

Design and Analysis of Algorithms

Part IV: Graph Algorithms

Lecture 10: Review of BFS/DFS, Topological Sort and Strongly Connected Components



Yongxin Tong (童咏昕)

School of CSE, Beihang University

yxtong@buaa.edu.cn

Outline

- Introduction to Part IV
- Review of Basic Graph Search Algorithms
 - Basic Concepts
 - The Breadth-First Search (BFS) Algorithm
 - The Depth-First Search (DFS) Algorithm
- Topological Sort
 - The Topological Sort Algorithm
 - Analysis of the Topological Sort Algorithm
- Strongly Connected Components
 - The Algorithm of Finding SCCs
 - Analysis of the Algorithm

Outline

- **Introduction to Part IV**
- **Review of Basic Graph Search Algorithms**
 - Basic Concepts
 - The Breadth-First Search (BFS) Algorithm
 - The Depth-First Search (DFS) Algorithm
- **Topological Sort**
 - The Topological Sort Algorithm
 - Analysis of the Topological Sort Algorithm
- **Strongly Connected Components**
 - The Algorithm of Finding SCCs
 - Analysis of the Algorithm

Introduction to Part IV

- In Part IV, we will illustrate several graph algorithm problems using several examples:
 - Basic Concepts of Graphs (图的基本概念)
 - Breadth-First Search [BFS] (广度优先搜索)
 - Depth-First Search [DFS] (深度优先搜索)
 - Topological Sort (拓扑排序)
 - Strongly Connected Components (强联通分量)
 - Minimum Spanning Trees (最小生成树)
 - Shortest Path (最短路径)
 - All-Pairs Shortest Paths (所有结点对的最短路径)
 - Maximum/Network Flows (最大流/网络流)

Introduction to Part IV

- In Part IV, we will illustrate several graph algorithm problems using several examples:
 - Basic Concepts of Graphs (图的基本概念)
 - Breadth-First Search [BFS] (广度优先搜索)
 - Depth-First Search [DFS] (深度优先搜索)
 - Topological Sort (拓扑排序)
 - Strongly Connected Components (强联通分量)
 - Minimum Spanning Trees (最小生成树)
 - Shortest Path (最短路径)
 - All-Pairs Shortest Paths (所有结点对的最短路径)
 - Maximum/Network Flows (最大流/网络流)

Outline

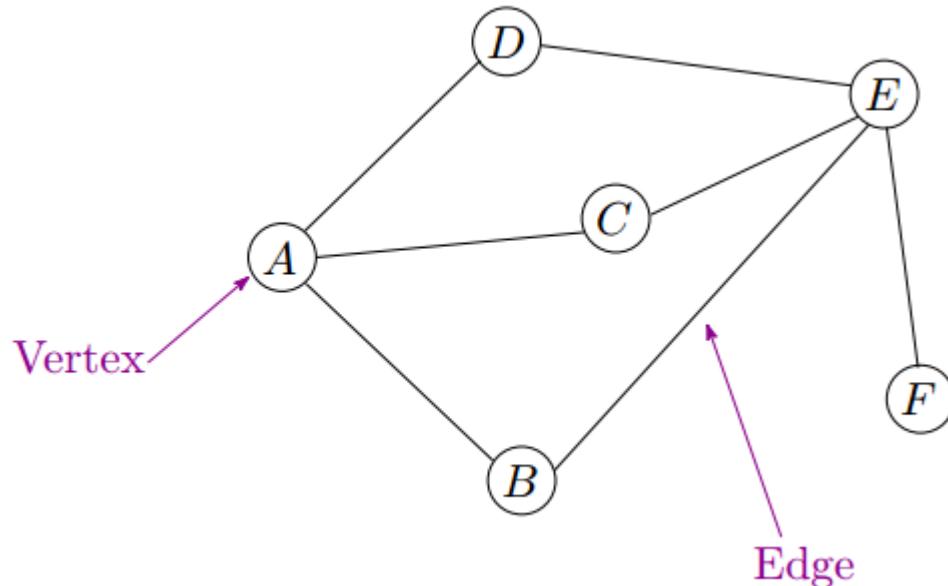
- Introduction to Part IV
- Review of Basic Graph Search Algorithms
 - Basic Concepts
 - The Breadth-First Search (BFS) Algorithm
 - The Depth-First Search (DFS) Algorithm
- Topological Sort
 - The Topological Sort Algorithm
 - Analysis of the Topological Sort Algorithm
- Strongly Connected Components
 - The Algorithm of Finding SCCs
 - Analysis of the Algorithm

Graphs

- Extremely useful tool in modeling problems

Graphs

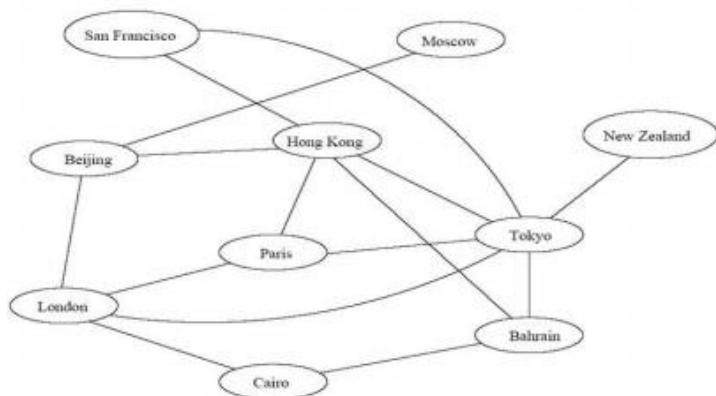
- Extremely useful tool in modeling problems
- Consist of:
 - Vertices
 - Edges



Vertices can be considered as “sites” or locations.

Edges represent connections.

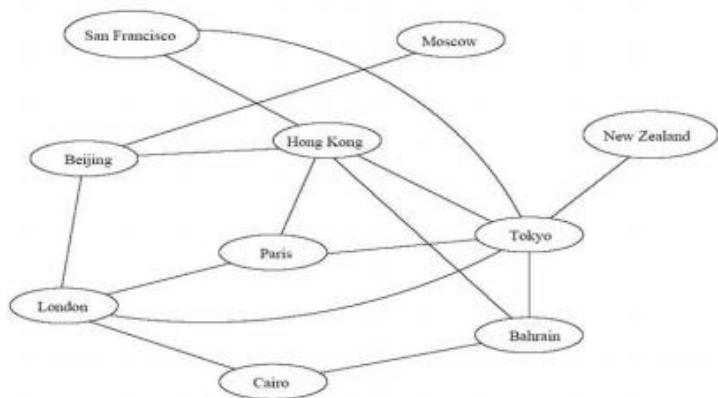
Graph Application



Air flight system



Graph Application

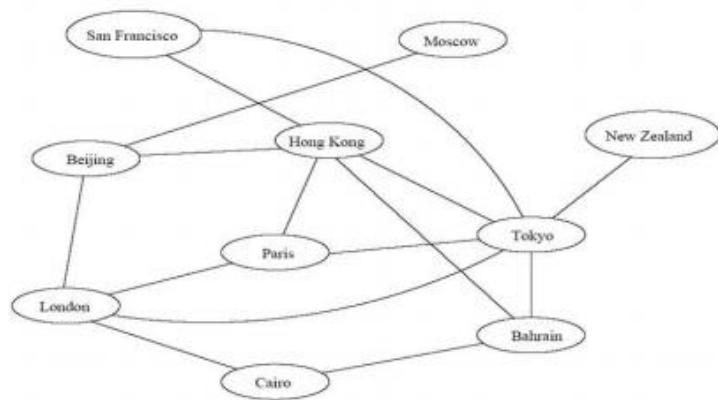


Air flight system



- Each vertex represents a city

Graph Application

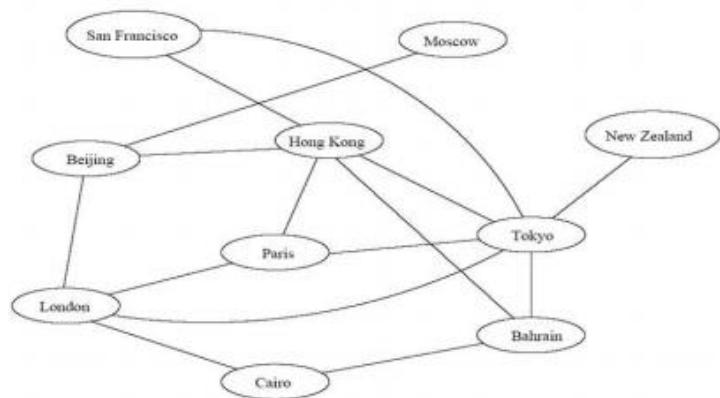


Air flight system



- Each vertex represents a city
- Each edge represents a direct flight between two cities

Graph Application

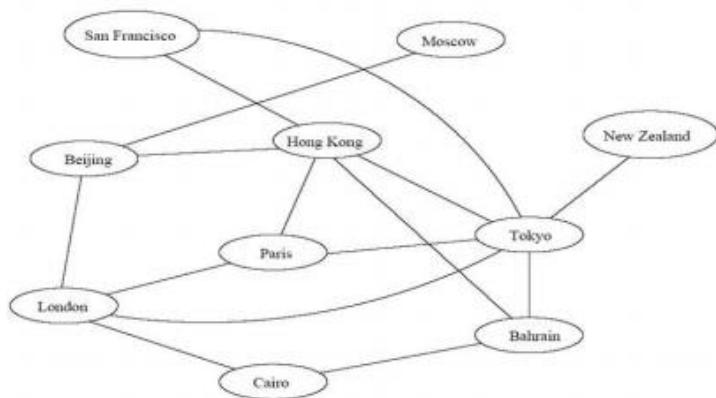


Air flight system



- Each vertex represents a city
- Each edge represents a direct flight between two cities
- A query on direct flight = a query on whether an edge exists

Graph Application

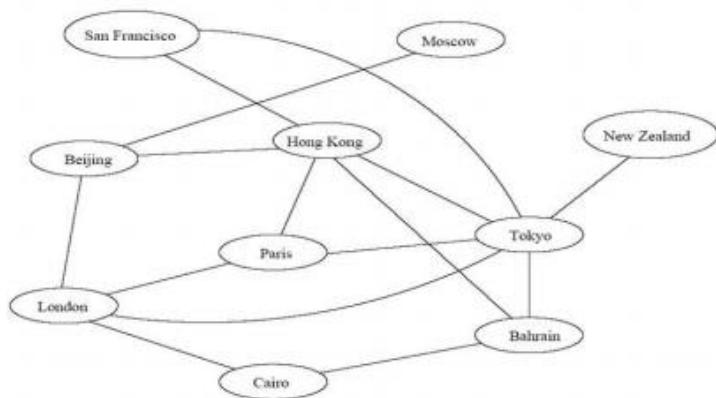


Air flight system



- Each vertex represents a city
- Each edge represents a direct flight between two cities
- A query on direct flight = a query on whether an edge exists
- A query on how to get to a location = does a path exist from A to B

Graph Application

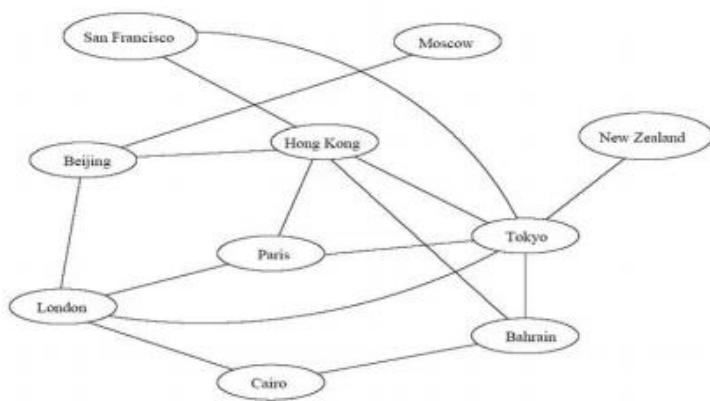


Air flight system

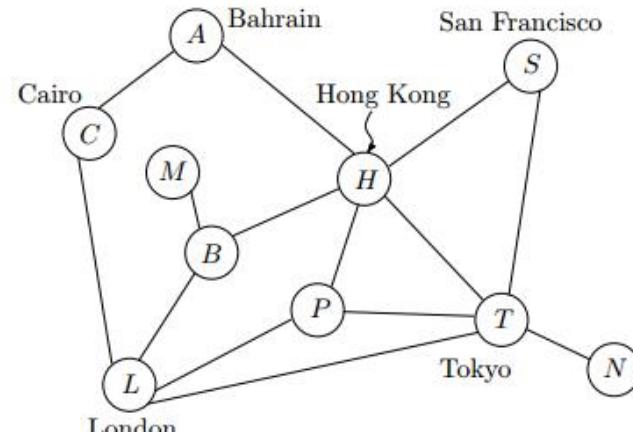


- Each vertex represents a city
- Each edge represents a direct flight between two cities
- A query on direct flight = a query on whether an edge exists
- A query on how to get to a location = does a path exist from A to B
- We can even associate costs to edges (weighted graphs), then ask “what is the cheapest path from A to B”

Graph Application



original graph



simplified graph

- Each vertex represents a city
- Each edge represents a direct flight between two cities
- A query on direct flight = a query on whether an edge exists
- A query on how to get to a location = does a path exist from A to B
- We can even associate costs to edges (weighted graphs), then ask “what is the cheapest path from A to B”

Why Graph Algorithms?

- Graph is a pervasive data structure in computer science

Why Graph Algorithms?

- Graph is a pervasive data structure in computer science
 - Networks: the Internet



Vinton G. Cerf & Robert E. Kahn

The Recipient of the Turing Award in 2004, USA.

Contributions: *for pioneering work on internetworking, including the design and implementation of the Internet's basic communications protocols, TCP/IP, etc.*

Why Graph Algorithms?

- Graph is a pervasive data structure in computer science
 - Networks: the Internet, World Wide Web



Tim Berners-Lee

The Recipient of the Turing Award in 2016, UK.

Contributions: *for inventing the World Wide Web, the first web browser, and the fundamental protocols and algorithms allowing the Web to scale.*

Why Graph Algorithms?

- Graph is a pervasive data structure in computer science
 - Networks: the Internet, World Wide Web, wireless networks

Why Graph Algorithms?

- Graph is a pervasive data structure in computer science
 - Networks: the Internet, World Wide Web, wireless networks
 - Logistics: transportation, supply chain management

Why Graph Algorithms?

- Graph is a pervasive data structure in computer science
 - Networks: the Internet, World Wide Web, wireless networks
 - Logistics: transportation, supply chain management
 - Relationship between objects: online dating, social networks (Facebook!)

Why Graph Algorithms?

- Graph is a pervasive data structure in computer science
 - Networks: the Internet, World Wide Web, wireless networks
 - Logistics: transportation, supply chain management
 - Relationship between objects: online dating, social networks (Facebook!)
- Hundreds of interesting computational problems defined on graphs

Why Graph Algorithms?

- Graph is a pervasive data structure in computer science
 - Networks: the Internet, World Wide Web, wireless networks
 - Logistics: transportation, supply chain management
 - Relationship between objects: online dating, social networks (Facebook!)
- Hundreds of interesting computational problems defined on graphs
- We will sample a few basic ones

Definition

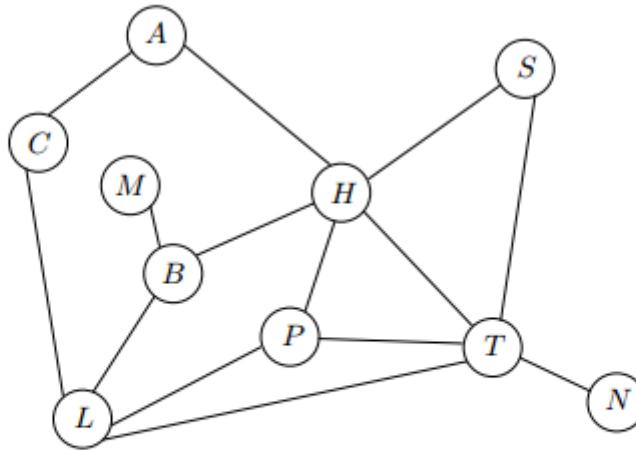
- A **graph** $G = (V, E)$ consists of
 - a set of **vertices** V , $|V| = n$, and
 - a set of **edges** E , $|E| = m$

Definition

- A **graph** $G = (V, E)$ consists of
 - a set of **vertices** V , $|V| = n$, and
 - a set of **edges** E , $|E| = m$
- Each edge is a pair of (u, v) , where u, v belongs to V

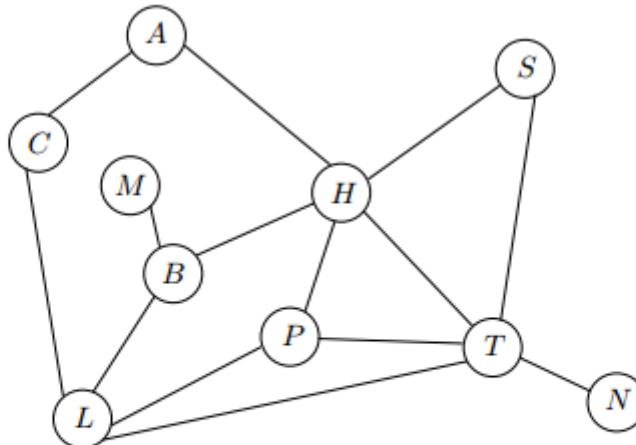
Definition

- A **graph** $G = (V, E)$ consists of
 - a set of **vertices** V , $|V| = n$, and
 - a set of **edges** E , $|E| = m$
- Each edge is a pair of (u, v) , where u, v belongs to V



Definition

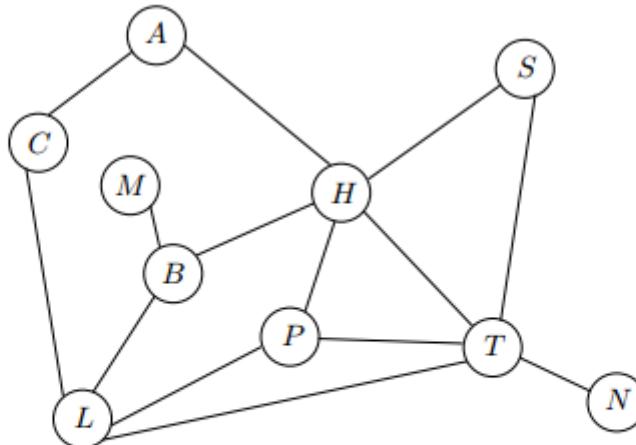
- A **graph** $G = (V, E)$ consists of
 - a set of **vertices** V , $|V| = n$, and
 - a set of **edges** E , $|E| = m$
- Each edge is a pair of (u, v) , where u, v belongs to V



$$V = \{A, B, C, H, L, M, N, P, S, T\}$$

Definition

- A **graph** $G = (V, E)$ consists of
 - a set of **vertices** V , $|V| = n$, and
 - a set of **edges** E , $|E| = m$
- Each edge is a pair of (u, v) , where u, v belongs to V

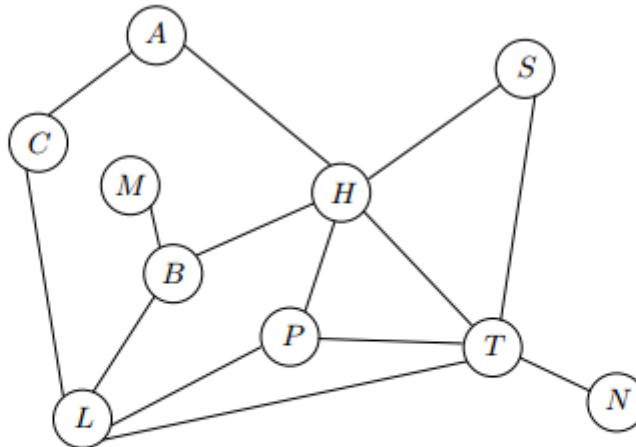


$$V = \{A, B, C, H, L, M, N, P, S, T\}$$

$$E = \{(A, C), (A, H), \dots, (H, P), \dots\}$$

Definition

- A **graph** $G = (V, E)$ consists of
 - a set of **vertices** V , $|V| = n$, and
 - a set of **edges** E , $|E| = m$
- Each edge is a pair of (u, v) , where u, v belongs to V



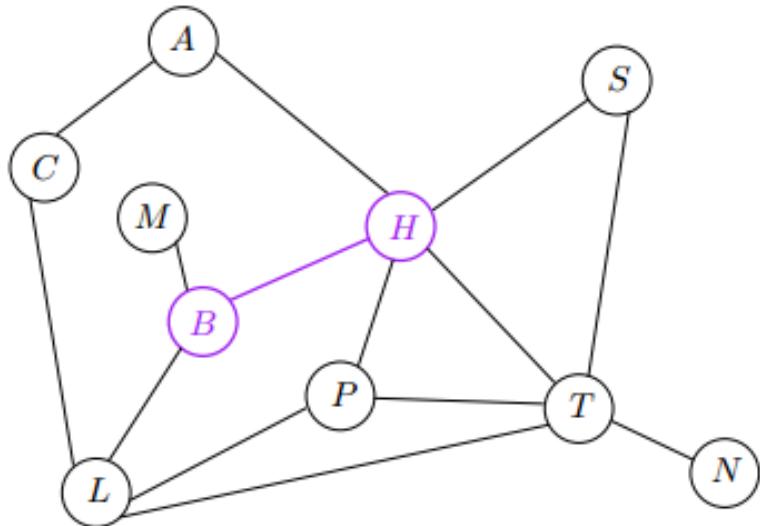
$$V = \{A, B, C, H, L, M, N, P, S, T\}$$

$$E = \{(A, C), (A, H), \dots, (H, P), \dots\}$$

- For directed graph, we distinguish between edge (u, v) and edge (v, u) ; for undirected graph, no such distinction is made.

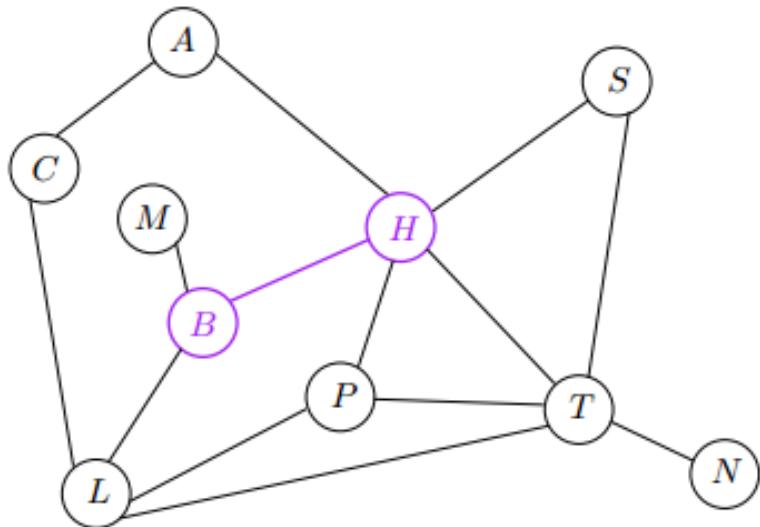
Terminology

- Each edge has two **endpoints**



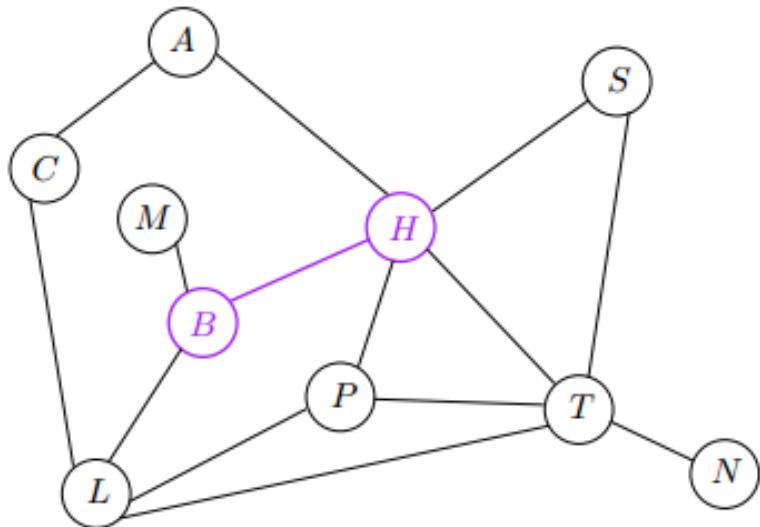
Terminology

- Each edge has two **endpoints**
 - H and B are the endpoints of (H, B)



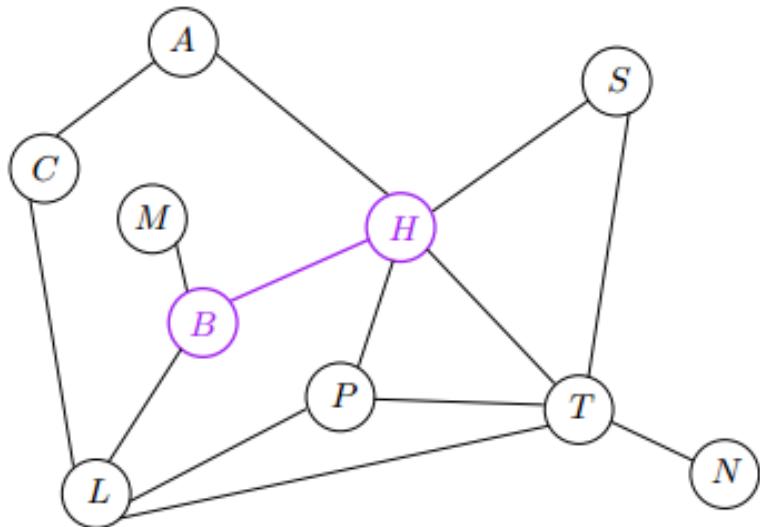
Terminology

- Each edge has two **endpoints**
 - H and B are the endpoints of (H, B)
- An edge **joins** its endpoints



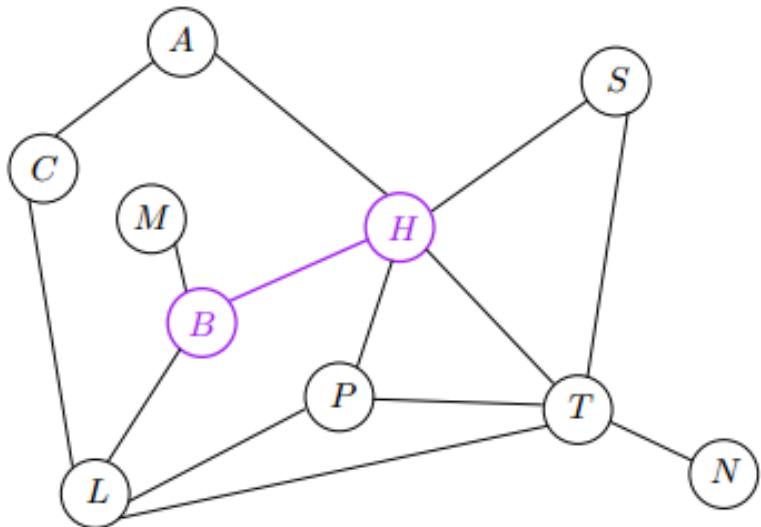
Terminology

- Each edge has two **endpoints**
 - H and B are the endpoints of (H, B)
- An edge **joins** its endpoints
 - (H, B) joins H and B



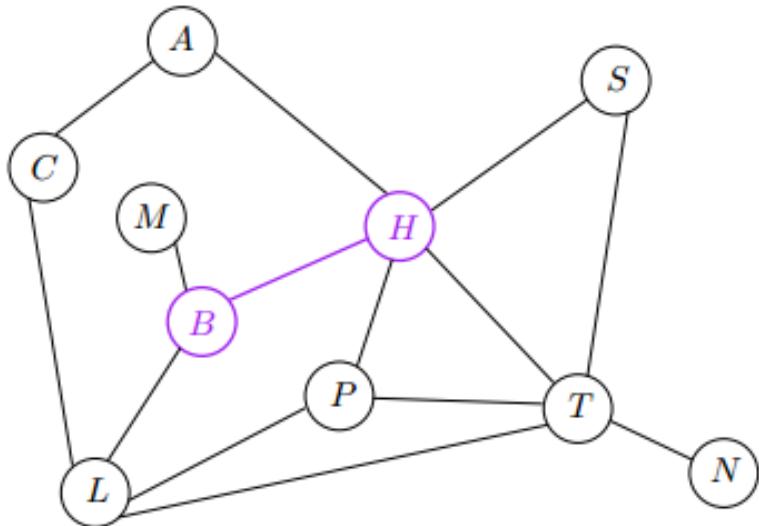
Terminology

- Each edge has two **endpoints**
 - H and B are the endpoints of (H, B)
- An edge **joins** its endpoints
 - (H, B) joins H and B
- Two vertices are **adjacent** (or **neighbors**) if they are joined by an edge.

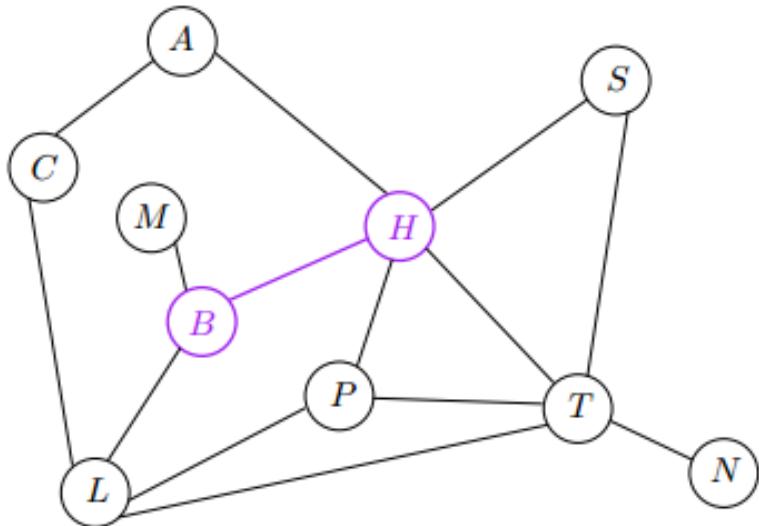


Terminology

- Each edge has two **endpoints**
 - H and B are the endpoints of (H, B)
- An edge **joins** its endpoints
 - (H, B) joins H and B
- Two vertices are **adjacent** (or **neighbors**) if they are joined by an edge.
 - H and B are adjacent

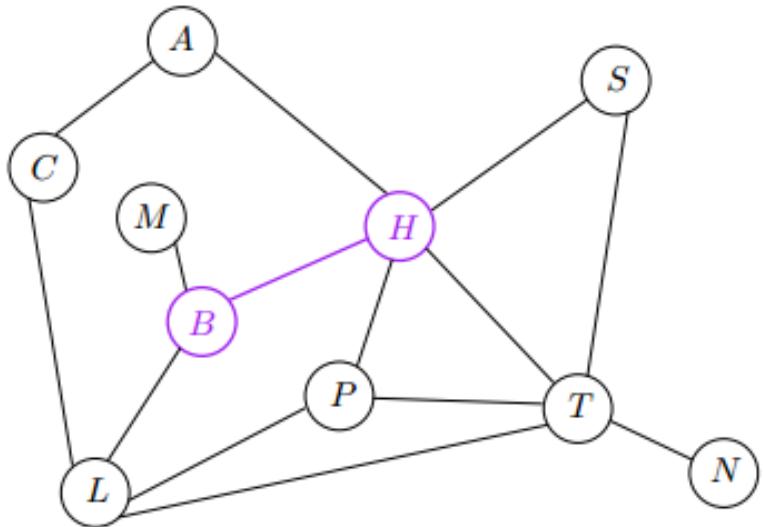


Terminology



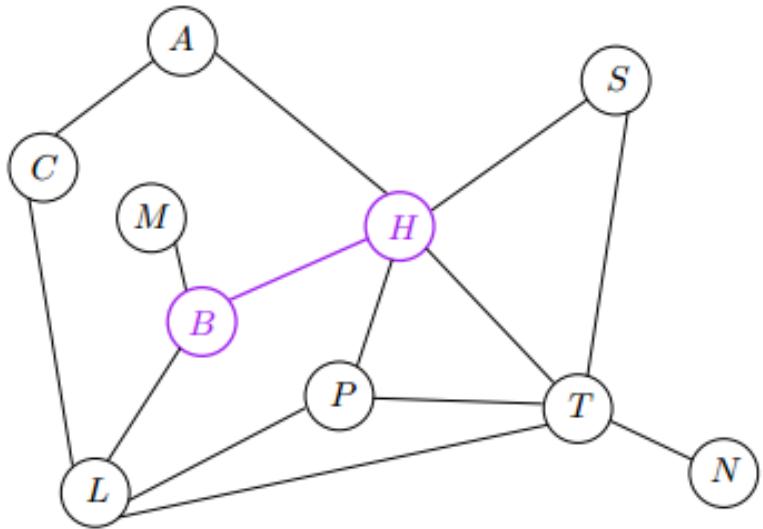
- Each edge has two **endpoints**
 - H and B are the endpoints of (H, B)
- An edge **joins** its endpoints
 - (H, B) joins H and B
- Two vertices are **adjacent** (or **neighbors**) if they are joined by an edge.
 - H and B are adjacent
 - H is a neighbor of B

Terminology



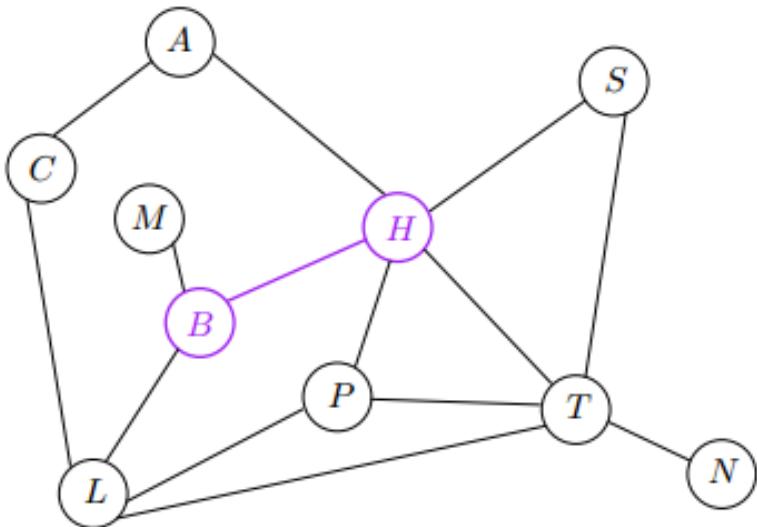
- Each edge has two **endpoints**
 - H and B are the endpoints of (H, B)
- An edge **joins** its endpoints
 - (H, B) joins H and B
- Two vertices are **adjacent** (or **neighbors**) if they are joined by an edge.
 - H and B are adjacent
 - H is a neighbor of B
- If vertex v is an endpoint of edge e, then the edge e is said to be **incident** on v. Also, the vertex v is said to be **incident** on e

Terminology



- Each edge has two **endpoints**
 - H and B are the endpoints of (H, B)
- An edge **joins** its endpoints
 - (H, B) joins H and B
- Two vertices are **adjacent** (or **neighbors**) if they are joined by an edge.
 - H and B are adjacent
 - H is a neighbor of B
- If vertex v is an endpoint of edge e, then the edge e is said to be **incident** on v. Also, the vertex v is said to be **incident** on e
 - (H, B) is incident on H and B

Terminology



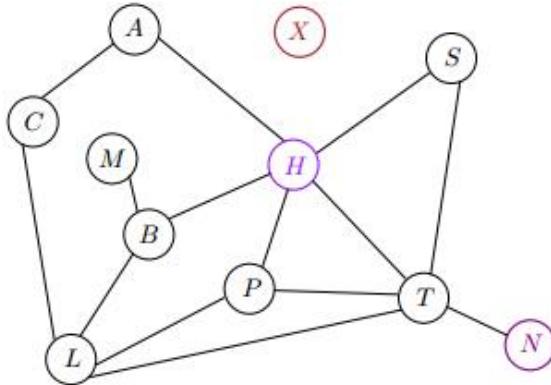
- Each edge has two **endpoints**
 - H and B are the endpoints of (H, B)
- An edge **joins** its endpoints
 - (H, B) joins H and B
- Two vertices are **adjacent** (or **neighbors**) if they are joined by an edge.
 - H and B are adjacent
 - H is a neighbor of B
- If vertex v is an endpoint of edge e, then the edge e is said to be **incident** on v. Also, the vertex v is said to be **incident** on e
 - (H, B) is incident on H and B
 - H and B are incident on (H, B)

The Degree of a Vertex

The **degree** of a vertex v ($\text{degree}(v)$) in a graph is the number of edges incident on it.

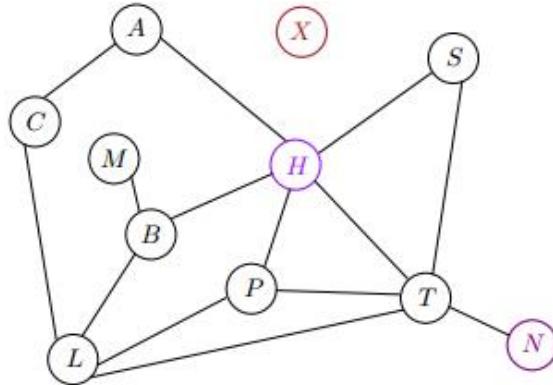
The Degree of a Vertex

The **degree** of a vertex v ($\text{degree}(v)$) in a graph is the number of edges incident on it.



The Degree of a Vertex

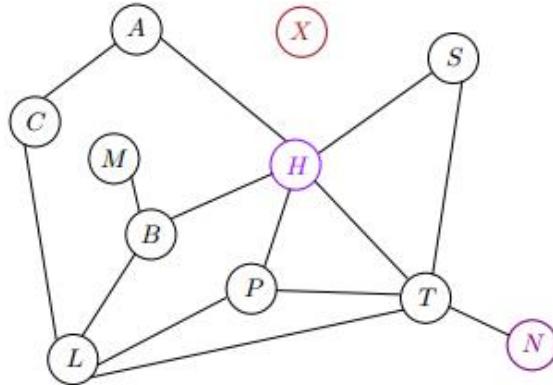
The **degree** of a vertex v ($\text{degree}(v)$) in a graph is the number of edges incident on it.



- Vertex H has degree 5

The Degree of a Vertex

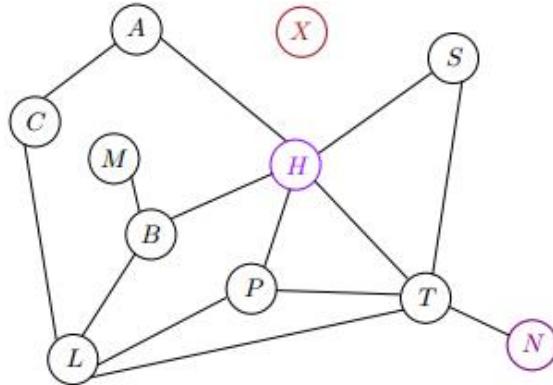
The **degree** of a vertex v ($\text{degree}(v)$) in a graph is the number of edges incident on it.



- Vertex H has degree 5
- Vertex N has degree 1

The Degree of a Vertex

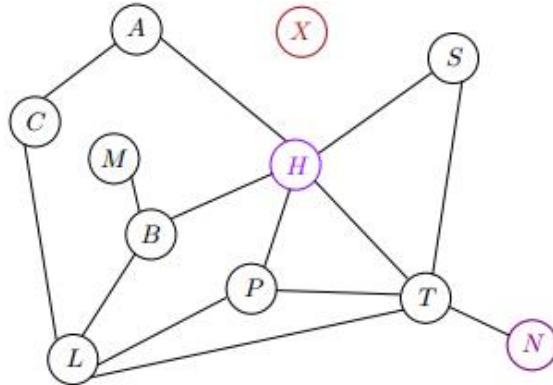
The **degree** of a vertex v ($\text{degree}(v)$) in a graph is the number of edges incident on it.



- Vertex H has degree 5
- Vertex N has degree 1
- Vertex X has degree 0

The Degree of a Vertex

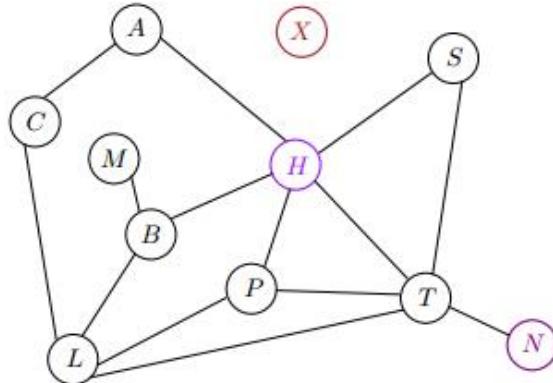
The **degree** of a vertex v ($\text{degree}(v)$) in a graph is the number of edges incident on it.



- Vertex H has degree 5
- Vertex N has degree 1
- Vertex X has degree 0
(It is called **isolated**)

The Degree of a Vertex

The **degree** of a vertex v ($\text{degree}(v)$) in a graph is the number of edges incident on it.



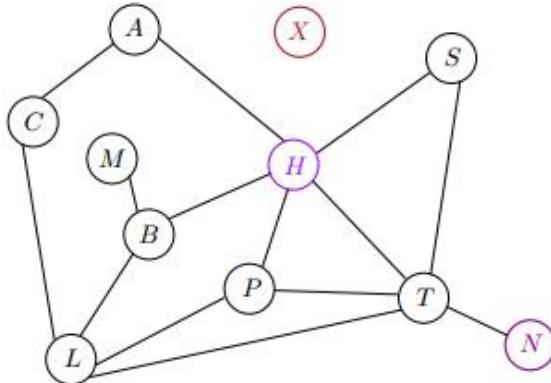
- Vertex H has degree 5
- Vertex N has degree 1
- Vertex X has degree 0
(It is called **isolated**)

Lemma

$$\sum_{v \in V} \text{degree}(v) = 2|E|.$$

The Degree of a Vertex

The **degree** of a vertex v ($\text{degree}(v)$) in a graph is the number of edges incident on it.



- Vertex H has degree 5
- Vertex N has degree 1
- Vertex X has degree 0
(It is called **isolated**)

Lemma

$$\sum_{v \in V} \text{degree}(v) = 2|E|.$$

Proof.

An edge $e = (u, v)$ in a graph contributes one to $\text{degree}(u)$ and contributes one to $\text{degree}(v)$.



Path

A **path** in a graph is a sequence $\langle v_0, v_1, v_2, \dots, v_k \rangle$ of vertices such that $(v_{i-1}, v_i) \in E$ for $i = 1, 2, \dots, k$

Path

A **path** in a graph is a sequence $\langle v_0, v_1, v_2, \dots, v_k \rangle$ of vertices such that $(v_{i-1}, v_i) \in E$ for $i = 1, 2, \dots, k$

- There is a path from v_0 to v_k

Path

A **path** in a graph is a sequence $\langle v_0, v_1, v_2, \dots, v_k \rangle$ of vertices such that $(v_{i-1}, v_i) \in E$ for $i = 1, 2, \dots, k$

- There is a path from v_0 to v_k
- **Length** of a path = # of edges on the path

Path

A **path** in a graph is a sequence $\langle v_0, v_1, v_2, \dots, v_k \rangle$ of vertices such that $(v_{i-1}, v_i) \in E$ for $i = 1, 2, \dots, k$

- There is a path from v_0 to v_k
- **Length** of a path = # of edges on the path
- The path **contains** the vertices v_0, v_1, \dots, v_k and the edges $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$

Path

A **path** in a graph is a sequence $\langle v_0, v_1, v_2, \dots, v_k \rangle$ of vertices such that $(v_{i-1}, v_i) \in E$ for $i = 1, 2, \dots, k$

- There is a path from v_0 to v_k
- **Length** of a path = # of edges on the path
- The path **contains** the vertices v_0, v_1, \dots, v_k and the edges $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$
- For any $0 \leq i \leq j \leq k$, $\langle v_i, v_{i+1}, \dots, v_j \rangle$ is its **subpath**

Path

A **path** in a graph is a sequence $\langle v_0, v_1, v_2, \dots, v_k \rangle$ of vertices such that $(v_{i-1}, v_i) \in E$ for $i = 1, 2, \dots, k$

- There is a path from v_0 to v_k
- **Length** of a path = # of edges on the path
- The path **contains** the vertices v_0, v_1, \dots, v_k and the edges $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$
- For any $0 \leq i \leq j \leq k$, $\langle v_i, v_{i+1}, \dots, v_j \rangle$ is its **subpath**
- If there is a path p from u to v , v is said to be **reachable** from u

Path

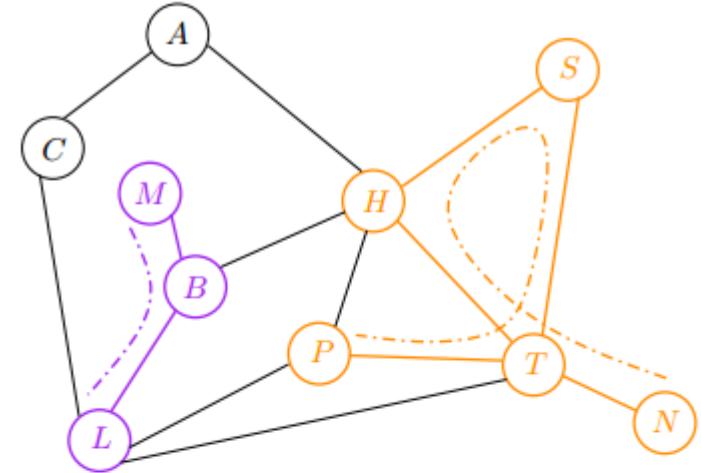
A **path** in a graph is a sequence $\langle v_0, v_1, v_2, \dots, v_k \rangle$ of vertices such that $(v_{i-1}, v_i) \in E$ for $i = 1, 2, \dots, k$

- There is a path from v_0 to v_k
- **Length** of a path = # of edges on the path
- The path **contains** the vertices v_0, v_1, \dots, v_k and the edges $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$
- For any $0 \leq i \leq j \leq k$, $\langle v_i, v_{i+1}, \dots, v_j \rangle$ is its **subpath**
- If there is a path p from u to v , v is said to be **reachable** from u
- A path is **simple** if all vertices in the path are distinct

Path

A **path** in a graph is a sequence $\langle v_0, v_1, v_2, \dots, v_k \rangle$ of vertices such that $(v_{i-1}, v_i) \in E$ for $i = 1, 2, \dots, k$

- There is a path from v_0 to v_k
- **Length** of a path = # of edges on the path
- The path **contains** the vertices v_0, v_1, \dots, v_k and the edges $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$
- For any $0 \leq i \leq j \leq k$, $\langle v_i, v_{i+1}, \dots, v_j \rangle$ is its **subpath**
- If there is a path p from u to v , v is said to be **reachable** from u
- A path is **simple** if all vertices in the path are distinct



- $\langle L, B, M \rangle$ is a path
 - length is 2
 - $\langle B, M \rangle$ is its subpath
 - M is reachable from L
 - a simple path
- $\langle N, T, H, S, T, P \rangle$ is a path
 - length is 5
 - $\langle T, H, S \rangle$ is its subpath
 - P is reachable from N
 - not a simple path

Cycle

A path $\langle v_0, v_1, v_2, \dots, v_k \rangle$ forms a **cycle** if $v_0 = v_k$ and all edges on the path are distinct

Cycle

A path $\langle v_0, v_1, v_2, \dots, v_k \rangle$ forms a **cycle** if $v_0 = v_k$ and all edges on the path are distinct

- A cycle is **simple** if v_1, v_2, \dots, v_k are distinct

Cycle

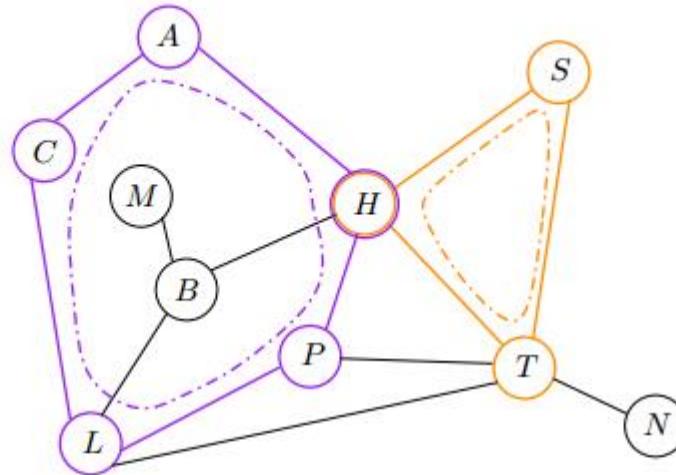
A path $\langle v_0, v_1, v_2, \dots, v_k \rangle$ forms a **cycle** if $v_0 = v_k$ and all edges on the path are distinct

- A cycle is **simple** if v_1, v_2, \dots, v_k are distinct
- A graph with no cycles is **acyclic**

Cycle

A path $\langle v_0, v_1, v_2, \dots, v_k \rangle$ forms a **cycle** if $v_0 = v_k$ and all edges on the path are distinct

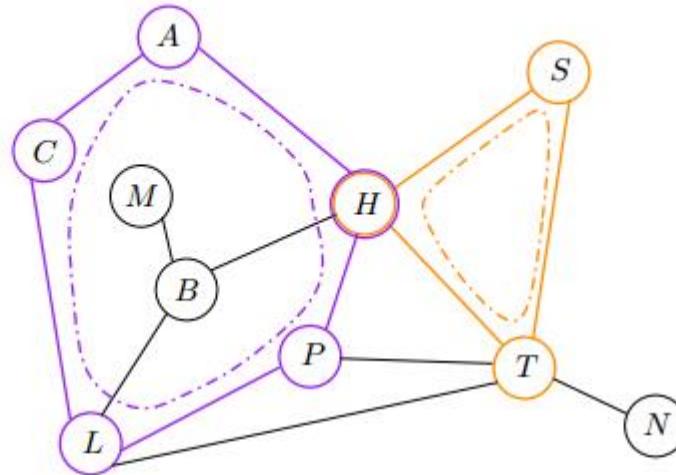
- A cycle is **simple** if v_1, v_2, \dots, v_k are distinct
- A graph with no cycles is **acyclic**



Cycle

A path $\langle v_0, v_1, v_2, \dots, v_k \rangle$ forms a **cycle** if $v_0 = v_k$ and all edges on the path are distinct

- A cycle is **simple** if v_1, v_2, \dots, v_k are distinct
- A graph with no cycles is **acyclic**

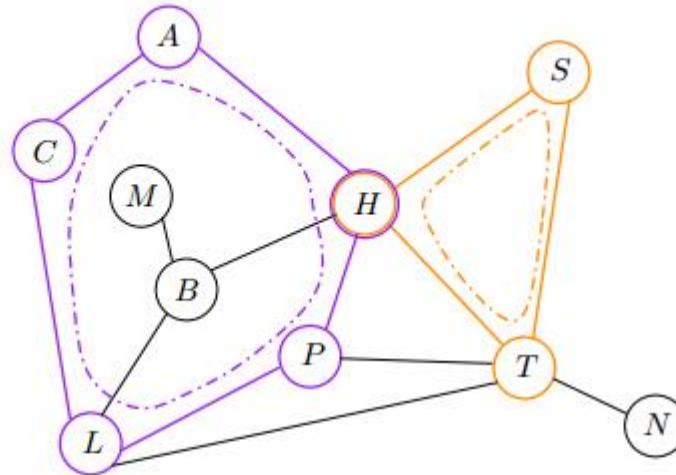


- $\langle T, S, H, T \rangle$ is a simple cycle

Cycle

A path $\langle v_0, v_1, v_2, \dots, v_k \rangle$ forms a **cycle** if $v_0 = v_k$ and all edges on the path are distinct

- A cycle is **simple** if v_1, v_2, \dots, v_k are distinct
- A graph with no cycles is **acyclic**



- $\langle T, S, H, T \rangle$ is a simple cycle
- $\langle A, C, L, P, H, A \rangle$ is a simple cycle

Connectivity

- Two vertices are **connected** if there is a path between them

Connectivity

- Two vertices are **connected** if there is a path between them
- A graph is **connected** if every pair of vertices is connected; otherwise, the graph is **disconnected**

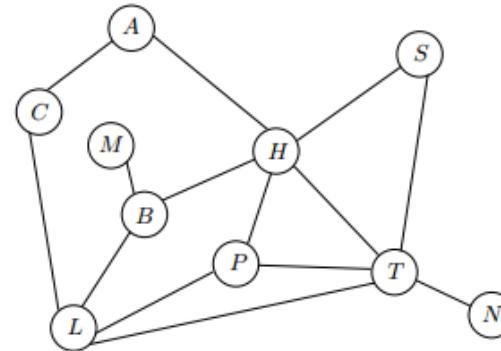
Connectivity

- Two vertices are **connected** if there is a path between them
- A graph is **connected** if every pair of vertices is connected; otherwise, the graph is **disconnected**
- The **connected components** of a graph are the equivalence classes of vertices under the "*is reachable from*" relation

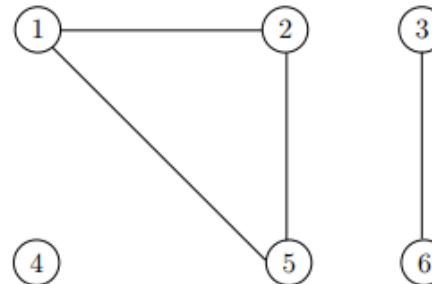
Connectivity

- Two vertices are **connected** if there is a path between them
- A graph is **connected** if every pair of vertices is connected; otherwise, the graph is **disconnected**
- The **connected components** of a graph are the equivalence classes of vertices under the "*is reachable from*" relation

- connected graph
- one connected component
 $\{A, B, C, H, L, M, N, P, S, T\}$



- disconnected graph
- 3 connected components
 - $\{1, 2, 5\}$
 - $\{3, 6\}$
 - $\{4\}$



Subgraph

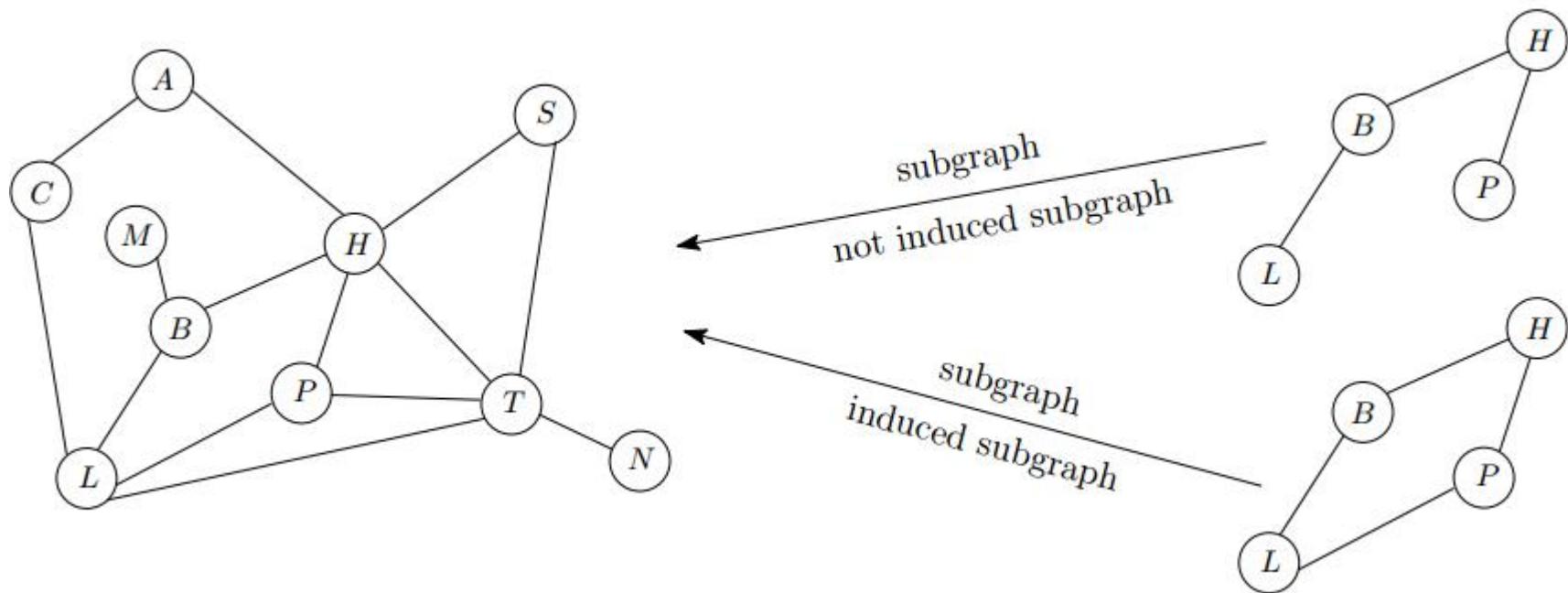
- Graph $G' = (V', E')$ is a **subgraph** of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$

Subgraph

- Graph $G' = (V', E')$ is a **subgraph** of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$
- G' is an **induced subgraph** of G if G' is a subgraph of G and every edge of G connecting vertices of G' is an edge of G' .

Subgraph

- Graph $G' = (V', E')$ is a **subgraph** of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$
- G' is an **induced subgraph** of G if G' is a subgraph of G and every edge of G connecting vertices of G' is an edge of G' .



Representations of Graphs: Adjacency List

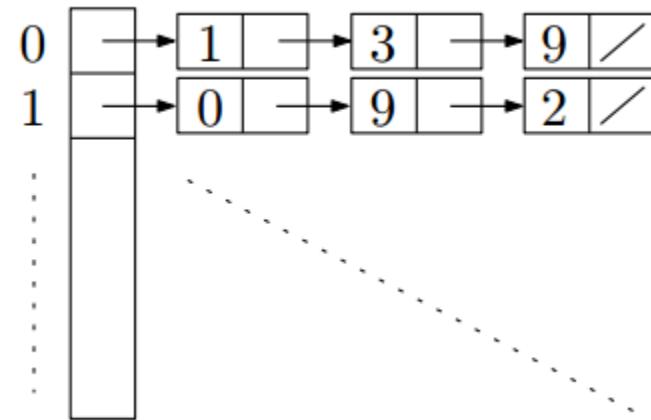
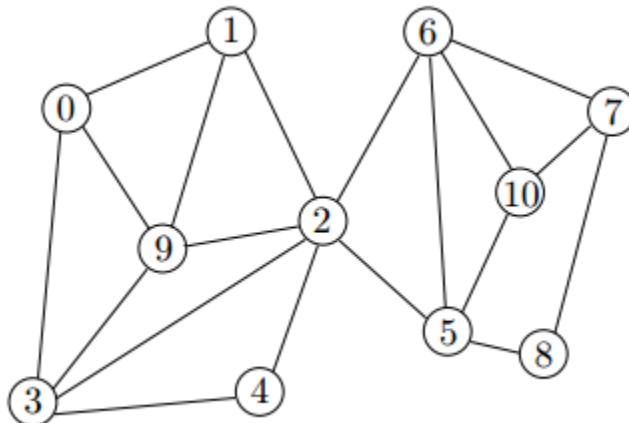
- V : set of vertices, E : set of edges. (As a convenience, we will sometimes use V to denote the number of vertices, and E to denote the number of edges.)
- **Adjacency list representation:** $O(V + E)$ storage $\text{Adj}[u]$ — linked list of all v such that $(u, v) \in E$.

Representations of Graphs: Adjacency List

- V : set of vertices, E : set of edges. (As a convenience, we will sometimes use V to denote the number of vertices, and E to denote the number of edges.)
- **Adjacency list representation:** $O(V + E)$ storage $\text{Adj}[u]$ — linked list of all v such that $(u, v) \in E$.
 - $\text{Adj}[0] = \{1, 3, 9\}; \text{Adj}[1] = \{0, 9, 2\}; \dots$

Representations of Graphs: Adjacency List

- V : set of vertices, E : set of edges. (As a convenience, we will sometimes use V to denote the number of vertices, and E to denote the number of edges.)
 - **Adjacency list representation:** $O(V + E)$ storage $\text{Adj}[u]$ – linked list of all v such that $(u, v) \in E$.
 - $\text{Adj}[0] = \{1, 3, 9\}; \text{Adj}[1] = \{0, 9, 2\}; \dots$
 - Can retrieve all the neighbors of u in $O(\text{degree}(u))$ time



Representations of Graphs: Adjacency Matrix

- **Adjacency matrix representation:** $O(V^2)$ storage

$A = [a_{ij}]$, $a_{ij} = 1$ if $(v_i, v_j) \in E$;

$a_{ij} = 0$ if $(v_i, v_j) \notin E$.

For an undirected graph, the adjacency matrix is always **symmetric**.

Representations of Graphs: Adjacency Matrix

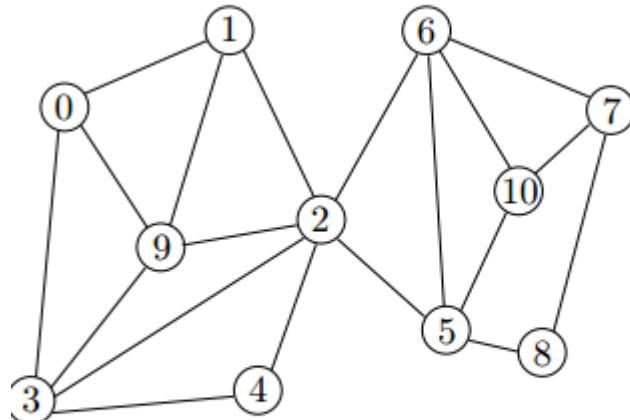
- **Adjacency matrix representation:** $O(V^2)$ storage

$A = [a_{ij}]$, $a_{ij} = 1$ if $(v_i, v_j) \in E$;

$a_{ij} = 0$ if $(v_i, v_j) \notin E$.

For an undirected graph, the adjacency matrix is always **symmetric**.

- Can check if u and v are connected in $O(1)$ time.



	0	1	2	3	4	5	6	7	8	9	10
0	0	1	0	1	0	0	0	0	0	1	0
1	1	0	1	0	0	0	0	0	0	1	0
2	0	1	0	1	1	1	1	0	0	1	0
3	1	0	1	0	1	0	0	0	0	1	0
4	0	0	1	1	0	0	0	0	0	0	0
5	0	0	1	0	0	0	1	0	1	0	1
6	0	0	1	0	0	1	0	1	0	0	1
7	0	0	0	0	0	0	1	0	1	0	1
8	0	0	0	0	0	1	0	1	0	0	0
9	1	1	1	1	0	0	0	0	0	0	0
10	0	0	0	0	0	1	1	1	0	0	0

Outline

- Introduction to Part IV
- Review of Basic Graph Search Algorithms
 - Basic Concepts
 - The Breadth-First Search (BFS) Algorithm
 - The Depth-First Search (DFS) Algorithm
- Topological Sort
 - The Topological Sort Algorithm
 - Analysis of the Topological Sort Algorithm
- Strongly Connected Components
 - The Algorithm of Finding SCCs
 - Analysis of the Algorithm

The Breadth-First Search (BFS) Algorithm

What does Breadth-First Search (BFS) do?

The Breadth-First Search (BFS) Algorithm

What does Breadth-First Search (BFS) do?

- Traverse all vertices in graph, and thereby
- Reveal properties of the graph

The Breadth-First Search (BFS) Algorithm

What does Breadth-First Search (BFS) do?

- Traverse all vertices in graph, and thereby
- Reveal properties of the graph

Three arrays are used to keep information gathered during traversal

The Breadth-First Search (BFS) Algorithm

What does Breadth-First Search (BFS) do?

- Traverse all vertices in graph, and thereby
- Reveal properties of the graph

Three arrays are used to keep information gathered during traversal

- `color[u]`: the **color** of each vertex visited

The Breadth-First Search (BFS) Algorithm

What does Breadth-First Search (BFS) do?

- Traverse all vertices in graph, and thereby
- Reveal properties of the graph

Three arrays are used to keep information gathered during traversal

- `color[u]`: the **color** of each vertex visited
 - **WHITE**: undiscovered

The Breadth-First Search (BFS) Algorithm

What does Breadth-First Search (BFS) do?

- Traverse all vertices in graph, and thereby
- Reveal properties of the graph

Three arrays are used to keep information gathered during traversal

- `color[u]`: the **color** of each vertex visited
 - **WHITE**: undiscovered
 - **GRAY**: discovered but not finished processing

The Breadth-First Search (BFS) Algorithm

What does Breadth-First Search (BFS) do?

- Traverse all vertices in graph, and thereby
- Reveal properties of the graph

Three arrays are used to keep information gathered during traversal

- **color[u]**: the **color** of each vertex visited
 - **WHITE**: undiscovered
 - **GRAY**: discovered but not finished processing
 - **BLACK**: finished processing

The Breadth-First Search (BFS) Algorithm

What does Breadth-First Search (BFS) do?

- Traverse all vertices in graph, and thereby
- Reveal properties of the graph

Three arrays are used to keep information gathered during traversal

- $\text{color}[u]$: the **color** of each vertex visited
 - **WHITE**: undiscovered
 - **GRAY**: discovered but not finished processing
 - **BLACK**: finished processing
- $\text{pred}[u]$: the **predecessor** pointer
 - pointing back to the vertex from which u was discovered

The Breadth-First Search (BFS) Algorithm

What does Breadth-First Search (BFS) do?

- Traverse all vertices in graph, and thereby
- Reveal properties of the graph

Three arrays are used to keep information gathered during traversal

- $\text{color}[u]$: the **color** of each vertex visited
 - **WHITE**: undiscovered
 - **GRAY**: discovered but not finished processing
 - **BLACK**: finished processing
- $\text{pred}[u]$: the **predecessor** pointer
 - pointing back to the vertex from which u was discovered
- $d[u]$: the **distance** from the source to vertex u

BFS Algorithm

$\text{BFS}(G)$

Input: A graph G

Output: None

//Initialize

for u in V **do**

$\text{color}[u] \leftarrow \text{WHITE}$; //undiscovered

$\text{pred}[u] \leftarrow \text{NULL}$; //no predecessor

end

BFS Algorithm

$\text{BFS}(G)$

Input: A graph G

Output: None

//Initialize

for u in V **do**

$\text{color}[u] \leftarrow \text{WHITE}$; //undiscovered

$\text{pred}[u] \leftarrow \text{NULL}$; //no predecessor

end

for u in V **do**

 //start a new tree

if $\text{color}[u]$ is equal to WHITE **then**

 |

end

end

BFS Algorithm

$\text{BFS}(G)$

Input: A graph G

Output: None

//Initialize

for u in V **do**

$\text{color}[u] \leftarrow \text{WHITE}$; //undiscovered

$\text{pred}[u] \leftarrow \text{NULL}$; //no predecessor

end

for u in V **do**

 //start a new tree

if $\text{color}[u]$ is equal to WHITE **then**

 | $\text{BFSVisit}(u);$

end

end

BFS Algorithm

BFSVisit(s)

Input: A vertex s

Output: None

$color[s] \leftarrow \text{GRAY}, d[s] \leftarrow 0;$

BFS Algorithm

BFSVisit(s)

Input: A vertex s

Output: None

$color[s] \leftarrow \text{GRAY}, d[s] \leftarrow 0;$

$Q \leftarrow \emptyset, \text{Enqueue}(Q, s);$

while $Q \neq \emptyset$ **do**

end

BFS Algorithm

BFSVisit(s)

Input: A vertex s

Output: None

$color[s] \leftarrow \text{GRAY}, d[s] \leftarrow 0;$

$Q \leftarrow \emptyset, \text{Enqueue}(Q, s);$

while $Q \neq \emptyset$ **do**

$u \leftarrow \text{Dequeue}(Q);$

end

BFS Algorithm

BFSVisit(s)

Input: A vertex s

Output: None

$color[s] \leftarrow \text{GRAY}, d[s] \leftarrow 0;$

$Q \leftarrow \emptyset, \text{Enqueue}(Q, s);$

while $Q \neq \emptyset$ **do**

$u \leftarrow \text{Dequeue}(Q);$

for $v \in Adj[u]$ **do**

if $color[v] \leftarrow \text{WHITE}$ **then**

$color[v] \leftarrow \text{GRAY};$

$d[v] \leftarrow d[u] + 1;$

$pred[v] \leftarrow u;$

$\text{Enqueue}(Q, v);$

end

BFS Algorithm

BFSVisit(s)

Input: A vertex s

Output: None

$color[s] \leftarrow \text{GRAY}, d[s] \leftarrow 0;$

$Q \leftarrow \emptyset, \text{Enqueue}(Q, s);$

while $Q \neq \emptyset$ **do**

$u \leftarrow \text{Dequeue}(Q);$

for $v \in Adj[u]$ **do**

if $color[v] \leftarrow \text{WHITE}$ **then**

$color[v] \leftarrow \text{GRAY};$

$d[v] \leftarrow d[u] + 1;$

$pred[v] \leftarrow u;$

$\text{Enqueue}(Q, v);$

end

end

$color[u] \leftarrow \text{BLACK};$

end

BFS Algorithm

BFSVisit(s)

Input: A vertex s

Output: None

$color[s] \leftarrow \text{GRAY}, d[s] \leftarrow 0;$

$Q \leftarrow \emptyset, \text{Enqueue}(Q, s);$

while $Q \neq \emptyset$ **do**

$u \leftarrow \text{Dequeue}(Q);$

for $v \in \text{Adj}[u]$ **do**

if $color[v] \leftarrow \text{WHITE}$ **then**

$color[v] \leftarrow \text{GRAY};$

$d[v] \leftarrow d[u] + 1;$

$\text{pred}[v] \leftarrow u;$

$\text{Enqueue}(Q, v);$

end

end

$color[u] \leftarrow \text{BLACK};$

end

Question

Which graph representation shall we use?

BFS Example

color

W	W	W	W	W	W	W	W
---	---	---	---	---	---	---	---

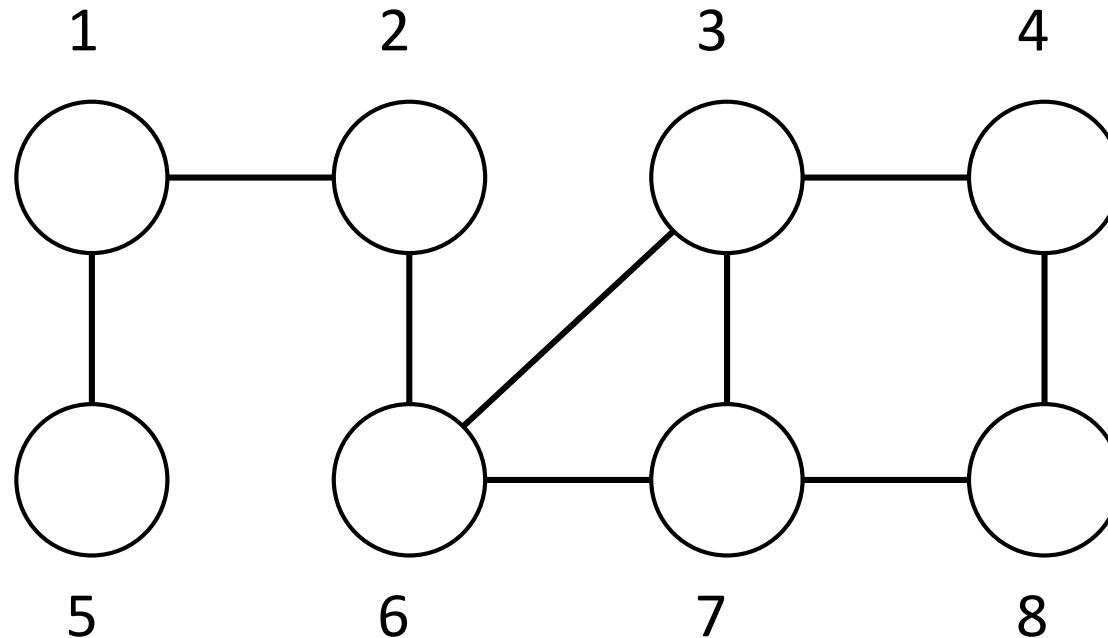
d

∞							
----------	----------	----------	----------	----------	----------	----------	----------

pred

N	N	N	N	N	N	N	N
---	---	---	---	---	---	---	---

Q



BFS Example

color

w	w	w	w	w	w	w	w
---	---	---	---	---	---	---	---

d

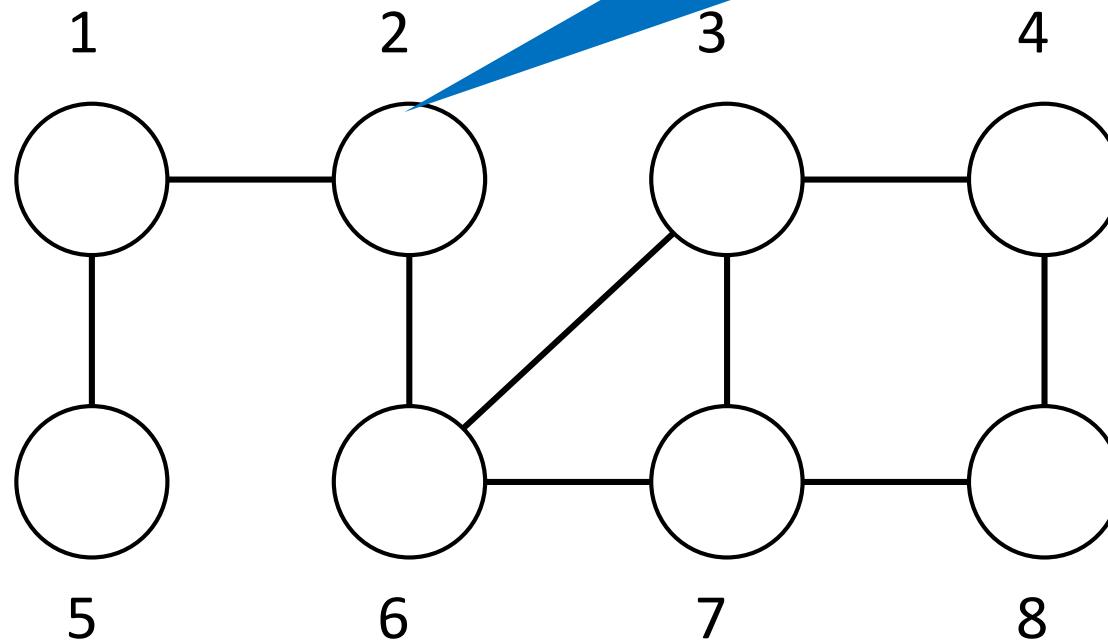
∞							
----------	----------	----------	----------	----------	----------	----------	----------

pred

N	N	N	N	N	N	N	N
---	---	---	---	---	---	---	---

Q

Start from this vertex



BFS Example

color

W	G	W	W	W	W	W	W
---	---	---	---	---	---	---	---

d

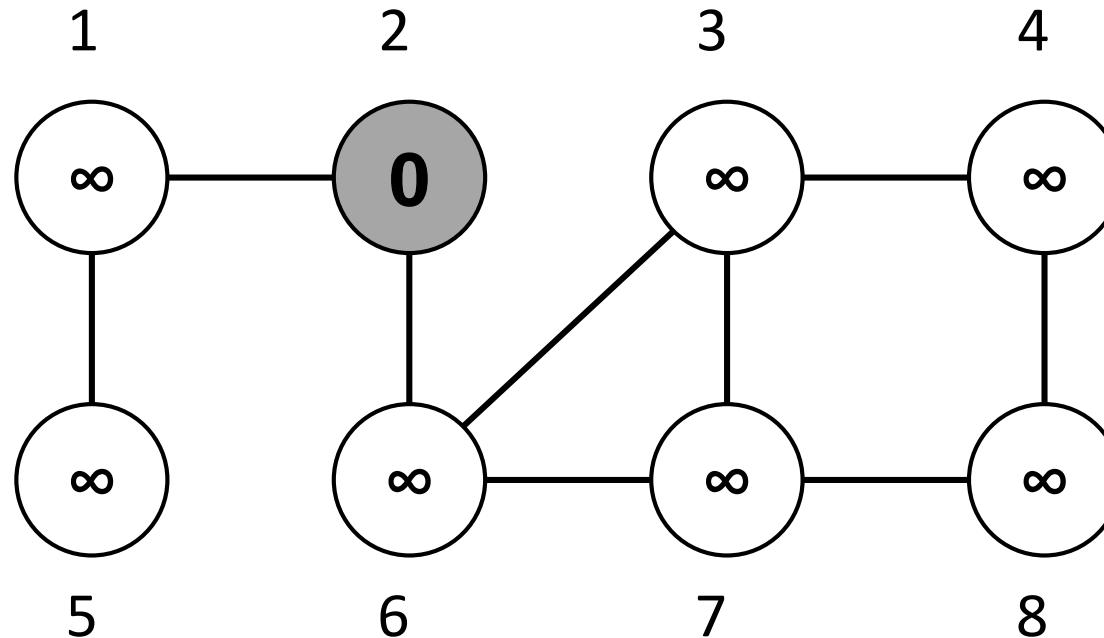
∞	0	∞	∞	∞	∞	∞	∞
----------	---	----------	----------	----------	----------	----------	----------

pred

N	N	N	N	N	N	N	N
---	---	---	---	---	---	---	---

Q

2



BFS Example

color

W	G	W	W	W	W	W	W
---	---	---	---	---	---	---	---

d

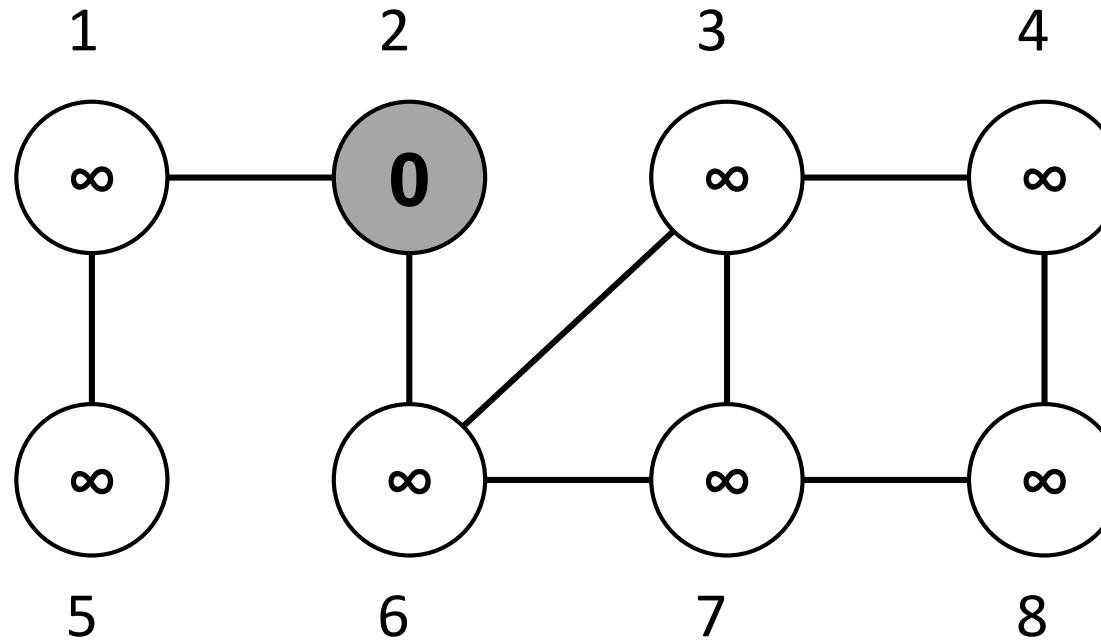
∞	0	∞	∞	∞	∞	∞	∞
----------	---	----------	----------	----------	----------	----------	----------

pred

N	N	N	N	N	N	N	N
---	---	---	---	---	---	---	---

Q

2



BFS Example

color

W	G	W	W	W	W	W	W
---	---	---	---	---	---	---	---

d

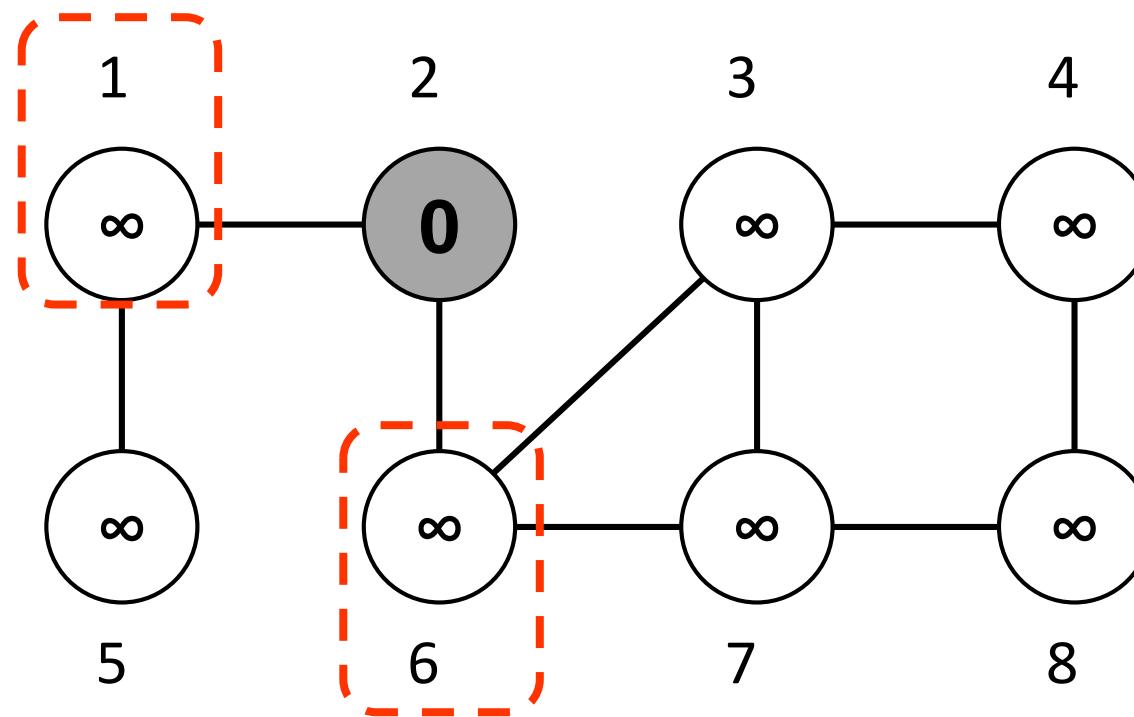
∞	0	∞	∞	∞	∞	∞	∞
----------	---	----------	----------	----------	----------	----------	----------

pred

N	N	N	N	N	N	N	N
---	---	---	---	---	---	---	---

Q

2



BFS Example

color

G	G	W	W	W	G	W	W
---	---	---	---	---	---	---	---

d

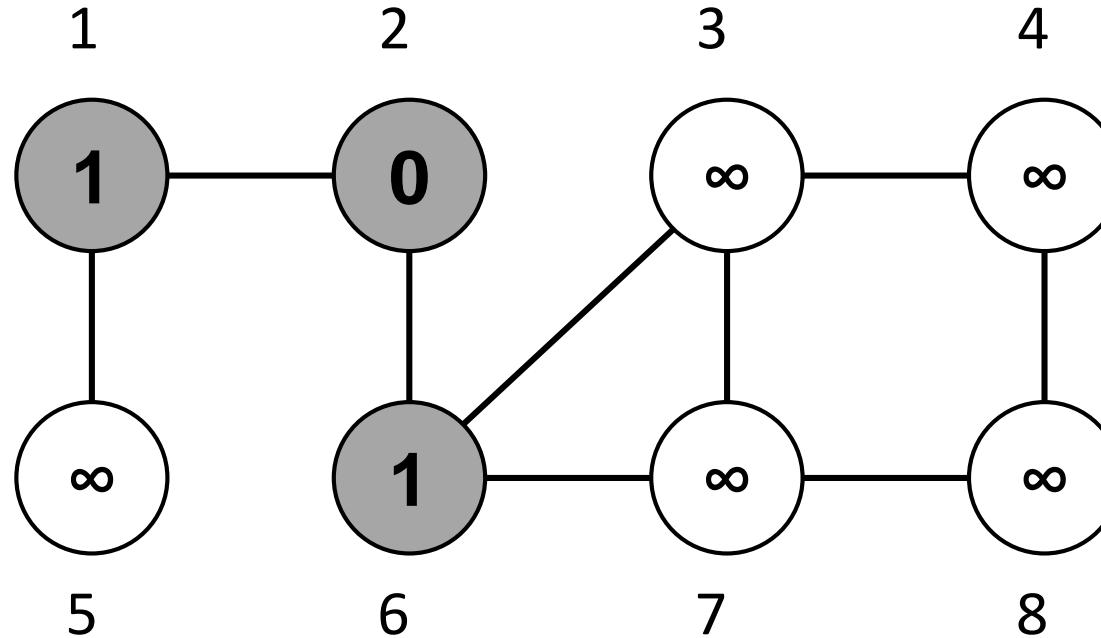
1	0	∞	∞	∞	1	∞	∞
---	---	----------	----------	----------	---	----------	----------

pred

2	N	N	N	N	2	N	N
---	---	---	---	---	---	---	---

Q

2	1	6
---	---	---



BFS Example

color

G	G	W	W	W	G	W	W
---	---	---	---	---	---	---	---

d

1	0	∞	∞	∞	1	∞	∞
---	---	----------	----------	----------	---	----------	----------

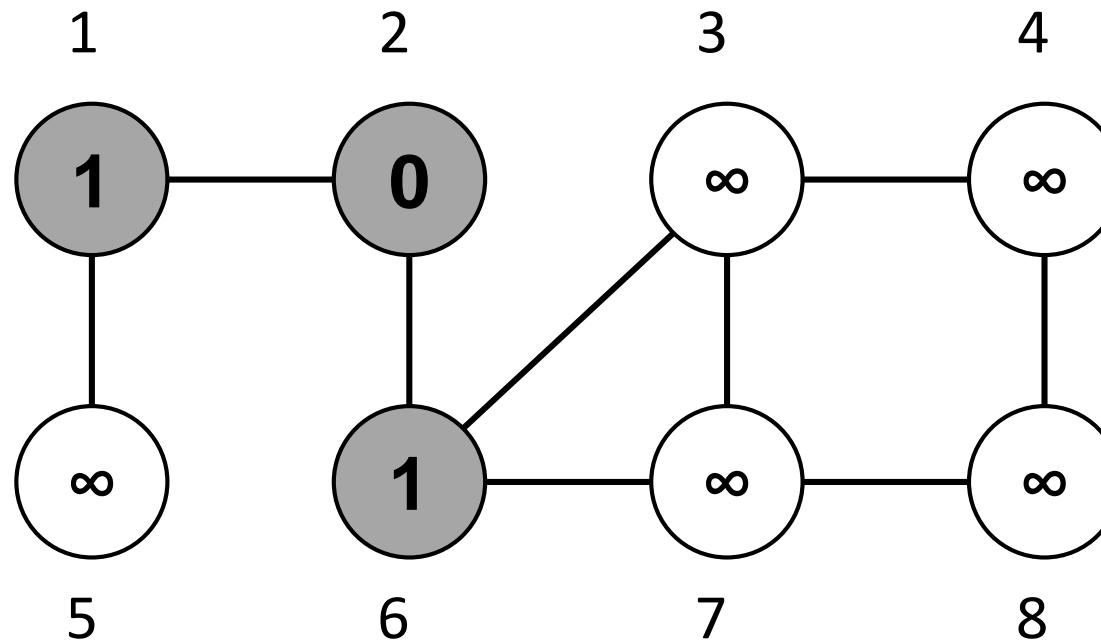
pred

2	N	N	N	N	2	N	N
---	---	---	---	---	---	---	---

Q

2	1	6
---	---	---

All adjacencies
of 2 have been
traversed.



BFS Example

color

G	B	W	W	W	G	W	W
---	---	---	---	---	---	---	---

d

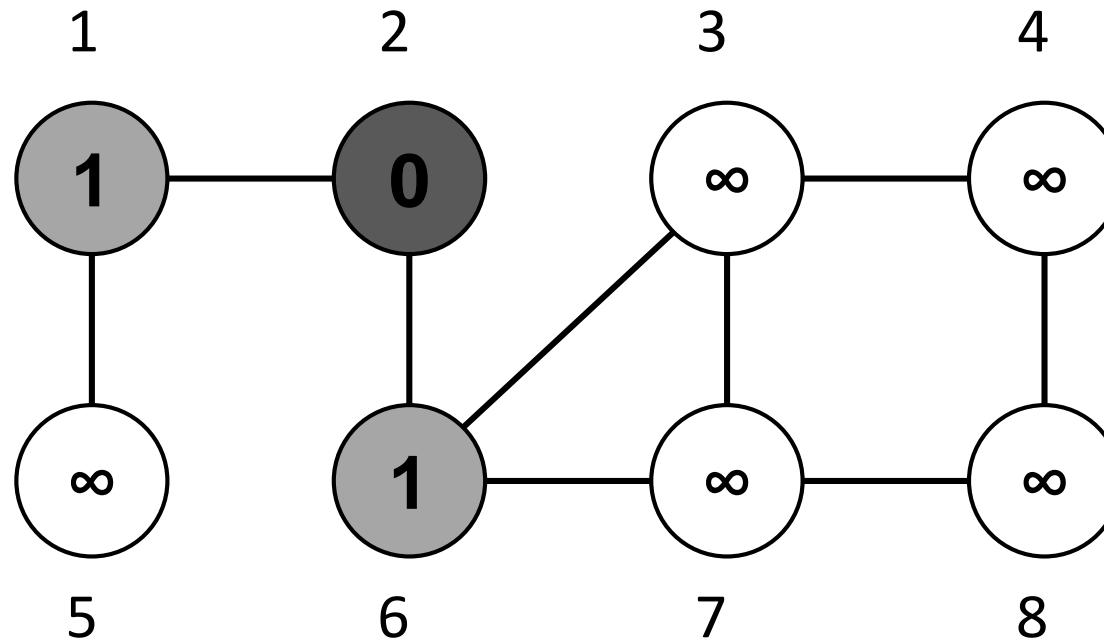
1	0	∞	∞	∞	1	∞	∞
---	---	----------	----------	----------	---	----------	----------

pred

2	N	N	N	N	N	2	N	N
---	---	---	---	---	---	---	---	---

Q

2	1	6
---	---	---



BFS Example

color

G	B	W	W	W	G	W	W
---	---	---	---	---	---	---	---

d

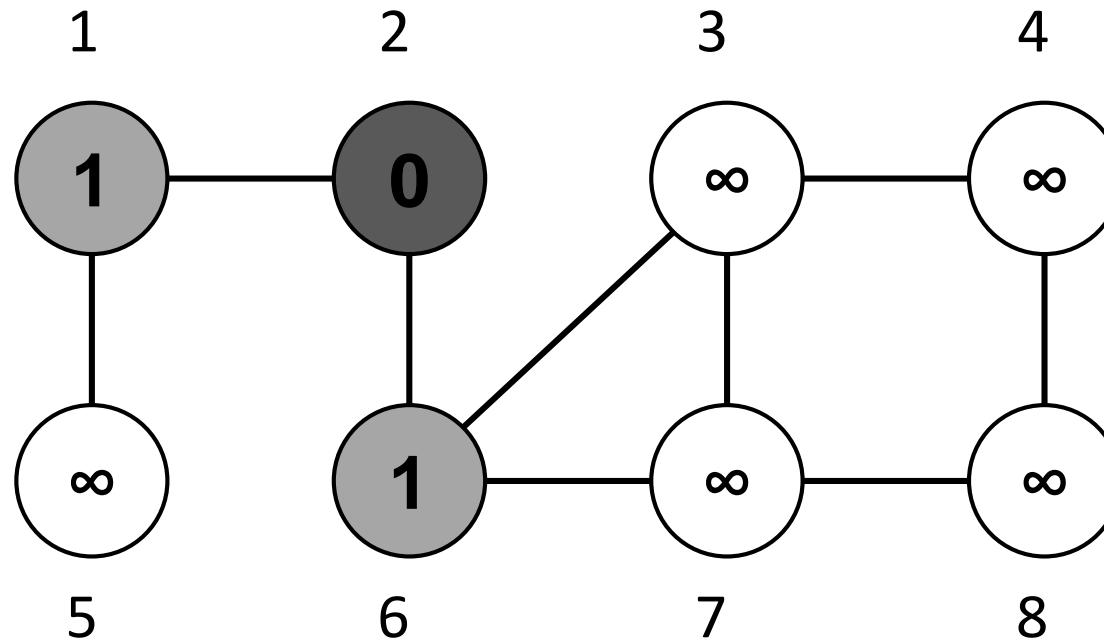
1	0	∞	∞	∞	1	∞	∞
---	---	----------	----------	----------	---	----------	----------

pred

2	N	N	N	N	N	2	N	N
---	---	---	---	---	---	---	---	---

Q

2	1	6
---	---	---



BFS Example

color

G	B	W	W	W	G	W	W
---	---	---	---	---	---	---	---

d

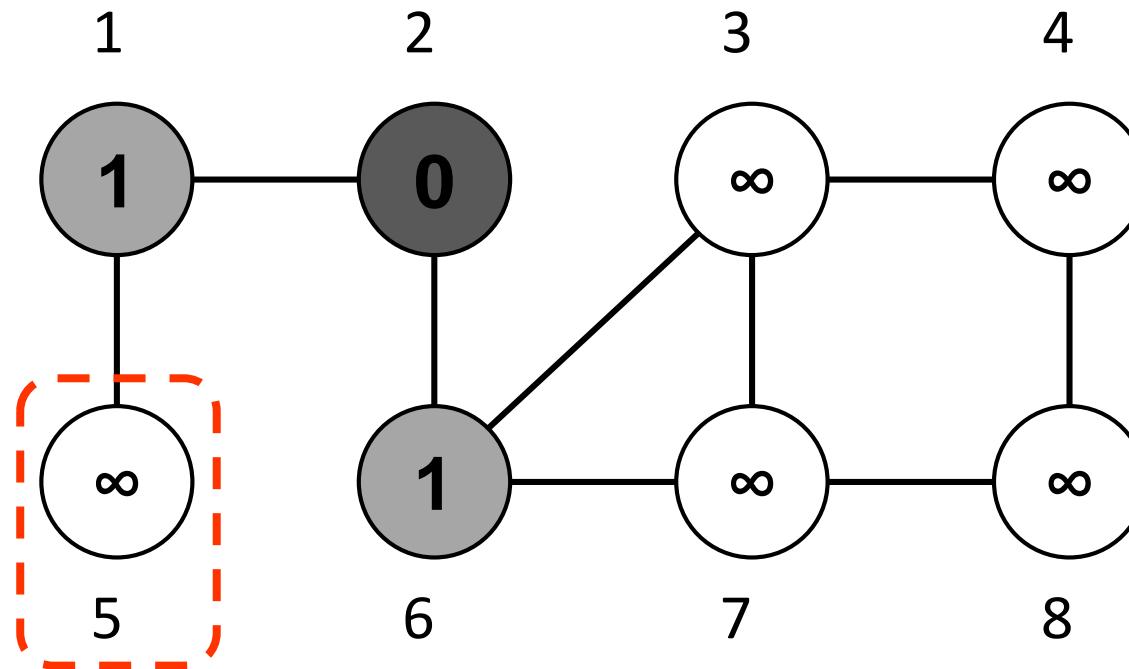
1	0	∞	∞	∞	1	∞	∞
---	---	----------	----------	----------	---	----------	----------

pred

2	N	N	N	N	N	2	N	N
---	---	---	---	---	---	---	---	---

Q

2	1	6
---	---	---



BFS Example

color

G	B	W	W	G	G	W	W
---	---	---	---	---	---	---	---

d

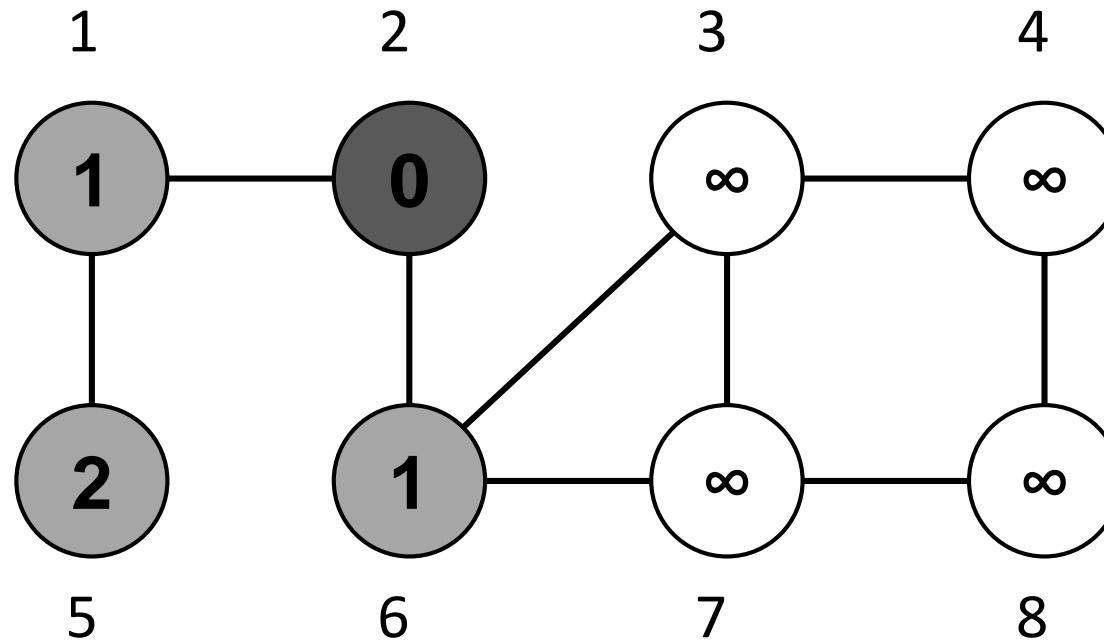
1	0	∞	∞	2	1	∞	∞
---	---	----------	----------	---	---	----------	----------

pred

2	N	N	N	1	2	N	N
---	---	---	---	---	---	---	---

Q

2	1	6	5
---	---	---	---



BFS Example

color

B	B	W	W	G	G	W	W
---	---	---	---	---	---	---	---

d

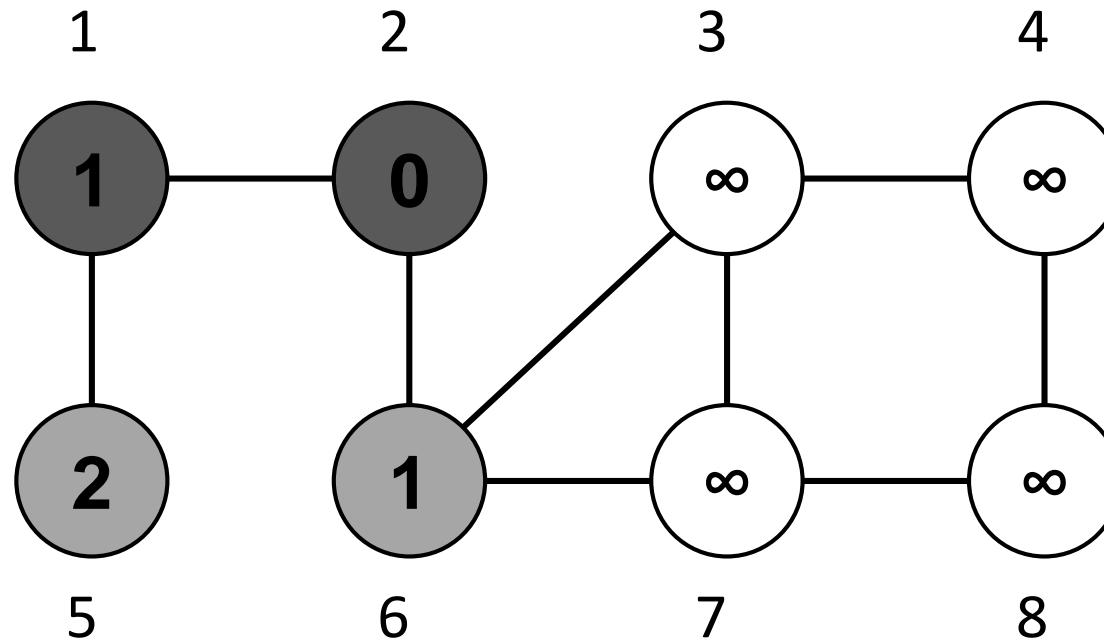
1	0	∞	∞	2	1	∞	∞
---	---	----------	----------	---	---	----------	----------

pred

2	N	N	N	1	2	N	N
---	---	---	---	---	---	---	---

Q

2	1	6	5
---	---	---	---



BFS Example

color

B	B	W	W	G	G	W	W
---	---	---	---	---	---	---	---

d

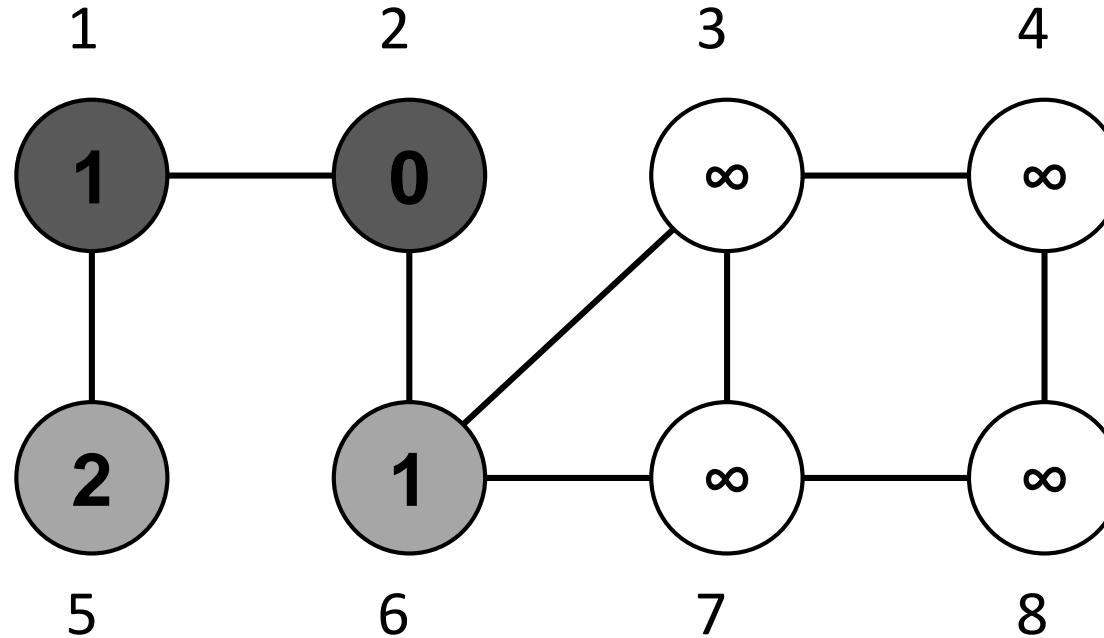
1	0	∞	∞	2	1	∞	∞
---	---	----------	----------	---	---	----------	----------

pred

2	N	N	N	1	2	N	N
---	---	---	---	---	---	---	---

Q

2	1	6	5
---	---	---	---



BFS Example

color

B	B	W	W	G	G	W	W
---	---	---	---	---	---	---	---

d

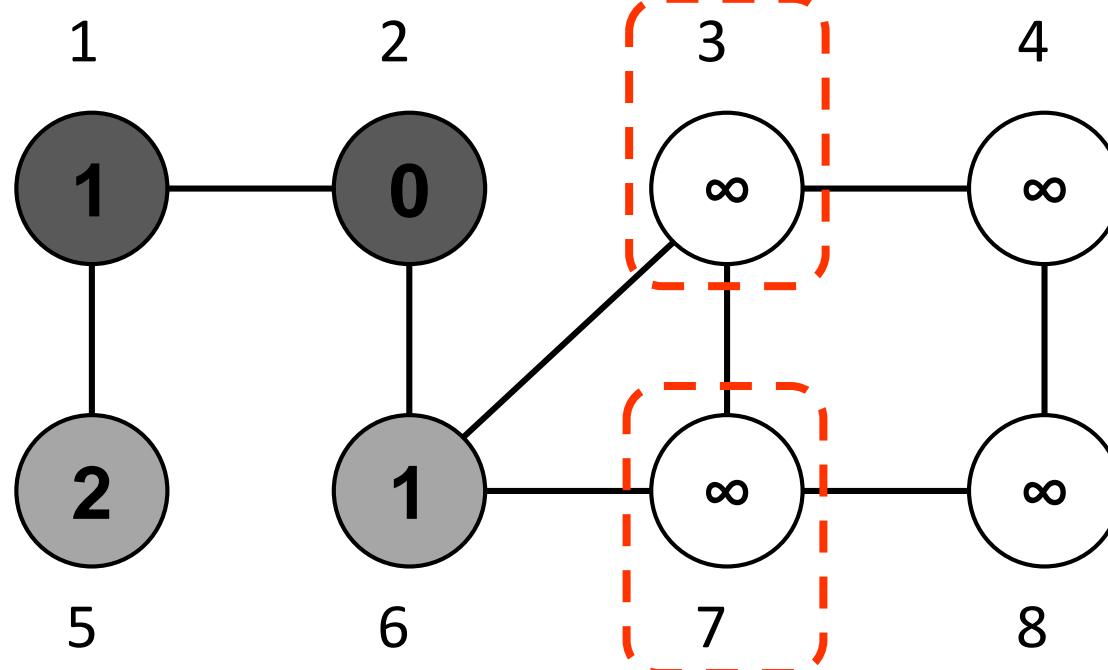
1	0	∞	∞	2	1	∞	∞
---	---	----------	----------	---	---	----------	----------

pred

2	N	N	N	1	2	N	N
---	---	---	---	---	---	---	---

Q

2	1	6	5
---	---	---	---



BFS Example

color

B	B	G	W	G	G	G	W
---	---	---	---	---	---	---	---

d

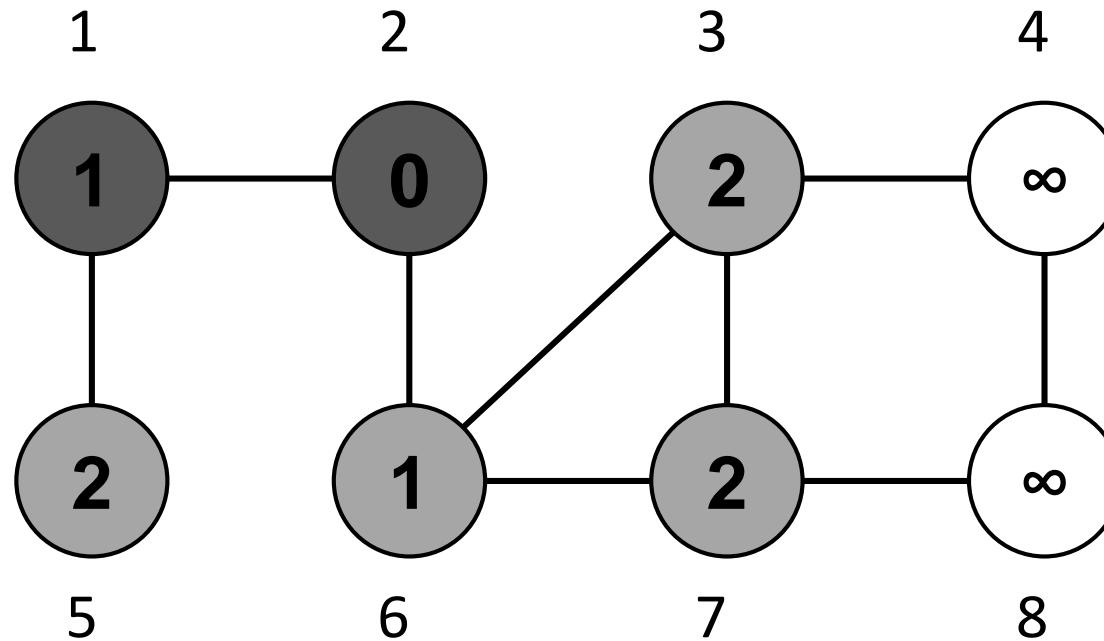
1	0	2	∞	2	1	2	∞
---	---	---	----------	---	---	---	----------

pred

2	N	6	N	1	2	6	N
---	---	---	---	---	---	---	---

Q

2	1	6	5	3	7
---	---	---	---	---	---



BFS Example

color

B	B	G	W	G	B	G	W
---	---	---	---	---	---	---	---

d

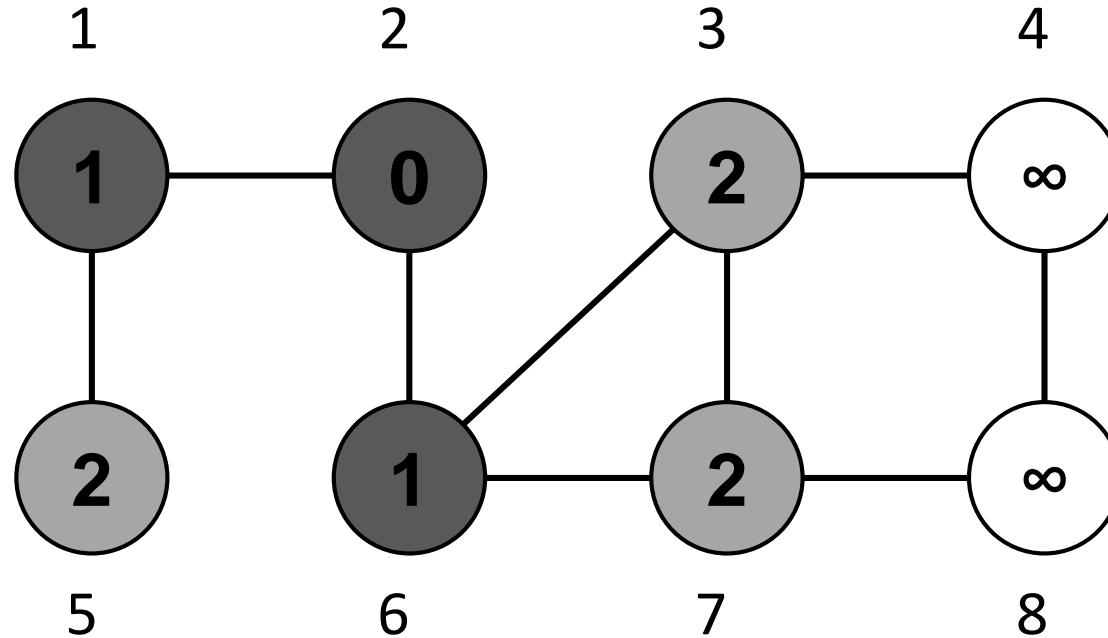
1	0	2	∞	2	1	2	∞
---	---	---	----------	---	---	---	----------

pred

2	N	6	N	1	2	6	N
---	---	---	---	---	---	---	---

Q

2	1	6	5	3	7
---	---	---	---	---	---



BFS Example

color

B	B	G	W	G	B	G	W
---	---	---	---	---	---	---	---

d

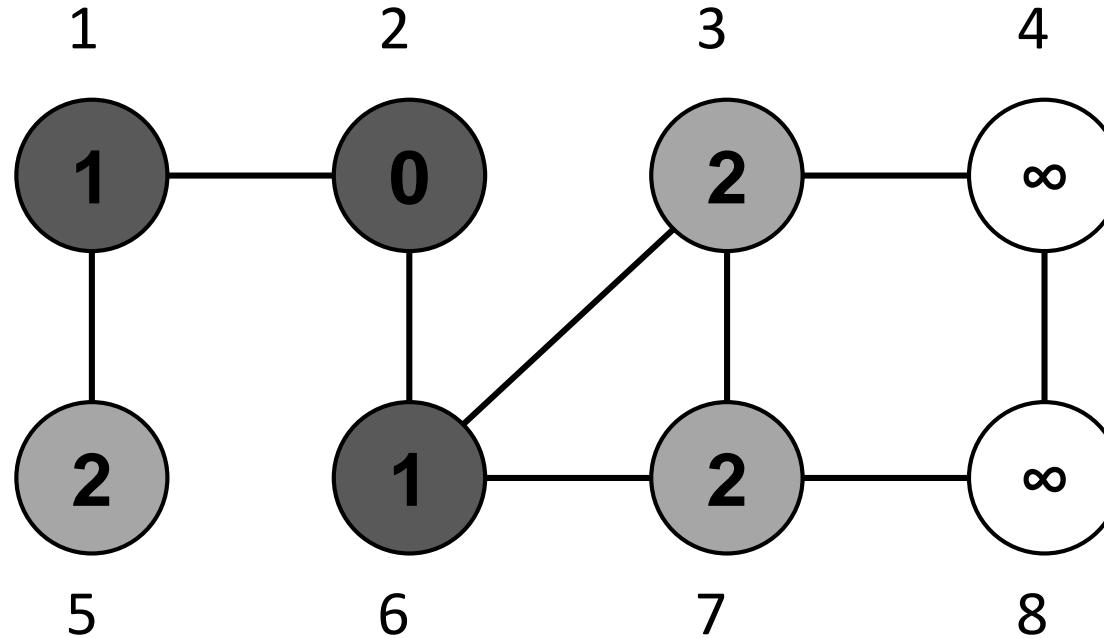
1	0	2	∞	2	1	2	∞
---	---	---	----------	---	---	---	----------

pred

2	N	6	N	1	2	6	N
---	---	---	---	---	---	---	---

Q

2	1	6	5	3	7
---	---	---	---	---	---



BFS Example

color

B	B	G	W	G	B	G	W
---	---	---	---	---	---	---	---

d

1	0	2	∞	2	1	2	∞
---	---	---	----------	---	---	---	----------

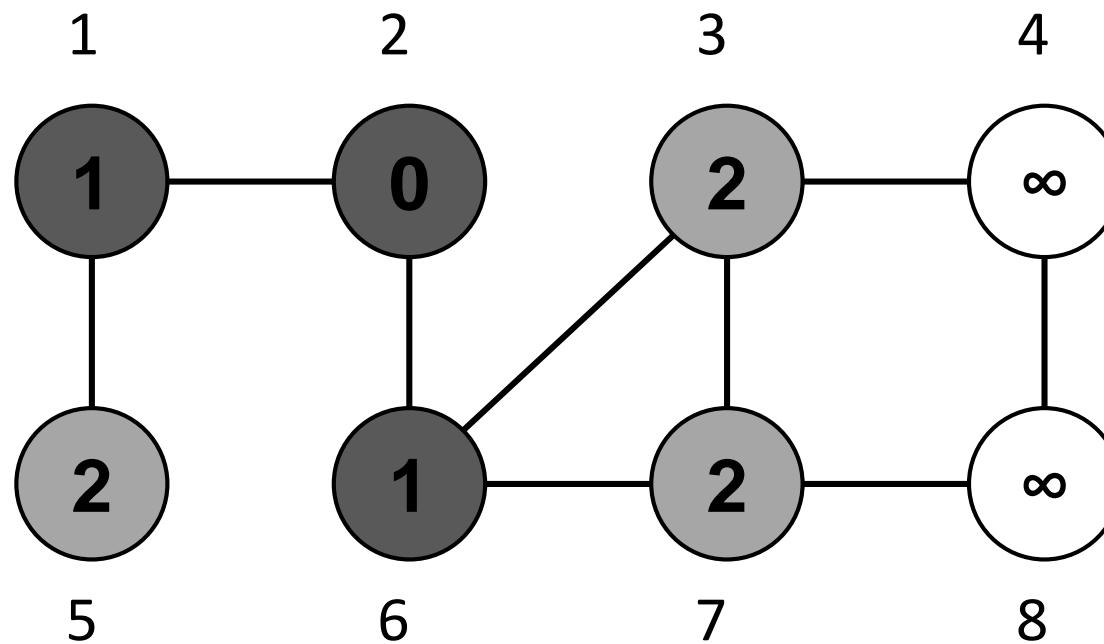
pred

2	N	6	N	1	2	6	N
---	---	---	---	---	---	---	---

Q

2	1	6	5	3	7
---	---	---	---	---	---

None of its
adjacencies
are WHITE.



BFS Example

color

B	B	G	W	B	B	G	W
---	---	---	---	---	---	---	---

d

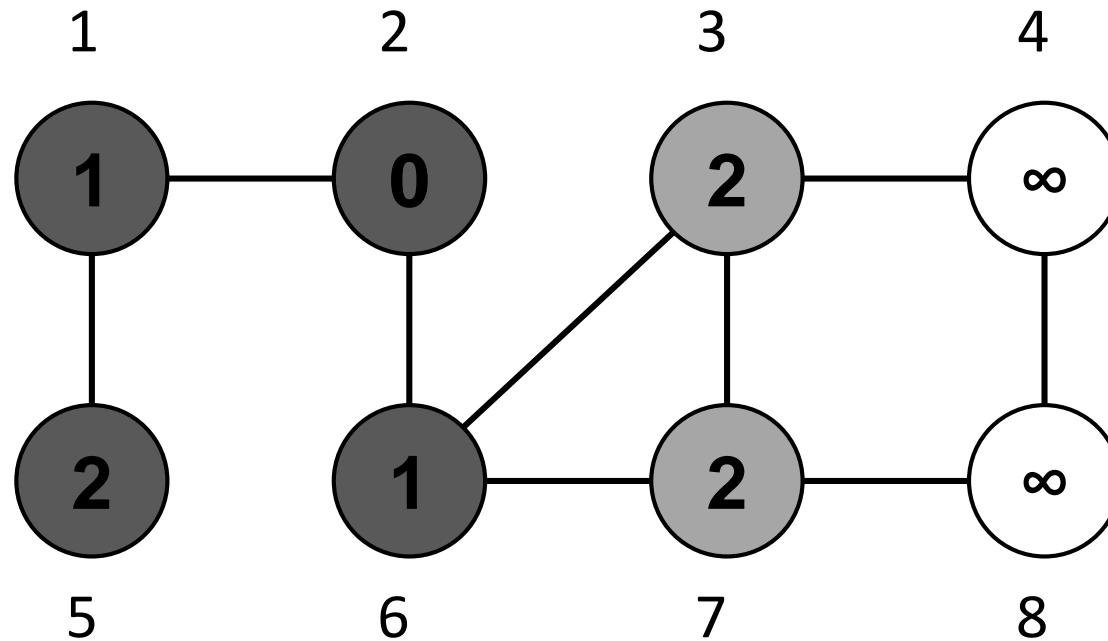
1	0	2	∞	2	1	2	∞
---	---	---	----------	---	---	---	----------

pred

2	N	6	N	1	2	6	N
---	---	---	---	---	---	---	---

Q

2	1	6	5	3	7
---	---	---	---	---	---



BFS Example

color

B	B	G	W	B	B	G	W
---	---	---	---	---	---	---	---

d

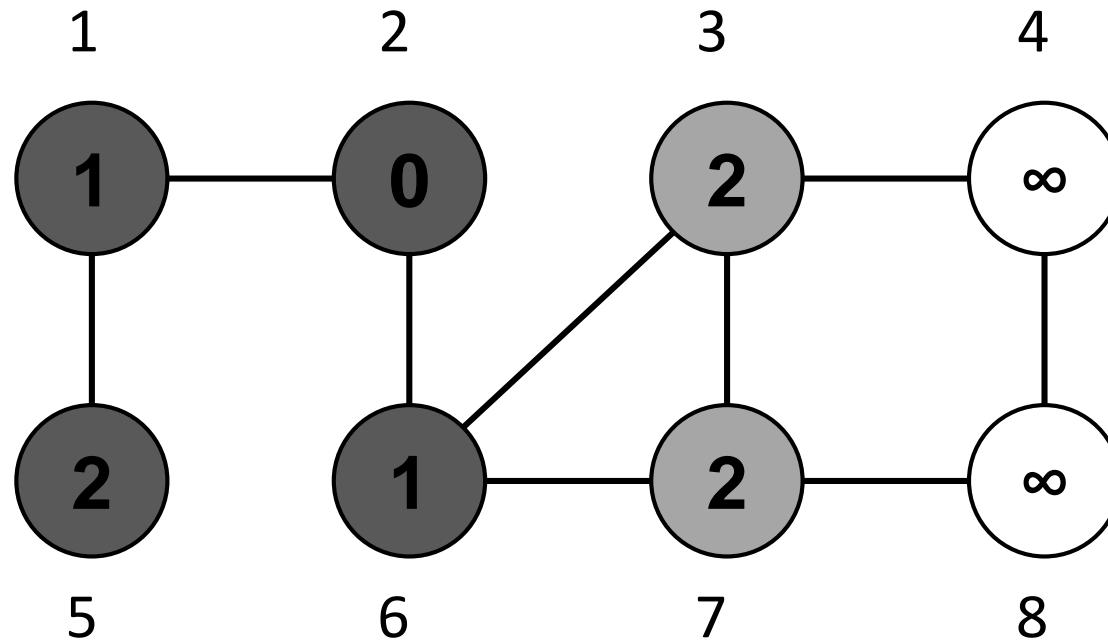
1	0	2	∞	2	1	2	∞
---	---	---	----------	---	---	---	----------

pred

2	N	6	N	1	2	6	N
---	---	---	---	---	---	---	---

Q

2	1	6	5	3	7
---	---	---	---	---	---



BFS Example

color

B	B	G	W	B	B	G	W
---	---	---	---	---	---	---	---

d

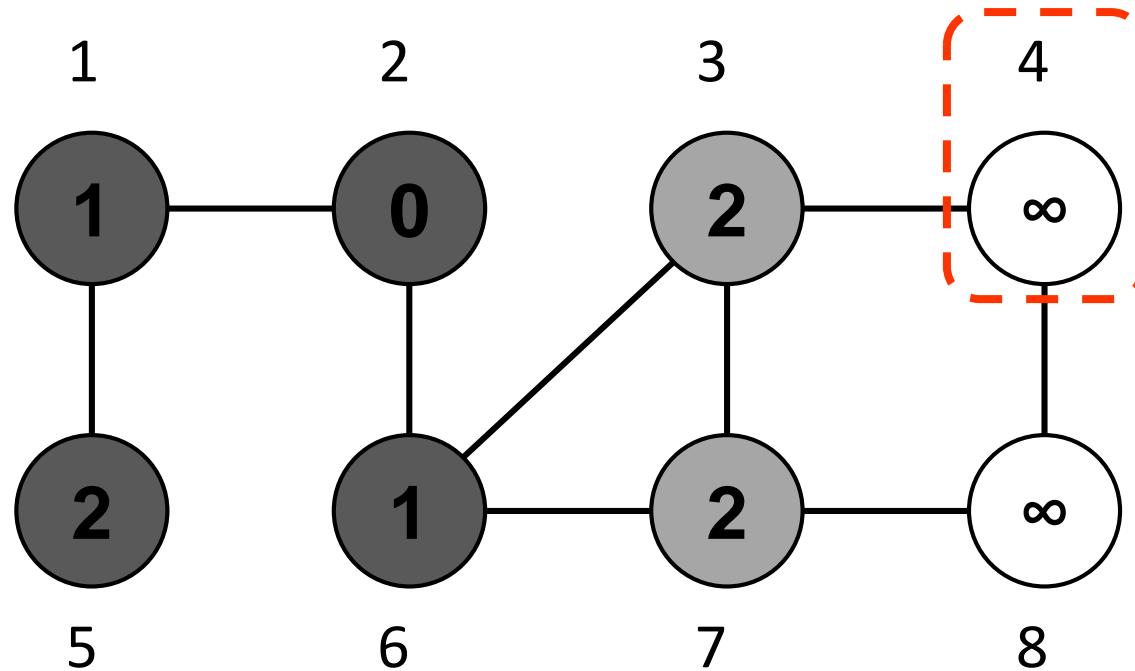
1	0	2	∞	2	1	2	∞
---	---	---	----------	---	---	---	----------

pred

2	N	6	N	1	2	6	N
---	---	---	---	---	---	---	---

Q

2	1	6	5	3	7
---	---	---	---	---	---



BFS Example

color

B	B	G	G	B	B	G	W
---	---	---	---	---	---	---	---

d

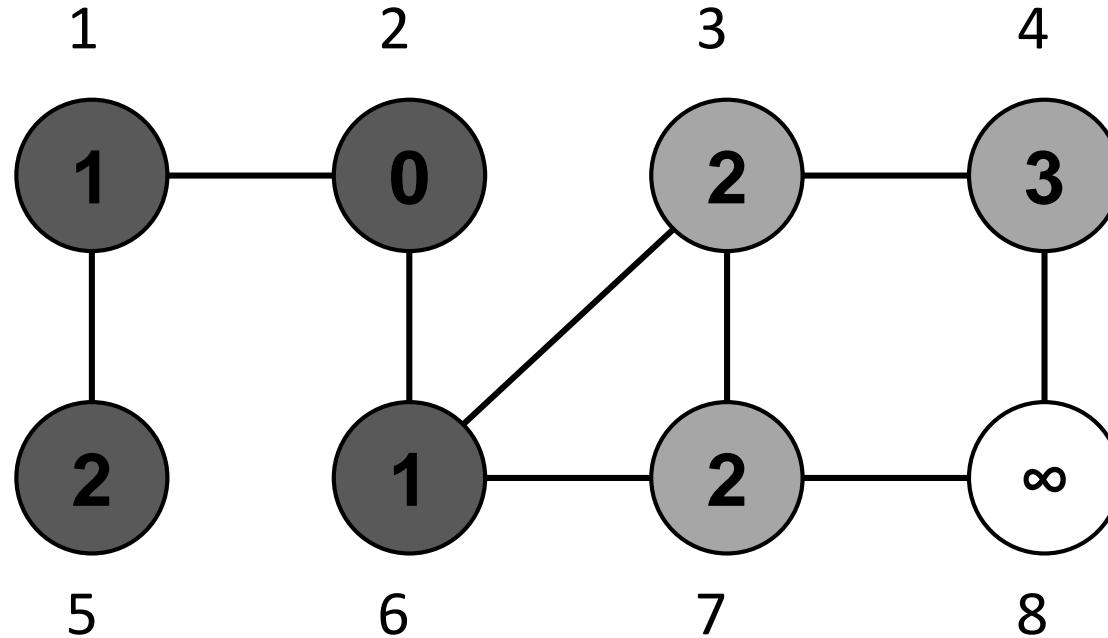
1	0	2	3	2	1	2	∞
---	---	---	---	---	---	---	----------

pred

2	N	6	3	1	2	6	N
---	---	---	---	---	---	---	---

Q

2	1	6	5	3	7	4
---	---	---	---	---	---	---



BFS Example

color

B	B	B	G	B	B	G	W
---	---	---	---	---	---	---	---

d

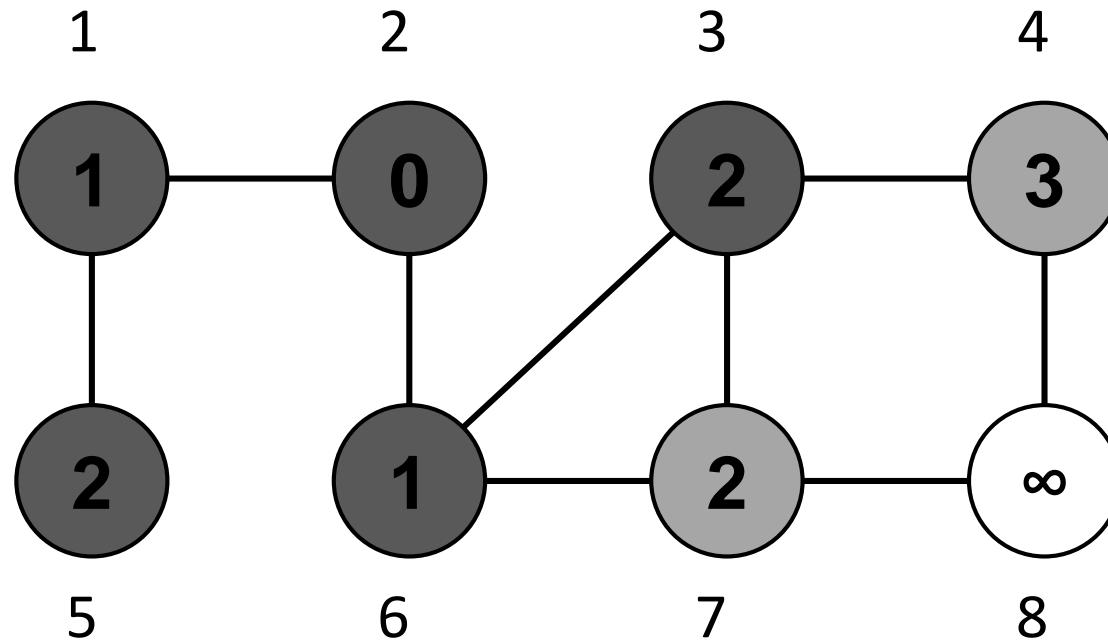
1	0	2	3	2	1	2	∞
---	---	---	---	---	---	---	----------

pred

2	N	6	3	1	2	6	N
---	---	---	---	---	---	---	---

Q

2	1	6	5	3	7	4
---	---	---	---	---	---	---



BFS Example

color

B	B	B	G	B	B	G	W
---	---	---	---	---	---	---	---

d

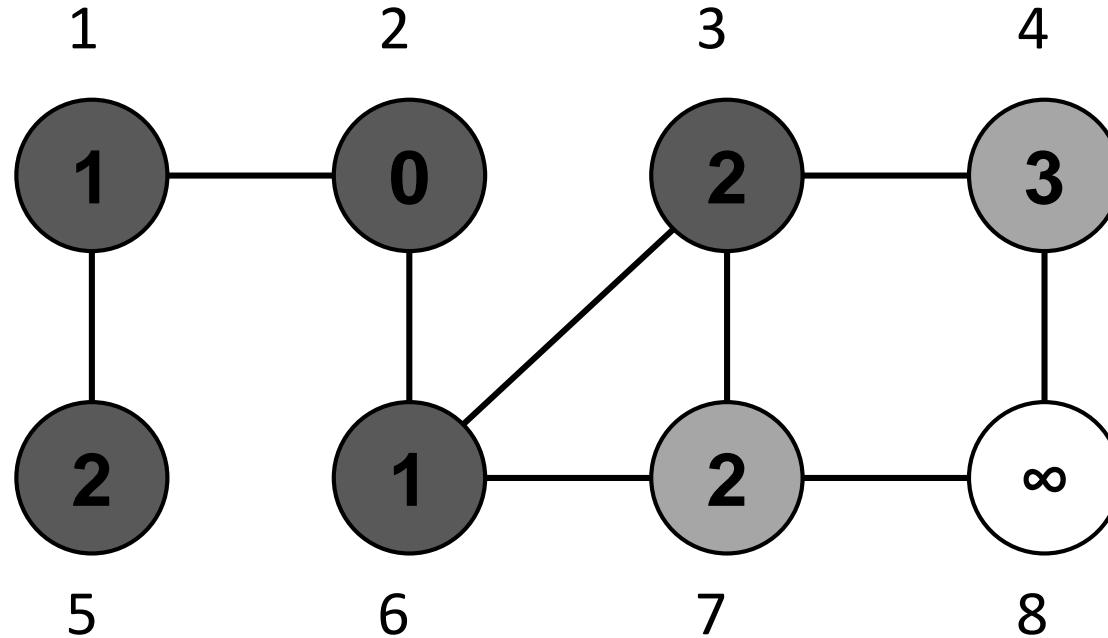
1	0	2	3	2	1	2	∞
---	---	---	---	---	---	---	----------

pred

2	N	6	3	1	2	6	N
---	---	---	---	---	---	---	---

Q

2	1	6	5	3	7	4
---	---	---	---	---	---	---



BFS Example

color

B	B	B	G	B	B	G	W
---	---	---	---	---	---	---	---

d

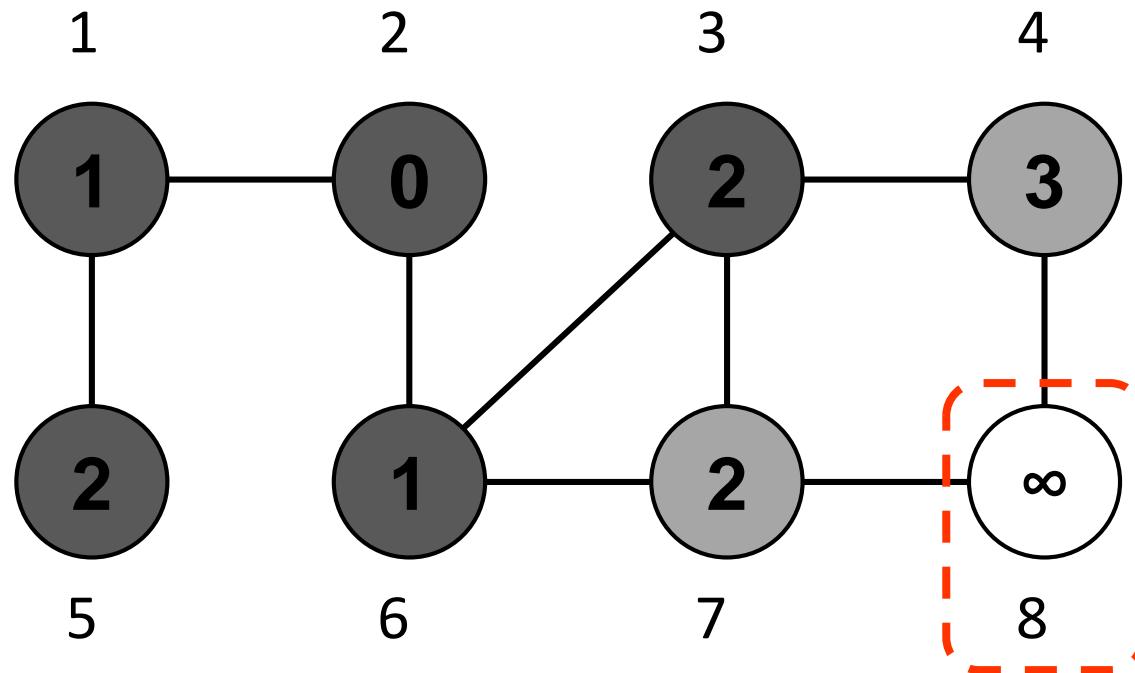
1	0	2	3	2	1	2	∞
---	---	---	---	---	---	---	----------

pred

2	N	6	3	1	2	6	N
---	---	---	---	---	---	---	---

Q

2	1	6	5	3	7	4
---	---	---	---	---	---	---



BFS Example

color

B	B	B	G	B	B	G	G
---	---	---	---	---	---	---	---

d

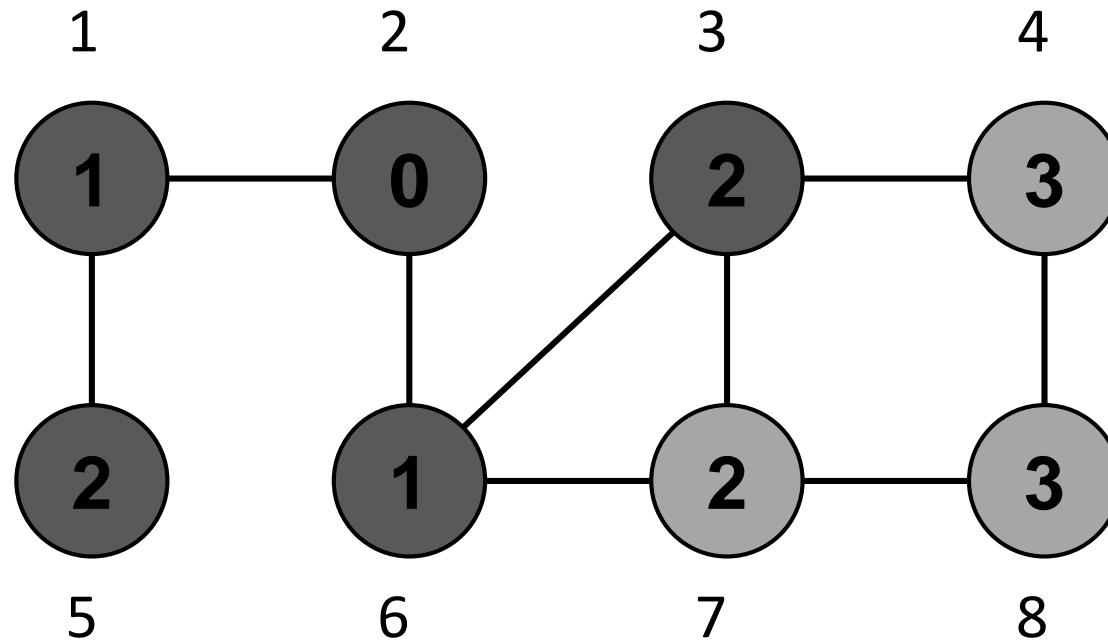
1	0	2	3	2	1	2	3
---	---	---	---	---	---	---	---

pred

2	N	6	3	1	2	6	7
---	---	---	---	---	---	---	---

Q

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



BFS Example

color

B	B	B	G	B	B	B	G
---	---	---	---	---	---	----------	---

d

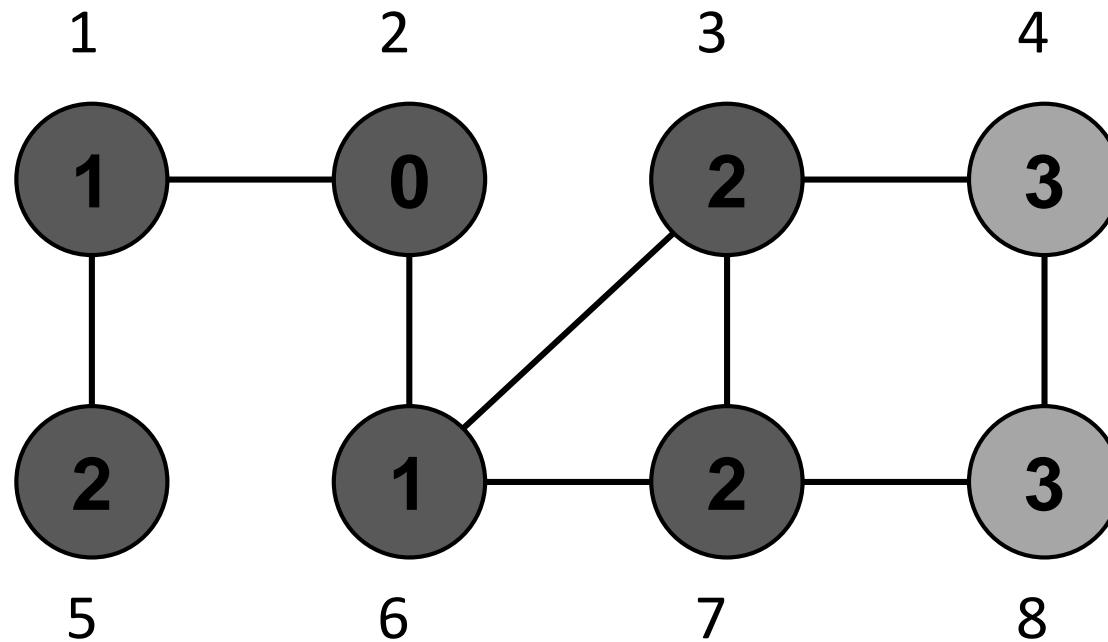
1	0	2	3	2	1	2	3
---	---	---	---	---	---	---	---

pred

2	N	6	3	1	2	6	7
---	---	---	---	---	---	---	---

Q

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



BFS Example

color

B	B	B	G	B	B	B	G
---	---	---	---	---	---	---	---

d

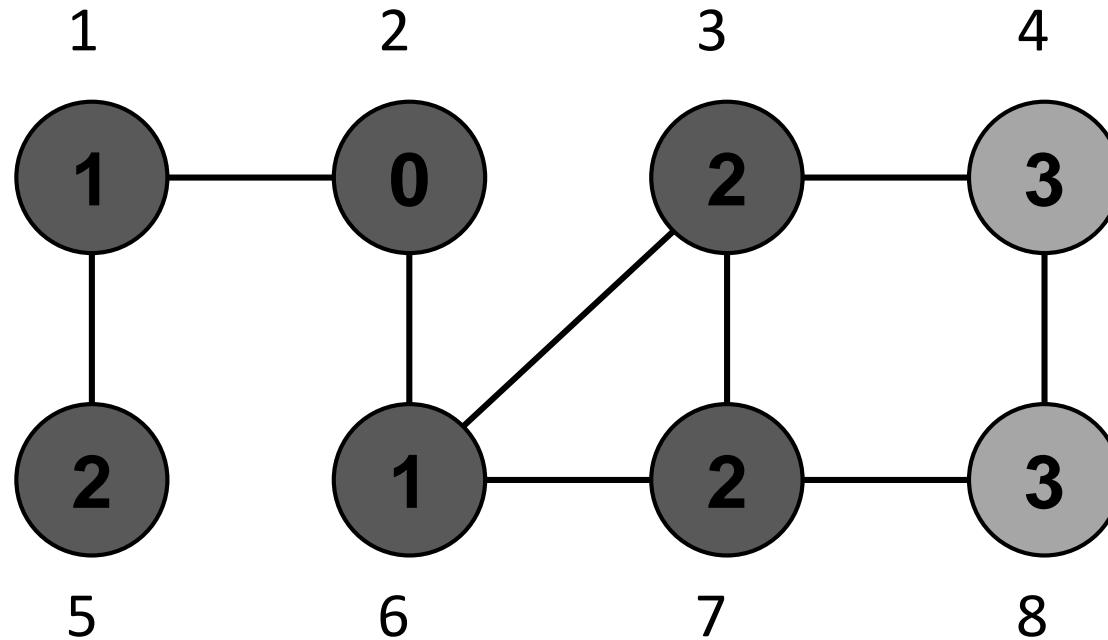
1	0	2	3	2	1	2	3
---	---	---	---	---	---	---	---

pred

2	N	6	3	1	2	6	7
---	---	---	---	---	---	---	---

Q

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



BFS Example

color

B	B	B	B	B	B	G
---	---	---	---	---	---	---

d

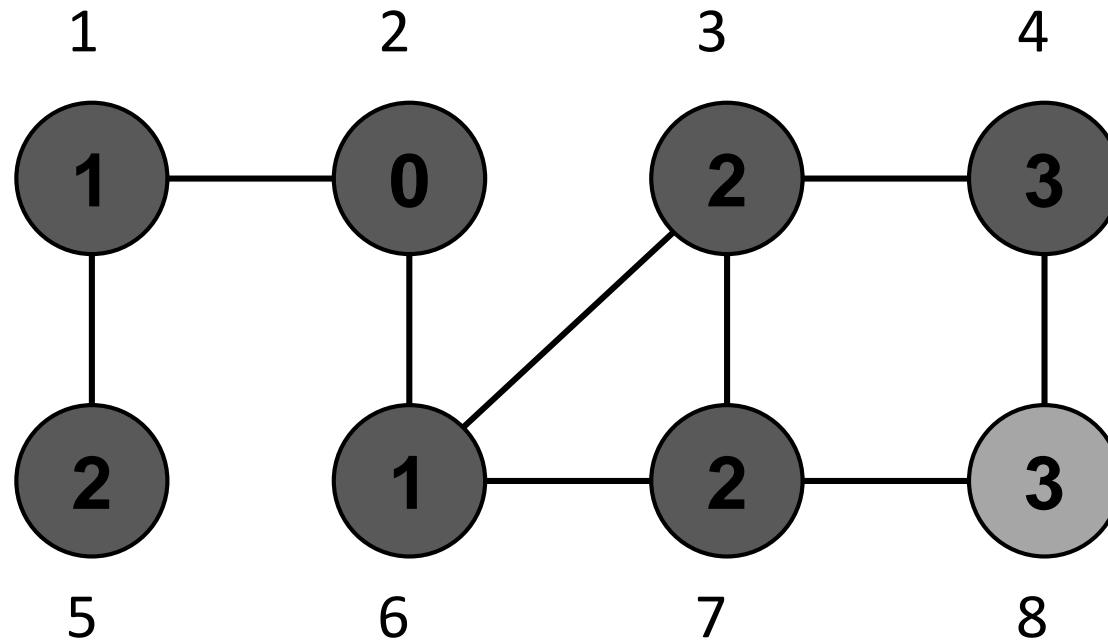
1	0	2	3	2	1	2	3
---	---	---	---	---	---	---	---

pred

2	N	6	3	1	2	6	7
---	---	---	---	---	---	---	---

Q

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



BFS Example

color

B	B	B	B	B	B	B	G
---	---	---	---	---	---	---	---

d

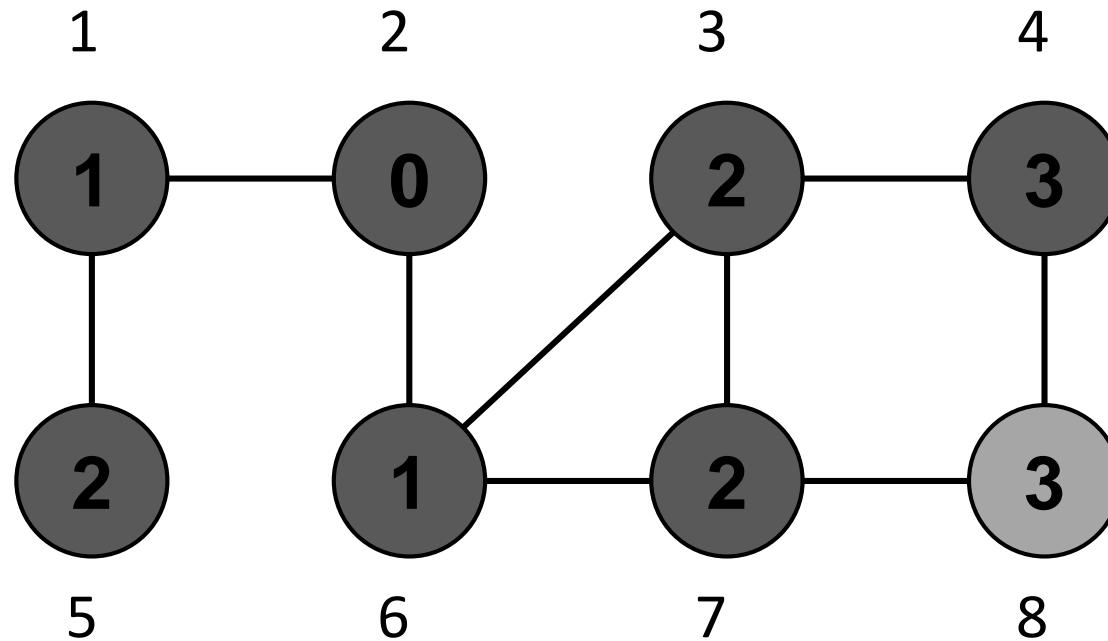
1	0	2	3	2	1	2	3
---	---	---	---	---	---	---	---

pred

2	N	6	3	1	2	6	7
---	---	---	---	---	---	---	---

Q

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



BFS Example

color

B	B	B	B	B	B	B	B
---	---	---	---	---	---	---	---

d

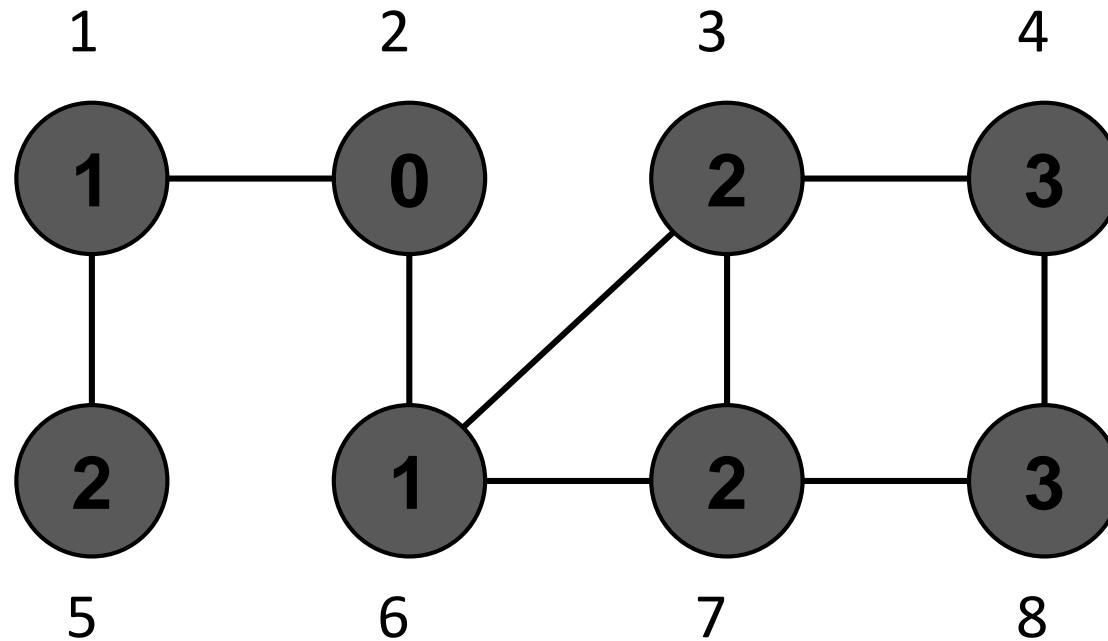
1	0	2	3	2	1	2	3
---	---	---	---	---	---	---	---

pred

2	N	6	3	1	2	6	7
---	---	---	---	---	---	---	---

Q

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



BFS Example

color

B	B	B	B	B	B	B	B
---	---	---	---	---	---	---	---

d

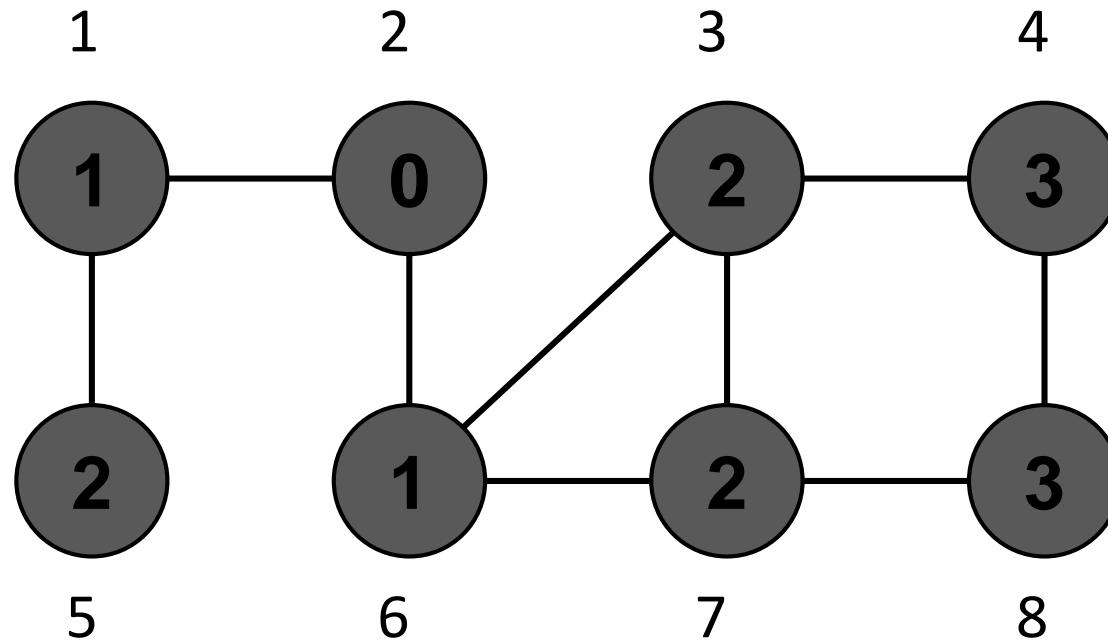
1	0	2	3	2	1	2	3
---	---	---	---	---	---	---	---

pred

2	N	6	3	1	2	6	7
---	---	---	---	---	---	---	---

Q

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



Running Time of BFS

On each vertex u , we spend time $T_u = O(1+\text{degree}(u))$

The total running time is

$$\sum_{u \in V} T_u \leq \sum_{u \in V} (O(1 + \text{degree}(u))) =$$

Running Time of BFS

On each vertex u , we spend time $T_u = O(1+\text{degree}(u))$

The total running time is

$$\sum_{u \in V} T_u \leq \sum_{u \in V} (O(1 + \text{degree}(u))) = O(V + E)$$

Running Time of BFS

On each vertex u , we spend time $T_u = O(1 + \text{degree}(u))$

The total running time is

$$\sum_{u \in V} T_u \leq \sum_{u \in V} (O(1 + \text{degree}(u))) = O(V + E)$$

Hence, the running of BFS on a graph with V vertices and E edges is **O(V + E)**

Running Time of BFS

On each vertex u , we spend time $T_u = O(1 + \text{degree}(u))$

The total running time is

$$\sum_{u \in V} T_u \leq \sum_{u \in V} (O(1 + \text{degree}(u))) = O(V + E)$$

Hence, the running of BFS on a graph with V vertices and E edges is **$O(V + E)$**

Applications:

- Shortest paths in a graph
 - What if the graph is weighted?
- Finding connected components

回顾：“啤酒-尿布”与数据挖掘

- Rakesh Agrawal

- 由于在数据挖掘领域中的先驱性研究，特别是关于关联规则挖掘与隐私保护的研究，其被称为数据挖掘之父。
- 其关于关联规则挖掘的论文，至今已经被全球学者引用超过4.4万次，并分别获得数据库领域国际顶级会议SIGMOD和VLDB的“时间检验奖”与“最具影响力奖”。



Rakesh Agrawal

Rakesh Agrawal, Tomasz Imielinski, Arun N. Swami. Mining Association Rules between Sets of Items in Large Databases.
In SIGMOD, Pages 207-216, 1993 . (CCF A类会议)

An Application of BFS: Apriori Algorithm

Supermarket Application

Raymond



apple



coke



coffee

Item

History or Transaction

David



diaper



coke

...

We want to find some associations between items.

Emily



milk



biscuit

...

An interesting association:

Diaper and Beer are usually bought together.

Derek



coke



milk

...



diaper



beer

Why? Is it strange?

An Application of BFS: Apriori Algorithm

Supermarket Application

An interesting association:

Diaper and Beer are usually bought together.

Why? Is it strange?



diaper



beer

An Application of BFS: Apriori Algorithm

Supermarket Application

An interesting association:

Diaper and Beer are usually bought together.

Why? Is it strange?



diaper



beer

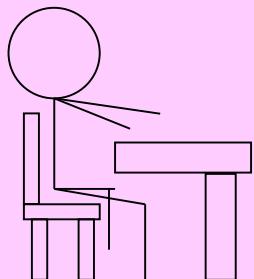
Reasons:

This pattern occurs frequently in the early evening.

Daytime

Office

Working...



An Application of BFS: Apriori Algorithm

Supermarket Application

Why? Is it strange?

An interesting association:

Diaper and Beer are usually bought together.



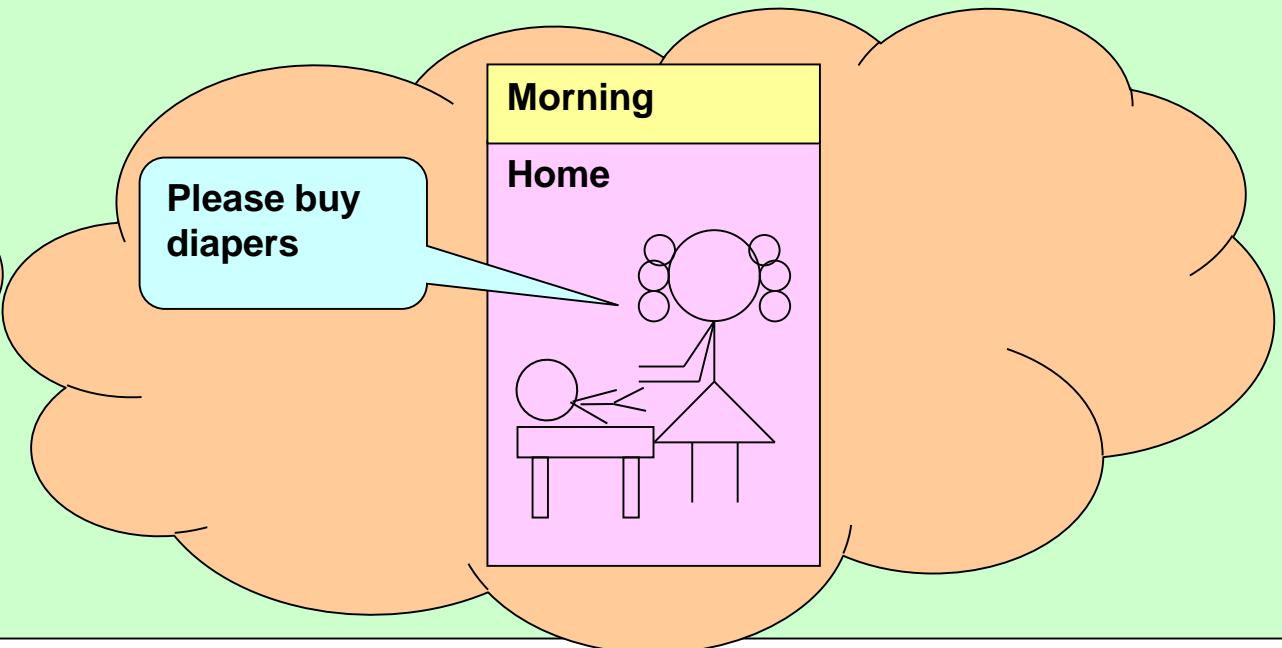
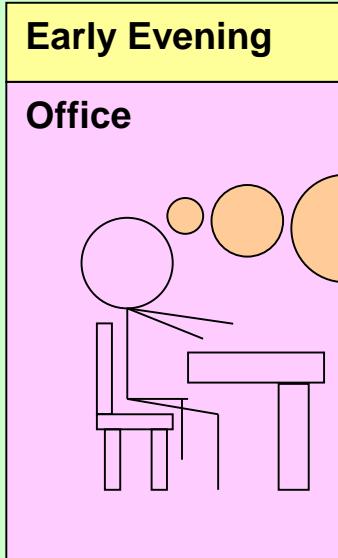
diaper



beer

Reasons:

This pattern occurs frequently in the early evening.



An Application of BFS: Apriori Algorithm



A Specific Threshold=2

Database TDB

Tid	Items
10	A, C, D
20	B, C, E
30	A, B, C, E
40	B, E

C_1

1st scan

Itemset	sup
{A}	2
{B}	3
{C}	3
{D}	1
{E}	3

L_1

Itemset	sup
{A}	2
{B}	3
{C}	3
{E}	3

Dr. R.
Agrawal

L_2

Itemset	sup
{A, C}	2
{B, C}	2
{B, E}	3
{C, E}	2

C_2

2nd scan

Itemset	sup
{A, B}	1
{A, C}	2
{A, E}	1
{B, C}	2
{B, E}	3
{C, E}	2

C_2

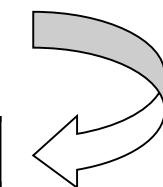
Itemset
{A, B}
{A, C}
{A, E}
{B, C}
{B, E}
{C, E}

C_3

3rd scan

L_3

Itemset	sup
{B, C, E}	2



Outline

- Introduction to Part IV
- Review of Basic Graph Search Algorithms
 - Basic Concepts
 - The Breadth-First Search (BFS) Algorithm
 - The Depth-First Search (DFS) Algorithm
- Topological Sort
 - The Topological Sort Algorithm
 - Analysis of the Topological Sort Algorithm
- Strongly Connected Components
 - The Algorithm of Finding SCCs
 - Analysis of the Algorithm

The Depth-First Search (DFS) Algorithm

What does Depth-First Search (DFS) do?

The Depth-First Search (DFS) Algorithm

What does Depth-First Search (DFS) do?

- Traverse all vertices in graph, and thereby
- Reveal properties of the graph.

The Depth-First Search (DFS) Algorithm

What does Depth-First Search (DFS) do?

- Traverse all vertices in graph, and thereby
- Reveal properties of the graph.

Four arrays are used to keep information gathered during traversal

The Depth-First Search (DFS) Algorithm

What does Depth-First Search (DFS) do?

- Traverse all vertices in graph, and thereby
- Reveal properties of the graph.

Four arrays are used to keep information gathered during traversal

- `color[u]`: the **color** of each vertex visited

The Depth-First Search (DFS) Algorithm

What does Depth-First Search (DFS) do?

- Traverse all vertices in graph, and thereby
- Reveal properties of the graph.

Four arrays are used to keep information gathered during traversal

- **color[u]**: the **color** of each vertex visited
 - WHITE: **undiscovered**

The Depth-First Search (DFS) Algorithm

What does Depth-First Search (DFS) do?

- Traverse all vertices in graph, and thereby
- Reveal properties of the graph.

Four arrays are used to keep information gathered during traversal

- **color[u]**: the **color** of each vertex visited
 - WHITE: **undiscovered**
 - GRAY: **discovered** but not finished processing

The Depth-First Search (DFS) Algorithm

What does Depth-First Search (DFS) do?

- Traverse all vertices in graph, and thereby
- Reveal properties of the graph.

Four arrays are used to keep information gathered during traversal

- **color[u]**: the **color** of each vertex visited
 - WHITE: **undiscovered**
 - GRAY: **discovered** but not finished processing
 - BLACK: **finished** processing

The Depth-First Search (DFS) Algorithm

What does Depth-First Search (DFS) do?

- Traverse all vertices in graph, and thereby
- Reveal properties of the graph.

Four arrays are used to keep information gathered during traversal

- $\text{color}[u]$: the **color** of each vertex visited
 - WHITE: **undiscovered**
 - GRAY: **discovered** but not finished processing
 - BLACK: **finished** processing
- $\text{pred}[u]$: the **predecessor** pointer
 - pointing back to the vertex from which u was discovered

The Depth-First Search (DFS) Algorithm

What does Depth-First Search (DFS) do?

- Traverse all vertices in graph, and thereby
- Reveal properties of the graph.

Four arrays are used to keep information gathered during traversal

- $\text{color}[u]$: the **color** of each vertex visited
 - WHITE: **undiscovered**
 - GRAY: **discovered** but not finished processing
 - BLACK: **finished** processing
- $\text{pred}[u]$: the **predecessor** pointer
 - pointing back to the vertex from which u was discovered
- $d[u]$: the **discovery time**
 - a counter indicating when vertex u is discovered

The Depth-First Search (DFS) Algorithm

What does Depth-First Search (DFS) do?

- Traverse all vertices in graph, and thereby
- Reveal properties of the graph.

Four arrays are used to keep information gathered during traversal

- $\text{color}[u]$: the **color** of each vertex visited
 - WHITE: **undiscovered**
 - GRAY: **discovered** but not finished processing
 - BLACK: **finished** processing
- $\text{pred}[u]$: the **predecessor** pointer
 - pointing back to the vertex from which u was discovered
- $d[u]$: the **discovery time**
 - a counter indicating when vertex u is discovered
- $f[u]$: the **finishing time**
 - a counter indicating when the processing of vertex u (and **all** its descendants) is finished

The DFS Algorithm

How does DFS work?

The DFS Algorithm

How does DFS work?

- It starts from an initial vertex
- After visiting a vertex, it recursively visits all of its neighbors
- The strategy is to search "deeper" in the graph whenever possible

DFS Algorithm

DFS(G)

Input: A graph G

Output: None

for u in V **do**

$\text{color}[u] \leftarrow \text{WHITE}$; //undiscovered

$\text{pred}[u] \leftarrow \text{NULL}$; //no predecessor

end

DFS Algorithm

DFS(G)

```
Input: A graph  $G$ 
Output: None
for  $u$  in  $V$  do
    |  $color[u] \leftarrow$  WHITE; //undiscovered
    |  $pred[u] \leftarrow$  NULL; //no predecessor
end
 $time \leftarrow 0$ ;
for  $u$  in  $V$  do
    |
end
```

DFS Algorithm

DFS(G)

```
Input: A graph  $G$ 
Output: None
for  $u$  in  $V$  do
    |  $color[u] \leftarrow$  WHITE; //undiscovered
    |  $pred[u] \leftarrow$  NULL; //no predecessor
end
 $time \leftarrow 0$ ;
for  $u$  in  $V$  do
    | //start a new tree
    | if  $color[u]$  is equal to WHITE then
    | |
    | end
end
```

DFS Algorithm

DFS(G)

```
Input: A graph  $G$ 
Output: None
for  $u$  in  $V$  do
    |  $color[u] \leftarrow$  WHITE; //undiscovered
    |  $pred[u] \leftarrow$  NULL; //no predecessor
end
 $time \leftarrow 0$ ;
for  $u$  in  $V$  do
    | //start a new tree
    | if  $color[u]$  is equal to WHITE then
    |   | DFSVisit( $u$ );
    | end
end
```

DFS Algorithm

DFSVisit(u)

Input: A vertex u

Output: None

$\text{color}[u] \leftarrow \text{GRAY}; // u \text{ is discovered}$

DFS Algorithm

DFSVisit(u)

Input: A vertex u

Output: None

$\text{color}[u] \leftarrow \text{GRAY}; // u \text{ is discovered}$

$d[u] \leftarrow ++\text{time}; // u \text{'s discovery time}$

DFS Algorithm

DFSVisit(u)

Input: A vertex u

Output: None

```
color[ $u$ ]  $\leftarrow$  GRAY; // $u$  is discovered  
 $d[u]$   $\leftarrow$   $++time$ ; // $u$ 's discovery time  
for  $v \in Adj(u)$  do
```

|

DFS Algorithm

DFSVisit(u)

Input: A vertex u

Output: None

```
color[ $u$ ]  $\leftarrow$  GRAY; // $u$  is discovered  
 $d[u]$   $\leftarrow$   $++time$ ; // $u$ 's discovery time  
for  $v \in Adj(u)$  do  
    //Visit undiscovered vertex  
    if color[ $v$ ] is equal to WHITE then
```

DFS Algorithm

DFSVisit(u)

Input: A vertex u

Output: None

```
color[ $u$ ]  $\leftarrow$  GRAY; // $u$  is discovered  
 $d[u]$   $\leftarrow$   $++time$ ; // $u$ 's discovery time  
for  $v \in Adj(u)$  do  
    //Visit undiscovered vertex  
    if color[ $v$ ] is equal to WHITE then  
        pred[ $v$ ]  $\leftarrow u$ ;
```

DFS Algorithm

DFSVisit(u)

Input: A vertex u

Output: None

```
color[ $u$ ]  $\leftarrow$  GRAY; // $u$  is discovered  
 $d[u]$   $\leftarrow$   $++time$ ; // $u$ 's discovery time  
for  $v \in Adj(u)$  do  
    //Visit undiscovered vertex  
    if color[ $v$ ] is equal to WHITE then  
        pred[ $v$ ]  $\leftarrow u$ ;  
        DFSVisit( $v$ );
```

DFS Algorithm

DFSVisit(u)

Input: A vertex u

Output: None

```
color[ $u$ ]  $\leftarrow$  GRAY; // $u$  is discovered
 $d[u] \leftarrow ++time$ ; // $u$ 's discovery time
for  $v \in Adj(u)$  do
    //Visit undiscovered vertex
    if color[ $v$ ] is equal to WHITE then
        pred[ $v$ ]  $\leftarrow u$ ;
        DFSVisit( $v$ );
    end
end
color[ $u$ ]  $\leftarrow$  BLACK; // $u$  has finished
```

DFS Algorithm

DFSVisit(u)

Input: A vertex u

Output: None

```
color[ $u$ ]  $\leftarrow$  GRAY; // $u$  is discovered
 $d[u] \leftarrow ++time$ ; // $u$ 's discovery time
for  $v \in Adj(u)$  do
    //Visit undiscovered vertex
    if color[ $v$ ] is equal to WHITE then
        pred[ $v$ ]  $\leftarrow u$ ;
        DFSVisit( $v$ );
    end
end
color[ $u$ ]  $\leftarrow$  BLACK; // $u$  has finished
 $f[u] \leftarrow ++time$ ; // $u$ 's finish time
```

DFS Example

time = 0

color

W	W	W	W	W	W	W	W
---	---	---	---	---	---	---	---

d

∞							
----------	----------	----------	----------	----------	----------	----------	----------

pred

N	N	N	N	N	N	N	N
---	---	---	---	---	---	---	---

f

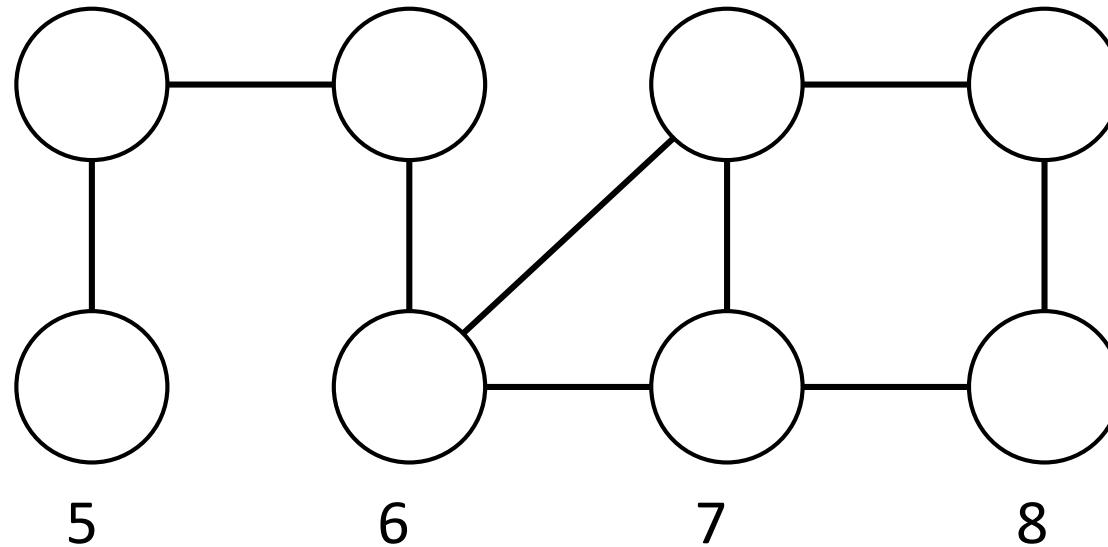
∞							
----------	----------	----------	----------	----------	----------	----------	----------

1

2

3

4



DFS Example

time = 0

color

W	W	W	W	W	W	W	W
---	---	---	---	---	---	---	---

d

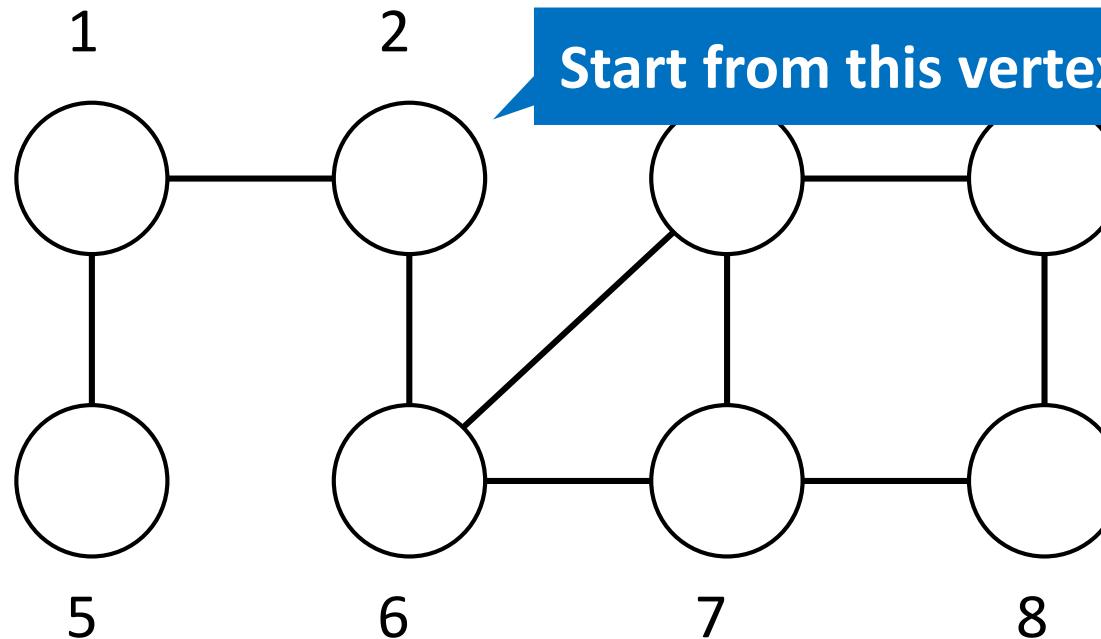
∞							
----------	----------	----------	----------	----------	----------	----------	----------

pred

N	N	N	N	N	N	N	N
---	---	---	---	---	---	---	---

f

∞							
----------	----------	----------	----------	----------	----------	----------	----------



DFS Example

time = 1

color

W	G	W	W	W	W	W	W
---	---	---	---	---	---	---	---

d

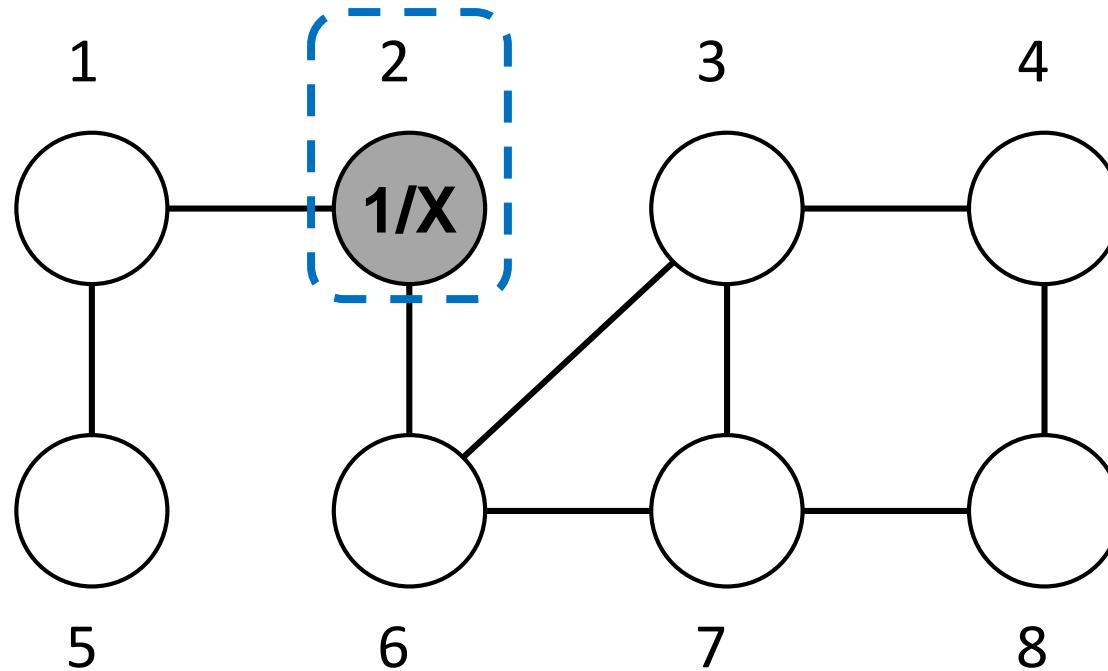
∞	1	∞	∞	∞	∞	∞	∞
----------	---	----------	----------	----------	----------	----------	----------

pred

N	N	N	N	N	N	N	N
---	---	---	---	---	---	---	---

f

∞							
----------	----------	----------	----------	----------	----------	----------	----------



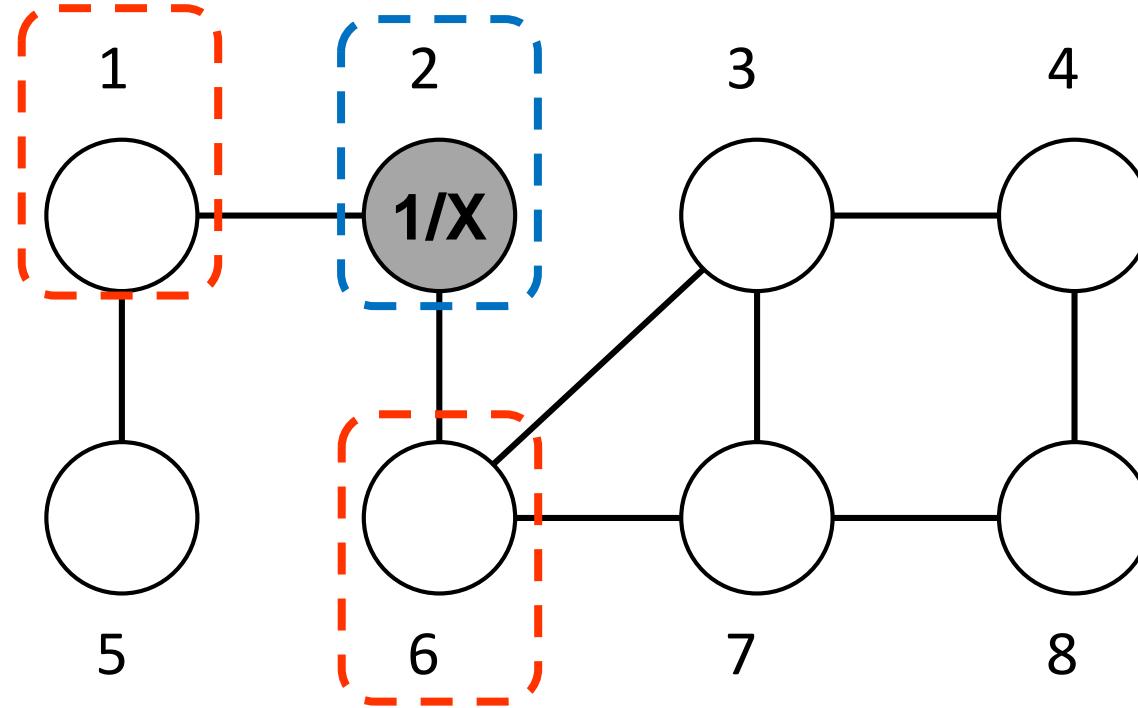
DFS Example

time = 1

color

W G W W W W W W

d



pred

N	N	N	N	N	N	N	N
---	---	---	---	---	---	---	---

f

DFS Example

time = 2

color

W	G	W	W	W	W	W	W
---	---	---	---	---	---	---	---

d

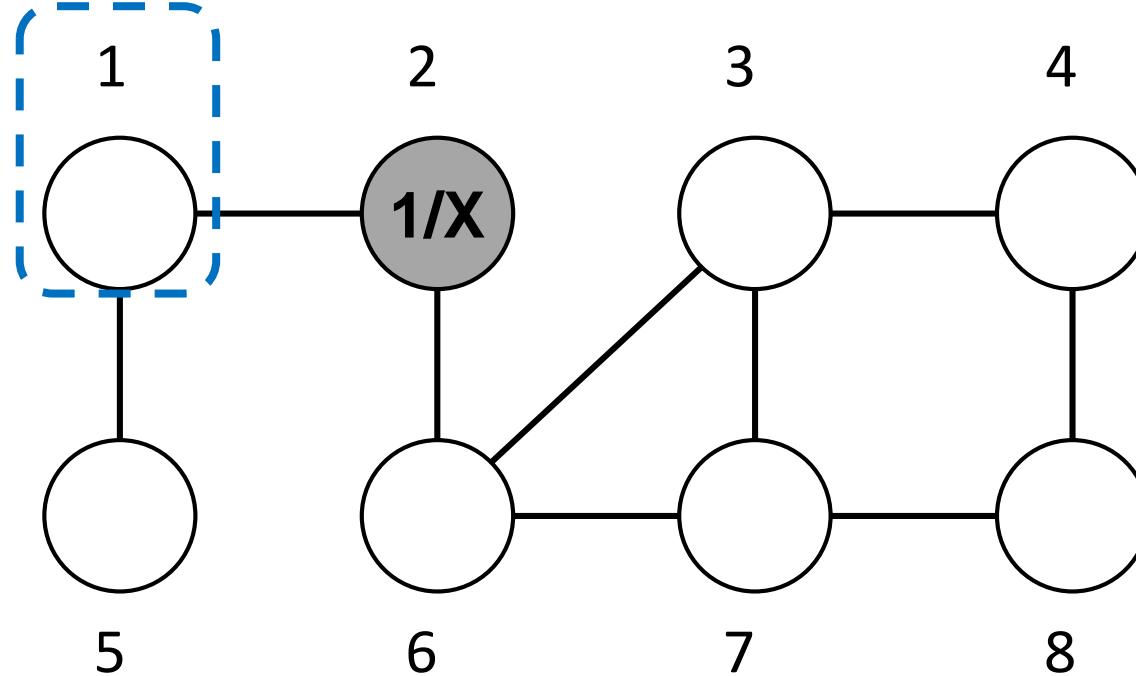
∞	1	∞	∞	∞	∞	∞	∞
----------	---	----------	----------	----------	----------	----------	----------

pred

N	N	N	N	N	N	N	N
---	---	---	---	---	---	---	---

f

∞							
----------	----------	----------	----------	----------	----------	----------	----------



DFS Example

time = 2

color

G	G	W	W	W	W	W	W
----------	----------	----------	----------	----------	----------	----------	----------

d

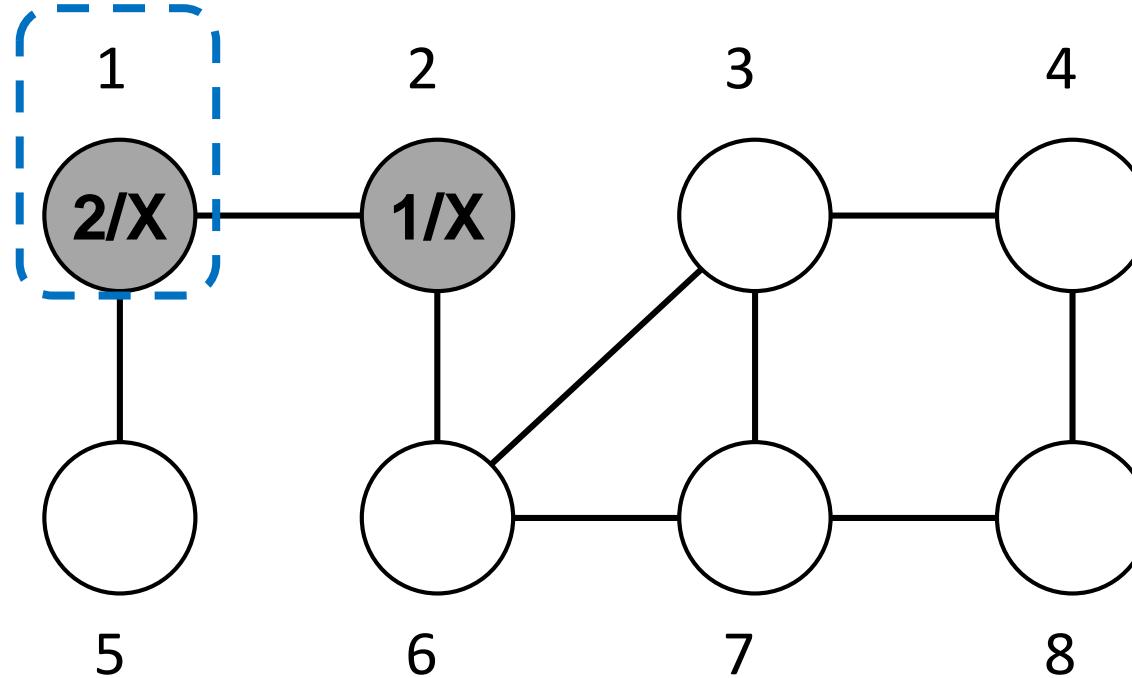
2	1	∞	∞	∞	∞	∞	∞
----------	----------	----------	----------	----------	----------	----------	----------

pred

2	N						
----------	----------	----------	----------	----------	----------	----------	----------

f

∞							
----------	----------	----------	----------	----------	----------	----------	----------



DFS Example

time = 2

color

G	G	W	W	W	W	W	W
----------	----------	----------	----------	----------	----------	----------	----------

d

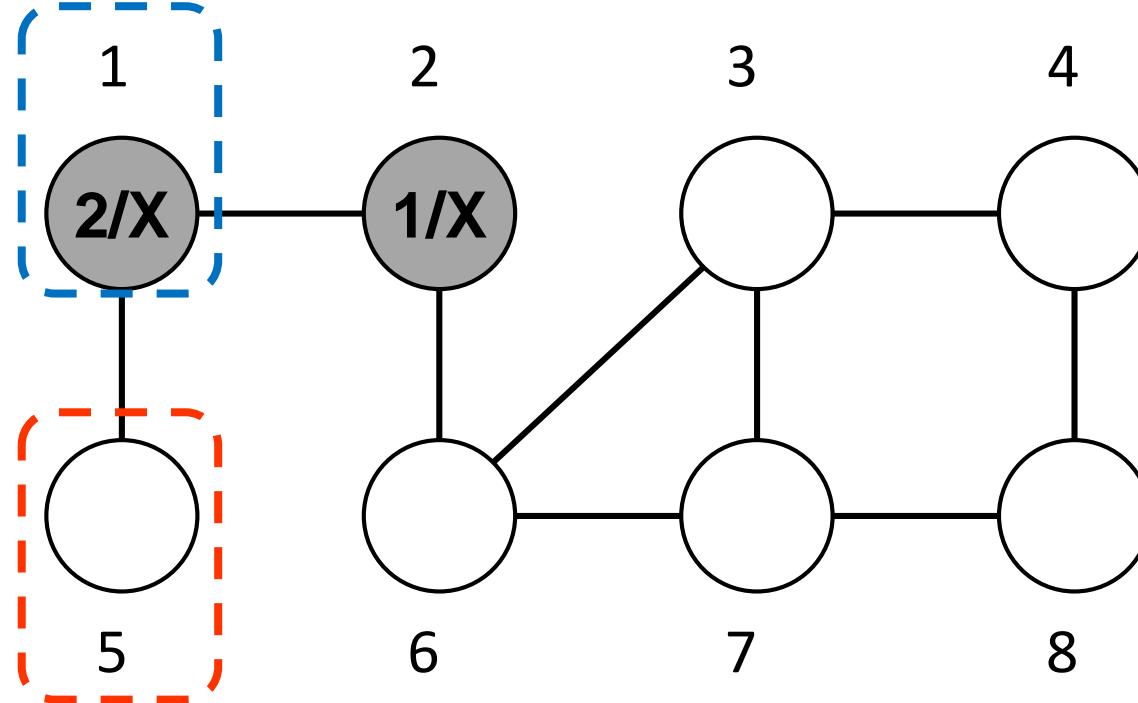
2	1	∞	∞	∞	∞	∞	∞
----------	----------	----------	----------	----------	----------	----------	----------

pred

2	N						
----------	----------	----------	----------	----------	----------	----------	----------

f

∞							
----------	----------	----------	----------	----------	----------	----------	----------



DFS Example

time = 3

color

G	G	W	W	W	W	W	W
---	---	---	---	---	---	---	---

d

2	1	∞	∞	∞	∞	∞	∞
---	---	----------	----------	----------	----------	----------	----------

pred

2	N	N	N	N	N	N	N
---	---	---	---	---	---	---	---

f

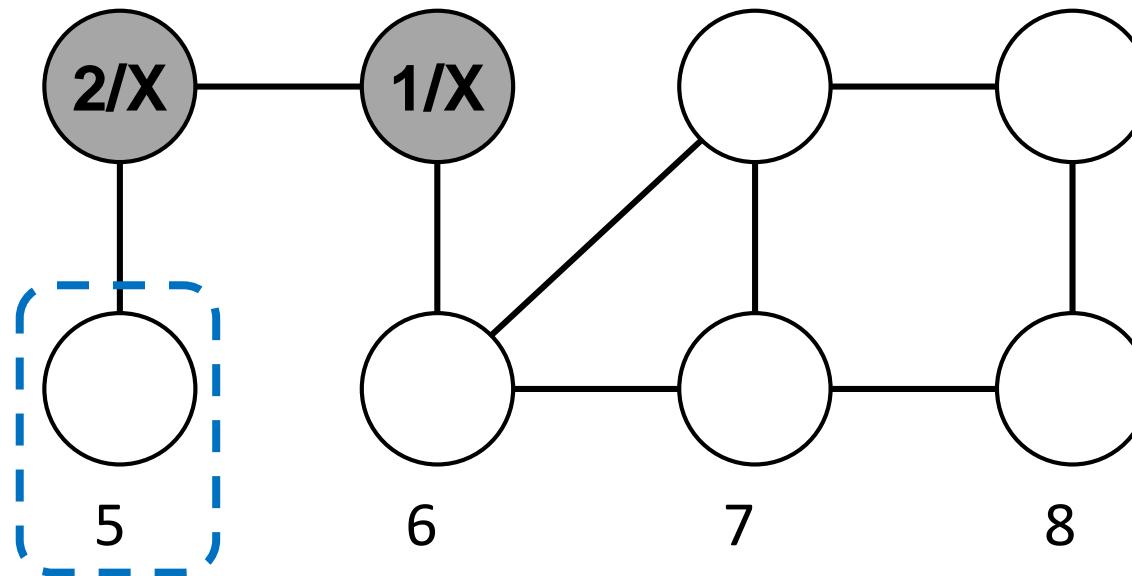
∞							
----------	----------	----------	----------	----------	----------	----------	----------

1

2

3

4



DFS Example

time = 3

color

G	G	W	W	G	W	W	W
---	---	---	---	---	---	---	---

d

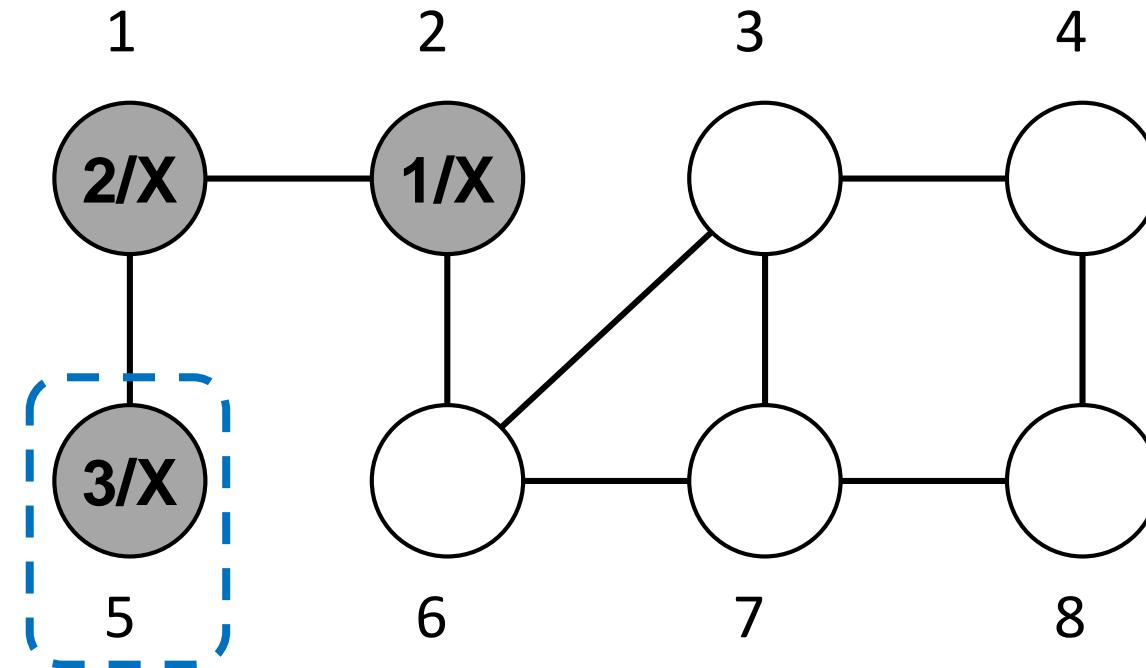
2	1	∞	∞	3	∞	∞	∞
---	---	----------	----------	---	----------	----------	----------

pred

2	N	N	N	1	N	N	N
---	---	---	---	---	---	---	---

f

∞							
----------	----------	----------	----------	----------	----------	----------	----------



DFS Example

time = 3

color

G	G	W	W	G	W	W	W
---	---	---	---	---	---	---	---

d

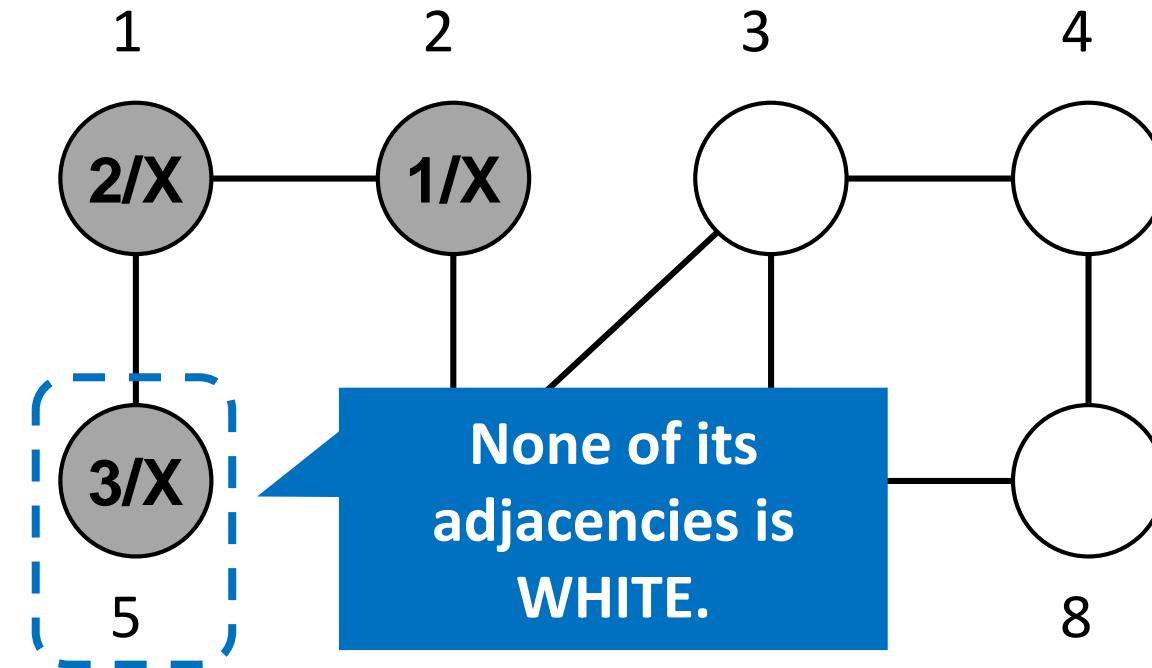
2	1	∞	∞	3	∞	∞	∞
---	---	----------	----------	---	----------	----------	----------

pred

2	N	N	N	1	N	N	N
---	---	---	---	---	---	---	---

f

∞							
----------	----------	----------	----------	----------	----------	----------	----------



DFS Example

time = 4

color

G	G	W	W	G	W	W	W
---	---	---	---	---	---	---	---

d

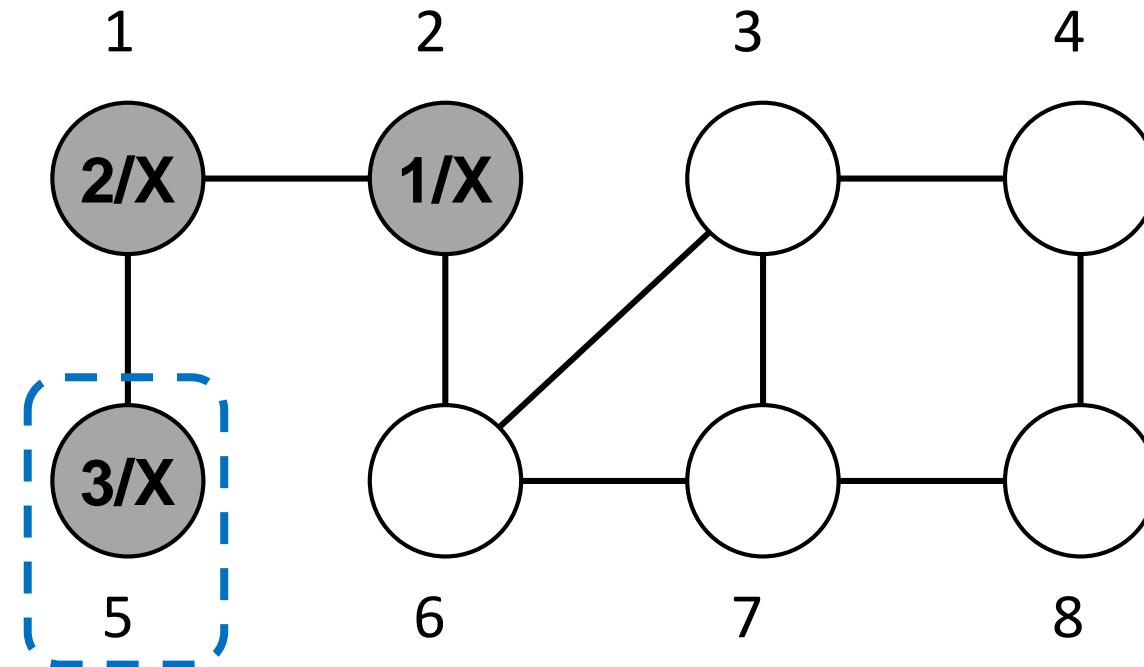
2	1	∞	∞	3	∞	∞	∞
---	---	----------	----------	---	----------	----------	----------

pred

2	N	N	N	1	N	N	N
---	---	---	---	---	---	---	---

f

∞							
----------	----------	----------	----------	----------	----------	----------	----------



DFS Example

time = 4

color

G	G	W	W	B	W	W	W
---	---	---	---	---	---	---	---

d

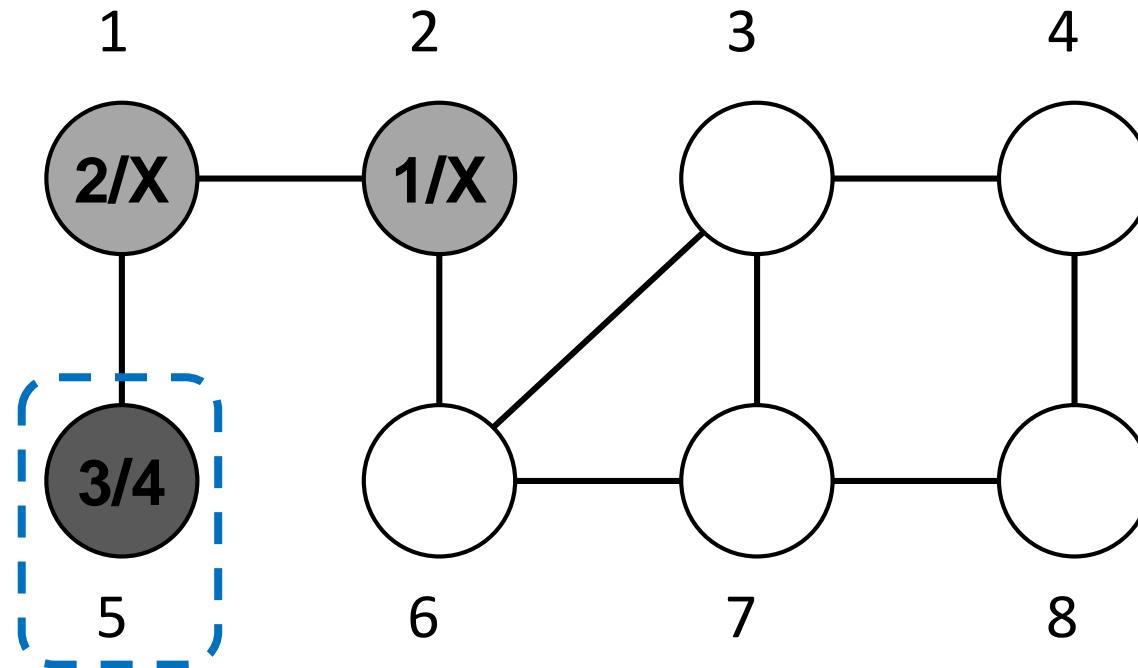
2	1	∞	∞	3	∞	∞	∞
---	---	----------	----------	---	----------	----------	----------

pred

2	N	N	N	1	N	N	N
---	---	---	---	---	---	---	---

f

∞	∞	∞	∞	4	∞	∞	∞
----------	----------	----------	----------	---	----------	----------	----------



DFS Example

time = 4

color

G	G	W	W	B	W	W	W
----------	----------	----------	----------	----------	----------	----------	----------

d

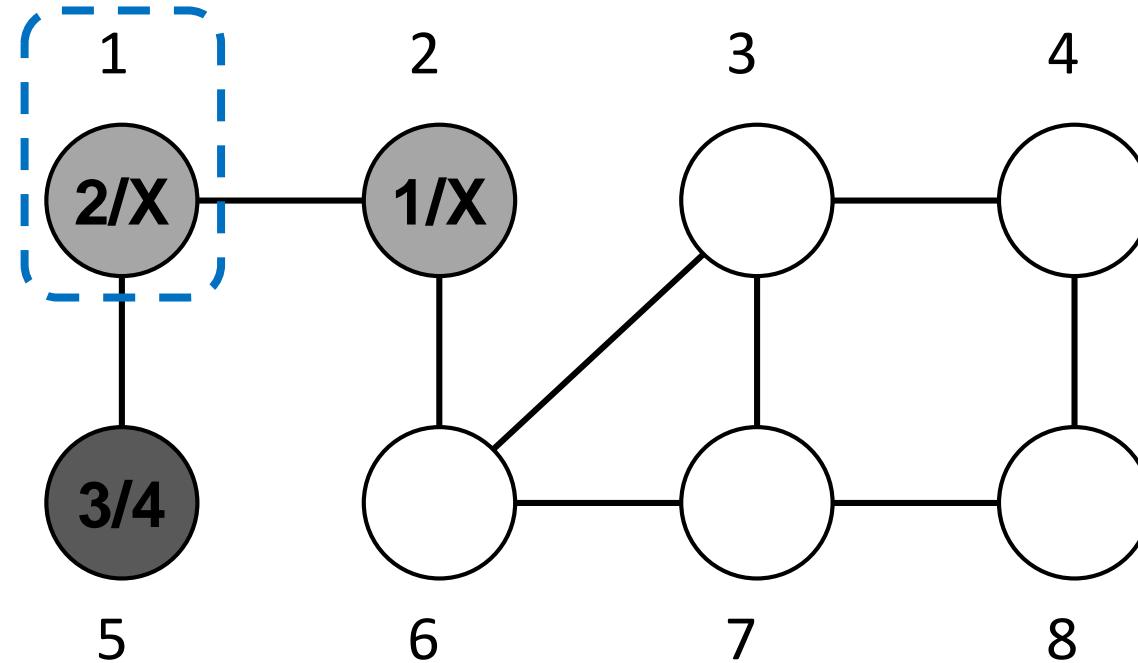
2	1	∞	∞	3	∞	∞	∞
---	---	----------	----------	---	----------	----------	----------

pred

2	N	N	N	1	N	N	N
---	---	---	---	---	---	---	---

f

∞	∞	∞	∞	4	∞	∞	∞
----------	----------	----------	----------	---	----------	----------	----------



DFS Example

time = 4

color

G	G	W	W	B	W	W	W
---	---	---	---	---	---	---	---

d

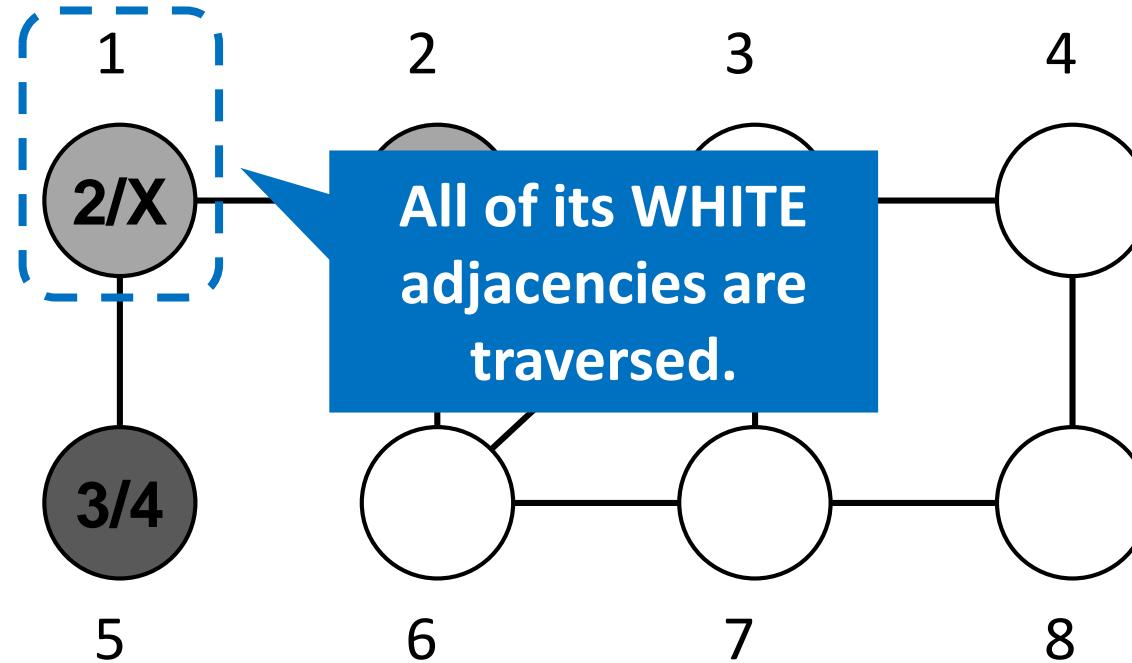
2	1	∞	∞	3	∞	∞	∞
---	---	----------	----------	---	----------	----------	----------

pred

2	N	N	N	1	N	N	N
---	---	---	---	---	---	---	---

f

∞	∞	∞	∞	4	∞	∞	∞
----------	----------	----------	----------	---	----------	----------	----------



DFS Example

time = 5

color

G	G	W	W	B	W	W	W
---	---	---	---	---	---	---	---

d

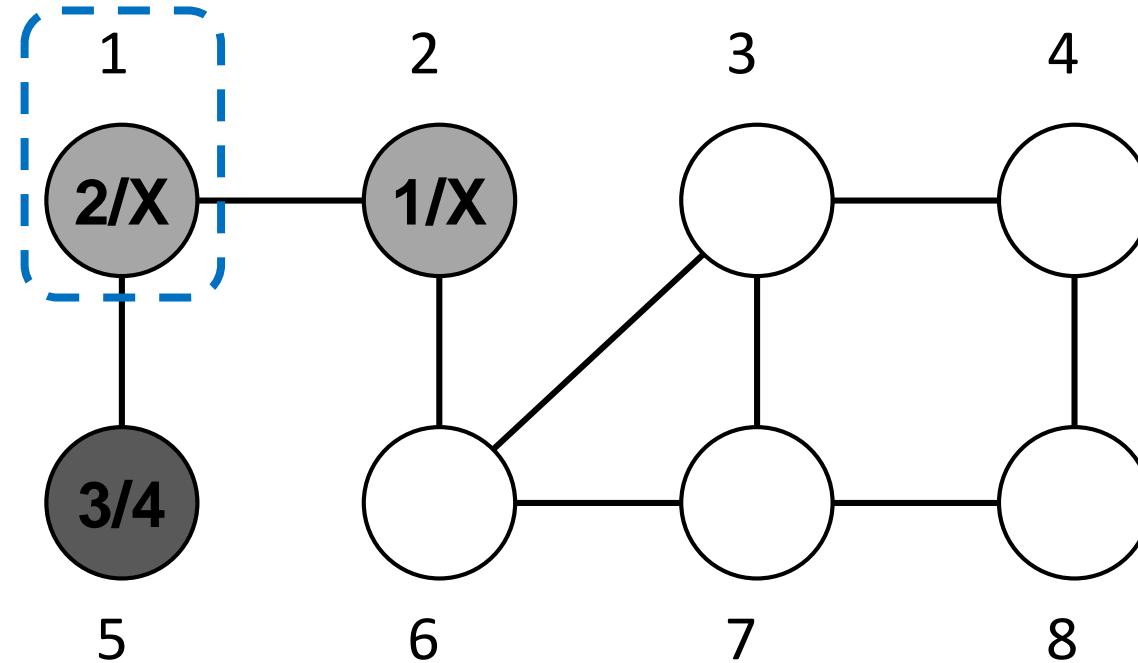
2	1	∞	∞	3	∞	∞	∞
---	---	----------	----------	---	----------	----------	----------

pred

2	N	N	N	1	N	N	N
---	---	---	---	---	---	---	---

f

∞	∞	∞	∞	4	∞	∞	∞
----------	----------	----------	----------	---	----------	----------	----------



DFS Example

time = 5

color

B	G	W	W	B	W	W	W
---	---	---	---	---	---	---	---

d

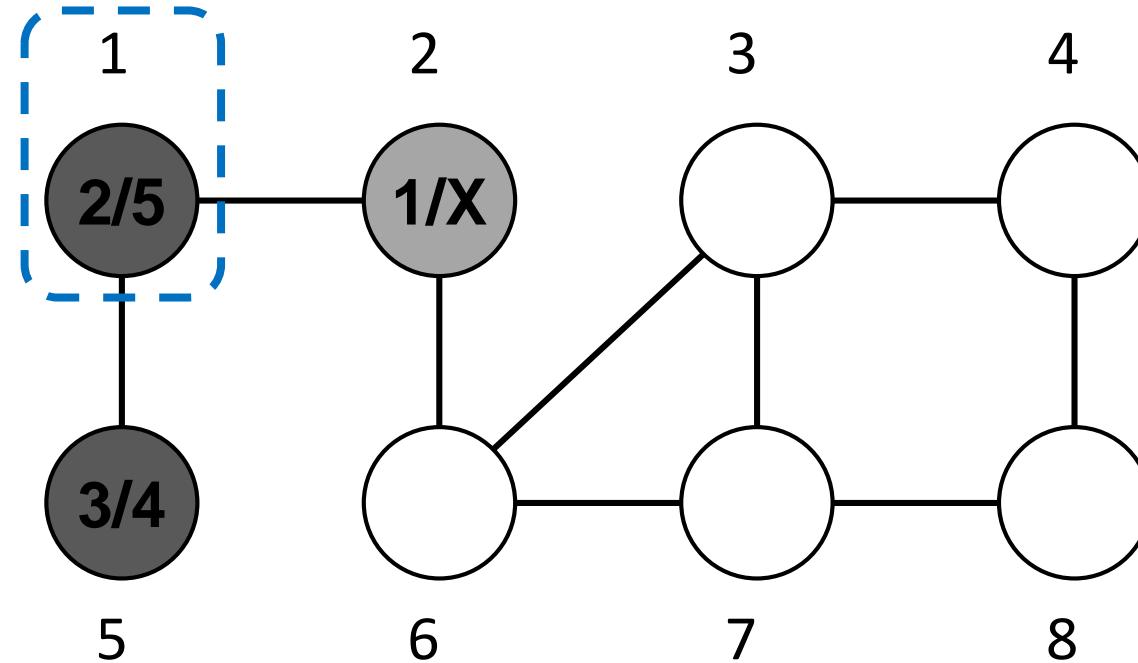
2	1	∞	∞	3	∞	∞	∞
---	---	----------	----------	---	----------	----------	----------

pred

2	N	N	N	1	N	N	N
---	---	---	---	---	---	---	---

f

5	∞	∞	∞	4	∞	∞	∞
---	----------	----------	----------	---	----------	----------	----------



DFS Example

time = 5

color

B	G	W	W	B	W	W	W
---	---	---	---	---	---	---	---

d

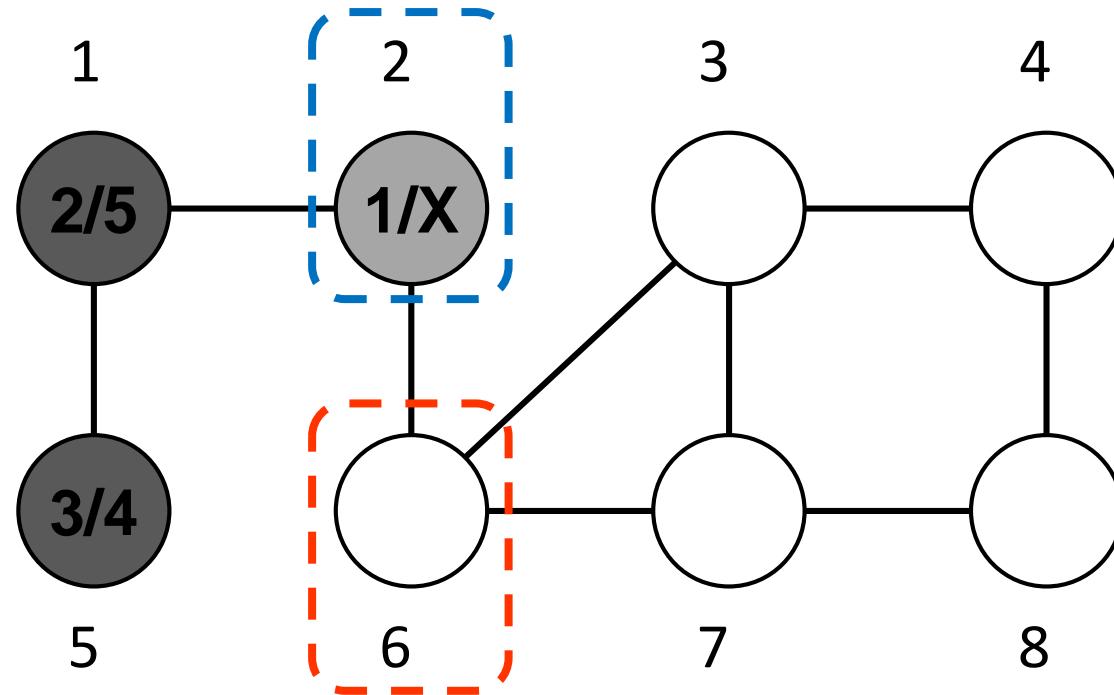
2	1	∞	∞	3	∞	∞	∞
---	---	----------	----------	---	----------	----------	----------

pred

2	N	N	N	1	N	N	N
---	---	---	---	---	---	---	---

f

5	∞	∞	∞	4	∞	∞	∞
---	----------	----------	----------	---	----------	----------	----------



DFS Example

time = 6

color

B	G	W	W	B	W	W	W
---	---	---	---	---	---	---	---

d

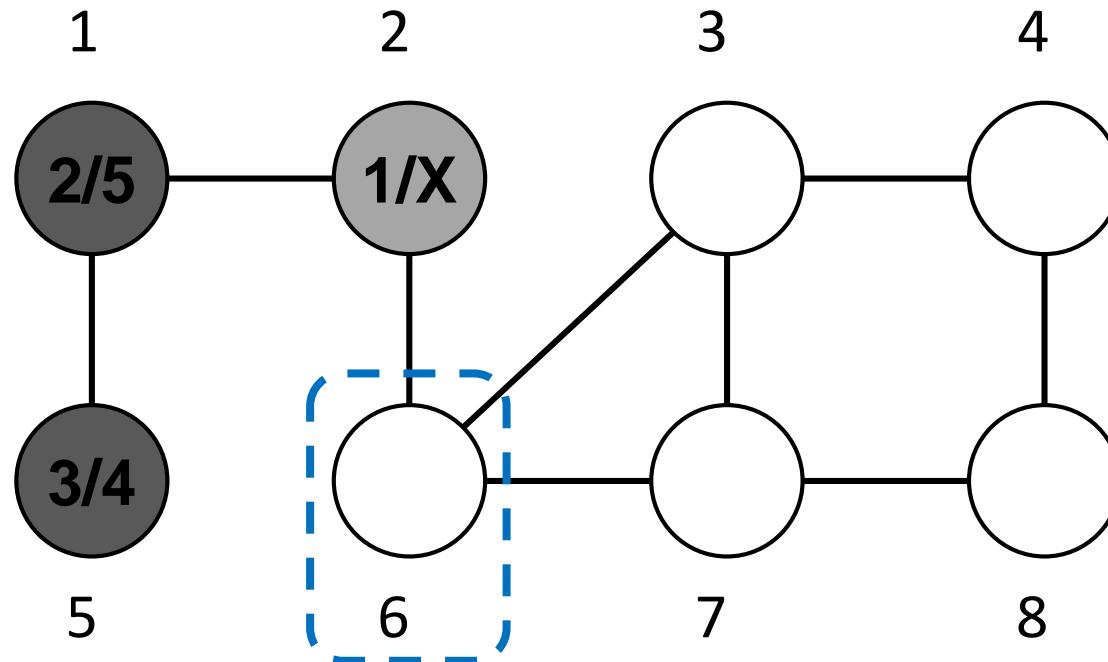
2	1	∞	∞	3	∞	∞	∞
---	---	----------	----------	---	----------	----------	----------

pred

2	N	N	N	1	N	N	N
---	---	---	---	---	---	---	---

f

5	∞	∞	∞	4	∞	∞	∞
---	----------	----------	----------	---	----------	----------	----------



DFS Example

time = 6

color

B	G	W	W	B	G	W	W
---	---	---	---	---	----------	---	---

d

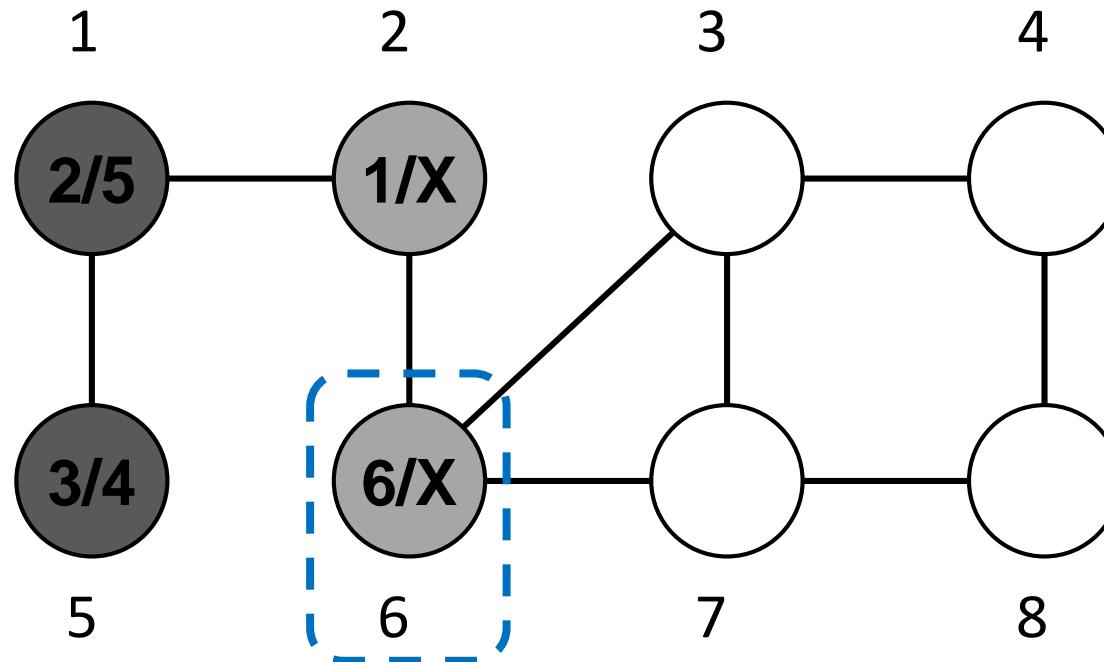
2	1	∞	∞	3	6	∞	∞
---	---	----------	----------	---	----------	----------	----------

pred

2	N	N	N	1	2	N	N
---	---	---	---	---	----------	---	---

f

5	∞	∞	∞	4	∞	∞	∞
---	----------	----------	----------	---	----------	----------	----------



DFS Example

time = 6

color

B	G	W	W	B	G	W	W
---	---	---	---	---	---	---	---

d

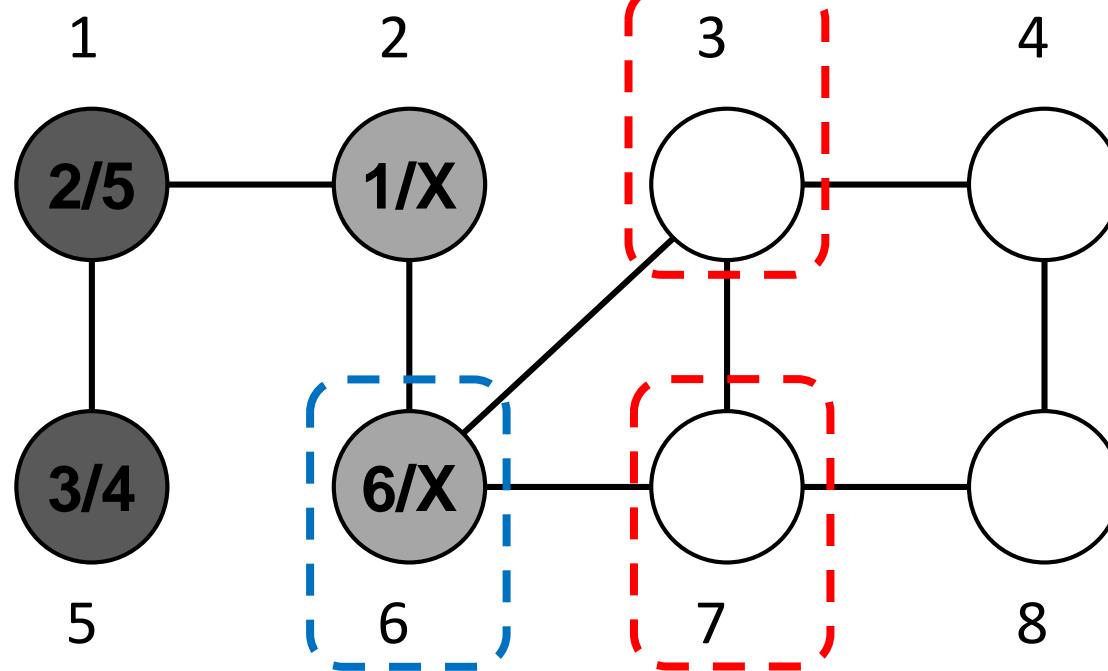
2	1	∞	∞	3	6	∞	∞
---	---	----------	----------	---	---	----------	----------

pred

2	N	N	N	1	2	N	N
---	---	---	---	---	---	---	---

f

5	∞	∞	∞	4	∞	∞	∞
---	----------	----------	----------	---	----------	----------	----------



DFS Example

time = 7

color

B	G	W	W	B	G	W	W
---	---	---	---	---	---	---	---

d

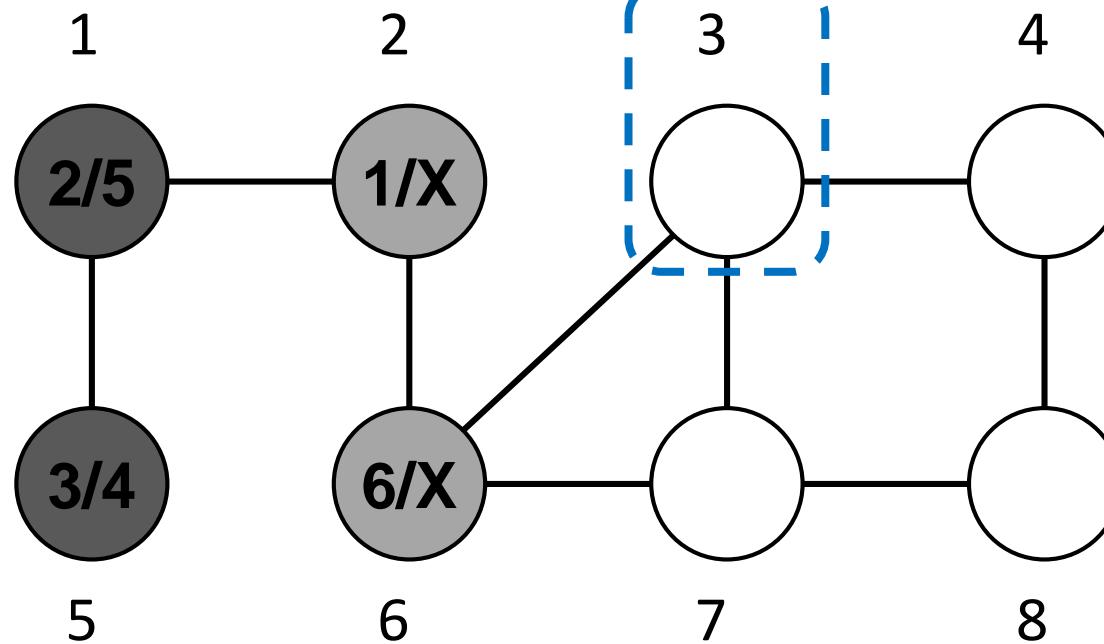
2	1	∞	∞	3	6	∞	∞
---	---	----------	----------	---	---	----------	----------

pred

2	N	N	N	1	2	N	N
---	---	---	---	---	---	---	---

f

5	∞	∞	∞	4	∞	∞	∞
---	----------	----------	----------	---	----------	----------	----------



DFS Example

time = 7

color

B	G	G	W	B	G	W	W
---	---	---	---	---	---	---	---

d

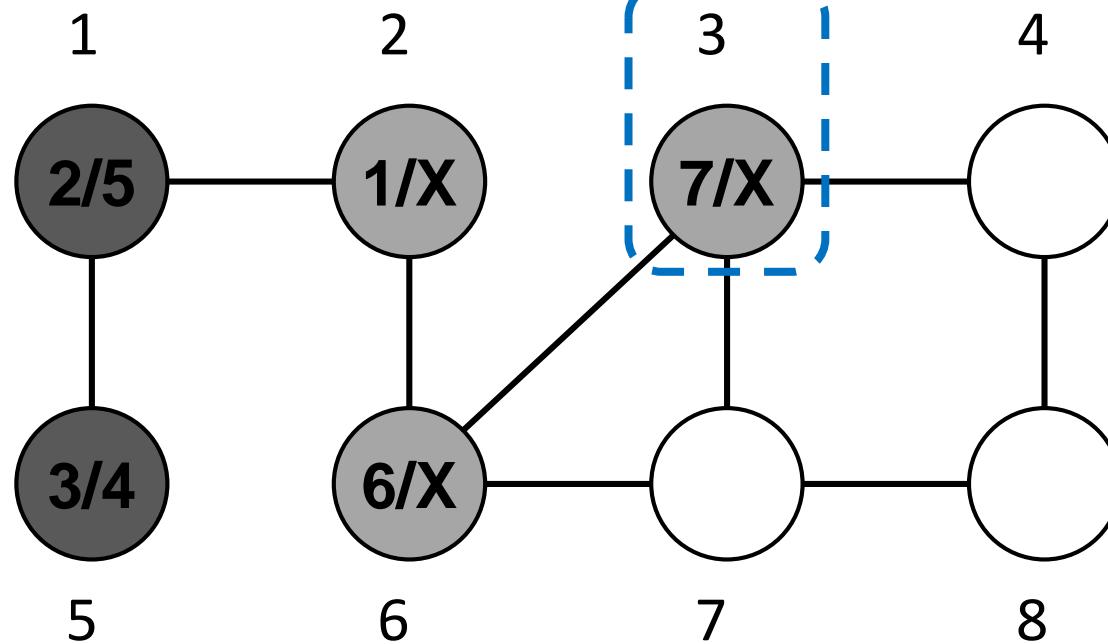
2	1	7	∞	3	6	∞	∞
---	---	---	----------	---	---	----------	----------

pred

2	N	6	N	1	2	N	N
---	---	---	---	---	---	---	---

f

5	∞	∞	∞	4	∞	∞	∞
---	----------	----------	----------	---	----------	----------	----------



DFS Example

time = 7

color

B	G	G	W	B	G	W	W
---	---	---	---	---	---	---	---

d

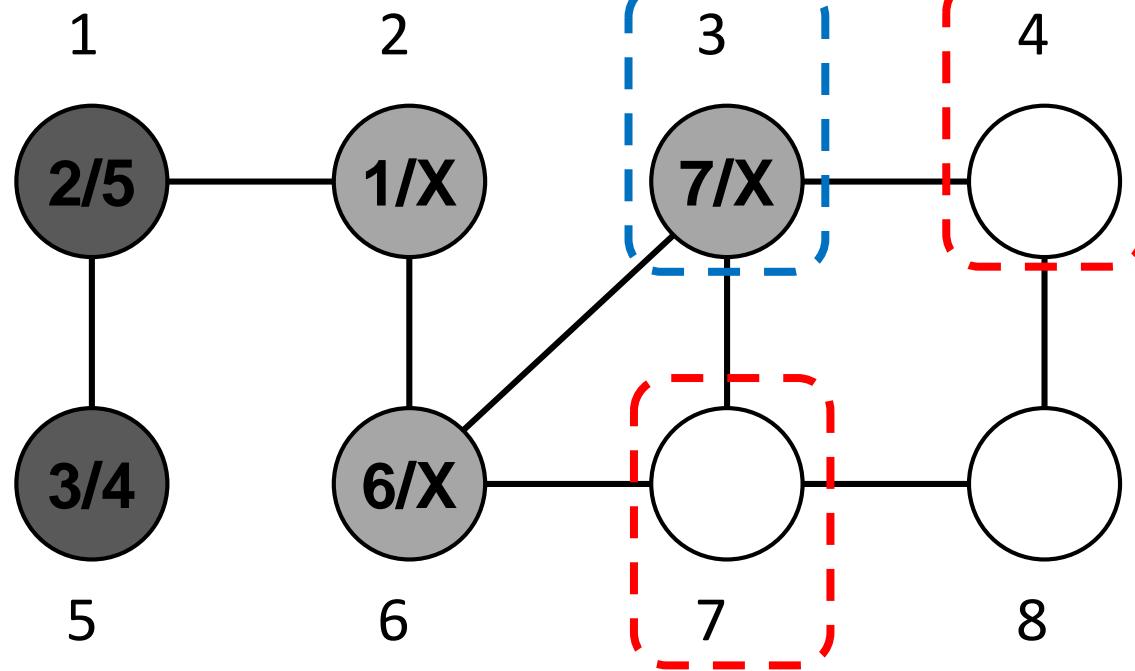
2	1	7	∞	3	6	∞	∞
---	---	---	----------	---	---	----------	----------

pred

2	N	6	N	1	2	N	N
---	---	---	---	---	---	---	---

f

5	∞	∞	∞	4	∞	∞	∞
---	----------	----------	----------	---	----------	----------	----------



DFS Example

time = 8

color

B	G	G	W	B	G	W	W
---	---	---	---	---	---	---	---

d

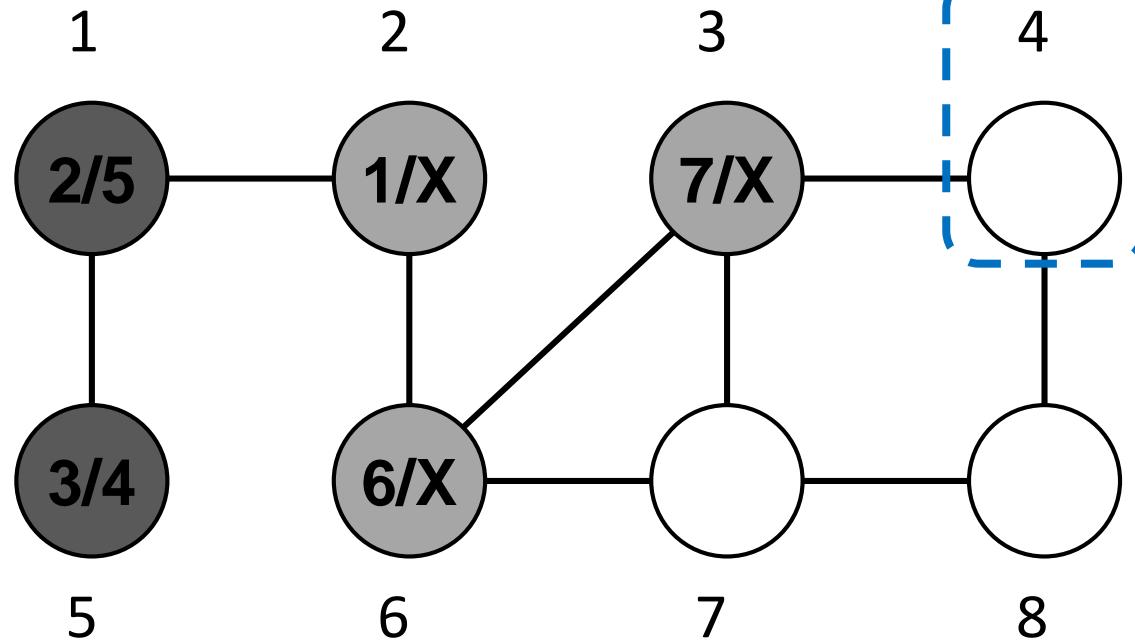
2	1	7	∞	3	6	∞	∞
---	---	---	----------	---	---	----------	----------

pred

2	N	6	N	1	2	N	N
---	---	---	---	---	---	---	---

f

5	∞	∞	∞	4	∞	∞	∞
---	----------	----------	----------	---	----------	----------	----------



DFS Example

time = 8

color

B	G	G	G	B	G	W	W
---	---	---	---	---	---	---	---

d

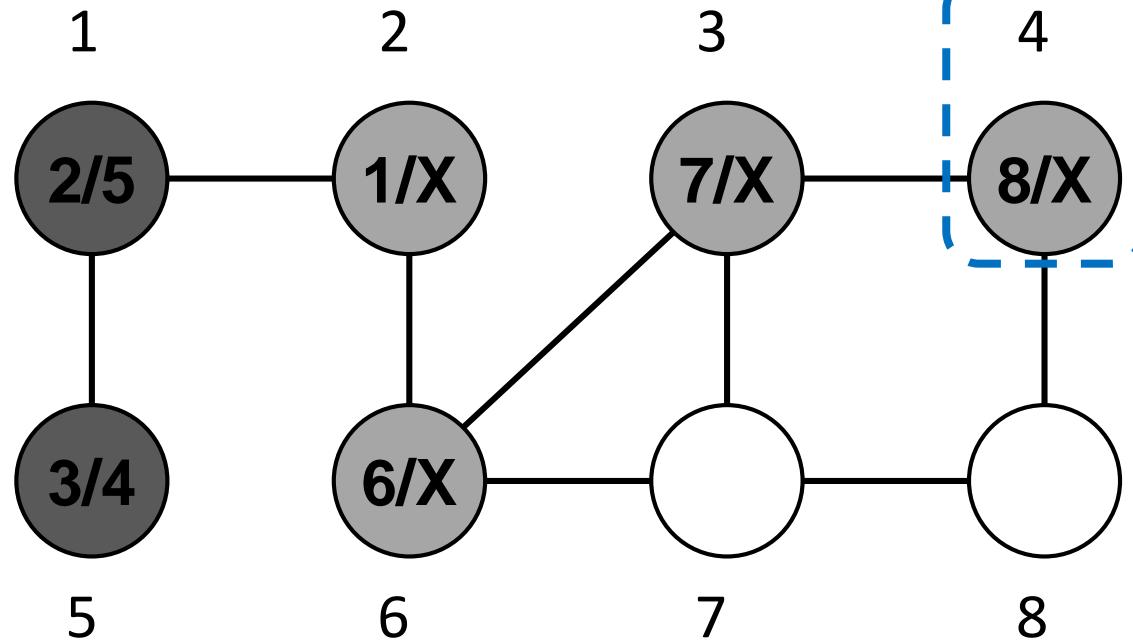
2	1	7	8	3	6	∞	∞
---	---	---	---	---	---	----------	----------

pred

2	N	6	3	1	2	N	N
---	---	---	---	---	---	---	---

f

5	∞	∞	∞	4	∞	∞	∞
---	----------	----------	----------	---	----------	----------	----------



DFS Example

time = 8

color

B	G	G	G	B	G	W	W
---	---	---	---	---	---	---	---

d

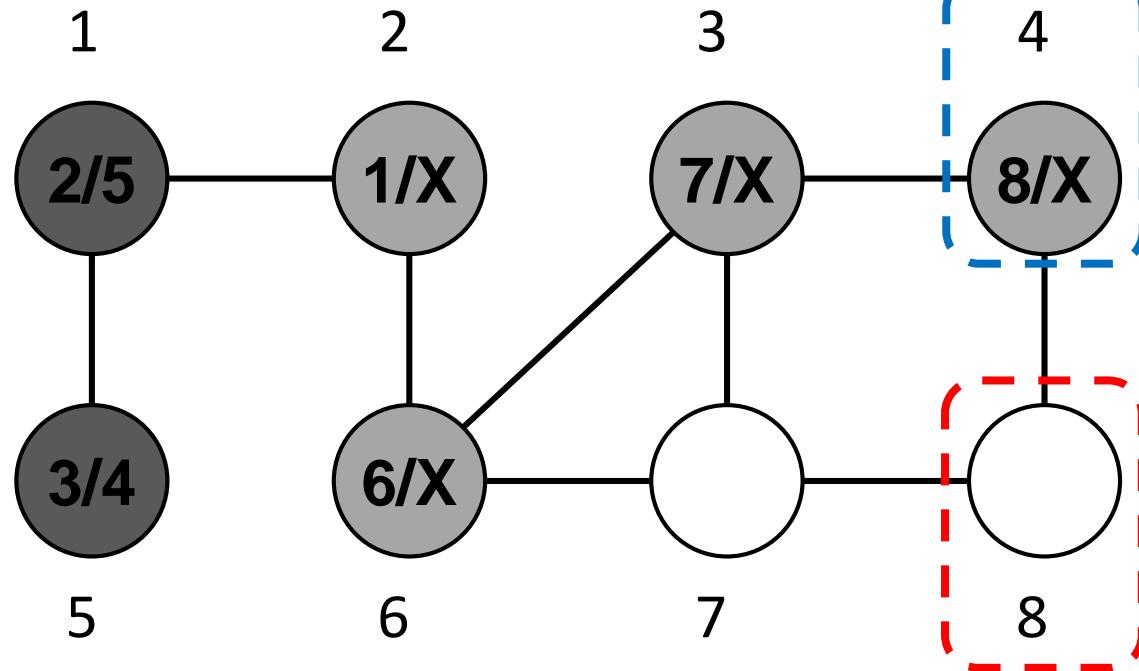
2	1	7	8	3	6	∞	∞
---	---	---	---	---	---	----------	----------

pred

2	N	6	3	1	2	N	N
---	---	---	---	---	---	---	---

f

5	∞	∞	∞	4	∞	∞	∞
---	----------	----------	----------	---	----------	----------	----------



DFS Example

time = 9

color

B	G	G	G	B	G	W	W
---	---	---	---	---	---	---	---

d

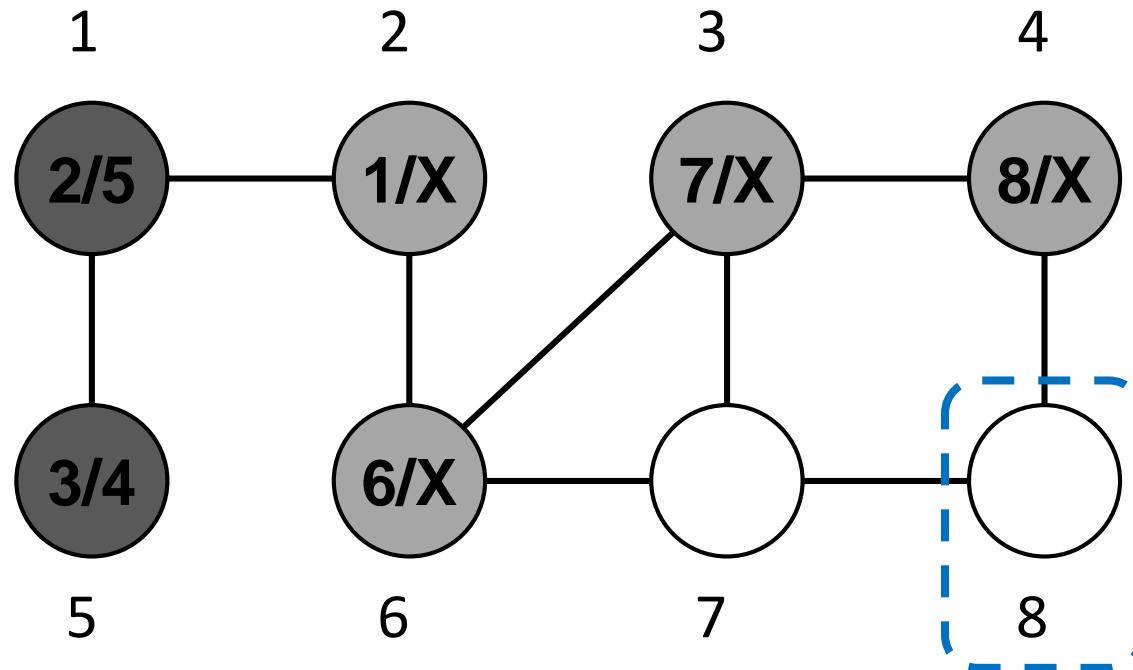
2	1	7	8	3	6	∞	∞
---	---	---	---	---	---	----------	----------

pred

2	N	6	3	1	2	N	N
---	---	---	---	---	---	---	---

f

5	∞	∞	∞	4	∞	∞	∞
---	----------	----------	----------	---	----------	----------	----------



DFS Example

time = 9

color

B	G	G	G	B	G	W	G
---	---	---	---	---	---	---	----------

d

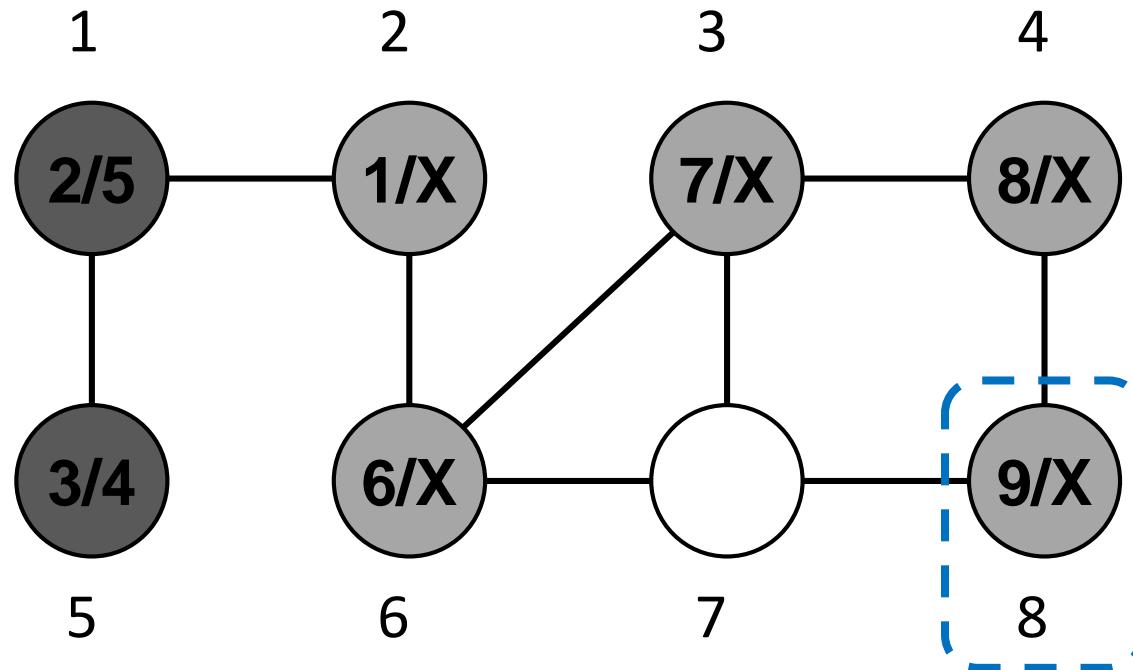
2	1	7	8	3	6	∞	9
---	---	---	---	---	---	----------	----------

pred

2	N	6	3	1	2	N	4
---	---	---	---	---	---	---	----------

f

5	∞	∞	∞	4	∞	∞	∞
---	----------	----------	----------	---	----------	----------	----------



DFS Example

time = 9

color

B	G	G	G	B	G	W	G
---	---	---	---	---	---	---	---

d

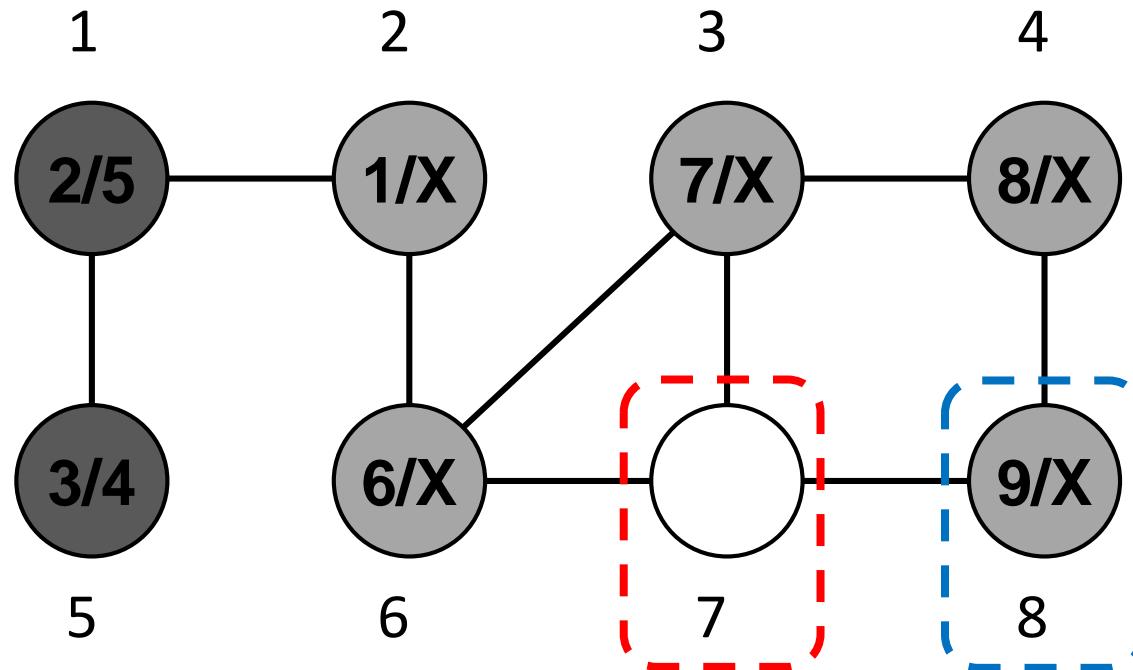
2	1	7	8	3	6	∞	9
---	---	---	---	---	---	----------	---

pred

2	N	6	3	1	2	N	4
---	---	---	---	---	---	---	---

f

5	∞	∞	∞	4	∞	∞	∞
---	----------	----------	----------	---	----------	----------	----------



DFS Example

time = 10

color

B	G	G	G	B	G	W	G
---	---	---	---	---	---	---	---

d

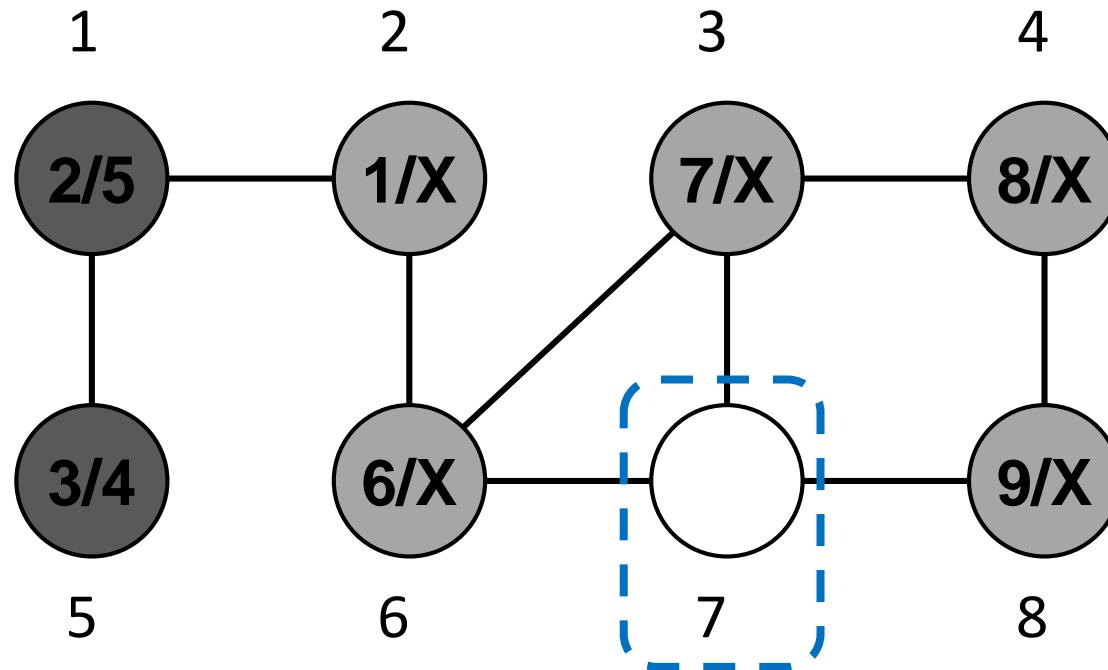
2	1	7	8	3	6	∞	9
---	---	---	---	---	---	----------	---

pred

2	N	6	3	1	2	N	4
---	---	---	---	---	---	---	---

f

5	∞	∞	∞	4	∞	∞	∞
---	----------	----------	----------	---	----------	----------	----------



DFS Example

time = 10

color

B	G	G	G	B	G	G	G
---	---	---	---	---	---	----------	---

d

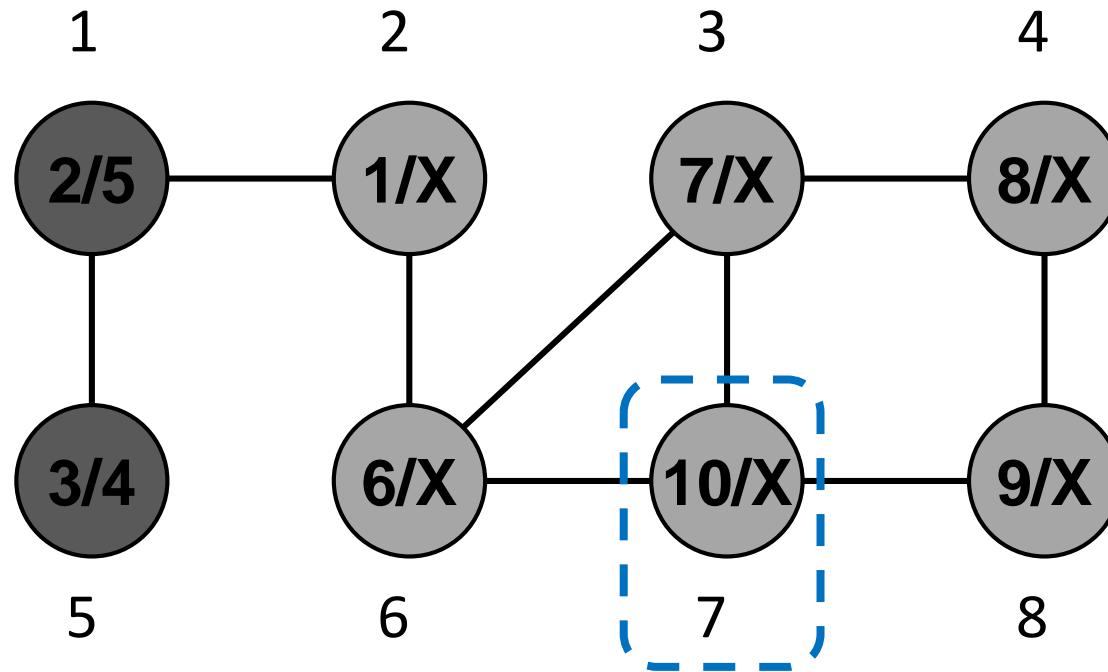
2	1	7	8	3	6	10	9
---	---	---	---	---	---	-----------	---

pred

2	N	6	3	1	2	8	4
---	---	---	---	---	---	----------	---

f

5	∞	∞	∞	4	∞	∞	∞
---	----------	----------	----------	---	----------	----------	----------



DFS Example

time = 10

color

B	G	G	G	B	G	G	G
---	---	---	---	---	---	---	---

d

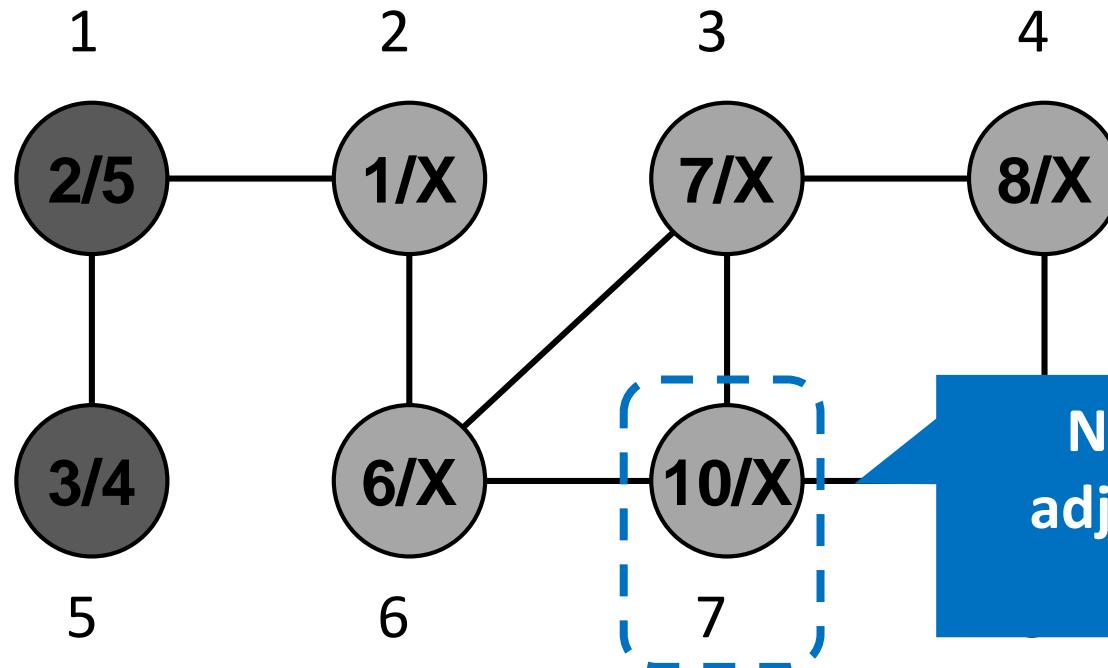
2	1	7	8	3	6	10	9
---	---	---	---	---	---	----	---

pred

2	N	6	3	1	2	8	4
---	---	---	---	---	---	---	---

f

5	∞	∞	∞	4	∞	∞	∞
---	----------	----------	----------	---	----------	----------	----------



None of its
adjacencies is
WHITE.

DFS Example

time = 11

color

B	G	G	G	B	G	G	G
---	---	---	---	---	---	---	---

d

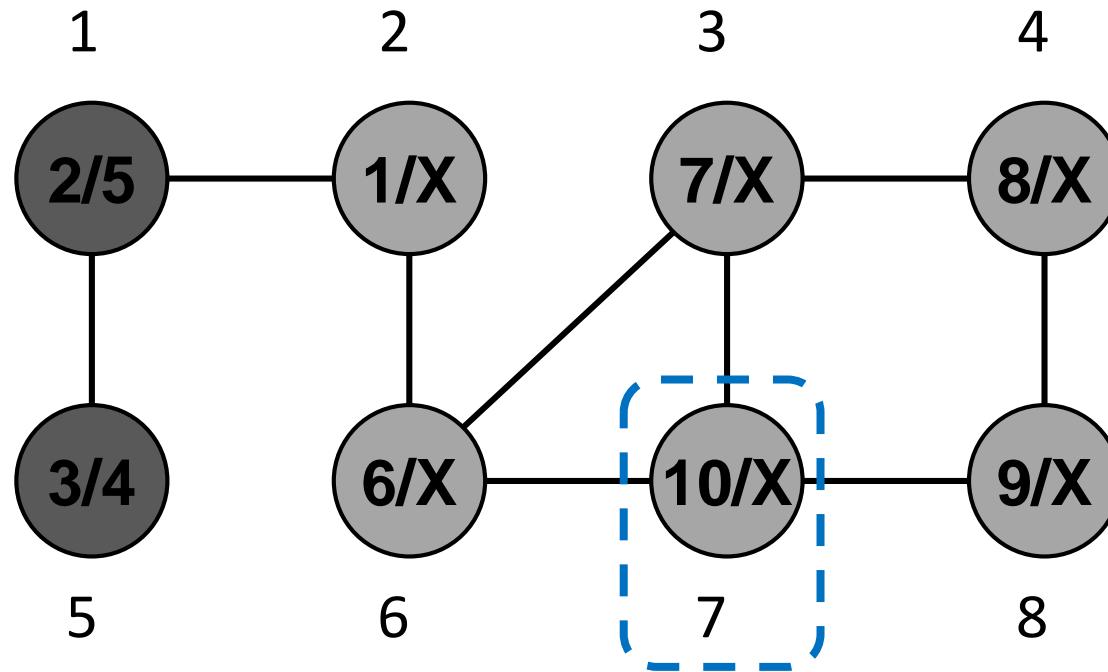
2	1	7	8	3	6	10	9
---	---	---	---	---	---	----	---

pred

2	N	6	3	1	2	8	4
---	---	---	---	---	---	---	---

f

5	∞	∞	∞	4	∞	∞	∞
---	----------	----------	----------	---	----------	----------	----------



DFS Example

time = 11

color

B	G	G	G	B	G	B	G
---	---	---	---	---	---	----------	---

d

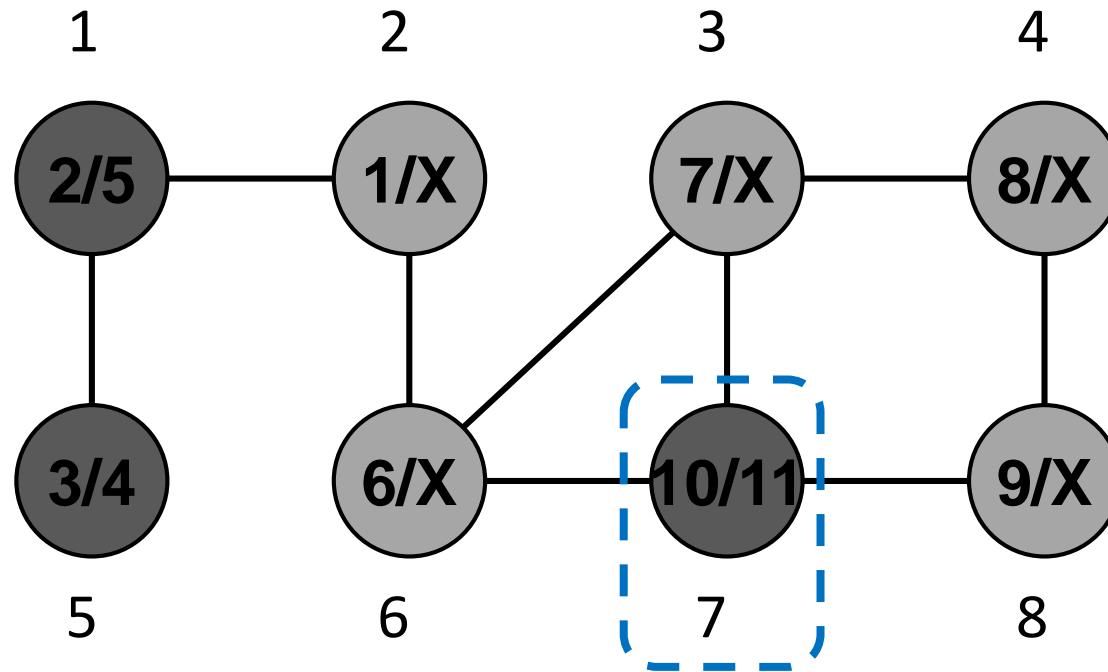
2	1	7	8	3	6	10	9
---	---	---	---	---	---	-----------	---

pred

2	N	6	3	1	2	8	4
---	---	---	---	---	---	---	---

f

5	∞	∞	∞	4	∞	11	∞
---	----------	----------	----------	---	----------	-----------	----------



DFS Example

time = 11

color

B	G	G	G	B	G	B	G
---	---	---	---	---	---	---	---

d

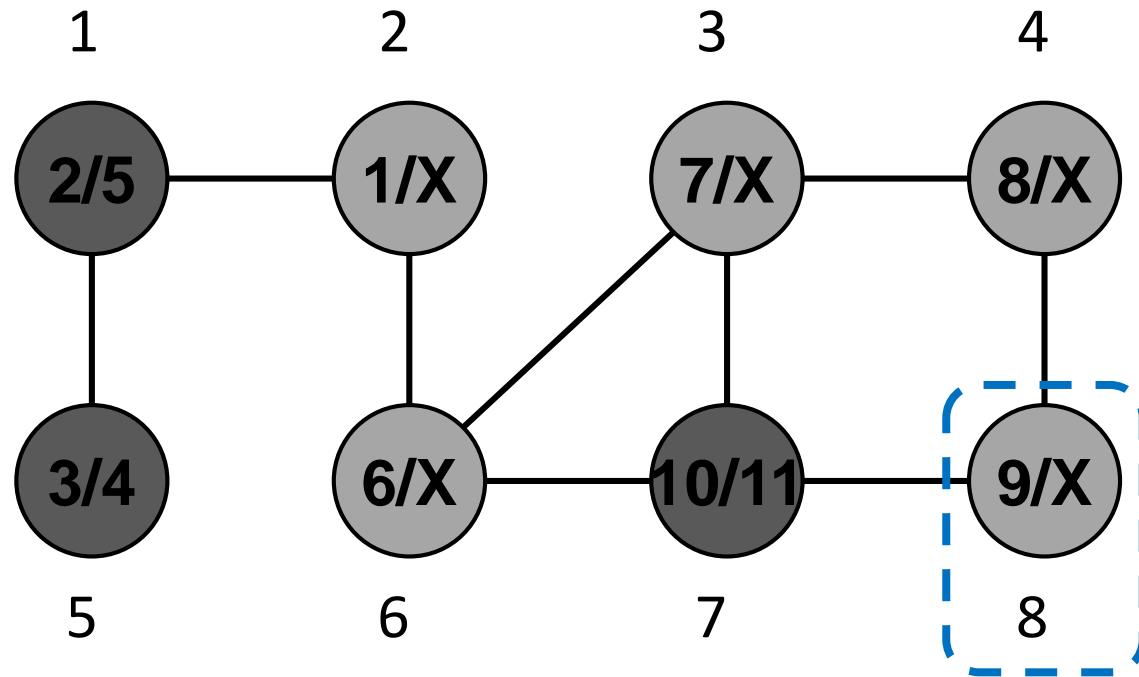
2	1	7	8	3	6	10	9
---	---	---	---	---	---	----	---

pred

2	N	6	3	1	2	8	4
---	---	---	---	---	---	---	---

f

5	∞	∞	∞	4	∞	11	∞
---	----------	----------	----------	---	----------	----	----------



DFS Example

time = 12

color

B	G	G	G	B	G	B	B
---	---	---	---	---	---	---	---

d

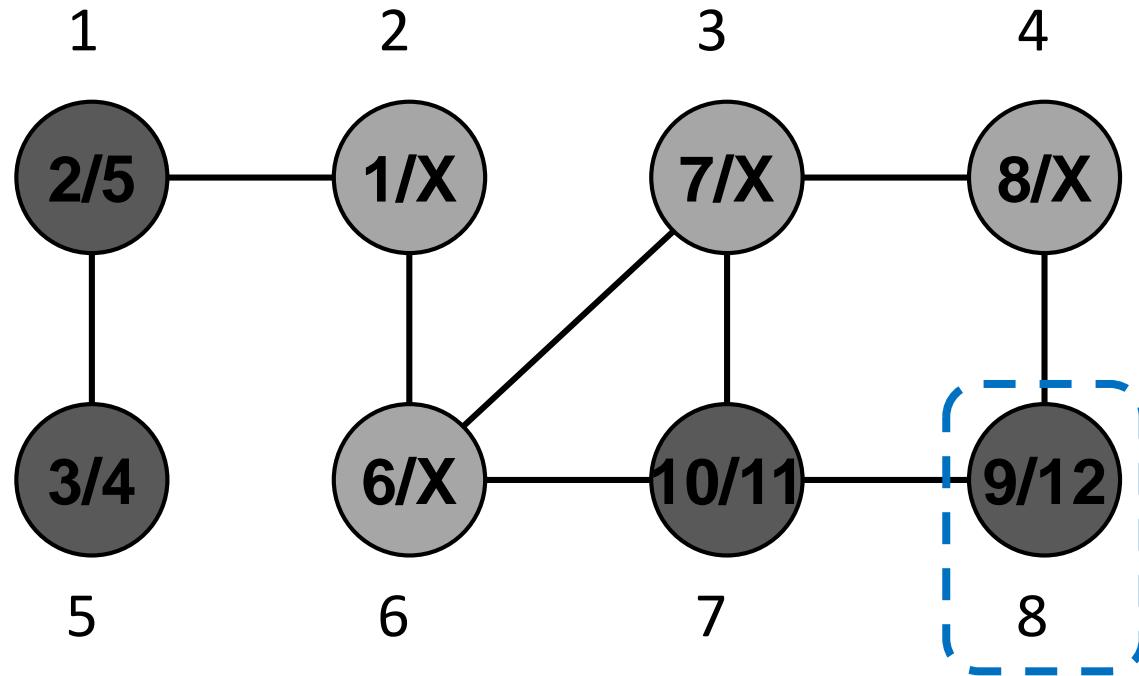
2	1	7	8	3	6	10	9
---	---	---	---	---	---	----	---

pred

2	N	6	3	1	2	8	4
---	---	---	---	---	---	---	---

f

5	∞	∞	∞	4	∞	11	12
---	----------	----------	----------	---	----------	----	----



DFS Example

time = 12

color

B	G	G	G	B	G	B	B
---	---	---	---	---	---	---	---

d

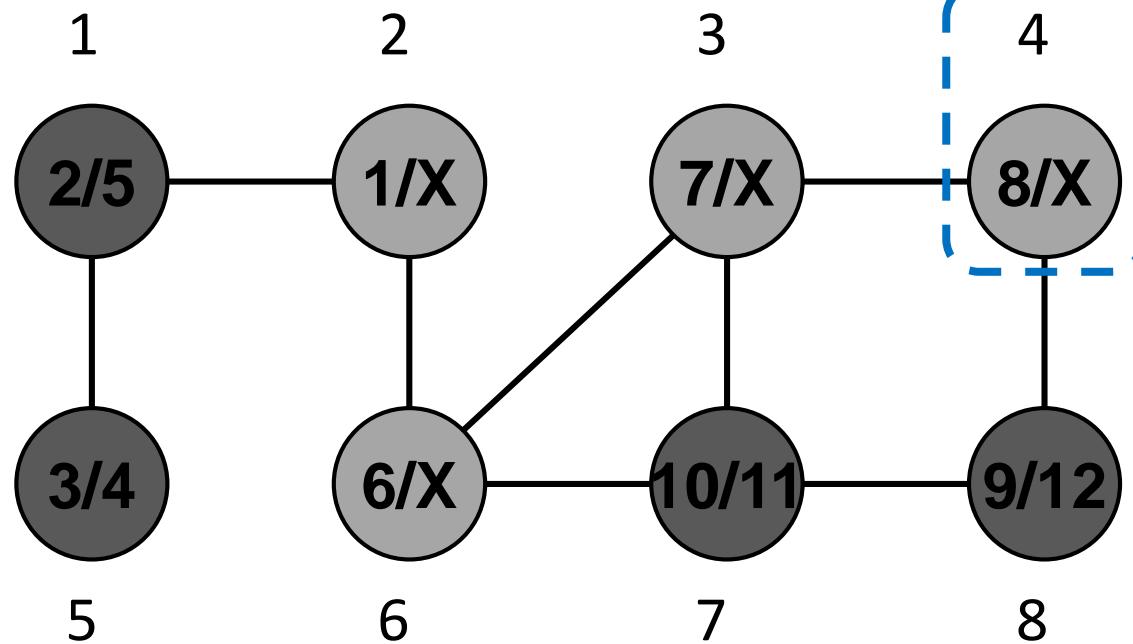
2	1	7	8	3	6	10	9
---	---	---	---	---	---	----	---

pred

2	N	6	3	1	2	8	4
---	---	---	---	---	---	---	---

f

5	∞	∞	∞	4	∞	11	12
---	----------	----------	----------	---	----------	----	----



DFS Example

time = 13

color

B	G	G	B	B	G	B	B
---	---	---	---	---	---	---	---

d

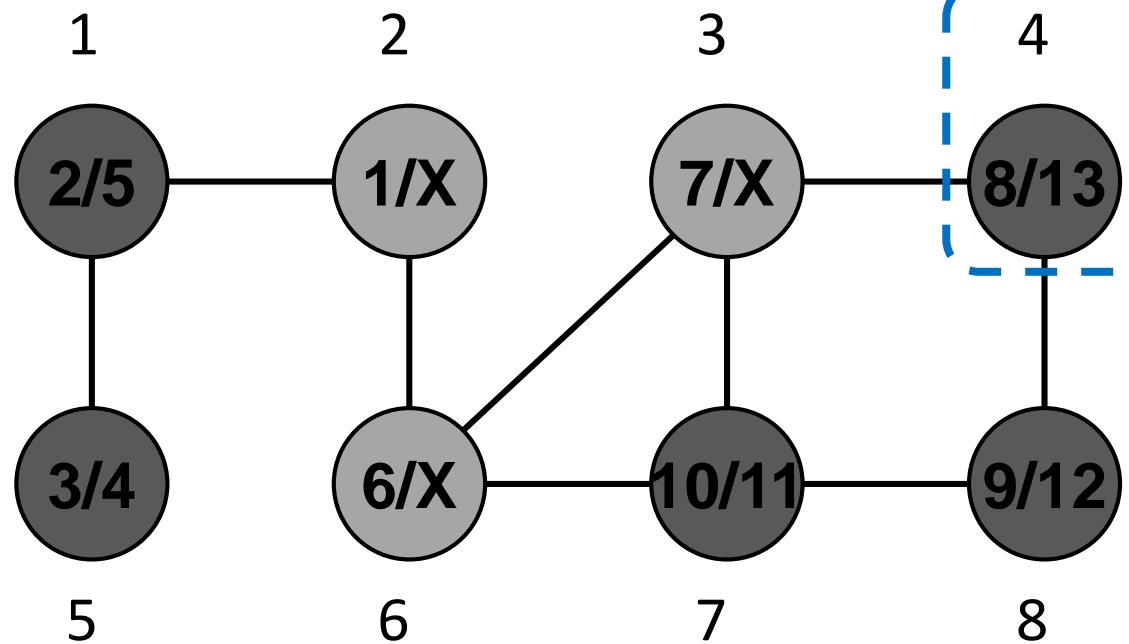
2	1	7	8	3	6	10	9
---	---	---	---	---	---	----	---

pred

2	N	6	3	1	2	8	4
---	---	---	---	---	---	---	---

f

5	∞	∞	13	4	∞	11	12
---	----------	----------	----	---	----------	----	----



DFS Example

time = 13

color

B	G	G	B	B	G	B	B
---	---	---	---	---	---	---	---

d

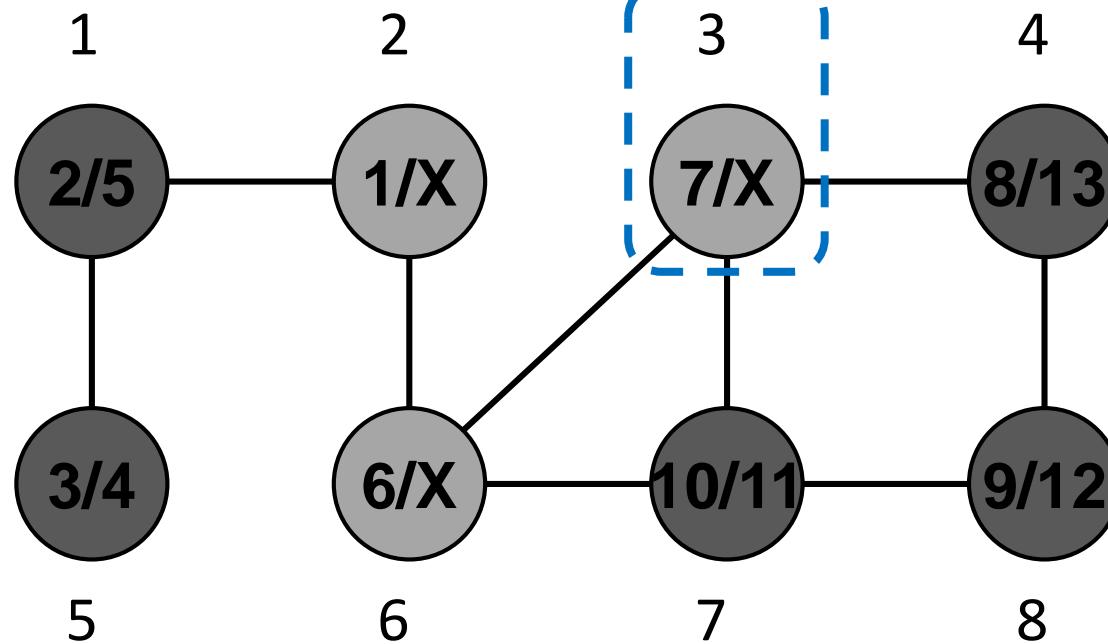
2	1	7	8	3	6	10	9
---	---	---	---	---	---	----	---

pred

2	N	6	3	1	2	8	4
---	---	---	---	---	---	---	---

f

5	∞	∞	13	4	∞	11	12
---	----------	----------	----	---	----------	----	----



DFS Example

time = 14

color

B	G	B	B	B	G	B	B
---	---	---	---	---	---	---	---

d

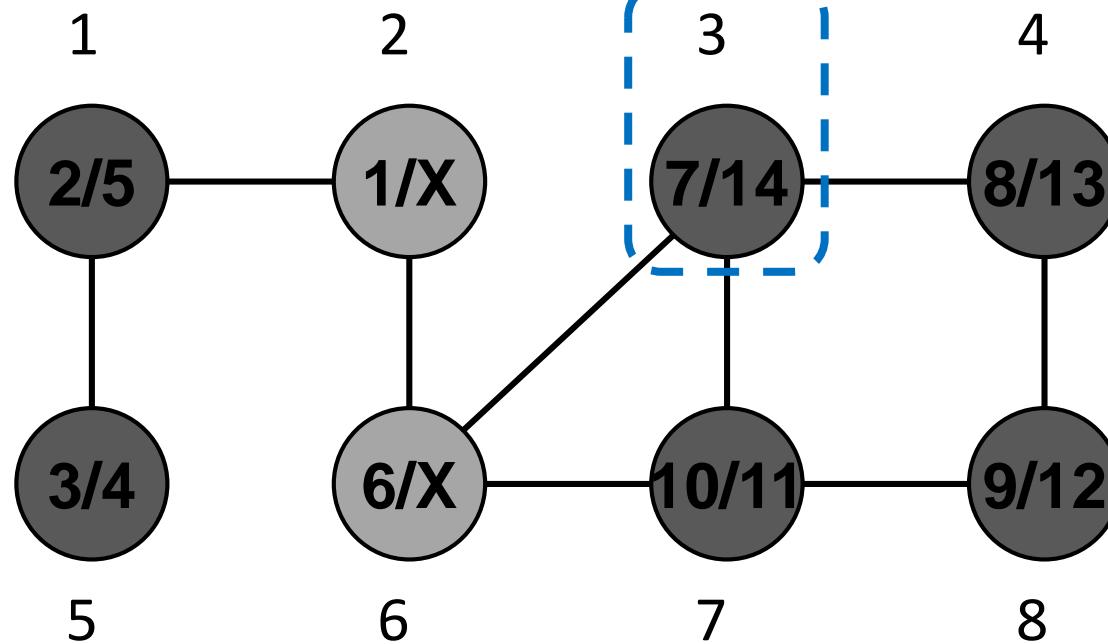
2	1	7	8	3	6	10	9
---	---	---	---	---	---	----	---

pred

2	N	6	3	1	2	8	4
---	---	---	---	---	---	---	---

f

5	∞	14	13	4	∞	11	12
---	----------	----	----	---	----------	----	----



DFS Example

time = 14

color

B	G	B	B	B	G	B	B
---	---	---	---	---	---	---	---

d

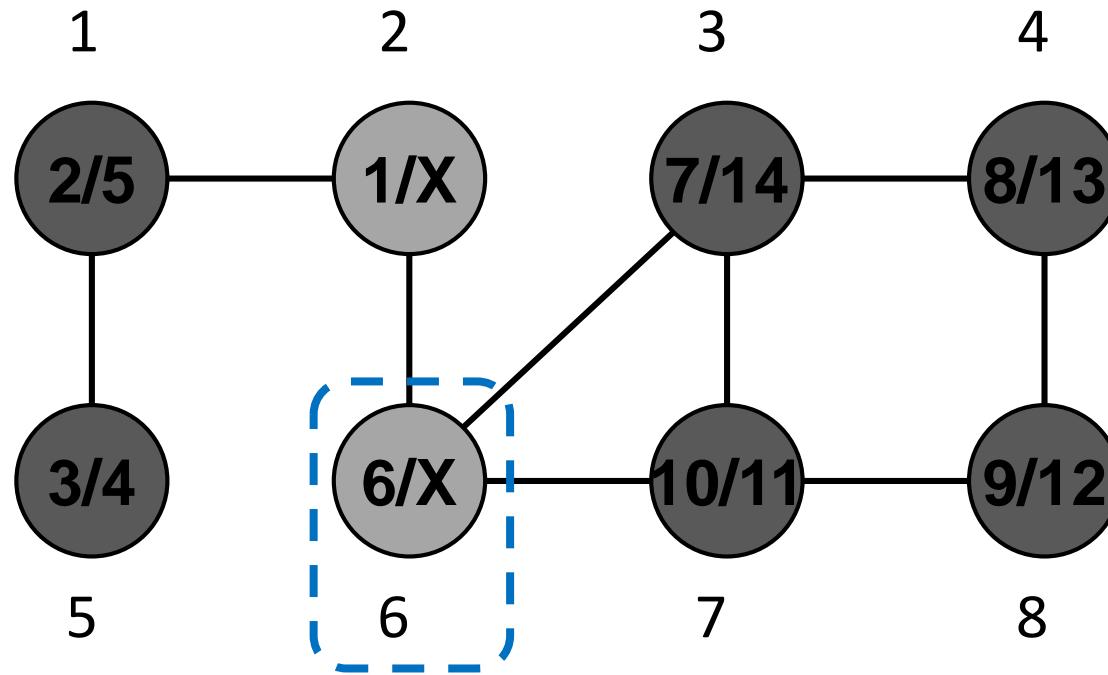
2	1	7	8	3	6	10	9
---	---	---	---	---	---	----	---

pred

2	N	6	3	1	2	8	4
---	---	---	---	---	---	---	---

f

5	∞	14	13	4	∞	11	12
---	----------	----	----	---	----------	----	----



DFS Example

time = 15

color

B	G	B	B	B	B	B	B
---	---	---	---	---	----------	---	---

d

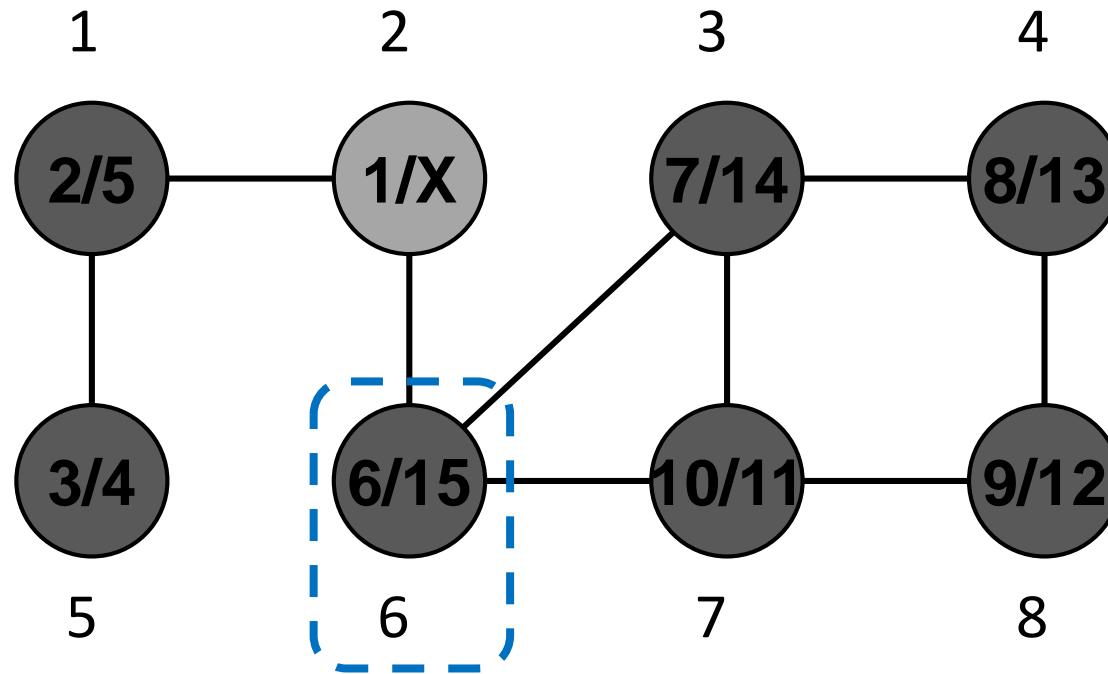
2	1	7	8	3	6	10	9
---	---	---	---	---	---	-----------	---

pred

2	N	6	3	1	2	8	4
---	---	---	---	---	---	---	---

f

5	∞	14	13	4	15	11	12
---	----------	----	----	---	-----------	----	----



DFS Example

time = 15

color

B	G	B	B	B	B	B	B
---	---	---	---	---	---	---	---

d

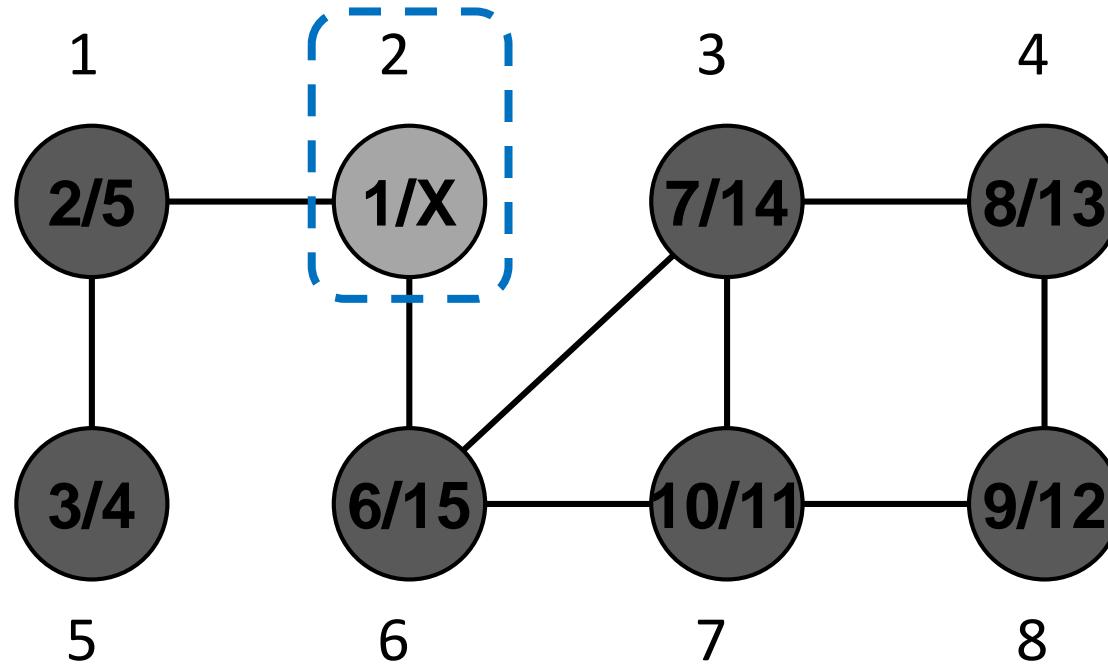
2	1	7	8	3	6	10	9
---	---	---	---	---	---	----	---

pred

2	N	6	3	1	2	8	4
---	---	---	---	---	---	---	---

f

5	∞	14	13	4	15	11	12
---	----------	----	----	---	----	----	----



DFS Example

time = 16

color

B	B	B	B	B	B	B	B
---	---	---	---	---	---	---	---

d

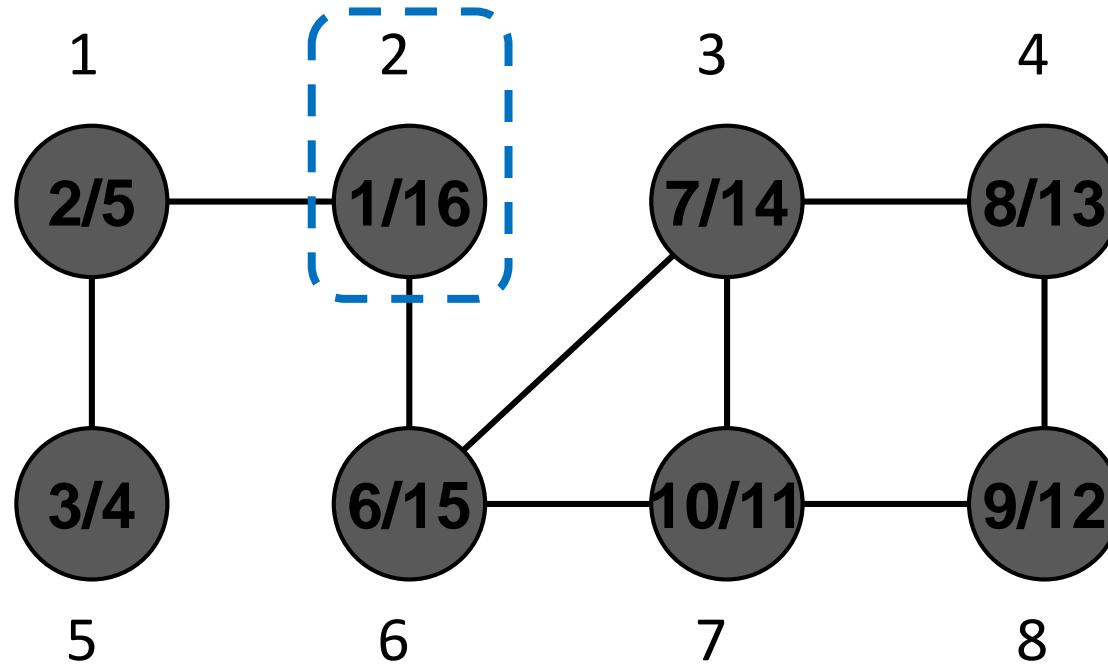
2	1	7	8	3	6	10	9
---	---	---	---	---	---	----	---

pred

2	N	6	3	1	2	8	4
---	---	---	---	---	---	---	---

f

5	16	14	13	4	15	11	12
---	----	----	----	---	----	----	----



DFS Example

time = 16

color

B	B	B	B	B	B	B	B
---	---	---	---	---	---	---	---

d

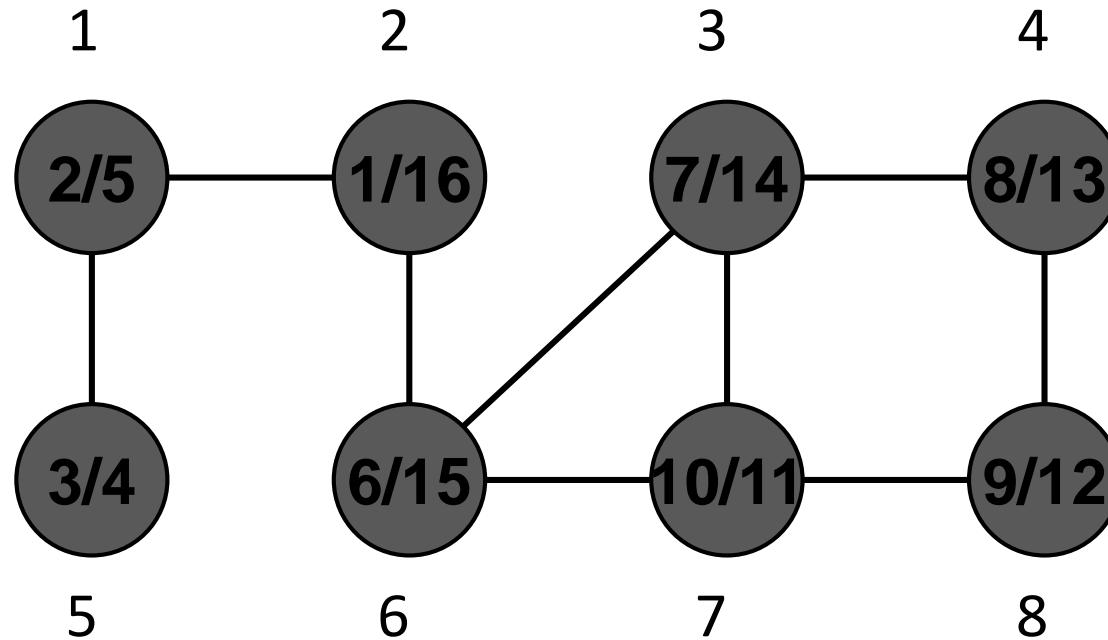
2	1	7	8	3	6	10	9
---	---	---	---	---	---	----	---

pred

2	N	6	3	1	2	8	4
---	---	---	---	---	---	---	---

f

5	16	14	13	4	15	11	12
---	----	----	----	---	----	----	----



Running Time of DFS

- The procedure `DFSVisit` is called exactly once for each vertex $u \in V$

Running Time of DFS

- The procedure `DFSVisit` is called exactly once for each vertex $u \in V$
 - since `DFSVisit` is invoked only on white vertices and the first thing it does is paint the vertex gray

Running Time of DFS

- The procedure `DFSVisit` is called exactly once for each vertex $u \in V$
 - since `DFSVisit` is invoked only on white vertices and the first thing it does is paint the vertex gray
- During an execution of `DFSVisit(u)`, the for loop is executed $|\text{Adj}(u)| = \text{degree}(u)$ times

Running Time of DFS

- The procedure `DFSVisit` is called exactly once for each vertex $u \in V$
 - since `DFSVisit` is invoked only on white vertices and the first thing it does is paint the vertex gray
- During an execution of `DFSVisit(u)`, the for loop is executed $|\text{Adj}(u)| = \text{degree}(u)$ times

On each vertex u , we spend time $T_u = O(1 + \text{degree}(u))$

Running Time of DFS

- The procedure `DFSVisit` is called exactly once for each vertex $u \in V$
 - since `DFSVisit` is invoked only on white vertices and the first thing it does is paint the vertex gray
- During an execution of `DFSVisit(u)`, the for loop is executed $|\text{Adj}(u)| = \text{degree}(u)$ times

On each vertex u , we spend time $T_u = O(1 + \text{degree}(u))$

The total running time is

$$\sum_{u \in V} T_u \leq \sum_{u \in V} O(1 + \text{degree}(u)) = O(V + E)$$

Running Time of DFS

- The procedure `DFSVisit` is called exactly once for each vertex $u \in V$
 - since `DFSVisit` is invoked only on white vertices and the first thing it does is paint the vertex gray
- During an execution of `DFSVisit(u)`, the for loop is executed $|\text{Adj}(u)| = \text{degree}(u)$ times

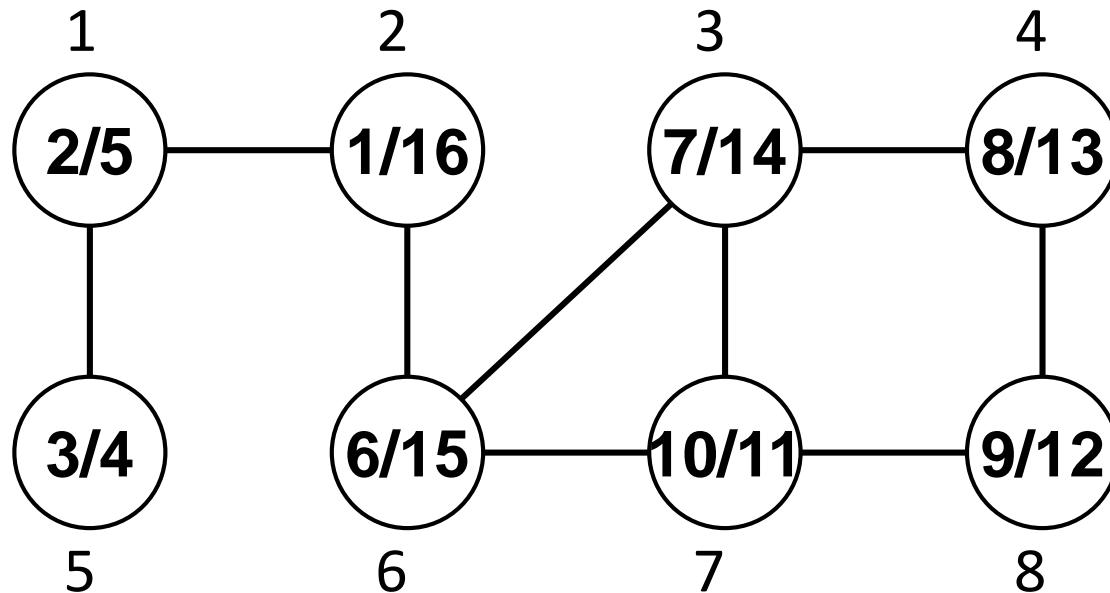
On each vertex u , we spend time $T_u = O(1 + \text{degree}(u))$

The total running time is

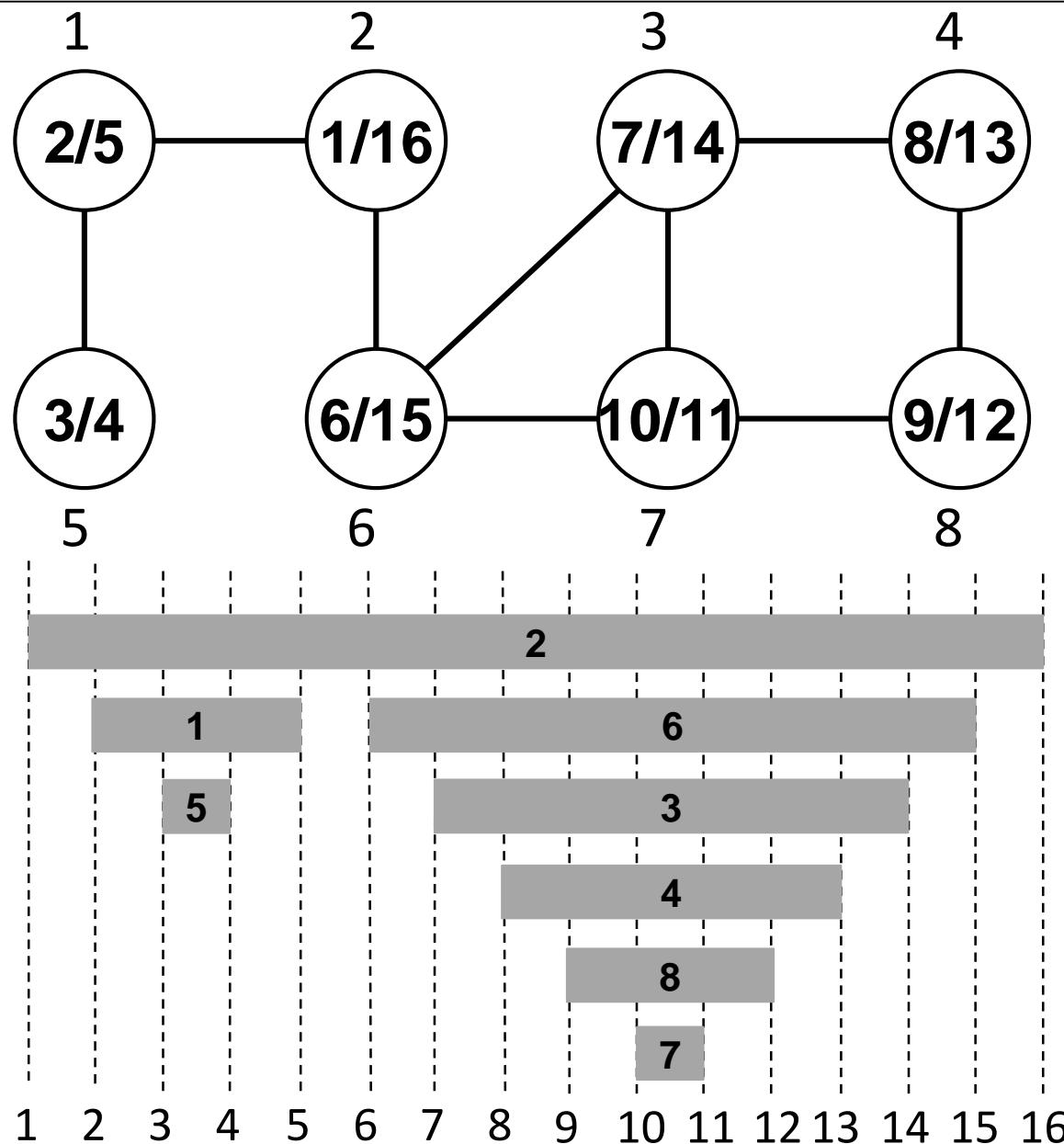
$$\sum_{u \in V} T_u \leq \sum_{u \in V} O(1 + \text{degree}(u)) = O(V + E)$$

Hence, the running of DFS on a graph with V vertices and E edges is **O(V + E)**

Time-Stamp (Parenthesis) Structure



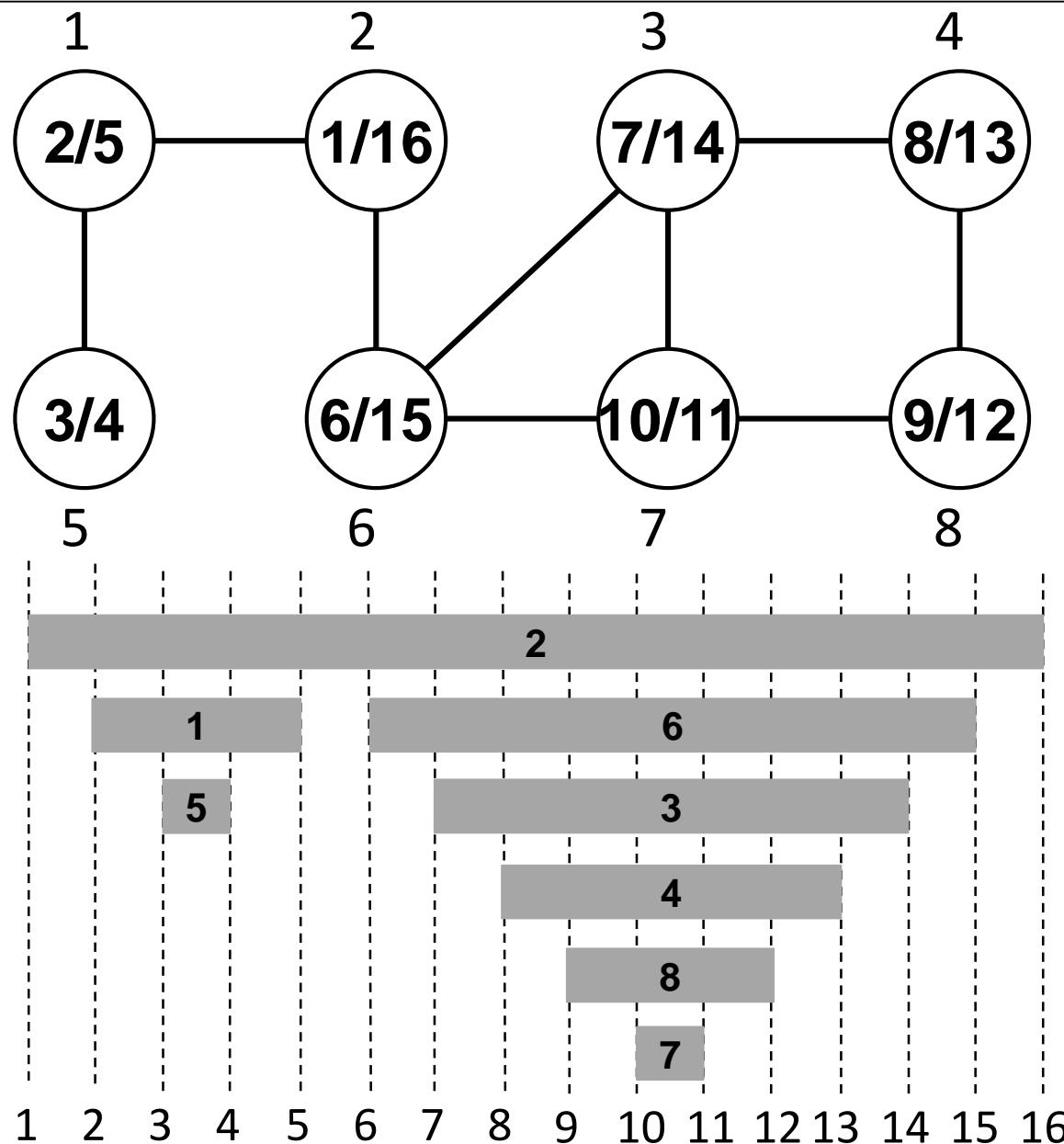
Time-Stamp (Parenthesis) Structure



Time-Stamp (Parenthesis) Structure

- u is a **descendant** (in DFS trees) of v , if and only if $[d[u], f[u]]$ is a **subinterval** of $[d[v], f[v]]$

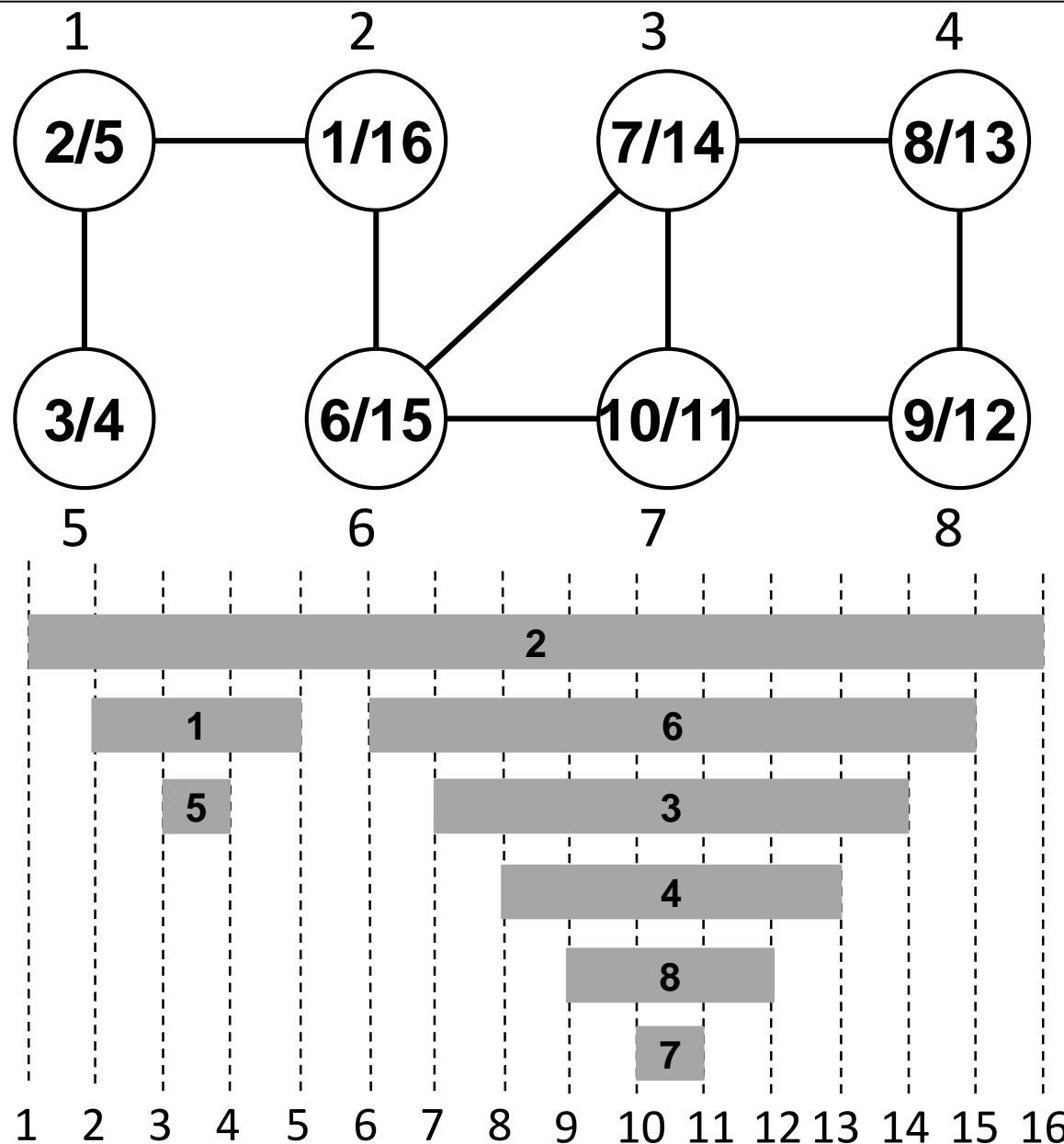
Time-Stamp (Parenthesis) Structure



Time-Stamp (Parenthesis) Structure

- u is a **descendant** (in DFS trees) of v , if and only if $[d[u], f[u]]$ is a **subinterval** of $[d[v], f[v]]$
- u is an **ancestor** of v , if and only if $[d[u], f[u]]$ **contains** $[d[v], f[v]]$

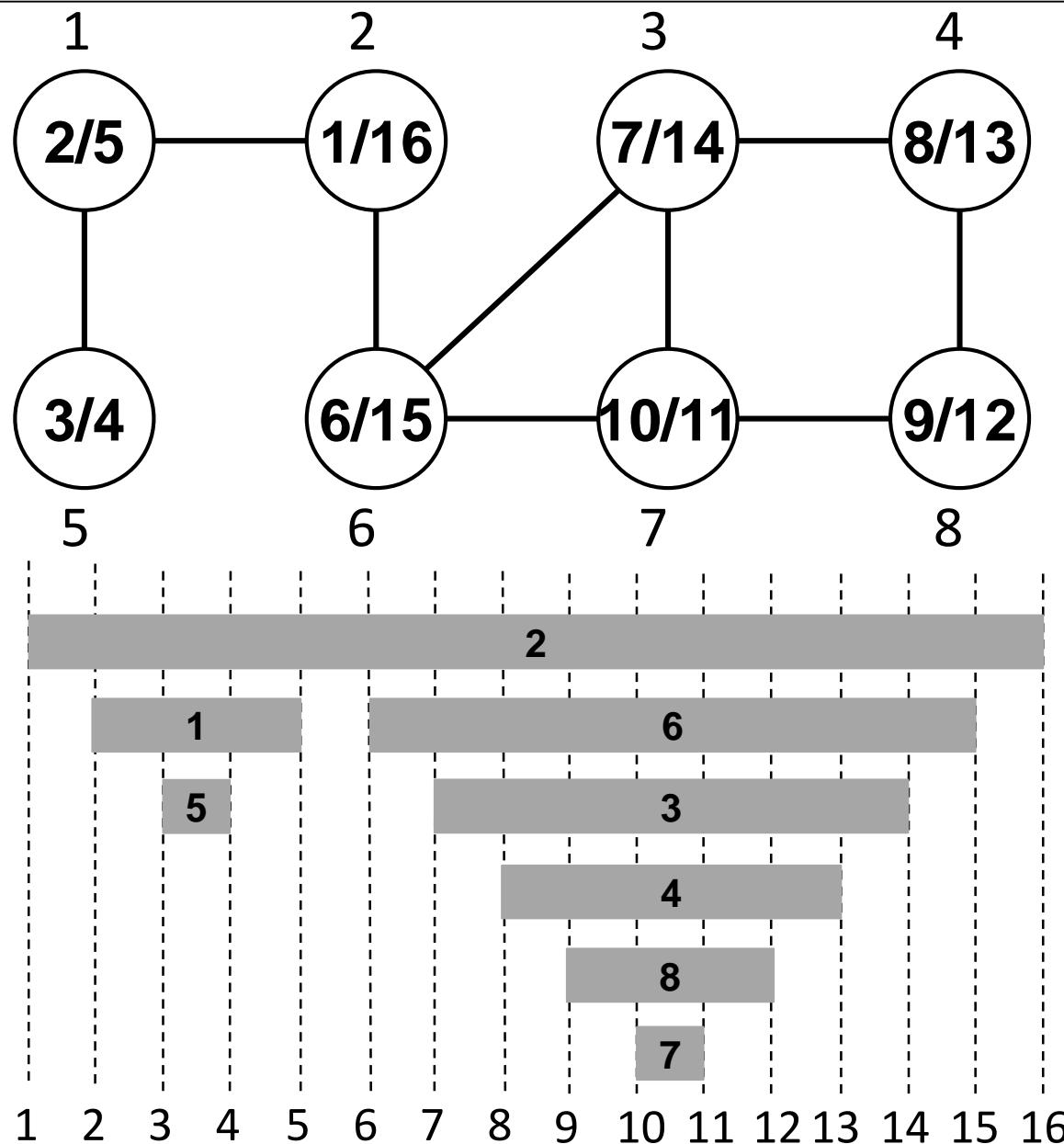
Time-Stamp (Parenthesis) Structure



Time-Stamp (Parenthesis) Structure

- u is a **descendant** (in DFS trees) of v , if and only if $[d[u], f[u]]$ is a **subinterval** of $[d[v], f[v]]$
- u is an **ancestor** of v , if and only if $[d[u], f[u]]$ **contains** $[d[v], f[v]]$
- u is **unrelated** to v , if and only if $[d[u], f[u]]$ and $[d[v], f[v]]$ are **disjoint** intervals

Time-Stamp (Parenthesis) Structure



Proof

The idea is to consider every case

Proof

The idea is to consider every case

We first consider $d[v] < d[u]$

Proof

The idea is to consider every case

We first consider $d[v] < d[u]$

- If $f[v] > d[u]$, then
 - u is discovered when v is still not finished yet (marked gray)
 $\Rightarrow u$ is a descendant of v

Proof

The idea is to consider every case

We first consider $d[v] < d[u]$

- If $f[v] > d[u]$, then
 - u is discovered when v is still not finished yet (marked gray)
 $\Rightarrow u$ is a descendant of v
 - u is discovered later than $v \Rightarrow u$ should finish before v

Proof

The idea is to consider every case

We first consider $d[v] < d[u]$

- If $f[v] > d[u]$, then
 - u is discovered when v is still not finished yet (marked gray)
 $\Rightarrow u$ is a descendant of v
 - u is discovered later than $v \Rightarrow u$ should finish before v
 - Hence we have $[d[u], f[u]]$ is a subinterval of $[d[v], f[v]]$

Proof

The idea is to consider every case

We first consider $d[v] < d[u]$

- If $f[v] > d[u]$, then
 - u is discovered when v is still not finished yet (marked gray)
⇒ u is a descendant of v
 - u is discovered later than v ⇒ u should finish before v
 - Hence we have $[d[u], f[u]]$ is a subinterval of $[d[v], f[v]]$
- If $f[v] < d[u]$, then
 - obviously $[d[v], f[v]]$ and $[d[u], f[u]]$ are disjoint

Proof

The idea is to consider every case

We first consider $d[v] < d[u]$

- If $f[v] > d[u]$, then
 - u is discovered when v is still not finished yet (marked gray)
⇒ u is a descendant of v
 - u is discovered later than $v \Rightarrow u$ should finish before v
 - Hence we have $[d[u], f[u]]$ is a subinterval of $[d[v], f[v]]$
- If $f[v] < d[u]$, then
 - obviously $[d[v], f[v]]$ and $[d[u], f[u]]$ are disjoint
 - It means that when u or v is discovered, the others are not marked gray

Proof

The idea is to consider every case

We first consider $d[v] < d[u]$

- If $f[v] > d[u]$, then
 - u is discovered when v is still not finished yet (marked gray)
⇒ u is a descendant of v
 - u is discovered later than $v \Rightarrow u$ should finish before v
 - Hence we have $[d[u], f[u]]$ is a subinterval of $[d[v], f[v]]$
- If $f[v] < d[u]$, then
 - obviously $[d[v], f[v]]$ and $[d[u], f[u]]$ are disjoint
 - It means that when u or v is discovered, the others are not marked gray
 - Hence neither vertex is a descendant of the other

Proof

The idea is to consider every case

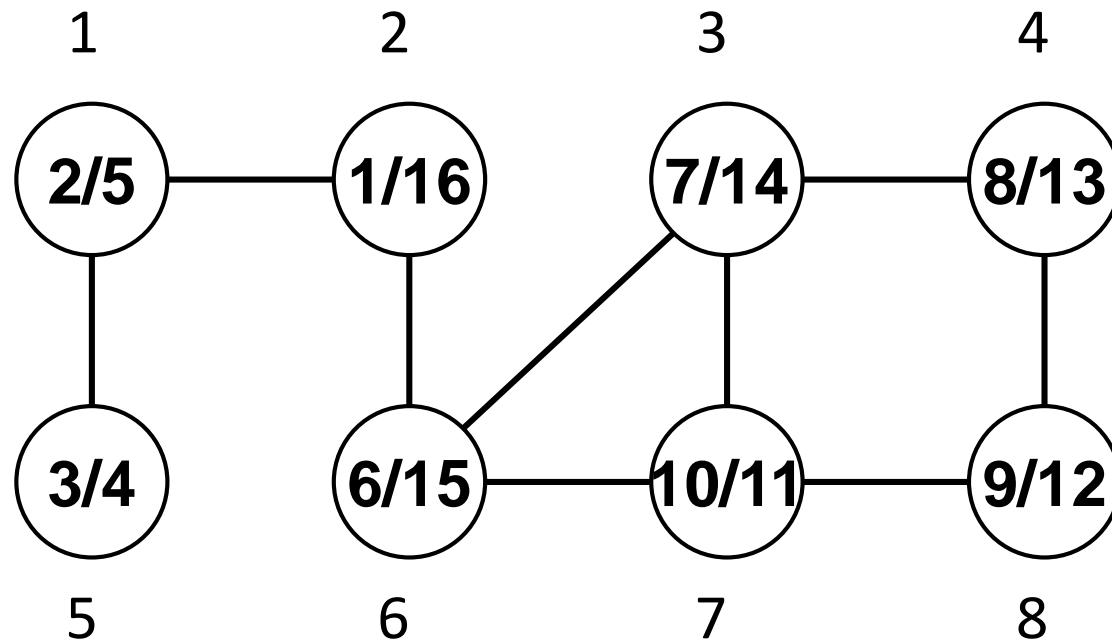
We first consider $d[v] < d[u]$

- If $f[v] > d[u]$, then
 - u is discovered when v is still not finished yet (marked gray)
⇒ u is a descendant of v
 - u is discovered later than $v \Rightarrow u$ should finish before v
 - Hence we have $[d[u], f[u]]$ is a subinterval of $[d[v], f[v]]$
- If $f[v] < d[u]$, then
 - obviously $[d[v], f[v]]$ and $[d[u], f[u]]$ are disjoint
 - It means that when u or v is discovered, the others are not marked gray
 - Hence neither vertex is a descendant of the other

The argument for other case, where $d[v] > d[u]$, is similar

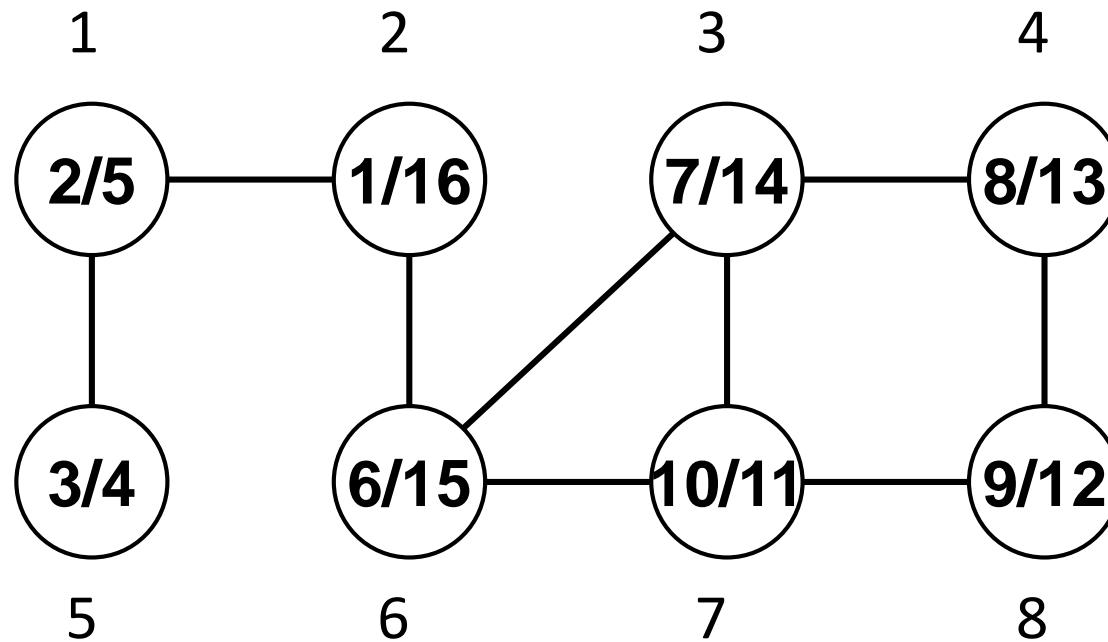
Tree Structure

- Undirected graph $G = (V, E)$, DFS forest $F = (V, E_f)$



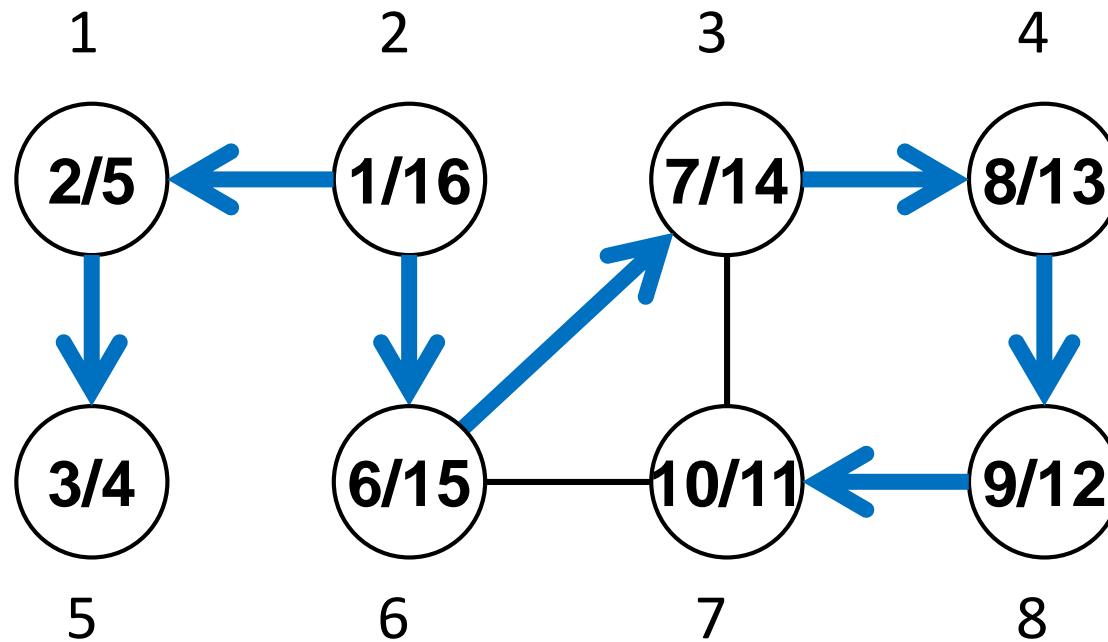
Tree Structure

- Undirected graph $G = (V, E)$, DFS forest $F = (V, E_f)$
- Consider $(u, v) \in E$
 - **tree edge**: if $(u, v) \in E_f$ or equivalently $u = \text{pred}[v]$, i.e. u is the predecessor of v in DFS trees



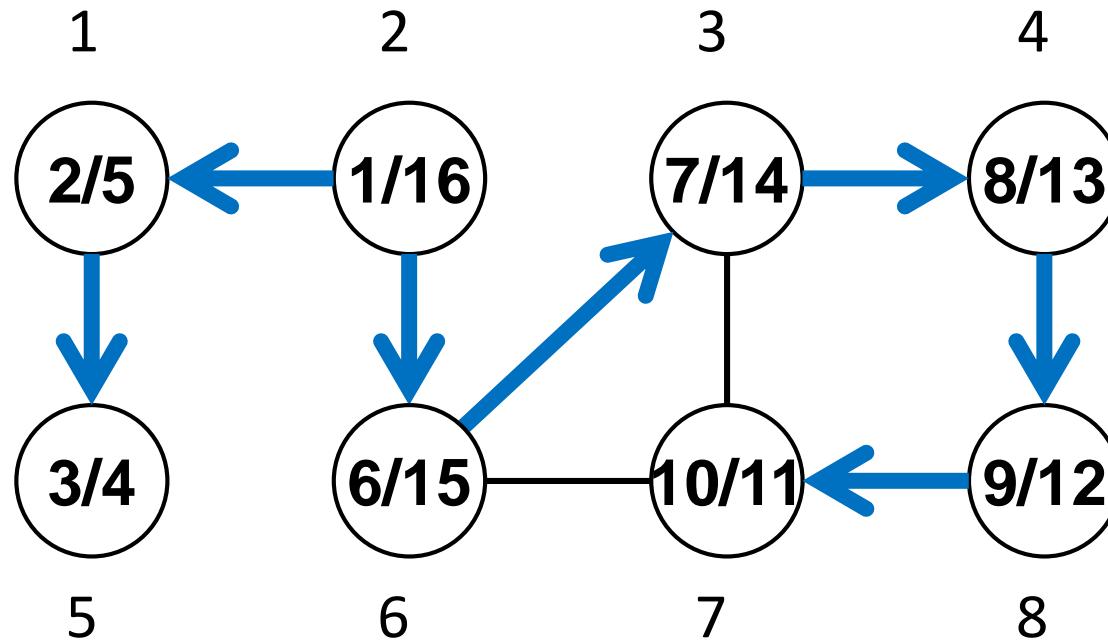
Tree Structure

- Undirected graph $G = (V, E)$, DFS forest $F = (V, E_f)$
- Consider $(u, v) \in E$
 - **tree edge**: if $(u, v) \in E_f$ or equivalently $u = \text{pred}[v]$, i.e. u is the predecessor of v in DFS trees



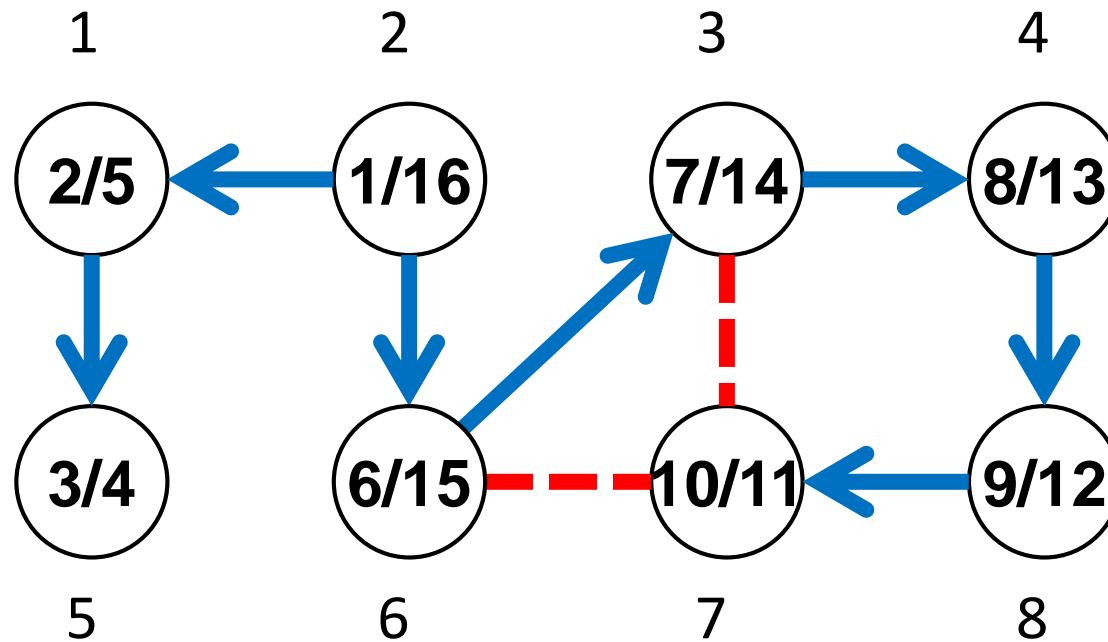
Tree Structure

- Undirected graph $G = (V, E)$, DFS forest $F = (V, E_f)$
- Consider $(u, v) \in E$
 - **tree edge**: if $(u, v) \in E_f$ or equivalently $u = \text{pred}[v]$, i.e. u is the predecessor of v in DFS trees
 - **back edge**: if v is an ancestor (excluding predecessor) of u in the DFS trees



Tree Structure

- Undirected graph $G = (V, E)$, DFS forest $F = (V, E_f)$
- Consider $(u, v) \in E$
 - **tree edge**: if $(u, v) \in E_f$ or equivalently $u = \text{pred}[v]$, i.e. u is the predecessor of v in DFS trees
 - **back edge**: if v is an ancestor (excluding predecessor) of u in the DFS trees



Tree Structure

Theorem

An edge in an undirected graph is either a tree edge or a back edge.

Tree Structure

Theorem

An edge in an undirected graph is either a tree edge or a back edge.

Proof:

- Let (u, v) be an arbitrary edge in an undirected graph G .
- Without loss of generality, assume $d(u) < d(v)$.

Tree Structure

Theorem

An edge in an undirected graph is either a tree edge or a back edge.

Proof:

- Let (u, v) be an arbitrary edge in an undirected graph G .
- Without loss of generality, assume $d(u) < d(v)$.
- Then v is discovered while u is gray.

Tree Structure

Theorem

An edge in an undirected graph is either a tree edge or a back edge.

Proof:

- Let (u, v) be an arbitrary edge in an undirected graph G .
- Without loss of generality, assume $d(u) < d(v)$.
- Then v is discovered while u is gray.
- Hence v is in the DFS subtree rooted at u .

Tree Structure

Theorem

An edge in an undirected graph is either a tree edge or a back edge.

Proof:

- Let (u, v) be an arbitrary edge in an undirected graph G .
- Without loss of generality, assume $d(u) < d(v)$.
- Then v is discovered while u is gray.
- Hence v is in the DFS subtree rooted at u .
 - If $\text{pred}[v] = u$, then (u, v) is a tree edge.

Tree Structure

Theorem

An edge in an undirected graph is either a tree edge or a back edge.

Proof:

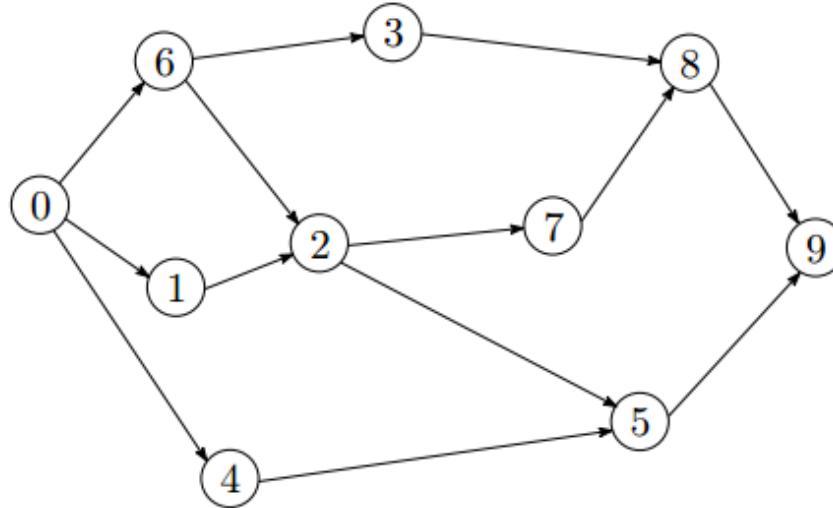
- Let (u, v) be an arbitrary edge in an undirected graph G .
- Without loss of generality, assume $d(u) < d(v)$.
- Then v is discovered while u is gray.
- Hence v is in the DFS subtree rooted at u .
 - If $\text{pred}[v] = u$, then (u, v) is a tree edge.
 - if $\text{prev}[v] \neq u$, then (u, v) is a back edge.

Outline

- Introduction to Part IV
- Review of Basic Graph Search Algorithms
 - Basic Concepts
 - The Breadth-First Search (BFS) Algorithm
 - The Depth-First Search (DFS) Algorithm
- Topological Sort
 - The Topological Sort Algorithm
 - Analysis of the Topological Sort Algorithm
- Strongly Connected Components
 - The Algorithm of Finding SCCs
 - Analysis of the Algorithm

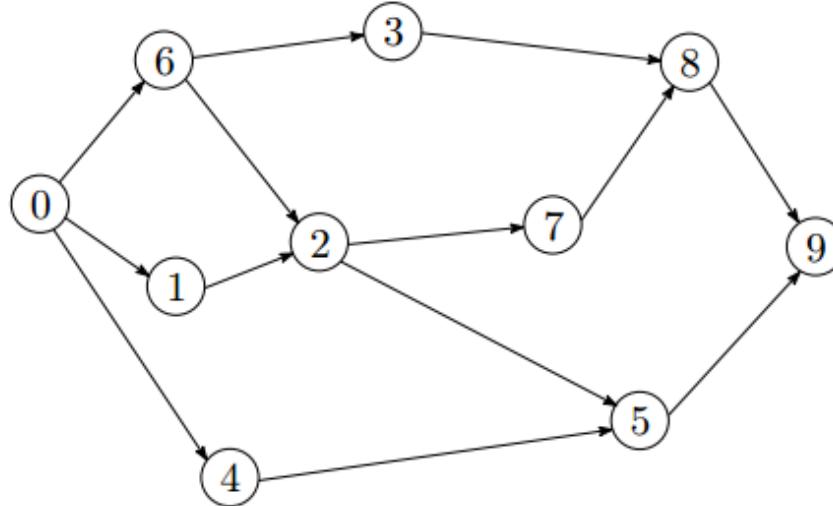
Directed Graph

For **directed graph**, we distinguish between edge (u, v) and edge (v, u)



Directed Graph

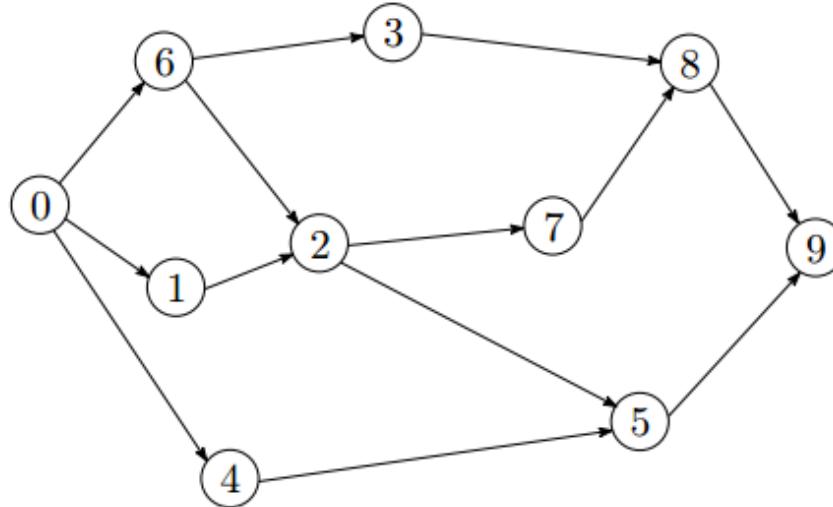
For **directed graph**, we distinguish between edge (u, v) and edge (v, u)



- **out-degree** of a vertex is the number of edges **leaving** it

Directed Graph

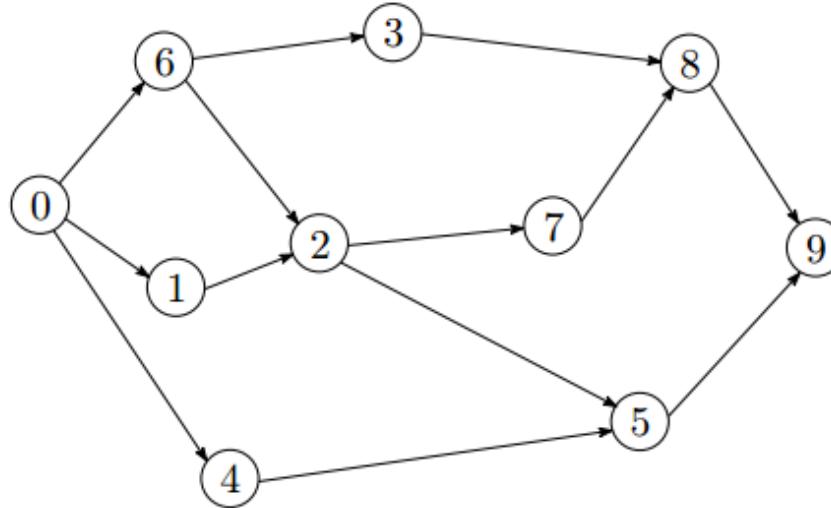
For **directed graph**, we distinguish between edge (u, v) and edge (v, u)



- **out-degree** of a vertex is the number of edges **leaving** it
- **in-degree** of a vertex is the number of edges **entering** it

Directed Graph

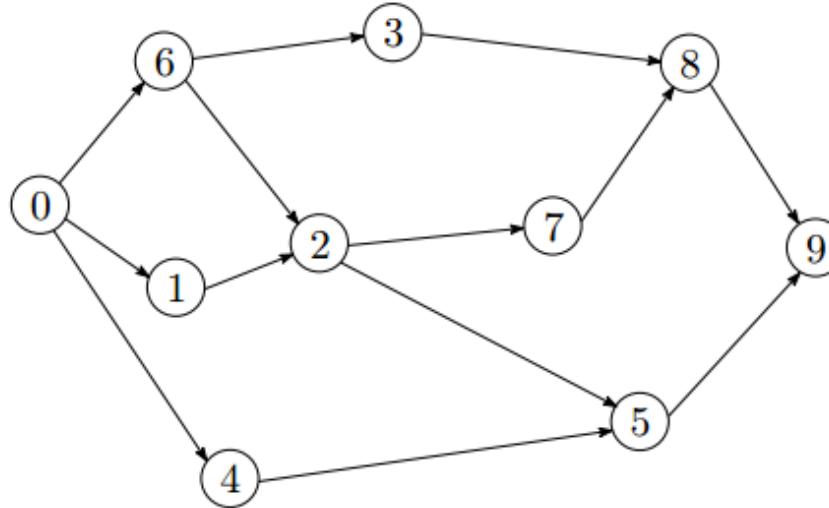
For **directed graph**, we distinguish between edge (u, v) and edge (v, u)



- **out-degree** of a vertex is the number of edges **leaving** it
- **in-degree** of a vertex is the number of edges **entering** it
- Each edge (u, v) contributes one to the out-degree of u and contributes one to the in-degree of v

Directed Graph

For **directed graph**, we distinguish between edge (u, v) and edge (v, u)



- **out-degree** of a vertex is the number of edges **leaving** it
- **in-degree** of a vertex is the number of edges **entering** it
- Each edge (u, v) contributes one to the out-degree of u and contributes one to the in-degree of v

$$\sum_{v \in V} \text{out-degree}(v) = \sum_{v \in V} \text{in-degree}(v) = |E|$$

Usage of Directed Graph

- Directed graphs are often used to represent **order-dependent** tasks

Usage of Directed Graph

- Directed graphs are often used to represent **order-dependent** tasks
 - That is, we cannot start a task before another task finishes

Usage of Directed Graph

- Directed graphs are often used to represent **order-dependent** tasks
 - That is, we cannot start a task before another task finishes
- An edge (u, v) means task v cannot start until task u is finished



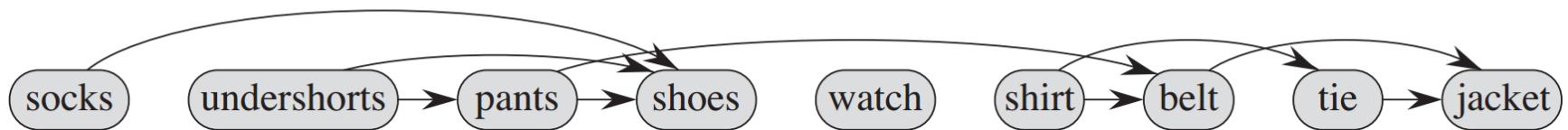
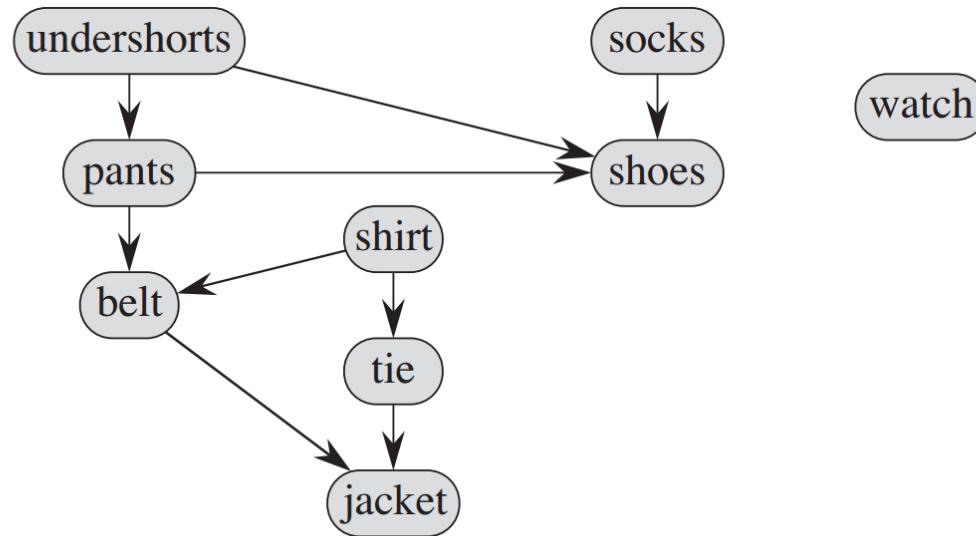
Usage of Directed Graph

- Directed graphs are often used to represent **order-dependent** tasks
 - That is, we cannot start a task before another task finishes
- An edge (u, v) means task v cannot start until task u is finished



- Clearly, for the system needs to hang, the graph must be **acyclic**
 - It must be a **directed acyclic graph (or DAG)**

An Example



Topological Sort

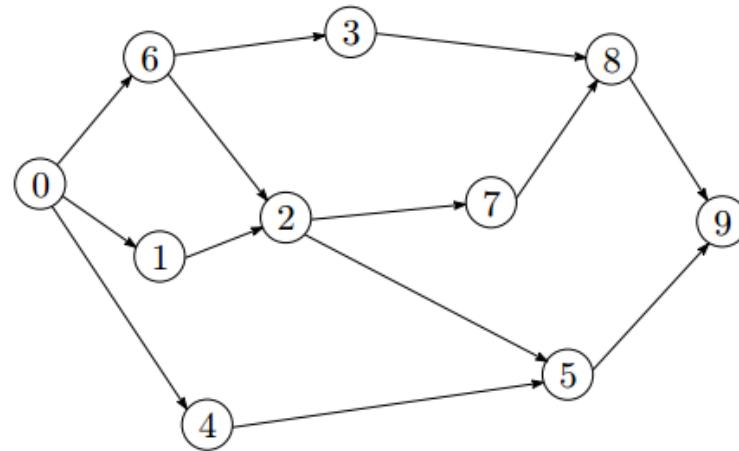
- **Topological sort** is an algorithm for a directed acyclic graph

Topological Sort

- Topological sort is an algorithm for a directed acyclic graph
- Linearly order the vertices so that for each edge (u, v) in the DAG, u appears before v in the linear ordering

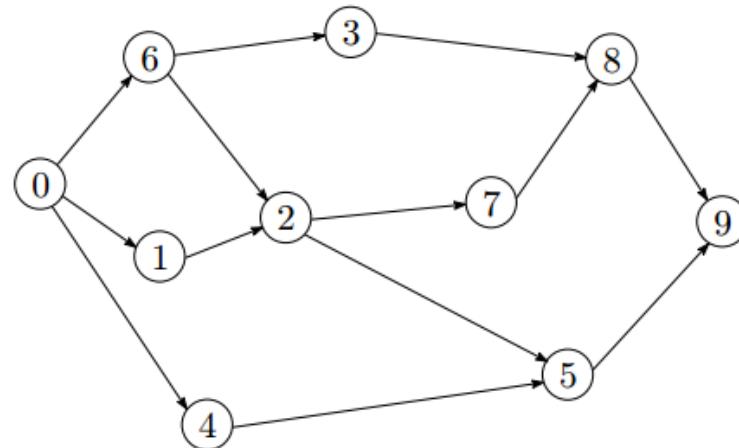
Topological Sort

- Topological sort is an algorithm for a directed acyclic graph
- Linearly order the vertices so that for each edge (u, v) in the DAG, u appears before v in the linear ordering



Topological Sort

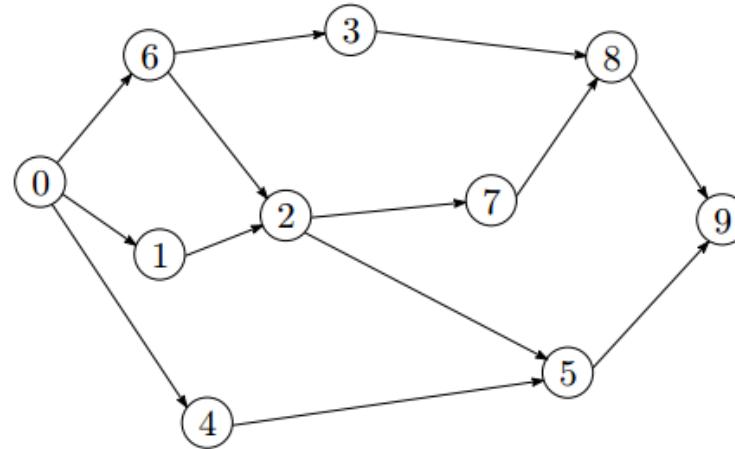
- Topological sort is an algorithm for a directed acyclic graph
- Linearly order the vertices so that for each edge (u, v) in the DAG, u appears before v in the linear ordering



- It may not be unique as there are many "equal" elements!

Topological Sort

- **Topological sort** is an algorithm for a directed acyclic graph
- **Linearly order the vertices** so that for each edge (u, v) in the DAG, u appears before v in the linear ordering



- It may not be unique as there are many "equal" elements!
- For the example, there are several topological orderings
 - 0, 6, 1, 4, 3, 2, 5, 7, 8, 9
 - 0, 4, 1, 6, 2, 5, 3, 7, 8, 9
 - ...

Topological Sort Algorithm

- Observations
 - Starting vertex must have zero in-degree

Topological Sort Algorithm

- Observations
 - Starting vertex must have zero in-degree
 - If such a vertex doesn't exist, the graph would not be acyclic

Topological Sort Algorithm

- Observations
 - Starting vertex must have zero in-degree
 - If such a vertex doesn't exist, the graph would not be acyclic
- Main idea of the algorithm
 - A vertex with zero in-degree is a task that can start right away. So we can output it first in the linear order

Topological Sort Algorithm

- Observations
 - Starting vertex must have zero in-degree
 - If such a vertex doesn't exist, the graph would not be acyclic
- Main idea of the algorithm
 - A vertex with zero in-degree is a task that can start right away. So we can output it first in the linear order
 - If a vertex u is output, then all the edges (u, v) are no longer useful, since tasks v do not need to wait for u anymore

Topological Sort Algorithm

- Observations
 - Starting vertex must have zero in-degree
 - If such a vertex doesn't exist, the graph would not be acyclic
- Main idea of the algorithm
 - A vertex with zero in-degree is a task that can start right away. So we can output it first in the linear order
 - If a vertex u is output, then all the edges (u, v) are no longer useful, since tasks v do not need to wait for u anymore
 - remove all the edges (u, v) (u 's outgoing edges)

Topological Sort Algorithm

- Observations
 - Starting vertex must have zero in-degree
 - If such a vertex doesn't exist, the graph would not be acyclic
- Main idea of the algorithm
 - A vertex with zero in-degree is a task that can start right away. So we can output it first in the linear order
 - If a vertex u is output, then all the edges (u, v) are no longer useful, since tasks v do not need to wait for u anymore
 - remove all the edges (u, v) (u 's outgoing edges)
 - With vertex u removed, the new graph is still a directed acyclic graph

Topological Sort Algorithm

- Observations
 - Starting vertex must have zero in-degree
 - If such a vertex doesn't exist, the graph would not be acyclic
- Main idea of the algorithm
 - A vertex with zero in-degree is a task that can start right away. So we can output it first in the linear order
 - If a vertex u is output, then all the edges (u, v) are no longer useful, since tasks v do not need to wait for u anymore
 - remove all the edges (u, v) (u 's outgoing edges)
 - With vertex u removed, the new graph is still a directed acyclic graph
 - repeat steps 1-2 until no vertex is left

Topological Sort Algorithm

Topological-Sort(G)

Input: A graph G

Output: None

Initialize Q to be an empty queue;

for $u \in V$ **do**

if $u.in_degree$ is equal to 0 **then**

 //Find all starting vertices

 Enqueue(Q, u);

end

end

while Q is not empty **do**

$u \leftarrow$ Dequeue(Q);

 Output u ;

for $v \in Adj(u)$ **do**

 //remove u 's outgoing edges

$v.in_degree \leftarrow v.in_degree - 1$;

if $v.in_degree$ is equal to 0 **then**

 | Enqueue(Q, v);

end

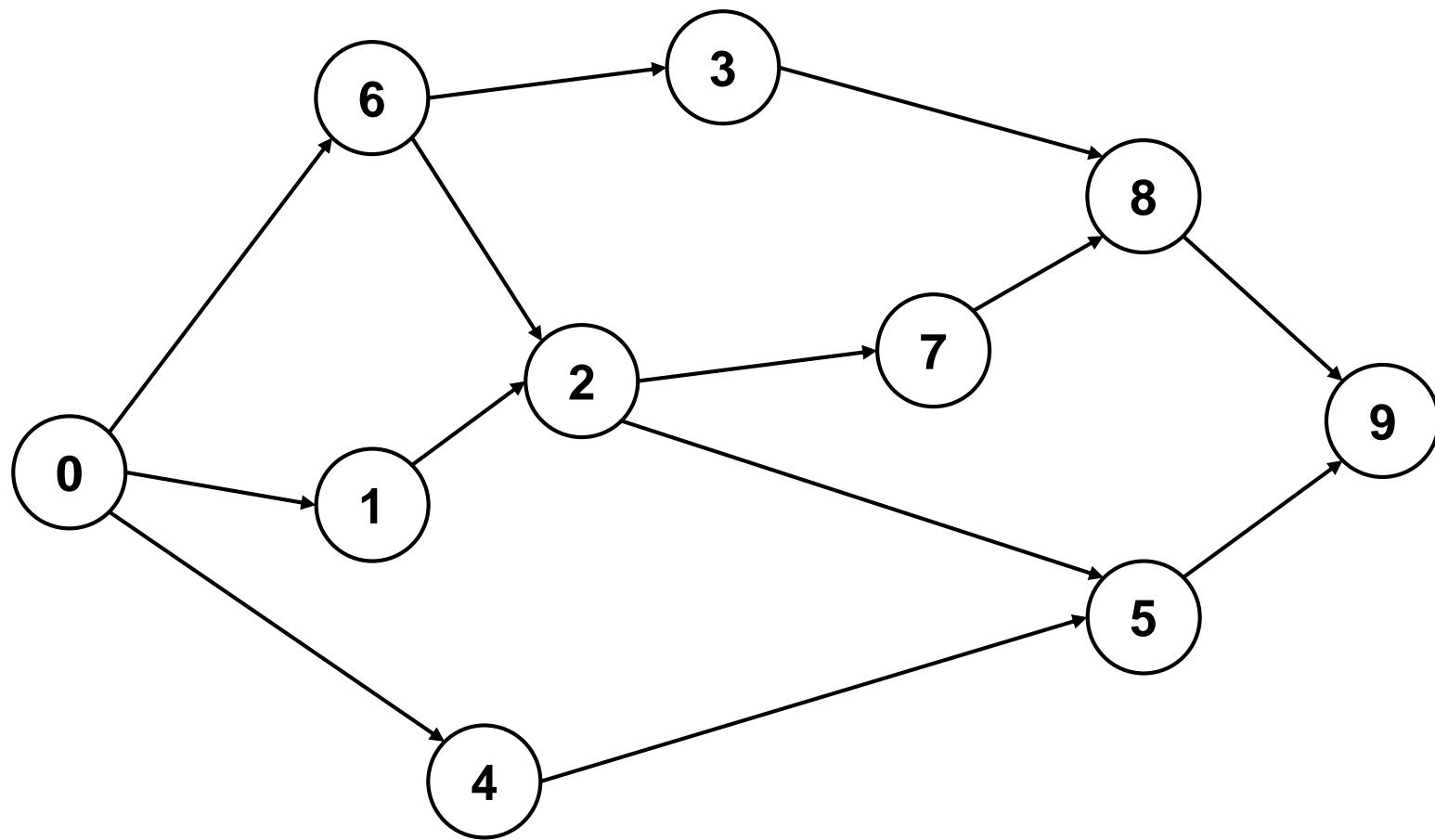
end

end

Topological Sort Example

Q

Output

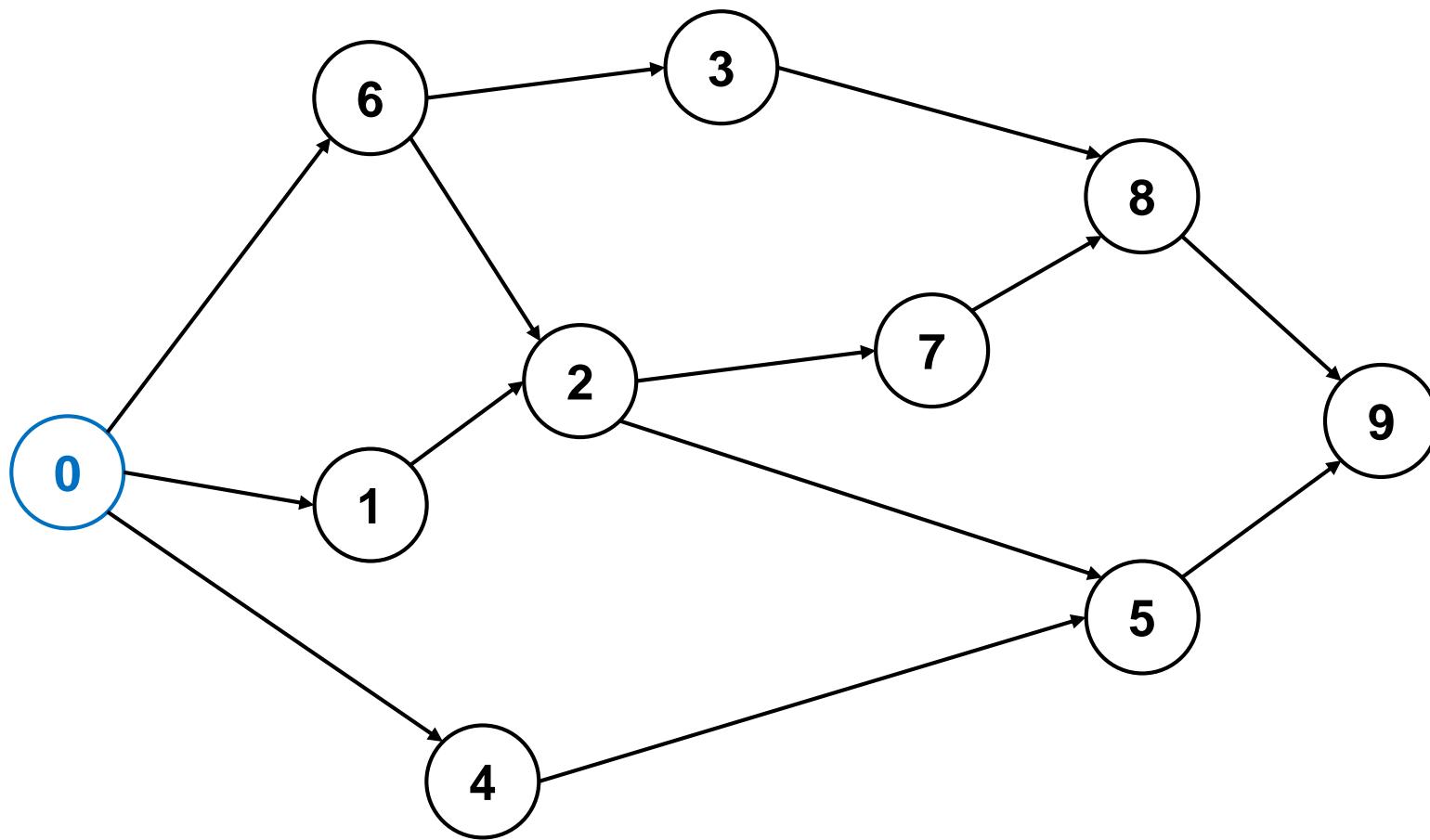


Topological Sort Example

Q

0

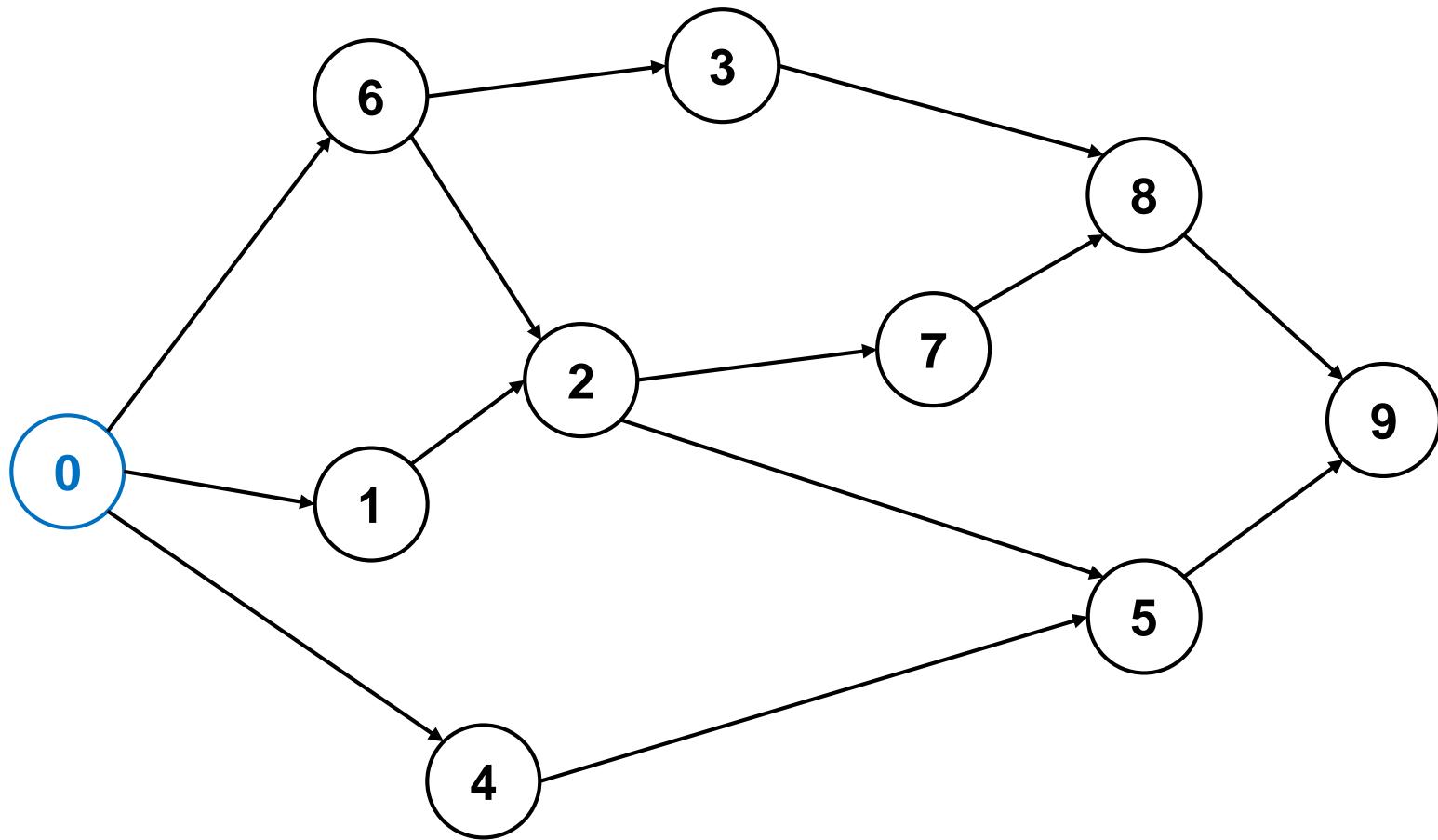
Output



Topological Sort Example

Q
0

Output



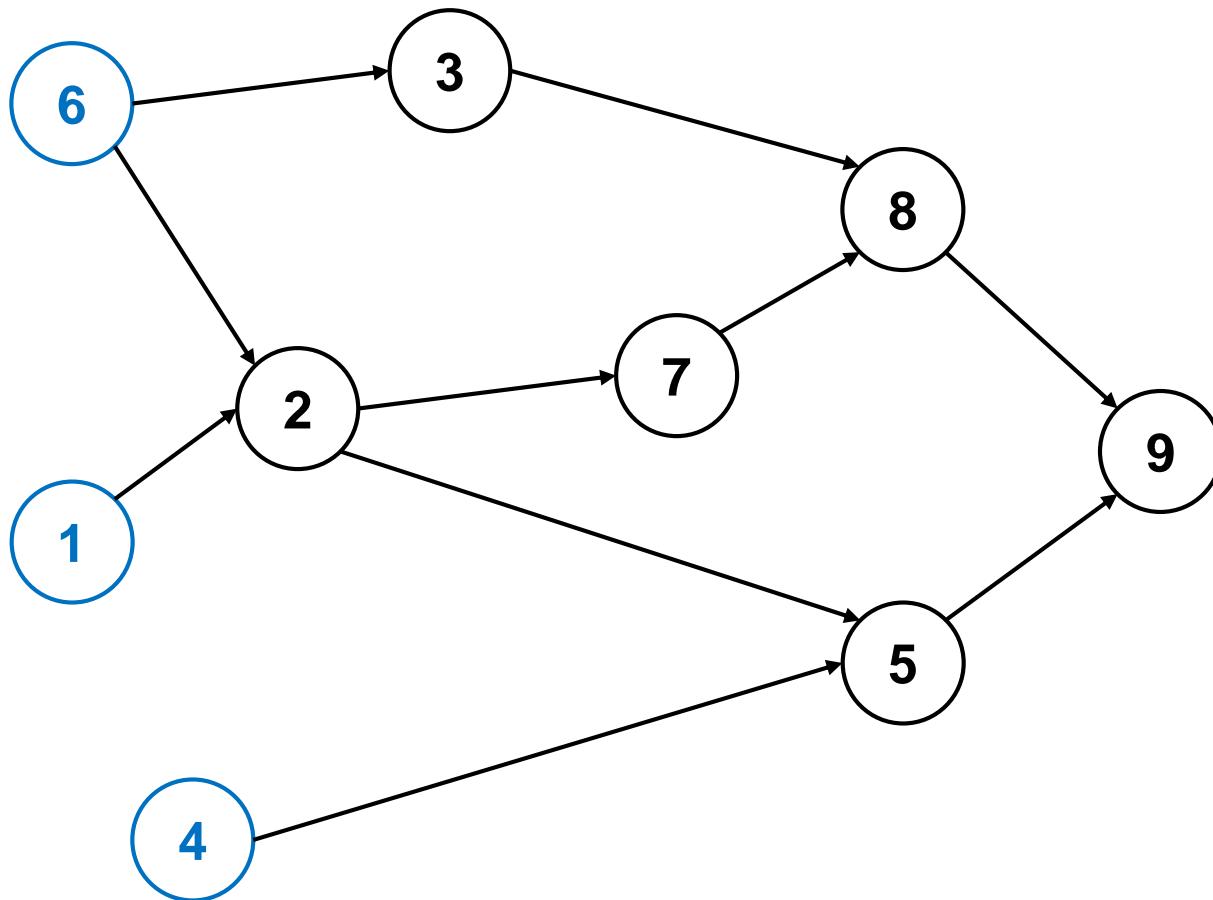
Topological Sort Example

Q

0	6	1	4
---	---	---	---

Output

0



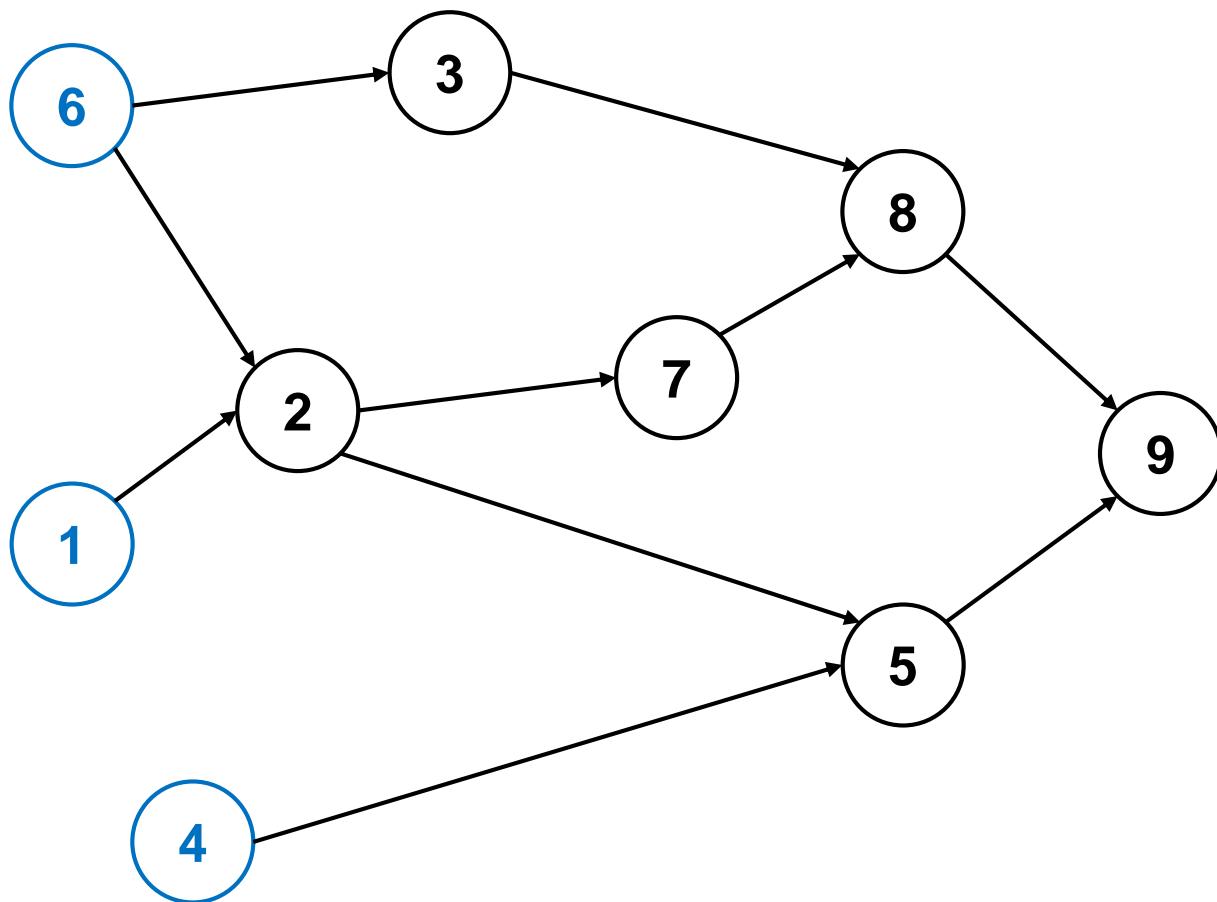
Topological Sort Example

Q

0	6	1	4
	6	1	4

Output

0



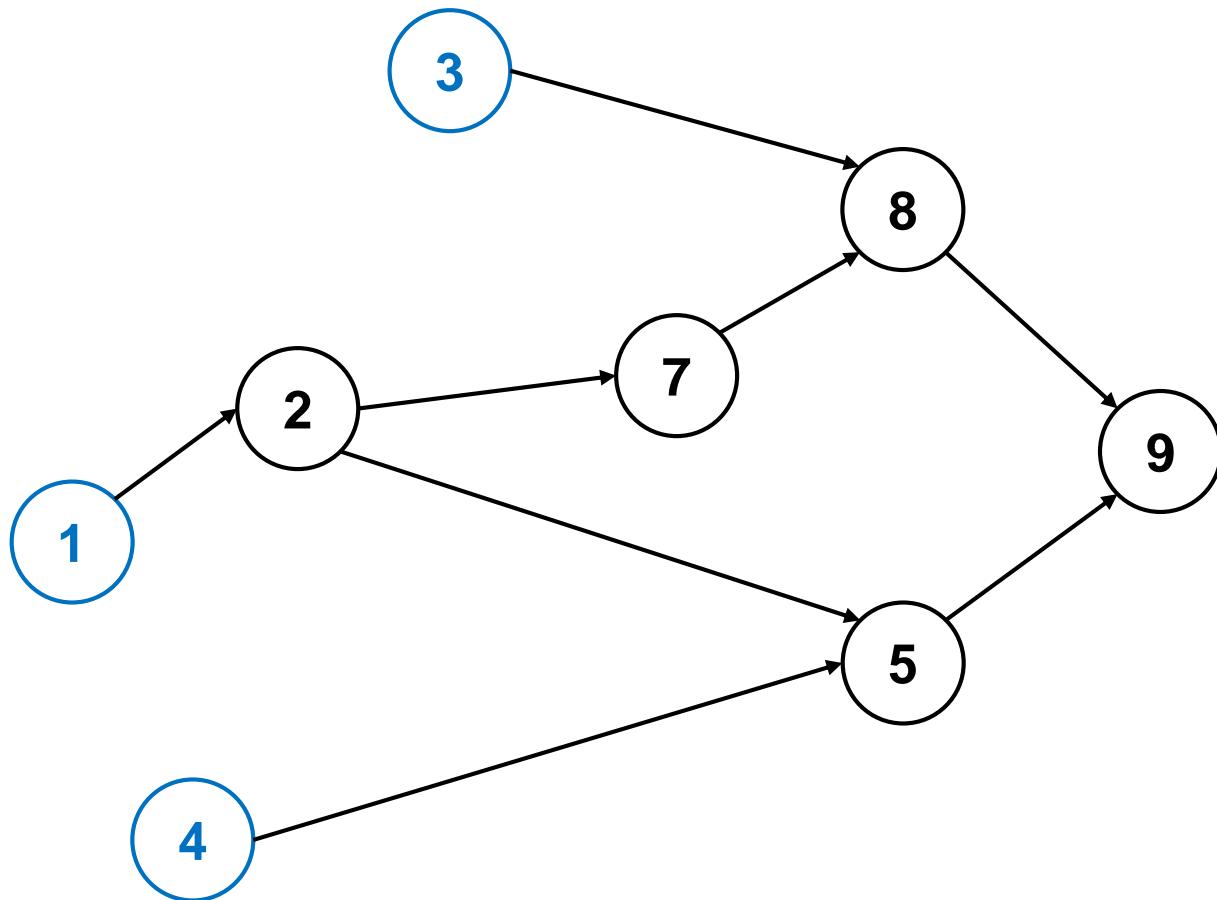
Topological Sort Example

Q

0	6	1	4	3
---	---	---	---	---

Output

0	6
---	---



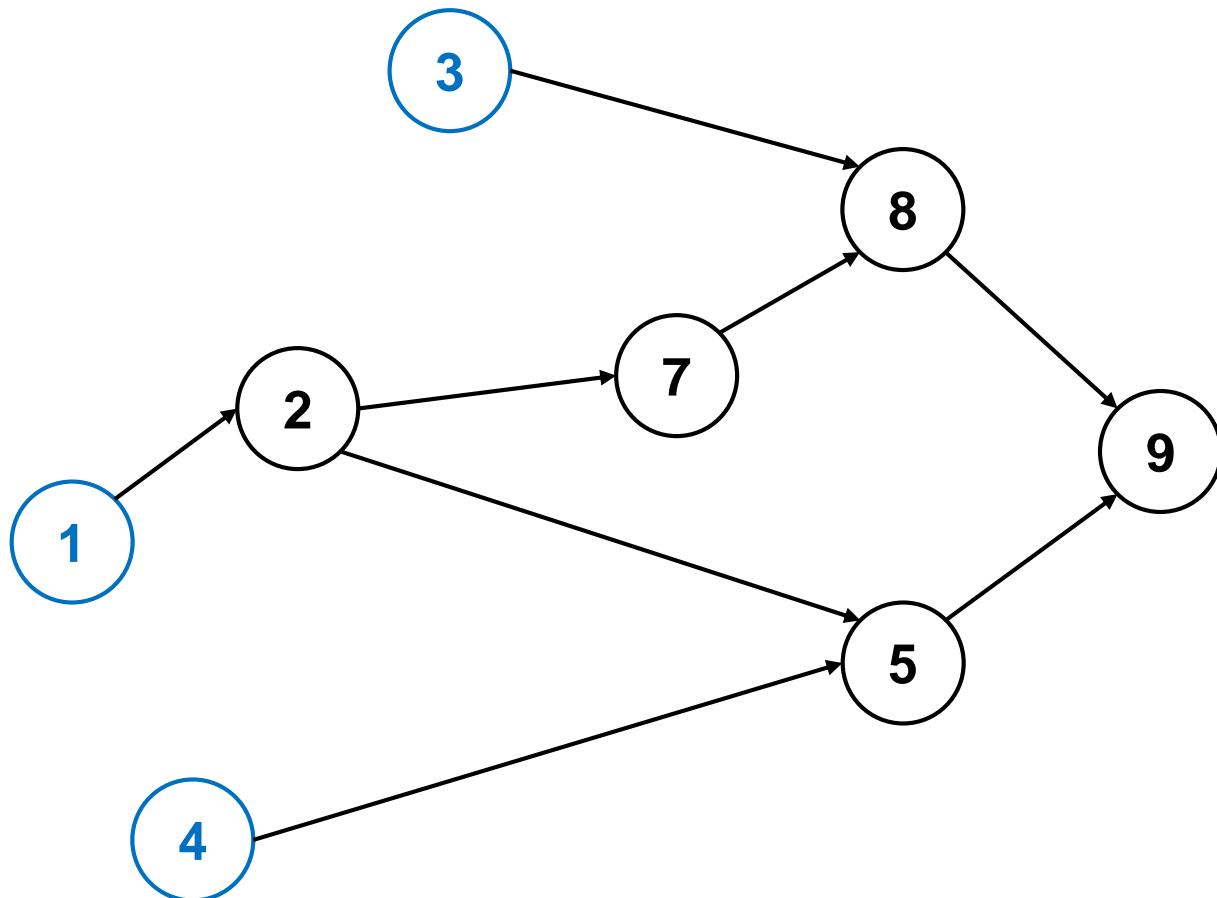
Topological Sort Example

Q

0	6	1	4	3
0	6	1	4	3

Output

0	6
---	---



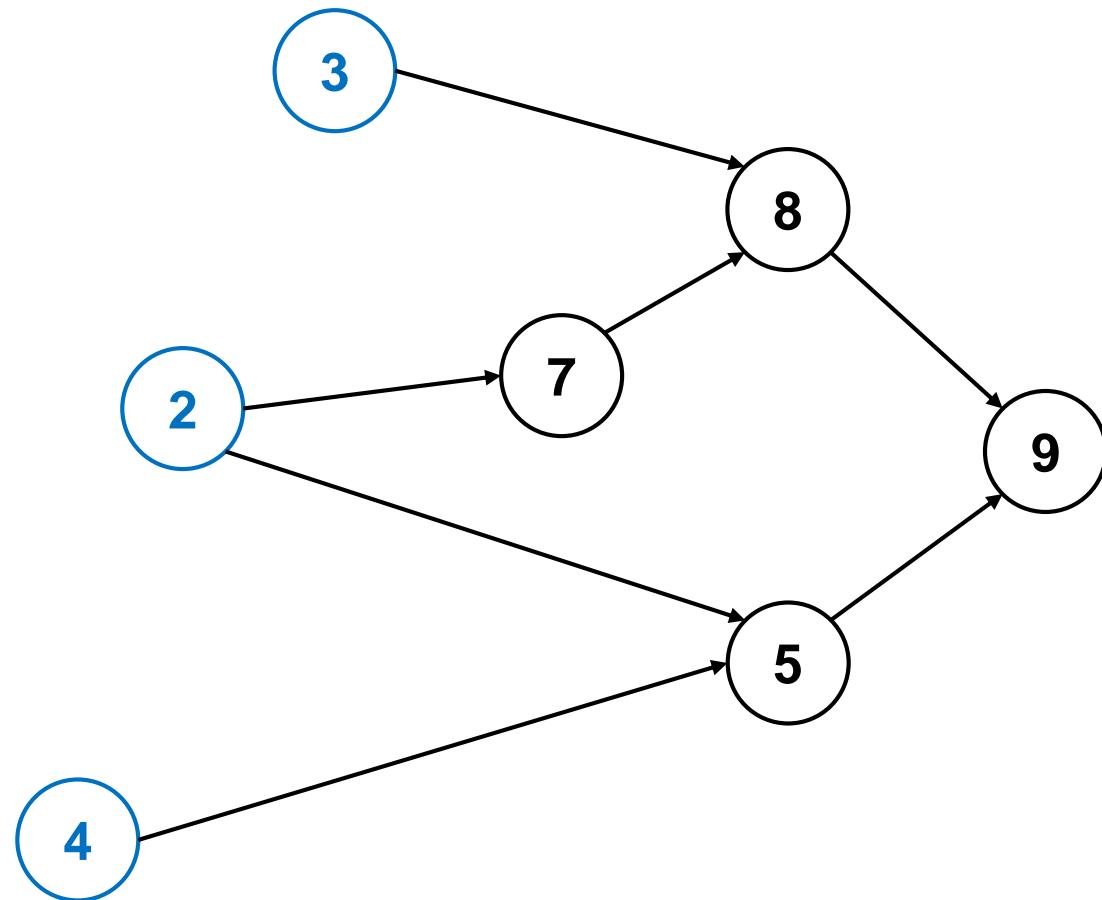
Topological Sort Example

Q

0	6	1	4	3	2
---	---	---	---	---	---

Output

0	6	1
---	---	---



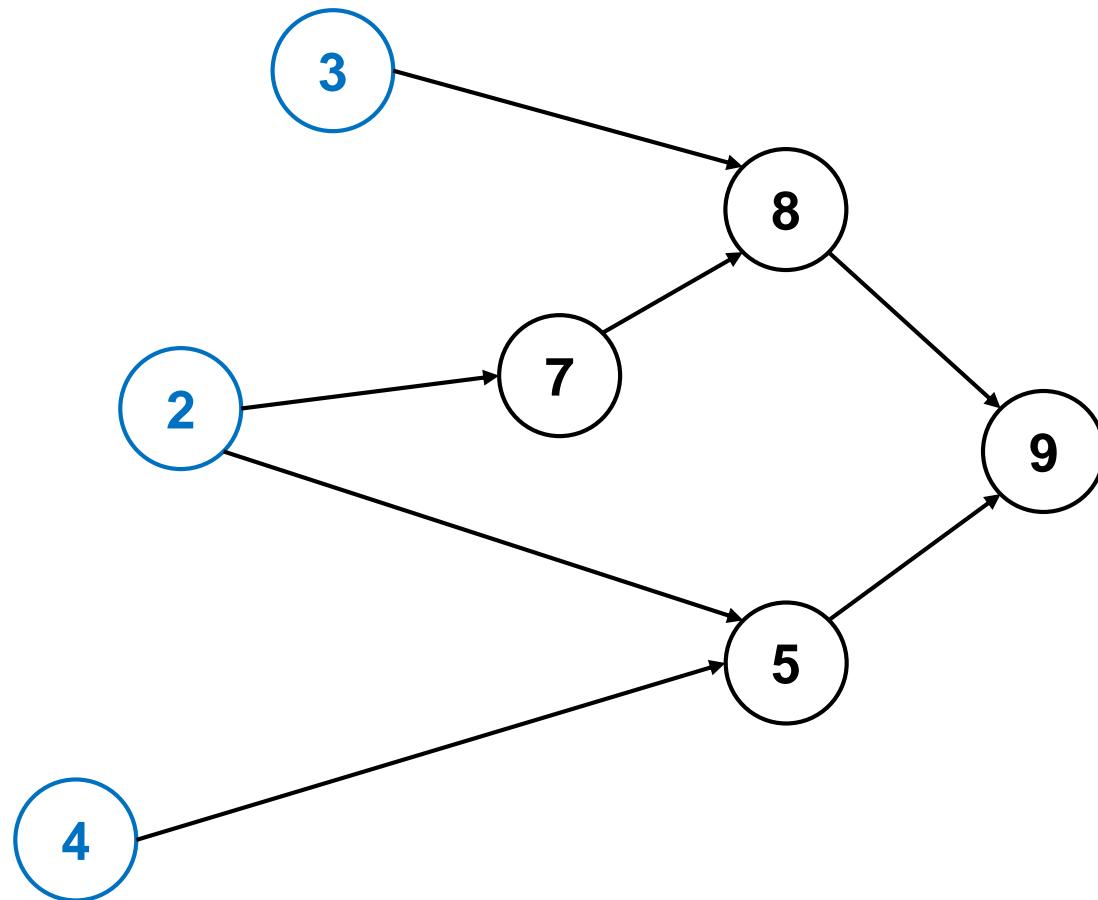
Topological Sort Example

Q

0	6	1	4	3	2
---	---	---	---	---	---

Output

0	6	1
---	---	---



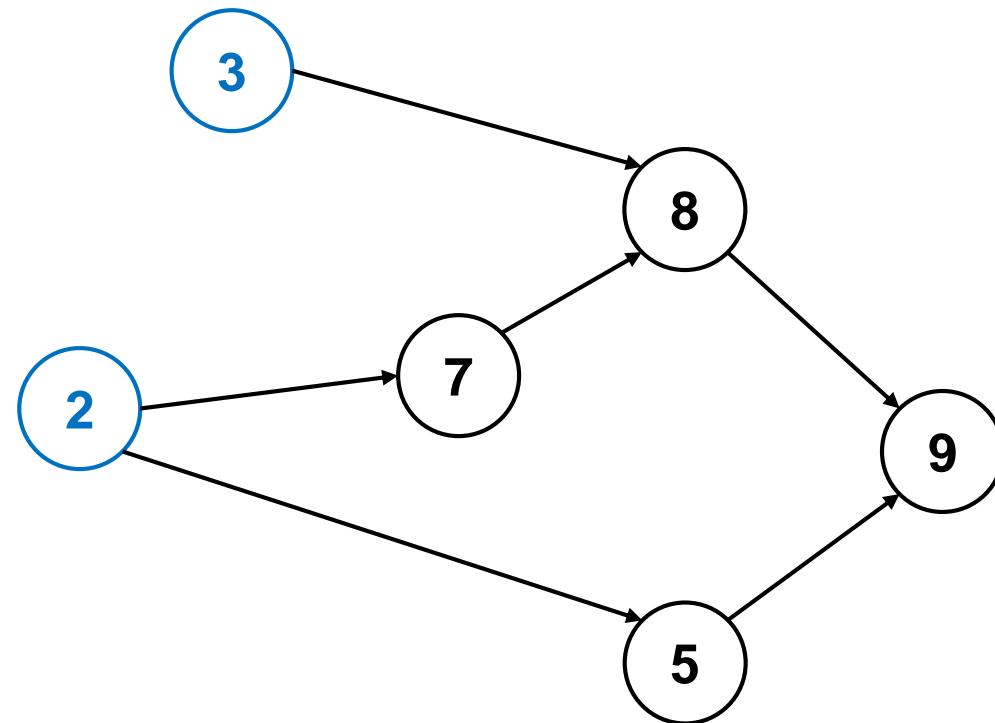
Topological Sort Example

Q

0	6	1	4	3	2
---	---	---	---	---	---

Output

0	6	1	4
---	---	---	---



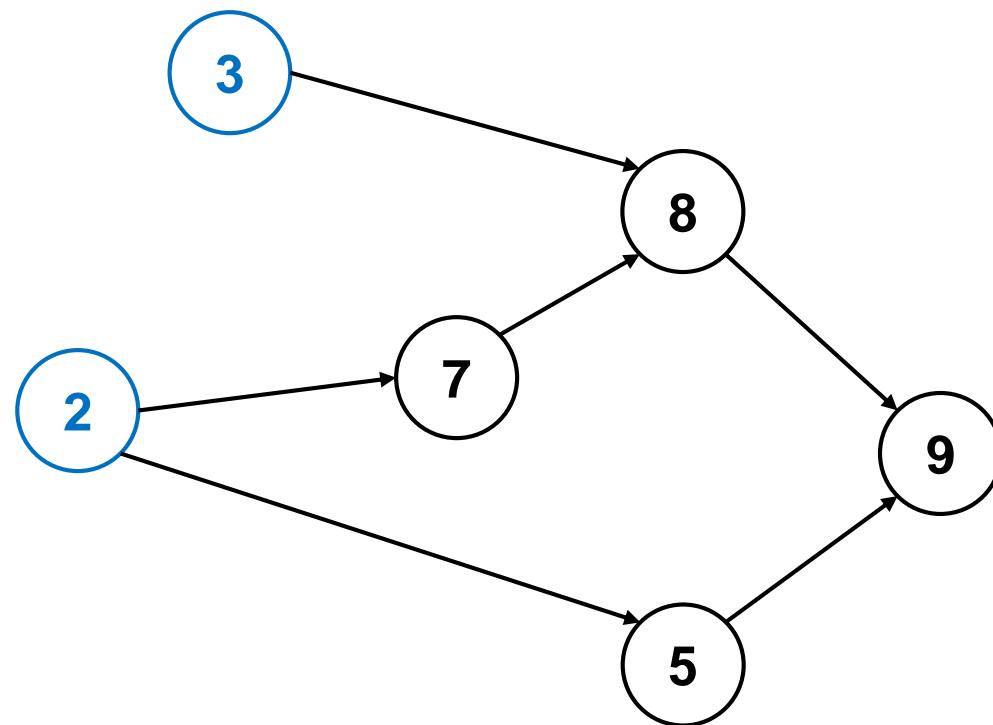
Topological Sort Example

Q

0	6	1	4	3	2
0	6	1	4	3	2

Output

0	6	1	4
---	---	---	---



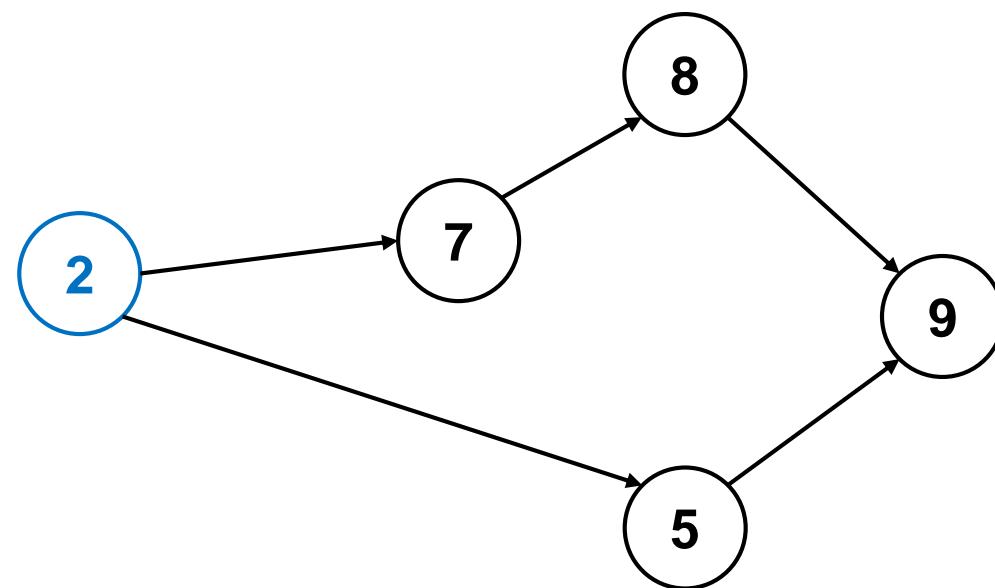
Topological Sort Example

Q

0	6	1	4	3	2
---	---	---	---	---	---

Output

0	6	1	4	3
---	---	---	---	---



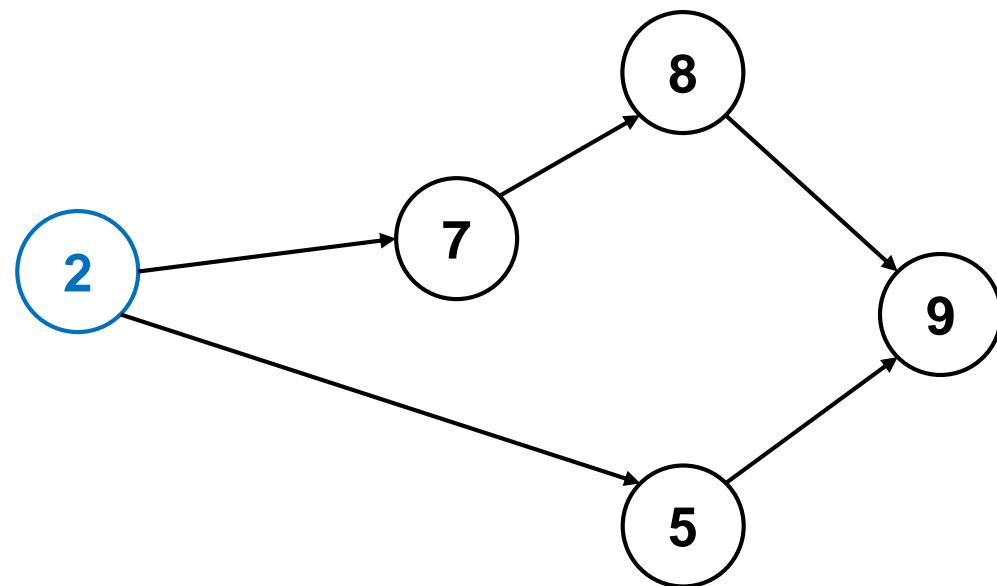
Topological Sort Example

Q

0	6	1	4	3	2
---	---	---	---	---	---

Output

0	6	1	4	3
---	---	---	---	---



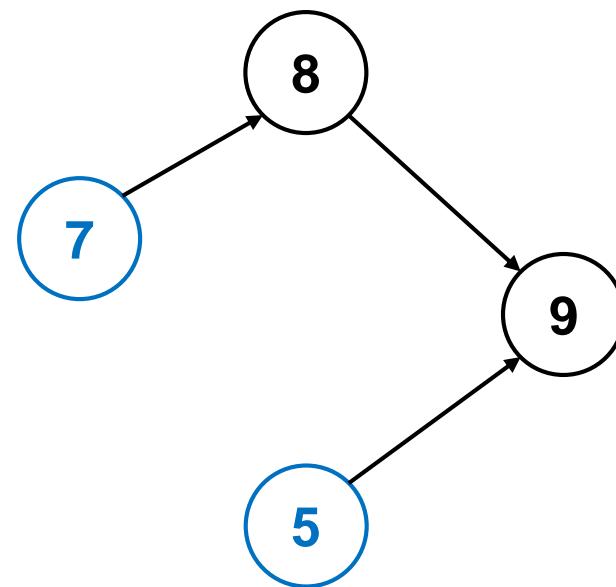
Topological Sort Example

Q

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

Output

0	6	1	4	3	2
---	---	---	---	---	---



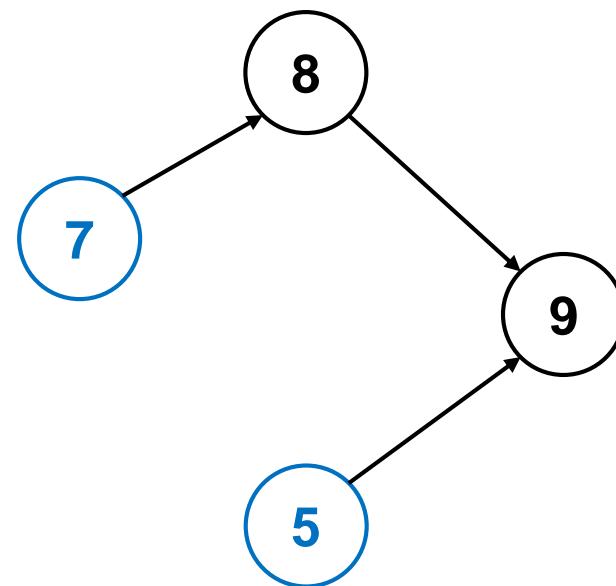
Topological Sort Example

Q

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

Output

0	6	1	4	3	2
---	---	---	---	---	---



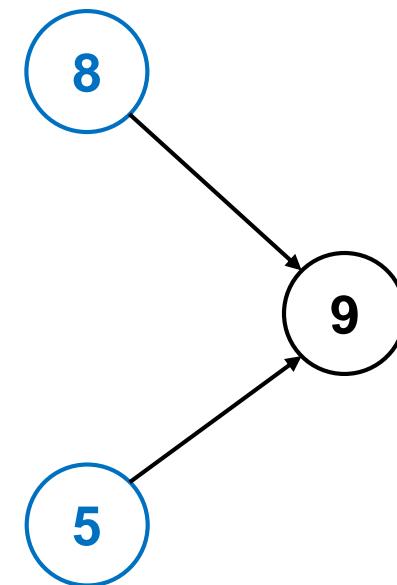
Topological Sort Example

Q

0	6	1	4	3	2	7	5	8
---	---	---	---	---	---	---	---	---

Output

0	6	1	4	3	2	7
---	---	---	---	---	---	---



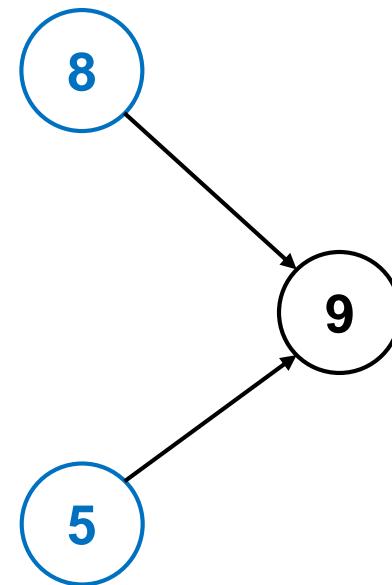
Topological Sort Example

Q

0	6	1	4	3	2	7	5	8
0	6	1	4	3	2	7	5	8

Output

0	6	1	4	3	2	7
---	---	---	---	---	---	---



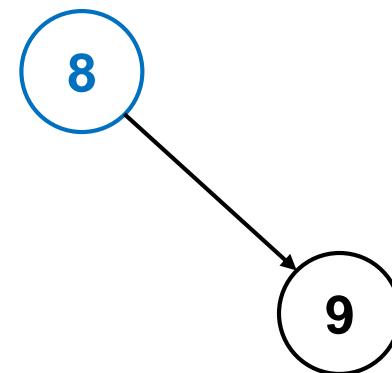
Topological Sort Example

Q

0	6	1	4	3	2	7	5	8
---	---	---	---	---	---	---	---	---

Output

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



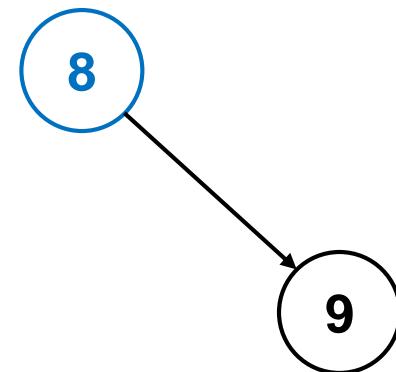
Topological Sort Example

Q

0	6	1	4	3	2	7	5	8
0	6	1	4	3	2	7	5	8

Output

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



Topological Sort Example

Q

0	6	1	4	3	2	7	5	8	9
---	---	---	---	---	---	---	---	---	---

Output

0	6	1	4	3	2	7	5	8
---	---	---	---	---	---	---	---	---

9

Topological Sort Example

Q

0	6	1	4	3	2	7	5	8	9
---	---	---	---	---	---	---	---	---	---

Output

0	6	1	4	3	2	7	5	8
---	---	---	---	---	---	---	---	---

9

Topological Sort Example

Q

0	6	1	4	3	2	7	5	8	9
---	---	---	---	---	---	---	---	---	---

Output

0	6	1	4	3	2	7	5	8	9
---	---	---	---	---	---	---	---	---	---

Outline

- Introduction to Part V
- Review of Basic Graph Search Algorithms
 - Basic Concepts
 - The Breadth-First Search (BFS) Algorithm
 - The Depth-First Search (DFS) Algorithm
- Topological Sort
 - The Topological Sort Algorithm
 - Analysis of the Topological Sort Algorithm
- Strongly Connected Components
 - The Algorithm of Finding SCCs
 - Analysis of the Algorithm

Correctness of Topological Sort Algorithm

- For any vertex v , if it depends on a vertex u (i.e., edge (u, v) exists), then for v to appear in the linear ordering, all of its incoming edges must be removed so that it has zero indegree.

Correctness of Topological Sort Algorithm

- For any vertex v , if it depends on a vertex u (i.e., edge (u, v) exists), then for v to appear in the linear ordering, all of its incoming edges must be removed so that it has zero in-degree.
- Therefore, v must appear after u in the linear ordering.

Topological Sort: Complexity

- We never visit a vertex more than once

Topological Sort: Complexity

- We never visit a vertex more than once
- For each vertex, we examine all outgoing edges
 - $\sum_{v \in V} \text{out-degree}(v) = E$

Topological Sort: Complexity

- We never visit a vertex more than once
- For each vertex, we examine all outgoing edges
 - $\sum_{v \in V} \text{out-degree}(v) = E$
- Therefore, the running time is $O(V + E)$

Topological Sort: Complexity

- We never visit a vertex more than once
- For each vertex, we examine all outgoing edges
 - $\sum_{v \in V} \text{out-degree}(v) = E$
- Therefore, the running time is $O(V + E)$

Question

Can we use DFS to implement topological sort?

Outline

- Introduction to Part IV
- Review of Basic Graph Search Algorithms
 - Basic Concepts
 - The Breadth-First Search (BFS) Algorithm
 - The Depth-First Search (DFS) Algorithm
- Topological Sort
 - The Topological Sort Algorithm
 - Analysis of the Topological Sort Algorithm
- **Strongly Connected Components**
 - The Algorithm of Finding SCCs
 - Analysis of the Algorithm

Strongly Connected Components

Let $G = (V, E)$ be a directed graph.

Strongly Connected Components

Let $G = (V, E)$ be a directed graph.

A **strongly connected component** (SCC) of G is a subset S of V such that

- For any two vertices $u, v \in S$, it must hold that:

Strongly Connected Components

Let $G = (V, E)$ be a directed graph.

A **strongly connected component** (SCC) of G is a subset S of V such that

- For any two vertices $u, v \in S$, it must hold that:
 - There is a path from u to v .
 - There is a path from v to u .

Strongly Connected Components

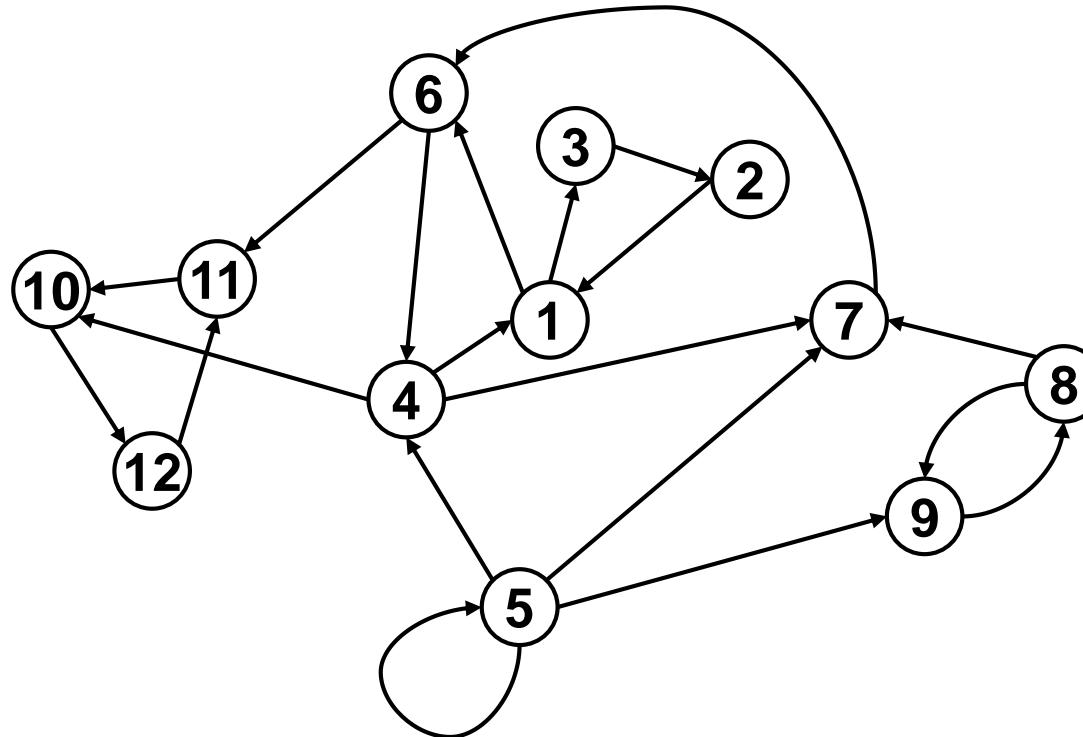
Let $G = (V, E)$ be a directed graph.

A **strongly connected component** (SCC) of G is a subset S of V such that

- For any two vertices $u, v \in S$, it must hold that:
 - There is a path from u to v .
 - There is a path from v to u .
- S is **maximal** in the sense that we cannot put any more vertex into S without violating the above property.

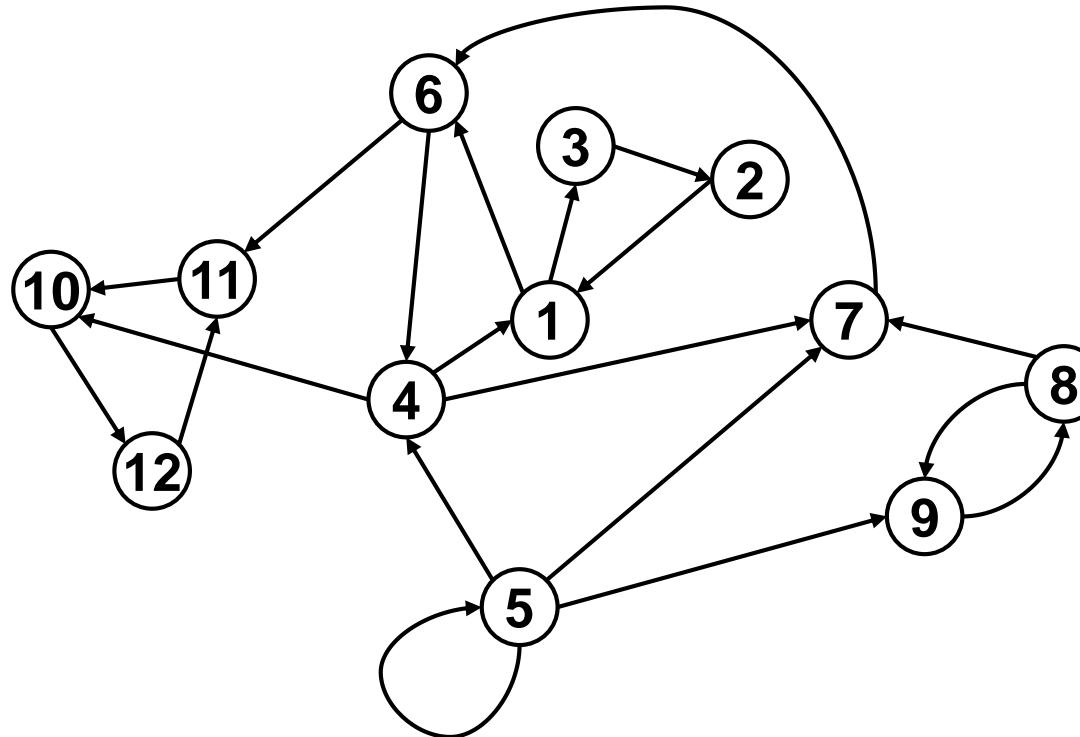
Example

Consider the following graph:



Example

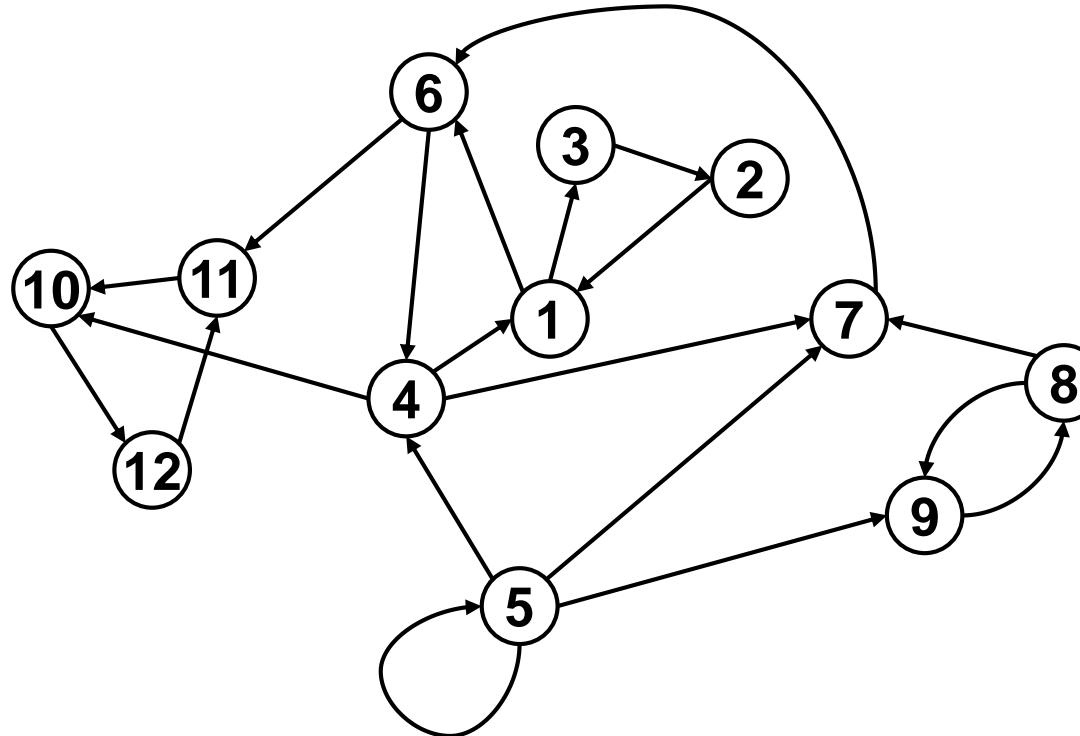
Consider the following graph:



- $\{10,11,12\}$ is an SCC.
- $\{6,10,11,12\}$ is not an SCC.

Example

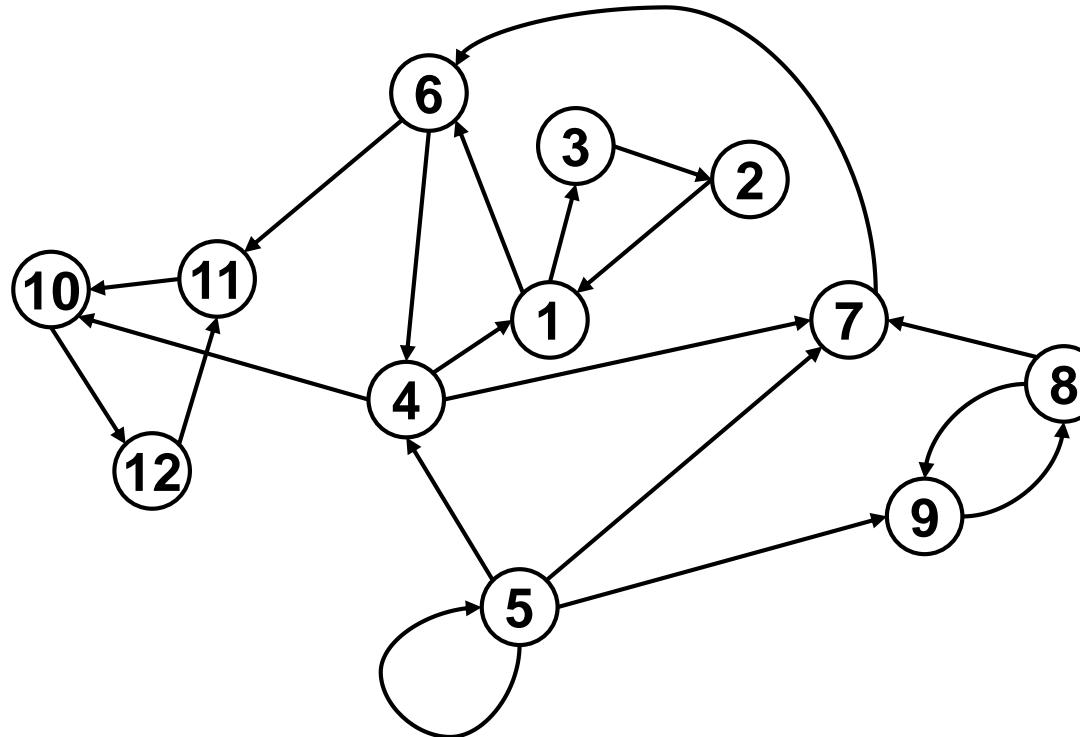
Consider the following graph:



- $\{10,11,12\}$ is an SCC.
- $\{6,10,11,12\}$ is not an SCC.
- $\{1,2,3,4,6\}$ is not an SCC (because we can still add vertex 7).

Example

Consider the following graph:



- $\{10,11,12\}$ is an SCC.
- $\{6,10,11,12\}$ is not an SCC.
- $\{1,2,3,4,6\}$ is not an SCC (because we can still add vertex 7).
- $\{1,2,3,4,6,7\}$ is an SCC.

SCCs are Disjoint

Theorem: Suppose that S_1 and S_2 are both SCCs of G.
Then, $S_1 \cap S_2 = \emptyset$.

SCCs are Disjoint

Theorem: Suppose that S_1 and S_2 are both SCCs of G.
Then, $S_1 \cap S_2 = \emptyset$.

Proof: Assume that there is a vertex v in both S_1 and S_2 .

SCCs are Disjoint

Theorem: Suppose that S_1 and S_2 are both SCCs of G .
Then, $S_1 \cap S_2 = \emptyset$.

Proof: Assume that there is a vertex v in both S_1 and S_2 .
Then, for any vertex $u_1 \in S_1$ and any vertex $u_2 \in S_2$:

SCCs are Disjoint

Theorem: Suppose that S_1 and S_2 are both SCCs of G .
Then, $S_1 \cap S_2 = \emptyset$.

Proof: Assume that there is a vertex v in both S_1 and S_2 .
Then, for any vertex $u_1 \in S_1$ and any vertex $u_2 \in S_2$:

- There is a path from u_1 to u_2 : we can first go from u_1 to v within S_1 , and then from v to u_2 within S_2 .

SCCs are Disjoint

Theorem: Suppose that S_1 and S_2 are both SCCs of G.
Then, $S_1 \cap S_2 = \emptyset$.

Proof: Assume that there is a vertex v in both S_1 and S_2 .
Then, for any vertex $u_1 \in S_1$ and any vertex $u_2 \in S_2$:

- There is a path from u_1 to u_2 : we can first go from u_1 to v within S_1 , and then from v to u_2 within S_2 .
- Likewise, there is also a path from u_2 to u_1 .

SCCs are Disjoint

Theorem: Suppose that S_1 and S_2 are both SCCs of G .
Then, $S_1 \cap S_2 = \emptyset$.

Proof: Assume that there is a vertex v in both S_1 and S_2 .

Then, for any vertex $u_1 \in S_1$ and any vertex $u_2 \in S_2$:

- There is a path from u_1 to u_2 : we can first go from u_1 to v within S_1 , and then from v to u_2 within S_2 .
- Likewise, there is also a path from u_2 to u_1 .

Hence, neither S_1 nor S_2 is maximal, contradicting the fact that they are SCCs.

The Problem of Finding SCCs

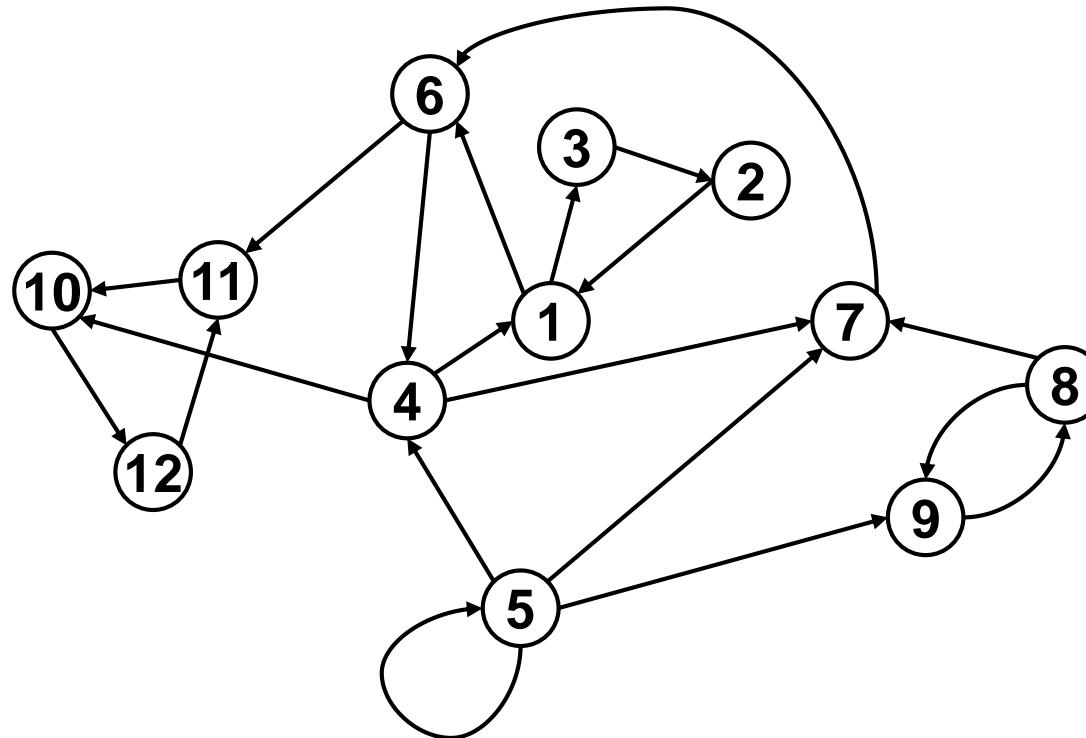
Given a directed graph $G = (V, E)$, the goal of the **finding strongly connected components problem** is to

The Problem of Finding SCCs

Given a directed graph $G = (V, E)$, the goal of the **finding strongly connected components problem** is to divide V into disjoint subsets, each of which is an SCC.

The Problem of Finding SCCs

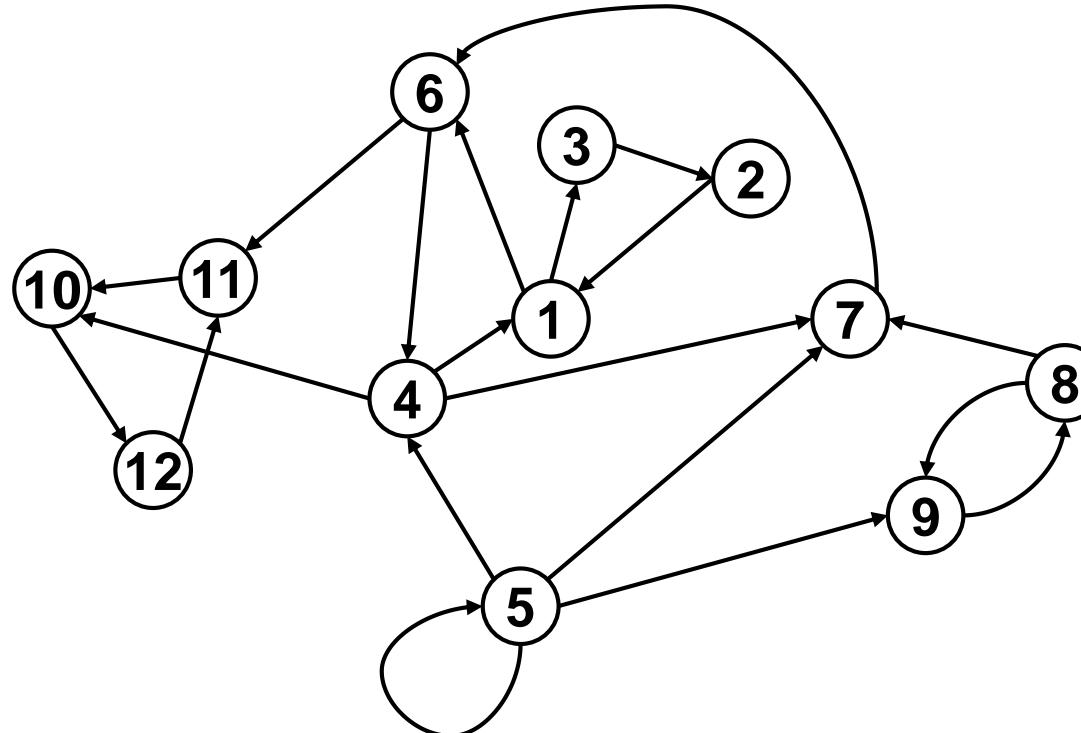
Given a directed graph $G = (V, E)$, the goal of the **finding strongly connected components problem** is to divide V into disjoint subsets, each of which is an SCC. Example:



The goal is to output the following 4 SCCs:

The Problem of Finding SCCs

Given a directed graph $G = (V, E)$, the goal of the **finding strongly connected components problem** is to divide V into disjoint subsets, each of which is an SCC. Example:



The goal is to output the following 4 SCCs:
 $\{10, 11, 12\}$, $\{1, 2, 3, 4, 6, 7\}$, $\{8, 9\}$, and $\{5\}$.

The Algorithm of Finding SCCs

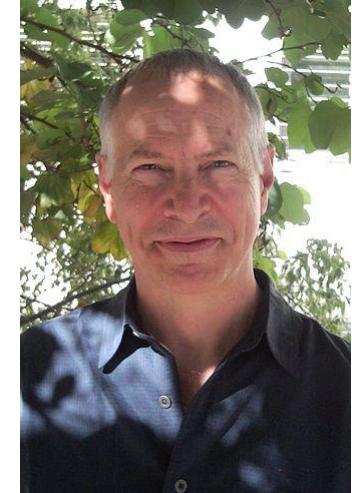
SIAM J. COMPUT.
Vol. 1, No. 2, June 1972

DEPTH-FIRST SEARCH AND LINEAR GRAPH ALGORITHMS*

ROBERT TARJAN†

Abstract. The value of depth-first search or “backtracking” as a technique for solving problems is illustrated by two examples. An improved version of an algorithm for finding the strongly connected components of a directed graph and an algorithm for finding the biconnected components of an undirected graph are presented. The space and time requirements of both algorithms are bounded by $k_1V + k_2E + k_3$ for some constants k_1, k_2 , and k_3 , where V is the number of vertices and E is the number of edges of the graph being examined.

Key words. Algorithm, backtracking, biconnectivity, connectivity, depth-first, graph, search, spanning tree, strong-connectivity.



Robert
Tarjan

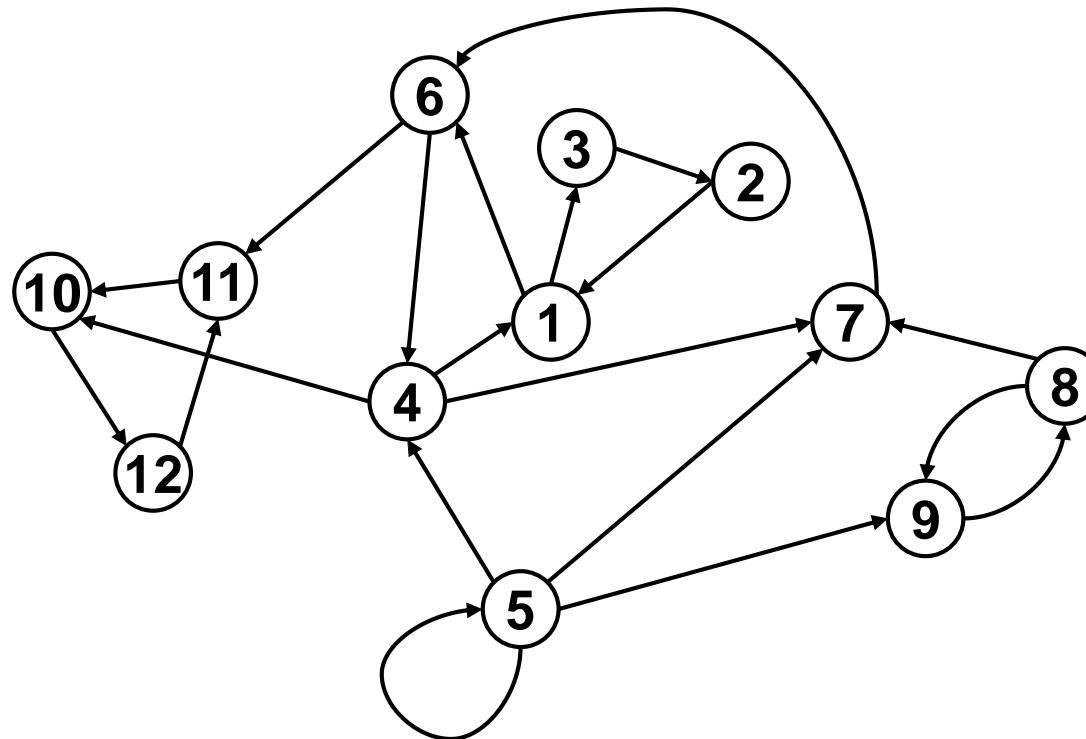
Robert Tarjan. Depth-First Search and Linear Graph Algorithms. SIAM Journal on Computing 1(2), 146-160, 1972. (CCF A类期刊)

Algorithm

Step 1: Obtain the **reverse graph G^R** by reversing the directions of all the edges in G .

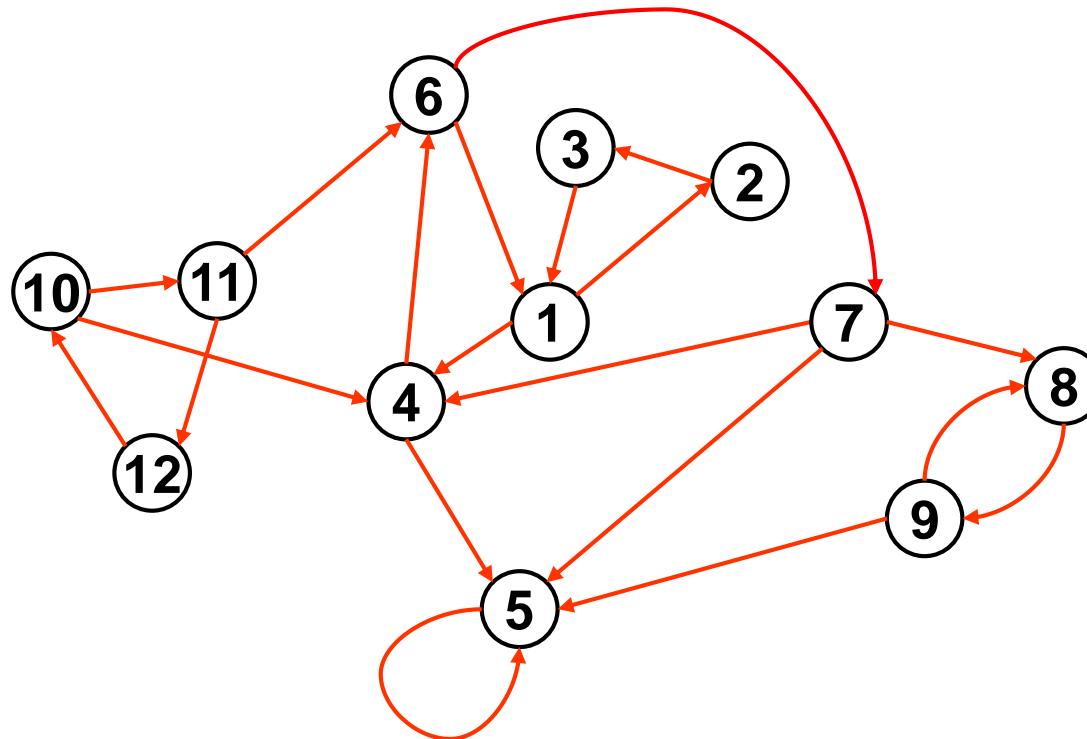
SCC Example

The original graph G :



SCC Example

Obtain the reverse graph G^R :



Algorithm

Step 1: Obtain the **reverse graph G^R** by reversing the directions of all the edges in G .

Step 2: Perform DFS on G^R ,

Algorithm

Step 1: Obtain the **reverse graph G^R** by reversing the directions of all the edges in G .

Step 2: Perform DFS on G^R , and obtain the sequence L^R that the vertices in G^R turn black (i.e., whenever a vertex is popped out of the stack, append it to L^R).

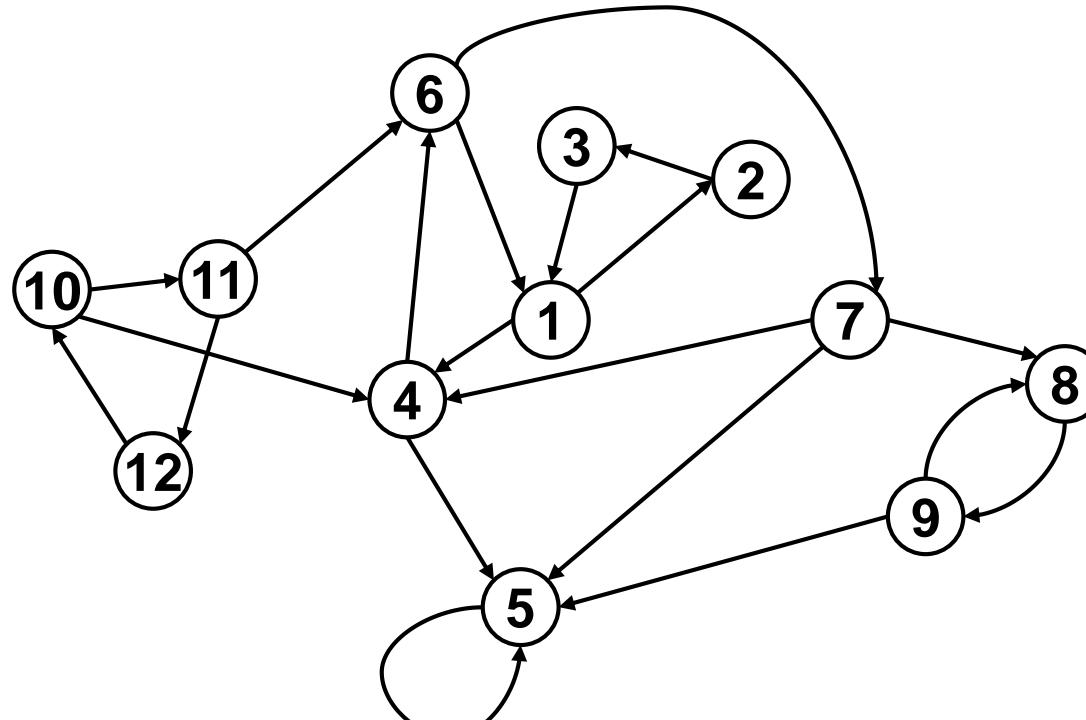
Algorithm

Step 1: Obtain the **reverse graph G^R** by reversing the directions of all the edges in G .

Step 2: Perform DFS on G^R , and obtain the sequence L^R that the vertices in G^R turn black (i.e., whenever a vertex is popped out of the stack, append it to L^R). Obtain L as the reverse order of L^R .

SCC Example

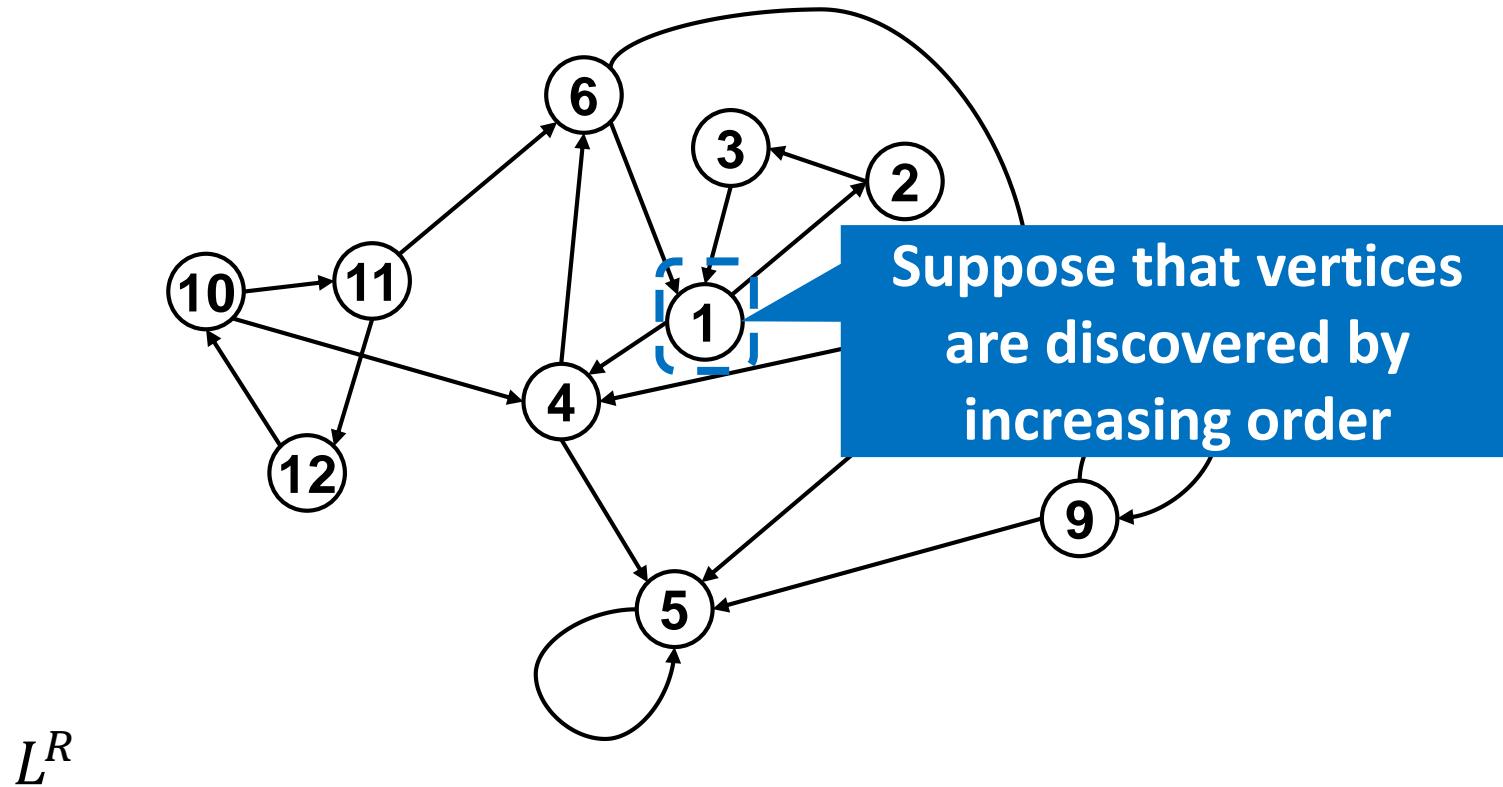
Reverse graph G^R :



L^R

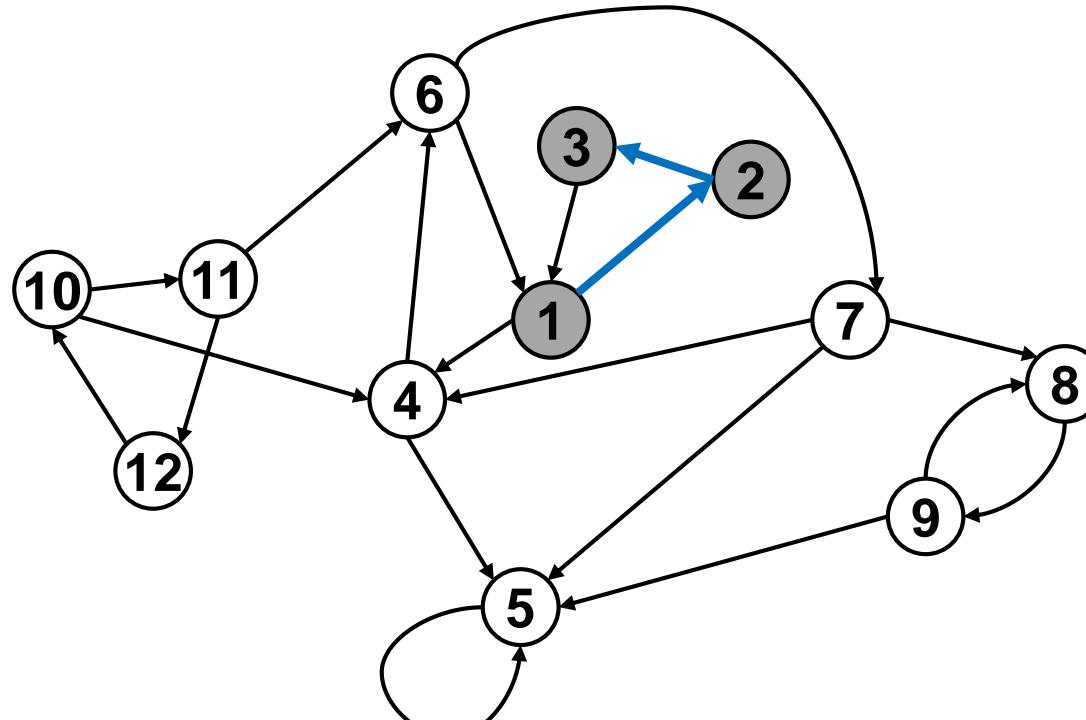
SCC Example

Reverse graph G^R :



SCC Example

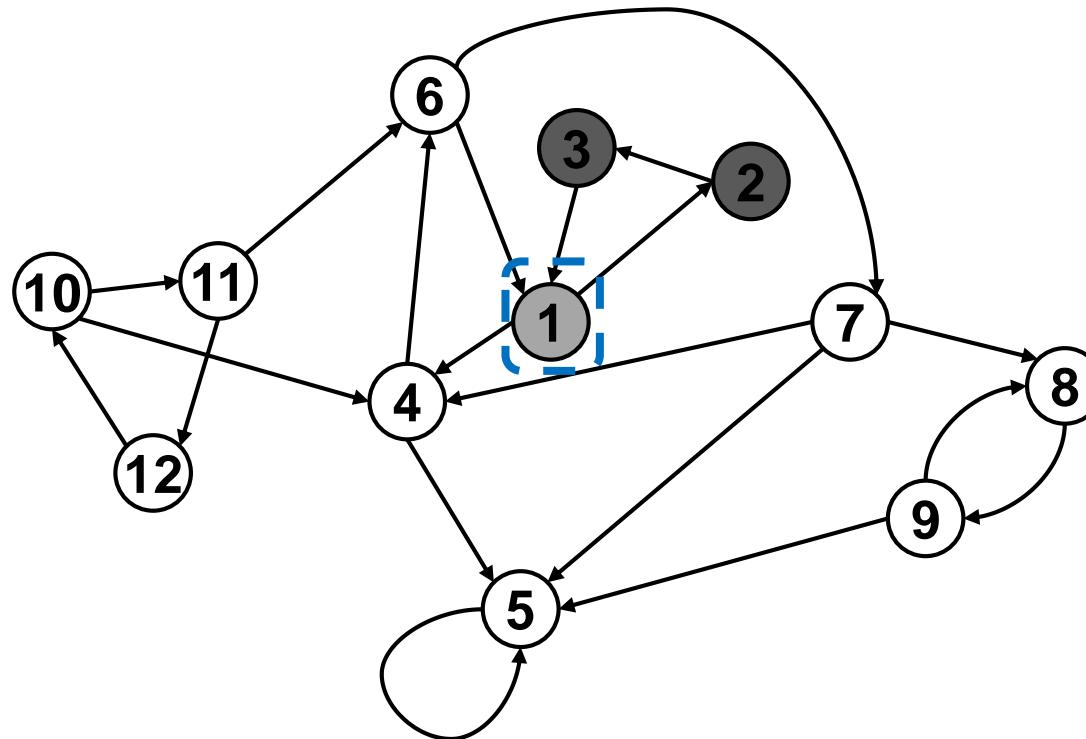
Reverse graph G^R :



L^R

SCC Example

Reverse graph G^R :

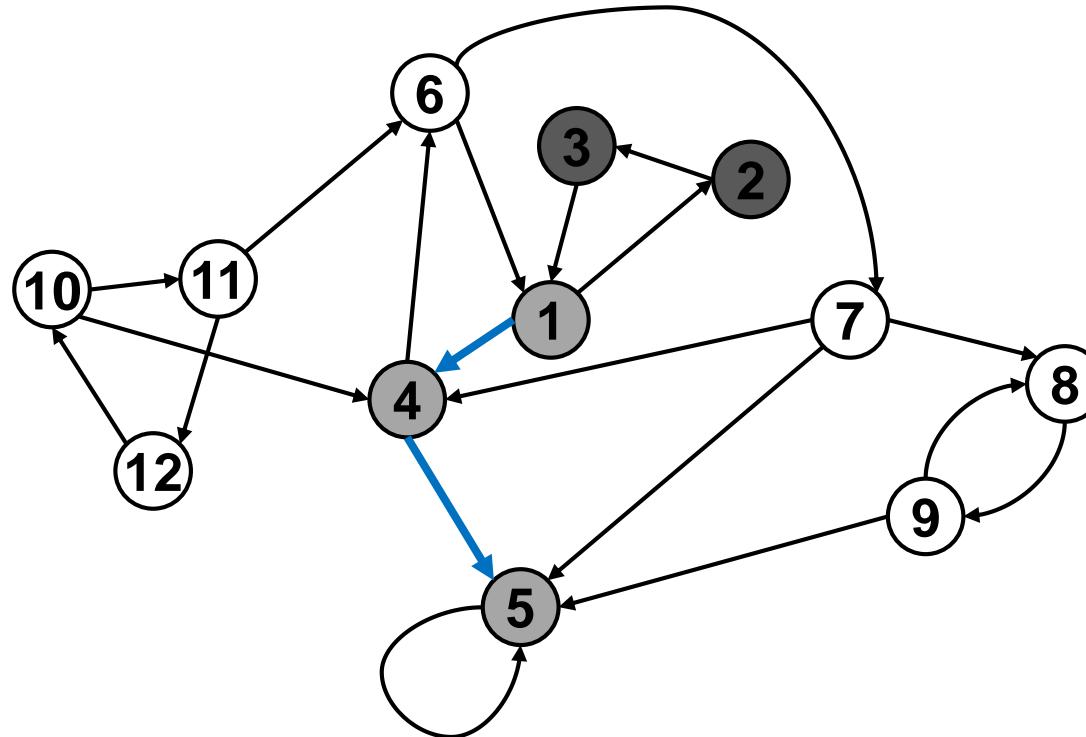


L^R

3	2
---	---

SCC Example

Reverse graph G^R :

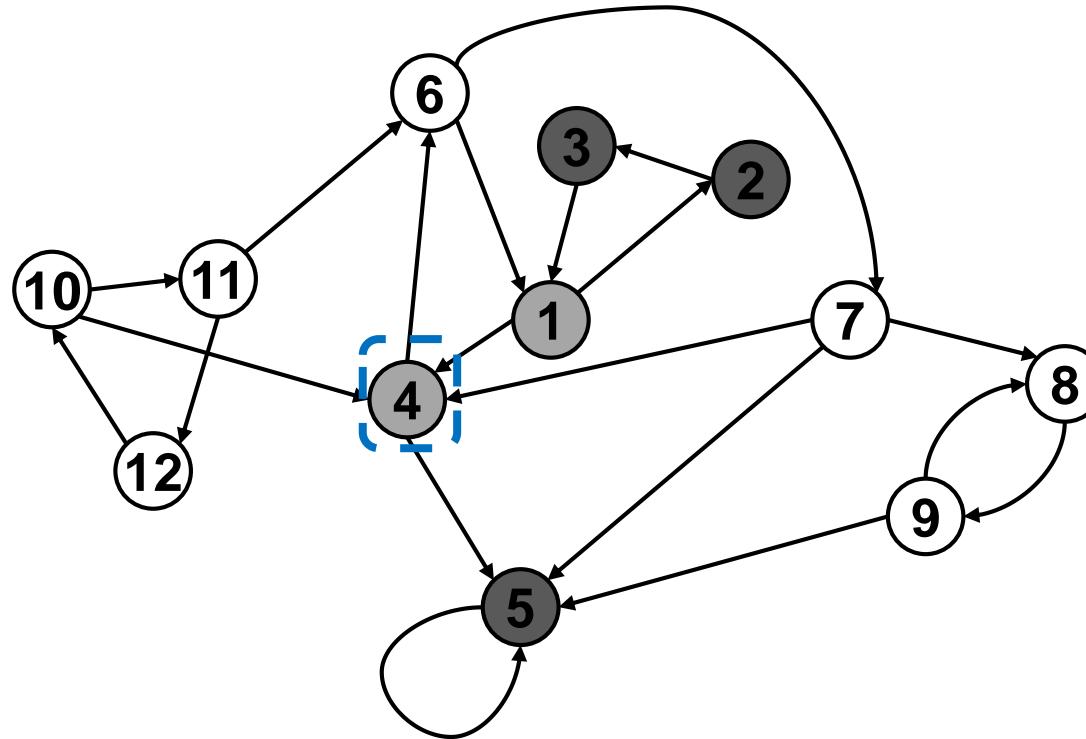


L^R

3	2
---	---

SCC Example

Reverse graph G^R :

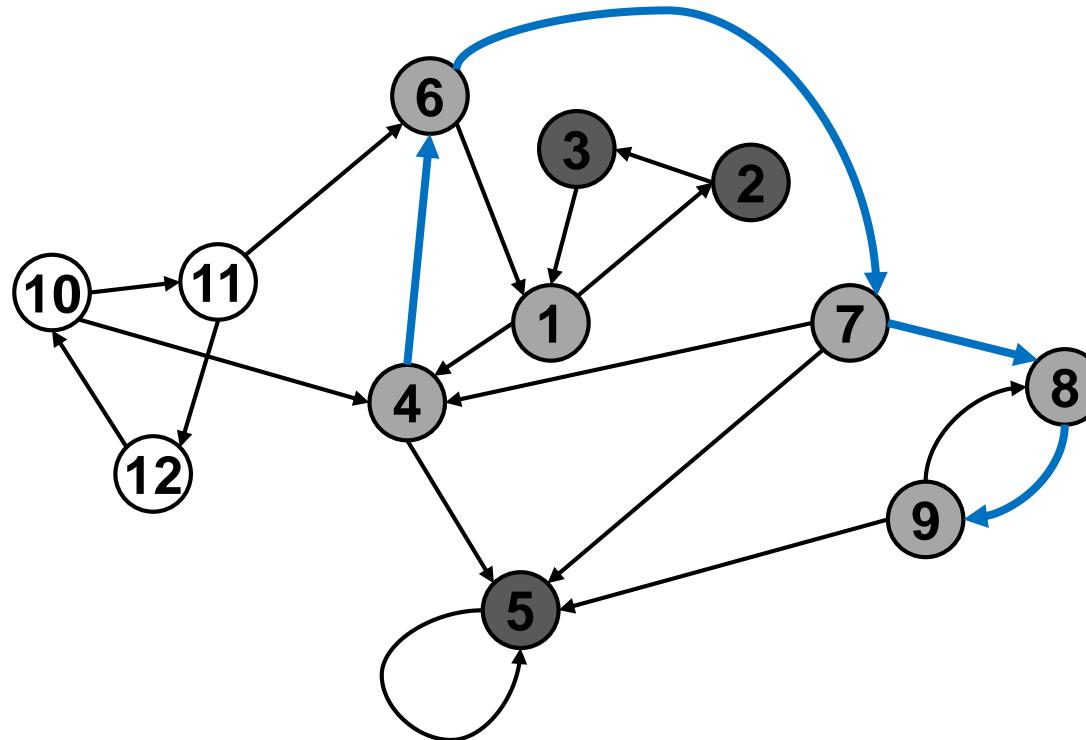


L^R

3	2	5
---	---	---

SCC Example

Reverse graph G^R :

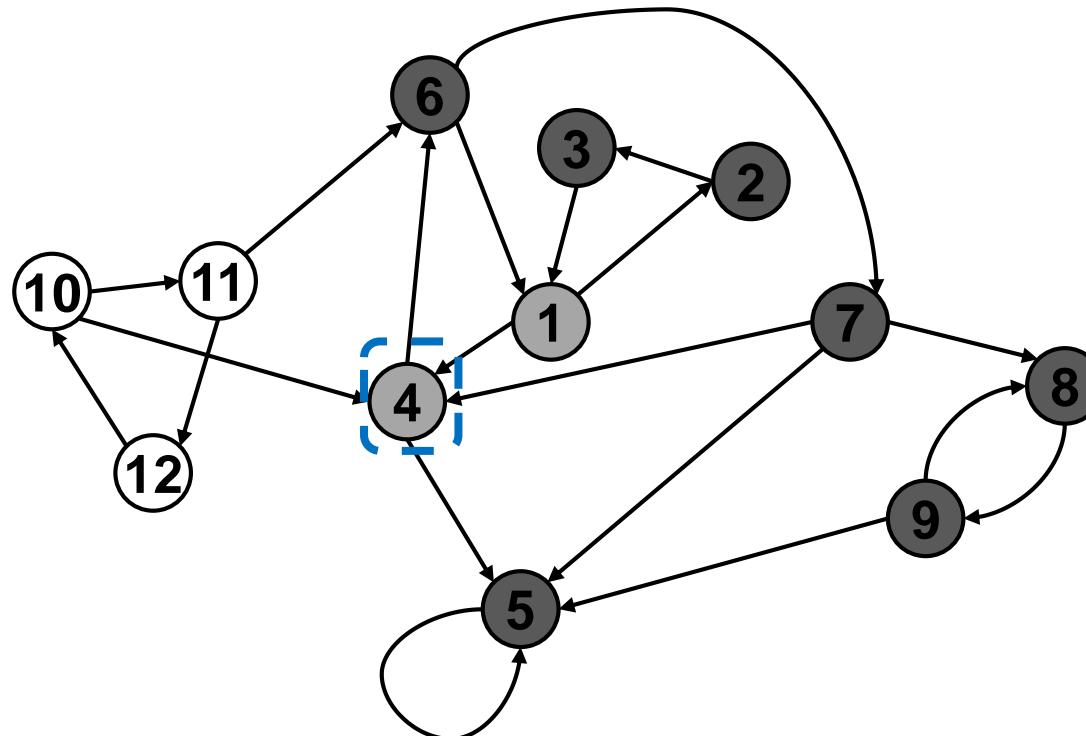


L^R

3	2	5
---	---	---

SCC Example

Reverse graph G^R :

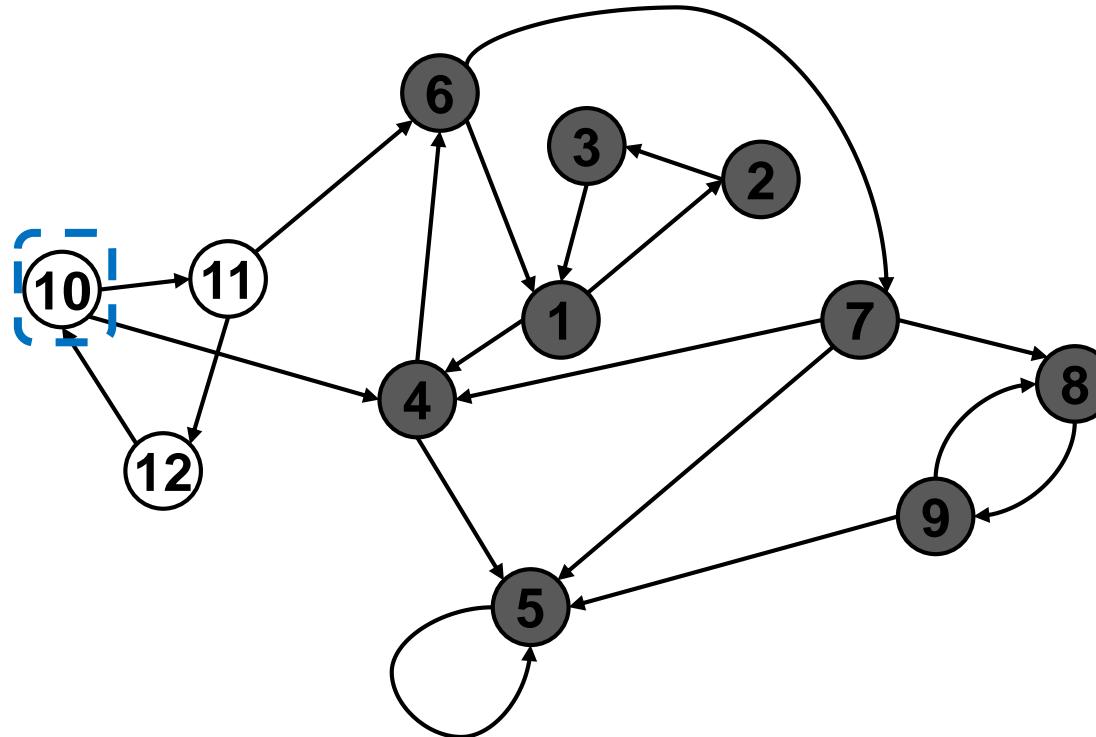


L^R

3	2	5	9	8	7	6
---	---	---	---	---	---	---

SCC Example

Reverse graph G^R :

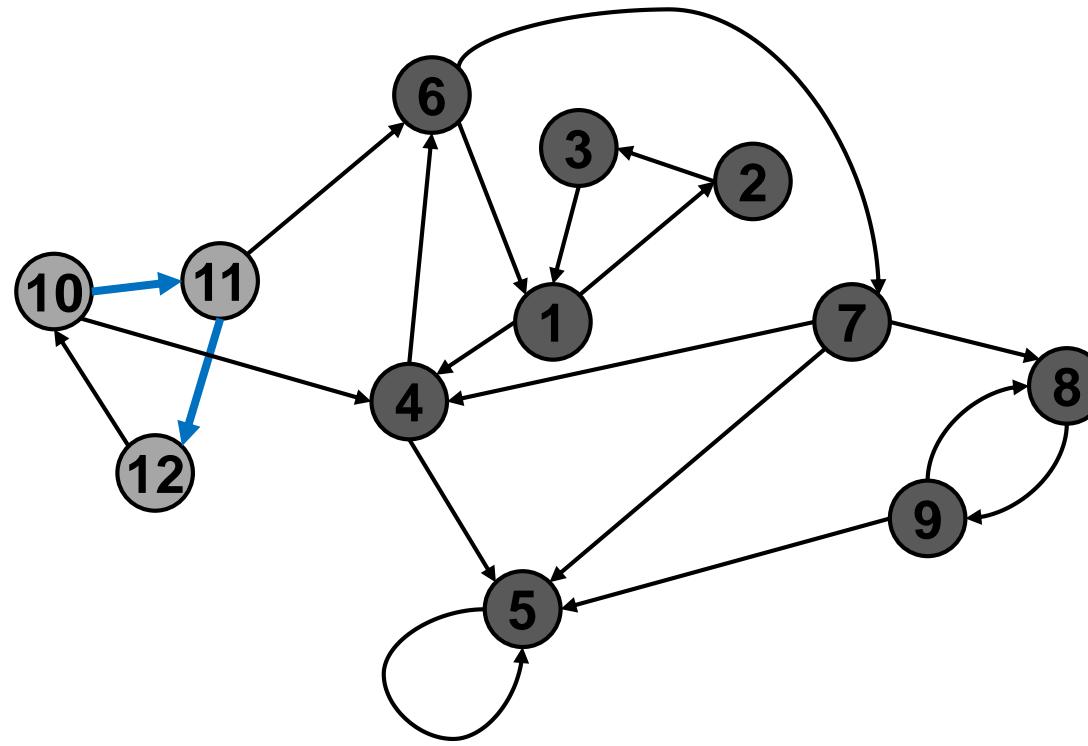


L^R

3	2	5	9	8	7	6	4	1
---	---	---	---	---	---	---	---	---

SCC Example

Reverse graph G^R :

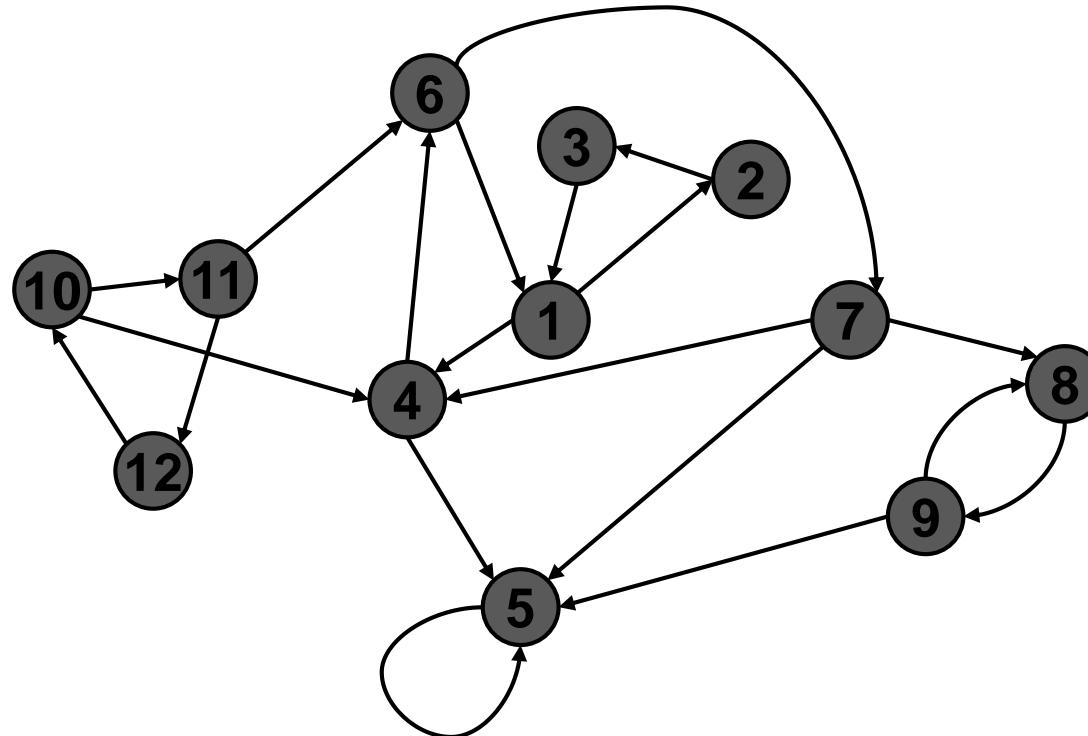


L^R

3	2	5	9	8	7	6	4	1
---	---	---	---	---	---	---	---	---

SCC Example

Reverse graph G^R :

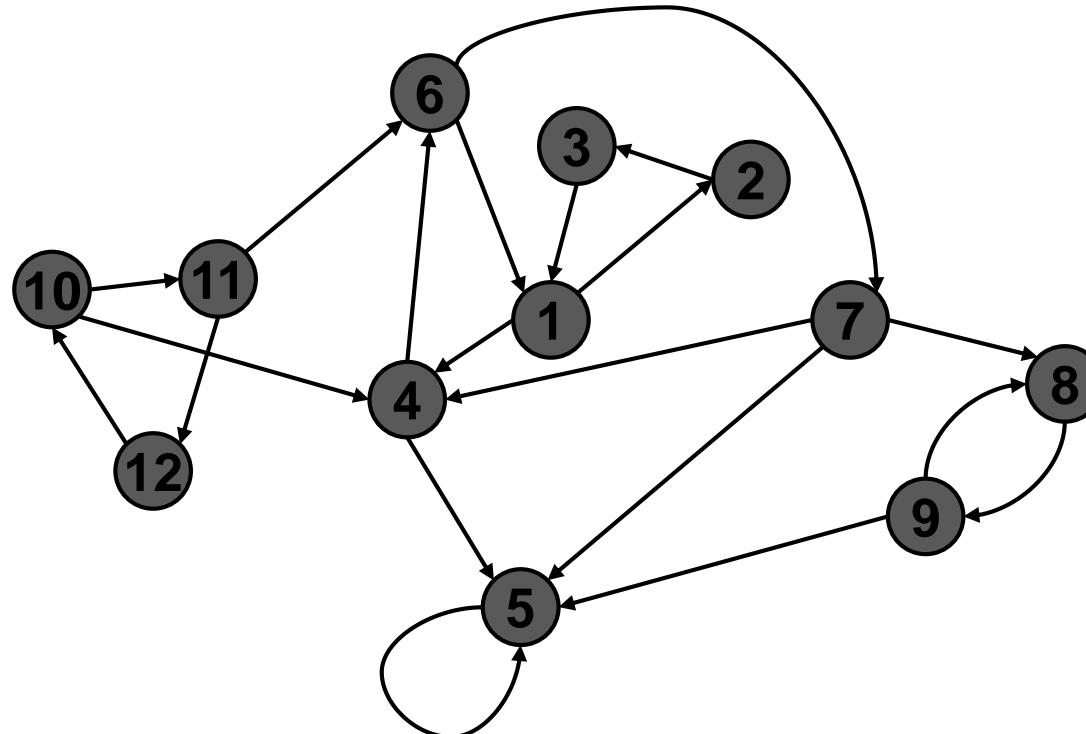


L^R

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

SCC Example

Reverse graph G^R :



L^R

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

L

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

Algorithm

Step 1: Obtain the **reverse graph** G^R by reversing the directions of all the edges in G .

Step 2: Perform DFS on G^R , and obtain the sequence L^R that the vertices in G^R turn black (i.e., whenever a vertex is popped out of the stack, append it to L^R). Obtain L as the reverse order of L^R .

Step 3: Perform DFS on the **original** graph G by obeying the following rules:

Algorithm

Step 1: Obtain the **reverse graph** G^R by reversing the directions of all the edges in G .

Step 2: Perform DFS on G^R , and obtain the sequence L^R that the vertices in G^R turn black (i.e., whenever a vertex is popped out of the stack, append it to L^R). Obtain L as the reverse order of L^R .

Step 3: Perform DFS on the **original** graph G by obeying the following rules:

- **Rule 1:** Start the DFS at the first vertex of L .

Algorithm

Step 1: Obtain the **reverse graph** G^R by reversing the directions of all the edges in G .

Step 2: Perform DFS on G^R , and obtain the sequence L^R that the vertices in G^R turn black (i.e., whenever a vertex is popped out of the stack, append it to L^R). Obtain L as the reverse order of L^R .

Step 3: Perform DFS on the **original** graph G by obeying the following rules:

- **Rule 1:** Start the DFS at the first vertex of L .
- **Rule 2:** Whenever a restart is needed, start from the first vertex of L that is still white.

Algorithm

Step 1: Obtain the **reverse graph G^R** by reversing the directions of all the edges in G .

Step 2: Perform DFS on G^R , and obtain the sequence L^R that the vertices in G^R turn black (i.e., whenever a vertex is popped out of the stack, append it to L^R). Obtain L as the reverse order of L^R .

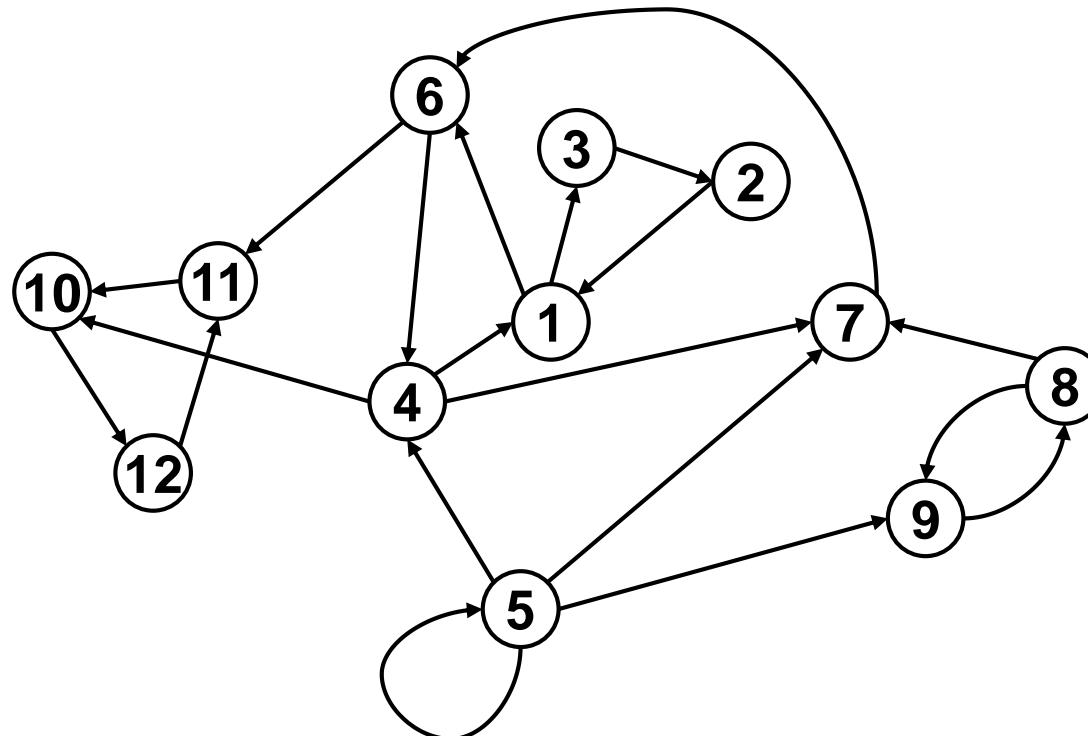
Step 3: Perform DFS on the **original graph G** by obeying the following rules:

- **Rule 1:** Start the DFS at the first vertex of L .
- **Rule 2:** Whenever a restart is needed, start from the first vertex of L that is still white.

Output the vertices in each DFS-tree as an SCC.

SCC Example

The original graph G :

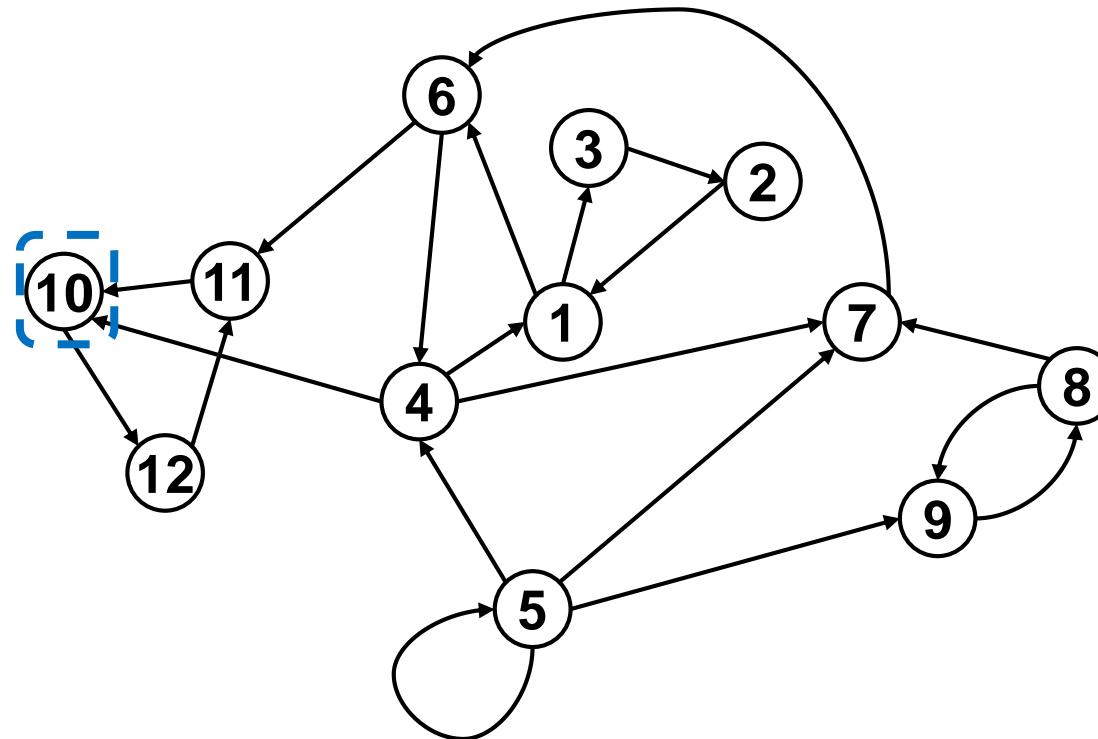


L

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

SCC Example

The original graph G :

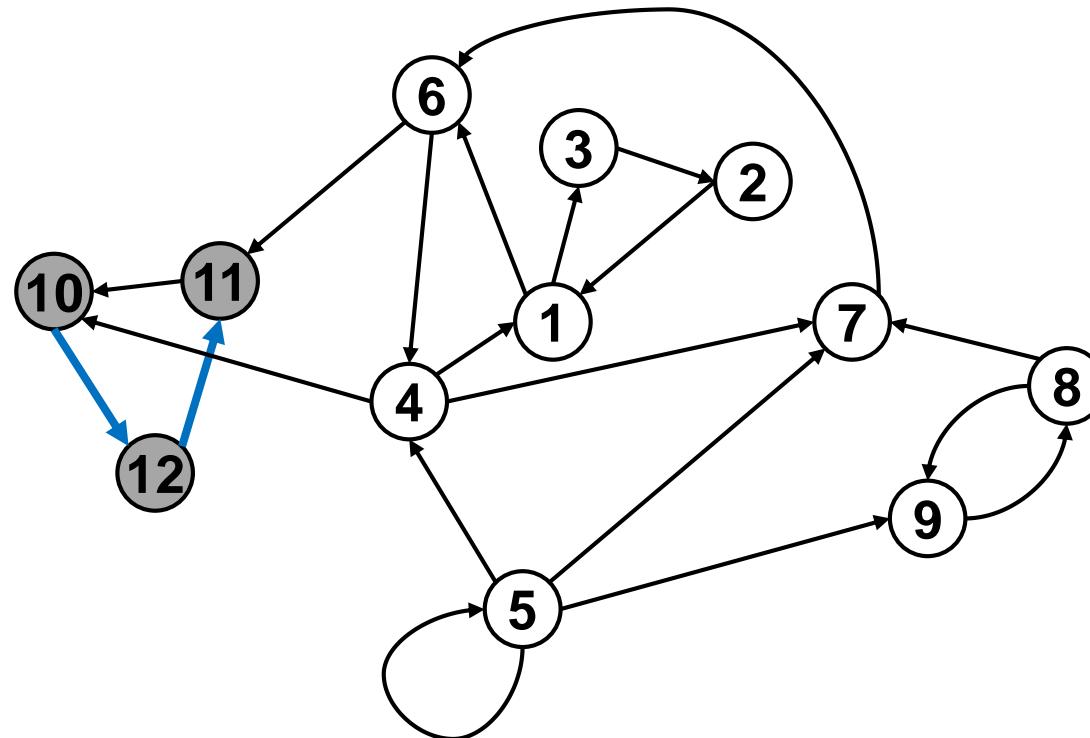


L

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

SCC Example

The original graph G :

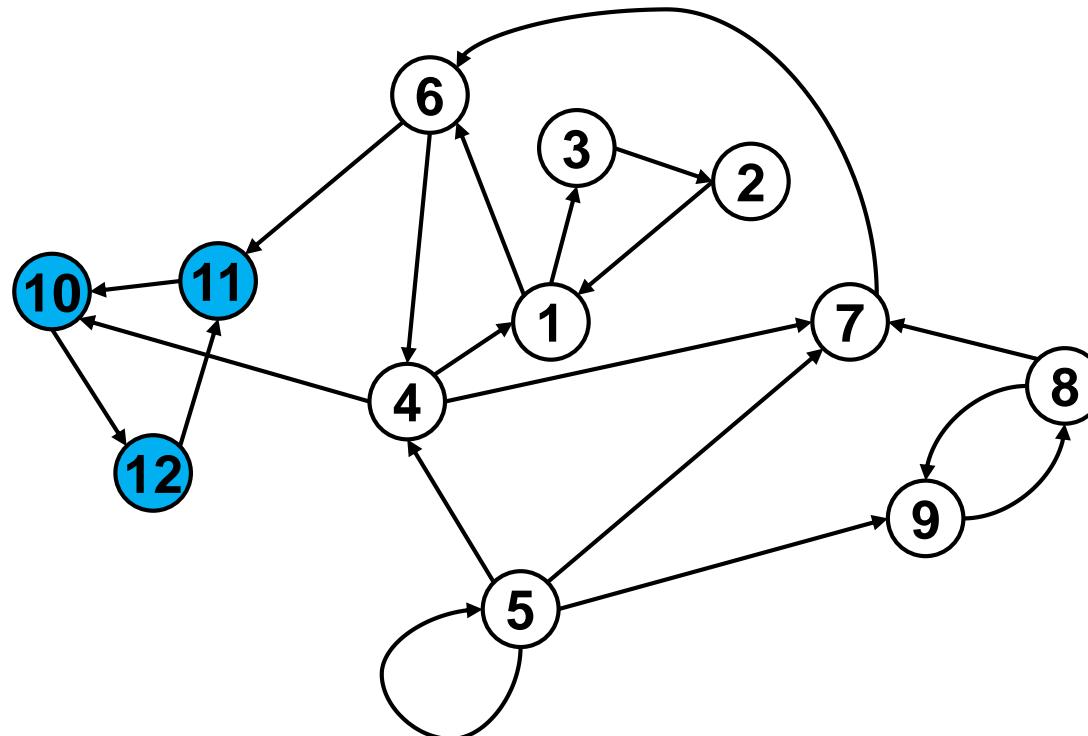


L

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

SCC Example

The original graph G :

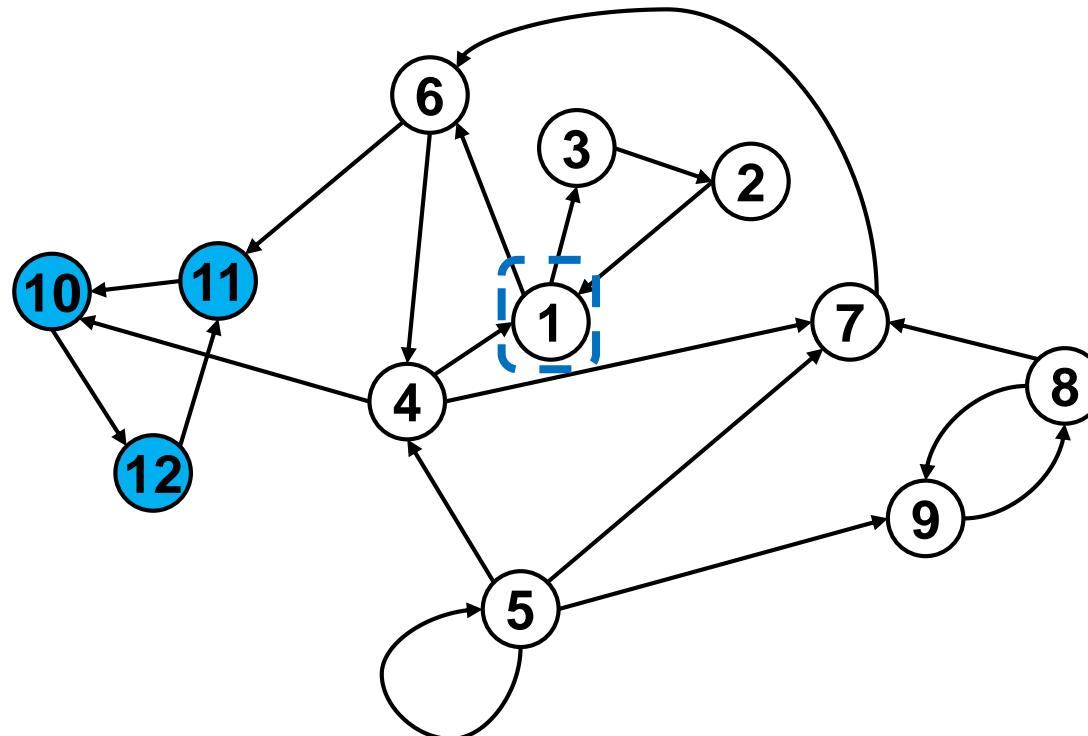


L

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

SCC Example

The original graph G :

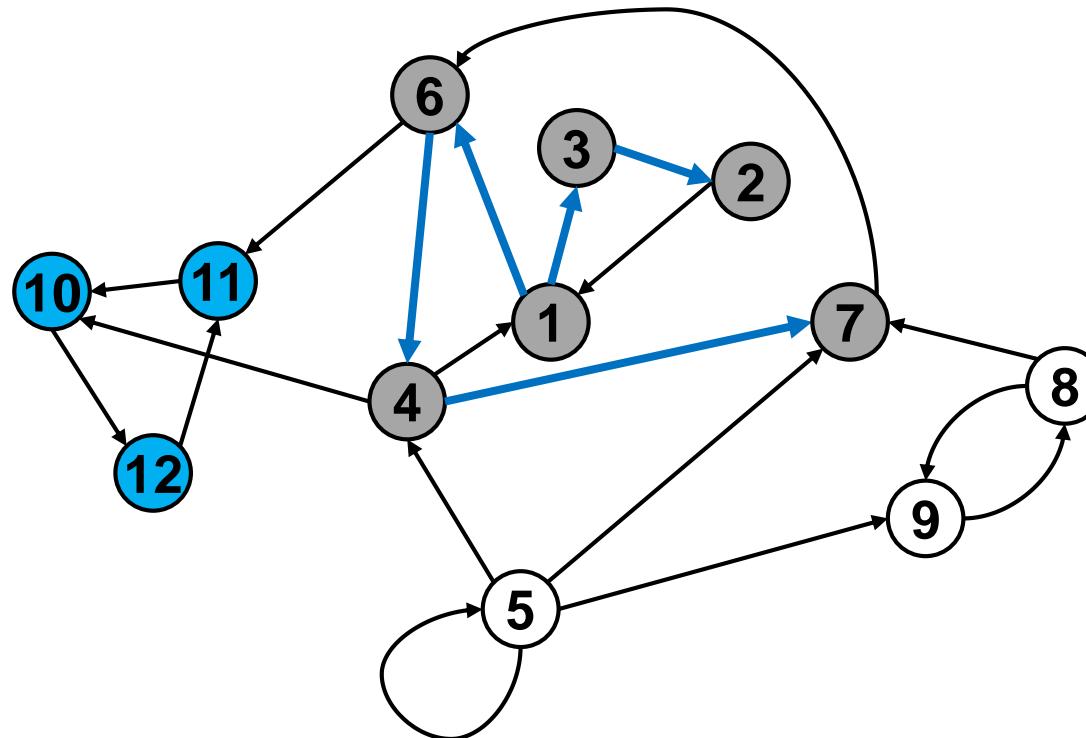


L

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

SCC Example

The original graph G :

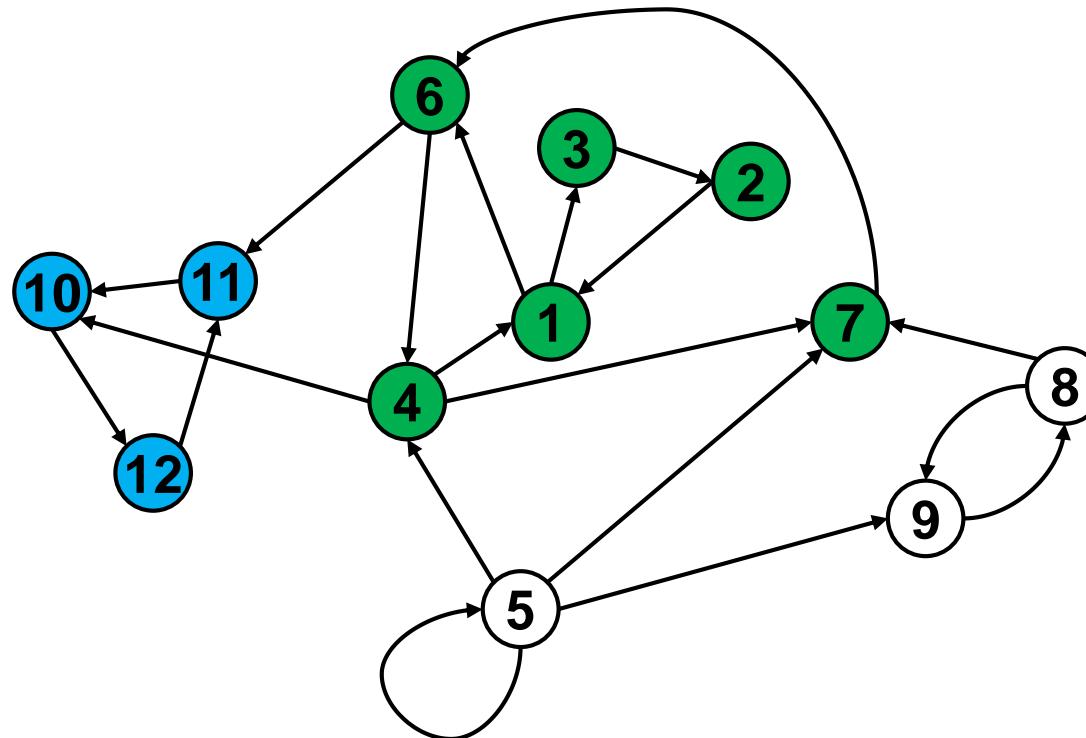


L

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

SCC Example

The original graph G :

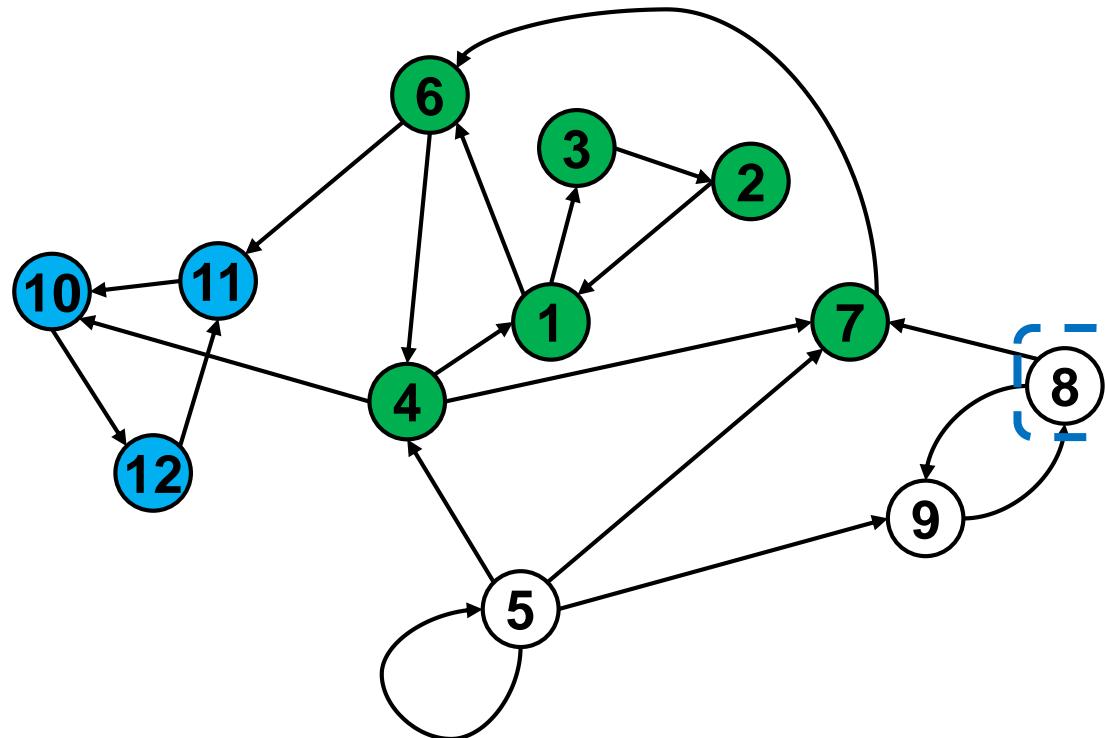


L

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

SCC Example

The original graph G :

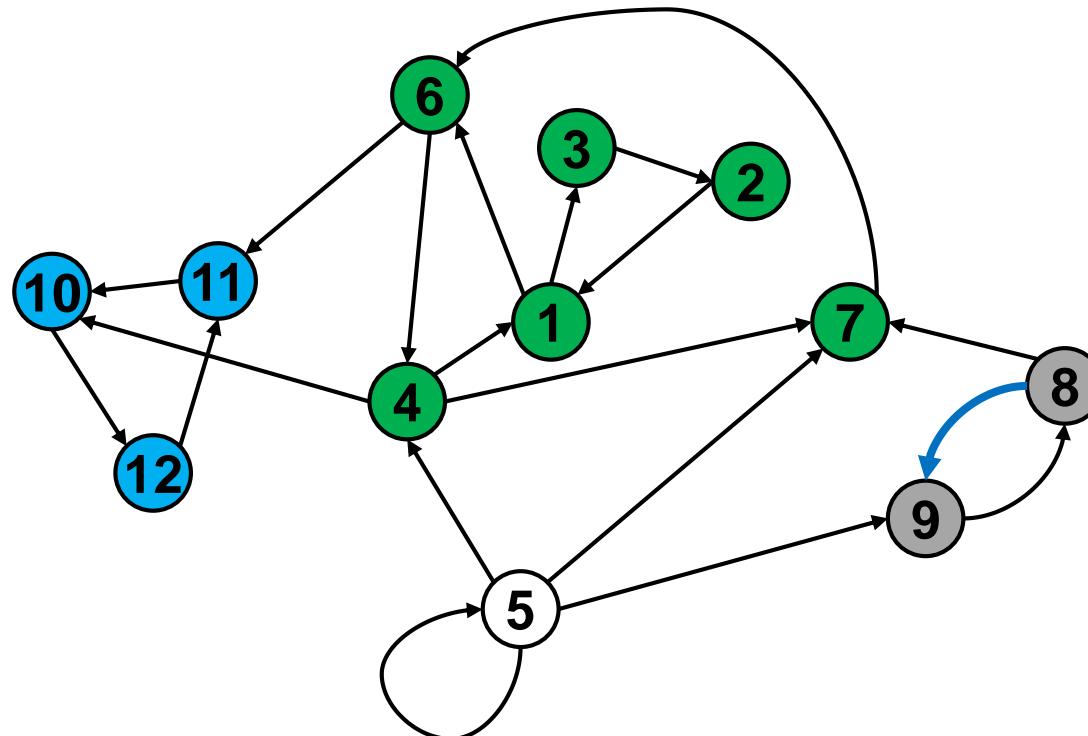


L

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

SCC Example

The original graph G :

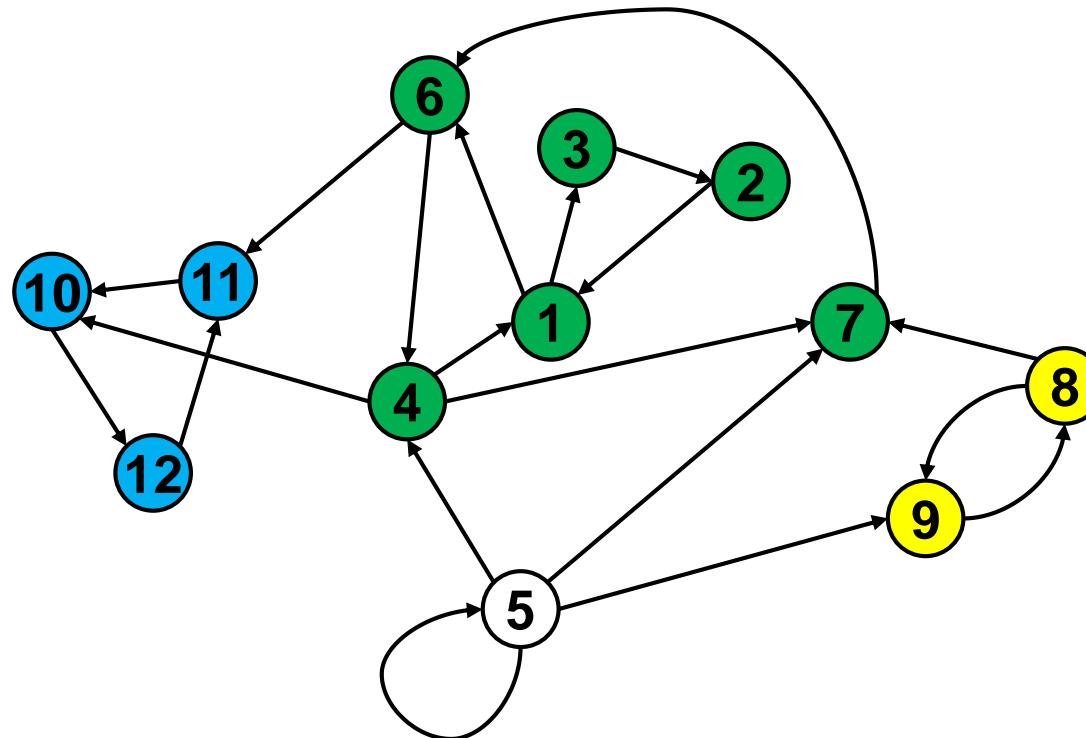


L

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

SCC Example

The original graph G :

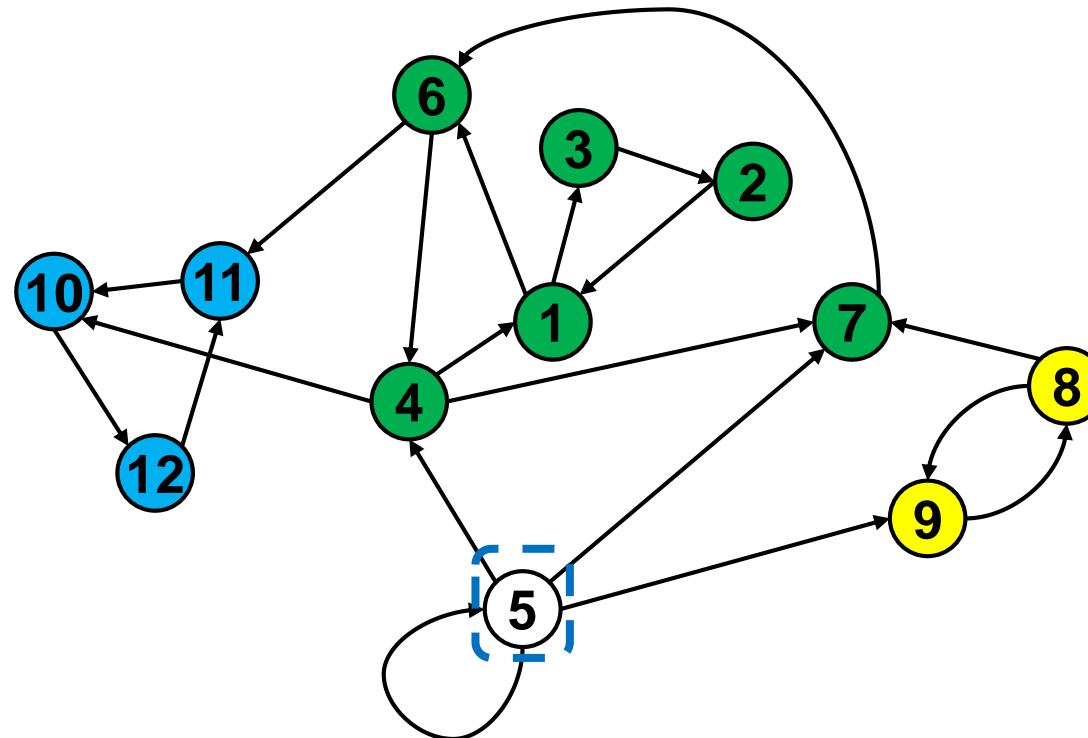


L

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

SCC Example

The original graph G :

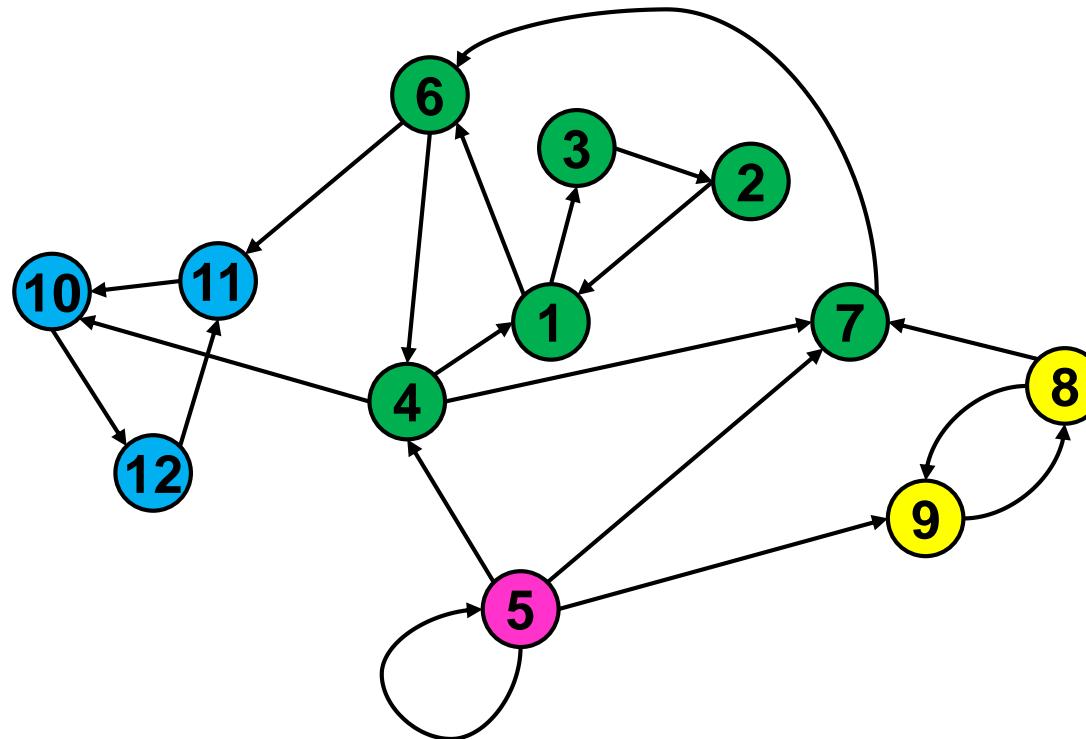


L

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

SCC Example

The original graph G :

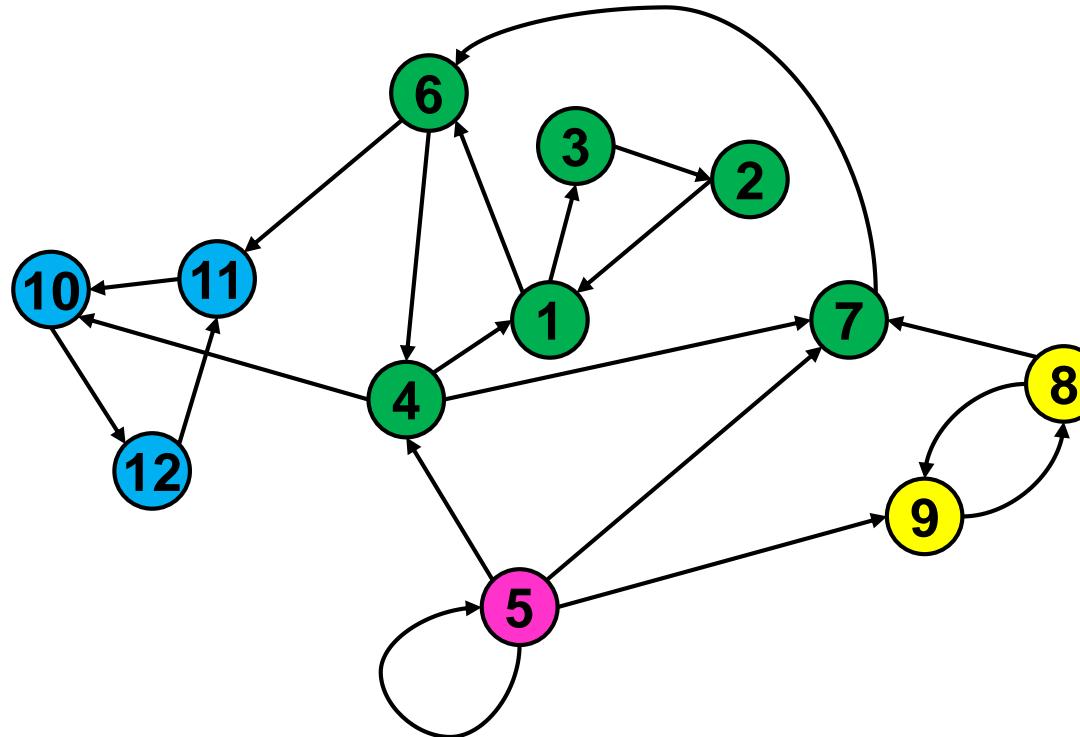


L

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

SCC Example

The original graph G :



Graph G contains 4 SCCs:

$$\{10, 11, 12\}, \{1, 2, 3, 4, 6, 7\}, \{8, 9\}, \{5\}$$

Pseudocode

SCC(G)

Input: A directed graph \mathbf{G}

Output: The set of strongly connected components \mathbf{R}

$R \leftarrow \{\}; // \text{ set of SCCs}$

Pseudocode

SCC(G)

Input: A directed graph \mathbf{G}

Output: The set of strongly connected components \mathbf{R}

$R \leftarrow \{\}; // \text{ set of SCCs}$

$G^R \leftarrow \text{reverse graph of } G;$

Pseudocode

$\text{SCC}(G)$

Input: A directed graph G

Output: The set of strongly connected components R

$R \leftarrow \{\}; // \text{ set of SCCs}$

$G^R \leftarrow \text{reverse graph of } G;$

$L^R \leftarrow \text{DFS-b}(G^R); // \text{ Perform DFS}$

$L \leftarrow \text{reverse order of } L^R;$

Pseudocode

$\text{SCC}(G)$

Input: A directed graph G

Output: The set of strongly connected components R

$R \leftarrow \{\}; // \text{ set of SCCs}$

$G^R \leftarrow \text{reverse graph of } G;$

$L^R \leftarrow \text{DFS-b}(G^R); // \text{ Perform DFS}$

$L \leftarrow \text{reverse order of } L^R;$

for $u \in L$ **do**

if

Pseudocode

SCC(G)

Input: A directed graph \mathbf{G}

Output: The set of strongly connected components \mathbf{R}

$R \leftarrow \{\}; // \text{ set of SCCs}$

$G^R \leftarrow \text{reverse graph of } G;$

$L^R \leftarrow \text{DFS-b}(G^R); // \text{ Perform DFS}$

$L \leftarrow \text{reverse order of } L^R;$

for $u \in L$ **do**

| **if** $\text{color}[u]$ *is equal to* WHITE **then**

Pseudocode

$\text{SCC}(G)$

Input: A directed graph G

Output: The set of strongly connected components R

$R \leftarrow \{\}; // \text{ set of SCCs}$

$G^R \leftarrow \text{reverse graph of } G;$

$L^R \leftarrow \text{DFS-b}(G^R); // \text{ Perform DFS}$

$L \leftarrow \text{reverse order of } L^R;$

for $u \in L$ **do**

if $\text{color}[u]$ *is equal to WHITE* **then**

$L_{\text{scc}} \leftarrow \text{DFSVISIT}(G, u); // \text{ Perform DFS starting at } u$

Pseudocode

$\text{SCC}(G)$

Input: A directed graph G

Output: The set of strongly connected components R

$R \leftarrow \{\}; // \text{ set of SCCs}$

$G^R \leftarrow \text{reverse graph of } G;$

$L^R \leftarrow \text{DFS-b}(G^R); // \text{ Perform DFS}$

$L \leftarrow \text{reverse order of } L^R;$

for $u \in L$ **do**

if $\text{color}[u]$ is equal to WHITE **then**

$L_{\text{scc}} \leftarrow \text{DFSVisit}(G, u); // \text{ Perform DFS starting at } u$

$R \leftarrow R \cup \text{Set}(L_{\text{scc}});$

Pseudocode

$\text{SCC}(G)$

Input: A directed graph G
Output: The set of strongly connected components R

```
 $R \leftarrow \{\}; // \text{set of SCCs}$ 
 $G^R \leftarrow \text{reverse graph of } G;$ 
 $L^R \leftarrow \text{DFS-b}(G^R); // \text{Perform DFS}$ 
 $L \leftarrow \text{reverse order of } L^R;$ 
for  $u \in L$  do
    if  $\text{color}[u]$  is equal to WHITE then
         $L_{\text{scc}} \leftarrow \text{DFSVisit}(G, u); // \text{Perform DFS starting at } u$ 
         $R \leftarrow R \cup \text{Set}(L_{\text{scc}});$ 
    end
end
return  $R;$ 
```

Outline

- Introduction to Part IV
- Review of Basic Graph Search Algorithms
 - Basic Concepts
 - The Breadth-First Search (BFS) Algorithm
 - The Depth-First Search (DFS) Algorithm
- Topological Sort
 - The Topological Sort Algorithm
 - Analysis of the Topological Sort Algorithm
- **Strongly Connected Components**
 - The Algorithm of Finding SCCs
 - **Analysis of the Algorithm**

Time Analysis

- Steps 1 and 2 obviously require only $O(|V| + |E|)$ time.

Time Analysis

- Steps 1 and 2 obviously require only $O(|V| + |E|)$ time.
- Regarding Step 3, the DFS itself takes $O(|V| + |E|)$ time, but we still need to discuss the time to implement Rule 2.

Time Analysis

- Steps 1 and 2 obviously require only $O(|V| + |E|)$ time.
- Regarding Step 3, the DFS itself takes $O(|V| + |E|)$ time, but we still need to discuss the time to implement Rule 2. Namely, whenever DFS needs a restart, how do we find the first white vertex in L efficiently? This will be left as an exercise.

Time Analysis

- Steps 1 and 2 obviously require only $O(|V| + |E|)$ time.
- Regarding Step 3, the DFS itself takes $O(|V| + |E|)$ time, but we still need to discuss the time to implement Rule 2. Namely, whenever DFS needs a restart, how do we find the first white vertex in L efficiently? This will be left as an exercise.
- Hence, the overall execution time is $O(|V| + |E|)$.

SCC Graph

Let \mathbf{G} be the input directed graph, with SCCs S_1, S_2, \dots, S_t for some $t \geq 1$.

Let us define a **SCC graph \mathbf{G}^{SCC}** as follows:

SCC Graph

Let \mathbf{G} be the input directed graph, with SCCs S_1, S_2, \dots, S_t for some $t \geq 1$.

Let us define a **SCC graph G^{SCC}** as follows:
Each vertex in G^{SCC} is a distinct SCC in G .

SCC Graph

Let G be the input directed graph, with SCCs S_1, S_2, \dots, S_t for some $t \geq 1$.

Let us define a **SCC graph** G^{SCC} as follows:

Each vertex in G^{SCC} is a distinct SCC in G .

Consider two vertices (a.k.a. SCCs) S_i and S_j ($1 \leq i, j \leq t$).

G^{SCC} has an edge from S_i to S_j if and only if

SCC Graph

Let G be the input directed graph, with SCCs S_1, S_2, \dots, S_t for some $t \geq 1$.

Let us define a **SCC graph G^{SCC}** as follows:

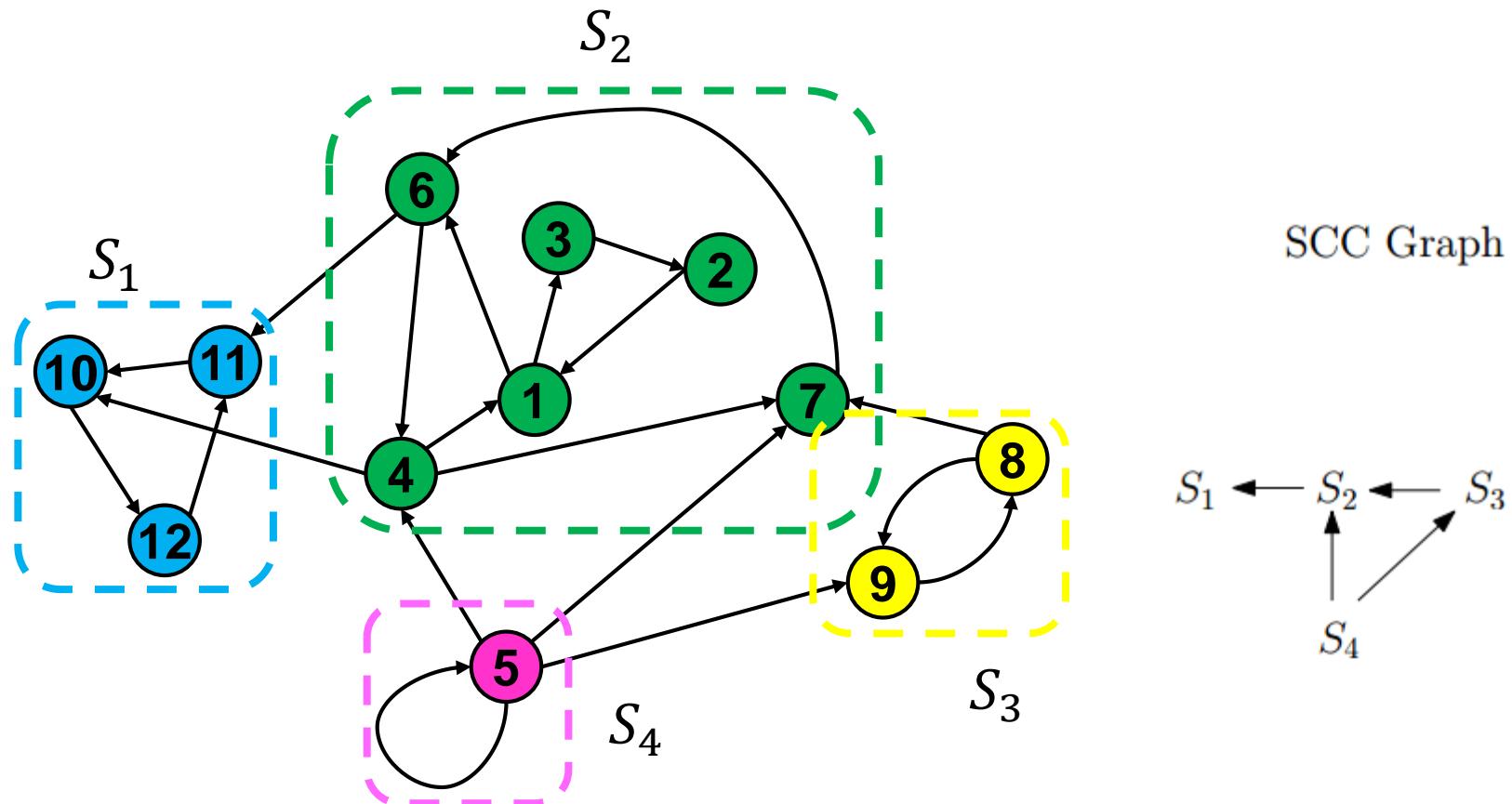
Each vertex in G^{SCC} is a distinct SCC in G .

Consider two vertices (a.k.a. SCCs) S_i and S_j ($1 \leq i, j \leq t$).

G^{SCC} has an edge from S_i to S_j if and only if

- $i \neq j$, and
- There is a path in G from a vertex in S_i to a vertex in S_j .

SCC Graph - Example



SCC Graph

Lemma: G^{SCC} is a DAG.

Proof: Suppose that there is a cycle in G^{SCC} , which must involve at least 2 SCCs (say S_i, S_j) as no vertex in G^{SCC} has an edge to itself.

SCC Graph

Lemma: G^{SCC} is a DAG.

Proof: Suppose that there is a cycle in G^{SCC} , which must involve at least 2 SCCs (say S_i, S_j) as no vertex in G^{SCC} has an edge to itself.

Then, any vertex in S_i is reachable from any vertex in S_j , and vice versa. This violates the fact that S_i, S_j are SCCs (violating maximality).

SCC Graph

Define an SCC as a **sink SCC** if it has no outgoing edge in G^{SCC} .

Lemma: There must be at least one sink SCC in G^{SCC} .

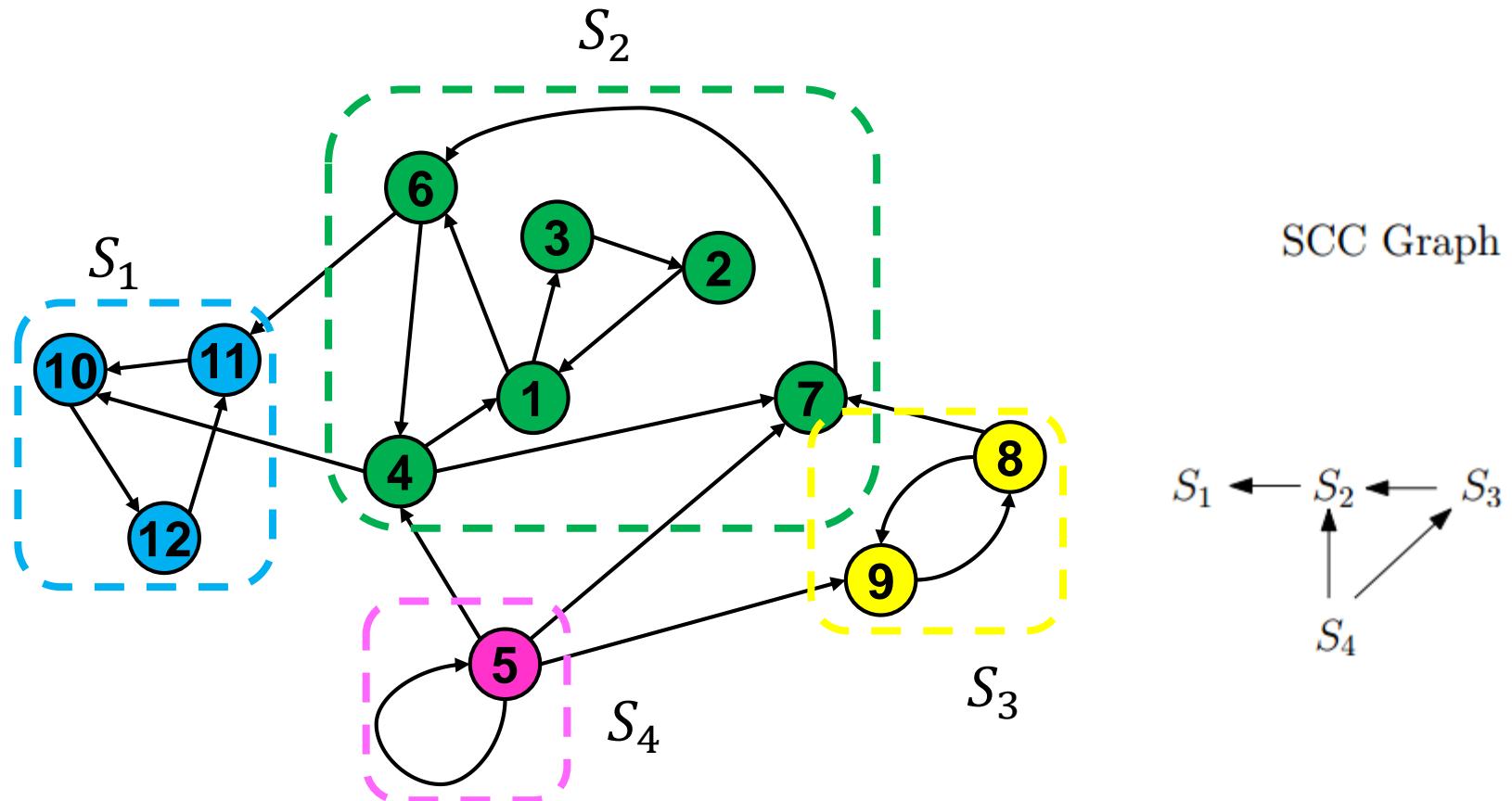
SCC Graph

Define an SCC as a **sink SCC** if it has no outgoing edge in G^{SCC} .

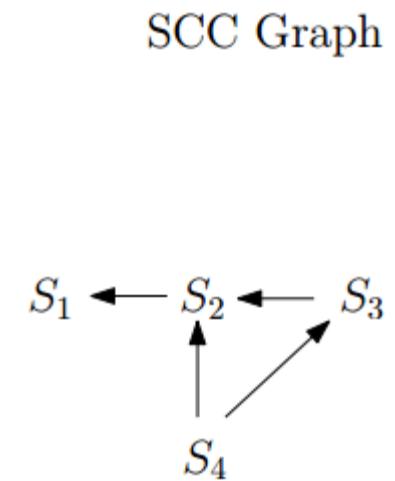
Lemma: There must be at least one sink SCC in G^{SCC} .

Proof: Since G^{SCC} is a DAG, it admits a topological order. The last vertex of the topological order cannot have any outgoing edges.

SCC Graph - Example



S_1 is a sink vertex.



DFS in a Sink SCC

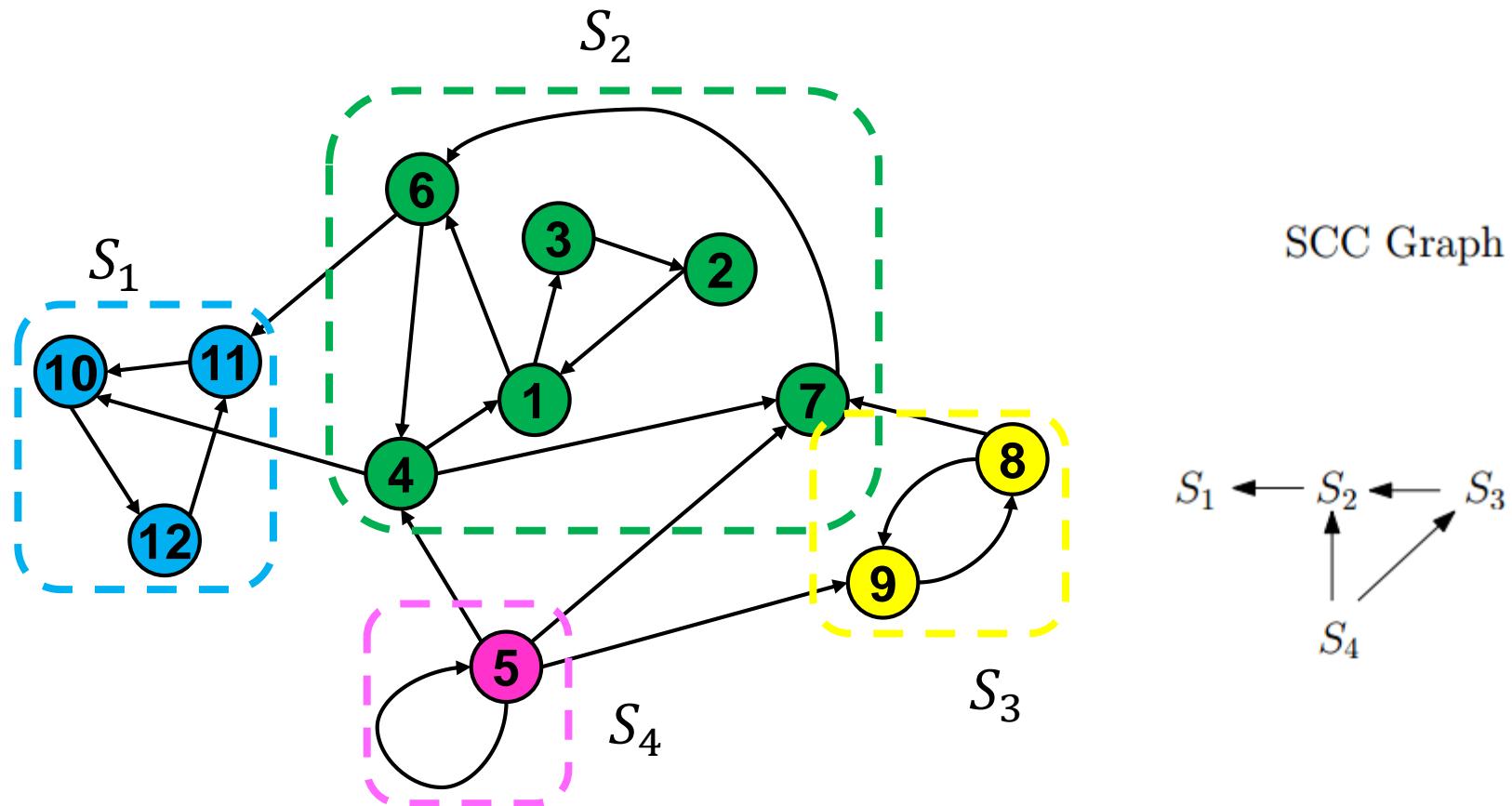
Lemma: Let S be a sink SCC of G^{SCC} . Suppose that we perform a DFS starting from any vertex in S . Then the first DFS-tree output must include all and only the vertices in S .

DFS in a Sink SCC

Lemma: Let S be a sink SCC of G^{SCC} . Suppose that we perform a DFS starting from any vertex in S . Then the first DFS-tree output must include all and only the vertices in S .

Proof: Let $v \in S$ be the starting vertex of DFS. By the visiting process of DFS, the DFS-tree must include all the vertices that v can reach. These are exactly the vertices in S .

DFS in a Sink SCC - Example



Performing DFS from any vertex in S_1 will discover S_1 as the first SCC.

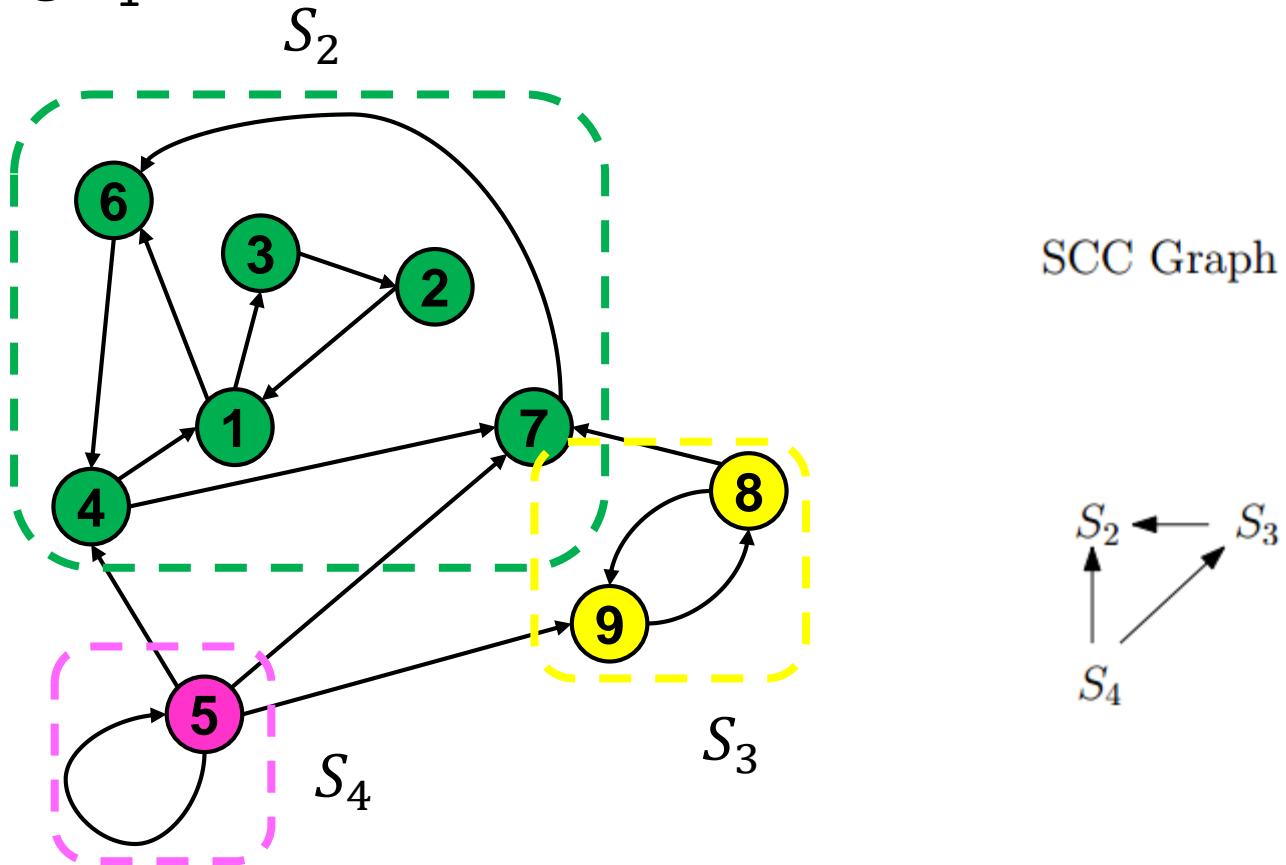
Finding SCCs - The Strategy

The previous lemma suggests the following strategy for finding all the SCCs:

- Performing DFS from any vertex in a sink SCC S .
- Delete all the vertices of S from G , as well as their edges.
- Accordingly, delete S from G^{SCC} , as well as its edges.
- Repeat from Step 1, until G is empty.

Finding SCCs - Example

After deleting S_1 , we have:

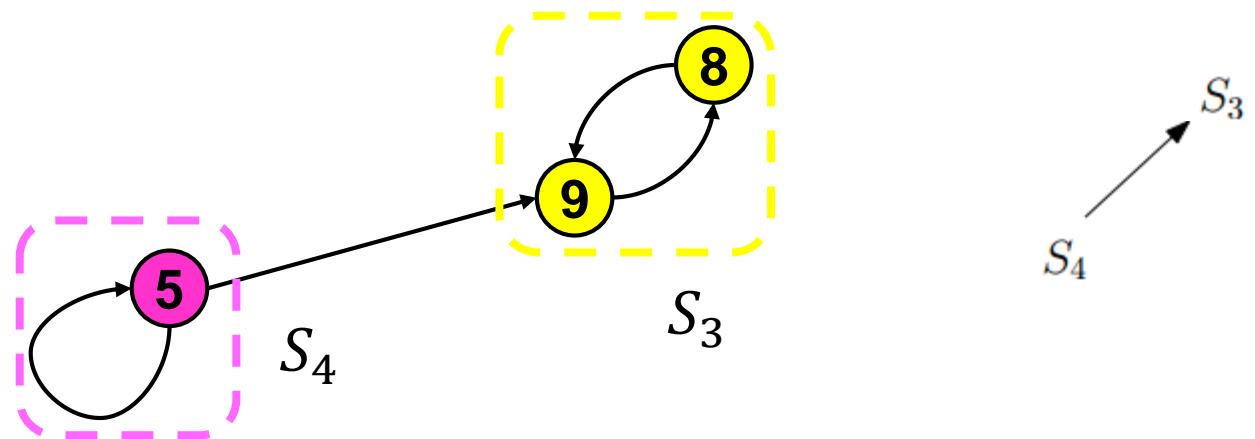


Now, S_2 becomes the sink SCC. Performing DFS from any vertex in S_2 discovers S_2 as the **second** SCC.

Finding SCCs - Example

After deleting S_2 , we have:

SCC Graph



Now, S_3 becomes the sink SCC. Performing DFS from any vertex in S_3 discovers S_3 as the **third** SCC.

Finding SCCs - Example

After deleting S_3 , we have:

SCC Graph



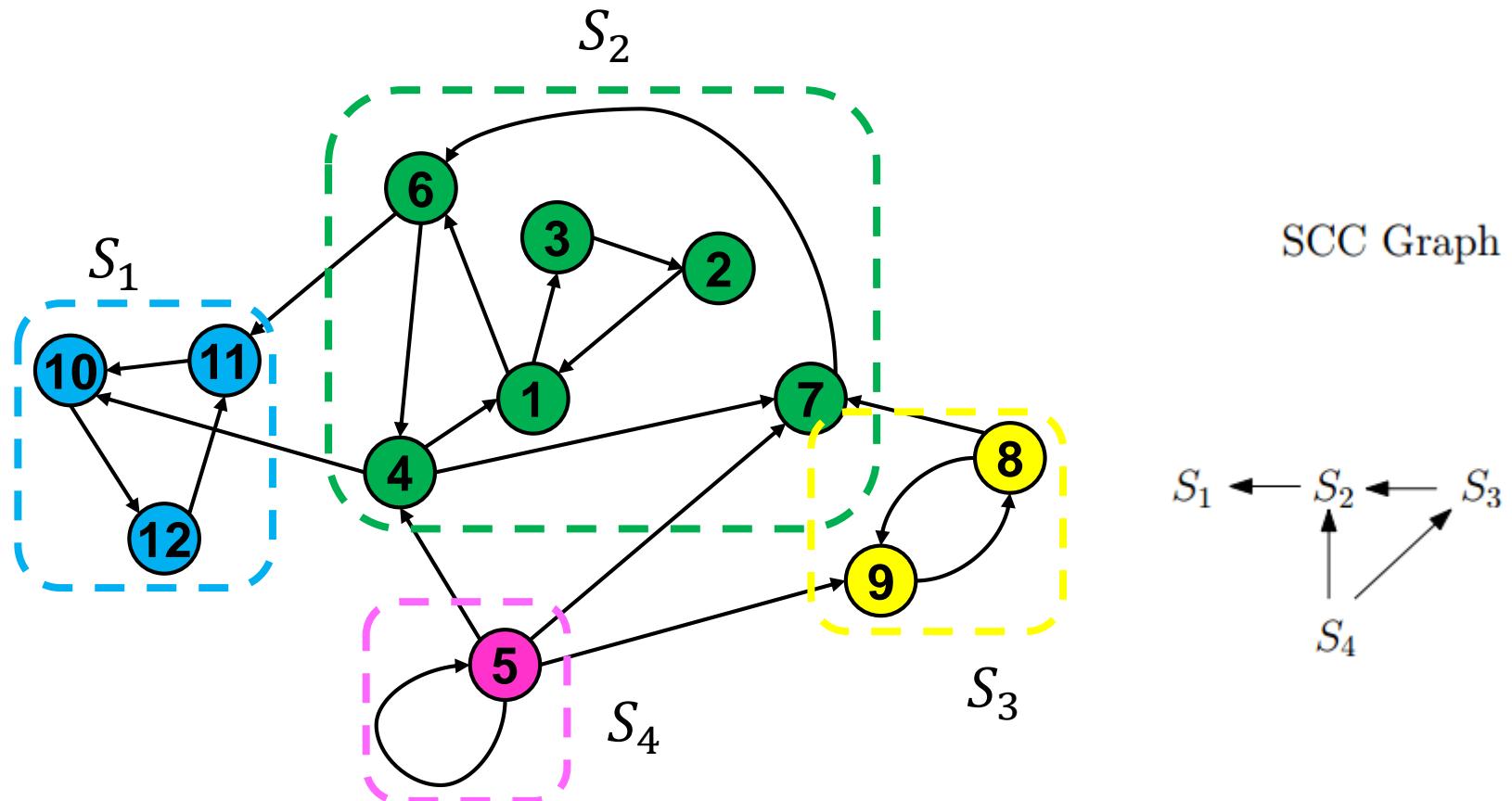
Now, S_4 becomes the sink SCC. Performing DFS from any vertex in S_4 discovers S_4 as the **last** SCC.

Correctness

A Property of the Ordering L

Lemma: Let S_1, S_2 be SCCs such that there is a path from S_1 to S_2 in G^{SCC} . In the ordering of L , the **earliest vertex in S_2** must come **before** the **earliest vertex in S_1** .

Correctness - Example



Recall that we obtained earlier

$L = (10, 11, 12, 1, 4, 6, 7, 8, 9, 5, 3, 2)$. The red vertices 10, 1, 8, 5 are, respectively, the earliest vertex in L of S_1 , S_2 , S_3 , and S_4 .

Correctness

This essentially completes the proof of the correctness of our SCC algorithm.

You may want to ask: but we never **delete** any vertices from G ! In fact, we did, as far as DFS is concerned.

To see this, recall that DFS colors all the "done" vertices black. These vertices are never touched again, and hence, effectively deleted.

