Lecture 12
Graphs

Department of Computer Science
Hofstra University

Acknowledgement: Lecture slides based on UofW Course on Data Structures

Inter-data Relationships

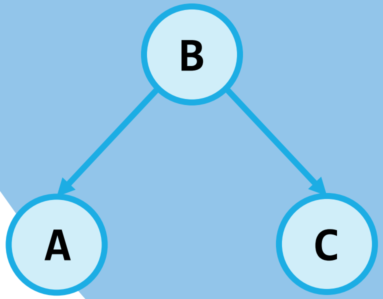
Arrays

- Elements only store pure data, no connection info
- Only relationship between data is order

0	1	2
A	B	C

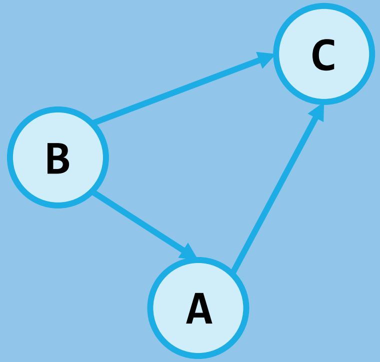
Trees

- Elements store data and connection info
- Directional relationships between nodes; limited connections



Graphs

- Elements AND connections can store data
- Relationships dictate structure; huge freedom with connections



Applications

Physical Maps

- Airline maps
  - Vertices are airports, edges are flight paths
- Traffic
  - Vertices are addresses, edges are streets

Relationships

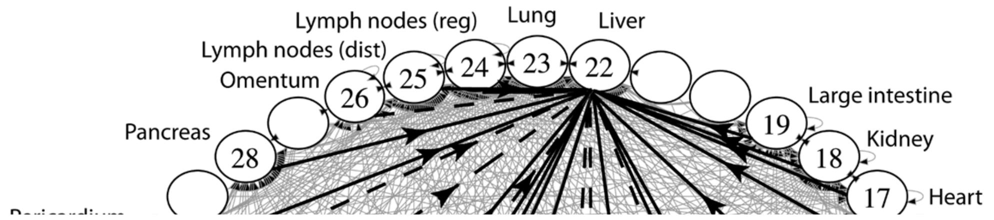
- Social media graphs
  - Vertices are accounts, edges are follower relationships
- Traffic
  - Vertices are classes, edges are usage

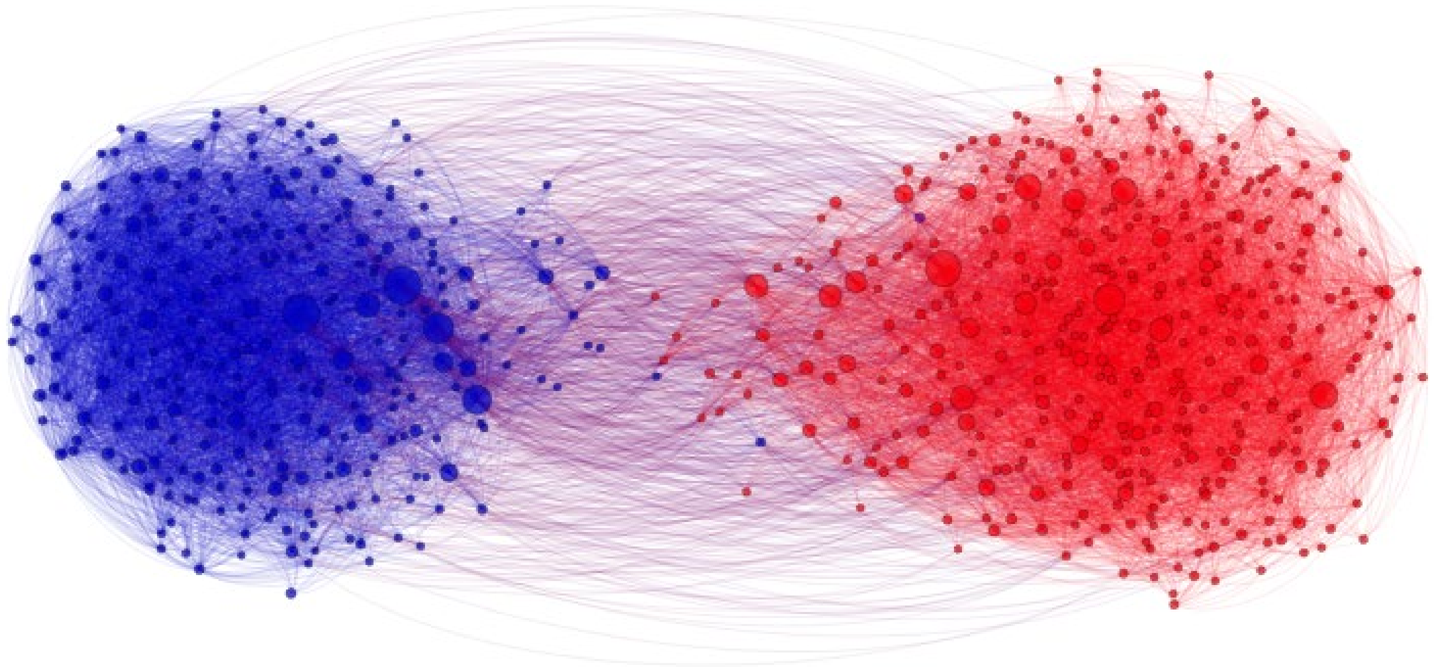
Influence

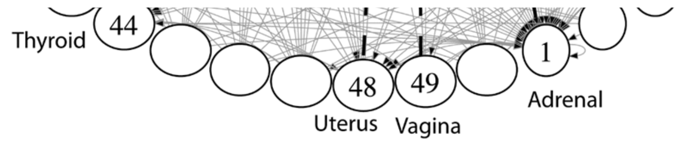
- Biology
  - Vertices are cancer cell desinations, edges are migration paths

Related topics

- Web Page Ranking
  - Vertices are web pages, edges are hyperlinks
- Wikipedia
  - Vertices are articles, edges are links







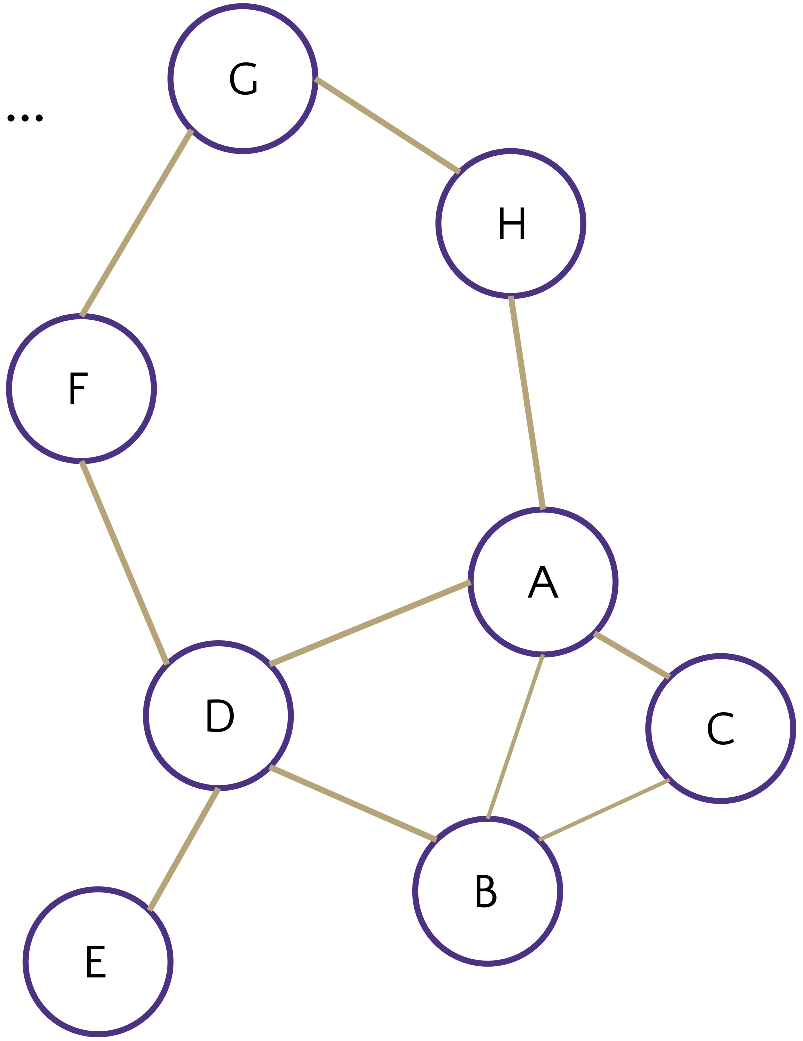
And so many more!!

www.allthingsgraphed.com

Graph: Formal Definition

A **graph** is defined by a pair of sets G = (V, E) where...

- V is a set of **vertices**
  - A vertex or "node" is a data entity
  - V = { A, B, C, D, E, F, G, H }
- E is a set of **edges**
  - An edge is a connection between two vertices
  - E = { (A, B), (A, C), (A, D), (A, H), (C, B), (B, D), (D, E), (D, F), (F, G), (G, H)}



Graph Terminology

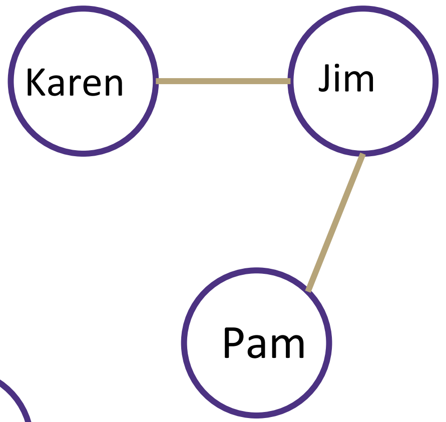
Graph Direction

- **Undirected graph** – edges have no direction and are two-way
  - V = { Karen, Jim, Pam }
  - E = { (Jim, Pam), (Jim, Karen) } *inferred (Karen, Jim) and (Pam, Jim)*
- **Directed graphs** – edges have direction and are thus one-way
  - V = { Gunther, Rachel, Ross }
  - E = { (Gunther, Rachel), (Rachel, Ross), (Ross, Rachel) }

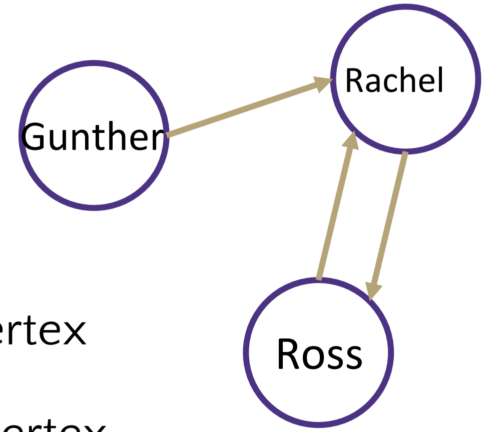
Degree of a Vertex

- **Degree** – the number of edges connected to that vertex
  - Karen : 1, Jim : 1, Pam : 1
- **In-degree** – the number of directed edges that point to a vertex
  - Gunther : 0, Rachel : 2, Ross : 1
- **Out-degree** – the number of directed edges that start at a vertex
  - Gunther : 1, Rachel : 1, Ross : 1

Undirected Graph:



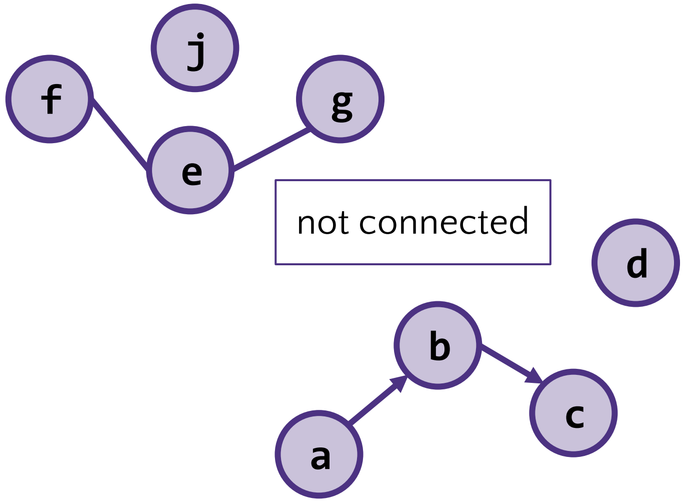
Directed Graph:

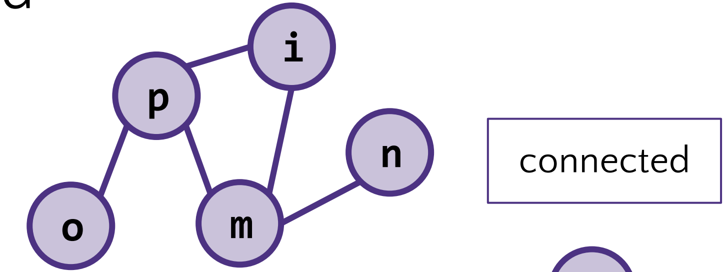


More Graph Terminology

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

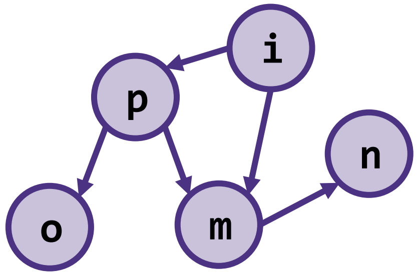
- If all the vertices are connected, we say the graph is **connected**
    - A directed graph is **weakly connected** if replacing every directed edge with an undirected edge results in a connected graph
    - A directed graph is **strongly connected** if a directed path exists between every pair of vertices
- The number of edges leaving a vertex is its **degree**





A **path** is a sequence of vertices connected by edges

- A **simple path** is a path without repeated vertices
- A **cycle** is a path whose first and last vertices are the same
    - A graph with a cycle is **cyclic**

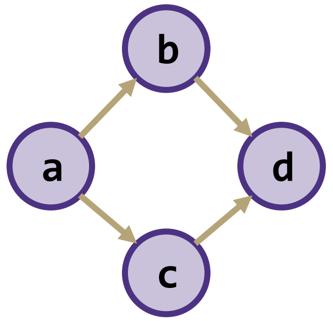


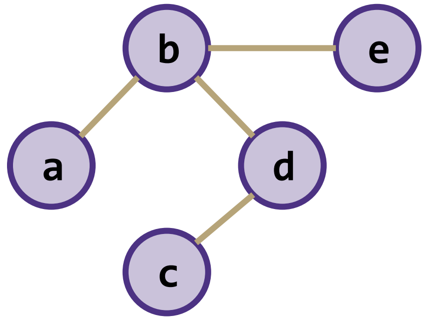
Directed vs Undirected; Acyclic vs Cyclic

Directed:

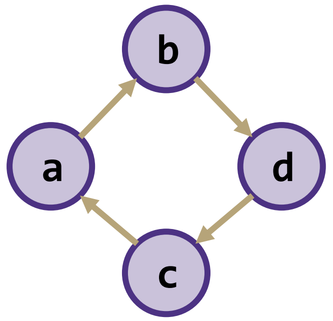
Undirected:

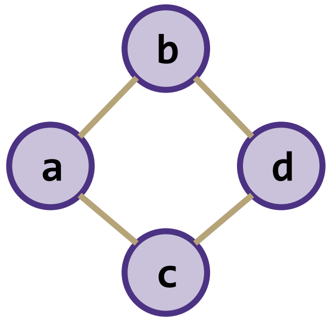
Acyclic:





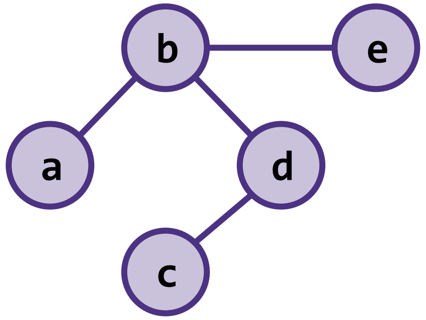
Cyclic:



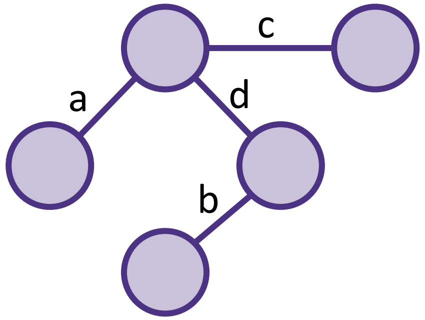


Labeled and Weighted Graphs

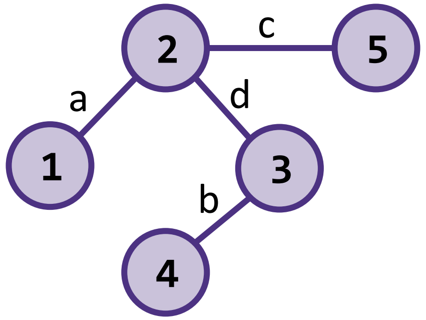
Vertex Labels



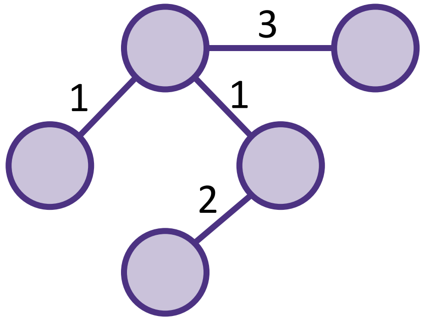
Edge Labels



Vertex & Edge Labels



Numeric Edge Labels
(Edge Weights)



Multi-Variable Analysis

- So far, we thought of everything as being in terms of some single argument “n”
- With graphs, we need to consider:
    - n (or |V|): total number of vertices (sometimes written as V)
    - m (or |E|): total number of edges (sometimes written as E)
    - deg(u): degree of node u (how many outgoing edges it has)

Adjacency Matrix

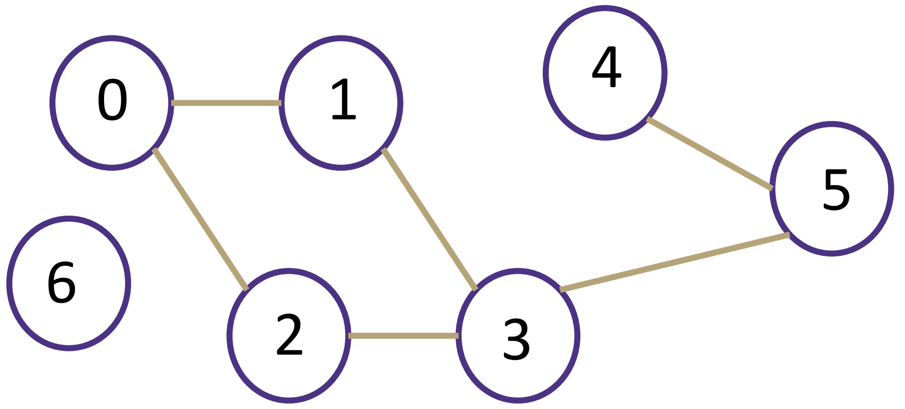
In an adjacency matrix a[u][v] is 1 if there is an edge (u,v), and 0 otherwise.

Worst-case Time Complexity (|V| = n, |E| = m):

- Add Edge: O(1)
- Remove Edge: O(1)
- Check edge exists from (u,v): O(1)
- Get out-neighbors of u: O(n)
- Get in-neighbors of u: O(n)

Space Complexity: O(n^2)

More suitable for dense graphs



	0	1	2	3	4	5	6
0	0	1	1	0	0	0	0
1	1	0	0	1	0	0	0
2	1	0	0	1	0	0	0
3	0	1	1	0	0	1	0
4	0	0	0	0	0	1	0
5	0	0	0	1	1	0	0
6	0	0	0	0	0	0	0

For an undirected graph, adjacency matrix is symmetric w.r.t diagonal line. A node u's out-neighbors are the same as its in-neighbors

Adjacency List

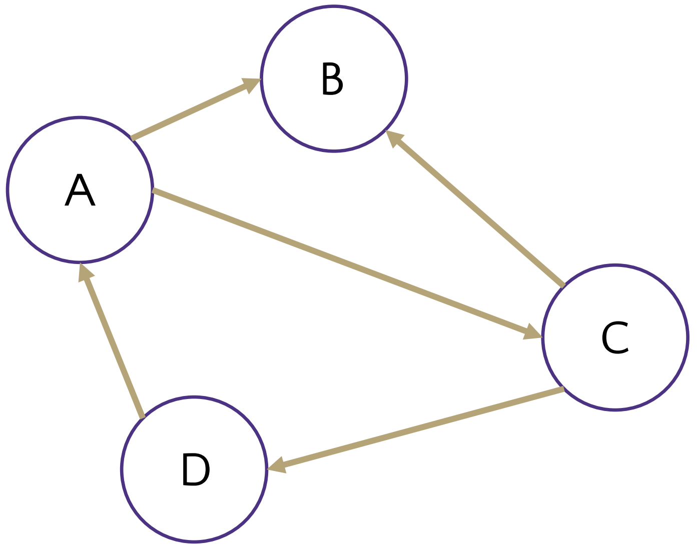
In an adjacency matrix a[u][v] is 1 if there is an edge (u,v), and 0 otherwise.

Worst-case Time Complexity (|V| = n, |E| = m):

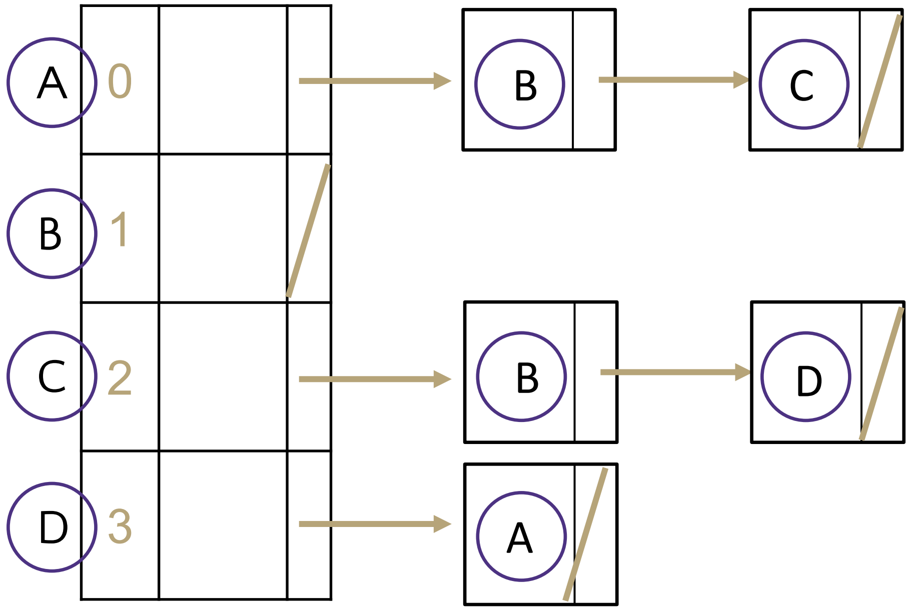
- Add Edge: O(1)
- Remove Edge: O(deg(u))
- Check edge exists from (u,v): O(deg (u))
- Get outneighbors of u: O(deg(u))
- Get inneighbors of u: O(n + m)

Space Complexity: O(n + m)

More suitable for sparse graphs



Linked Lists



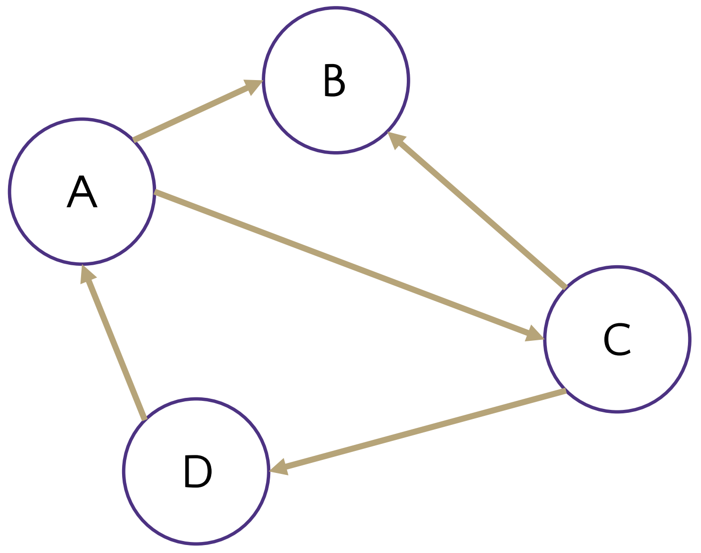
Adjacency List

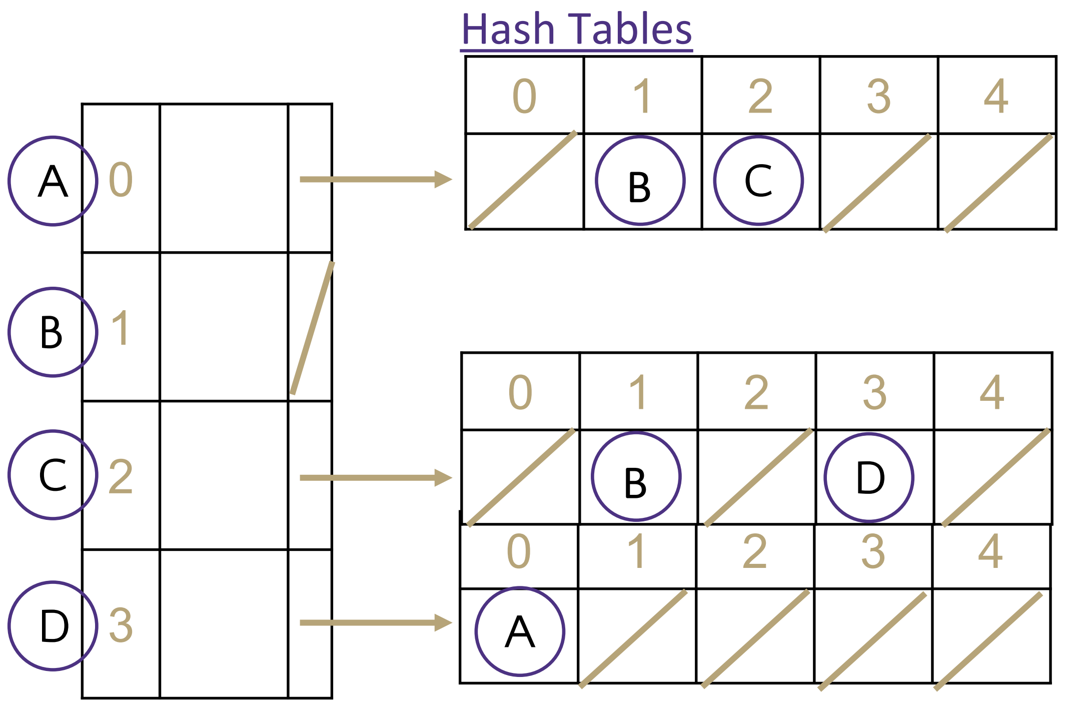
In an adjacency matrix a[u][v] is 1 if there is an edge (u,v), and 0 otherwise.

Worst-case Time Complexity (assuming a good hash function so all hash table operations are O(1)) (|V| = n, |E| = m):

- Add Edge: O(1)
- Remove Edge: O(1)
- Check edge exists from (u,v): O(1)
- Get outneighbors of u: O(deg(u))
- Get inneighbors of u: O(n)

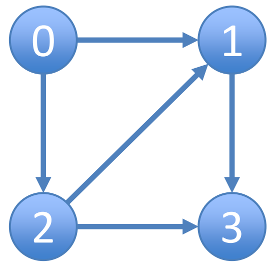
Space Complexity: O(n + m)





2-Hop Neighbors (through Matrix Multiplication)

- Matrix multiplication for finding two-hop neighbors



A graph and its adjacency matrix representation

$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 2 & 0 & 1 & 0 & 1 \\ 3 & 0 & 0 & 0 & 0 \end{array}$$

The adjacency matrix representation

$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 1 & 0 & 2 \\ 1 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 1 \\ 3 & 0 & 0 & 0 & 0 \end{array}$$

Node 3 is a two-hop neighbor of node 0 along two different paths

Node 3 is a two-hop neighbor of node 2 along 1 path

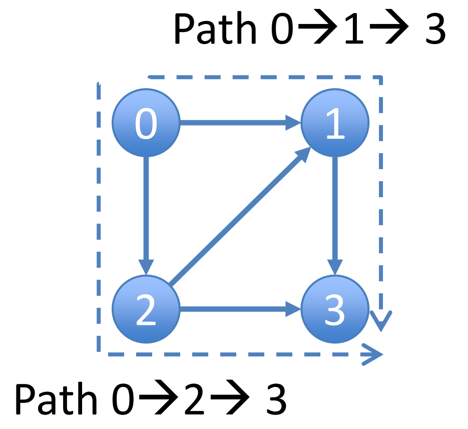
The adjacency matrix representation for two-hop neighbors, obtained by matrix-matrix product of the adjacency matrix

$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}^2 = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & 2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

2-Hop Neighbors (through Matrix Multiplication)

- Consider the multiplication of the first row of the left matrix wit the last column of the right matrix:
  - 0*0 + 1*1 + 1*1 + 0*0 = 2.
- This means that there are two 2-hop paths from 1 to 3:
  - Path 0→1→ 3 consisting of two edges 0→1 & 1→ 3, corresponding to the first term of 1*1
  - Path 0→2→ 3 consisting of two edges 0→2 &2→ 3, corresponding to the second term of 1*1

$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & \boxed{2} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

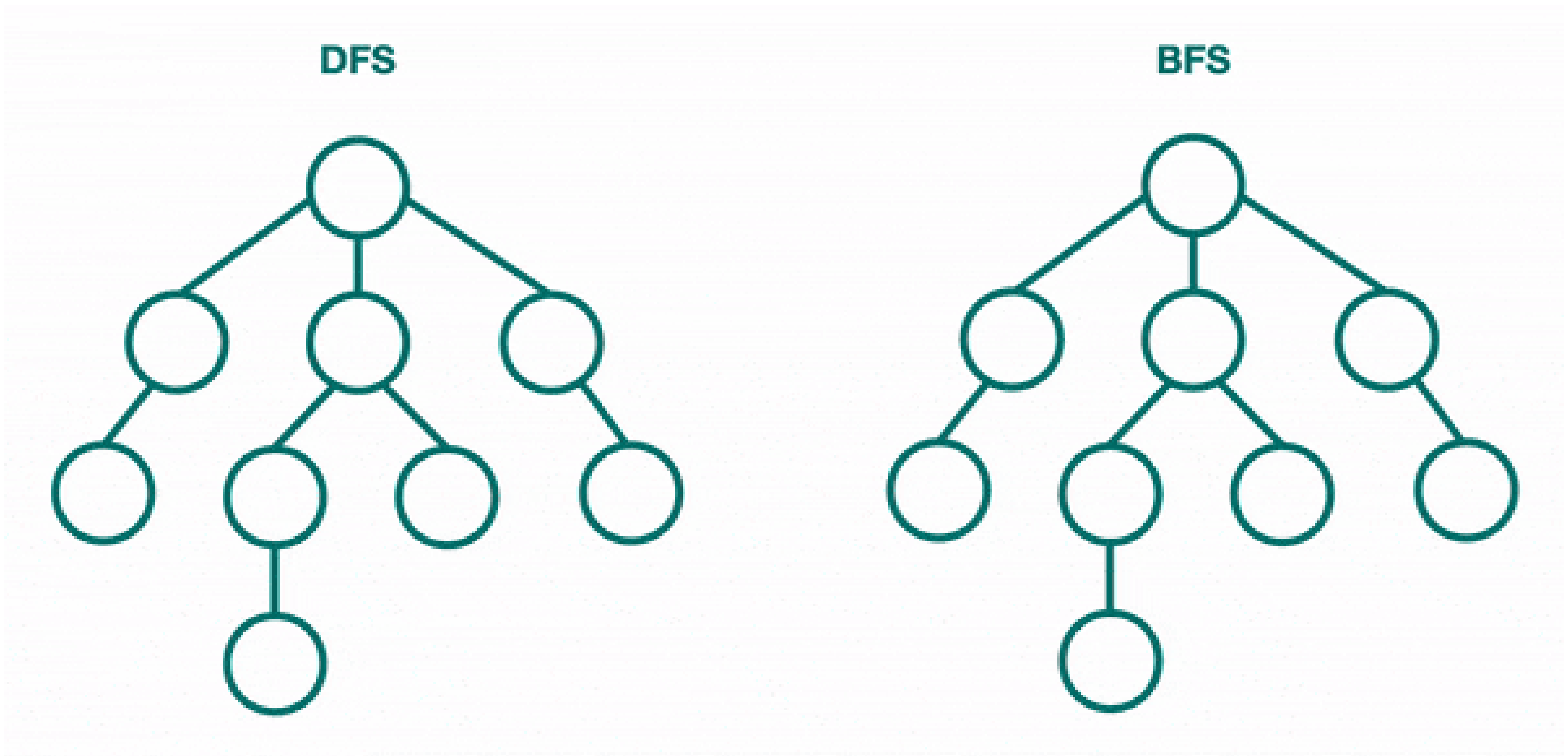


Graph Traversals

Topological Sort

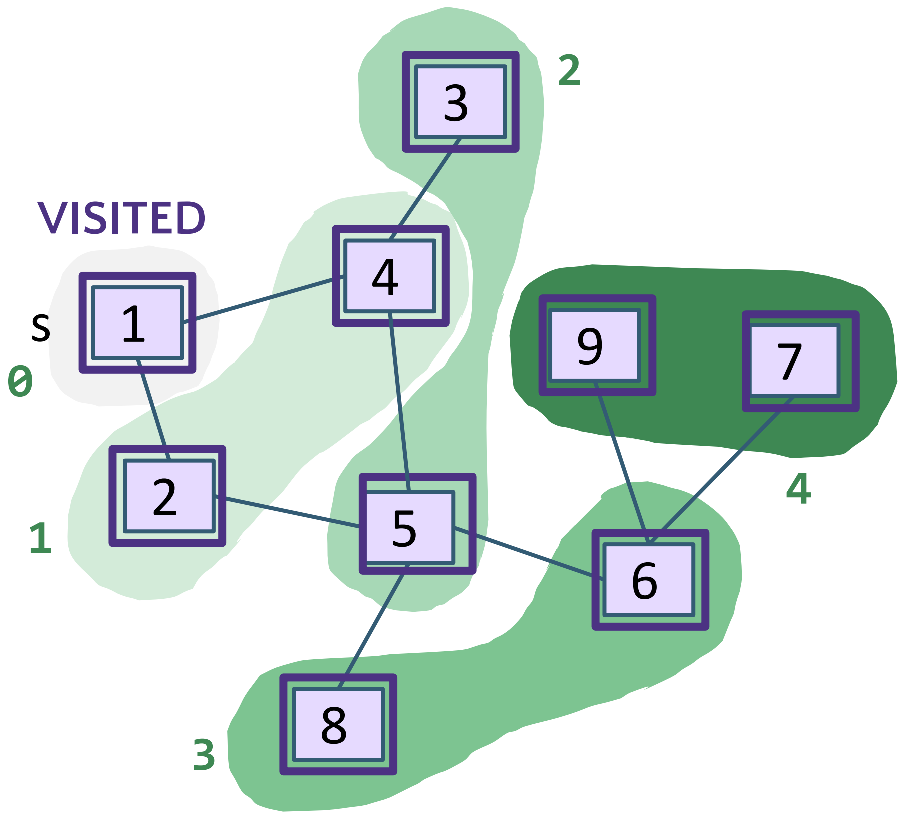
Shortest Path

BFS & DFS for Trees



Breadth-First Search (BFS)

BFS: Explore nodes "layer by layer"; like level-order traversal of a tree, now generalized to any graph; visit 1-hop neighbors, then 2-hop neighbors, …, until all nodes have been visited

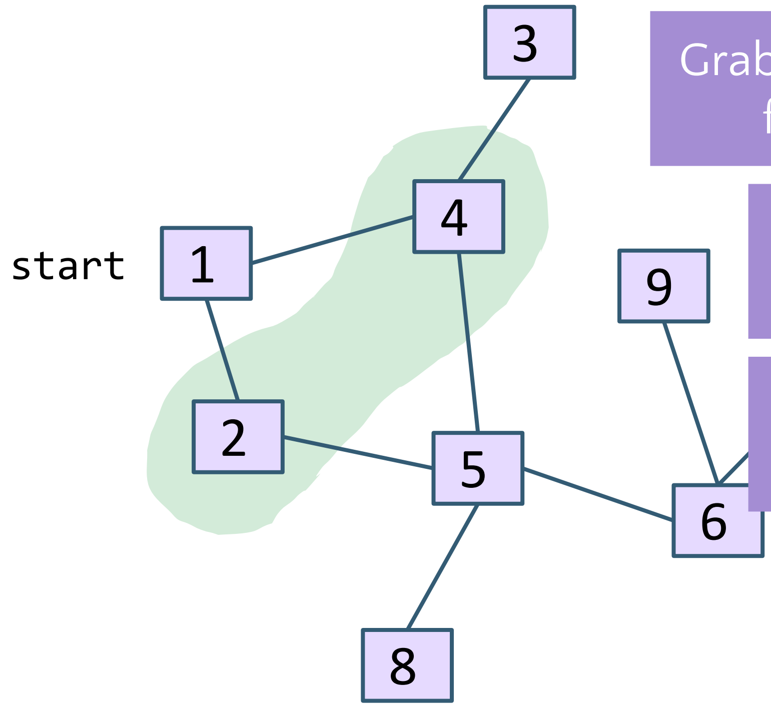


This is our goal, but how do we translate into code?

- Use a data structure to "queue up" children…

```
for (Vertex n : s.neighbors) {
```

BFS Implementation



A queue keeps track of "**outer edge**" of nodes still to explore

Kick off the algorithm by adding start to perimeter

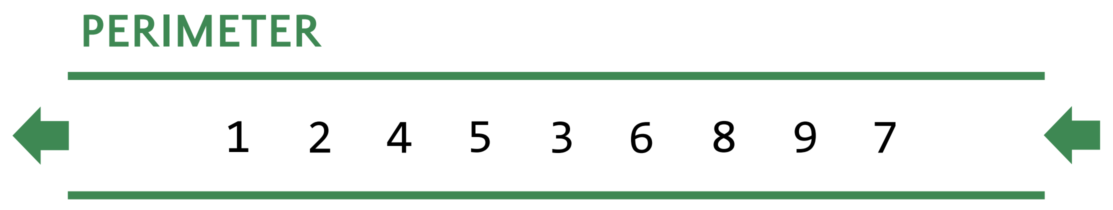
Grab one element at a time from the perimeter

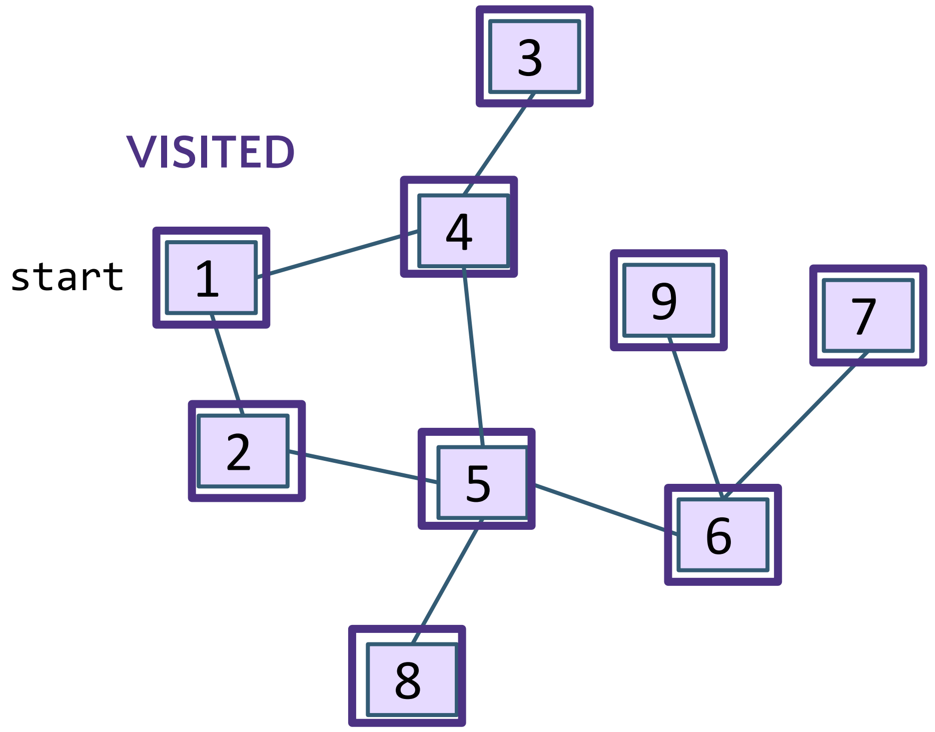
Look at all that element's children

Add new ones to perimeter!

```
bfs(Graph graph, Vertex start) {
   Queue<Vertex> perimeter = new Queue<>();
   Set<Vertex> visited = new Set<>();

  perimeter.add(start);
  visited.add(start);

  while (!perimeter.isEmpty()) {
    Vertex from = perimeter.remove();
    for (Edge edge : graph.edgesFrom(from)) {
          Vertex to = edge.to();
          if (!visited.contains(to)) {
            perimeter.add(to);
            visited.add(to);
          }
        }
      }
    }
```

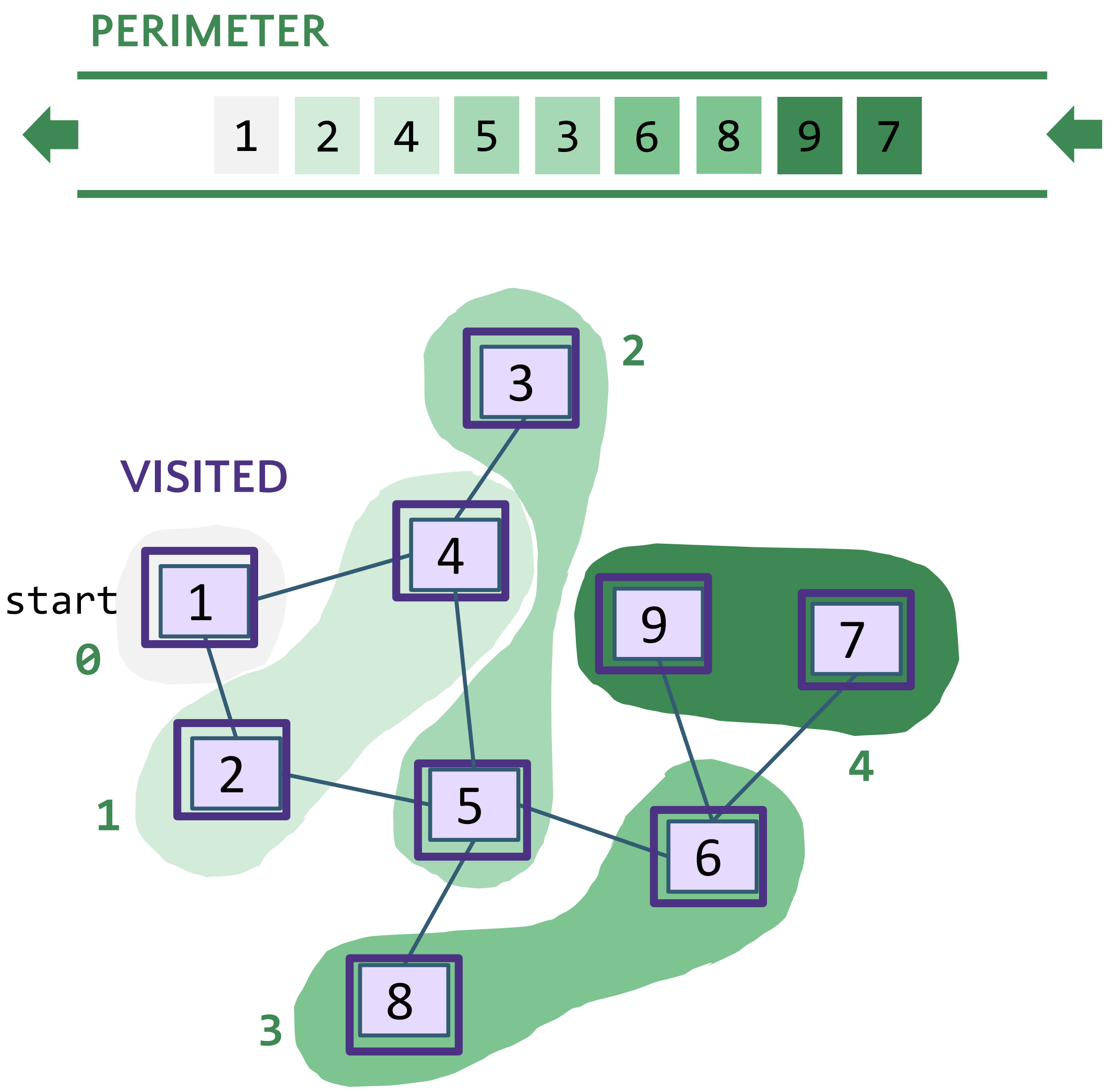
BFS Implementation: In Action





```
bfs(Graph graph, Vertex start) {
  Queue<Vertex> perimeter = new Queue<>();
  Set<Vertex> visited = new Set<>();

  perimeter.add(start);
  visited.add(start);

  while (!perimeter.isEmpty()) {
    Vertex from = perimeter.remove();
    for (Edge edge : graph.edgesFrom(from)) {
      Vertex to = edge.to();
      if (!visited.contains(to)) {
        perimeter.add(to);
        visited.add(to);
      }
    }
  }
}
```

BFS Intuition: Why Does it Work?

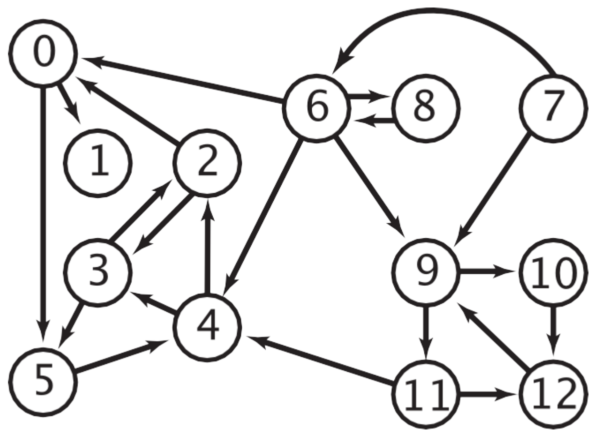


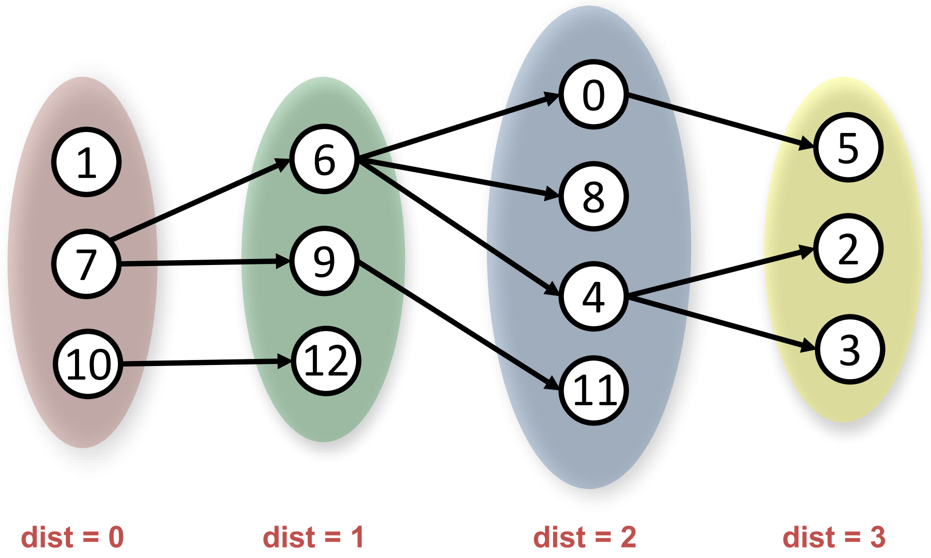
- Using FIFO queue means we explore an entire layer before moving on to the next layer
- Keep going until `perimeter` is empty

```
  perimeter.add(start);
  visited.add(start);

  while (!perimeter.isEmpty()) {
    Vertex from = perimeter.remove();
    for (Edge edge : graph.edgesFrom(from)) {
      Vertex to = edge.to();
      if (!visited.contains(to)) {
        perimeter.add(to);
        visited.add(to);
      }
    }
  }
}
```

BFS Application: Multiple-Source Shortest Paths Problem

- Given a digraph and a set of source vertices, find shortest path from any vertex in the set to every other vertex, assuming all edges have weight 1.
    - e.g., S = { 1, 7, 10 }.
    - Shortest path to 4 is 7 → 6 → 4.
    - Shortest path to 5 is 7 → 6 → 0 → 5.
    - Shortest path to 12 is 10 → 12.
- Can be done with BFS, and initialize by enqueuing all source nodes

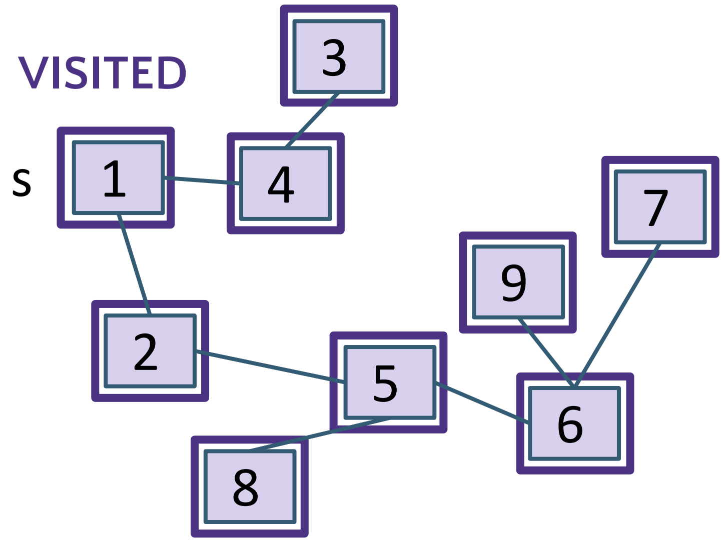




Depth-First Search (DFS) (Recursive Algorithm)

- DFS explores one branch “all the way down” before coming back to try branches
- Depending on the order of visiting branches, many possible orderings: e.g., {1, 2, 5, 6, 9, 7, 8, 4, 3}, {1, 4, 3, 2, 5, 8, 6, 7, 9}, etc.

```
Set<Vertex> visited; // assume global
connected(Vertex s, Vertex t) {
  if (s == t) {
    return true;
  } else {
    visited.add(s);
    for (Vertex n : s.neighbors) {
      if (!visited.contains(n)) {
        if (connected(n, t)) {
          return true;
        }
      }
    }
    return false;
  }
}
```



DFS w/ Stack vs. BFS w/ Queue

```
dfs(Graph graph, Vertex start) {
  Stack<Vertex> perimeter = new Stack<>();
  Set<Vertex> visited = new Set<>();

  perimeter.add(start);

  while (!perimeter.isEmpty()) {
    Vertex from = perimeter.remove();
        if (!visted.contains(from)) {
      for (Edge edge:graph.edgesFrom(from)) {
        Vertex to = edge.to();
                perimeter.add(to)
      }

      visited.add(from);
    }
  }
}
```

```
bfs(Graph graph, Vertex start) {
  Queue<Vertex> perimeter = new Queue<>();
  Set<Vertex> visited = new Set<>();

  perimeter.add(start);
  visited.add(start);

  while (!perimeter.isEmpty()) {
    Vertex from = perimeter.remove();
    for (Edge edge : graph.edgesFrom(from)) {
      Vertex to = edge.to();
      if (!visited.contains(to)) {
        perimeter.add(to);
        visited.add(to);
      }
    }
  }
}
```

Recap: Graph Traversals

DFS

(Iterative)

- Follow a "choice" all the way to the end, then come back to revisit other choices
- Uses a **stack**!

DFS

(Recursive)

On huge graphs, might overflow the call stack

BFS

(Iterative)

- Explore layer-by-layer: examine every node at a certain distance from start, then examine nodes that are one level farther
- Uses a **queue**!

Topological Sort

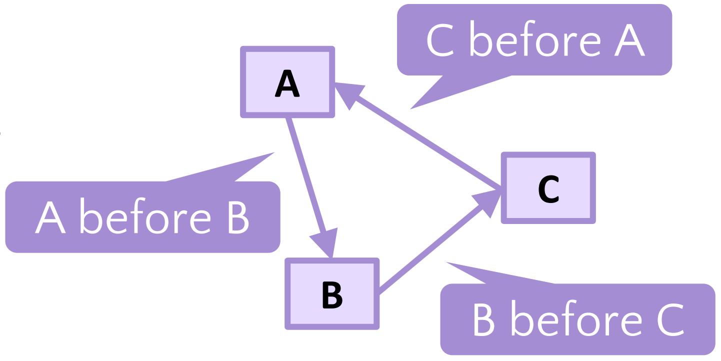
A Directed Acyclic Graph (DAG) : A directed graph (digraph) without any cycles

A DAG encodes a "dependency graph"

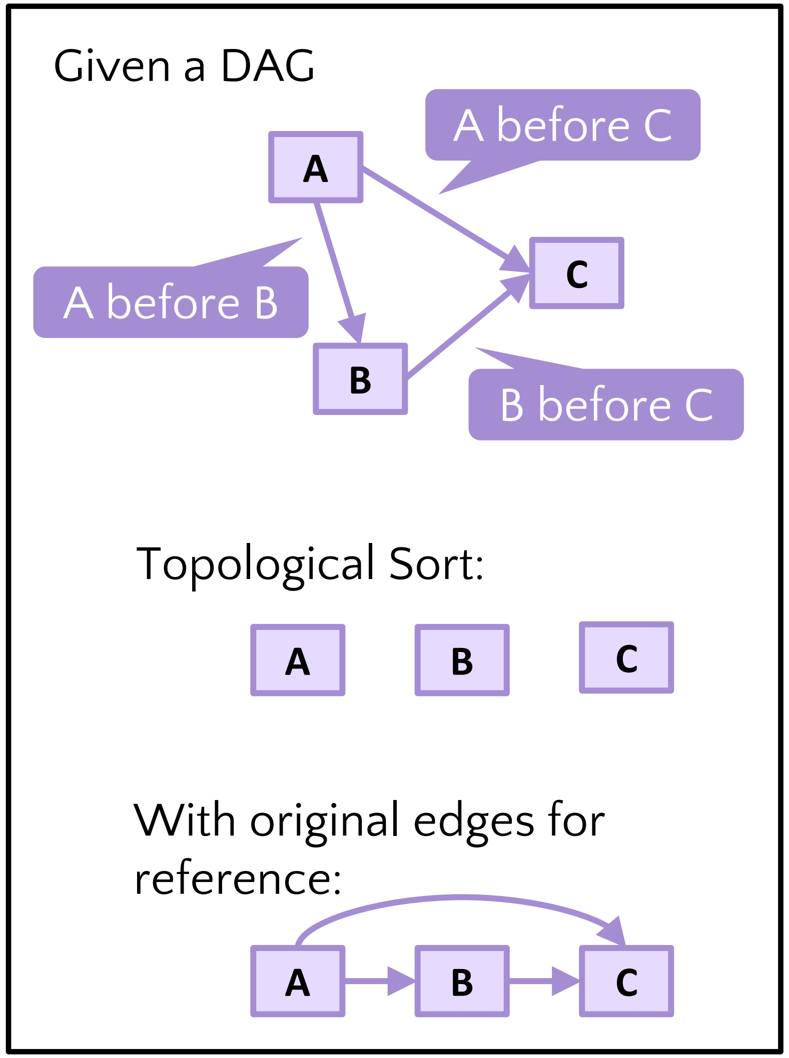
- An edge (u, v) means u must precede v
- A topological sort or topological ordering of a DAG gives a total node ordering that **respects dependencies**

Applications:

- Compiling multiple Java files
- Multi-job Workflows

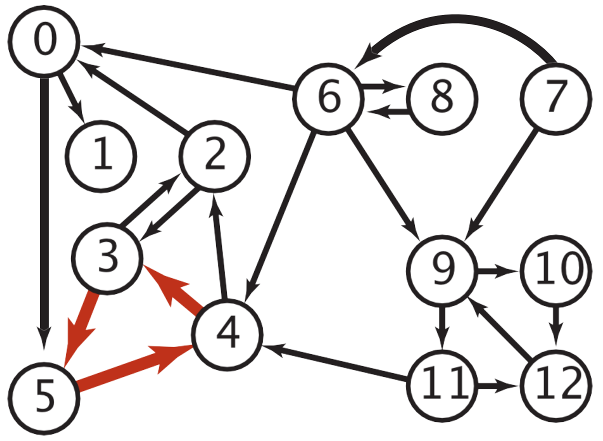


Not a DAG. No possible topological sort



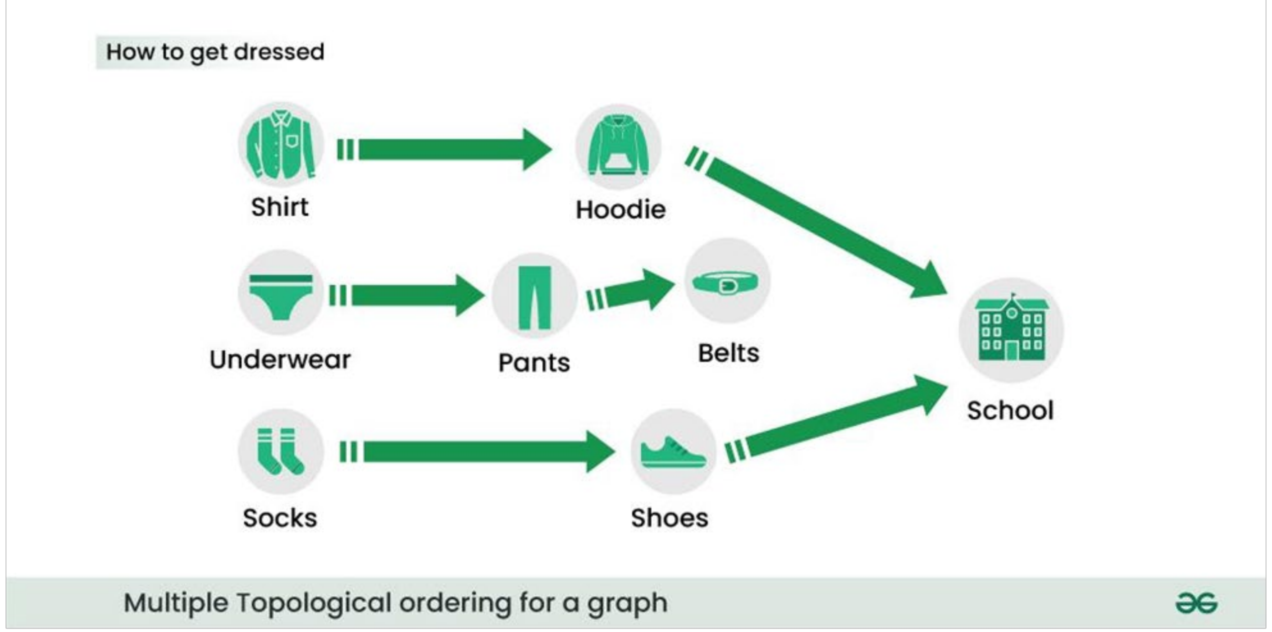
Cycles and Undirected Edges

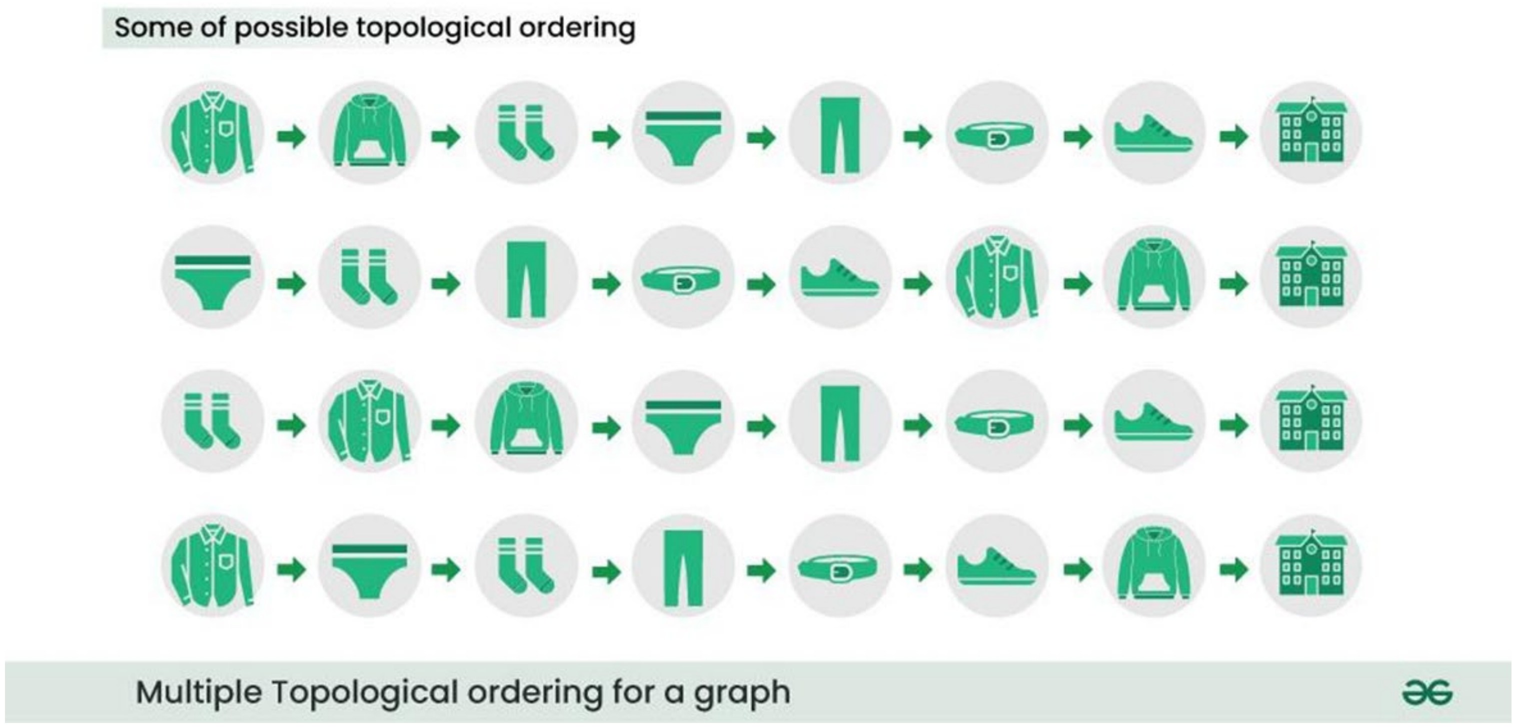
- Why is topological sort not possible for digraphs containing cycles?
    - Imagine a graph with 3 vertices and edges = {1 to 2 , 2 to 3, 3 to 1} forming a cycle. Now if we try to topologically sort this graph starting from any vertex, it will always create a contradiction to our definition. All the vertices in a cycle are indirectly dependent on each other hence topological sortfails
- Why is topological sort not possible for graphs with undirected edges?
    - Special case of a cycle. Undirected edge between two vertices u and v means, there is an edge from u to v as well as from v to u. Because of this both the nodes u and v depend upon each other and none of them can appear before the other in the topological sortwithout creating a contradiction



a digraph with a directed cycle

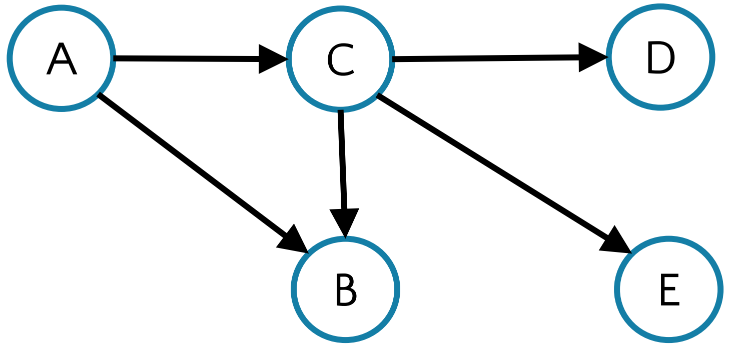
Topological Sort Example I




Some of possible topological ordering
Multiple Topological ordering for a graph

Topological Sort Example II

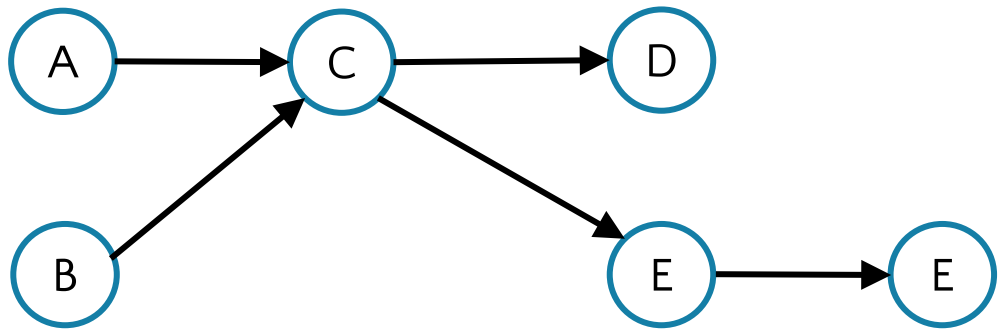
Give all possible topological orderings of this DAG



By observation, we can get (A, C, B, D, E), (A, C, B, E, D)

Topological Sort Example III

Give all possible topological orderings of this DAG



By observation, we can get: all possible interleavings with (AB) or (BA) before C, (DEF) or (DFE) or (EFD) after C. e.g., ABCDEF, BACEDF, etc.

Topological Sort by DFS Post-Order Traversal

- Perform DFS Post-Order Traversal starting from any node (often, but not necessarily, a node with no incoming edges (no predecessors)) to get an ordered list of nodes, then reverse the node order to get a Topological Sort
    - Upon finishing traversal starting from one node, restart from another unvisited node with no incoming edges
    - This results in one of multiple possible Topological Sorts
- Intuition: DFS Post-Order Traversal outputs nodes from the deepest (furthest away from the starting node) to the starting mode, hence the reverse order is a Topological Sort from the starting node

DFS Traversal: Pre-order, Post-order

```
function preOrderTraversal(node) {
  if (node !== null) {
    visitNode(node);
    preOrderTraversal(node.left);
    preOrderTraversal(node.right);
  }
}
```

```
function postOrderTraversal(node) {
  if (node !== null) {
    postOrderTraversal(node.left);
    postOrderTraversal(node.right);
    visitNode(node);
  }
}
```

Recall: Binary Tree traversal with DFS: pre-order, post-order

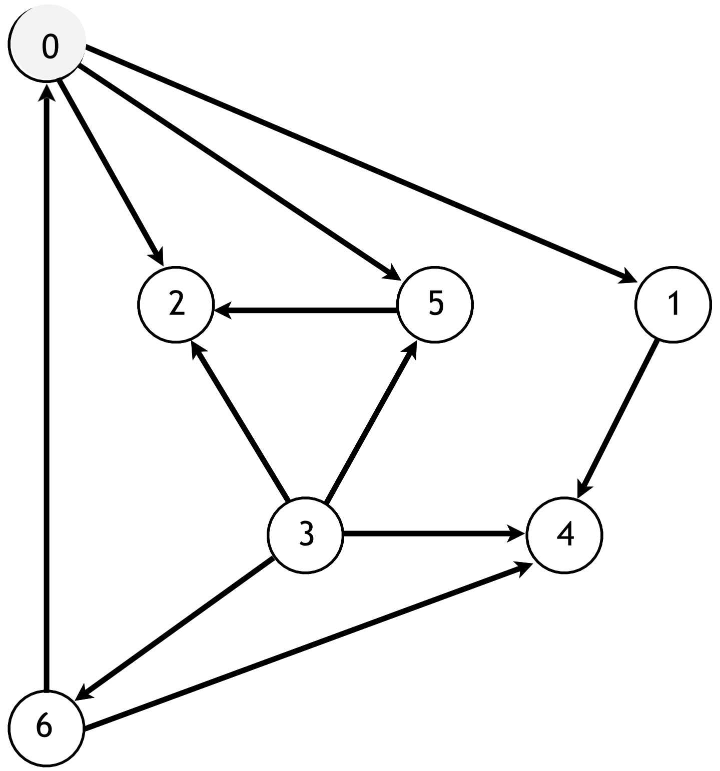
```
function preOrderTraversal(node) {
  if (node !== null) {
    visitNode(node);
    foreach(c ∈ node.children) {
        preOrderTraversal(c);}
    }
}
```

```
function postOrderTraversal(node) {
  if (node !== null) {
        foreach(c ∈ node.children) {
            postOrderTraversal(c);}
        visitNode(node);
  }
}
```

Graph traversal with DFS: pre-order, post-order (for post-order, visit a node during backtrack when all its children have been visited)

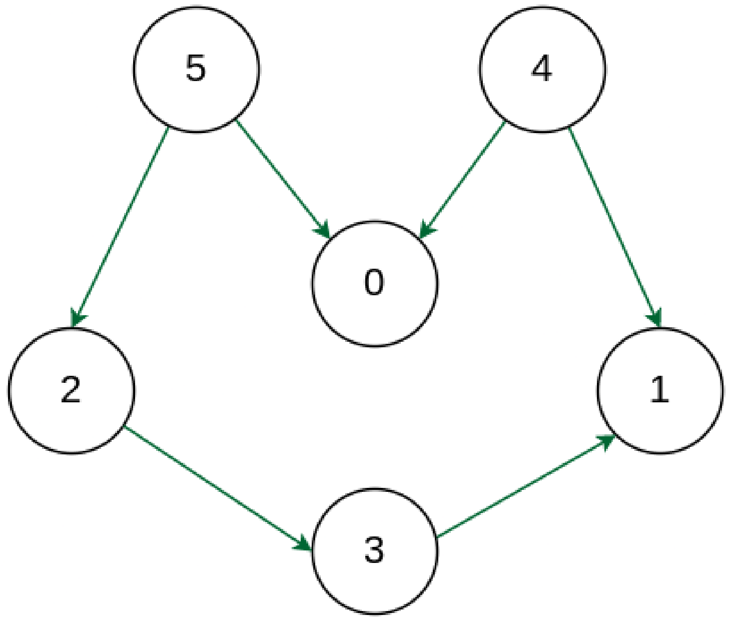
DFS Traversal Example I

- Start from node 0
- Visit nodes 0->1->4. Since 4 has no successors, backtrack to 0.
  - Pre-order: (0, 1, 4); post-order: (4, 1)
- Visit node 2. Since 2 has no successors, backtrack to 0
  - Pre-order: (0, 1, 4, 2); post-order: (4, 1, 2)
- Visit node 5. Since 5's successor 2 has been visited, backtrack to 0. Since all of node 0's successors have been visited, we visit it in post-order
  - Pre-order: (0, 1, 4, 2, 5); post-order: (4, 1, 2, 5, 0)
- Restart from node 3, visit its successor 6, then backtrack. Since all of node 3's successors have been visited, we visit it in post-order
  - Pre-order: (0, 1, 4, 2, 5, 3, 6); post-order: (4, 1, 2, 5, 0, 6, 3)
- All nodes and their successors have been visited, so the algorithm terminates. A topological sort corresponding to this post-order is (3, 6, 0, 5, 2, 1, 4)
- This is one of many possible traversals, e.g., if we start from another node 3 with no predecessors, then we have the traversals: pre-order: (3, 5, 2, 4, 6, 0, 1); post-order: (2, 5, 4, 3, 1, 0, 6); Topological sort: (6, 0, 1, 3, 4, 5, 2); BFS traversal: (3, 2, 5, 4, 6, 0, 1)
- A BFS traversal: (0, 2, 5, 1, 4, 3, 6)



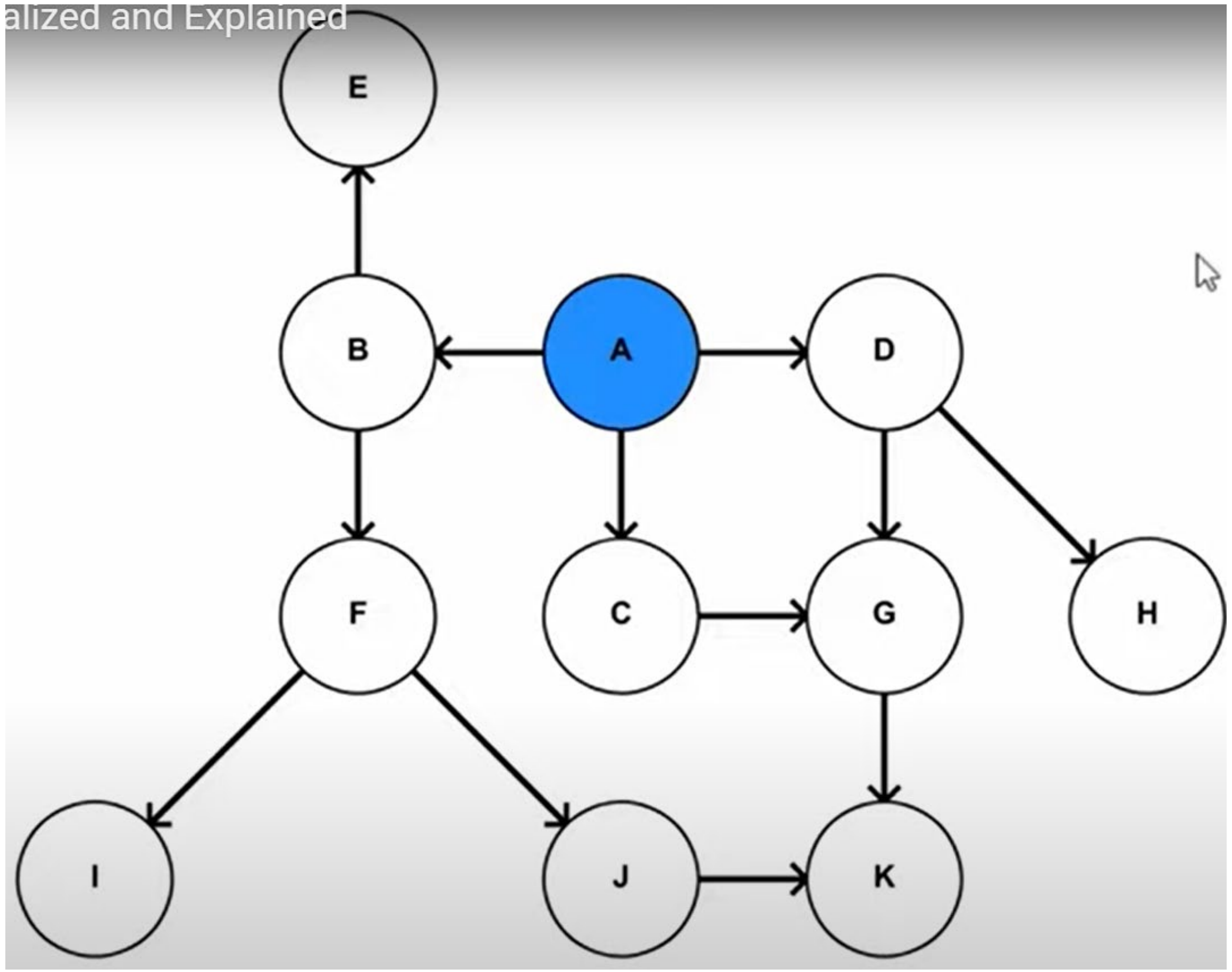
DFS Traversal Example II

- Starting from node 5:
  - Pre-order traversal: (5, 2, 3, 1, 0, 4)
  - Post-order traversal: (1, 3, 2, 0, 5, 4)
  - Topological sort: (4, 5, 0, 2, 3, 1)
  - BFS traversal: (5, 0, 2, 3, 1, 4)
- Starting from node 4:
  - Pre-order traversal: (4, 0, 1, 5, 2, 3)
  - Post-order traversal: (0, 1, 4, 3, 2, 5)
  - Topological sort: (5, 2, 3, 4, 1, 0)
  - BFS traversal: (4, 0, 1, 5, 2, 3)
- Starting from node 0:
  - Pre-order traversal: (0, 5, 2, 3, 1, 4)
  - Post-order traversal: (0, 1, 3, 2, 5, 4)
  - Topological sort: (4, 5, 2, 3, 1, 0)
  - BFS traversal: (0, 5, 2, 3, 1, 4)
- You may try starting any other node.



DFS Traversal Example III

- Starting from node A:
- Pre-order traversal: (A, B, F, I, J, K, E, C, G, D, H)
- Post-order traversal: (I, K, J, F, E, B, G, C, H, D, A)
- Topological sort: (A, D, H, C, G, B, E, F, J, K, I)
- BFS traversal: (A, B, C, D, E, F, G, H, I, J, K)
- Starting from a different node will give a different topological sort, but all of them must start with A, since it is the only node without any predecessors (i.e., it must precede all the other nodes based on the DAG)
- Any post-order traversal must visit A last, since all of A's sucessors must be visited before visiting A

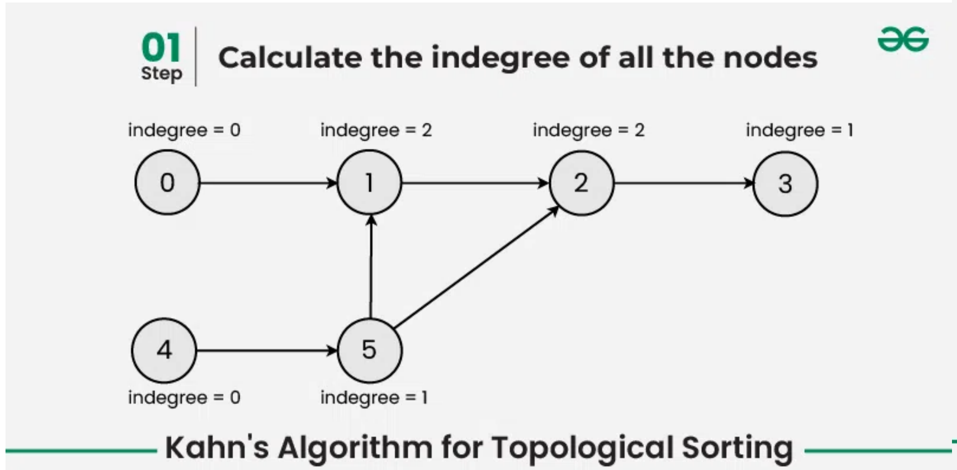


Topological Sort Visualized and Explained
https://www.youtube.com/watch?v=7J3GadLzydI

Kahn's algorithm for Topological Sort

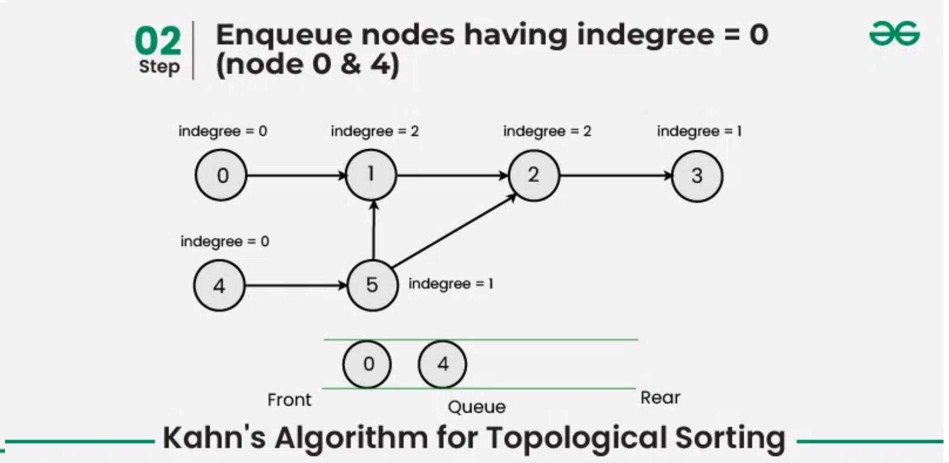
- The algorithm works by repeatedly finding vertices with no incoming edges, removing them from the graph, and updating the incoming edges of the remaining vertices. This process continues until all vertices have been ordered.
    - Add all nodes with in-degree 0 to a queue.
    - While the queue is not empty:
        - Remove a node from the queue.
        - For each outgoing edge from the removed node, decrement the in-degree of the destination node by 1.
        - If the in-degree of a destination node becomes 0, add it to the queue.
    - If the queue is empty and there are still nodes in the graph, the graph contains a cycle and cannot be topologically sorted.
    - The nodes in the queue represent the topological sort of the graph.
- Time Complexity: O(V+E).
    - The outer for loop will be executed V number of times and the inner for loop will be executed E number of times.

https://www.geeksforgeeks.org/topological-sorting-indegree-based-solution/



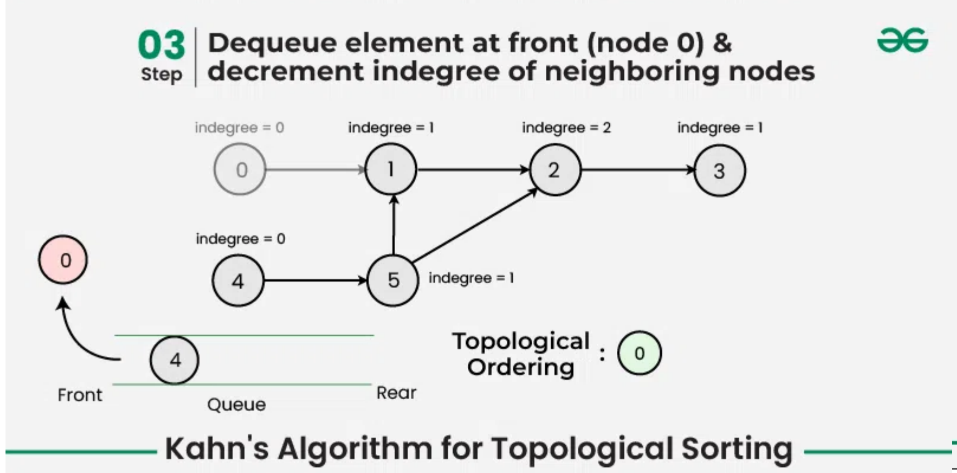
01 Step | Calculate the indegree of all the nodes

Kahn's Algorithm for Topological Sorting



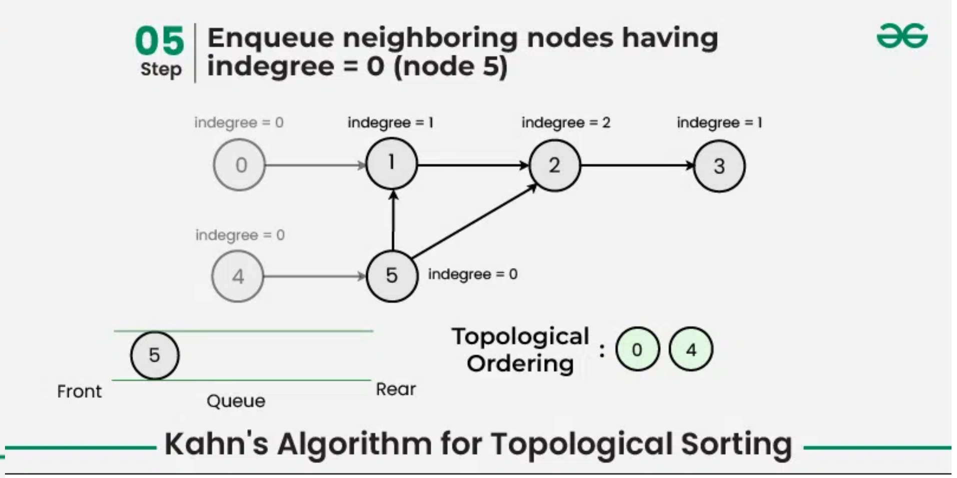
02 Step | Enqueue nodes having indegree = 0 (node 0 & 4)

Kahn's Algorithm for Topological Sorting



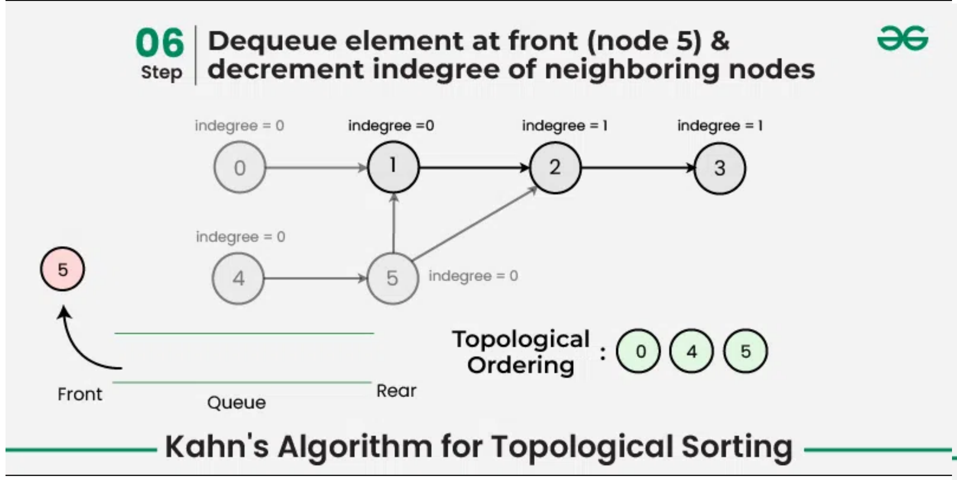
03 Step | Dequeue element at front (node 0) & decrement indegree of neighboring nodes

Kahn's Algorithm for Topological Sorting



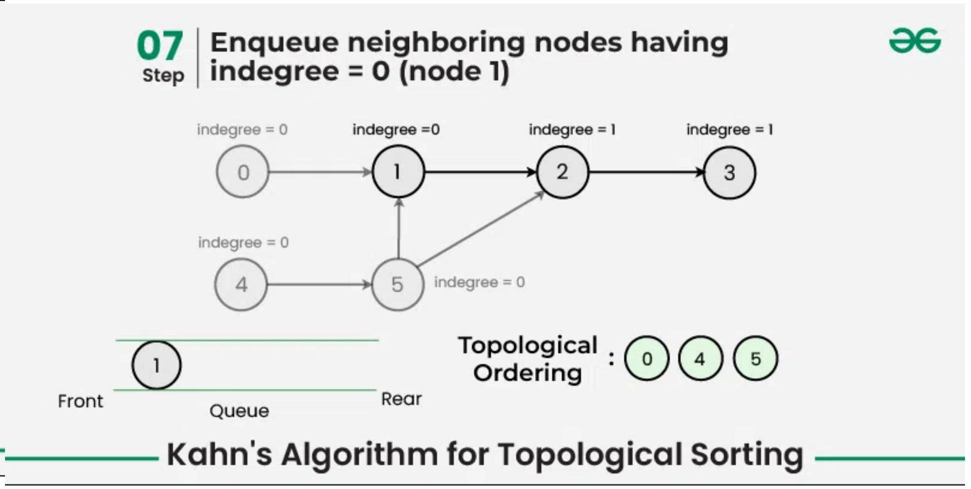
05 Step | Enqueue neighboring nodes having indegree = 0 (node 5)

Kahn's Algorithm for Topological Sorting



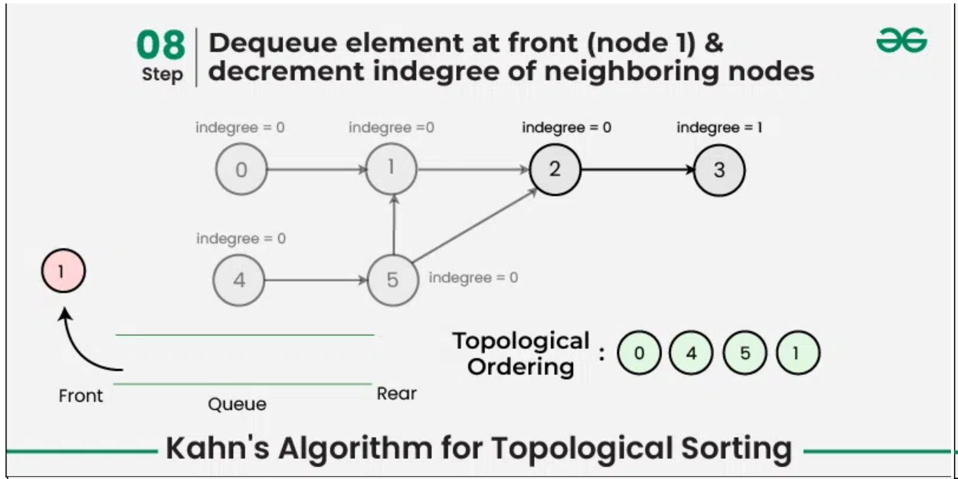
06 Step | Dequeue element at front (node 5) & decrement indegree of neighboring nodes

Kahn's Algorithm for Topological Sorting

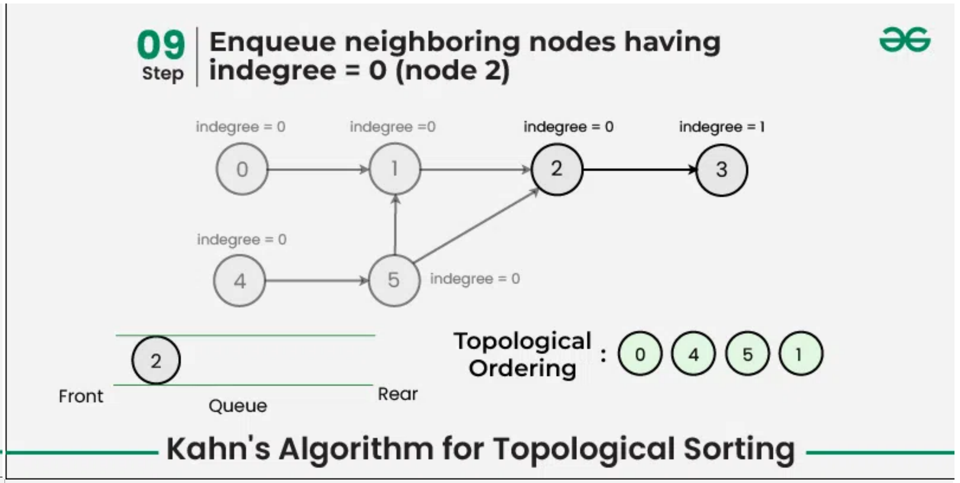


07 Step | Enqueue neighboring nodes having indegree = 0 (node 1)

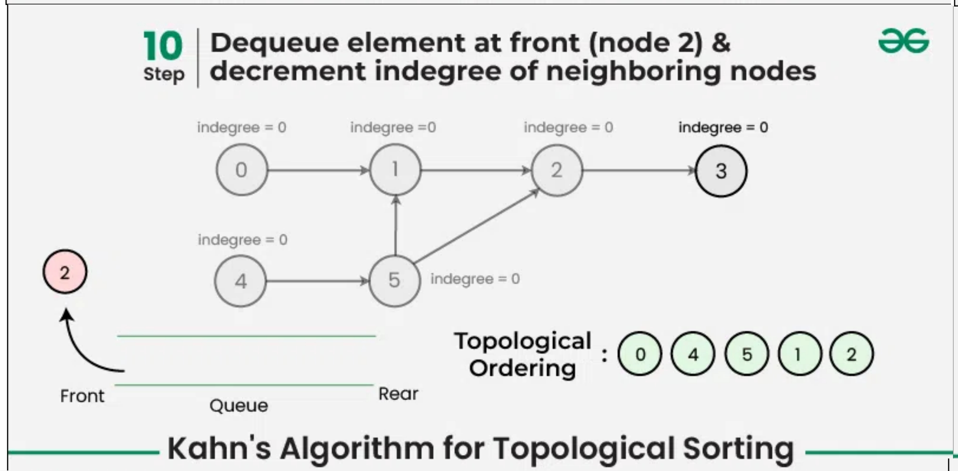
Kahn's Algorithm for Topological Sorting



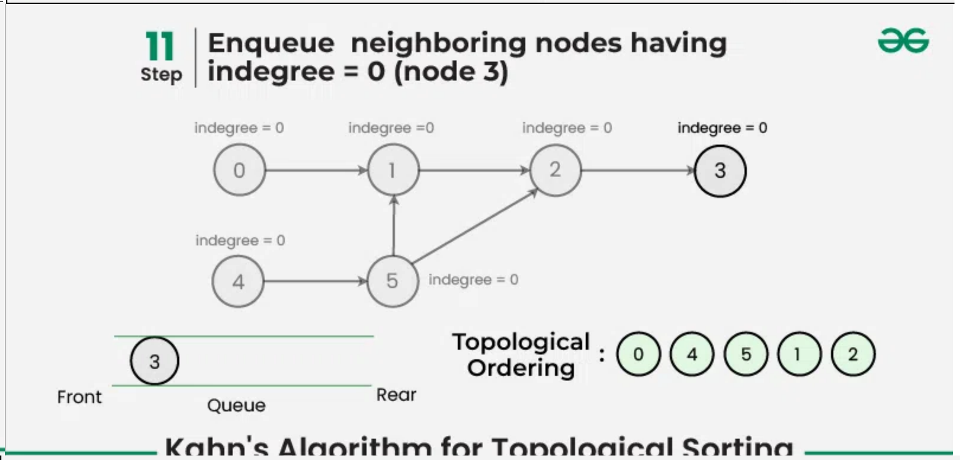
08 Step Dequeue element at front (node 1) & decrement indegree of neighboring nodes



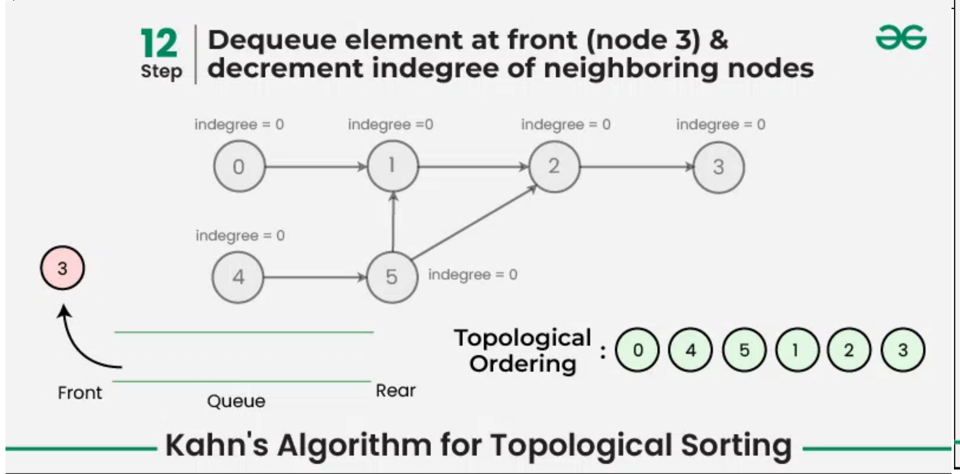
09 Step Enqueue neighboring nodes having indegree = 0 (node 2)



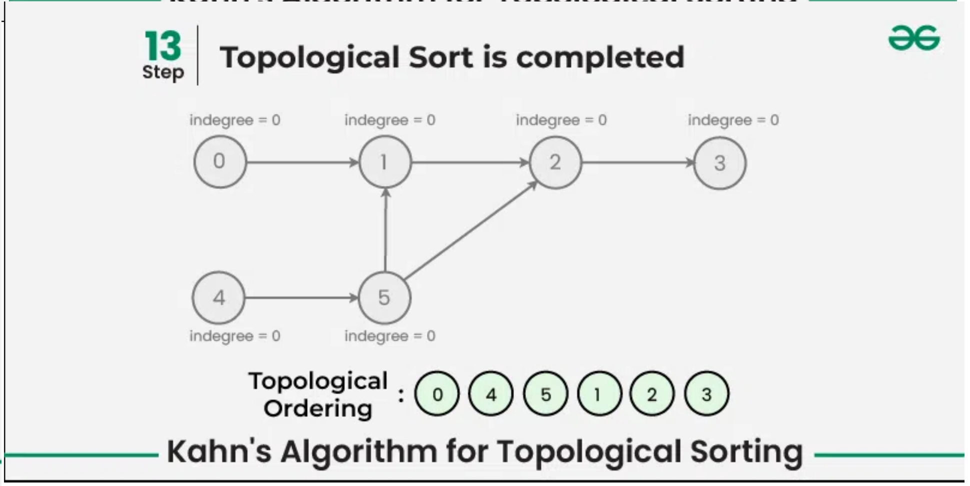
10 Step Dequeue element at front (node 2) & decrement indegree of neighboring nodes



11 Step Enqueue neighboring nodes having indegree = 0 (node 3)



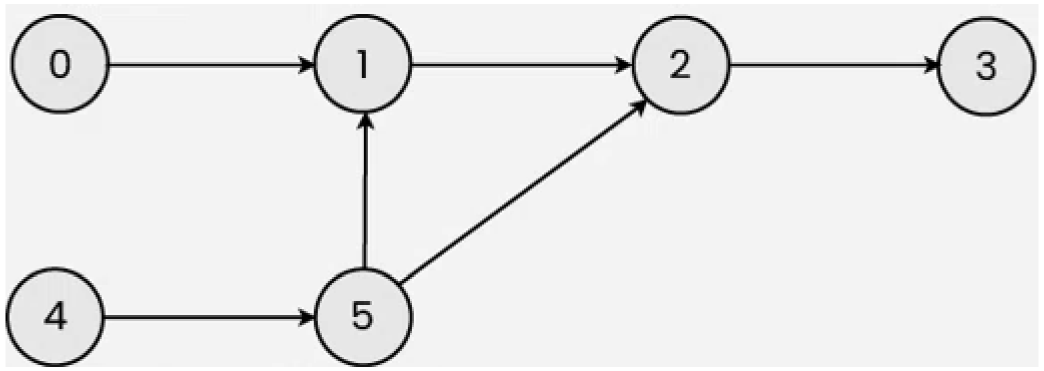
12 Step Dequeue element at front (node 3) & decrement indegree of neighboring nodes



13 Step Topological Sort is completed

Graph Traversals

- Give one DFS pre-order, in-order and post-order traversal of the following graph, and a topological sort.
- Starting from node 4:
  - Pre-order traversal: (4, 5, 1, 2, 3, 0)
  - Post-order traversal: (3, 2, 1, 5, 4, 0)
  - Topological Sort: (0, 4, 5, 1, 2, 3)
  - BFS: (4, 5, 1, 2, 3, 0)
- Starting from node 0:
  - Pre-order traversal: (0, 1, 2, 3, 4, 5)
  - Post-order traversal: (3, 2, 1, 0, 5, 4)
  - Topological Sort: (4, 5, 0, 1, 2, 3)
  - BFS: (0, 1, 2, 3, 4, 5)



References

- Breadth-first search in 4 minutes (for a tree)
  - https://www.youtube.com/watch?v=HZ5YTanv5QE
- Depth-first search in 4 minutes (for a tree)
  - https://www.youtube.com/watch?v=Urx87-NMm6c
- Graph Traversals - Breadth First and Depth First (for an undirected graph)
  - https://www.youtube.com/watch?v=bIA8HEEUxZI

Graph Traversals

Topological Sort

Shortest Path

The Shortest Path Problem

(Unweighted) Shortest Path Problem

Given source vertex **s** and a target vertex **t**, how long is the shortest path from **s** to **t**? What edges makeup that path?

This is a little harder, but still totally doable! We just need a way to keep track of how far each node is from the start.

- Sounds like a job for?

Using BFS for the Shortest Path Problem

(Unweighted) Shortest Path Problem

Given source vertex **s** and a target vertex **t**, how long is the shortest path from **s** to **t**? What edges makeup that path?

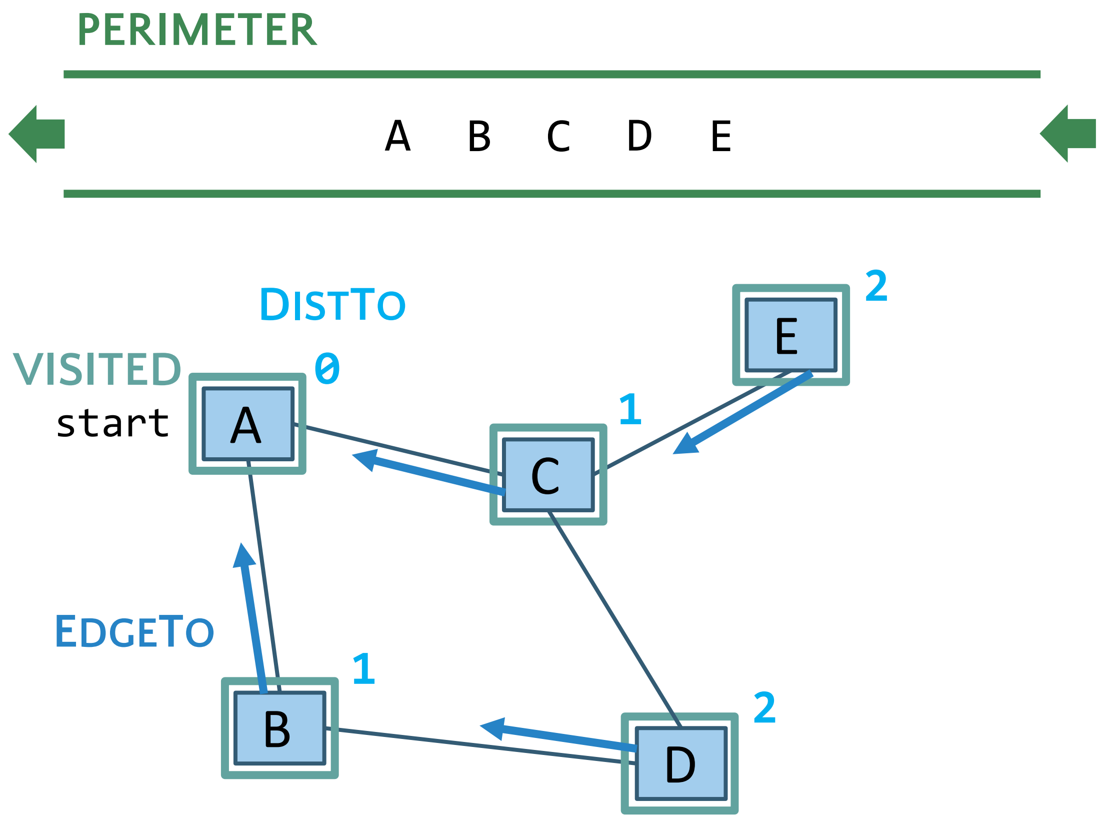
We need to keep track of how far each node is from the start, using BFS

```
  ...
  Map<Vertex, Edge> edgeTo = ...
  Map<Vertex, Double> distTo = ...

  edgeTo.put(start, null);
  distTo.put(start, 0.0);

  while (!perimeter.isEmpty()) {
    Vertex from = perimeter.remove();
    for (Edge edge : graph.edgesFrom(from)) {
      Vertex to = edge.to();
      if (!visited.contains(to)) {
        edgeTo.put(to, edge);
        distTo.put(to, distTo.get(from) + 1);
        perimeter.add(to);
        visited.add(to);
      }
    }
  }
  return edgeTo;
}
```

The start required no edge to arrive at, and is on level 0

Remember how we got to this point, and what layer this vertex is part of

BFS for Shortest Paths: Example



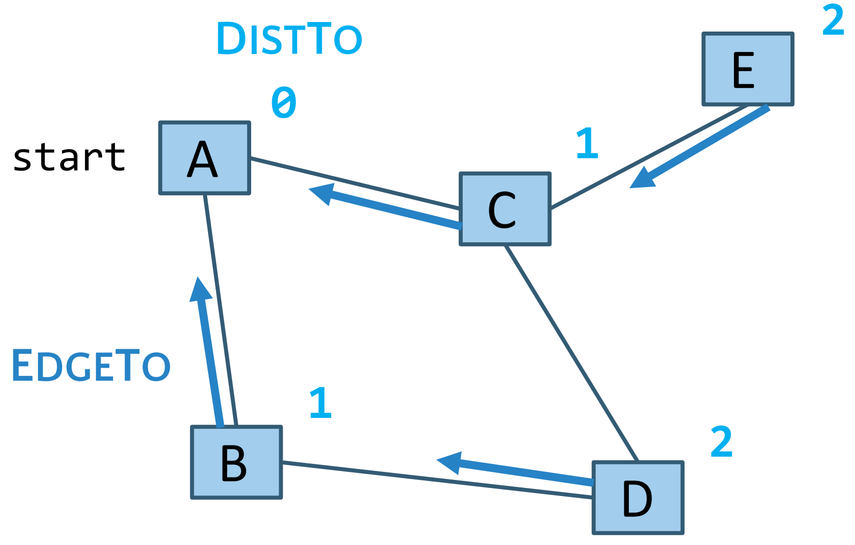
The **edgeTo** map stores **backpointers**: each vertex remembers what vertex was used to arrive at it!

Note: this code stores `visited`, `edgeTo`, and `distTo` as **external maps** (only drawn on graph for convenience). Another implementation option: store them as fields of the nodes themselves

```
  ...
  Map<Vertex, Edge> edgeTo = ...
  Map<Vertex, Double> distTo = ...

  edgeTo.put(start, null);
  distTo.put(start, 0.0);

  while (!perimeter.isEmpty()) {
    Vertex from = perimeter.remove();
    for (Edge edge : graph.edgesFrom(from)) {
      Vertex to = edge.to();
      if (!visited.contains(to)) {
        edgeTo.put(to, edge);
        distTo.put(to, distTo.get(from) + 1);
        perimeter.add(to);
        visited.add(to);
      }
    }
  }
  return edgeTo;
}
```

What about the Target Vertex?

Shortest Path Tree:



This modification on BFS didn't mention the target vertex at all!

Instead, it calculated the shortest path and distance from start to *every other vertex*

- This is called the **shortest path tree**
  - A general concept: in this implementation, made up of **distances** and **backpointers**

Shortest path tree has all the answers!

- **Length of shortest path from A to D?**
  - Lookup in **distTo** map: **2**
- **What's the shortest path from A to D?**
  - Build up backwards from **edgeTo** map: start at D, follow **backpointer** to B, follow **backpointer** to A – our shortest path is **A • B • D**

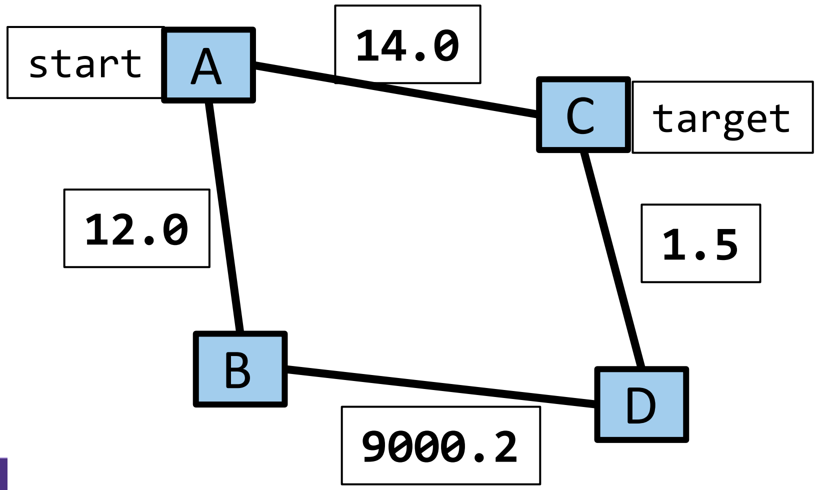
All our shortest path algorithms will have this property

- If you only care about t, you can sometimes stop early!

Recap: Graph Problems

Just like everything is Graphs, every problem is a Graph Problem

BFS and DFS are very useful tools to solve these! We'll see plenty more.



- What about the Shortest Path Problem on a weighted graph?
- Suppose we want to find shortest path from A to C, using weight of each edge as "distance"

s-t Connectivity Problem

Given source vertex **s** and a target vertex **t**, does there exist a path between **s** and **t**?

BFS or DFS + check if we've hit t

(Unweighted) Shortest Path Problem

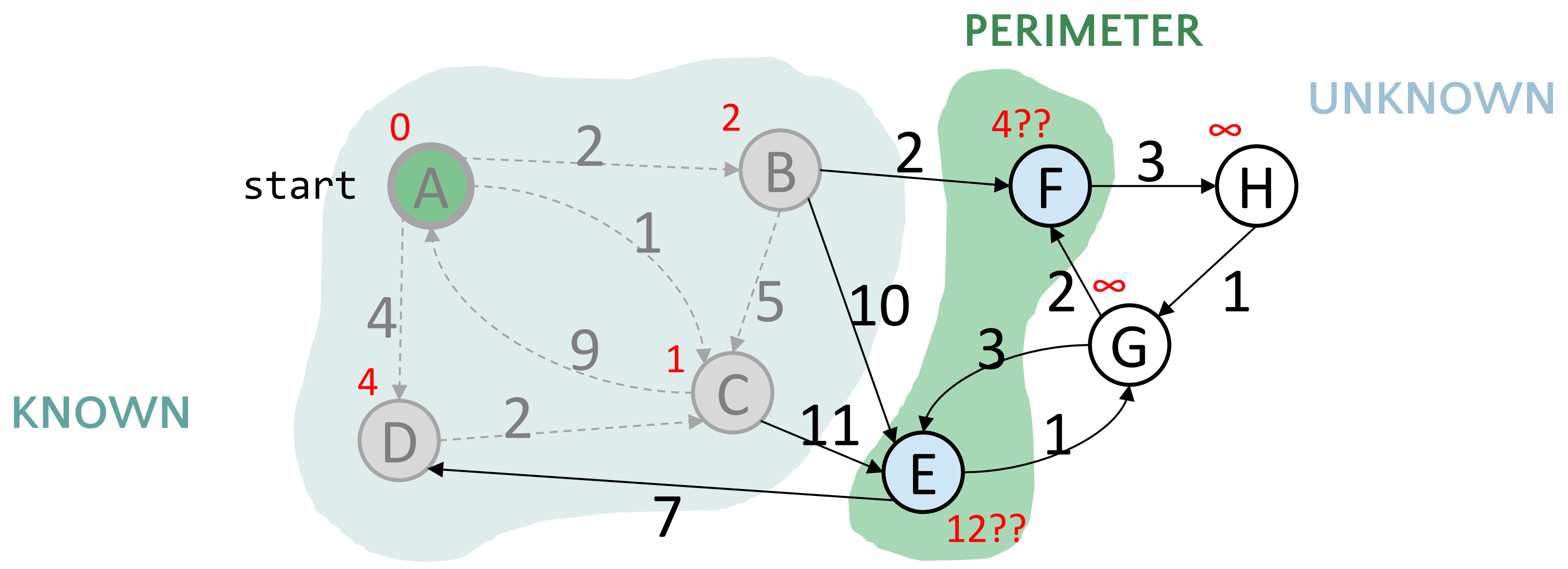
Given source vertex **s** and a target vertex **t**, how long is the shortest path from **s** to **t**? What edges make up that path?

BFS + generate shortest path tree as we go

Dijkstra's Algorithm

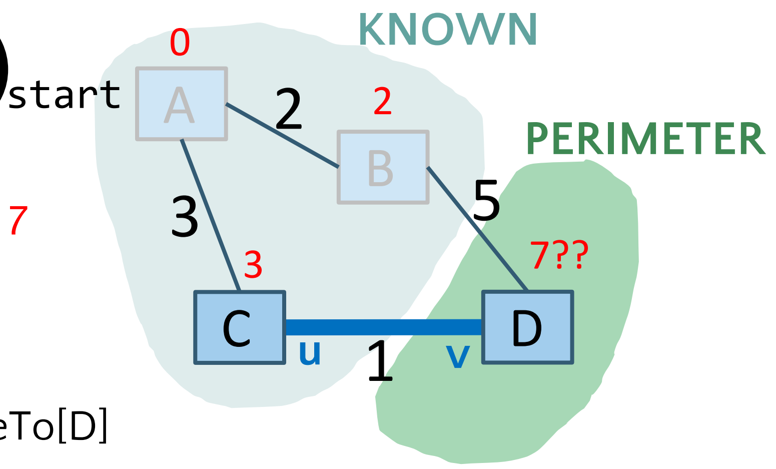
- Named after its inventor, Edsger Dijkstra (1930–2002)
    - Truly one of the "founders" of computer science
    - 1972 Turing Award
- The idea: reminiscent of BFS, but adapted to handle weights
    - Grow the set of nodes whose shortest distance has been computed
    - Nodes not in the set will have a "best distance so far"

Dijkstra's Algorithm: Idea



- Initialization:
    - Start vertex has distance **0**; all other vertices have distance **∞**
- At each step:
    - Pick closest unknown vertex v
    - Add it to the "cloud" of known vertices
    - Update "best-so-far" distances for vertices with edges from v

Dijkstra's Pseudocode (High-Level)



- Suppose we already visited B, distTo[D] = 7
- Now considering edge (C, D):
    - oldDist = 7
    - newDist = 3 + 1
    - That's better! Update distTo[D], edgeTo[D]

Similar to "visited" in BFS, "known" is nodes that are finalized (we know their path)

Dijkstra's algorithm is all about updating "best-so-far" in distTo if we find shorter path! Init all paths to infinite.

Order matters: always visit closest first!

Consider all vertices reachable from me: would getting there *through* me be a shorter path than they currently know about?

```
dijkstraShortestPath(G graph, V start)
  Set known; Map edgeTo, distTo;
  initialize distTo with all nodes mapped to ∞, except start to 0

  while (there are unknown vertices):
    let u be the closest unknown vertex
    known.add(u);
    for each edge (u,v) from u with weight w:
      oldDist = distTo.get(v)      // previous best path to v
      newDist = distTo.get(u) + w  // what if we went through u?
      if (newDist < oldDist):
        distTo.put(v, newDist)
        edgeTo.put(v, u)
```

Dijkstra's Algorithm: Key Properties

Once a vertex is marked known, its shortest path is known

- Can reconstruct path by following back-pointers (in edgeTo map)

While a vertex is not known, another shorter path might be found

- We call this update **relaxing** the distance because it only ever shortens the current best path

Going through closest vertices first lets us confidently say no shorter path will be found once known

- Because not possible to find a shorter path that uses a farther vertex we'll consider later

```
dijkstraShortestPath(G graph, V start)
  Set known; Map edgeTo, distTo;
  initialize distTo with all nodes mapped to ∞, except start to 0

  while (there are unknown vertices):
    let u be the closest unknown vertex
    known.add(u)
    for each edge (u,v) to unknown v with weight w:
      oldDist = distTo.get(v)      // previous best path to v
      newDist = distTo.get(u) + w  // what if we went through u?
      if (newDist < oldDist):
        distTo.put(v, newDist)
        edgeTo.put(v, u)
```

Dijkstra's Algorithm: Runtime

Important for P4!

```
dijkstraShortestPath(G graph, V start)
  Set known; Map edgeTo, distTo;
  initialize distTo with all nodes mapped to ∞, except start to 0

  while (there are unknown vertices):
    let u be the closest unknown vertex
    known.add(u)
    for each edge (u,v) to unknown v with weight w:
      oldDist = distTo.get(v)      // previous best path to v
      newDist = distTo.get(u) + w  // what if we went through u?
      if (newDist < oldDist):
        distTo.put(v, newDist)
        edgeTo.put(v, u)
```

O(|V|)

come back…

How do we find this??

O(1) for HashSet

O(|E|) worst case

O(1) for HashMap

We can use an optimized structure that will tell us the "minimum" distance vertex, and let us "update distance" as we go…

Use a **HeapMinPriorityQueue**! (like the one from P3)

Dijkstra's Algorithm: Runtime

```
dijkstraShortestPath(G graph, V start)
  Set known; Map edgeTo, distTo;
  initialize distTo with all nodes mapped to ∞, except start to 0

  while (there are unknown vertices):
    let u be the closest unknown vertex
    known.add(u)
    for each edge (u,v) to unknown v with weight w:
      oldDist = distTo.get(v)      // previous best path to v
      newDist = distTo.get(u) + w  // what if we went through u?
      if (newDist < oldDist):
        distTo.put(v, newDist)
        edgeTo.put(v, u)
        update distance in list of unknown vertices
```

O(|V|)

O(|V|)

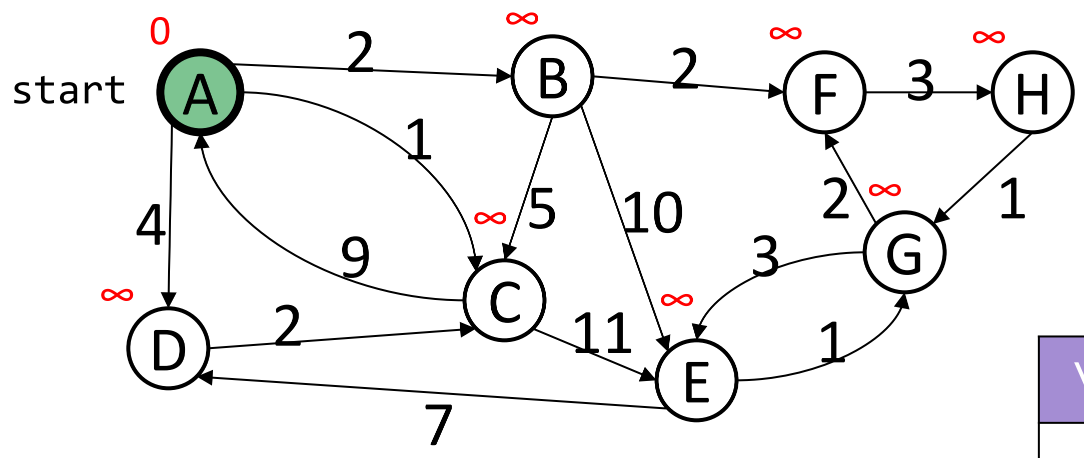
O(log|V|)

O(|E|)

O(log|V|)

Final runtime: $O(|V|\log|V| + |E|\log|V|)$

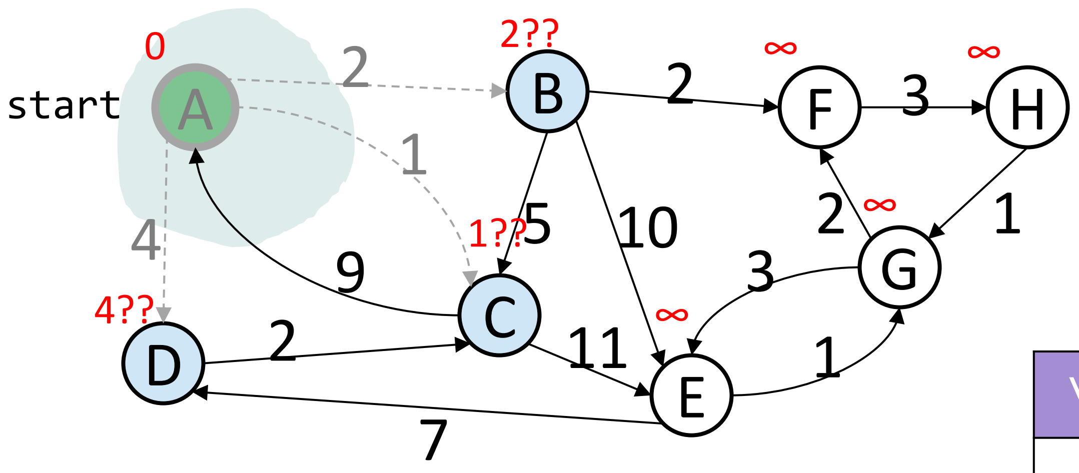
Dijkstra's Algorithm: Example #1



Order Added to Known Set:

Vertex	Known?	distTo	edgeTo
A		∞	
B		∞	
C		∞	
D		∞	
E		∞	
F		∞	
G		∞	
H		∞	

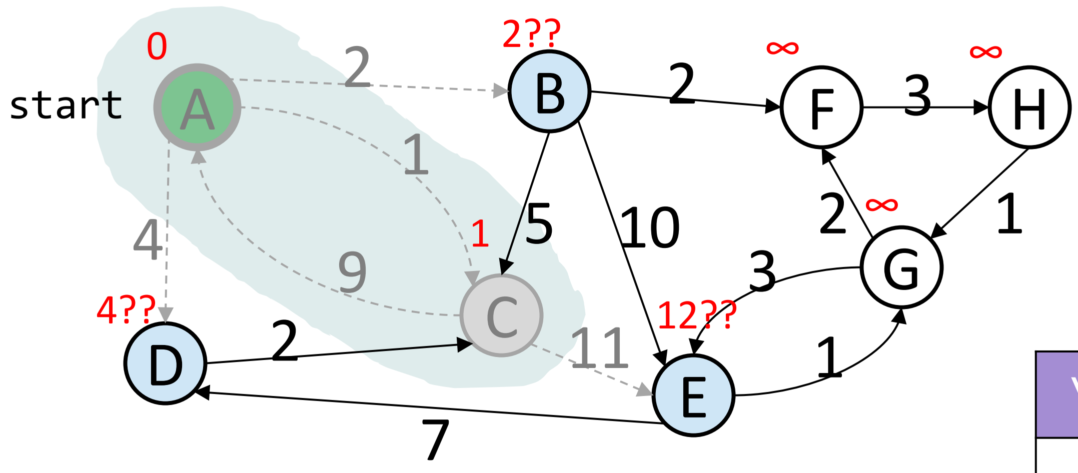
Dijkstra's Algorithm: Example #1



Vertex	Known?	distTo	edgeTo
A	Y	0	/
B		≤ 2	A
C		≤ 1	A
D		≤ 4	A
E		∞	
F		∞	
G		∞	
H		∞	

Order Added to Known Set:
A

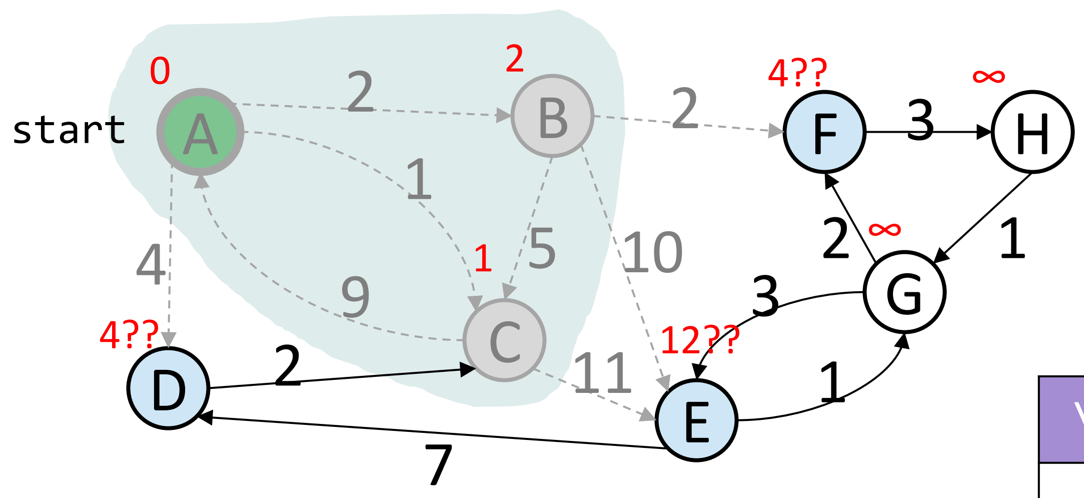
Dijkstra's Algorithm: Example #1



Order Added to Known Set:
A, C

Vertex	Known?	distTo	edgeTo
A	Y	0	/
B		≤ 2	A
C	Y	1	A
D		≤ 4	A
E		**≤ 12**	**C**
F		∞	
G		∞	
H		∞	

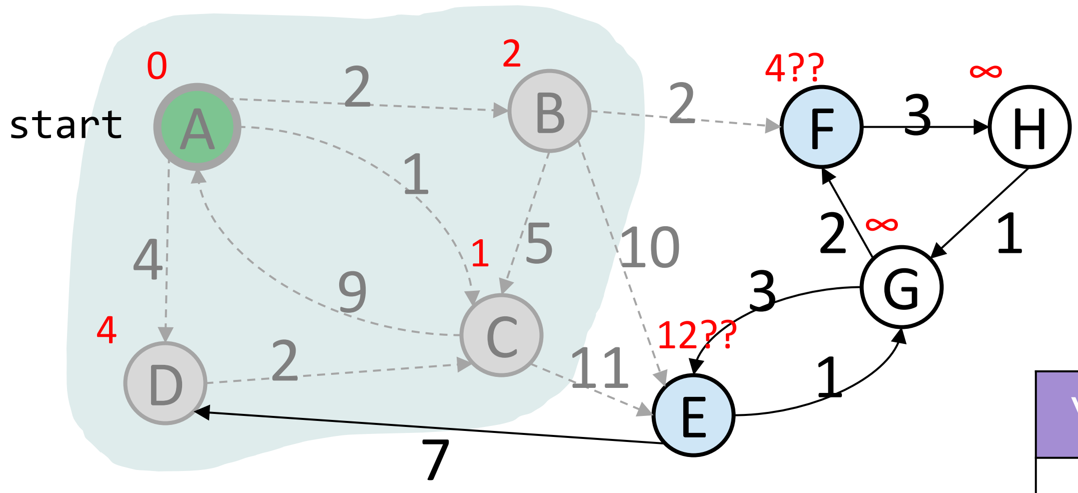
Dijkstra's Algorithm: Example #1



Vertex	Known?	distTo	edgeTo
A	Y	0	/
B	Y	2	A
C	Y	1	A
D		≤ 4	A
E		≤ 12	C
F		**≤ 4**	**B**
G		∞	
H		∞	

Order Added to Known Set:
A, C, B

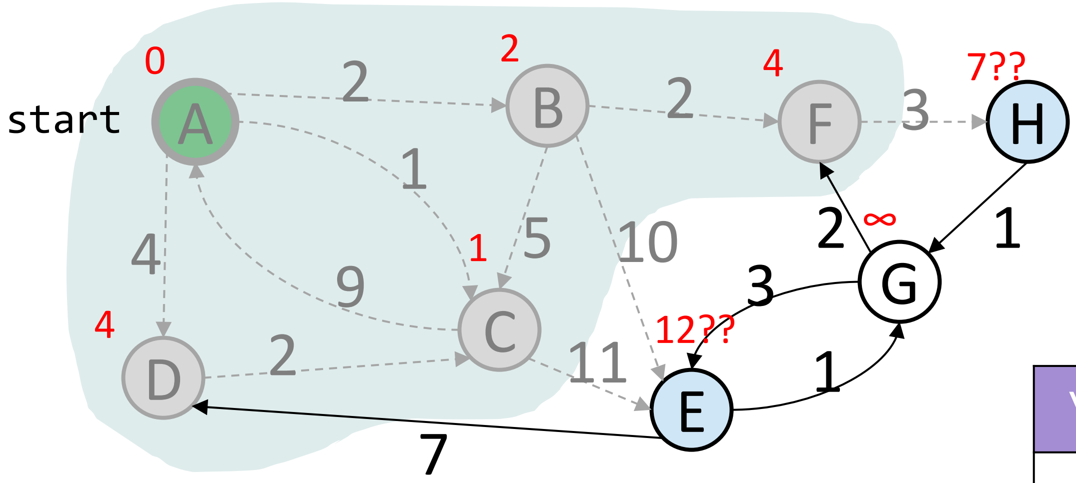
Dijkstra's Algorithm: Example #1



Vertex	Known?	distTo	edgeTo
A	Y	0	/
B	Y	2	A
C	Y	1	A
D	Y	4	A
E		≤ 12	C
F		≤ 4	B
G		∞	
H		∞	

<u>Order Added to Known Set:</u>
A, C, B, D

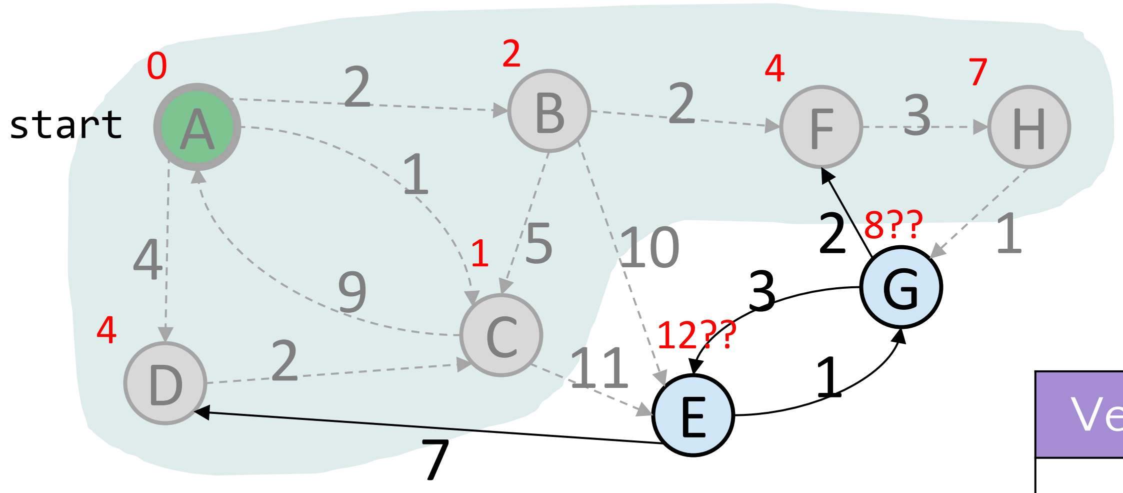
Dijkstra's Algorithm: Example #1



Order Added to Known Set:
A, C, B, D, F

Vertex	Known?	distTo	edgeTo
A	Y	0	/
B	Y	2	A
C	Y	1	A
D	Y	4	A
E		≤ 12	C
F	Y	4	B
G		∞	
H		**≤ 7**	**F**

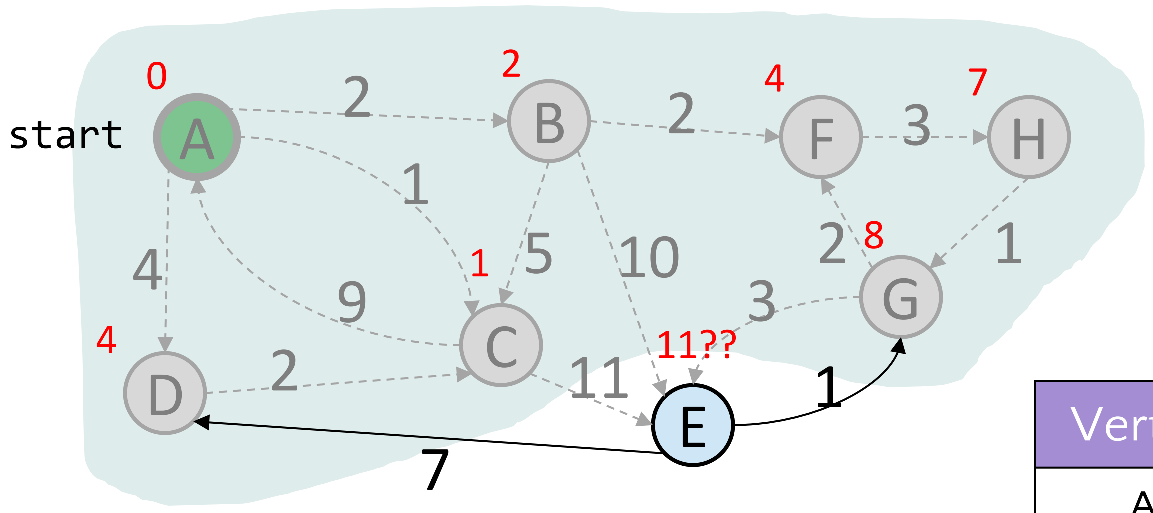
Dijkstra's Algorithm: Example #1



Order Added to Known Set:
A, C, B, D, F, H

Vertex	Known?	distTo	edgeTo
A	Y	0	/
B	Y	2	A
C	Y	1	A
D	Y	4	A
E		≤ 12	C
F	Y	4	B
G		**≤ 8**	**H**
H	Y	7	F

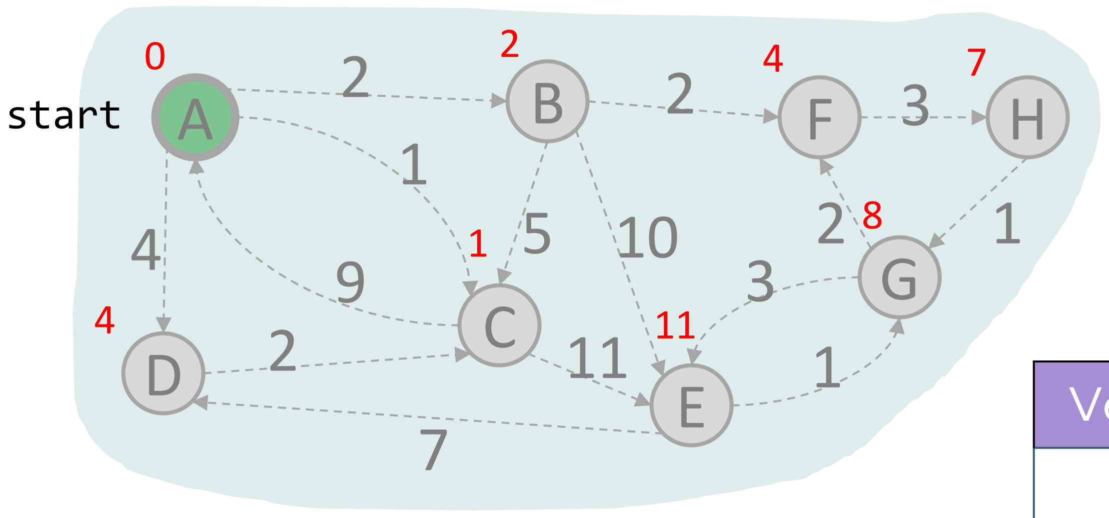
Dijkstra's Algorithm: Example #1



Vertex	Known?	distTo	edgeTo
A	Y	0	/
B	Y	2	A
C	Y	1	A
D	Y	4	A
E		≤ 11	G
F	Y	4	B
G	Y	8	H
H	Y	7	F

Order Added to Known Set:
A, C, B, D, F, H, G

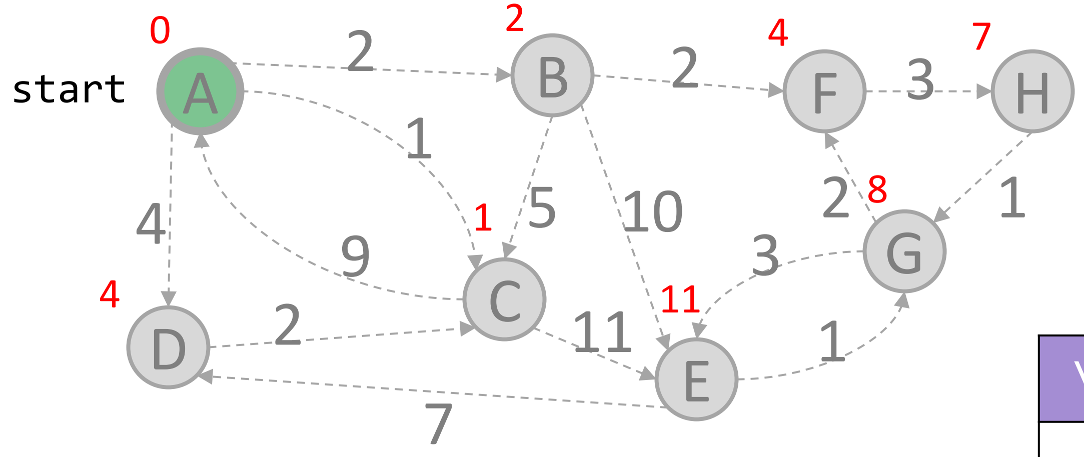
Dijkstra's Algorithm: Example #1



Order Added to Known Set:
A, C, B, D, F, H, G, E

Vertex	Known?	distTo	edgeTo
A	Y	0	/
B	Y	2	A
C	Y	1	A
D	Y	4	A
E	Y	11	G
F	Y	4	B
G	Y	8	H
H	Y	7	F

Dijkstra's Algorithm: Interpreting the Results



Now that we're done, how do we get the path from A to E?

- Follow edgeTo backpointers!
- distTo and edgeTo make up the **shortest path tree**

Vertex	Known?	distTo	edgeTo
A	Y	0	/
B	Y	2	A
C	Y	1	A
D	Y	4	A
E	Y	11	G
F	Y	4	B
G	Y	8	H
H	Y	7	F

Order Added to Known Set:
A, C, B, D, F, H, G, E

Review: Key Features

- Once a vertex is marked known, its shortest path is known
  - Can reconstruct path by following backpointers
- While a vertex is not known, another shorter path might be found!
- The "Order Added to Known Set" is unimportant
  - A detail about how the algorithm works *(client doesn't care)*
  - Not used by the algorithm *(implementation doesn't care)*
  - It is sorted by path-distance; ties are resolved "somehow"
- If we only need path to a specific vertex, can stop early once that vertex is known
  - Because its shortest path cannot change!
  - Return a partial **shortest path tree**

Greedy Algorithms

- At each step, do what seems best at that step
    - “instant gratification”
    - “make the locally optimal choice at each stage”
- Dijkstra’s is “greedy” because once a vertex is marked as “processed” we never revisit
    - This is why Dijkstra’s does not work with negative edge weights

Other examples of greedy algorithms are:

- Kruskal and Prim’s minimum spanning tree algorithms (*next week*)
- Huffman compression

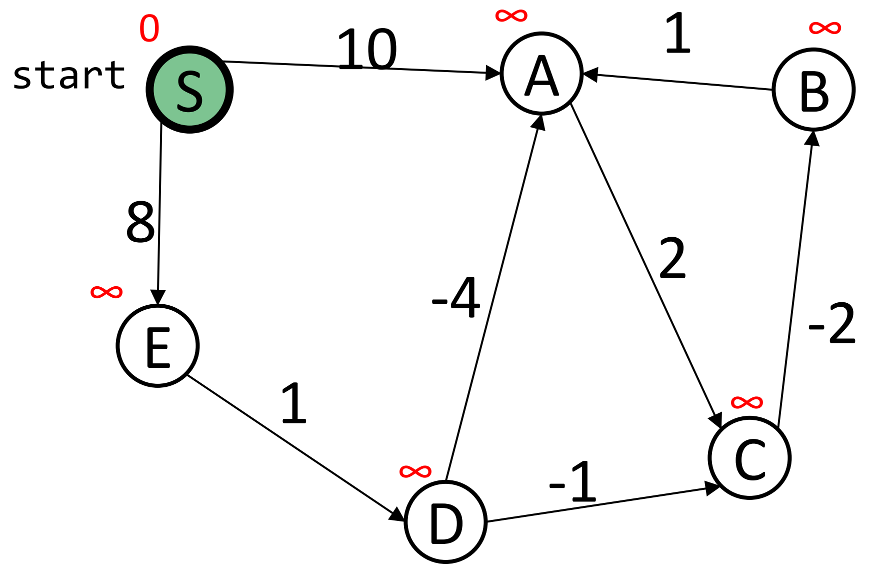
Bellman-Ford Shortest Path

- A shortest path algorithm that will work with negative edge weights
  - Will **not** work if a negative cycle exists- in this case no shortest path exists
- **Not** a greedy algorithm
- Originally proposed by Alfonso Shimbel, then published by Edward F. Moore (Moore's Finite State Machine, not of Moore's law), then republished by Lester Ford Jr and finally named after Richard Bellman (invented dynamic programming) who's final publication built off of Ford's

Bellman-Ford Basics

- There can be at most |V| - 1 edges in our shortest path
    - If there are |V| or more edges in a path that means there's a cycle/repeated Vertex
- Run |V| - 1 iterations of shortest path analysis through the graph
    - This means we will repeatedly revisit the "distance from" selected per vertex
- Look at each vertex's outgoing edges in each iteration
- It is slower than Dijkstra's for the same problem because it will revisit previously assessed vertices

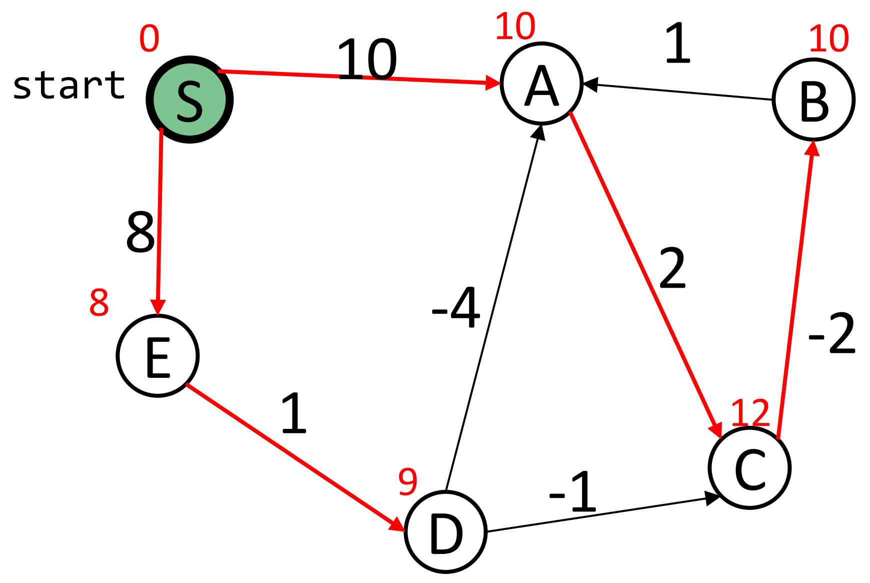
Bellman-Ford Example



Vertex	distTo	edgeTo
S	0	
A	∞	
B	∞	
C	∞	
D	∞	
E	∞	

Example Walk Through Video (5min)

Bellman-Ford Example

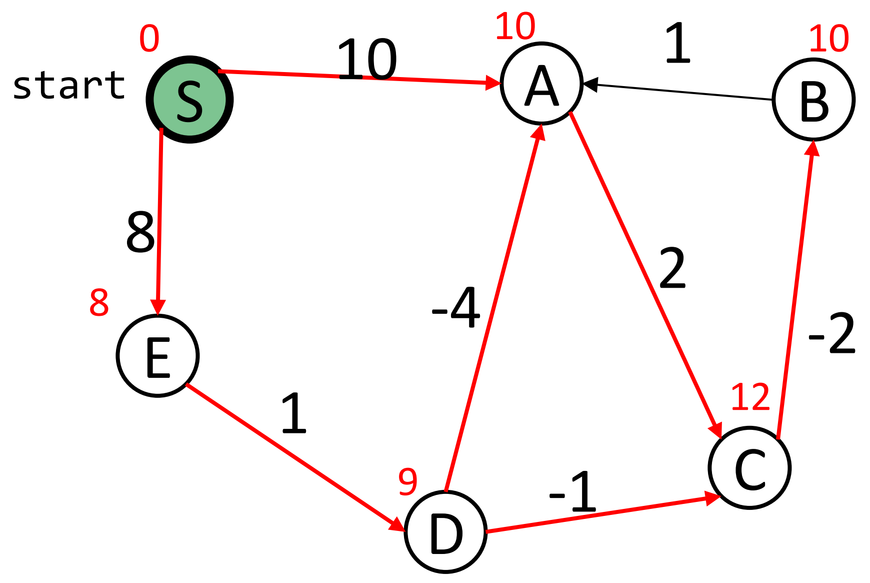


Iteration 1 – for each Vertex's outgoing edge, does that give us a shorter way to get to a new vertex?

Vertex	distTo	edgeTo
S	0	-
A	10	S
B	10	C
C	12	A
D	9	E
E	8	A

Example Walk Through Video (5min)

Bellman-Ford Example



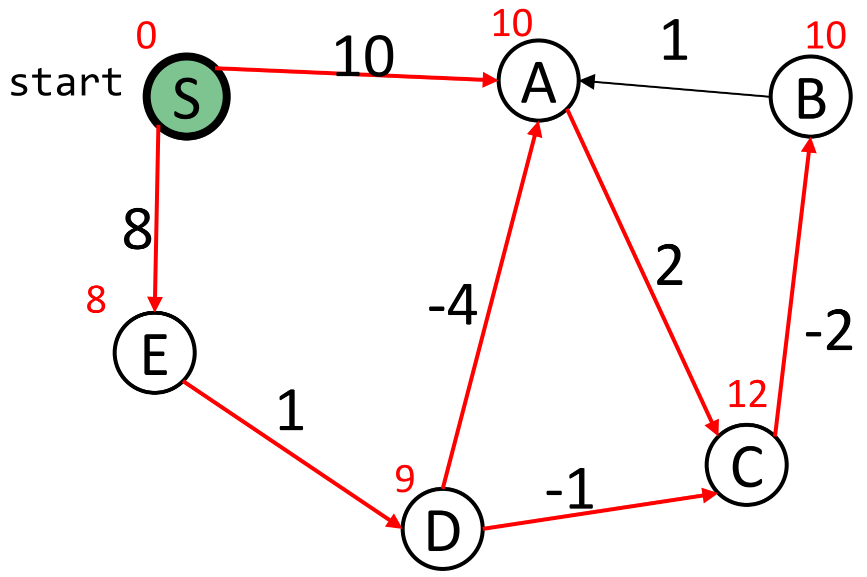
Iteration 2 - re-examining outgoing edges, can we improve the distance to any given Vertex?

Vertex	distTo	edgeTo
S	0	-
A	~~10~~ **5**	~~S~~ **D**
B	10	C
C	~~12~~ **8**	~~A~~ **D**
D	9	E
E	8	A

* Because a distance to D is known by the time we process D we can include D's outgoing edges for consideration

Example Walk Through Video (5min)

Bellman-Ford Example



Iteration 3 – repeat!

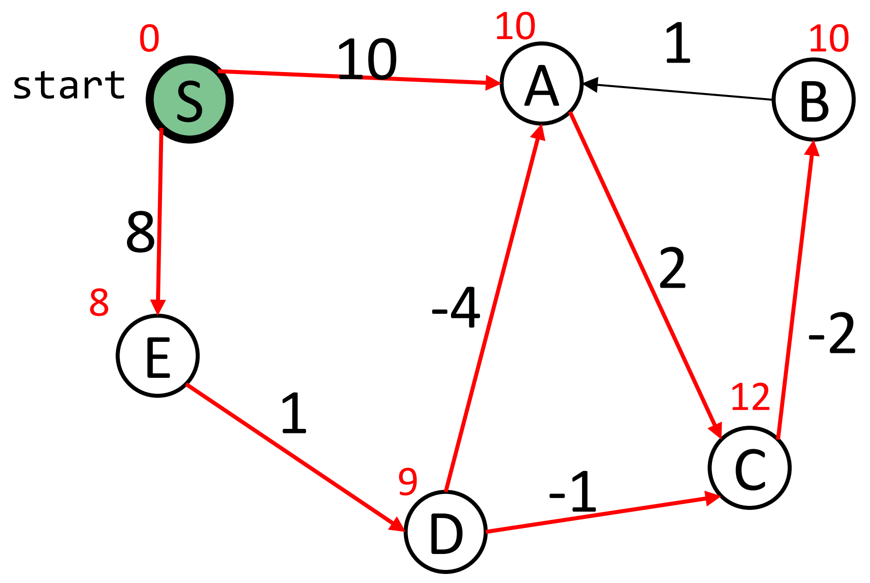
Vertex	distTo	edgeTo
S	0	-
A	5	D
B	~~10~~ **5**	C
C	~~8~~ **7**	A
D	9	E
E	8	A

* With a shortened distance to C from this iteration we can improve distance to B

* With a shortened distance to A from iteration 2 we can improve the distance to C

Example Walk Through Video (5min)

Bellman-Ford Example



Iteration 4 – repeat!

Vertex	distTo	edgeTo
S	0	–
A	5	D
B	5	C
C	7	A
D	9	E
E	8	A

No changes!
this means we can stop early

Example Walk Through Video (5min)

- Breadth-First Search Visualized and Explained
  - https://www.youtube.com/watch?v=N6wicLpEmHY&list=PLnZHgAO8ocBv6XRqZkqQjrsIJijn82UUC&index=5
- Depth-First Search Visualized and Explained
  - https://www.youtube.com/watch?v=5GcSvYDgiSo&list=PLnZHgAO8ocBv6XRqZkqQjrsIJijn82UUC&index=6
- Topological Sort Visualized and Explained
  - https://www.youtube.com/watch?v=7J3GadLzydI&list=PLnZHgAO8ocBv6XRqZkqQjrsIJijn82UUC&index=7