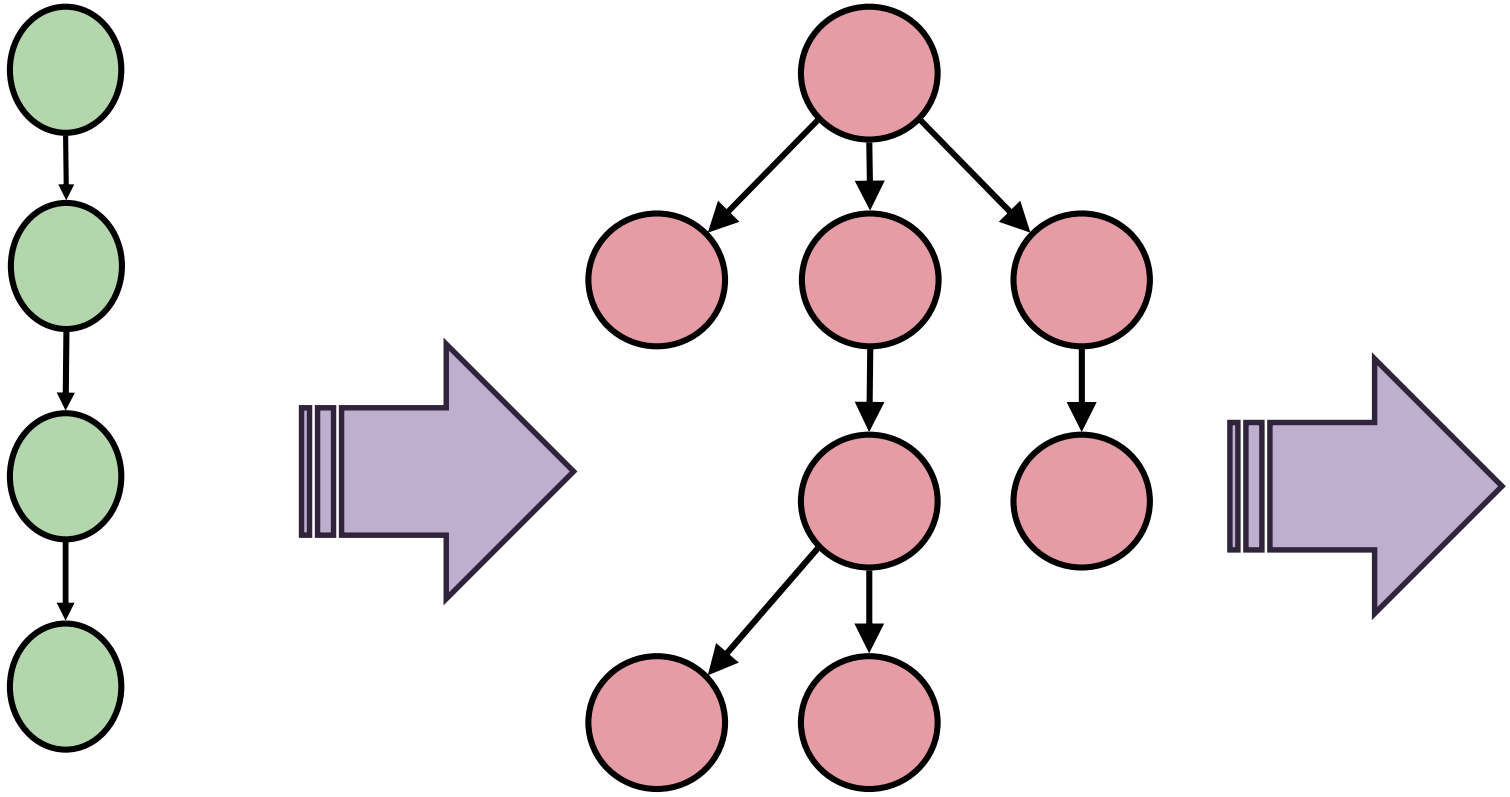
Graph

IT5003: Data Structures and Algorithms
(**AY2019/20 Semester 1**)

Lecture Outline

- **Graph**
  - Motivation
  - Definitions
  - Representations
    - Adjacency Matrix / List / Map
    - Incidence Matrix
  - Major Algorithms:
    - Breadth-First Search
    - Topological Sort
    - Shortest Path Algorithm

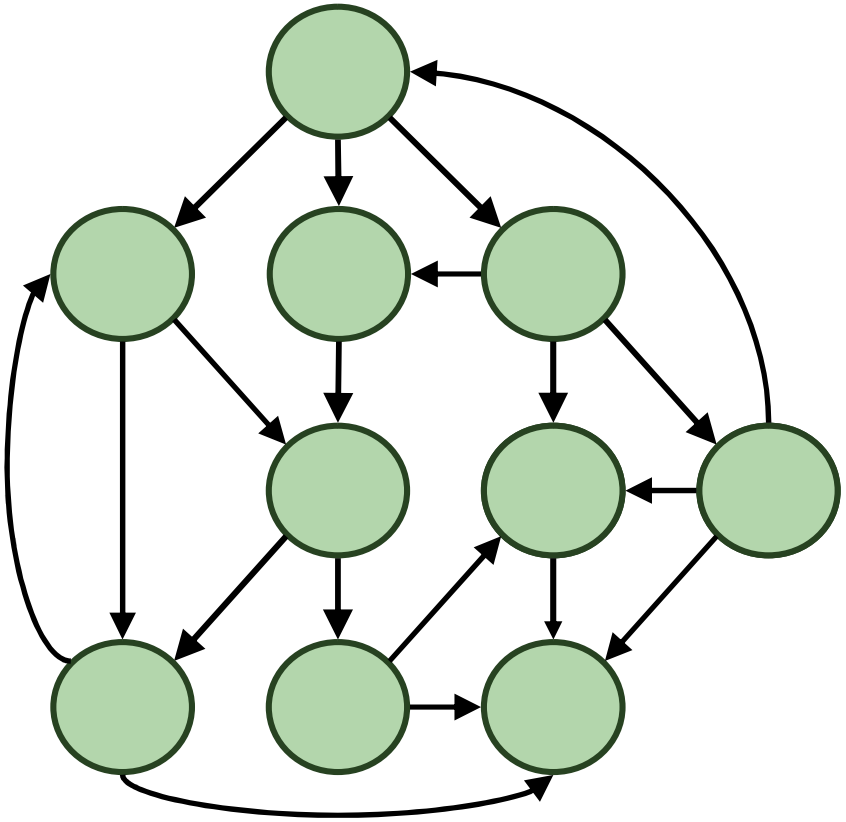
Motivation: Why Graph?



Linked List
- 1 parent
- 1 child

Tree
- 1 parent
- Multiple child

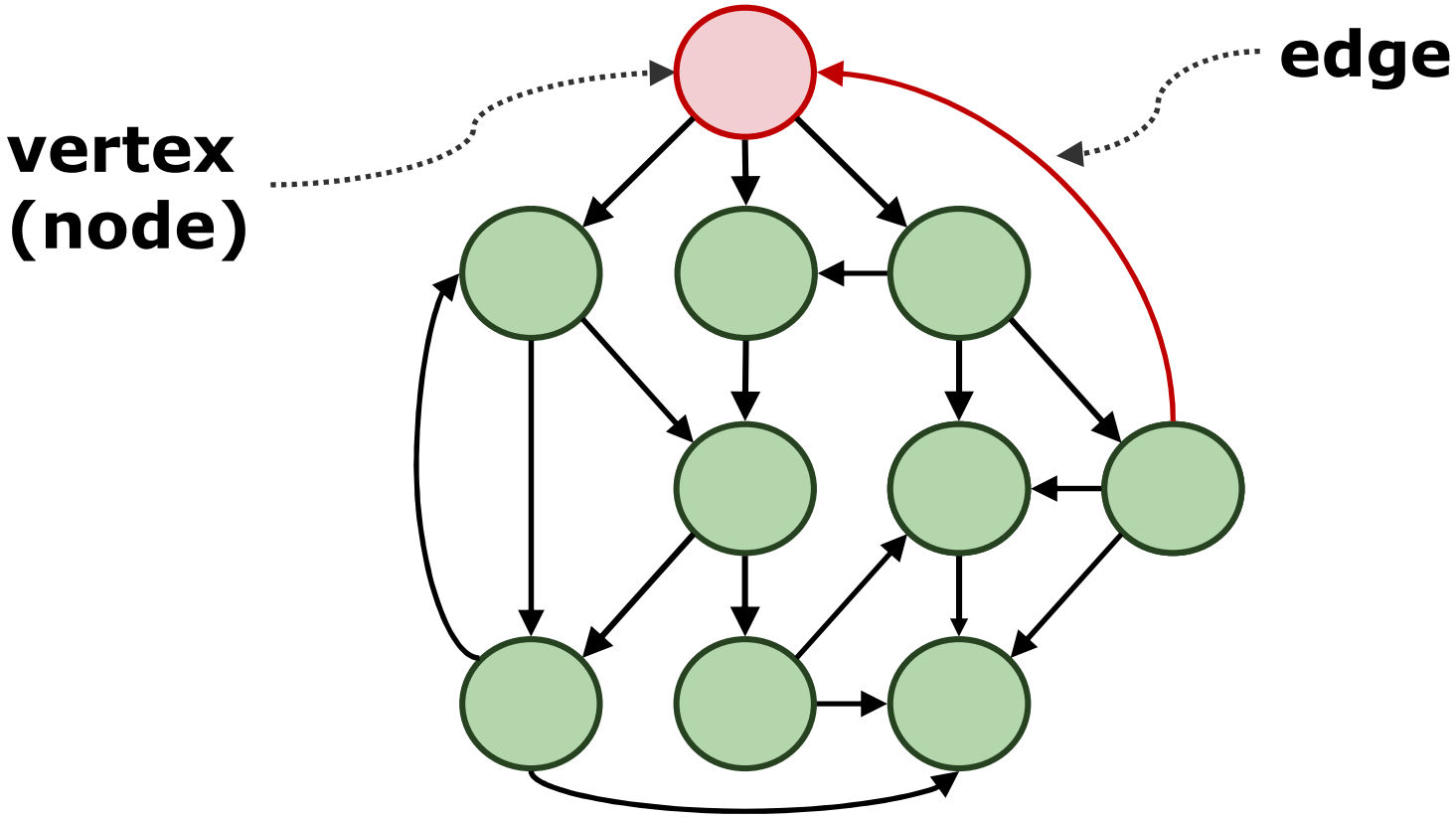
Motivation: Why Graph?



- Easy to represent complex relationship
- Example:
  - Map, links between webpages,

Graph
- Multiple parent
- Multiple child

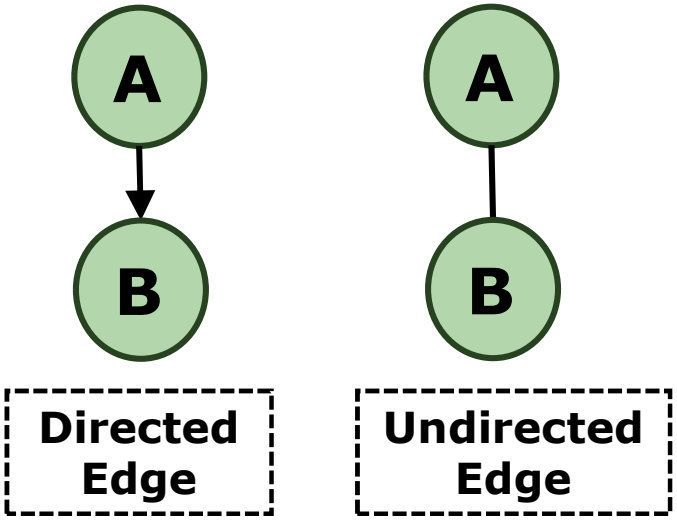
Terminology: **Edge & Vertex**



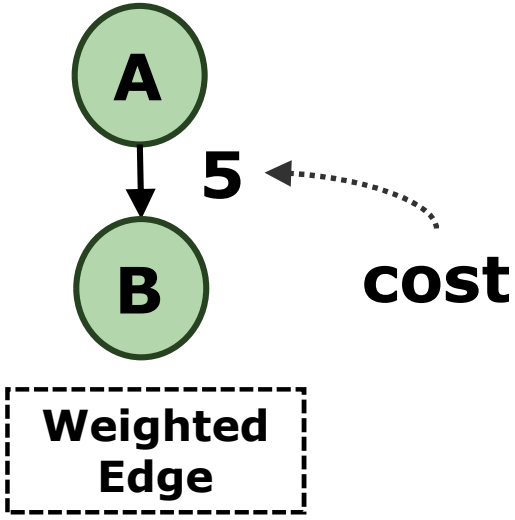
- **Vertices:** Data objects in a graph
- **Edges:** Links between nodes

Terminology: **Types of Edge**

- Characteristics of an edge:
  1. Directed / Undirected
     - Is travelling allowed in both directions?

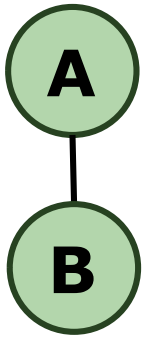


  2. Weighted / Non-weighted
     - What is the cost associated with the edge?



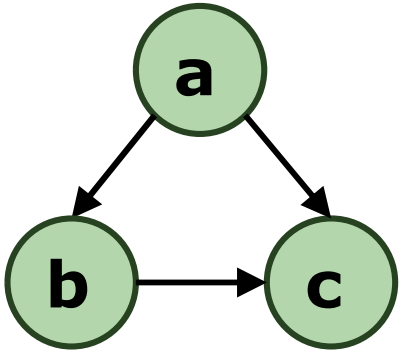
Terminology: **Adjacency**

- Node A is adjacent to node B if there is an edge between them
  - A and B are neighbor



- `adj(v)` = the set of vertices adjacent to `v`

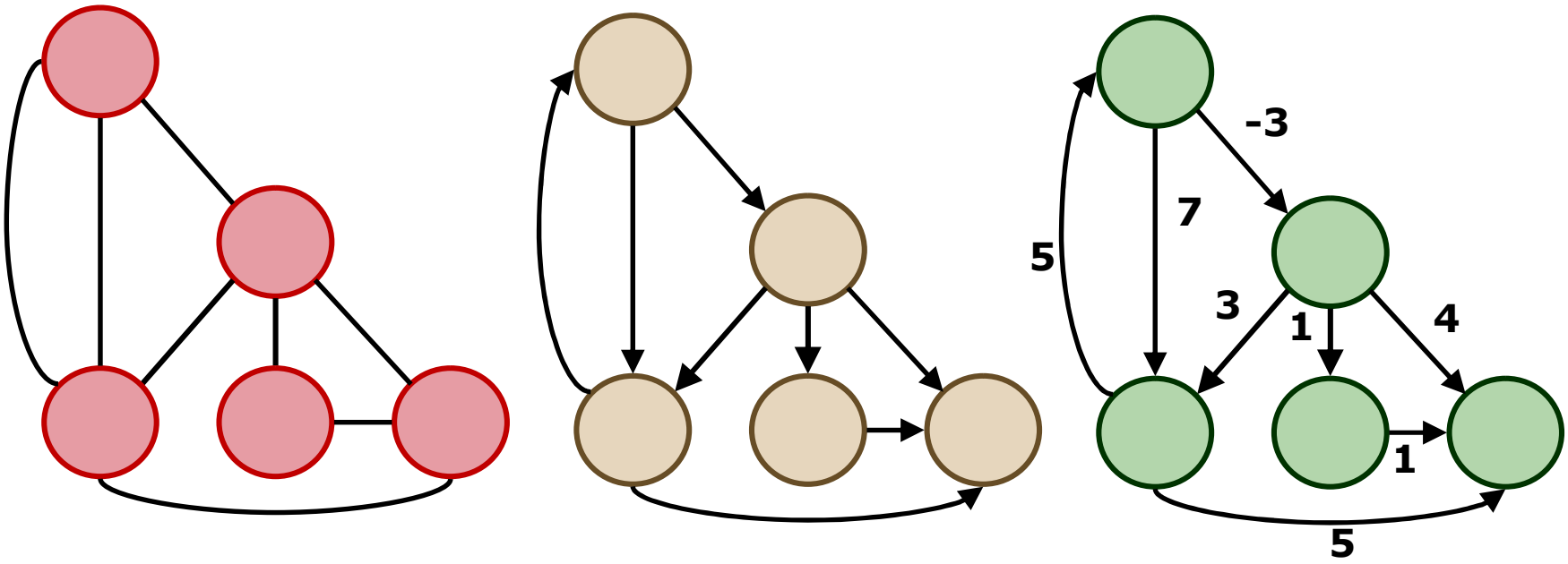
- Example:
  - `adj(a) = {b, c}`
  - `adj(b) = {c}`
  - `adj(c) = {}`



Definition: **Graph**

- Formally, graph consists of:
  - **V** = The set of vertices
  - **E** = The set of edges
  - w = The weight function
    - w(E_1) = the cost associated with edge E_1
- Hence,
  - **G = {V, E, w}, where G is a graph**

Example Graphs



Undirected Graph

Directed Graph

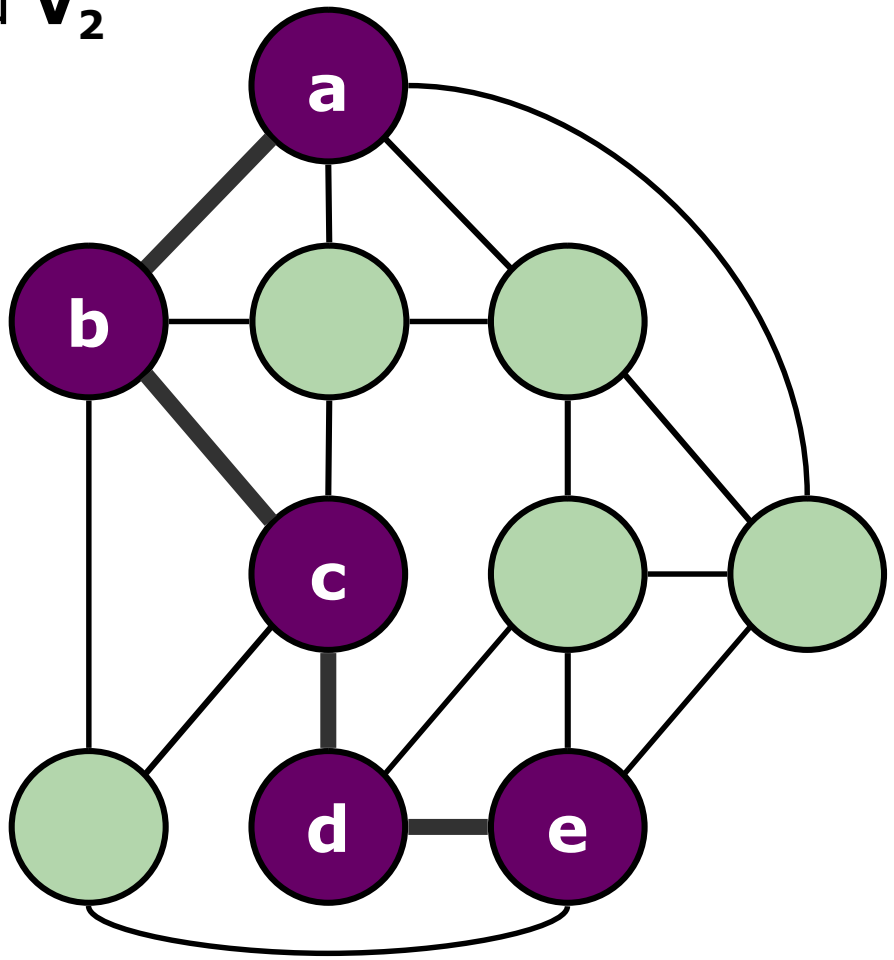
Directed Weighted Graph

Definition: **Path**

Path between vertices $\mathbf{V_1}$ and $\mathbf{V_2}$
= Sequence of edges that
begin at $\mathbf{V_1}$ and end at $\mathbf{V_2}$

Length of a path p
= The number of edges in p

Simple path
= Vertices in the path
are visited once only

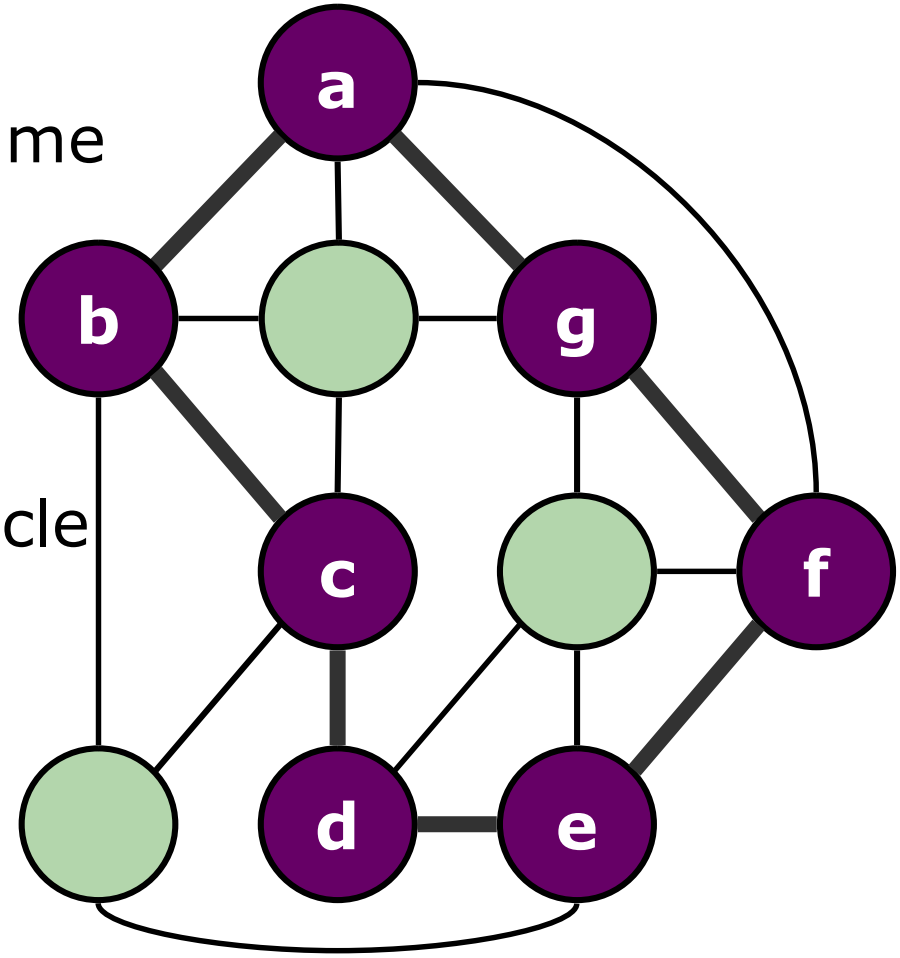


Definition: Cycle

Cycle
= A **path** that
begins and ends at the same
vertex

Simple cycle
= A simple path that is a cycle

Acyclic Graph
= Graph with no cycle



Definition: Connected / Disconnected

Connected graph

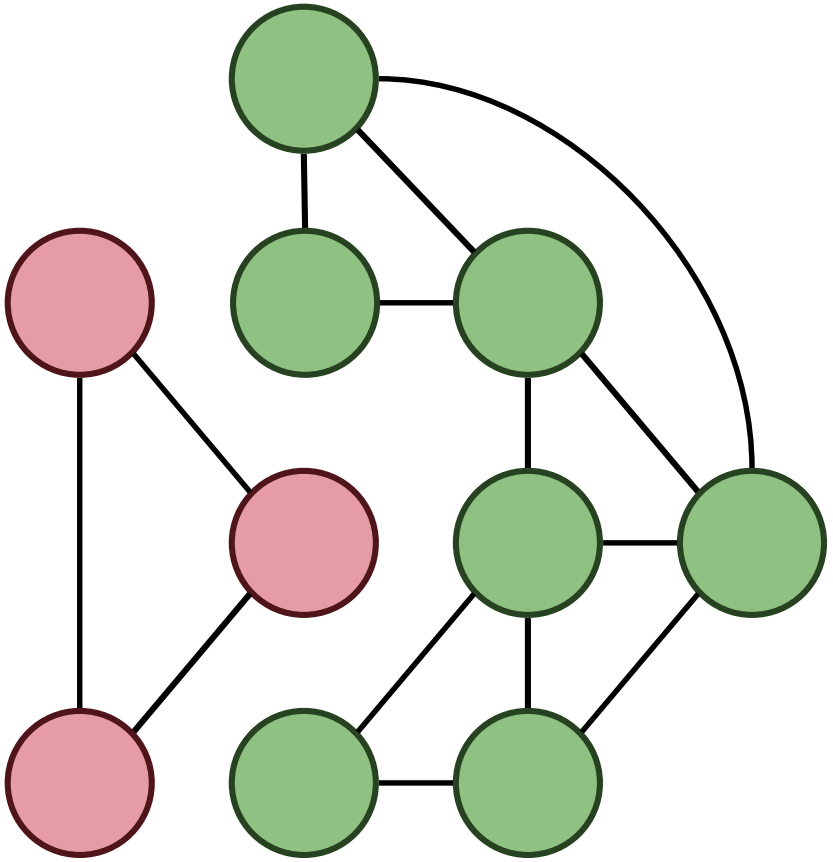
= There is a path between every pair of nodes

Disconnected graph

= There is at least one pair of nodes with no path between them

Component

= Connected subgraph

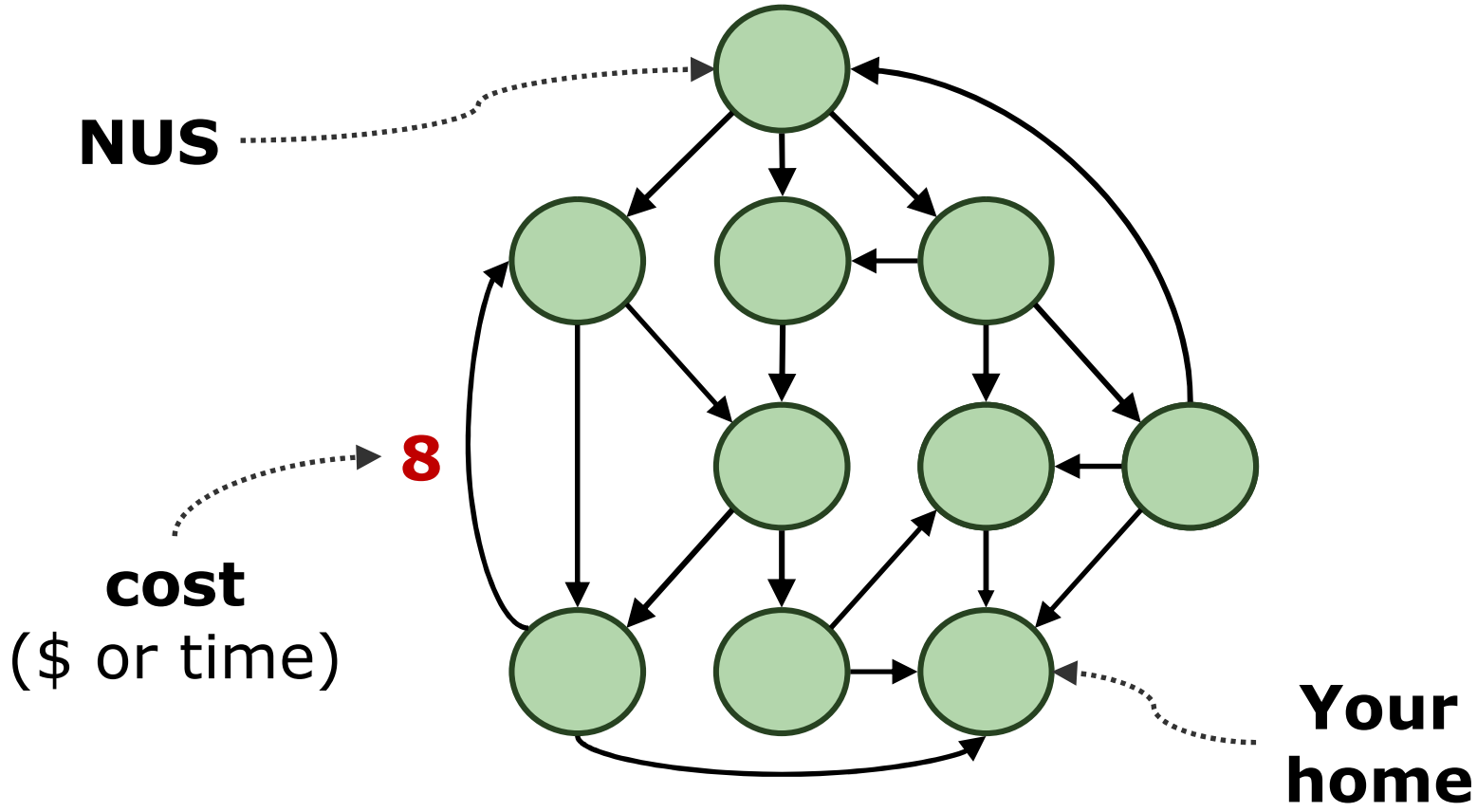


Disconnected graph with two components

Is Graph any good?

EXAMPLE APPLICATION

Transport Route as Graph



Interesting Questions

- With the graph representation, we can ask:

What is the fastest way / cheapest route to travel between "Home" and "NUS"?

"SHORTEST PATH PROBLEM"

How to find the simple cycle that visit every nodes and has the minimum cost?

"TRAVELING SALESMAN PROBLEM"

Graph implementations

REPRESENTATION

Overview

- Recall that
  - **G = {V, E, w}, where G is a graph**
  - ➔ **We need to represent {V, E, w} in some ways**

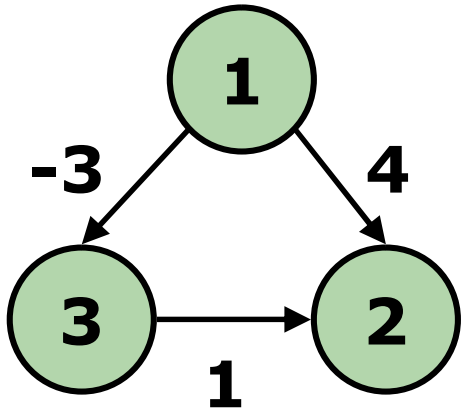
- Observe the **adjacency relationship**:
  - **adj(V) = set of vertices adjacent to v**
  - $\sum_v |adj(v)| = |E|$

  - i.e. **The adjacency information for all vertices == The set of all edges**

Option 1. Adjacency Matrix

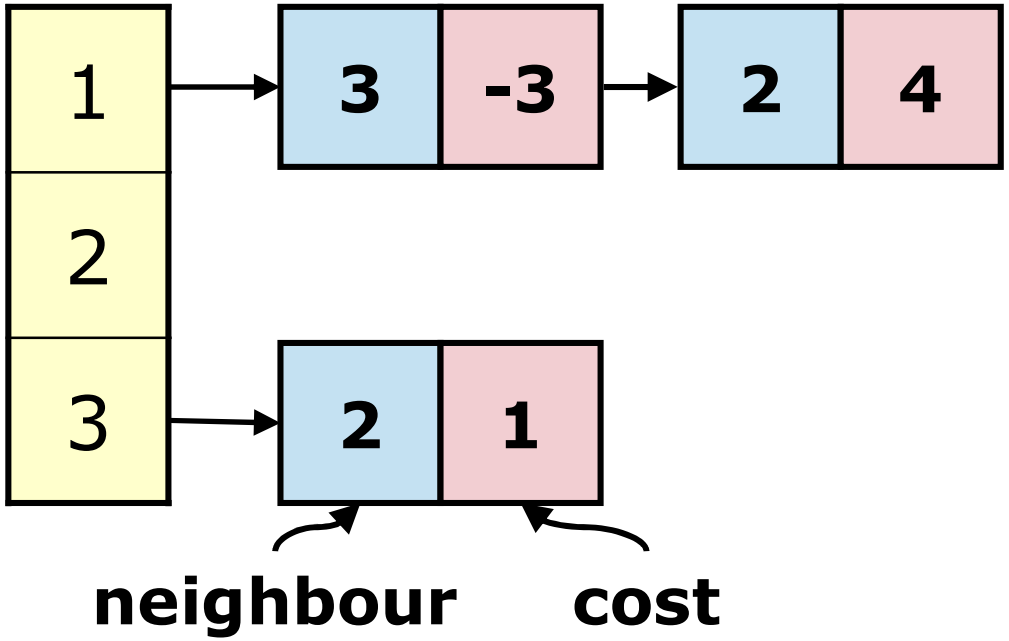
- Use a matrix **M** of size N x N
  - Where N is the number of vertices
  - M[***x***][***y***] = the edge from vertex ***x*** to vertex ***y***

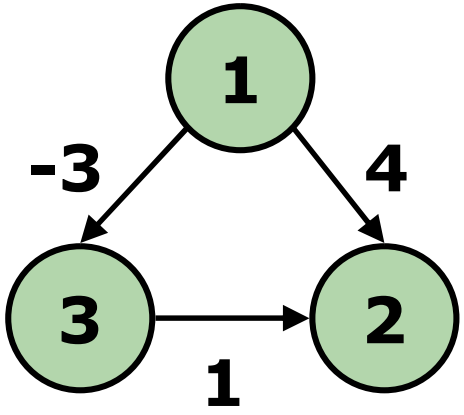
	1	2	3
1	-	4	-3
2	-	-	-
3	-	1	-



Option 2. Adjacency List

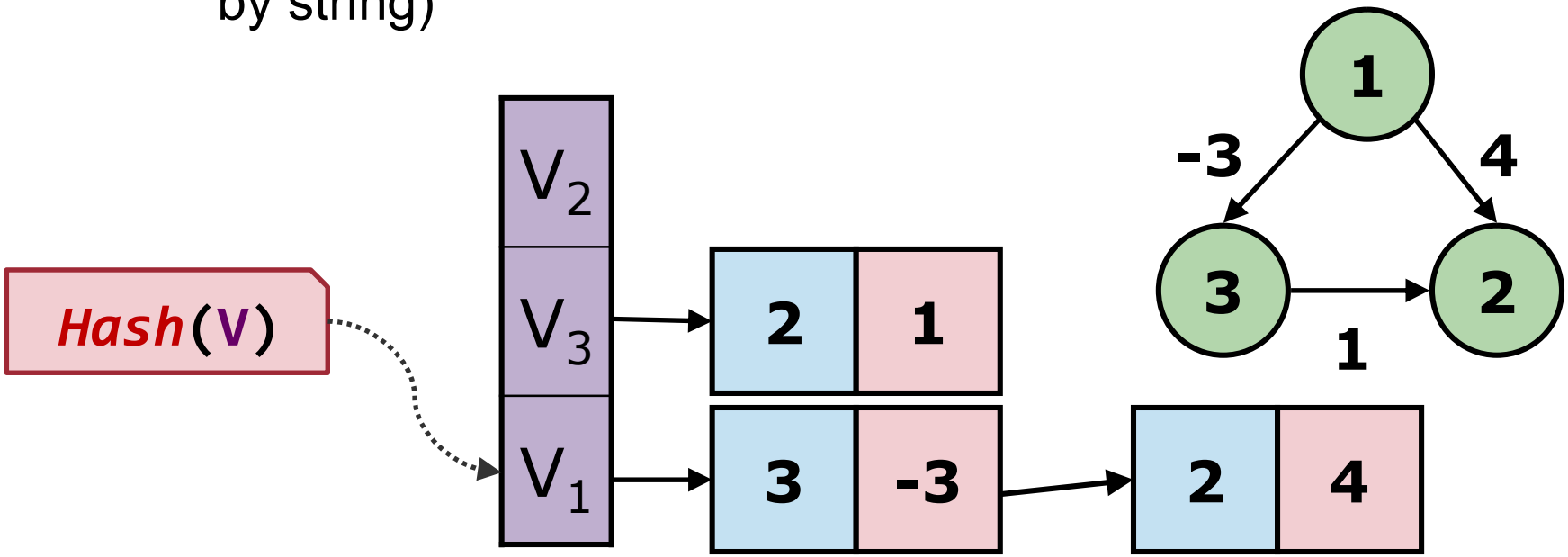
- Use an array **A** of **N linked lists**
  - N is the number of vertices
  - The linked list at A[***x***] represents the edges going out from vertex ***x***





Option 3. Adjacency Map

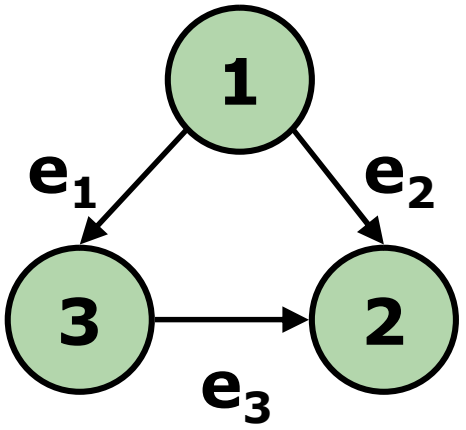
- Use Hash Table **H** of at least size **N**
  - **N** is the number of vertices
  - The hash table maps the vertex to the index
    - Each entry contains information of outgoing edges from this vertex (e.g. can stored as linked list)
    - Works for non-integer vertex too (e.g. vertex is identified by string)



Option 4. Incidence Matrix

- Use a matrix of **N** x **E**
  - Each row represent a single node
  - Each column represent a single edge
- `Matrix[V][E] = 1` if edge `E` has `V` as one of the endpoints

	e_1	e_2	e_3
1	**1**	**1**	**0**
2	**0**	**1**	**1**
3	**1**	**0**	**1**



Question:

a. How to represent **directed graph**?

b. How to represent **weighted graph**?

One way to traverse a graph

BREADTH-FIRST SEARCH

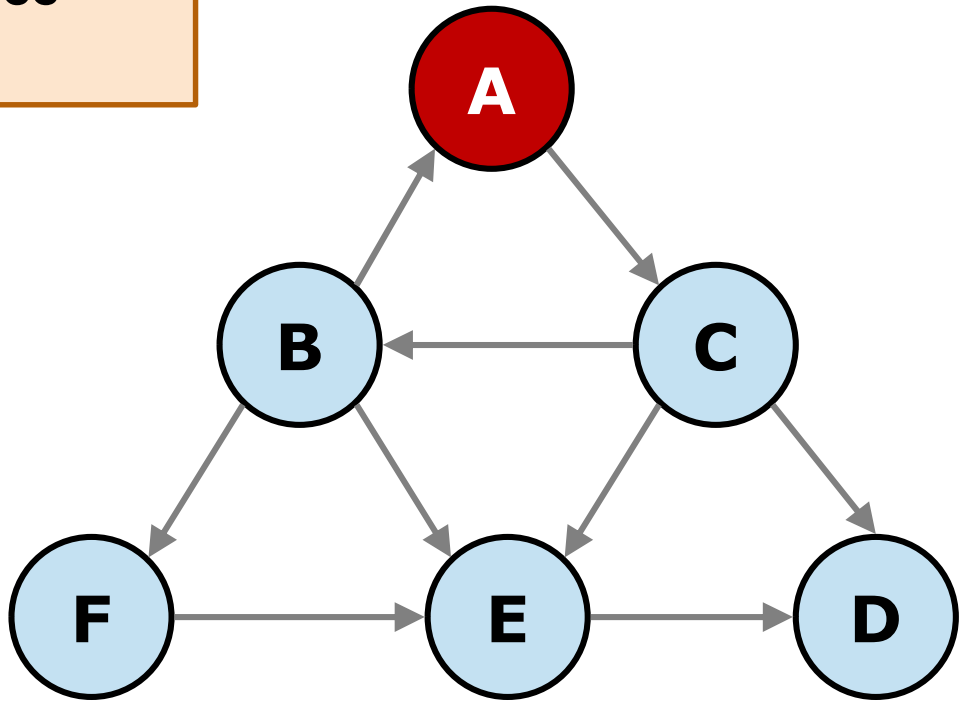
Breadth-First Search (BFS)

- Idea:
  - We explore the graph from a staring vertex **V**
  - All vertices that are of distance **D** away from **V** are visited before vertices that are of distance **D+1** away from **V**, and so on…
    - Distance = number of edges between two vertices
- Result of the traversal = **BFS sequence**
  - **Not unique** (same graph may give several equivalent traversals)
  - **Depends on the order** in which the neighboring vertices are visited

Example: **BFS**

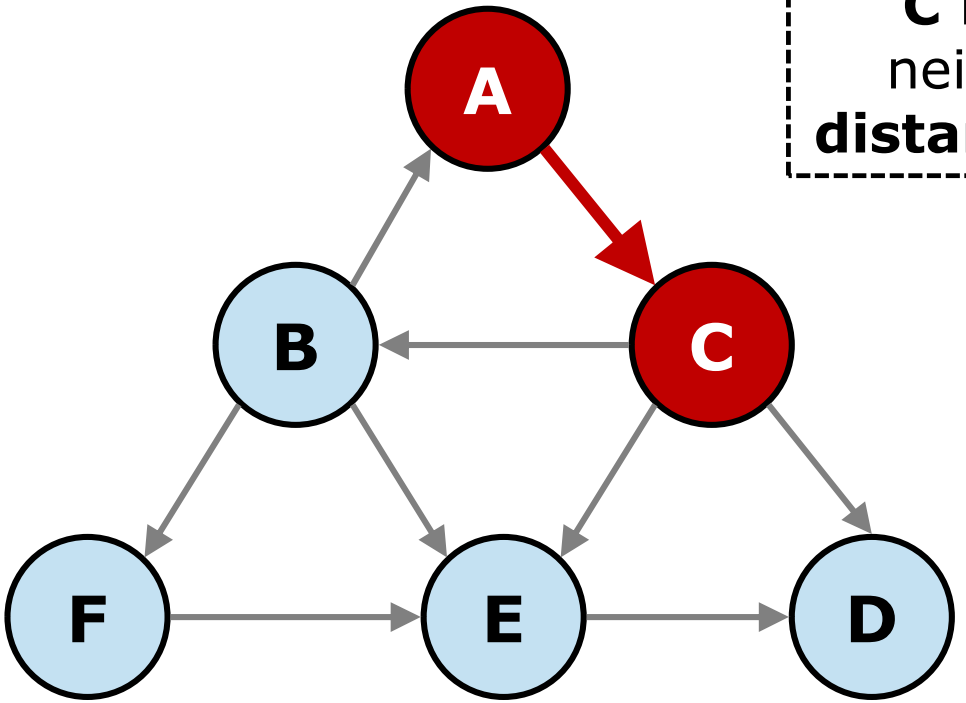
- Let's start the BFS from vertex A:

BFS Sequence
= { A,



Example: **BFS**

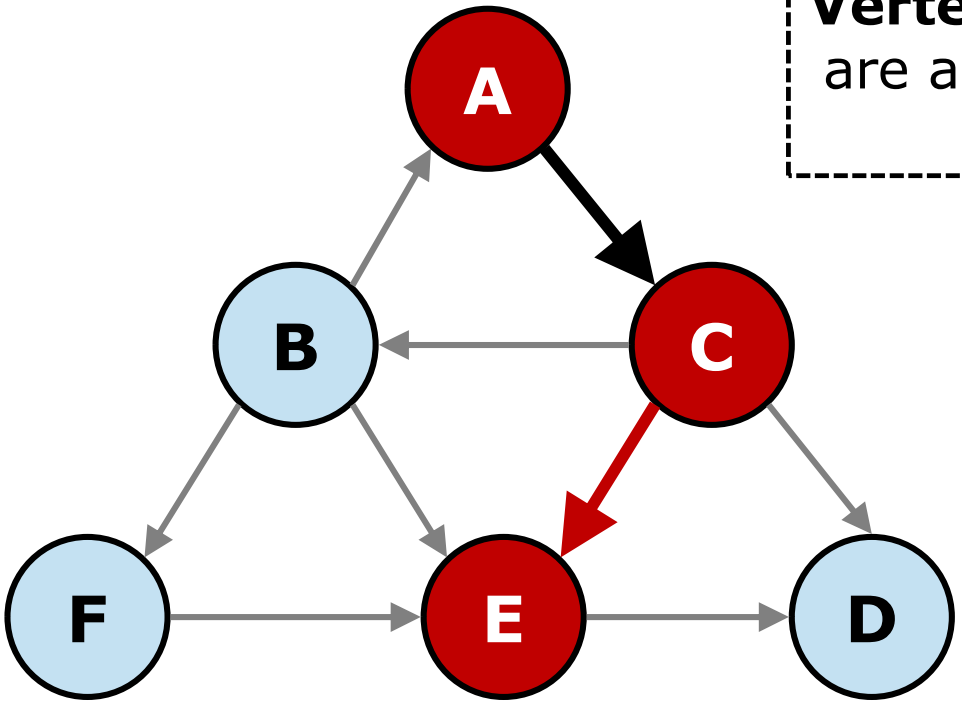
BFS Sequence
= { A, C



C is the only neighbor with **distance 1** from **A**

Example: **BFS**

BFS Sequence
= { A, C, E

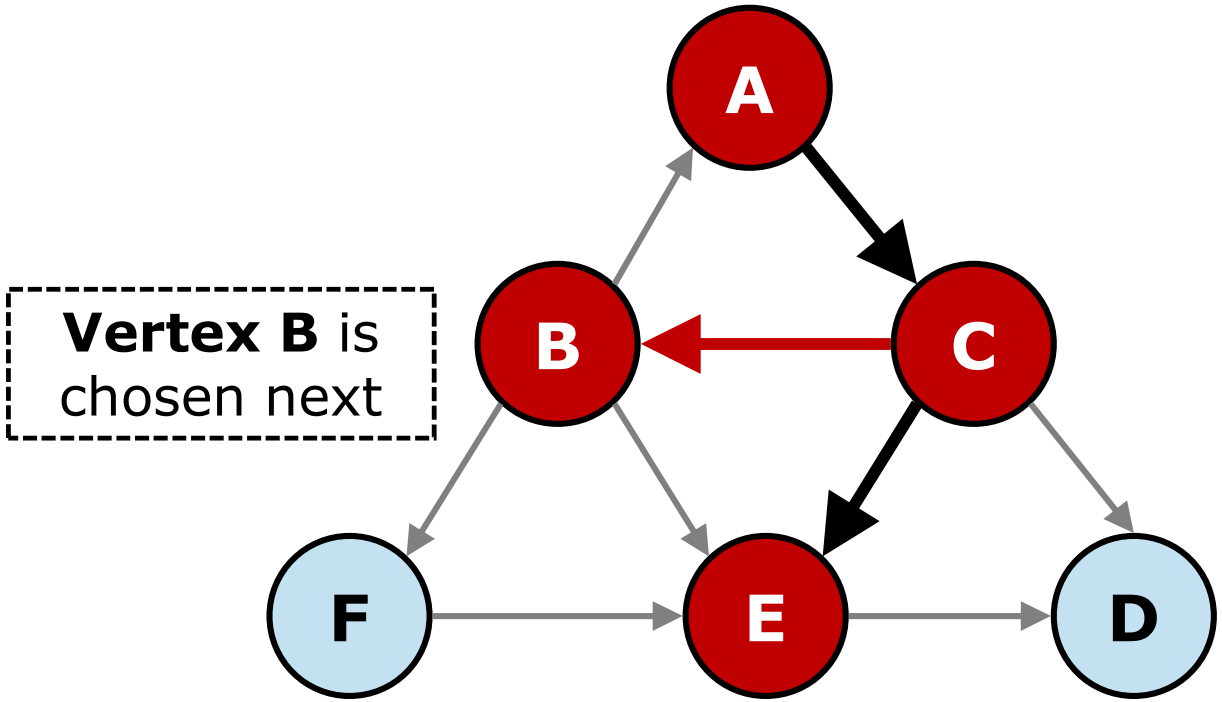


Vertex B, D and E are all **distance 2** from **A**

Vertex E is chosen in this example

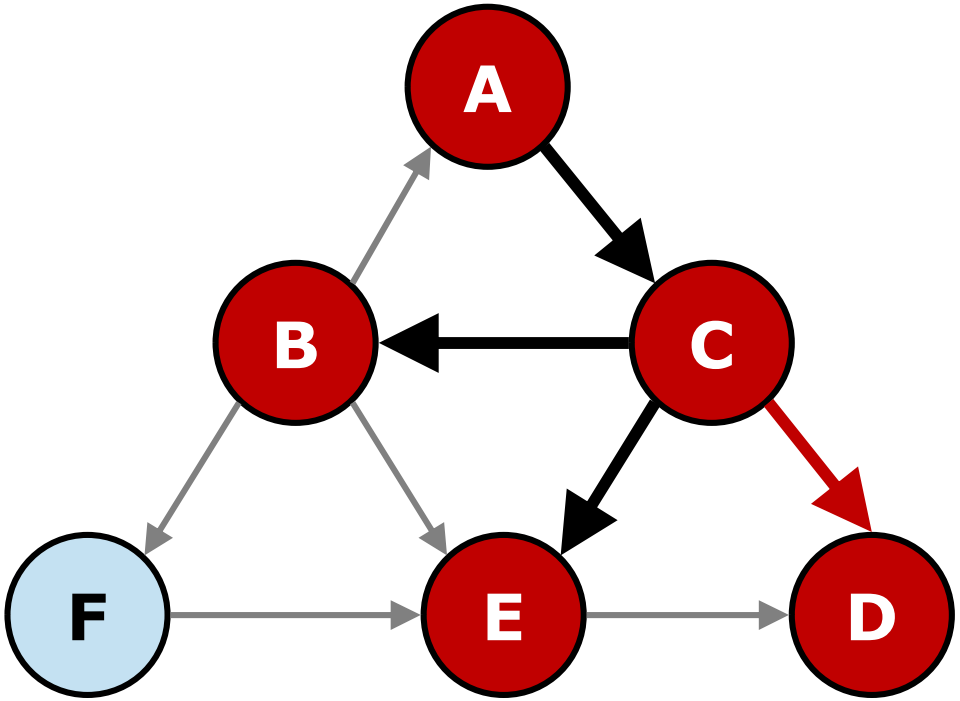
Example: **BFS**

BFS Sequence
= { A, C, E, B



Example: **BFS**

BFS Sequence
= { A, C, E, B, D

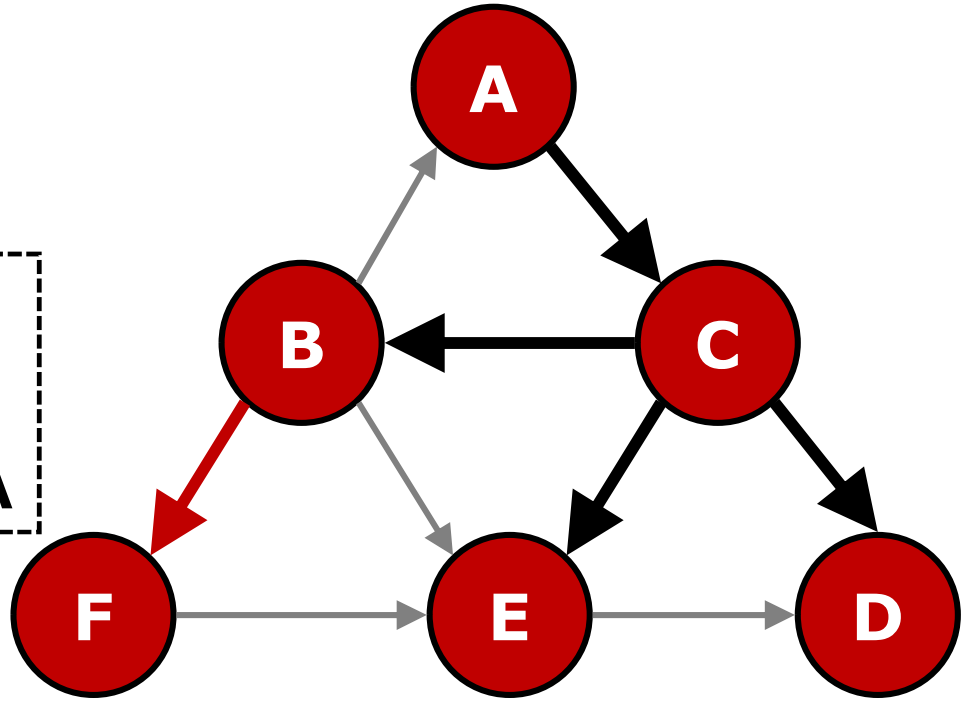


Vertex D is the last **distance 2** vertex from **A**

Example: **BFS**

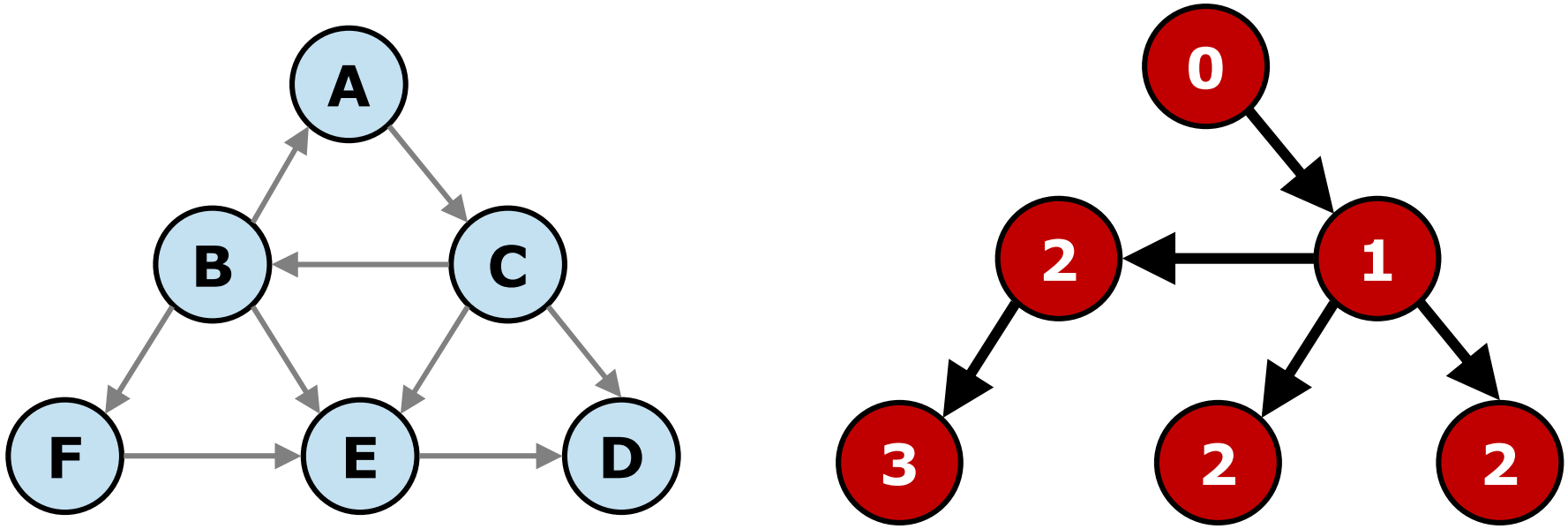
BFS Sequence
= { A, C, E, B, D, F }

Vertex F is the only **distance 3** vertex from **A**



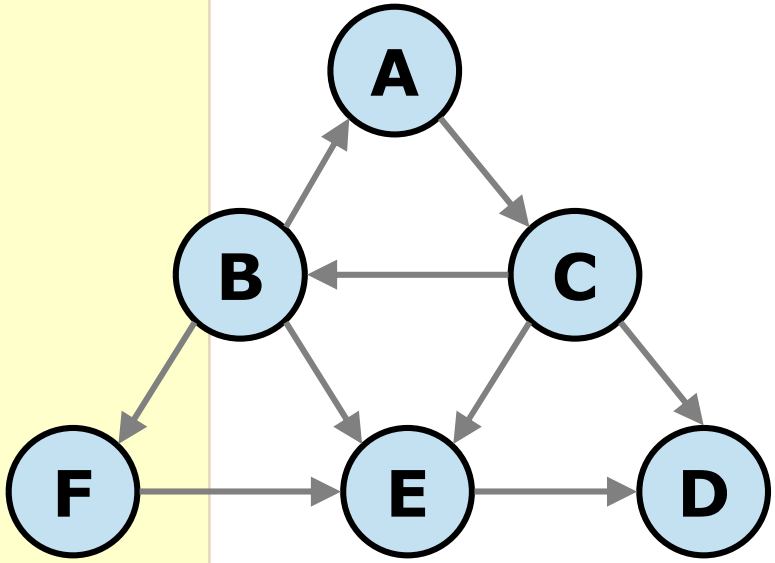
Example: **BFS**

- We essentially built a tree with root at **A**:
  - Known as the **BFS tree**



Breath-First Search: **Pseudo Code**

```
def BFS( V ):
    vQ = Queue()

    vQ.enqueue( V )

    while vQ not empty:
        cur = vQ.dequeue()
        print cur

        foreach w in adj(cur)
            if w is not visited
                vQ.enqueue(w)
                mark w as visited
```



Complexity Analysis

$O(|V| + |E|)$

```
def BFS( V ):
    vQ = Queue()

    vQ.enqueue( V )

    while vQ not empty:
        cur = vQ.dequeue()
        print cur

        foreach w in adj(cur)
            if w is not visited
                vQ.enqueue(w)
                mark w as visited
```

$O(|V|)$

$O(adj(V))$

$O(|E|)$

Other Traversal Methods?

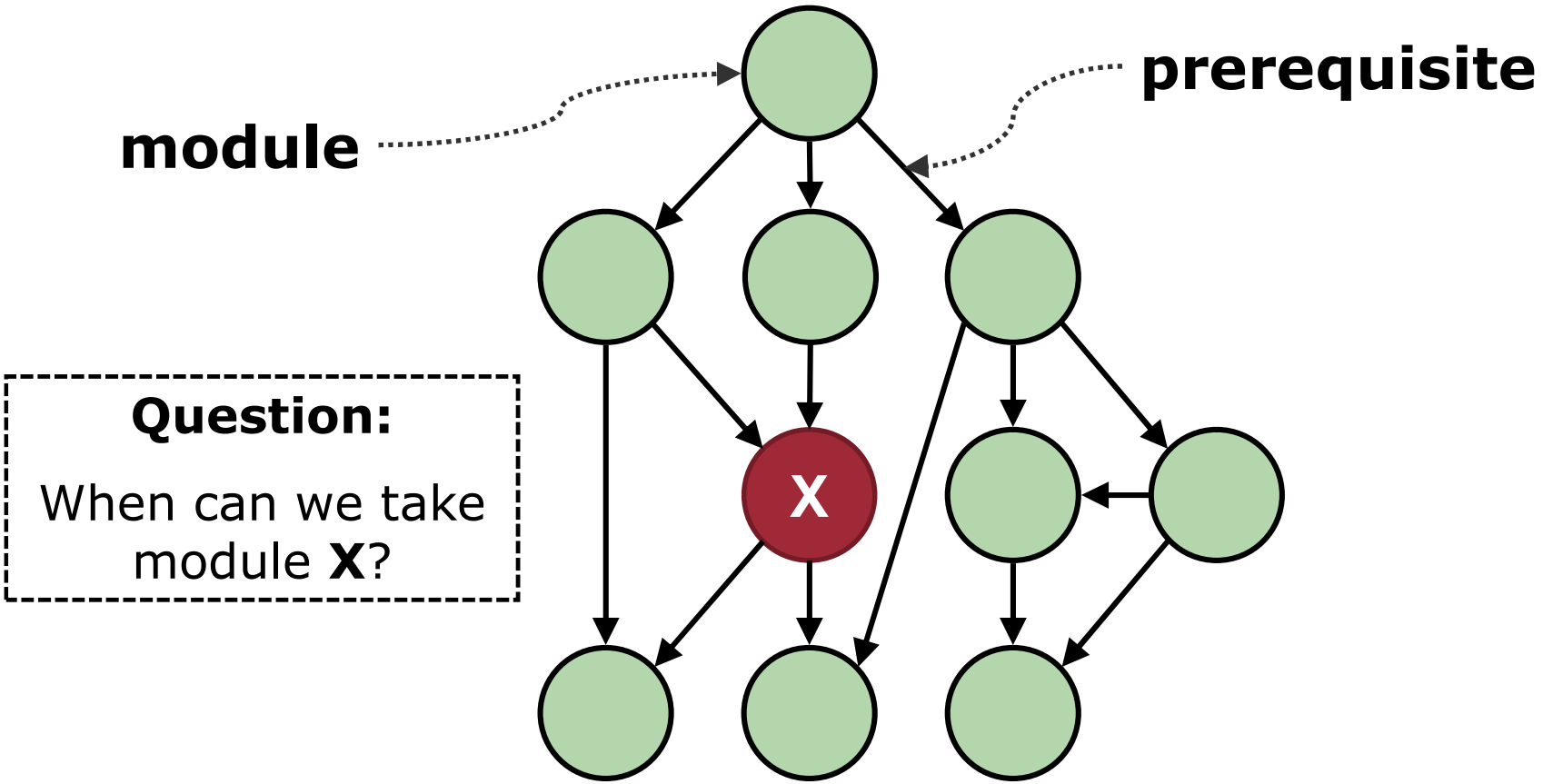
- Instead of visiting nodes of equal distance, we can choose to go as **deep as possible**
  - When we hit a deadend, we can backtrack and then explore another branch in the path
  - Known as **depth-first search (DFS)**

- Interestingly, a minor change of code + a different data structure will transform **BFS** into a **DFS**!
  - Give it a try!

I need to do A and B first, then followed by C, E but not D…..

TOPOLOGICAL SORT

Module Selection: Example



Question:

When can we take module **X**?

Topological Sort: **Idea**

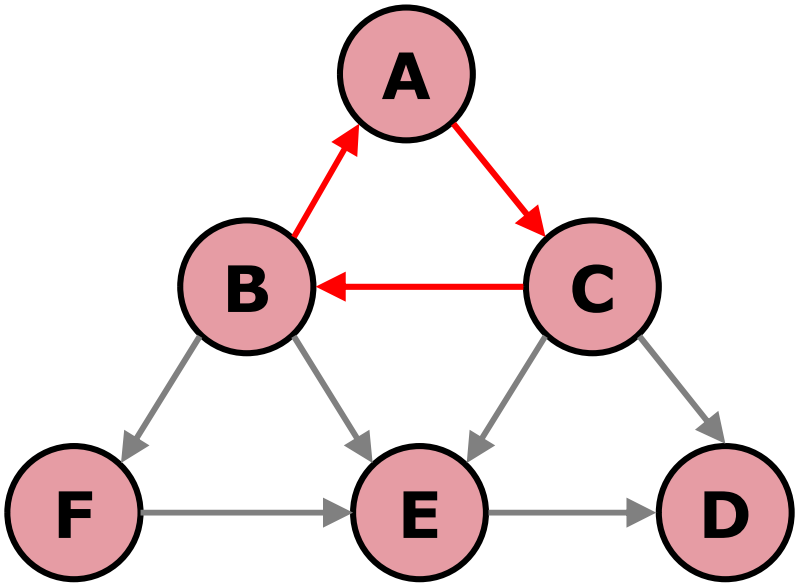
- An ordering on the vertices, such that:
  - If there is a path from ***u*** to ***v***, ***u* appears before *v*** in the sequence
  - In particular, if there is an edge from u to v, then **u appears before v** in the sequence

➔all parent vertex visited before a node is visited

- Topological sort can only be applied to certain type of graph, namely **Directed Acyclic Graph (DAG)**

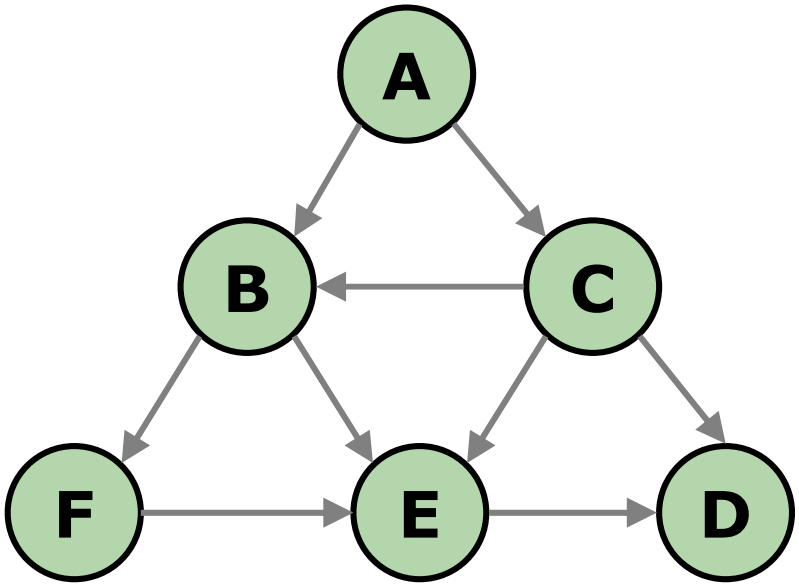
Directed Acyclic Graph: **Example**

- **Directed Acyclic Graph (DAG)**
  - A directed graph with no cycle



Not DAG

Topological Sort not possible (why?)

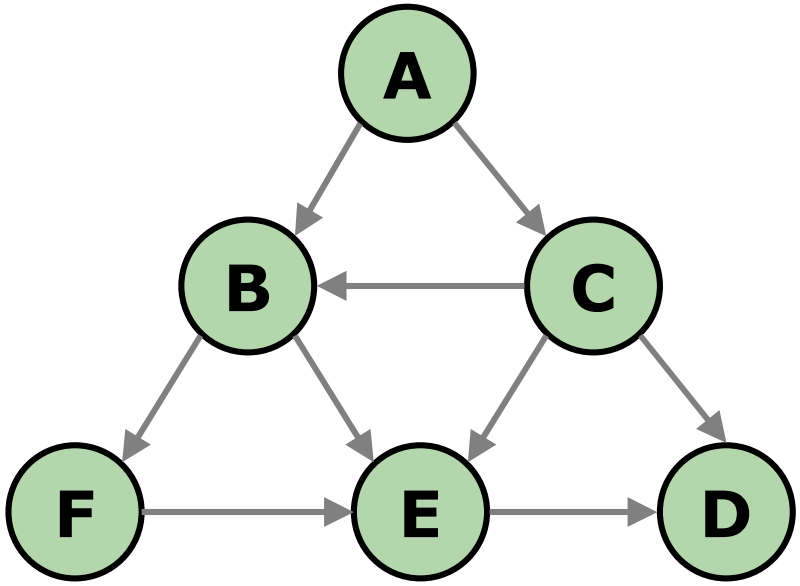


DAG

Topological Sort can be performed

Topological Sort: **Example**

- Topological Sort example results:
  - **ACBEFD**
  - **ACBEDF**
- **Incorrect** results:
  - **ACDBEF**

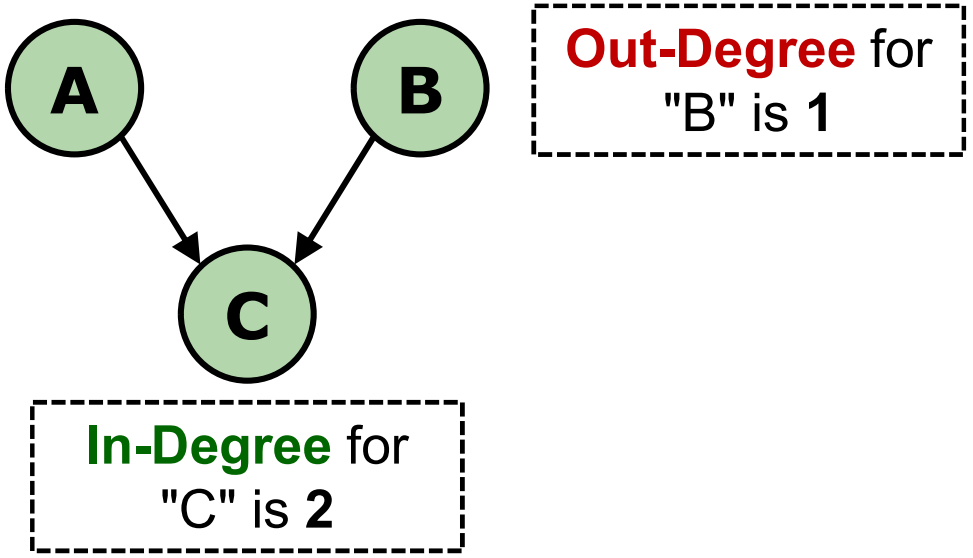


- How do we know which result is possible?
  - The reasoning forms the basis of topological sort algorithm

In- Out- Degree: Definition

- **In-Degree** of a vertex
  - number of incoming edges

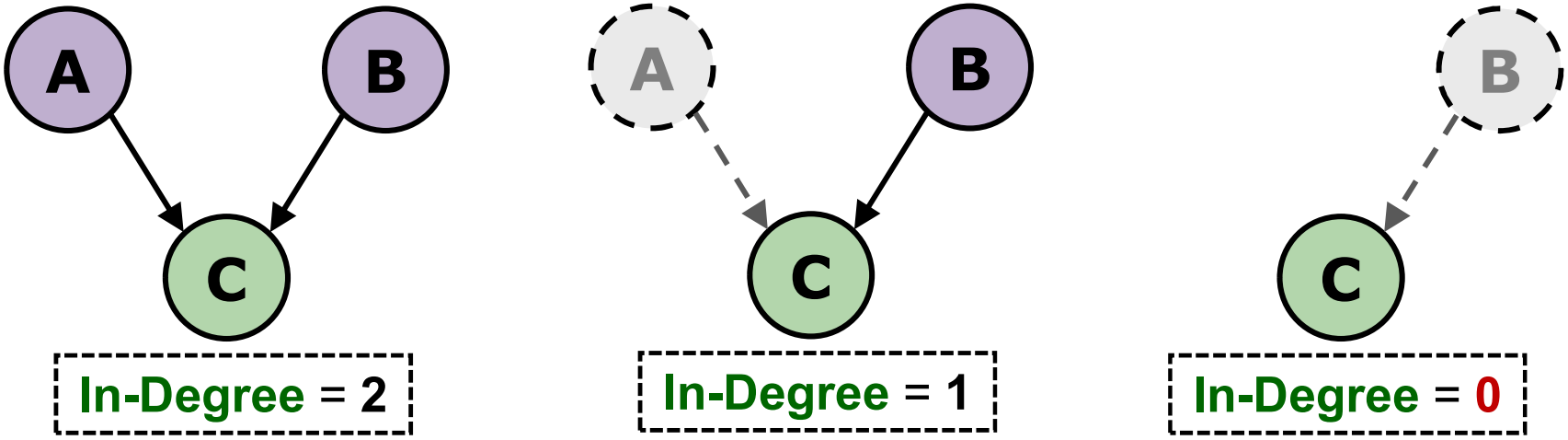
- **Out-Degree** of a vertex
  - number of outgoing edges



Topological Sort: **Main Ideas**

1. When a node is remove with its outgoing edges:
   - Its "children" will have one less incoming edge

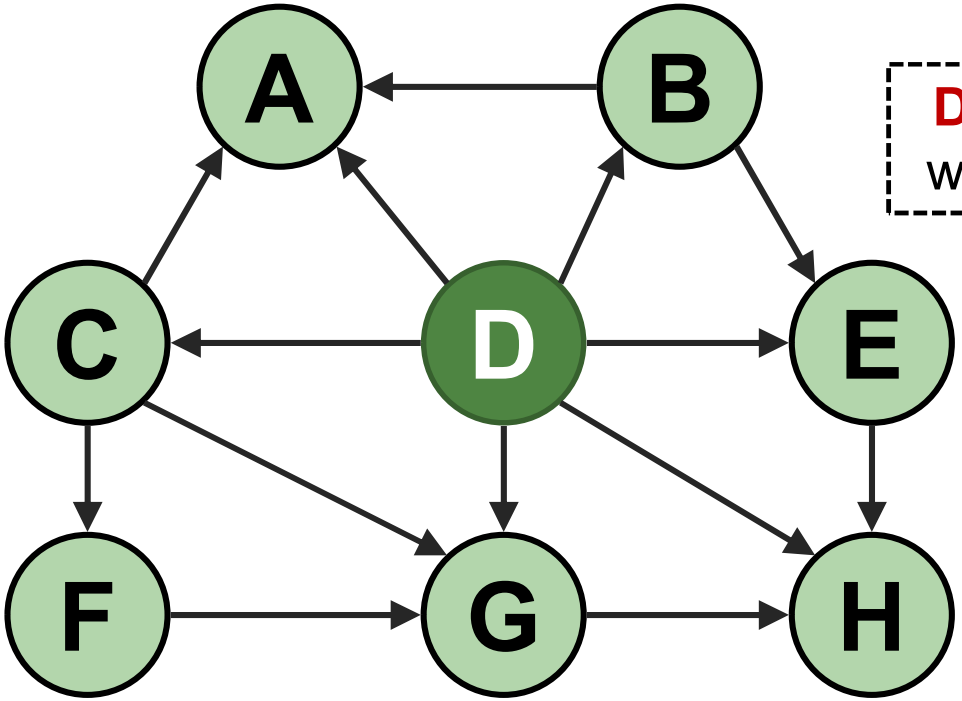
   ➔ When all parent nodes of node ***V*** are removed, then **in-degree** of ***V*** becomes **0**
2. In DAG, there must be **at least one node with 0 indegree**



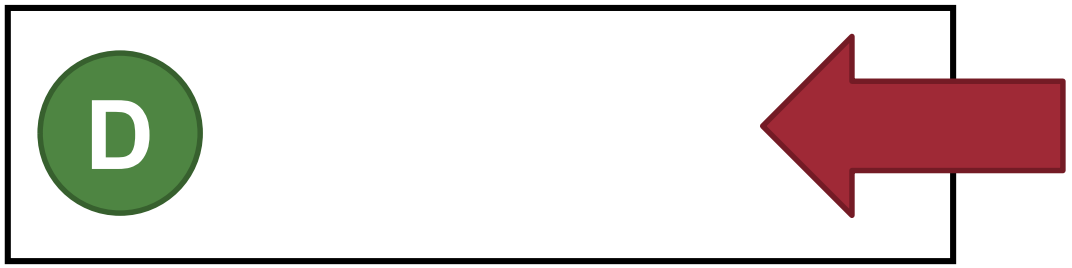
Topological Sort: **Pseudo Code**

```
def TopologicalSort( ):
    Q = Queue()
    Q.enqueue(all vertices with in-degree 0)

    while Q is not empty
        V = Q.dequeue()
        print V

        remove V and related edges from G
        Q.enqueue( adj(V) with in-degree 0 )
```

Example: Initialization



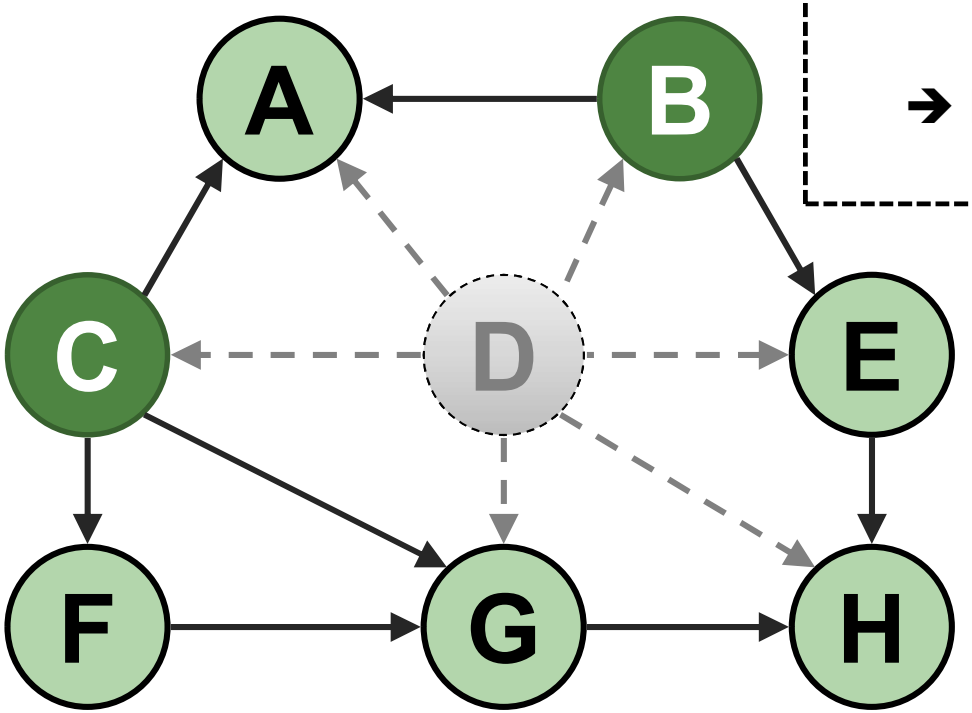
D is the only node with **in-degree = 0**

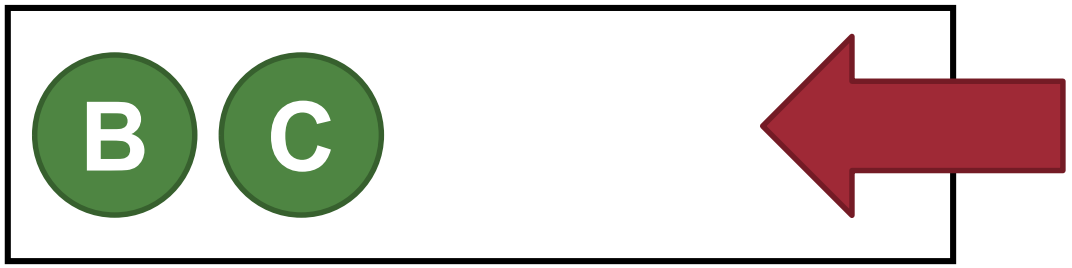


Output: D

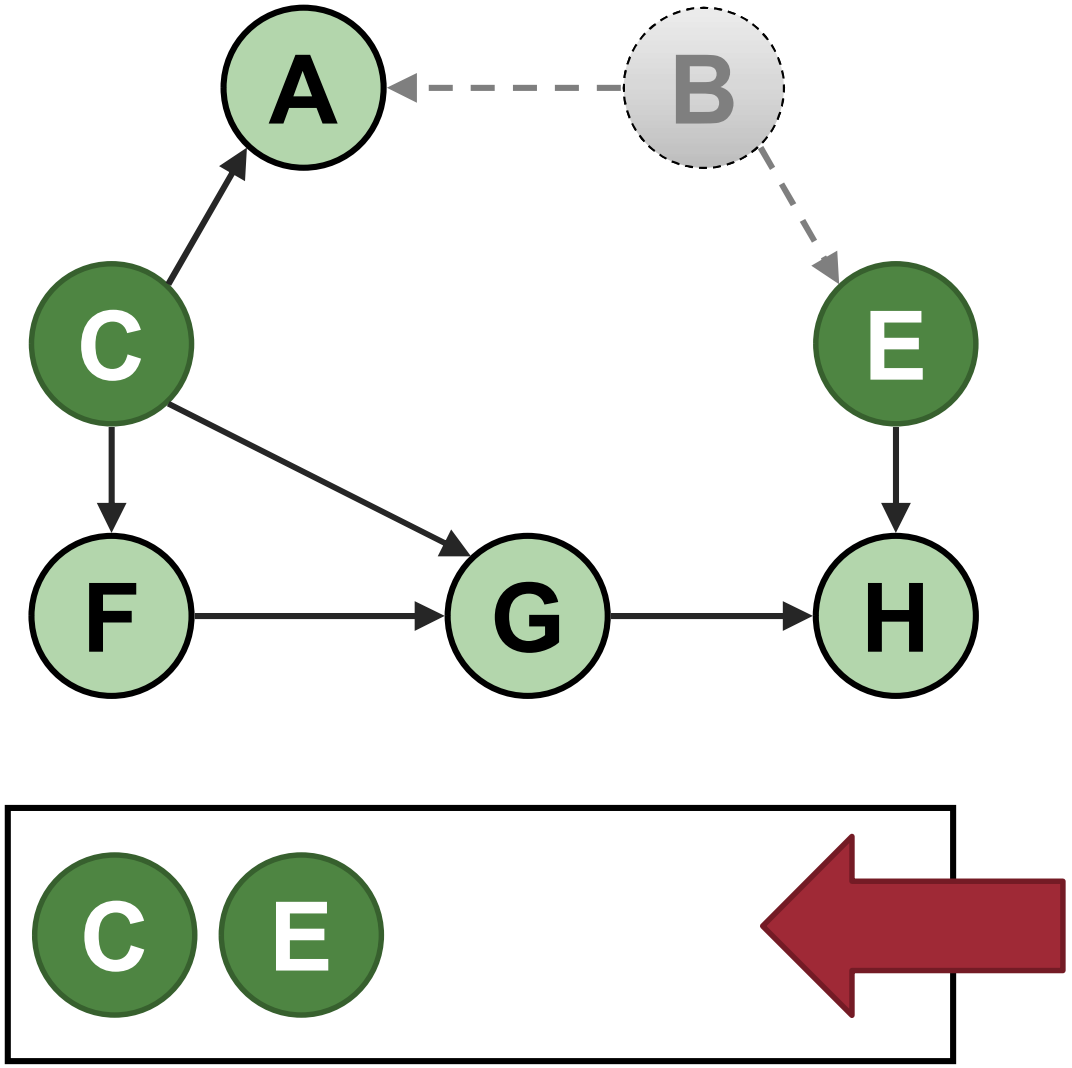
D removed from graph and printed

➔ **B** and **C** now has **0 in-degree**

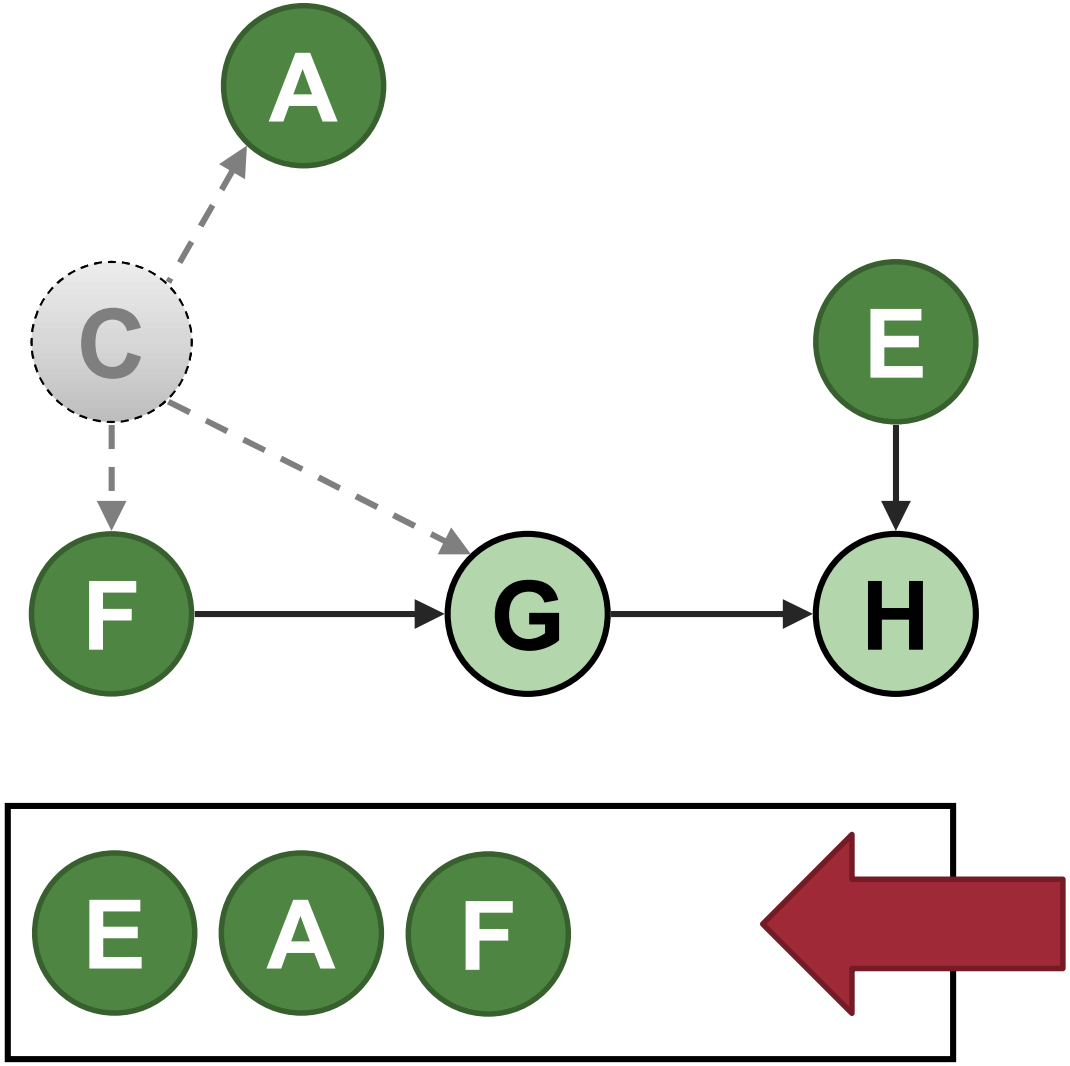




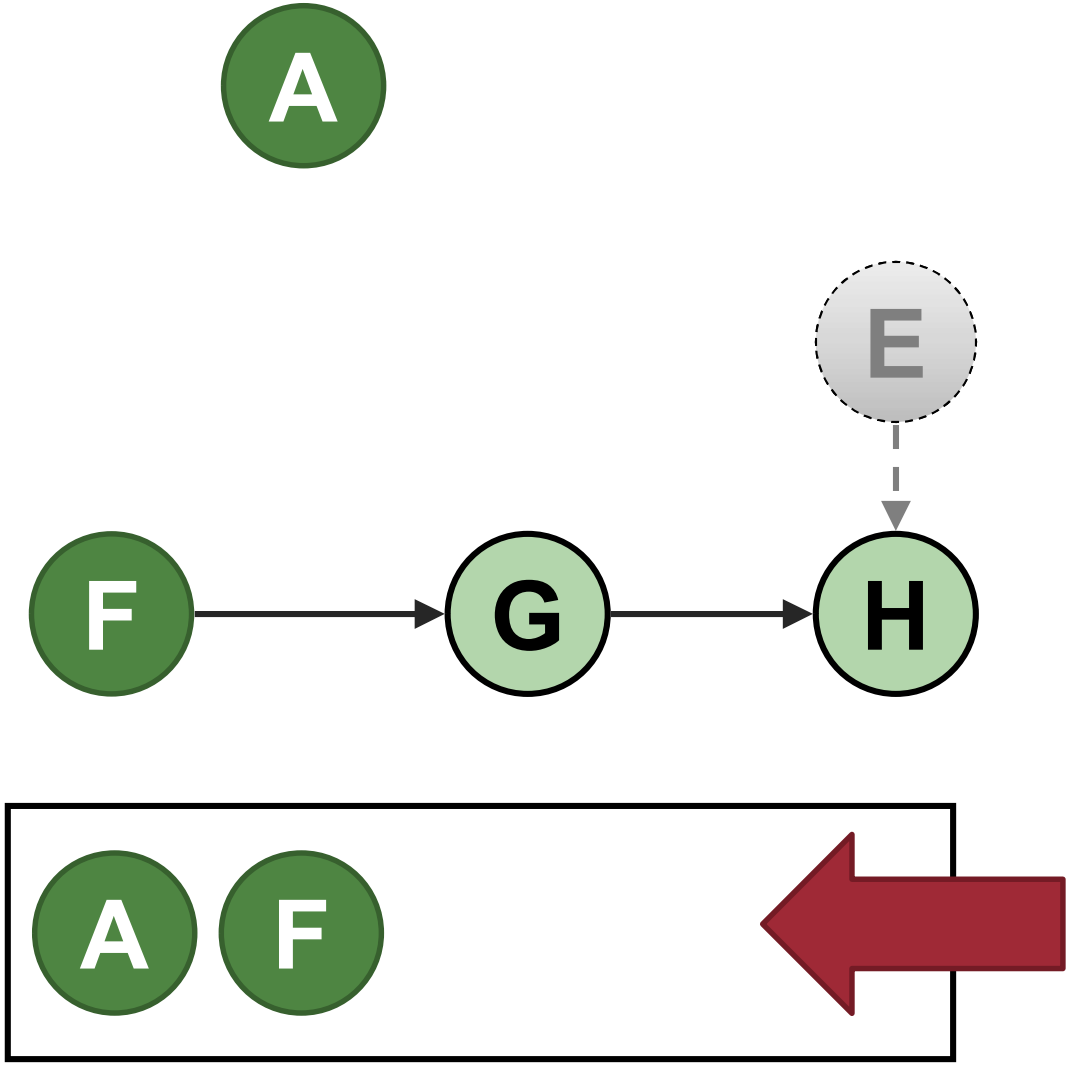
Output: **DB**



Output: **DBC**

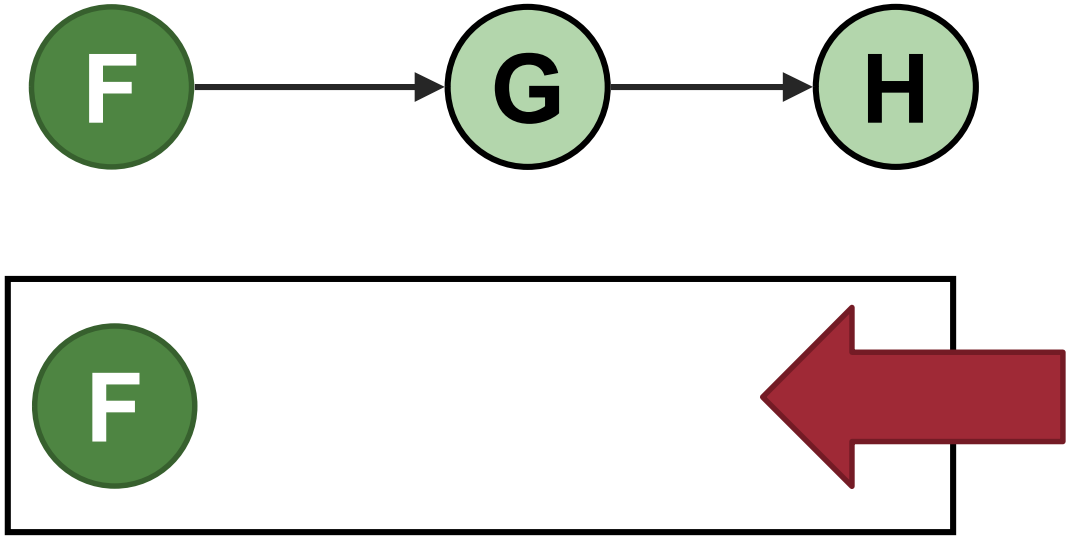


Output: **DBCE**



Output: **DBCEA**

- The rest is straightforward
- Final Output: **DBCEAFGH**



A to B in shortest time!

SHORTEST PATH

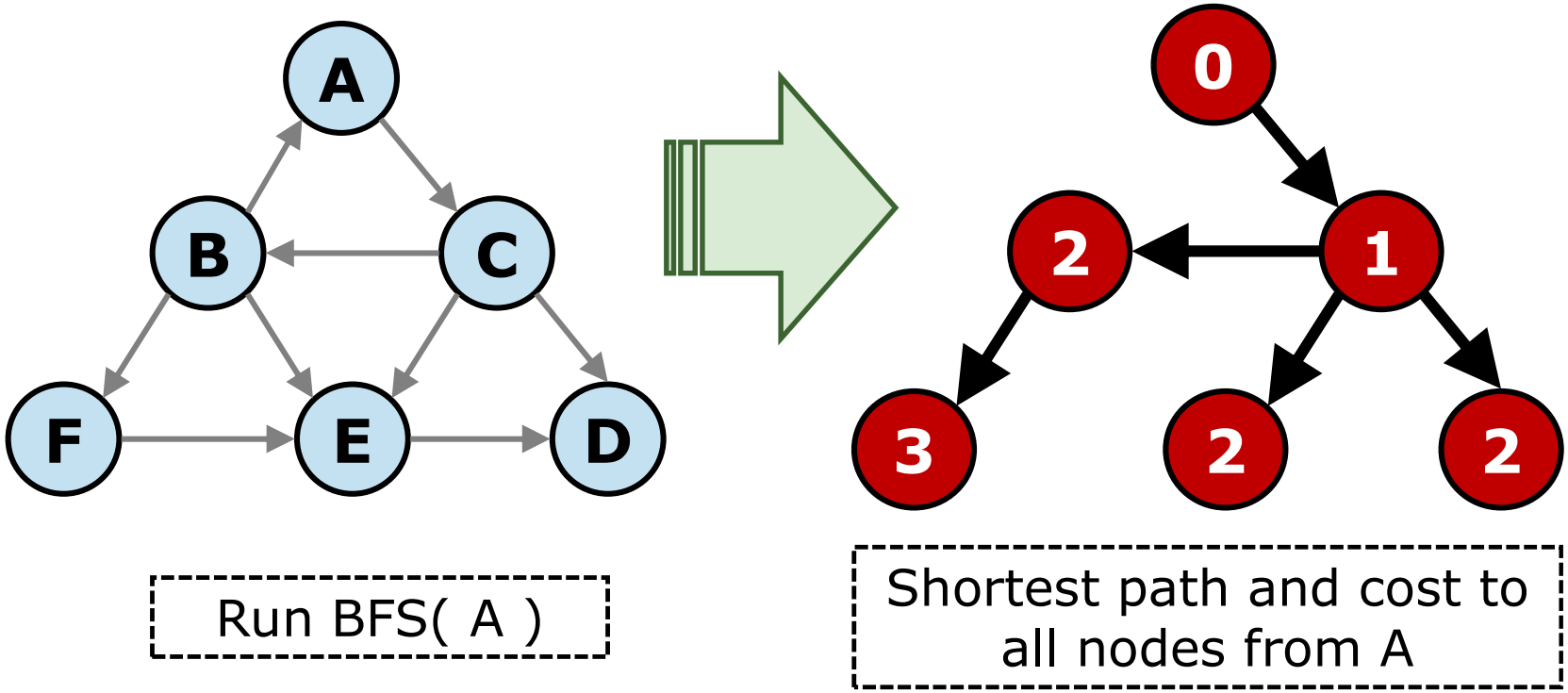
Definition: **Single Source Shortest Path**

- Each path has an associated **cost**:
  - The sum of weight of all edges in the path
  - For **unweighted graph**:
    - Equivalent to the number of edges in the path

- **Single Source Shortest Path problem**:
  - Given a vertex **V**, find the path with minimum cost to **all other vertices**
  - We are interested in **both the path and the cost**

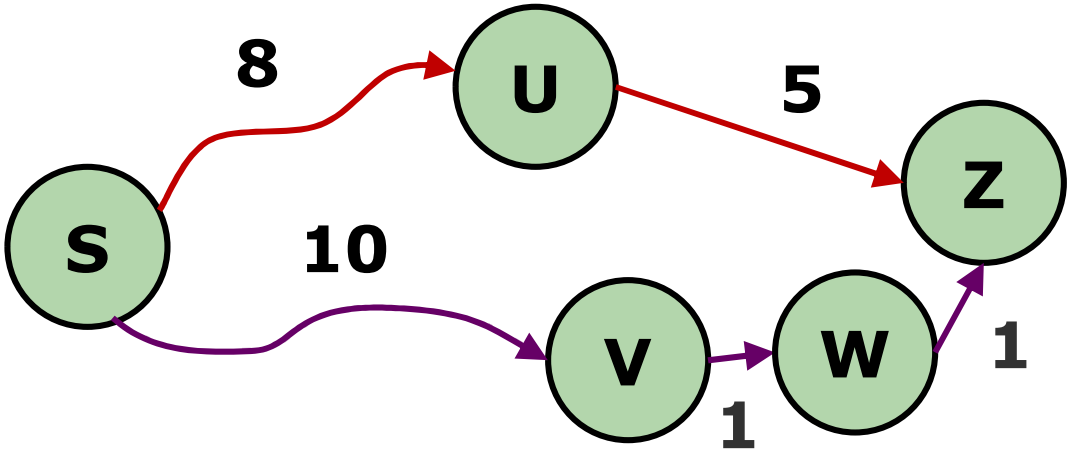
SSSP: Unweighted Graph

- Use **Breath-First Search** on source vertex **V**:
  - Compute the “cost” as we progress



SSSP: **Weighted Graph**

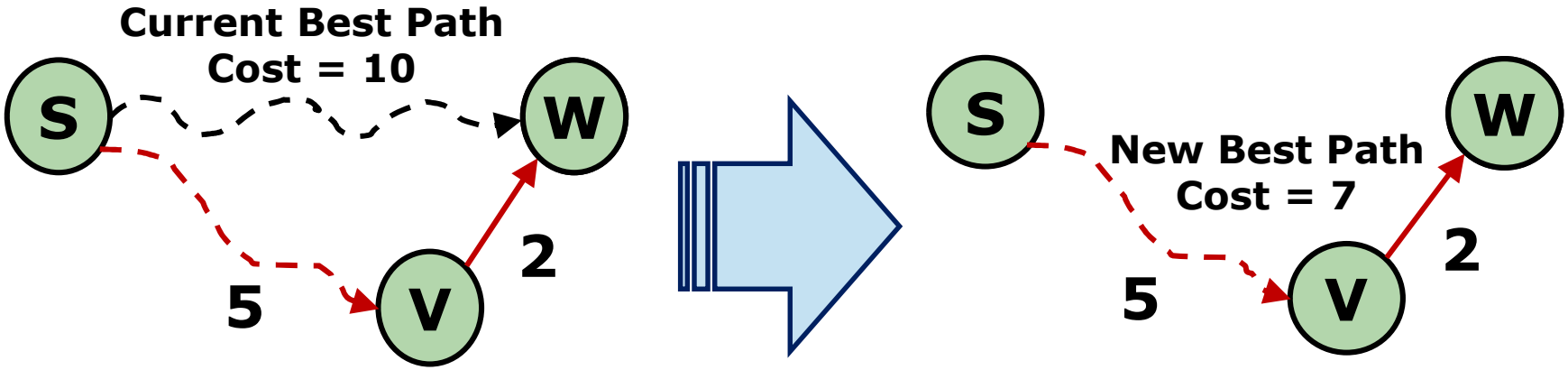
- Weighted graph is trickier:
  - The number of edges != The total cost



- S ➔U ➔ Z = 2 edges, **cost = 8+5 = 13**
- S ➔ V ➔ W ➔ Z = 3 edges, **cost = 10+1+1 = 12**

SSSP: Main Idea - **Relaxation**

- Instead of fixing the path of a vertex:
  - We should progressively update the path as cheaper alternative is found!
  - Known as **Relaxation**

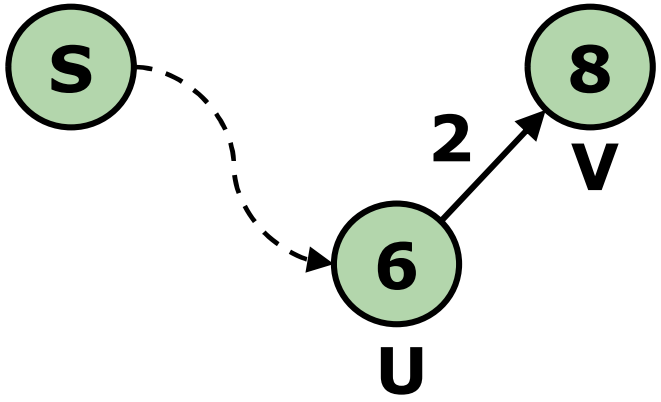


Cheaper path found via V to W

Relax the old path and use the new discovery!

Definitions

- Given a source vertex **S**
  - **distance**(**V**) = shortest distance so far from **S** to **V**
  - **parent**(**V**) = previous vertex on the current shortest path from **S** to **V**
  - **weight**(**U**, **V**) = weight of the edge from **U** to **V**



distance (U) = 6

distance(V) = 8

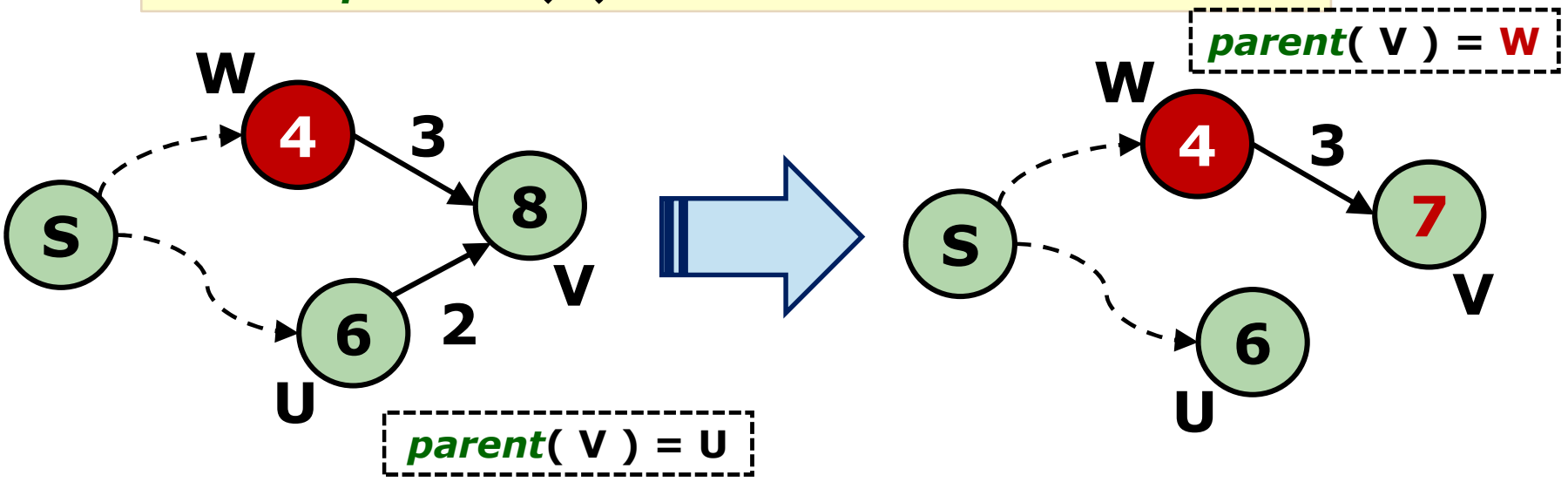
parent(V) = U

weight(U, V) = 2

Relaxation Technique: **Pseudo Code**

- When we encounter a new vertex **W**, check for better path via **W** to reach **V**:

```
def Relax( W, V ):
    d = distance(W) + weight(W, V)
    if distance(V) > d:
        distance(V) = d
        parent(V) = W
```



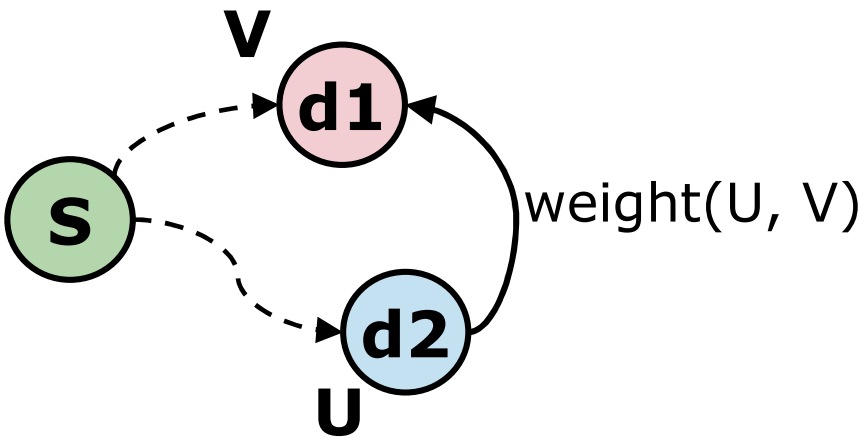
Observation So Far

- It is not hard to imagine a brute force approach:
  - Go through each vertex in the graph
    - Repeatedly relax all known paths
  - ➔ Extremely inefficient!

- **Dijkstra** made a **key observation for positive weighted graph**
  - There is a way to fix the path for some vertices (i.e. the path is impossible to get any better)
  - Remove them from future computation ➔ Much more efficient!

Dijkstra's Key Observation

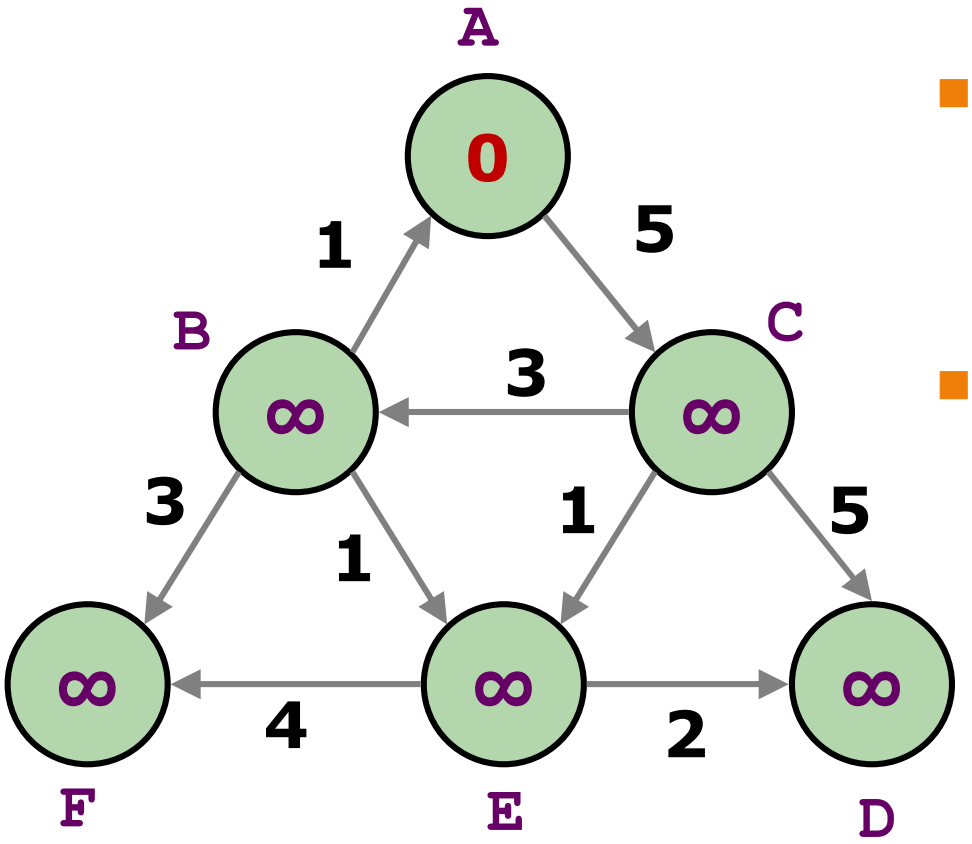
- When choosing a vertex to visit next:
  - We should choose a vertex **V** with currently known **minimum distance from source S**
  - ➔ It is guaranteed that there is NO better path to **V**
  - ➔ **Distance(V) can be fixed and remove from consideration!**



Given `d1 <= d2`, then

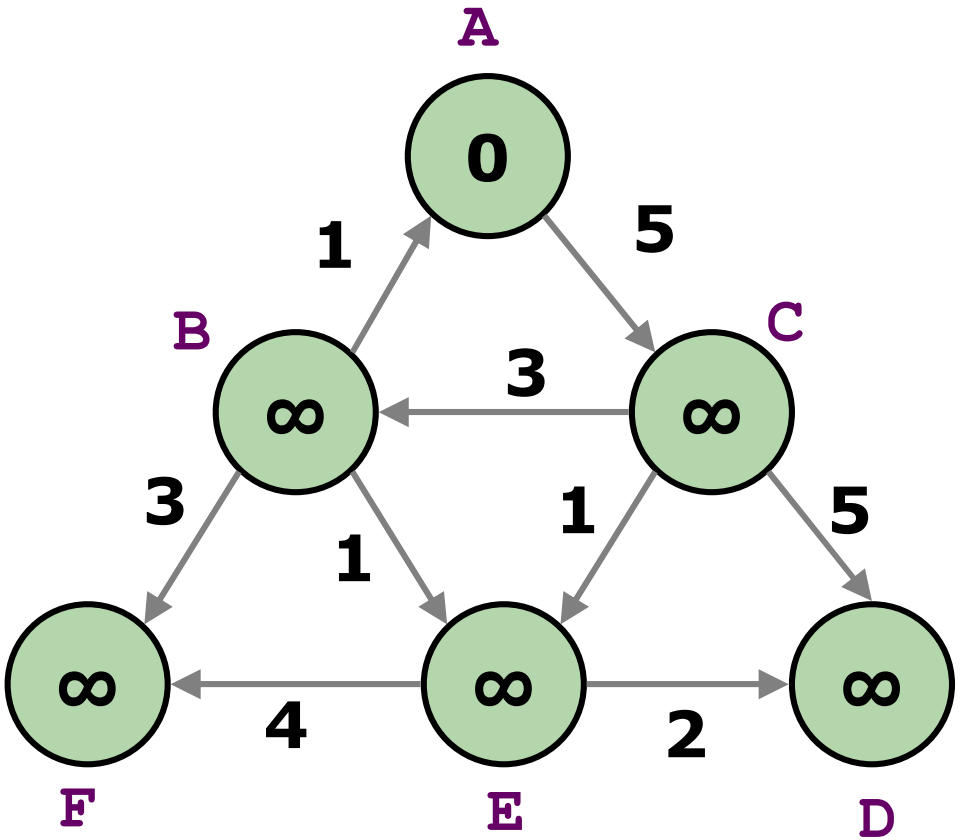
`d2 + weight(U, V) > d1`

Dijkstra's Algorithm: **Setup (1/2)**



- A is the source vertex
  - Distance(A) = 0
- Initialize distance to all other vertices as **infinity**

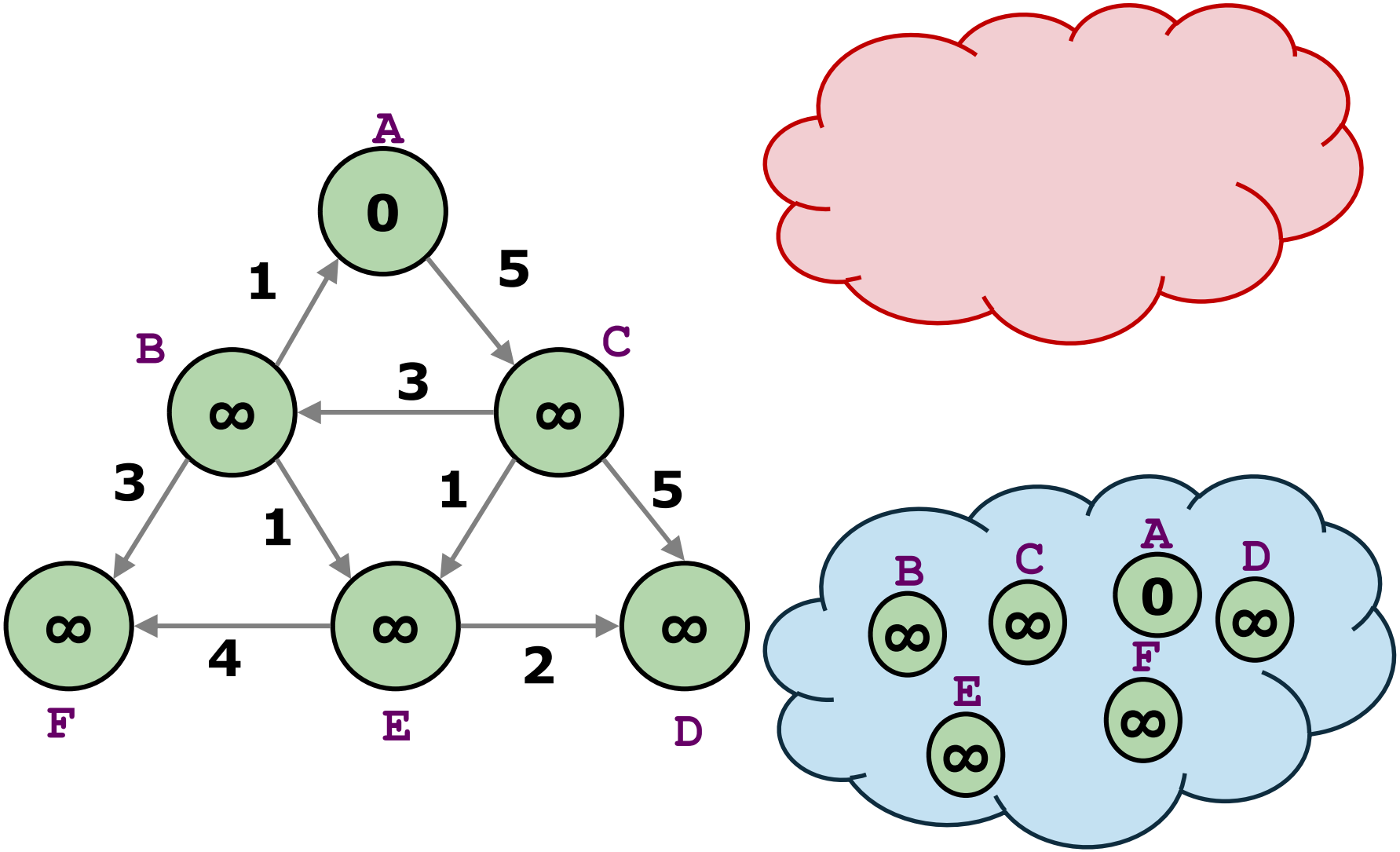
Dijkstra's Algorithm: **Setup (2/2)**



Fixed Vertices
(Vertices with best path)

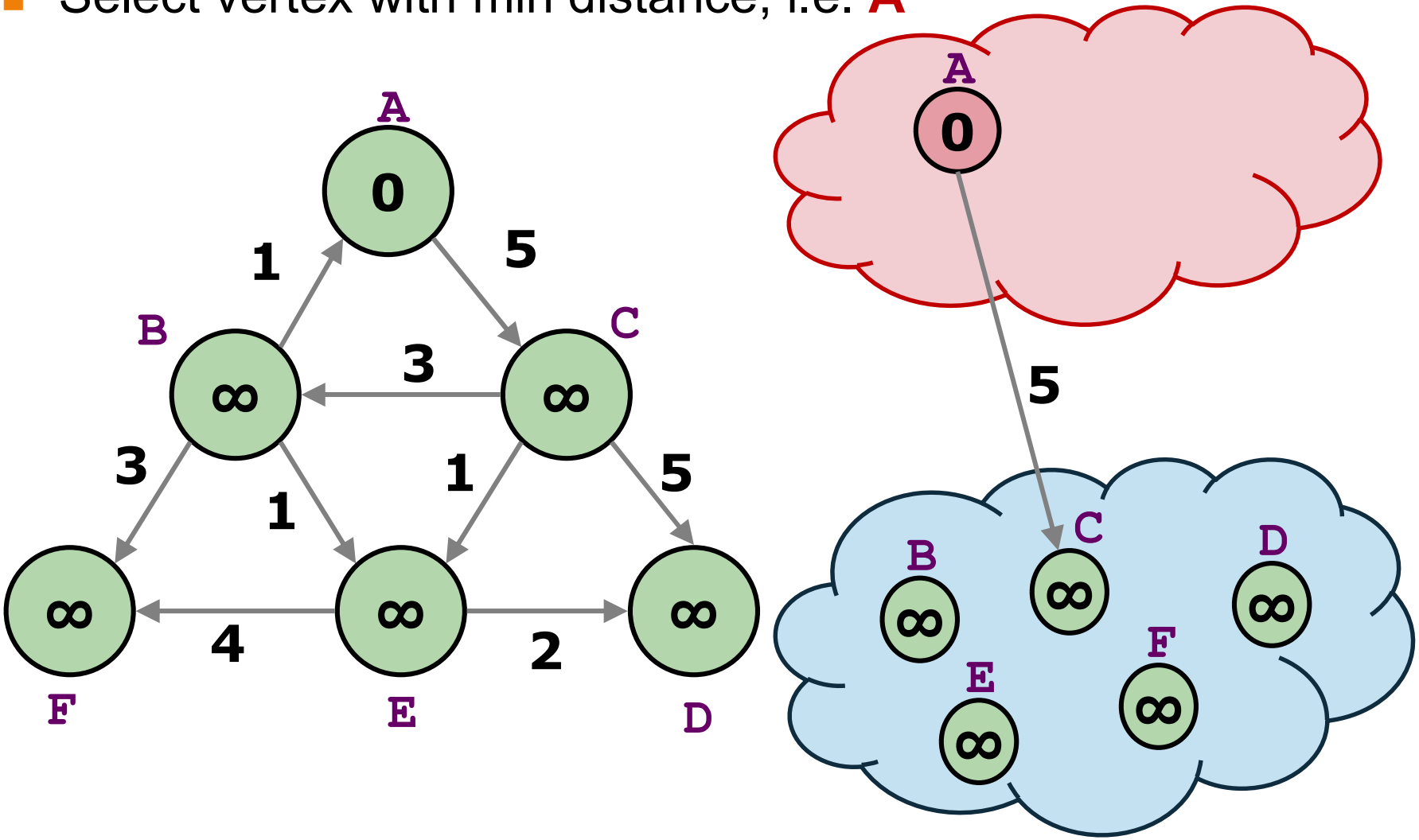
Non-fixed Vertices
(Best path still being determined)

Dijkstra's Algorithm: **Example (1/10)**



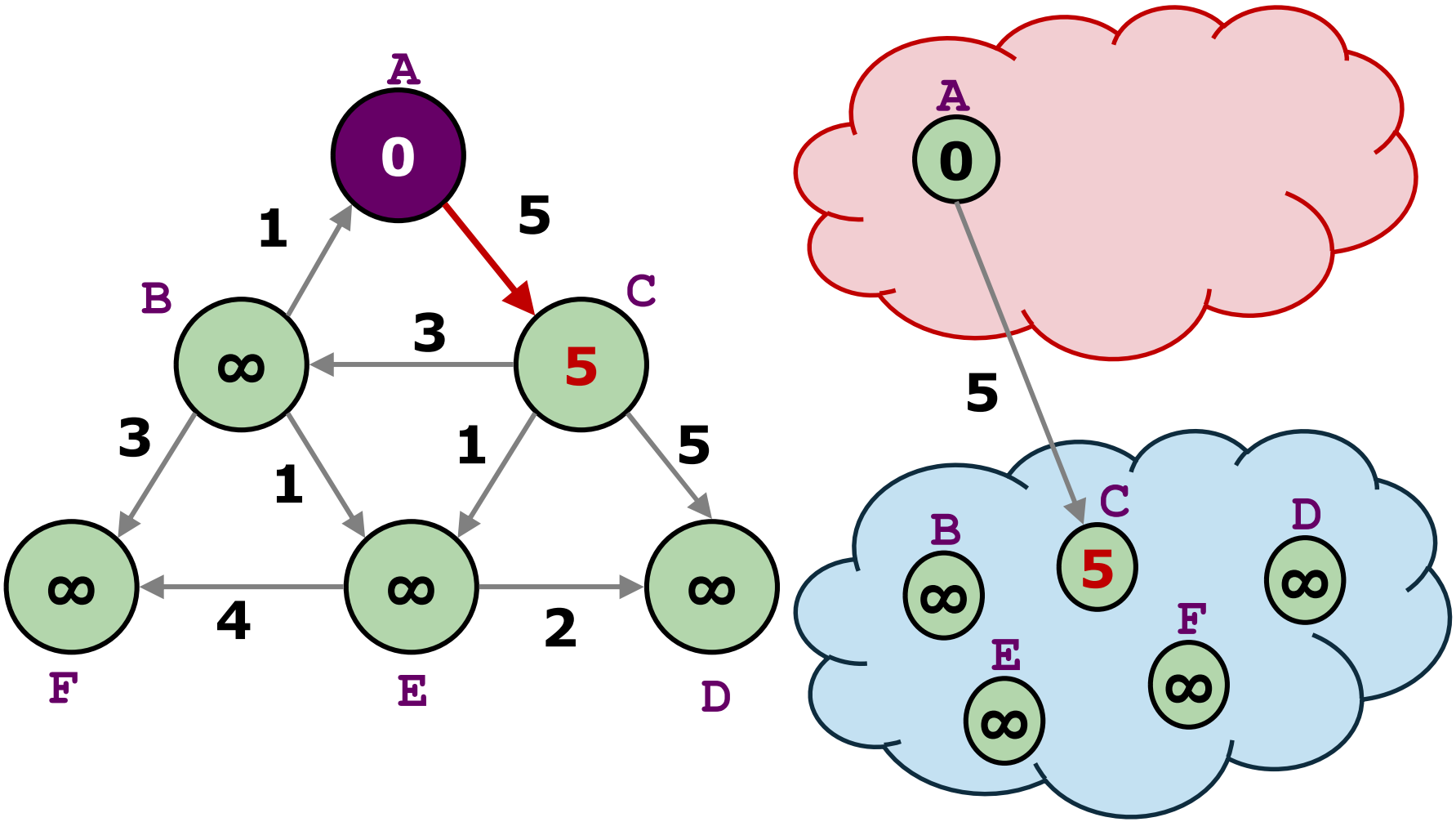
Dijkstra's Algorithm: **Example (2/10)**

- Select vertex with min distance, i.e. **A**



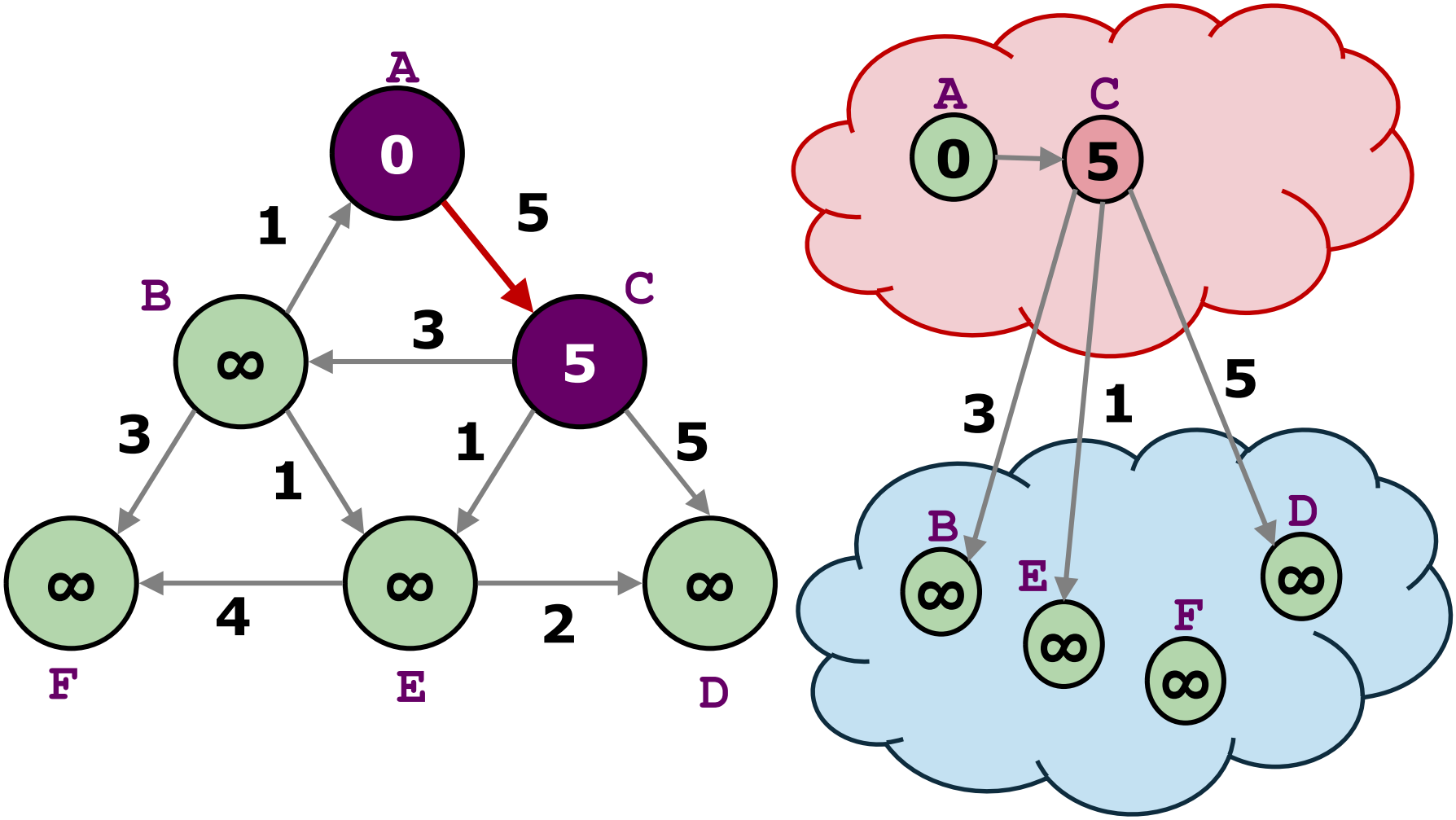
Dijkstra's Algorithm: **Example (3/10)**

- Relax neighbors of **A**



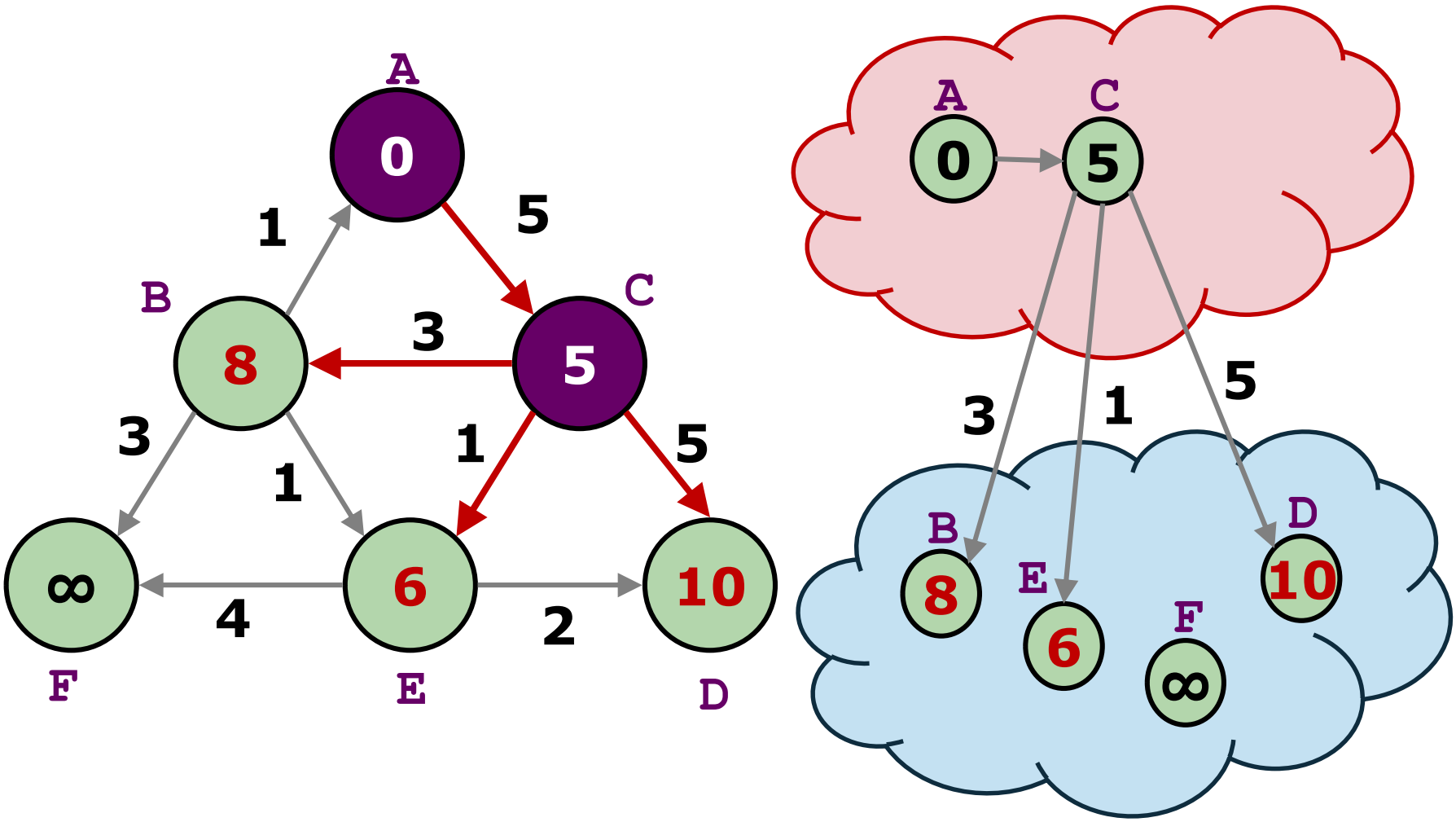
Dijkstra's Algorithm: **Example (4/10)**

- **C** is the next vertex with min distance



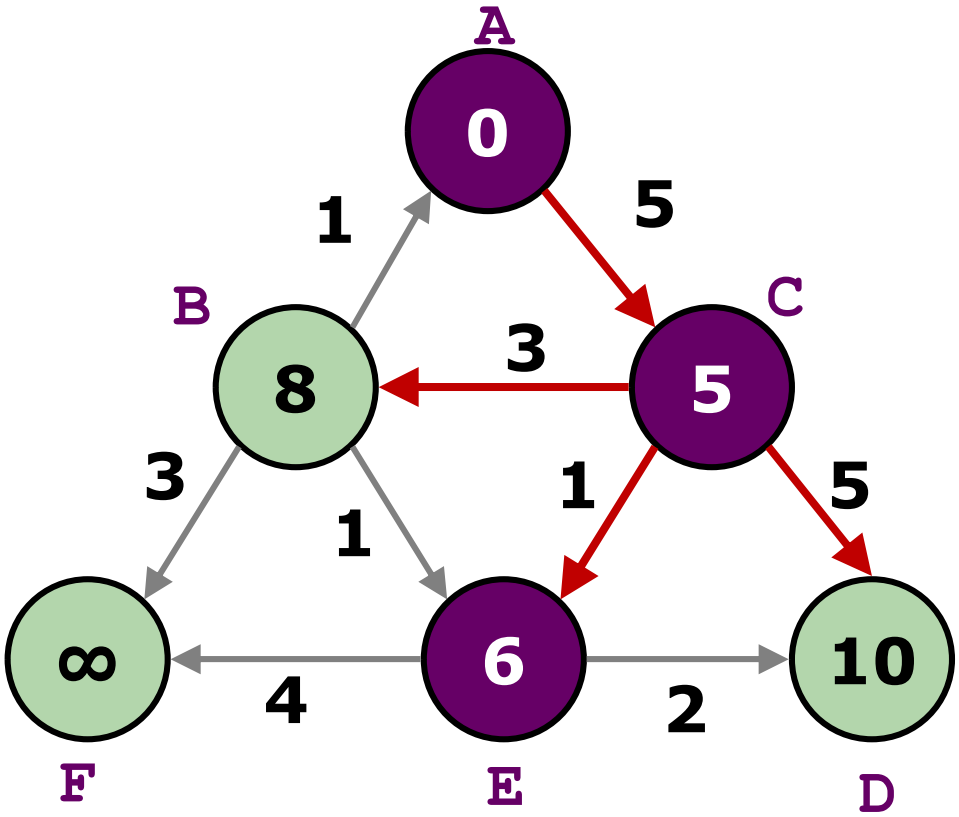
Dijkstra's Algorithm: **Example (5/10)**

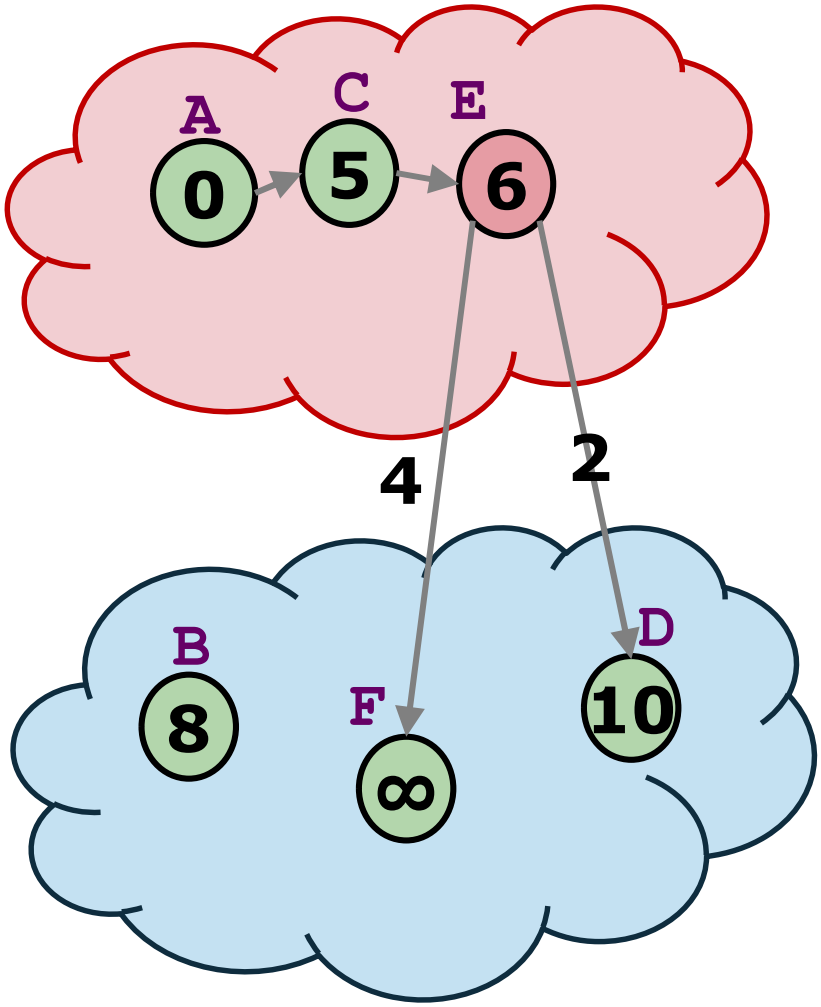
- Relax the neighbors



Dijkstra's Algorithm: **Example (6/10)**

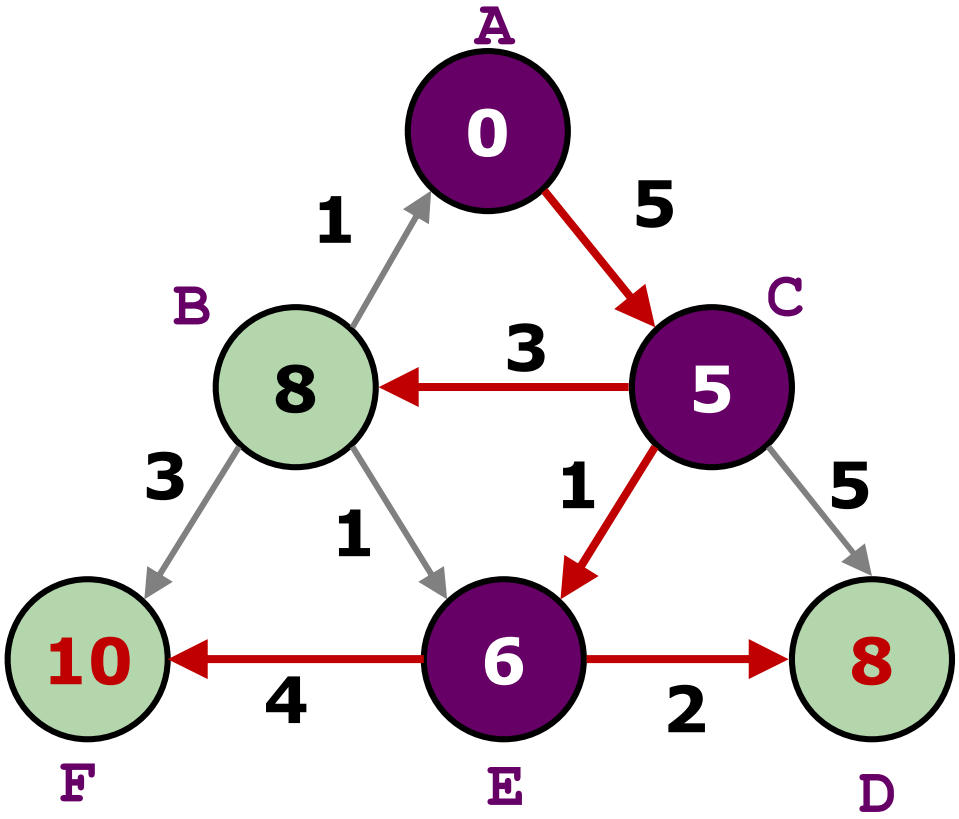
- **E** is the next target

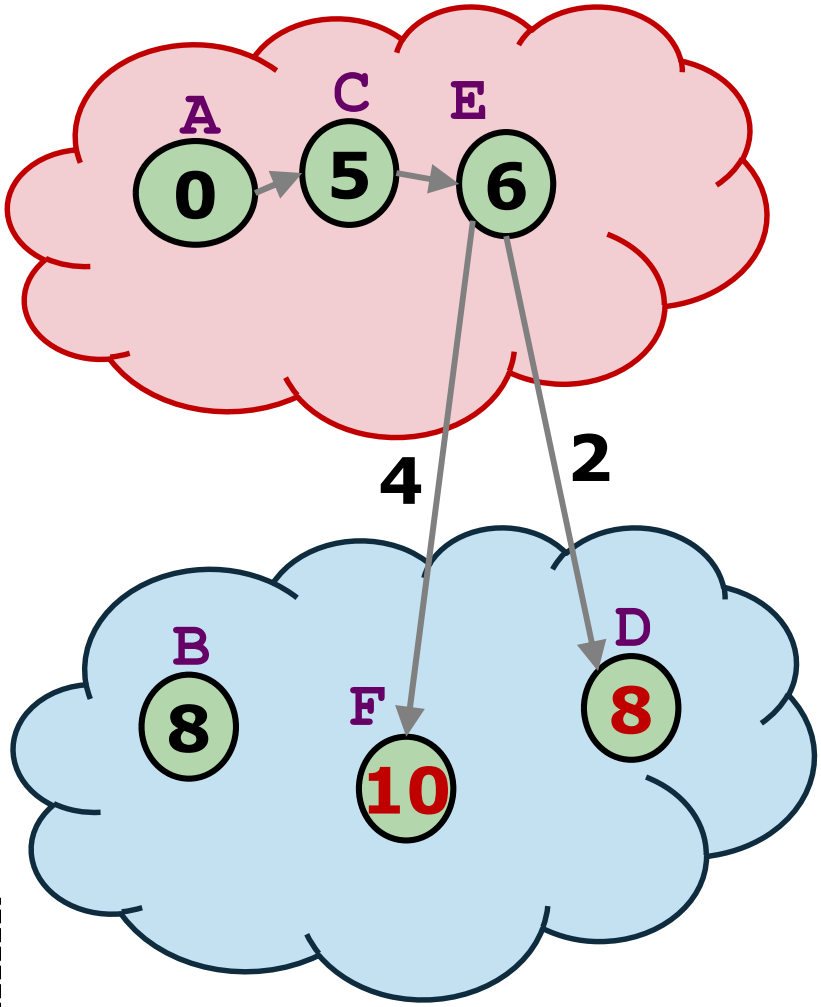




Dijkstra's Algorithm: **Example (7/10)**

- Relax the neighbors

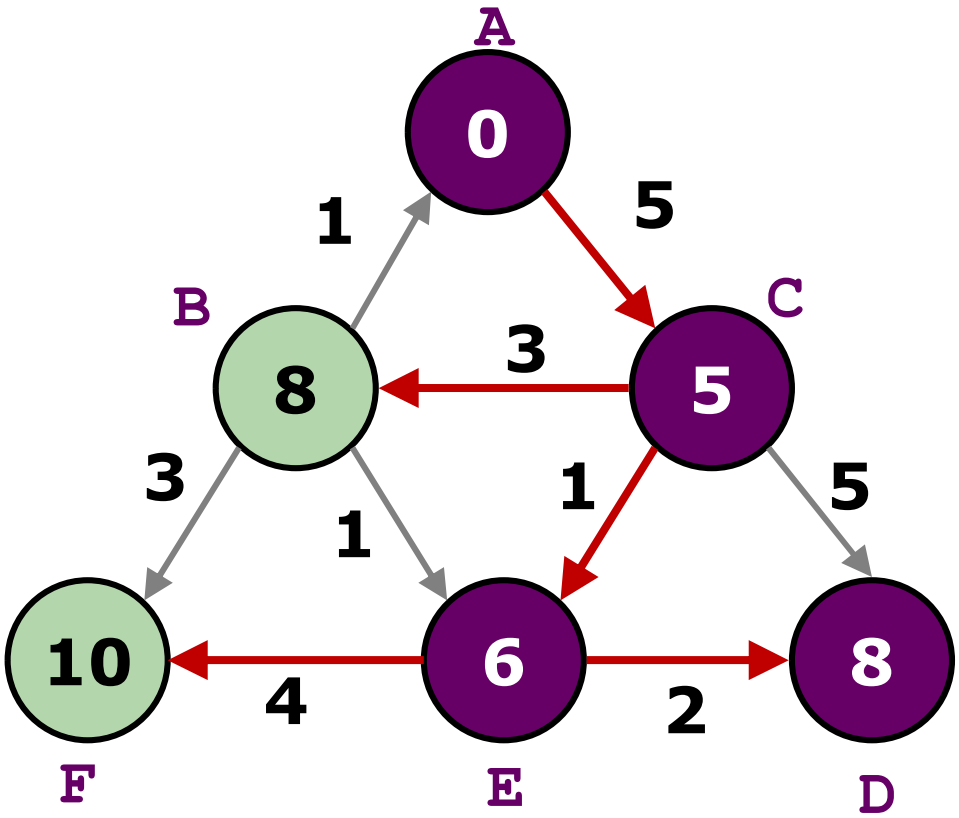


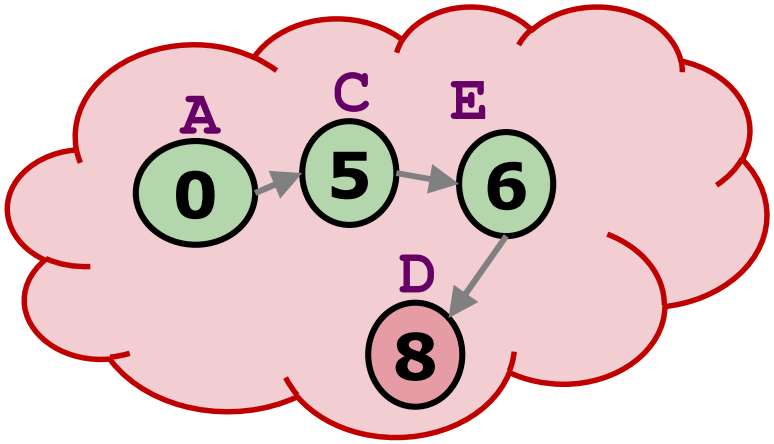


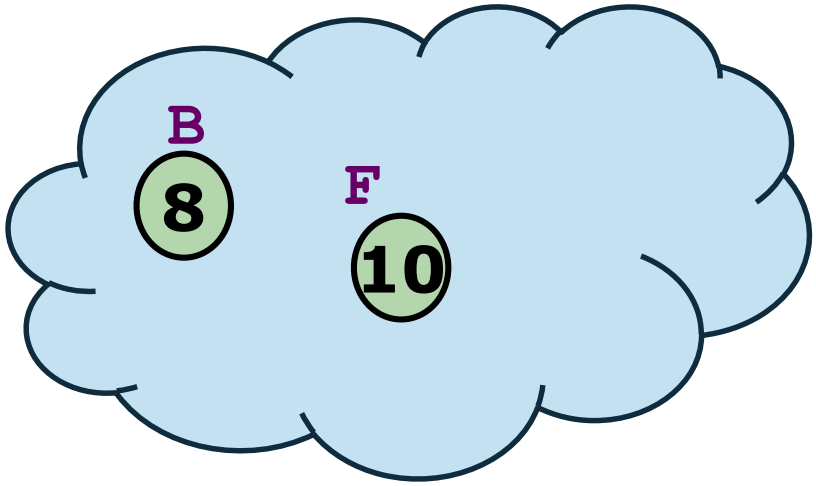
Note the change of parent(D)

Dijkstra's Algorithm: **Example (8/10)**

- **D** is the next target, no change

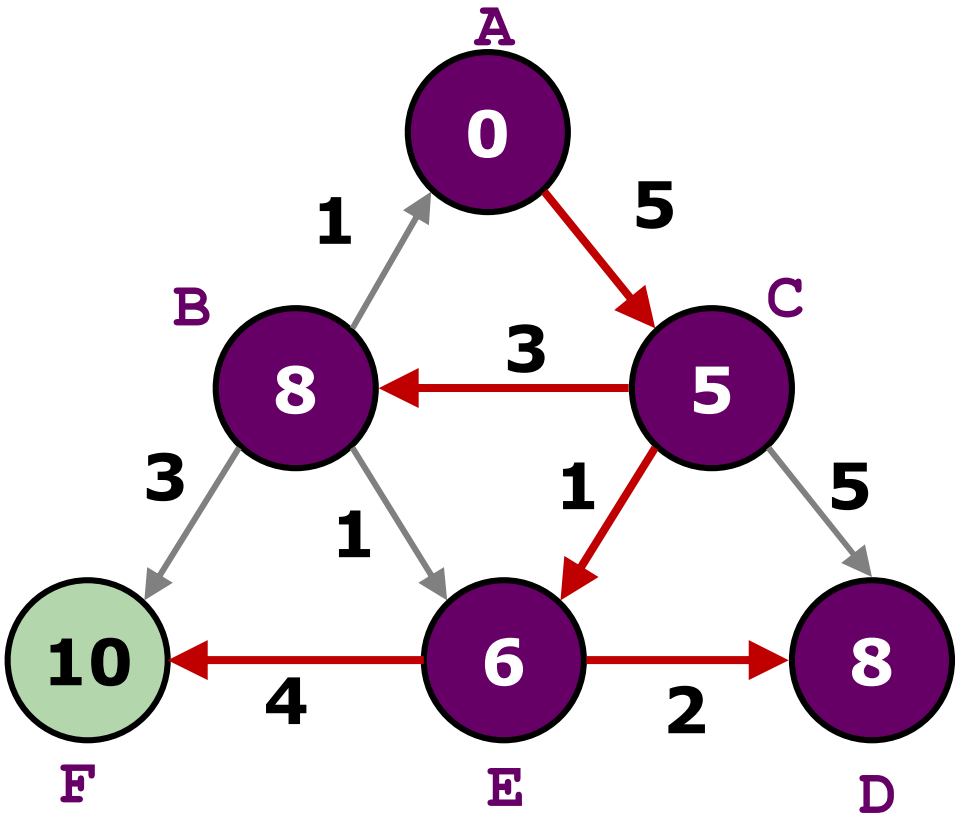


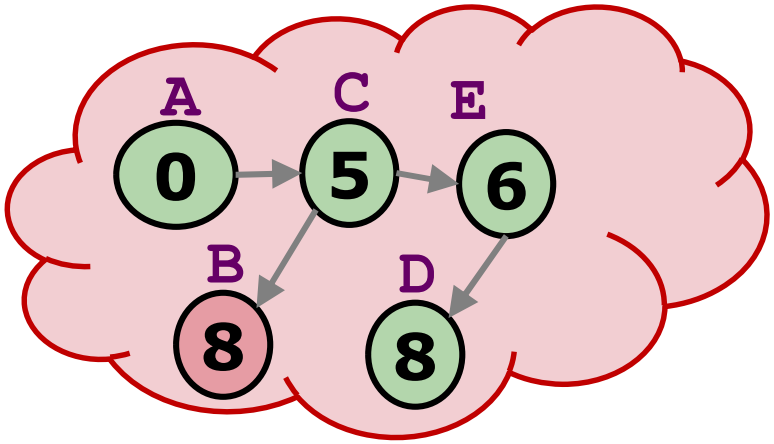


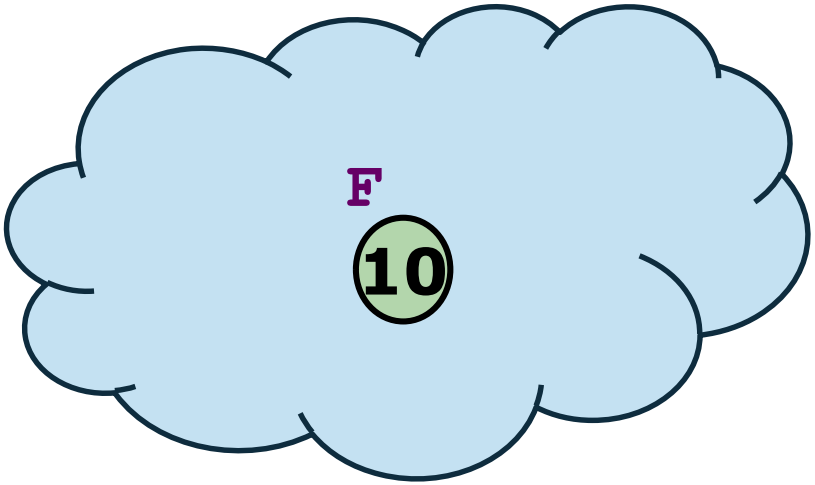


Dijkstra's Algorithm: **Example (9/10)**

- **B** is the next target, no change

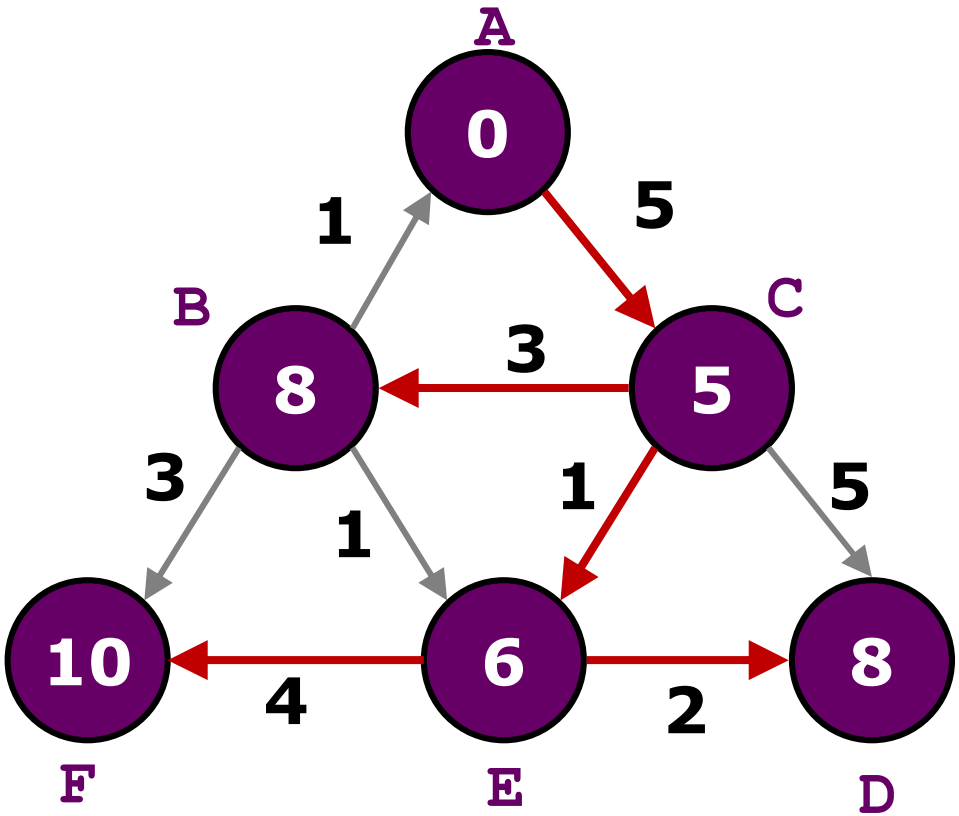


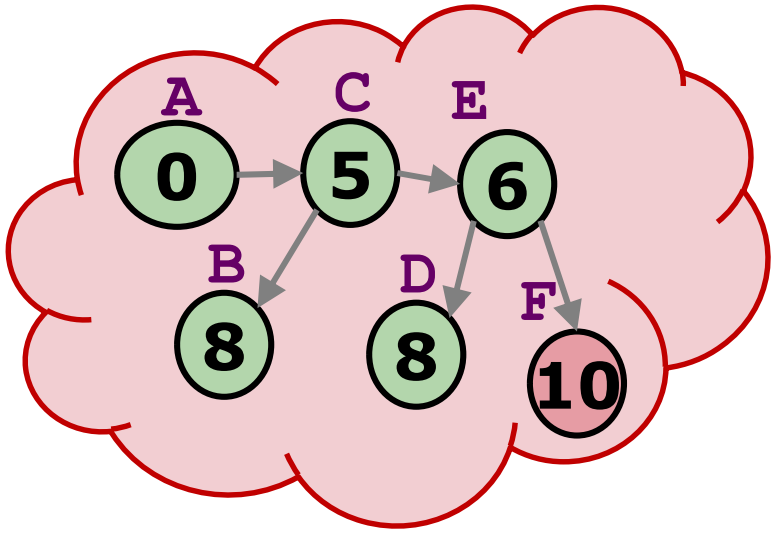


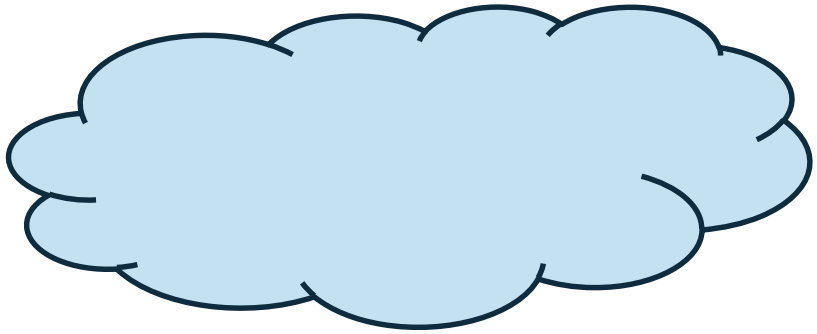


Dijkstra's Algorithm: **Example (10/10)**

- **F** is the next target, done!







Dijkstra's Shorted Path Algorithm

- It is a **single source shortest path algorithm** for **positive weighted graph**

```
def Dijkstra( S ):
    color all vertices blue
    foreach vertex w
        distance( w ) = INFINITY
    distance( S ) = 0

    while there are blue vertices
        v = blue vertex with min distance()
        color v red
        foreach neighbour w of v
            relax( v, w )
```

Complexity Analysis

$O(|V|^2 + |E|)$

```
def Dijkstra( S ):

   color all vertices blue
   foreach vertex w
      distance( w ) = INFINITY
   distance( S ) = 0

   while there are blue vertices

      v = blue vertex
              with min distance()
      color v red


      foreach neighbour w of v
         relax( v, w )
```

$O(|V|)$

$O(|V|)$

$O(|V|)$ $O(|V|^2)$

$O(adj(V))$ $O(|E|)$

Dijkstra's Algorithm: **Improvement**

- Main inefficiency = Choosing min blue vertex
  - Use **min heap** to store the blue vertices
  - Delete root get the minimum vertex will be cheaper!

```
def Dijkstra( S ):
    foreach vertex w
        distance( w ) = INFINITY
    distance( S ) = 0
    mh = minHeap( vertices with distance as key)

    while mh is not empty
        v = mh.delete()
        foreach neighbour w of v
            relax( v, w )
```

Complexity Analysis II

$O((|V| + |E|)\, lg|V|)$

```
def Dijkstra( S ):

    foreach vertex w
        distance( w ) = INFINITY
    distance( S ) = 0
    mh = minHeap( vertices with distance as key)


    while mh is not empty

        v = mh.delete()


        foreach neighbour w of v

            relax( v, w )
```

$O(|V|)$

$O(|V|)$

$O(lg\,|V|)$

$O(|V|\, lg\,|V|)$

$O(adj(V))$

$O(|E|\, lg\,|V|)$

$O(lg\,|V|)$

Complexity Analysis II

- Since the blue vertices are now kept in a min-heap:
  - Using ***distance***() as the key
  - ***relax***(v, w) may change the distance of blue vertices
  - ➔ Key change = Heap property can be violated!

- Recall that we only need a simple bubbling operation to restore heap property:
  - **O(lg N)** where N is number of nodes

  ➔ **O(lg |V|)** in this case

Summary

- Graph can be used to model more complex relationship
- Many representations possible for graph
  - Mainly representing the {V, E, w} in some ways
- Algorithms discussed:
  - BFS, Topological Sort, Single Source Shortest Path
- Many advance algorithms possible!