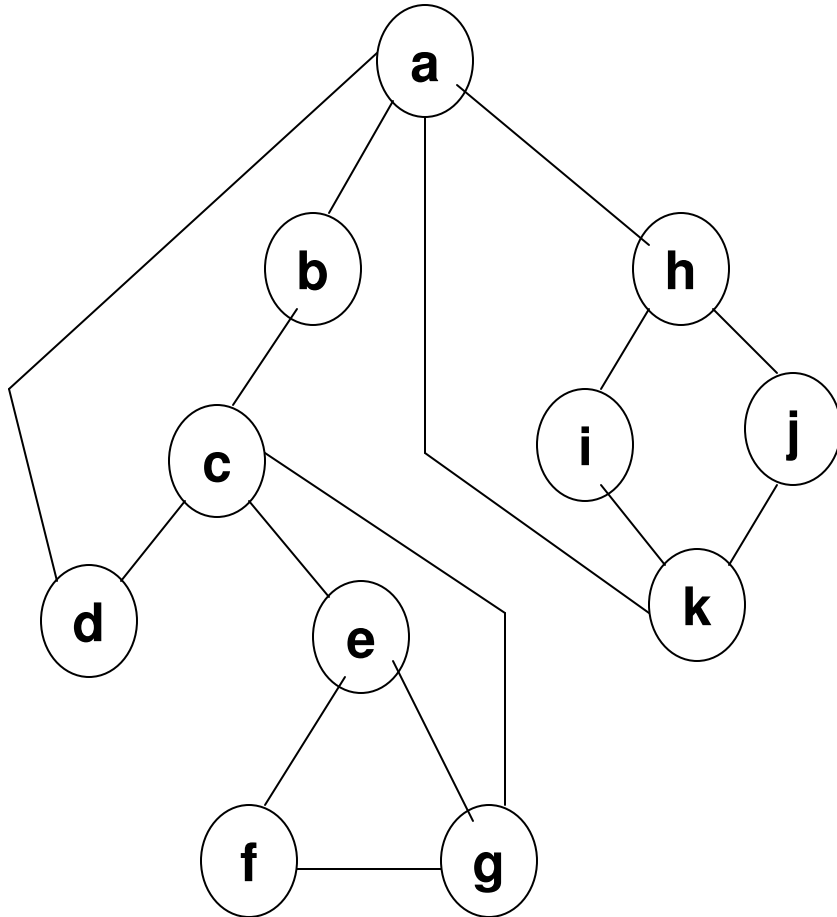
- In the above new graph (different from the previous example: note edge $a - e$ and $b - f$ are added back; but $a - d$ is missing):
 - Vertices 'a' and 'b' are articulation points
 - Vertex 'c' is not an articulation point

Example 4: DFS and Articulation Points



- In the above new graph (different from the previous example: note edges $b - f$, $a - d$ and $a - e$ are missing), vertices 'a', 'b', 'c', 'd' and 'g' are articulation points, because:
 - Vertex 'a' is the root node of the DFS tree and it has more than one child node
 - Vertex 'b' is an intermediate node; it has one sub tree rooted at its child node (d) that does not have any node, including 'd', to climb higher than 'b'. So, vertex 'b' is an articulation point.
 - Vertex 'c' is also an articulation point, by the same argument as above – this time, applied to the sub tree rooted at child node 'g'.
 - Vertices 'd' and 'g' are articulation points; because, they have one child node ('f' and 'e' respectively) that are not connected to any other vertex higher than 'd' and 'g' respectively.

Example 5: DFS and Articulation Points



DFS TREE

Identification of the Articulation Points of the Graph in Example 5

1) **Root Vertex 'a'** has more than one child; so, it is an articulation point.

2) Vertices '**d**', '**g**' and '**j**' are leaf nodes

3) Vertex '**b**' is not an articulation point because the only sub tree rooted at its child node 'c' has a back edge to a vertex higher than 'b' (in this case to the root vertex 'a')

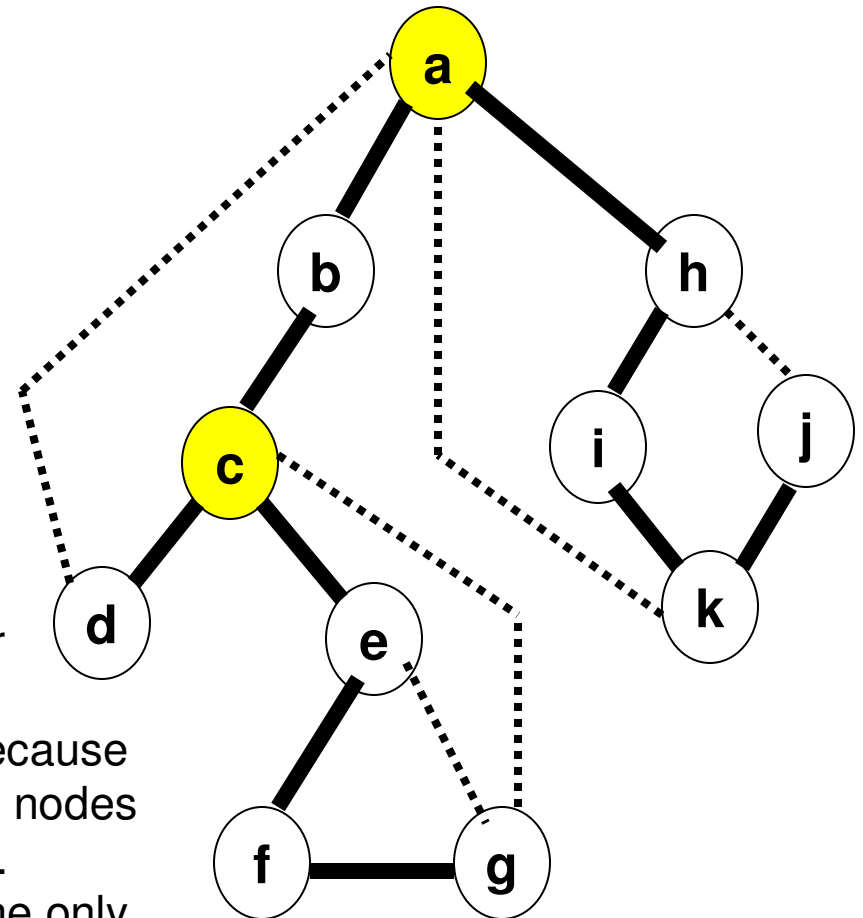
4) **Vertex 'c' is an articulation point.** One of its child vertex 'd' does not have any sub tree rooted at it. The other vertex 'e' has a sub tree rooted at it and this sub tree has no back edge higher up than 'c'.

5) By argument (4), it follows that vertex '**e**' is not an articulation point because the sub tree rooted at its child node 'f' has a back edge higher up than 'e' (to vertex 'c');

6) Vertices 'f' and 'k' are not articulation points because they have only one child node each and the child nodes are connected to a vertex higher above 'f' and 'k'.

7) Vertex 'i' is not an articulation point because the only sub tree rooted at its child has a back edge higher up (to vertices 'a' and 'h').

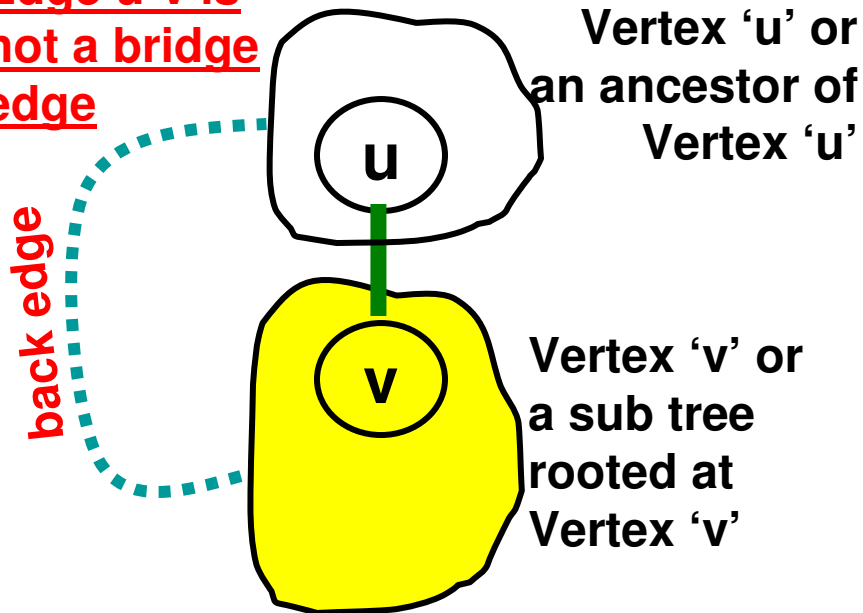
8) Vertex 'h' is not an articulation point because the only sub tree rooted at 'h' has a back edge higher up (to the root vertex 'a').



Bridge Edges in a Graph

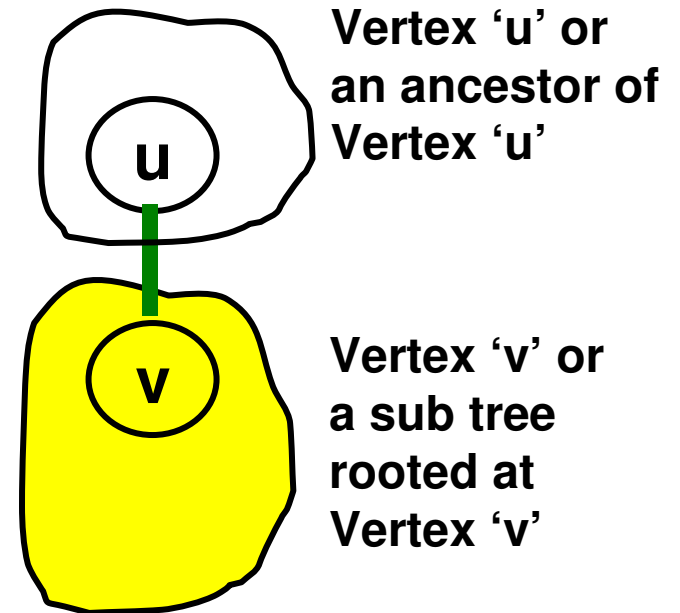
- An edge is a “bridge” edge in a connected graph if its removal would disconnect the vertices in the graph.
- We can identify the bridge edges of a graph using DFS
 - The “back” edges are not bridge edges
 - A tree edge $u - v$ is a bridge if there do not exist any back edge to reach u or an ancestor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

Edge $u-v$ is not a bridge edge

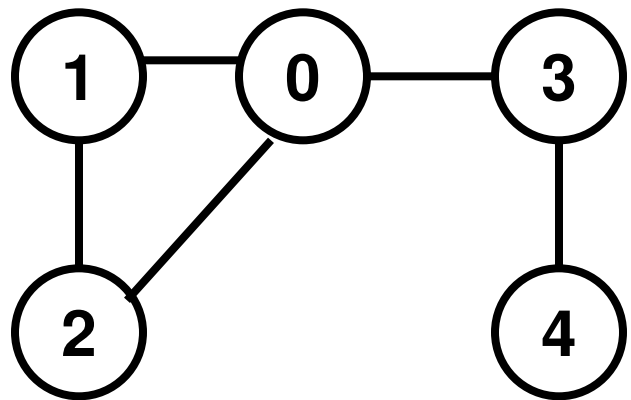


There is no back edge from vertex 'v' or a sub tree rooted at vertex 'v' to vertex 'u' or an ancestor of vertex 'u'

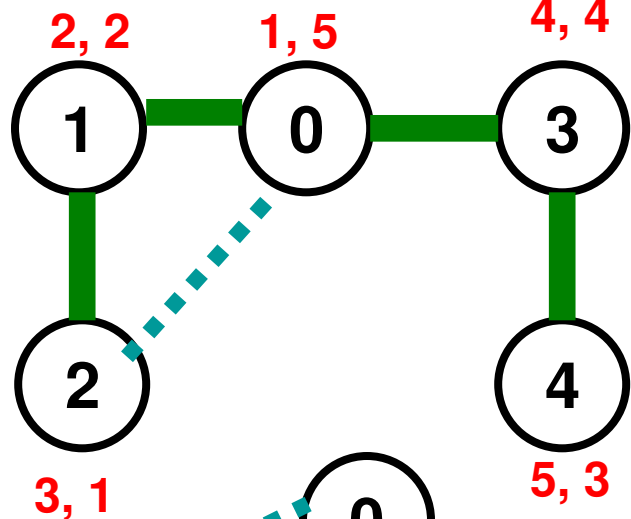
Edge $u-v$ is a bridge edge



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

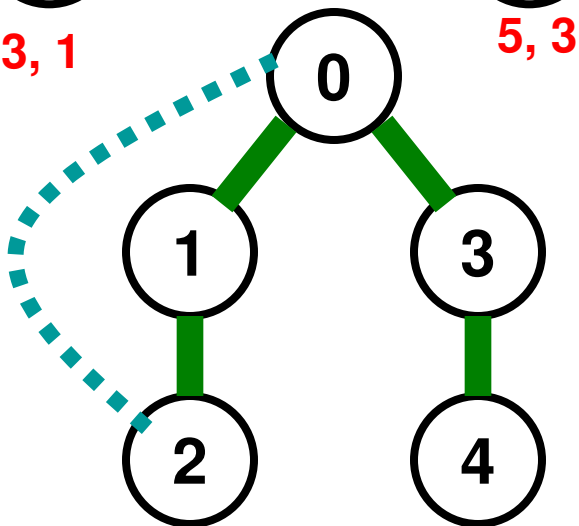


Edge 1 – 2 is not a bridge edge as there is a back edge from the sub tree rooted at vertex 2 to an ancestor of vertex 1

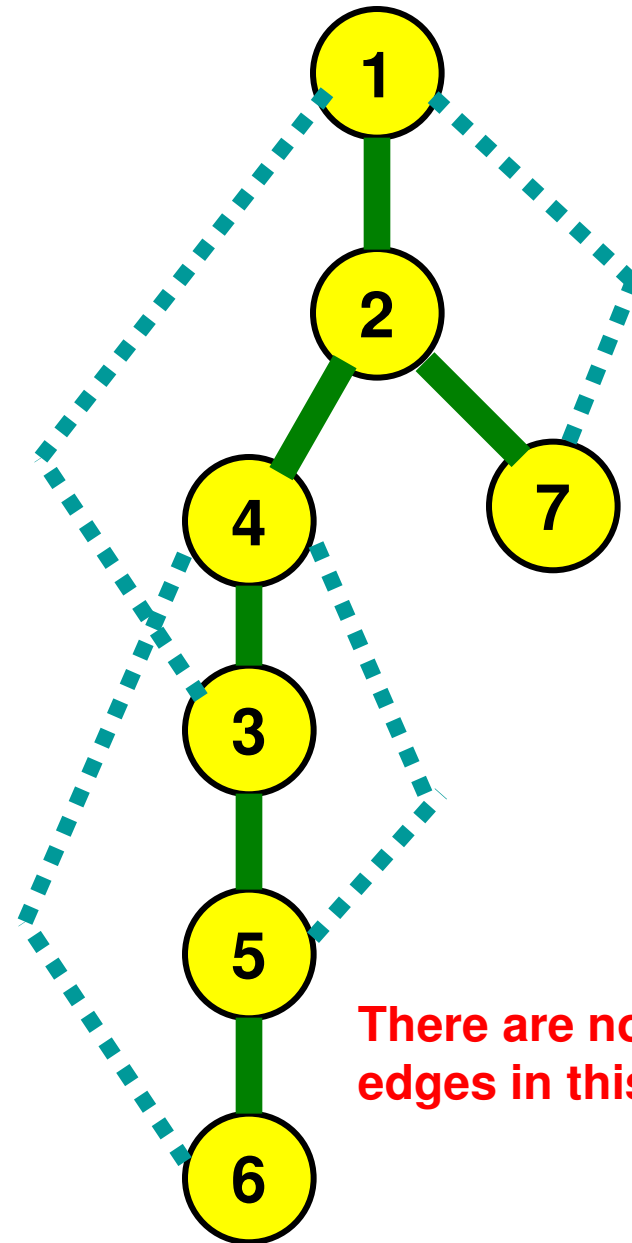
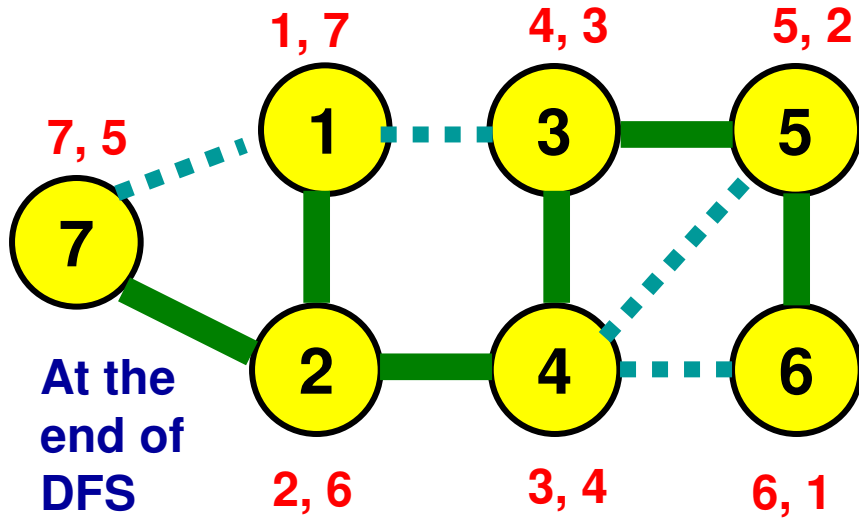
Edge 0 – 2 is not a bridge edge as it is a back edge

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.



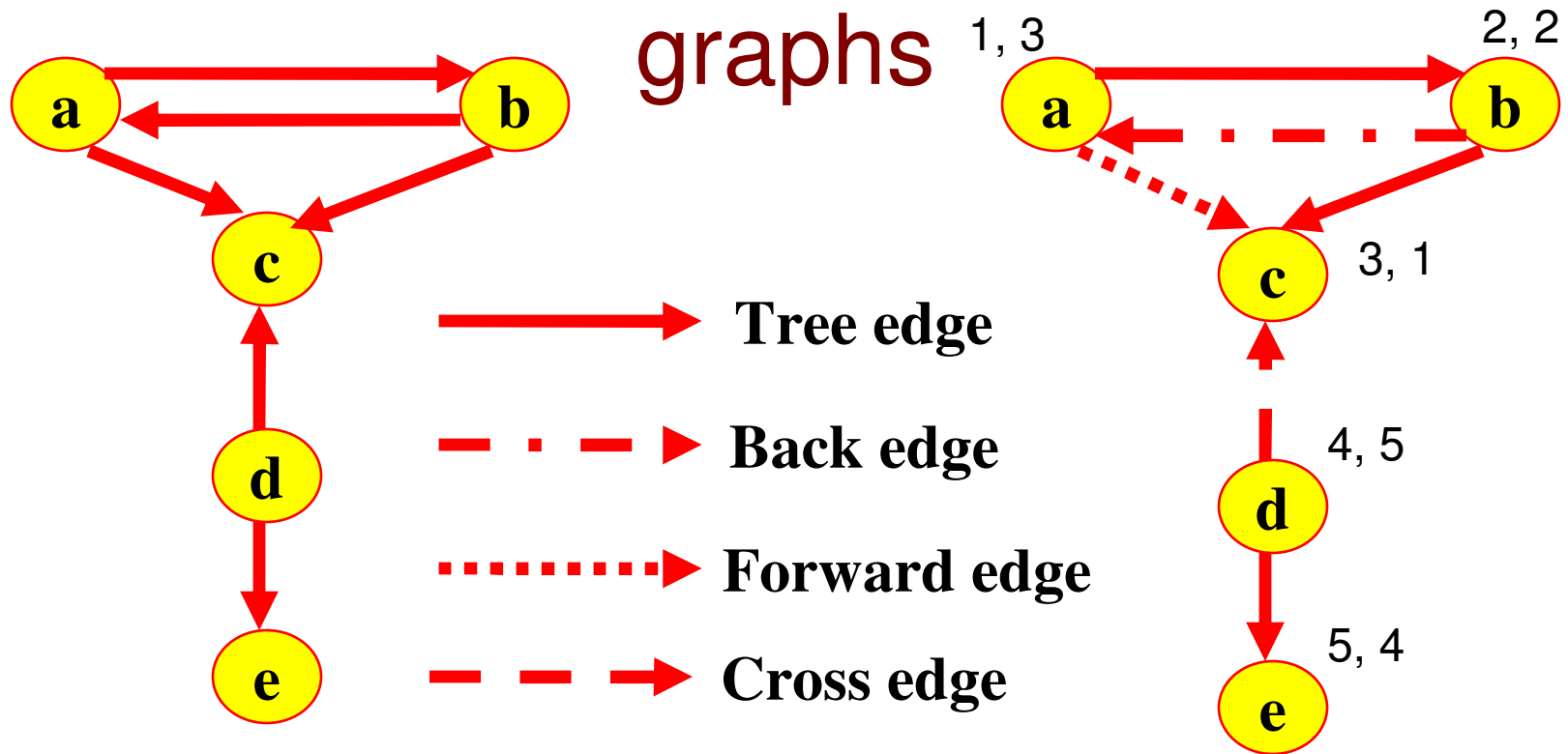
Bridge Edges in a Graph: Example 2



There are no bridge edges in this graph

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

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.

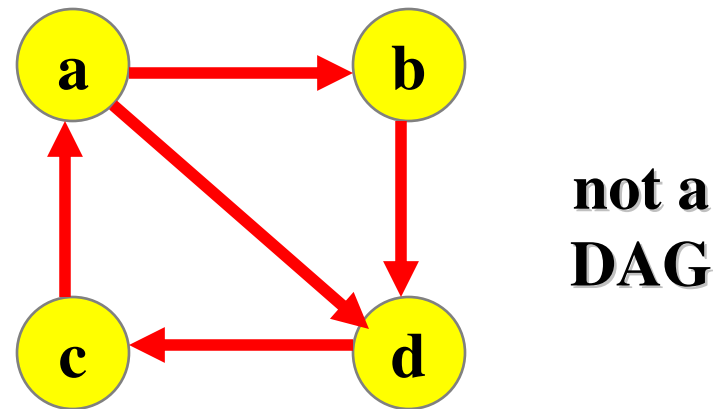
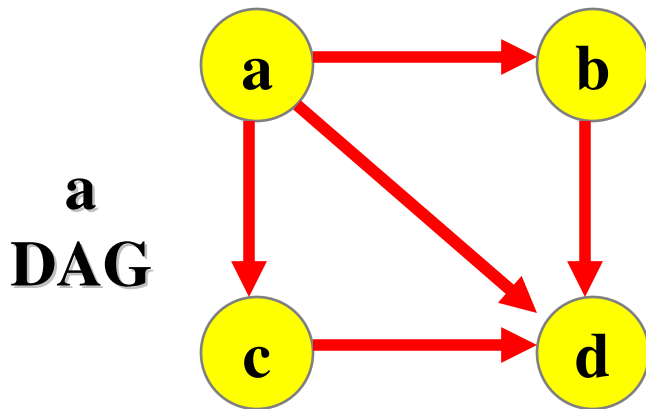
The two nodes do not have a parent-child relationship. The back and forward edges are in a single component (the DFS tree).

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

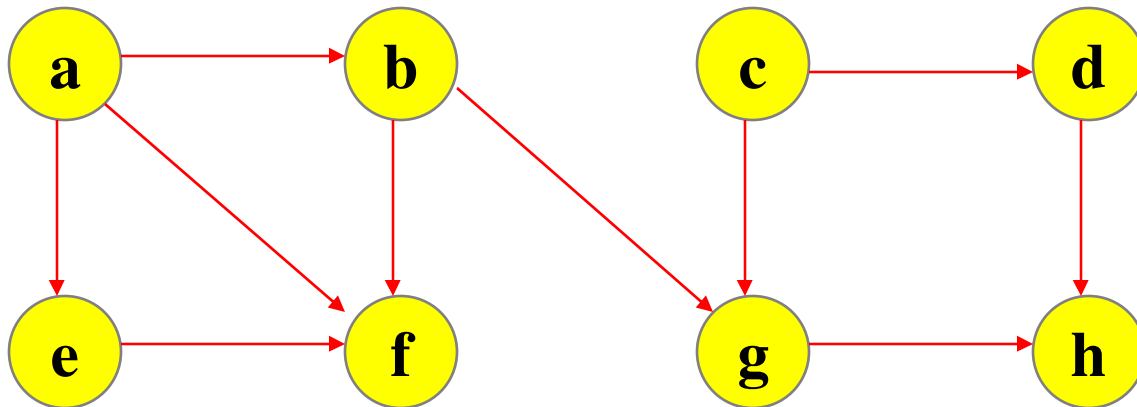
Directed Acyclic Graphs (DAG)

- A directed graph is a graph with directed edges between its vertices (e.g., $u \rightarrow v$).
- 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)

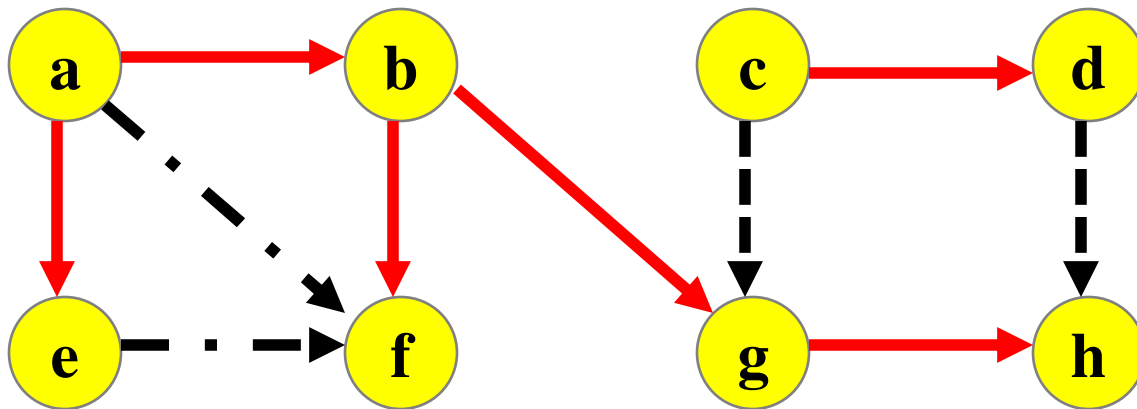


To test whether a directed graph is a DAG, run DFS on the directed graph. If a back edge is not encountered, then the directed graph is a DAG.

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

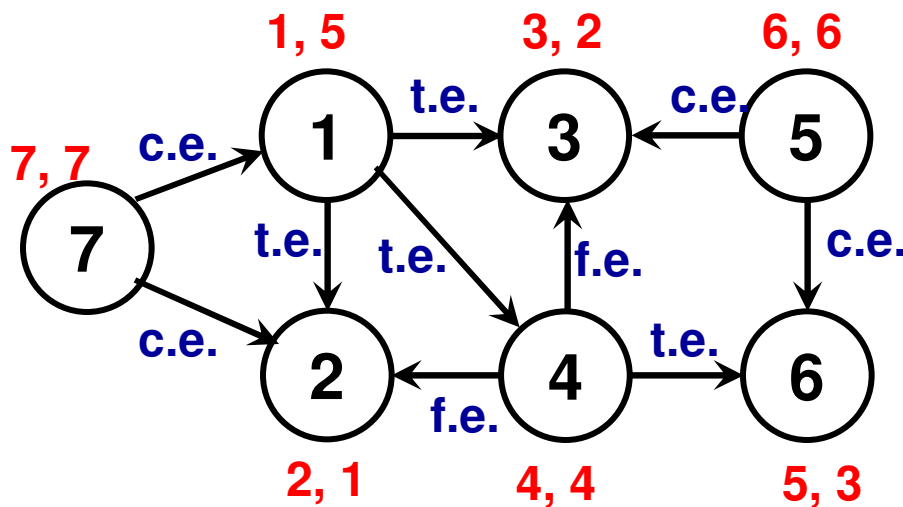
Reverse the order

Topological Sort

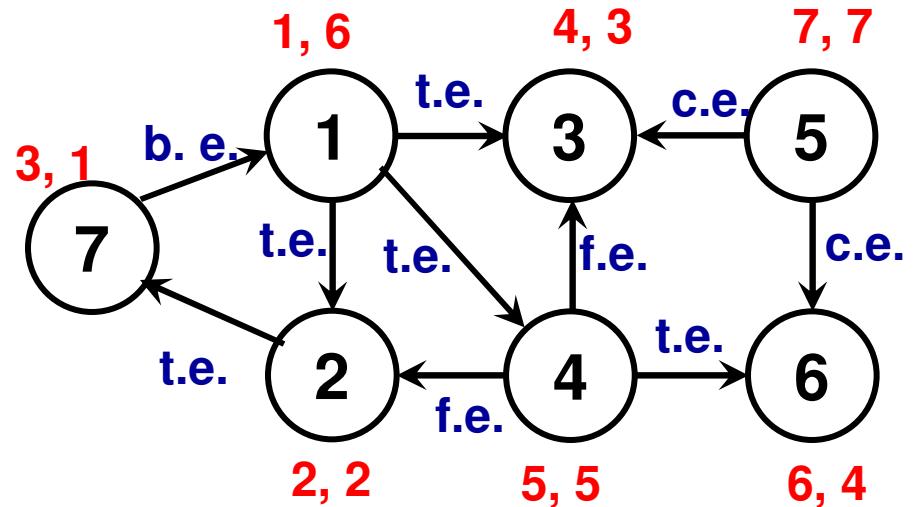
c d a e b g h f

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 $u \rightarrow v$, u appears somewhere before v ($u \dots v$) 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 \rightarrow Topological sort exists



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



Topological Sort does not exist!

Topological Sort

- Topological sort is an ordering of the vertices of a directed acyclic graph (DAG) – a directed graph (a.k.a. digraph) without cycles.
 - This implies if there is an edge $u \rightarrow v$ in the digraph, then u should be listed ahead of v in the topological sort: ... u ... v ...
 - Being a DAG is the necessary and sufficient condition to be able to do a topological sorting for a digraph.
 - **Proof for Necessary Condition:** If a digraph is not a DAG and lets say it has a topological sorting. Consider a cycle (in the digraph) comprising of vertices $u_1, u_2, u_3, \dots, u_k, u_1$. In other words, there is an edge from u_k to u_1 and there is a directed path to u_k from u_1 . So, it is not possible to decide whether u_1 should be ahead of u_k or after u_k in the topological sorting of the vertices of the digraph. Hence, there cannot be a topological sorting of the vertices of a digraph, if the digraph has even one cycle. **To be able to topologically sort the vertices of a digraph, the digraph has to first of all be a DAG. [Necessary Condition].** We will next prove that this is also the sufficient condition.

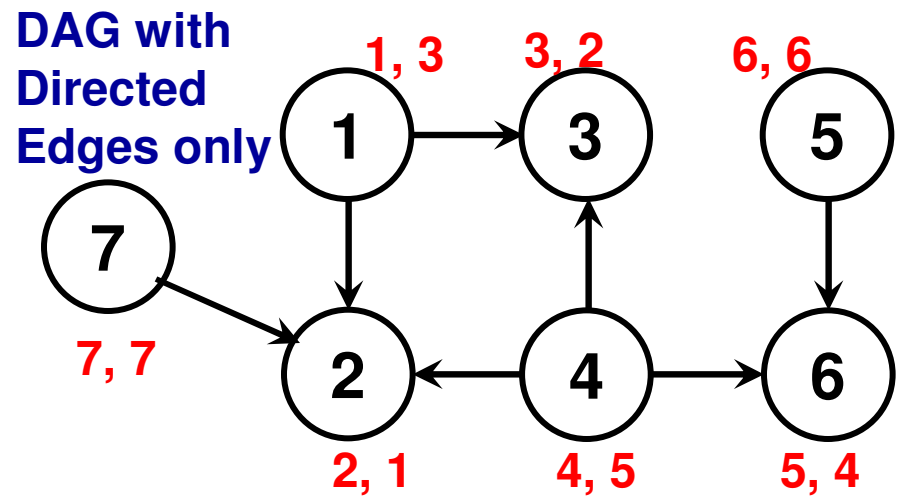
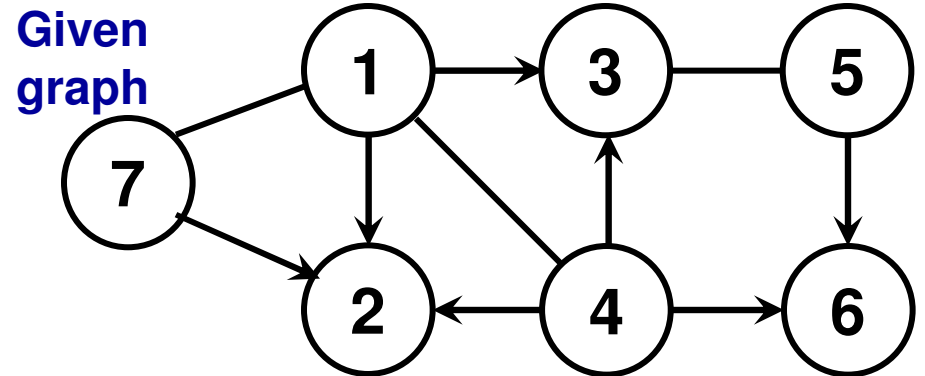
Topological Sort

Proof for Sufficient Condition

- After running DFS on the directed graph (also a DAG), the topological sorting is the listing of the vertices of the DAG in the reverse order according to which they are removed from the stack.
 - Consider an edge $u \rightarrow v$ in the DAG.
 - If there exists an ordering that lists v ahead of u , then it implies that u was popped out from the stack ahead of v . That is, vertex v has been already added to the stack and we were to be able to visit vertex u by exploring a path leading from v to u . This means the edge $u \rightarrow v$ has to be a **back edge**. This implies, the digraph has a cycle and is not a DAG. We had earlier proved that if a digraph has a cycle, we cannot generate a topological sort of its vertices.
 - For an edge $u \rightarrow v$, if v is listed ahead of $u \implies$ the graph is not a DAG (Note that $a \implies b$, then $!b \implies !a$)
 - If the graph is a DAG $\implies u$ should be listed ahead of v for every edge $u \rightarrow v$.
 - Hence, it is sufficient for a directed to be DAG to generate a topological sort for it.

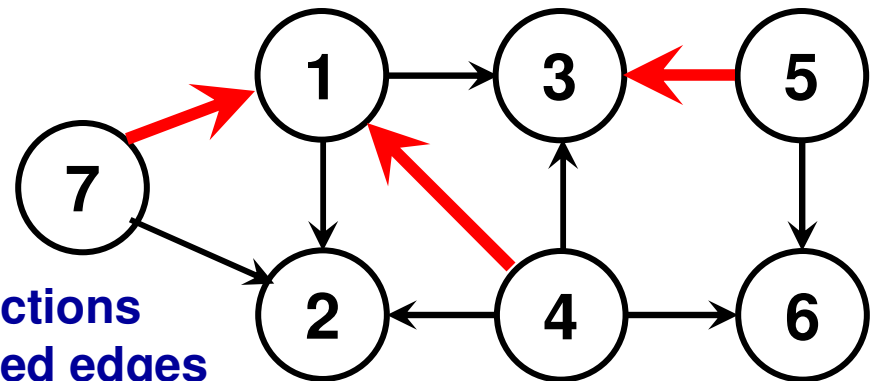
Assigning Directions to Undirected Edges in a Graph (DAG)

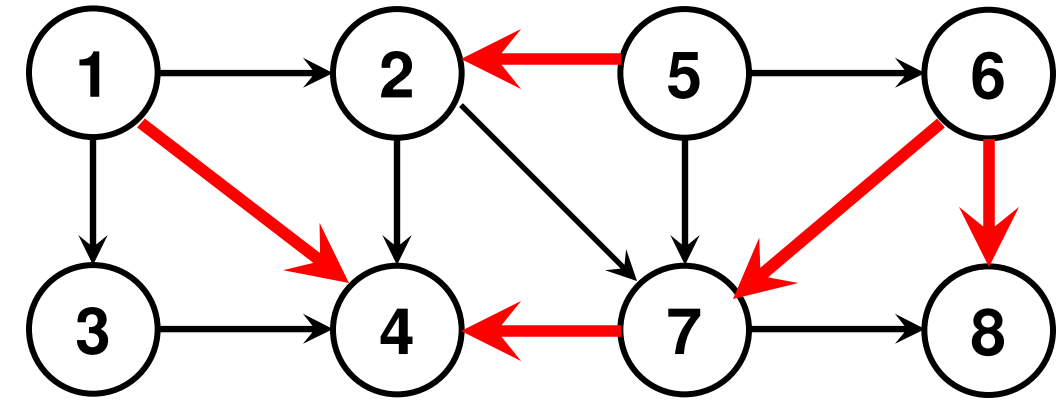
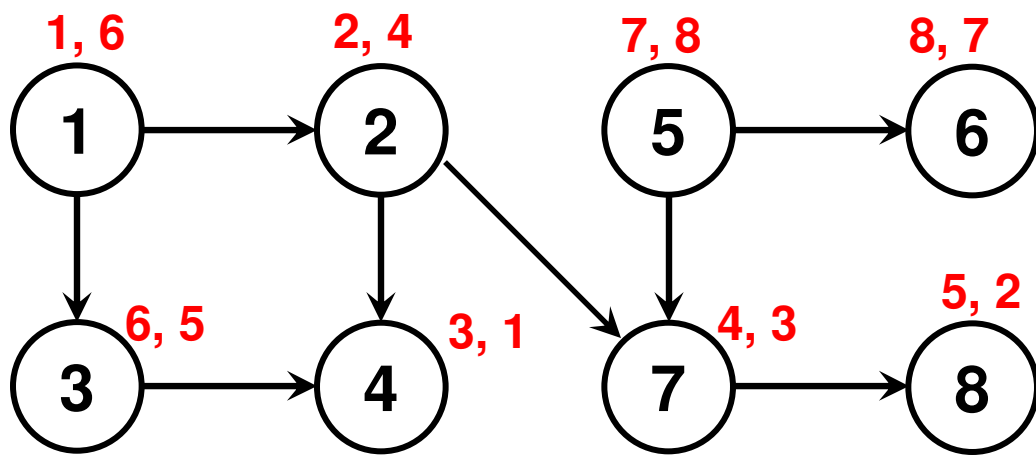
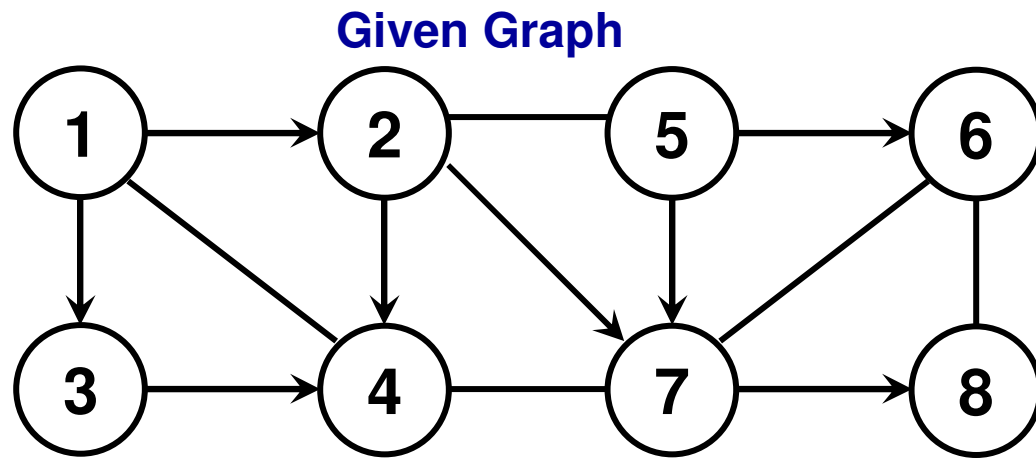
- Consider a directed graph that has a mix of undirected and directed edges. The directed edges of the graph do not create a cycle. We want to assign directions to the undirected edges so that the directed graph remains acyclic.
- Solution:
- Run DFS on the directed edges alone and determine a topological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



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

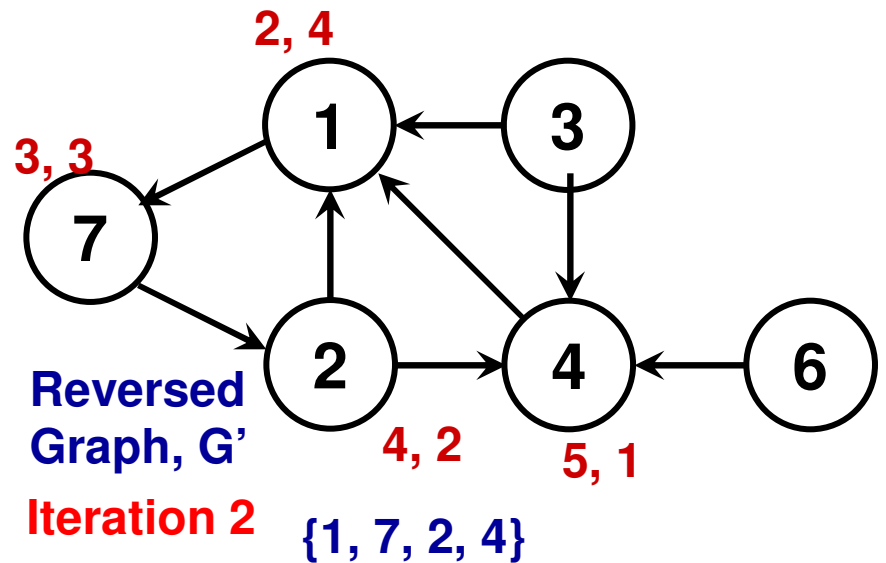
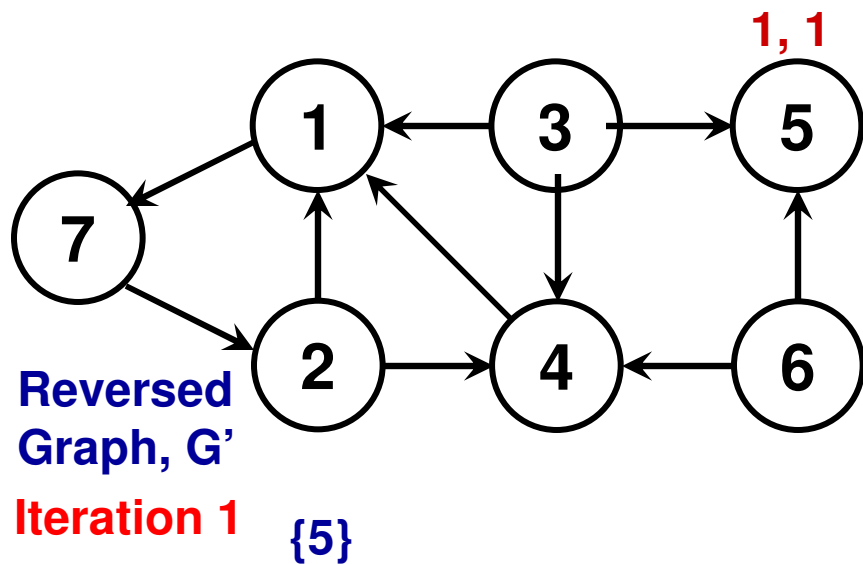
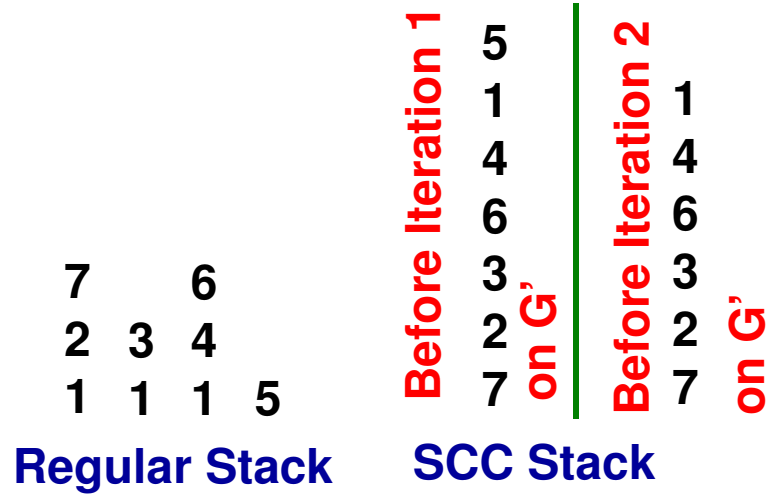
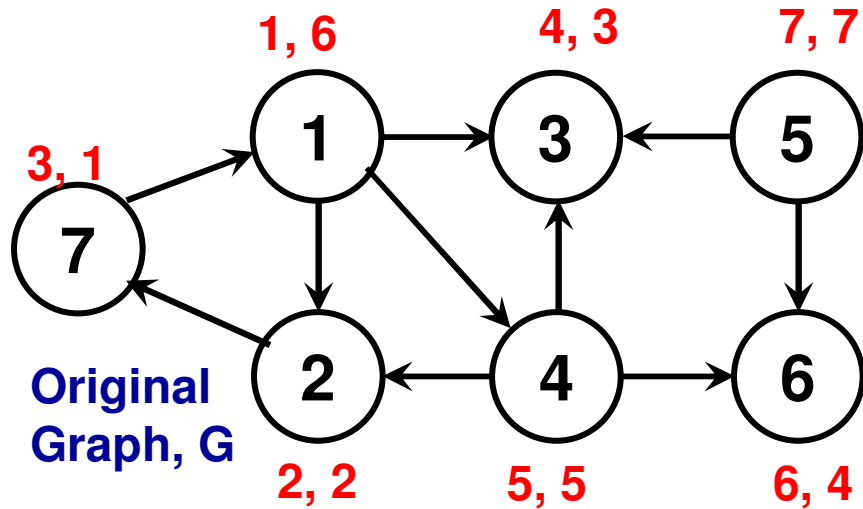
Strongly and Weakly Connected Components of a Directed Graph

- 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.

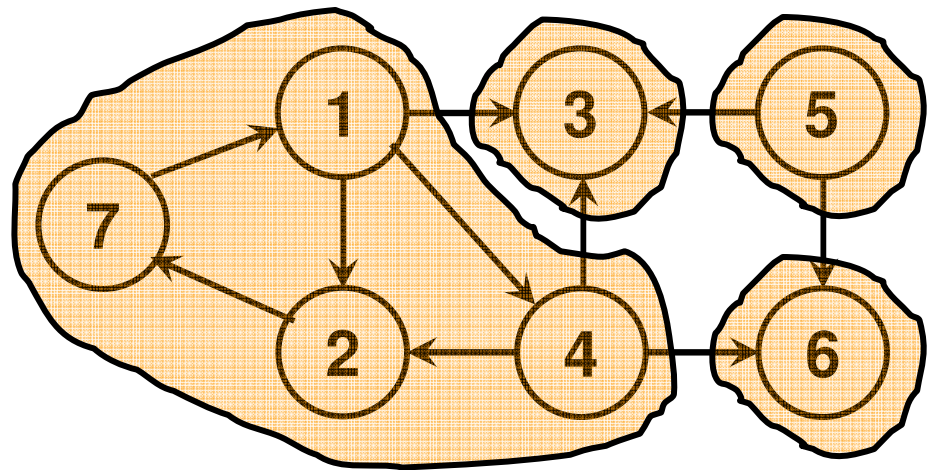
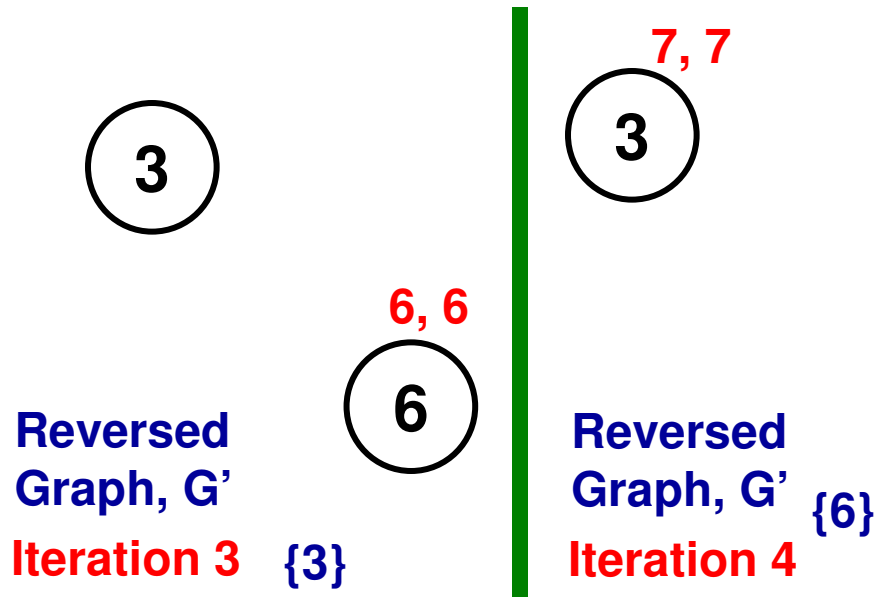
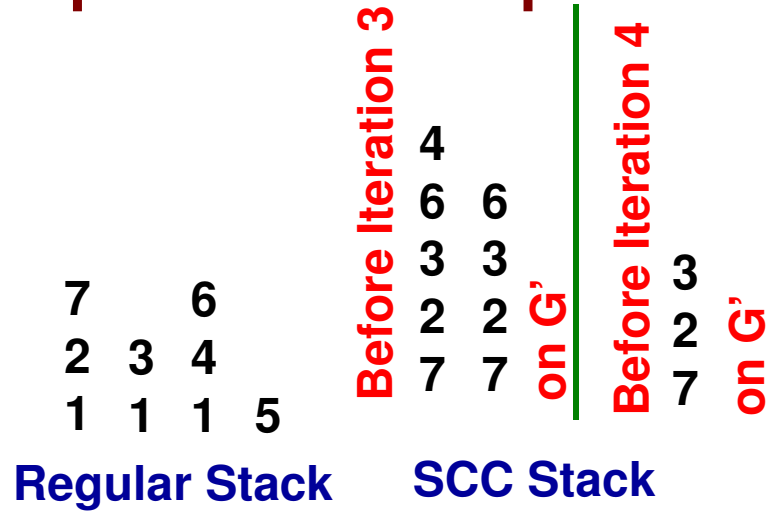
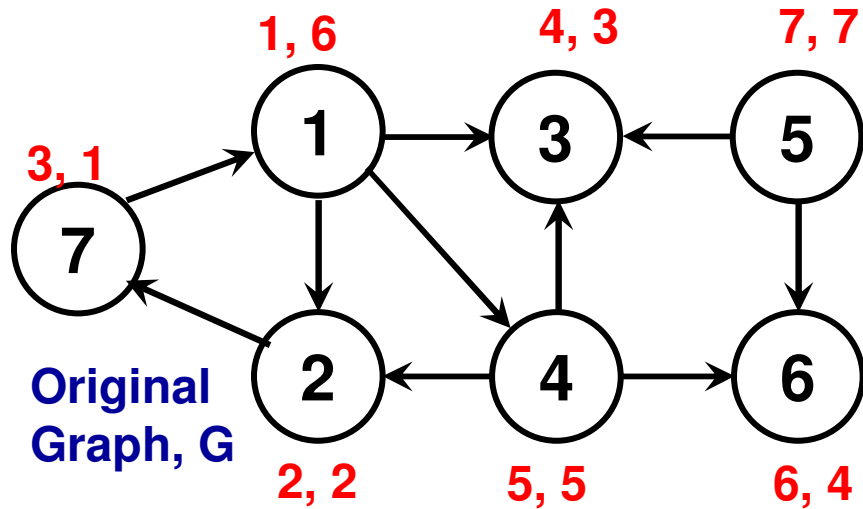
Strongly Connected Components (SCC) of a Directed Graph

- 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 all vertices in G' are visited.

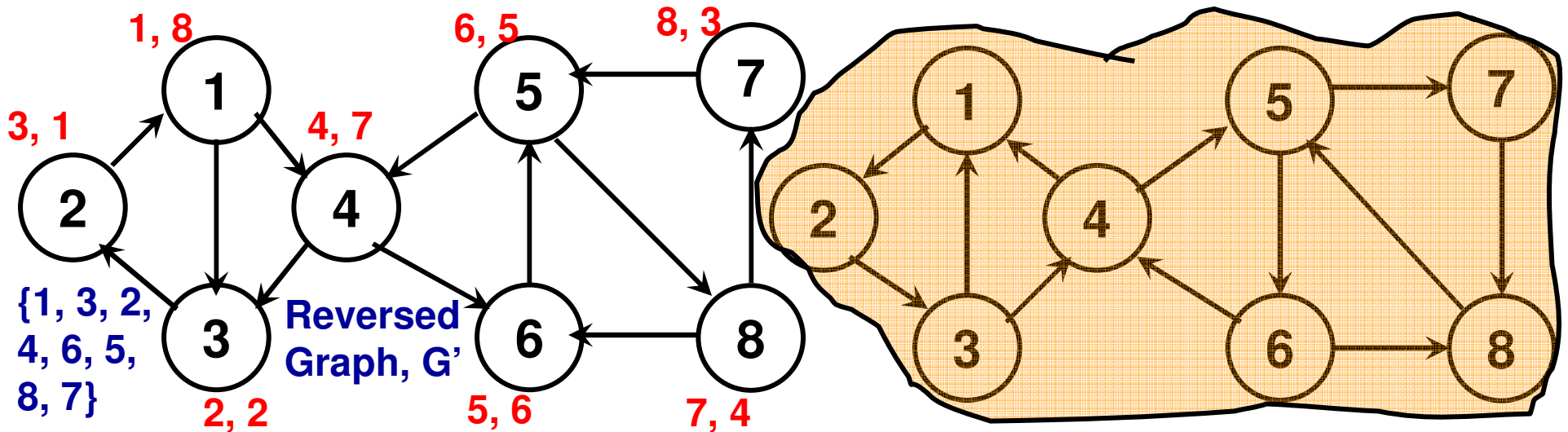
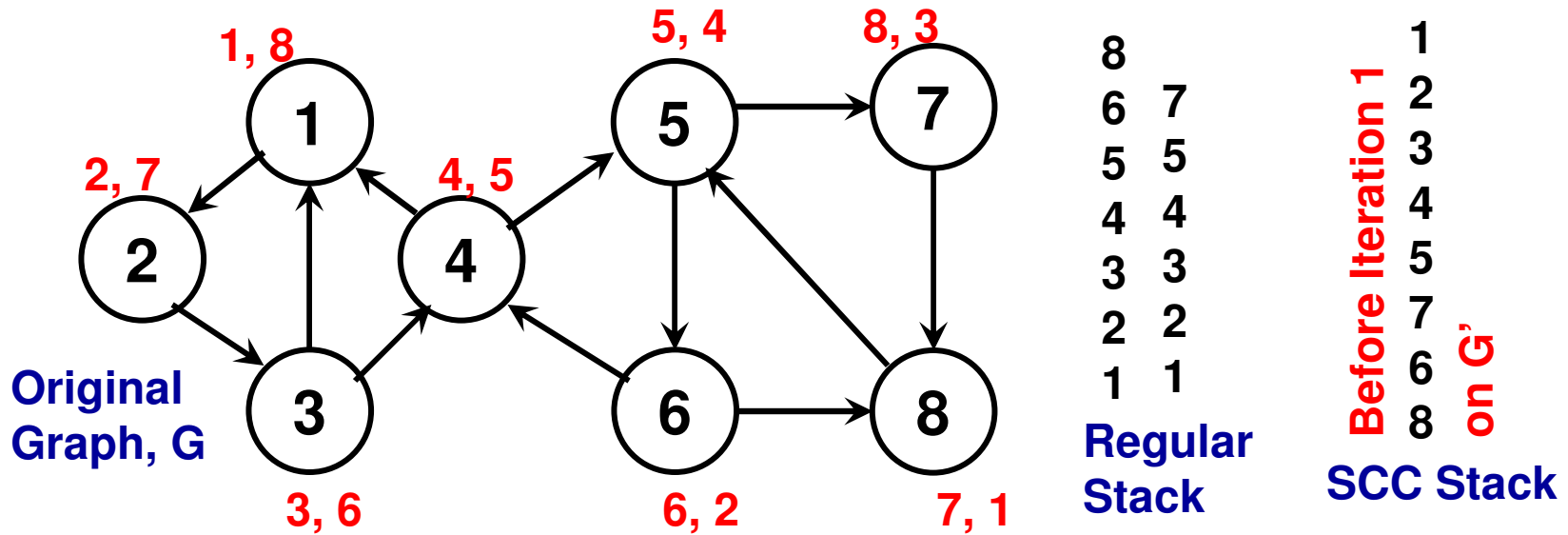
Strongly Connected Components for a Directed Graph: Example 1



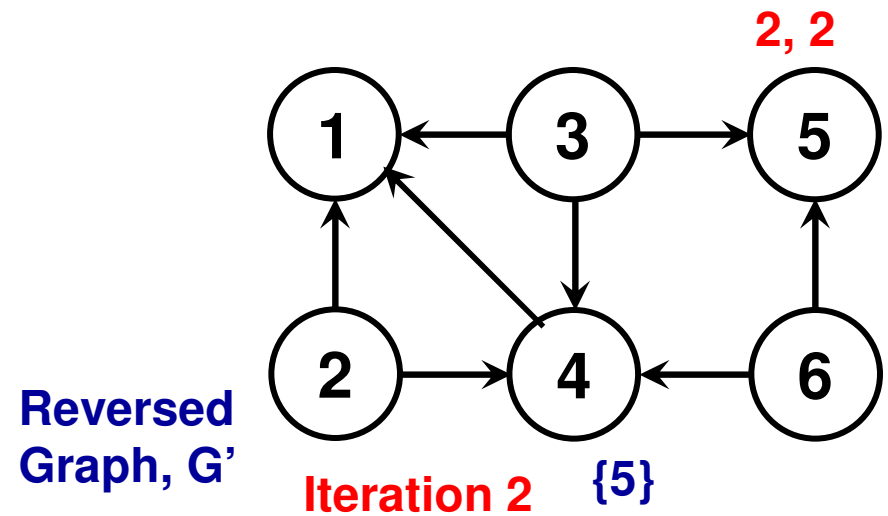
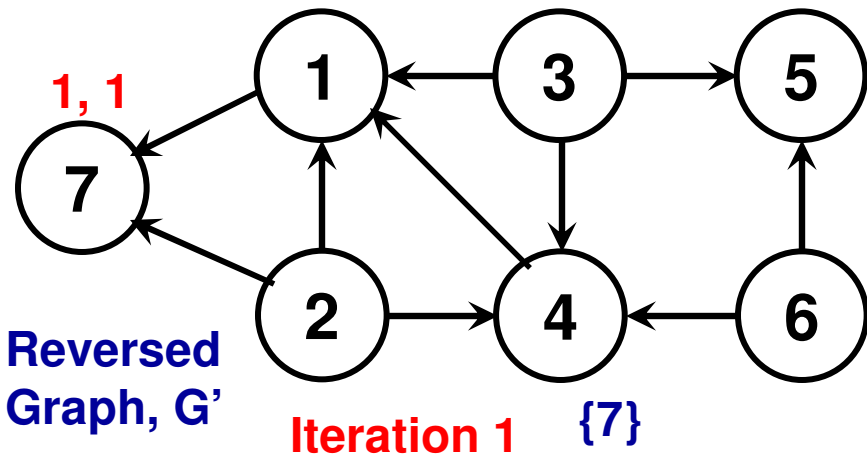
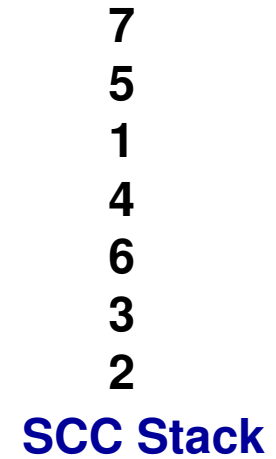
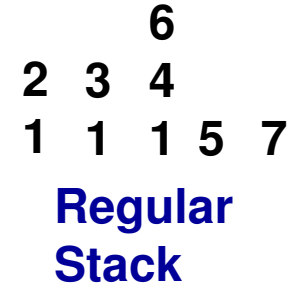
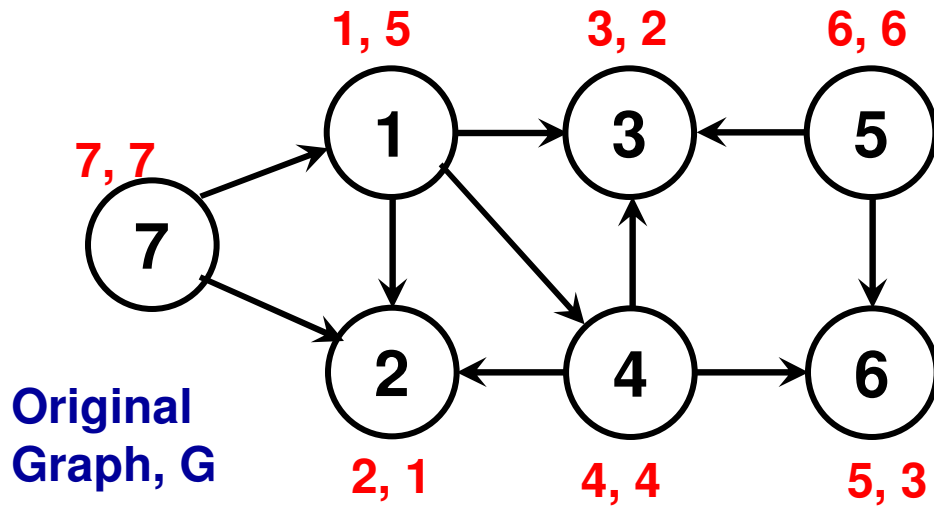
Strongly Connected Components for a Directed Graph: Example 1



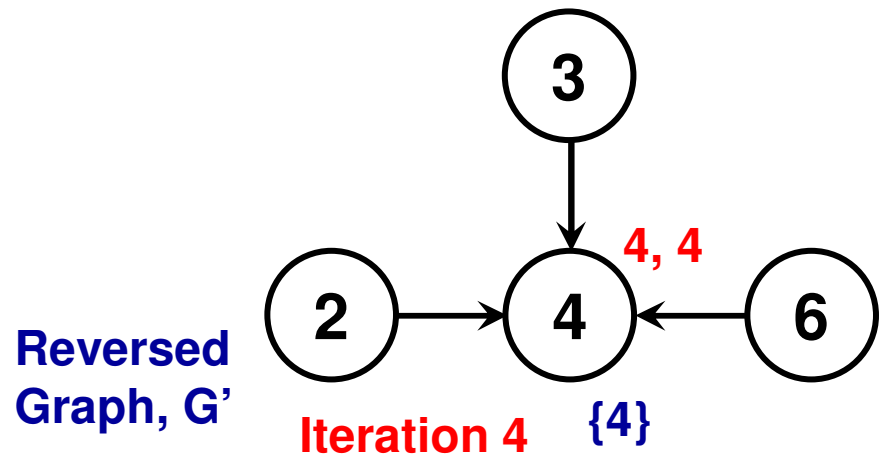
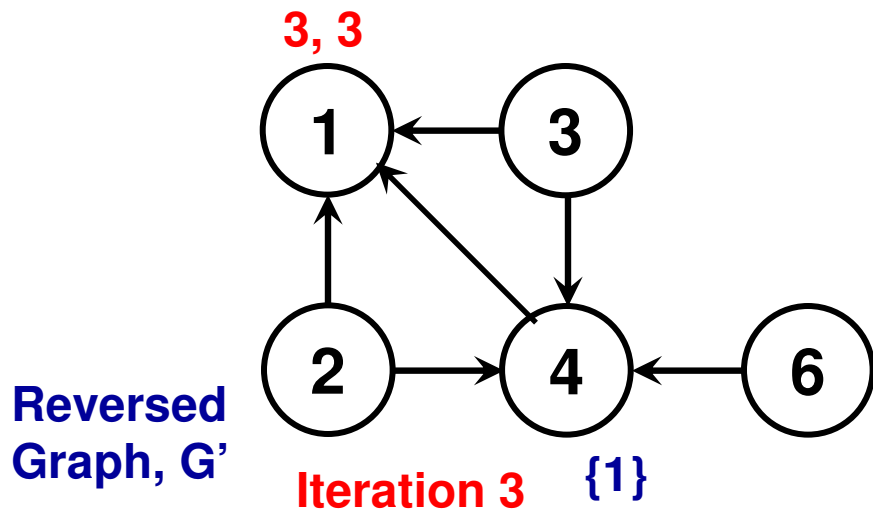
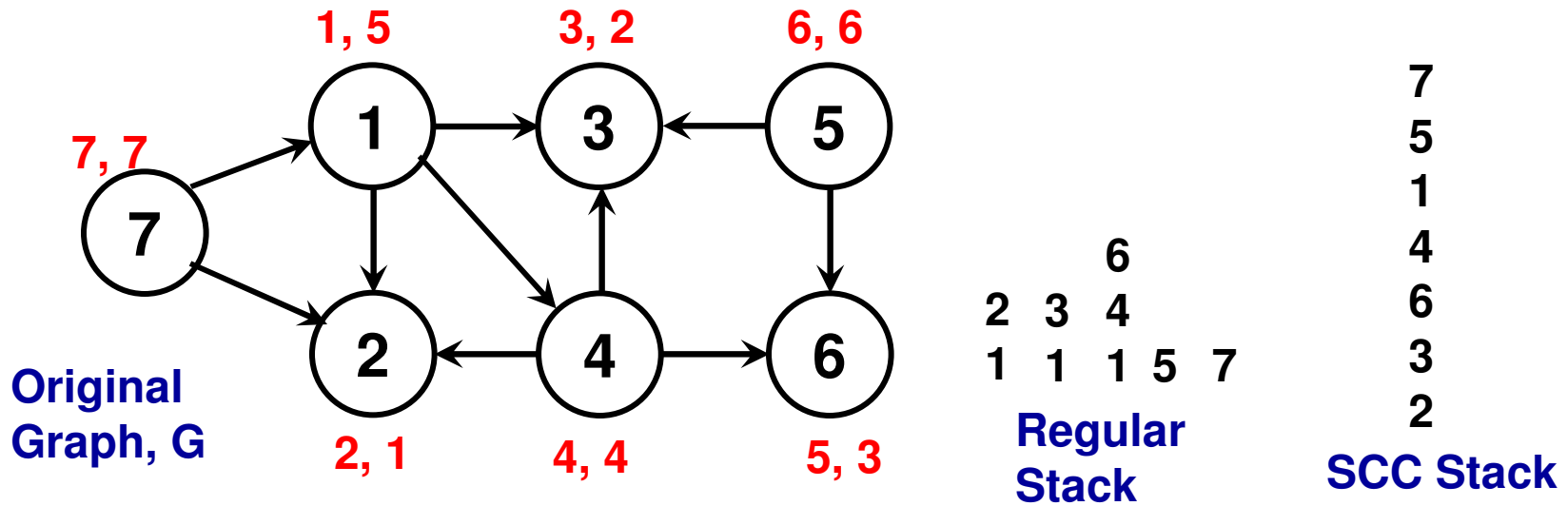
Strongly Connected Components of a Directed Graph: Example 2



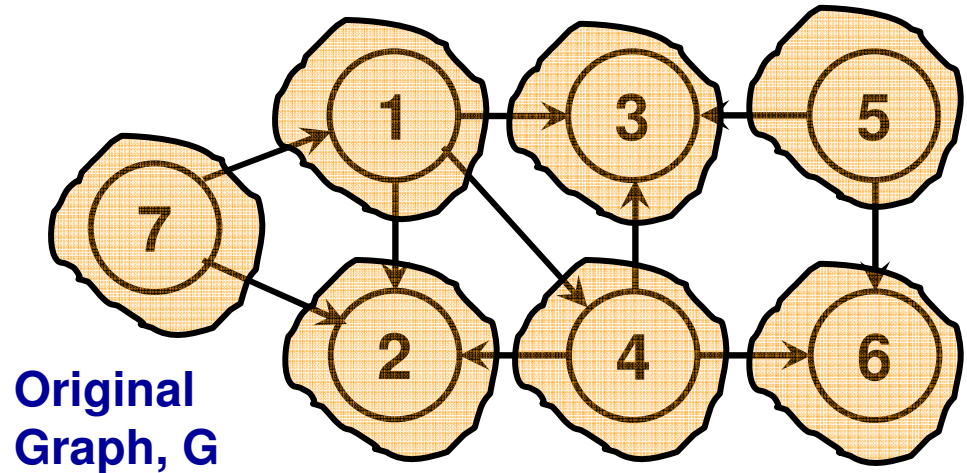
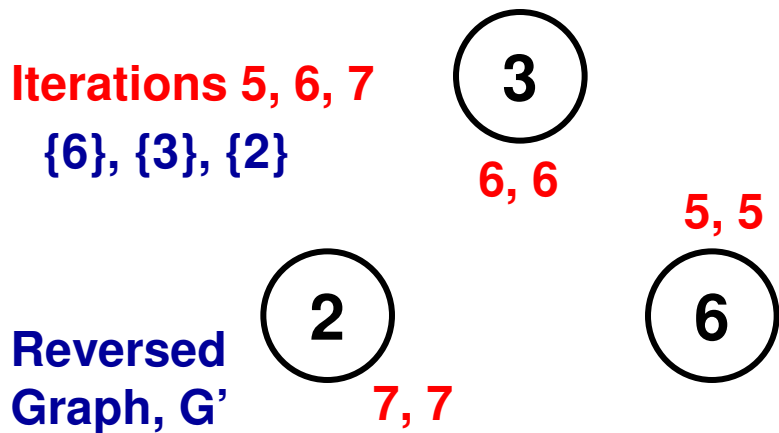
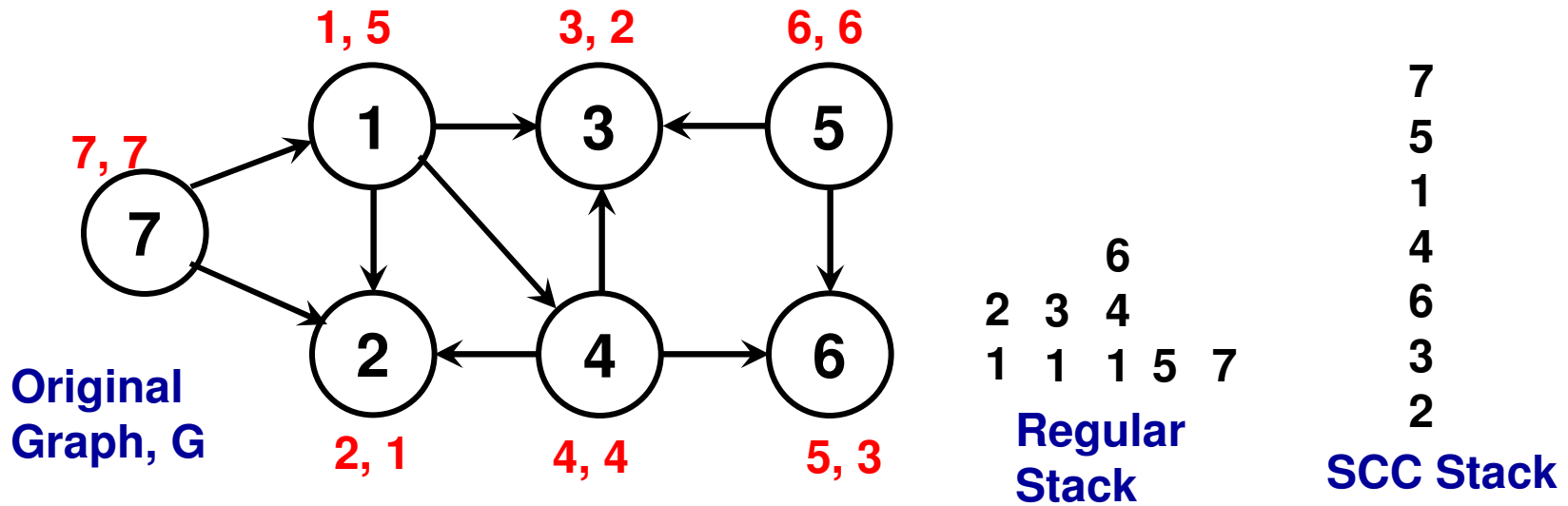
Strongly Connected Components of a Directed Graph: Example 3



Strongly Connected Components of a Directed Graph: Example 3



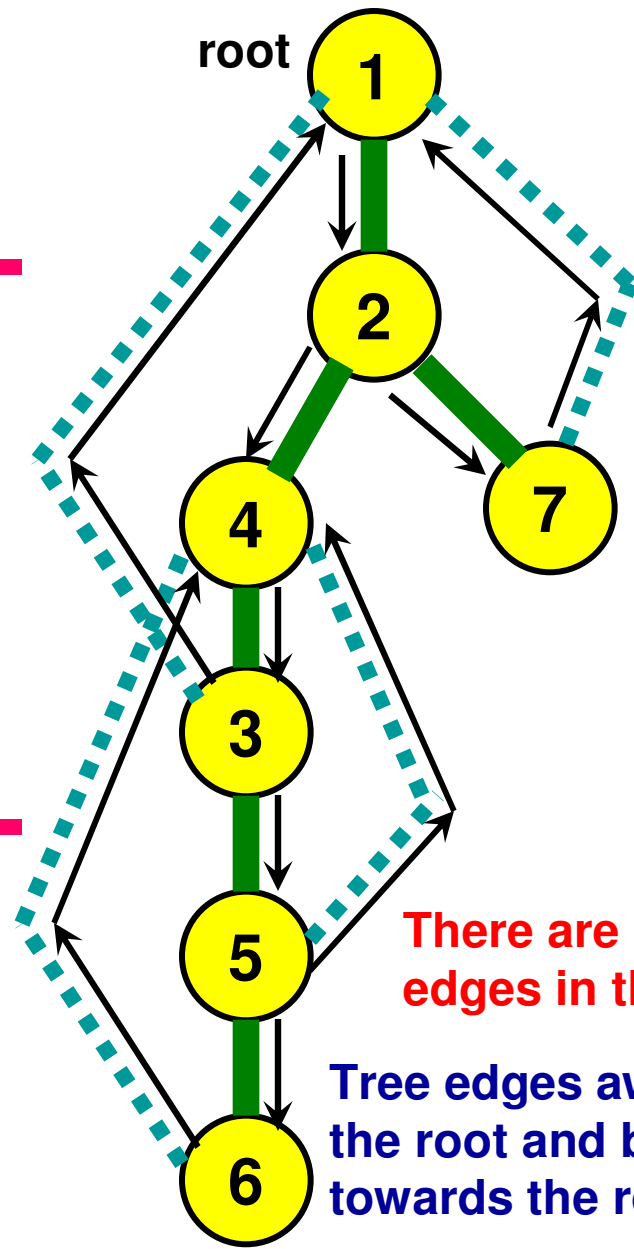
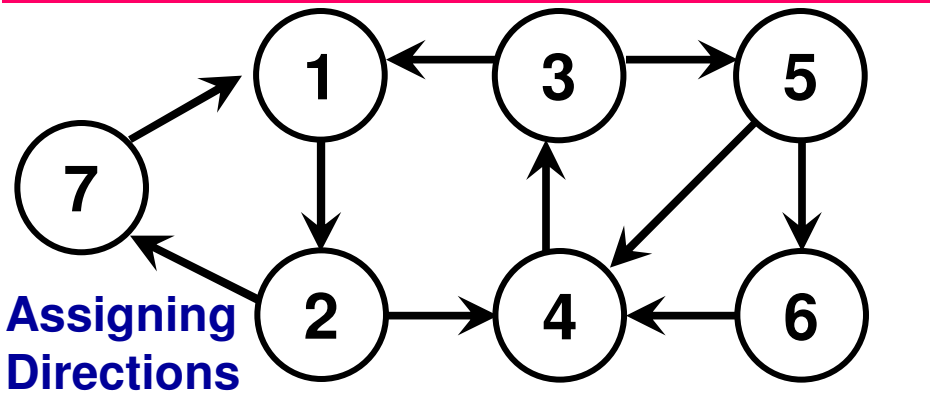
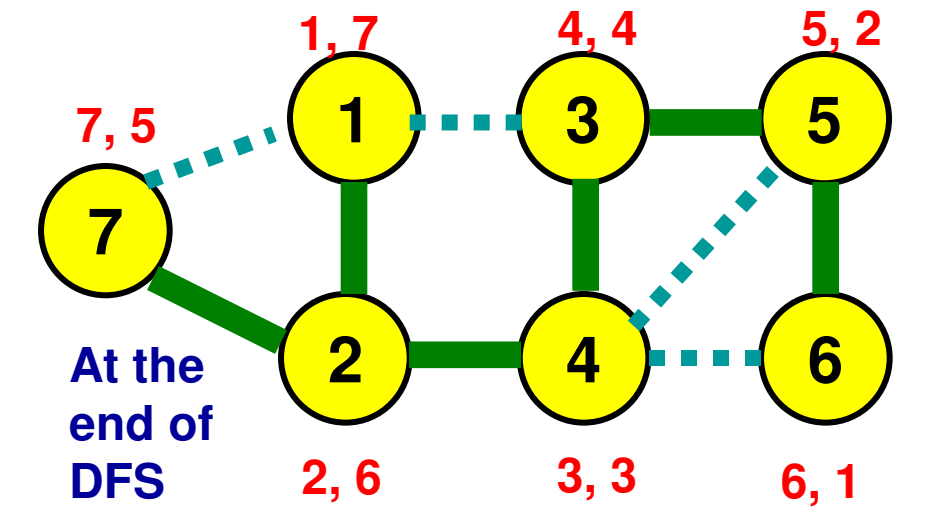
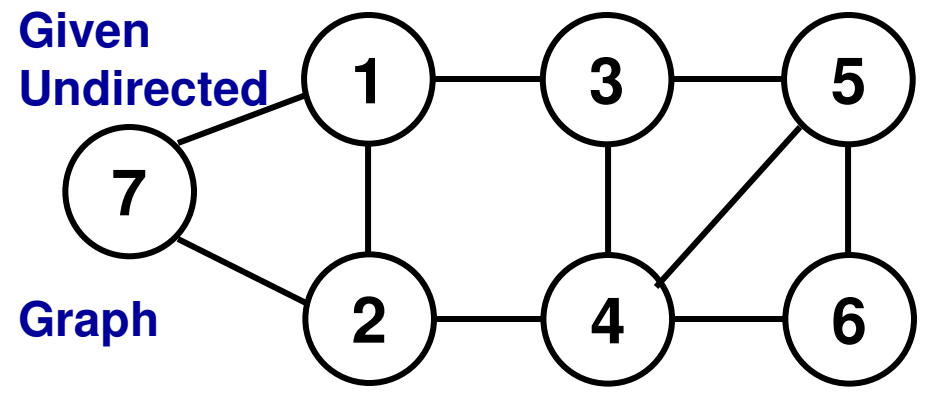
Strongly Connected Components of a Directed Graph: Example 3



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

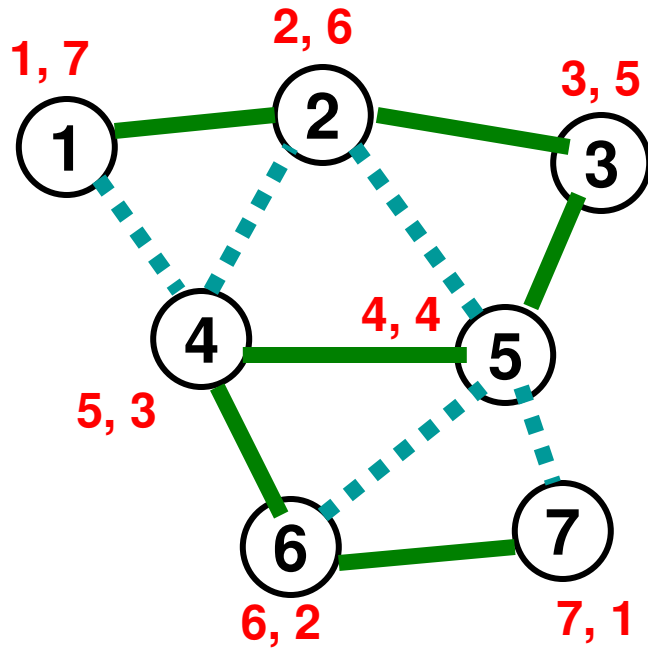
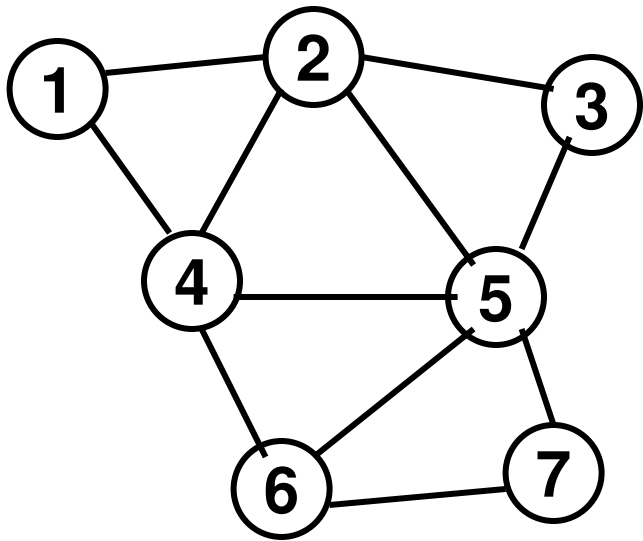
- We are given an undirected graph that is connected and has no bridges (i.e., the graph is 2-edge connected).
- We want to assign directions to the edges of this graph so that the resulting directed graph has all the vertices in a single strongly connected component (i.e., the resulting directed graph 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



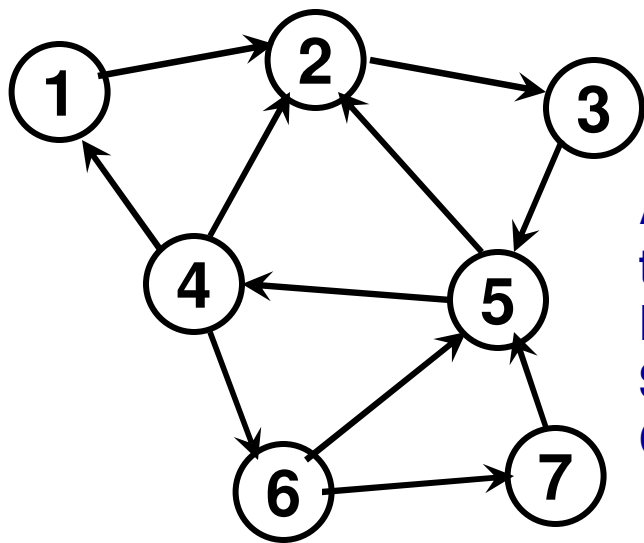
There are no bridge edges in this graph

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

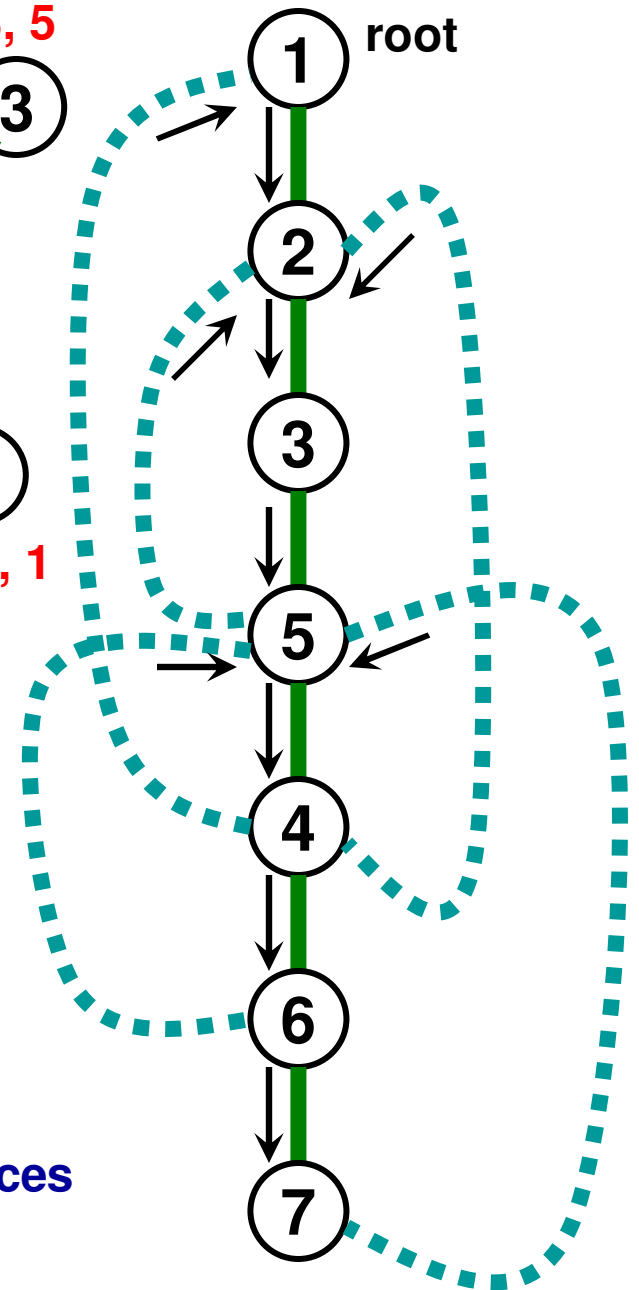


Ex-2

There are no Bridge edges



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



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

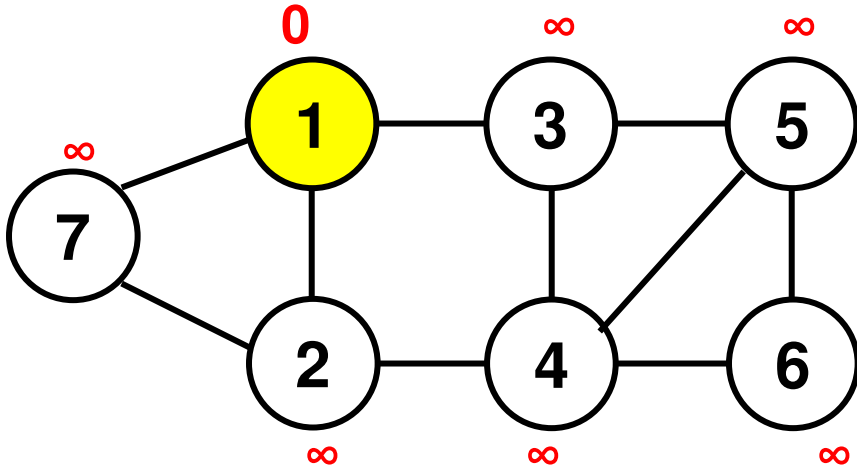
end while

End BFS

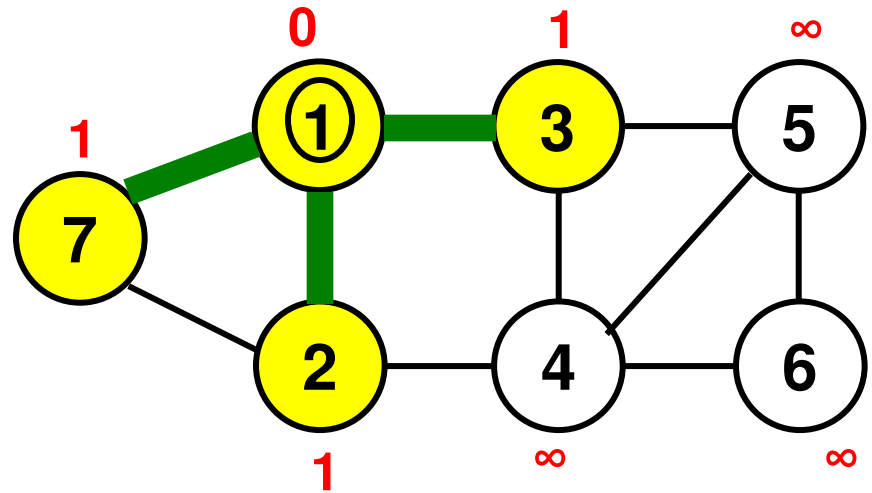
Time Complexity:

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 $\Theta(V+E)$ when implemented using an Adjacency list and $\Theta(V^2)$ when implemented using an Adjacency matrix.

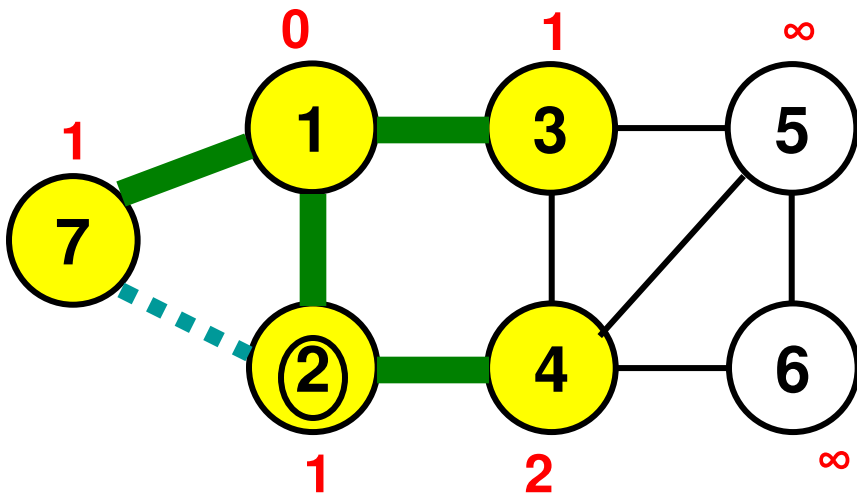
BFS: Example 1



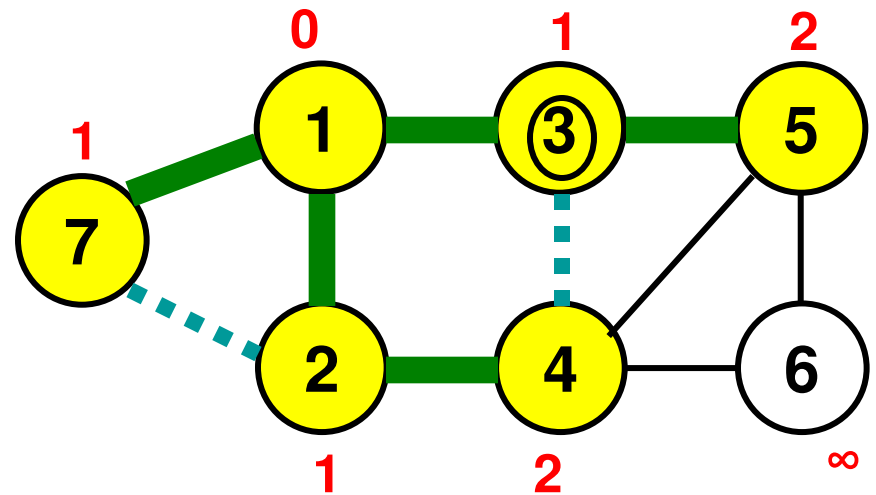
Initialization: Queue = {1}



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

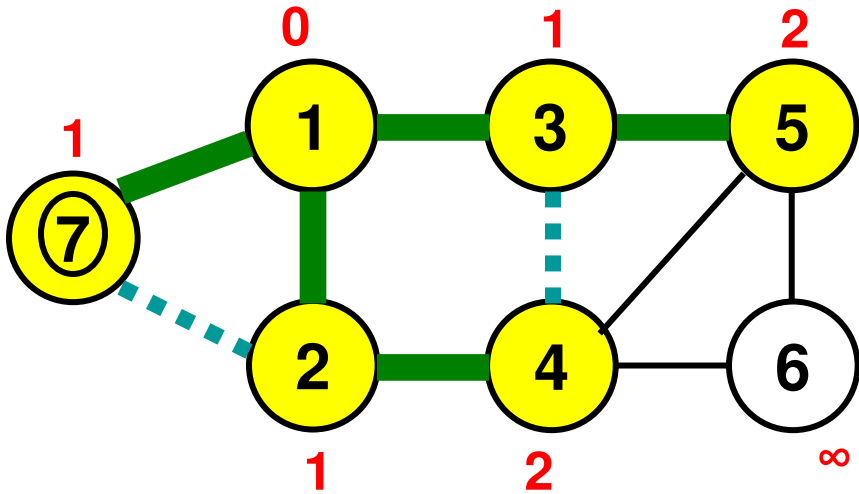


Iteration 2: Queue = {3, 7, 4}

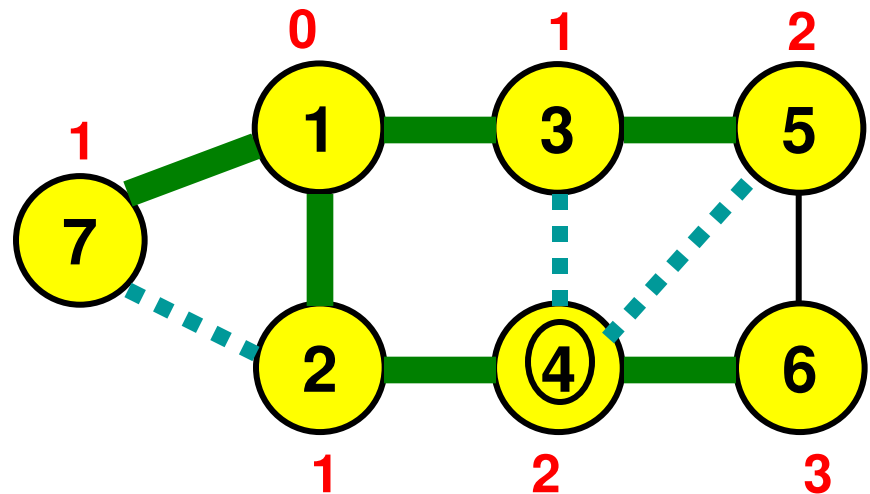


Iteration 3: Queue = {7, 4, 5}

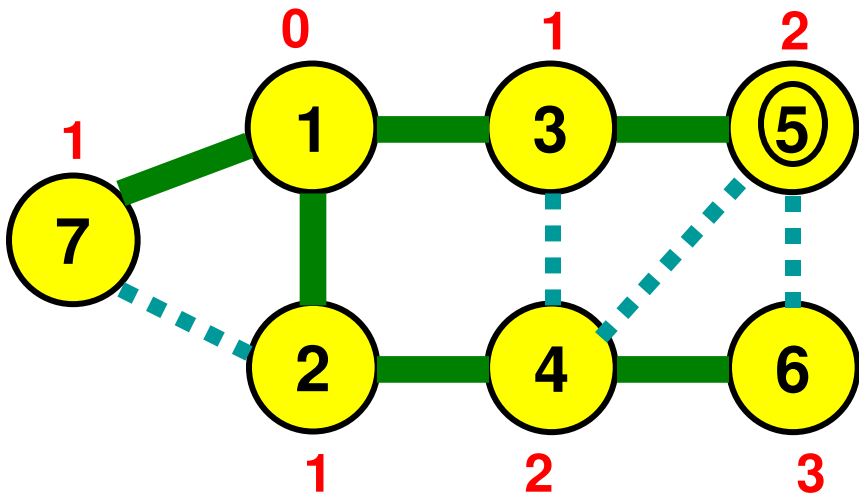
BFS: Example 1



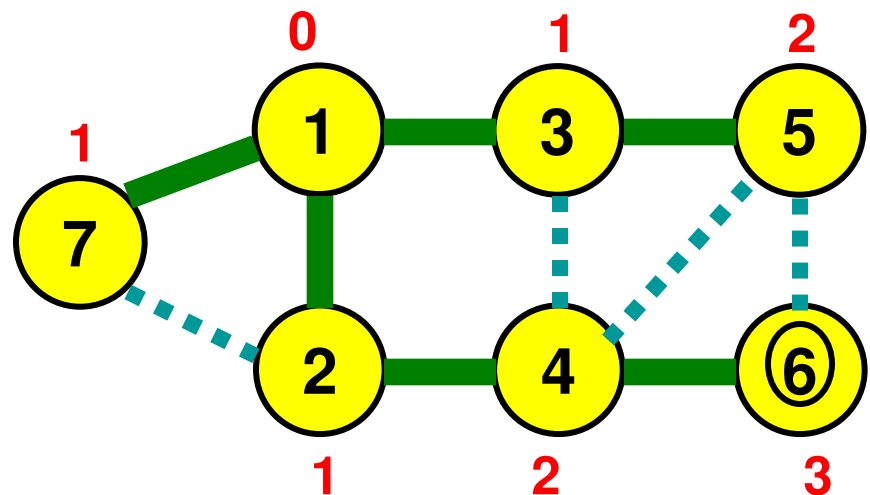
Iteration 4: Queue = {4, 5}



Iteration 5: Queue = {5, 6}



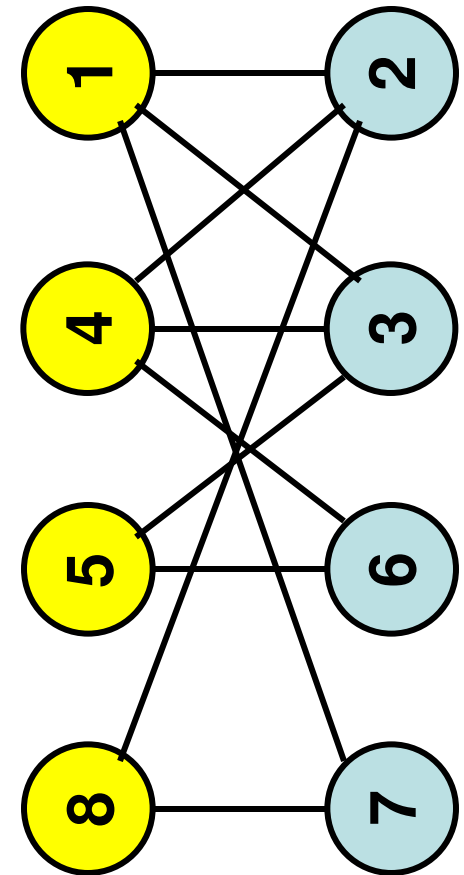
Iteration 6: Queue = {6}



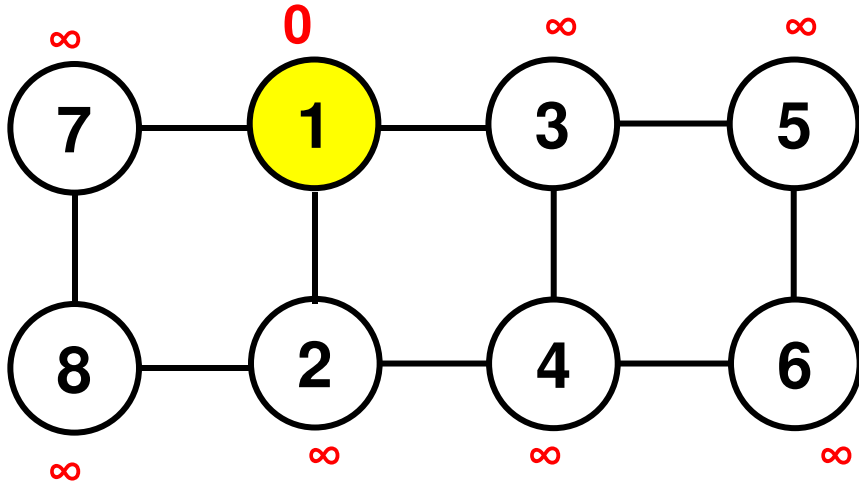
Iteration 7: Queue = {}

Bipartite Graph

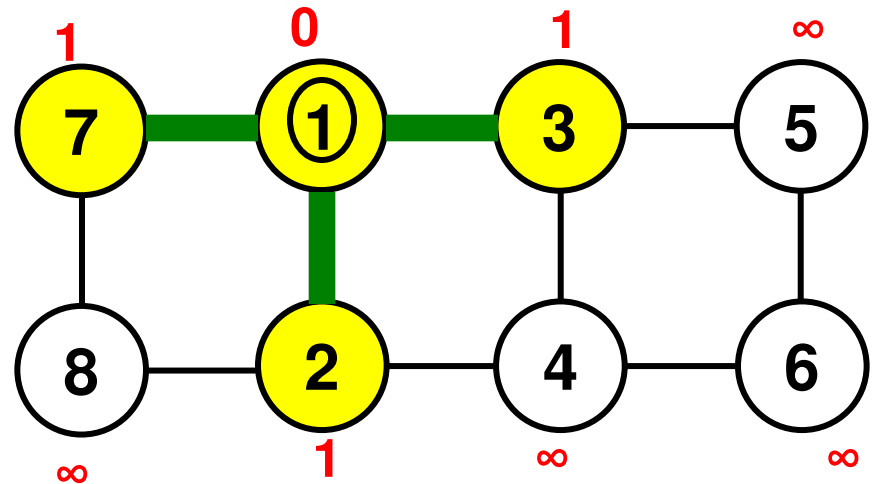
- A graph in which the vertices could be partitioned to two disjoint sets such that all the edges in the graph are between the vertices in the two sets and there are no edges between vertices in the same set.
- An undirected graph is bipartite if there are no cross edges between vertices at the same 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.



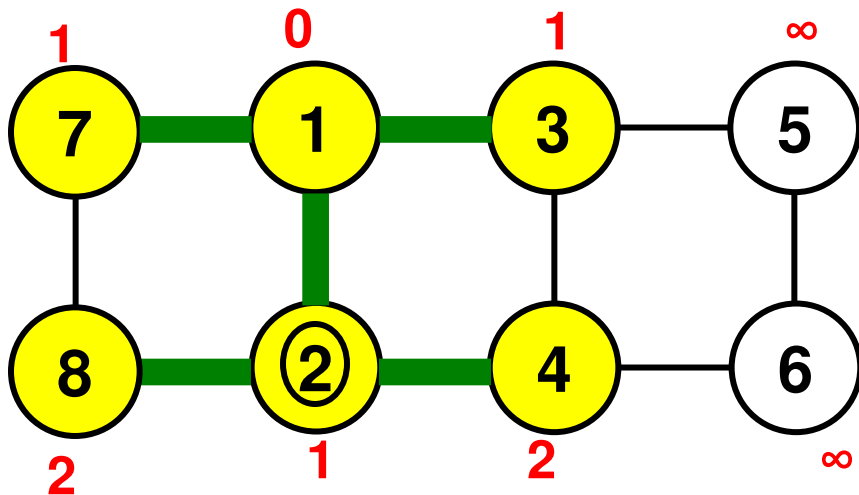
BFS: Example 2



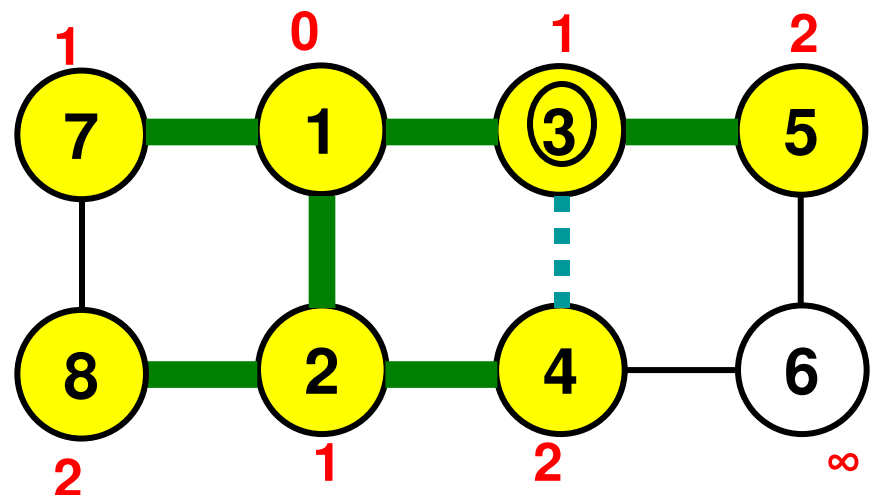
Initialization: Queue = {1}



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

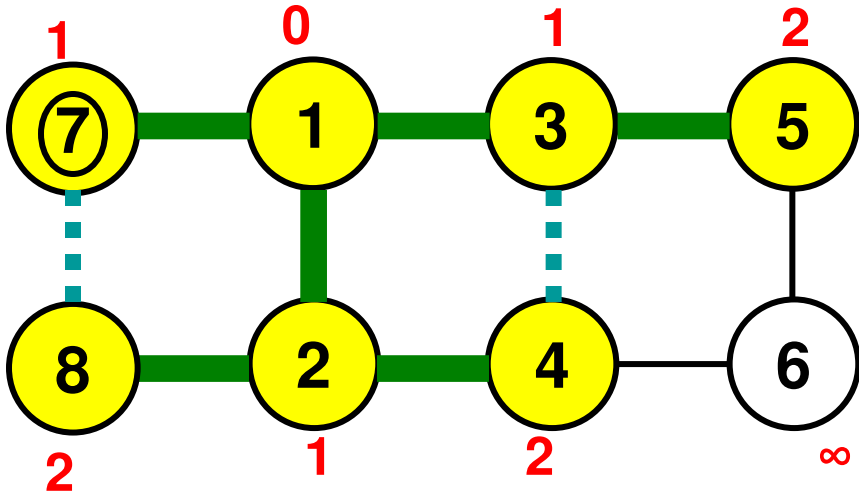


Iteration 2: Queue = {3, 7, 4, 8}

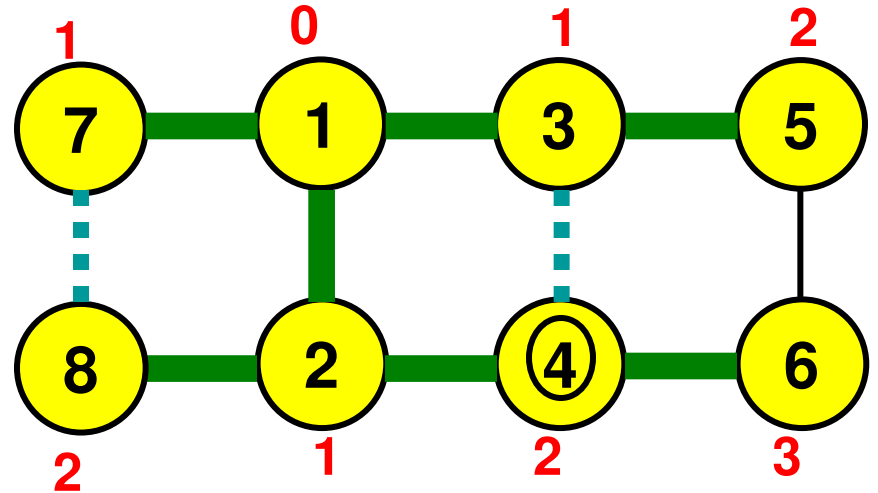


Iteration 3: Queue = {7, 4, 8, 5}

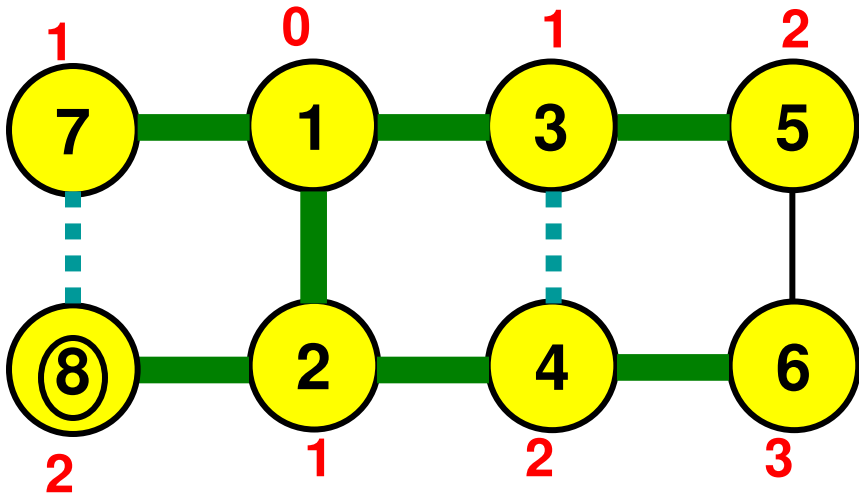
BFS: Example 2



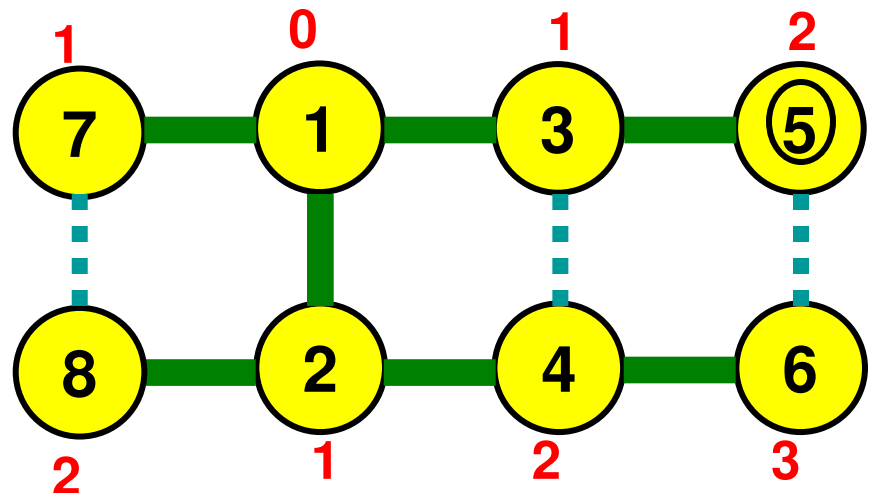
Iteration 4: Queue = {4, 8, 5}



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

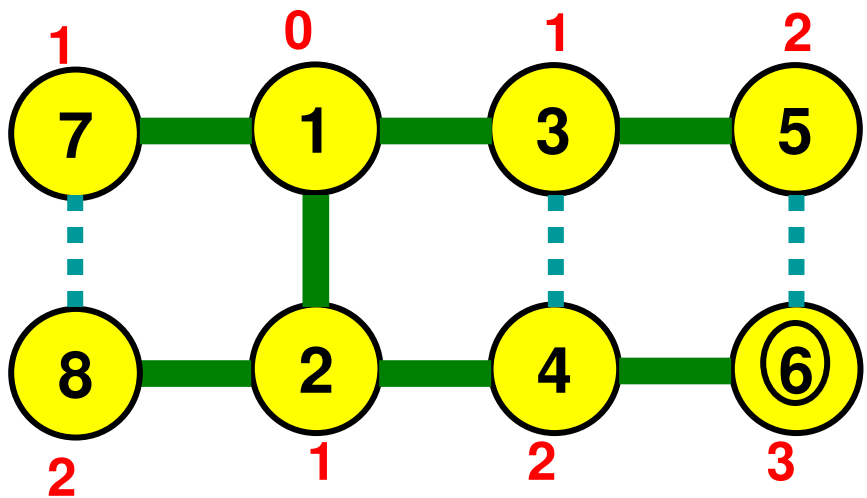


Iteration 6: Queue = {5, 6}



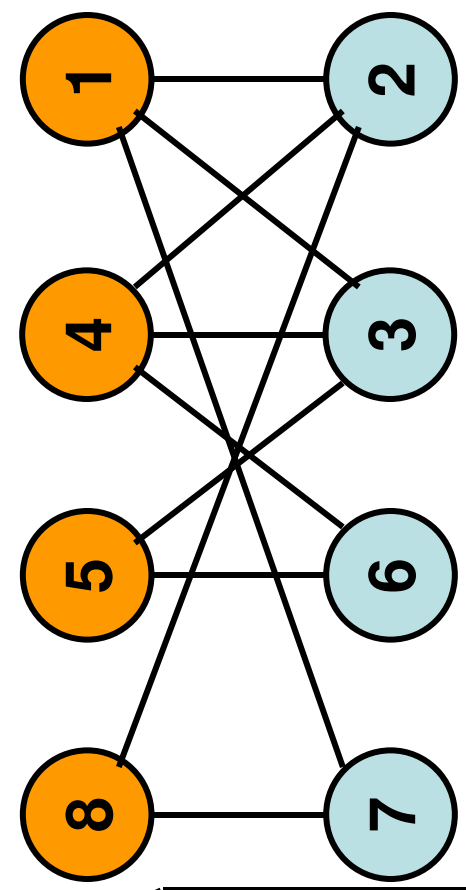
Iteration 7: Queue = {6}

BFS: Example 2



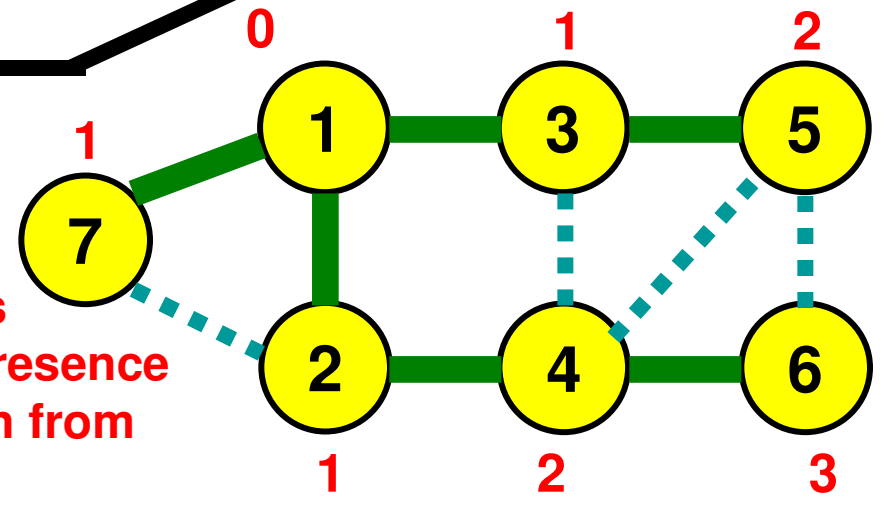
Iteration 8: Queue = {}

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



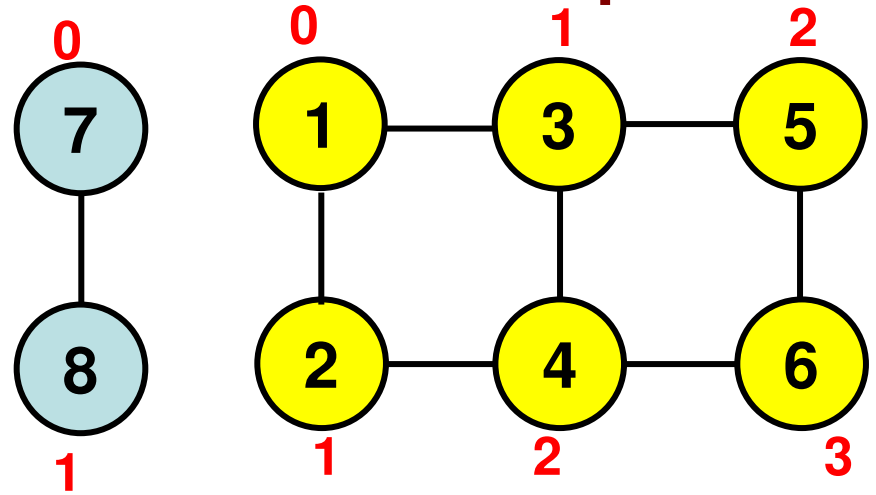
BFS: Example 1

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.

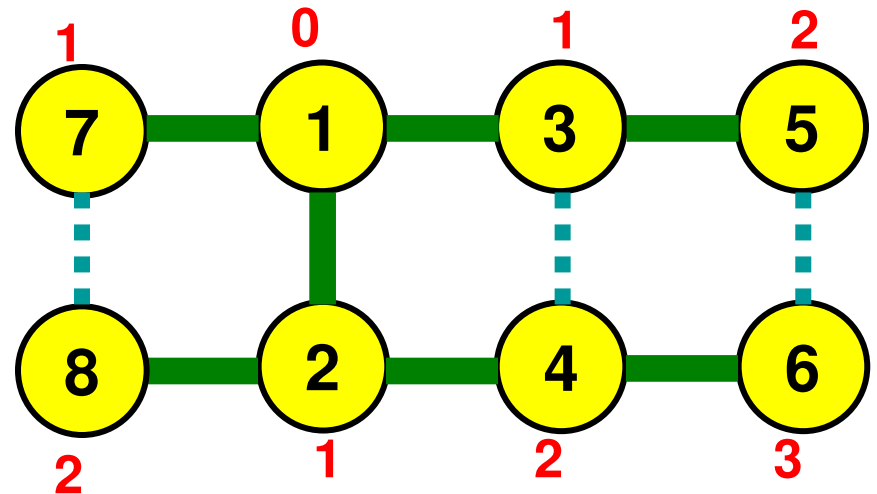


Connected Undirected Graph

- 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 one 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



Comparison of DFS and BFS

| | DFS | BFS |
|---------------------------------|---|--|
| Data structure | a stack | a queue |
| Number of vertex orderings | two orderings | one ordering |
| Edge types (undirected graphs) | tree and back edges | tree and cross edges |
| Applications | connectivity, acyclicity, articulation points | connectivity, acyclicity, minimum-edge paths |
| Efficiency for adjacency matrix | $\Theta(V ^2)$ | $\Theta(V ^2)$ |
| Efficiency for adjacency lists | $\Theta(V + E)$ | $\Theta(V + E)$ |

With the levels of a tree, referenced starting from the root node, A back edge in a DFS tree could connect vertices at different levels; whereas, a cross edge in a BFS tree always connects vertices that are either at the same level or at adjacent levels.

There is always only a unique ordering of the vertices, according to BFS, in the order they are visited (added and removed from the queue in the same order).

On the other hand, with DFS – vertices could be ordered in the order they are pushed to the Stack, typically different from the order in which they are popped from the stack.

Source: Table 3.1: Levitin, 3rd Edition: Introduction to the Design and Analysis of Algorithms, 2012.

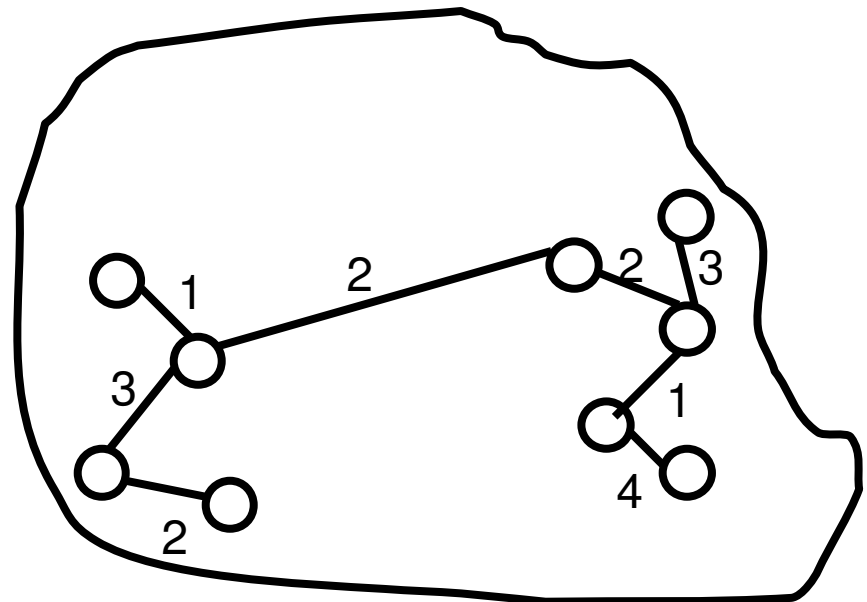
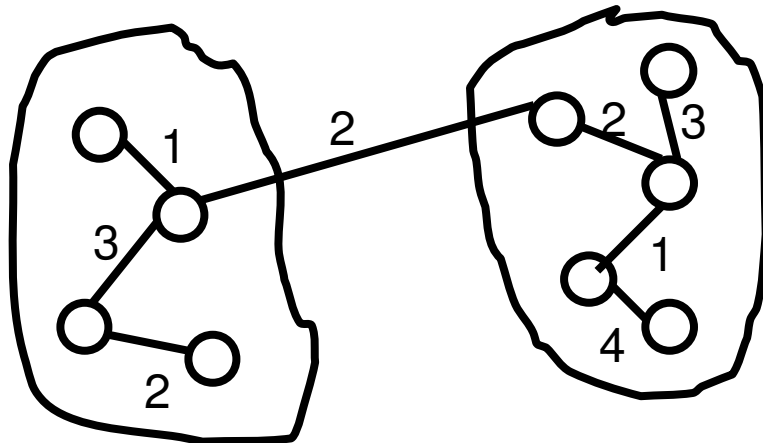
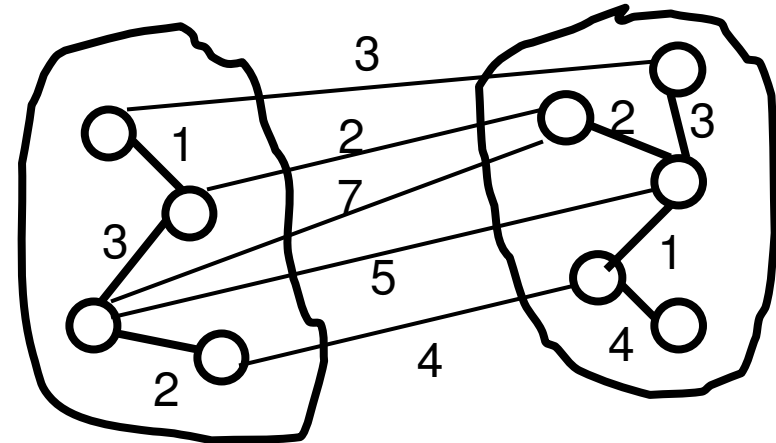
Minimum Spanning Trees

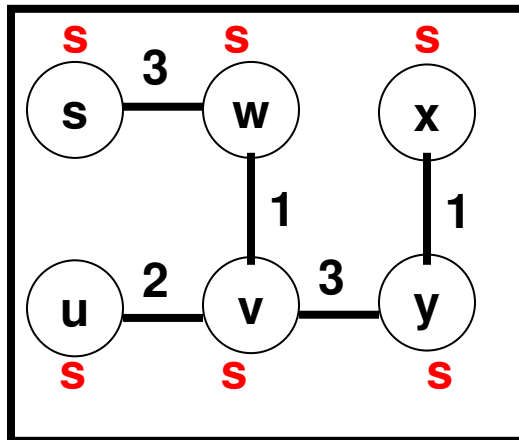
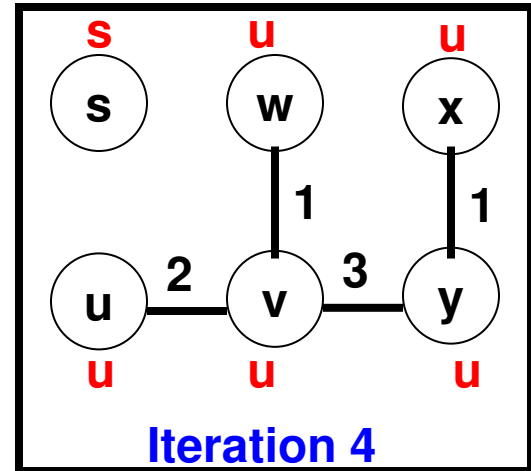
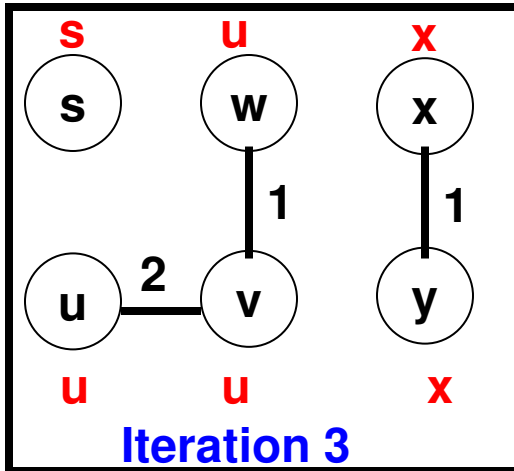
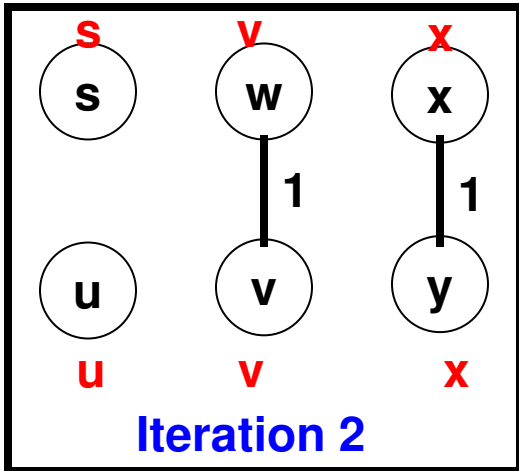
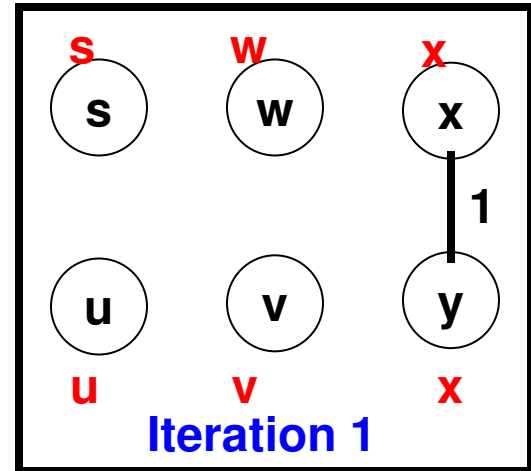
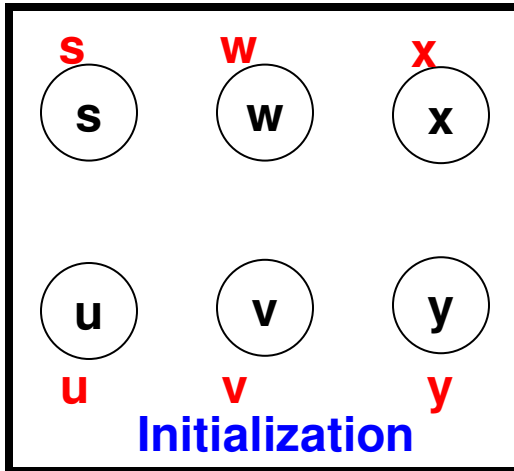
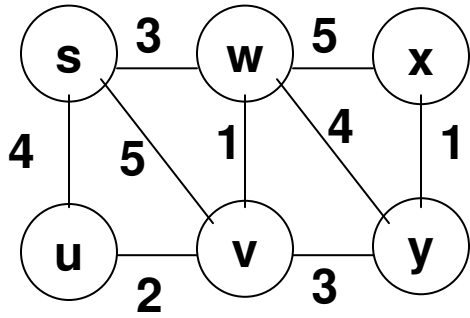
Minimum Spanning Tree Problem

- Given a weighted graph, we want to determine a tree that spans all the vertices in the tree and the sum of the weights of all the edges in such a spanning tree should be minimum.
- Kruskal algorithm: Consider edges in the increasing order of their weights and include an edge in the tree, if and only if, by including the edge in the tree, we do not create a cycle!!
 - For a graph of E edges, we spend $\Theta(E \cdot \log E)$ time to sort the edges and this is the most time consuming step of the algorithm.
- To start with, each vertex is in its own component.
- In each iteration, we merge two components using an edge of minimum weight connecting the vertices across the two components.
 - The merged component does not have a cycle and the sum of all the edge weights within a component is the minimum possible.
- To detect a cycle, the vertices within a component are identified by a component ID. If the edge considered for merging two components comprises of end vertices with the same component ID, then the edge is not considered for the merger.
 - An edge is considered for merging two components only if its end vertices are identified with different component IDs.

Property of any MST Algorithm

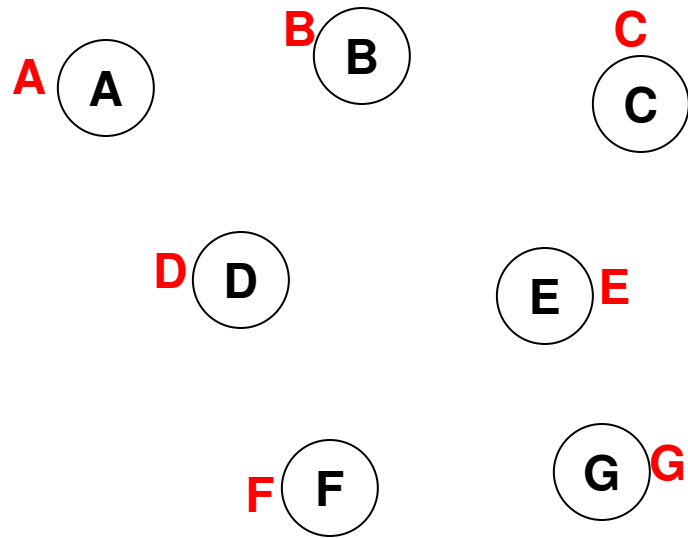
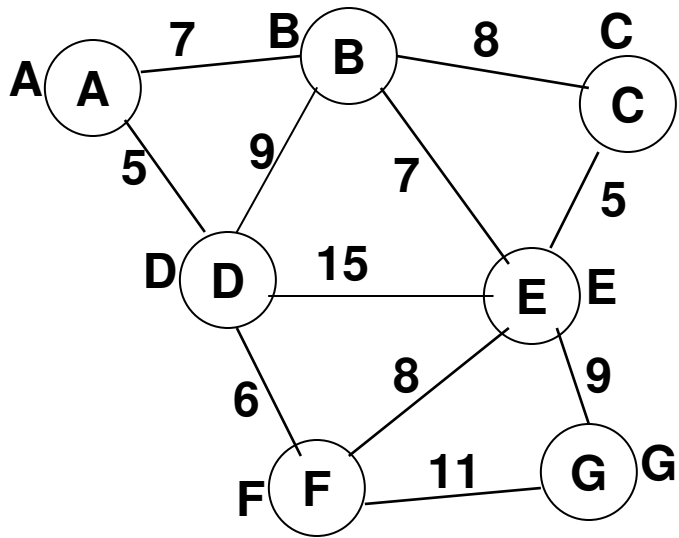
- Given two components of vertices (that are a tree by themselves of the smallest possible weights), any MST algorithm would choose an edge of the smallest weight that could connect the two components such that the merger of the two components is also a tree and is of the smallest possible weight.



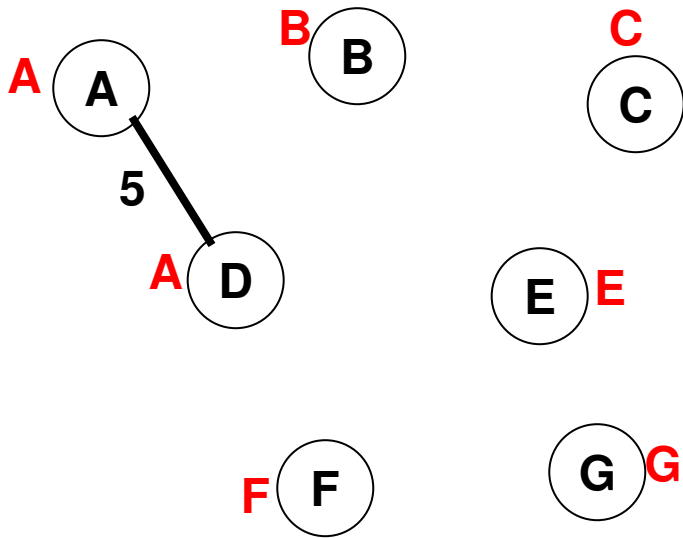


Iteration 5
Min. Spanning Tree

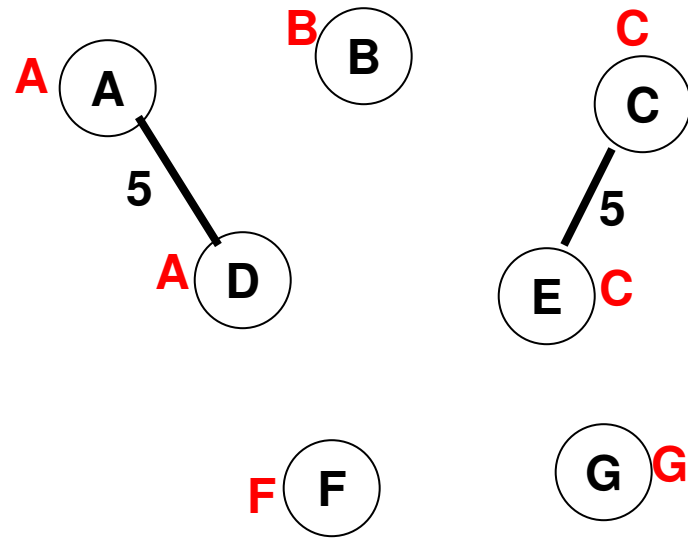
MST
Weight
10



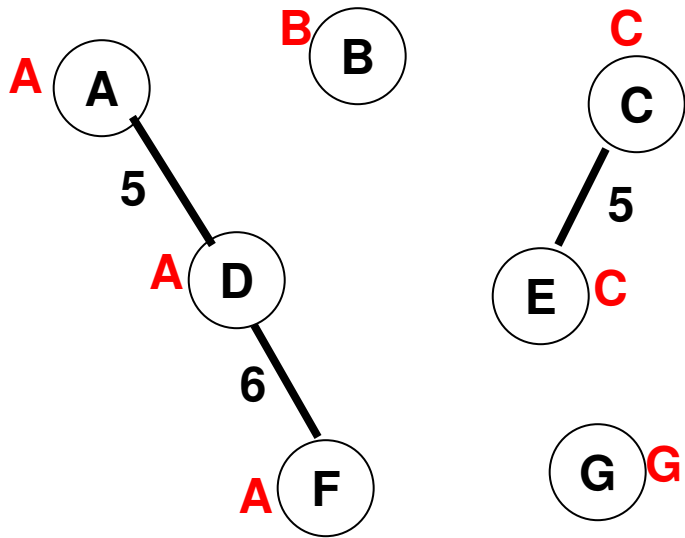
Initialization



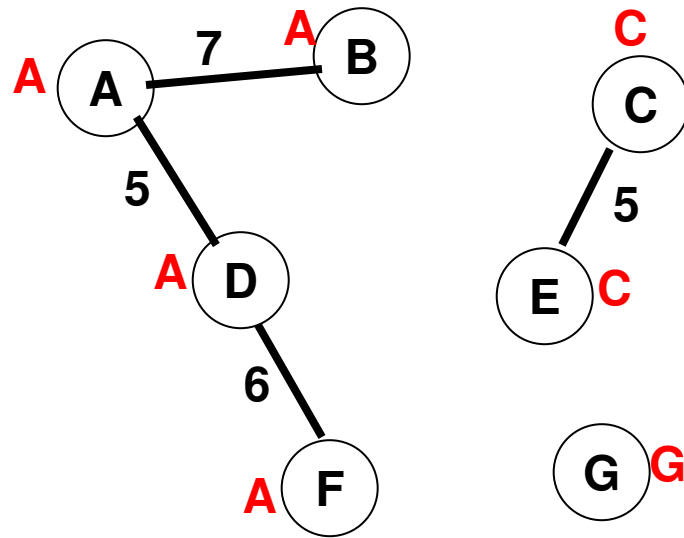
Iteration 1



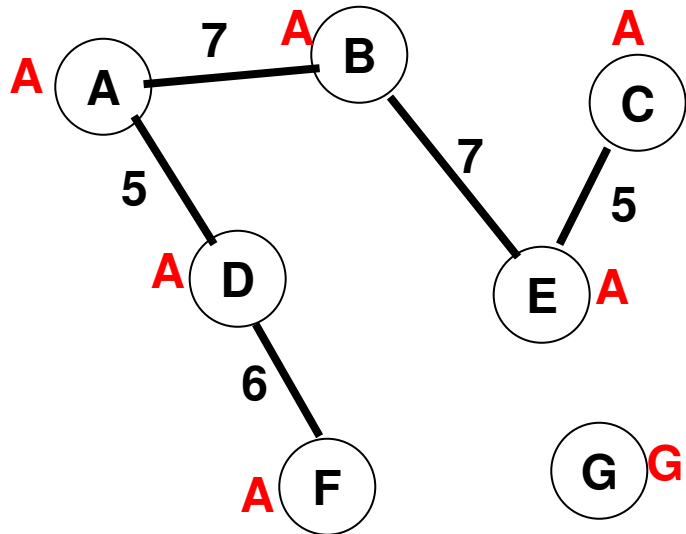
Iteration 2



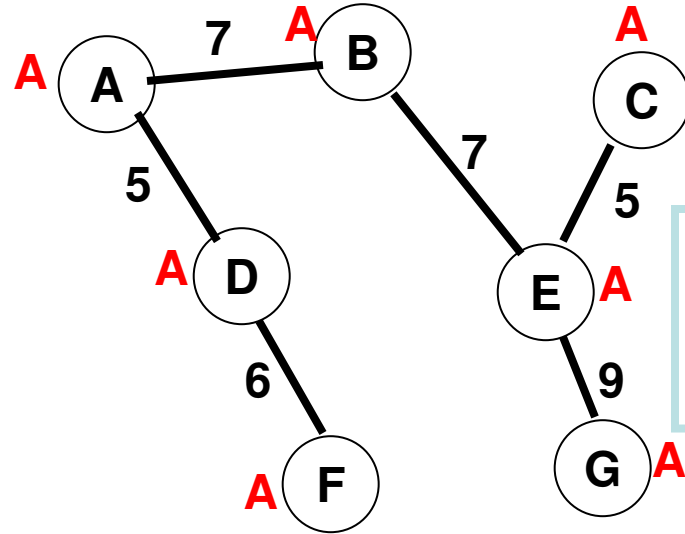
Iteration 3



Iteration 4



Iteration 5



Iteration 6: Min. Sp Tree

MST
Weight
39

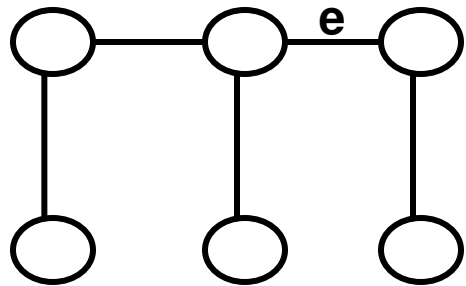
Proof of Correctness: Kruskal's Algorithm

- Let T be the spanning tree generated by Kruskal's algorithm for a graph G . Let T' be a minimum spanning tree for G . We need to show that both T and T' have the same weight.
- Assume that $\text{wt}(T') < \text{wt}(T)$.
- Hence, there should be an edge e in T that is not in T' and likewise there should be an edge e' in T' that is not in T . Because, if every edge of T is in T' , then $T = T'$ and $\text{wt}(T) = \text{wt}(T')$.
- Remove the edge e' that is in T' . This would disconnect the T' to two components. The edge e that was in T and not in T' should be one of the edges (along with e') that cross the two split components of T' .
- Depending on how Kruskal's algorithm works, $\text{wt}(e) \leq \text{wt}(e')$. Hence, the two components of T' could be merged using edge e (instead of e') and this would only lower the weight of T' from what it was before (and not increase it).
- That is, $\text{wt}(\text{modified } T') = \text{wt}(T' - \{e'\} \cup \{e\}) \leq \text{wt}(T')$.
- We could repeat the above procedure for all edges that are in T' and not in T , and eventually transform T' to T , without increasing the cost of the spanning tree.
- Hence, T is a minimum spanning tree.

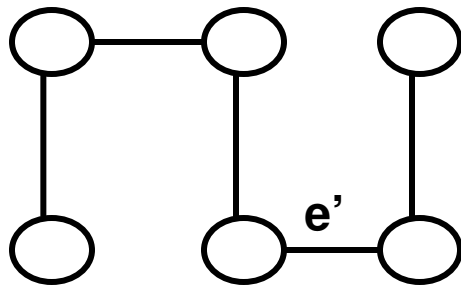
Proof of Correctness

Let T be the spanning tree determined using Kruskal's

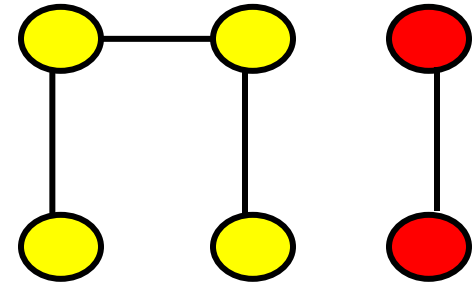
Let T' be a hypothetical spanning tree that is a MST such that $W(T') < W(T)$



T

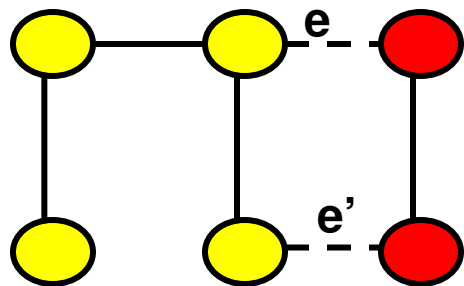


T'

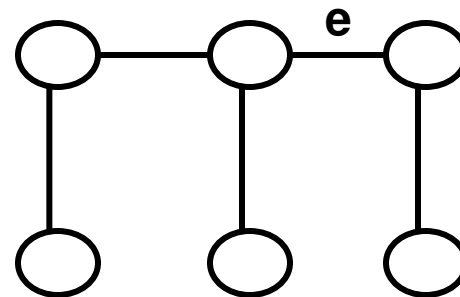


$$Wt(e) \leq Wt(e')$$

$Wt(T' - \{e'\} \cup \{e\}) \leq Wt(T')$. Hence, by reducing the edge difference and making T' approach T , we are able to only decrease the weight of T' further, if possible, making T' not a MST to start with, a contradiction.



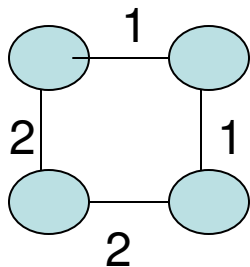
Candidate edges to merge the two components



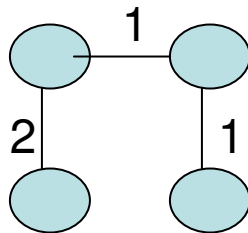
Modified $T' = T' - \{e'\} \cup \{e\}$

Properties of Minimum Spanning Tree

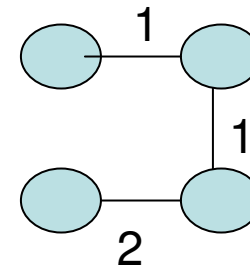
- **Property 2:** If a graph does not have unique edge weights, there could be more than one minimum spanning tree for the graph.
- **Proof (by Example)**



Graph



One Min. Spanning Tree



Another Min. Spanning Tree

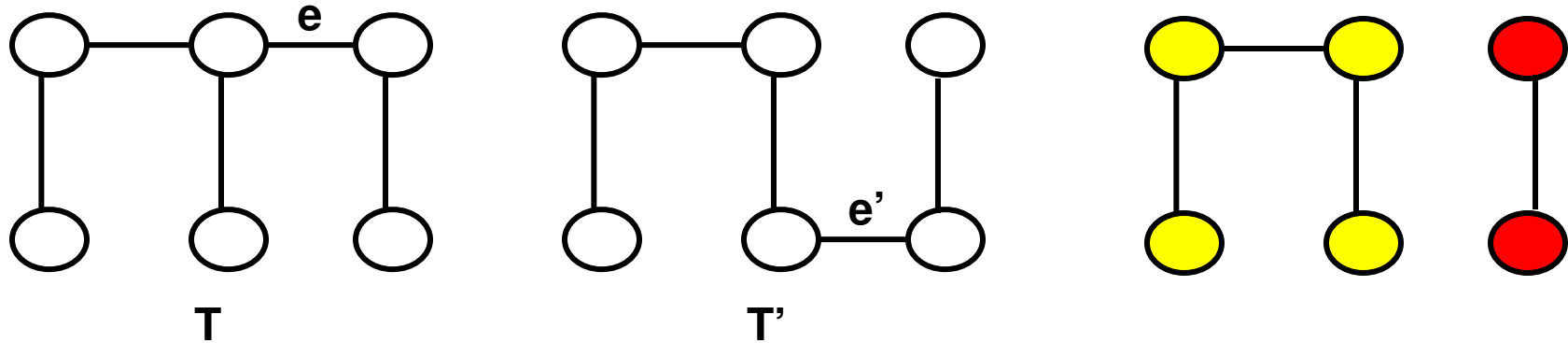
- **Property 3:** If all the edges in a weighted graph have unique weights, then there can be only one minimum spanning tree of the graph.
- **Proof:** Consider a graph G whose edges are of distinct weights. Assume there are two different spanning trees T and T' , both are of minimum weight; but have at least one edge difference. Let e' be an edge in T' that is not in T . Removing e' from T' will split the latter into two components. There should be an edge e that is not part of T' but part of T and should also be a candidate edge to connect the two components of the split T' .

Properties of Minimum Spanning Tree

- **Property 3:** If all the edges in a weighted graph have unique weights, then there can be only one minimum spanning tree of the graph.
- **Proof (continued..):** If $wt(e) < wt(e')$, then we could merge the two components of T' using e and this would lower the weight of T' from what it was before. Hence, $wt(e) \geq wt(e')$.
- However, since the graph has unique edge weights, $wt(e) > wt(e')$. But, if this is the case, then we could indeed remove e from T and have e' to merge the two components of T resulting from the removal of e . This would only lower the weight of T from what it was before.
- So, if T and T' have to be two different MSTs $\rightarrow wt(e) = wt(e')$.
 - This is a contradiction to the given statement that the graph has unique edge weights.
- Not $(wt(e) = wt(e')) \rightarrow$ Not (T and T' have to be two different MSTs)
- That is, $wt(e) \neq wt(e') \rightarrow T$ and T' have to be the same MST.
- Hence, if a graph has unique edge weights, there can be only one MST for the graph.

Property 3

Assume that both T and T' are MSTs, but different MSTs to start with.



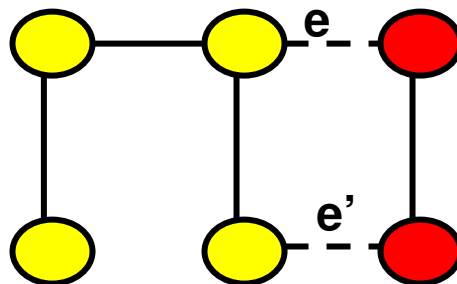
$W(e) < W(e') \Rightarrow T'$ is not a MST

$W(e) > W(e') \Rightarrow T$ is not a MST

Hence, for both T and T' to be two different MSTs $\rightarrow W(e) = W(e')$.

But the graph has unique edge weights.

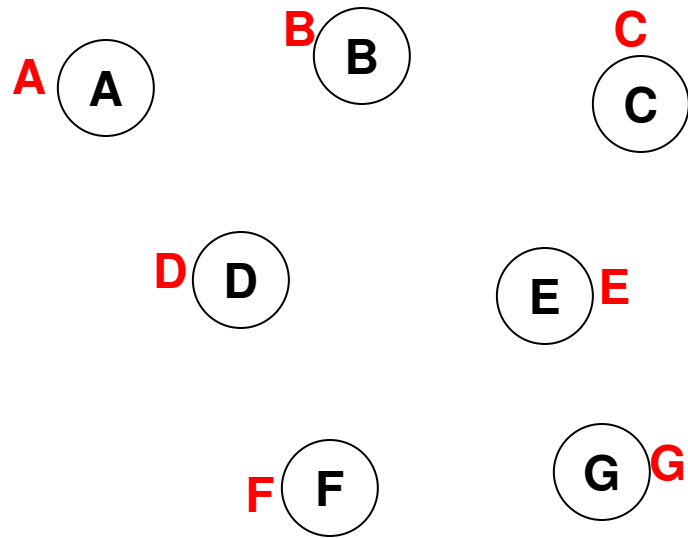
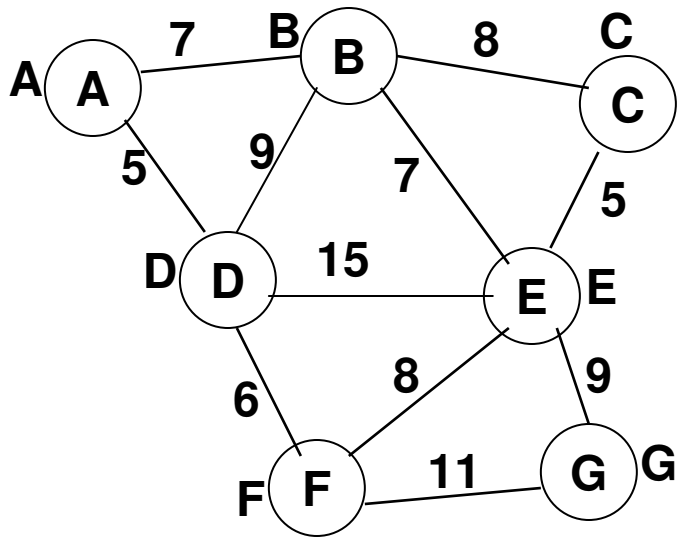
$W(e) \neq W(e) \rightarrow$ Both T and T' have to be the same.



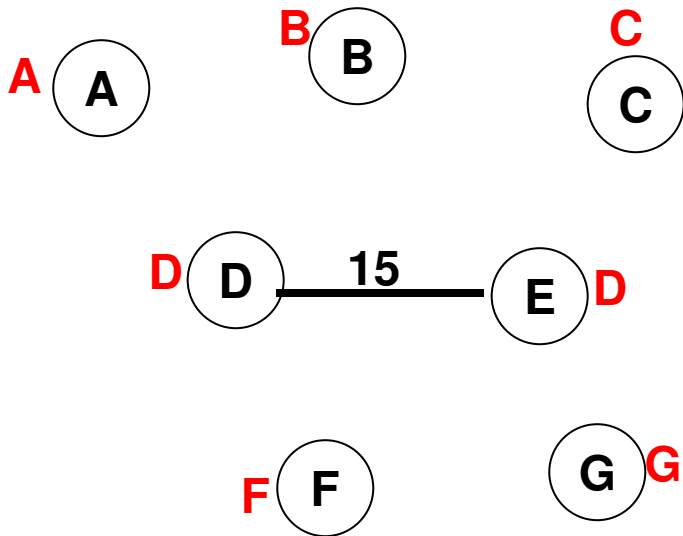
**Candidate edges to merge
the two components**

Maximum Spanning Tree

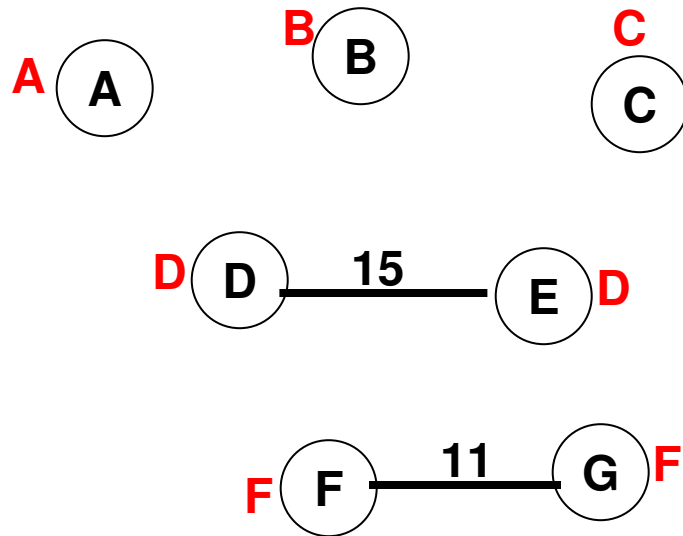
- A Maximum Spanning Tree is a spanning tree such that the sum of its edge weights is the maximum.
- We can find a Maximum Spanning Tree through any one of the following ways:
 - Straightforward approach: Run Kruskal's algorithm by selecting edges in the decreasing order of edge weights (i.e., edge with the largest weight is chosen first) as long as the end vertices of an edge are in two different components
 - Alternate approach (Example for Transform and Conquer): Given a weighted graph, set all the edge weights to be negative, run a minimum spanning tree algorithm on the negative weight graph, then turn all the edge weights to positive on the minimum spanning tree to get a maximum spanning tree.



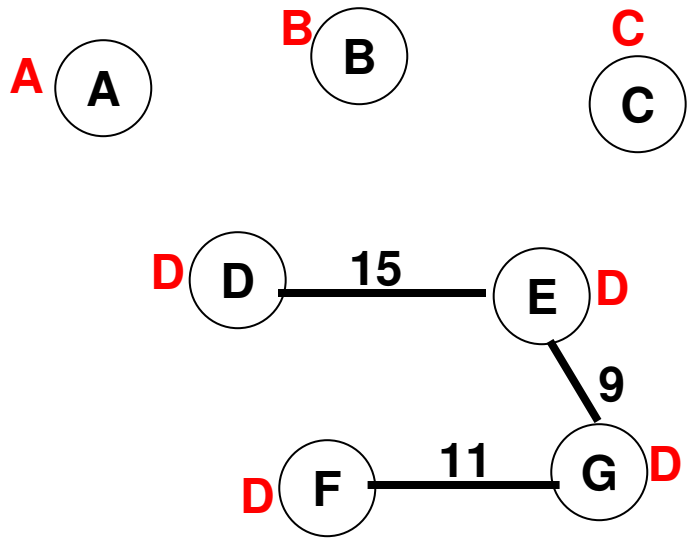
Initialization



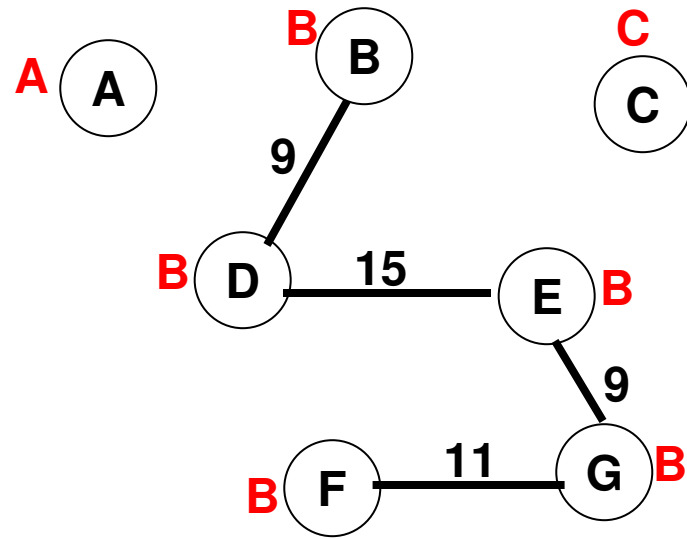
Iteration 1



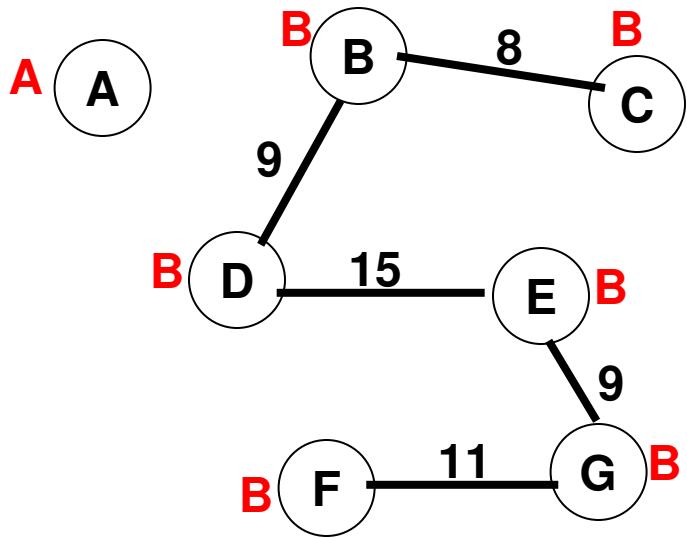
Iteration 2



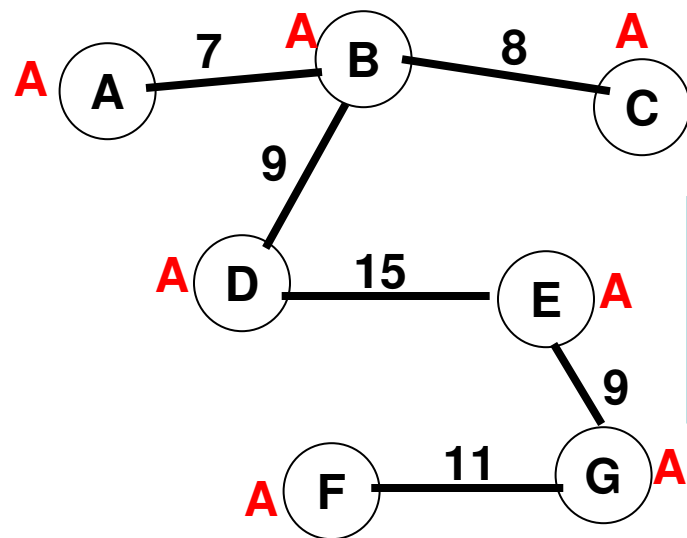
Iteration 3



Iteration 4



Iteration 5



Iteration 6: Max. Sp Tree

MST
Weight
59

Practice Proofs

- Similar to the proof of correctness that we saw for the Minimum Spanning Trees, write the proof of correctness for the Kruskal's algorithm to find Maximum Spanning Trees.
- Prove the following property: If all the edges in a weighted graph have unique weights, then there can be only one maximum spanning tree of the graph.

Dijkstra's Shortest Path Algorithm

Shortest Path (Min. Wt. Path) Problem

- Path p of length k from a vertex s to a vertex d is a sequence $(v_0, v_1, v_2, \dots, v_k)$ of vertices such that $v_0 = s$ and $v_k = d$ and $(v_{i-1}, v_i) \in E$, for $i = 1, 2, \dots, k$

- Weight of a path $p = (v_0, v_1, v_2, \dots, v_k)$ is $w(p) = \sum_{i=1}^k w(v_{i-1}, v_i)$

- The weight of a shortest path from s to d is given by
$$\delta(s, d) = \begin{cases} \min \{w(p) : s \xrightarrow{p} d \text{ if there is a path from } s \text{ to } d\} \\ \infty & \text{otherwise} \end{cases}$$

:

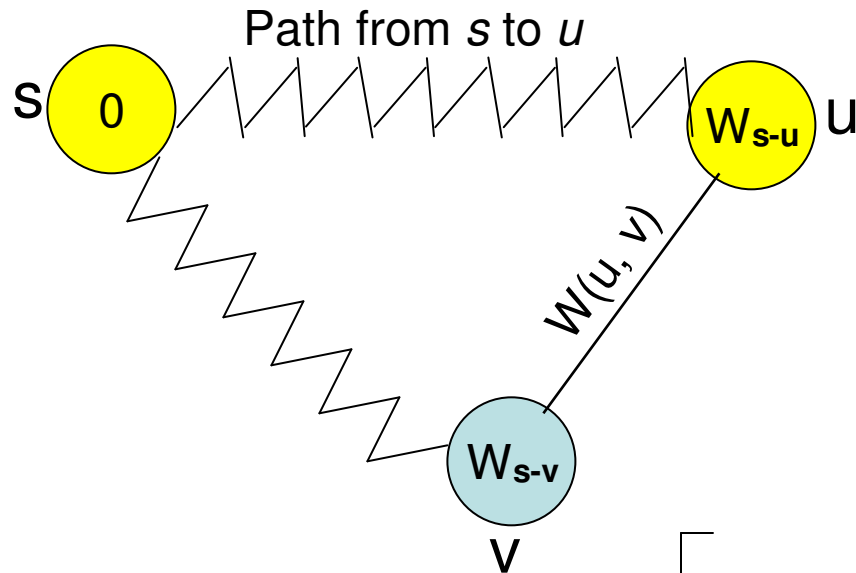
Dijkstra Algorithm

- **Assumption:** $w(u, v) > 0$ for each edge $(u, v) \in E$ (i.e., the edge weights are positive)
- **Objective:** Given $G = (V, E, w)$, find the shortest weight path between a given source s and destination d
- **Principle:** Greedy strategy
- Maintain a minimum weight path estimate $d[v]$ from s to each other vertex v .
- At each step, pick the vertex that has the smallest minimum weight path estimate
- **Output:** After running this algorithm for $|V|$ iterations, we get the shortest weight path from s to all other vertices in G
- **Time Complexity:** Dijkstra algorithm – $\Theta(|E| \log |V|)$

Dr. Meg's YouTube Video Explanation:

<https://www.youtube.com/watch?v=V8VxK1cr0x0>

Principle of Dijkstra Algorithm



Relaxation Condition

If $W_{s-v} > W_{s-u} + W(u, v)$ then

$$W_{s-v} = W_{s-u} + W(u, v)$$

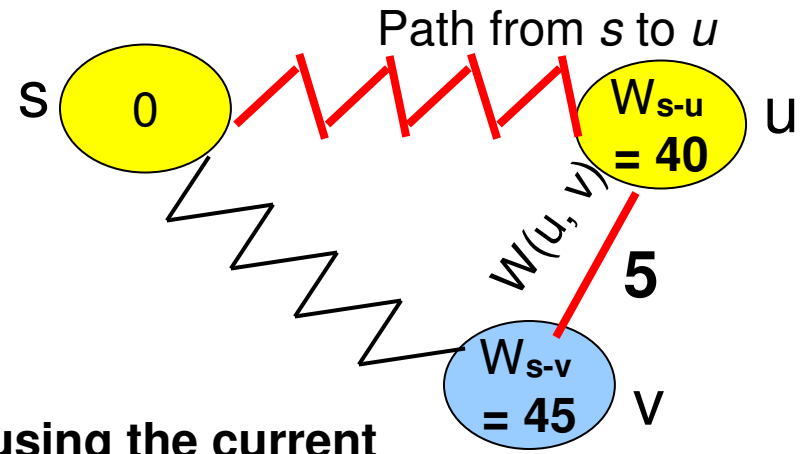
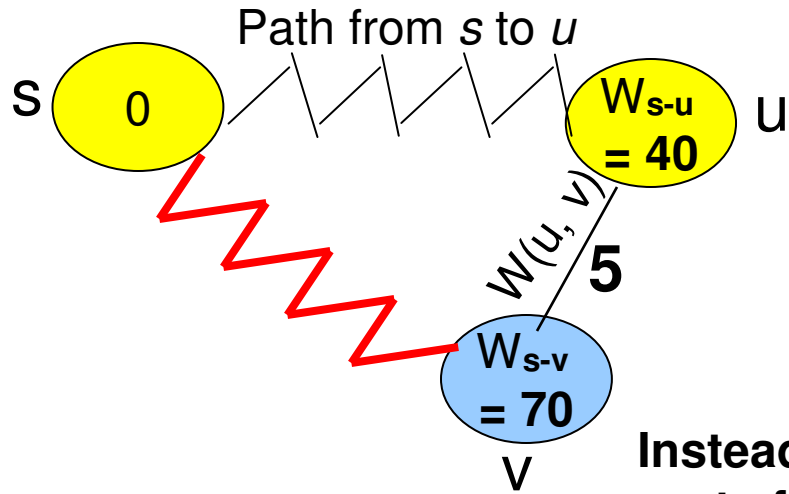
$$\text{Predecessor}(v) = u$$

else

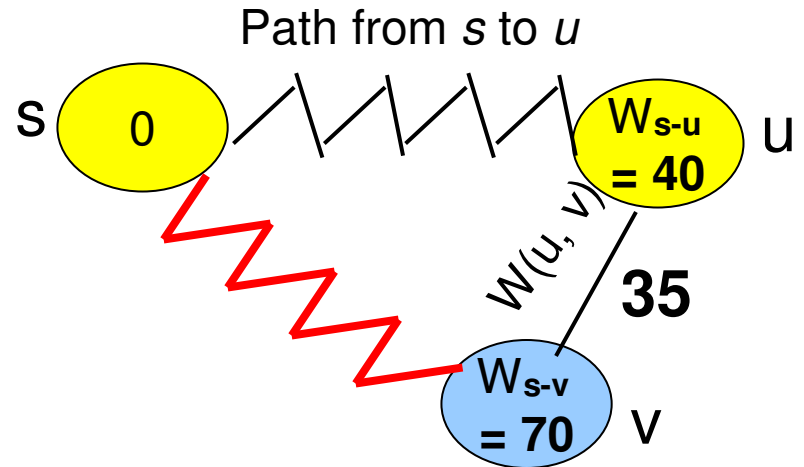
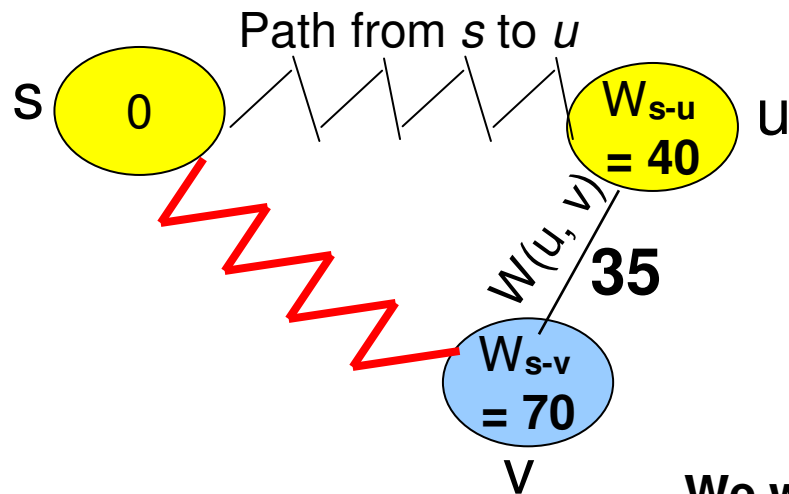
Retain the current path from s to v

Principle in a nutshell

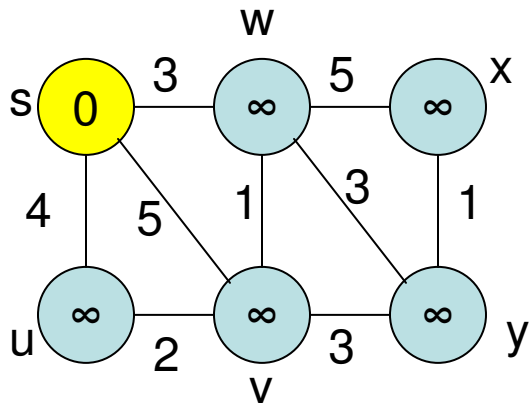
During the beginning of each iteration we will pick a vertex u that has the minimum weight path to s . We will then explore the neighbors of u for which we have not yet found a minimum weight path. We will try to see if by going through u , we can reduce the weight of path from s to v , where v is a neighbor of u .



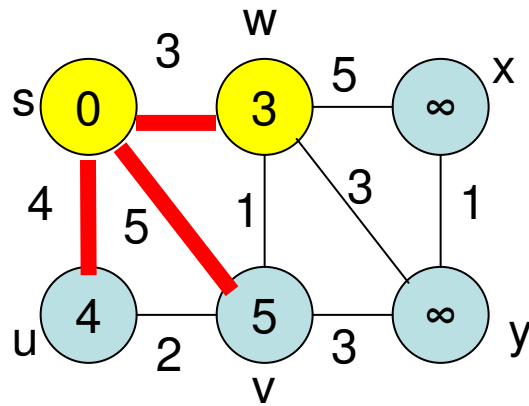
Instead of using the current route from s to v , we will go through u to reach v from s



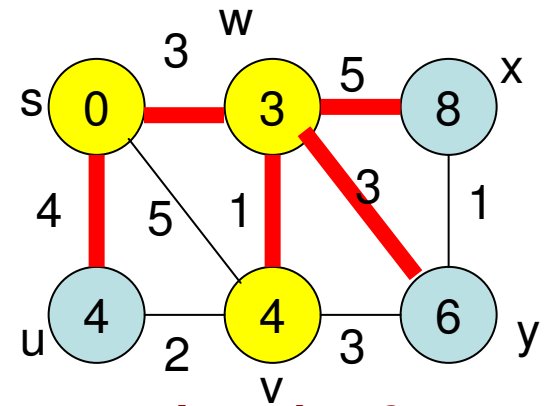
We will stay with the current route we know from s to v .



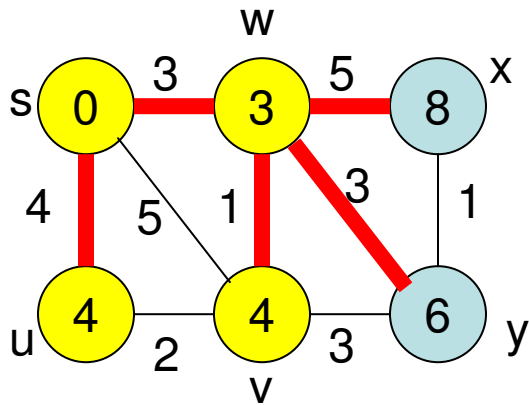
Given Graph, Initialization



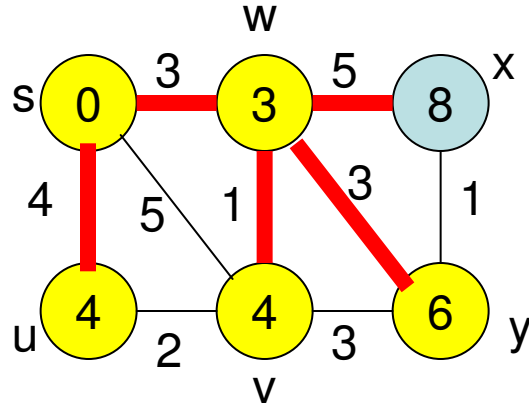
Iteration 1



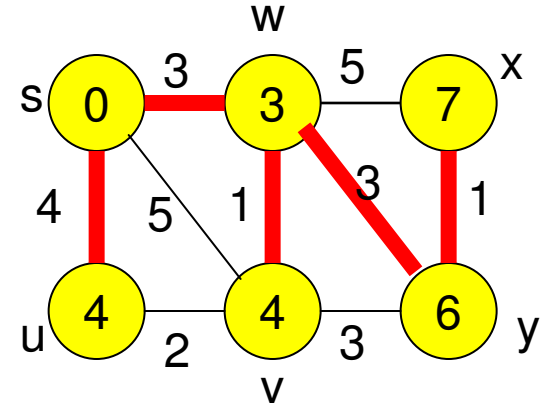
Iteration 2



Iteration 3



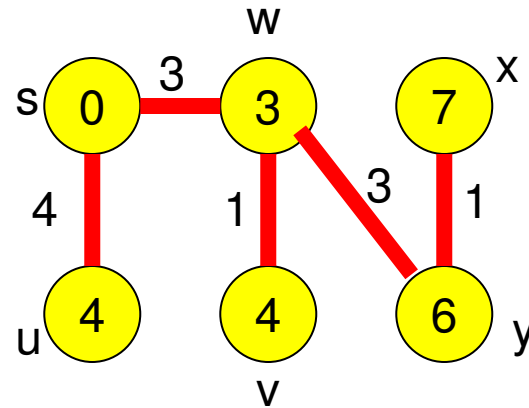
Iteration 4

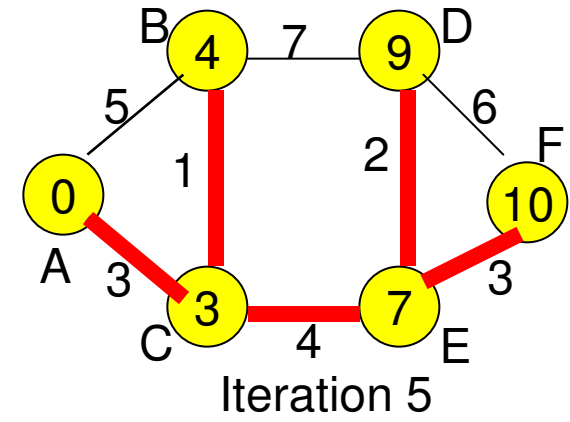
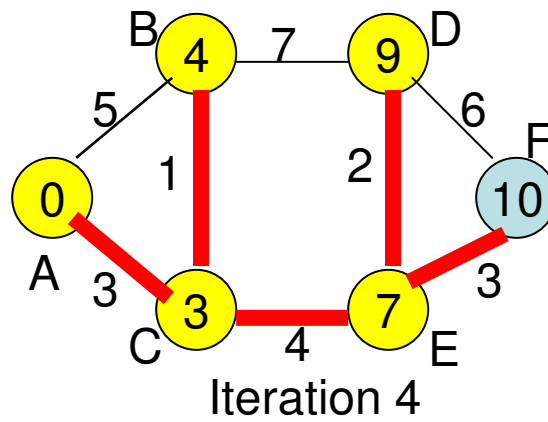
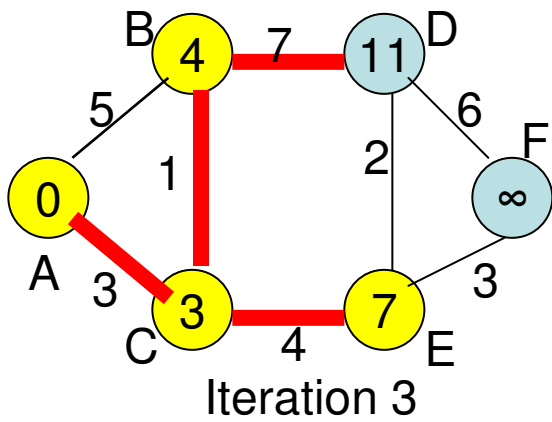
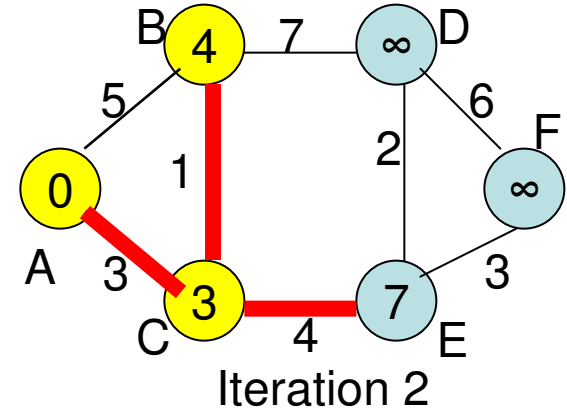
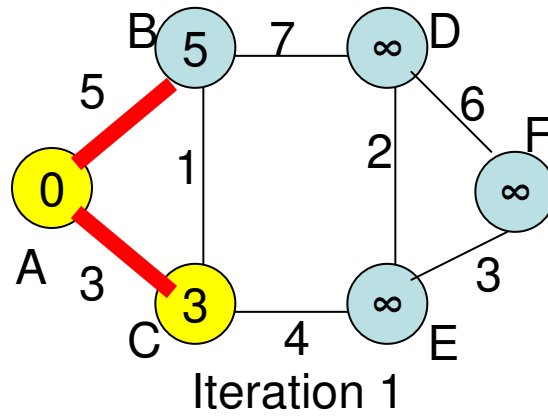
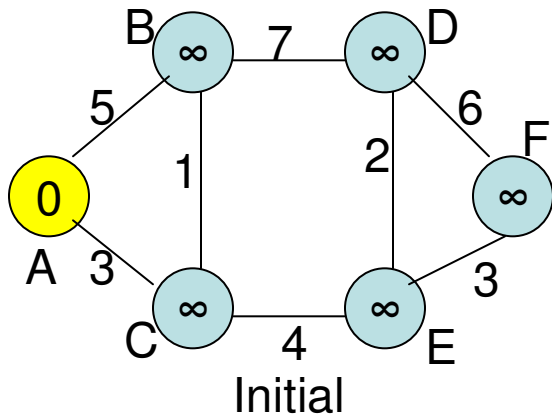


Iteration 5

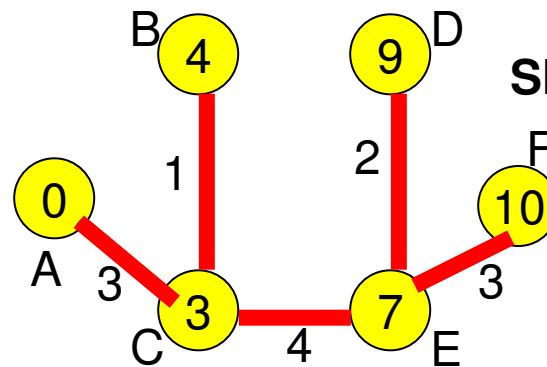
**Dijkstra Algorithm
Example 1**

Shortest Path Tree





Dijkstra Algorithm Example 2



Dijkstra Algorithm

Begin Algorithm *Dijkstra* (G, s)

1 **For** each vertex $v \in V$

2 $d[v] \leftarrow \infty$ // an estimate of the min-weight path from s to v

3 **End For**

4 $d[s] \leftarrow 0$

5 $S \leftarrow \Phi$ // set of nodes for which we know the min-weight path from s

6 $Q \leftarrow V$ // set of nodes for which we know estimate of min-weight path from s

7 **While** $Q \neq \Phi$

8 $u \leftarrow \text{EXTRACT-MIN}(Q)$

9 $S \leftarrow S \cup \{u\}$

10 **For** each vertex v such that $(u, v) \in E$

11 **If** $v \in Q$ and $d[v] > d[u] + w(u, v)$ then

12 $d[v] \leftarrow d[u] + w(u, v)$

13 Predecessor(v) = u

13 **End If**

14 **End For**

15 **End While**

16 **End** *Dijkstra*

Dijkstra Algorithm: Time Complexity

Begin Algorithm *Dijkstra* (G, s)

```

1  For each vertex  $v \in V$ 
2       $d[v] \leftarrow \infty$  // an estimate of the min-weight path from  $s$  to  $v$ 
3  End For
4   $d[s] \leftarrow 0$ 
5   $S \leftarrow \emptyset$  // set of nodes for which we know the min-weight path from  $s$ 
6   $Q \leftarrow V$  // set of nodes for which we know estimate of min-weight path from  $s$ 
7  While  $Q \neq \emptyset$  done  $|V|$  times =  $\Theta(V)$  time
8       $u \leftarrow \text{EXTRACT-MIN}(Q)$  done Each extraction takes  $\Theta(\log V)$  time
9       $S \leftarrow S \cup \{u\}$ 
10     For each vertex  $v$  such that  $(u, v) \in E$  done  $\Theta(E)$  times totally
11         If  $v \in Q$  and  $d[v] > d[u] + w(u, v)$  then
12              $d[v] \leftarrow d[u] + w(u, v)$ 
13             Predecessor( $v$ ) =  $u$ 
14         End If
15     End For
16 End While
17 End Dijkstra

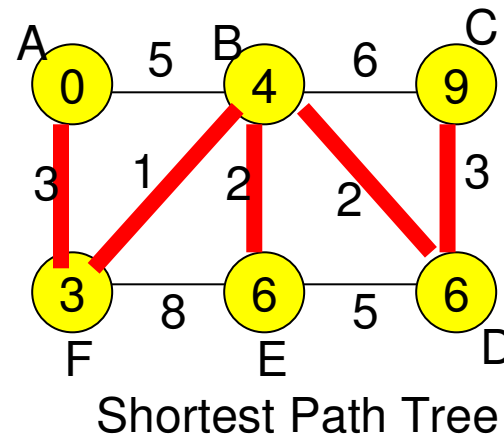
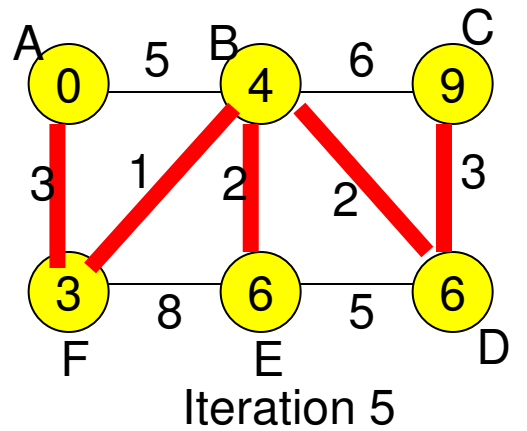
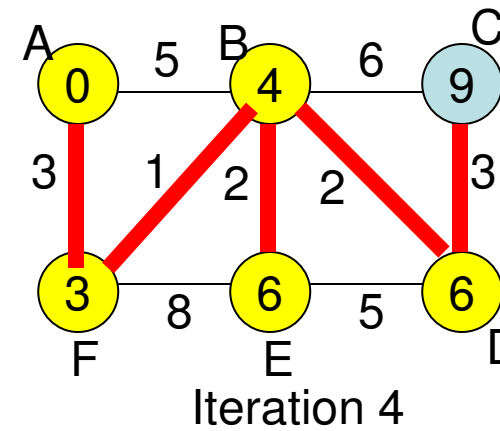
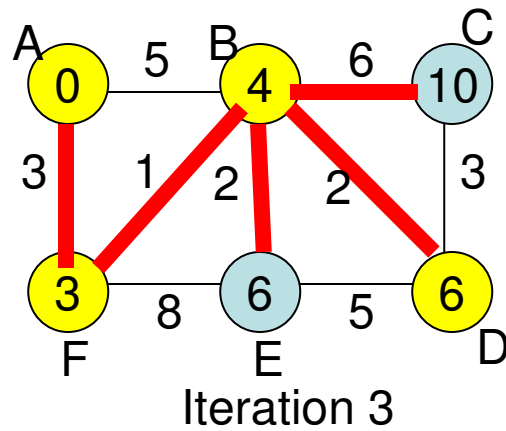
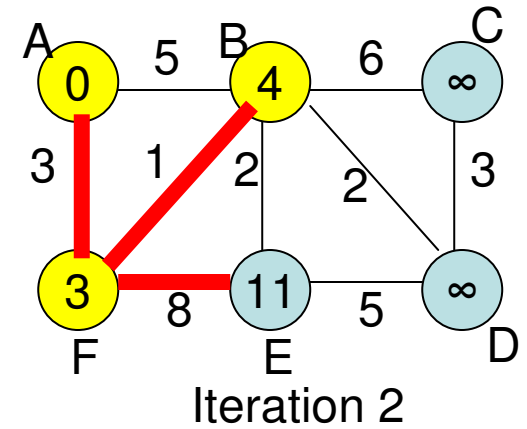
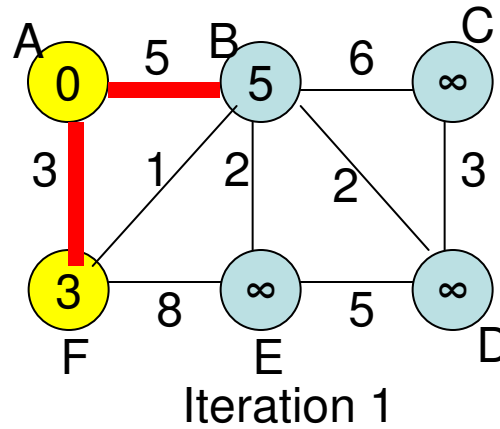
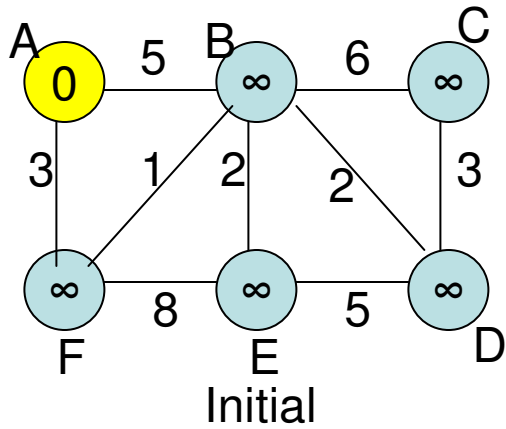
```

$\Theta(V)$ time

$\Theta(V)$ time to Construct a Min-heap

It takes $\Theta(\log V)$ time when done once

Overall Complexity: $\Theta(V) + \Theta(V) + \Theta(V \log V) + \Theta(E \log V)$
 Since the $|E| \geq |V| - 1$, the $V \log V$ term is dominated by the $E \log V$ term. Hence, overall complexity = $\Theta(|E| \cdot \log |V|)$



Dijkstra Algorithm Example 3

Theorems on Shortest Paths and Dijkstra Algorithm

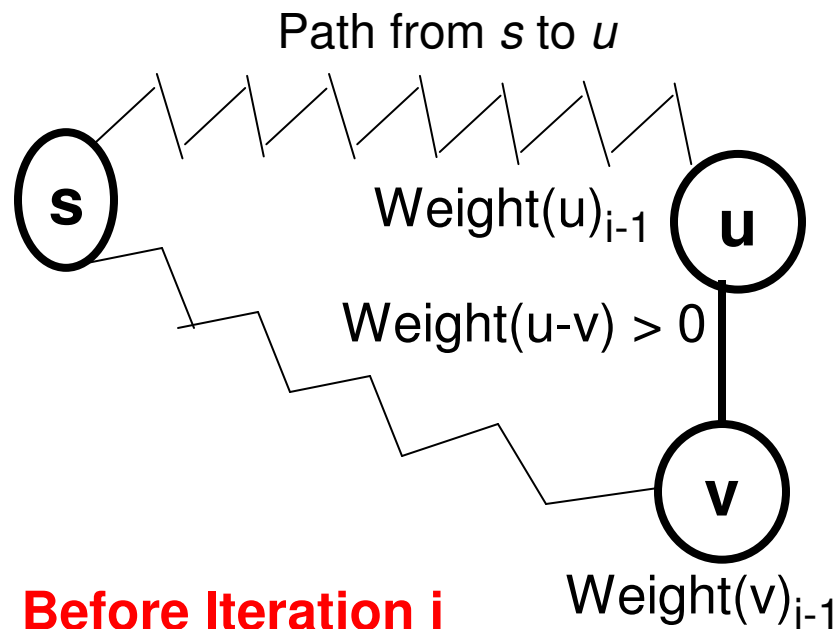
- **Theorem 1:** Sub path of a shortest path is also shortest.
- **Proof:** Lets say there is a shortest path from s to d through the vertices $s - a - b - c - d$.
- Then, the shortest path from a to c is also $a - b - c$.
- If there is a path of lower weight than the weight of the path from $a - b - c$, then we could have gone from s to d through this alternate path from a to c of lower weight than $a - b - c$.
- However, if we do that, then the weight of the path $s - a - b - c - d$ is not the lowest and there exists an alternate path of lower weight.
- This contradicts our assumption that $s - a - b - c - d$ is the shortest (lowest weight) path.

Theorems on Shortest Paths and Dijkstra Algorithm

- **Theorem 2**: The weights of the vertices that are optimized are in the non-decreasing (i.e., typically increasing) order.
- **Proof**: We want to prove that if a vertex u is optimized in an earlier iteration (say iteration i), then the weight of the vertex v optimized at a later iteration (say iteration j ; $i < j$) is always greater than or equal to that of vertex u .
- Vertex v could be either a neighbor of vertex u or not. In either case, the $\text{weight}(v)_{i-1} \geq \text{weight}(u)_{i-1}$ during the beginning of iteration i as vertex u was considered to have been optimized instead of vertex v during this iteration.
- During iteration i : we relax the neighbors of vertex u
 - If vertex v is a neighbor of vertex u , $\text{weight}(v)_i$ could have become less than $\text{weight}(v)_{i-1}$, but $\text{weight}(v)_i$ could never become $\text{weight}(u)_{i-1}$ as all edge weights are positive (including the weight of the edge $u-v$). Hence, $\text{weight}(v)_i$ could have become $\text{weight}(u)_{i-1} + \text{weight}(u-v)$, but it will still be only less than $\text{weight}(u)_{i-1}$, as $\text{weight}(u-v) > 0$.
- If vertex v is not a neighbor of vertex u , then vertex v should ultimately get optimized through some neighbor x (that is not u). But all such neighbors x should have $\text{weight}(x)_{i-1} \geq \text{weight}(u)_{i-1}$, as x was not picked for optimization in iteration i . Hence, by going through such neighbors x , the $\text{weight}(v)$ during iterations i or later, could never become still less than $\text{weight}(u)_{i-1}$, as all the edge weights $w(x-v)$ are greater than 0.

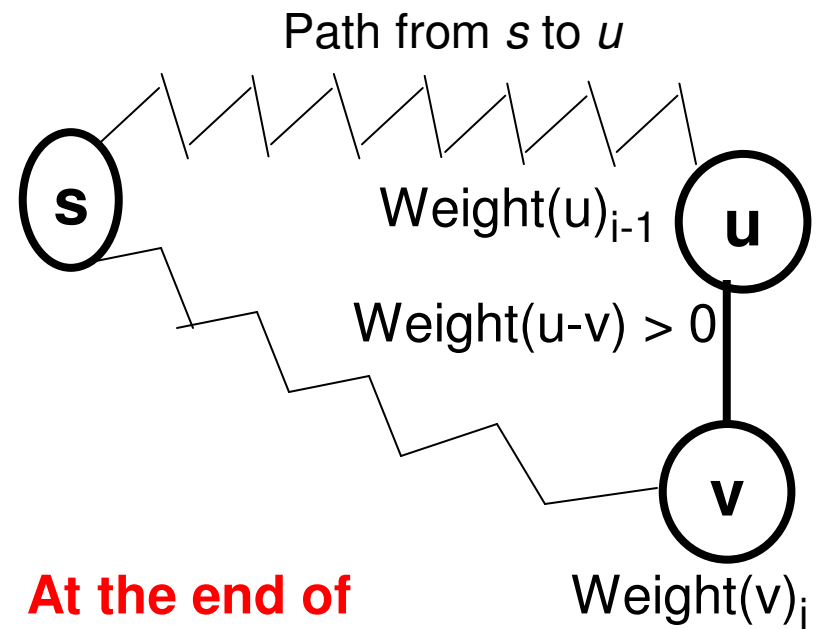
Proof for Theorem 2

Scenario: Vertex v is a neighbor of Vertex u



**Before Iteration i
(at the end of
Iteration $i-1$)**

$$Weight(v)_{i-1} \geq Weight(u)_{i-1}$$

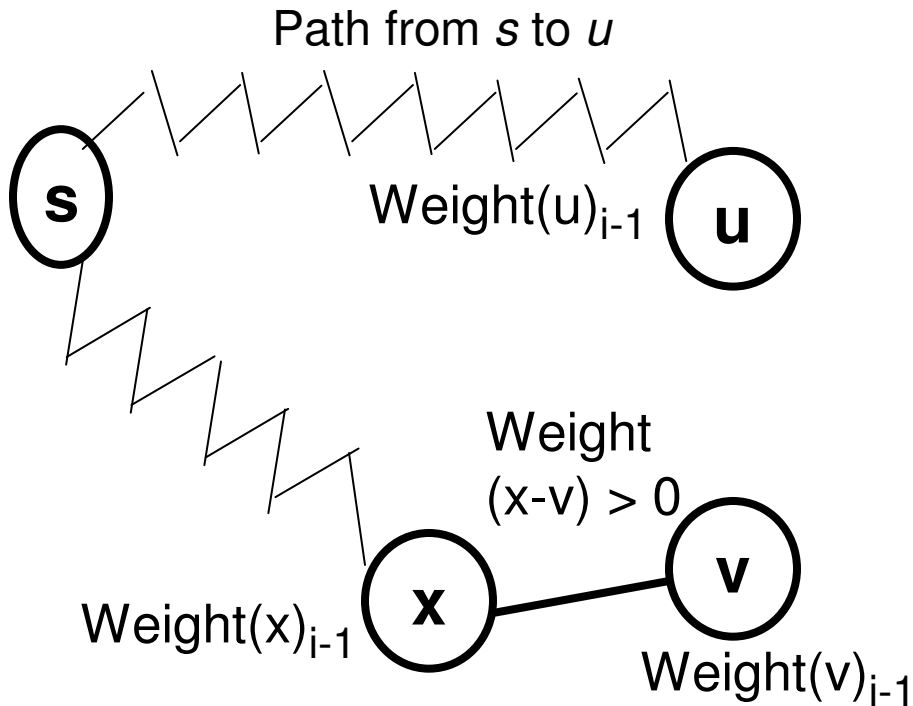


**At the end of
Iteration i**

$$Weight(v)_i \geq Weight(u)_{i-1}$$

Proof for Theorem 2

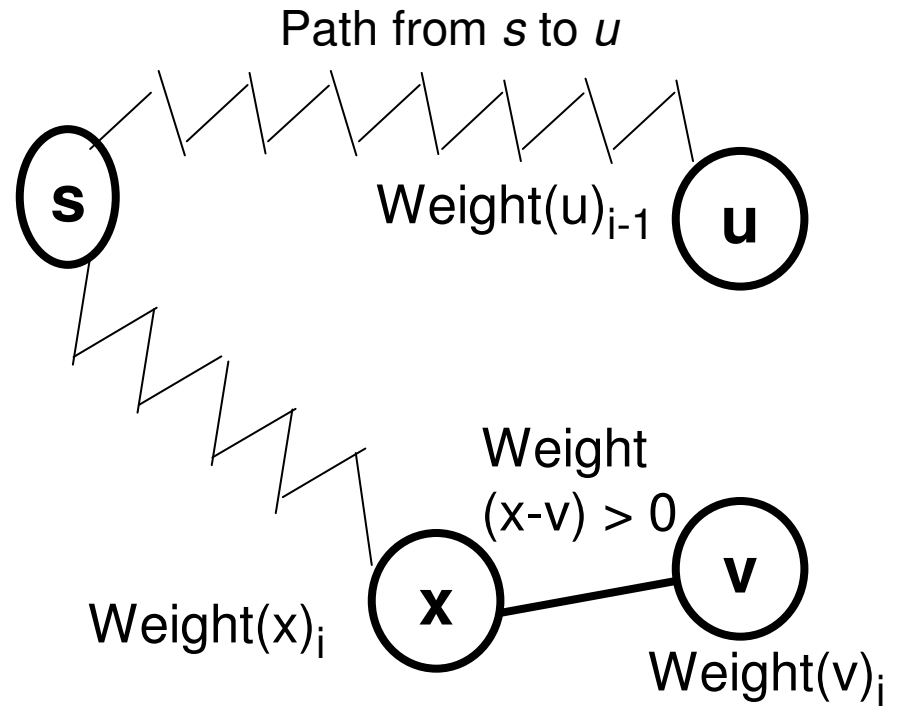
Scenario: Vertex v is NOT a neighbor of Vertex u , but a neighbor of some other vertex x



Before Iteration i (at the end of Iteration $i-1$)

$$\text{Weight}(x)_{i-1} \geq \text{Weight}(u)_{i-1}$$

$$\text{Weight}(v)_{i-1} \geq \text{Weight}(u)_{i-1}$$



At the end of Iteration i

$$\text{Weight}(x)_i \geq \text{Weight}(u)_{i-1}$$

$$\text{Weight}(v)_i \geq \text{Weight}(u)_{i-1}$$

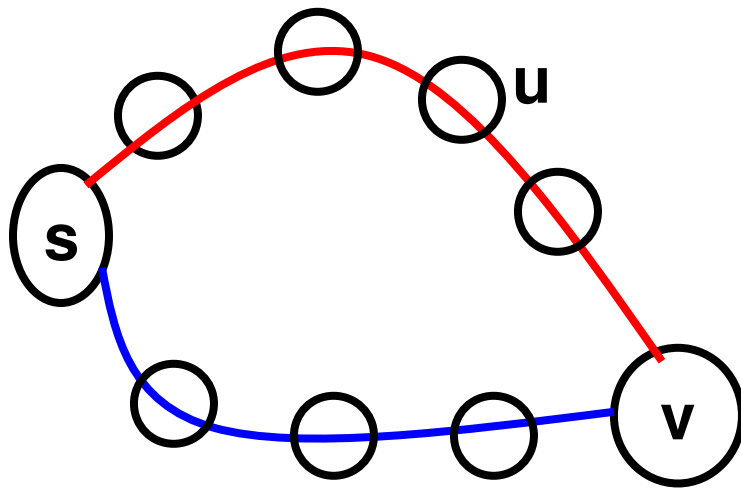
Theorems on Shortest Paths and Dijkstra Algorithm

- **Theorem 3:** When a vertex v is picked for relaxation/optimization, every intermediate vertex on the $s \dots v$ shortest path is already optimized.
- **Proof:** Let there be a path from s to v that includes a vertex x (i.e., $s \dots x \dots v$) for which we have not yet found the shortest path.
- From Theorem 1, shortest path weight($s \dots x$) $<$ shortest path weight($s \dots v$).
- From Theorem 2, vertices are optimized in the non-decreasing order of shortest path weights.
- So, if vertex v is picked for optimization based on the path $s \dots x \dots v$, then the intermediate vertex x should have been already picked (before v) for optimization. A contradiction.

- **Theorem 4:** When a vertex v is picked for relaxation, we have optimized the vertex (i.e., found the shortest path for the vertex from a source vertex s).
- **Proof:** Let P be the path from source s to vertex v based on whose weight we decide to relax the vertex. We want to prove P is the optimal path of minimum weight from s to v . We will prove this by contradiction.
- Let P' be a hypothetical shortest path from s to v such that $w(P') < w(P)$
- If all the intermediate vertices from s to v on the path P' are already optimized, we would have indeed found the shortest path from s to v of weight $w(P')$.
- If P' is not chosen and P is chosen by Dijkstra algorithm for optimizing vertex v , then there should be at least one intermediate vertex (say vertex ' u ') on the path P' from s to v that is not yet optimized (and because of this we were not able to optimize v from s on path P').
- From the earlier Theorems, the $\text{weight}(s \dots u \text{ in } P') \geq \text{weight}(s \dots v \text{ in } P)$ because the algorithm picks vertices for optimization in the non-decreasing (i.e., increasing) order of shortest path weights.
 - So, even if vertex u on path P' is chosen for optimization after vertex v on path P , the weight of the $s \dots u \dots v$ path (P') would be only larger than that of the $s \dots v$ path (P). Hence, a contradiction.
- Thus, the path P found by Dijkstra algorithm is the shortest path from the source s to a vertex v .

Proof for Theorem 4 (by Contradiction)

**Hypothetical Path P' that
We assume:
 $\text{Weight}(s\dots v)_{P'} < \text{Weight}(s\dots v)_P$**



**Path P found by
Dijkstra algorithm**

From Theorem 3,

If P' is an optimal path from s to v , then all the intermediate vertices on the path should have been already optimized, and as a result of the accompanying relaxations, we would have traced the path P' from s to v as the optimal path instead of the path P . Hence, if the algorithm did not pick P' as the optimal path, there should be some intermediate vertex u on the path P' that is not yet optimized and all the subsequent vertices on the path P' are not optimized either.

From Theorem 2,

$\text{Weight}(s\dots u)_{P'} \geq \text{Weight}(s\dots v)_P$

From Theorem 1,

$\text{Weight}(s\dots u\dots v)_{P'} > \text{Weight}(s\dots u)_{P'}$

Hence:

$\text{Weight}(s\dots u\dots v)_{P'} > \text{Weight}(s\dots v)_P$

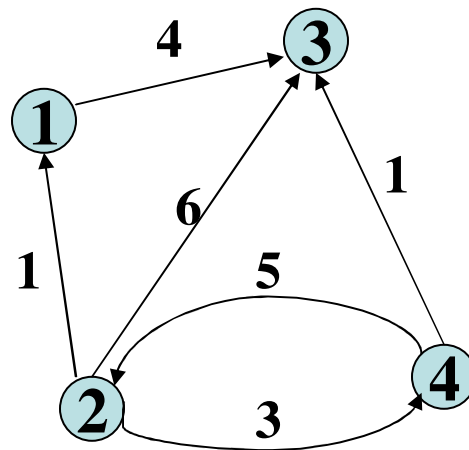
All Pairs Shortest Paths Problem

Dynamic Programming Algorithm for All Pairs Shortest Paths

Problem: In a weighted (di)graph, find shortest paths between every pair of vertices

idea: construct solution through series of matrices $D^{(0)}, \dots, D^{(n)}$ using increasing subsets of the vertices allowed as intermediate

Example:



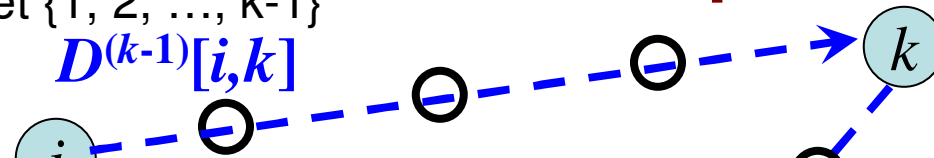
The algorithm we are going to see was developed by two people Floyd and Warshall. We will shortly refer to the algorithm as the FW algorithm

FW Algorithm: Operating Principle

- **Operating Principle:** The vertices are numbered from 1 to n. There are 'n' iterations. In the kth iteration, the candidate set of vertices available to choose from as intermediate vertices are {1, 2, 3, ..., k}.
- **Initialization:** No vertex is a candidate intermediate vertex. There is a path between two vertices only if there is a direct edge between them (i.e., $i \rightarrow j$); otherwise, not.
- **Iteration 1:** Candidate intermediate vertex {1}. Hence, the candidate paths to choose from are (depending on the graph, the following two may be true):
 $i \rightarrow j$ (or) $i \rightarrow 1 \rightarrow j$
- **Iteration 2:** Candidate intermediate vertices {1, 2}. Hence, the candidate paths to choose from are (depending on the graph; the following in an exhaustive list for a complete graph in case of a brute force approach):
 $i \rightarrow j$ (or) $i \rightarrow 1 \rightarrow j$ (or) $i \rightarrow 2 \rightarrow j$ (or) $i \rightarrow 1 \rightarrow 2 \rightarrow j$ (or) $i \rightarrow 2 \rightarrow 1 \rightarrow j$
- **Iteration 3:** Candidate intermediate vertices {1, 2, 3}. Hence, the candidate paths to choose from are (depending on the graph; the following in an exhaustive list for a complete graph in case of a brute force approach):
 $i \rightarrow j$ (or) $i \rightarrow 1 \rightarrow j$ (or) $i \rightarrow 2 \rightarrow j$ (or) $i \rightarrow 3 \rightarrow j$ (or) $i \rightarrow 1 \rightarrow 2 \rightarrow j$ (or) $i \rightarrow 2 \rightarrow 1 \rightarrow j$ (or) $i \rightarrow 1 \rightarrow 3 \rightarrow j$ (or) $i \rightarrow 3 \rightarrow 1 \rightarrow j$ (or) $i \rightarrow 2 \rightarrow 3 \rightarrow j$ (or) $i \rightarrow 3 \rightarrow 2 \rightarrow j$ (or) $i \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow j$ (or) $i \rightarrow 3 \rightarrow 2 \rightarrow 1 \rightarrow j$ (or) $i \rightarrow 1 \rightarrow 3 \rightarrow 2 \rightarrow j$ (or) $i \rightarrow 3 \rightarrow 1 \rightarrow 2 \rightarrow j$ (or) $i \rightarrow 2 \rightarrow 3 \rightarrow 1 \rightarrow j$ (or) $i \rightarrow 2 \rightarrow 1 \rightarrow 3 \rightarrow j$

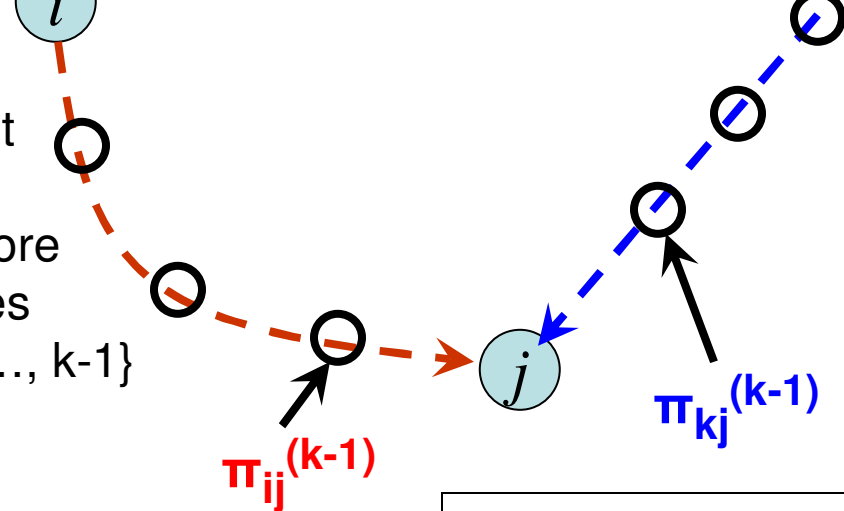
FW Algorithm: Operating Principle

the minimum weight path from i to k involving zero or more intermediate vertices from the set $\{1, 2, \dots, k-1\}$



the minimum weight path from i to j involving zero or more intermediate vertices from the set $\{1, 2, \dots, k-1\}$

$$D^{(k-1)}[i,j]$$



the minimum weight path from k to j involving zero or more intermediate vertices from the set $\{1, 2, \dots, k-1\}$

$$D^{(k-1)}[k,j]$$

$$\pi_{kj}^{(k-1)}$$

$$D^{(0)}[i,j] = w_{ij} \quad \text{if } i \rightarrow j \in E$$

$$D^{(0)}[i,j] = \infty \quad \text{if } i \rightarrow j \notin E$$

$$\pi_{ij}^{(0)} = i \quad \text{if } i \rightarrow j \in E$$

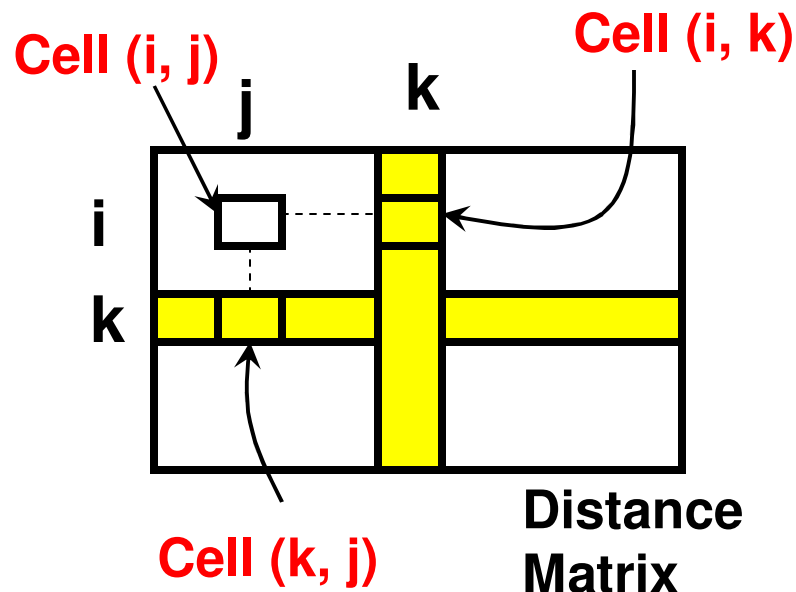
$$\pi_{ij}^{(0)} = N/A \quad \text{if } i \rightarrow j \notin E$$

$$D^{(k)}[i,j] = \min \{ D^{(k-1)}[i,j], D^{(k-1)}[i,k] + D^{(k-1)}[k,j] \}$$

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } D_{ij}^{(k-1)} \leq D_{ik}^{(k-1)} + D_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} & \text{if } D_{ij}^{(k-1)} > D_{ik}^{(k-1)} + D_{kj}^{(k-1)} \end{cases}$$

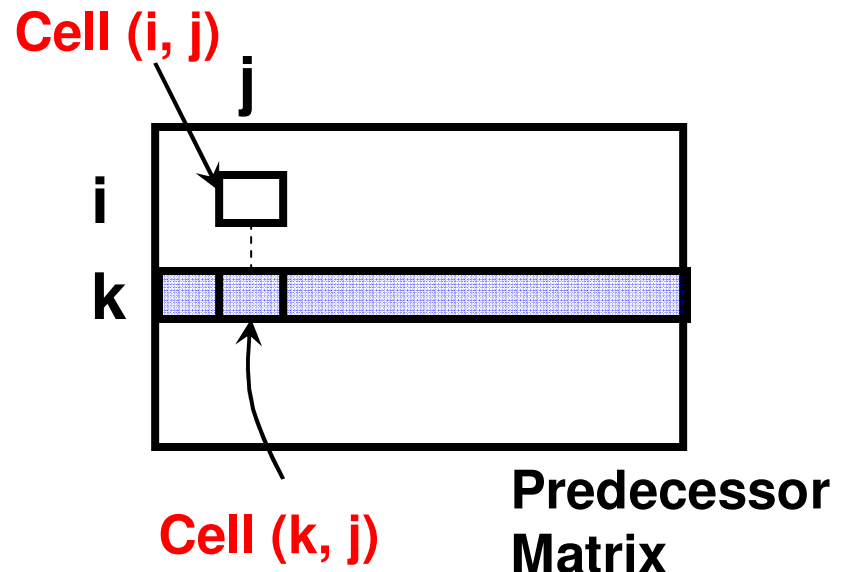
FW Algorithm: Working Principle

- In iteration k , we highlight the row and column corresponding to vertex k , and check whether the values for each of the other cells could be reduced from what they were prior to that iteration. We do not change the values for the cells in the row and column corresponding to vertex k .

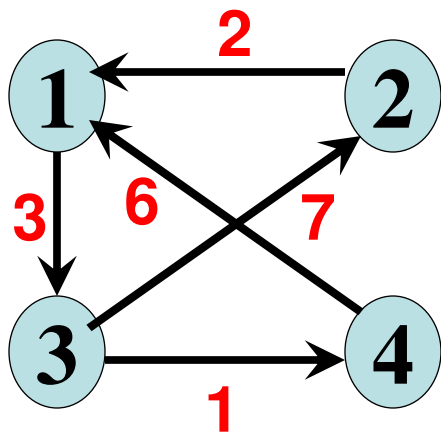


We update a cell (i, j) if the value in the cell is greater than the sum of the Values of the cells (i, k) and (k, j)

If we update cell (i, j) , we also update the predecessor for (i, j) to be the value corresponding to the predecessor for (k, j) in row k .



FW Algorithm: Example 1 (1)



Iteration 1

$D^{(0)}$

| | v1 | v2 | v3 | v4 |
|----|----------|----------|----------|----------|
| v1 | 0 | ∞ | 3 | ∞ |
| v2 | 2 | 0 | ∞ | ∞ |
| v3 | ∞ | 7 | 0 | 1 |
| v4 | 6 | ∞ | ∞ | 0 |

$\Pi^{(0)}$

| | v1 | v2 | v3 | v4 |
|----|-----|-----|-----|-----|
| v1 | N/A | N/A | v1 | N/A |
| v2 | v2 | N/A | N/A | N/A |
| v3 | N/A | v3 | N/A | v3 |
| v4 | v4 | N/A | N/A | N/A |

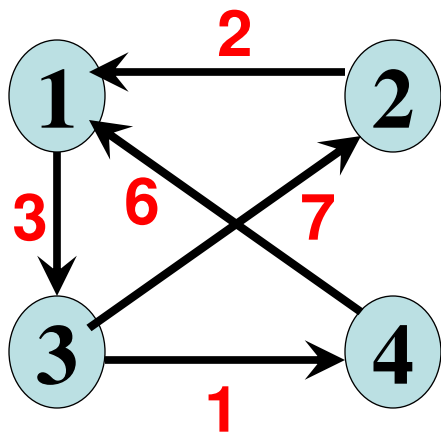
$D^{(1)}$

| | v1 | v2 | v3 | v4 |
|----|----------|----------|----|----------|
| v1 | 0 | ∞ | 3 | ∞ |
| v2 | 2 | | | |
| v3 | ∞ | | | |
| v4 | 6 | | | |

$\Pi^{(1)}$

| | v1 | v2 | v3 | v4 |
|----|-----|-----|----|-----|
| v1 | N/A | N/A | v1 | N/A |
| v2 | | | | |
| v3 | | | | |
| v4 | | | | |

FW Algorithm: Example 1 (1)



Iteration 1

$D^{(0)}$

| | v1 | v2 | v3 | v4 |
|----|----------|----------|----------|----------|
| v1 | 0 | ∞ | 3 | ∞ |
| v2 | 2 | 0 | ∞ | ∞ |
| v3 | ∞ | 7 | 0 | 1 |
| v4 | 6 | ∞ | ∞ | 0 |

$\Pi^{(0)}$

| | v1 | v2 | v3 | v4 |
|----|-----|-----|-----|-----|
| v1 | N/A | N/A | v1 | N/A |
| v2 | v2 | N/A | N/A | N/A |
| v3 | N/A | v3 | N/A | v3 |
| v4 | v4 | N/A | N/A | N/A |

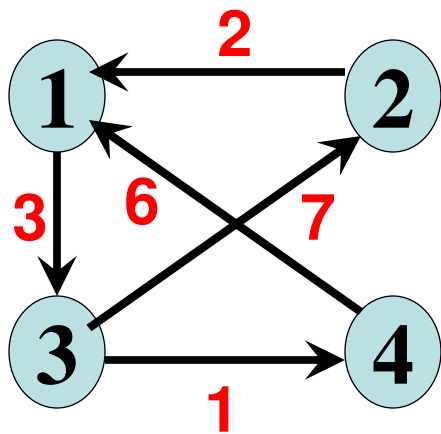
$D^{(1)}$

| | v1 | v2 | v3 | v4 |
|----|----------|----------|----|----------|
| v1 | 0 | ∞ | 3 | ∞ |
| v2 | 2 | 0 | 5 | ∞ |
| v3 | ∞ | 7 | 0 | 1 |
| v4 | 6 | ∞ | 9 | 0 |

$\Pi^{(1)}$

| | v1 | v2 | v3 | v4 |
|----|-----|-----|-----|-----|
| v1 | N/A | N/A | v1 | N/A |
| v2 | v2 | N/A | v1 | N/A |
| v3 | N/A | v3 | N/A | v3 |
| v4 | v4 | N/A | v1 | N/A |

FW Algorithm: Example 1 (2)



Iteration 2

$D^{(1)}$

| | v1 | v2 | v3 | v4 |
|----|----------|----------|----|----------|
| v1 | 0 | ∞ | 3 | ∞ |
| v2 | 2 | 0 | 5 | ∞ |
| v3 | ∞ | 7 | 0 | 1 |
| v4 | 6 | ∞ | 9 | 0 |

$\Pi^{(1)}$

| | v1 | v2 | v3 | v4 |
|----|-----|-----|-----|-----|
| v1 | N/A | N/A | v1 | N/A |
| v2 | v2 | N/A | v1 | N/A |
| v3 | N/A | v3 | N/A | v3 |
| v4 | v4 | N/A | v1 | N/A |

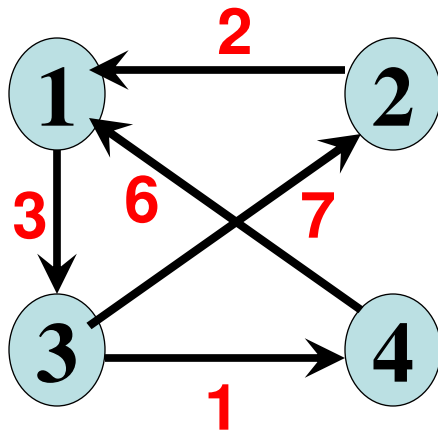
$D^{(2)}$

| | v1 | v2 | v3 | v4 |
|----|----|----------|----|----------|
| v1 | | ∞ | | |
| v2 | 2 | 0 | 5 | ∞ |
| v3 | | 7 | | |
| v4 | | ∞ | | |

$\Pi^{(2)}$

| | v1 | v2 | v3 | v4 |
|----|----|-----|----|-----|
| v1 | | | | |
| v2 | v2 | N/A | v1 | N/A |
| v3 | | | | |
| v4 | | | | |

FW Algorithm: Example 1 (2)



Iteration 2

$D^{(1)}$

| | v1 | v2 | v3 | v4 |
|----|----------|----------|----|----------|
| v1 | 0 | ∞ | 3 | ∞ |
| v2 | 2 | 0 | 5 | ∞ |
| v3 | ∞ | 7 | 0 | 1 |
| v4 | 6 | ∞ | 9 | 0 |

$\Pi^{(1)}$

| | v1 | v2 | v3 | v4 |
|----|-----|-----|-----|-----|
| v1 | N/A | N/A | v1 | N/A |
| v2 | v2 | N/A | v1 | N/A |
| v3 | N/A | v3 | N/A | v3 |
| v4 | v4 | N/A | v1 | N/A |

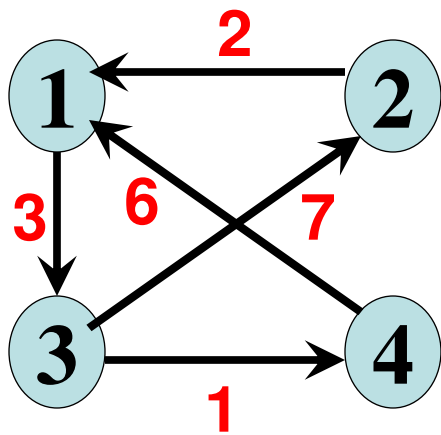
$D^{(2)}$

| | v1 | v2 | v3 | v4 |
|----|----|----------|----|----------|
| v1 | 0 | ∞ | 3 | ∞ |
| v2 | 2 | 0 | 5 | ∞ |
| v3 | 9 | 7 | 0 | 1 |
| v4 | 6 | ∞ | 9 | 0 |

$\Pi^{(2)}$

| | v1 | v2 | v3 | v4 |
|----|-----|-----|-----|-----|
| v1 | N/A | N/A | v1 | N/A |
| v2 | v2 | N/A | v1 | N/A |
| v3 | v2 | v3 | N/A | v3 |
| v4 | v4 | N/A | v1 | N/A |

FW Algorithm: Example 1 (3)



Iteration 3

$D^{(2)}$

| | v1 | v2 | v3 | v4 |
|----|----|----------|----|----------|
| v1 | 0 | ∞ | 3 | ∞ |
| v2 | 2 | 0 | 5 | ∞ |
| v3 | 9 | 7 | 0 | 1 |
| v4 | 6 | ∞ | 9 | 0 |

$\Pi^{(2)}$

| | v1 | v2 | v3 | v4 |
|----|-----|-----|-----|-----|
| v1 | N/A | N/A | v1 | N/A |
| v2 | v2 | N/A | v1 | N/A |
| v3 | v2 | v3 | N/A | v3 |
| v4 | v4 | N/A | v1 | N/A |

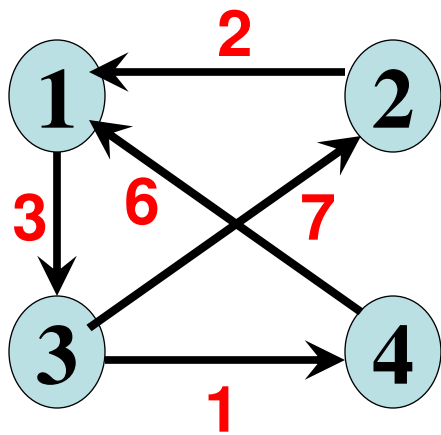
$D^{(3)}$

| | v1 | v2 | v3 | v4 |
|----|----|----|----|----|
| v1 | | | 3 | |
| v2 | | | 5 | |
| v3 | 9 | 7 | 0 | 1 |
| v4 | | | 9 | |

$\Pi^{(3)}$

| | v1 | v2 | v3 | v4 |
|----|----|----|-----|----|
| v1 | | | | |
| v2 | | | | |
| v3 | v2 | v3 | N/A | v3 |
| v4 | | | | |

FW Algorithm: Example 1 (3)



Iteration 3

$D^{(2)}$

| | v1 | v2 | v3 | v4 |
|----|----|----------|----|----------|
| v1 | 0 | ∞ | 3 | ∞ |
| v2 | 2 | 0 | 5 | ∞ |
| v3 | 9 | 7 | 0 | 1 |
| v4 | 6 | ∞ | 9 | 0 |

$\Pi^{(2)}$

| | v1 | v2 | v3 | v4 |
|----|-----|-----|-----|-----|
| v1 | N/A | N/A | v1 | N/A |
| v2 | v2 | N/A | v1 | N/A |
| v3 | v2 | v3 | N/A | v3 |
| v4 | v4 | N/A | v1 | N/A |

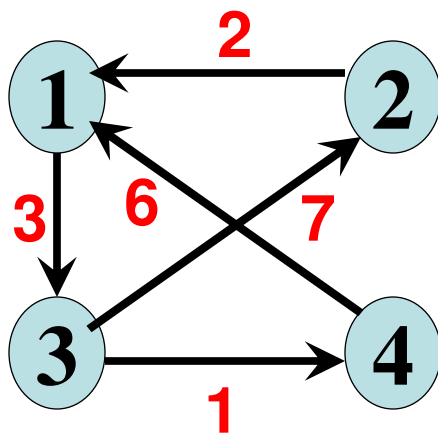
$D^{(3)}$

| | v1 | v2 | v3 | v4 |
|----|----|----|----|----|
| v1 | 0 | 10 | 3 | 4 |
| v2 | 2 | 0 | 5 | 6 |
| v3 | 9 | 7 | 0 | 1 |
| v4 | 6 | 16 | 9 | 0 |

$\Pi^{(3)}$

| | v1 | v2 | v3 | v4 |
|----|-----|-----|-----|-----|
| v1 | N/A | v3 | v1 | v3 |
| v2 | v2 | N/A | v1 | v3 |
| v3 | v2 | v3 | N/A | v3 |
| v4 | v4 | v3 | v1 | N/A |

FW Algorithm: Example 1 (4)



Iteration 4

$D^{(3)}$

| | v1 | v2 | v3 | v4 |
|----|----|----|----|----|
| v1 | 0 | 10 | 3 | 4 |
| v2 | 2 | 0 | 5 | 6 |
| v3 | 9 | 7 | 0 | 1 |
| v4 | 6 | 16 | 9 | 0 |

$\Pi^{(3)}$

| | v1 | v2 | v3 | v4 |
|----|-----|-----|-----|-----|
| v1 | N/A | v3 | v1 | v3 |
| v2 | v2 | N/A | v1 | v3 |
| v3 | v2 | v3 | N/A | v3 |
| v4 | v4 | v3 | v1 | N/A |

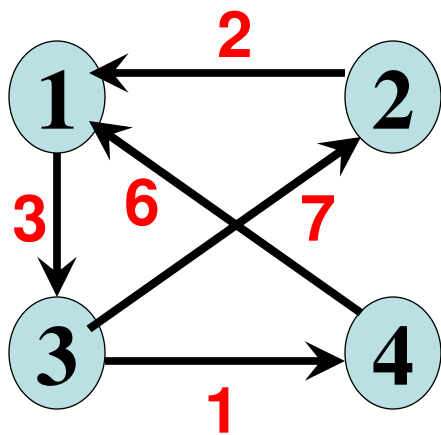
$D^{(4)}$

| | v1 | v2 | v3 | v4 |
|----|----|----|----|----|
| v1 | | | | 4 |
| v2 | | | | 6 |
| v3 | | | | 1 |
| v4 | 6 | 16 | 9 | 0 |

$\Pi^{(4)}$

| | v1 | v2 | v3 | v4 |
|----|----|----|----|-----|
| v1 | | | | |
| v2 | | | | |
| v3 | | | | |
| v4 | v4 | v3 | v1 | N/A |

FW Algorithm: Example 1 (4)



Iteration 4

$D^{(3)}$

| | v1 | v2 | v3 | v4 |
|----|----|----|----|----|
| v1 | 0 | 10 | 3 | 4 |
| v2 | 2 | 0 | 5 | 6 |
| v3 | 9 | 7 | 0 | 1 |
| v4 | 6 | 16 | 9 | 0 |

$\Pi^{(3)}$

| | v1 | v2 | v3 | v4 |
|----|-----|-----|-----|-----|
| v1 | N/A | v3 | v1 | v3 |
| v2 | v2 | N/A | v1 | v3 |
| v3 | v2 | v3 | N/A | v3 |
| v4 | v4 | v3 | v1 | N/A |

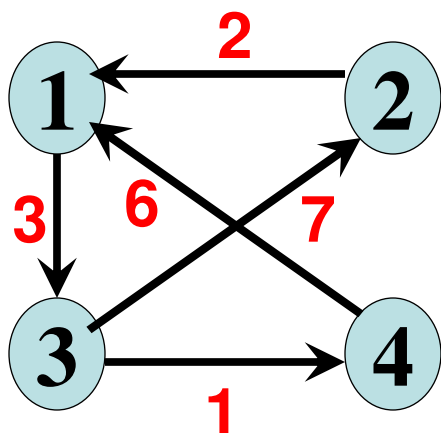
$D^{(4)}$

| | v1 | v2 | v3 | v4 |
|----|----|----|----|----|
| v1 | 0 | 10 | 3 | 4 |
| v2 | 2 | 0 | 5 | 6 |
| v3 | 7 | 7 | 0 | 1 |
| v4 | 6 | 16 | 9 | 0 |

$\Pi^{(4)}$

| | v1 | v2 | v3 | v4 |
|----|-----|-----|-----|-----|
| v1 | N/A | v3 | v1 | v3 |
| v2 | v2 | N/A | v1 | v3 |
| v3 | v4 | v3 | N/A | v3 |
| v4 | v4 | v3 | v1 | N/A |

FW Algorithm: Example 1 (5)



$D^{(4)}$

| | v1 | v2 | v3 | v4 |
|----|----|----|----|----|
| v1 | 0 | 10 | 3 | 4 |
| v2 | 2 | 0 | 5 | 6 |
| v3 | 7 | 7 | 0 | 1 |
| v4 | 6 | 16 | 9 | 0 |

$\Pi^{(4)}$

| | v1 | v2 | v3 | v4 |
|----|-----|-----|-----|-----|
| v1 | N/A | v3 | v1 | v3 |
| v2 | v2 | N/A | v1 | v3 |
| v3 | v4 | v3 | N/A | v3 |
| v4 | v4 | v3 | v1 | N/A |

Path from v2 to v4

$\pi(v2 \dots v4)$

$= \pi(v2 \dots v3) \rightarrow v3 \rightarrow v4$

$= \pi(v2 \dots v1) \rightarrow v1 \rightarrow v3 \rightarrow v4$

$= v2 \rightarrow v1 \rightarrow v3 \rightarrow v4$

Path from v4 to v2

$\pi(v4 \dots v2)$

$= \pi(v4 \dots v3) \rightarrow v3 \rightarrow v2$

$= \pi(v4 \dots v1) \rightarrow v1 \rightarrow v3 \rightarrow v2$

$= v4 \rightarrow v1 \rightarrow v3 \rightarrow v2$

FW Algorithm (pseudocode and analysis)

ALGORITHM *Floyd*($W[1..n, 1..n]$)

//Implements Floyd's algorithm for the all-pairs shortest-paths problem

//Input: The weight matrix W of a graph with no negative-length cycle

//Output: The distance matrix of the shortest paths' lengths

$D \leftarrow W$ //is not necessary if W can be overwritten

for $k \leftarrow 1$ **to** n **do**

for $i \leftarrow 1$ **to** n **do**

for $j \leftarrow 1$ **to** n **do**

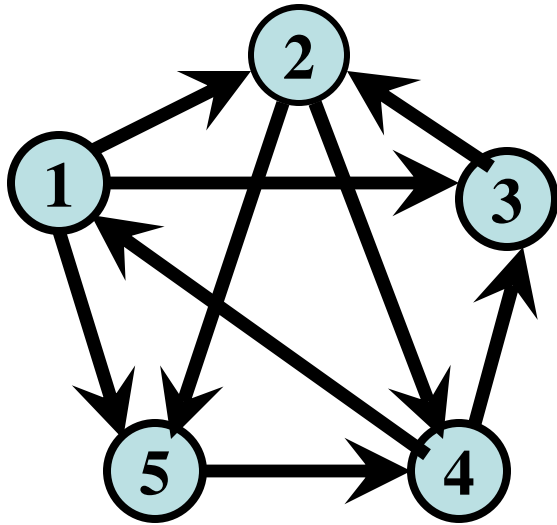
$D[i, j] \leftarrow \min\{D[i, j], D[i, k] + D[k, j]\}$

return D

Time efficiency: $\Theta(n^3)$

Space efficiency: $\Theta(n^2)$

FW Algorithm: Example 2(1)



$D^{(0)}$

| | v1 | v2 | v3 | v4 | v5 |
|----|----------|----------|----------|----------|----------|
| v1 | 0 | 3 | 8 | ∞ | -4 |
| v2 | ∞ | 0 | ∞ | 1 | 7 |
| v3 | ∞ | 4 | 0 | ∞ | ∞ |
| v4 | 2 | ∞ | -5 | 0 | ∞ |
| v5 | ∞ | ∞ | ∞ | 6 | 0 |

$\Pi^{(0)}$

| | v1 | v2 | v3 | v4 | v5 |
|----|-----|-----|-----|-----|-----|
| v1 | N/A | v1 | v1 | N/A | v1 |
| v2 | N/A | N/A | N/A | v2 | v2 |
| v3 | N/A | v3 | N/A | N/A | N/A |
| v4 | v4 | N/A | v4 | N/A | N/A |
| v5 | N/A | N/A | N/A | v5 | N/A |

| | | Weight Matrix | | | | |
|----|----------|---------------|----------|----------|----------|--|
| | v1 | v2 | v3 | v4 | v5 | |
| v1 | 0 | 3 | 8 | ∞ | -4 | |
| v2 | ∞ | 0 | ∞ | 1 | 7 | |
| v3 | ∞ | 4 | 0 | ∞ | ∞ | |
| v4 | 2 | ∞ | -5 | 0 | ∞ | |
| v5 | ∞ | ∞ | ∞ | 6 | 0 | |

$D^{(1)}$

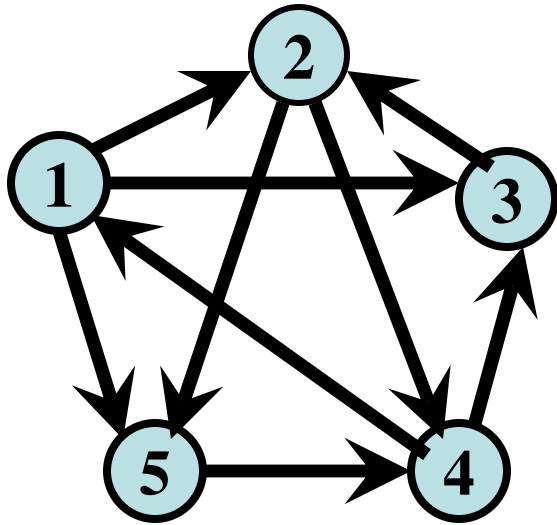
| | v1 | v2 | v3 | v4 | v5 |
|----|----------|----|----|----------|----|
| v1 | 0 | 3 | 8 | ∞ | -4 |
| v2 | ∞ | | | | |
| v3 | ∞ | | | | |
| v4 | 2 | | | | |
| v5 | ∞ | | | | |

$\Pi^{(1)}$

| | v1 | v2 | v3 | v4 | v5 |
|----|-----|----|----|-----|----|
| v1 | N/A | v1 | v1 | N/A | v1 |
| v2 | | | | | |
| v3 | | | | | |
| v4 | | | | | |
| v5 | | | | | |

Iteration 1

FW Algorithm: Example 2(1)



| | Weight Matrix | | | | |
|----|---------------|----------|----------|----------|----------|
| | v1 | v2 | v3 | v4 | v5 |
| v1 | 0 | 3 | 8 | ∞ | -4 |
| v2 | ∞ | 0 | ∞ | 1 | 7 |
| v3 | ∞ | 4 | 0 | ∞ | ∞ |
| v4 | 2 | ∞ | -5 | 0 | ∞ |
| v5 | ∞ | ∞ | ∞ | 6 | 0 |

$$D^{(0)}$$

| | v1 | v2 | v3 | v4 | v5 |
|----|----------|----------|----------|----------|----------|
| v1 | 0 | 3 | 8 | ∞ | -4 |
| v2 | ∞ | 0 | ∞ | 1 | 7 |
| v3 | ∞ | 4 | 0 | ∞ | ∞ |
| v4 | 2 | ∞ | -5 | 0 | ∞ |
| v5 | ∞ | ∞ | ∞ | 6 | 0 |

$$\Pi^{(0)}$$

| | v1 | v2 | v3 | v4 | v5 |
|----|-----|-----|-----|-----|-----|
| v1 | N/A | v1 | v1 | N/A | v1 |
| v2 | N/A | N/A | N/A | v2 | v2 |
| v3 | N/A | v3 | N/A | N/A | N/A |
| v4 | v4 | N/A | v4 | N/A | N/A |
| v5 | N/A | N/A | N/A | v5 | N/A |

$$D^{(1)}$$

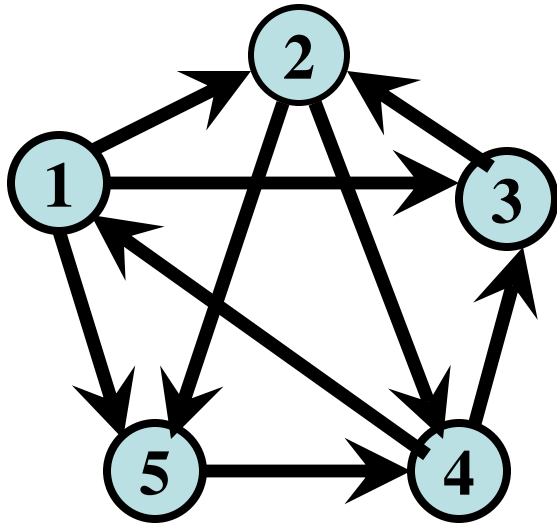
| | v1 | v2 | v3 | v4 | v5 |
|----|----------|----------|----------|----------|----------|
| v1 | 0 | 3 | 8 | ∞ | -4 |
| v2 | ∞ | 0 | ∞ | 1 | 7 |
| v3 | ∞ | 4 | 0 | ∞ | ∞ |
| v4 | 2 | 5 | -5 | 0 | -2 |
| v5 | ∞ | ∞ | ∞ | 6 | 0 |

$$\Pi^{(1)}$$

| | v1 | v2 | v3 | v4 | v5 |
|----|-----|-----|-----|-----|-----|
| v1 | N/A | v1 | v1 | N/A | v1 |
| v2 | N/A | N/A | N/A | v2 | v2 |
| v3 | N/A | v3 | N/A | N/A | N/A |
| v4 | v4 | v1 | v4 | N/A | v1 |
| v5 | N/A | N/A | N/A | v5 | N/A |

Iteration 1

FW Algorithm: Example 2(2)



| | Weight Matrix | | | | |
|----|---------------|----------|----------|----------|----------|
| | v1 | v2 | v3 | v4 | v5 |
| v1 | 0 | 3 | 8 | ∞ | -4 |
| v2 | ∞ | 0 | ∞ | 1 | 7 |
| v3 | ∞ | 4 | 0 | ∞ | ∞ |
| v4 | 2 | ∞ | -5 | 0 | ∞ |
| v5 | ∞ | ∞ | ∞ | 6 | 0 |

$D^{(1)}$

| | v1 | v2 | v3 | v4 | v5 |
|----|----------|----------|----------|----------|----------|
| v1 | 0 | 3 | 8 | ∞ | -4 |
| v2 | ∞ | 0 | ∞ | 1 | 7 |
| v3 | ∞ | 4 | 0 | ∞ | ∞ |
| v4 | 2 | 5 | -5 | 0 | -2 |
| v5 | ∞ | ∞ | ∞ | 6 | 0 |

$\Pi^{(1)}$

| | v1 | v2 | v3 | v4 | v5 |
|----|-----|-----|-----|-----|-----|
| v1 | N/A | v1 | v1 | N/A | v1 |
| v2 | N/A | N/A | N/A | v2 | v2 |
| v3 | N/A | v3 | N/A | N/A | N/A |
| v4 | v4 | v1 | v4 | N/A | v1 |
| v5 | N/A | N/A | N/A | v5 | N/A |

$D^{(2)}$

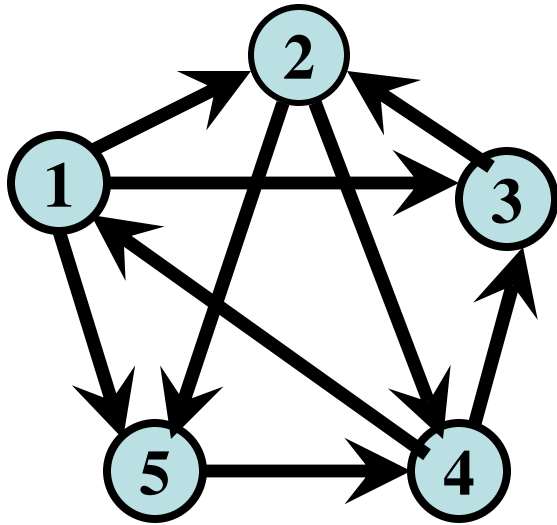
| | v1 | v2 | v3 | v4 | v5 |
|----|----------|----------|----------|----|----|
| v1 | | 3 | | | |
| v2 | ∞ | 0 | ∞ | 1 | 7 |
| v3 | | 4 | | | |
| v4 | | 5 | | | |
| v5 | | ∞ | | | |

$\Pi^{(2)}$

| | v1 | v2 | v3 | v4 | v5 |
|----|-----|-----|-----|----|----|
| v1 | | | | | |
| v2 | N/A | N/A | N/A | v2 | v2 |
| v3 | | | | | |
| v4 | | | | | |
| v5 | | | | | |

Iteration 2

FW Algorithm: Example 2(2)



| | Weight Matrix | | | | |
|----|---------------|----|----|----|----|
| | v1 | v2 | v3 | v4 | v5 |
| v1 | 0 | 3 | 8 | ∞ | -4 |
| v2 | ∞ | 0 | ∞ | 1 | 7 |
| v3 | ∞ | 4 | 0 | ∞ | ∞ |
| v4 | 2 | ∞ | -5 | 0 | ∞ |
| v5 | ∞ | ∞ | ∞ | 6 | 0 |

$D^{(1)}$

| | v1 | v2 | v3 | v4 | v5 |
|----|----|----|----|----|----|
| v1 | 0 | 3 | 8 | ∞ | -4 |
| v2 | ∞ | 0 | ∞ | 1 | 7 |
| v3 | ∞ | 4 | 0 | ∞ | ∞ |
| v4 | 2 | 5 | -5 | 0 | -2 |
| v5 | ∞ | ∞ | ∞ | 6 | 0 |

$\Pi^{(1)}$

| | v1 | v2 | v3 | v4 | v5 |
|----|-----|-----|-----|-----|-----|
| v1 | N/A | v1 | v1 | N/A | v1 |
| v2 | N/A | N/A | N/A | v2 | v2 |
| v3 | N/A | v3 | N/A | N/A | N/A |
| v4 | v4 | v1 | v4 | N/A | v1 |
| v5 | N/A | N/A | N/A | v5 | N/A |

$D^{(2)}$

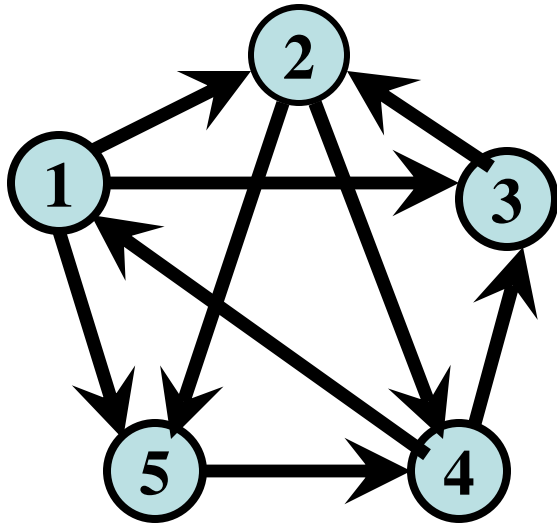
| | v1 | v2 | v3 | v4 | v5 |
|----|----|----|----|----|----|
| v1 | 0 | 3 | 8 | 4 | -4 |
| v2 | ∞ | 0 | ∞ | 1 | 7 |
| v3 | ∞ | 4 | 0 | 5 | 11 |
| v4 | 2 | 5 | -5 | 0 | -2 |
| v5 | ∞ | ∞ | ∞ | 6 | 0 |

$\Pi^{(2)}$

| | v1 | v2 | v3 | v4 | v5 |
|----|-----|-----|-----|-----|-----|
| v1 | N/A | v1 | v1 | v2 | v1 |
| v2 | N/A | N/A | N/A | v2 | v2 |
| v3 | N/A | v3 | N/A | v2 | v2 |
| v4 | v4 | v1 | v4 | N/A | v1 |
| v5 | N/A | N/A | N/A | v5 | N/A |

Iteration 2

FW Algorithm: Example 2(3)



$D^{(2)}$

| | v1 | v2 | v3 | v4 | v5 |
|----|----------|----------|----------|----|----|
| v1 | 0 | 3 | 8 | 4 | -4 |
| v2 | ∞ | 0 | ∞ | 1 | 7 |
| v3 | ∞ | 4 | 0 | 5 | 11 |
| v4 | 2 | 5 | -5 | 0 | -2 |
| v5 | ∞ | ∞ | ∞ | 6 | 0 |

$\Pi^{(2)}$

| | v1 | v2 | v3 | v4 | v5 |
|----|-----|-----|-----|-----|-----|
| v1 | N/A | v1 | v1 | v2 | v1 |
| v2 | N/A | N/A | N/A | v2 | v2 |
| v3 | N/A | v3 | N/A | v2 | v2 |
| v4 | v4 | v1 | v4 | N/A | v1 |
| v5 | N/A | N/A | N/A | v5 | N/A |

$D^{(3)}$

| | v1 | v2 | v3 | v4 | v5 |
|----|----------|----------|----------|----|----|
| v1 | 0 | 3 | 8 | 4 | -4 |
| v2 | ∞ | 0 | ∞ | 1 | 7 |
| v3 | ∞ | 4 | 0 | 5 | 11 |
| v4 | 2 | -1 | -5 | 0 | -2 |
| v5 | ∞ | ∞ | ∞ | 6 | 0 |

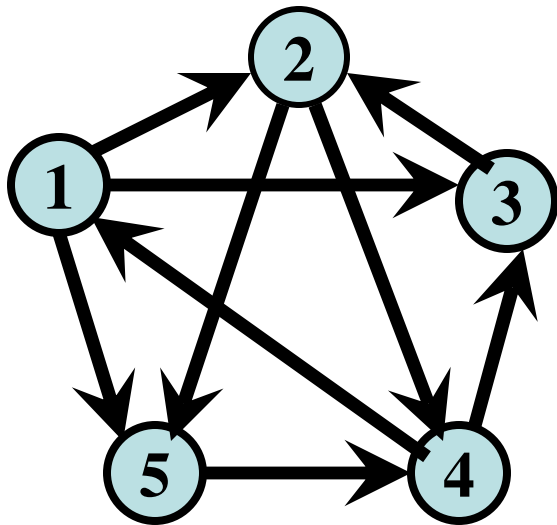
$\Pi^{(3)}$

| | v1 | v2 | v3 | v4 | v5 |
|----|-----|-----|-----|-----|-----|
| v1 | N/A | v1 | v1 | v2 | v1 |
| v2 | N/A | N/A | N/A | v2 | v2 |
| v3 | N/A | v3 | N/A | v2 | v2 |
| v4 | v4 | v3 | v4 | N/A | v1 |
| v5 | N/A | N/A | N/A | v5 | N/A |

| | | Weight Matrix | | | | |
|----|----------|---------------|----------|----------|----------|--|
| | v1 | v2 | v3 | v4 | v5 | |
| v1 | 0 | 3 | 8 | ∞ | -4 | |
| v2 | ∞ | 0 | ∞ | 1 | 7 | |
| v3 | ∞ | 4 | 0 | ∞ | ∞ | |
| v4 | 2 | ∞ | -5 | 0 | ∞ | |
| v5 | ∞ | ∞ | ∞ | 6 | 0 | |

Iteration 3

FW Algorithm: Example 2(4)



| | Weight Matrix | | | | |
|----|---------------|----------|----------|----------|----------|
| | v1 | v2 | v3 | v4 | v5 |
| v1 | 0 | 3 | 8 | ∞ | -4 |
| v2 | ∞ | 0 | ∞ | 1 | 7 |
| v3 | ∞ | 4 | 0 | ∞ | ∞ |
| v4 | 2 | ∞ | -5 | 0 | ∞ |
| v5 | ∞ | ∞ | ∞ | 6 | 0 |

$D^{(3)}$

| | v1 | v2 | v3 | v4 | v5 |
|----|----------|----------|----------|----|----|
| v1 | 0 | 3 | 8 | 4 | -4 |
| v2 | ∞ | 0 | ∞ | 1 | 7 |
| v3 | ∞ | 4 | 0 | 5 | 11 |
| v4 | 2 | -1 | -5 | 0 | -2 |
| v5 | ∞ | ∞ | ∞ | 6 | 0 |

$\Pi^{(3)}$

| | v1 | v2 | v3 | v4 | v5 |
|----|-----|-----|-----|-----|-----|
| v1 | N/A | v1 | v1 | v2 | v1 |
| v2 | N/A | N/A | N/A | v2 | v2 |
| v3 | N/A | v3 | N/A | v2 | v2 |
| v4 | v4 | v3 | v4 | N/A | v1 |
| v5 | N/A | N/A | N/A | v5 | N/A |

$D^{(4)}$

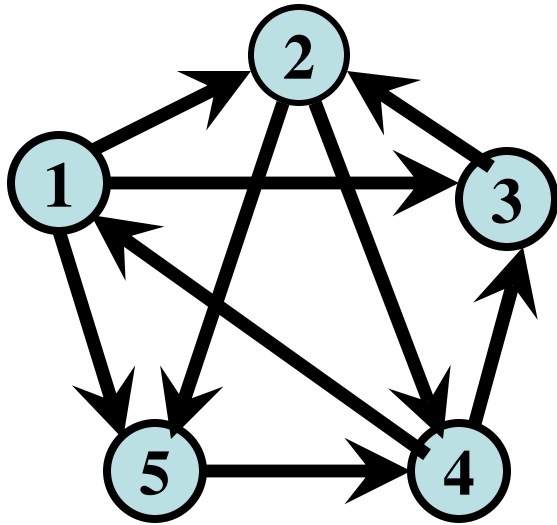
| | v1 | v2 | v3 | v4 | v5 |
|----|----|----|----|----|----|
| v1 | 0 | 3 | -1 | 4 | -4 |
| v2 | 3 | 0 | -4 | 1 | -1 |
| v3 | 7 | 4 | 0 | 5 | 3 |
| v4 | 2 | -1 | -5 | 0 | -2 |
| v5 | 8 | 5 | 1 | 6 | 0 |

$\Pi^{(4)}$

| | v1 | v2 | v3 | v4 | v5 |
|----|-----|-----|-----|-----|-----|
| v1 | N/A | v1 | v4 | v2 | v1 |
| v2 | v4 | N/A | v4 | v2 | v1 |
| v3 | v4 | v3 | N/A | v2 | v1 |
| v4 | v4 | v3 | v4 | N/A | v1 |
| v5 | v4 | v3 | v4 | v5 | N/A |

Iteration 4

FW Algorithm: Example 2(5)



| | Weight Matrix | | | | |
|----|---------------|----------|----------|----------|----------|
| | v1 | v2 | v3 | v4 | v5 |
| v1 | 0 | 3 | 8 | ∞ | -4 |
| v2 | ∞ | 0 | ∞ | 1 | 7 |
| v3 | ∞ | 4 | 0 | ∞ | ∞ |
| v4 | 2 | ∞ | -5 | 0 | ∞ |
| v5 | ∞ | ∞ | ∞ | 6 | 0 |

$D^{(4)}$

| | v1 | v2 | v3 | v4 | v5 |
|----|----|----|----|----|----|
| v1 | 0 | 3 | -1 | 4 | -4 |
| v2 | 3 | 0 | -4 | 1 | -1 |
| v3 | 7 | 4 | 0 | 5 | 3 |
| v4 | 2 | -1 | -5 | 0 | -2 |
| v5 | 8 | 5 | 1 | 6 | 0 |

$\Pi^{(4)}$

| | v1 | v2 | v3 | v4 | v5 |
|----|-----|-----|-----|-----|-----|
| v1 | N/A | v1 | v4 | v2 | v1 |
| v2 | v4 | N/A | v4 | v2 | v1 |
| v3 | v4 | v3 | N/A | v2 | v1 |
| v4 | v4 | v3 | v4 | N/A | v1 |
| v5 | v4 | v3 | v4 | v5 | N/A |

$D^{(5)}$

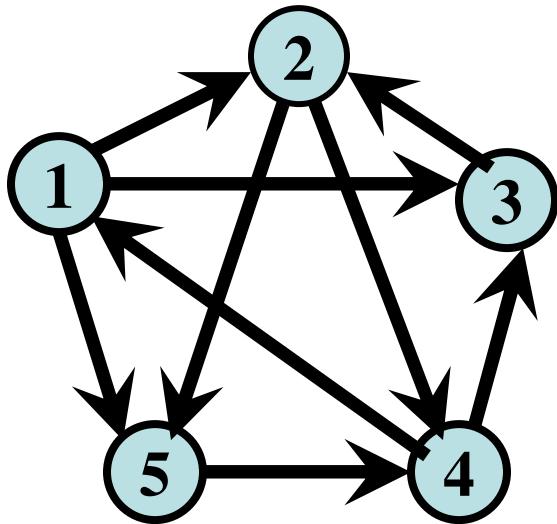
| | v1 | v2 | v3 | v4 | v5 |
|----|----|----|----|----|----|
| v1 | 0 | 1 | -3 | 2 | -4 |
| v2 | 3 | 0 | -4 | 1 | -1 |
| v3 | 7 | 4 | 0 | 5 | 3 |
| v4 | 2 | -1 | -5 | 0 | -2 |
| v5 | 8 | 5 | 1 | 6 | 0 |

$\Pi^{(5)}$

| | v1 | v2 | v3 | v4 | v5 |
|----|-----|-----|-----|-----|-----|
| v1 | N/A | v3 | v4 | v5 | v1 |
| v2 | v4 | N/A | v4 | v2 | v1 |
| v3 | v4 | v3 | N/A | v2 | v1 |
| v4 | v4 | v3 | v4 | N/A | v1 |
| v5 | v4 | v3 | v4 | v5 | N/A |

Iteration 5

FW Algorithm: Example 2(6)



$D^{(5)}$

| | v1 | v2 | v3 | v4 | v5 |
|----|----|----|----|----|----|
| v1 | 0 | 1 | -3 | 2 | -4 |
| v2 | 3 | 0 | -4 | 1 | -1 |
| v3 | 7 | 4 | 0 | 5 | 3 |
| v4 | 2 | -1 | -5 | 0 | -2 |
| v5 | 8 | 5 | 1 | 6 | 0 |

$\Pi^{(5)}$

| | v1 | v2 | v3 | v4 | v5 |
|----|-----|-----|-----|-----|-----|
| v1 | N/A | v3 | v4 | v5 | v1 |
| v2 | v4 | N/A | v4 | v2 | v1 |
| v3 | v4 | v3 | N/A | v2 | v1 |
| v4 | v4 | v3 | v4 | N/A | v1 |
| v5 | v4 | v3 | v4 | v5 | N/A |

Path from v3 to v1

$\pi(v3 \dots v1)$

$= \pi(v3 \dots v4) \rightarrow v4 \rightarrow v1$

$= \pi(v3 \dots v2) \rightarrow v2 \rightarrow v4 \rightarrow v1$

$= v3 \rightarrow v2 \rightarrow v4 \rightarrow v1$

Path from v1 to v3

$\pi(v1 \dots v3)$

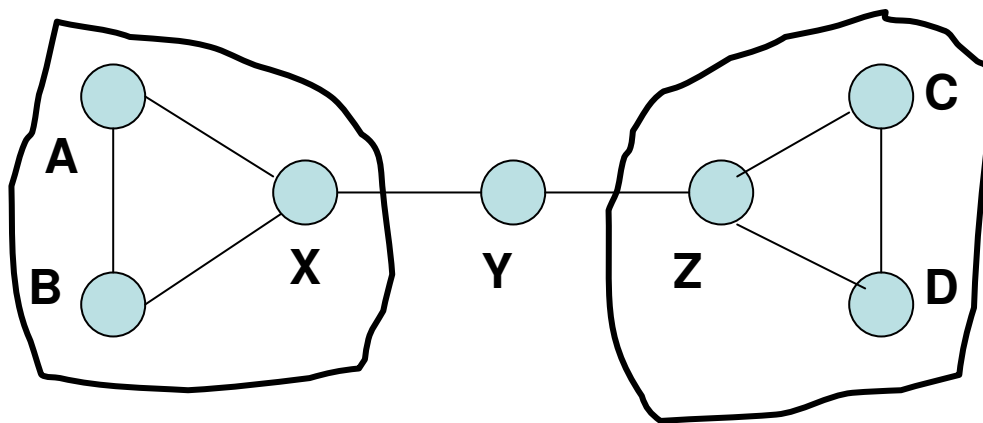
$= \pi(v1 \dots v4) \rightarrow v4 \rightarrow v3$

$= \pi(v1 \dots v5) \rightarrow v5 \rightarrow v4 \rightarrow v3$

$= v1 \rightarrow v5 \rightarrow v4 \rightarrow v3$

Centrality Metrics

- Centrality metrics quantify the importance of a vertex based on its position (topological importance) in the graph
- Among the various centrality metrics that exist, we will look at one of the most important metrics called the Betweenness Centrality (BWC) of a vertex.
- The BWC of a vertex is a measure of the presence of the vertex (as an intermediate vertex) on the shortest paths between any two pairs of vertices



In the graph shown here, there exists only one shortest path between any two vertices.

Vertex Y is an intermediate vertex on all the '9' shortest paths between any vertex in the set $\{A, B, X\}$ and any vertex in the set $\{Z, C, D\}$.

Vertices A, B, C and D do not lie on the shortest path for any two vertices.

Betweenness Centrality

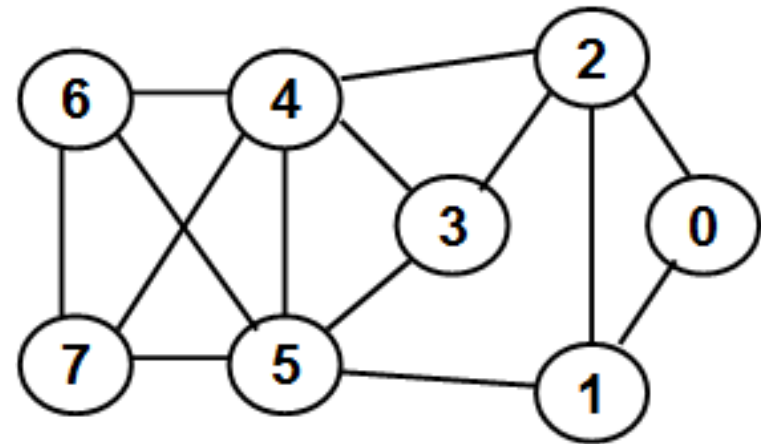
$$BWC(i) = \sum_{j \neq k \neq i} \frac{sp_{jk}(i)}{sp_{jk}}$$

($j < k$ for undirected graphs)

Time Complexity: $\Theta(VE + V^2)$

- We will now discuss how to find the total number of shortest paths between any two vertices j and k as well as to find out how many of these shortest paths go through a vertex i ($j \neq k \neq i$).
- Use Breadth First Search (BFS) to find the shortest path tree from vertex j to every other vertex k
 - Root vertex j is at level 0
 - Vertices that are 1-hop away from j are at level 1; 2-hops away from j are at level 2, and so on.
 - The number of shortest paths from j to a vertex k at level p is the sum of the number of shortest paths from j to the neighbors of k in the original graph that are at level $p-1$
 - The number of shortest paths from j to k that go through vertex i is the maximum of the number of shortest paths from j to i and the number of shortest paths from k to i .

Betweenness Centrality Example



BWC for node 0: 0.0

BWC for node 1

Pair (0, 5): $\rightarrow 1 / 1$

Pair (0, 6): $\rightarrow 1 / 2$

Pair (0, 7): $\rightarrow 1 / 2$

Pair (2, 5): $\rightarrow 1 / 3$

BWC (1) = 2.333

BWC for node 2

Pair (0, 3): $\rightarrow 1 / 1$

Pair (0, 4): $\rightarrow 1 / 1$

Pair (0, 6): $\rightarrow 1 / 2$

Pair (0, 7): $\rightarrow 1 / 2$

Pair (1, 3): $\rightarrow 1 / 2$

Pair (1, 4): $\rightarrow 1 / 2$

BWC (2): 4.0

BWC for node 3

Pair (2, 5) $\rightarrow 1 / 3$

BWC (3) = 0.333

BWC for node 4

Pair (0, 6) $\rightarrow 1 / 2$

Pair (0, 7) $\rightarrow 1 / 2$

Pair (2, 5) $\rightarrow 1 / 3$

Pair (2, 6) $\rightarrow 1 / 1$

Pair (2, 7) $\rightarrow 1 / 1$

Pair (3, 6) $\rightarrow 1 / 2$

Pair (3, 7) $\rightarrow 1 / 2$

BWC (4) = 4.333

BWC for node 6: 0.0

BWC for node 7: 0.0

BWC for node 5

Pair (0, 6) $\rightarrow 1 / 2$

Pair (0, 7) $\rightarrow 1 / 2$

Pair (1, 3) $\rightarrow 1 / 2$

Pair (1, 4) $\rightarrow 1 / 2$

Pair (1, 6) $\rightarrow 1 / 1$

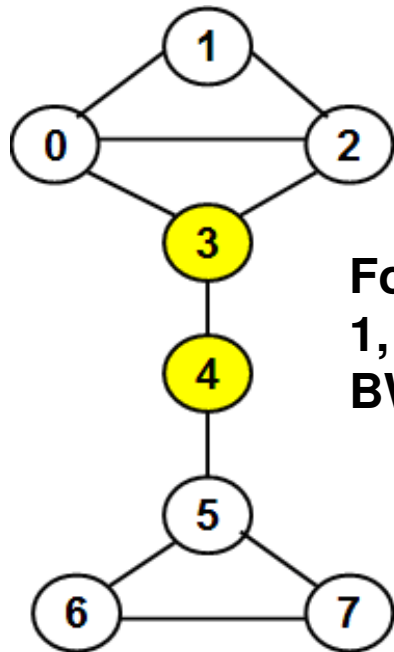
Pair (1, 7) $\rightarrow 1 / 1$

Pair (3, 6) $\rightarrow 1 / 2$

Pair (3, 7) $\rightarrow 1 / 2$

BWC (5) = 5.0

| ID | BWC |
|----|-------|
| 0 | 0.0 |
| 1 | 2.333 |
| 2 | 4.0 |
| 3 | 0.333 |
| 4 | 4.333 |
| 5 | 5.0 |
| 6 | 0.0 |
| 7 | 0.0 |



For vertices
1, 6 and 7
BWC = 0

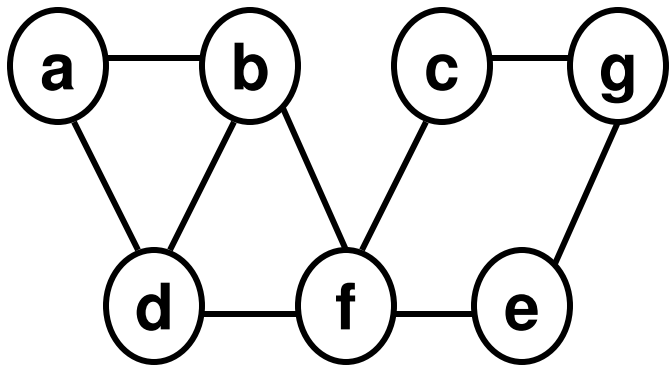
Betweenness for Vertex 0
 Pair (3,1) ---> 1 / 2
 Pair (4,1) ---> 1 / 2
 Pair (5,1) ---> 1 / 2
 Pair (6,1) ---> 1 / 2
 Pair (7,1) ---> 1 / 2
 Total of all Betweenness
 (Vertex 0): 2.5

Betweenness for Vertex 2
 Pair (3,1) ---> 1 / 2
 Pair (4,1) ---> 1 / 2
 Pair (5,1) ---> 1 / 2
 Pair (6,1) ---> 1 / 2
 Pair (7,1) ---> 1 / 2
 Total of all Betweenness
 (Vertex 2): 2.5

Betweenness for Vertex 5
 Pair (6,0) ---> 1 / 1
 Pair (7,0) ---> 1 / 1
 Pair (6,1) ---> 2 / 2
 Pair (7,1) ---> 2 / 2
 Pair (6,2) ---> 1 / 1
 Pair (7,2) ---> 1 / 1
 Pair (6,3) ---> 1 / 1
 Pair (7,3) ---> 1 / 1
 Pair (6,4) ---> 1 / 1
 Pair (7,4) ---> 1 / 1
 Total of all Betweenness
 (Vertex 5): 10

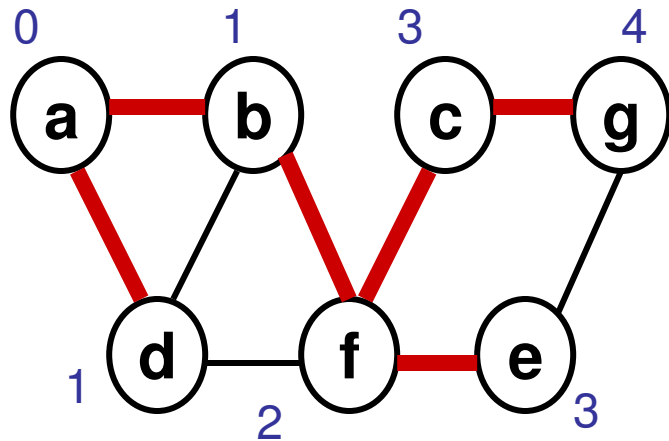
Betweenness for Vertex 3
 Pair (4,0) ---> 1 / 1
 Pair (5,0) ---> 1 / 1
 Pair (6,0) ---> 1 / 1
 Pair (7,0) ---> 1 / 1
 Pair (4,1) ---> 2 / 2
 Pair (5,1) ---> 2 / 2
 Pair (6,1) ---> 2 / 2
 Pair (7,1) ---> 2 / 2
 Pair (4,2) ---> 1 / 1
 Pair (5,2) ---> 1 / 1
 Pair (6,2) ---> 1 / 1
 Pair (7,2) ---> 1 / 1
 Total of all Betweenness
 (Vertex 3): 12

Betweenness for Vertex 4
 Pair (5,0) ---> 1 / 1
 Pair (6,0) ---> 1 / 1
 Pair (7,0) ---> 1 / 1
 Pair (5,1) ---> 2 / 2
 Pair (6,1) ---> 2 / 2
 Pair (7,1) ---> 2 / 2
 Pair (5,2) ---> 1 / 1
 Pair (6,2) ---> 1 / 1
 Pair (7,2) ---> 1 / 1
 Pair (5,3) ---> 1 / 1
 Pair (6,3) ---> 1 / 1
 Pair (7,3) ---> 1 / 1
 Total of all Betweenness
 (Vertex 4): 12

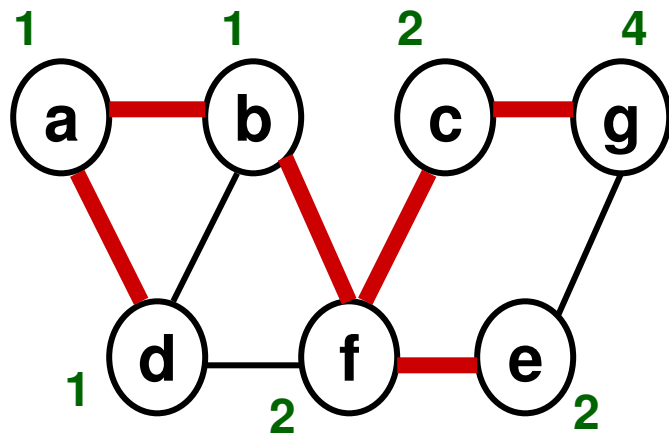
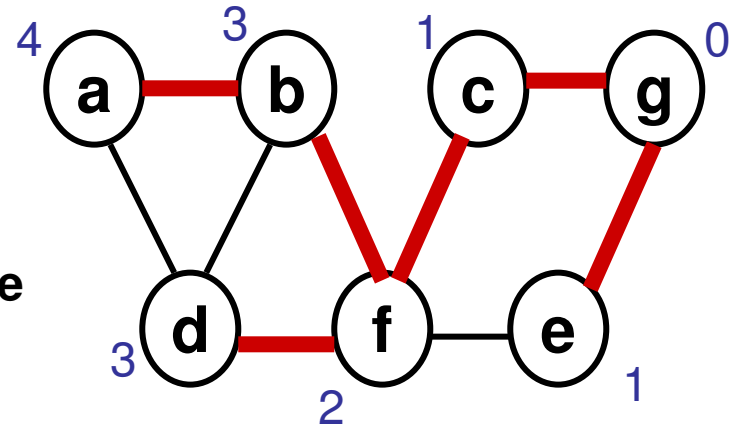


shortest paths from a to g that go through c
 is the product(# shortest paths from a to c,
 # shortest paths from g to c)
 = product (2, 1) = 2

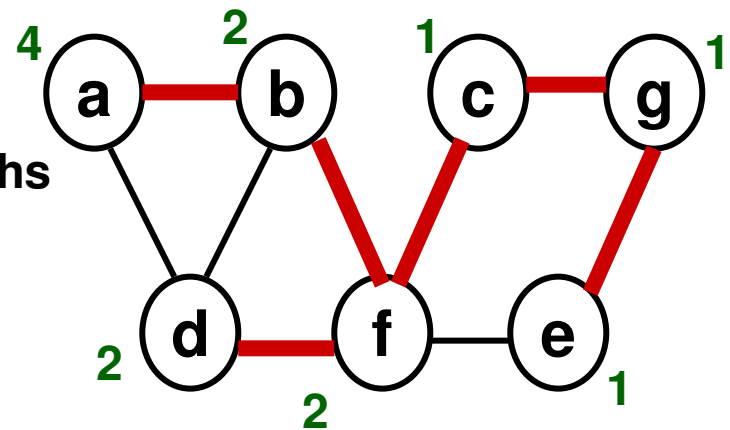
BWC ('c' with respect to pair a-g) = 2/4



Levels of
 Vertices on
 the BFS tree

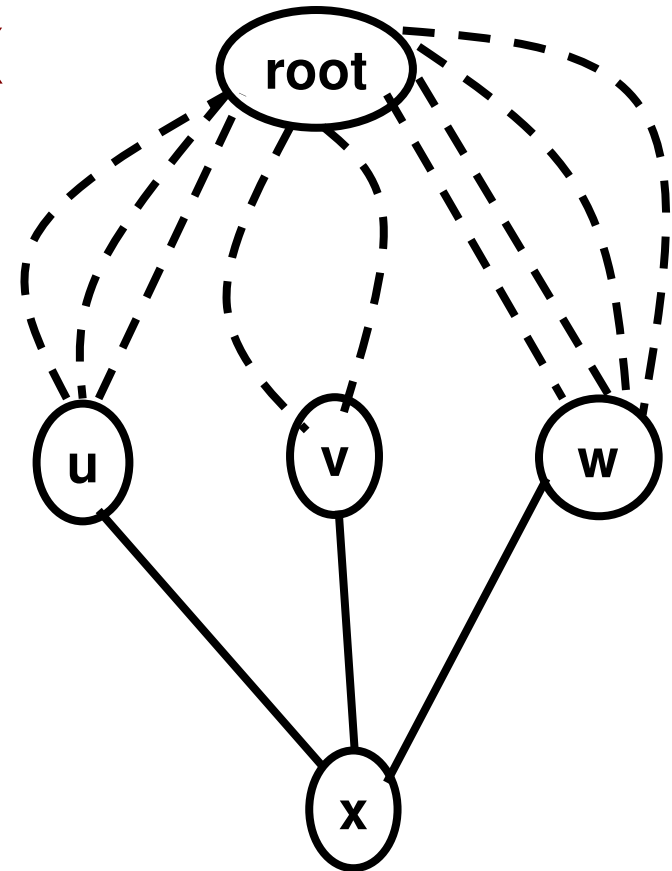


shortest paths
 from the root
 to the other
 vertices

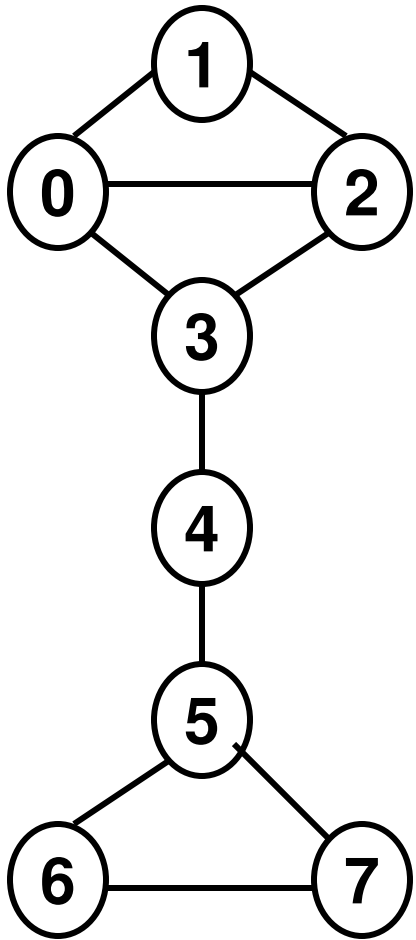


Shortest Paths from the Root to a Vertex

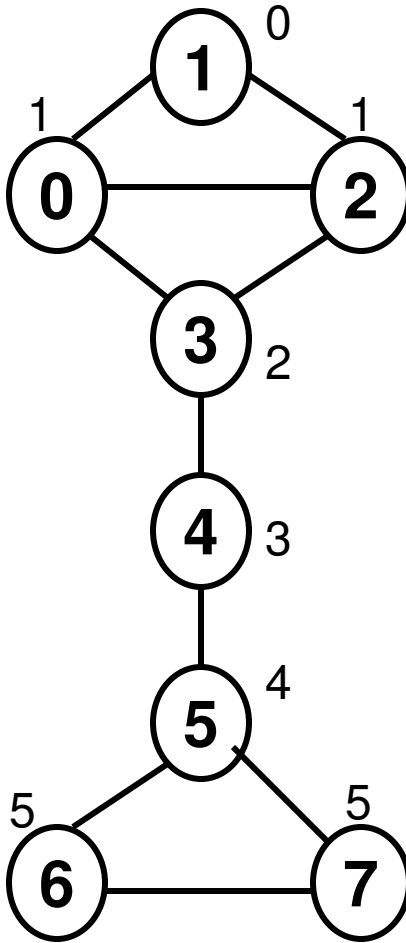
- The number of shortest paths from the root of a Breadth First Search (BFS) tree to a vertex 'x' (at level 'L' in the BFS tree) is the sum of the number of shortest paths from the root to the predecessors of 'x' (that are at level 'L-1' in the BFS tree) to which 'x' has an edge in the graph.
- In the example shown on the right side, vertex 'x' (at level L) has an edge to vertices 'u', 'v' and 'w' (at level L-1 in the BFS tree) in the given graph. The # shortest paths from the root to 'x' is the sum of the # shortest paths from the root to each of 'u', 'v' and 'w'.



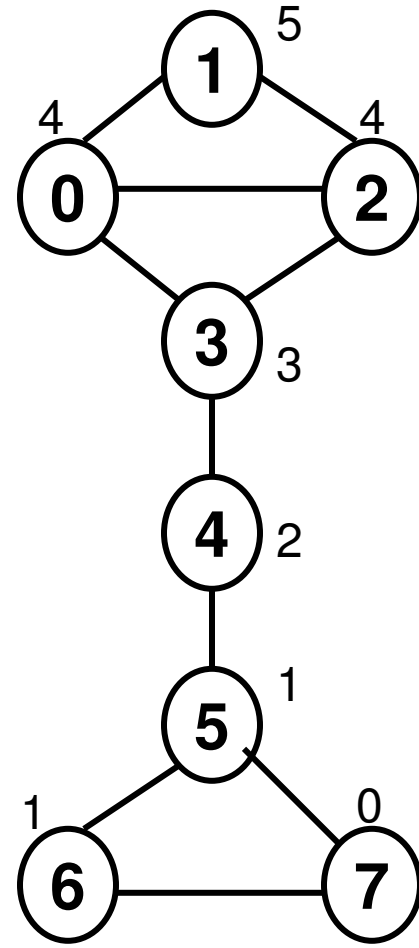
$$\begin{aligned} & \# \text{ Shortest Paths from the root to 'x'} \\ &= \# \text{ Shortest Paths from the root to 'u'} + \\ & \quad \# \text{ Shortest Paths from the root to 'v'} + \\ & \quad \# \text{ Shortest Paths from the root to 'w'} \\ &= 3 + 2 + 4 = 9. \end{aligned}$$



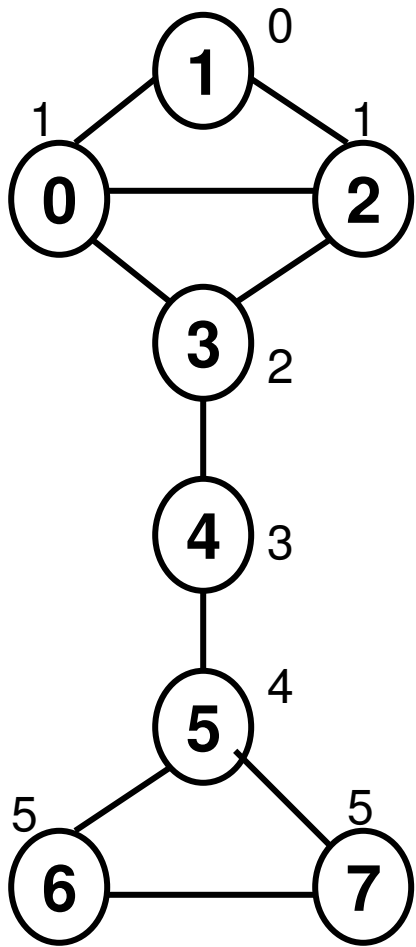
To determine how many Shortest paths from nodes 1 to 7 that go through node 4.



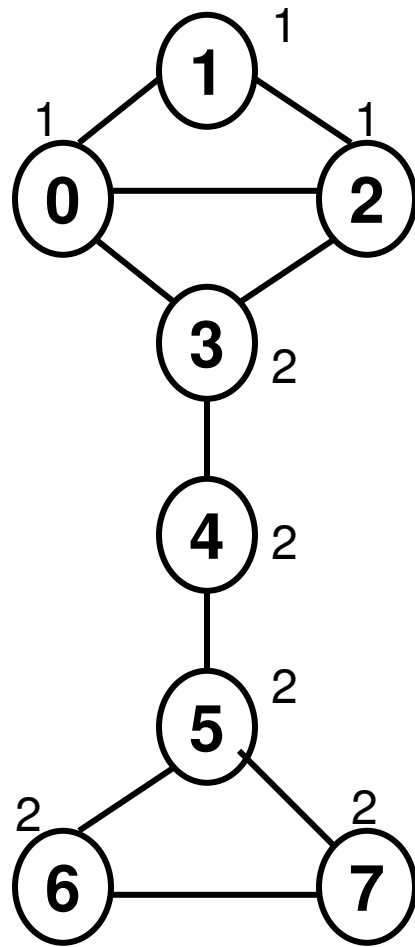
Level Numbers of the vertices starting from root '1'.



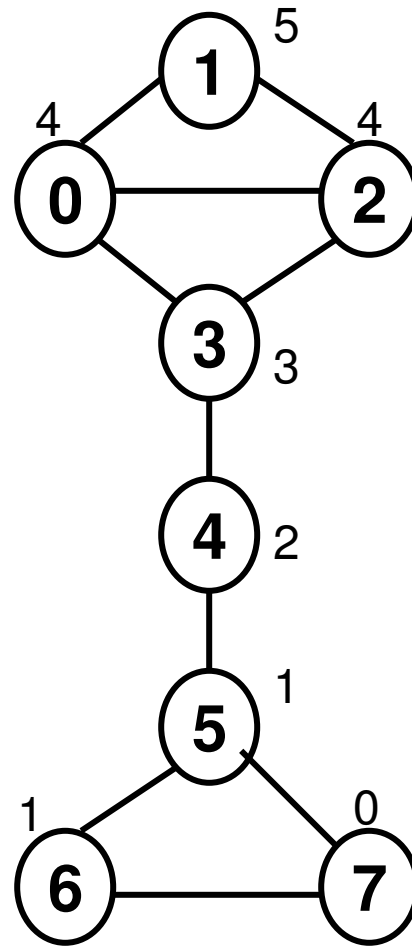
Level Numbers of the vertices starting from root '7'.



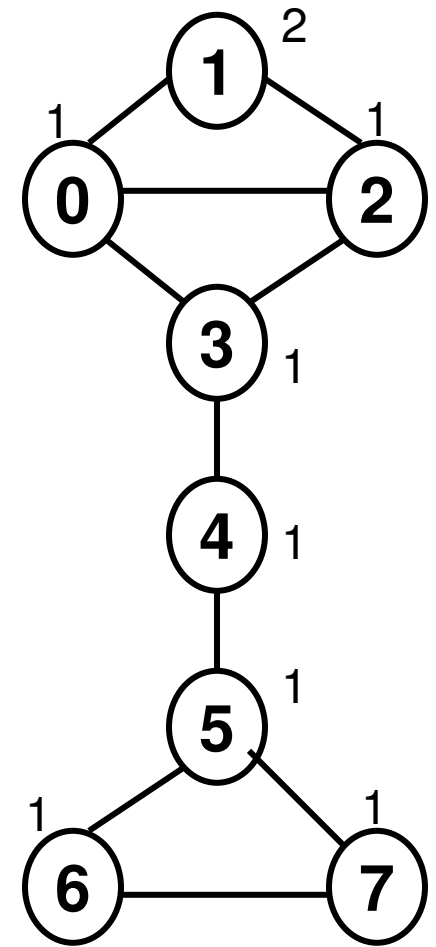
**BFS Tree
rooted at
Vertex 1**



**# shortest paths
from vertex 1 to
the other vertices**



**BFS Tree
rooted at
Vertex 7**



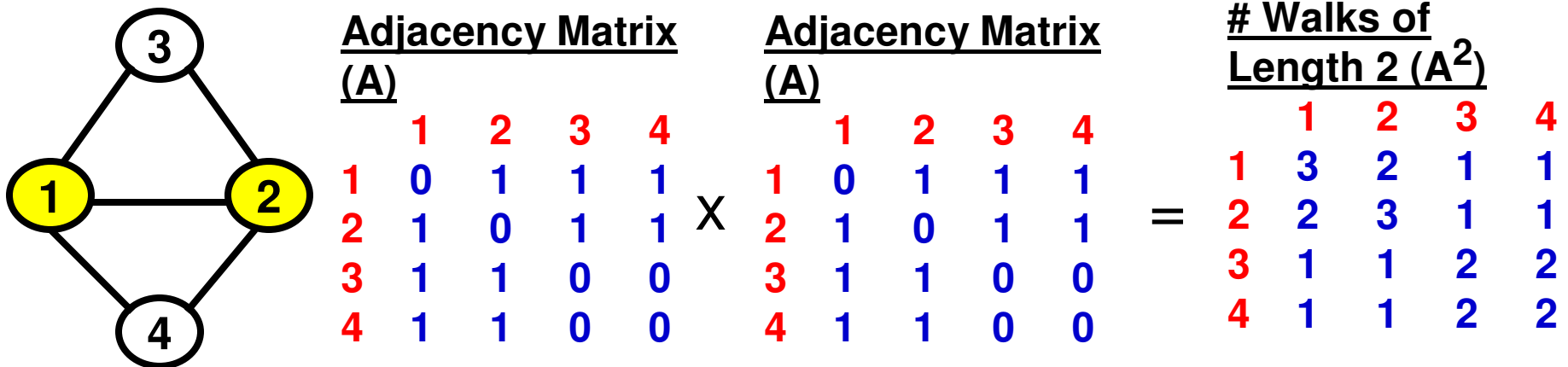
**# shortest paths
from vertex 7 to
the other vertices**

To determine how many Shortest paths from nodes 1 to 7 that go through node 4: = Product(2, 1) = 2

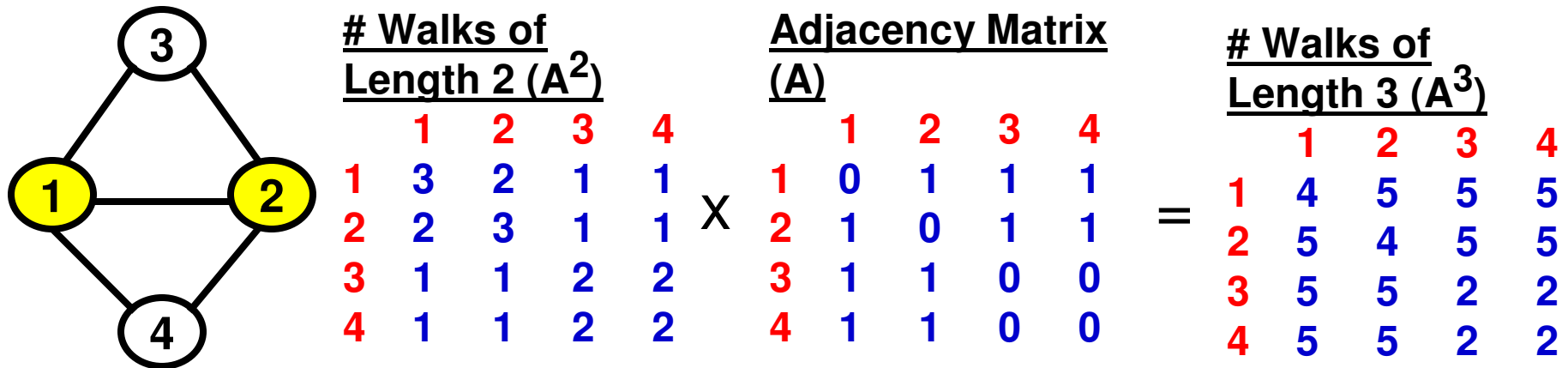
BWC(node 4 with respect to pair 1-7) = 2/2

Number of Walks in a Graph

- An u - v walk between two vertices u and v is a sequence of zero or more intermediate vertices (that could be even repeated).
- The length of a walk is one plus the number of intermediate vertices
 - Example: **2 – 3 – 1 – 4 – 1** is a walk of length 4.
- A walk is a path if the intermediate vertices, if any, are not repeated.
 - Example: **2 – 3 – 1** is a walk as well as a path, but the walk **2 – 3 – 1 – 4 – 1** is not a path.
- The number of walks of length k between any two vertices in a graph could be determined by finding A^k where A is the binary adjacency matrix of the graph.



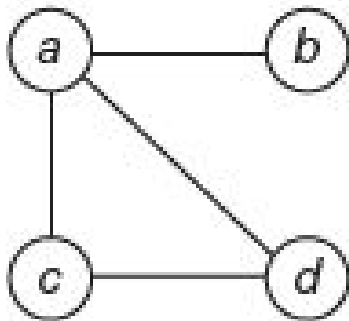
Number of Walks in a Graph



- **Basic Rules for Matrix Multiplication**
- To multiply two matrices A and B and get a product matrix $P = A * B$:
- (1) The number of columns in the first matrix A should be equal to the number of rows in the second matrix B
- (2) To get the value of a cell (i, j) in the product matrix P, do a pair-wise multiplication of the elements in row i of the first matrix with the elements in column j of the second matrix.

To find # Walks of Length 'n'

Walks of Length 4: Find A^4 .



$$A^2 = \begin{matrix} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \begin{bmatrix} 3 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 2 & 1 \\ 1 & 1 & 1 & 2 \end{bmatrix} \end{matrix} \quad \mathbf{x} \quad A^2 = \begin{matrix} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \begin{bmatrix} 3 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 2 & 1 \\ 1 & 1 & 1 & 2 \end{bmatrix} \end{matrix}$$

$$A^4 = \begin{matrix} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \begin{bmatrix} 11 & 2 & 6 & 6 \\ 2 & 3 & 4 & 4 \\ 6 & 4 & 5 & 6 \\ 6 & 4 & 6 & 7 \end{bmatrix} \end{matrix}$$

To find the number of walks length 4 between vertices b and c, just simply do a pair-wise multiplication and addition of the elements corresponding to the row for vertex 'b' in A^2 with the elements corresponding to the column for vertex 'c' in A^2 .

Note: Rule for Matrix Multiplication

To find the value of an entry in cell (i, j) in the product matrix $P = A * B$,

Do a pair-wise multiplication and addition of the elements in row 'i' of the first matrix A and the elements in column 'j' of the second matrix B.