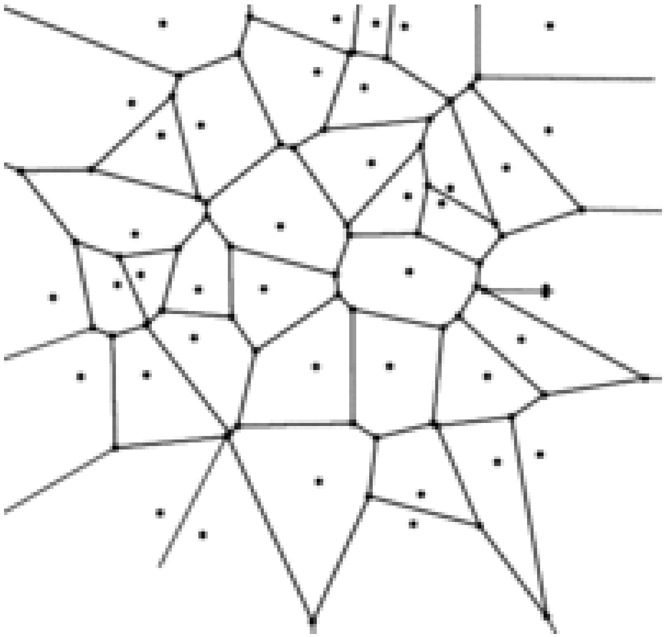
# Data Structures & Algorithms

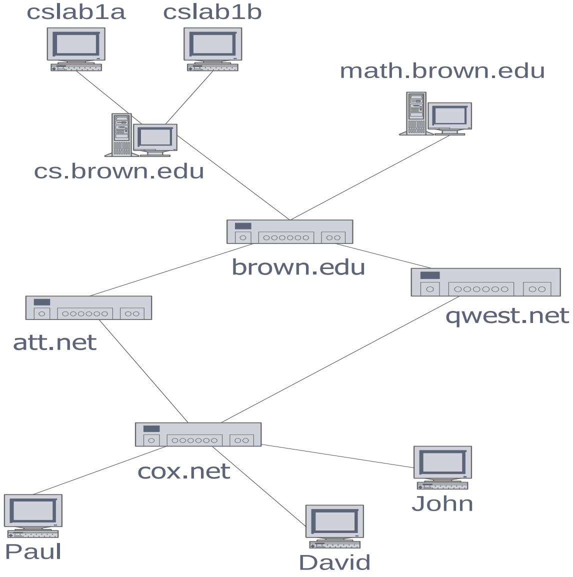
Graph

Maastricht University

# When do we need Graphs?

- Representations of:
  - Social networks
  - Computer networks
  - Class diagrams
  - Geographic relations
  - Routes on a map
  - …





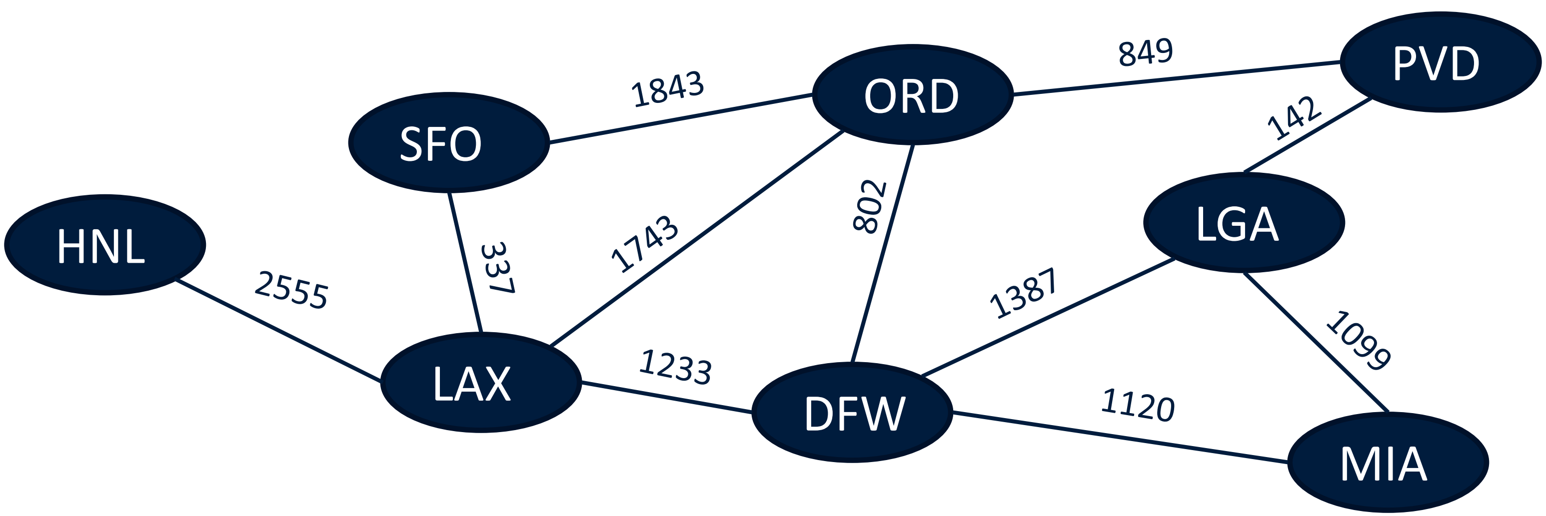
# Graphs

- Formally, a graph is a pair **(V, E)** where:
  - ***V*** is a set of *vertices* (or nodes)
  - ***E*** is a set of edges
    - Vertices and edges can store information
    - Vertices usually hold elements
    - Edges can have a weight, and other info

- Generally: $n = |V|$, and $m = |E|$

# Graphs – Flight routes

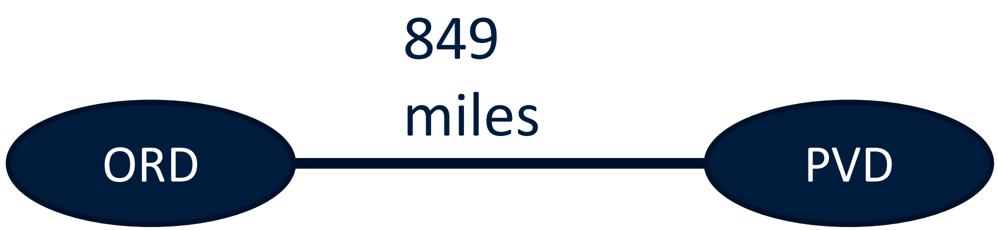
- Vertices represent airports and store the three-letter airport code
- Edges represent flight routes between two airports, store the distance of the route



# Edge Types – Directed / Undirected

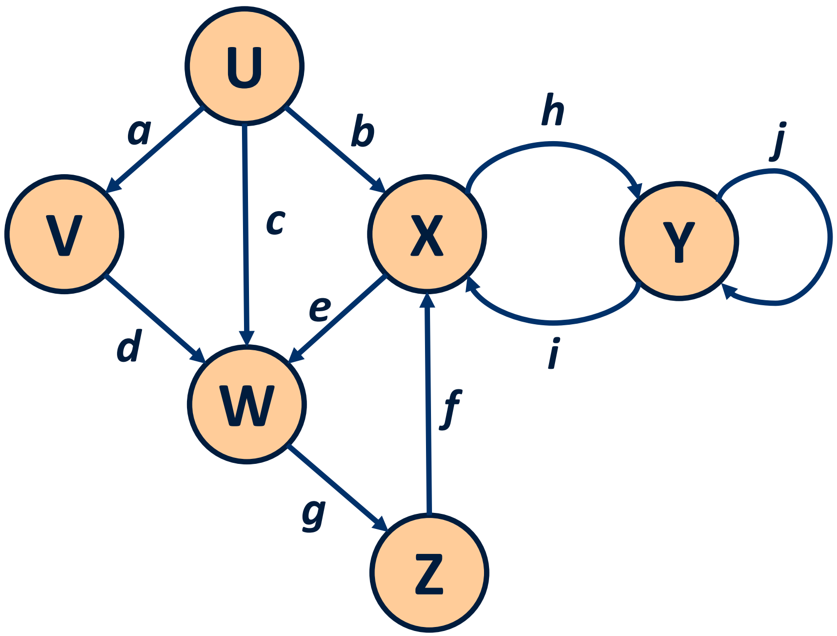
- Directed
  - Vertices are ordered pairs → origin/destination


- Undirected
  - Vertices are unordered

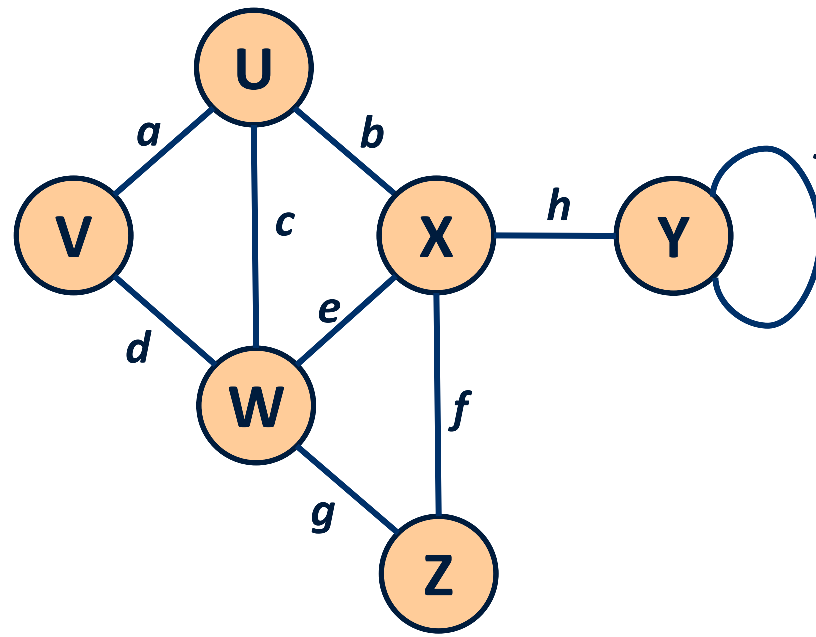


# Terminology - Directed Graph (Digraph)

- A **digraph** is a graph with no undirected edges



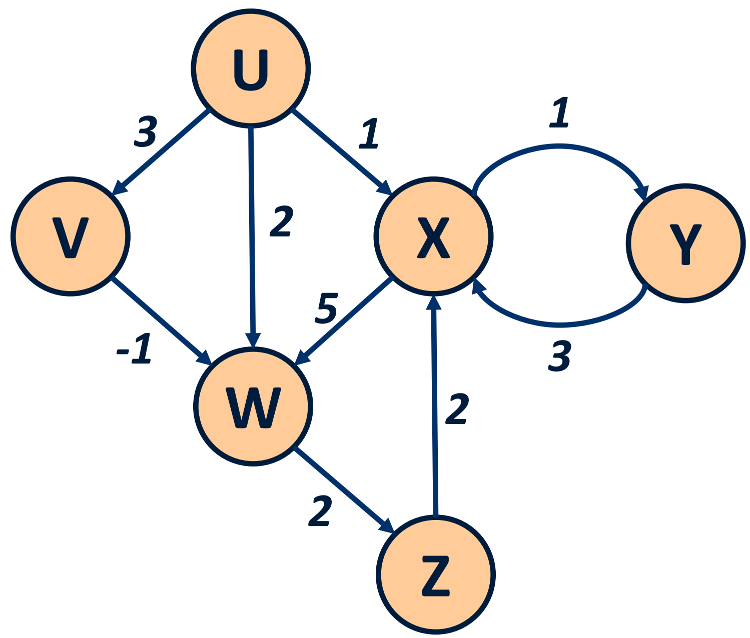

Directed Graph - Digraph



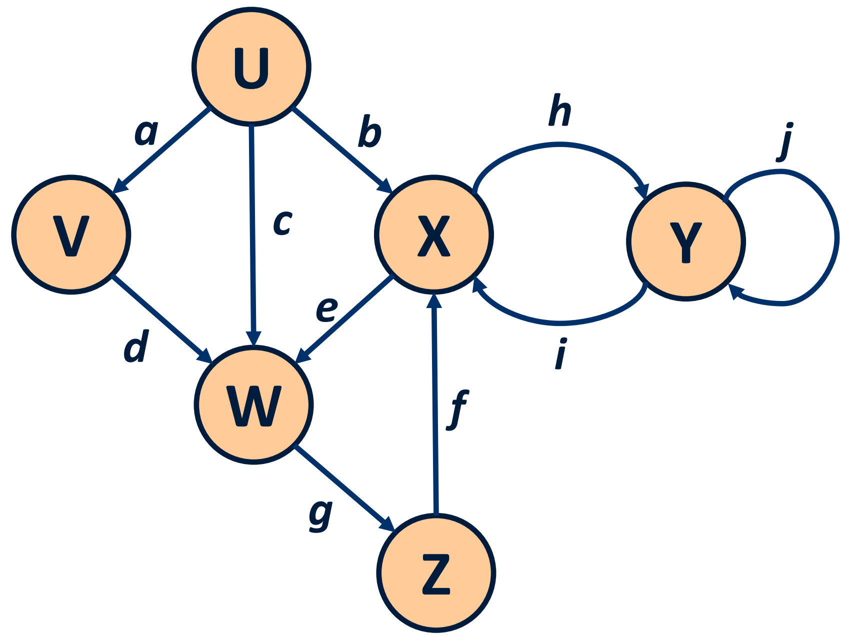

Undirected Graph

# Terminology - Weighted Graph

- A **weighted graph** is a graph where edges are labeled with weights
  - Weights can also be negative

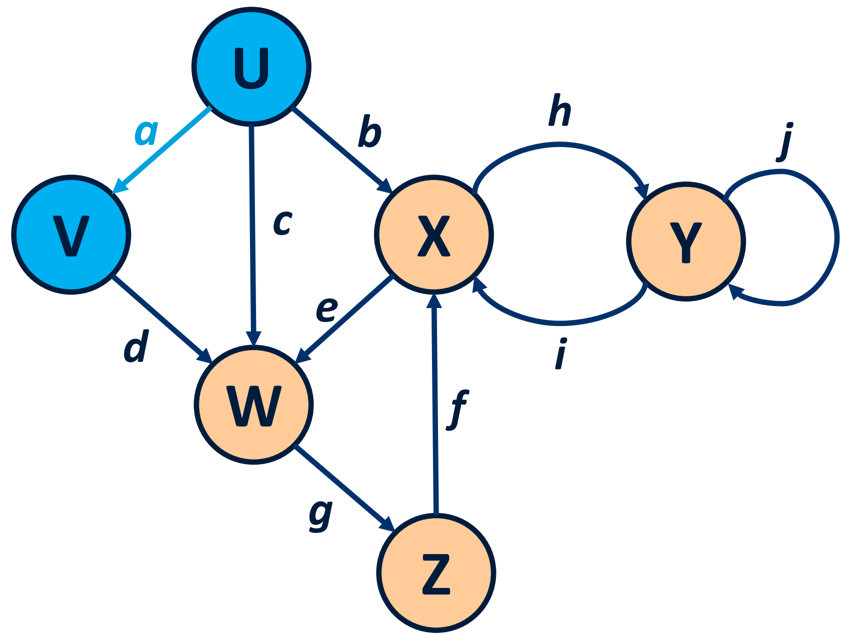


# Terminology – Vertices and Edges



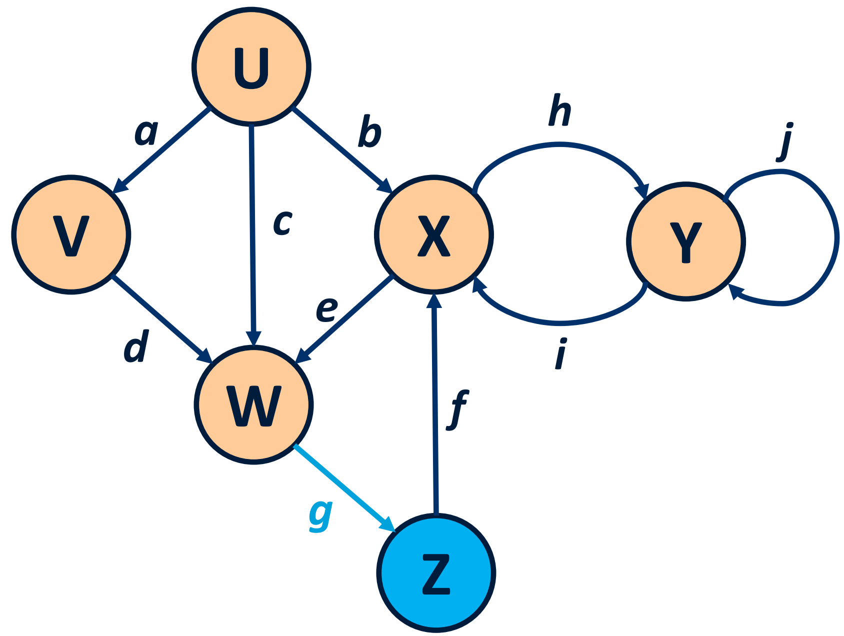
# Terminology – Vertices and Edges

- ***V*** is **endpoint** of ***a***
  - ***U*** is **start point** of ***a***



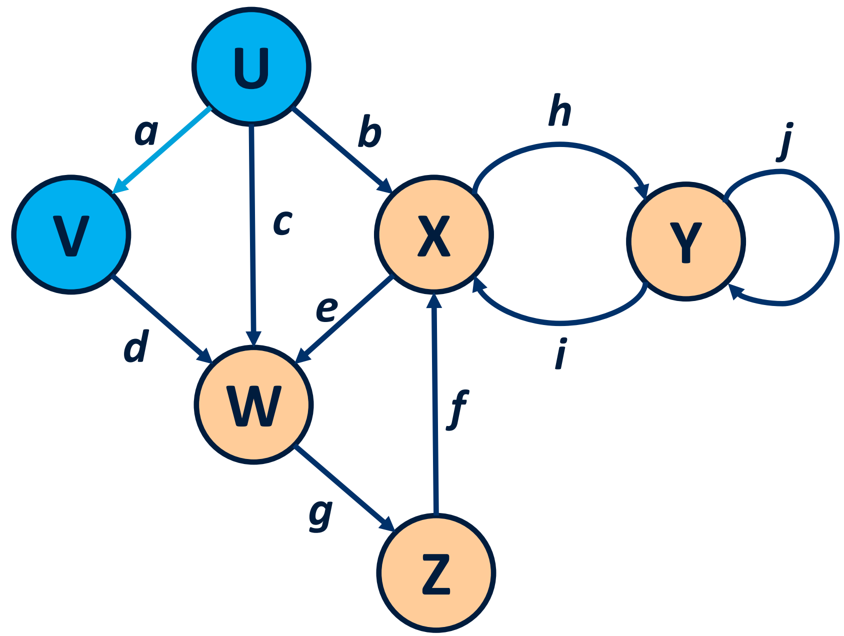
# Terminology – Vertices and Edges

- ***V*** is **endpoint** of ***a***
  - ***U*** is **start point** of ***a***
- ***g*** is **incident** on ***Z***



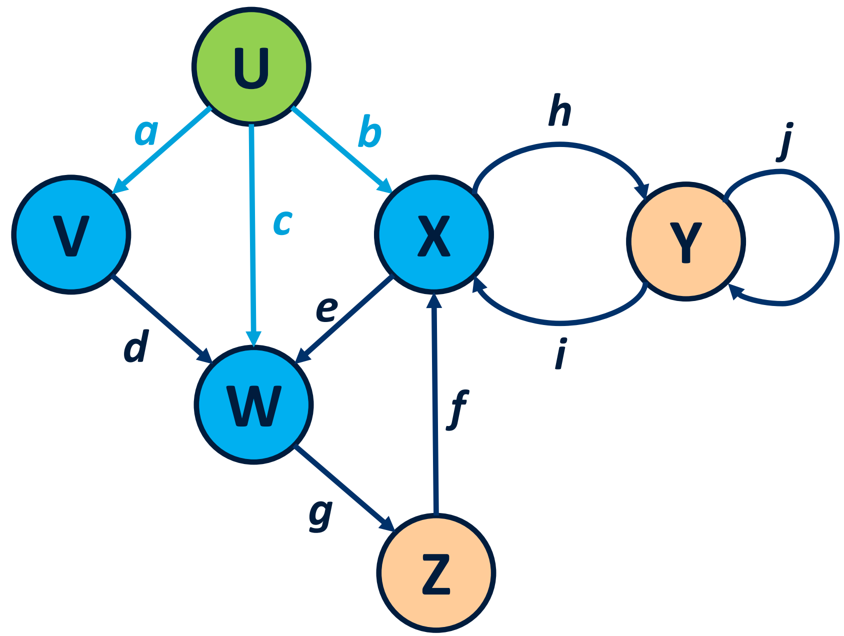
# Terminology – Vertices and Edges

- ***V*** is **endpoint** of ***a***
  - ***U*** is **start point** of ***a***
- ***g*** is **incident** on ***Z***
- ***V*** is **adjacent** to ***U***



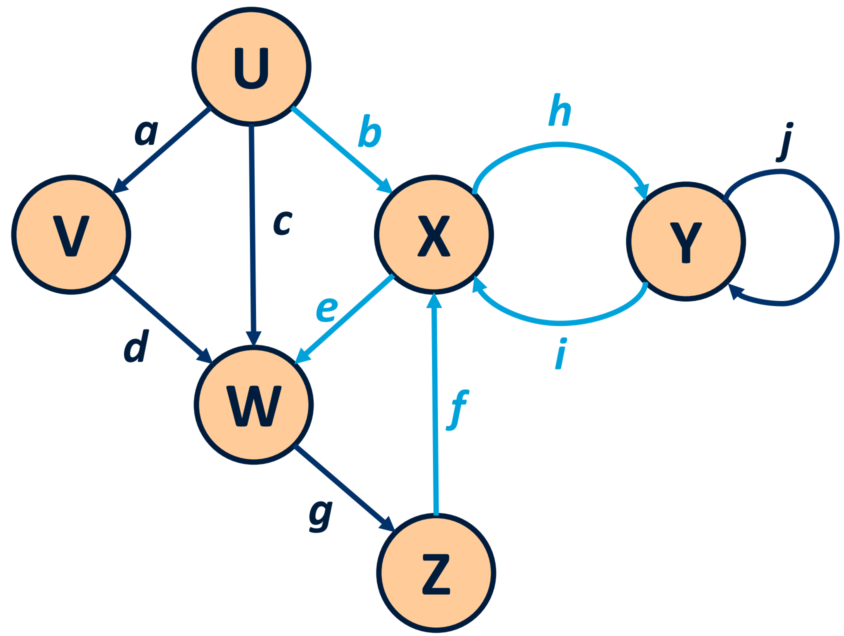
# Terminology – Vertices and Edges

- ***V*** is **endpoint** of ***a***
  - ***U*** is **start point** of ***a***
- ***g*** is **incident** on ***Z***
- ***V*** is **adjacent** to ***U***
- ***V***, ***W***, and ***X*** are **neighbors** of ***U***



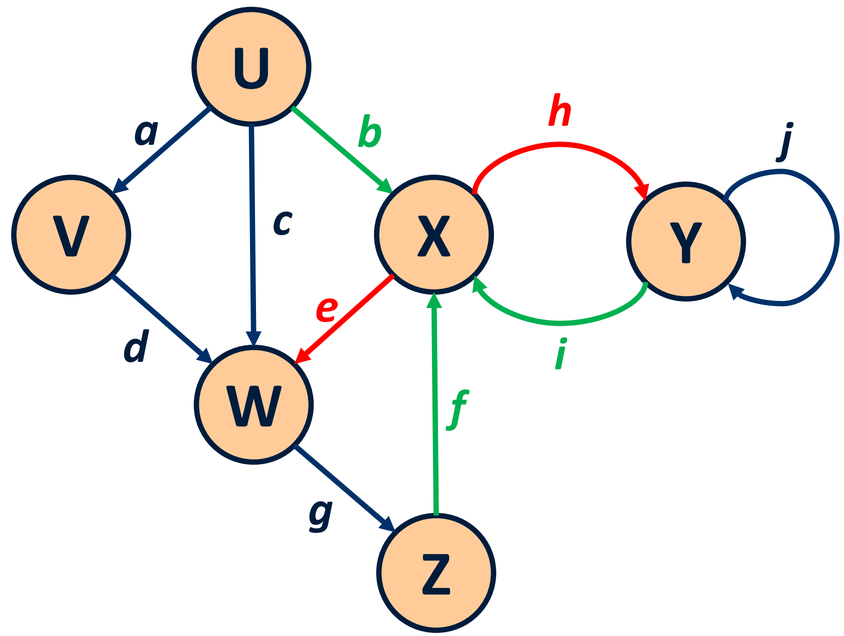
# Terminology – Vertices and Edges

- ***V*** is **endpoint** of ***a***
  - ***U*** is **start point** of ***a***
- ***g*** is **incident** on ***Z***
- ***V*** is **adjacent** to ***U***
- ***V***, ***W***, and ***X*** are **neighbors** of ***U***
- ***X*** has **degree** 5



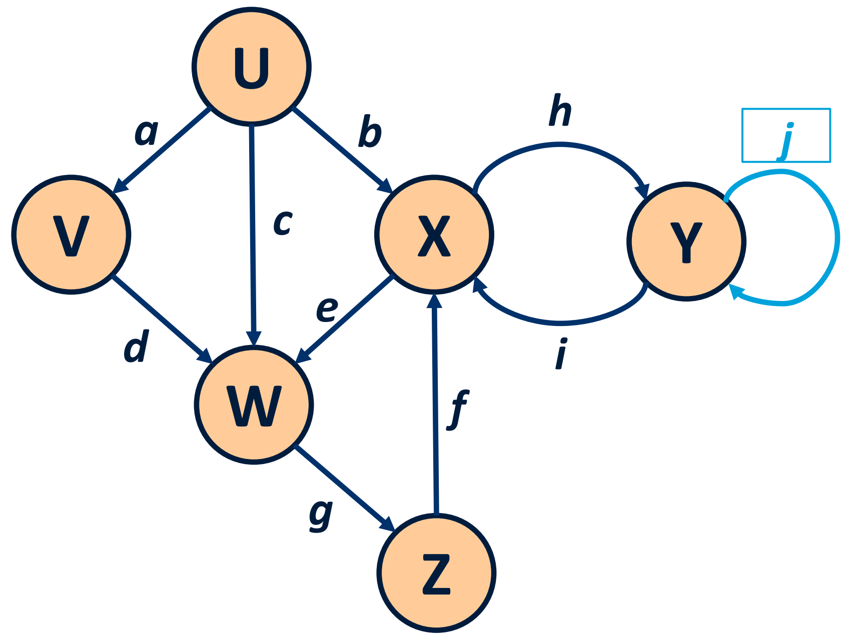
# Terminology – Vertices and Edges

- ***V*** is **endpoint** of ***a***
  - ***U*** is **start point** of ***a***
- ***g*** is **incident** on ***Z***
- ***V*** is **adjacent** to ***U***
- ***V***, ***W***, and ***X*** are **neighbors** of ***U***
- ***X*** has **degree** 5
  - **In-degree** is 3
  - **Out-degree** is 2



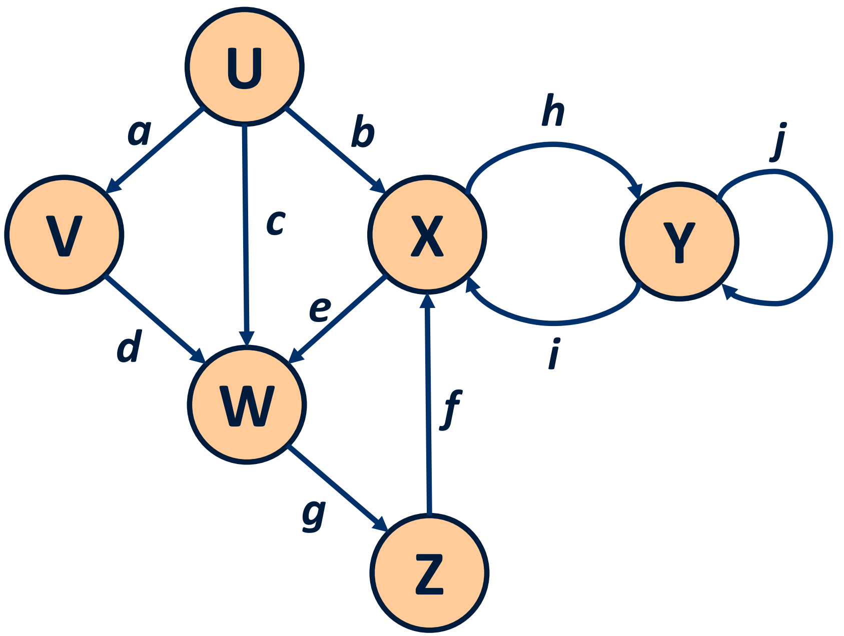
# Terminology – Vertices and Edges

- ***V*** is **endpoint** of ***a***
  - ***U*** is **start point** of ***a***
- ***g*** is **incident** on ***Z***
- ***V*** is **adjacent** to ***U***
- ***V***, ***W***, and ***X*** are **neighbors** of ***U***
- ***X*** has **degree** 5
  - **In-degree** is 3
  - **Out-degree** is 2
- ***j*** is a **self-loop**



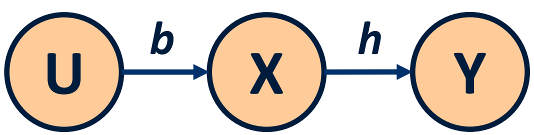
# Terminology - Paths

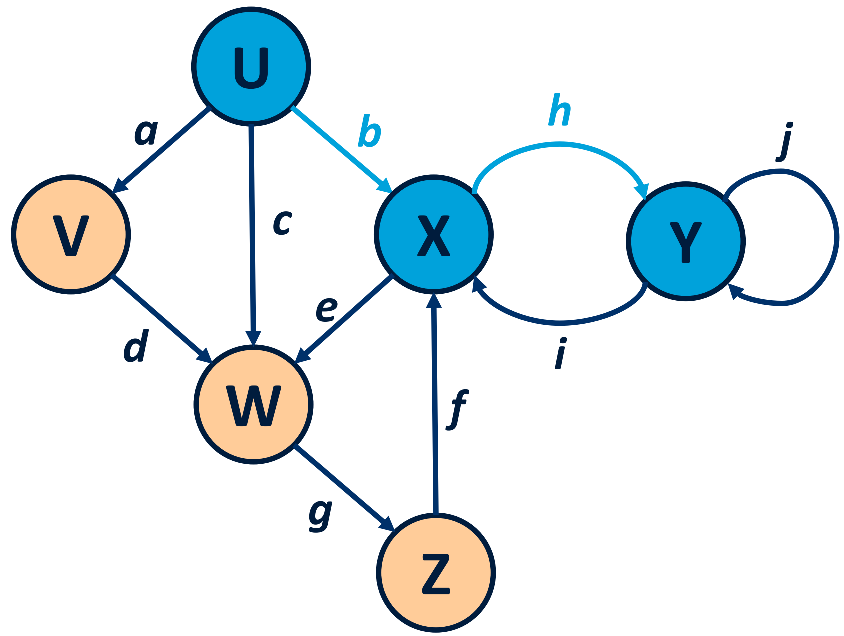
- A sequence of alternating vertices and edges is a **path**



# Terminology - Paths

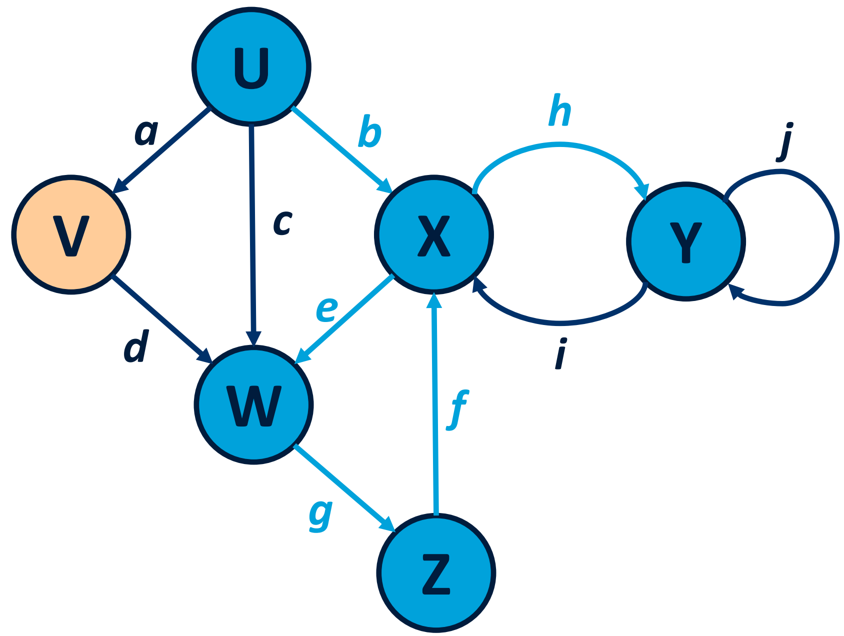
- A sequence of alternating vertices and edges is a **path**
- In a **simple path** all vertices and edges are distinct

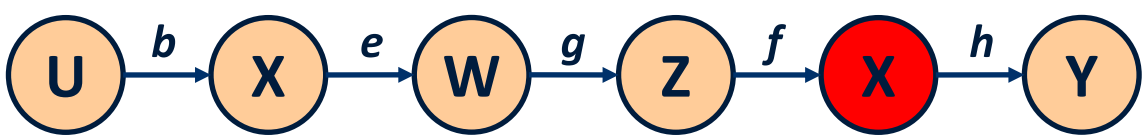


# Terminology - Paths

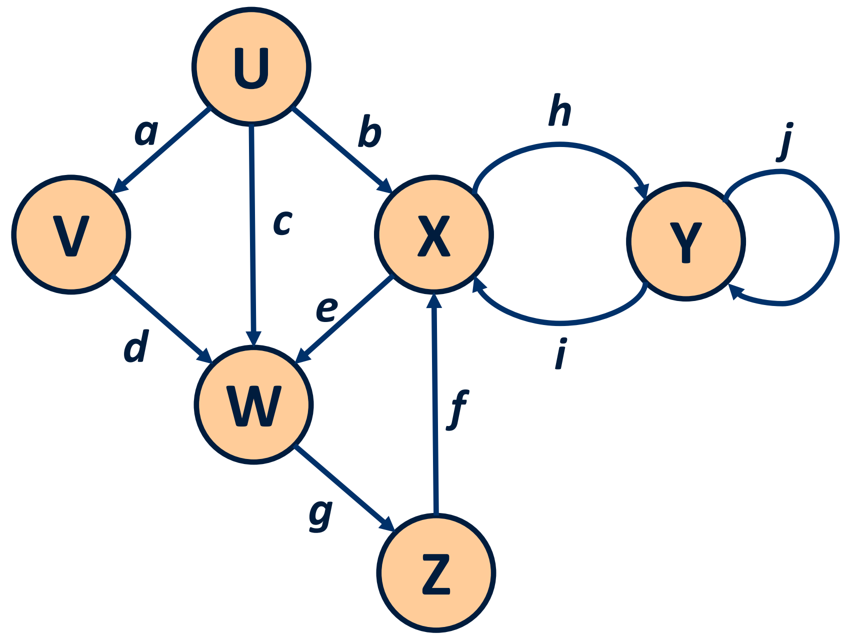
- A sequence of alternating vertices and edges is a **path**
  In a **simple path** all vertices and edges are distinct
- Otherwise not a simple path



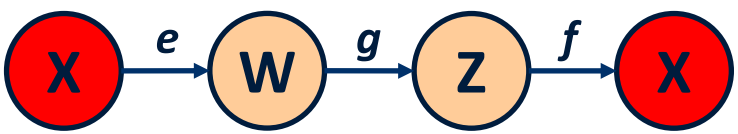
# Terminology - Cycles

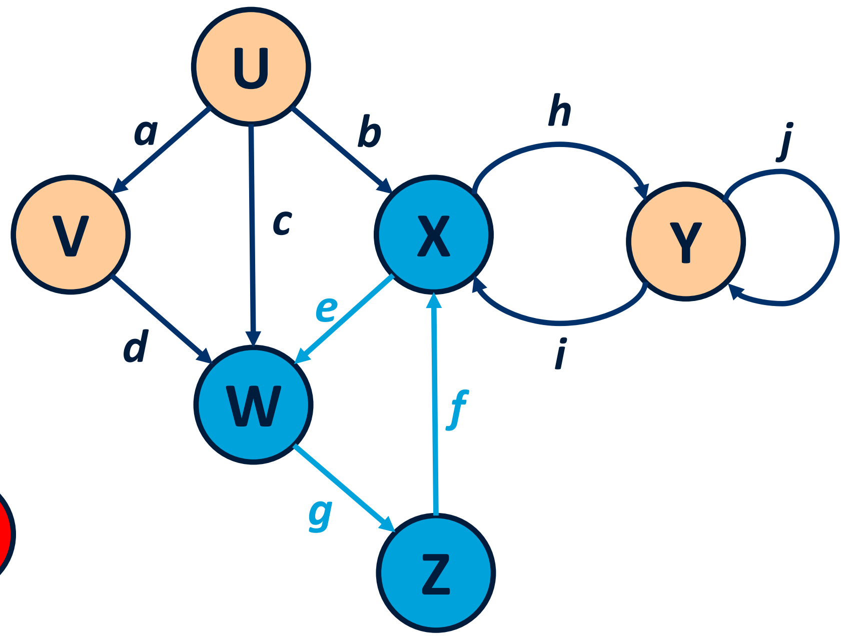
- A **cycle** is a path with the same start and end vertex



# Terminology - Cycles

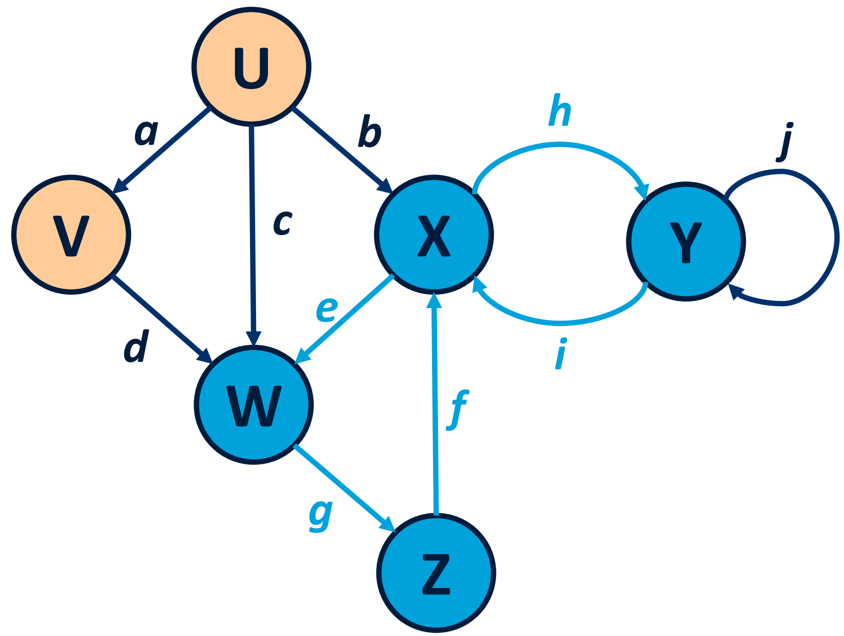
- A **cycle** is a path with the same start and end vertex
- A **simple cycle** is a closed walk with no repeated vertices

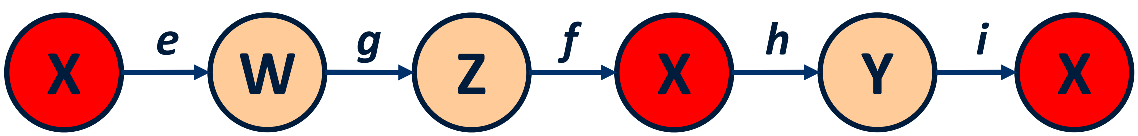


# Terminology - Cycles

- A **cycle** is a path with the same start and end vertex
- A **simple cycle** is a closed path with no repeated vertices
- Otherwise not a simple cycle



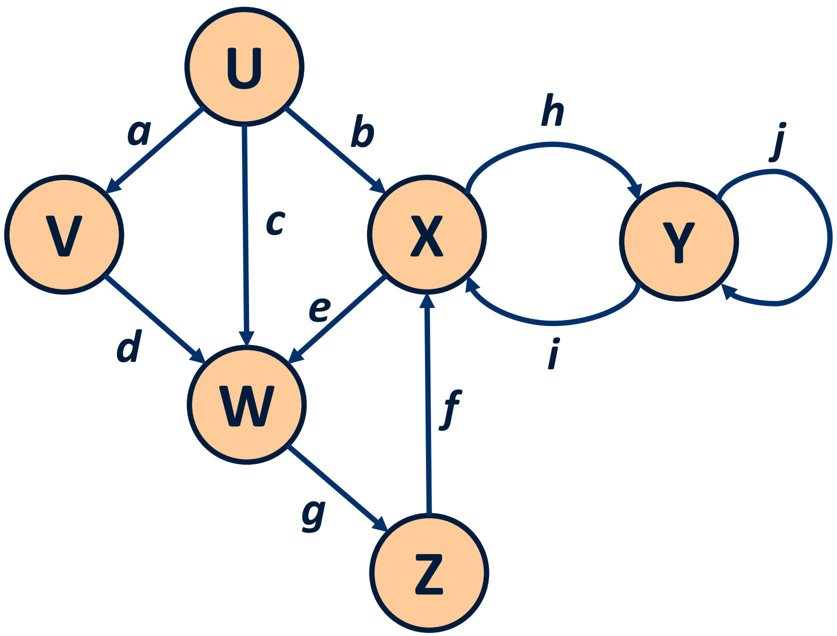



# Terminology - Subgraphs

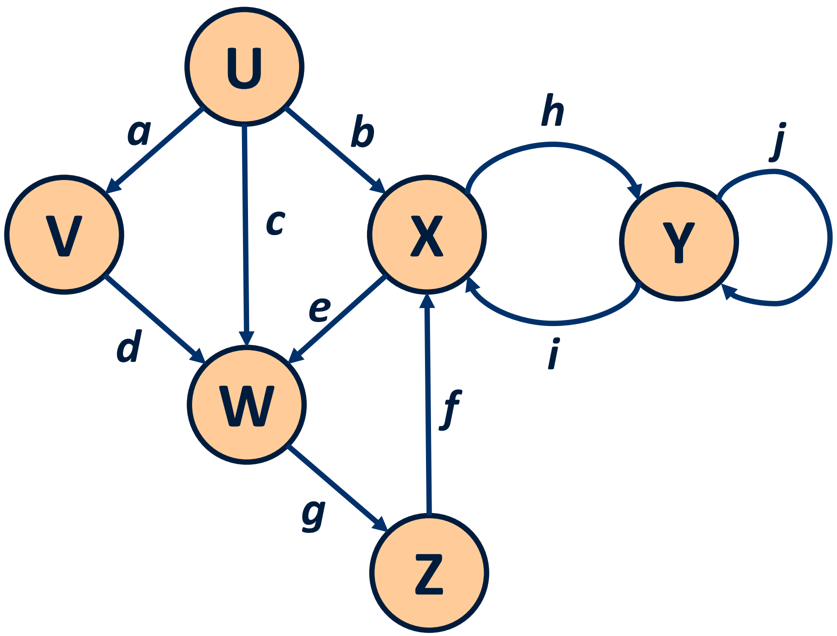
- A **subgraph** of a graph $\boldsymbol{G = (V, E)}$ is a graph $\boldsymbol{S = (V^s, E^s)}$ such that:
  - $\boldsymbol{V^s \subseteq V}$
    - all vertices of *S* are a subset of vertices in G
  - $\boldsymbol{E^s \subseteq E}$
    - all edges of *S* are a subset of *G*'s edges

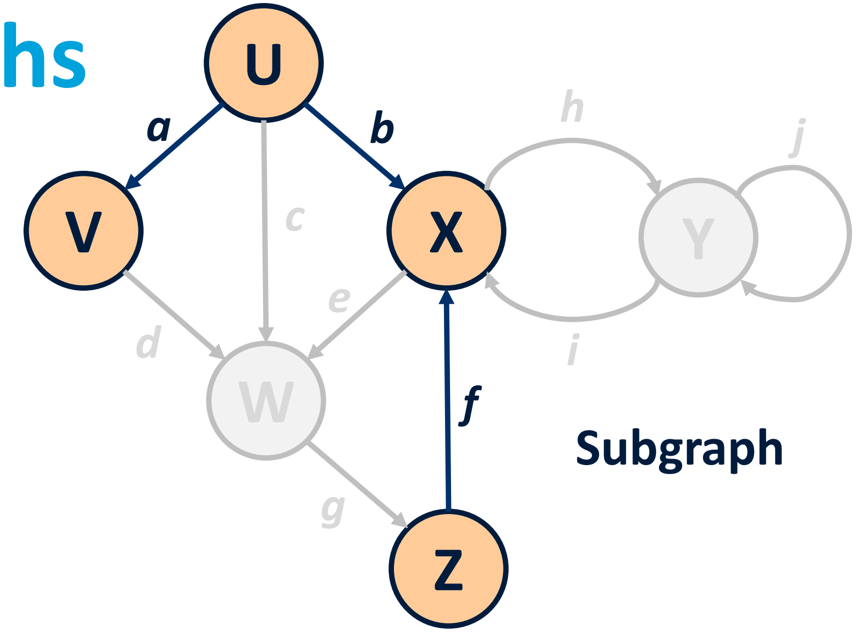
- A **spanning subgraph** of *G* is a subgraph that contains all the vertices of *G*

# Terminology - Subgraphs



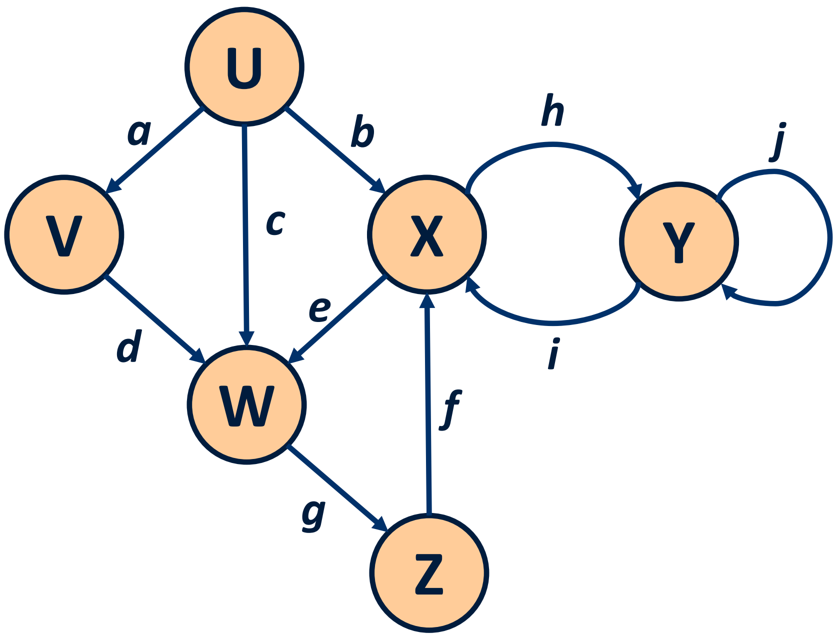
# Terminology - Subgraphs

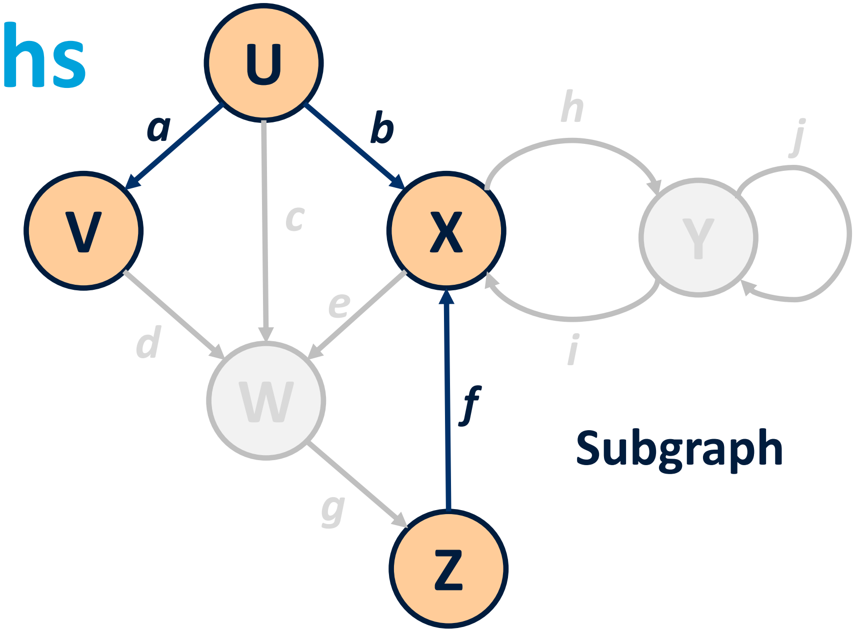


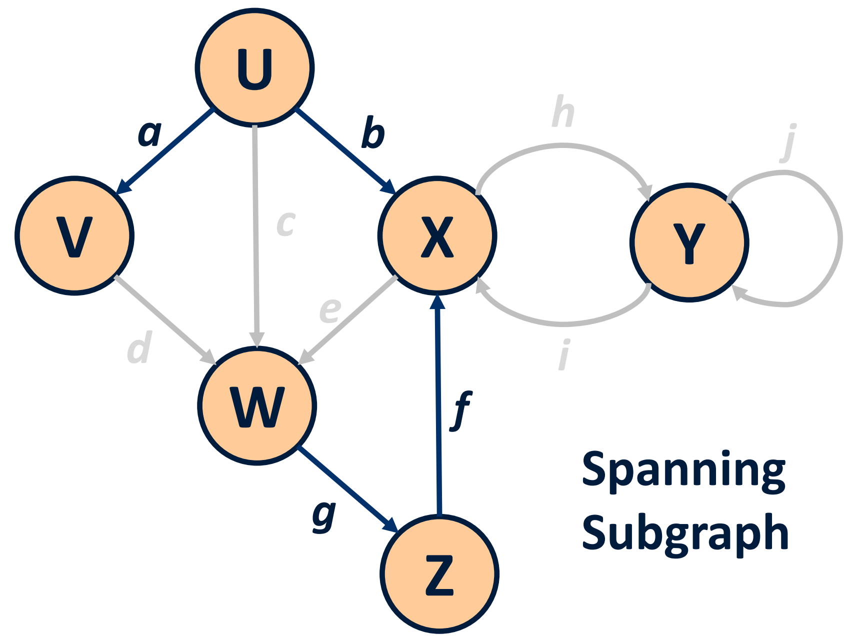

Subgraph

# Terminology - Subgraphs



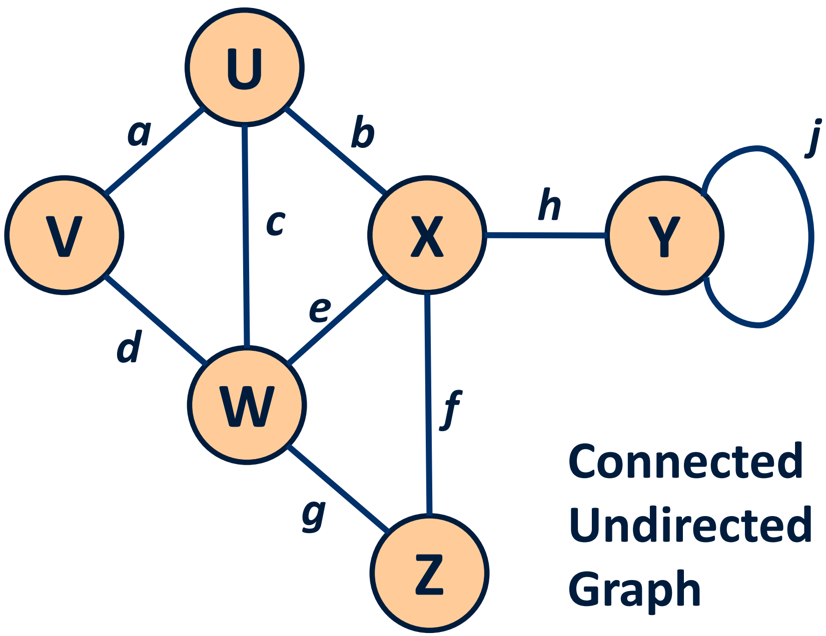

Subgraph



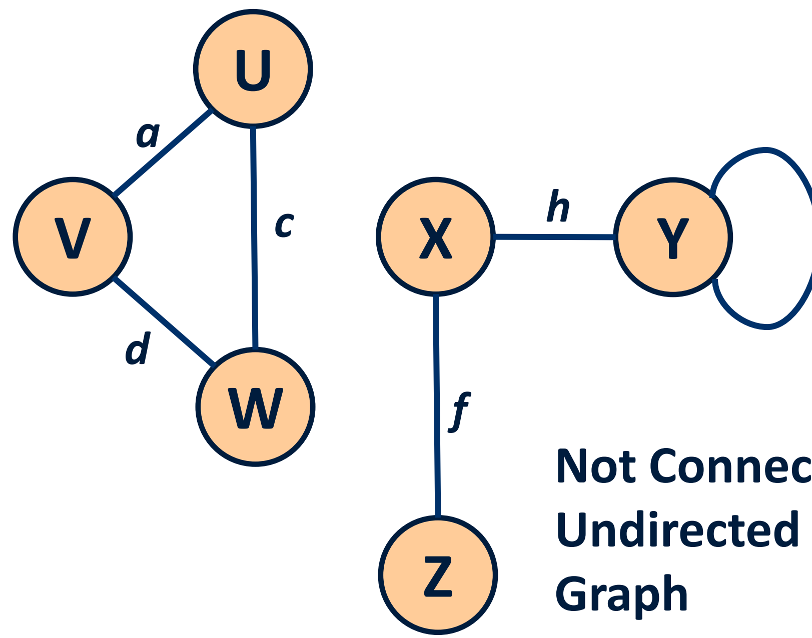

Spanning Subgraph

# Terminology - Connectivity

- An **Undirected** graph is **connected** if there’s a path between ***<u>every pair of vertices</u>***



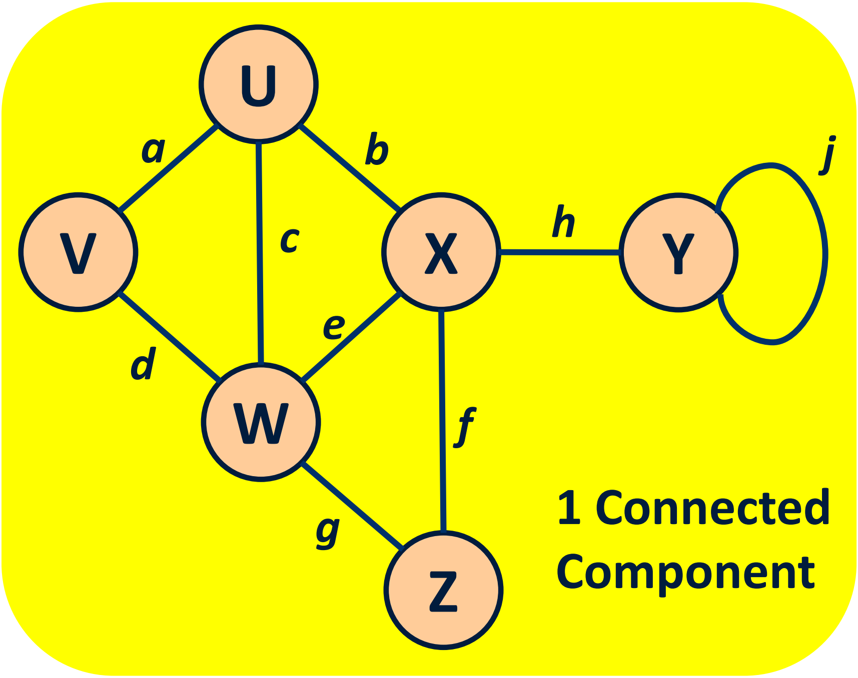

Connected Undirected Graph

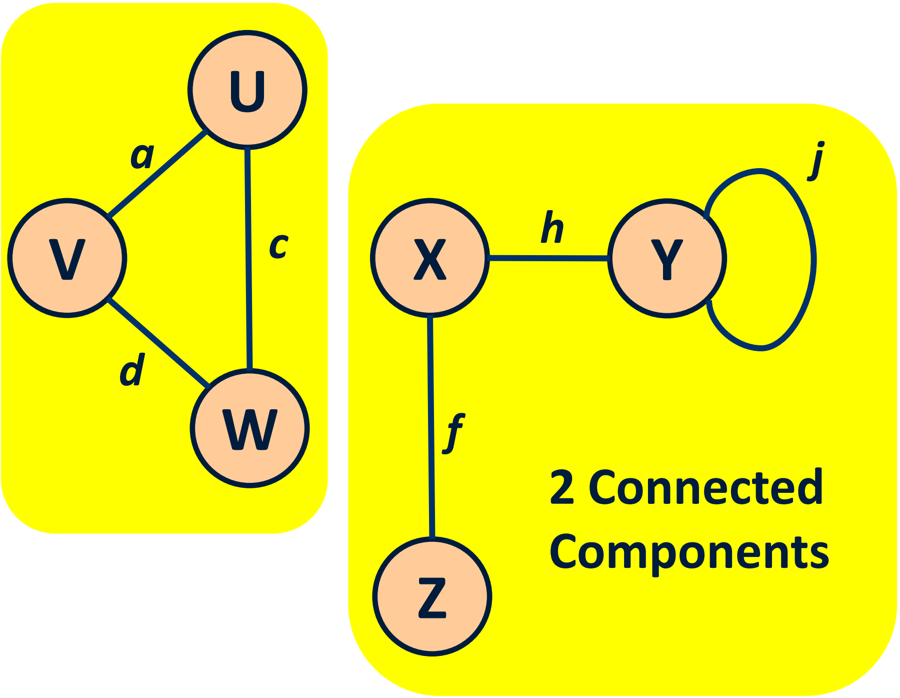



Not Connected Undirected Graph

# Terminology - Connectivity

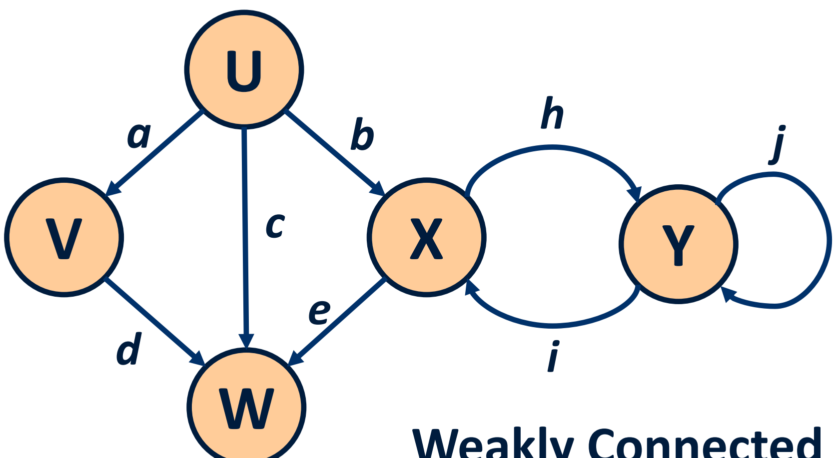
- In an **Undirected** graph, a **connected component** is a connected subgraph disconnected from the rest of the graph



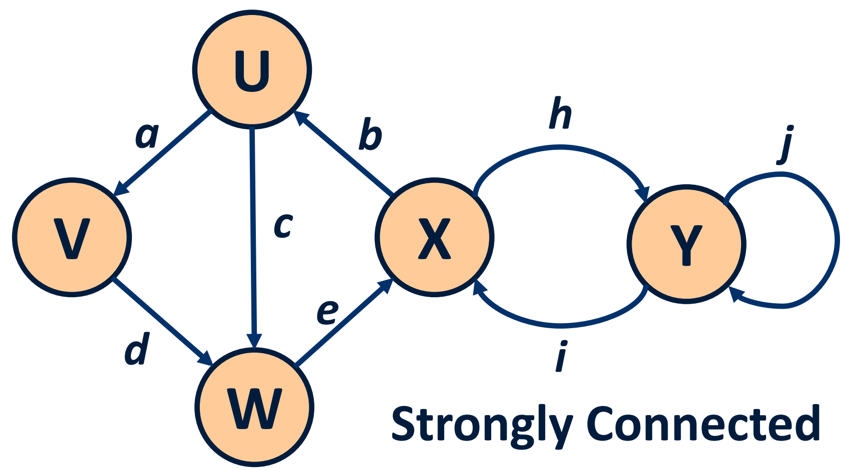



# Terminology - Connectivity

- A **Directed** graph is
  - **Weakly Connected**, if there's an **undirected** path between ***<u>every pair of vertices</u>***
  - **Strongly Connected**, if there's a **directed** path between ***<u>every pair of vertices</u>***



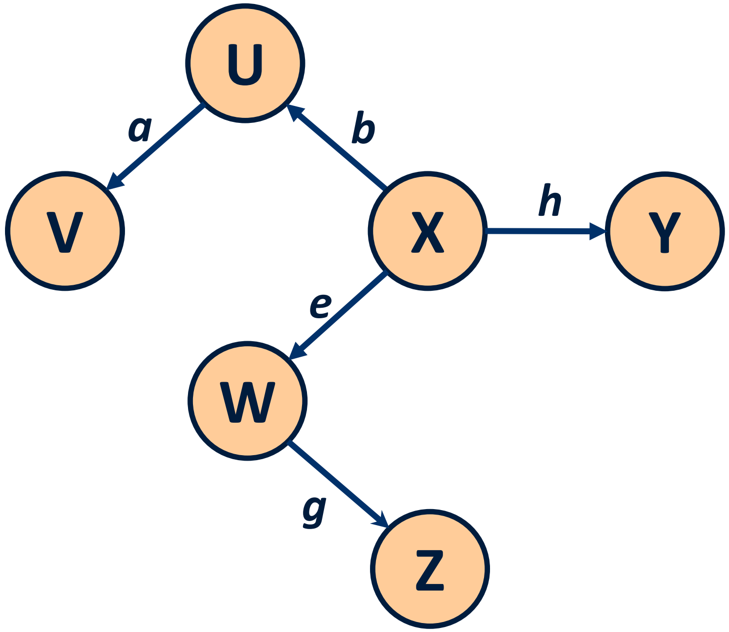

Weakly Connected




Strongly Connected

# Terminology – Trees and Forests

**A tree**

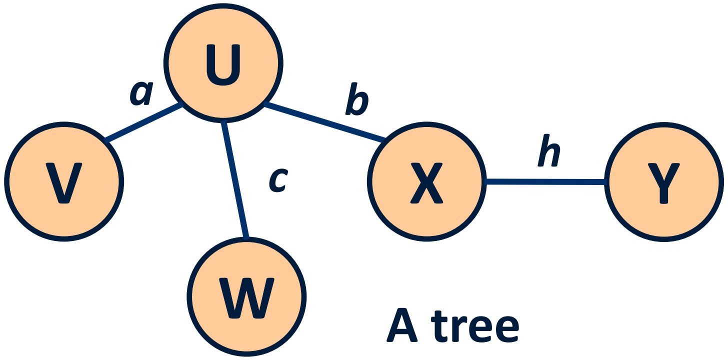



**Not a tree**

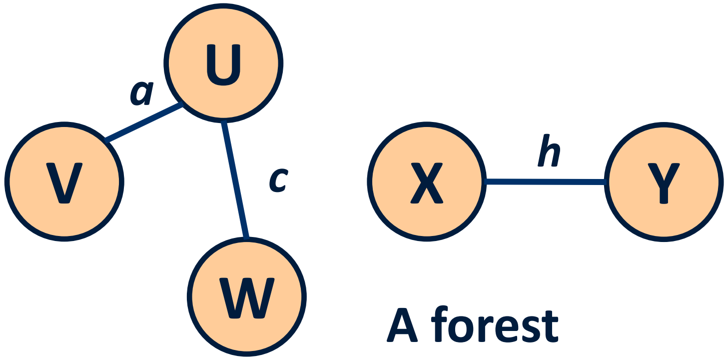
**What is a tree?**

# Terminology – Trees and Forests

- In an **Undirected** graph
  - A **tree** is a graph that is connected, and has no cycles
  - A **forest** is a not connected graph without cycles
  - All connected components of forests are trees



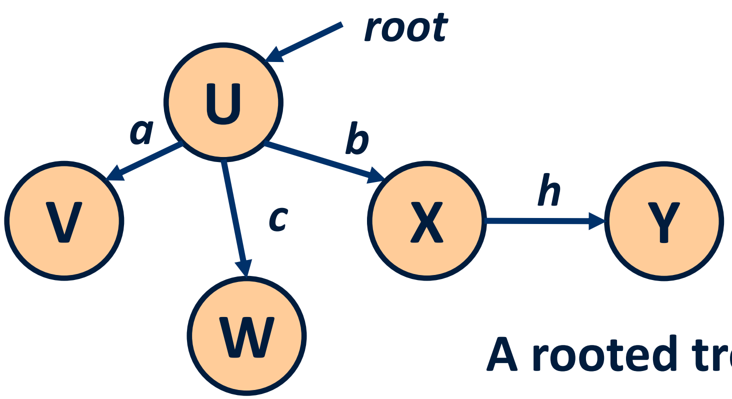
A tree



A forest

# Terminology – Trees and Forests

- In a **Directed** graph
  - A **rooted tree** (or tree) is a digraph that is weakly connected, and has no cycles and there is a **root** node, such that there is a single path from the root to any other node

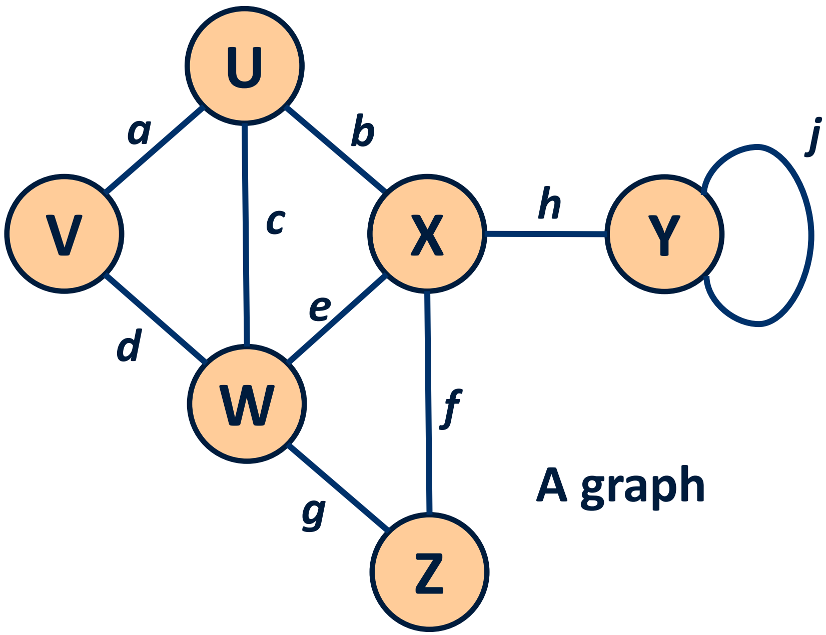



A rooted tree

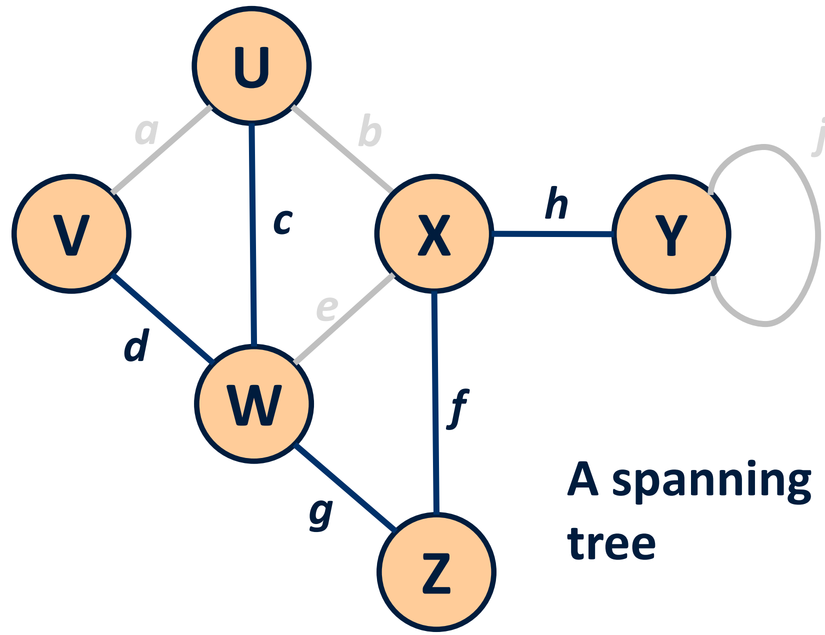
- Every node has 1 incoming edge, except for the root that has no incoming edges

# Terminology – Spanning Trees

- A **spanning tree** is a spanning subgraph that is a tree
  - Not unique unless the original graph is a tree



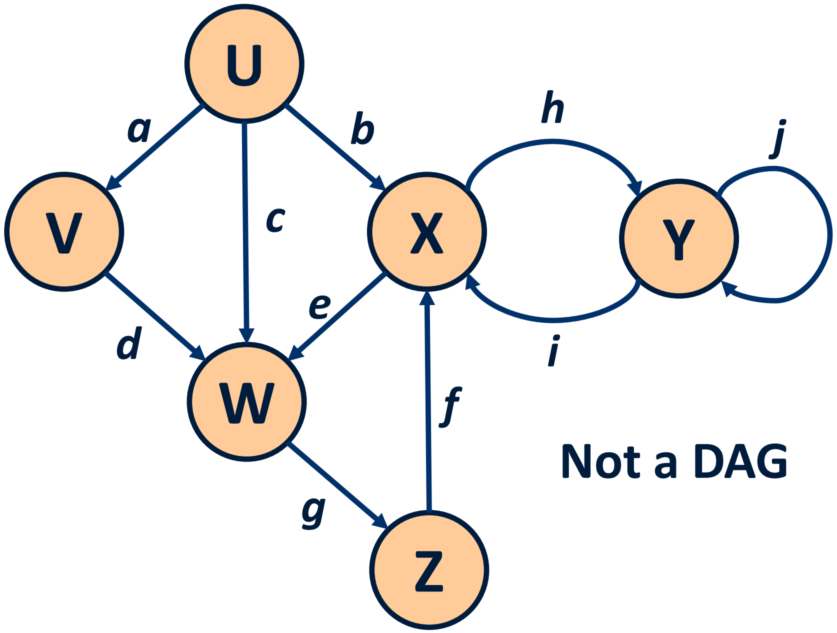

A graph

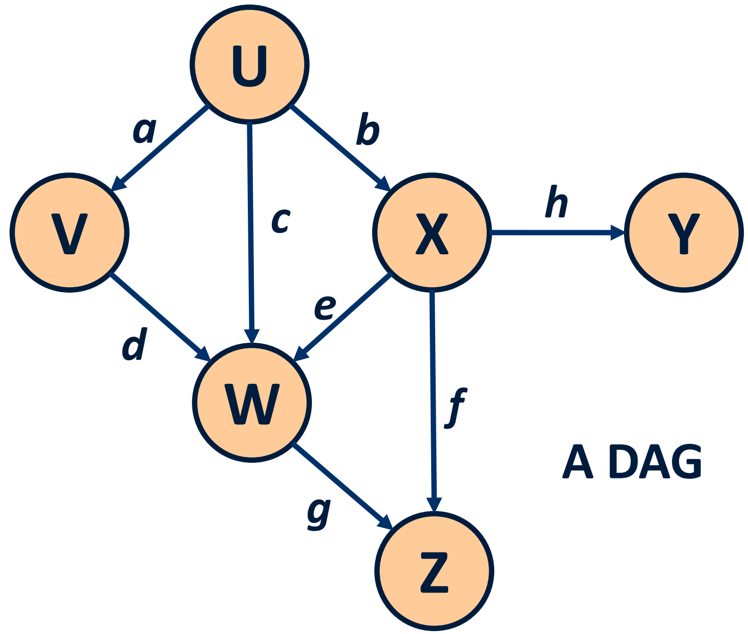



A spanning tree

# Terminology - Directed Graphs

- A **directed acyclic graph** (DAG) is a digraph without cycles



# The Graph ADT

| | |
|---|---|
| **adjacent(*G*, *x*, *y*)** | whether there's an edge from *x* to *y* |
| **neighbors(*G*, *x*)** | all vertices *y* s.t. there's an edge from *x* to *y* |
| **add_vertex(*G*, *x*)** | adds the vertex *x* |
| **remove_vertex(*G*, *x*)** | removes the vertex *x* |
| **add_edge(*G*, *x*, *y*)** | adds edge from the vertices *x* to *y* |
| **remove_edge(*G*, *x*, *y*)** | removes edge from the vertices *x* to *y* |
| **get_vertex_value(*G*, *x*)** | returns value associated with the vertex *x* |
| **set_vertex_value(*G*, *x*, *v*)** | sets value associated with the vertex *x* to *v* |
| **get_edge_value(*G*, *x*, *y*)** | returns value associated with edge (*x*, *y*) |
| **set_edge_value(*G*, *x*, *y*, *v*)** | sets value associated with edge (*x*, *y*) to *v* |

# Graph implementations

# Graph Implementations

- What to store?
  - Vertices/Edges
  - Vertex and/or edge values
- Application of the data structure
  - Find paths
  - Frequent visits to all neighbors
  - Following edges based on weights
  - Frequent mutations of the graph

# Graph Implementations

- Three common implementations:
  - Edge list (array-based or linked-based)
  - Adjacency list (array+linked-based)
  - Adjacency matrix (array-based)

- Each has specific benefits and drawbacks

- Complexity of operations differ

# Edge List

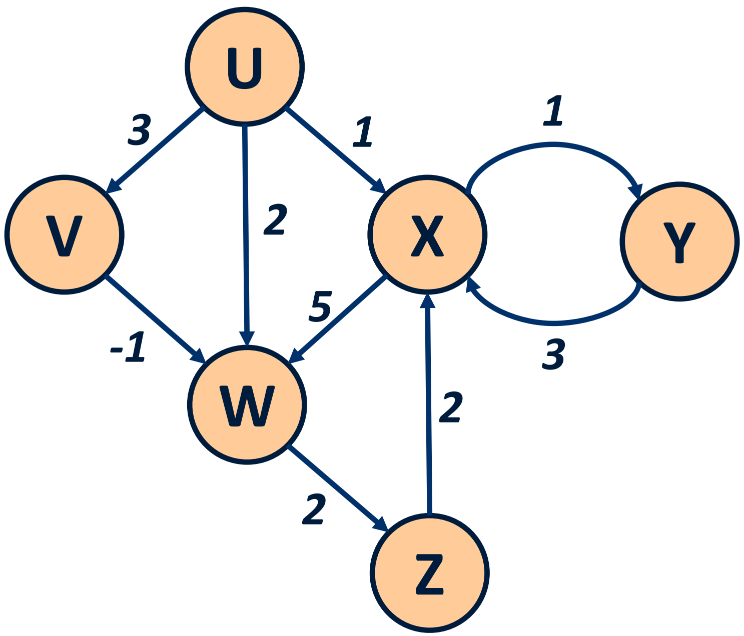
- An unordered list of all edges in the graph
  - Vertices stored implicitly
- Easy to:
  - Iterate over all edges
  - Find endpoints of edges
- Hard to determine:
  - If edge exists between vertices
  - The degrees of vertices
- Problem:
  - Unconnected Vertices

# Edge List

- Edge contains fields for:
  - Connected vertices
  - Weight data (if weighted graph)
  - Boolean value directed/undirected

- Vertex key/values need to be stored in separate data structure

# Edge List - array-based

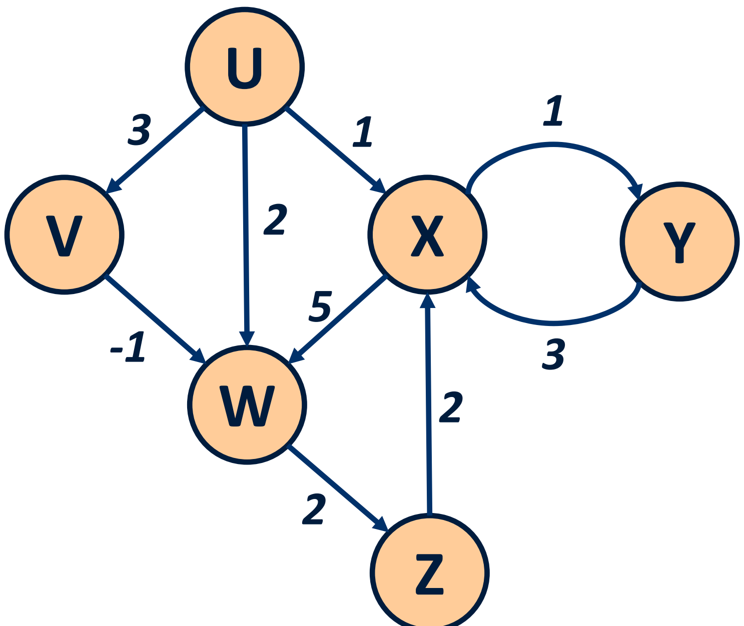



**Edge Object**

| |
|---|
| Vertex 1 |
| Vertex 2 |
| weight |
| Directed (T/F) |

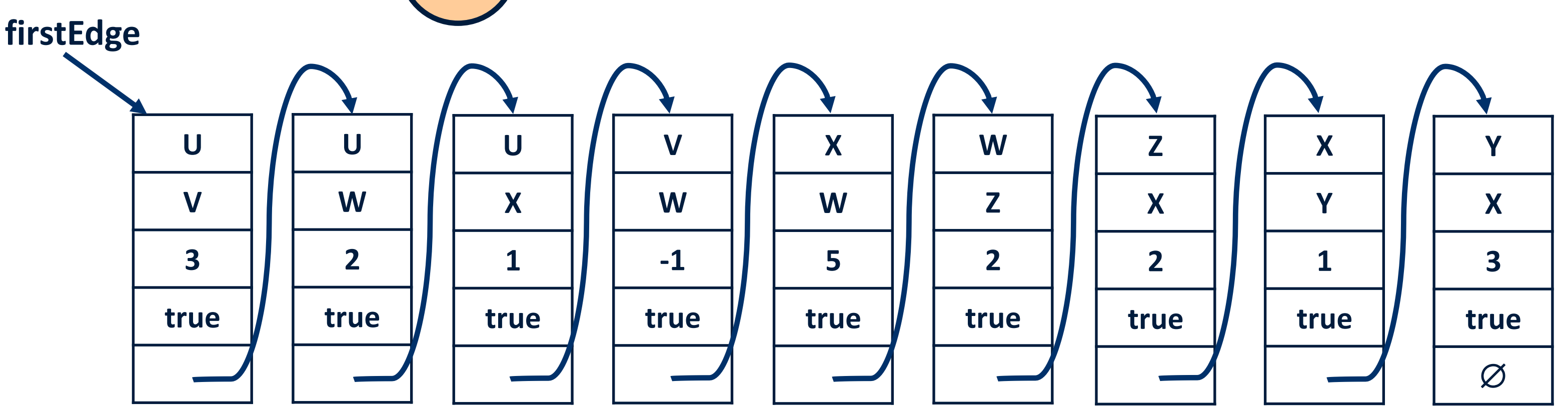
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| U | U | U | V | X | W | Z | X | Y |
| V | W | X | W | W | Z | X | Y | X |
| 3 | 2 | 1 | -1 | 5 | 2 | 2 | 1 | 3 |
| true | true | true | true | true | true | true | true | true |

# Edge List - linked-based




## Edge Object

| |
|---|
| Vertex 1 |
| Vertex 2 |
| weight |
| Directed (T/F) |
| next |



# Adjacency Lists

- For each vertex stores
  - edges as individual linked lists of references to each vertex's neighbors
- Easiest to implement if no information about edges is required.
  - Or needs an auxiliary edge data structure
- Easy to:
  - Add new vertices
  - Find incident edges on a vertex
- Hard to determine:
  - Whether an edge exists between two vertices

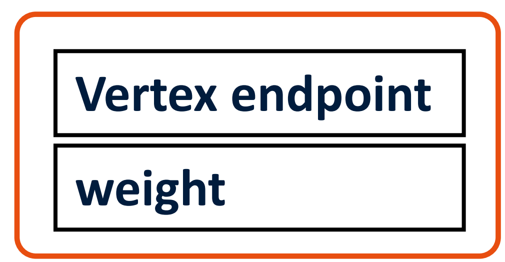
# Adjacency Lists

- Vertex objects have the following fields:
  - Element data
  - List of adjacent verteces

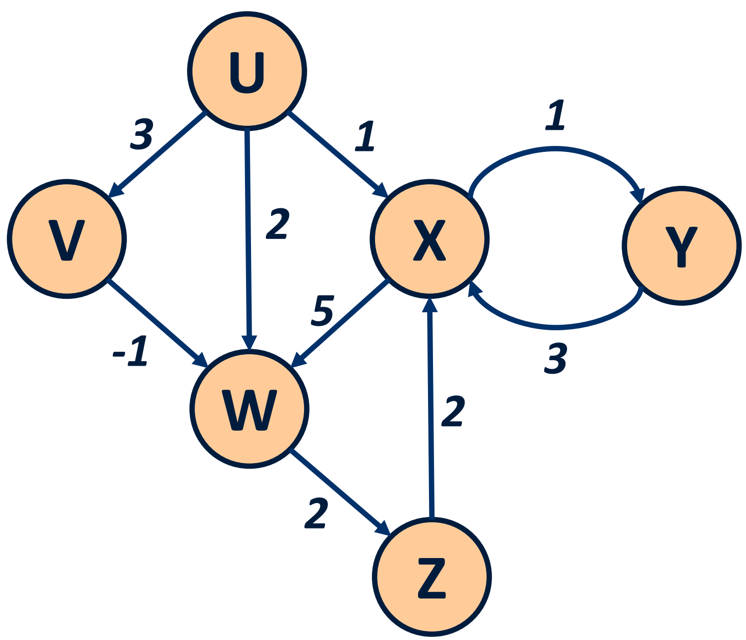
Vertex Object


- Edges objects keep
  - Vertex “endpoint” of the edge
  - weight

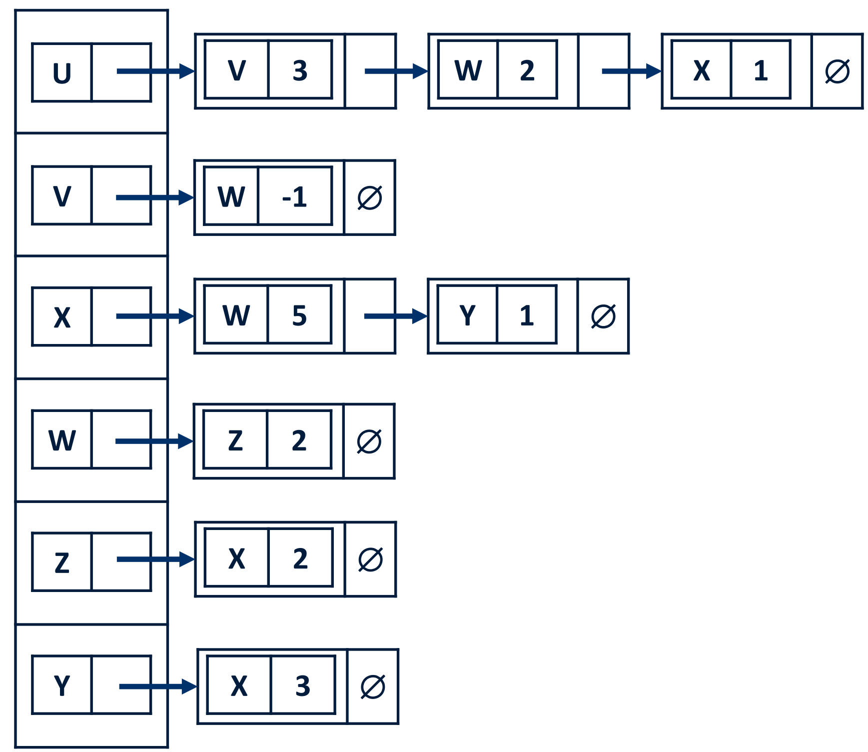
Edge Object



# Adjacency Lists

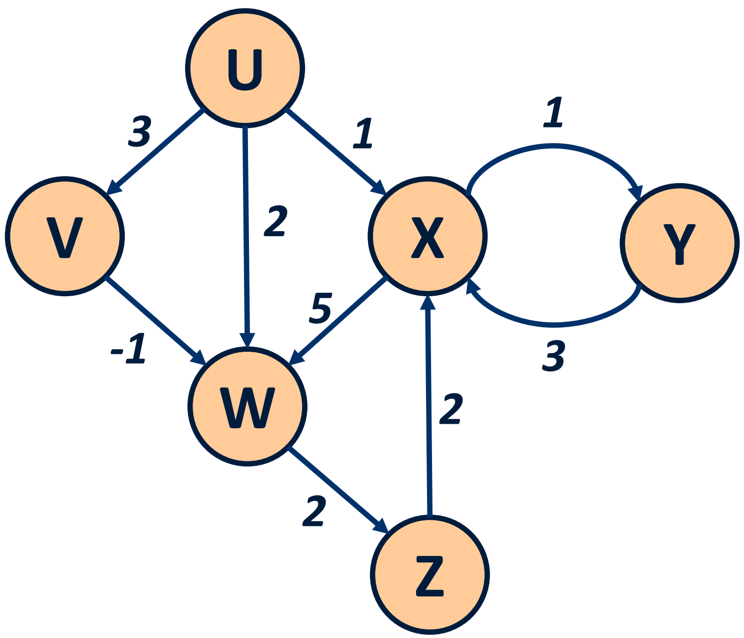



**List<Edge>[]**



# Adjacency Matrix

- Matrix: $a_{ij}$ weight of the edge $i$ -> $j$
  - We can store the entries element->index in a Map
  - The matrix can also store Edge objects




| | U | V | W | X | Y | Z |
|---|---|---|---|---|---|---|
| **U** | ∅ | 3 | 2 | 1 | ∅ | ∅ |
| **V** | ∅ | ∅ | -1 | ∅ | ∅ | ∅ |
| **W** | ∅ | ∅ | ∅ | ∅ | ∅ | 2 |
| **X** | ∅ | ∅ | 5 | ∅ | 1 | ∅ |
| **Y** | ∅ | ∅ | ∅ | 3 | ∅ | ∅ |
| **Z** | ∅ | ∅ | ∅ | 2 | ∅ | ∅ |

# Graph Operations – running time

| *G = (V, E), \|V\|=n, \|E\|=m*<br>***n*** vertices, ***m*** edges | **Edge List** | **Adjacency List** | **Adjacency Matrix** |
|---|---|---|---|
| **Space** | $n + m$ | $n + m$ | $n^2$ |
| Finding incident vertices to ***v*** | ***m*** | deg(***v***) | ***n*** |
| Determining if ***v*** is adjacent to ***w*** | ***m*** | min(deg(***v***), deg(***w***)) | 1 |
| inserting a vertex | 1 | 1 | $n^2$ |
| inserting an edge | 1 | 1 | 1 |
| removing vertex ***v*** | ***m*** | deg(***v***) | $n^2$ |
| removing an edge | ***m*** | deg(***v***) | 1 |

# Graph Traversal

# Graph Traversal

- Traversal of a graph
  - Depth-first search
  - Breadth-first search

# Depth-First search

- **Depth-first search (DFS)**: finds a path between two vertices by exploring each possible path as many steps as possible before backtracking

***dfs***(*Vertex v):*

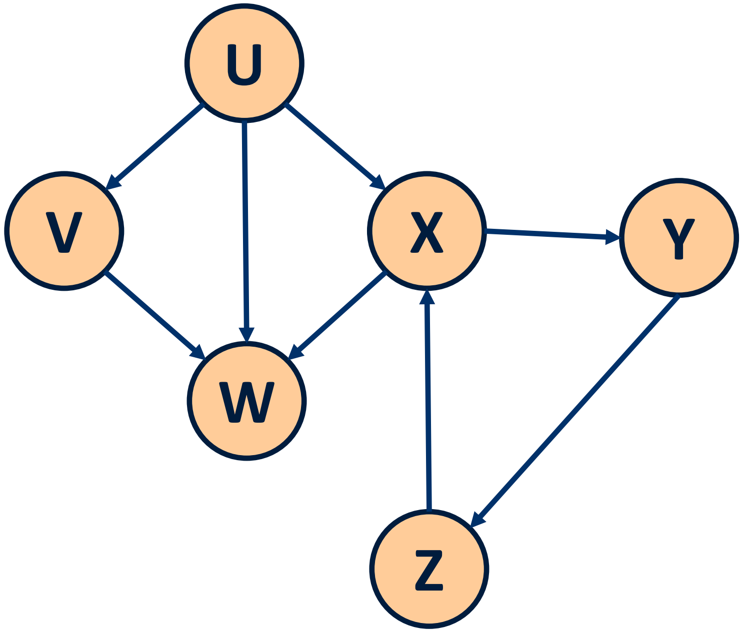
*mark* ***v*** *as visited*

***for each*** *unvisited neighbor* $\boldsymbol{v_i}$ *of* ***v***

***dfs***($v_i$)

# DFS Walk-Through

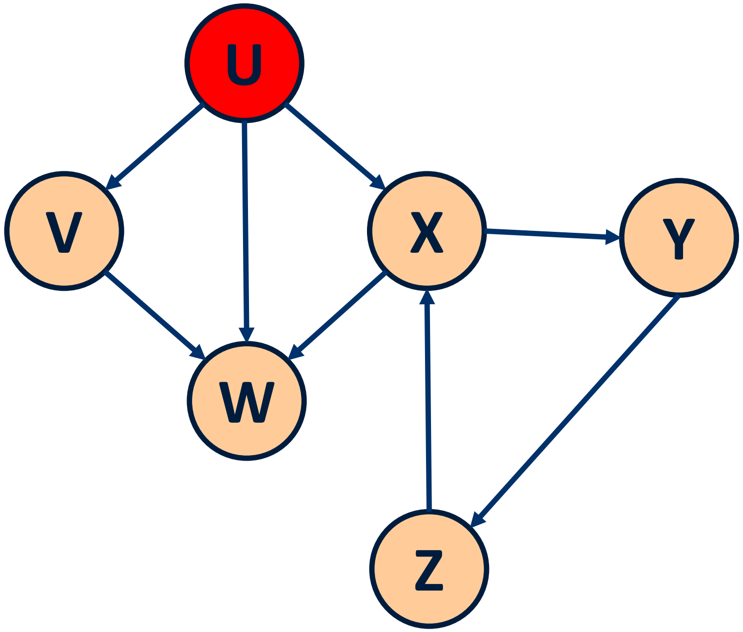
- **DFS** from node **U**




**Visited**

# DFS Walk-Through

- **DFS** from node **U**

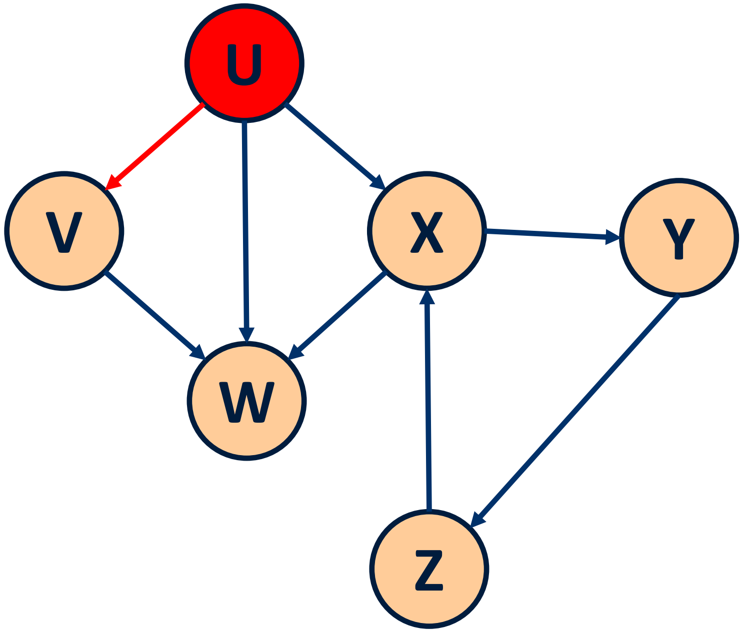



**Visited**

**U**

# DFS Walk-Through

- **DFS** from node **U**

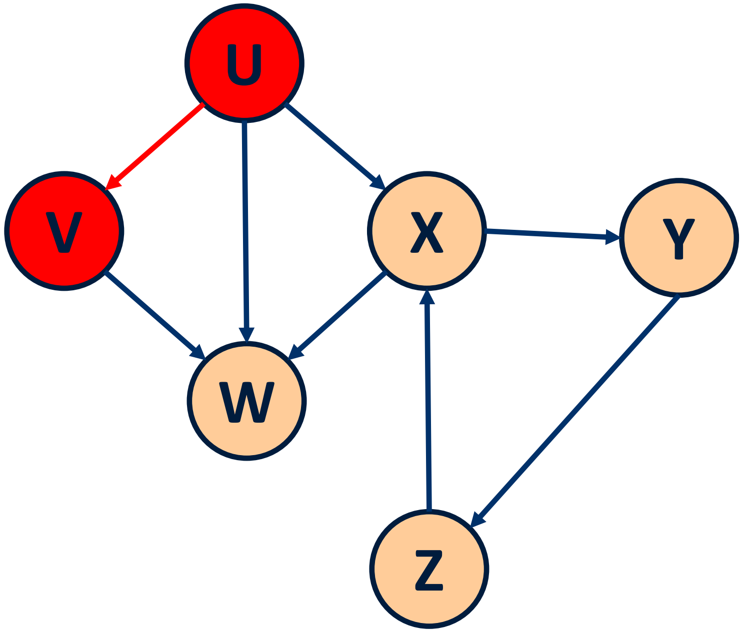



**Visited**

**U**

# DFS Walk-Through

- **DFS** from node **U**

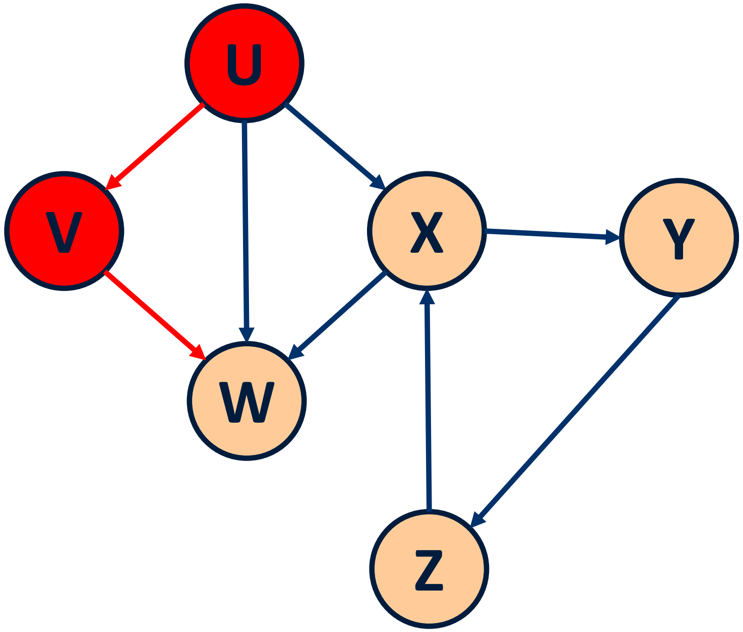



**Visited**

**U - V**

# DFS Walk-Through

- **DFS** from node **U**

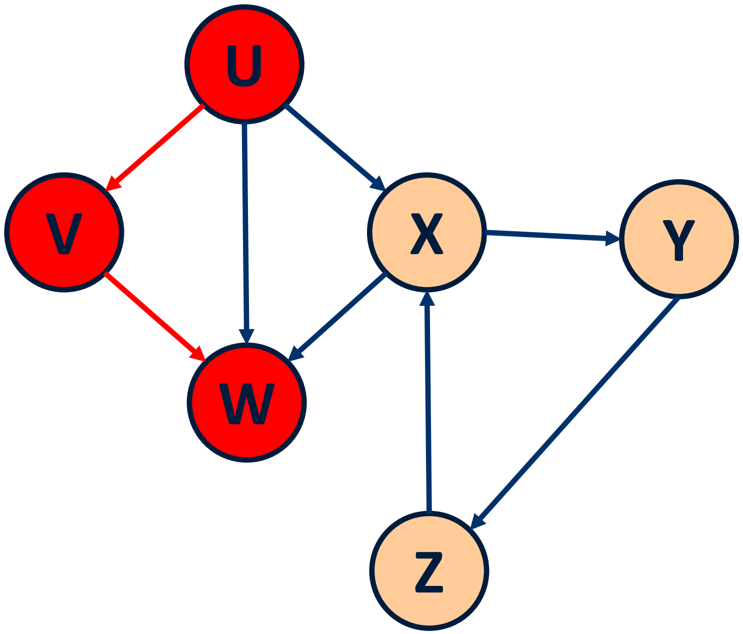



**Visited**

**U - V**

# DFS Walk-Through

- **DFS** from node **U**

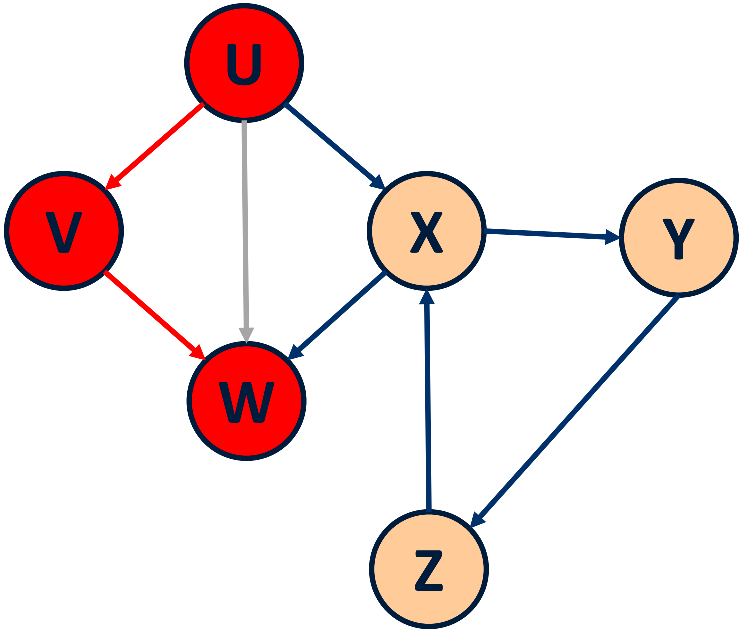



**Visited**

**U - V - W**

# DFS Walk-Through

- **DFS** from node **U**

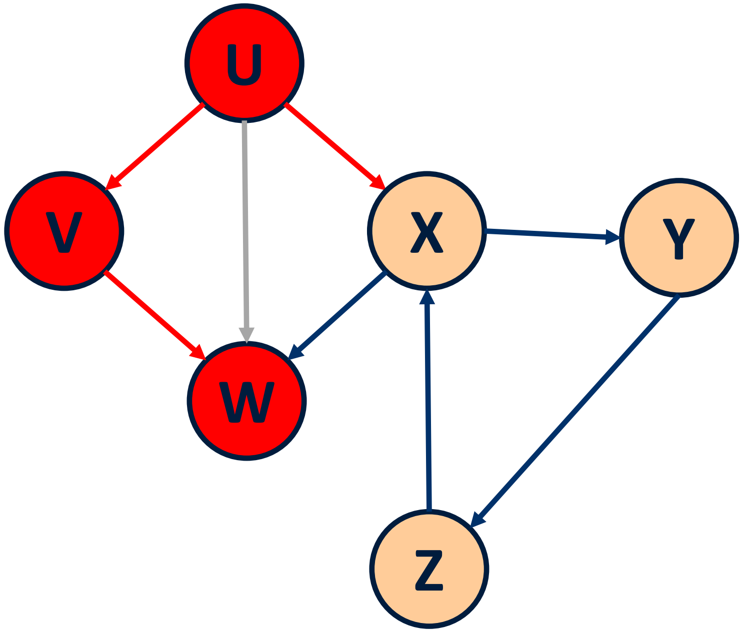



**Visited**

**U - V - W**

# DFS Walk-Through

- **DFS** from node **U**

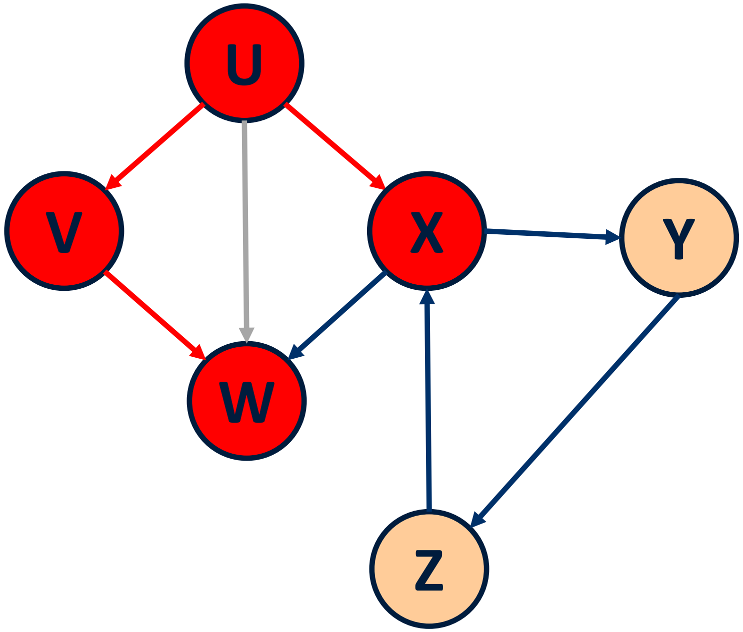



**Visited**

**U - V - W**

# DFS Walk-Through

- **DFS** from node **U**

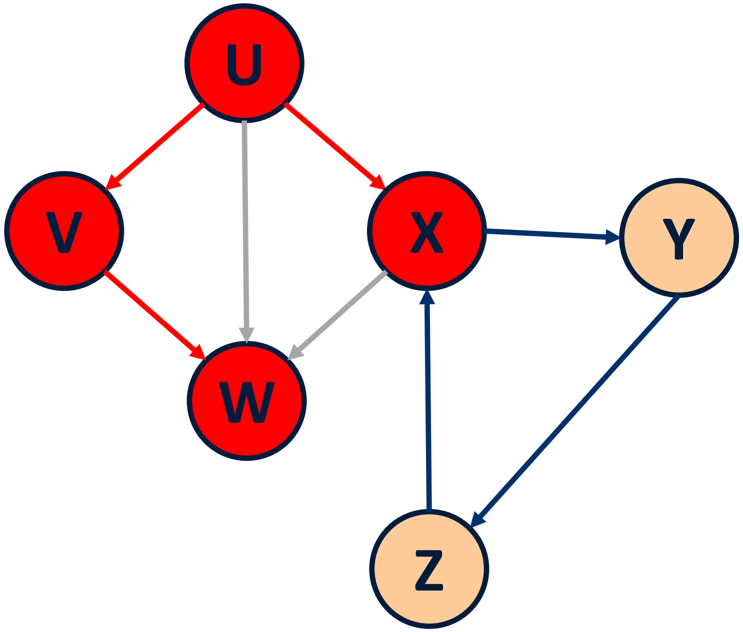



**Visited**

**U - V - W - X**

# DFS Walk-Through

- **DFS** from node **U**

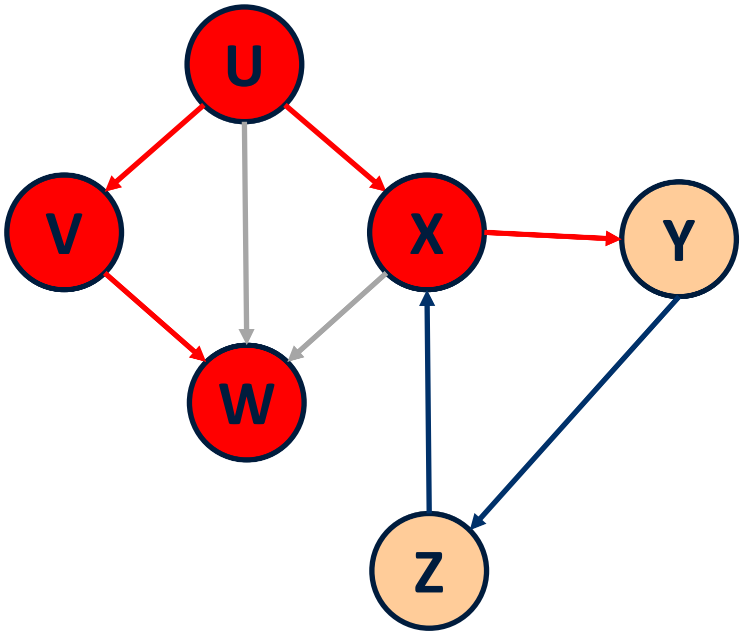



**Visited**

**U - V - W - X**

# DFS Walk-Through

- **DFS** from node **U**

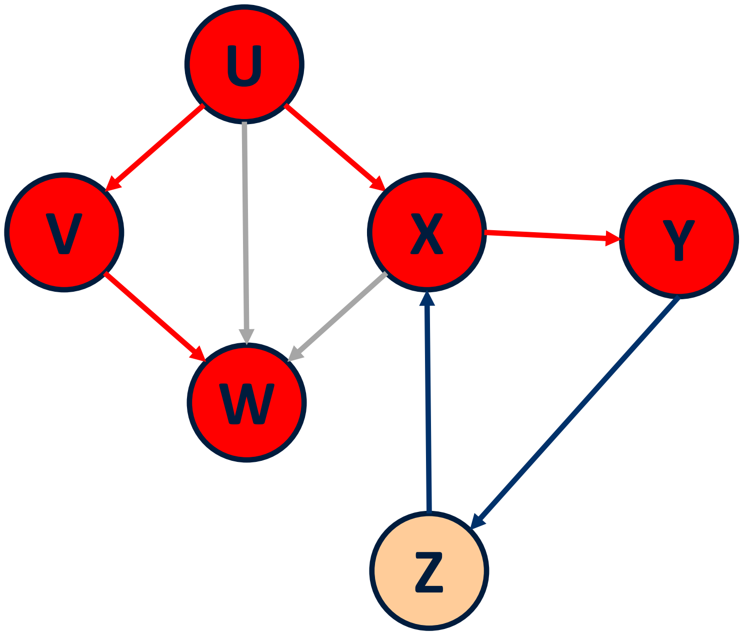



**Visited**

**U - V - W - X**

# DFS Walk-Through

- **DFS** from node **U**

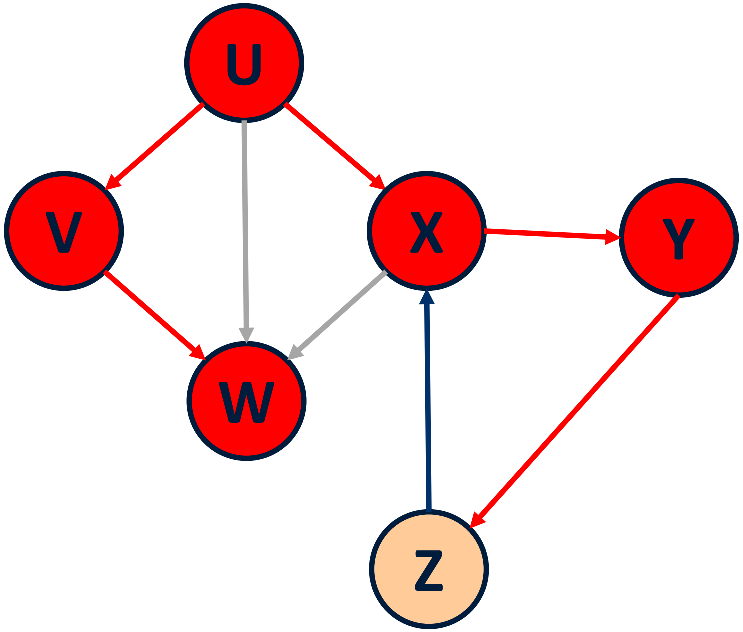



**Visited**

**U - V - W - X - Y**

# DFS Walk-Through

- **DFS** from node **U**

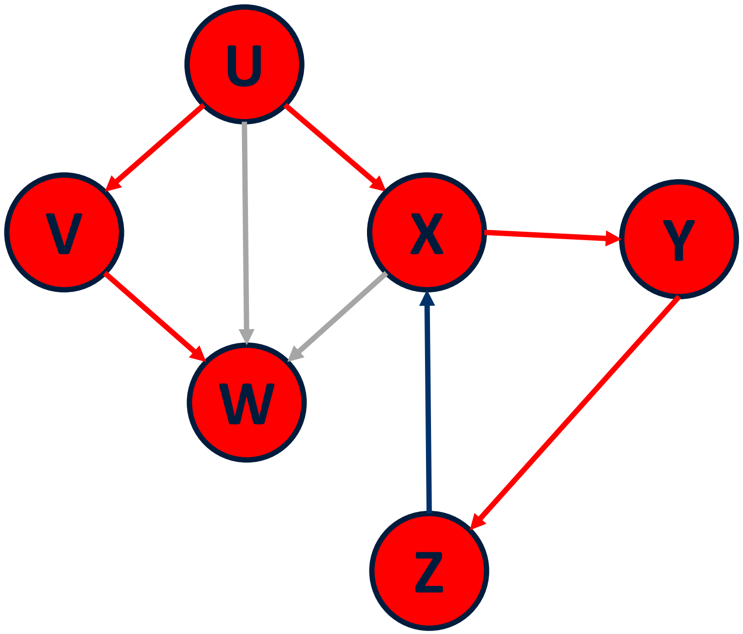



**Visited**

**U - V - W - X - Y**

# DFS Walk-Through

- **DFS** from node **U**

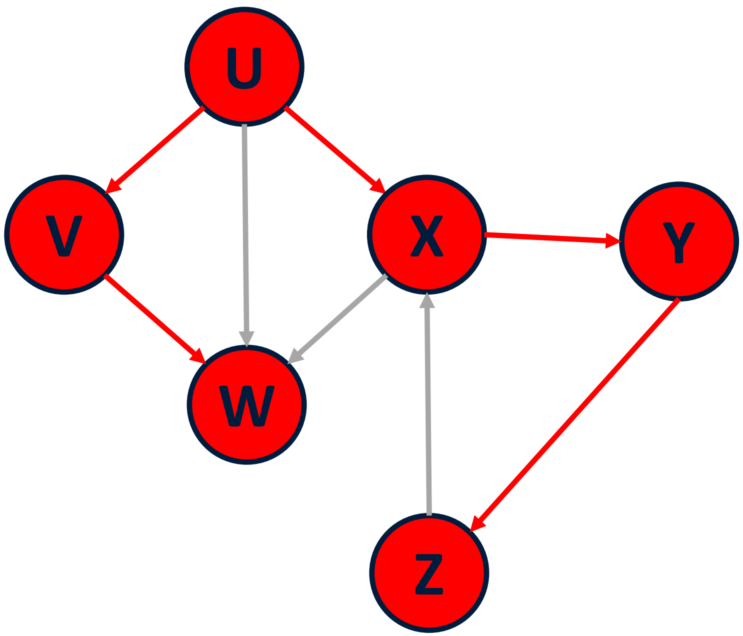



**Visited**

**U - V - W - X - Y - Z**

# DFS Walk-Through

- **DFS** from node **U**




**Visited**

**U - V - W - X - Y - Z**

# Breadth-First Search

- **Breadth-first search (BFS)** finds a path between two nodes by taking one step down all paths and then immediately backtracking

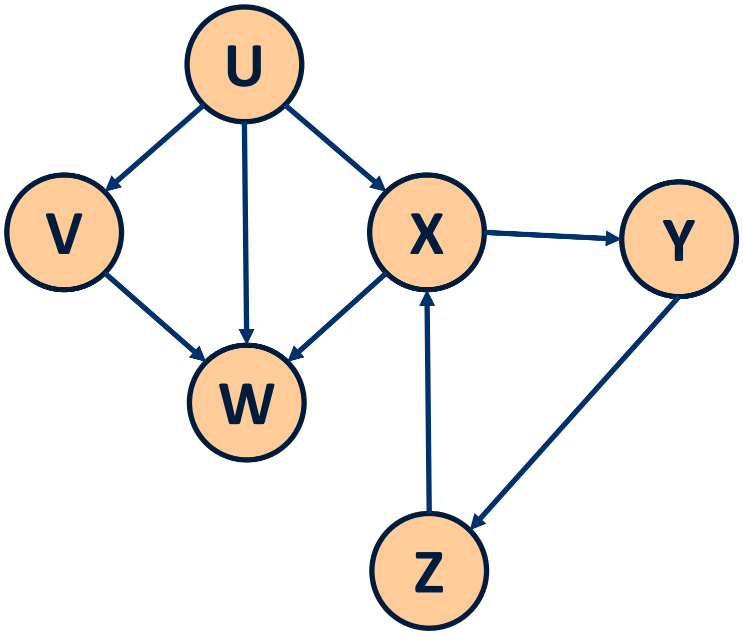
- BFS always returns the path with the fewest edges between the start and the goal vertices (shortest path for not weighted graphs)

# Breadth-First Search

```
set all nodes to "not visited";
q = new Queue();
q.enqueue(initial node);

while ( q ≠ empty ) do
    x = q.dequeue();

    if ( x has not been visited )
        visited[x] = true;

    for ( every edge (x, y))
        if ( y has not been visited)
            q.enqueue(y);
```

# BFS Walk-Through

- **BFS** from node **U**



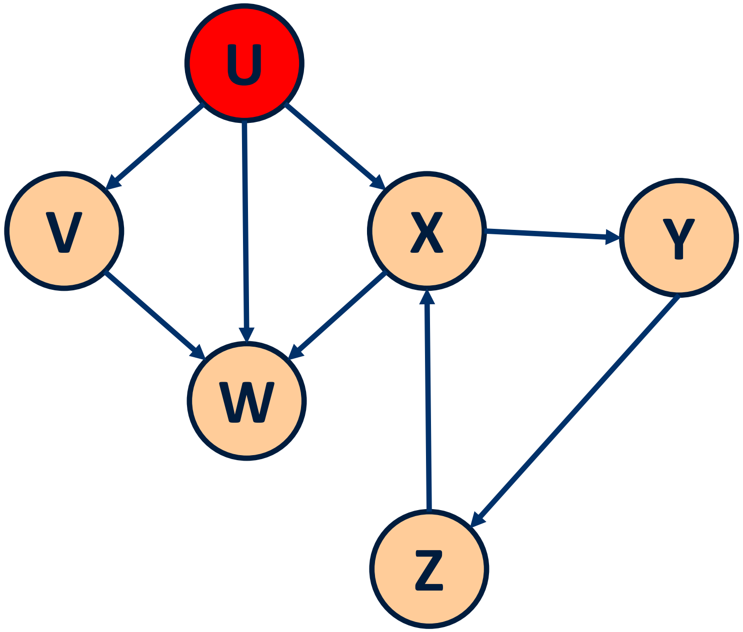

**Queue**

**U**

**Visited**

# BFS Walk-Through

- **BFS** from node **U**




**Queue**

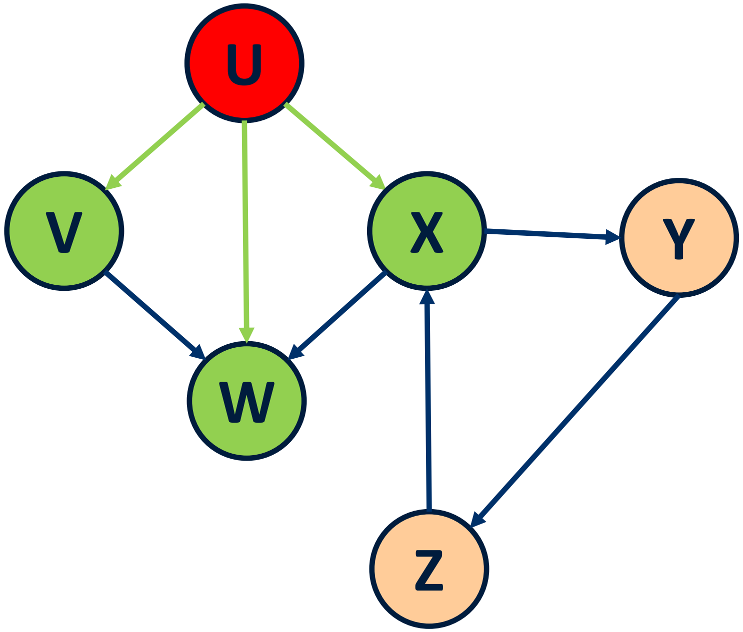
**~~U~~** **Dequeue(U)**

**Visited**

**U -**

# BFS Walk-Through

- **BFS** from node **U**




**Queue**

**V, W, X**

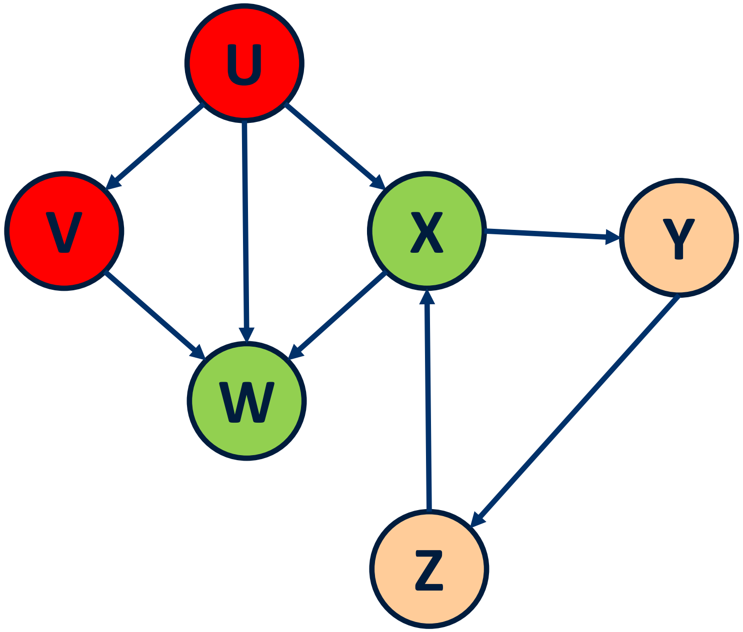
**Dequeue(U)**
**Enqueue(V, W, X)**

**Visited**

**U -**

# BFS Walk-Through

- **BFS** from node **U**




**Queue**

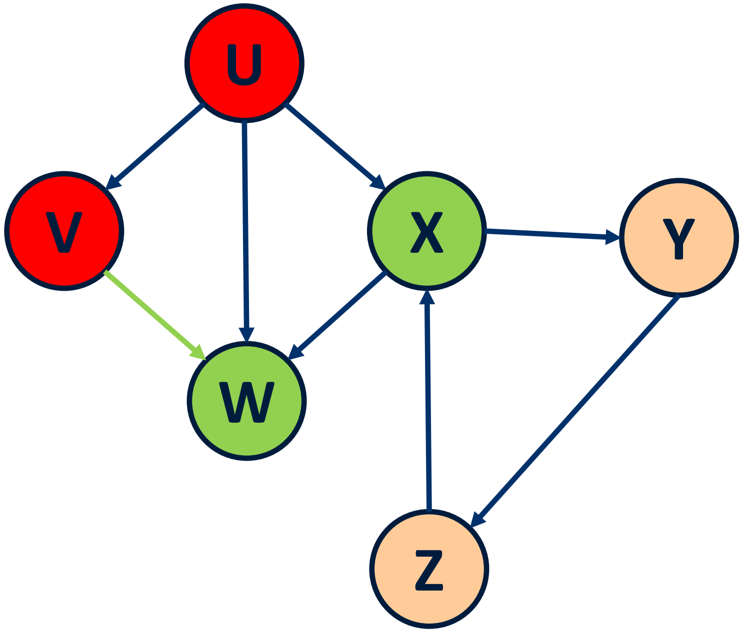
**~~V~~, W, X** **Dequeue(V)**

**Visited**

**U - V -**

# BFS Walk-Through

- **BFS** from node **U**




**Queue**

**~~V~~, W, X, W**

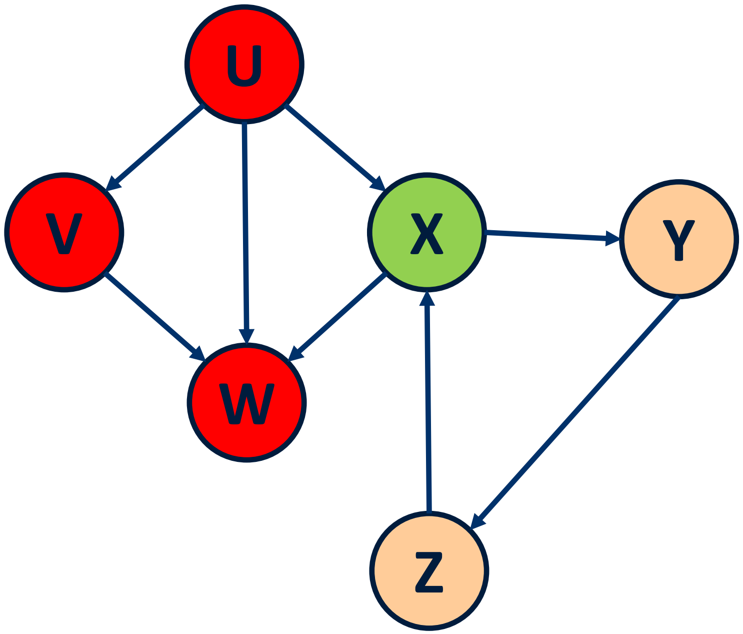
**Dequeue(V)**
**Enqueue(W)**

**Visited**

**U - V -**

# BFS Walk-Through

- **BFS** from node **U**




**Queue**

**~~W~~, X, W**

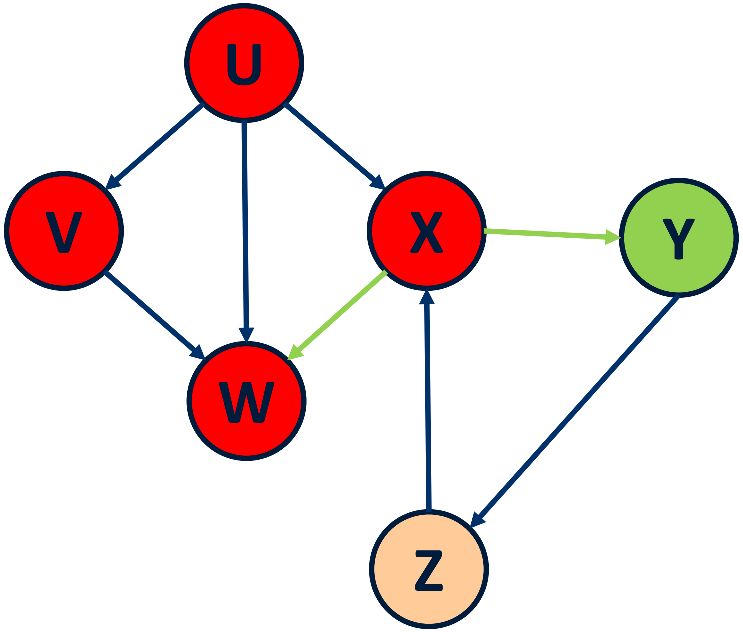
**Dequeue(W)**

**Visited**

**U - V - W -**

# BFS Walk-Through

- **BFS** from node **U**




**Queue**

**~~X~~, W, W, Y**

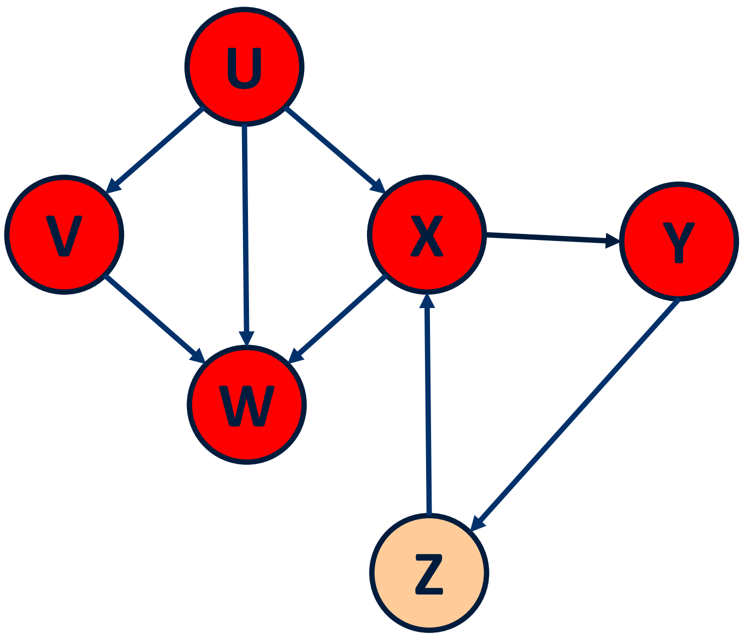
**Dequeue(X)**
**Enqueue(W, Y)**

**Visited**

**U - V - W - X -**

# BFS Walk-Through

- **BFS** from node **U**




**Queue**

**~~W~~, ~~W~~, Y**

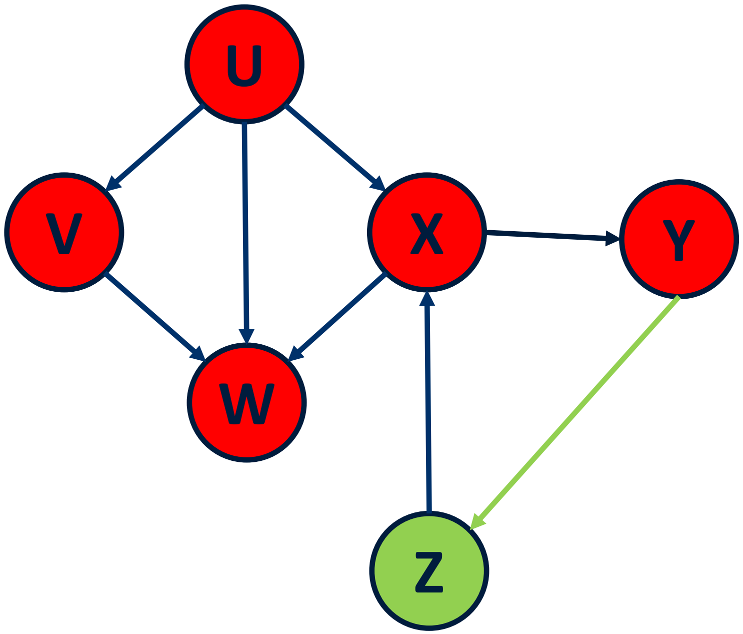
**Dequeue(W)**
**Dequeue(W)**
**Dequeue(Y)**

**Visited**

**U - V - W - X**

# BFS Walk-Through

- **BFS** from node **U**




**Queue**

**~~W~~, ~~W~~, ~~Y~~, Z**

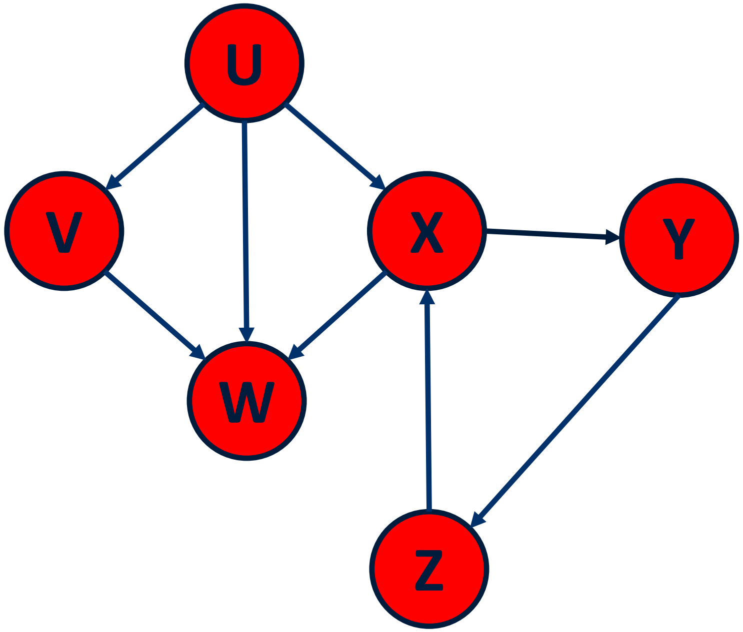
**Visited**

**U - V - W - X - Y**

**Dequeue(W)**
**Dequeue(W)**
**Dequeue(Y)**
**Enqueue(Z)**

# BFS Walk-Through

- **BFS** from node **U**




**Queue**

**~~Z~~** **Dequeue(Z)**

**Visited**

**U - V - W - X - Y - Z**

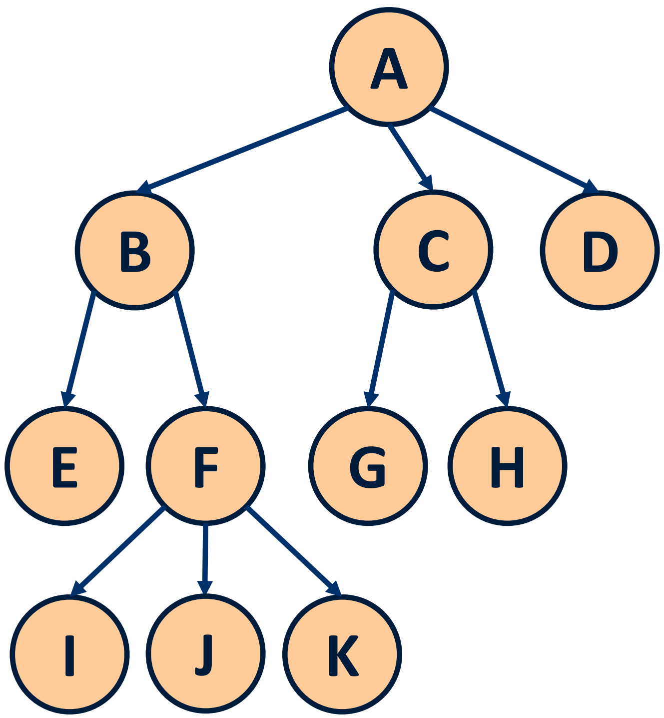
# DFS, BFS runtime

- What is the runtime of DFS and BFS, in terms of the number of vertices *V* and the number of edges *E* ?

- Answer: ***O(|V| + |E|)***
  - each algorithm must potentially visit every node and/or examine every edge once.

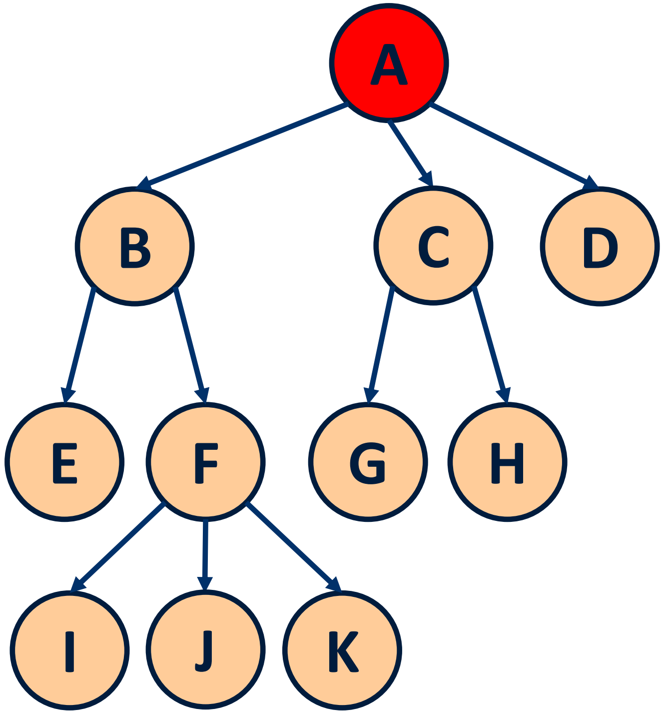
# DFS, BFS in Trees

## DFS



# DFS, BFS in Trees

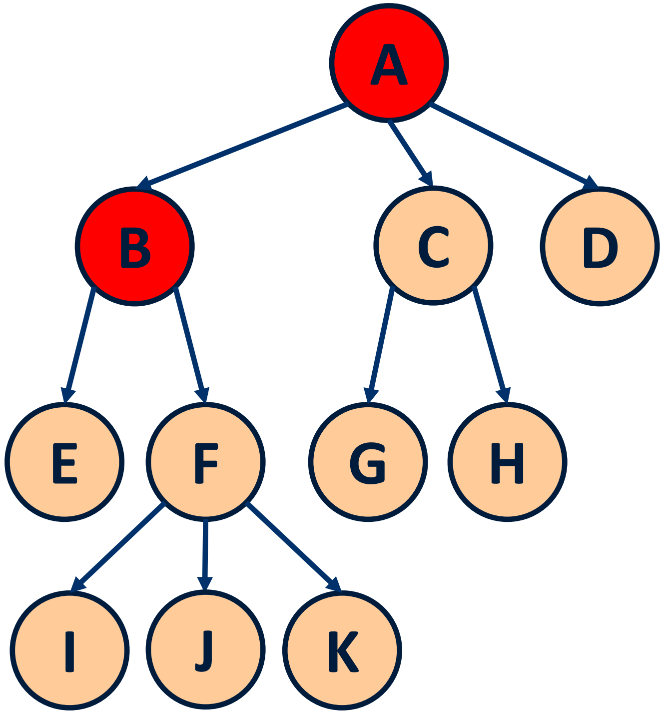
## DFS




A

# DFS, BFS in Trees

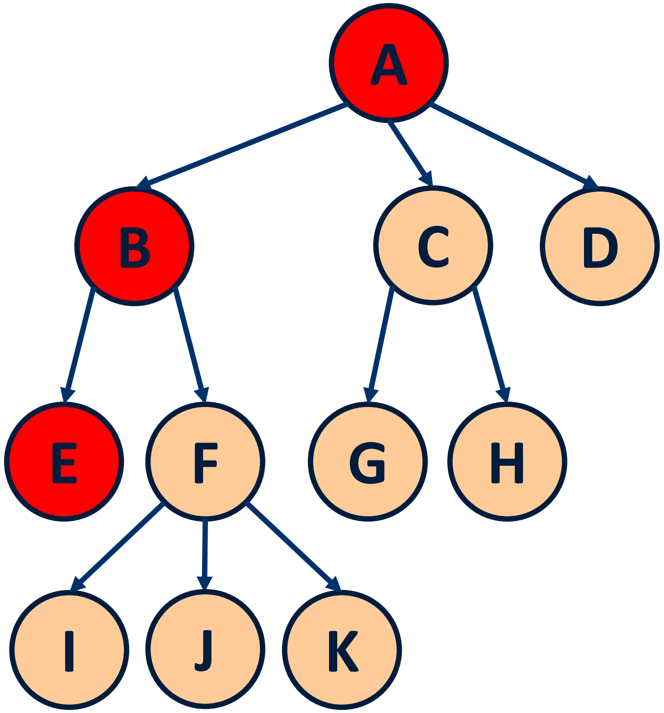
## DFS




**A - B**

# DFS, BFS in Trees

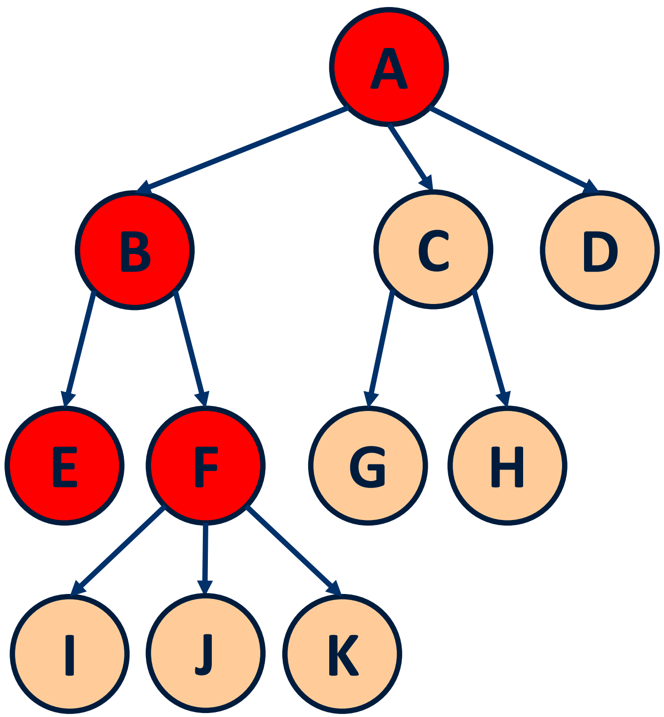
## DFS




A - B - E

# DFS, BFS in Trees

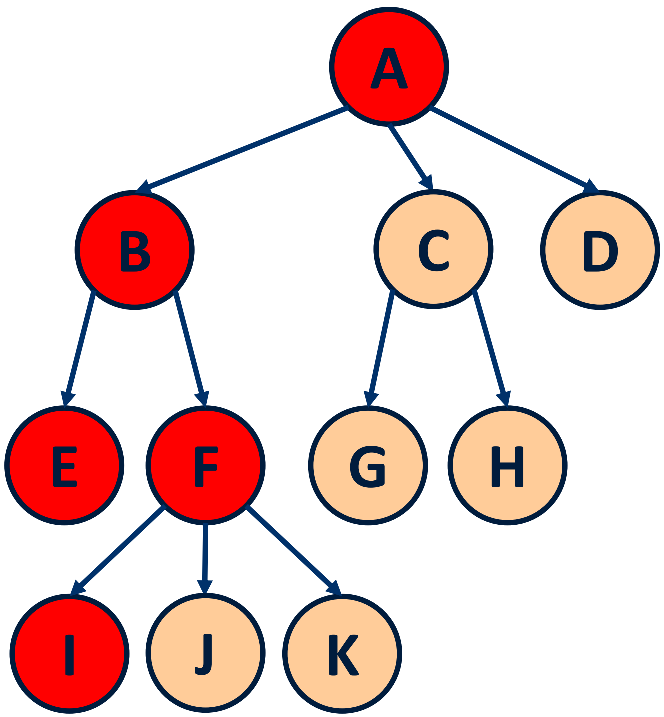
## DFS




A - B - E - F

# DFS, BFS in Trees

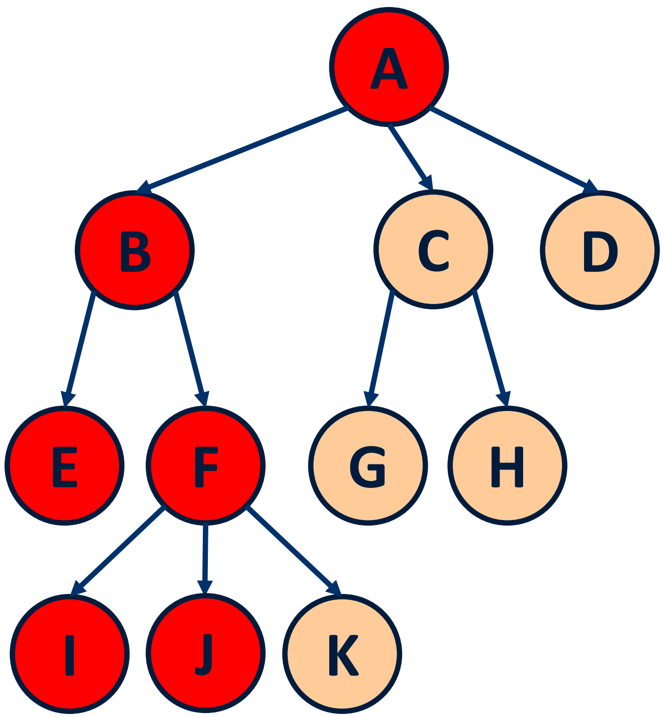
## DFS




A - B - E - F - I

# DFS, BFS in Trees

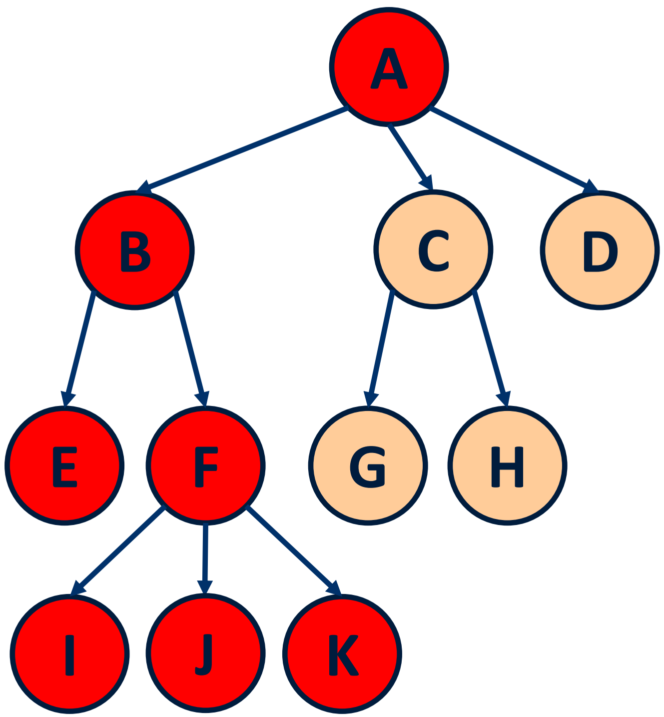
## DFS




A - B - E - F - I - J

# DFS, BFS in Trees

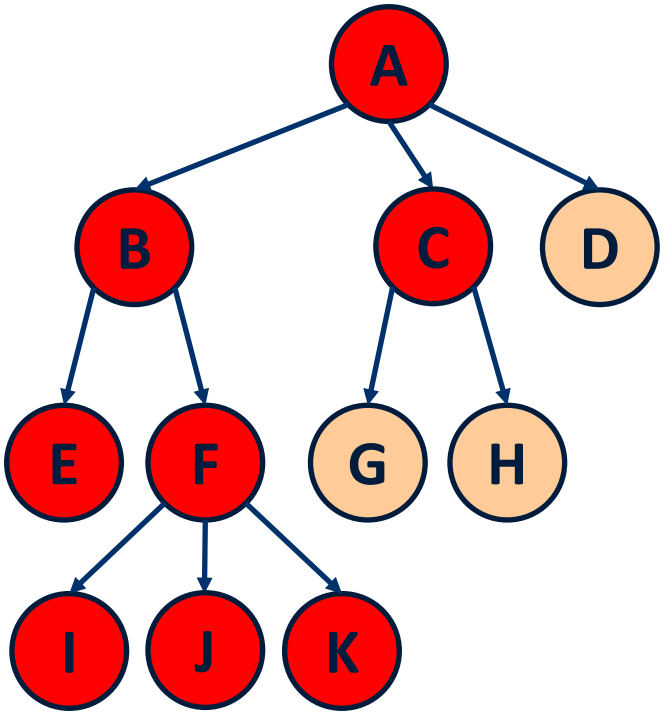
## DFS




A - B - E - F - I - J - K

# DFS, BFS in Trees

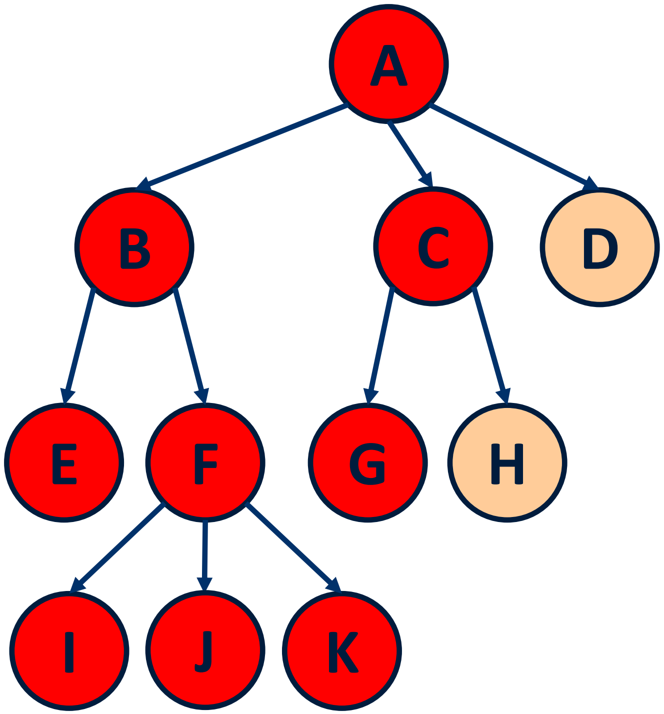
## DFS




A - B - E - F - I - J - K - C

# DFS, BFS in Trees

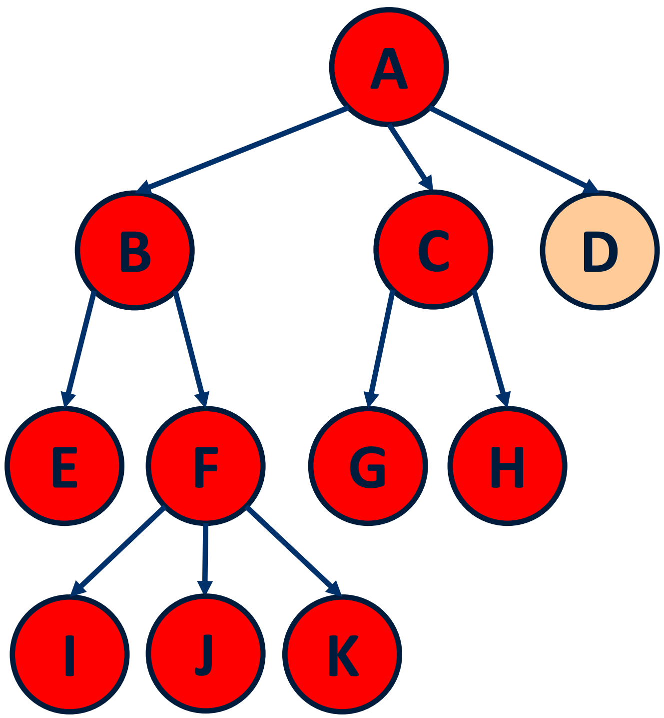
## DFS




A - B - E - F - I - J - K - C - G

# DFS, BFS in Trees

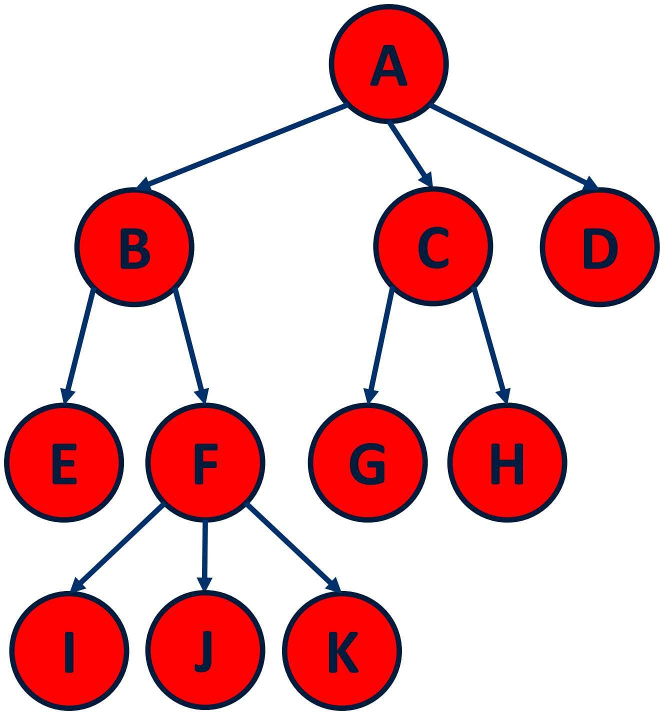
## DFS




A - B - E - F - I - J - K - C - G - H

# DFS, BFS in Trees

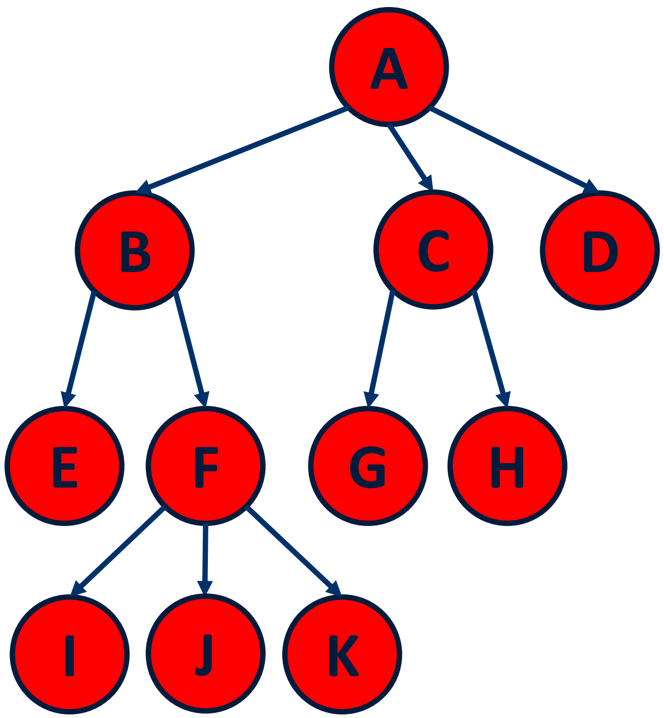
## DFS




A - B - E - F - I - J - K - C - G - H - D

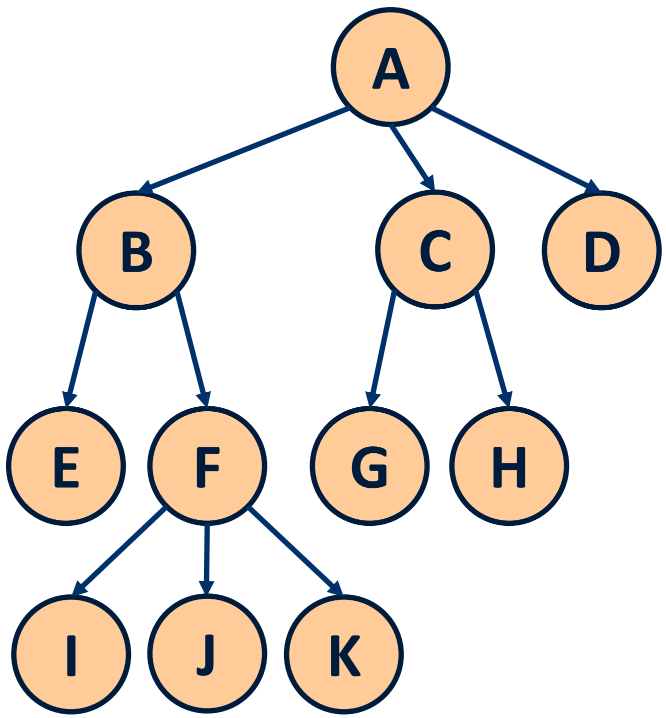
# DFS, BFS in Trees

## DFS



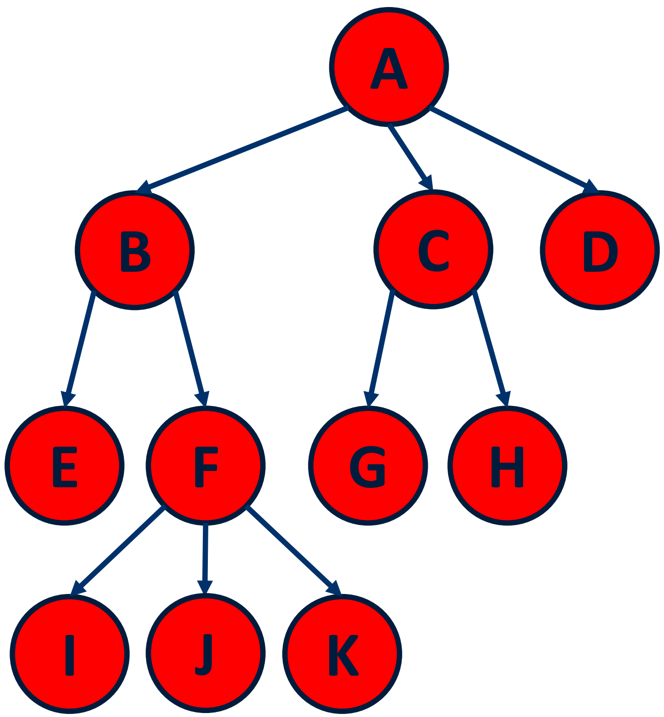

A - B - E - F - I - J - K - C - G - H - D

## BFS



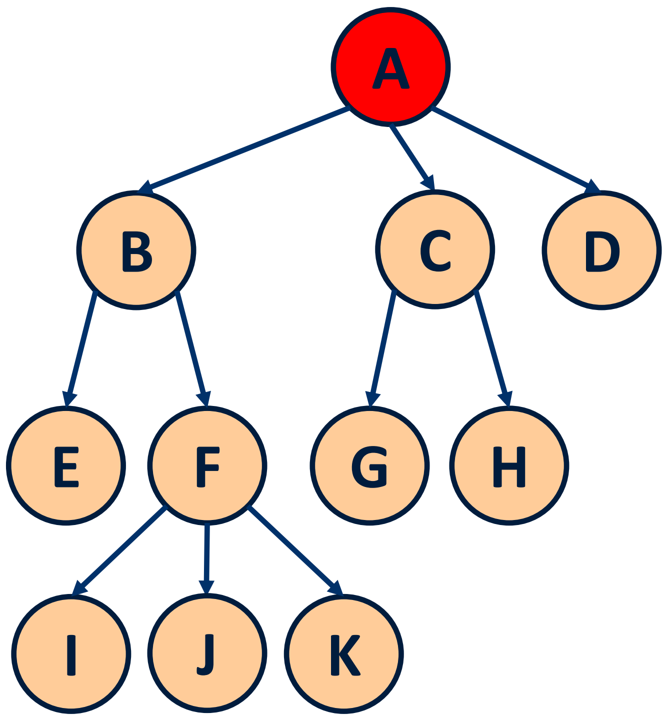
# DFS, BFS in Trees

## DFS



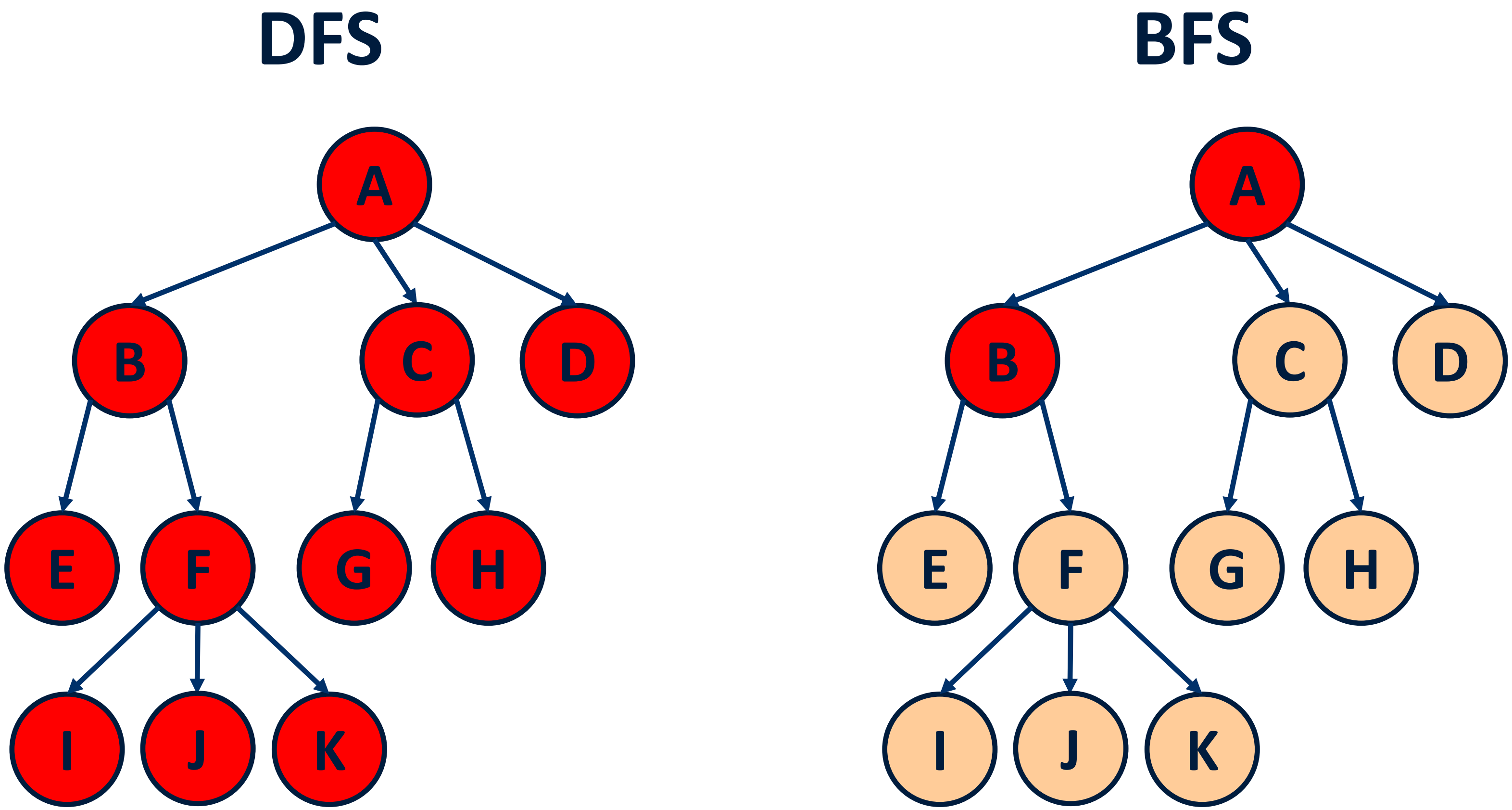

A - B - E - F - I - J - K - C - G - H - D

## BFS




A

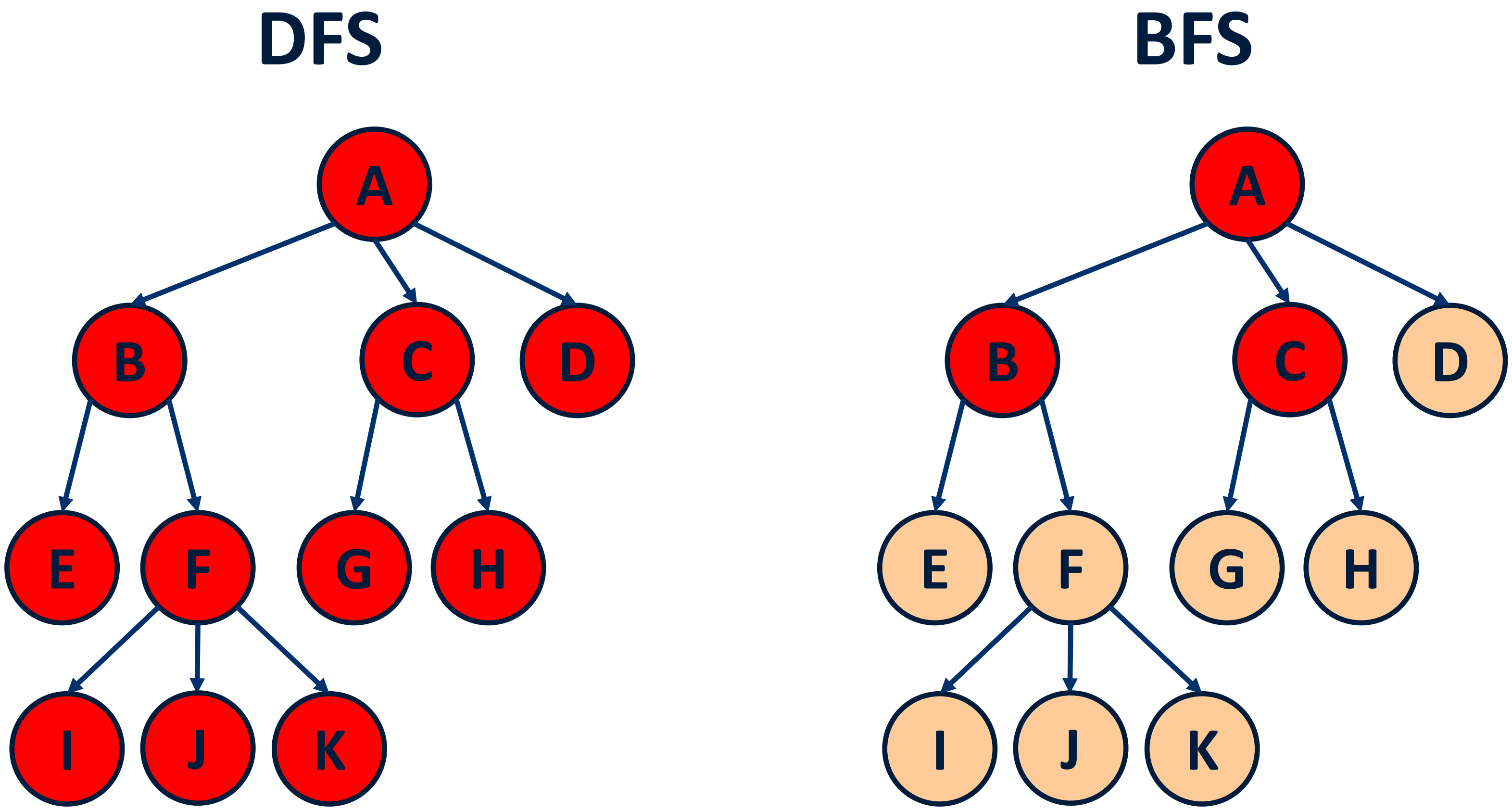
# DFS, BFS in Trees




A - B - E - F - I - J - K - C - G - H - D

A - B

# DFS, BFS in Trees

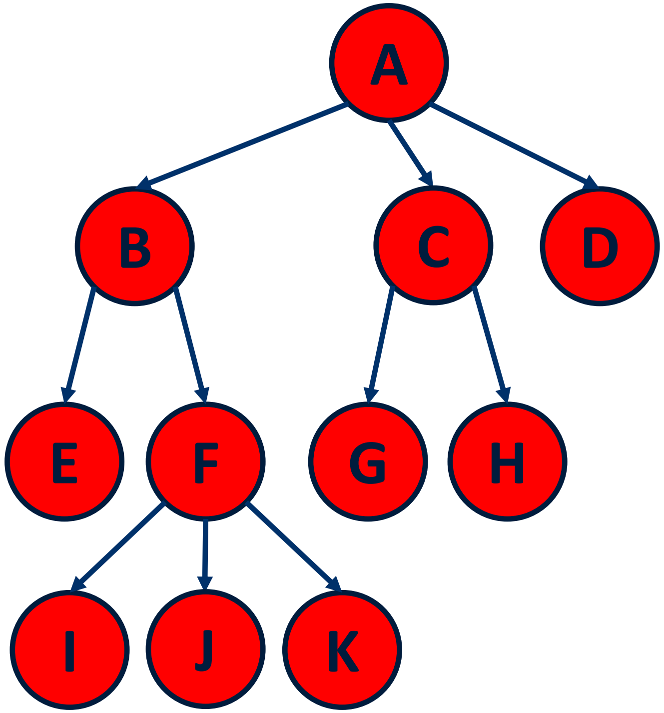



A - B - E - F - I - J - K - C - G - H - D

A - B - C

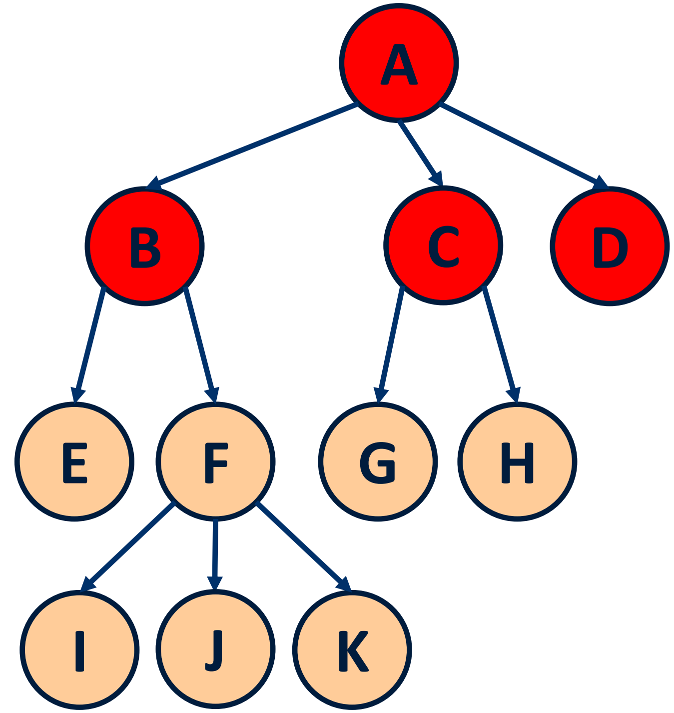
# DFS, BFS in Trees

## DFS



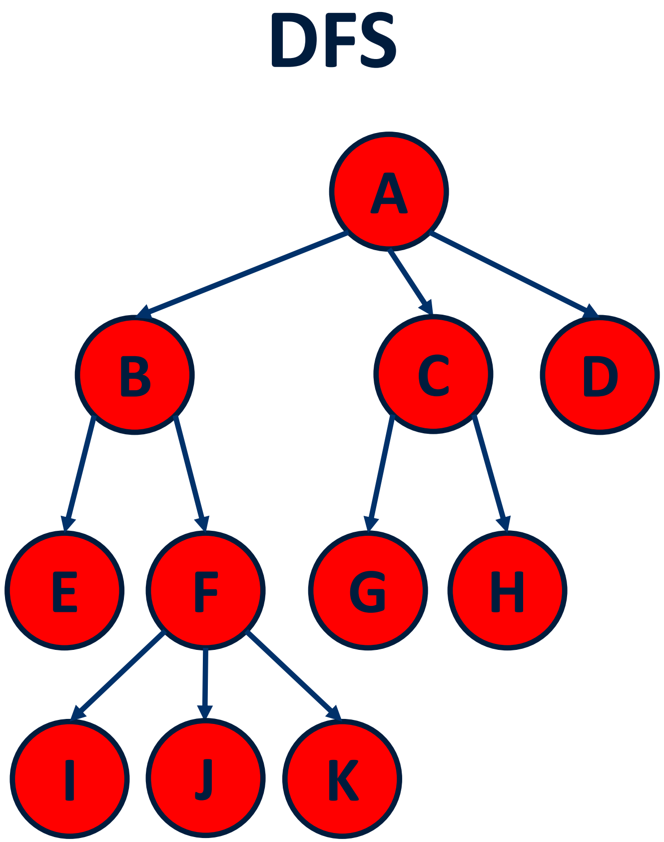

A - B - E - F - I - J - K - C - G - H - D

## BFS

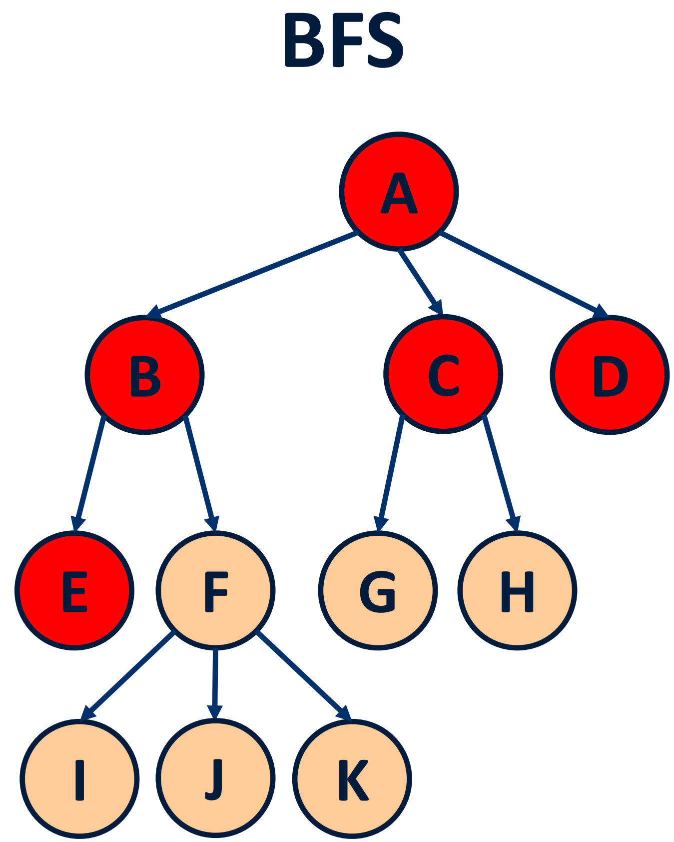



A - B - C - D

# DFS, BFS in Trees

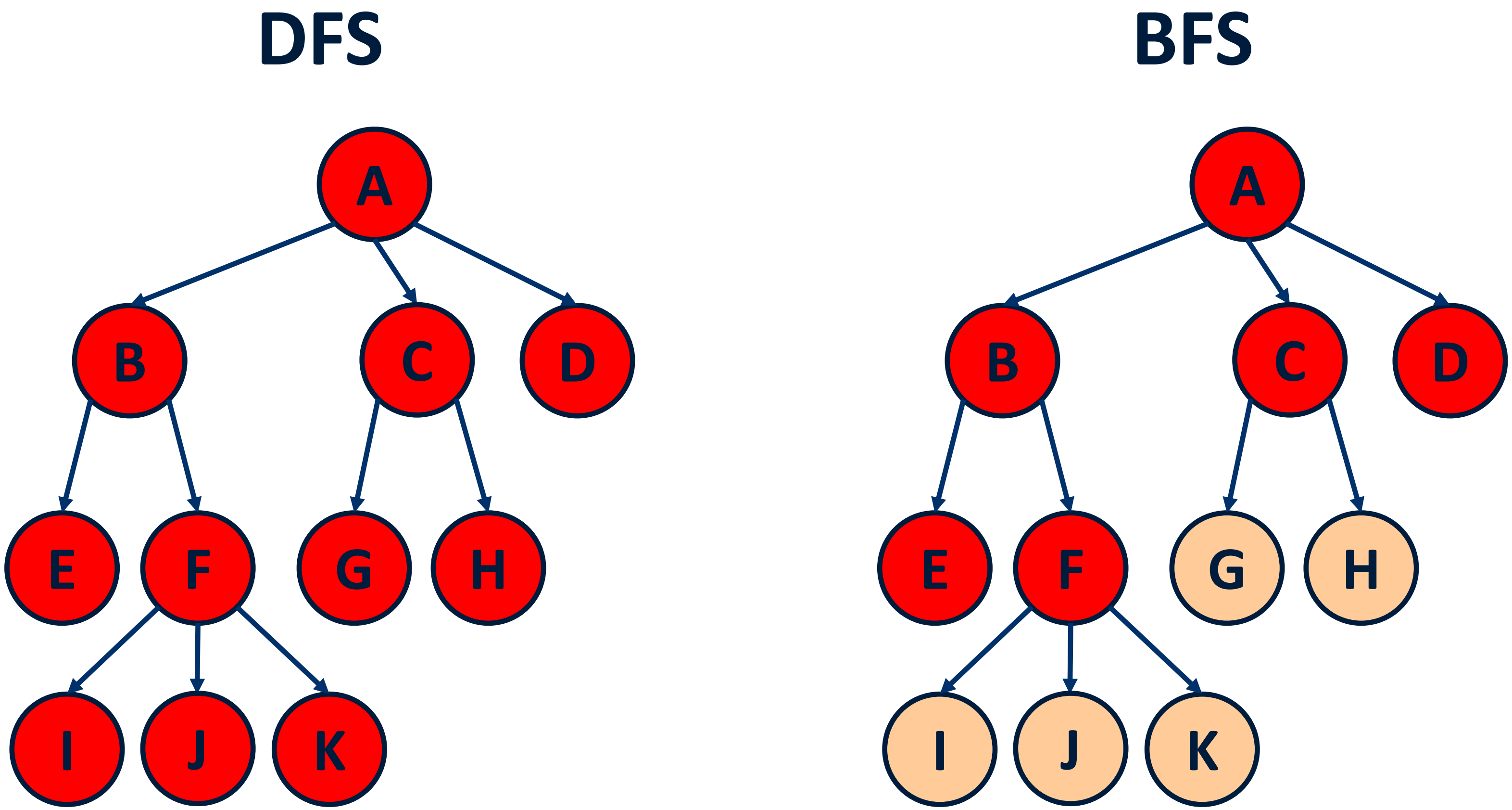



A - B - E - F - I - J - K - C - G - H - D




A - B - C - D - E

# DFS, BFS in Trees

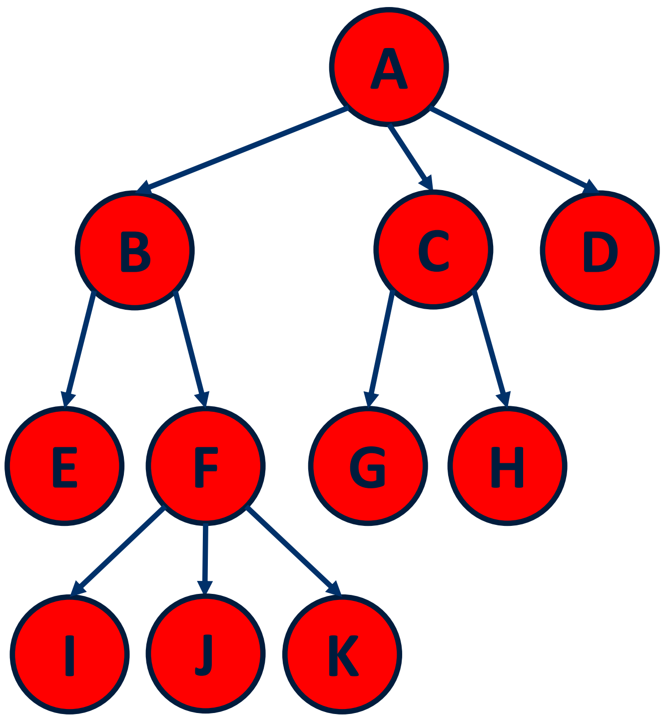



A - B - E - F - I - J - K - C - G - H - D

A - B - C - D - E - F

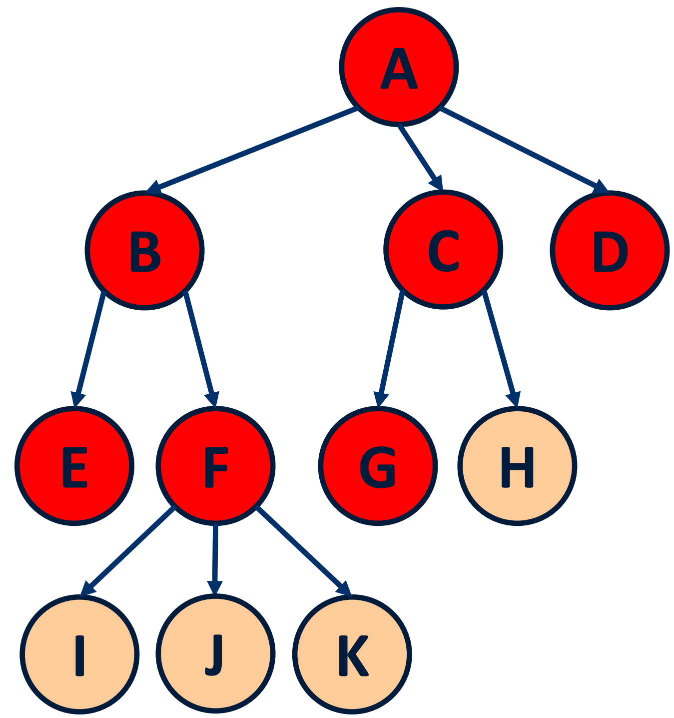
# DFS, BFS in Trees

**DFS**




A - B - E - F - I - J - K - C - G - H - D

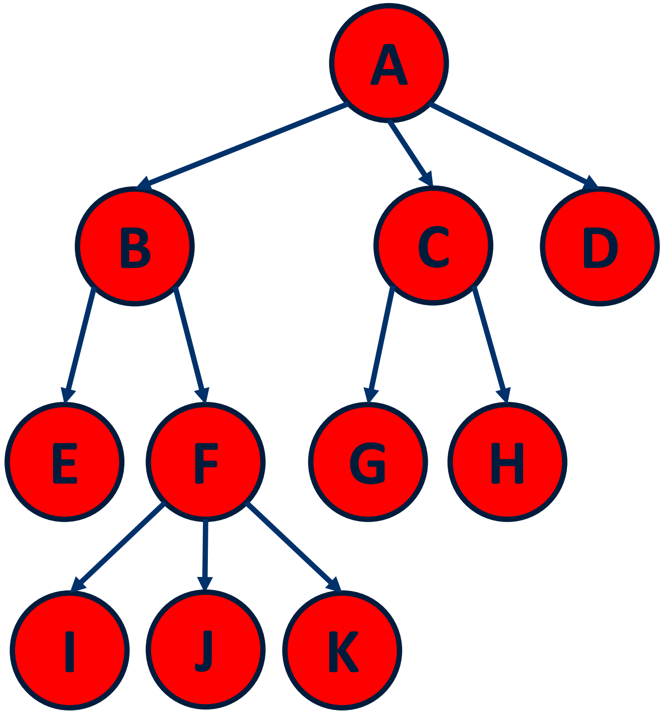
**BFS**




A - B - C - D - E - F - G

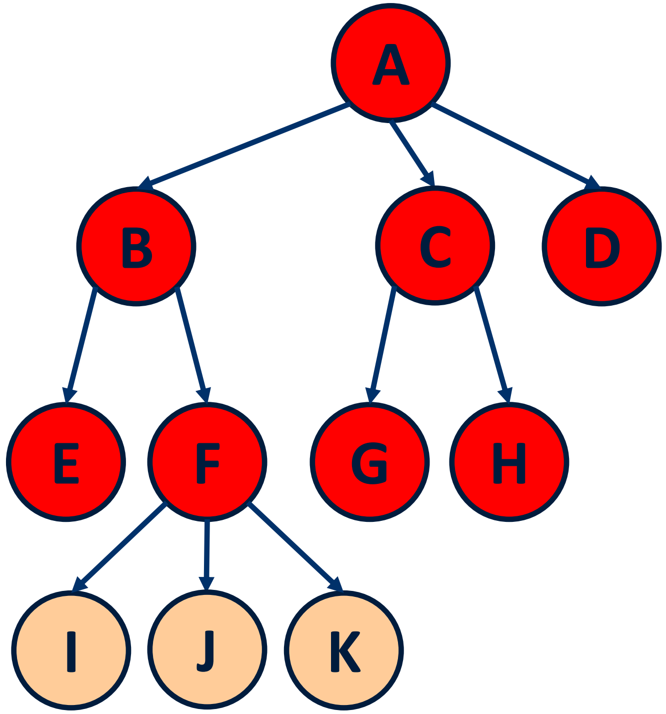
# DFS, BFS in Trees

## DFS



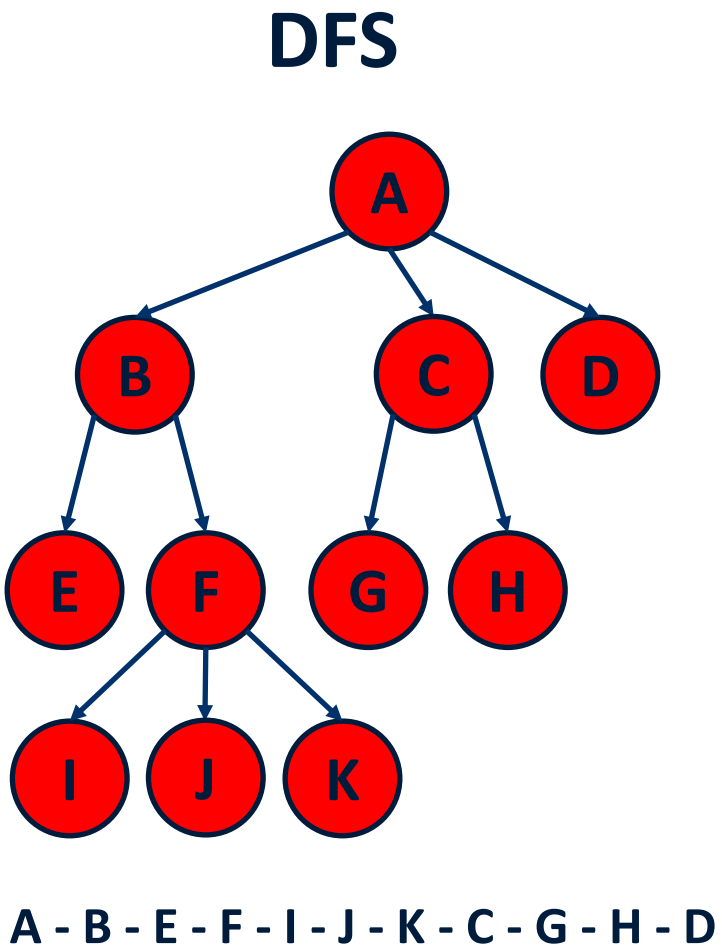

A - B - E - F - I - J - K - C - G - H - D

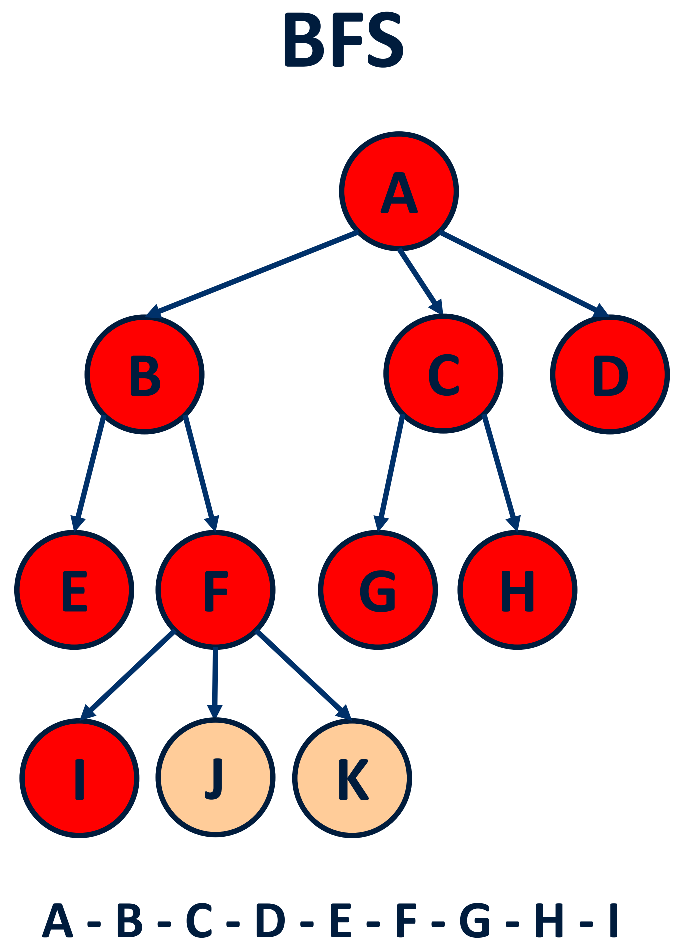
## BFS




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

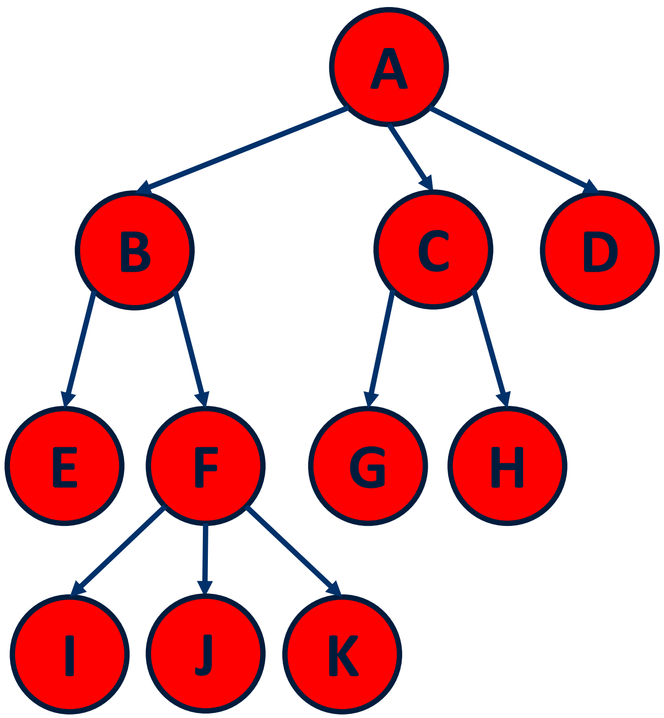
# DFS, BFS in Trees



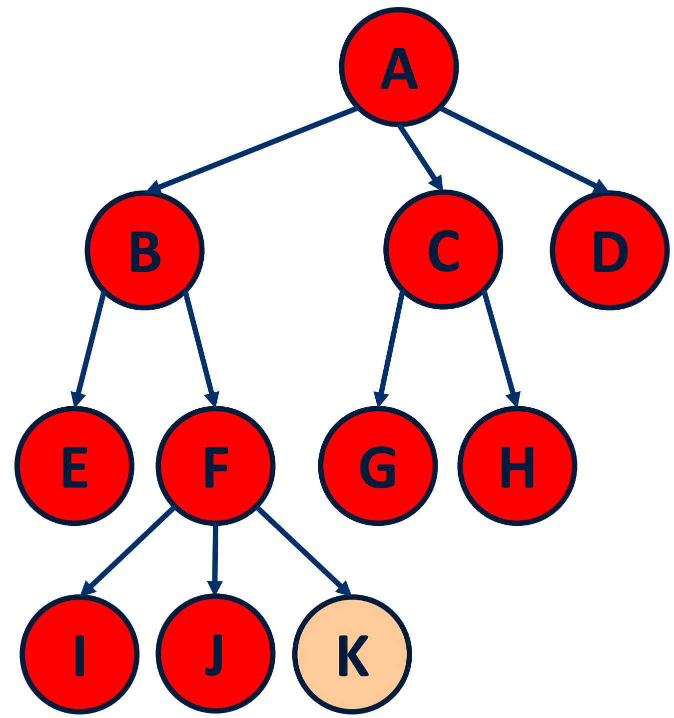
# DFS, BFS in Trees

## DFS




A - B - E - F - I - J - K - C - G - H - D

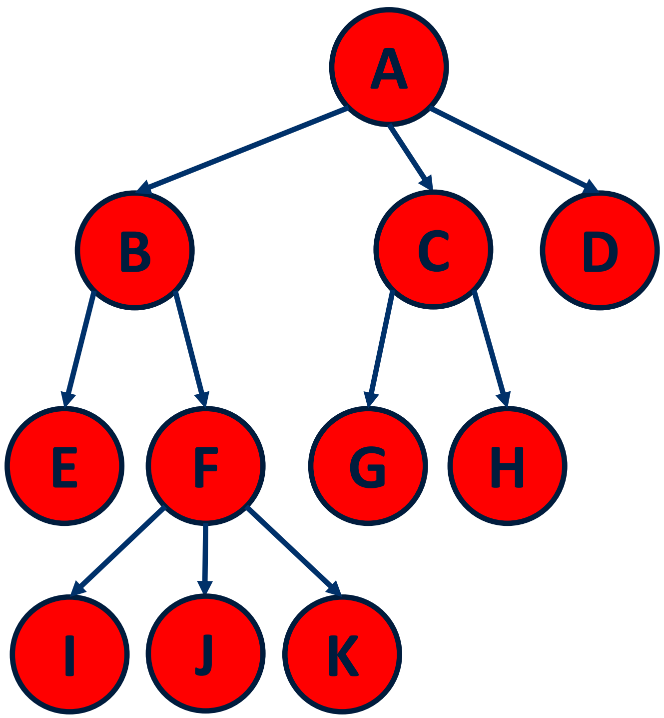
## BFS




A - B - C - D - E - F - G - H - I - J

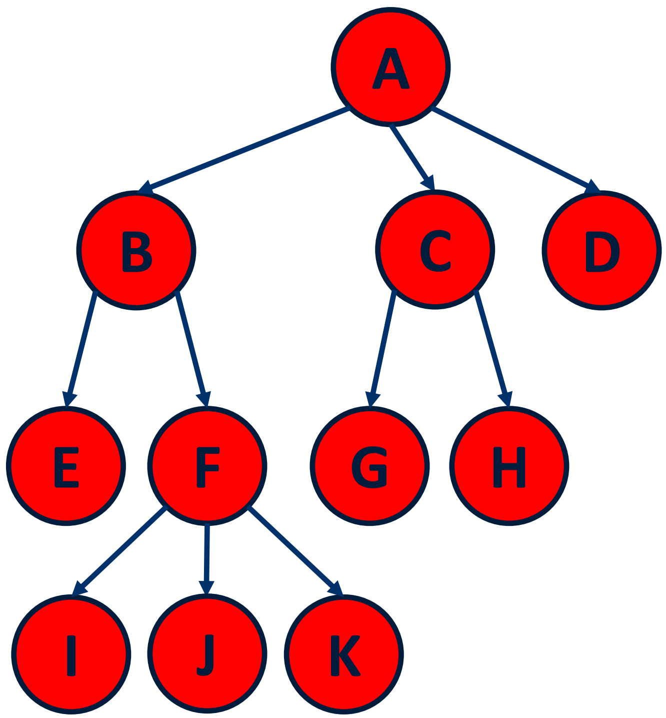
# DFS, BFS in Trees

**DFS**




**A - B - E - F - I - J - K - C - G - H - D**

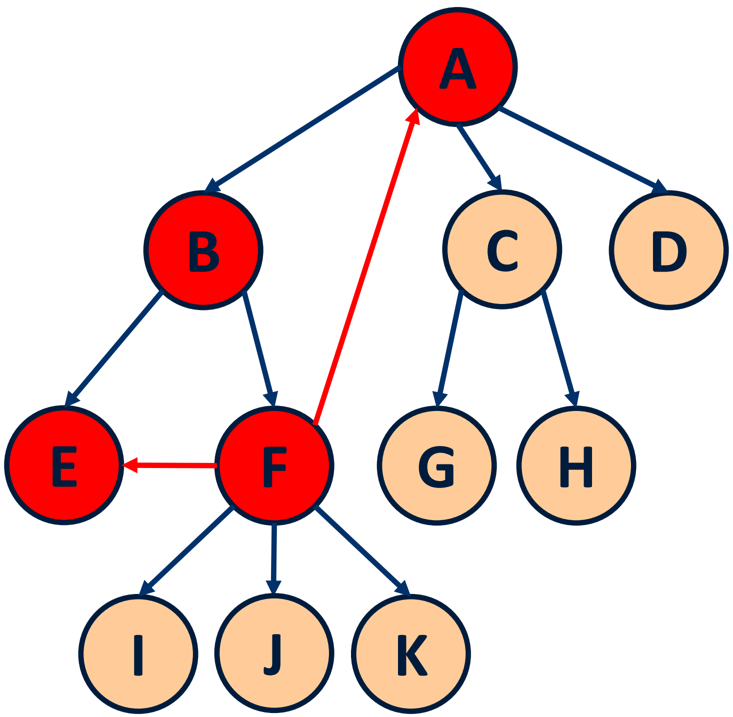
**BFS**




**A - B - C - D - E - F - G - H - I - J - K**

# DFS, BFS in Trees

**DFS**




**A - B - E - F**

- During a visit on a graph from a potential root
  - If we find a node already visited
  - either a cycle or there are two paths from the start node to that node
  - **NOT a tree**

# DFS or BFS

- We can use the BFS or DFS traversal algorithm, for a graph ***G*,** to solve the following problems in $O(|V| + |E|)$ time:

  - Compute the **connected components** of ***G***
  - Compute a **spanning forest** of ***G***
  - Find a **simple cycle** in ***G***, or report that ***G*** is a forest
  - Given two vertices of ***G*,** find a path in ***G*** between them or report that no such path exists.
  - **Check if a graph is a tree**
  - Find **shortest paths** in a not weighted graph

# Dijkstra's Algorithm

# Dijkstra's Algorithm

- Given:
  - ***weighted*** directed graph with **nonnegative weights**
- Task:
  - finds the shortest path between two vertices
- Bonus Task:
  - Finds the shortest path from one node to all other nodes in the graph

Why not use Breath-First-Search?

- Not correct for weighted graphs

# Dijkstra's Algorithm

- Idea:
  - Create a table about current best way to each vertex
    - Distance
    - Previous vertex (Backpointer)

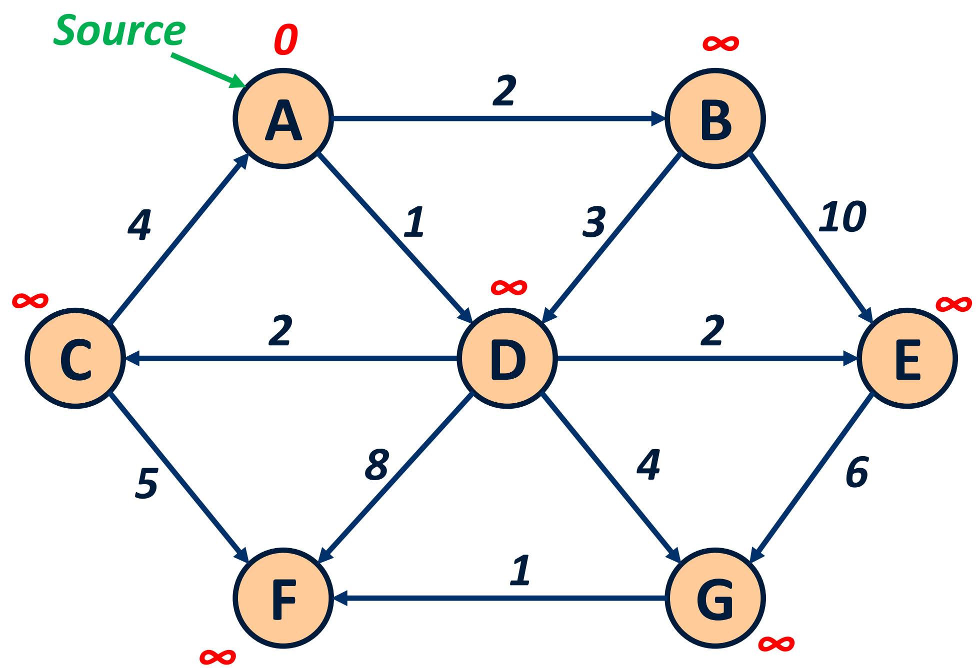
  - Improve it until it reaches the best solution
    - Select nearest not visited node
    - Update distance of its neighbours

  - Key operation: **Edge relaxation**

# Example: Algorithm

- Initialization
  - Create  set Q of verteces to visit (all)
  - Set all distances to infinity
  - Set source distance to zero
  - Set all previous to null

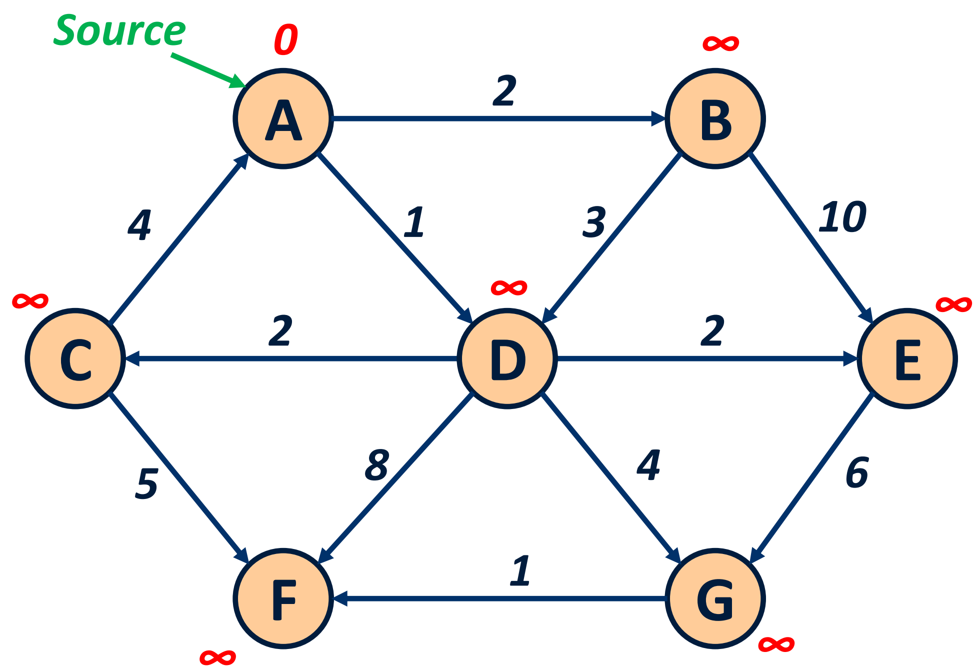
| Node | A | B | C | D | E | F | G |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Dist. | 0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| Prev. | - | - | - | - | - | - | - |



# Example: Algorithm

- While Q not empty
  - Select from Q node ***v*** with minimum distance
  - Remove ***v*** from Q
  - ***Relax all outgoing edges***

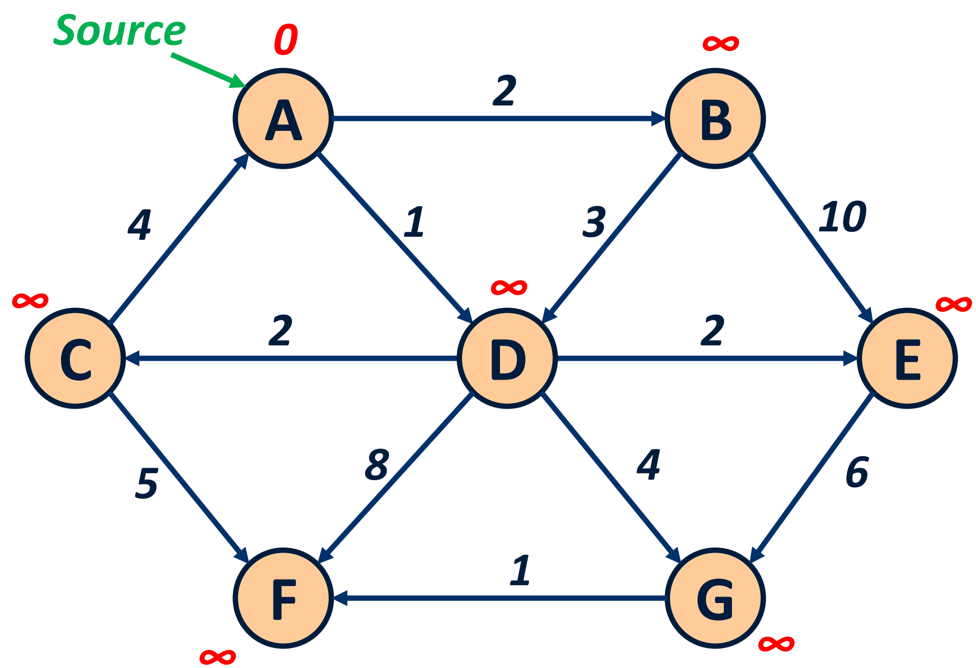
| Node | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| Dist. | 0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| Prev. | - | - | - | - | - | - | - |



# Example: Algorithm

- While Q not empty
  - Select from Q node ***v*** with minimum distance
  - Remove ***v*** from Q
  - ***Relax all outgoing edges***
    - Compute the distance of neighbors passing by ***v***
    - If this is shorter than the current, update distance and pred to ***v***

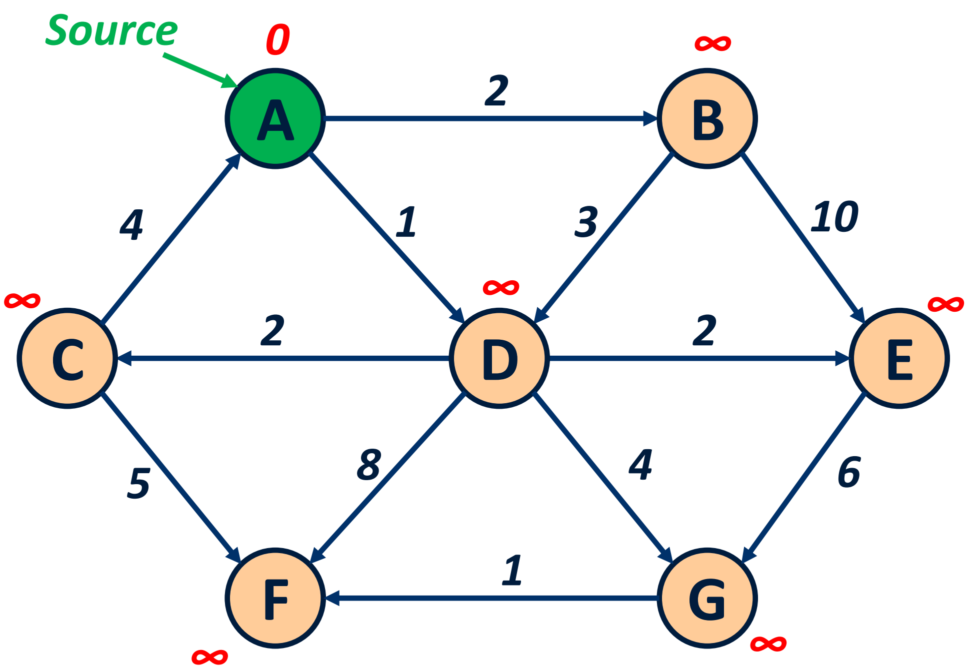
| Node | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| **Dist.** | 0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| **Prev.** | - | - | - | - | - | - | - |



# Example: Algorithm

- While Q not empty
  - Select from Q node ***v*** with minimum distance
  - Remove ***v*** from Q
  - ***Relax all outgoing edges***
    - Compute the distance of neighbors passing by ***v***
    - If this is shorter than the current, update distance and pred to ***v***

| Node | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| Dist. | 0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| Prev. | - | - | - | - | - | - | - |

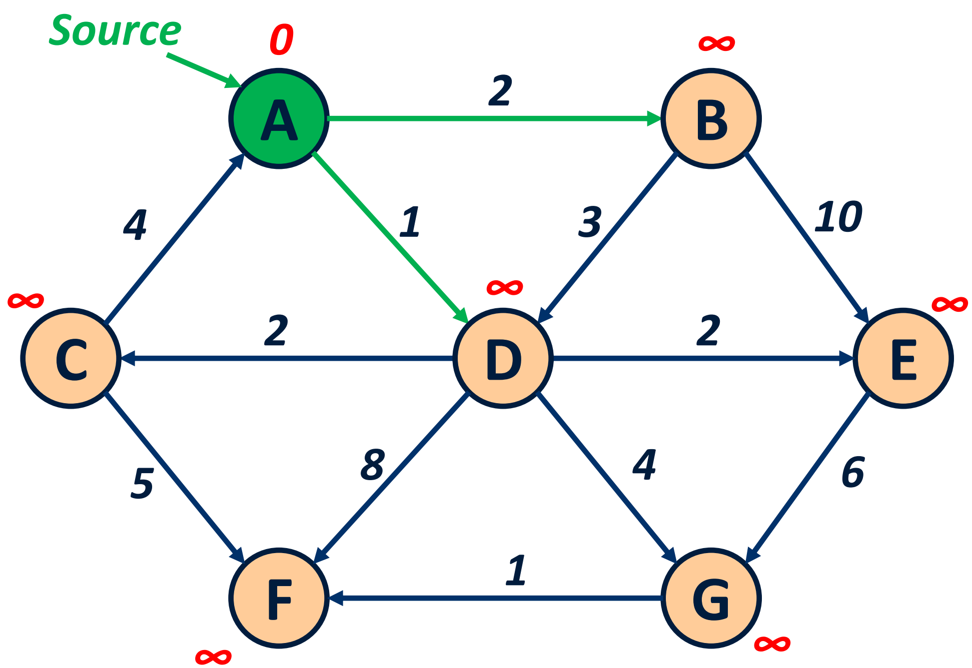



**Select A**

# Example: Algorithm

- While Q not empty
  - Select from Q node ***v*** with minimum distance
  - Remove ***v*** from Q
  - ***Relax all outgoing edges***
    - Compute the distance of neighbors passing by ***v***
    - If this is shorter than the current, update distance and pred to ***v***

| Node | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| Dist. | 0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| Prev. | - | - | - | - | - | - | - |

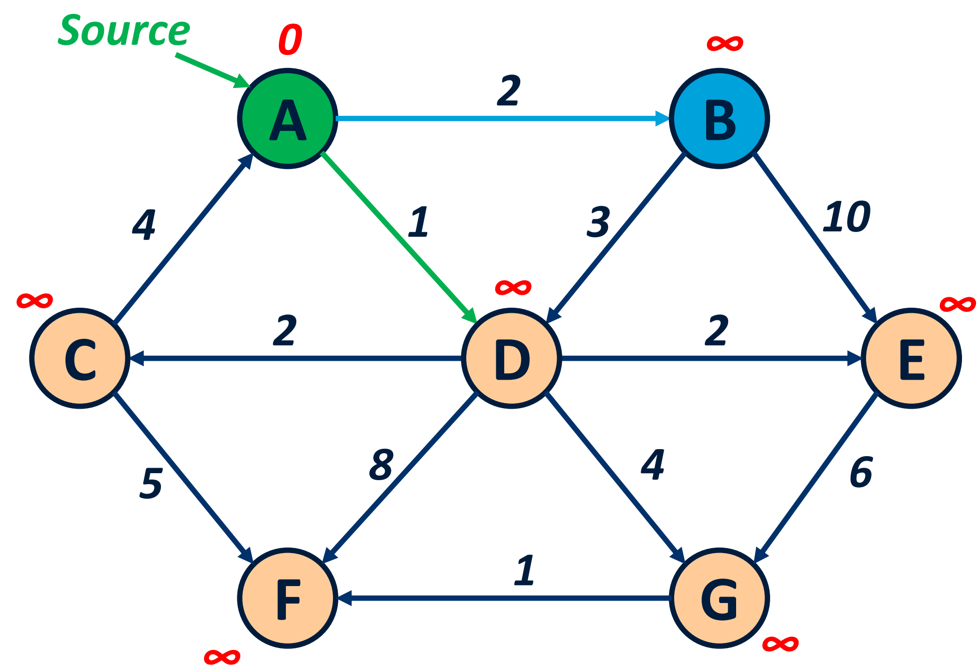



Relax outgoing edges

# Example: Algorithm

- While Q not empty
  - Select from Q node ***v*** with minimum distance
  - Remove ***v*** from Q
  - ***Relax all outgoing edges***
    - Compute the distance of neighbors passing by ***v***
    - If this is shorter than the current, update distance and pred to ***v***

| Node | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| Dist. | 0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| Prev. | - | - | - | - | - | - | - |

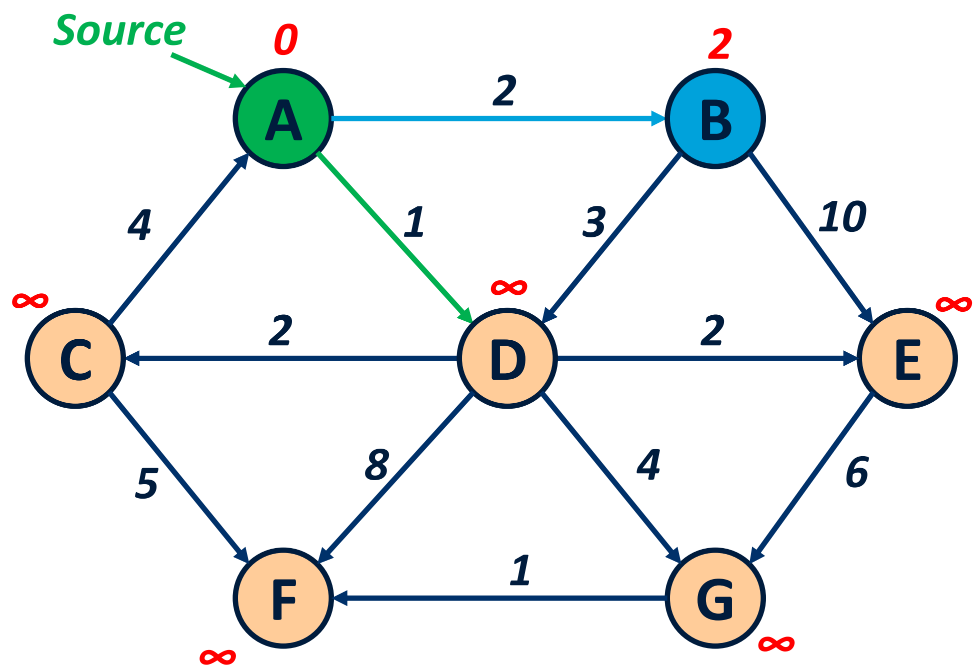



Dist(A) + 2 = 0 + 2 = 2 < ∞ => Update B

# Example: Algorithm

- While Q not empty
  - Select from Q node ***v*** with minimum distance
  - Remove ***v*** from Q
  - ***Relax all outgoing edges***
    - Compute the distance of neighbors passing by ***v***
    - If this is shorter than the current, update distance and pred to ***v***

| Node | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| **Dist.** | 0 | 2 | ∞ | ∞ | ∞ | ∞ | ∞ |
| **Prev.** | - | A | - | - | - | - | - |

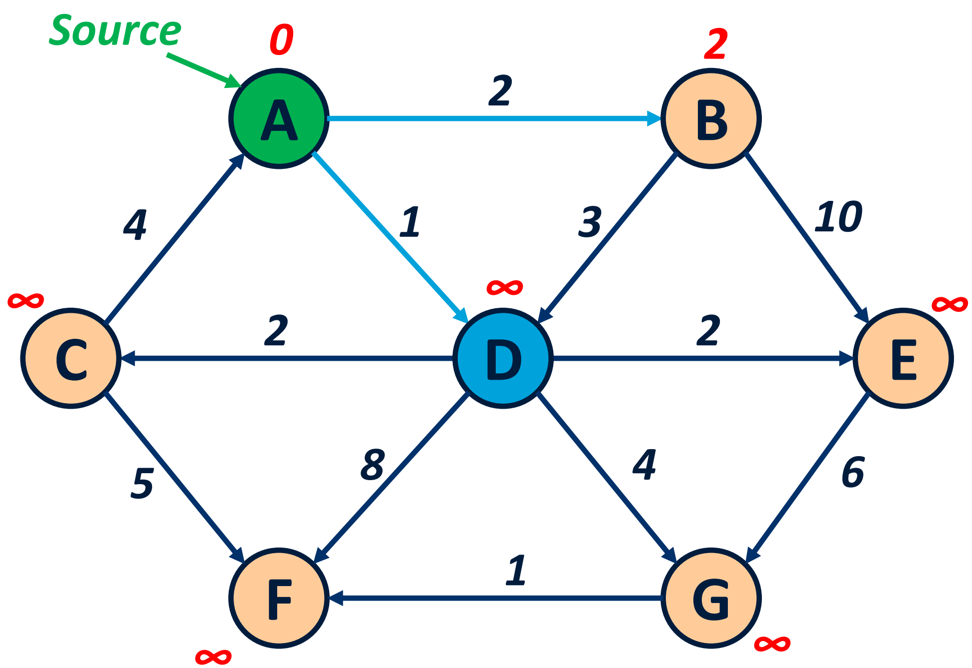



**Dist(A) + 2 = 0 + 2 = 2 < ∞ => Update B**

# Example: Algorithm

- While Q not empty
  - Select from Q node ***v*** with minimum distance
  - Remove ***v*** from Q
  - ***Relax all outgoing edges***
    - Compute the distance of neighbors passing by ***v***
    - If this is shorter than the current, update distance and pred to ***v***

| Node | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| Dist. | 0 | 2 | ∞ | ∞ | ∞ | ∞ | ∞ |
| Prev. | - | A | - | - | - | - | - |

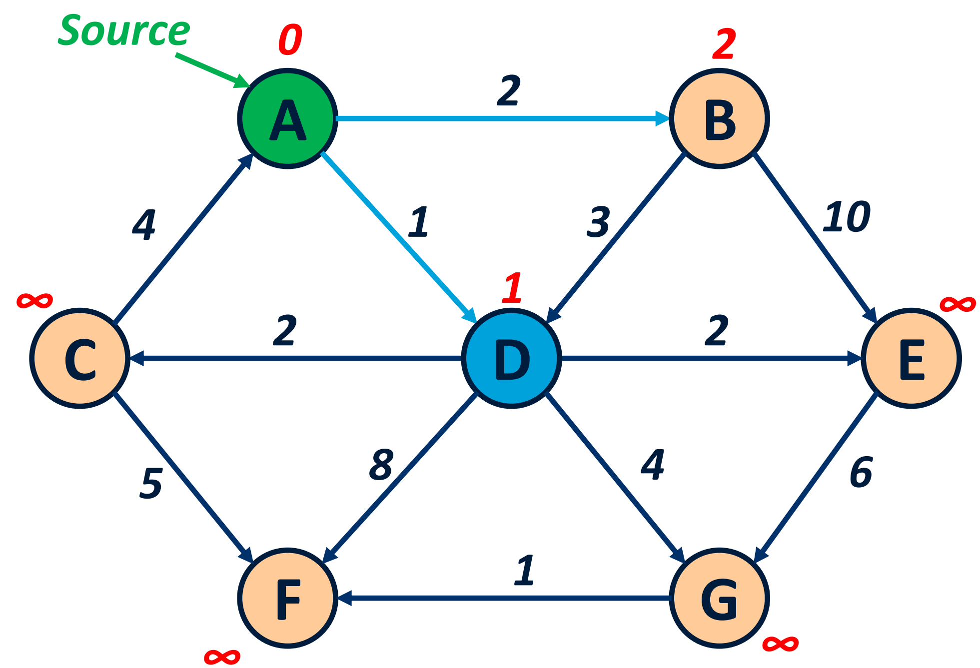



Dist(A) + 1 = 0 + 1 = 1 < ∞ => Update D

# Example: Algorithm

- While Q not empty
  - Select from Q node ***v*** with minimum distance
  - Remove ***v*** from Q
  - ***Relax all outgoing edges***
    - Compute the distance of neighbors passing by ***v***
    - If this is shorter than the current, update distance and pred to ***v***

| Node | A | B | C | D | E | F | G |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Dist. | 0 | 2 | ∞ | 1 | ∞ | ∞ | ∞ |
| Prev. | - | A | - | A | - | - | - |

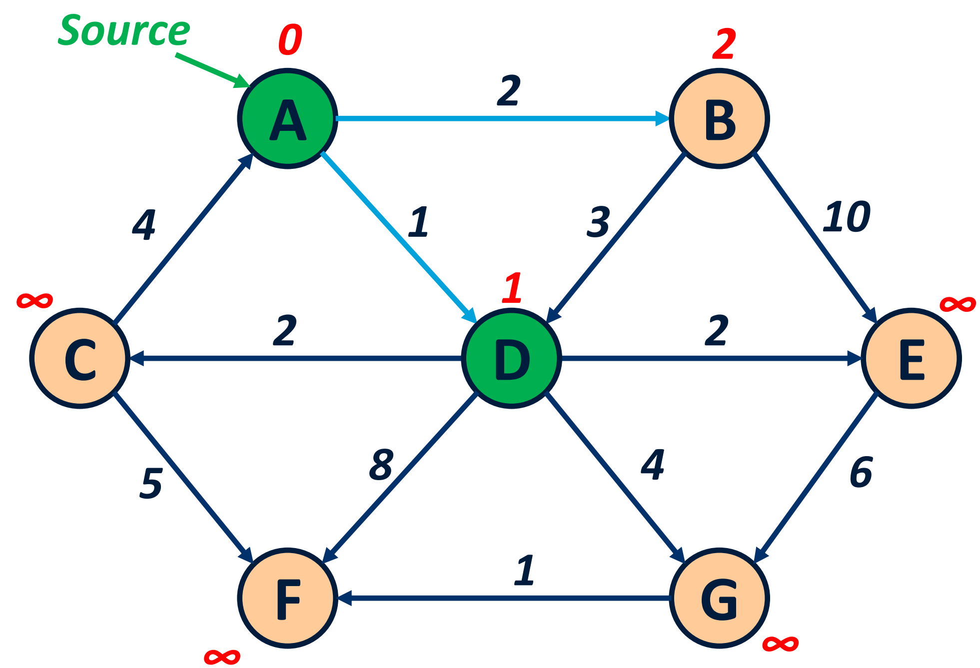



Dist(A) + 1 = 0 + 1 = 1 < ∞ => Update D

# Example: Algorithm

- While Q not empty
  - Select from Q node ***v*** with minimum distance
  - Remove ***v*** from Q
  - ***Relax all outgoing edges***
    - Compute the distance of neighbors passing by ***v***
    - If this is shorter than the current, update distance and pred to ***v***

| Node | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| Dist. | 0 | 2 | ∞ | 1 | ∞ | ∞ | ∞ |
| Prev. | - | A | - | A | - | - | - |

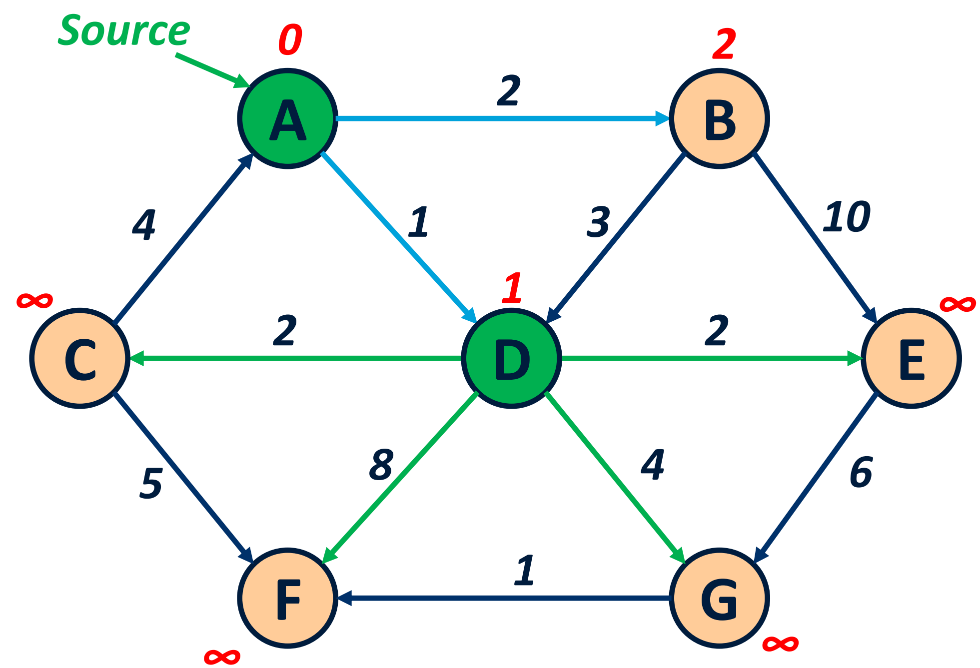



Select D

# Example: Algorithm

- While Q not empty
  - Select from Q node ***v*** with minimum distance
  - Remove ***v*** from Q
  - ***Relax all outgoing edges***
    - Compute the distance of neighbors passing by ***v***
    - If this is shorter than the current, update distance and pred to ***v***

| Node | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| Dist. | 0 | 2 | ∞ | 1 | ∞ | ∞ | ∞ |
| Prev. | - | A | - | A | - | - | - |

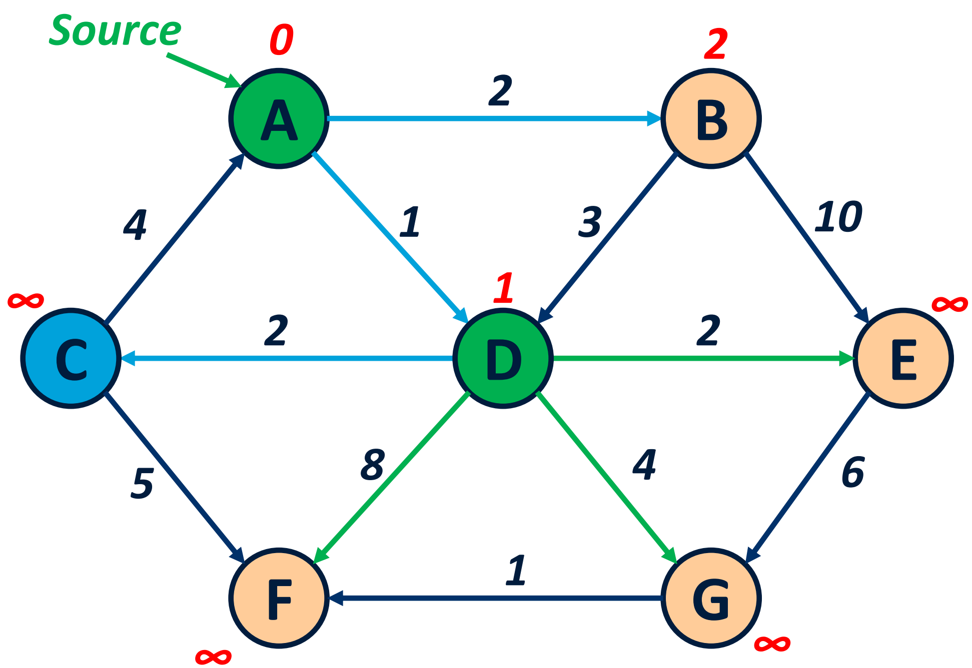



Relax outgoing edges

# Example: Algorithm

- While Q not empty
  - Select from Q node ***v*** with minimum distance
  - Remove ***v*** from Q
  - ***Relax all outgoing edges***
    - Compute the distance of neighbors passing by ***v***
    - If this is shorter than the current, update distance and pred to ***v***

| Node | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| Dist. | 0 | 2 | ∞ | 1 | ∞ | ∞ | ∞ |
| Prev. | - | A | - | A | - | - | - |

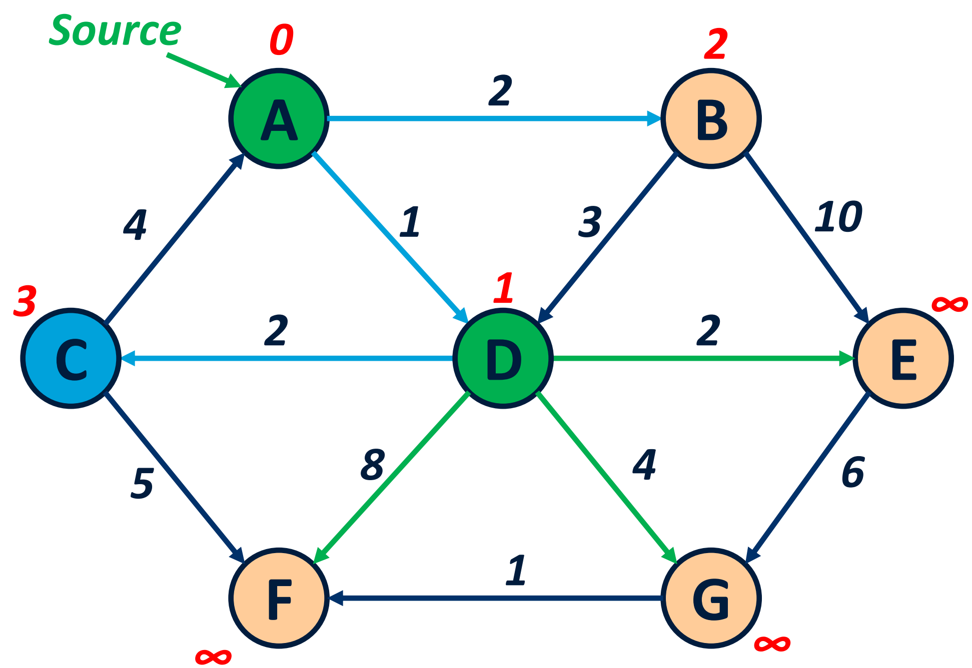



Dist(D) + 2 = 1 + 2 = 3 < ∞ => Update C

# Example: Algorithm

- While Q not empty
  - Select from Q node ***v*** with minimum distance
  - Remove ***v*** from Q
  - ***Relax all outgoing edges***
    - Compute the distance of neighbors passing by ***v***
    - If this is shorter than the current, update distance and pred to ***v***

| Node | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| Dist. | 0 | 2 | 3 | 1 | ∞ | ∞ | ∞ |
| Prev. | - | A | D | A | - | - | - |

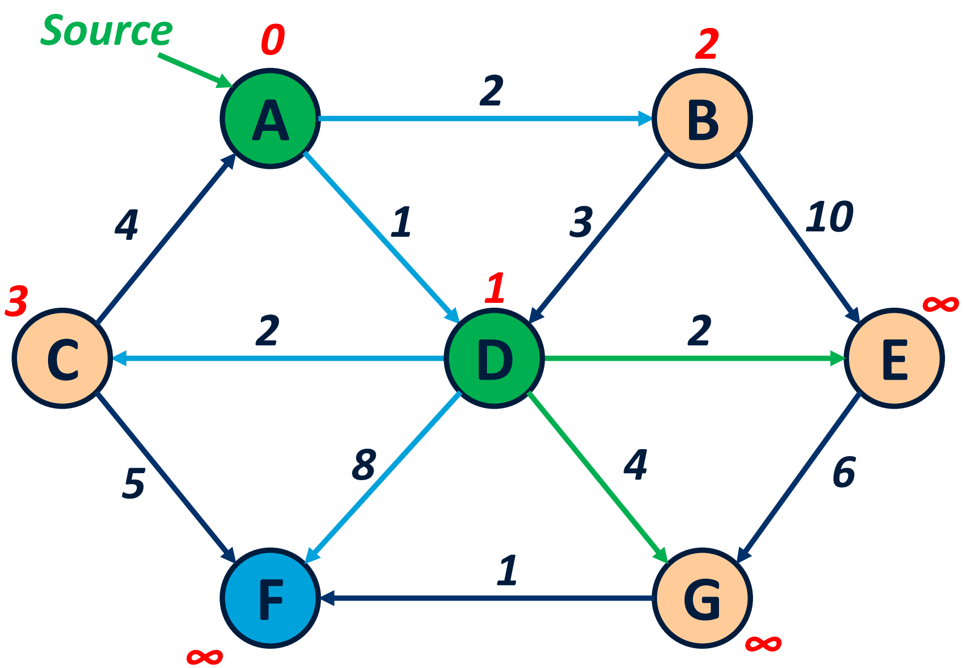



**Dist(D) + 2 = 1 + 2 = 3 < ∞ => Update C**

# Example: Algorithm

- While Q not empty
  - Select from Q node ***v*** with minimum distance
  - Remove ***v*** from Q
  - ***Relax all outgoing edges***
    - Compute the distance of neighbors passing by ***v***
    - If this is shorter than the current, update distance and pred to ***v***

| Node | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| Dist. | 0 | 2 | 3 | 1 | ∞ | ∞ | ∞ |
| Prev. | - | A | D | A | - | - | - |

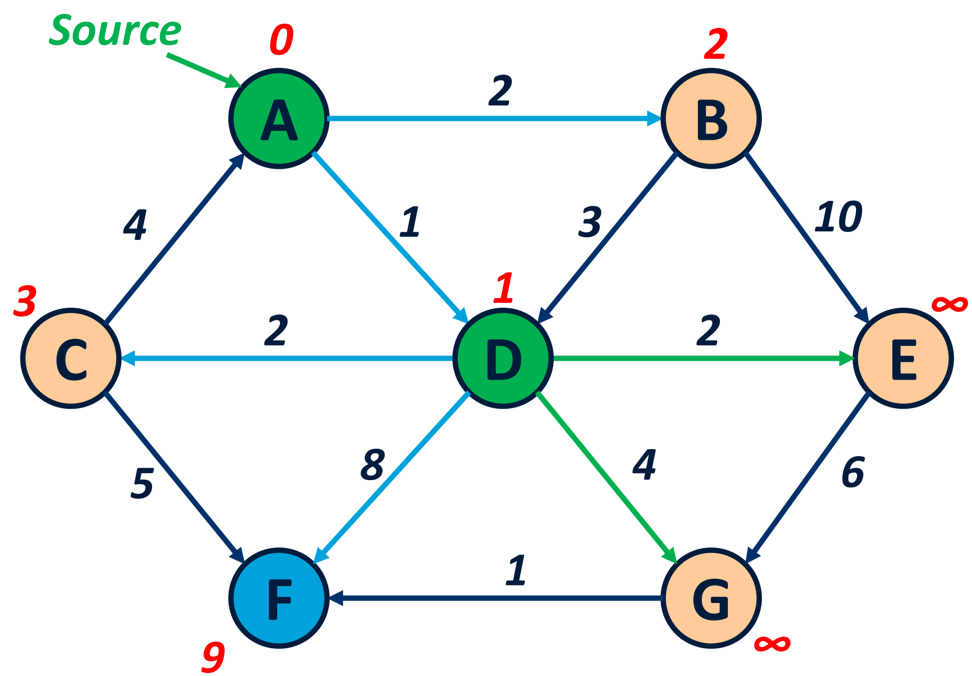



Dist(D) + 8 = 1 + 8 = 9 < ∞ => Update F

# Example: Algorithm

- While Q not empty
  - Select from Q node ***v*** with minimum distance
  - Remove ***v*** from Q
  - ***Relax all outgoing edges***
    - Compute the distance of neighbors passing by ***v***
    - If this is shorter than the current, update distance and pred to ***v***

| Node | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| Dist. | 0 | 2 | 3 | 1 | ∞ | 9 | ∞ |
| Prev. | - | A | D | A | - | D | - |

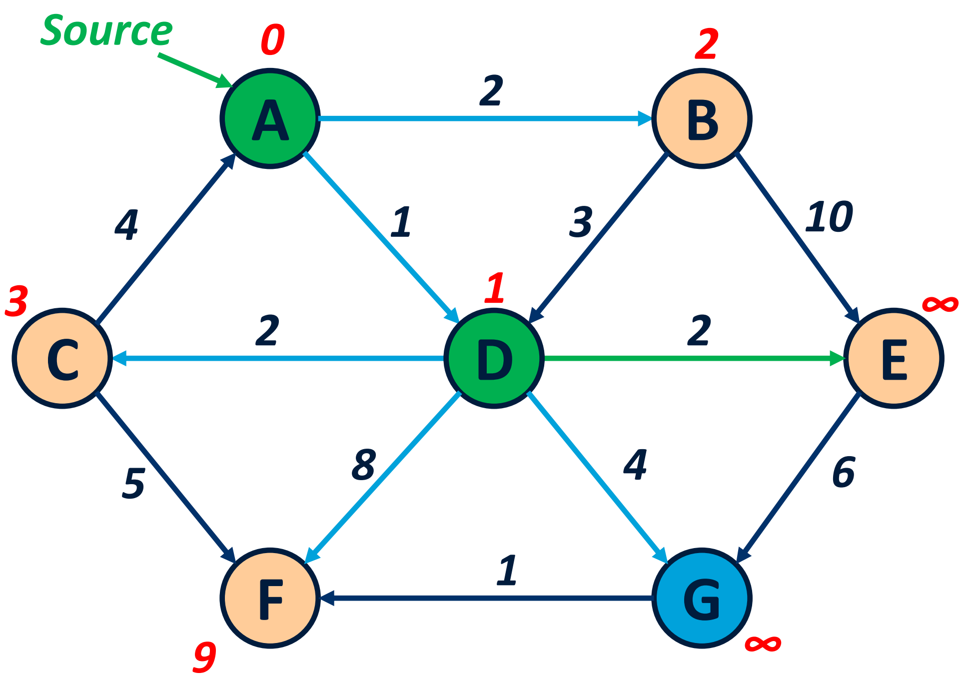



**Dist(D) + 8 = 1 + 8 = 9 < ∞ => Update F**

# Example: Algorithm

- While Q not empty
  - Select from Q node ***v*** with minimum distance
  - Remove ***v*** from Q
  - ***Relax all outgoing edges***
    - Compute the distance of neighbors passing by ***v***
    - If this is shorter than the current, update distance and pred to ***v***

| Node | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| Dist. | 0 | 2 | 3 | 1 | ∞ | 9 | ∞ |
| Prev. | - | A | D | A | - | D | - |

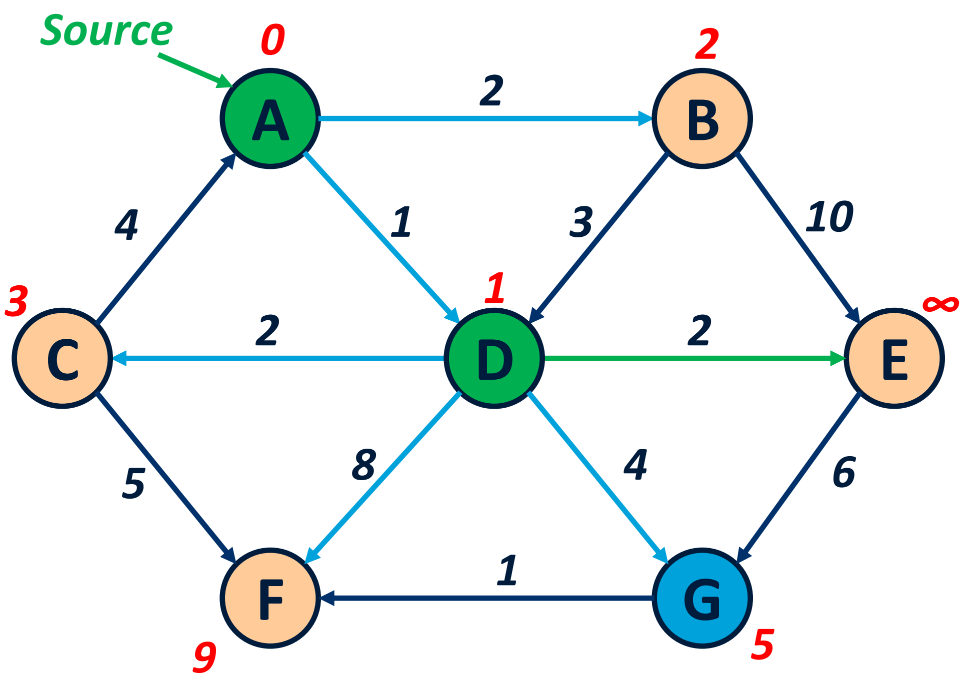



Dist(D) + 4 = 1 + 4 = 5 < ∞ => Update G

# Example: Algorithm

- While Q not empty
  - Select from Q node ***v*** with minimum distance
  - Remove ***v*** from Q
  - ***Relax all outgoing edges***
    - Compute the distance of neighbors passing by ***v***
    - If this is shorter than the current, update distance and pred to ***v***

| Node | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| Dist. | 0 | 2 | 3 | 1 | ∞ | 9 | 5 |
| Prev. | - | A | D | A | - | D | D |

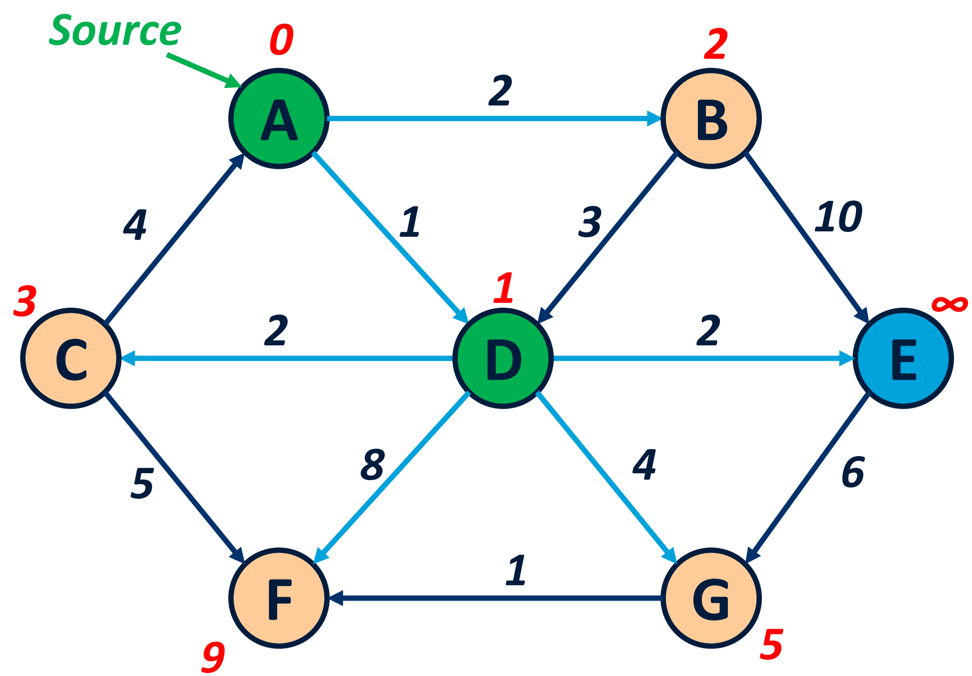



Dist(D) + 4 = 1 + 4 = 5 < ∞ => Update G

# Example: Algorithm

- While Q not empty
  - Select from Q node ***v*** with minimum distance
  - Remove ***v*** from Q
  - ***Relax all outgoing edges***
    - Compute the distance of neighbors passing by ***v***
    - If this is shorter than the current, update distance and pred to ***v***

| Node | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| Dist. | 0 | 2 | 3 | 1 | ∞ | 9 | 5 |
| Prev. | - | A | D | A | - | D | D |

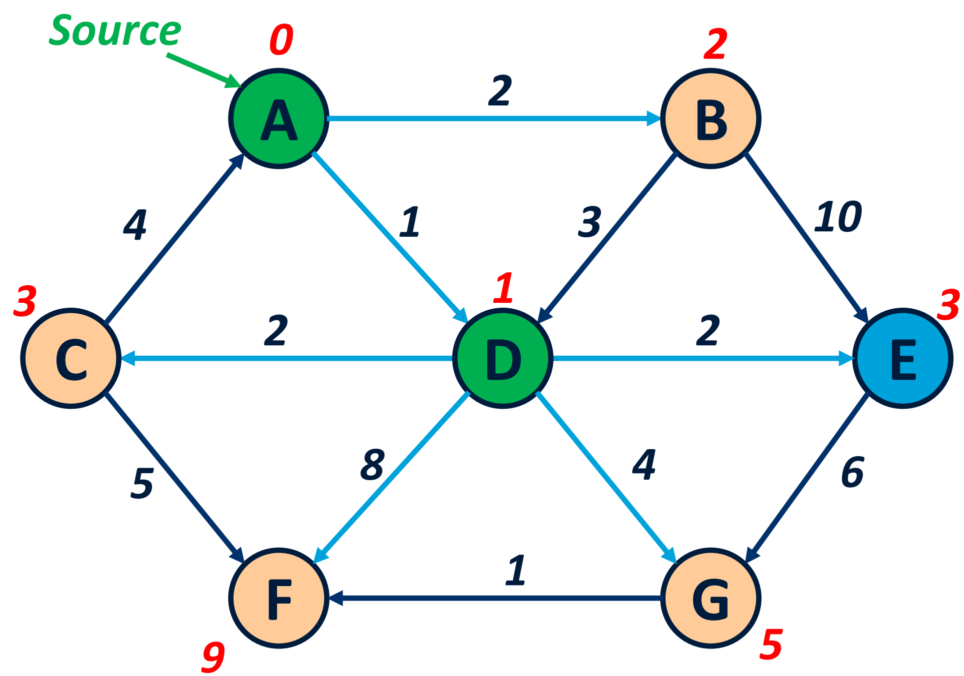



**Dist(D) + 2 = 1 + 2 = 3 < ∞ => Update E**

# Example: Algorithm

- While Q not empty
  - Select from Q node ***v*** with minimum distance
  - Remove ***v*** from Q
  - ***Relax all outgoing edges***
    - Compute the distance of neighbors passing by ***v***
    - If this is shorter than the current, update distance and pred to ***v***

| Node | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| Dist. | 0 | 2 | 3 | 1 | 3 | 9 | 5 |
| Prev. | - | A | D | A | D | D | D |

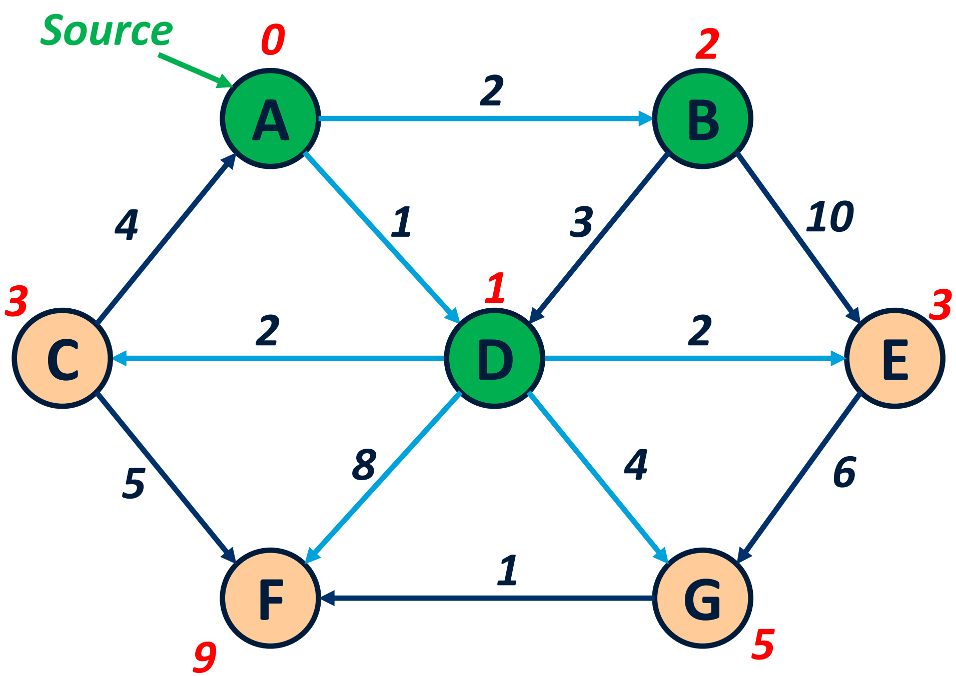



Dist(D) + 2 = 1 + 2 = 3 < ∞ => Update E

# Example: Algorithm

- While Q not empty
  - Select from Q node ***v*** with minimum distance
  - Remove ***v*** from Q
  - ***Relax all outgoing edges***
    - Compute the distance of neighbors passing by ***v***
    - If this is shorter than the current, update distance and pred to ***v***

| Node | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| Dist. | 0 | 2 | 3 | 1 | 3 | 9 | 5 |
| Prev. | - | A | D | A | D | D | D |

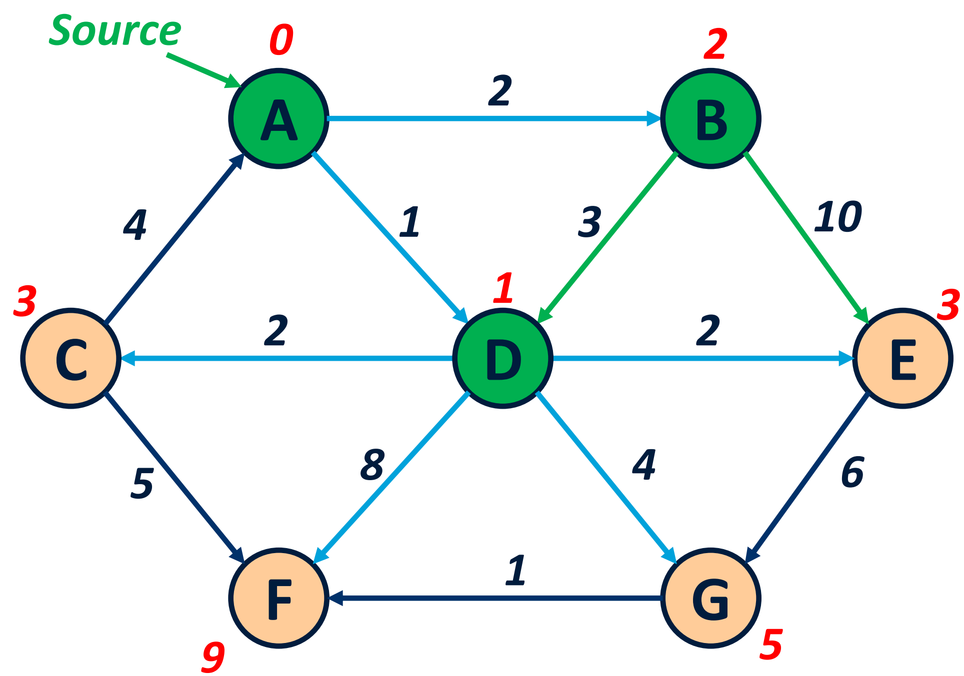



Select B

# Example: Algorithm

- While Q not empty
  - Select from Q node ***v*** with minimum distance
  - Remove ***v*** from Q
  - ***Relax all outgoing edges***
    - Compute the distance of neighbors passing by ***v***
    - If this is shorter than the current, update distance and pred to ***v***

| Node | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| **Dist.** | 0 | 2 | 3 | 1 | 3 | 9 | 5 |
| **Prev.** | - | A | D | A | D | D | D |

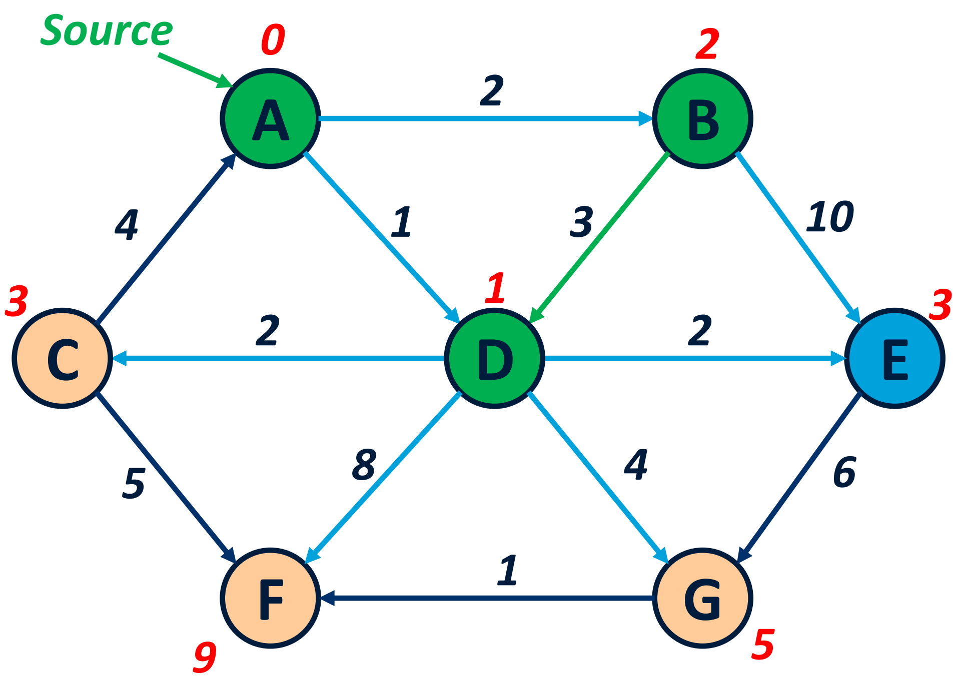



Relax outgoing edges

# Example: Algorithm

- While Q not empty
  - Select from Q node ***v*** with minimum distance
  - Remove ***v*** from Q
  - ***Relax all outgoing edges***
    - Compute the distance of neighbors passing by ***v***
    - If this is shorter than the current, update distance and pred to ***v***

| Node | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| Dist. | 0 | 2 | 3 | 1 | 3 | 9 | 5 |
| Prev. | - | A | D | A | D | D | D |

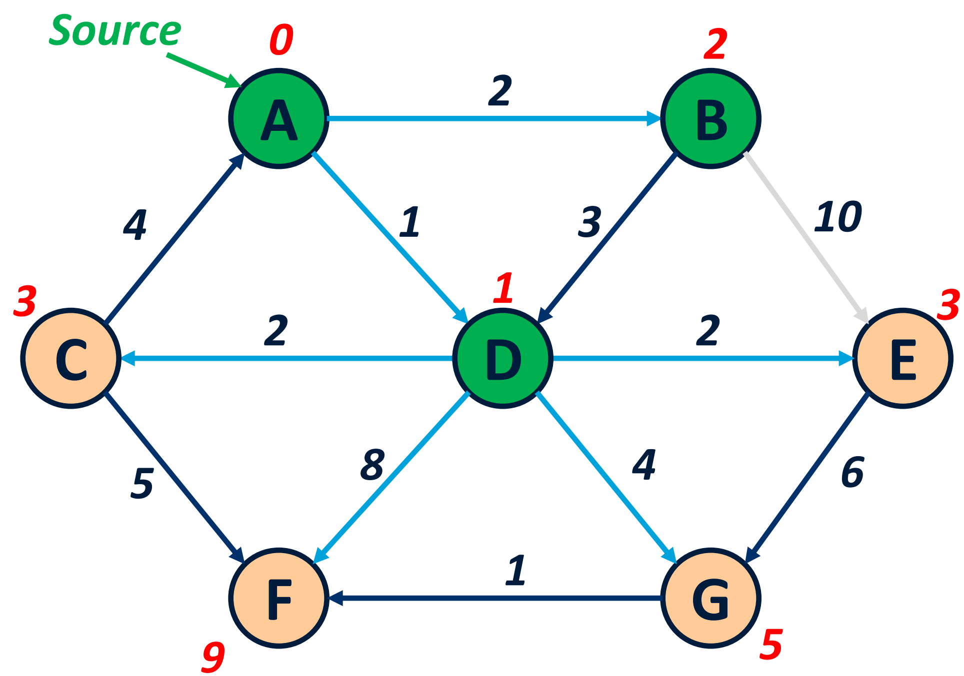



**Dist(B) + 10 = 2 + 10 = 12 > 3 => Not update E**

# Example: Algorithm

- While Q not empty
  - Select from Q node ***v*** with minimum distance
  - Remove ***v*** from Q
  - ***Relax all outgoing edges***
    - Compute the distance of neighbors passing by ***v***
    - If this is shorter than the current, update distance and pred to ***v***

| **Node** | **A** | **B** | **C** | **D** | **E** | **F** | **G** |
|---|---|---|---|---|---|---|---|
| **Dist.** | 0 | 2 | 3 | 1 | 3 | 9 | 5 |
| **Prev.** | - | A | D | A | D | D | D |

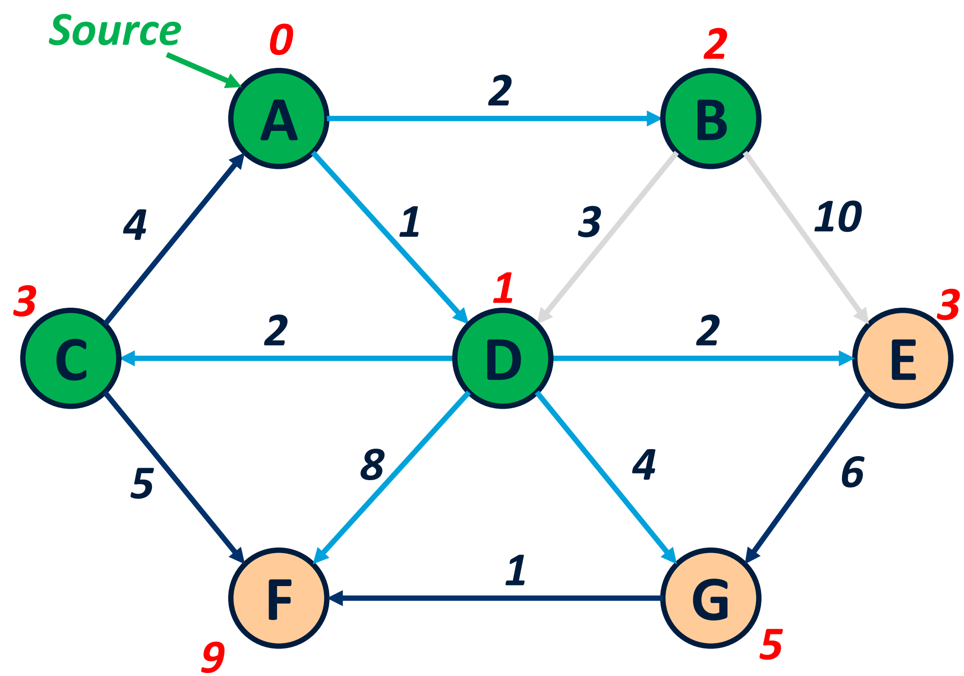



**D visited => we cannot find a shortest path**

# Example: Algorithm

- While Q not empty
  - Select from Q node ***v*** with minimum distance
  - Remove ***v*** from Q
  - ***Relax all outgoing edges***
    - Compute the distance of neighbors passing by ***v***
    - If this is shorter than the current, update distance and pred to ***v***

| Node | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| **Dist.** | 0 | 2 | 3 | 1 | 3 | 9 | 5 |
| **Prev.** | - | A | D | A | D | D | D |

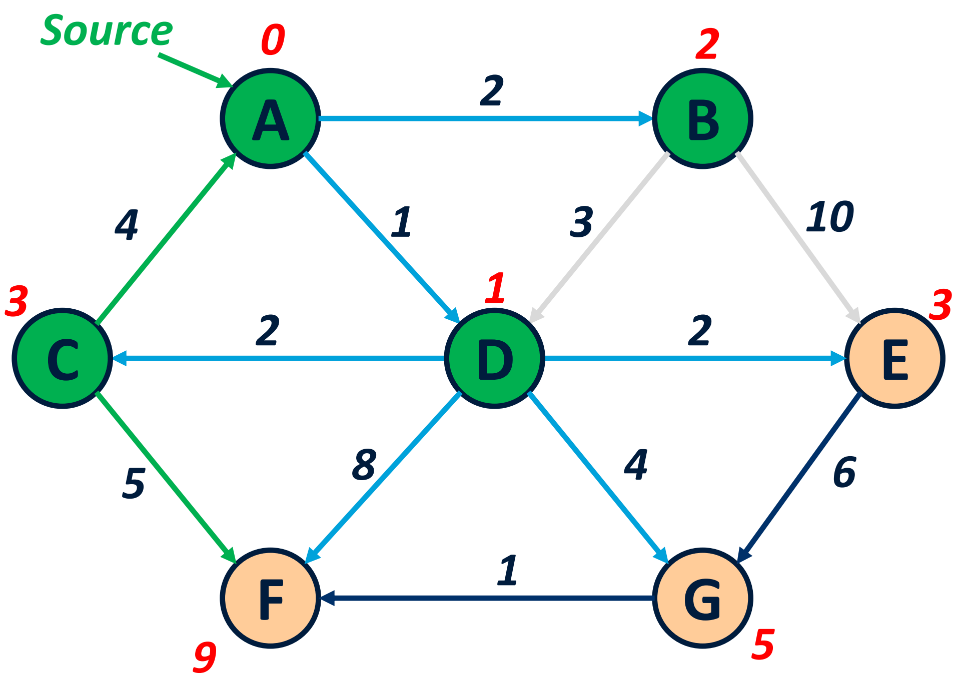



**Select C**

# Example: Algorithm

- While Q not empty
  - Select from Q node ***v*** with minimum distance
  - Remove ***v*** from Q
  - ***Relax all outgoing edges***
    - Compute the distance of neighbors passing by ***v***
    - If this is shorter than the current, update distance and pred to ***v***

| Node | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| Dist. | 0 | 2 | 3 | 1 | 3 | 9 | 5 |
| Prev. | - | A | D | A | D | D | D |

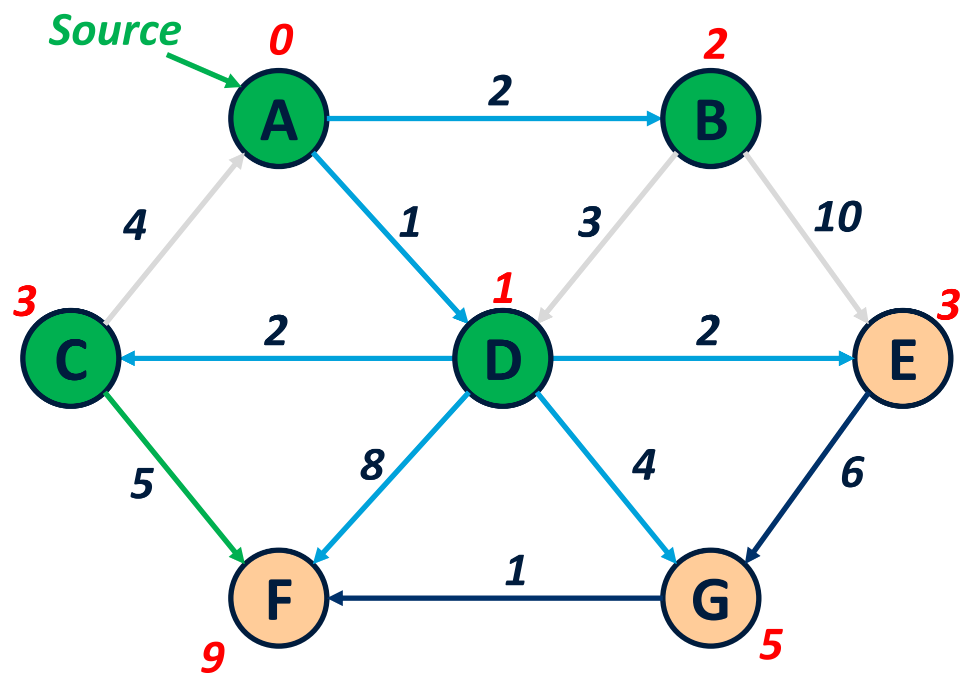



Relax outgoing edges

# Example: Algorithm

- While Q not empty
  - Select from Q node ***v*** with minimum distance
  - Remove ***v*** from Q
  - ***Relax all outgoing edges***
    - Compute the distance of neighbors passing by ***v***
    - If this is shorter than the current, update distance and pred to ***v***

| **Node** | **A** | **B** | **C** | **D** | **E** | **F** | **G** |
|---|---|---|---|---|---|---|---|
| **Dist.** | 0 | 2 | 3 | 1 | 3 | 9 | 5 |
| **Prev.** | - | A | D | A | D | D | D |

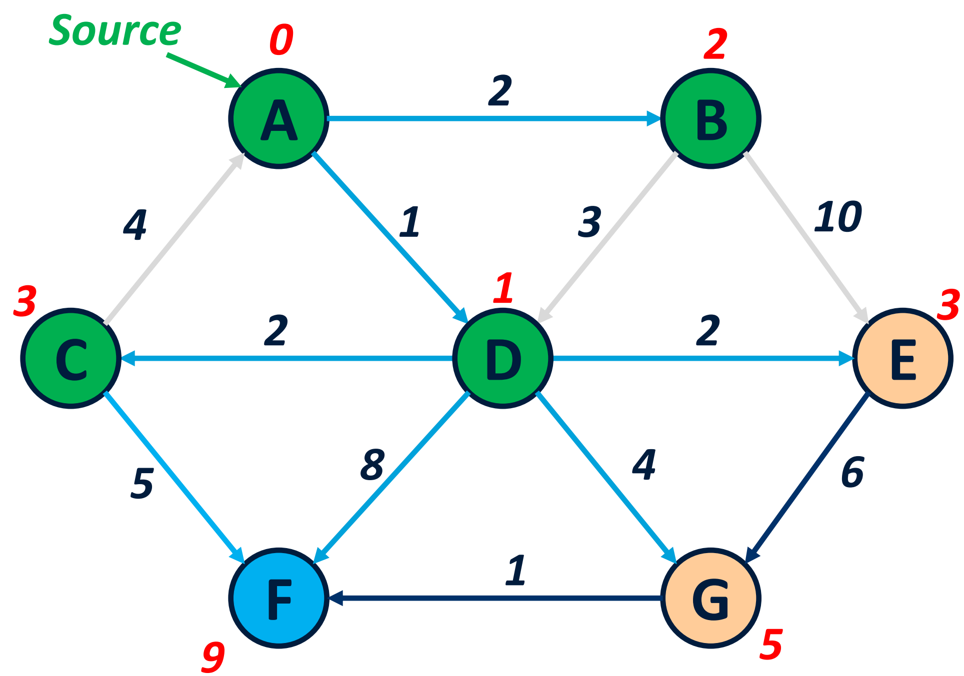



**A visited => we cannot find a shortest path**

# Example: Algorithm

- While Q not empty
  - Select from Q node ***v*** with minimum distance
  - Remove ***v*** from Q
  - ***Relax all outgoing edges***
    - Compute the distance of neighbors passing by ***v***
    - If this is shorter than the current, update distance and pred to ***v***

| Node | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| Dist. | 0 | 2 | 3 | 1 | 3 | 9 | 5 |
| Prev. | - | A | D | A | D | D | D |

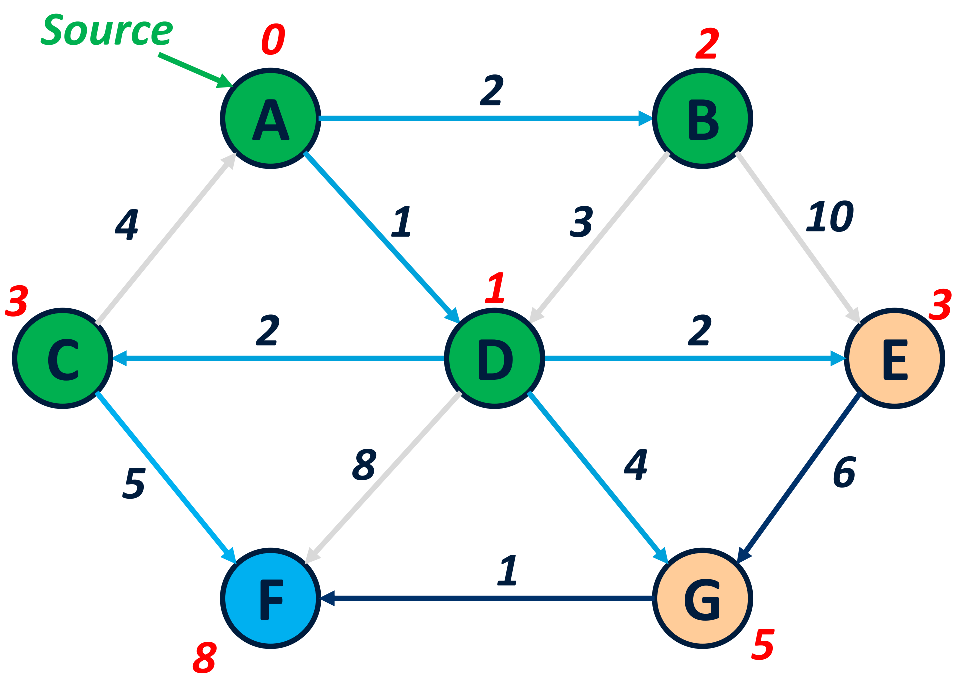



Dist(C) + 5 = 3 + 5 = 8 < 9 => Update F

# Example: Algorithm

- While Q not empty
  - Select from Q node ***v*** with minimum distance
  - Remove ***v*** from Q
  - ***Relax all outgoing edges***
    - Compute the distance of neighbors passing by ***v***
    - If this is shorter than the current, update distance and pred to ***v***

| Node | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| **Dist.** | 0 | 2 | 3 | 1 | 3 | 8 | 5 |
| **Prev.** | - | A | D | A | D | C | D |

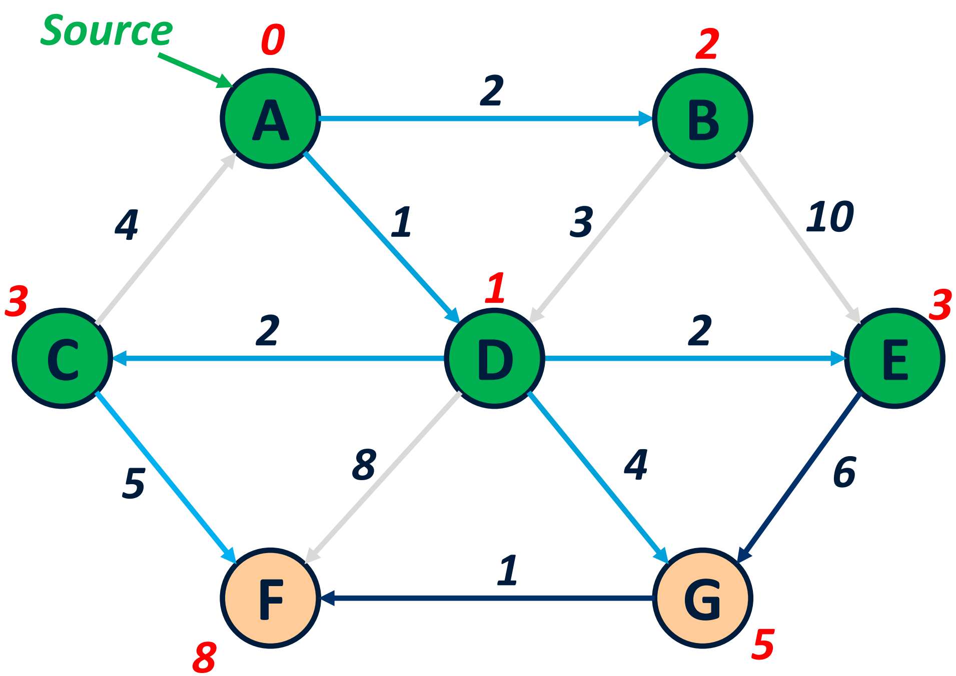



**Dist(C) + 5 = 3 + 5 = 8 < 9 => Update F**

# Example: Algorithm

- While Q not empty
  - Select from Q node ***v*** with minimum distance
  - Remove ***v*** from Q
  - ***Relax all outgoing edges***
    - Compute the distance of neighbors passing by ***v***
    - If this is shorter than the current, update distance and pred to ***v***

| Node | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| Dist. | 0 | 2 | 3 | 1 | 3 | 8 | 5 |
| Prev. | - | A | D | A | D | C | D |

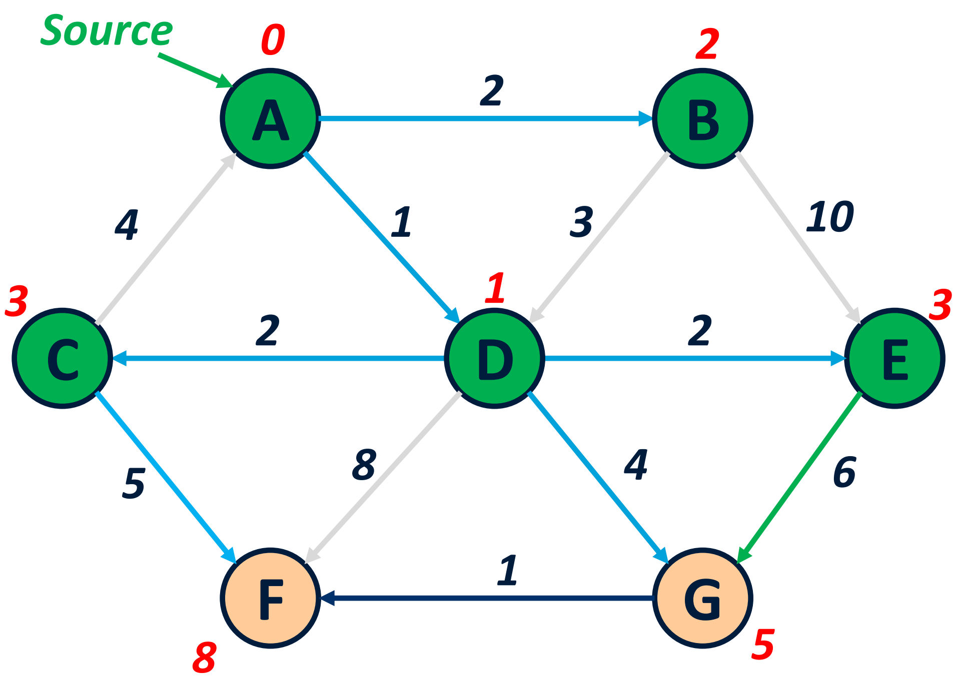



Select E

# Example: Algorithm

- While Q not empty
  - Select from Q node ***v*** with minimum distance
  - Remove ***v*** from Q
  - ***Relax all outgoing edges***
    - Compute the distance of neighbors passing by ***v***
    - If this is shorter than the current, update distance and pred to ***v***

| Node | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| Dist. | 0 | 2 | 3 | 1 | 3 | 8 | 5 |
| Prev. | - | A | D | A | D | C | D |

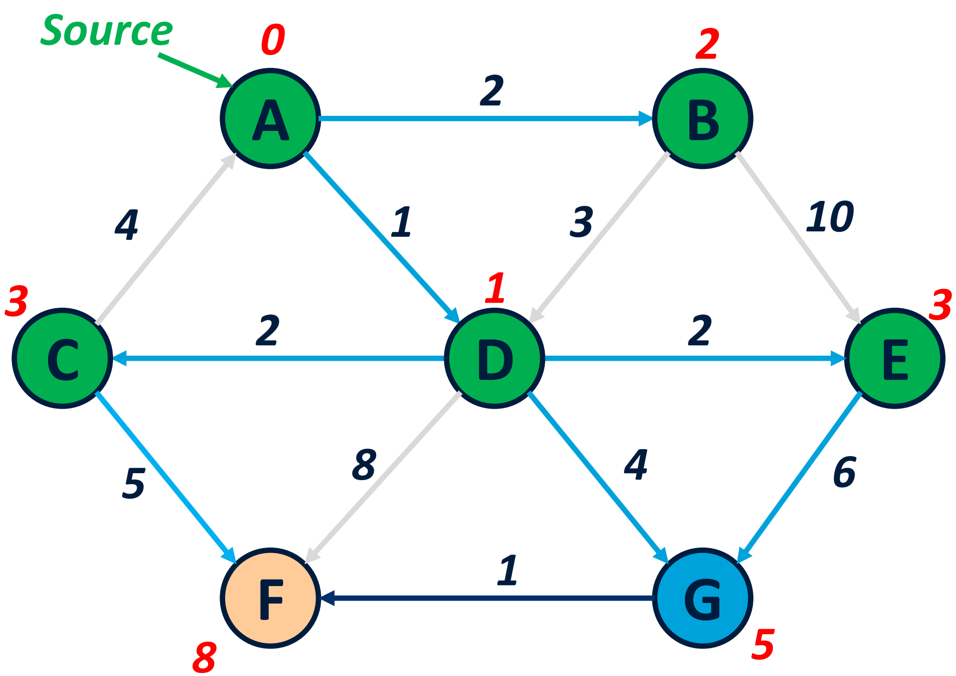



**Relax outgoing edges**

# Example: Algorithm

- While Q not empty
  - Select from Q node ***v*** with minimum distance
  - Remove ***v*** from Q
  - ***Relax all outgoing edges***
    - Compute the distance of neighbors passing by ***v***
    - If this is shorter than the current, update distance and pred to ***v***

| Node | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| **Dist.** | 0 | 2 | 3 | 1 | 3 | 8 | 5 |
| **Prev.** | - | A | D | A | D | C | D |

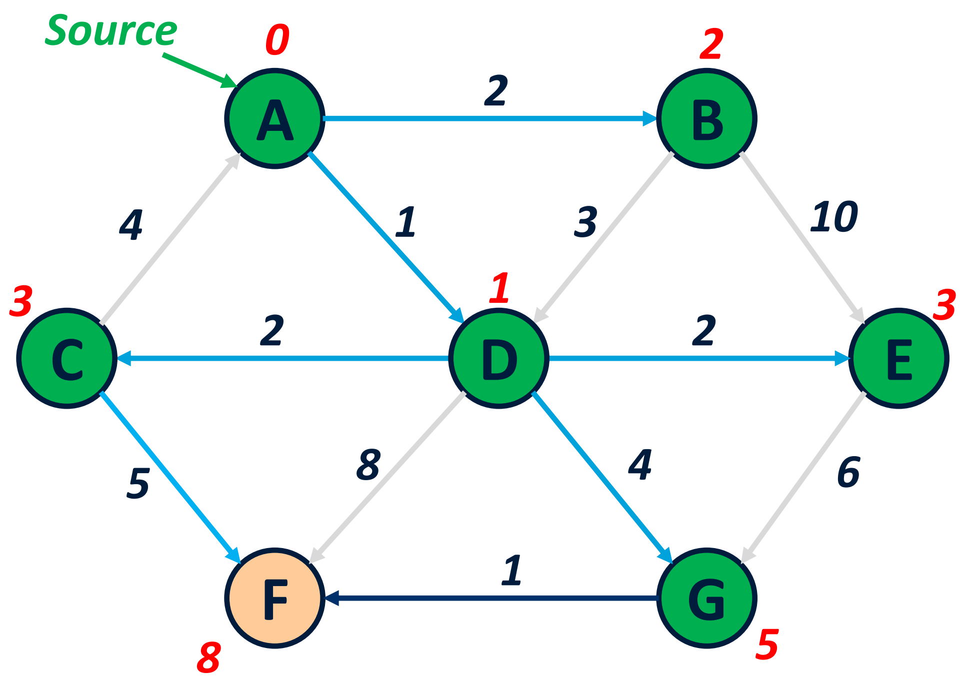



**Dist(E) + 6 = 3 + 6 = 9 > 5 => Don't update G**

# Example: Algorithm

- While Q not empty
  - Select from Q node ***v*** with minimum distance
  - Remove ***v*** from Q
  - ***Relax all outgoing edges***
    - Compute the distance of neighbors passing by ***v***
    - If this is shorter than the current, update distance and pred to ***v***

| Node | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| Dist. | 0 | 2 | 3 | 1 | 3 | 8 | 5 |
| Prev. | - | A | D | A | D | C | D |

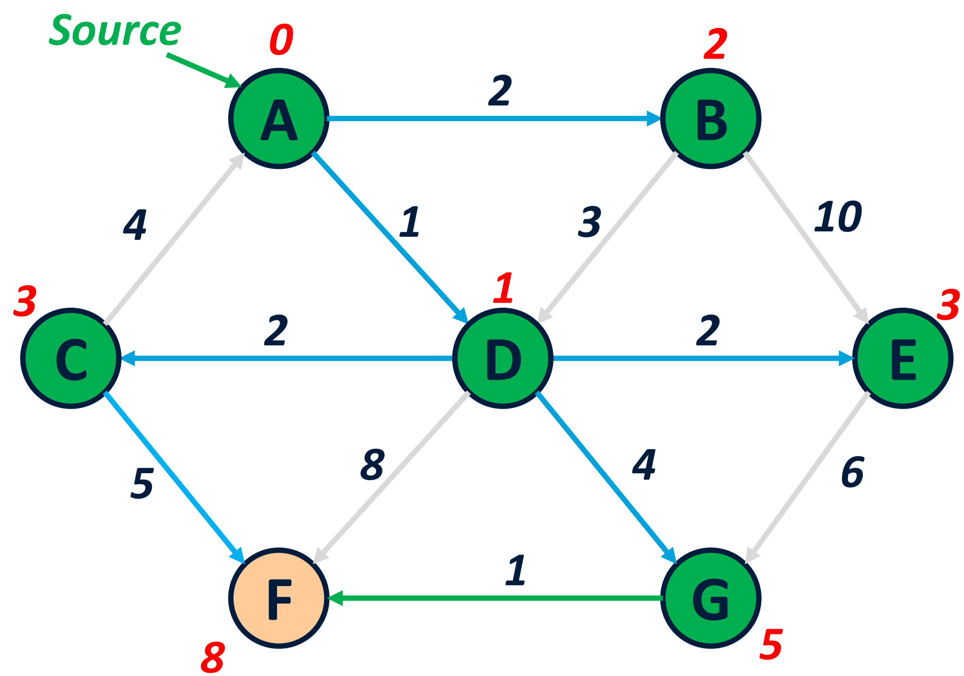



Select G

# Example: Algorithm

- While Q not empty
  - Select from Q node ***v*** with minimum distance
  - Remove ***v*** from Q
  - ***Relax all outgoing edges***
    - Compute the distance of neighbors passing by ***v***
    - If this is shorter than the current, update distance and pred to ***v***

| Node | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| Dist. | 0 | 2 | 3 | 1 | 3 | 8 | 5 |
| Prev. | - | A | D | A | D | C | D |

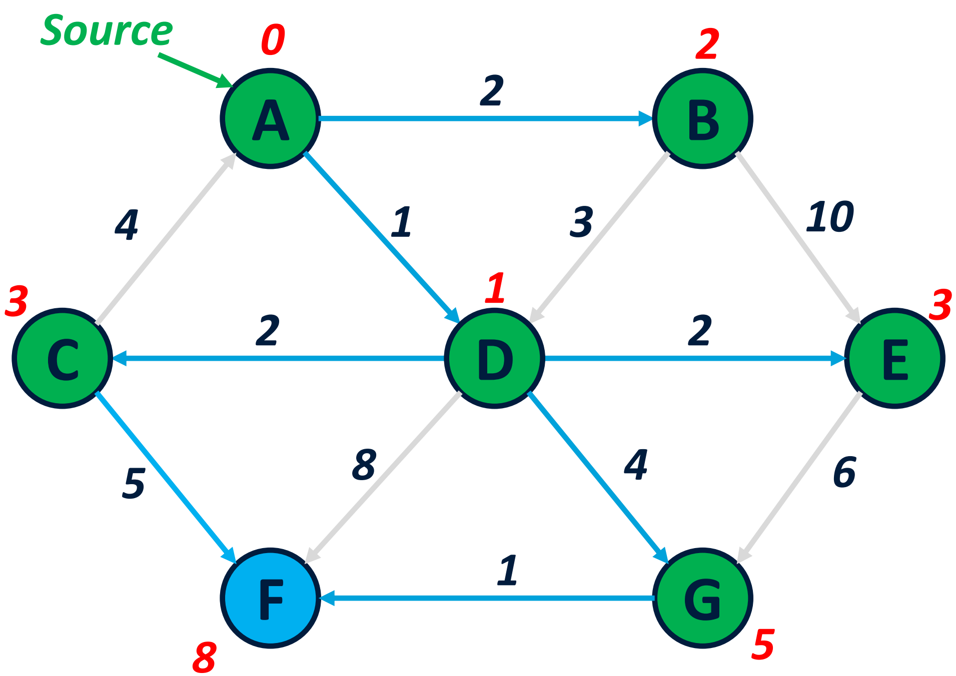



Relax outgoing edges

# Example: Algorithm

- While Q not empty
  - Select from Q node ***v*** with minimum distance
  - Remove ***v*** from Q
  - ***Relax all outgoing edges***
    - Compute the distance of neighbors passing by ***v***
    - If this is shorter than the current, update distance and pred to ***v***

| Node | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| **Dist.** | 0 | 2 | 3 | 1 | 3 | 8 | 5 |
| **Prev.** | - | A | D | A | D | C | D |

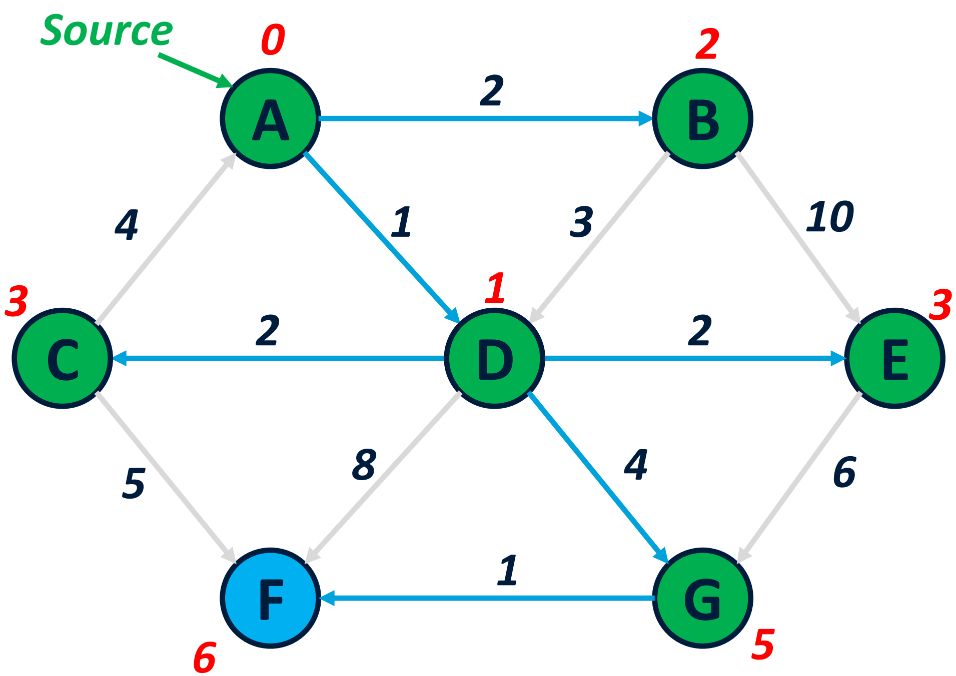



**Dist(G) + 1 = 5 + 1 = 6 < 8 => Update F**

# Example: Algorithm

- While Q not empty
  - Select from Q node ***v*** with minimum distance
  - Remove ***v*** from Q
  - ***Relax all outgoing edges***
    - Compute the distance of neighbors passing by ***v***
    - If this is shorter than the current, update distance and pred to ***v***

| Node | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| Dist. | 0 | 2 | 3 | 1 | 3 | 6 | 5 |
| Prev. | - | A | D | A | D | G | D |

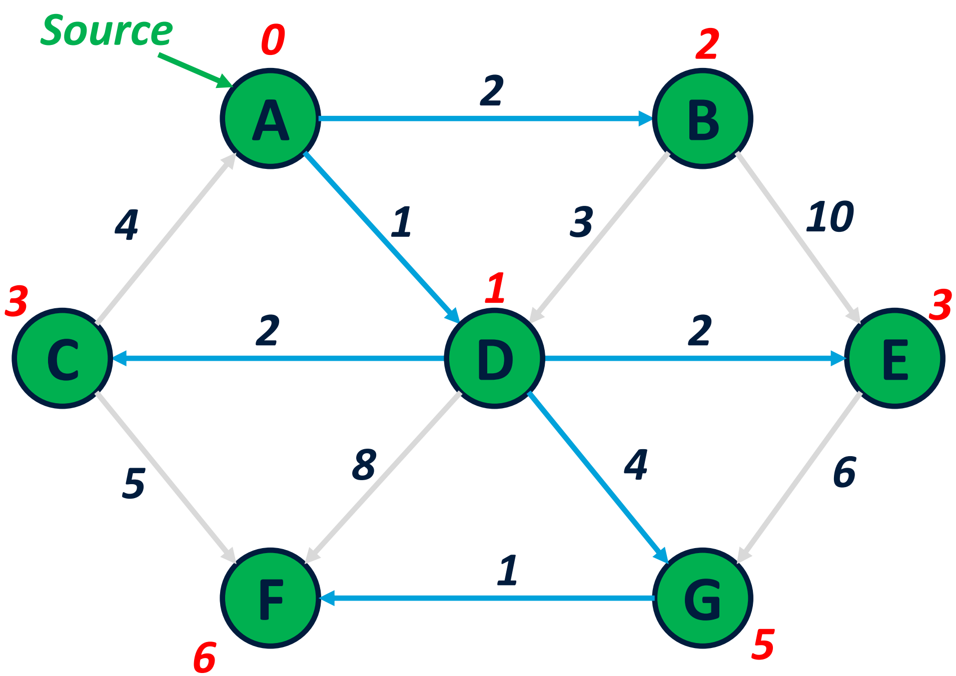



**Dist(G) + 1 = 5 + 1 = 6 < 8 => Update F**

# Example: Algorithm

- While Q not empty
  - Select from Q node ***v*** with minimum distance
  - Remove ***v*** from Q
  - ***Relax all outgoing edges***
    - Compute the distance of neighbors passing by ***v***
    - If this is shorter than the current, update distance and pred to ***v***

| Node | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| Dist. | 0 | 2 | 3 | 1 | 3 | 6 | 5 |
| Prev. | - | A | D | A | D | G | D |



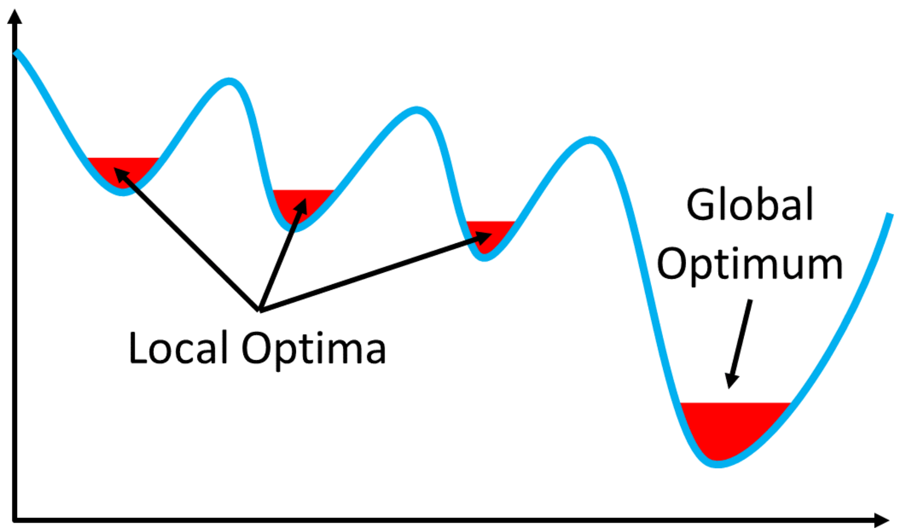

**Select F – No outgoing edges**

# Dijkstra's Algorithm - CODE

```
Dijkstra(Graph, source)

    // Initialization
    create vertex set Q
    for each vertex v in Graph
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q

    dist[source] ← 0

// Algorithm
while Q is not empty:
    u ← vertex in Q with min dist[u]
    remove u from Q

    for each neighbor v of u
        // only v that are still in Q

        alt ← dist[u] + length(u, v)
        if alt < dist[v]
            dist[v] ← alt
            prev[v] ← u

return dist[], prev[]
```

# Dijkstra's Algorithm

- Dijkstra's algorithm is a greedy algorithm
  - Make choices that currently seem the best
- Locally optimal does not always mean globally optimal



# Dijkstra’s Algorithm

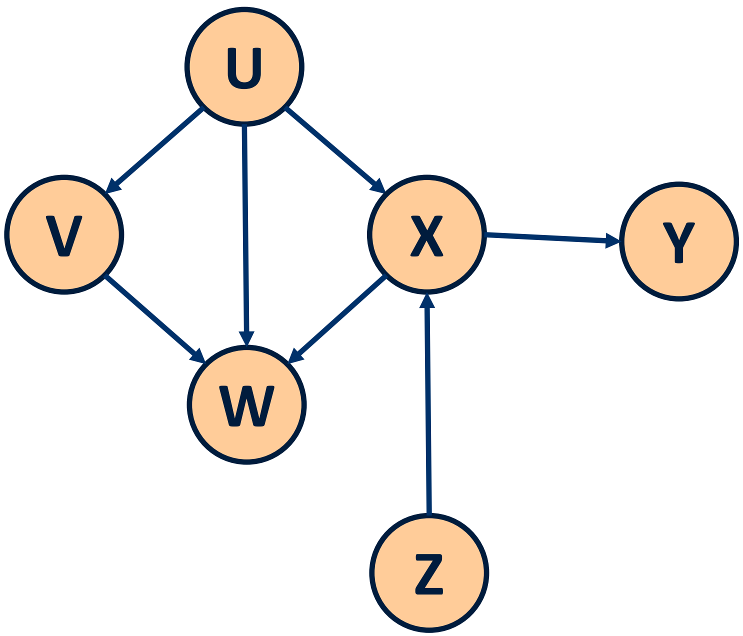
- Dijkstra’s algorithm is correct because:
  - For every known vertex, recorded distance is shortest distance to that vertex from source vertex
  - For every unknown vertex v, its recorded distance is shortest path distance to v from source vertex, considering only currently known vertices and v
  -
- Don’t forget that **having no negative-weighted edges is a requirement**

# Topological Sort

# Directed Acyclic Graphs (DAG)

- A **directed acyclic graph** (DAG) is a digraph without cycles
- Given a DAG, a **Topological Sort** outputs all vertices in an order so that no vertex appears before another vertex that points to it
- More formally, if there is a path from ***v*** to ***w***, than ***v*** appears before ***w*** in the topological sort

# Topological Sort




## Topological Sorts

U – V – Z – X – W – Y

Z – U – V – X – W – Y

U – Z – V – X – W – Y

# Topological Sort

- Why do we perform topological sorts only on DAGs?
  - If there is a cycle, there are no topological sorts

- Is there always a unique answer?
  - No, there can more topological sorts

- Do some DAGs have exactly 1 answer?
  - Yes

# Topological Sort

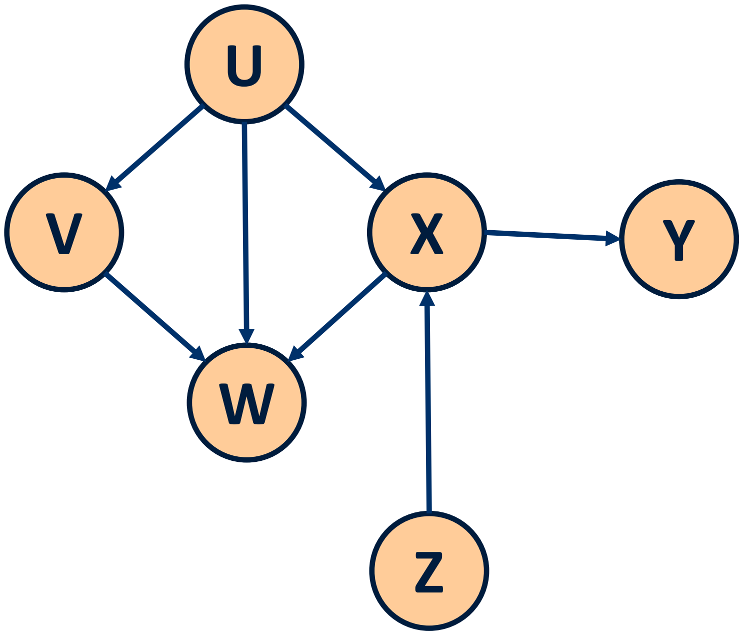
- Algorithm:
  - Compute in-degree of each node
  - Use queue to keep the topological sort
  - Until the graph has verteces
    - Select a vertex with in-degree = 0
    - Enqueue it in the topological sort
    - Decrement in-degree for all neighbors
  - If we cannot find a vertex with in-degree = 0
    - No topological sorts

# Topological Sort

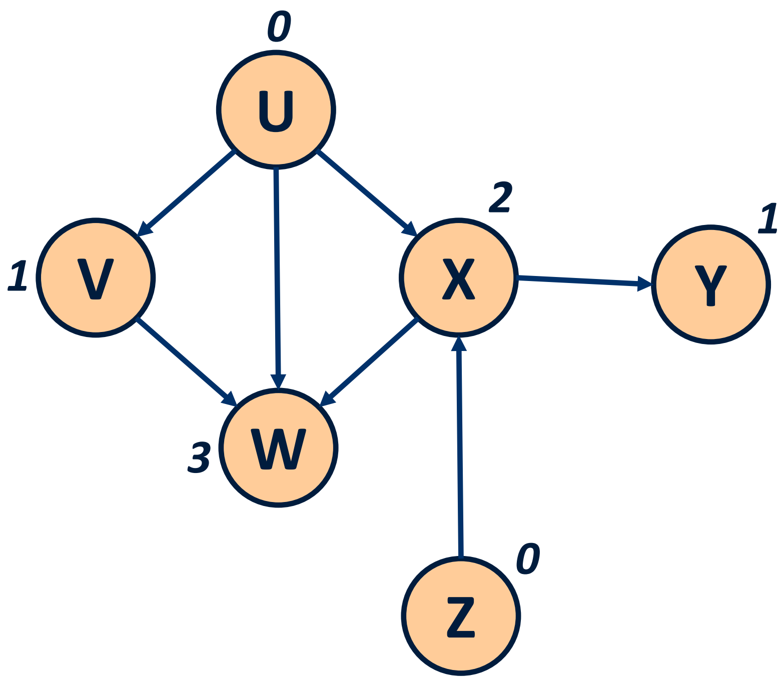
```
Topological Sort (Graph)
    create queue Q
    create list L

    for each vertex v in Graph
        compute in_degree[v]
        if in_degree[v]==0
            enqueue(v)

    while not empty(Q)
        v ← dequeue(Q)
        L.append(v)
        for each vertex w neighbor of v
            in_degree[w]--
            if in_degree[w]==0
                enqueue(w)
    return L
```

- At the end, if L does not contains all verteces
  - No topological sorts
  - The graph has cycles

# Topological Sort - Example



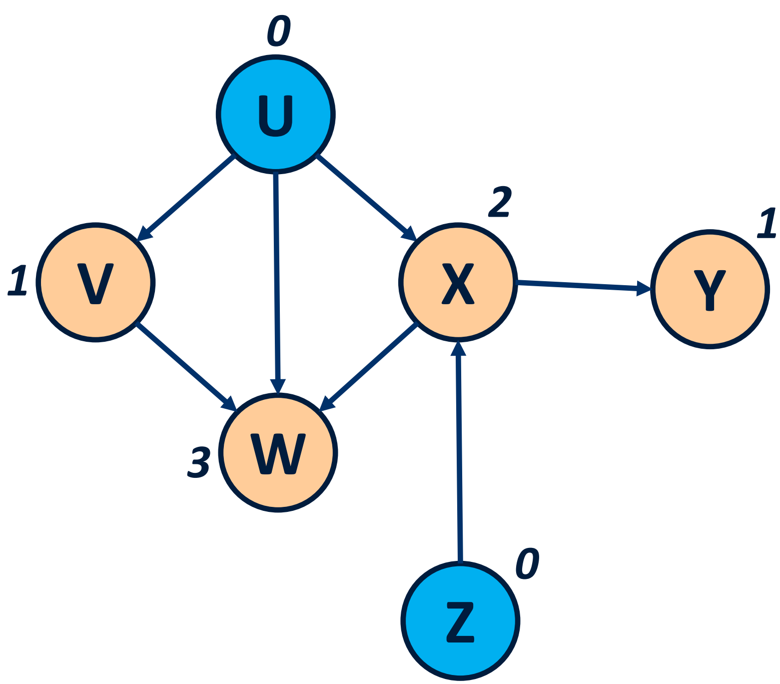
# Topological Sort - Example




- Compute **in_degree**

**Topological Sort:**

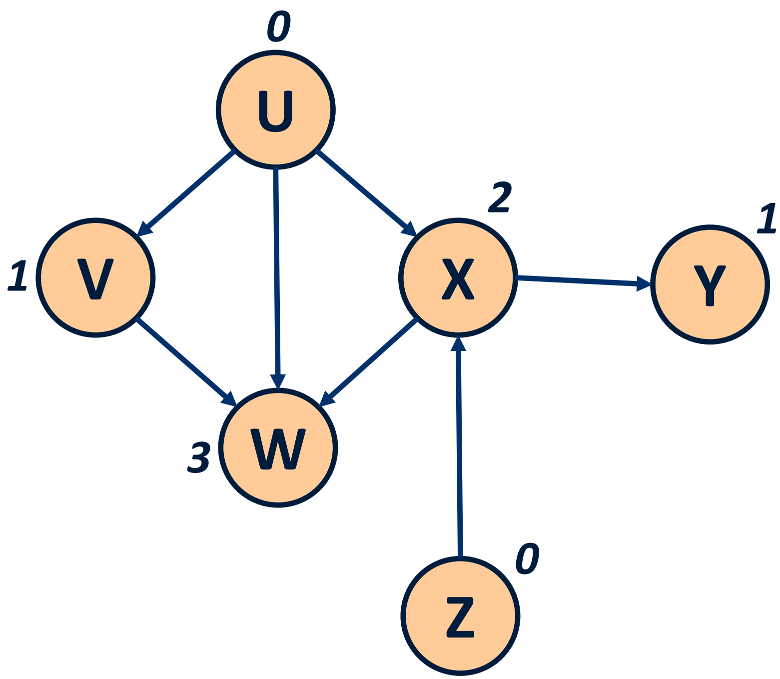
# Topological Sort - Example




- Compute **in_degree**
- Enqueue U and Z

**Topological Sort:**

# Topological Sort - Example

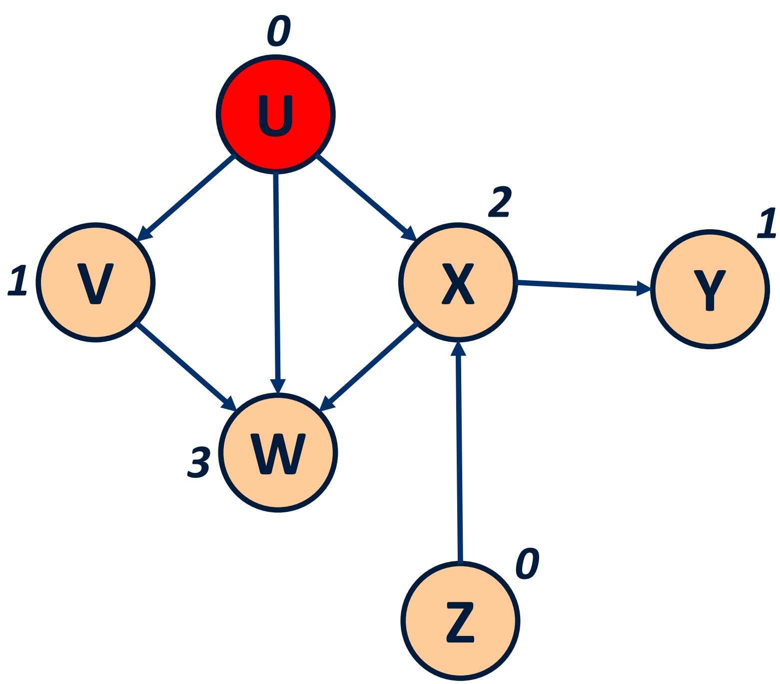



- Compute **in_degree**
- Enqueue U and Z

Q = U, Z

**Topological Sort:**

# Topological Sort - Example



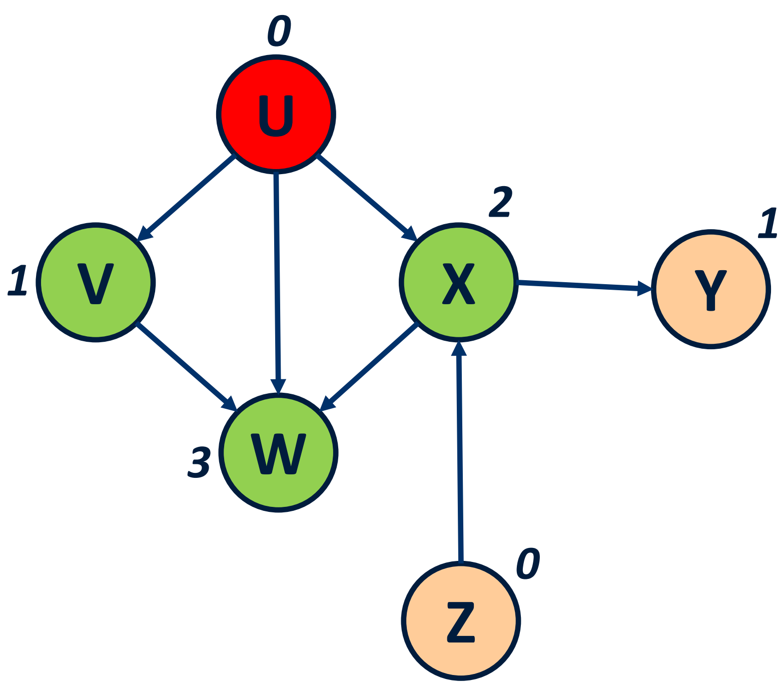

- Compute **in_degree**
- Enqueue U and Z

Q = U, Z

- Dequeue -> U

**Topological Sort: U -**

# Topological Sort - Example



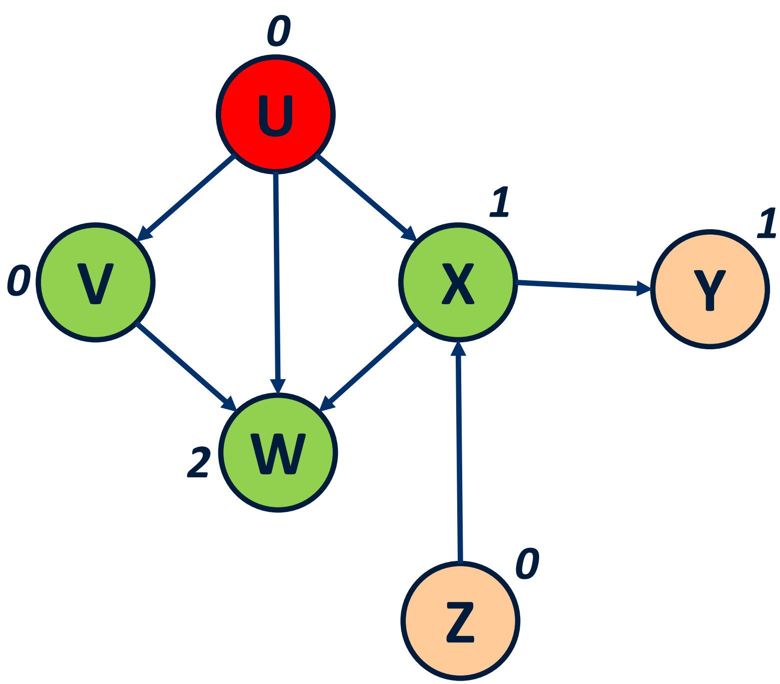

- Compute **in_degree**
- Enqueue U and Z

Q = Z

- Dequeue -> U
- Decrease in-degree for neighbors

**Topological Sort: U -**

# Topological Sort - Example



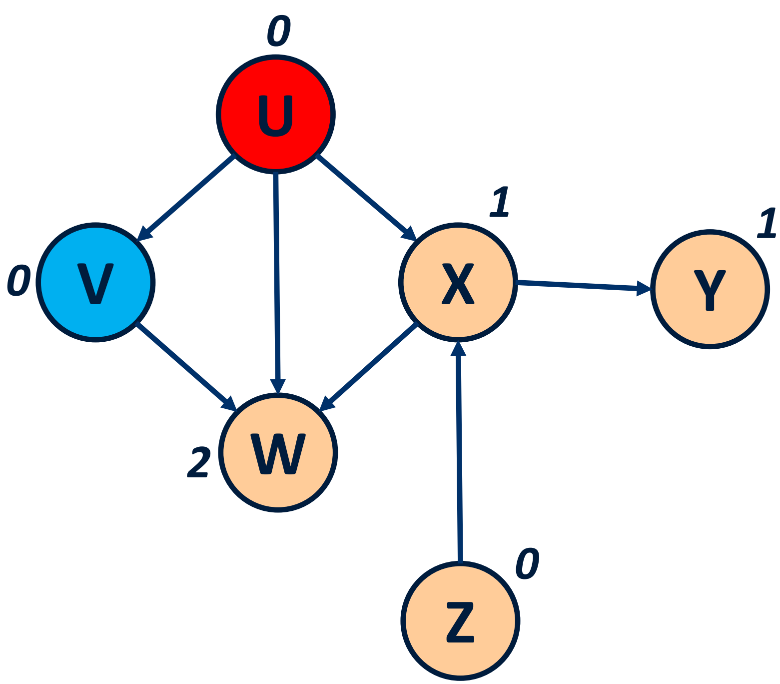

- Compute **in_degree**
- Enqueue U and Z

Q = Z

- Dequeue -> U
- Decrease in-degree for neighbors

**Topological Sort: U -**

# Topological Sort - Example



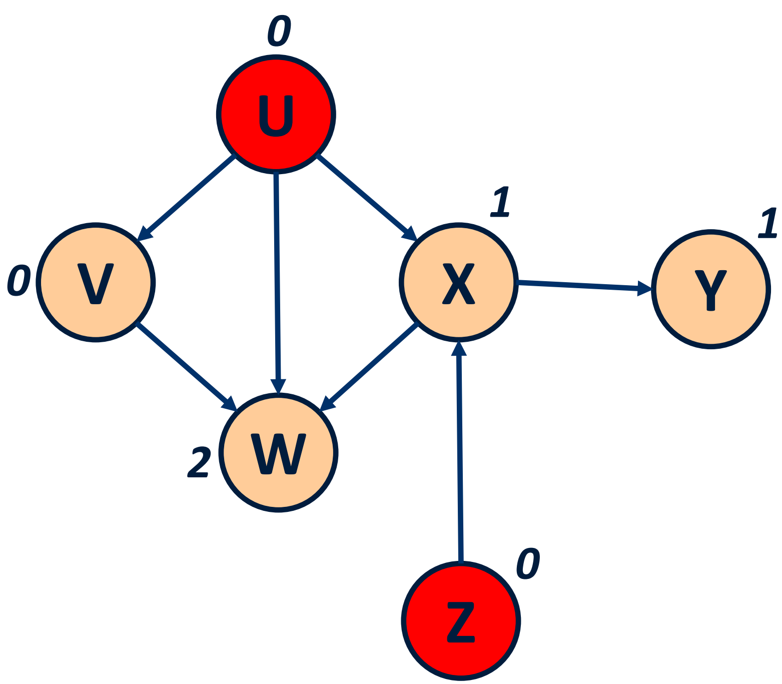

- Compute **in_degree**
- Enqueue U and Z

Q = Z, V

- Dequeue -> U
- Decrease in-degree for neighbors
- Enqueue V

**Topological Sort: U -**

# Topological Sort - Example



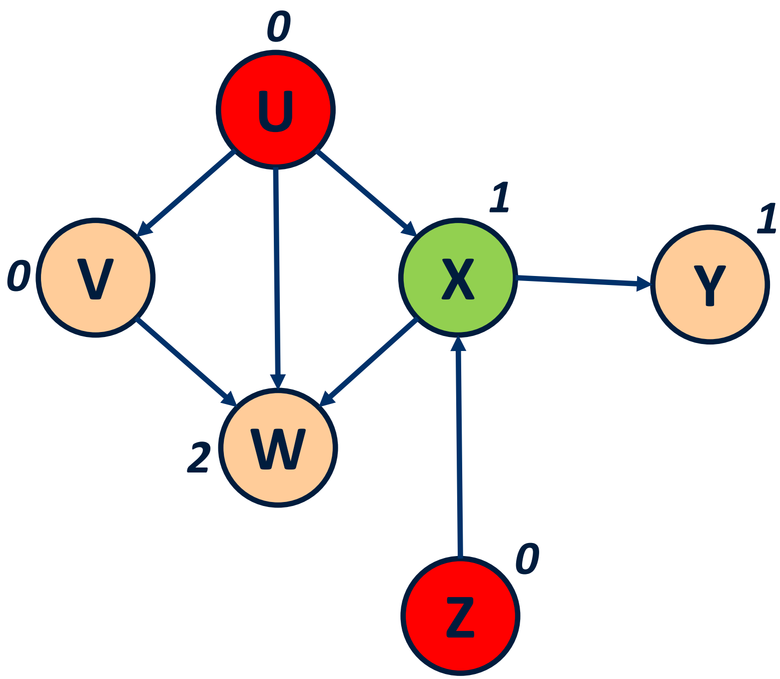

- Compute **in_degree**
- Enqueue U and Z

Q = V

- Dequeue -> Z

**Topological Sort: U - Z**

# Topological Sort - Example



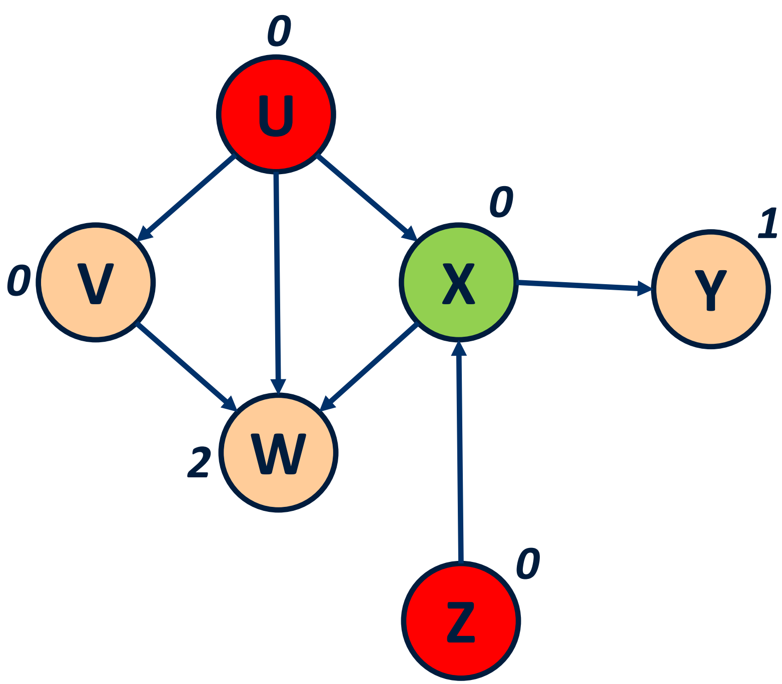

- Compute **in_degree**
- Enqueue U and Z

Q = V

- Dequeue -> Z
- Decrease in-degree for neighbors

**Topological Sort: U - Z**

# Topological Sort - Example



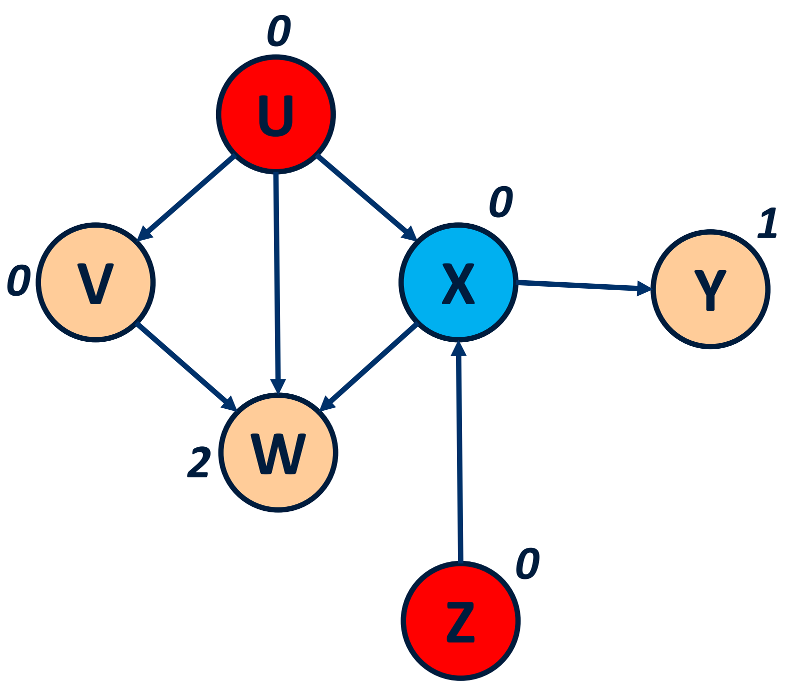

- Compute **in_degree**
- Enqueue U and Z

Q = V

- Dequeue -> Z
- Decrease in-degree for neighbors

**Topological Sort: U - Z**

# Topological Sort - Example



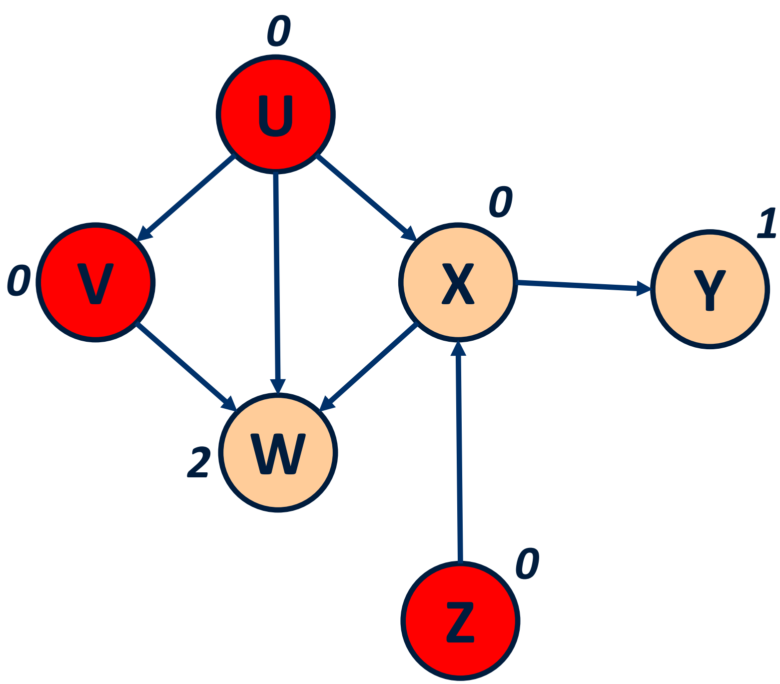

- Compute **in_degree**
- Enqueue U and Z

Q = V, X

- Dequeue -> Z
- Decrease in-degree for neighbors
- Enqueue X

**Topological Sort: U - Z**

# Topological Sort - Example



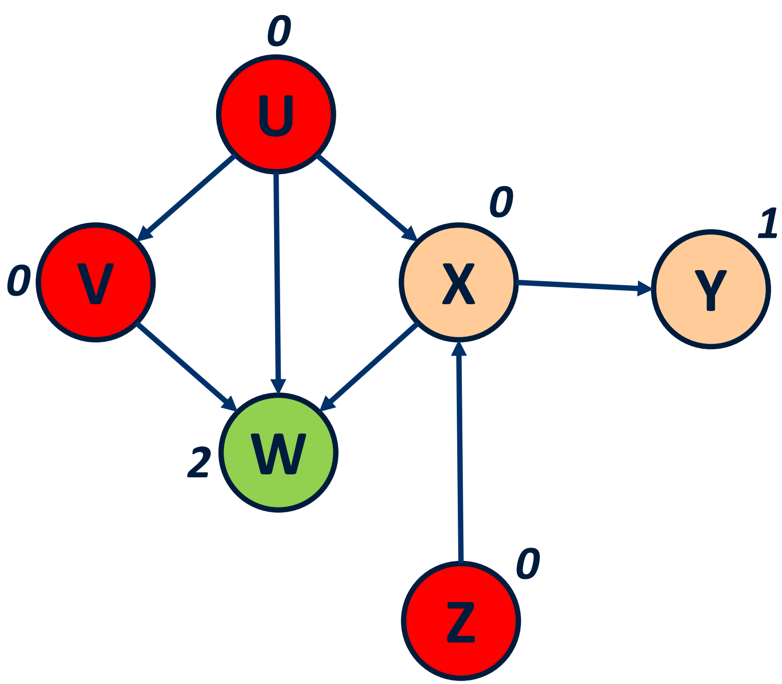

- Compute **in_degree**
- Enqueue U and Z

Q = X

- Dequeue -> V

**Topological Sort: U - Z - V**

# Topological Sort - Example



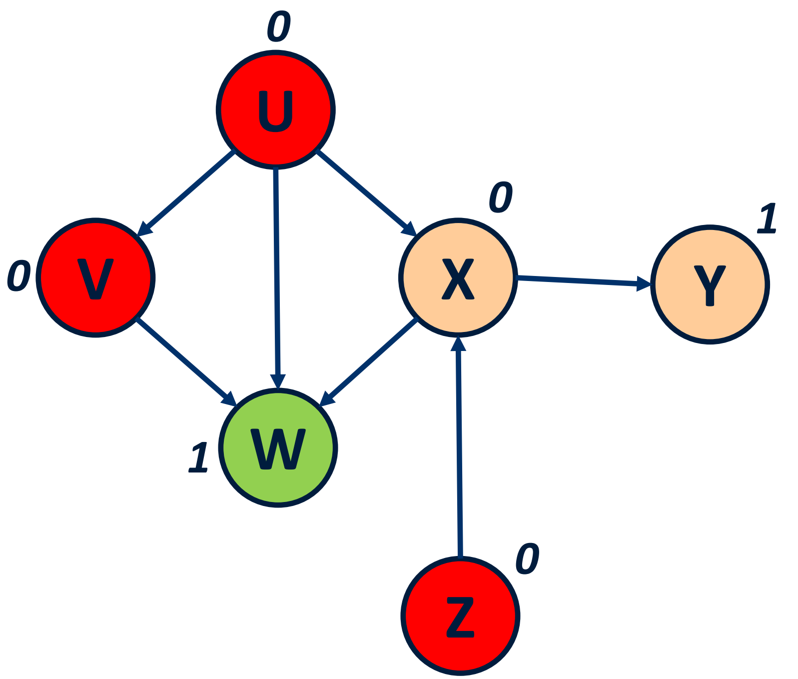

- Compute **in_degree**
- Enqueue U and Z

Q = X

- Dequeue -> V
- Decrease in-degree for neighbors

**Topological Sort: U - Z - V**

# Topological Sort - Example



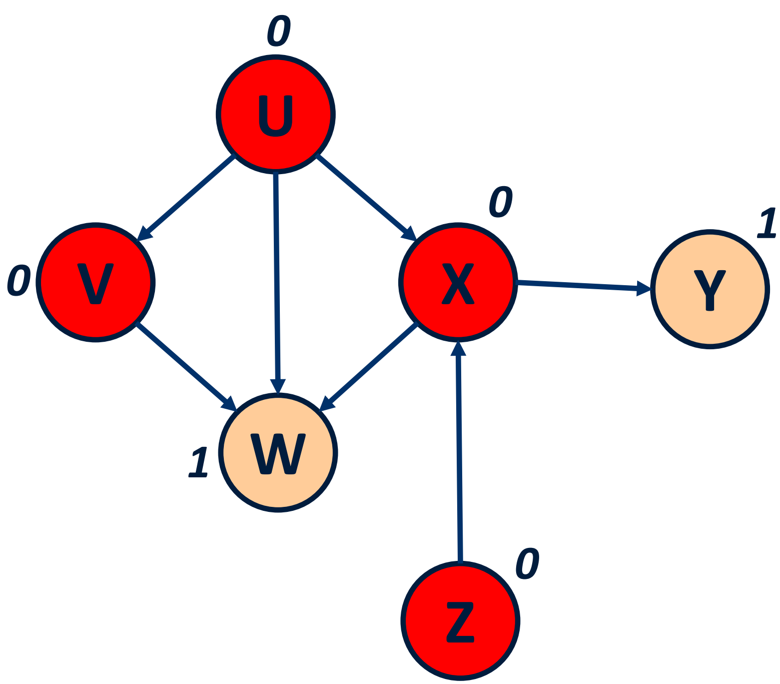

- Compute **in_degree**
- Enqueue U and Z

Q = X

- Dequeue -> V
- Decrease in-degree for neighbors

**Topological Sort: U - Z - V**

# Topological Sort - Example



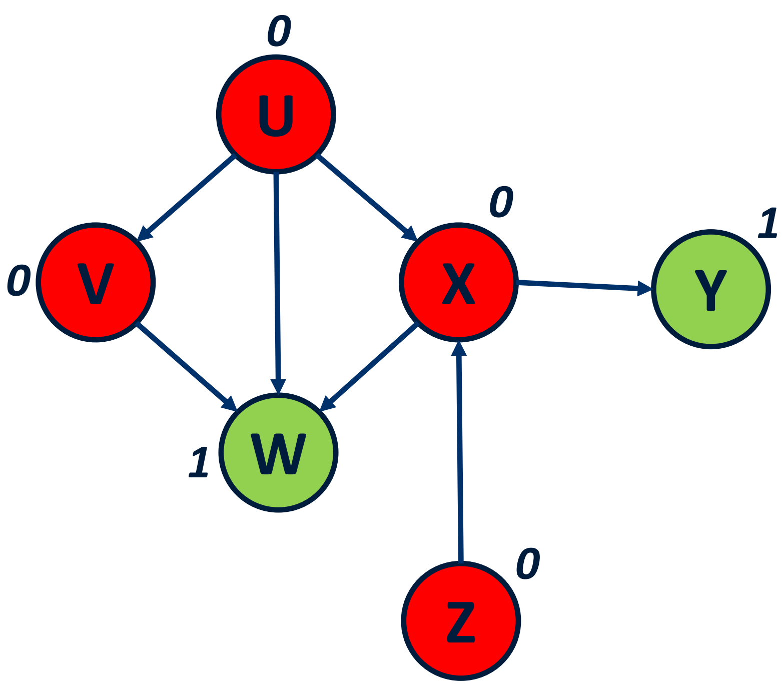

- Compute **in_degree**
- Enqueue U and Z

Q = empty

- Dequeue -> X

**Topological Sort: U - Z - V - X**

# Topological Sort - Example



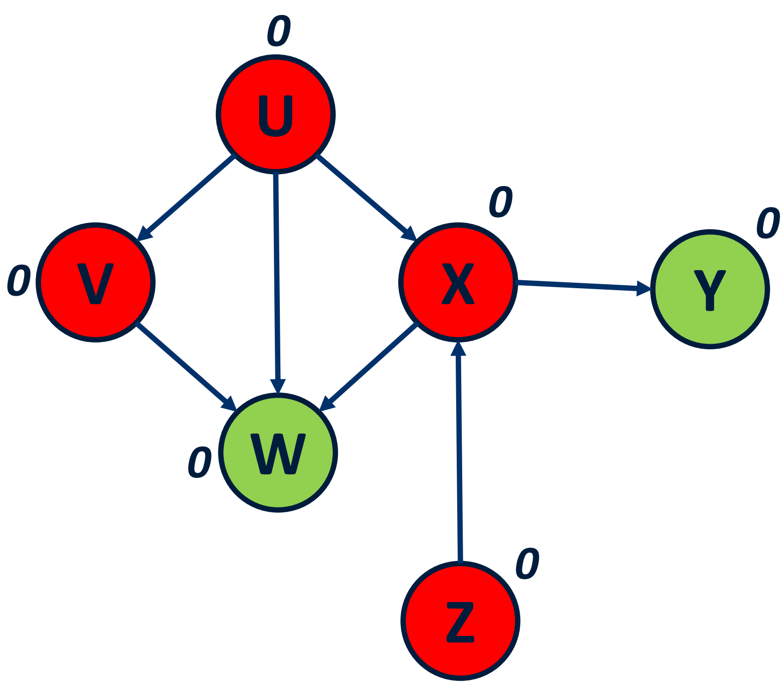

- Compute **in_degree**
- Enqueue U and Z

Q = empty

- Dequeue -> X
- Decrease in-degree for neighbors

**Topological Sort: U - Z - V - X**

# Topological Sort - Example



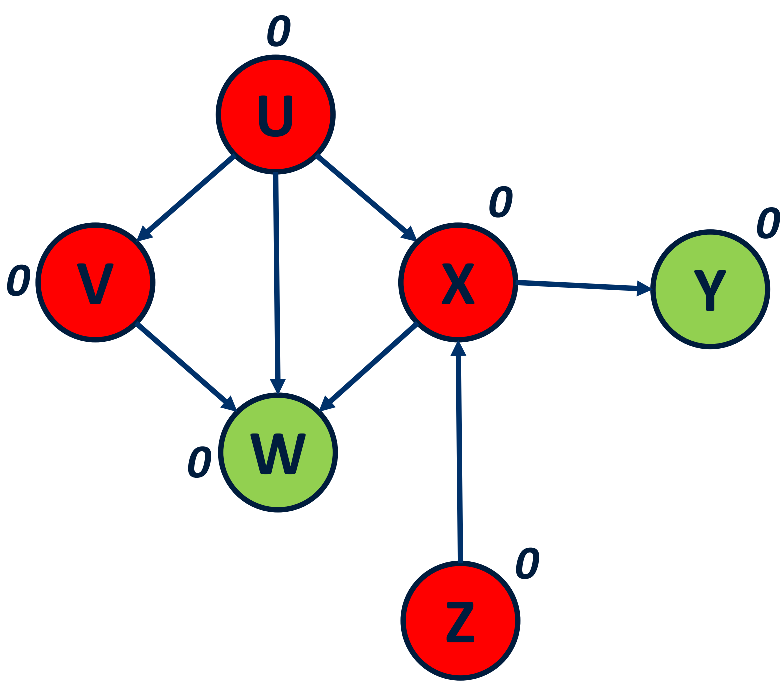

- Compute **in_degree**
- Enqueue U and Z

Q = empty

- Dequeue -> X
- Decrease in-degree for neighbors

**Topological Sort: U - Z - V - X**

# Topological Sort - Example



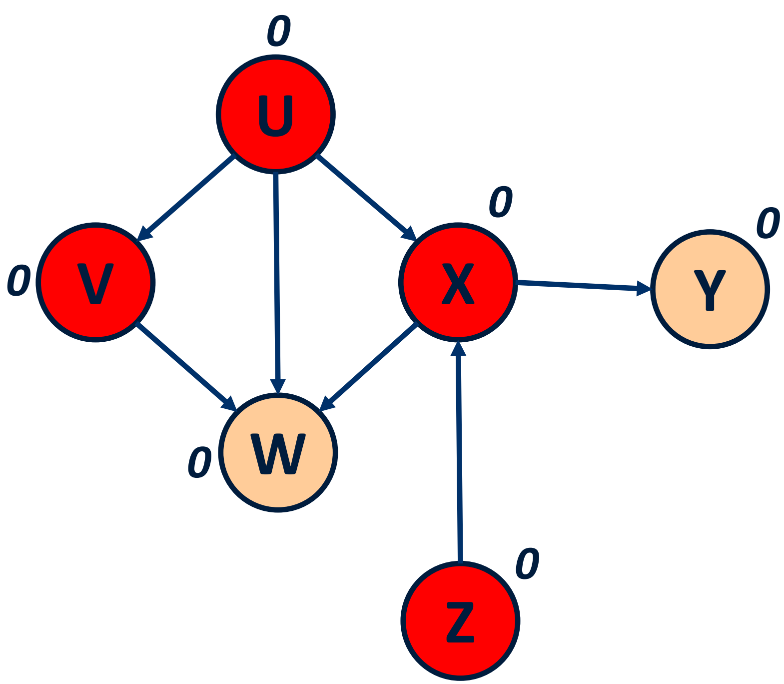

- Compute **in_degree**
- Enqueue U and Z

Q = empty

- Dequeue -> X
- Decrease in-degree for neighbors
- Enqueue W and Y

**Topological Sort: U - Z - V - X**

# Topological Sort - Example



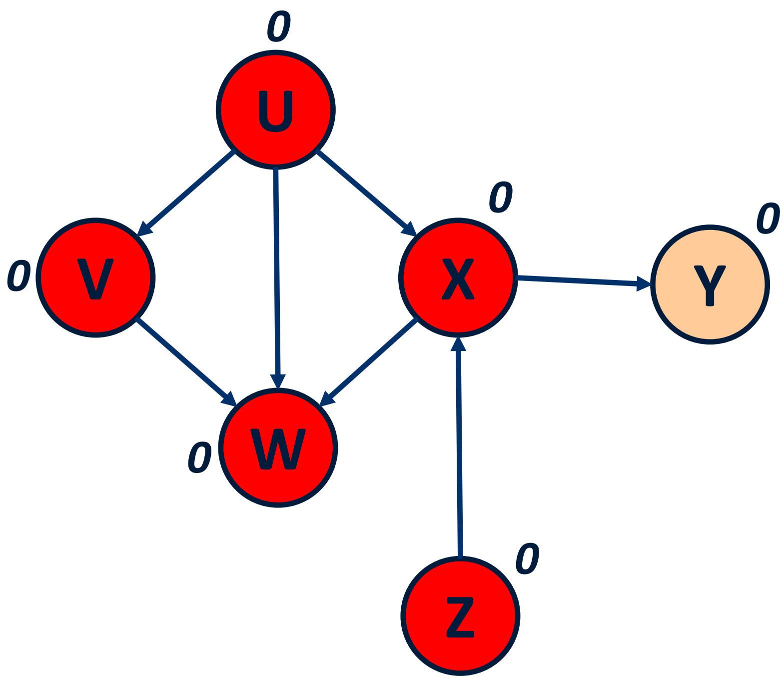

- Compute **in_degree**
- Enqueue U and Z

Q = W, Y

- Dequeue -> X
- Decrease in-degree for neighbors
- Enqueue W and Y

**Topological Sort: U - Z - V - X**

# Topological Sort - Example



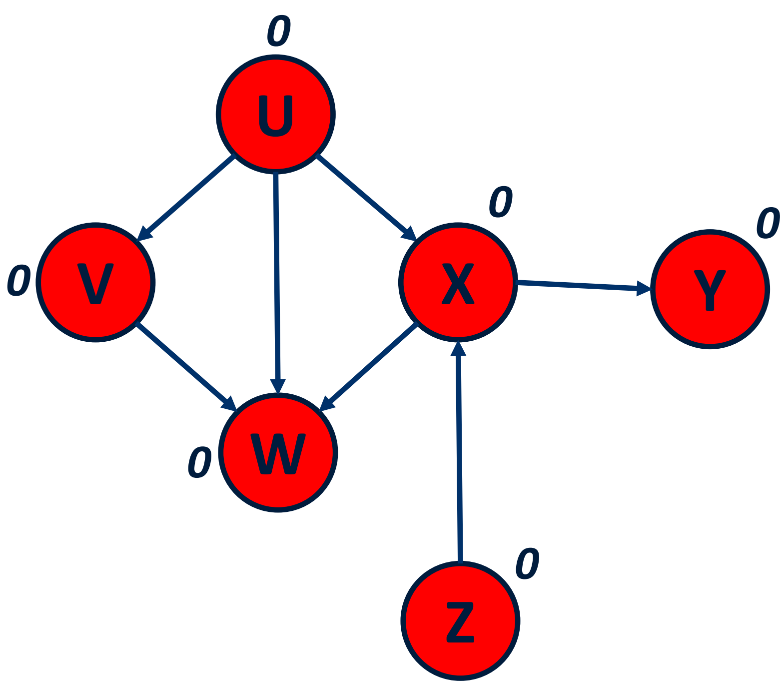

- Compute **in_degree**
- Enqueue U and Z

Q = Y

- Dequeue -> W
- No neighbors

**Topological Sort: U - Z - V - X - W**

# Topological Sort - Example




- Compute **in_degree**
- Enqueue U and Z

Q = empty

- Dequeue -> Y
- No neighbors

**Topological Sort: U - Z - V - X - W - Y**

# Topological Sort

- Running Time
  - Initialization: $O(|V|+|E|)$ (assuming adjacency list)
  - Sum of all enqueues and dequeues: $O(|V|)$
  - Sum of all decrements: $O(|E|)$ (assuming adjacency list)

So total is $O(|E|+|V|)$