Graphs

- Motivation and Terminology
- Representations
- Traversals
- Three Problems

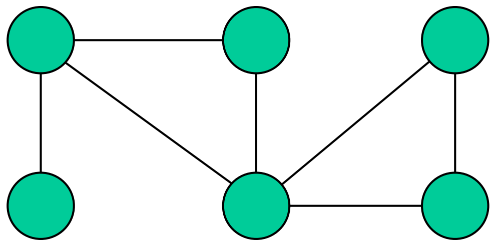
Graphs

A graph *G* consists of a set of *vertices V* together with a set *E* of vertex pairs or *edges*.

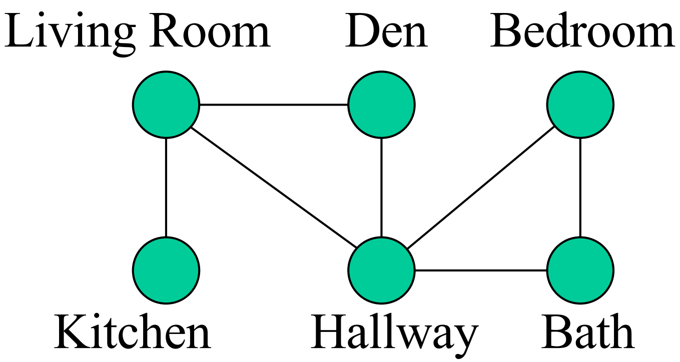
G = (V,E) [in some texts they use G(V,E)].

We also use V and E to represent # of nodes and edges

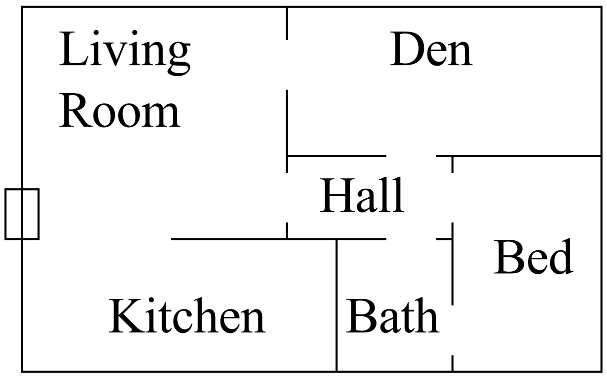
Graphs are important because any binary relation is a graph, so graphs can be used to represent essentially *any* relationship.



Graph Interpretations



The vertices could represent rooms in a house, and the edges could indicate which of those rooms are connected to each other.



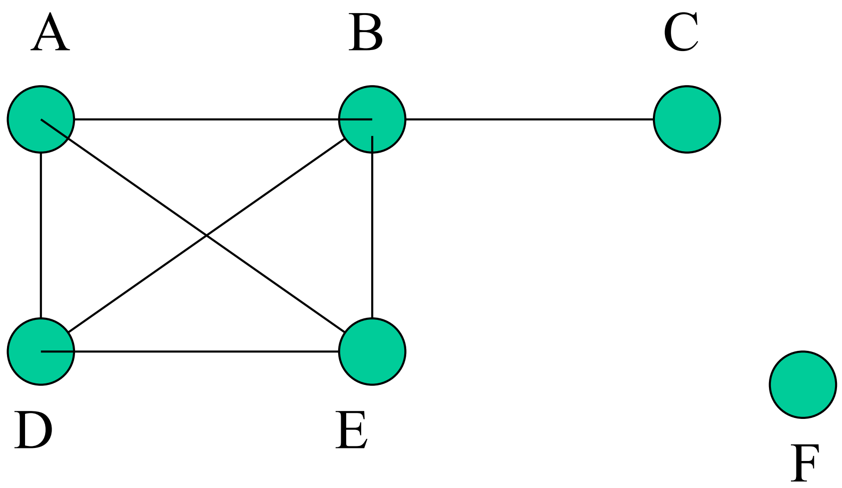
Apartment Blueprint

Sometimes a using a graph will be an easy simplification for a problem.

More interpretations

- Vertices are cities and edges are the roads connecting them.
- Edges are the components in a circuit and vertices are junctions where they connect.
- Vertices are software packages and edges indicate those that can interact.
- Edges are phone conversations and vertices are the households being connected.

Friendship Graphs



Each vertex represents a person, and each edge indicates that the two people are friends.

Graph Terminology

Directed and undirected graphs

A graph is said to be ***undirected*** if edge (x, y) always implies (y, x). Otherwise it is said to be ***directed***. Often called an ***arc***.

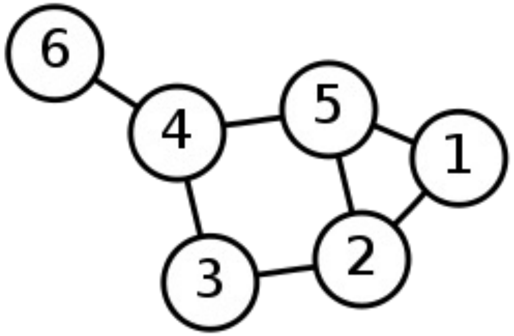
Loops, multiedges, and simple graphs

An edge of the form (x, x) is said to be a ***loop***. If x was y's friend several times over, we can model this relationship using ***multiedges***. A graph is said to be ***simple*** if it contains no loops or multiedges.

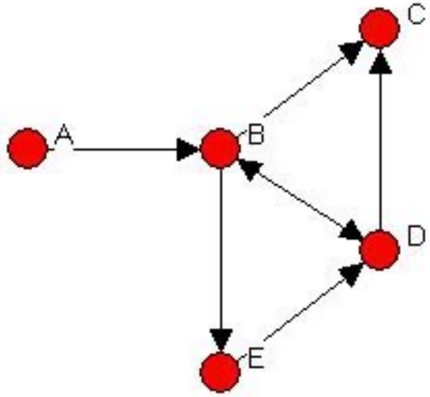
Weighted edges

A graph is said to be ***weighted*** if each edge has an associated numerical attribute. In an ***unweighted*** graph, all edges are assumed to be of equal weight.

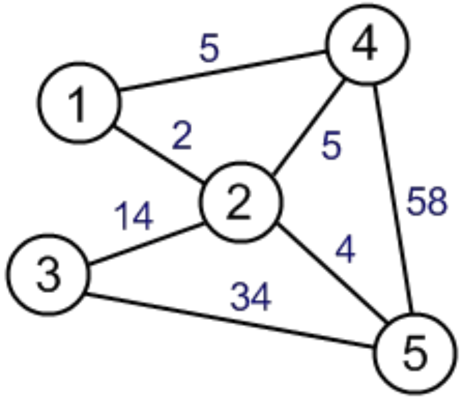
Kinds of Graphs



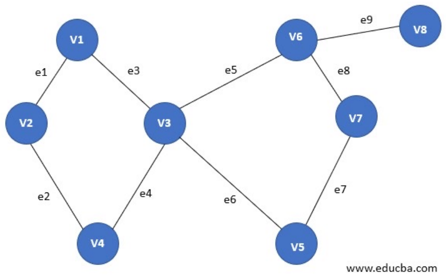
Undirected graph



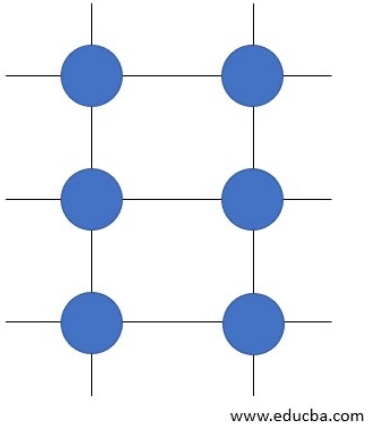
Directed graph



Weighted Undirected graph



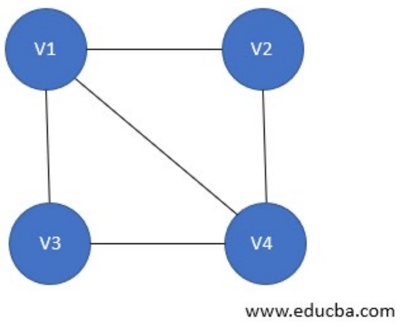
Finite graph



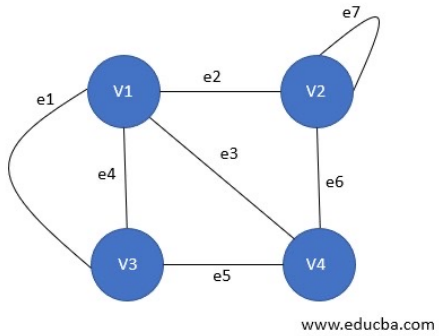
Infinite graph

Trivial graph

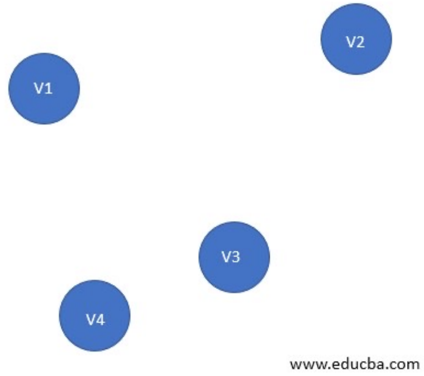
Kinds of Graphs



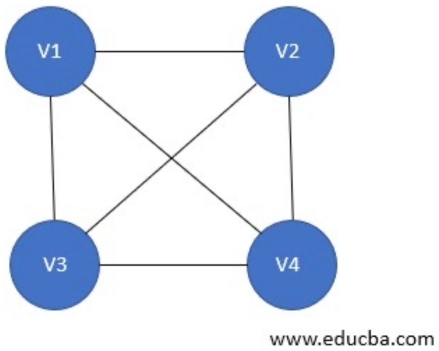
Simple graph



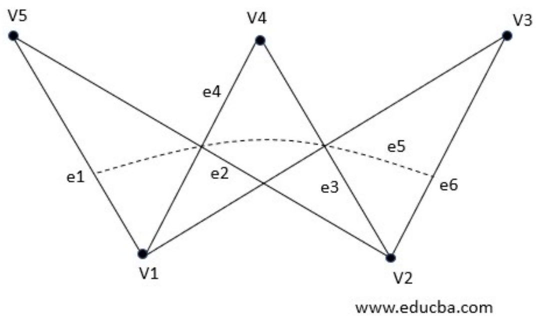
Parallel graph



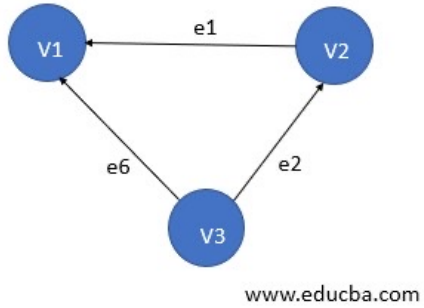
Null graph



Complete graph



Bipartite graph



Directed Acyclic graph

Terminology continued...

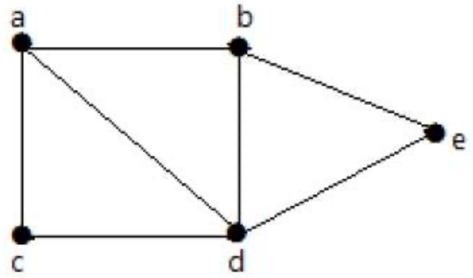
Paths

A ***path*** is any sequence of edges that connect two vertices. A ***simple path*** never goes through any vertex more than once. The ***shortest path*** is the minimum number edges needed to connect two vertices.

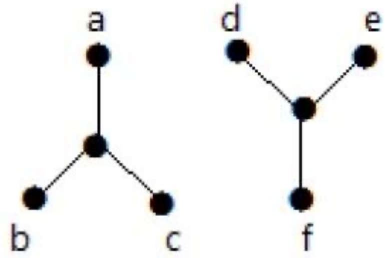
Connectivity

The “six degrees of separation” theory argues that there is always a short path between any two people in the world. A graph is ***connected*** if there a path between any two vertices. A directed graph is ***strongly connected*** if there is always a directed path between vertices. Any subgraph that is connected can be referred to as a ***connected component***.

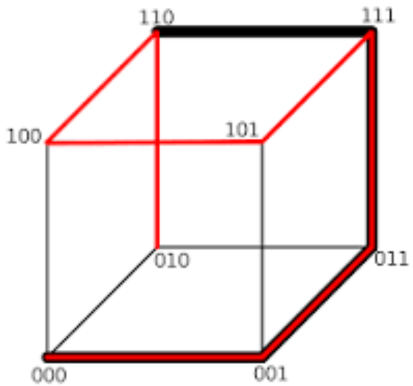
Path and Connectivity



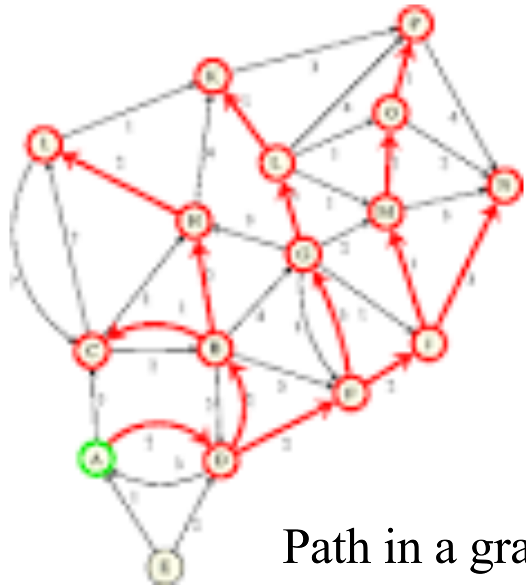
Connected graph



Disconnected graph



Path in a graph



Path in a graph

Still More Terminology...

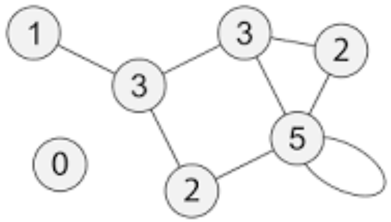
Degree and graph types

The ***degree*** of a vertex is the number of edges connected to it. The most popular person will have a vertex of the highest degree. Remote hermits may have degree-zero vertices. In ***dense*** graphs, most vertices have high degree. In ***sparse*** graphs, most vertices have low degree. In a ***regular graph***, all vertices have exactly the same degree. Degree can be **Indegree** and/or **Outdegree.**

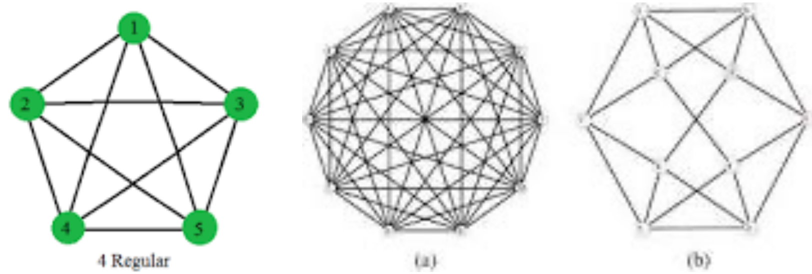
Clique

A graph is called ***complete*** if every pair of vertices is connected by an edge. A ***clique*** is a sub-graph that is complete.

Degree

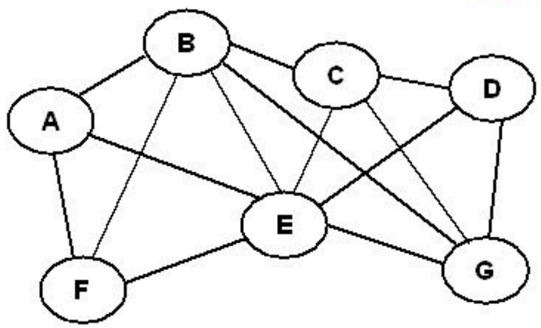


Graph with varying degree

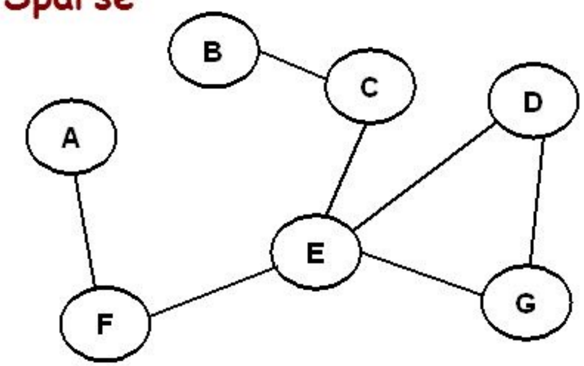


Regular Graph (with same degree)

Dense vs. Sparse

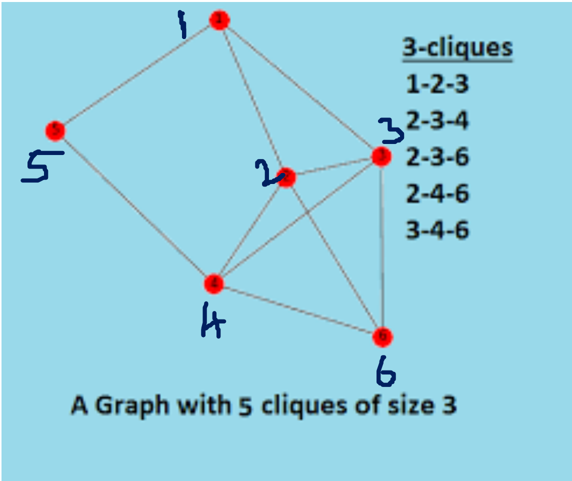


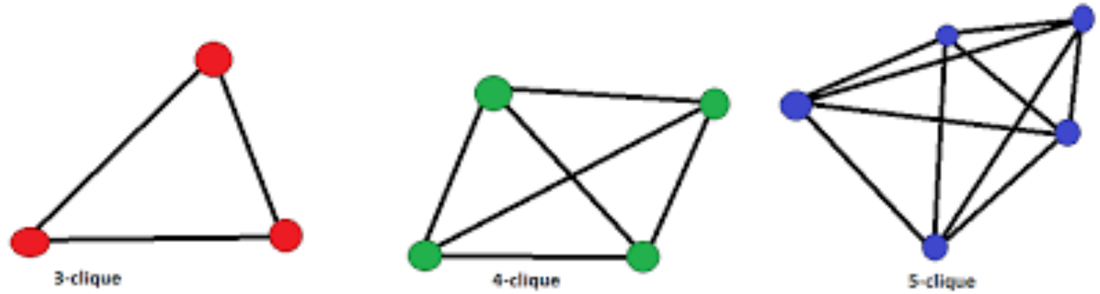
Dense graphs (many edges between nodes)

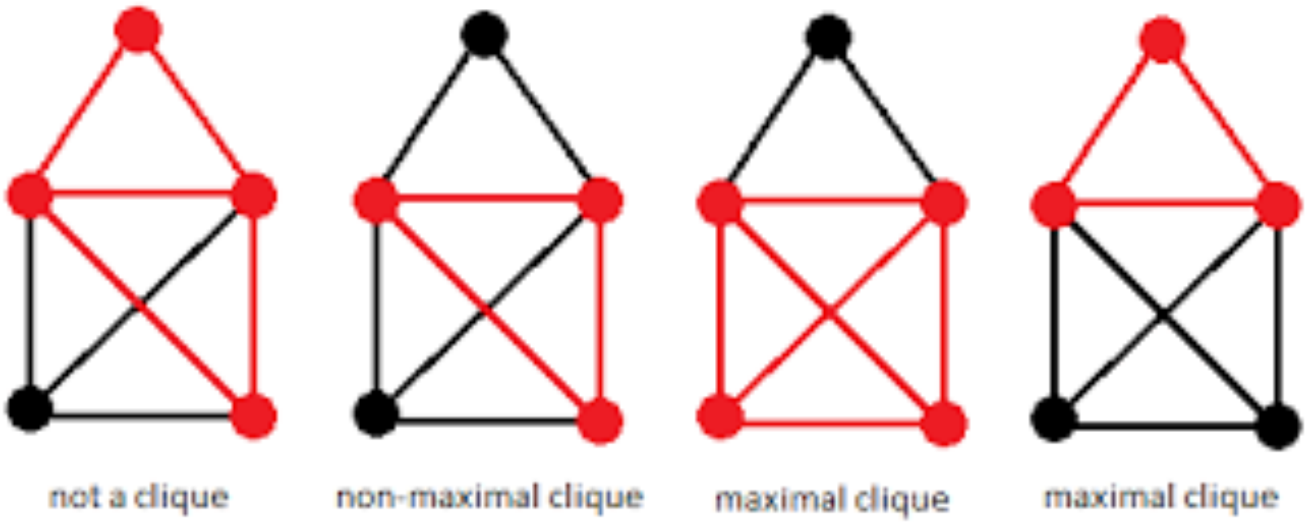


Sparse graphs (few edges between nodes)

Clique


3-cliques
1-2-3
2-3-4
2-3-6
2-4-6
3-4-6
1
2
3
4
5
6
A Graph with 5 cliques of size 3


3-clique
4-clique
5-clique

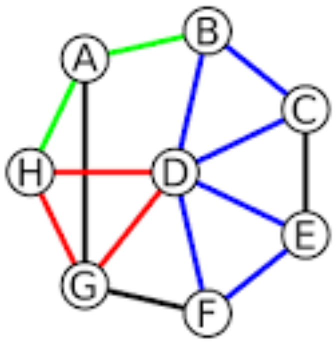

not a clique
non-maximal clique
maximal clique
maximal clique

Yet More Terminology...

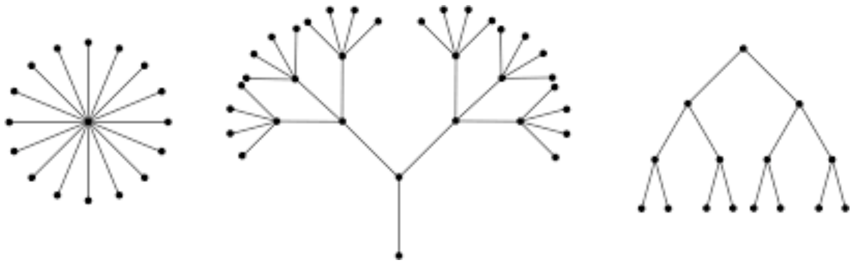
Cycles and Dags

A ***cycle*** is a path where the last vertex is adjacent to the first. A cycle in which no vertex is repeated is said to be a ***simple cycle***. The shortest cycle in a graph determines the graph's ***girth***. A simple cycle that passes through every vertex is said to be a ***Hamiltonian cycle***. An undirected graph with no cycles is a ***tree*** if it is connected, or a ***forest*** if it is not. A directed graph with no directed cycles is said to be a ***directed acyclic graph*** (or a DAG)

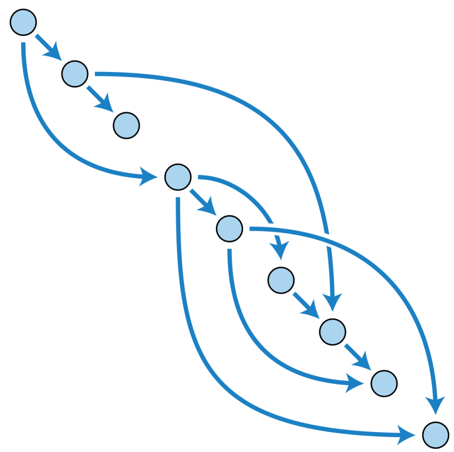
Cycles and DAGs



Graph with cycles



Trees in a Forest



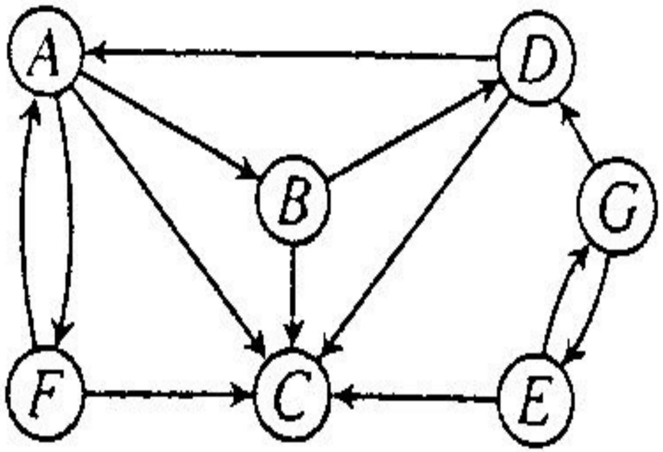
DAG

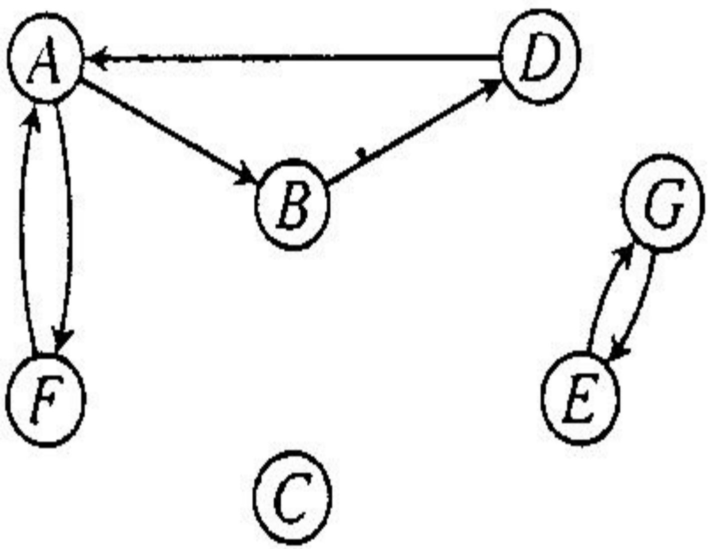
Graphs

- ***Graph* G = (*V*, *E*)**
  – V = set of vertices
  – E = set of edges $\subseteq (V \times V)$
- Types of graphs
  – **Undirected:** edge $(u, v) = (v, u)$; for all v, $(v, v) \notin E$ (No self loops.)
  – **Directed:** (u, v) is edge from u to v, denoted as $u \rightarrow v$. Self loops are allowed (also called as **digraph**)
  – **Weighted:** each edge has an associated weight, given by a weight function $w : E \rightarrow \mathbf{R}$.
  – **Dense:** $|E| \approx |V|^2$.
  – **Sparse:** $|E| << |V|^2$.
- $|E| = O(|V|^2)$

Strongly Connected Components of a Digraph

- Strongly connected:
  - A directed graph is strongly connected if and only if, for each pair of vertices v and w, there is a path from v to w.
- Strongly connected component:





Strongly connected Components and Equivalence Relations

- Strongly Connected Components may be defined in terms of an equivalence relation, S, on the vertices
  - vSw iff there is a path from v to w and
  - a path from w to v
- Then, a strongly connected component consists of one equivalence class, C, along with all edges vw such that v and w are in C.

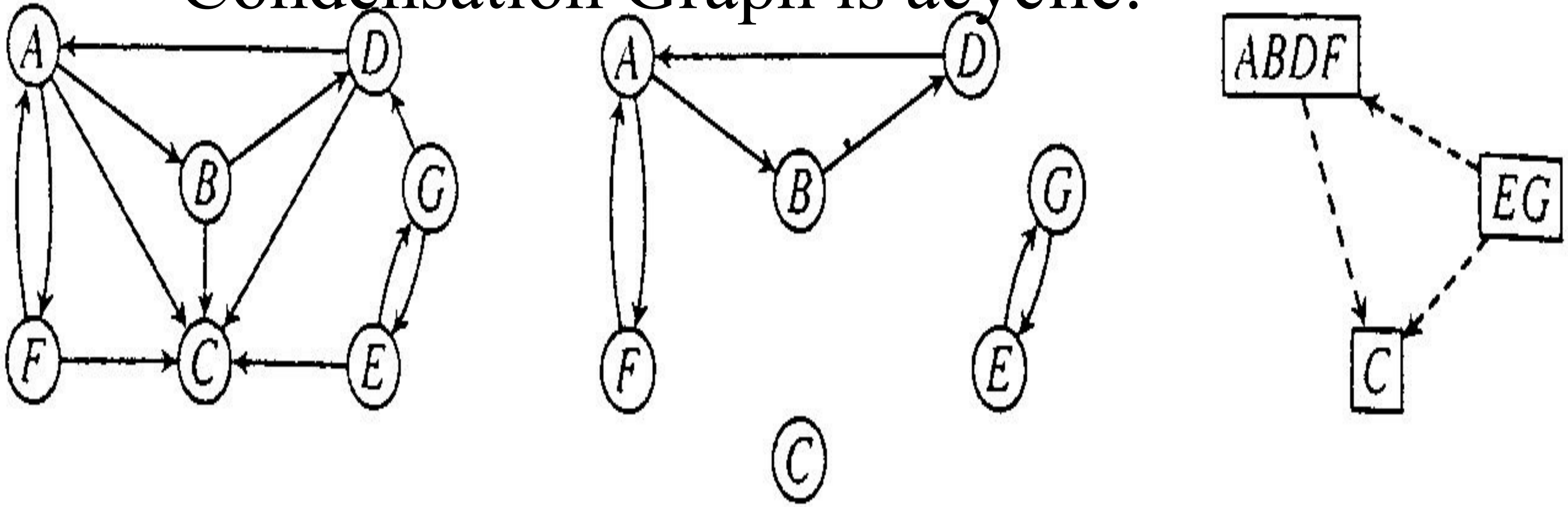
Condensation graph

- The strongly connected components of a digraph can each be collapsed to a single vertex yielding a new digraph that has no cycles.

- Condensation graph:
  - Let $S_1, S_2, \ldots S_p$ be the strong components of G.
  - The condensation graph of G denoted as G↓, is the digraph G↓ = (V',E'),
  - where V' has p elements $s_1, s_2, \ldots s_p$ and
  - $s_i s_j$ is in E' if and only if $i \neq j$ and
  - there is an edge in E from some vertex in S_i to some vertex in S_j.

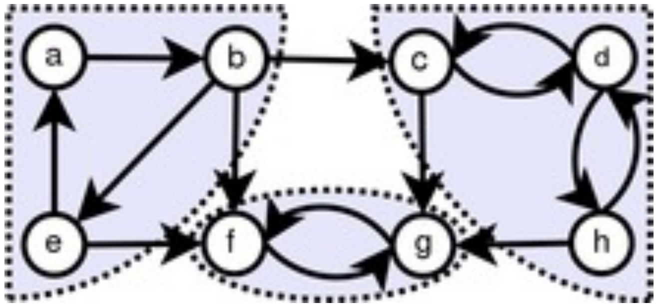
Condensation graph and its strongly connected components

- Condensation Graph is acyclic.

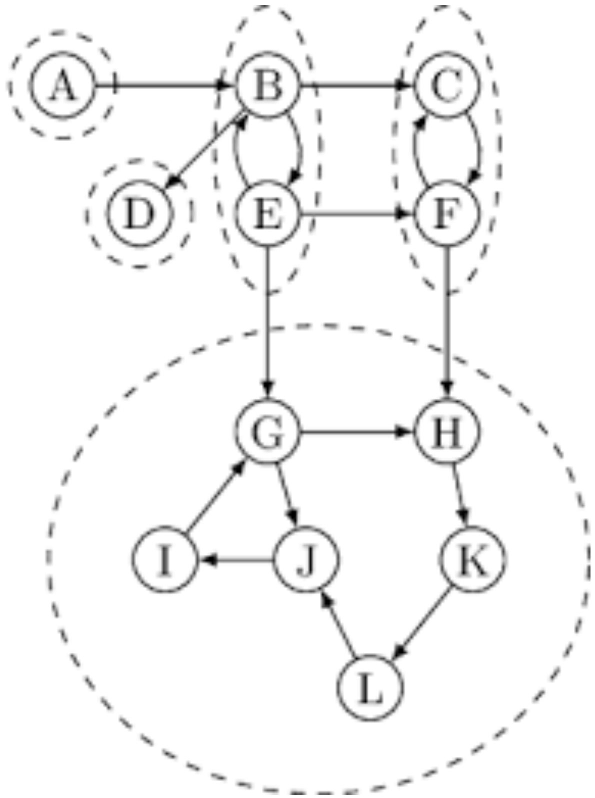


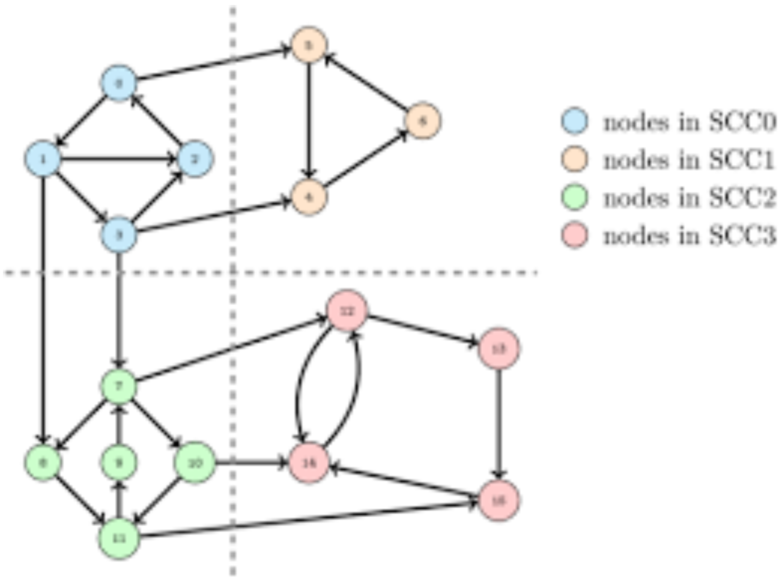
(a) The digraph (b) Its strong components (c) Its condensation graph

Examples



Graph with strongly connected components marked

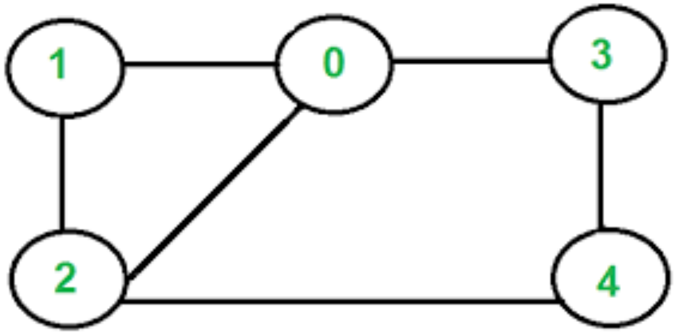




Bi-connected components of an Undirected graph

- Problem:
  - If any *one* vertex (and the edges incident upon it) are removed from a connected graph,
  - is the remaining subgraph still connected?
- Biconnected graph: **(No separation edge and separation vertex)**
  - A connected undirected graph G is said to be biconnected if it remains connected after removal of any one vertex and the edges that are incident upon that vertex. **Separation edge is also called cut edge and separation vertex is called cut vertex.**
- Biconnected component:
  - A biconnected component of a undirected graph is a maximal biconnected subgraph, that is, a biconnected subgraph not contained in any larger biconnected subgraph.
- Articulation point:
  - A vertex v is an articulation point for an undirected graph G if there are distinct vertices w and x (distinct from v also) such that v is in every path from w to x.

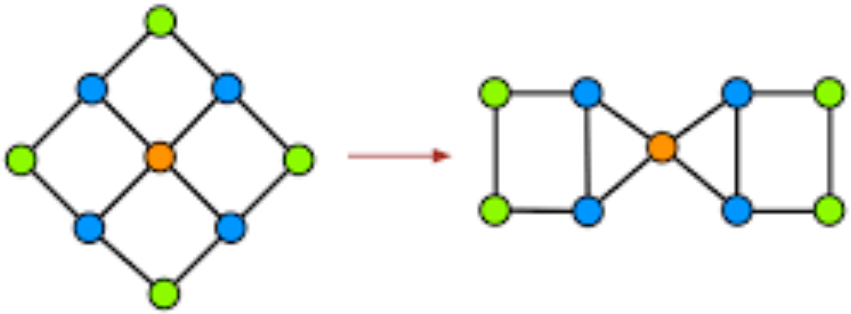
Bi-connected graph

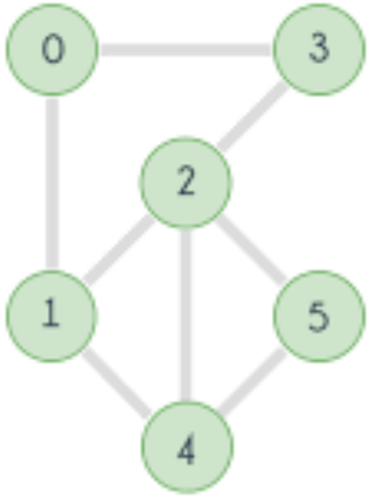


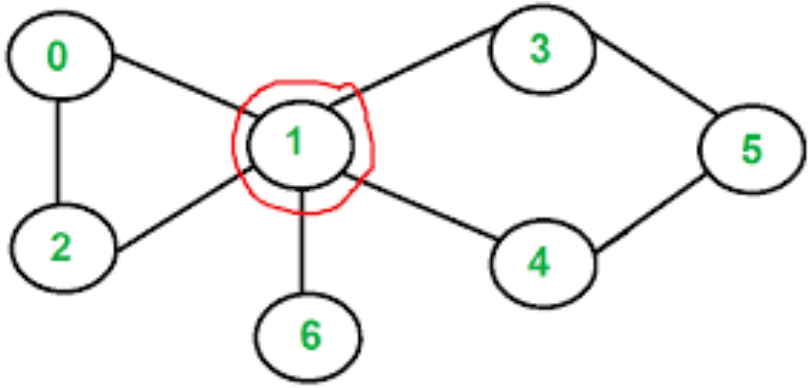
Biconnected

Biconnected

Not biconnected





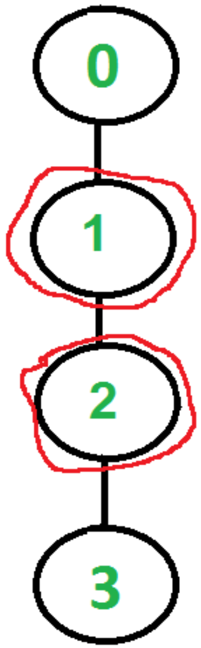


Articulation Point is 1

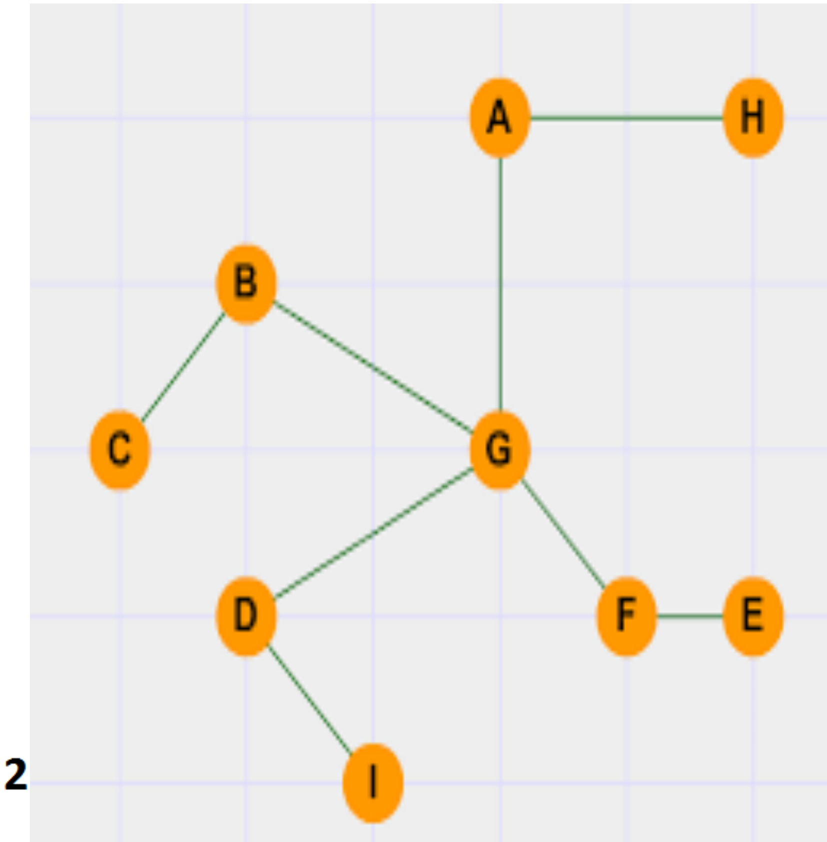
Not a biconnected graph

Articulation point

- An articulation point of a graph is a vertex v such that when we remove v and all edges incident upon v , we break a **connected component** of the graph into two or more pieces.
- A **connected** graph with no articulation points is said to be **biconnected**.

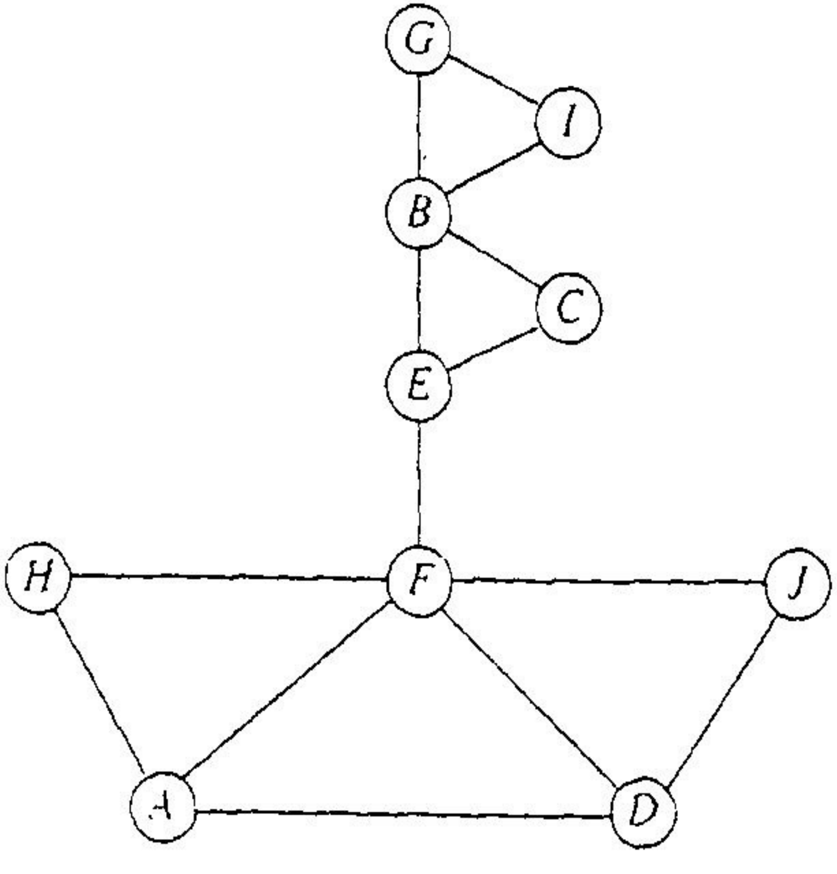


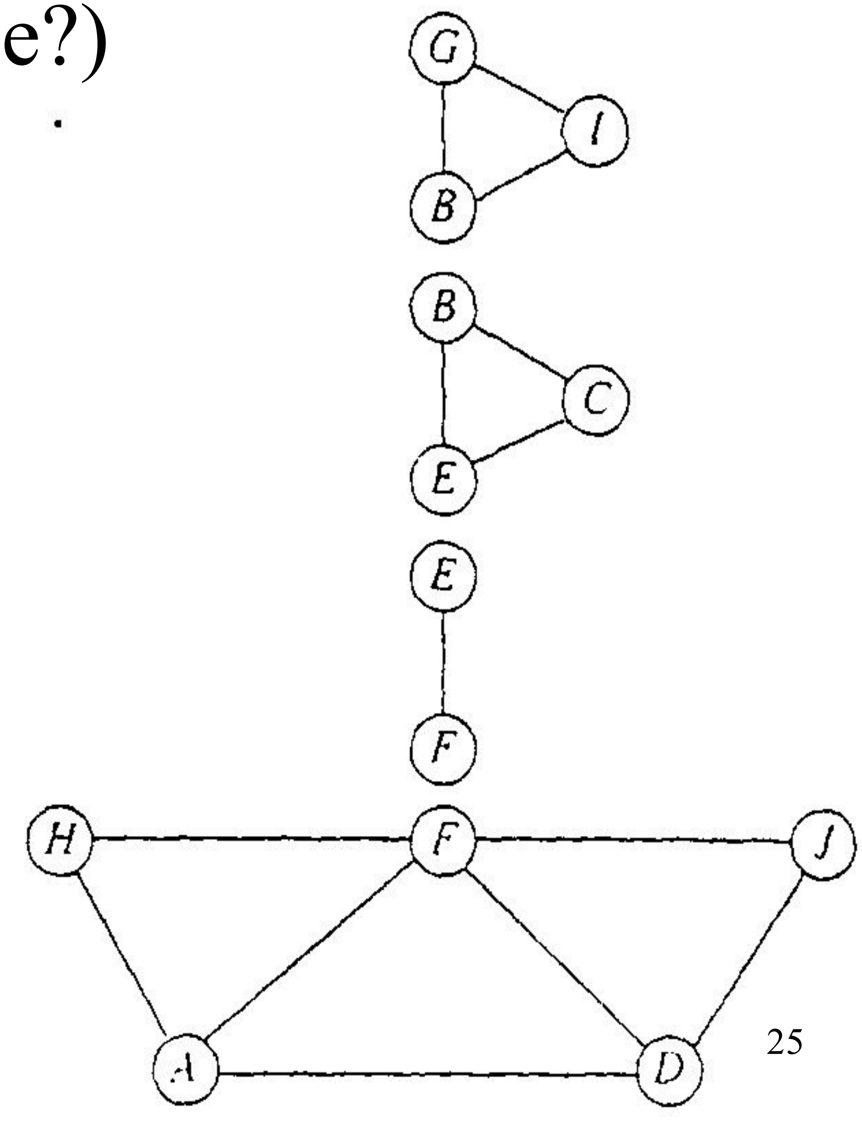
Artiulation Points are 1 & 2



Bi-connected components, e.g.

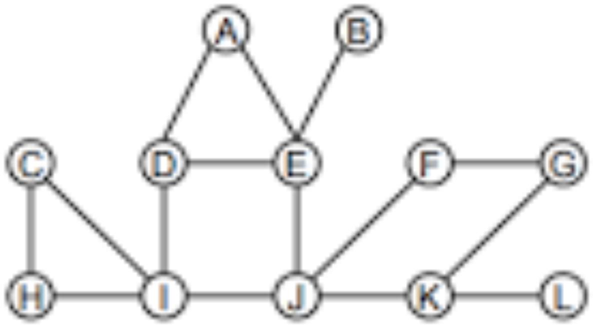
- Some vertices are in more than one component
- (which vertices are these?)





Example

Find and list the biconnected components of the graph in problem 1.



- Component 1: $\{\{A, D\}, \{A, E\}, \{D, E\}, \{D, I\}, \{E, J\}, \{I, J\}\}$
- Component 2: $\{\{B, E\}\}$
- Component 3: $\{\{C, H\}, \{C, I\}, \{H, I\}\}$
- Component 4: $\{\{F, G\}, \{F, J\}, \{G, K\}, \{J, K\}\}$, and
- Component 5: $\{\{K, L\}\}$

Example

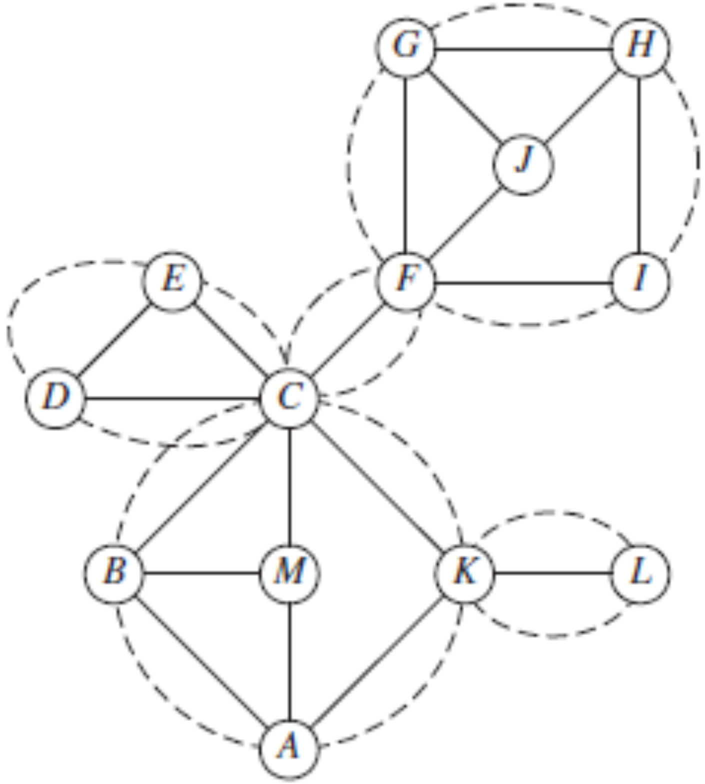
The number of Bi-connected components of the graph is:

1. {10-12, 12-13, 10-13}
2. {10-11}
3. {8-9}
4. {7-8}
5. {1-2}
6. {1-3, 1-6, 3-7, 5-7, 4-5,5-6, 4-6, }

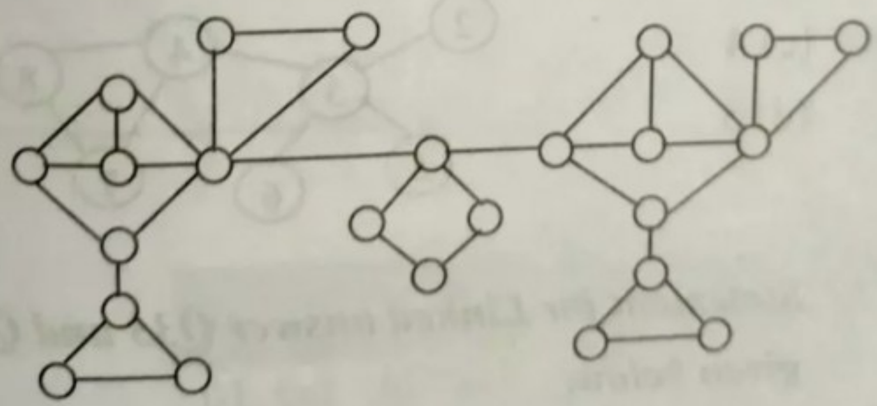
(A) 11 (B) 10

(C) 2 (D) 6

Example



22. A maximal connected subgraph which does not contain articulation points and bridges is called biconnected component. Then in the following graph, how many biconnected components are there?



(a) 6 (b) 5 (c) 7 (d) None

Exercises

Bob loves foreign languages and wants to plan his course schedule to take the following nine language courses: LA15, LA16, LA22, LA31, LA32, LA126, LA127, LA141, and LA169. The course prerequisites are:

- LA15: (none)
- LA16: LA15
- LA22: (none)
- LA31: LA15
- LA32: LA16, LA31
- LA126: LA22, LA32
- LA127: LA16
- LA141: LA22, LA16
- LA169: LA32.

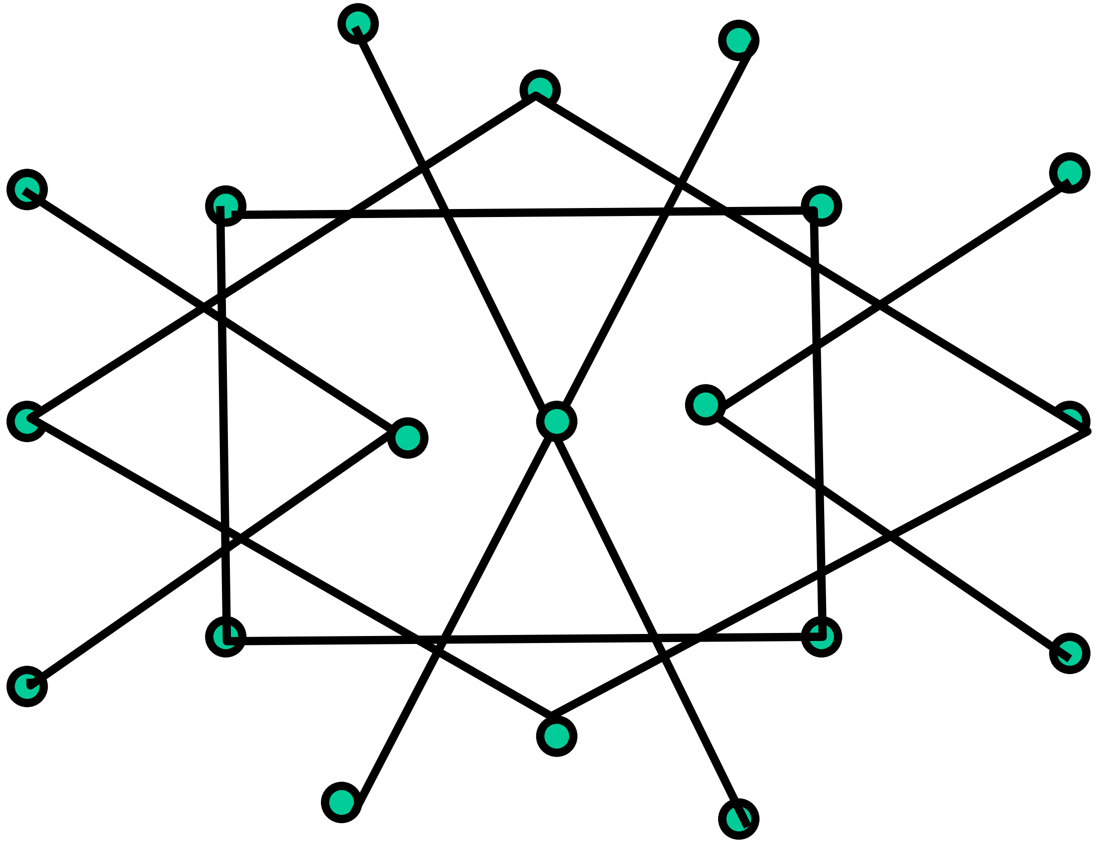
Find a sequence of courses that allows Bob to satisfy all the prerequisites.

Draw a directed graph, having as its eight vertices the strings 'ape', 'ate', 'eat', 'era', 'pea', 'rap', 'rat', and 'tea', and including an edge from word *x* to word *y* whenever the last two letters of *x* are the same as the first two letters of *y*; for instance, you should include an edge from 'ape' to 'pea'.

Consider the vertices 1 to 10. An edge is created iff the sum of two vertices are divisible by 4. Draw the graph and find the total number of edges.

Exercises

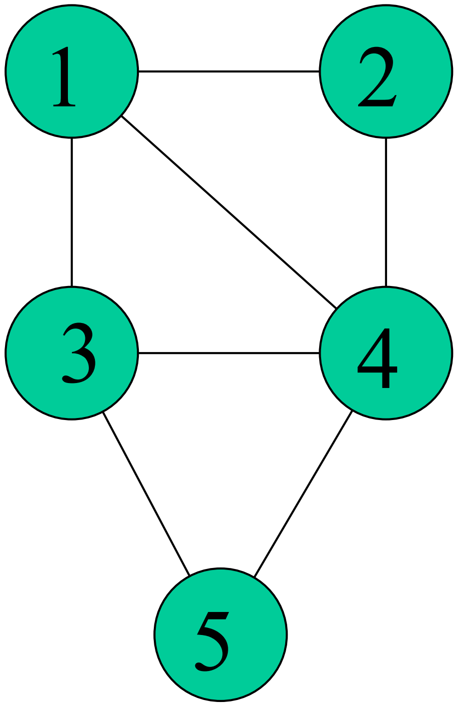
- Is the graph connected? If not identify the connected components.



Graphs

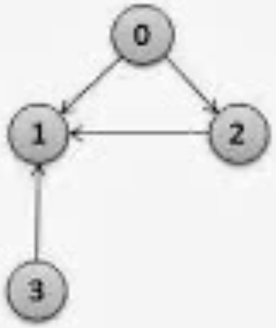
- Motivation and Terminology
- Representations
- Traversals
- Three Problems

Adjacency Matrix



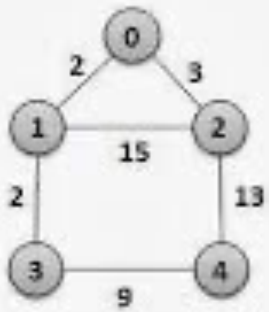
	1	2	3	4	5
1	0	1	1	1	0
2	1	0	0	1	0
3	1	0	0	1	1
4	1	1	1	0	1
5	0	0	1	1	0

Adjacency Matrix



	0	1	2	3
0	0	1	1	0
1	0	0	0	0
2	0	1	0	0
3	0	1	0	0

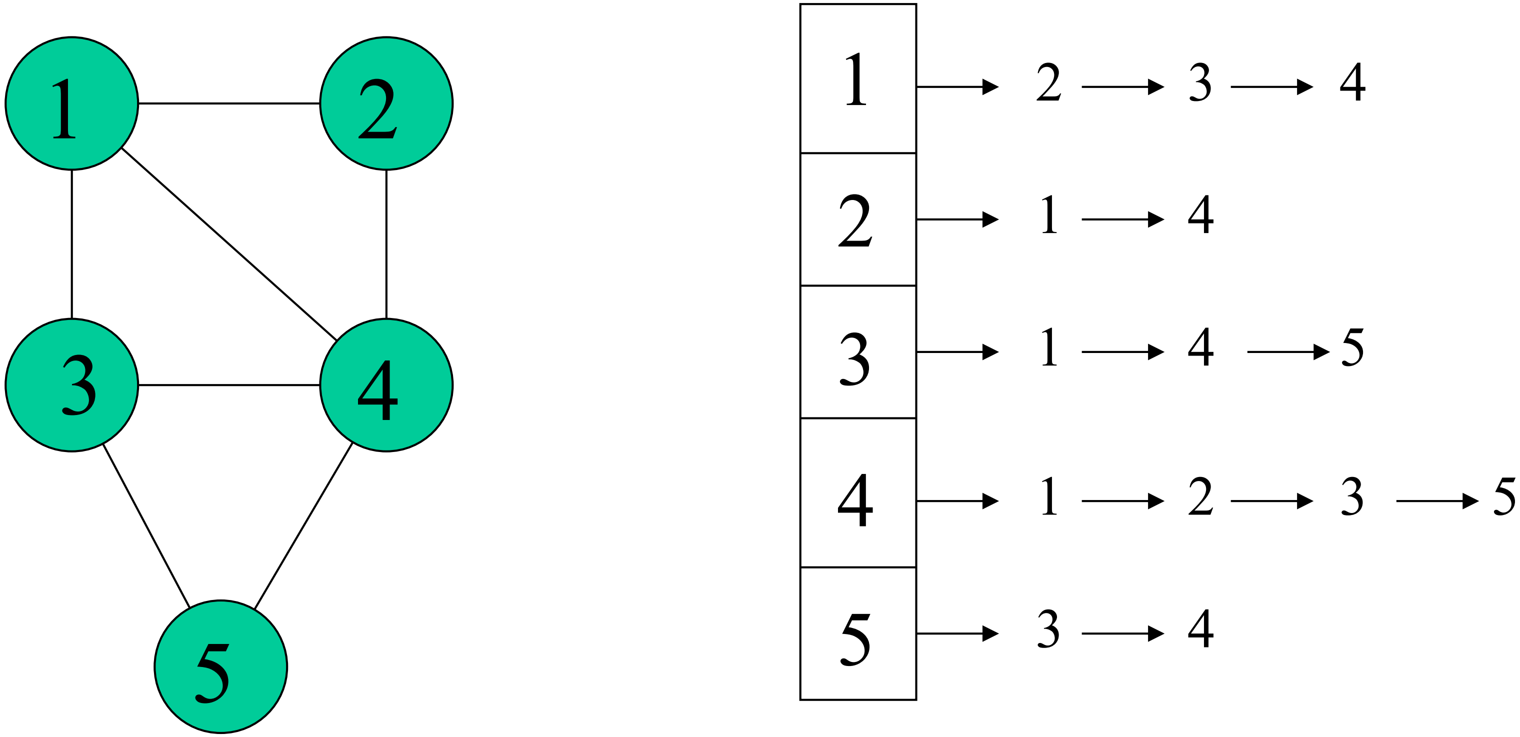
Adjacency Matrix Representation of Directed Graph



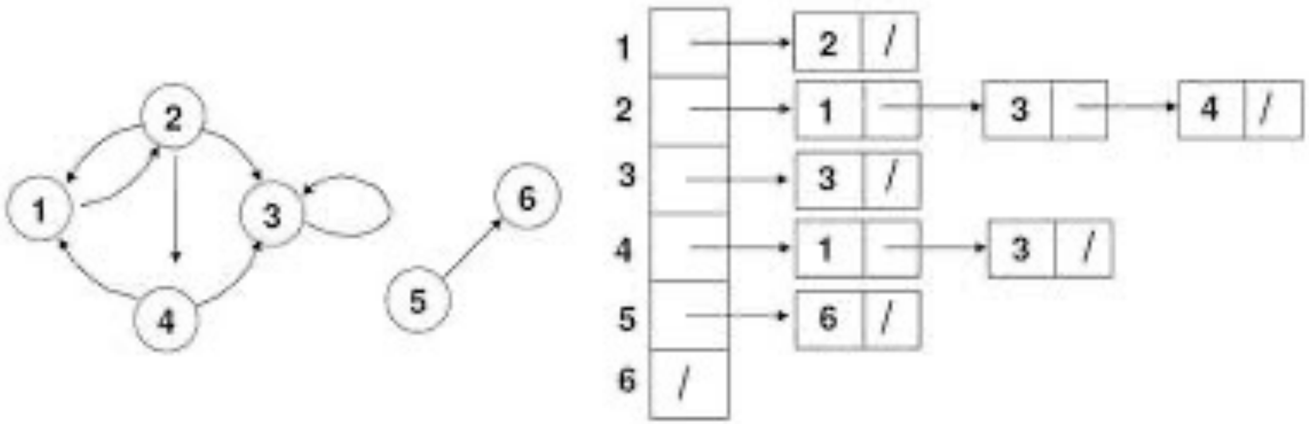
	0	1	2	3	4
0	0	2	3	0	0
1	2	0	15	2	0
2	3	15	0	0	13
3	0	2	0	0	9
4	0	0	13	9	0

Adjacency Matrix Representation of Weighted Graph

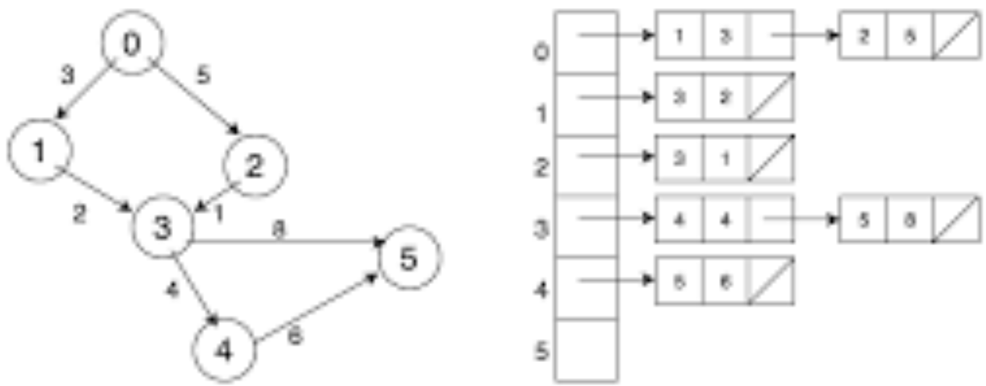
Adjacency List



Adjacency List



Directed Graph **Adjacency List Representation**



Tradeoffs Between Adjacency Lists and Adjacency Matrices

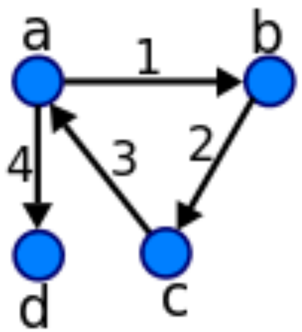
Comparison	**Winner (for worst case)**
Faster to test if (x, y) exists?	matrices: $\Theta(1)$ vs. $\Theta(V)$
Faster to find vertex degree?	lists: $\Theta(1)$ vs. $\Theta(V)$
Less memory on sparse graphs?	lists: $\Theta(V+E)$ vs. $\Theta(V^2)$
Less memory on dense graphs?	matrices: (small win)
Edge insertion or deletion?	matrices: $\Theta(1)$ vs. $\Theta(V)$
Faster to traverse the graph?	lists: $\Theta(E+V)$ vs. $\Theta(V^2)$
Better for most problems?	**lists**

Incidence matrix

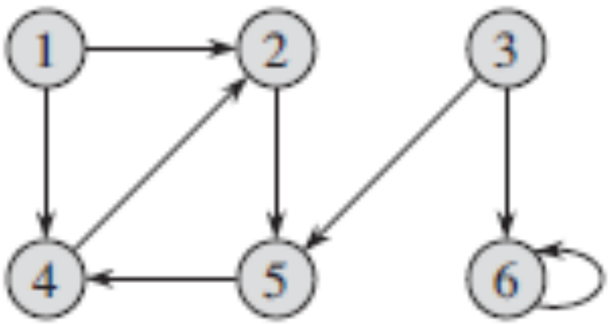
The ***incidence matrix*** of a directed graph $G = (V, E)$ with no self-loops is a $|V| \times |E|$ matrix $B = (b_{ij})$ such that

$$b_{ij} = \begin{cases} 1 & \text{if edge } j \text{ leaves vertex } i \text{ ,} \\ \text{-1} & \text{if edge } j \text{ enters vertex } i \text{ ,} \\ 0 & \text{otherwise .} \end{cases}$$

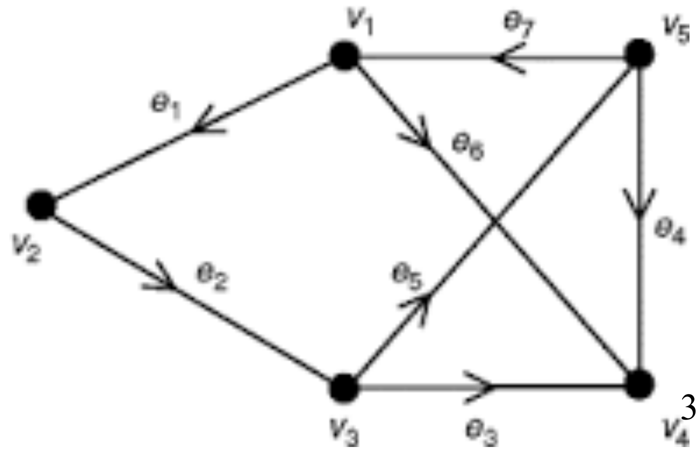
$$\begin{array}{c|c} & \begin{array}{ccccccc} e_1 & e_2 & e_3 & e_4 & e_5 & e_6 & e_7 \end{array} \\ \begin{array}{c} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{array} & \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 & -1 \\ -1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & -1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 & 1 \end{bmatrix} \end{array}$$



	1	2	3	4
a	1	0	-1	1
b	-1	1	0	0
c	0	-1	1	0
d	0	0	0	-1



←Try for this…



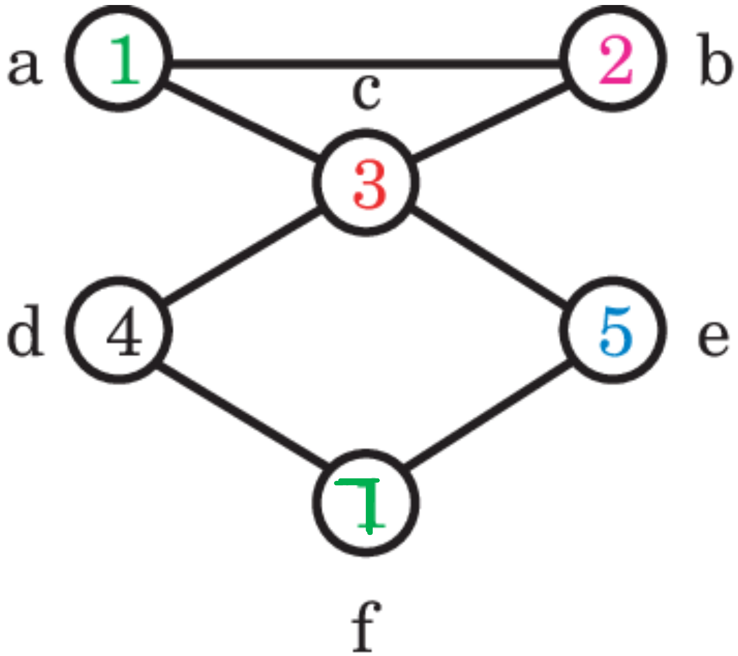
Graph Squaring

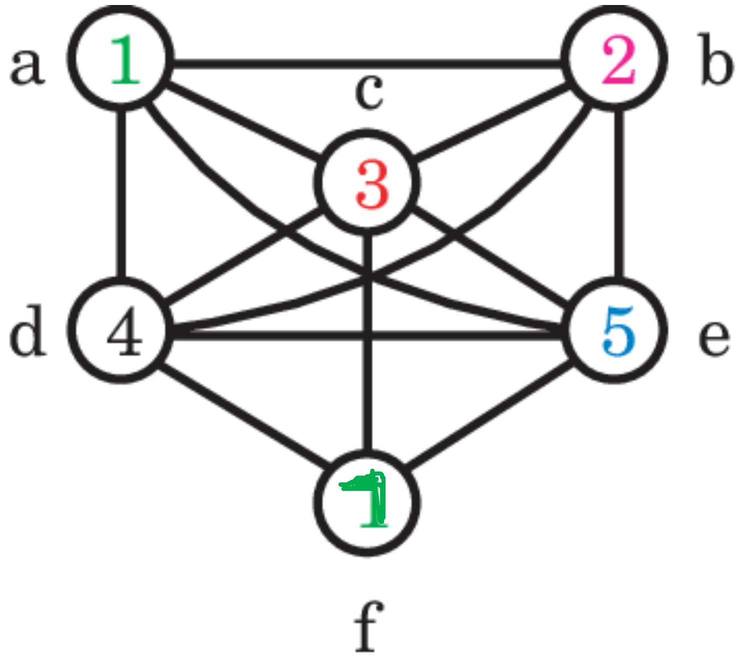
The square of a directed graph $G = (V, E)$ is the graph $G^2 = (V, E^2)$, such that $(x, y) \in E^2$ iff, for some z, both (x, z) and $(z, y) \in E$; i.e., there is a path of exactly two edges.

Try yourself….

Give efficient algorithms to square a graph on both adjacency lists and matrices.

Example





G^2 with Adjacency Matrices

To discover whether there is an edge (x, y) in E^2, we do the following.

> For each possible intermediate vertex z, we check whether (x, z) and (z, y) exist in O(1); if so for any z, mark (x,y) in E^2.

Since there are $O(V)$ intermediate vertices to check, and $O(V^2)$ pairs of vertices to ask about, this takes $O(V^3)$ time.

G^2 with Adjacency Lists

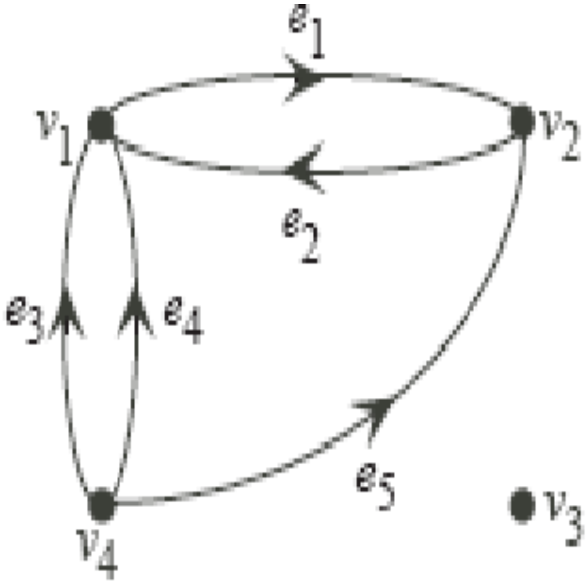
We use an adjacency matrix as temporary storage.

For each edge (x, z), we run through all the edges (z,y) from z in O(V) time, updating the adjacency matrix for edge (x,y). We convert back to adjacency lists at the end.

It takes O(VE) to construct the edges, and O(V^2) to initialize and read the adjacency matrix, for a total of $O((V+E)V)$. Since $E+1 \geq V$ (unless the graph is disconnected), this is usually simplified to $O(VE)$, and is faster than the previous algorithm on sparse graphs.

Exercises

- Write the adjacency matrix, incidence matrix and adjacency list representation of this graph



Graphs

- Motivation and Terminology
- Representations
- Traversals
- Three Problems

Traversing a Graph

One of the most fundamental graph problems is to traverse every edge and vertex in a graph. Applications include:

- Printing out the contents of each edge and vertex.
- Counting the number of edges.
- Identifying connected components of a graph.

For *correctness*, we must do the traversal in a systematic way so that we don't miss anything.

For *efficiency*, we must make sure we visit each edge at most twice.

Marking Vertices

The idea in graph traversal is that we mark each vertex when we first visit it, and keep track of what is not yet completely explored.

For each vertex, we maintain two flags:

- ***discovered*** - have we encountered this vertex before?
- ***explored*** - have we finished exploring this vertex?

We must maintain a structure containing all the vertices we have discovered but not yet completely explored.

Initially, only a single start vertex is set to be discovered.

Correctness of Graph Traversal

Every edge and vertex in the connected component is eventually visited.

Suppose not, i.e. there exists a vertex which was unvisited whose neighbor *was* visited. This neighbor will eventually be explored so we *would* visit it….

Traversal Orders

The order we explore the vertices depends upon the data structure used to hold the discovered vertices yet to be fully explored:

- ***Queue*** - by storing the vertices in a first-in, first out (FIFO) queue, we explore the oldest unexplored vertices first. Thus we radiate out slowly from the starting vertex, defining a so-called *breadth-first search*.
- ***Stack*** - by storing the vertices in a last-in, first-out (LIFO) stack, we explore the vertices by constantly visiting a new neighbor if one is available; we back up only when surrounded by previously discovered vertices. This defines a so-called *depth-first search*.

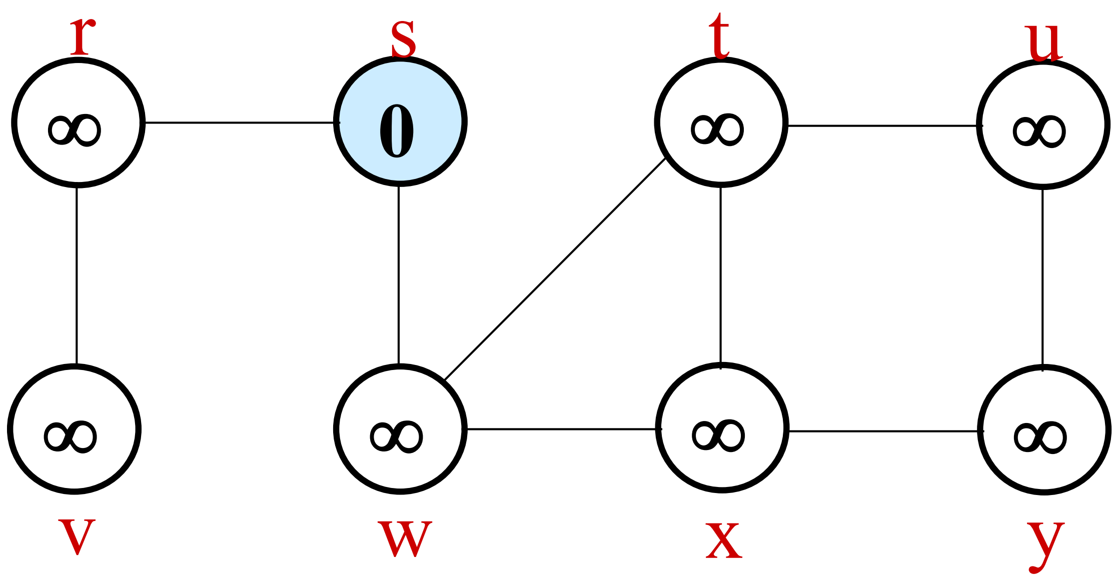
BFS(G,s)

```
for each vertex u in V[G] –  {s}
          do color[u] ← white
               d[u] ← ∝
               π[u] ← nil
color[s] ← gray
d[s] ← 0
 π[s] ← nil
Q ← Φ
enqueue(Q,s)
while Q ≠ Φ
          do u ← dequeue(Q)
                         for each v in Adj[u]
                                        do if color[v] = white
                                                       then color[v] ← gray
                                                                d[v] ← d[u] + 1
                                                                π[v] ← u
                                                                enqueue(Q,v)
                         color[u] ← black
```

white: undiscovered
gray: discovered
black: finished

Q: a queue of discovered vertices
color[v]: color of v
d[v]: distance from s to v
π[u]: predecessor of v

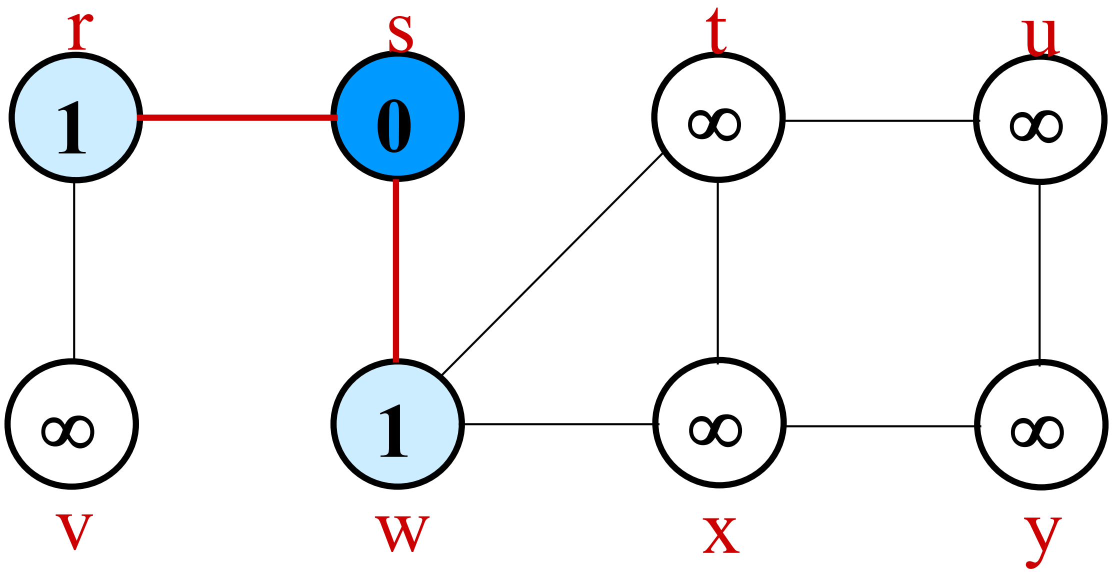
Example (BFS)



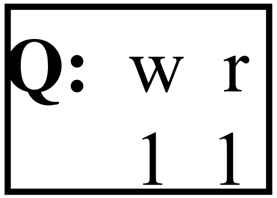
Order of visit:
s

Q:	s
	0

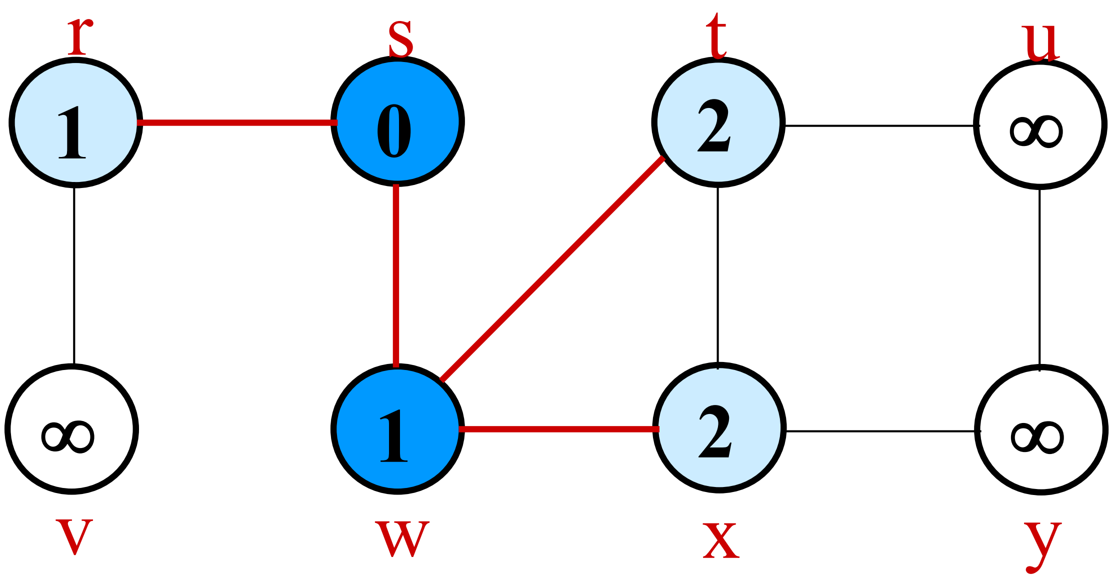
Example (BFS)



Order of visit:
s, w, r



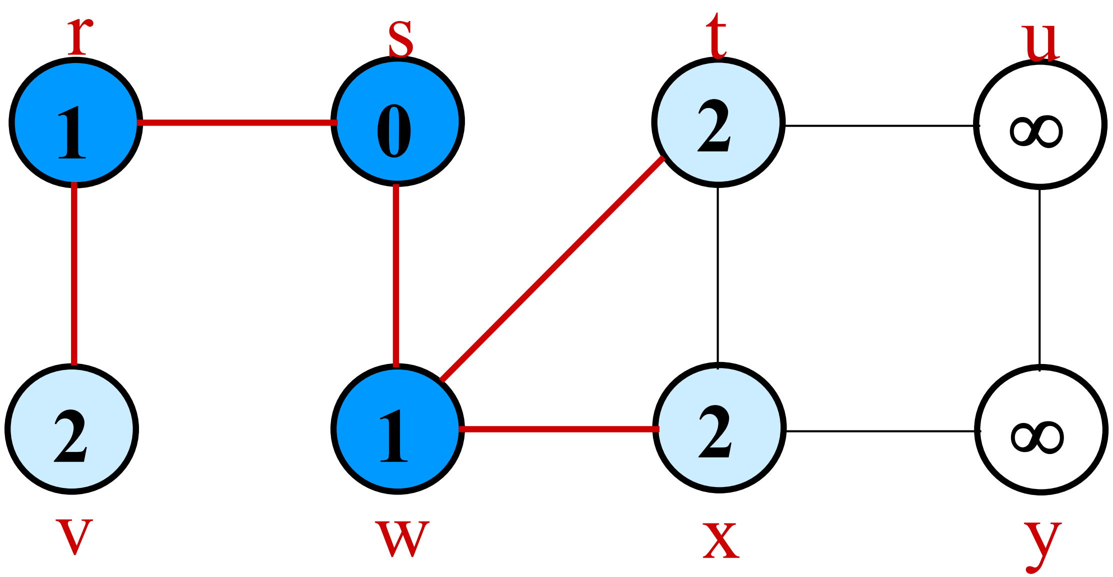
Example (BFS)



Order of visit:
s, w, r, t, x

Q:	r	t	x
	1	2	2

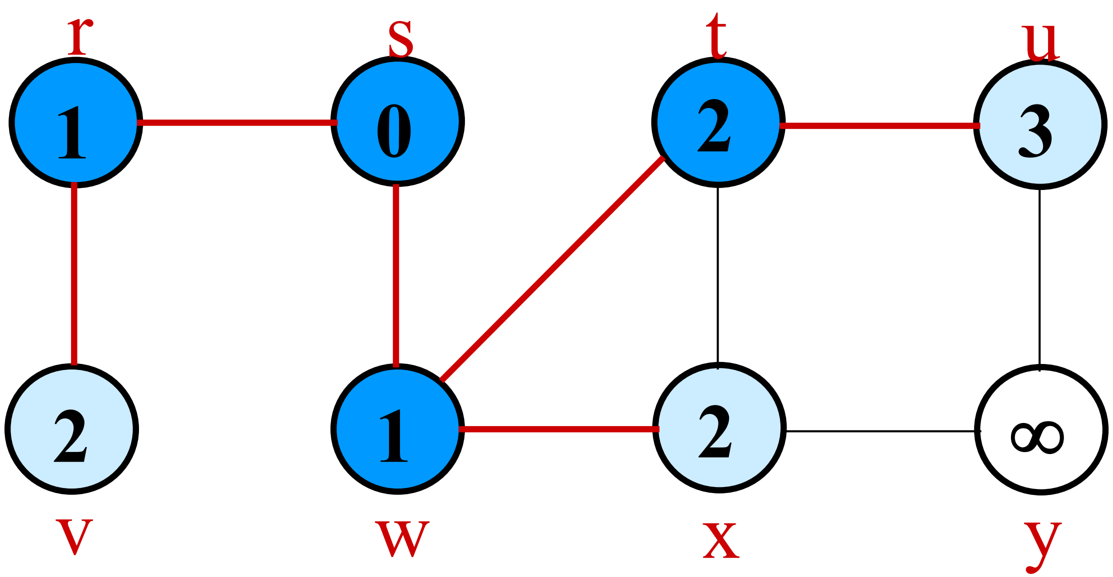
Example (BFS)



Q: t x v
2 2 2

Order of visit:
s, w, r, t, x, v

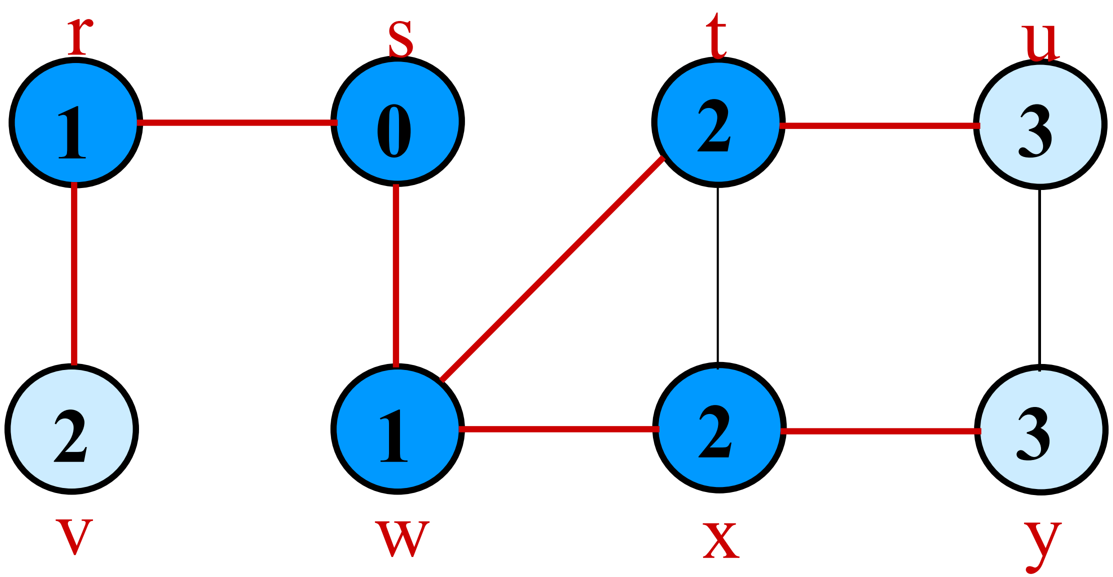
Example (BFS)



Q:	x	v	u
	2	2	3

Order of visit:
s, w, r, t, x, v, u

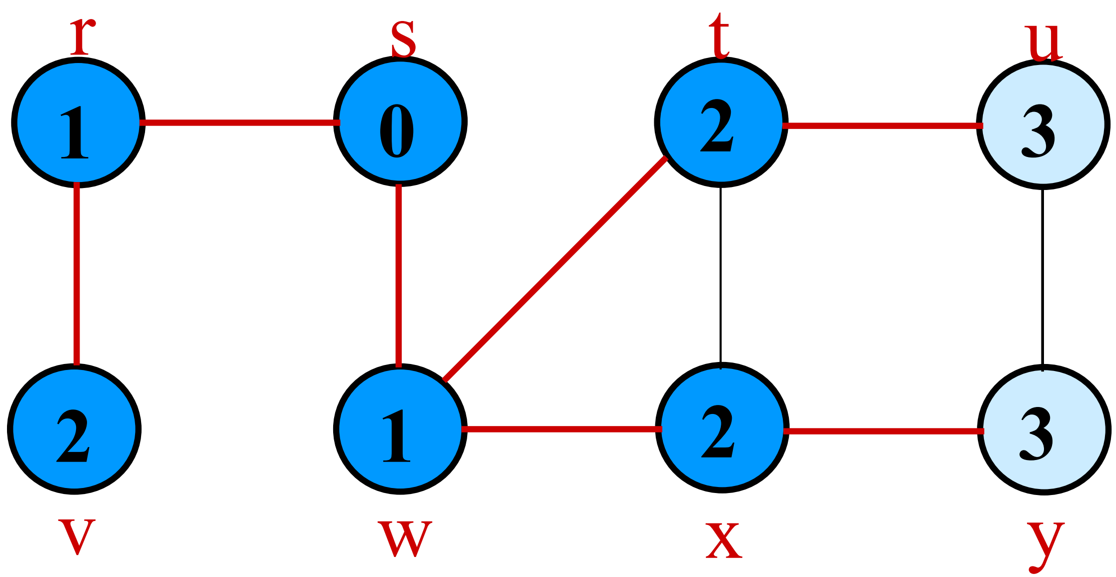
Example (BFS)



Q: v u y
2 3 3

Order of visit:
s, w, r, t, x, v, u, y

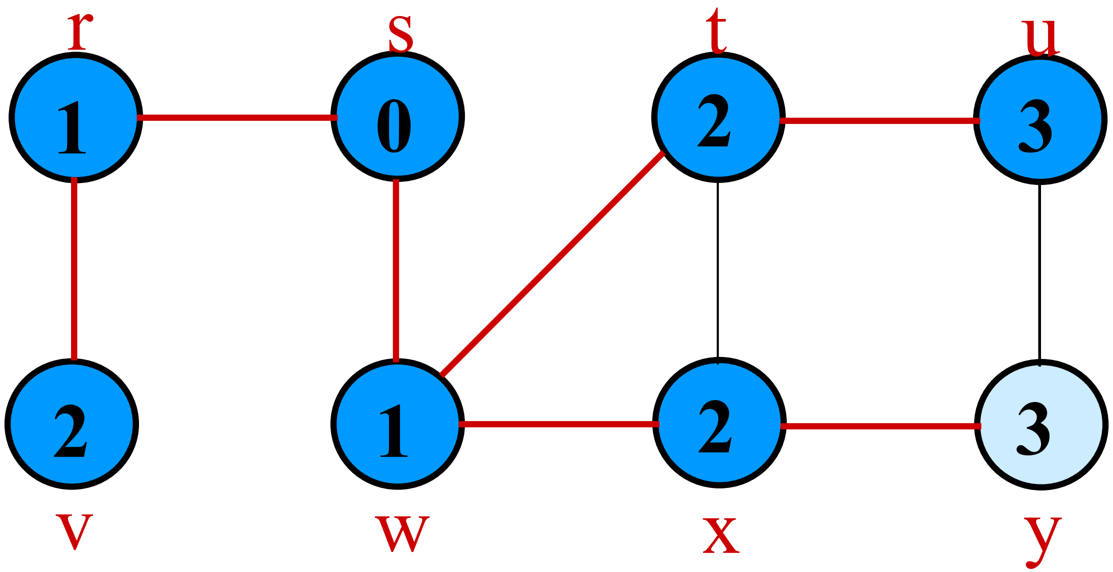
Example (BFS)



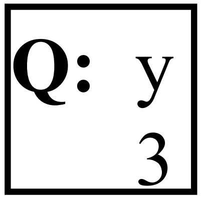
Order of visit:
s, w, r, t, x, v, u, y

Q:	u	y
	3	3

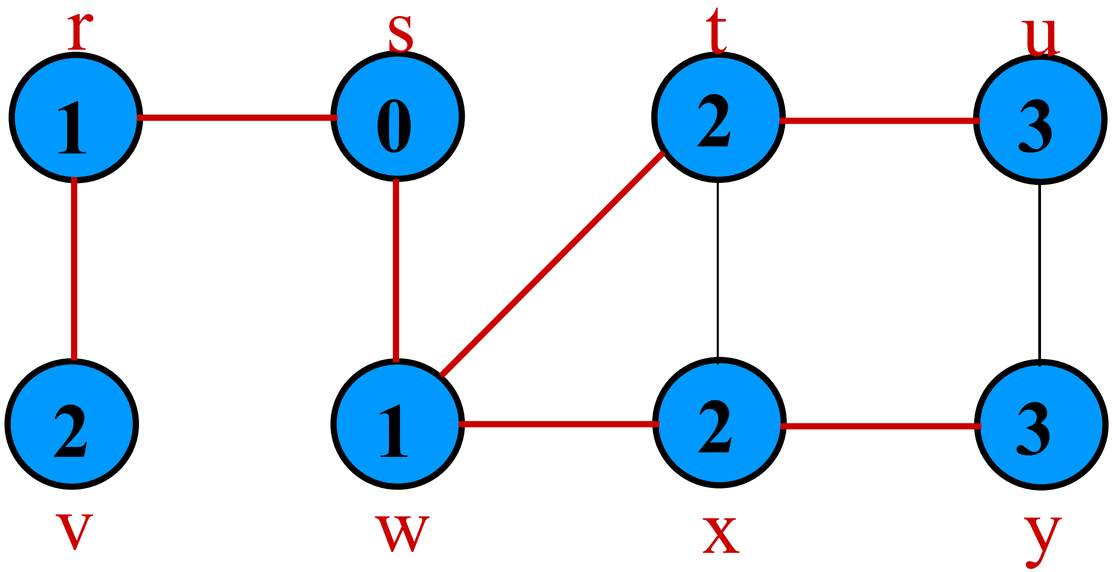
Example (BFS)



Order of visit:
s, w, r, t, x, v, u, y



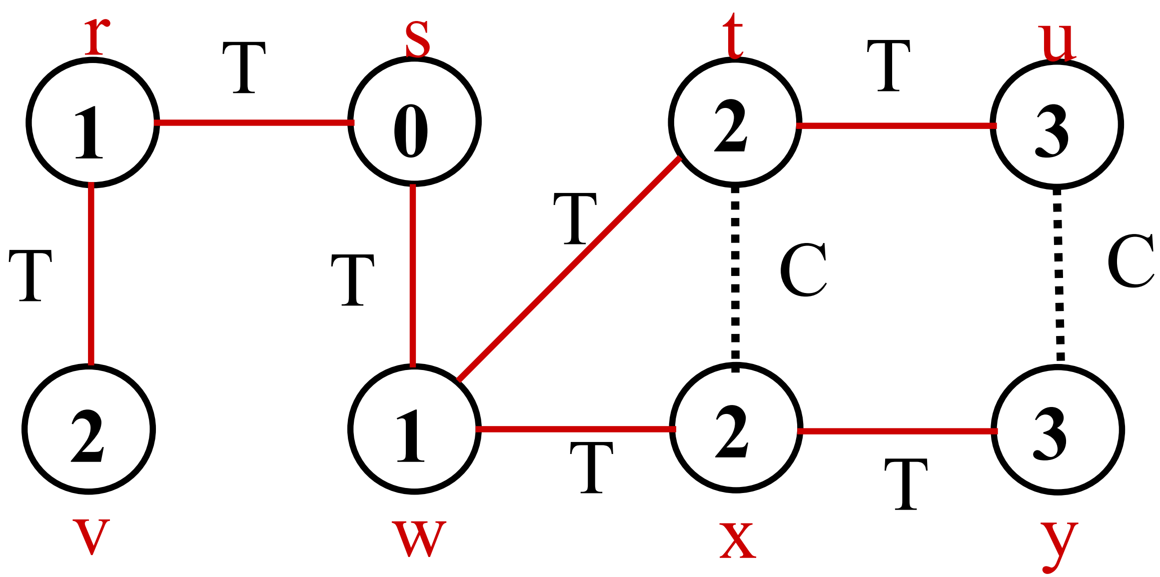
Example (BFS)



Q: ∅

Order of visit:
s, w, r, t, x, v, u, y

Example (BFS)



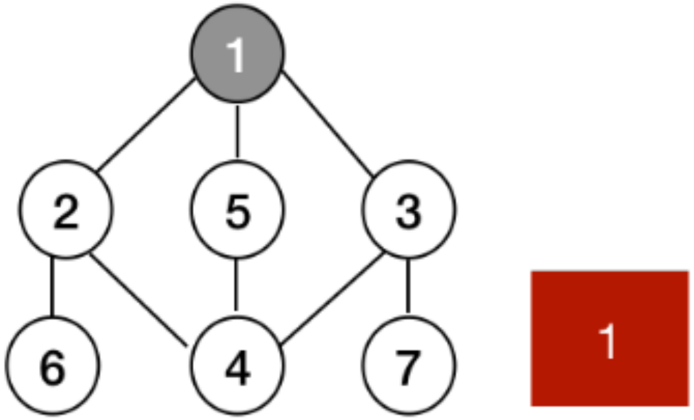
BF Tree

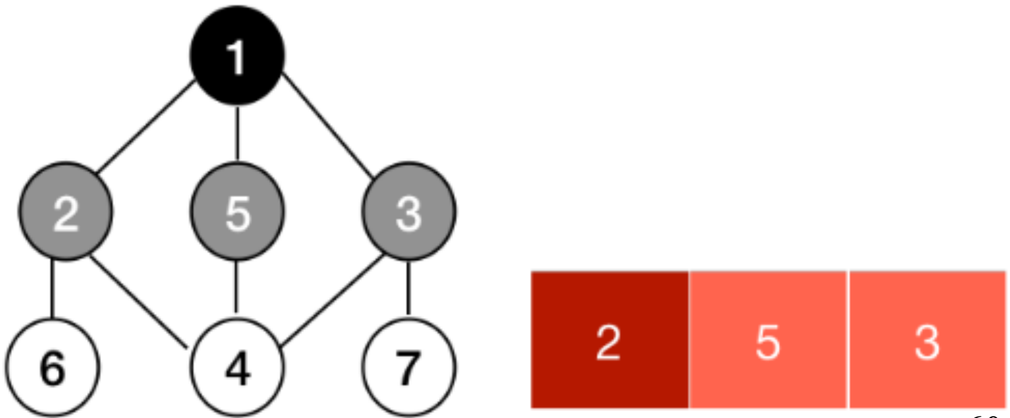
BFS traversal:
s, w, r, t, x, v, u, y

Analysis of BFS

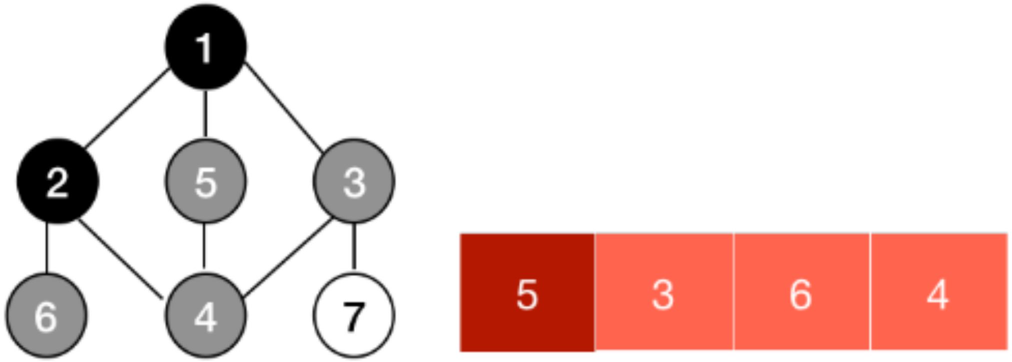
- Initialization takes $O(V)$.
- Traversal Loop
  - After initialization, each vertex is enqueued and dequeued at most once, and each operation takes $O(1)$. So, total time for queuing is $O(V)$.
  - The adjacency list of each vertex is scanned at most once. The sum of lengths of all adjacency lists is $\Theta(E)$.
- Summing up over all vertices => total running time of BFS is $O(V+E)$, linear in the size of the adjacency list representation of graph.

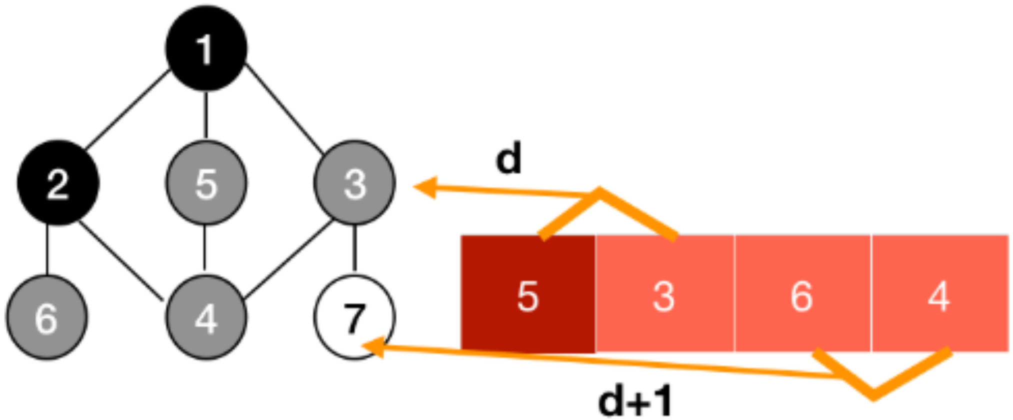
BFS- Example 2



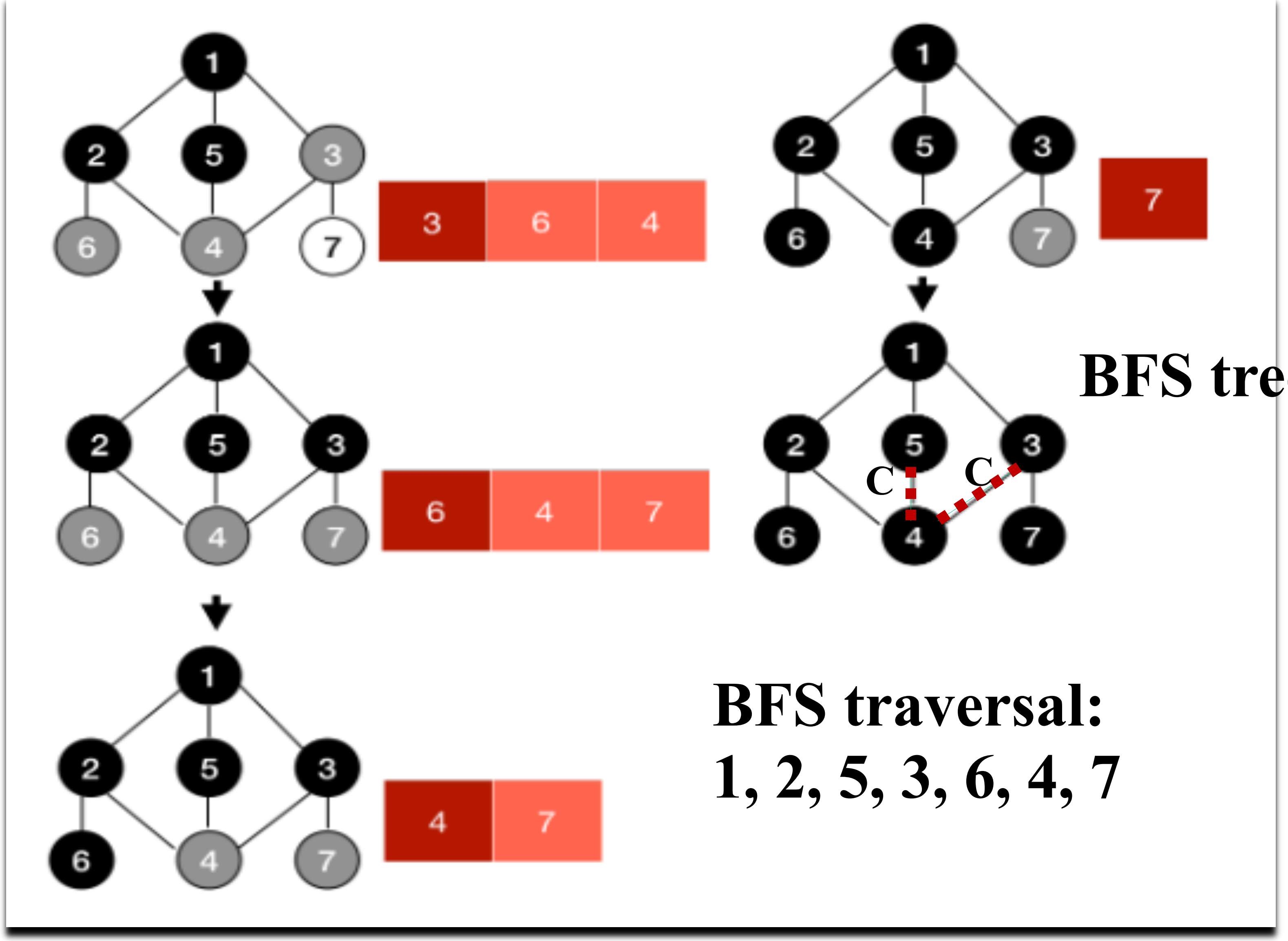


BFS- Example 2





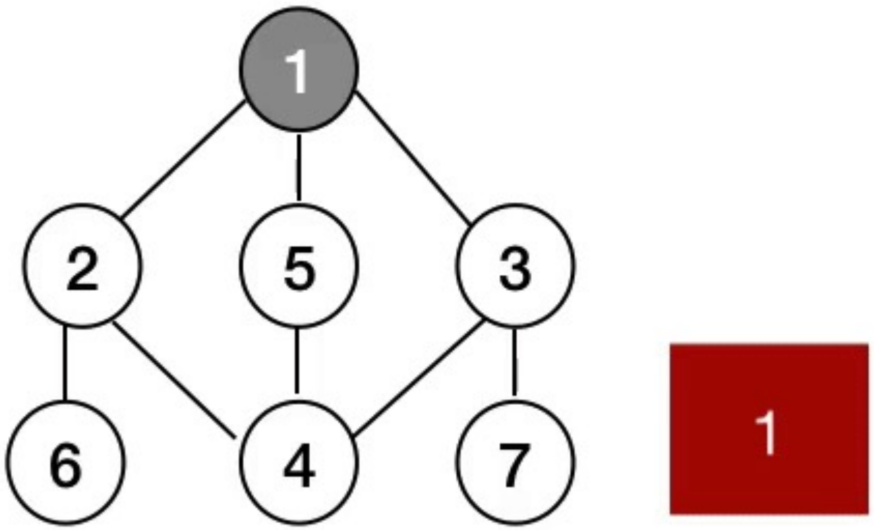
BFS- Example 2



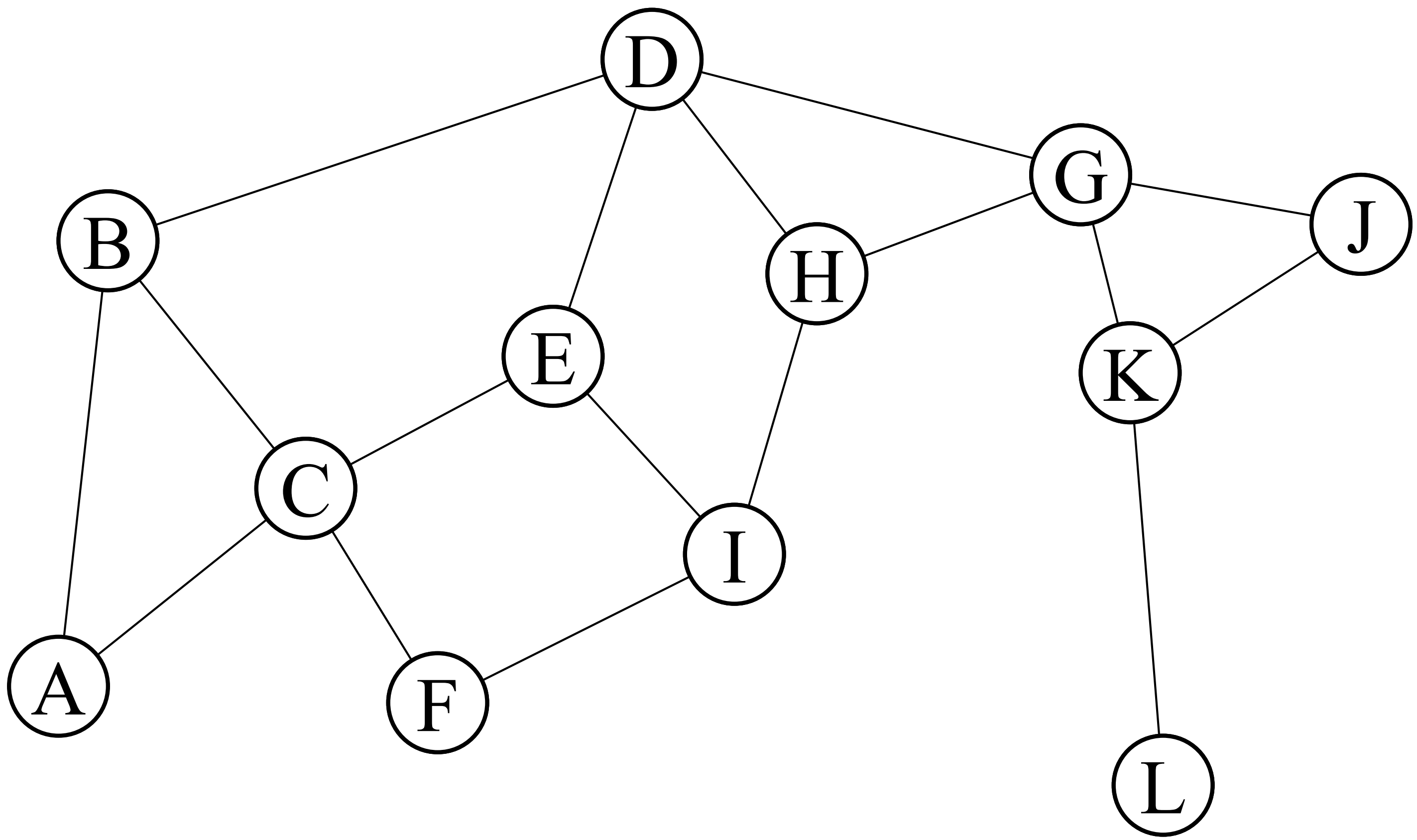
BFS tree

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

BFS- Example 2



Exercise- Find the BFS tree and the traversal



Depth-first Search (DFS)

- Explore edges out of the most recently discovered vertex v.
- When all edges of v have been explored, backtrack to explore other edges leaving the vertex from which v was discovered (its *predecessor*).
- "Search as deep as possible first."
- Continue until all vertices reachable from the original source are discovered.
- If any undiscovered vertices remain, then one of them is chosen as a new source and search is repeated from that source.

Depth-first Search

- **Input:** $G = (V, E)$, directed or undirected. No source vertex given!
- **Output:**
  - 2 **timestamps** on each vertex. Integers between 1 and 2|V|.
    - $d[v]$ = ***discovery time*** (v turns from white to gray)
    - $f[v]$ = ***finishing time*** (v turns from gray to black)
  - $\pi[v]$: predecessor of $v = u$, such that v was discovered during the scan of u's adjacency list.
- Uses the same coloring scheme for vertices as BFS.

Pseudo-code

DFS(G)

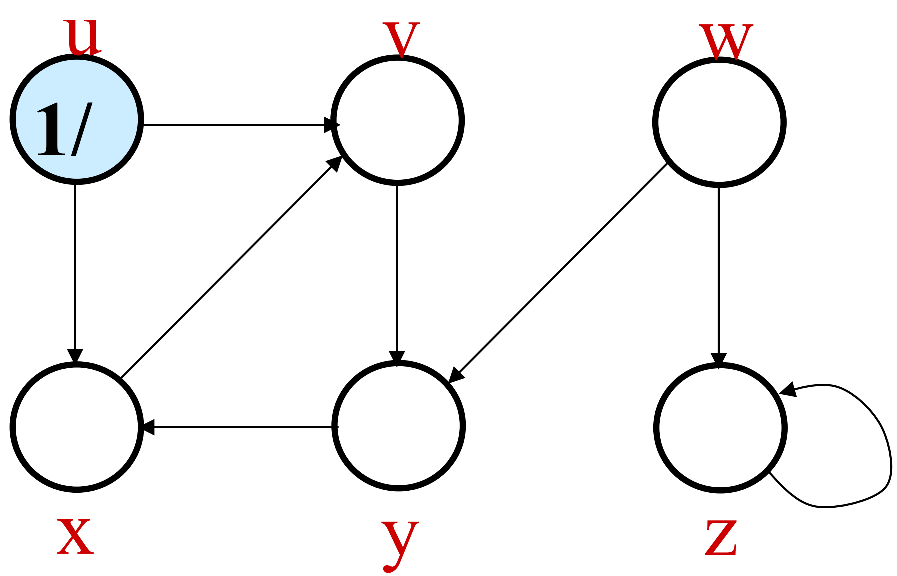
1. **for** each vertex $u \in V[G]$
2. **do** $color[u] \leftarrow$ white
3. $\pi[u] \leftarrow$ NIL
4. $time \leftarrow 0$
5. **for** each vertex $u \in V[G]$
6. **do if** $color[u] =$ white
7. **then** DFS-Visit(u)

Uses a global timestamp ***time***.

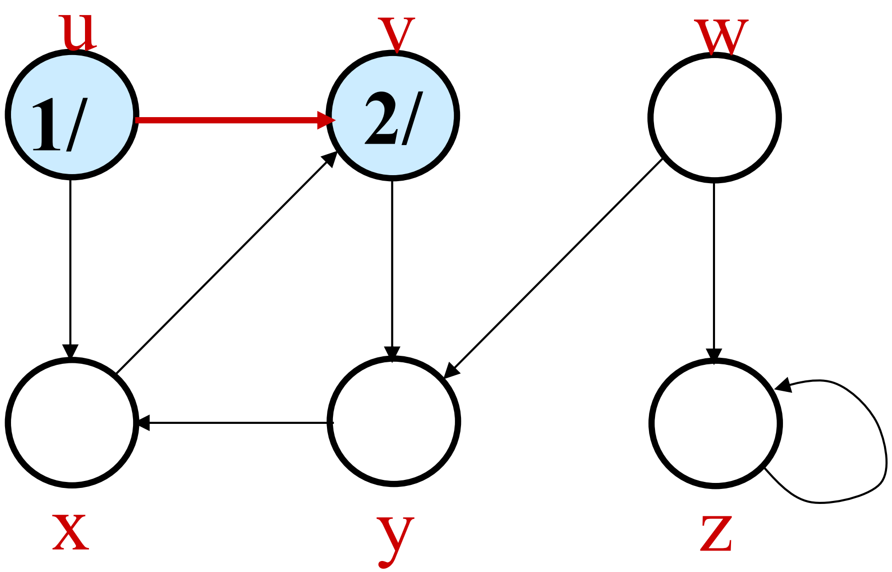
DFS-Visit(u)

1. $color[u] \leftarrow$ GRAY ∇ White vertex u has been discovered
2. $time \leftarrow time + 1$
3. $d[u] \leftarrow time$
4. **for** each $v \in Adj[u]$
5. **do if** $color[v] =$ WHITE
6. **then** $\pi[v] \leftarrow u$
7. DFS-Visit(v)
8. $color[u] \leftarrow$ BLACK ∇ Blacken u; it is finished.
9. $f[u] \leftarrow time \leftarrow time + 1$

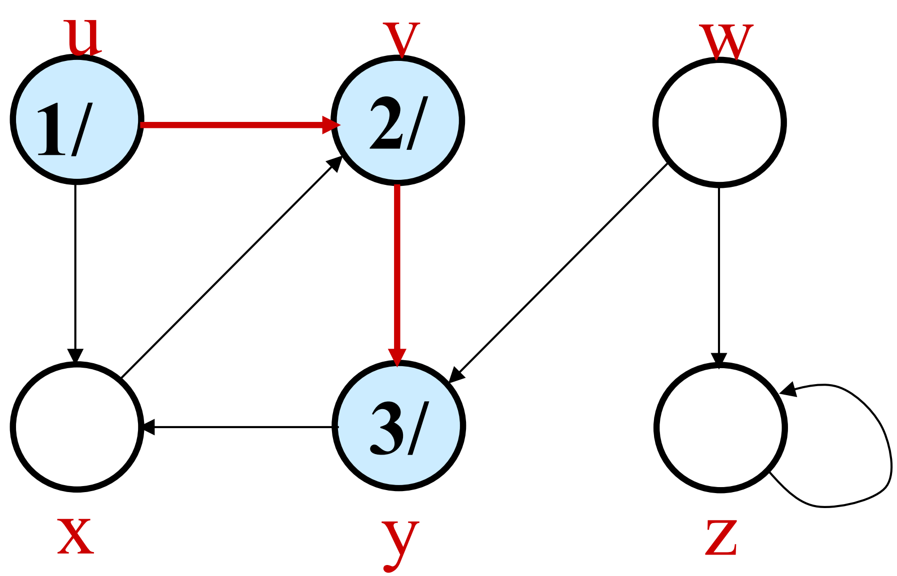
Example (DFS)



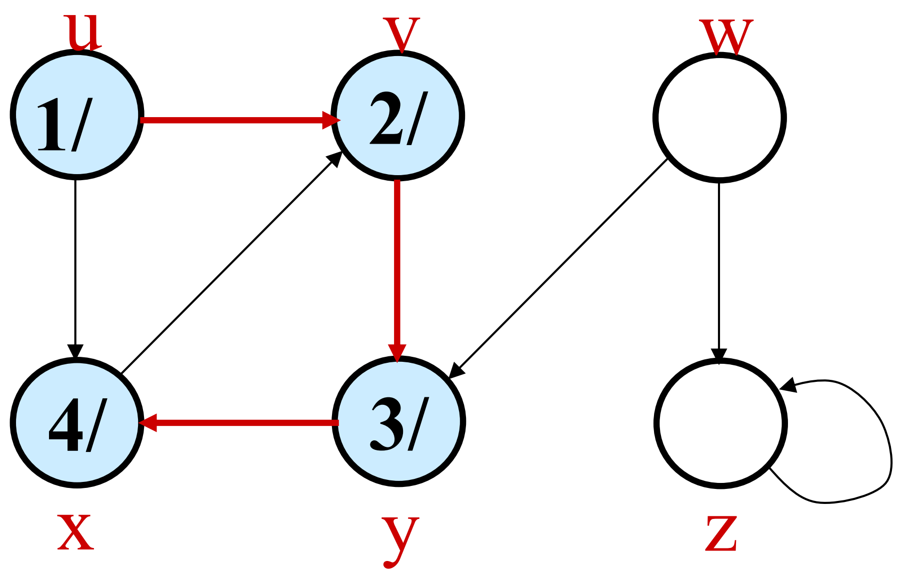
Example (DFS)



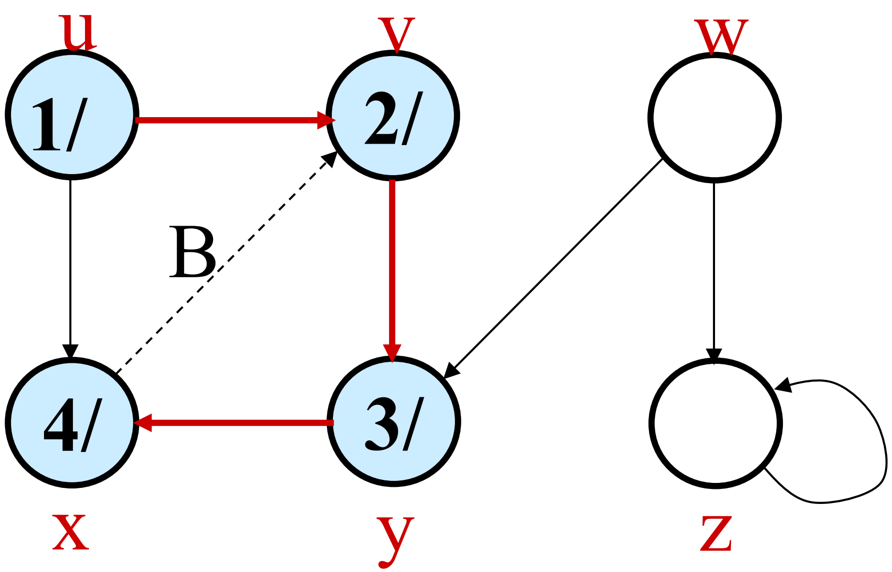
Example (DFS)



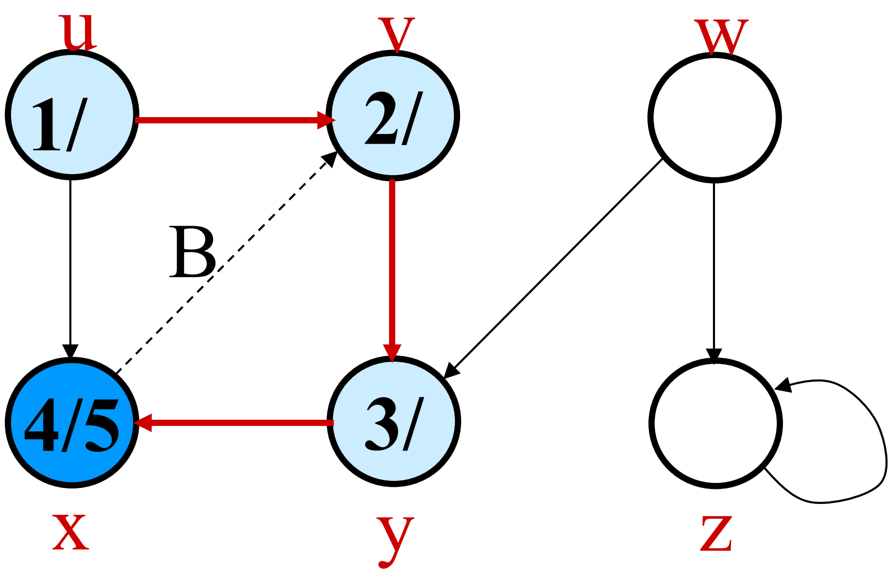
Example (DFS)



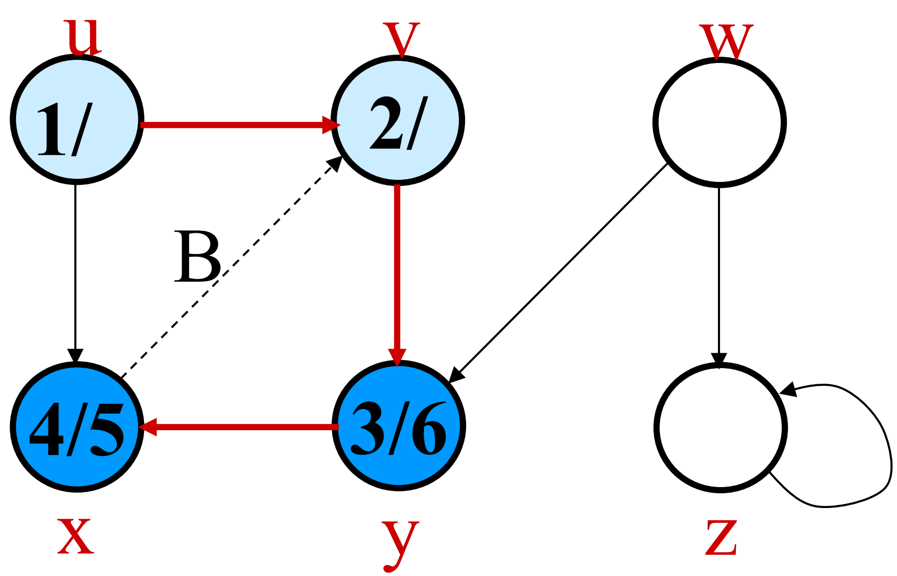
Example (DFS)



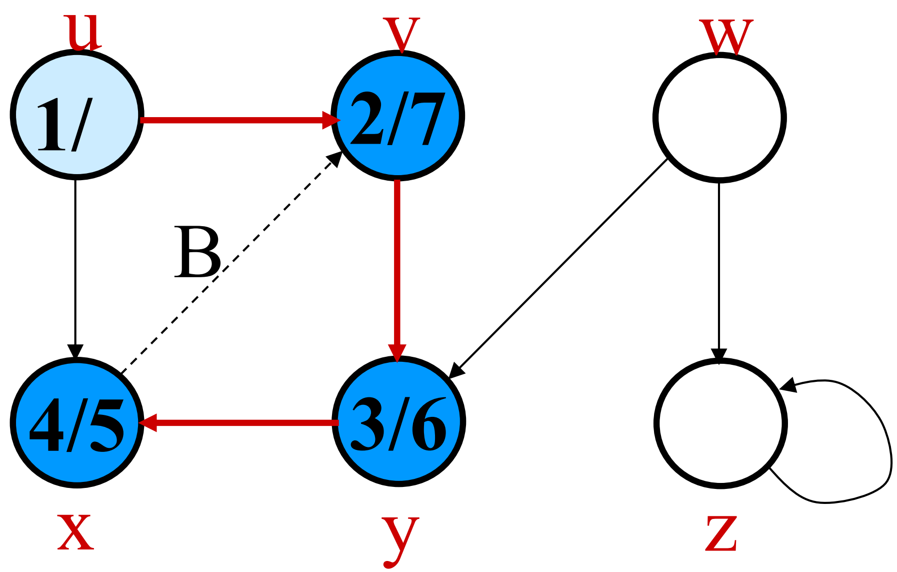
Example (DFS)



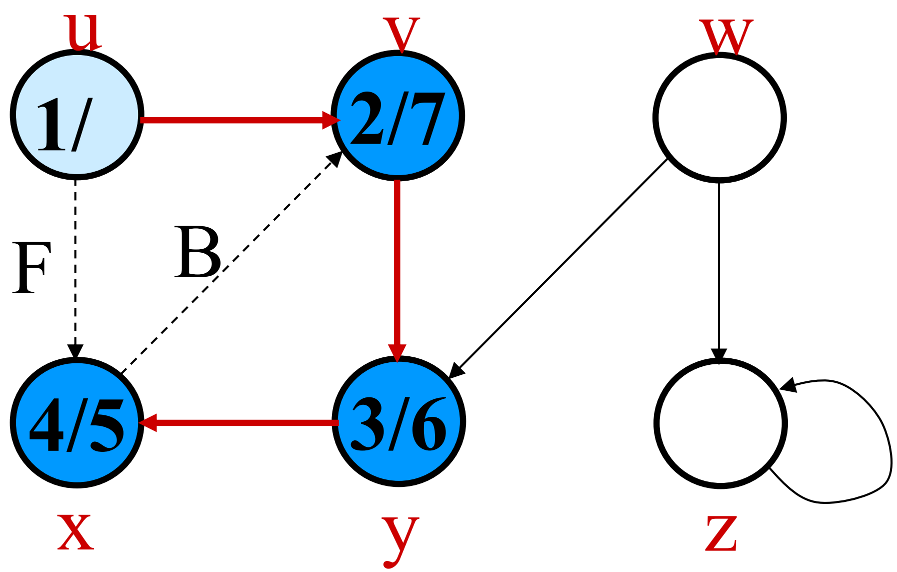
Example (DFS)



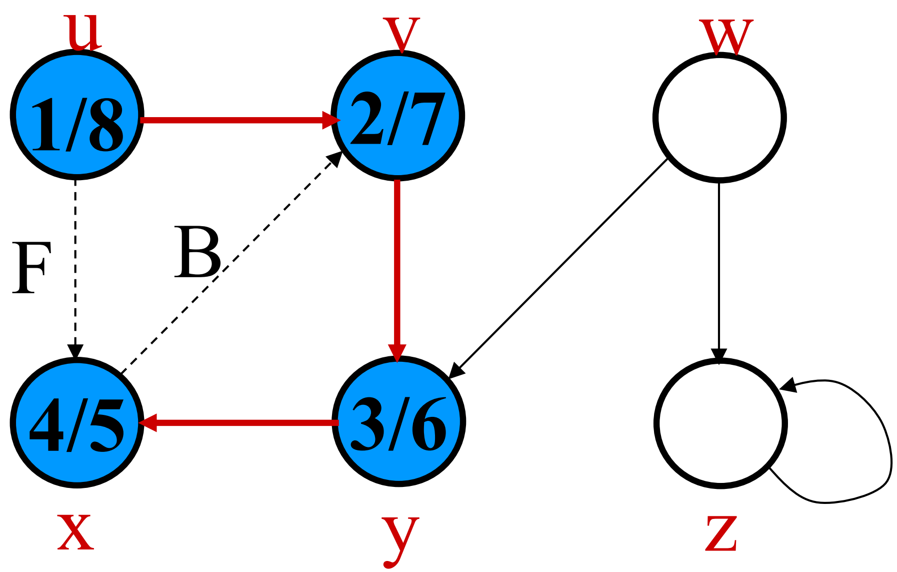
Example (DFS)



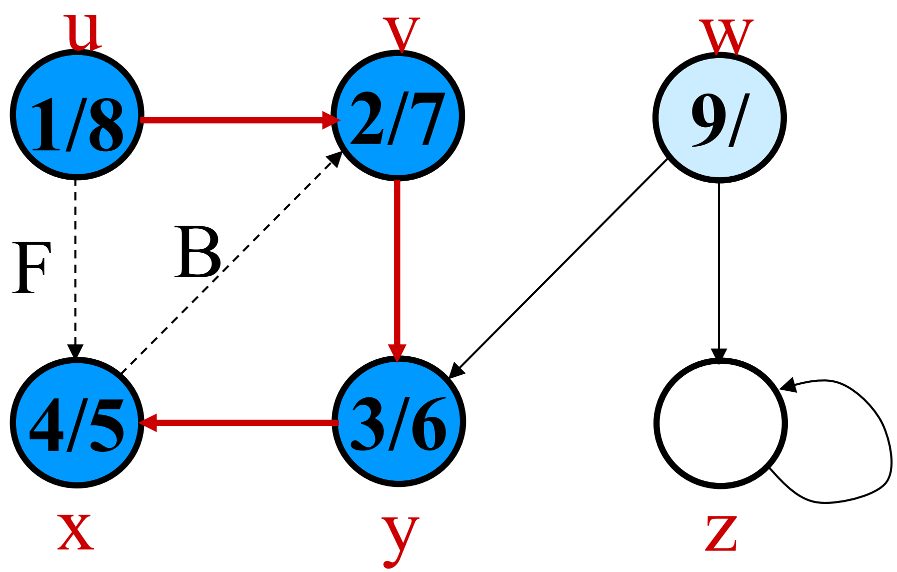
Example (DFS)



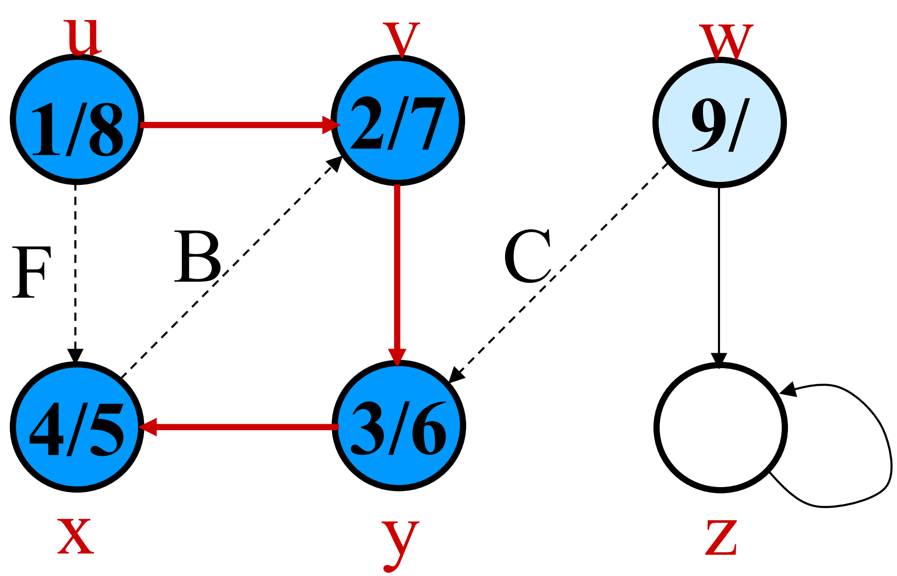
Example (DFS)



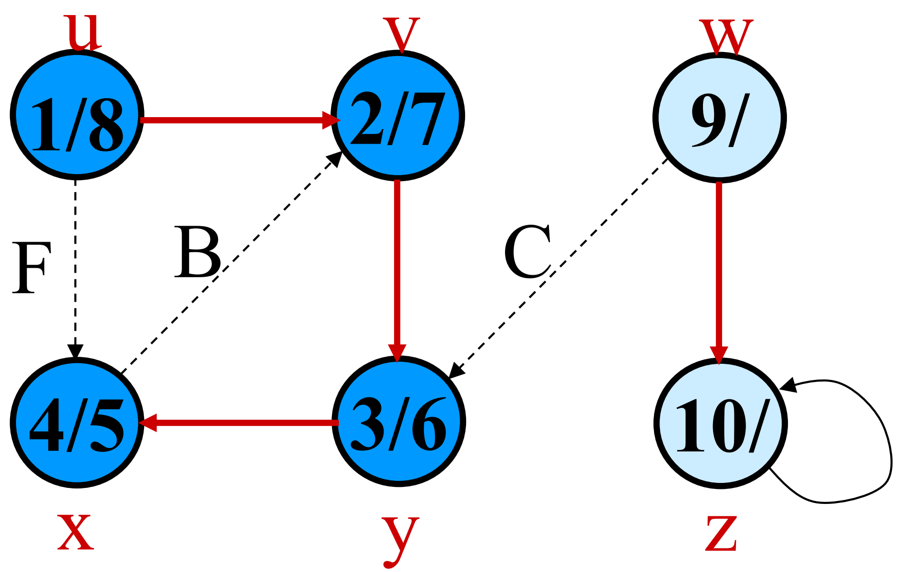
Example (DFS)



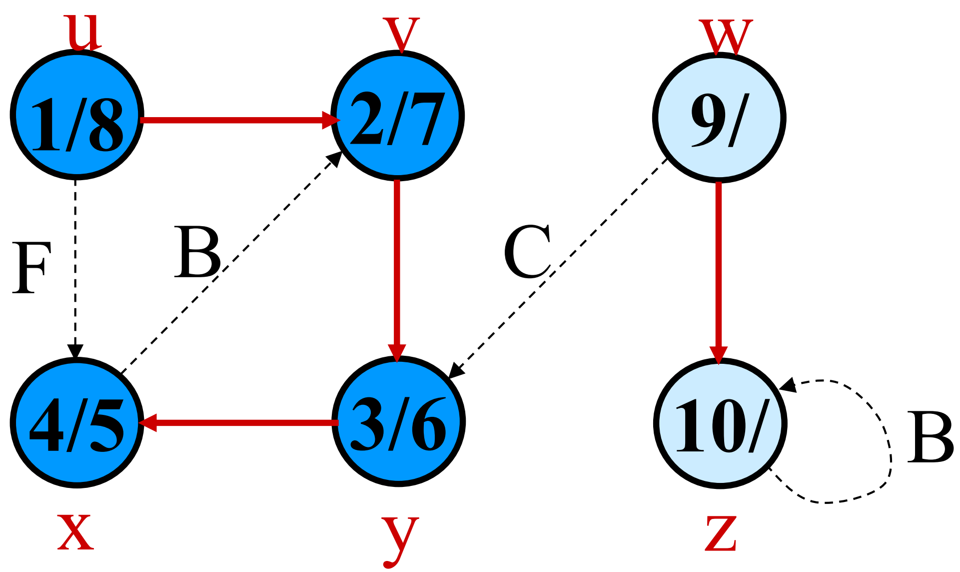
Example (DFS)



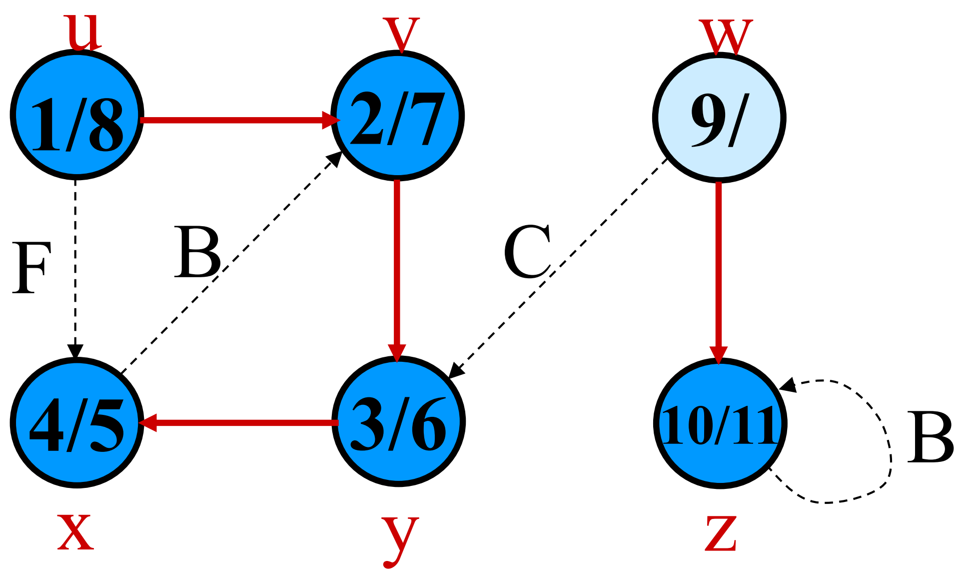
Example (DFS)



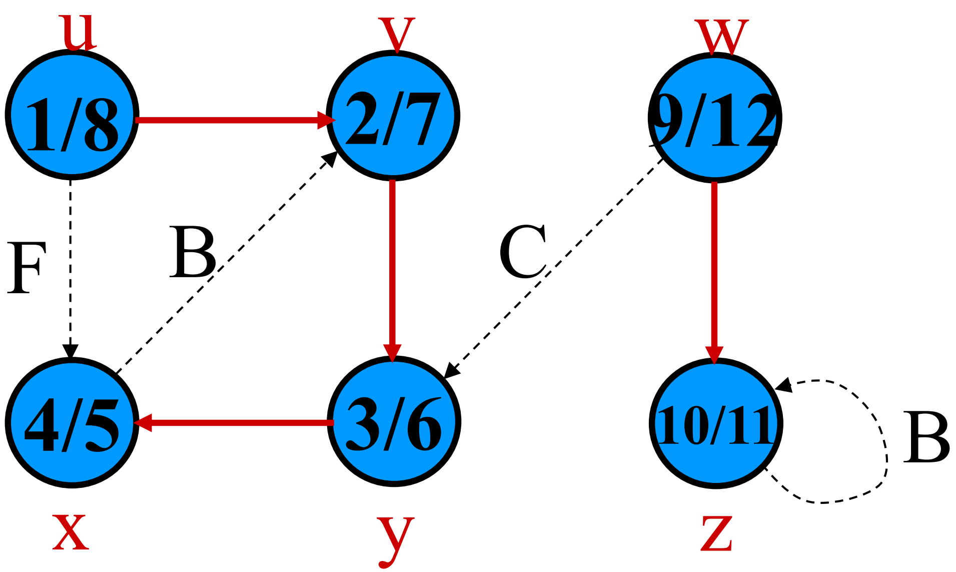
Example (DFS)



Example (DFS)



Example (DFS)



DFS traversal:

u, v, y, x, w, z

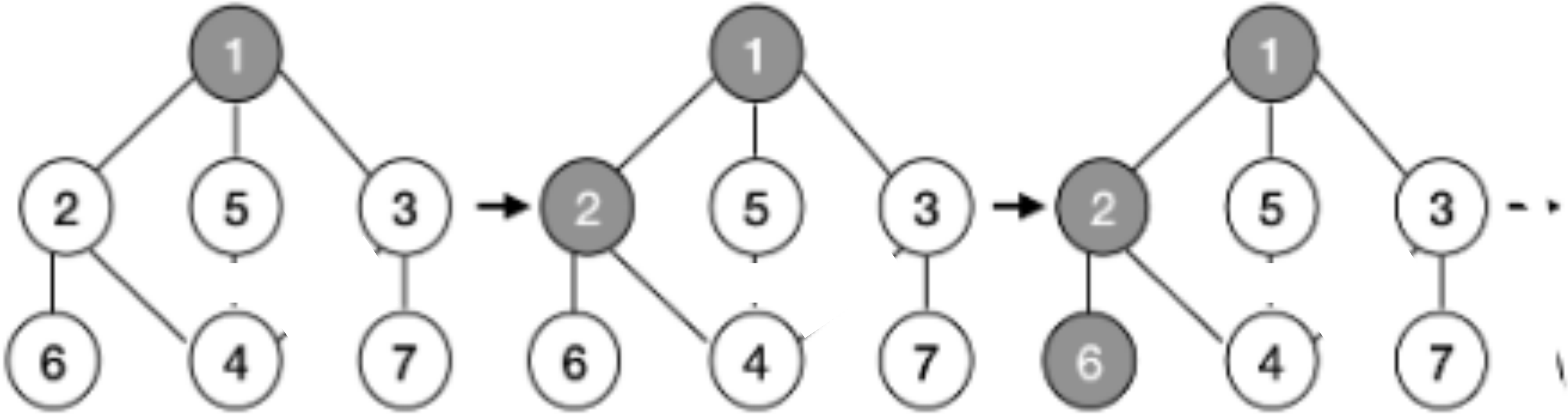
Analysis of DFS

- Loops on lines 1-2 & 5-7 take $\Theta(V)$ time, excluding time to execute DFS-Visit.

- DFS-Visit is called once for each white vertex $v \in V$ when it's painted gray the first time. Lines 3-6 of DFS-Visit is executed $|\text{Adj}[v]|$ times. The total cost of executing DFS-Visit is $\sum_{v \in V} |\text{Adj}[v]| = \Theta(E)$

- Total running time of DFS is $\Theta(V+E)$.

DFS – Example 2

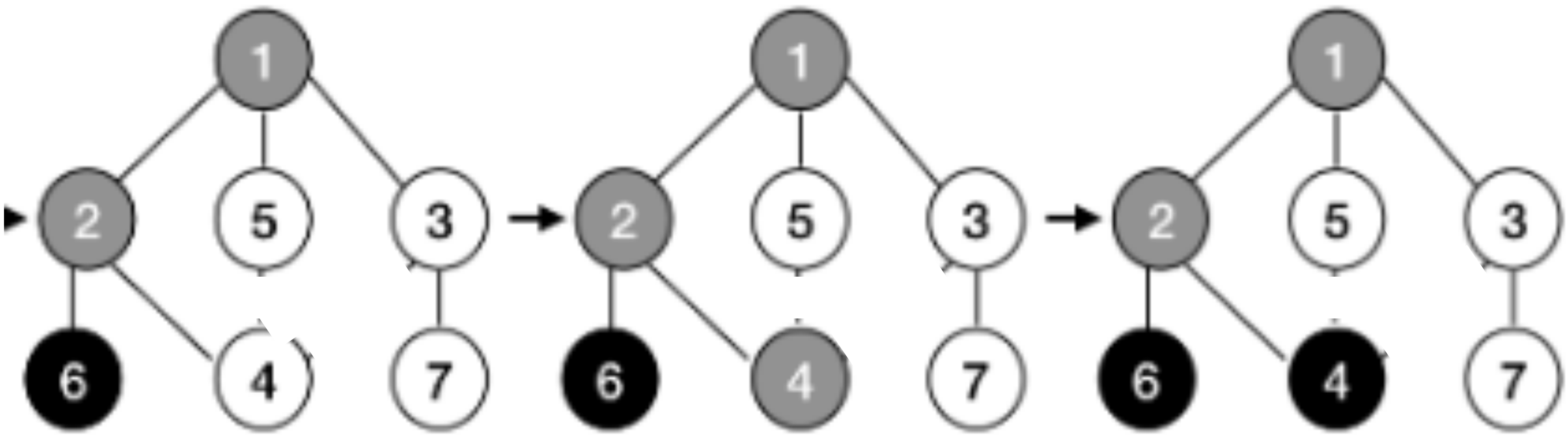


DFS traversal: 1

DFS traversal: 1, 2,

DFS traversal: 1, 2, 6

DFS – Example 2

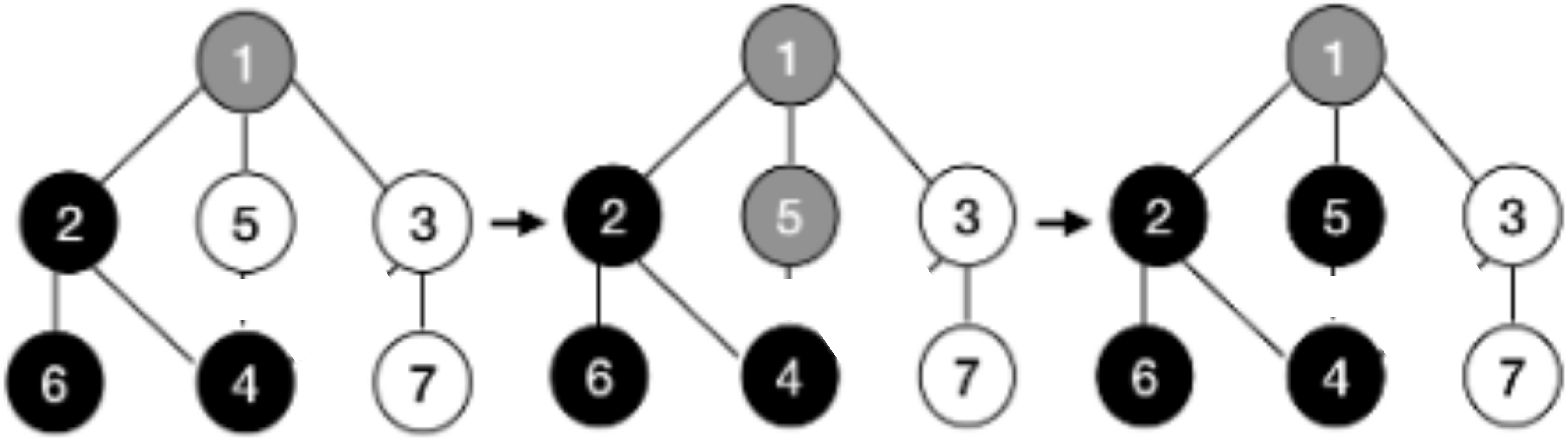


DFS traversal:
1, 2, 6

DFS traversal:
1, 2, 6, 4

DFS traversal:
1, 2, 6, 4

DFS – Example 2

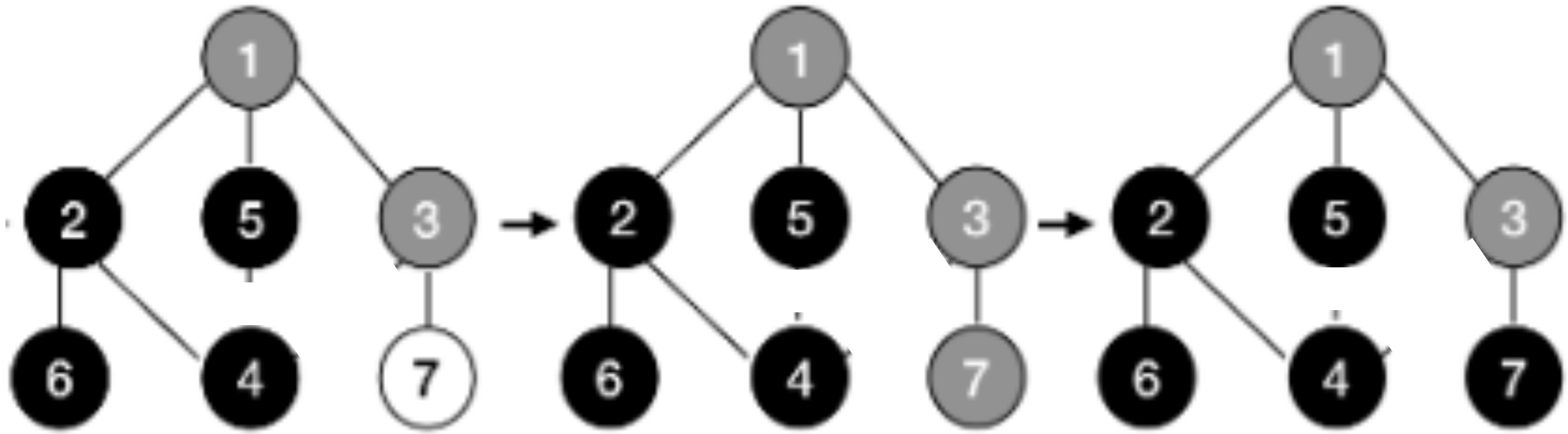


DFS traversal:
1, 2, 6, 4

DFS traversal:
1, 2, 6, 4, 5

DFS traversal:
1, 2, 6, 4, 5

DFS – Example 2

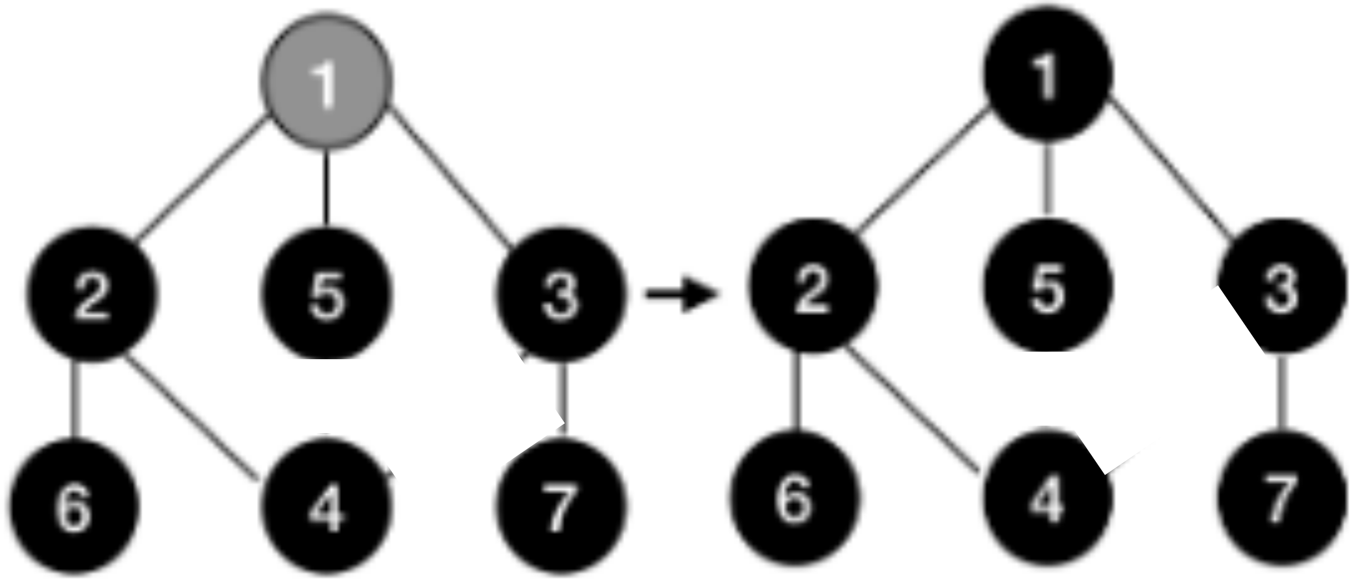


DFS traversal:
1, 2, 6, 4, 5, 3

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

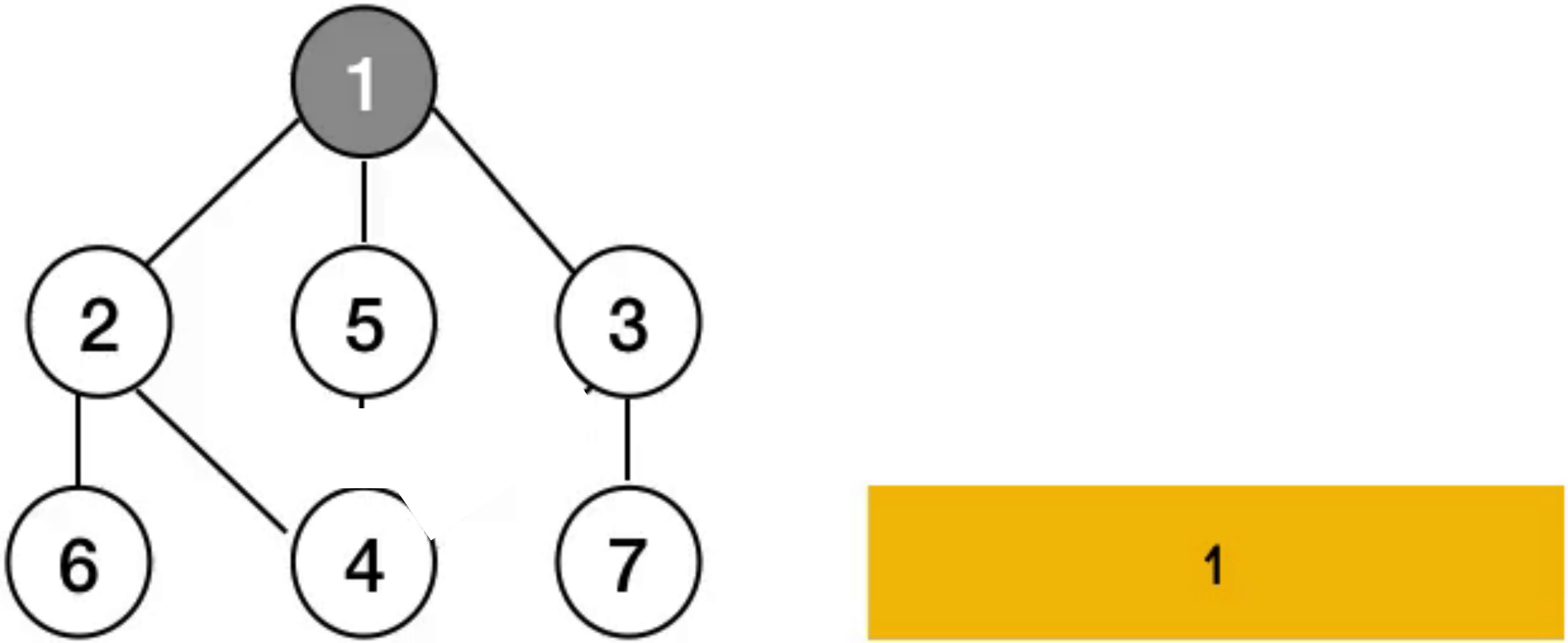
DFS traversal:
1, 2, 6, 4, 5, 3, 7

DFS – Example 2



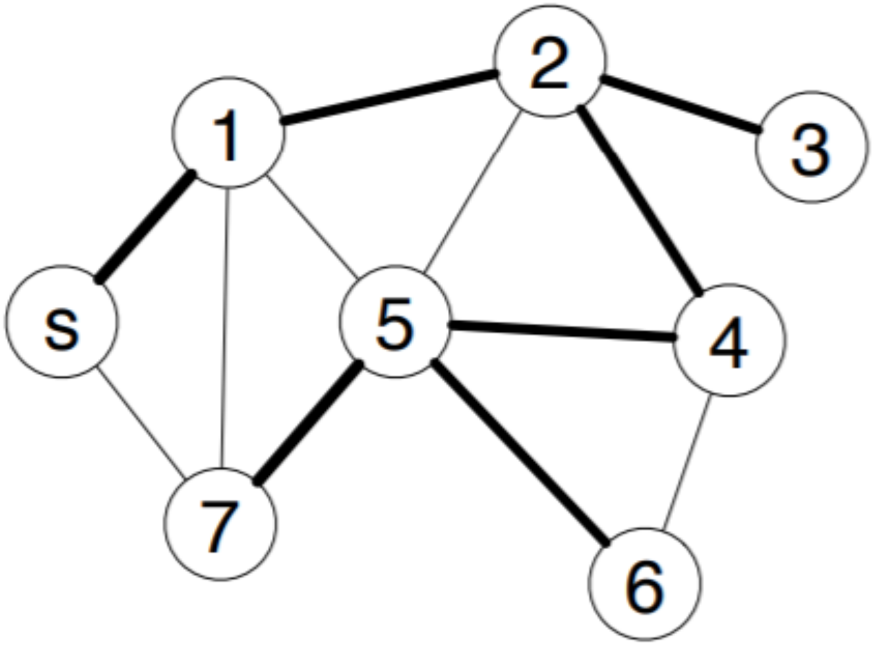
DFS traversal:
1, 2, 6, 4, 5, 3, 7

DFS – Example 2



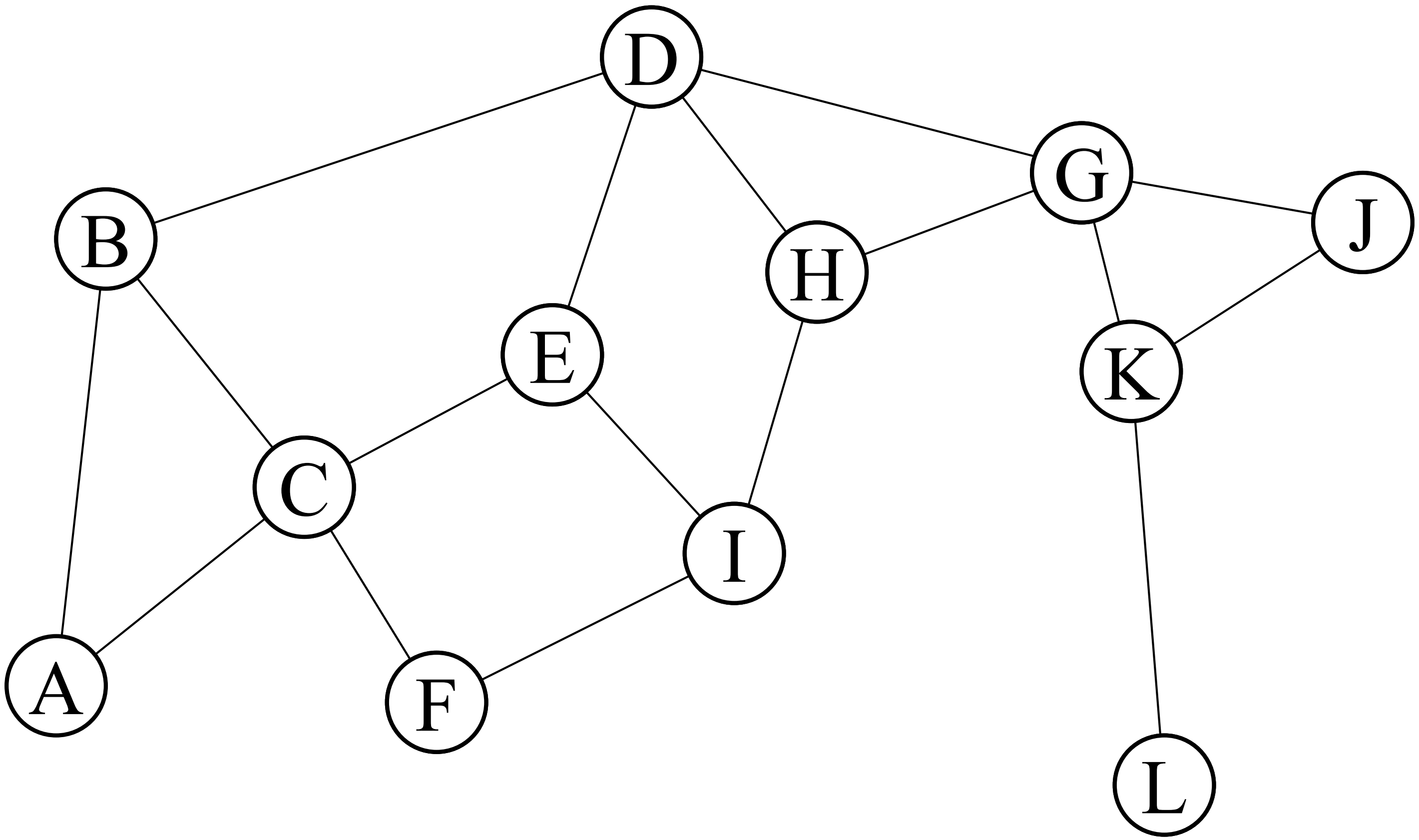
DFS traversal:
1, 2, 6, 4, 5, 3, 7

Example 3 – Identify the edges



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

Exercise – Perform DFS



Directed Graph DFS trees

- Things are a bit more complicated
  - Forward edges (ancestor to descendant)
  - Cross Edges
    - Not from an ancestor to a descendant
- Parenthesis structure still holds
  - The discovery and finishing times of nodes in a DFS tree have a parenthesis structure
- Can create a forest of DFS trees

Classification of Edges

- **Tree edge:** in the depth-first forest. Found by exploring (u, v).
- **Back edge:** (u, v), where u is a descendant of v (in the depth-first tree).
- **Forward edge:** (u, v), where v is a descendant of u, but not a tree edge.
- **Cross edge:** any other edge. Can go between vertices in same depth-first tree or in different depth-first trees.

In DFS of an undirected graph, we get only tree and back edges. No forward or cross edges.

Exercises

Let G be a graph whose vertices are the integers 1 through 8, and let the adjacent vertices of each vertex be given by the table below:

vertex	adjacent vertices
1	(2, 3, 4)
2	(1, 3, 4)
3	(1, 2, 4)
4	(1, 2, 3, 6)
5	(6, 7, 8)
6	(4, 5, 7)
7	(5, 6, 8)
8	(5, 7)

Assume that, in a traversal of G, the adjacent vertices of a given vertex are returned in the same order as they are listed in the above table.

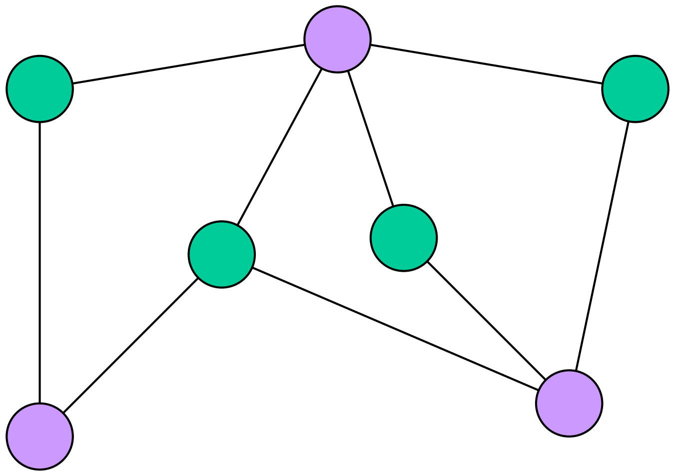
a. Draw G.
b. Order the vertices as they are visited in a DFS traversal starting at vertex 1.
c. Order the vertices as they are visited in a BFS traversal starting at vertex 1.

Graphs

- Motivation and Terminology
- Representations
- Traversals
- Three Problems

Bipartite Graph?

Give an efficient algorithm to determine if a graph is bipartite. (Bipartite means that the graph can be colored with 2 colors such that all edges connect vertices of different colors.)



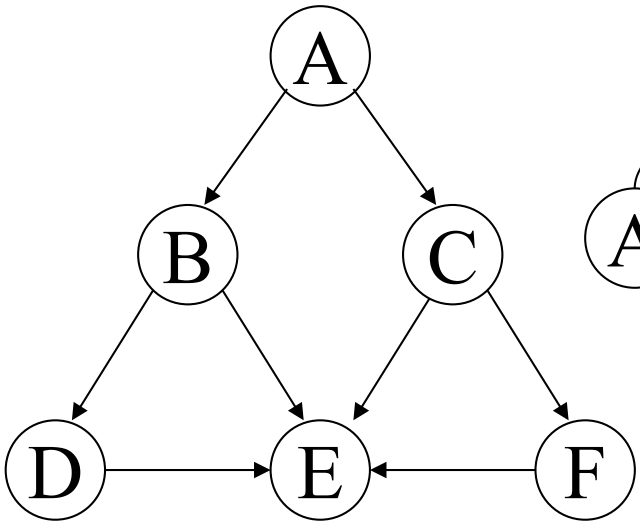
Cycle?

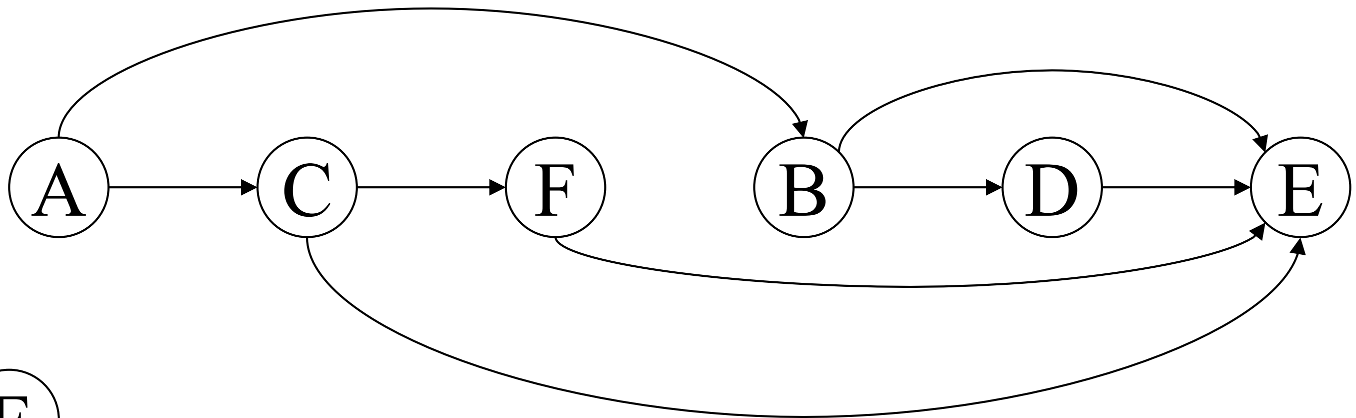
Give an O(V) algorithm to determine if an undirected graph contains a cycle.

Note: Full traversal by BFS and DFS both take O(V+E) time.

Topological Sorting?

•A *directed acyclic graph* (DAG) is a directed graph with no directed cycles.
•A *topological sort* is an ordering of nodes where all edges go from left to right.
•How can BFS or DFS help us topologically sort a directed graph (or determine the graph is not a DAG)?





DAGs and Topological Ordering

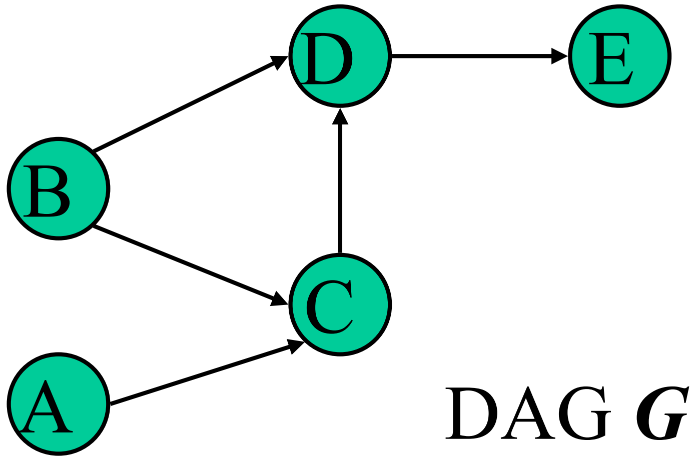
- A directed acyclic graph (DAG) is a digraph that has no directed cycles
- A topological ordering of a digraph is a numbering

$$v_1, \ldots, v_n$$

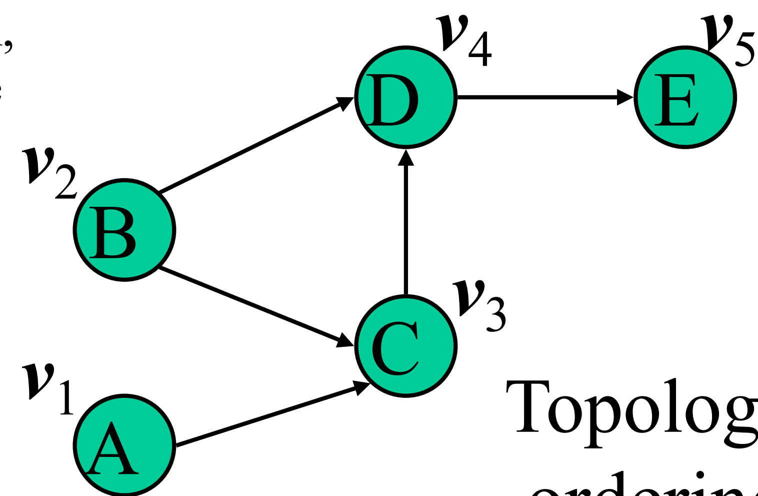
of the vertices such that for every edge (v_i, v_j), we have $i < j$
- Example: in a task scheduling digraph, a topological ordering a task sequence that satisfies the precedence constraints

Theorem

A digraph admits a topological ordering if and only if it is a DAG

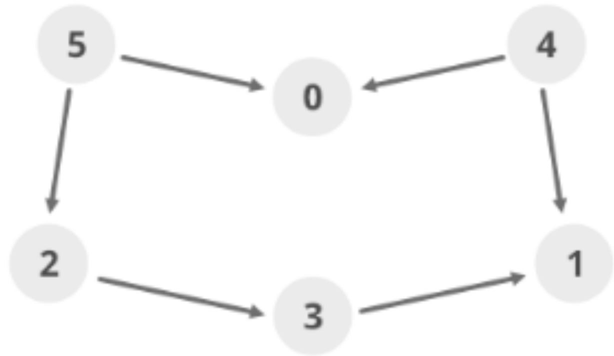


DAG G



Topological ordering of G

Example 1 – Topological Ordering

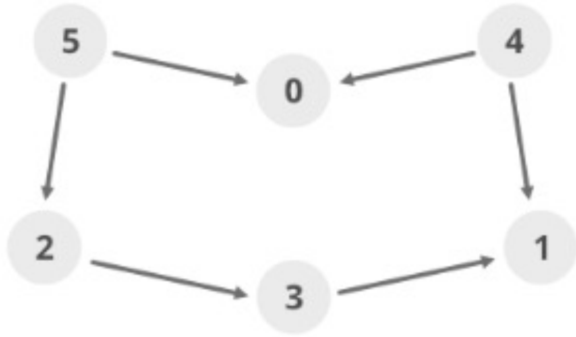


Adja cent list (G)
0 →
1 →
2 → 3
3 → 1
4 → 0, 1
5 → 2, 0

	0	1	2	3	4	5
visited	false	false	false	false	false	false

Stack(empty)

Example 1



Step 1: Topological Sort(0), visited[0] = true

↓

List is empty. No more recursion call.

Stack | 0 |

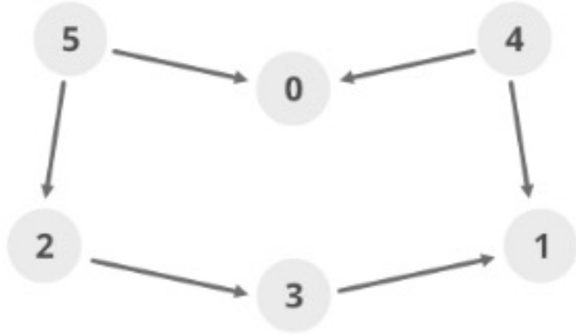
Step 2: Topological Sort(1), visited[1] = true

↓

List is empty. No more recursion call.

Stack | 0 | 1 |

Example 1



Step 3: Topological Sort(2), visited[2] = true

↓

Topological Sort(3), visited[3] = true

↓

'1' is already visited. No more recurrsion call

Stack	0	1	3	2

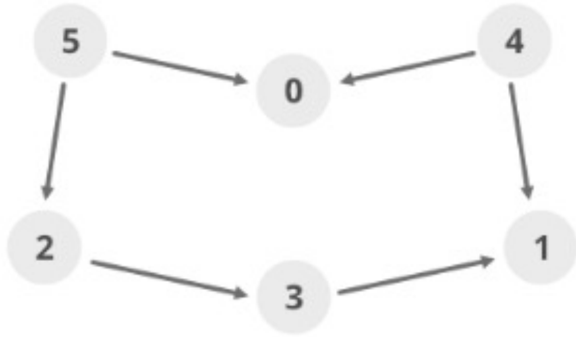
Step 4: Topological Sort(4), visited[4] = true

↓

'0' , '1' are already visited. No more recurrsion call

Stack	0	1	3	2	4

Example 1



Step 5: Topological Sort(5), visited[5] = true

'2' , '0' are already visited. No more recurrsion call

Stack						
	0	1	3	2	4	5

Step 6: Print all elements of stack from top to bottom

Algorithm

performTopologicalSorting(Graph)

Input: The given directed acyclic graph.
Output: Sequence of nodes.

```
Begin
   initially mark all nodes as unvisited
   for all nodes v of the graph, do
      if v is not visited, then
         topoSort(i, visited, stack)
   done
   pop and print all elements from the stack
End.
```

Algorithm

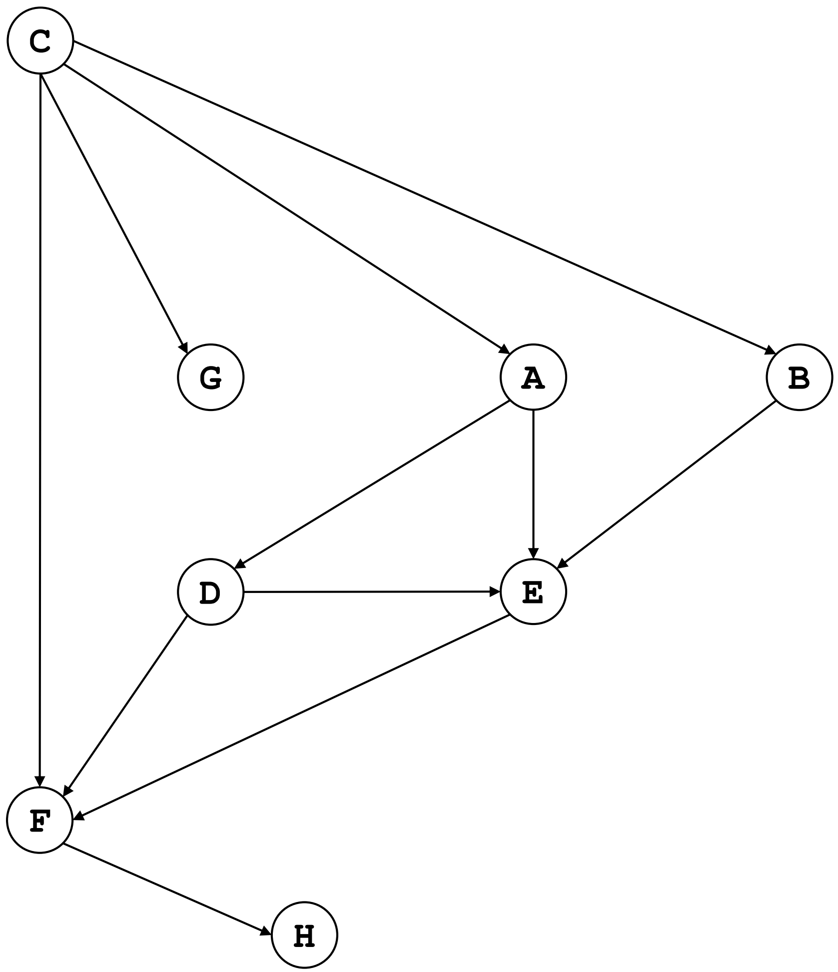
topoSort(u, visited, stack)

Input: The start vertex u, An array to keep track of which node is visited or not. A stack to store nodes.
Output: Sorting the vertices in topological sequence in the stack.

```
Begin
   mark u as visited
   for all vertices v which is adjacent with u, do
      if v is not visited, then
         topoSort(c, visited, stack)
   done

   push u into a stack
End
```

Example 2 - Topological Sort



Adjacency List	
A	D, E
B	E
C	A, B, F, G
D	E, F
E	F
F	H
G	-
H	-

Example 2 - Topological Sort

Step 1

TS(A), V[A]=T => TS(D), V[D]=T => TS(E), V[E]=T => TD(F), V[F]=T => TS(H), V[H]= T , no more recursive calls

Stack						
Stack	H	F	E			

Visited	
A	TRUE
B	
C	
D	TRUE
E	TRUE
F	TRUE
G	
H	TRUE

Step 1 continued

TS(F) , but F is already visited, recursive call ends

Stack	H	F	E	D	

Step 1 continued

TS(E), but E already visited, recursive call ends

Stack	H	F	E	D	A	

Example 2 - Topological Sort

Step 2

TS(B), V[B]=T => TS€, but E already visited, recursive call ends

Stack	H	F	E	D	A	B			

Visited	
A	TRUE
B	TRUE
C	
D	TRUE
E	TRUE
F	TRUE
G	
H	TRUE

Example 2 - Topological Sort

Step 3

TS(C), V[C]=T => TS(A) already visited, TS(B) already visited, TS(F) already visited, TS(G), V[G]=T, recursive call ends

Stack	H	F	E	D	A	B	G	C	

Visited	
A	TRUE
B	TRUE
C	TRUE
D	TRUE
E	TRUE
F	TRUE
G	TRUE
H	TRUE

Example 2 - Topological Sort

Step 4

TS(D) already visited, no recursive calls

Step 5

TS(E) , already visited, no recursive calls

Step 6

TS(F), already visited, no recursive calls

Step 7

TS(G), already visited, no recursive calls

Step H

TS(H), already visited, no recursive calls

Example 2 - Topological Sort

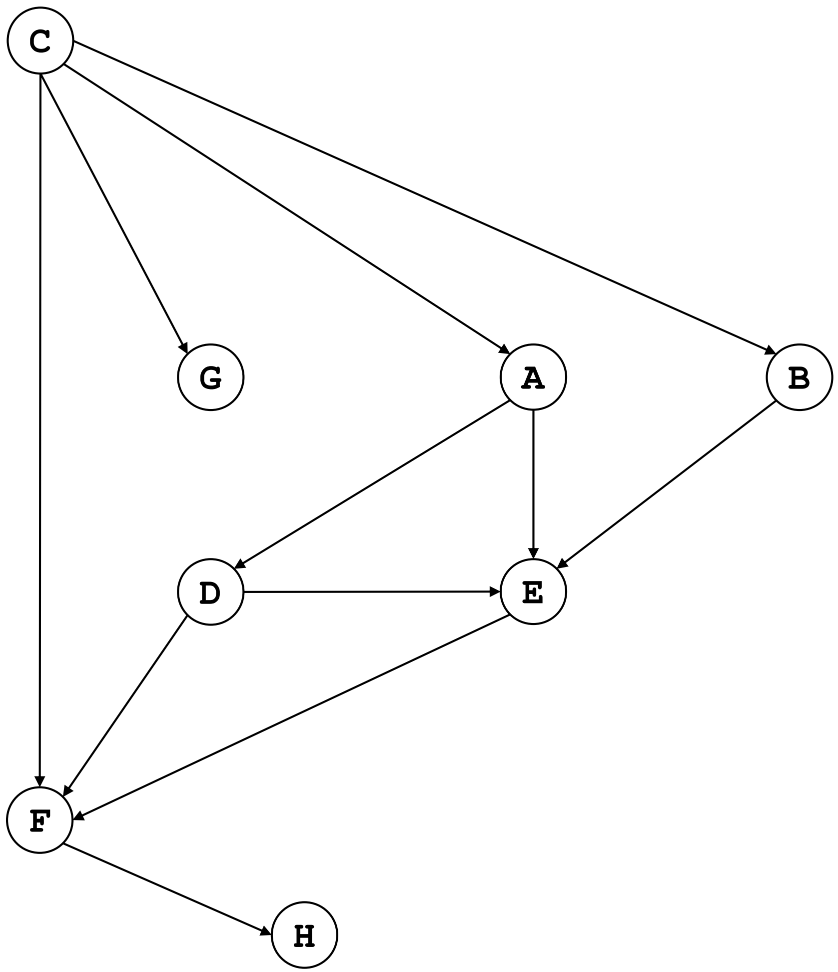
Read the stack in reverse

Stack	H	F	E	D	A	B	G	C	

Topological Order

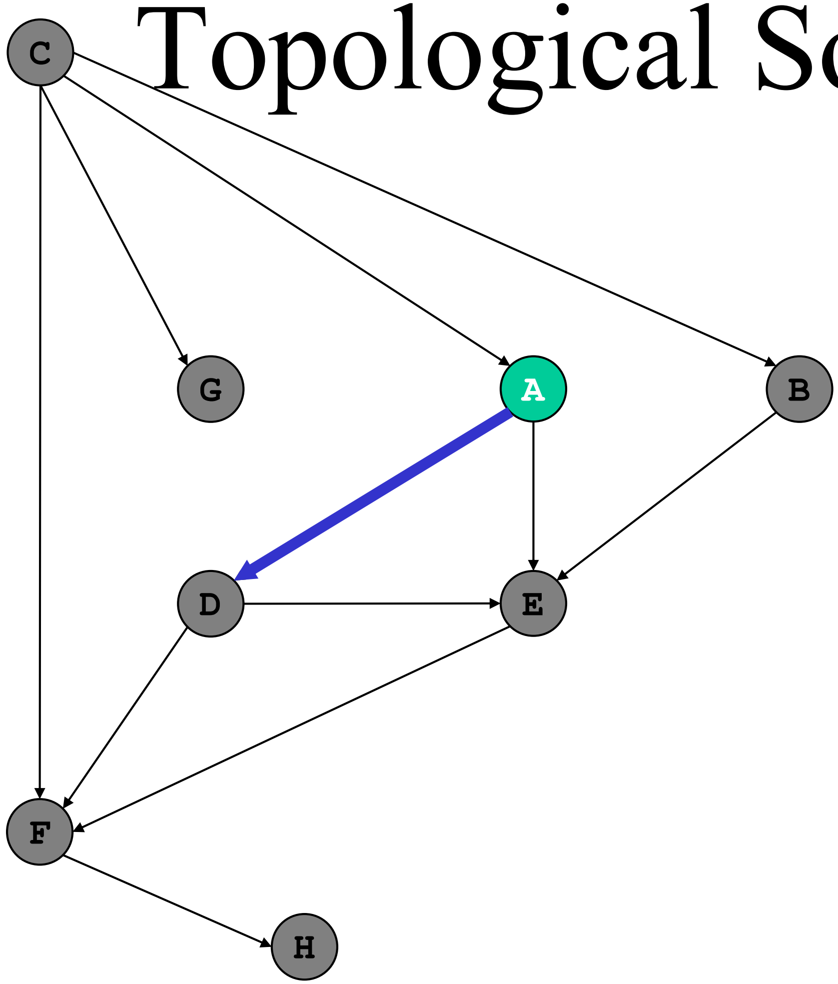
C, G, B, A, D, E, F, H

Example 2 - Topological Sort (Alternate Way)

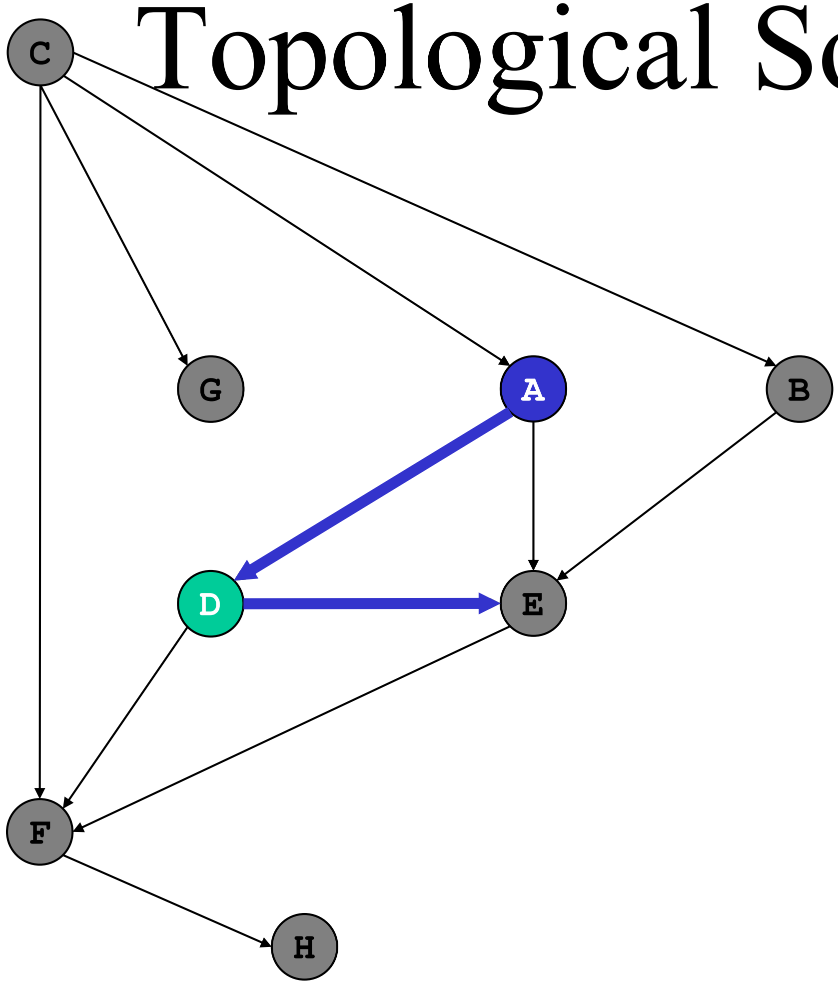


Topological Sort: DFS

`dfs(A)`

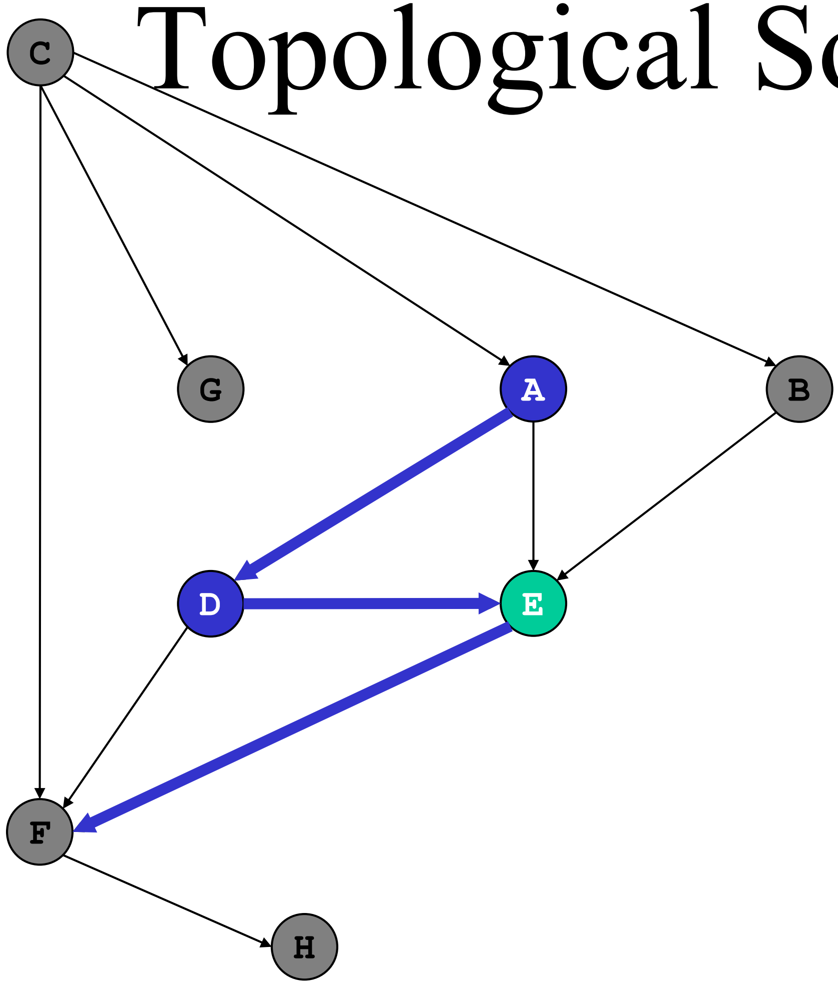


Topological Sort: DFS



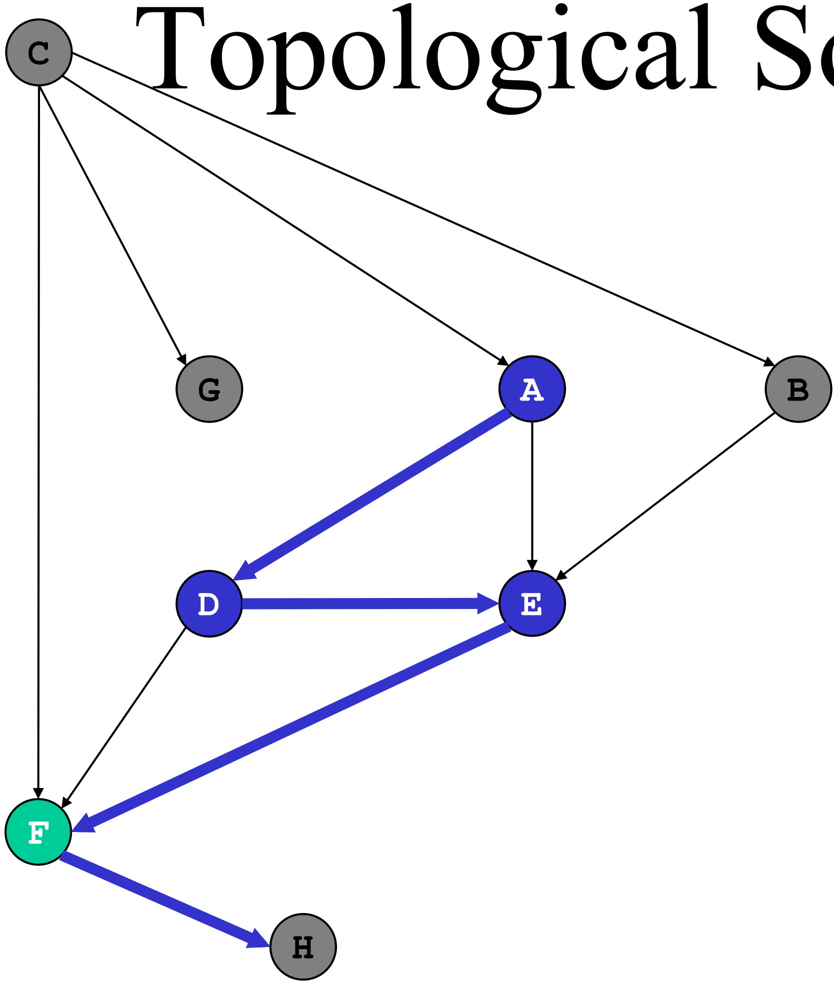

```
dfs(A)
dfs(D)
```

Topological Sort: DFS



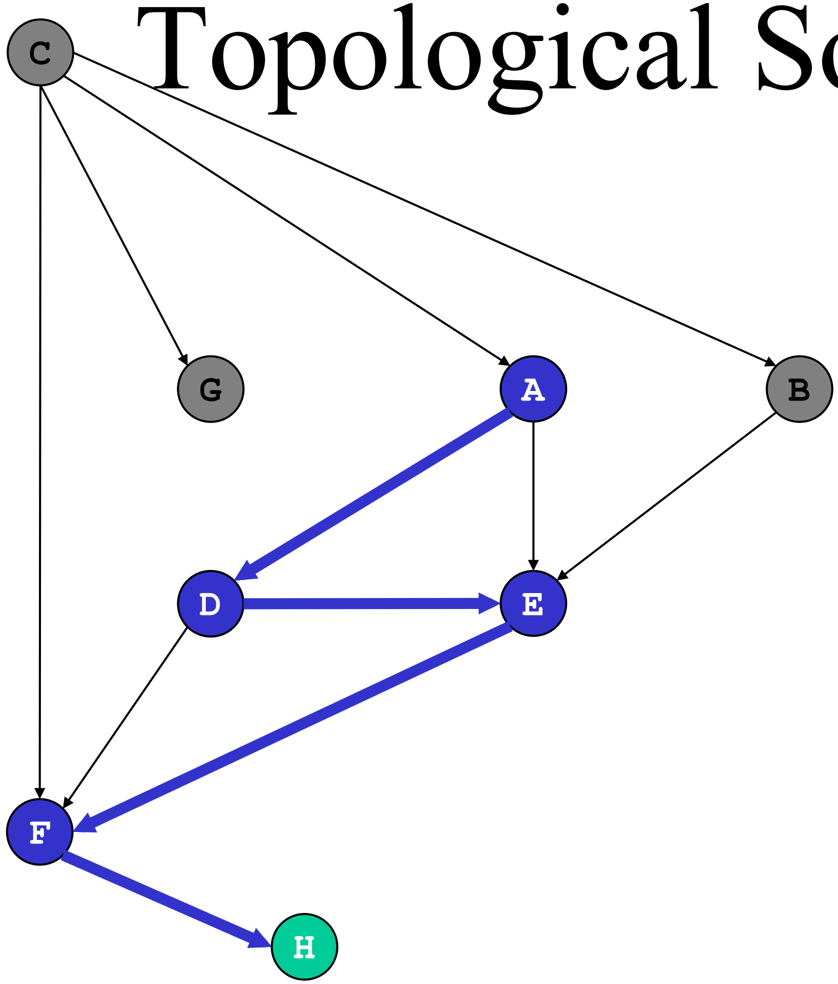

```
dfs(A)
dfs(D)
dfs(E)
```

Topological Sort: DFS



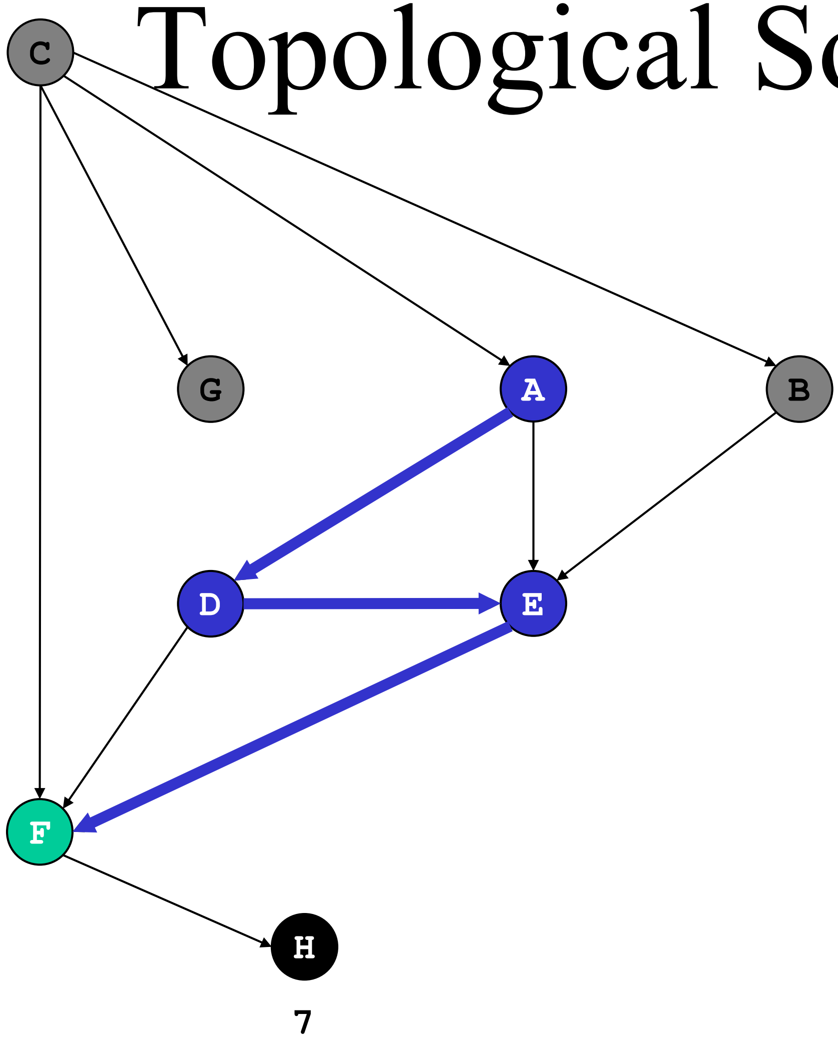

```
dfs(A)
dfs(D)
dfs(E)
dfs(F)
```

Topological Sort: DFS



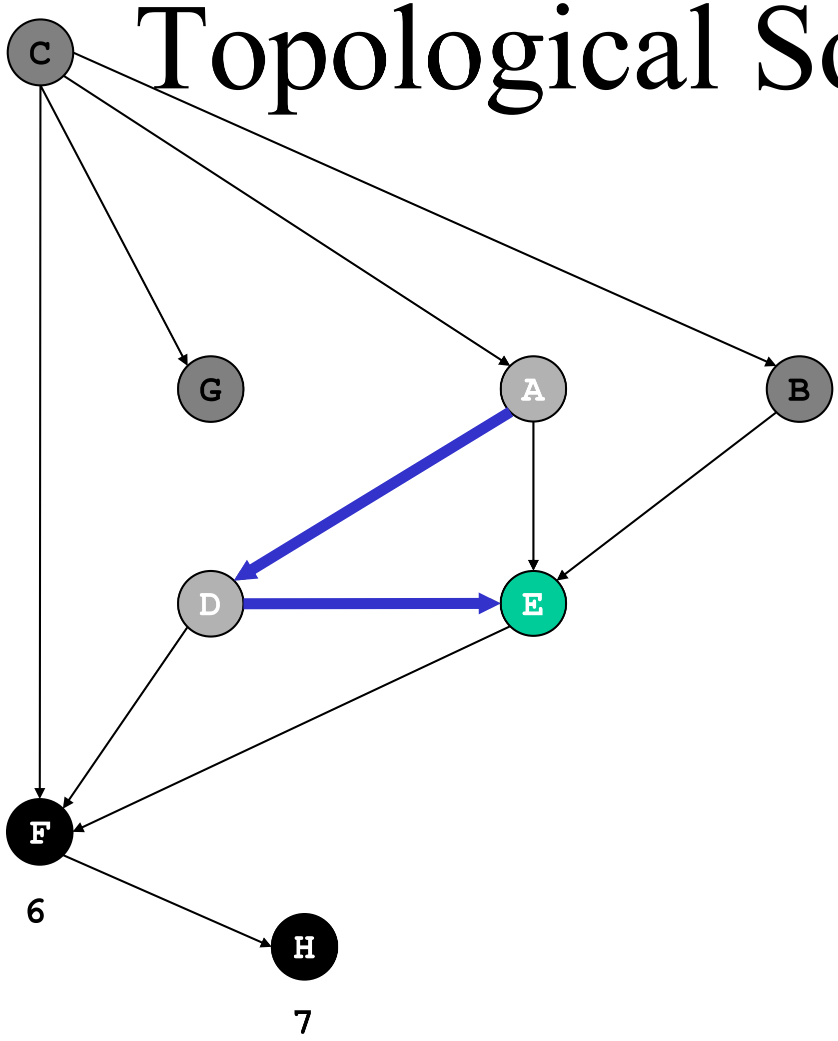

```
dfs(A)
dfs(D)
dfs(E)
dfs(F)
dfs(H)
```

Topological Sort: DFS




```
dfs(A)
dfs(D)
dfs(E)
dfs(F)
```

Topological Sort: DFS

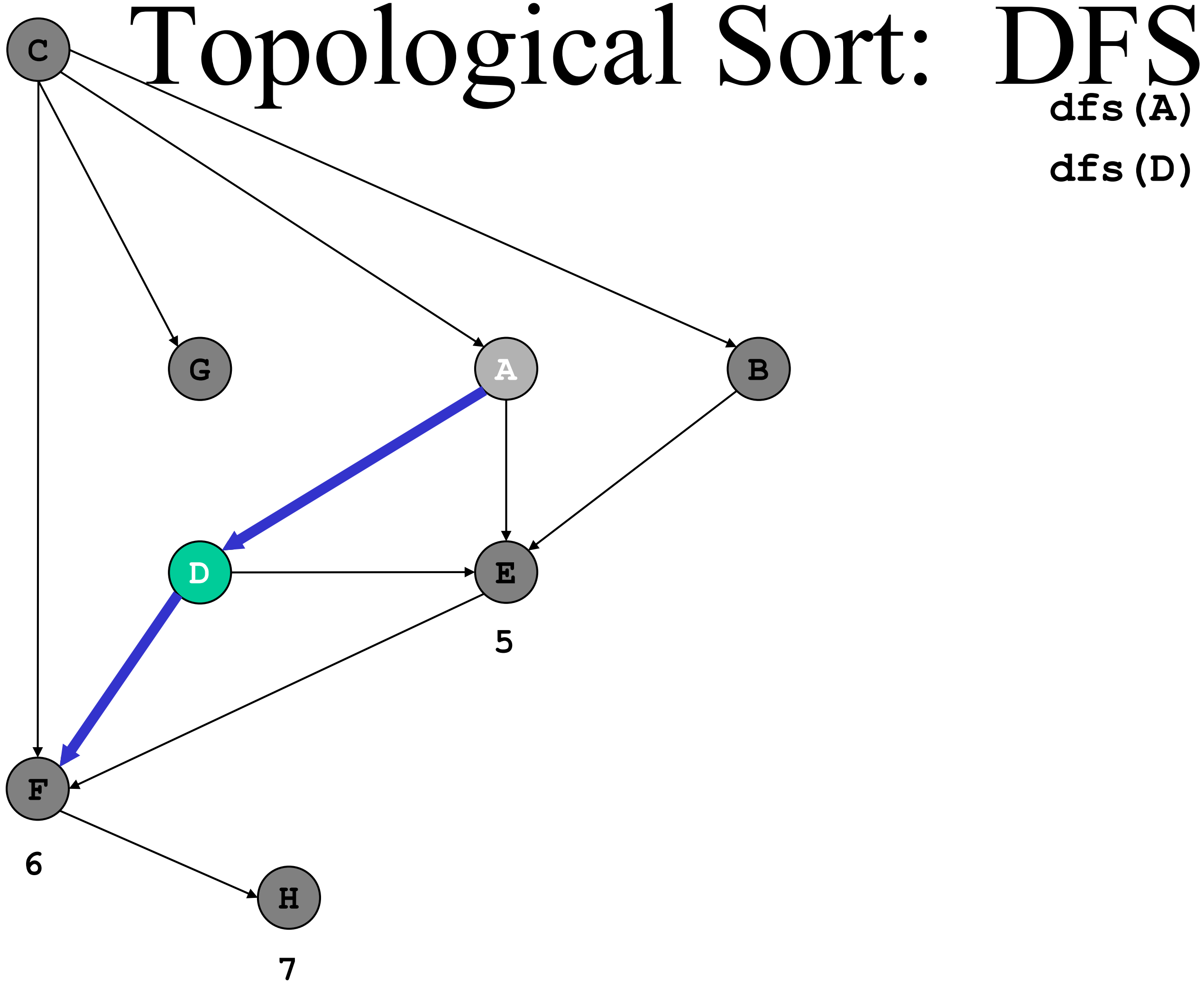


dfs(A)

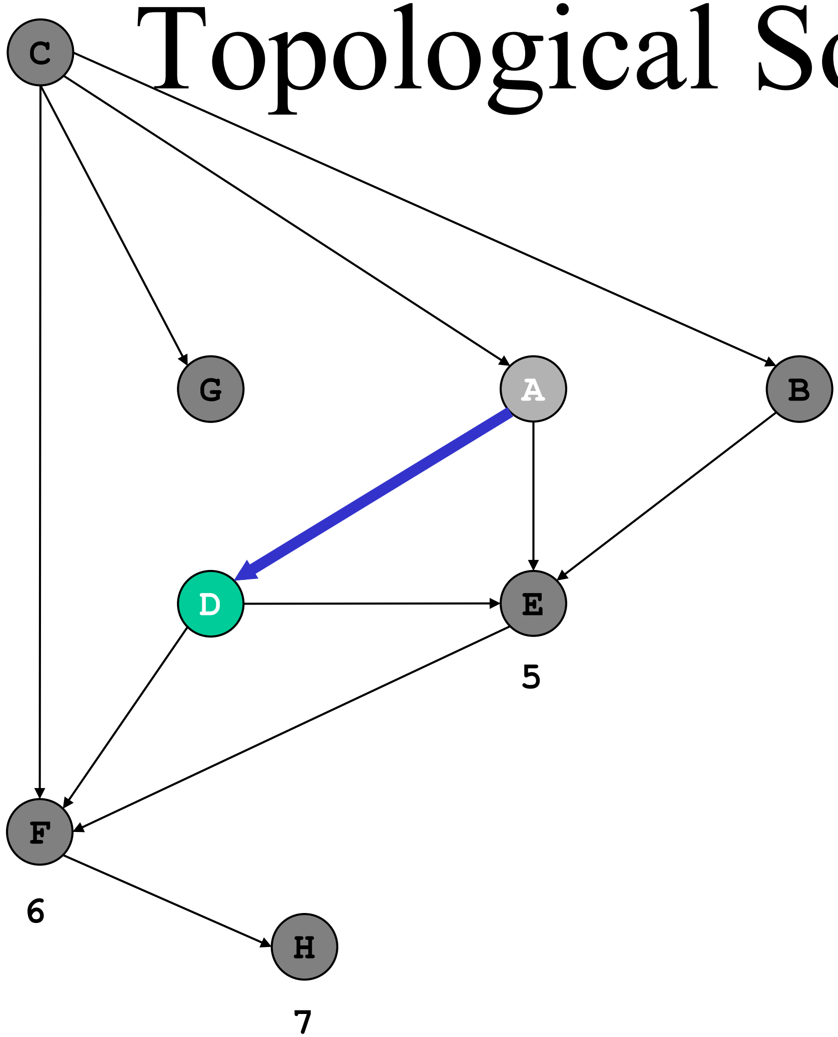
dfs(D)

dfs(E)

Topological Sort: DFS

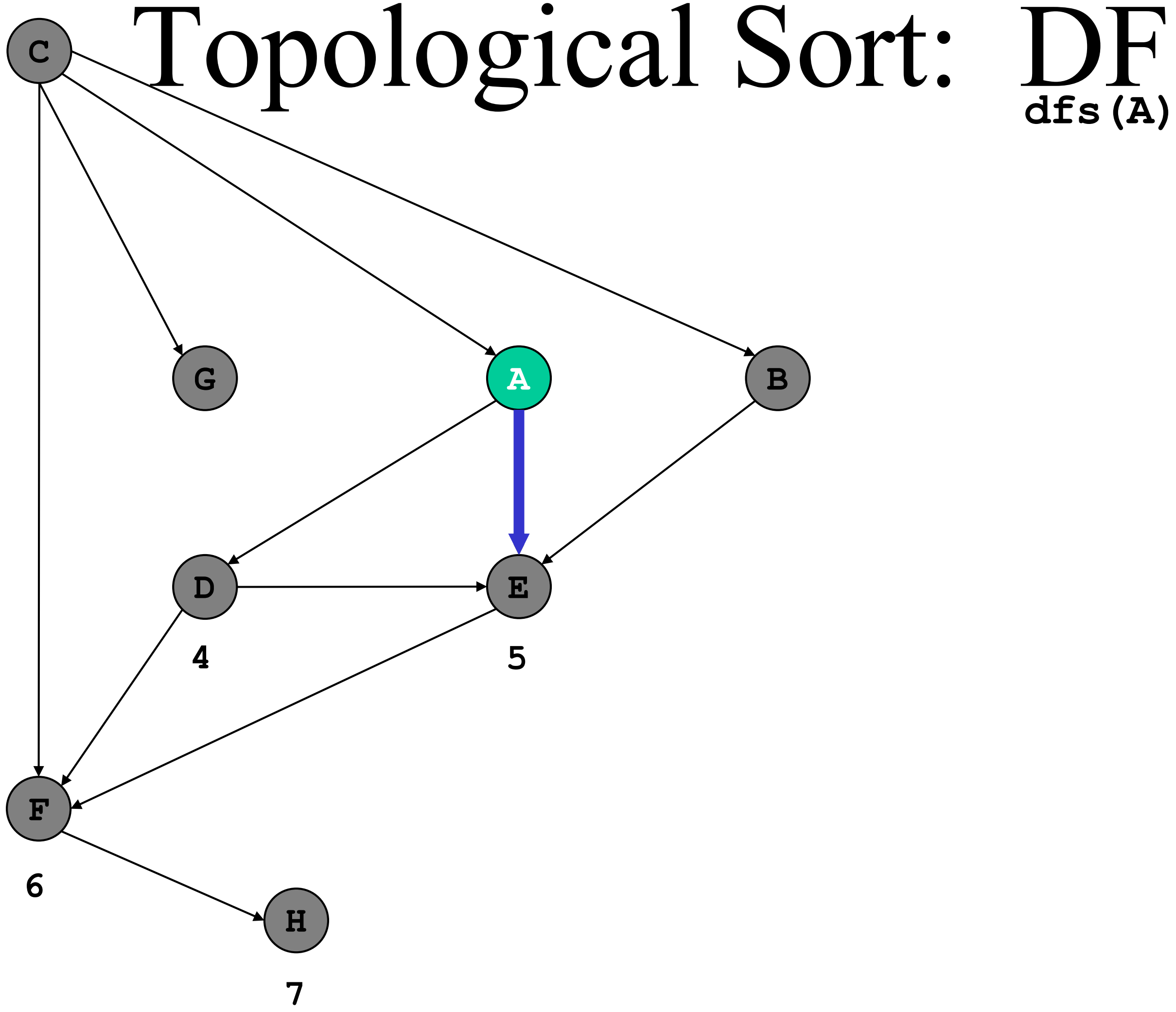


Topological Sort: DFS

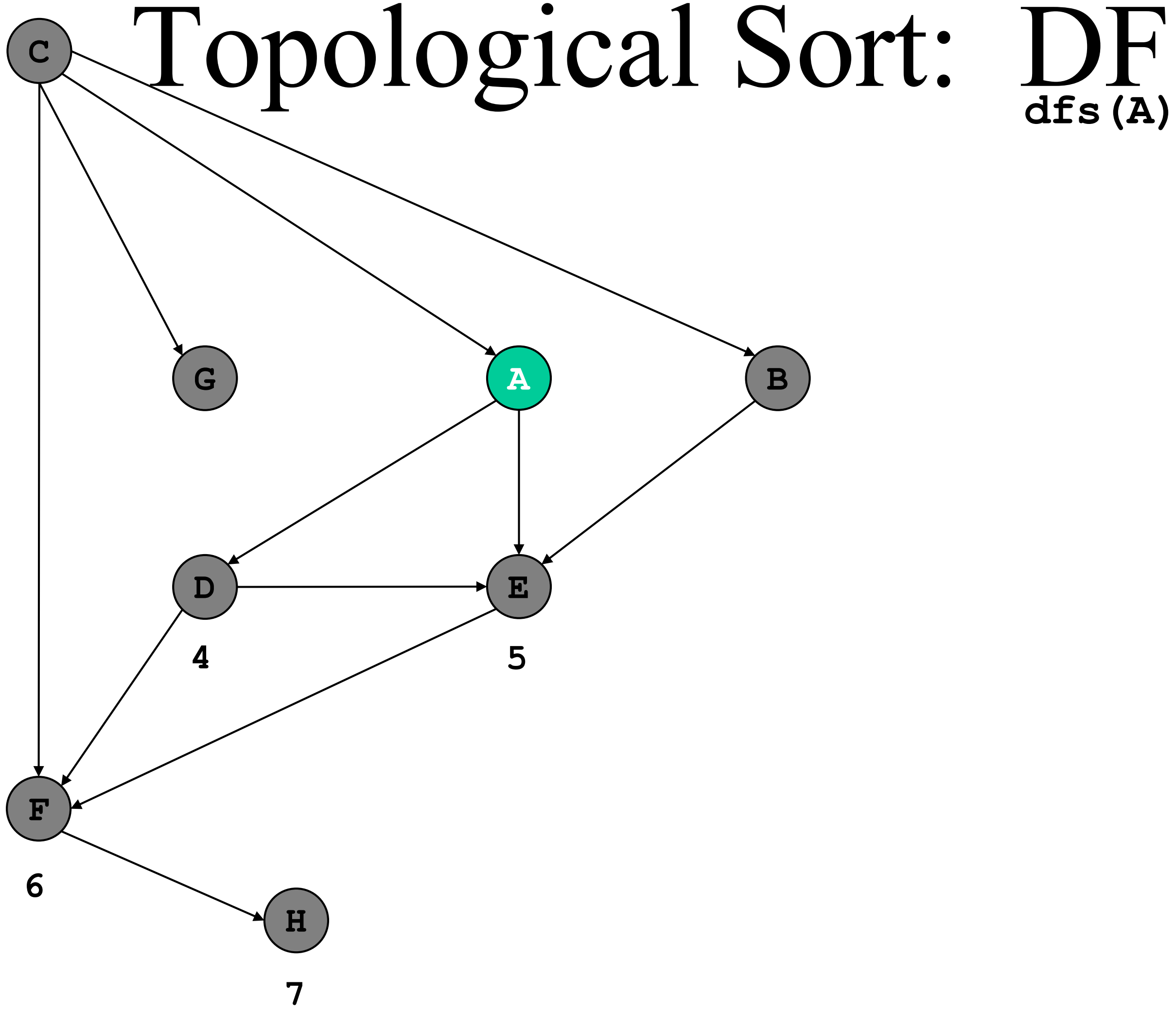



```
dfs(A)
dfs(D)
```

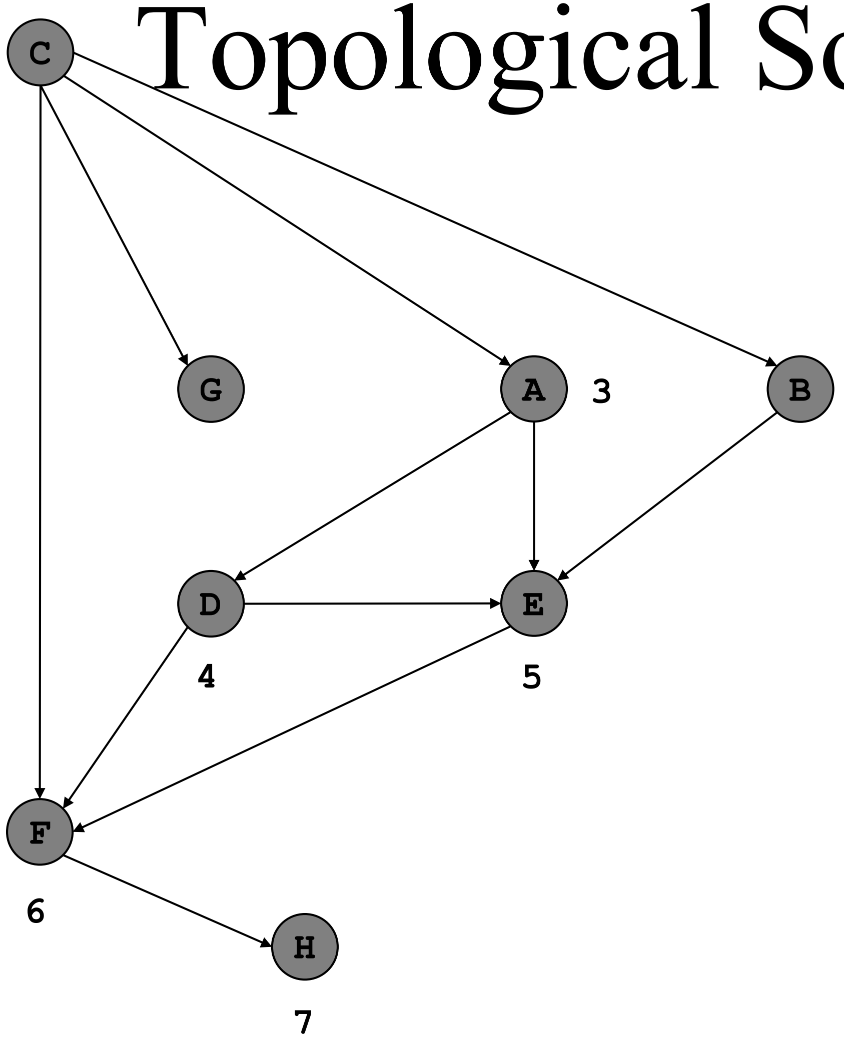
Topological Sort: DFS



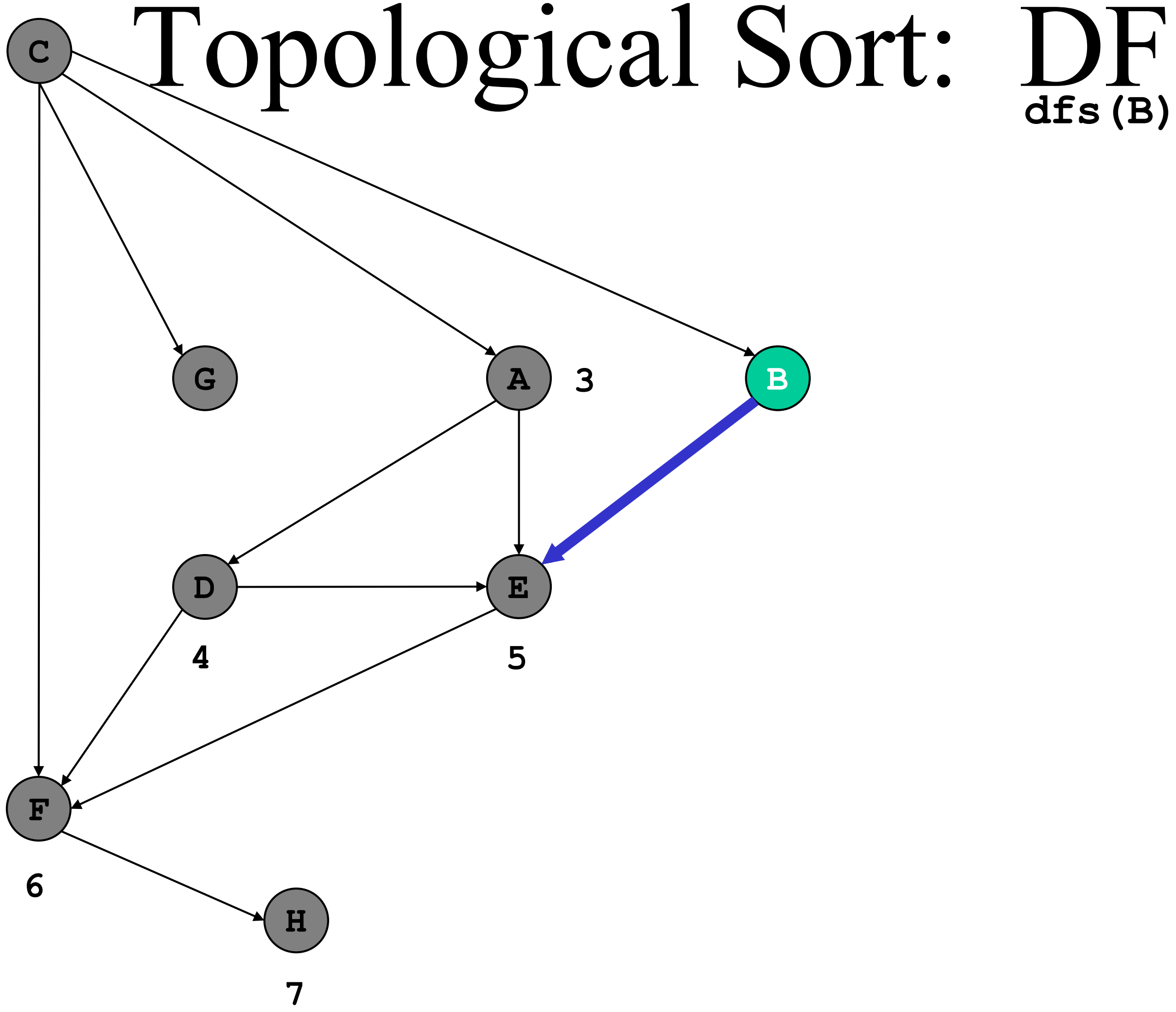
Topological Sort: DFS



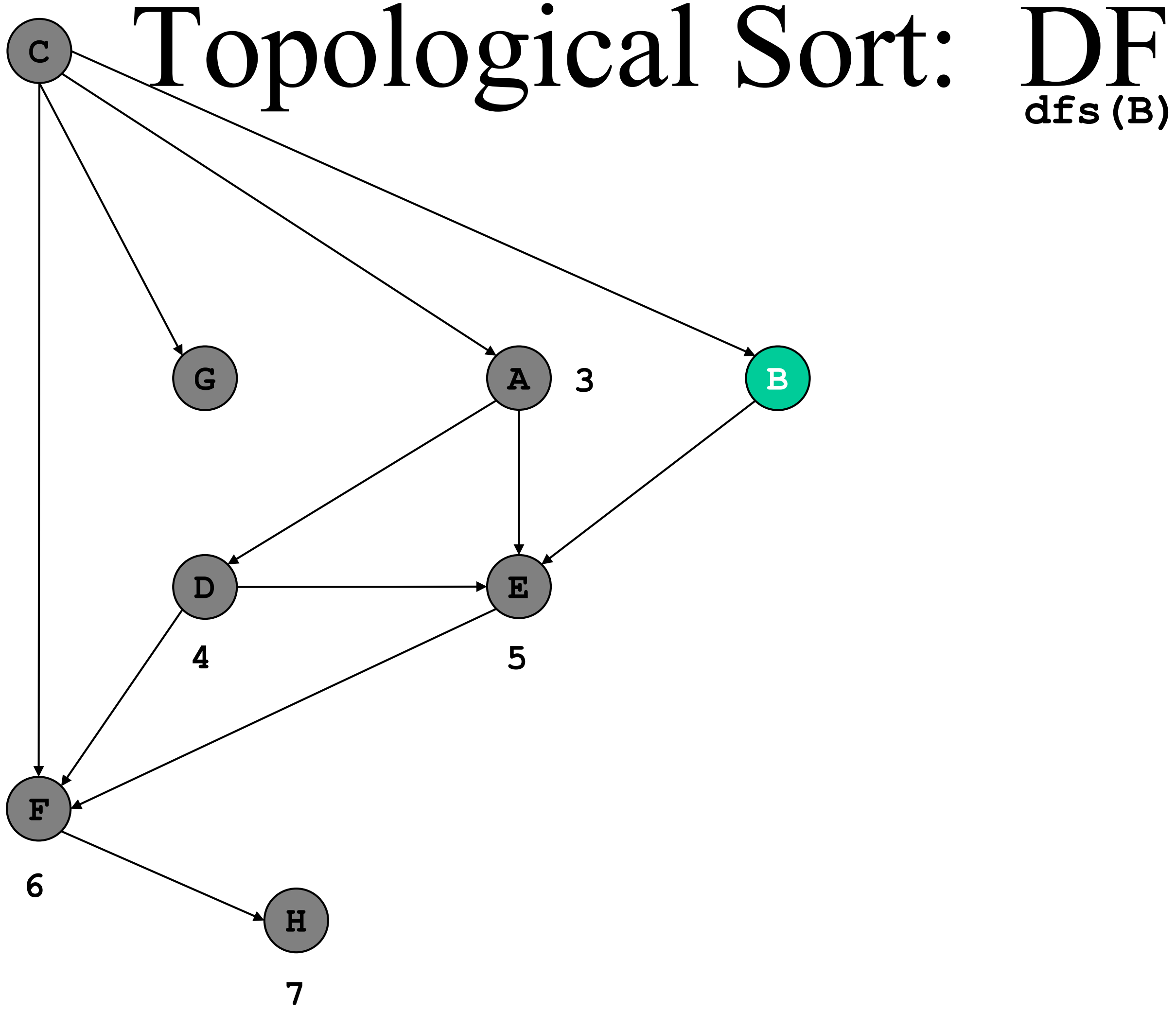
Topological Sort: DFS



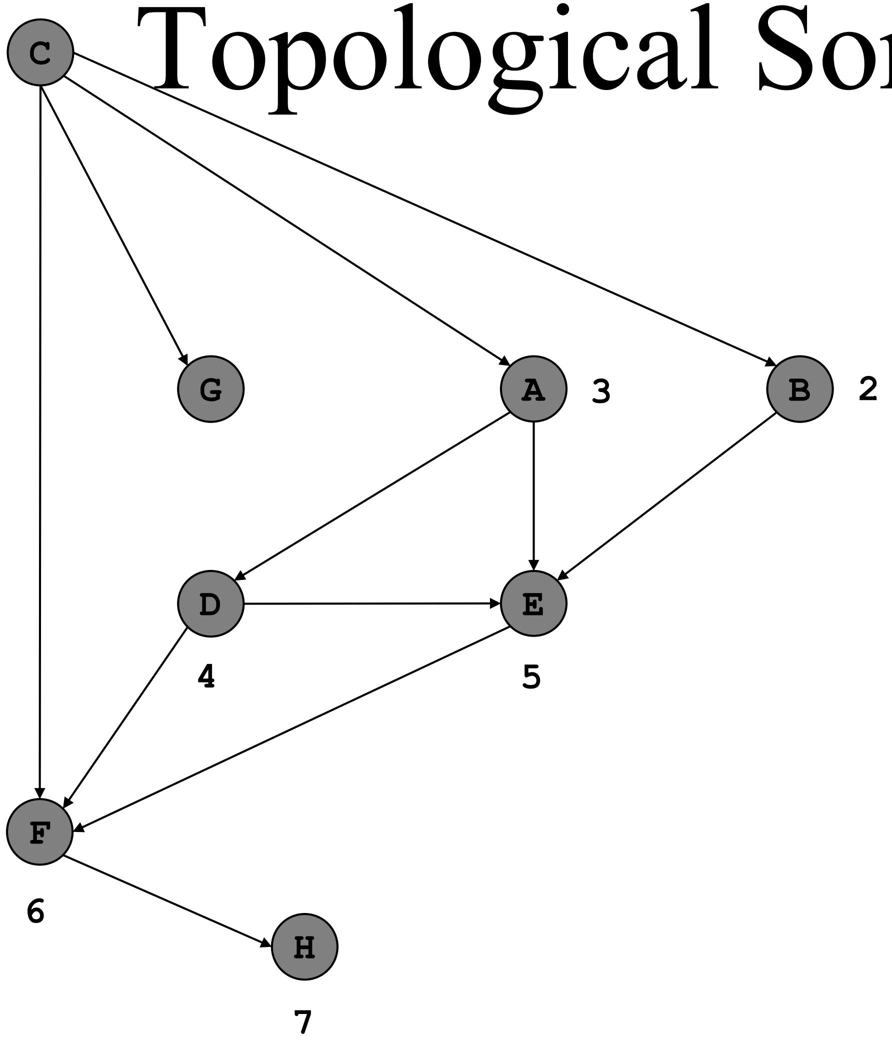
Topological Sort: DFS



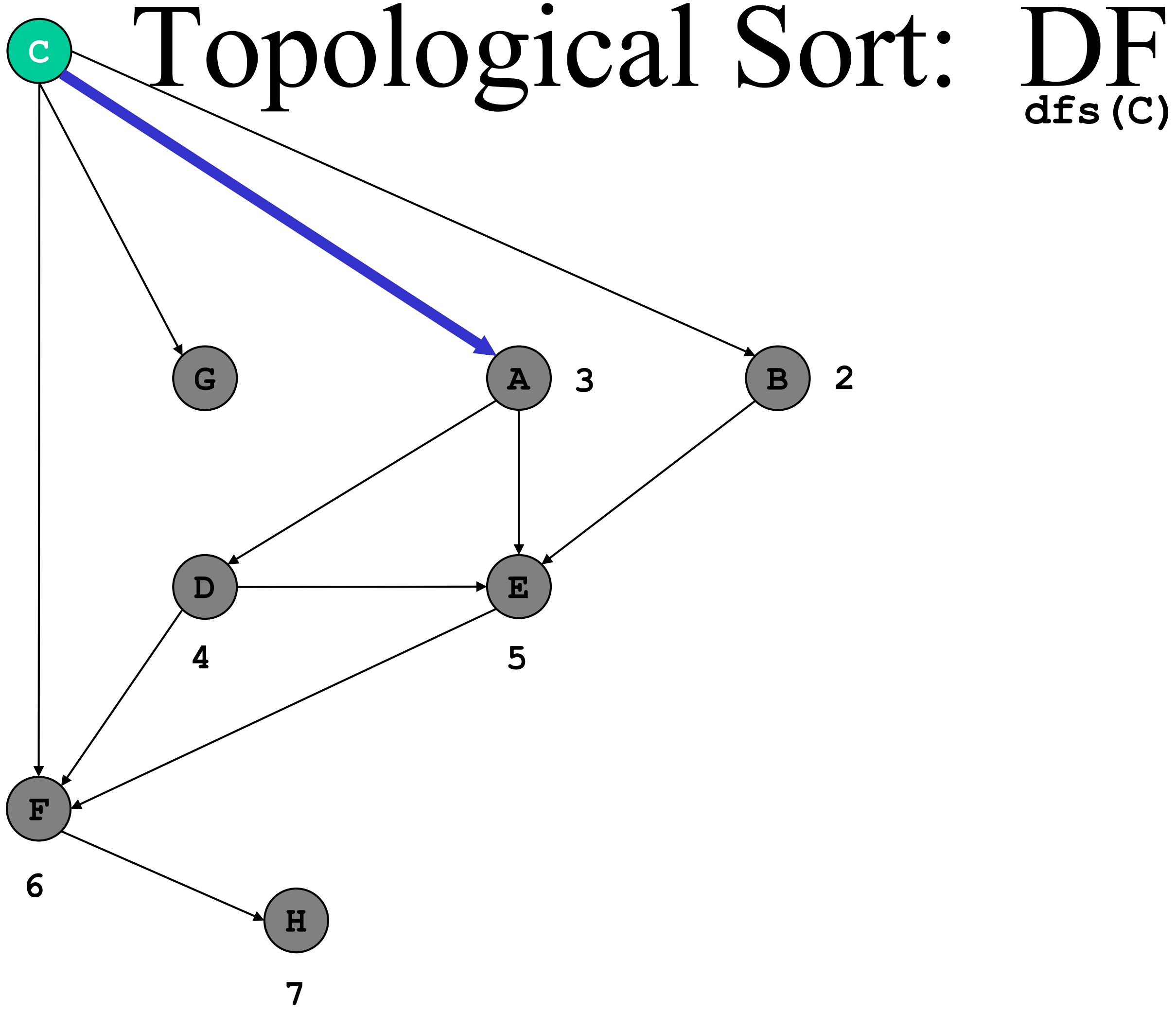
Topological Sort: DFS



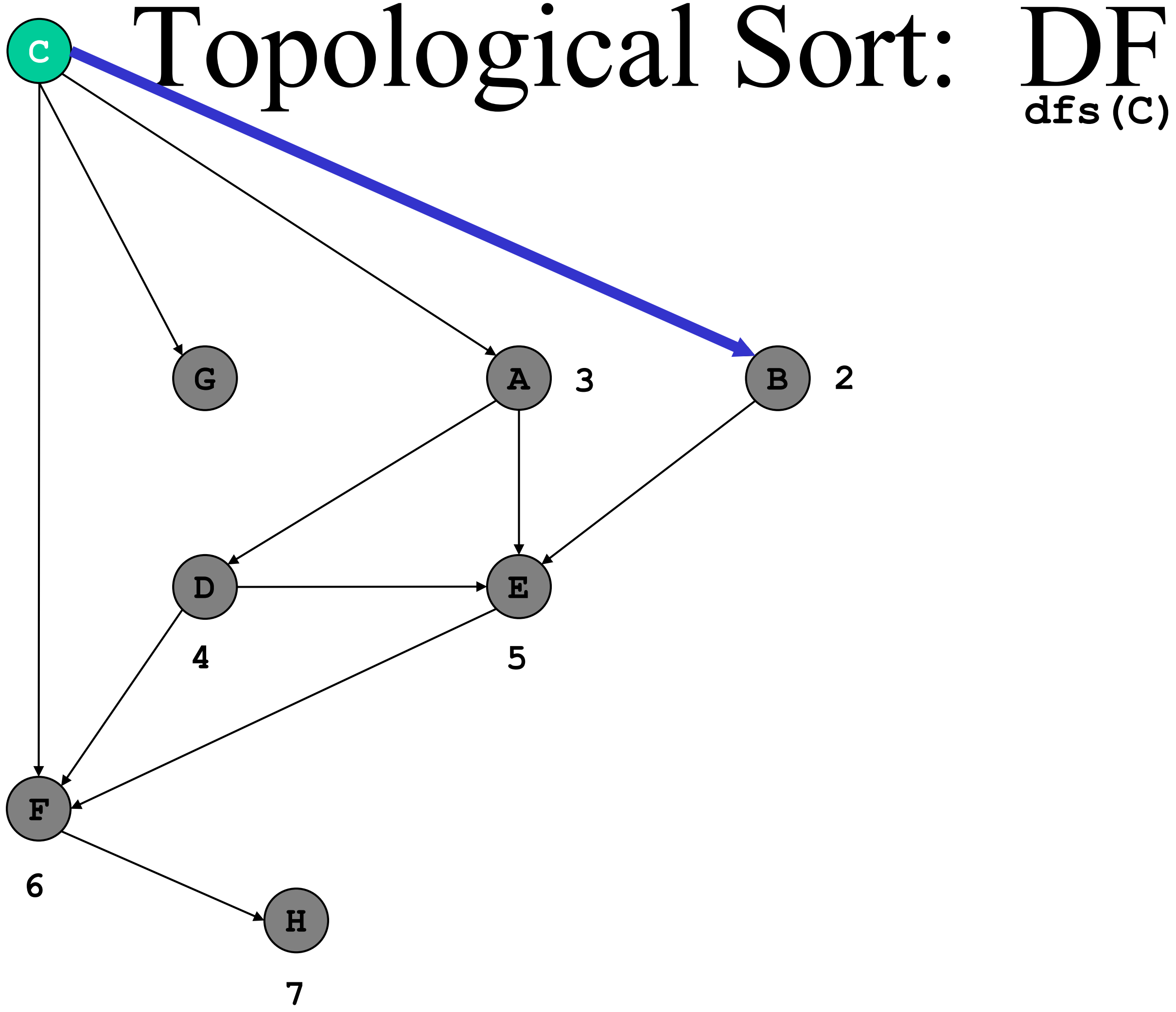
Topological Sort: DFS



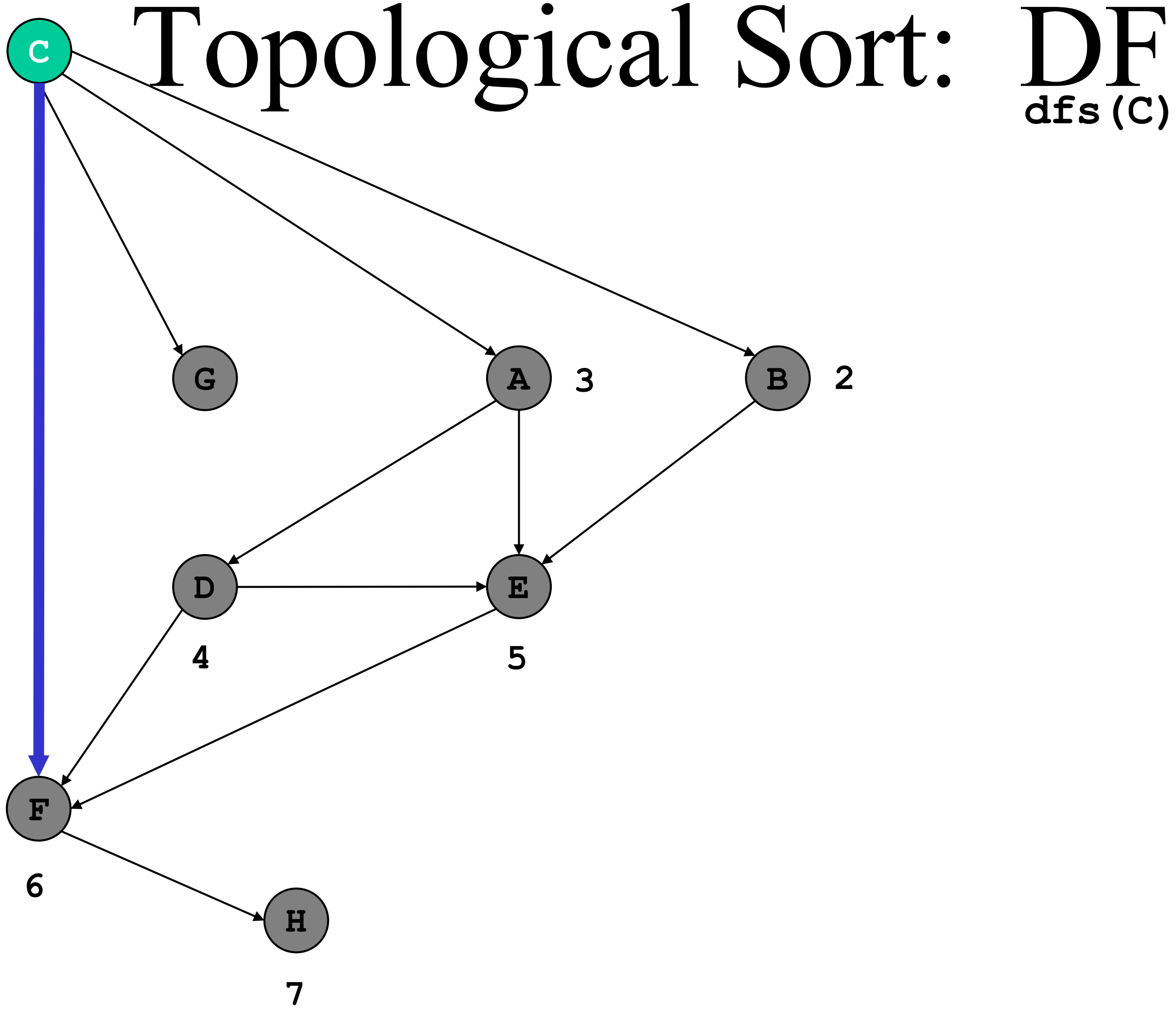
Topological Sort: DFS



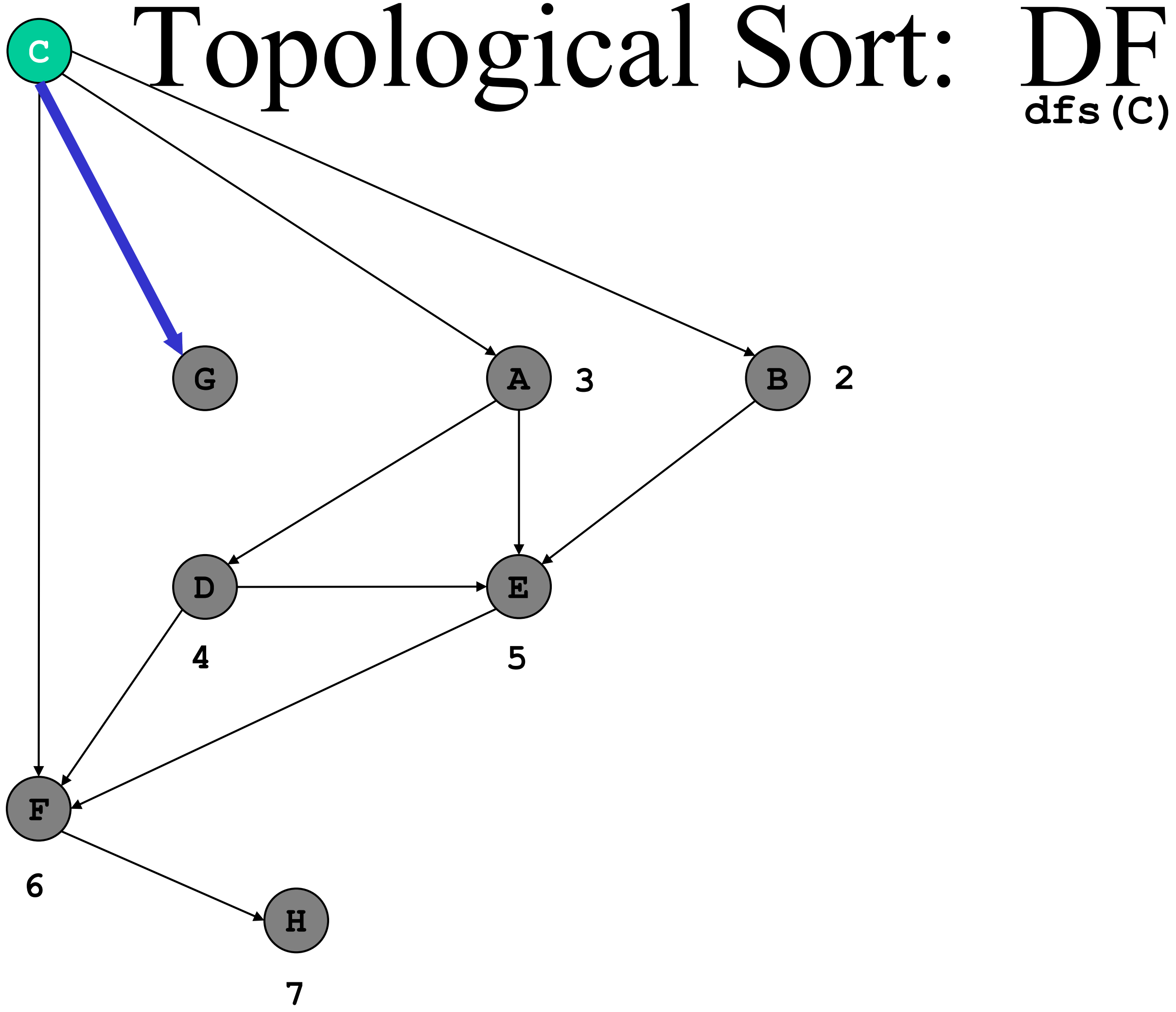
Topological Sort: DFS



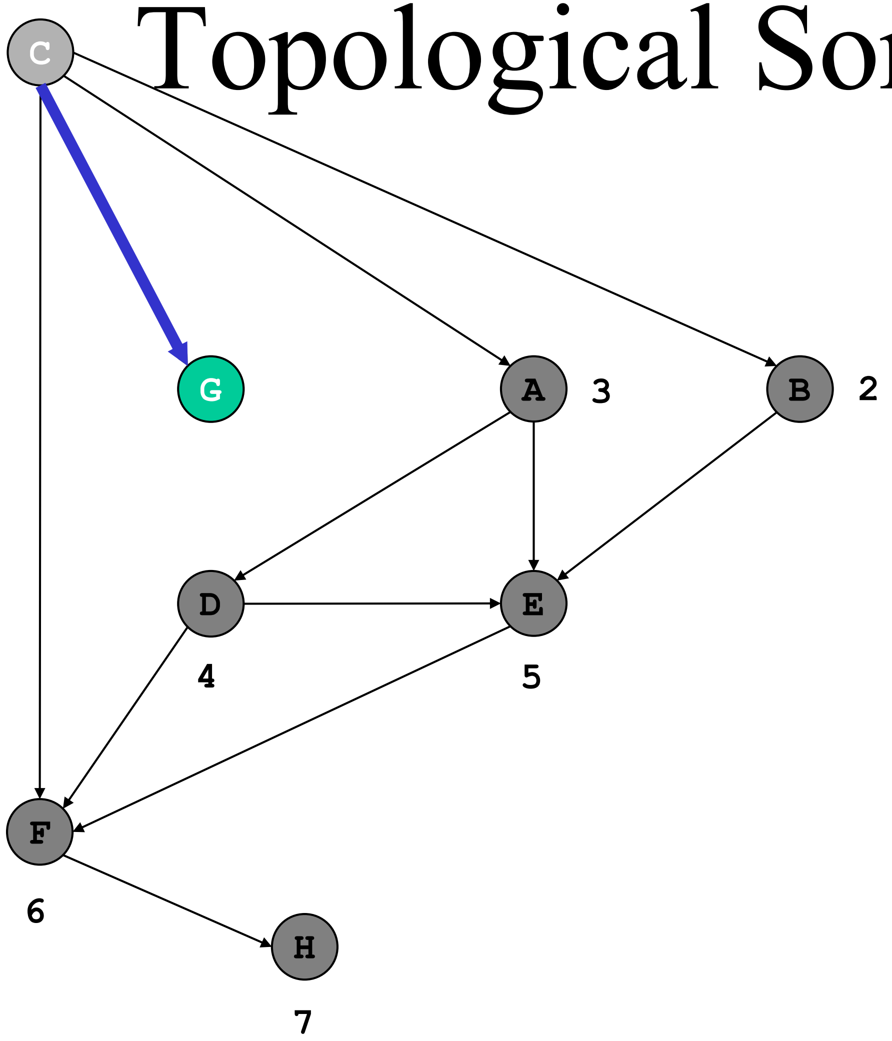
Topological Sort: DFS



Topological Sort: DFS



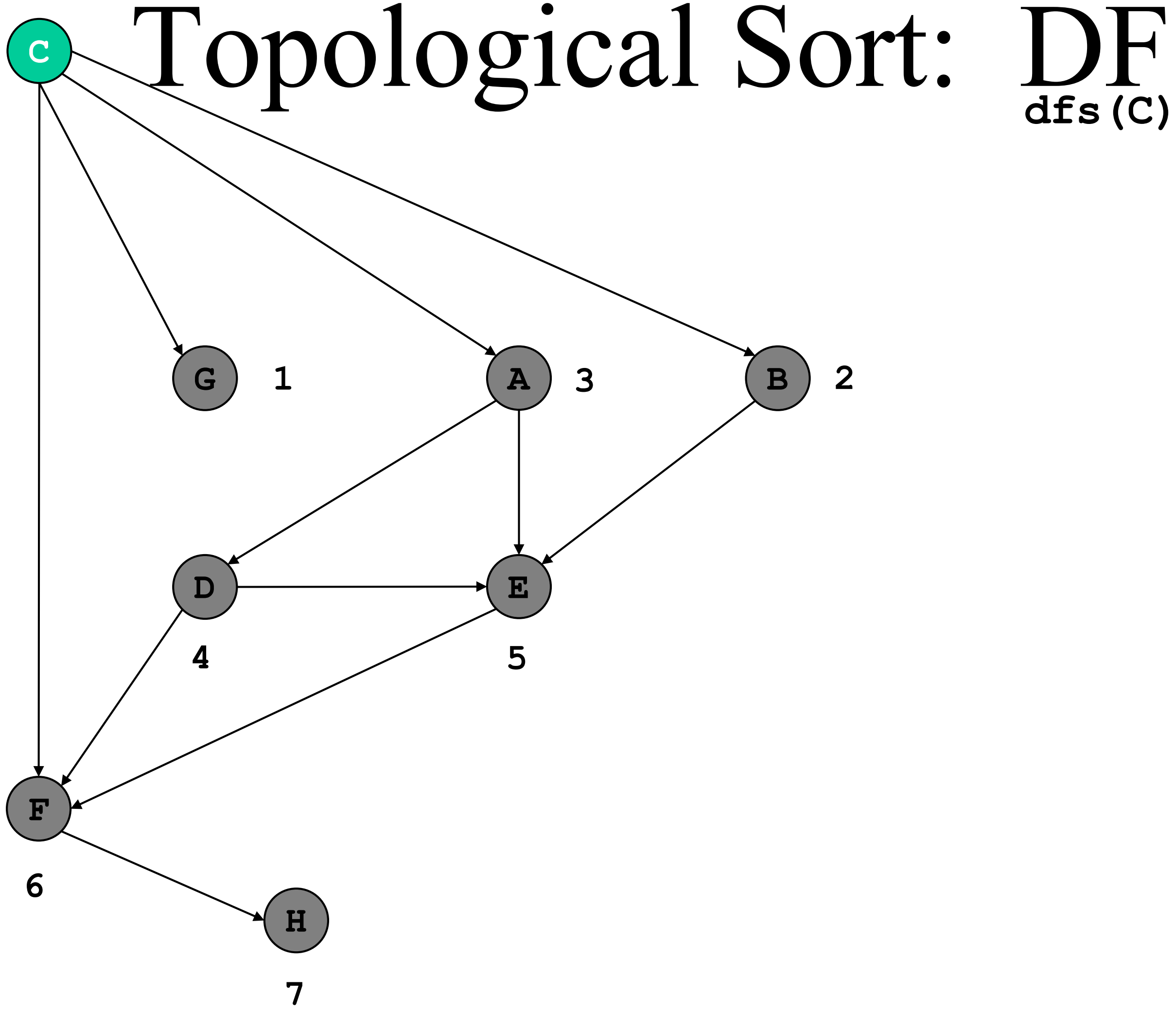
Topological Sort: DFS



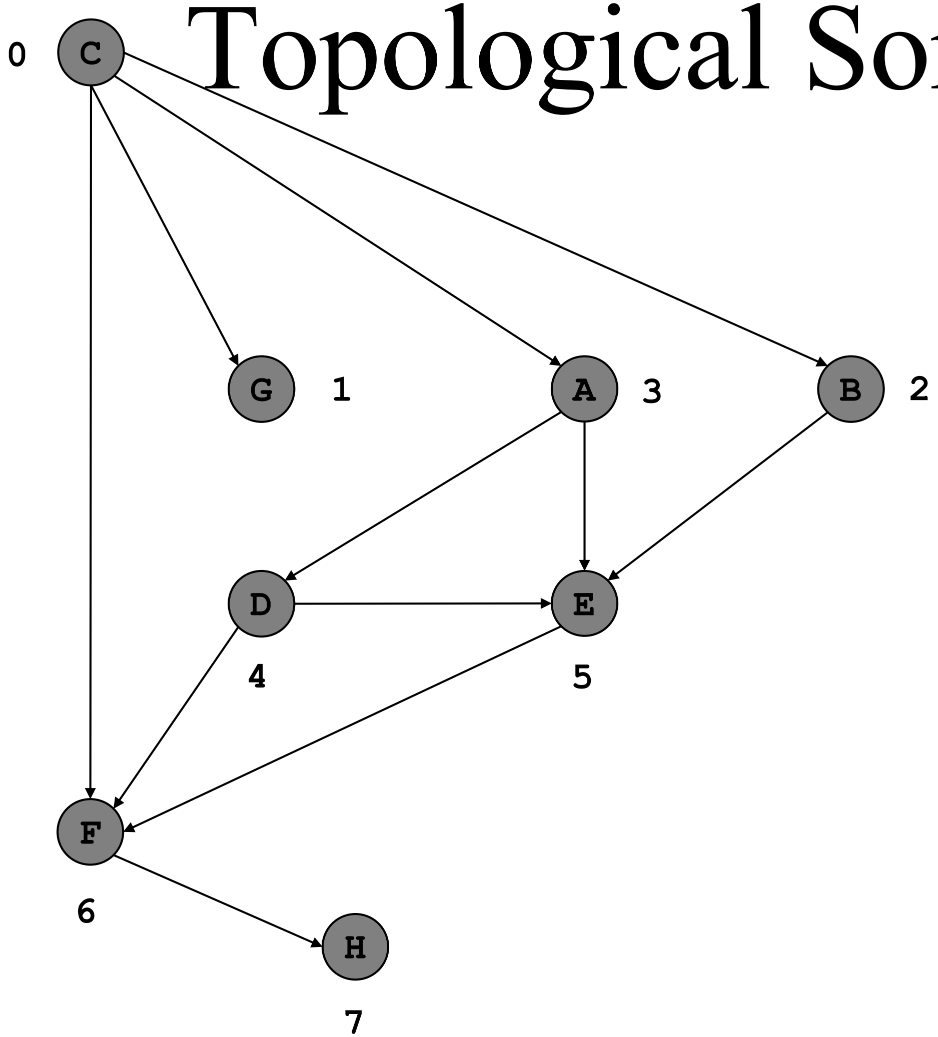
dfs(C)

dfs(G)

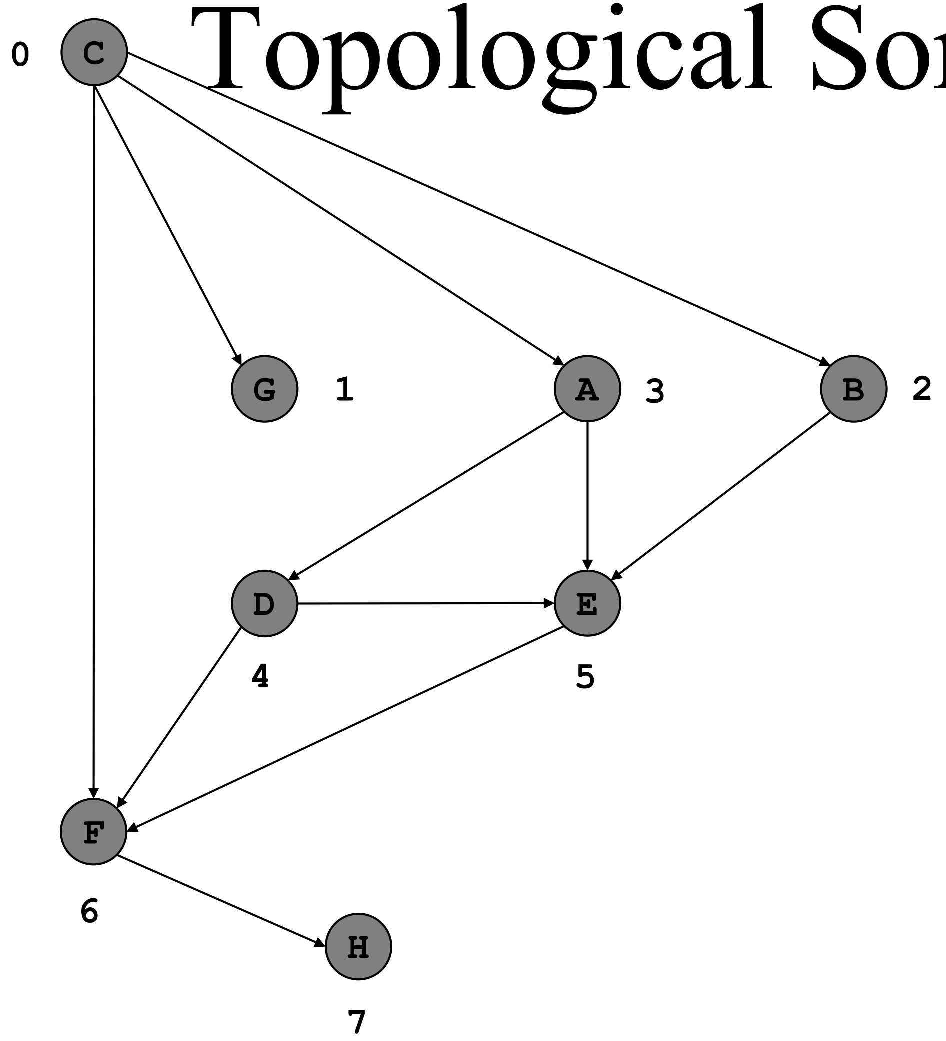
Topological Sort: DFS



Topological Sort: DFS



Topological Sort: DFS



Topological order: **C G B A D E F H**

Exercise – Find the Topological order