Lecture 12
Graphs

Department of Computer Science
Hofstra University

Acknowledgement: Lecture slides based on UofW Course on Data Structures

Inter-data Relationships

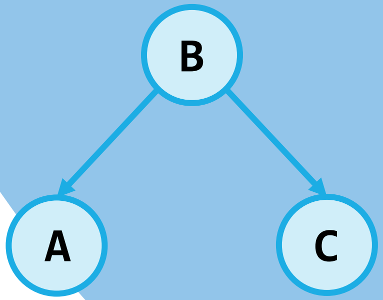
Arrays

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

0	1	2
A	B	C

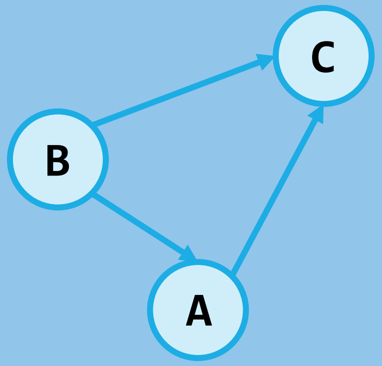
Trees

- 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

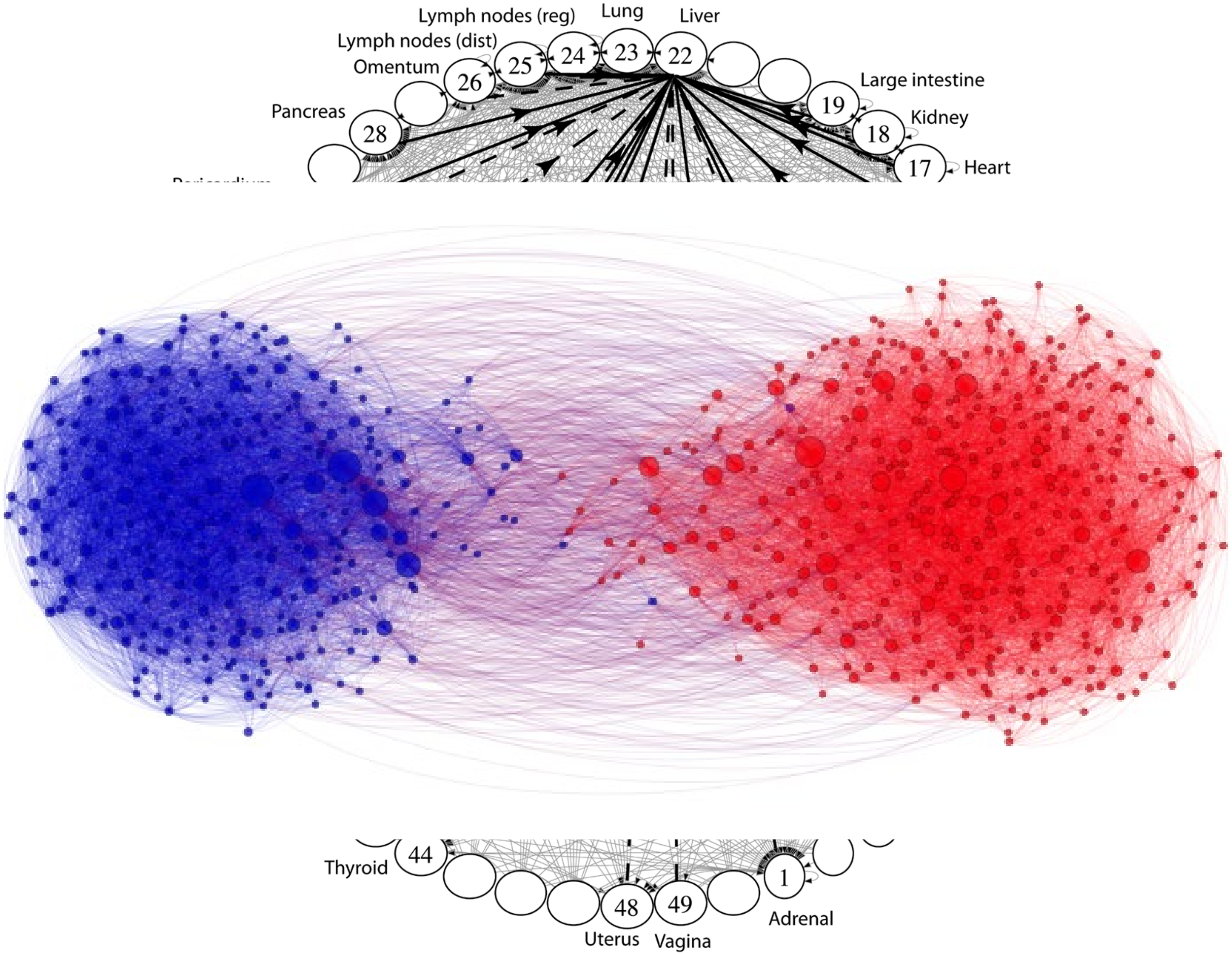
- Social media graphs
  - Vertices are accounts, edges are follower relationships
- Traffic
  - Vertices are classes, edges are usage

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



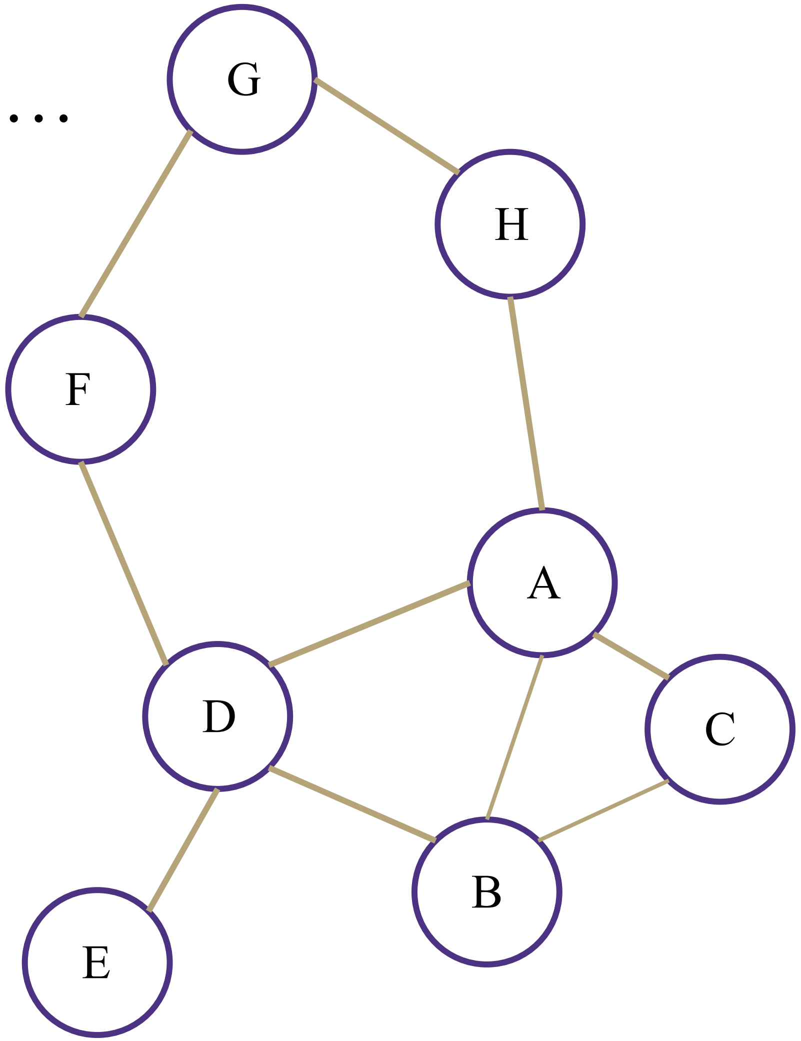
And so many more!!

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)}

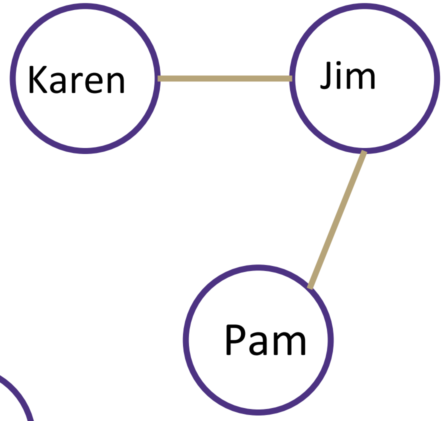


Graph Terminology

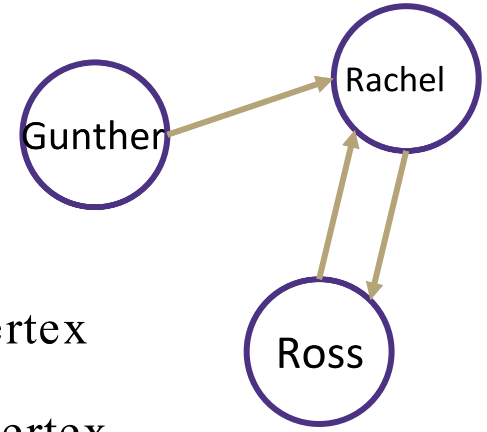
Graph Direction

- 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) }

Undirected Graph:



Directed Graph:



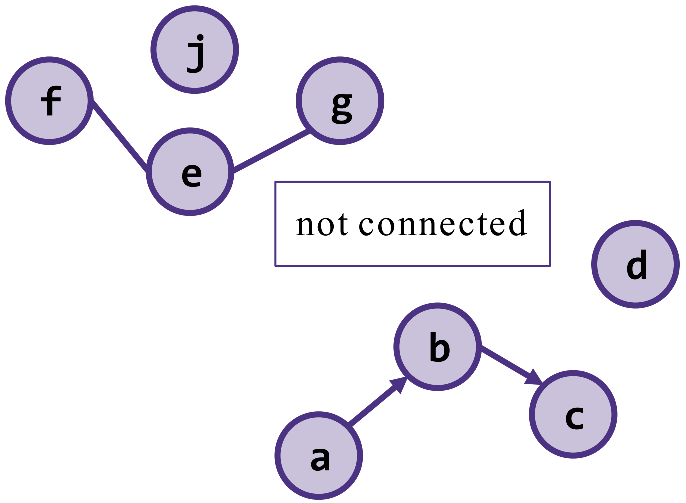
Degree of a Vertex

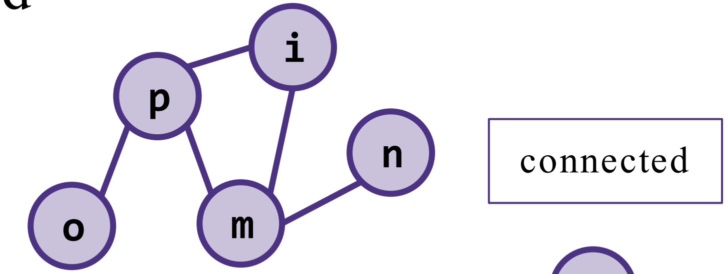
- Degree – the number of edges connected to that vertex
  - Karen : 1, Jim : 1, Pam : 1
- In-degree – the number of directed edges that point to a vertex
  - Gunther : 0 , Rachel : 2, Ross : 1
- Out-degree – the number of directed edges that start at a vertex
  - Gunther : 1, Rachel : 1, Ross : 1

More Graph Terminology

Two vertices are connected if there is a path between them

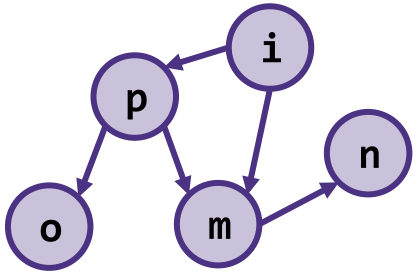
- If all the vertices are connected, we say the graph is connected
    - A directed graph is weakly connected if replacing every directed edge with an undirected edge results in a connected graph
    - A directed graph is strongly connected if a directed path exists between every pair of vertices
- The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



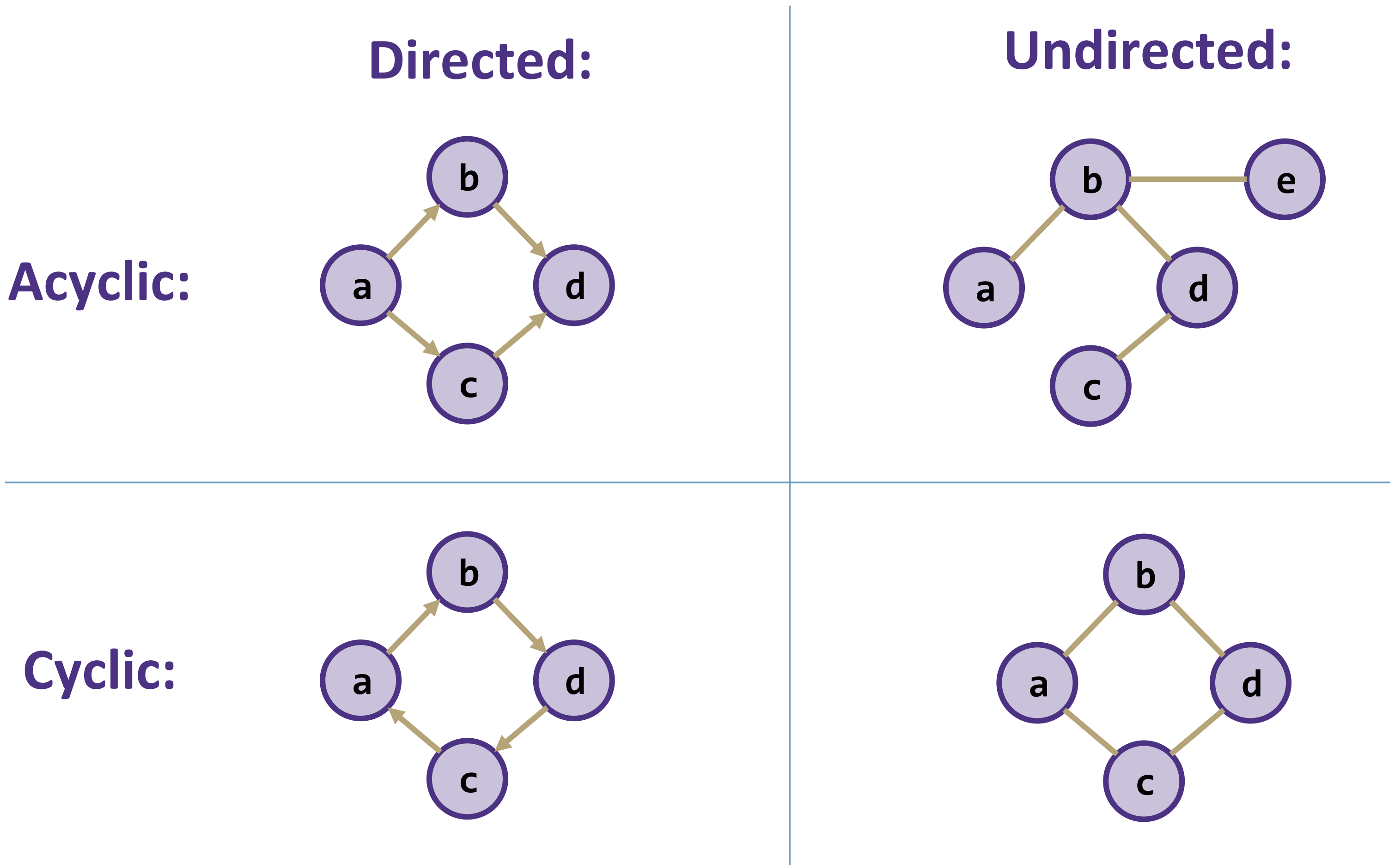


A path is a sequence of vertices connected by edges

- 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

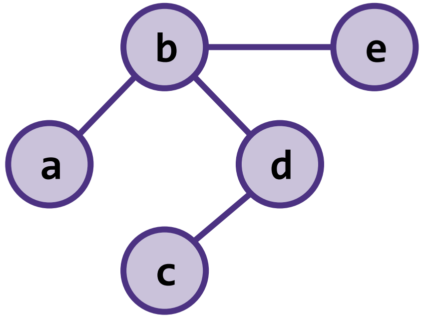


Directed vs Undirected; Acyclic vs Cyclic

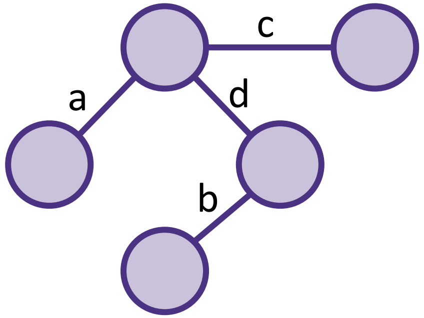


Labeled and Weighted Graphs

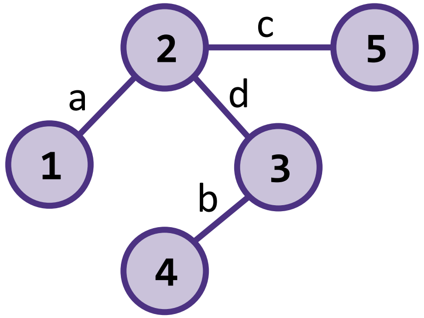
Vertex Labels



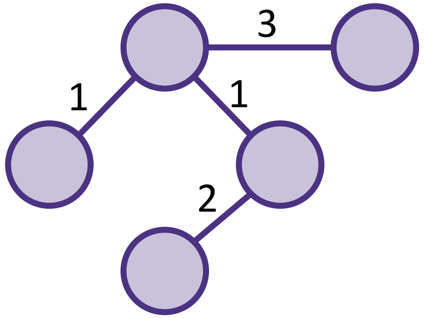
Edge Labels



Vertex & Edge Labels



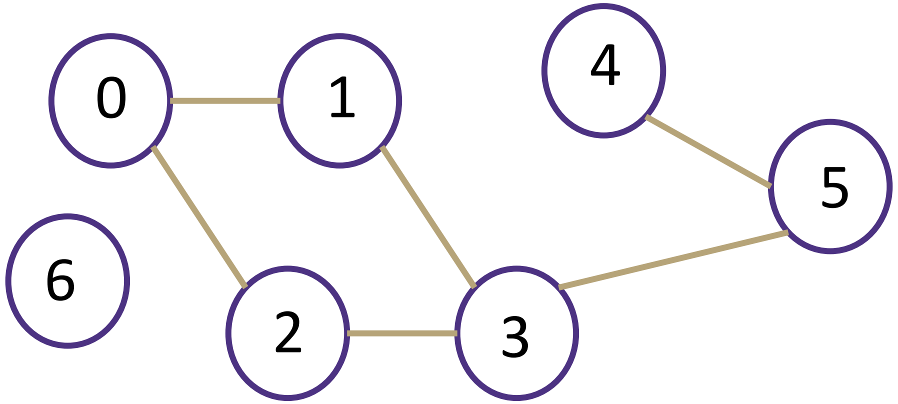
Numeric Edge Labels (Edge Weights)



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.

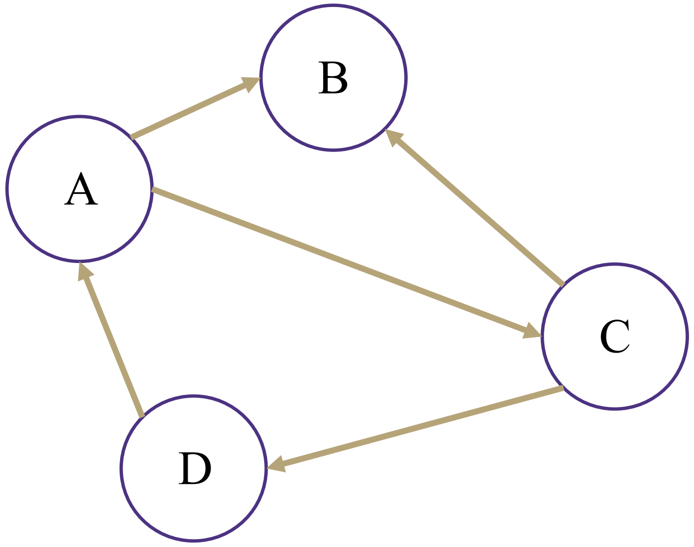
Worst-case Time Complexity (|V| = n, |E| = m):

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

Space Complexity: O(n * n)

	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

Adjacency List



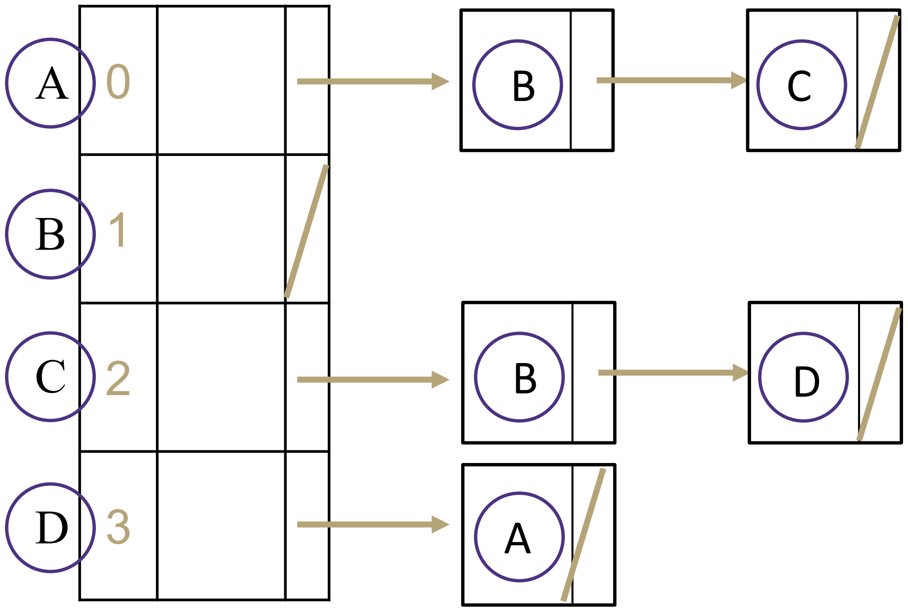
In an adjacency matrix a[u][v] is 1 if there is an edge (u,v), and 0 otherwise.

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: $O(n + m)$

Linked Lists



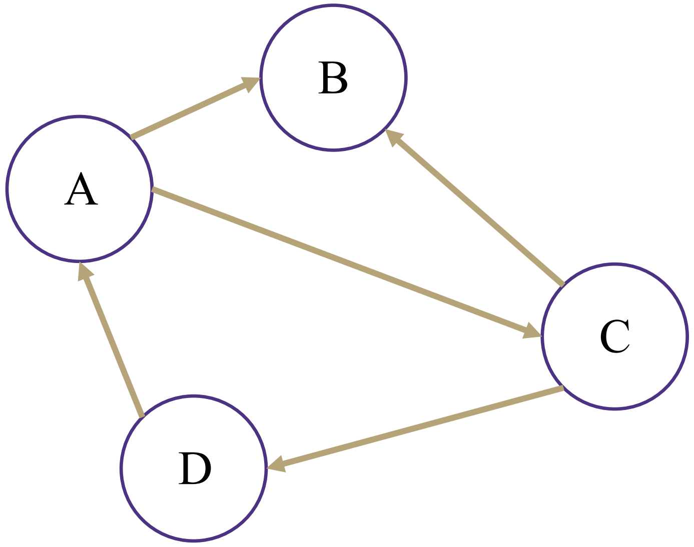
Adjacency List

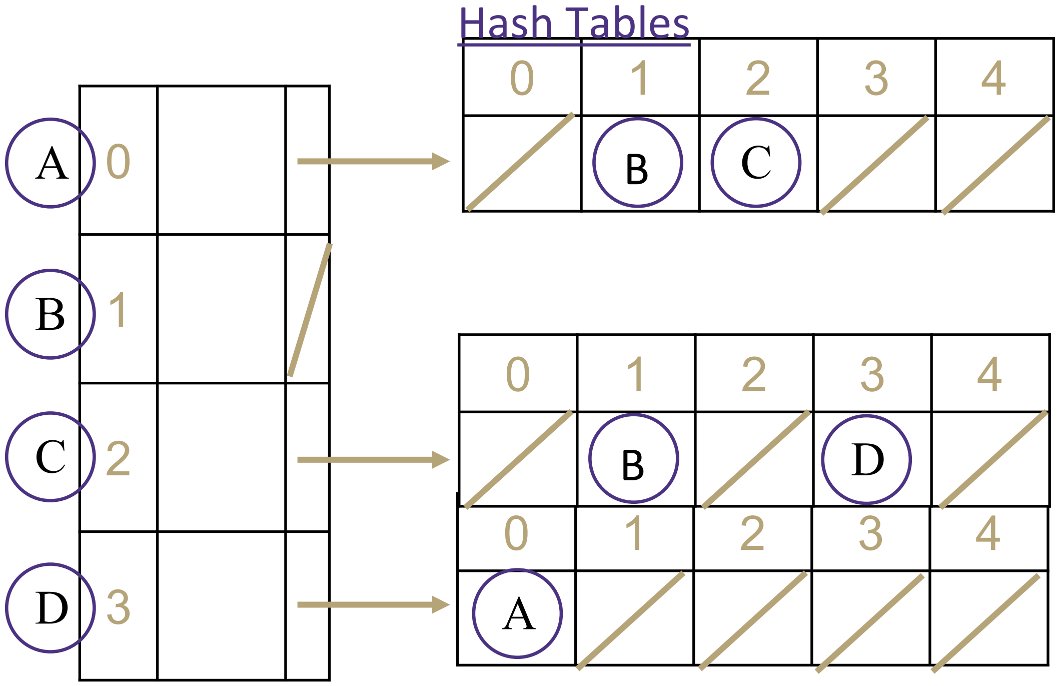
In an adjacency matrix a[u][v] is 1 if there is an edge (u,v), and 0 otherwise.

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)





Tradeoffs

Adjacency Matrices take more space, why would you use them?

- For dense graphs (where m is close to n^2), the running times will be close
- And the constant factors can be much better for matrices than for lists

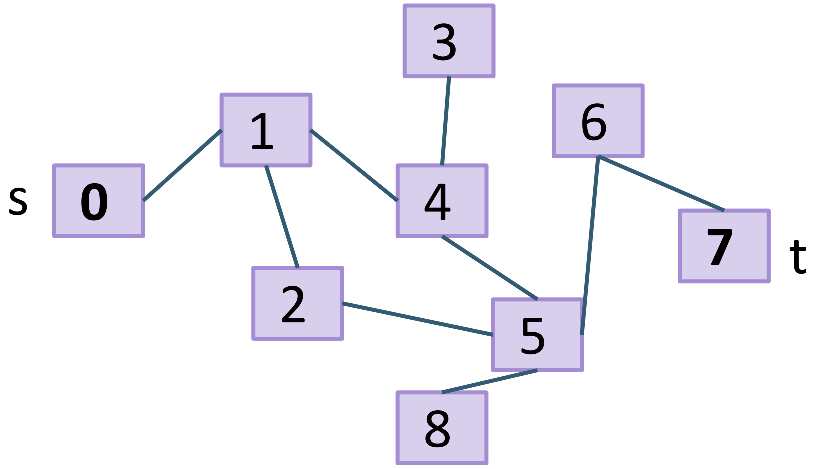
Graph Traversals

Topological Sort

Shortest Path

s-t Connectivity Problem

s-t Connectivity Problem
Given source vertex s and a target vertex t, does there exist a path between s and t?



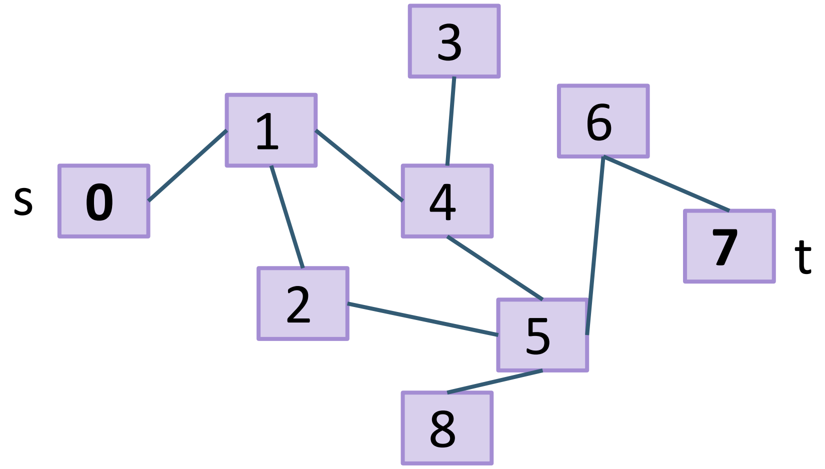
An algorithm for `connected(s, t)`

- We can use recursion: if a neighbor of s is connected to t, that means s is also connected to t!

s-t Connectivity Problem with Recursion

Solution: Mark each node as visited!

```
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;
  }
}
```



This general approach for crawling through a graph is going to be the basis for a LOT of algorithms!

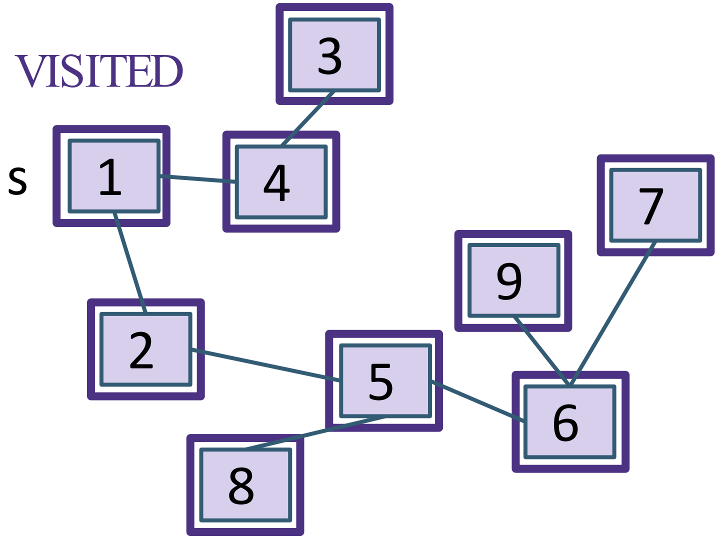
Recursive Depth-First Search (DFS)

What order does this algorithm use to visit nodes?

- Assume order of `s.neighbors` is arbitrary!

```
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;
  }
}
```

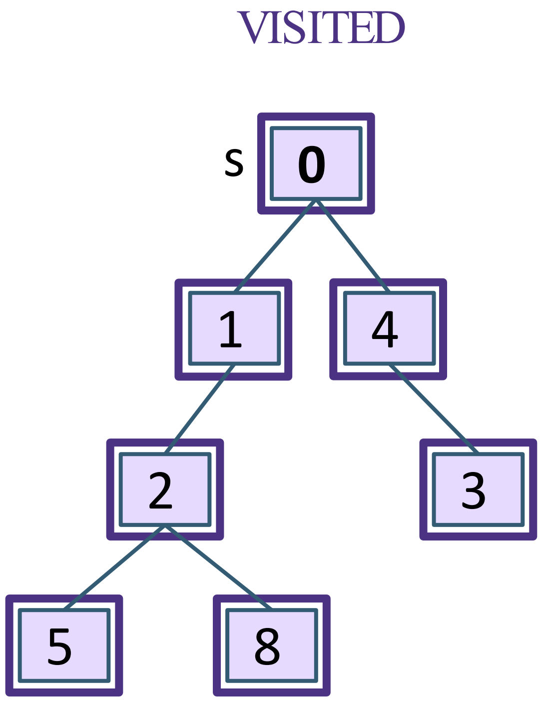
- It will explore one option "all the way down" before coming back to try other options
  - Many possible orderings: e.g., {1, 2, 5, 6, 9, 7, 8, 4, 3} or {1, 4, 3, 2, 5, 8, 6, 7, 9} both possible
- We call this approach a depth-first search (DFS)



Aside Tree Traversals

We could also apply this code to a tree (recall: a type of graph) to do a depth-first search on it

```
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;
  }
}
```



Recall: DFS traversal of a binary tree has 3 options:

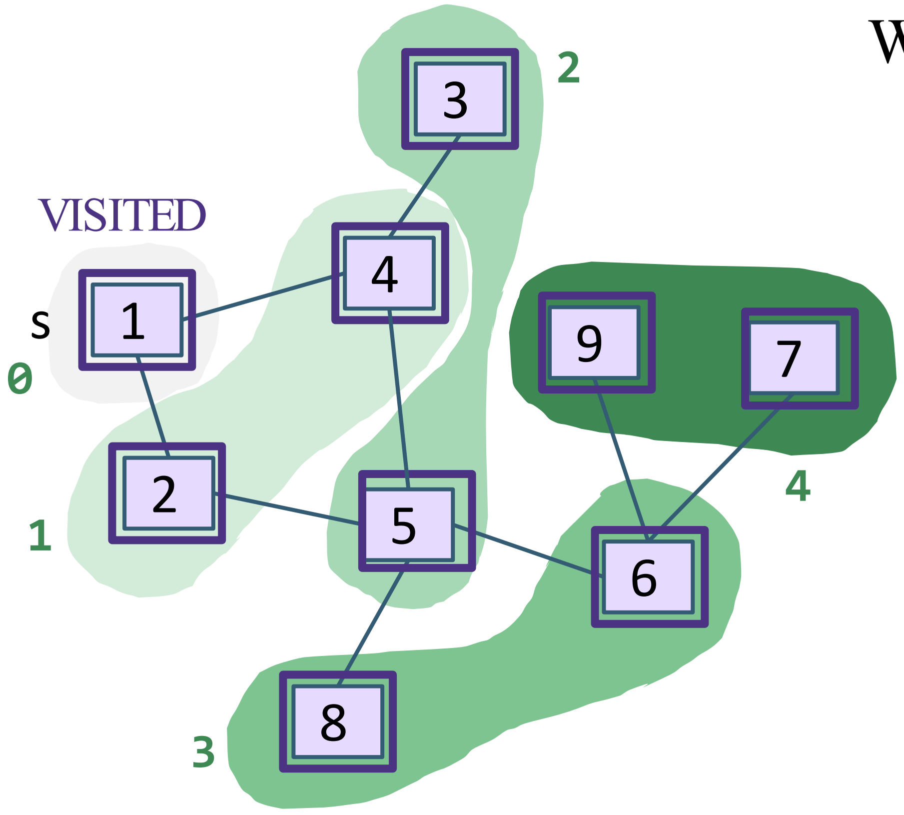
- Pre-order: visit node before its children
- In-order: visit node between its children
- Post-order: visit node after its children

The difference between these orderings is when we "process" the root – all are DFS!

Breadth-First Search (BFS)

Suppose we want to visit closer nodes first, instead of following one choice all the way to the end

- Just like level-order traversal of a tree, now generalized to any graph



We call this approach a breadth-first search (BFS)

- Explore "layer by layer"

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

- Key observation: recursive calls interrupted s.neighbors loop to immediately process children
- For BFS, instead we want to complete that loop before processing children
- Recursion isn't the answer! Need a data structure to "queue up" children…

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

BFS Implementation

Let's make this a bit more realistic and add a Graph

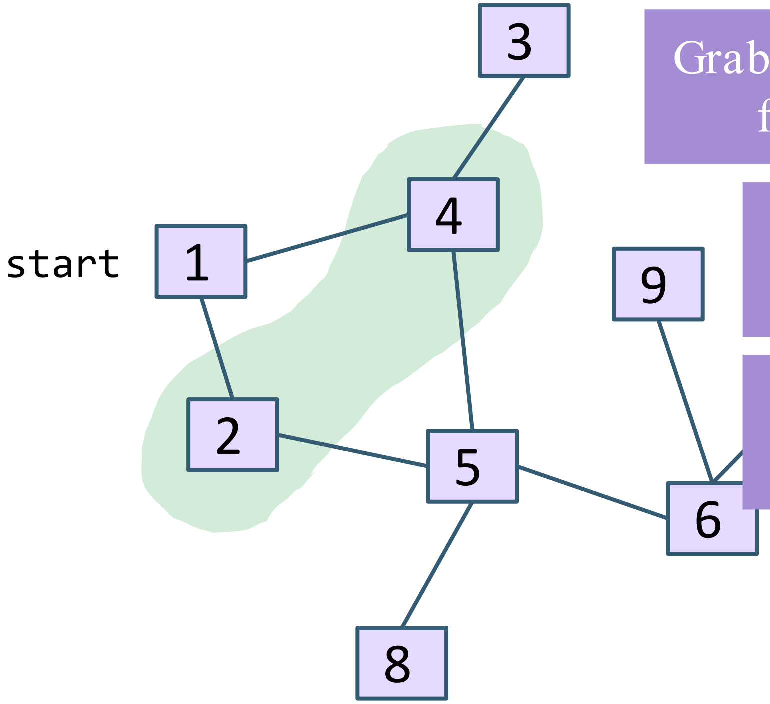
Our extra data structure! Will keep track of "outer edge" of nodes still to explore

Kick off the algorithm by adding start to perimeter

Grab one element at a time from the perimeter

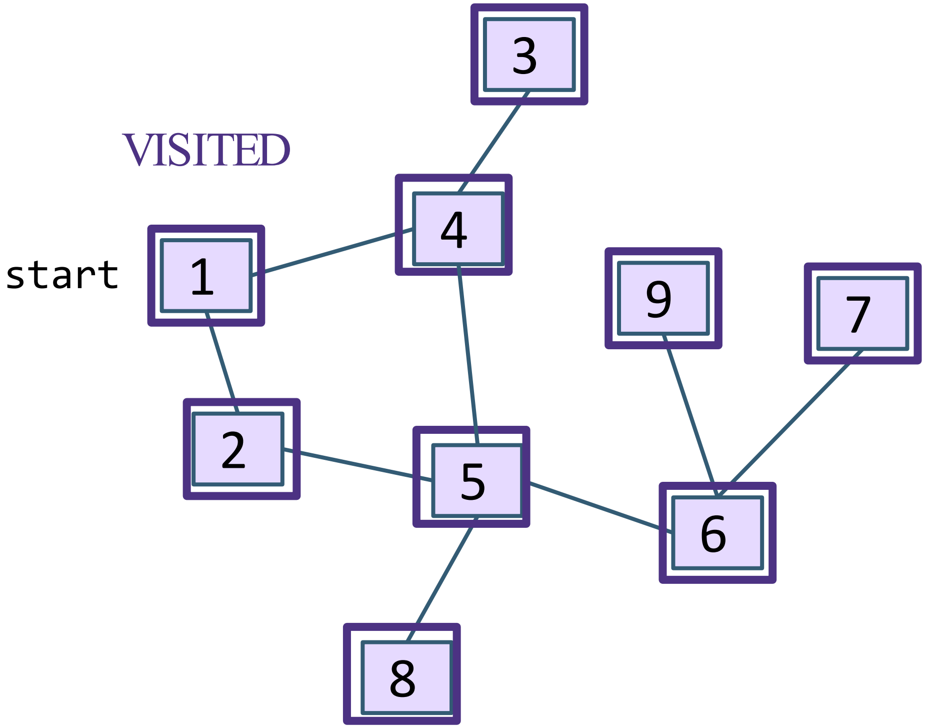
Look at all that element's children

Add new ones to perimeter!



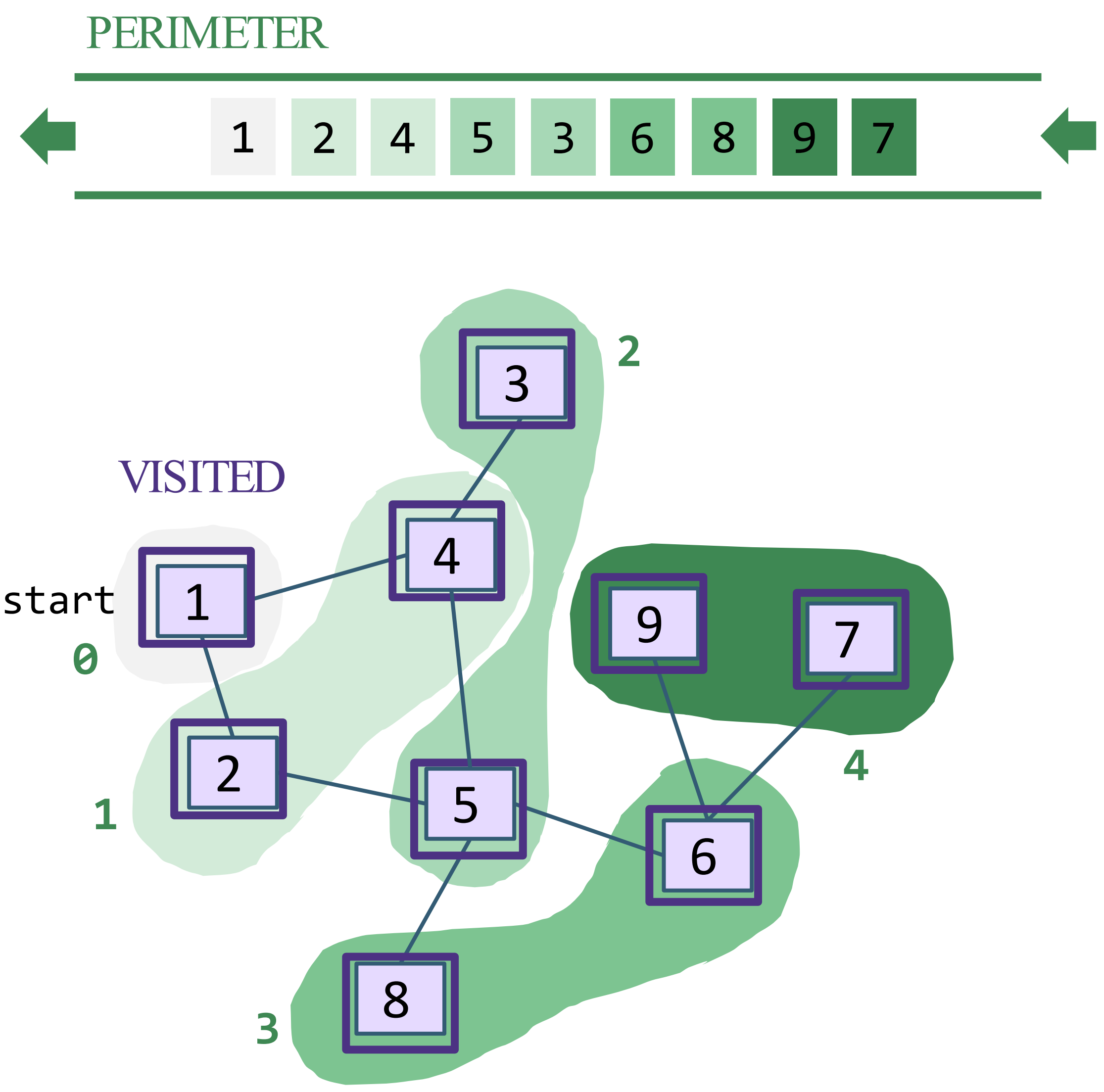

```
bfs(Graph graph, Vertex start) {
  Queue<Vertex> perimeter = new Queue<>();
  Set<Vertex> visited = new Set<>();

  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);
      }
    }
  }
}
```

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);
    }
  }
}
```

Change Queue for order to process neighbors to a Stack

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

       add(start);
       (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);
      }
```

In DFS we can't immediately add a node as "visited". We need to make sure we are only marking the node when it is popped.

Recap: Graph Traversals

DFS

(Iterative)

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

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 <u>queue</u>!

Using BFS for the s-t Connectivity Problem

s-t Connectivity Problem

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

BFS is a great building block – all we have to do is check each node to see if we've reached t!

- Note: we're not using any specific properties of BFS here, we just needed a traversal. DFS would also work.

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

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

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

Graph Traversals

Topological Sort

Shortest Path

Topological Sort

Given: a Directed Acyclic Graph (DAG) G, where we have an edge from u to v if u must happen before v.

Find: an ordering of the vertices so all edges go from left to right (all the dependency arrows are satisfied and the vertices can be processed left to right with no problems) .

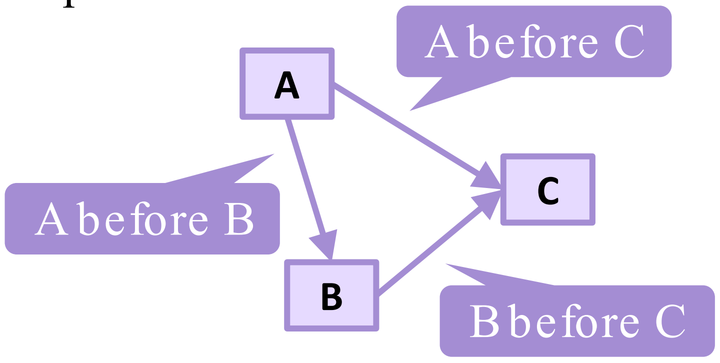
A DAG encodes a "dependency graph"

- An edge (u, v) means u must precede v
- A topological sort of a dependency graph gives an ordering that respects dependencies

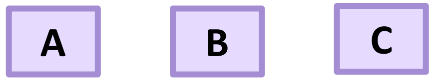
Applications:

- Compiling multiple Java files
- Multi-job Workflows

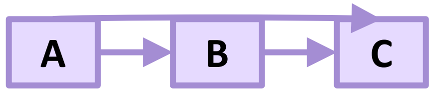
Input:



Topological Sort:

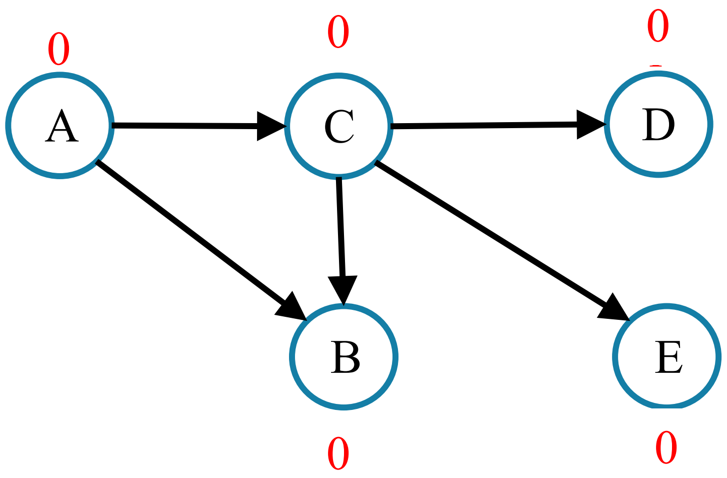


With original edges for reference:



Ordering a DAG

Does this graph have a topological ordering? If so find one.



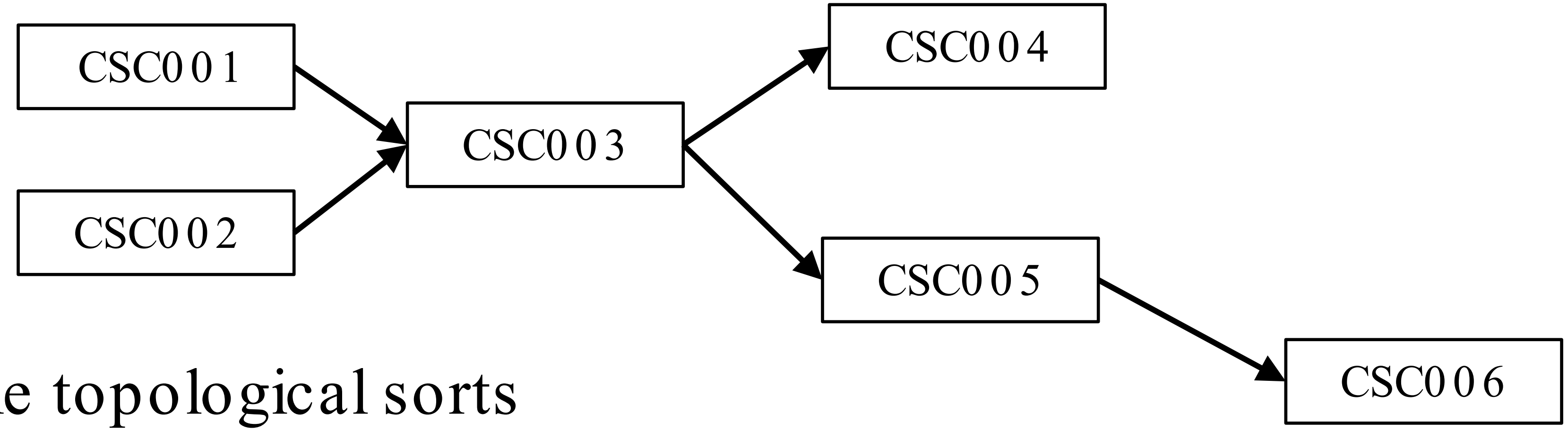
A C B D E

A C B E D

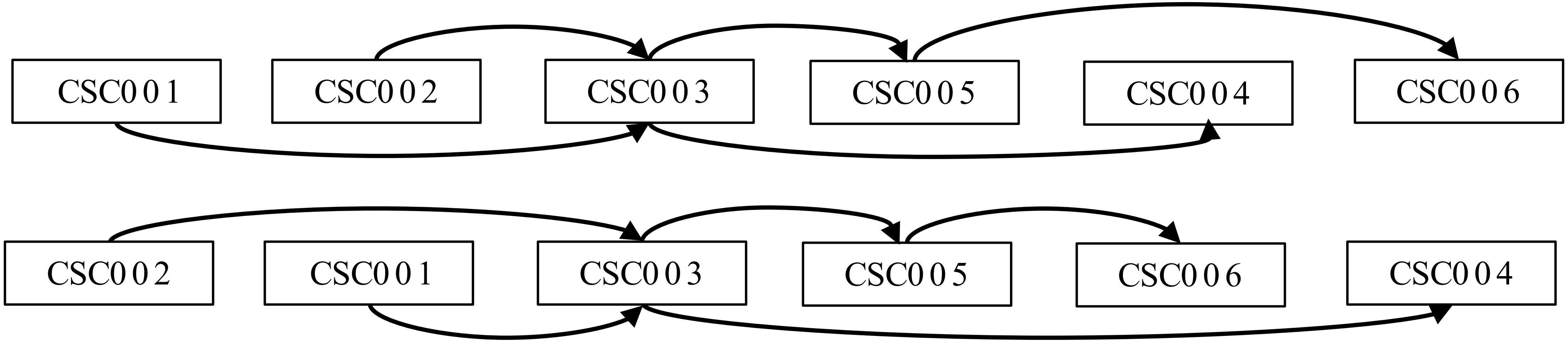
If a vertex doesn't have any edges going into it, we can add it to the ordering.
In general, topological sorts are not unique

Problem 1: Ordering Dependencies

Given a set of courses with prerequisites, find an order to take the courses in.

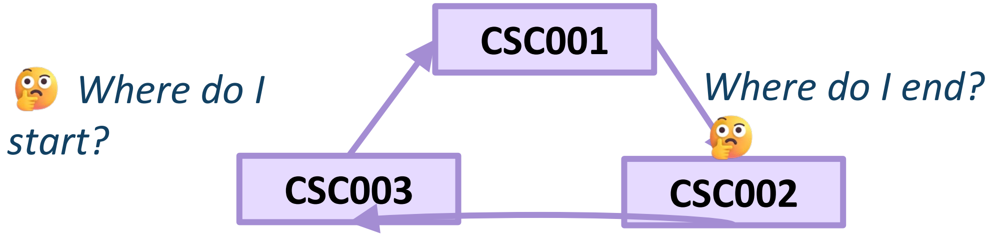


Two possible topological sorts



Can We Always Topo Sort a Graph?

Can you topologically sort this graph?



No

A graph has a topological ordering if it is a DAG

- But a DAG can have multiple orderings

DIRECTED ACYCLIC GRAPH

- A directed graph without any cycles
- Edges may or may not be weighted

Topological Sort Pseudocode

```
toposort(Graph graph) {
  Queue<Vertex> perimeter = new Queue<>();
  Set<Vertex> visited = new Set<>();
  Map<Vertex, Integer> indegree = countInDegree(graph);

  for (Vertex v : indegree.keySet()) {
    if(indegree.get(v) == 0) {
      perimeter.add(v);
      visited.add(v);
    }
  }
```

```
while (!perimeter.isEmpty()) {
  Vertex from = perimeter.remove();
  for (Edge edge : graph.edgesFrom(from)) {
    Vertex to = edge.to();
    if (!visited.contains(to)) {
      int inDeg = indegree.get(to);
      inDeg--;
      if (inDeg == 0) {
        perimeter.add(to);
        visited.add(to);
      }
      indegree.put(to, inDeg);
    }...
```

Start with BFS code (Queue to visit neighbors, List to mark visited)
Count the in-degree of each vertex
  queue up the 0 in-degree nodes to visit
Loop over Queue
  for each neighbor of a visited node reduce their in-degree count
  for nodes that hit 0, add them to Queue
Toposort is order nodes are "visited" (could create separate List to track order, could print out as you add to Set)

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?

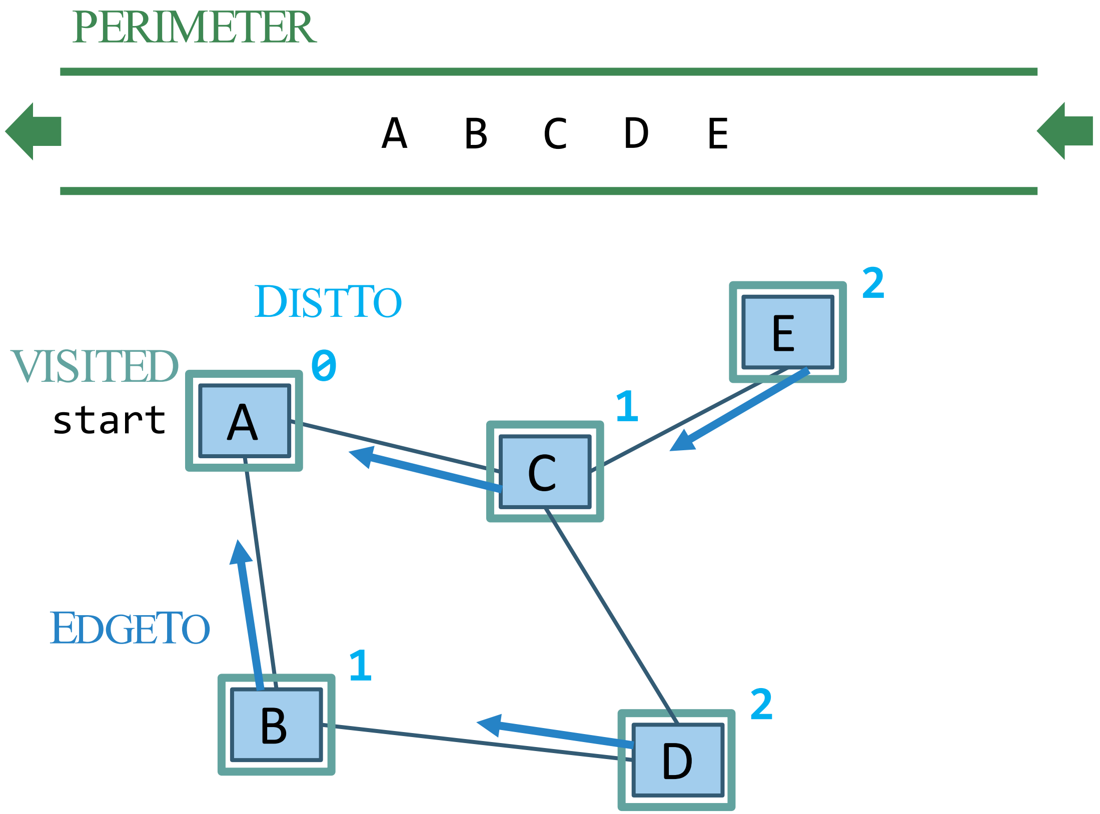
We need to keep track of how far each node is from the start, using BFS

```
  ...
  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



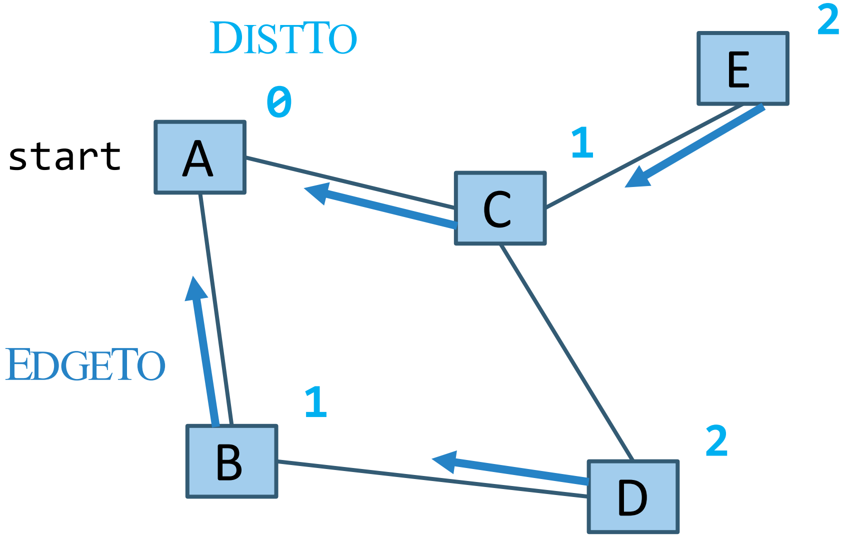
The `edgeTo` map stores backpointers: each vertex remembers what vertex was used to arrive at it!

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

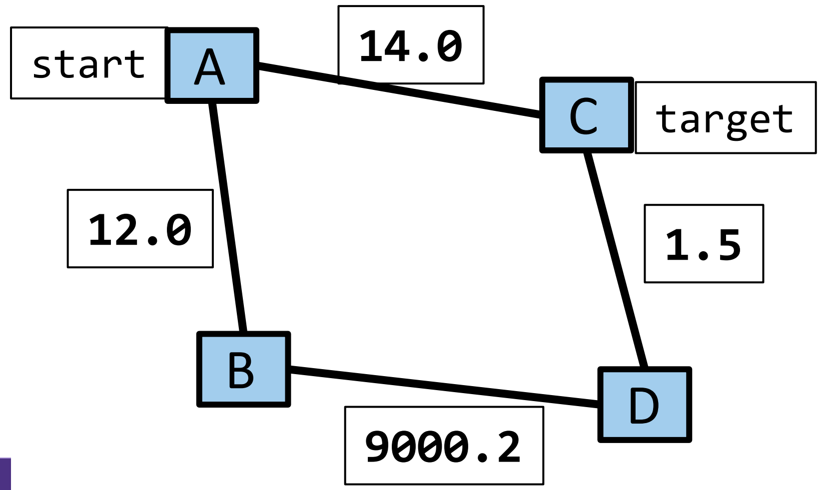
All our shortest path algorithms will have this property

- 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.



s-t Connectivity Problem	(Unweighted) Shortest Path Problem
Given source vertex s and a target vertex t, does there exist a path between s and t?	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 or DFS + check if we've hit t

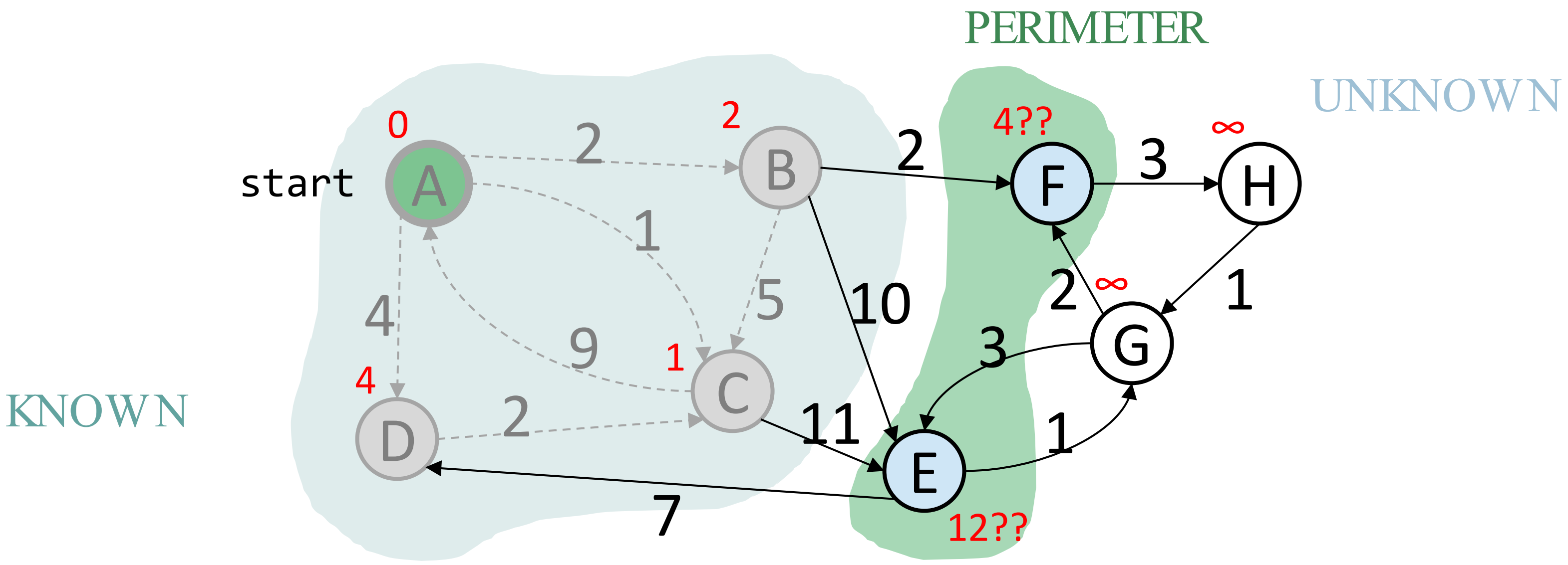
BFS + generate shortest path tree as we go

- 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"

Dijkstra's Algorithm

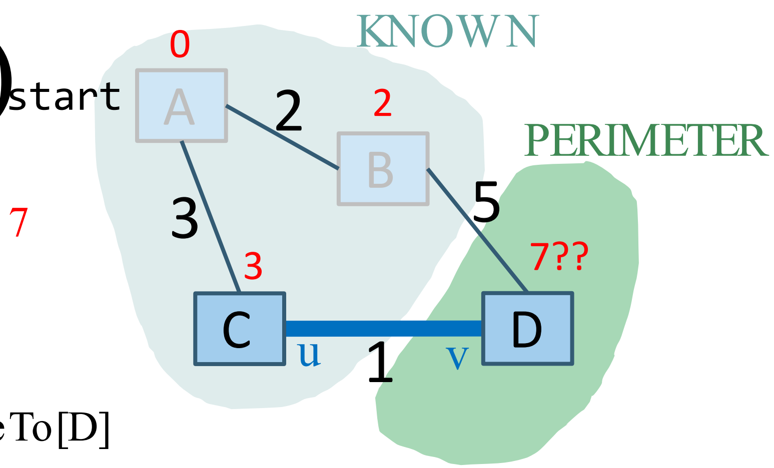
- Named after its inventor, Edsger Dijkstra (1930-2002)
  - Truly one of the "founders" of computer science
  - 1972 Turing Award
- The idea: reminiscent of BFS, but adapted to handle weights
  - Grow the set of nodes whose shortest distance has been computed
  - Nodes not in the set will have a "best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

O(|V|)

come back…

How do we find this??

O(1) for HashSet

O(|E|) worst case

O(1) for HashMap

```
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)
```

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

O(|V|)

O(|V|)

O(log|V|)

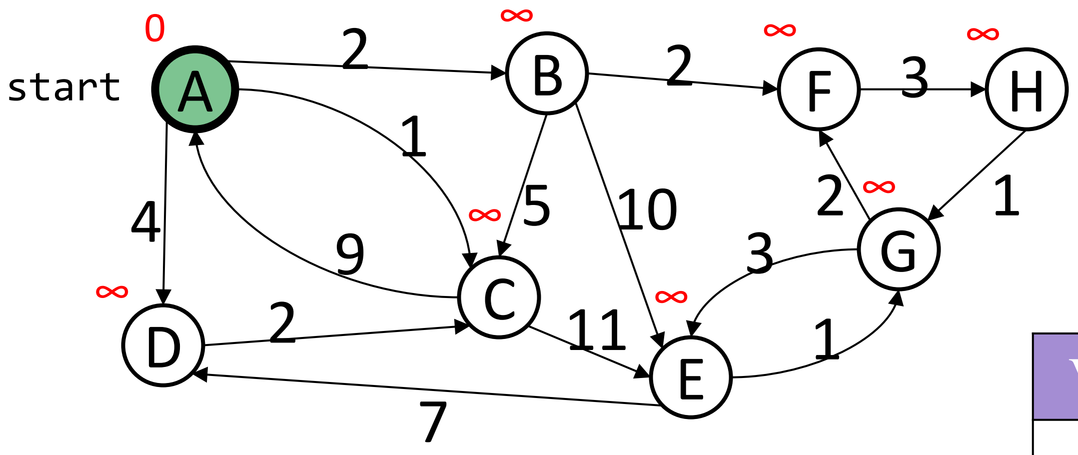
O(|E|)

O(log|V|)

```
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
```

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

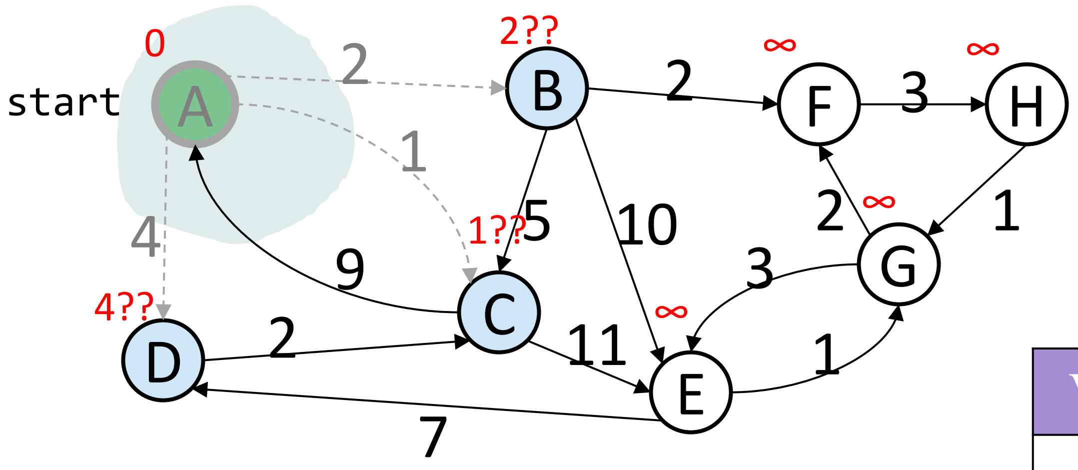
Dijkstra's Algorithm: Example #1



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

<u>Order Added to Known Set:</u>

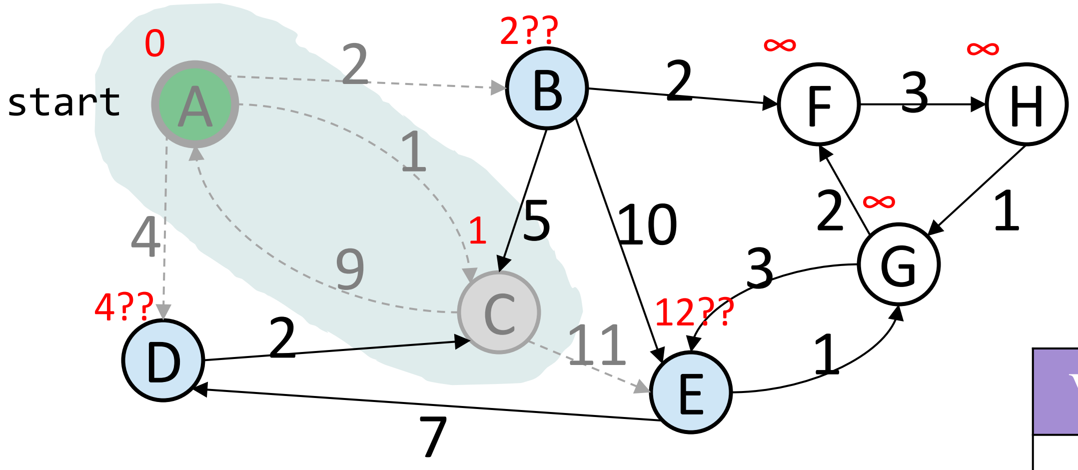
Dijkstra's Algorithm: Example #1



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

<u>Order Added to Known Set:</u>
A

Dijkstra's Algorithm: Example #1

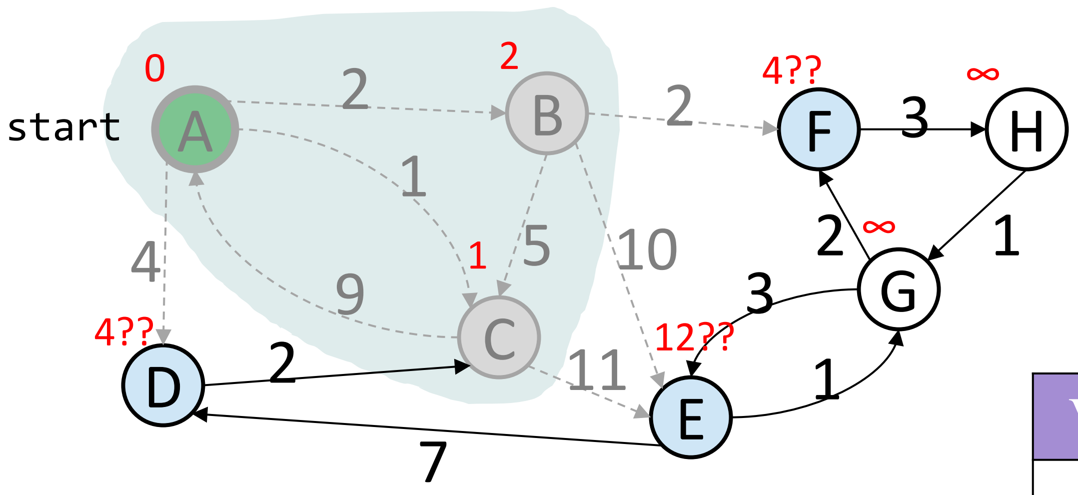


Vertex	Known?	distTo	edgeTo
A	Y	0	/
B		≤ 2	A
C	Y	1	A
D		≤ 4	A
E		≤ 12	C
F		∞	
G		∞	
H		∞	

<u>Order Added to Known Set:</u>

A, C

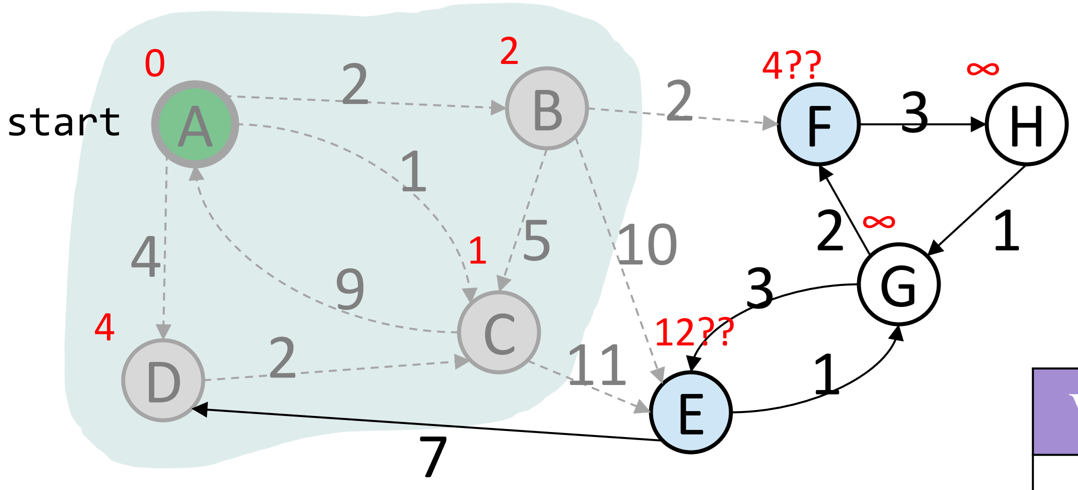
Dijkstra’s Algorithm: Example #1



Order Added to Known Set:
A, C, B

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		∞	

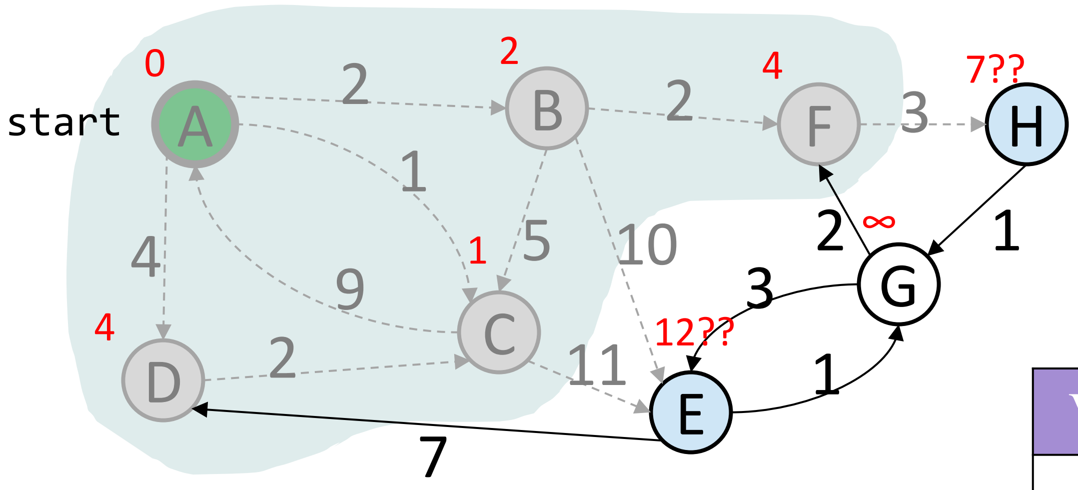
Dijkstra's Algorithm: Example #1



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

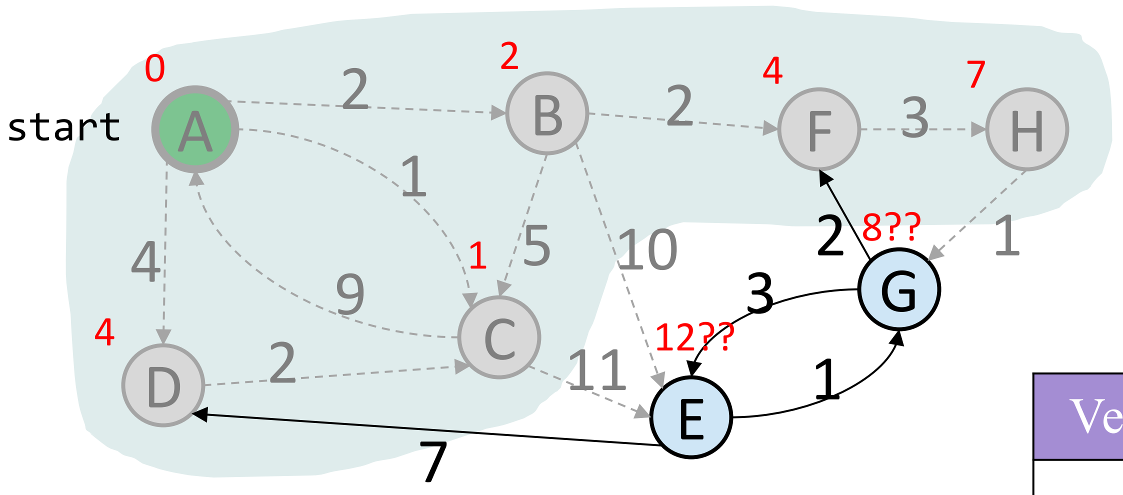
Dijkstra's Algorithm: Example #1



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

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

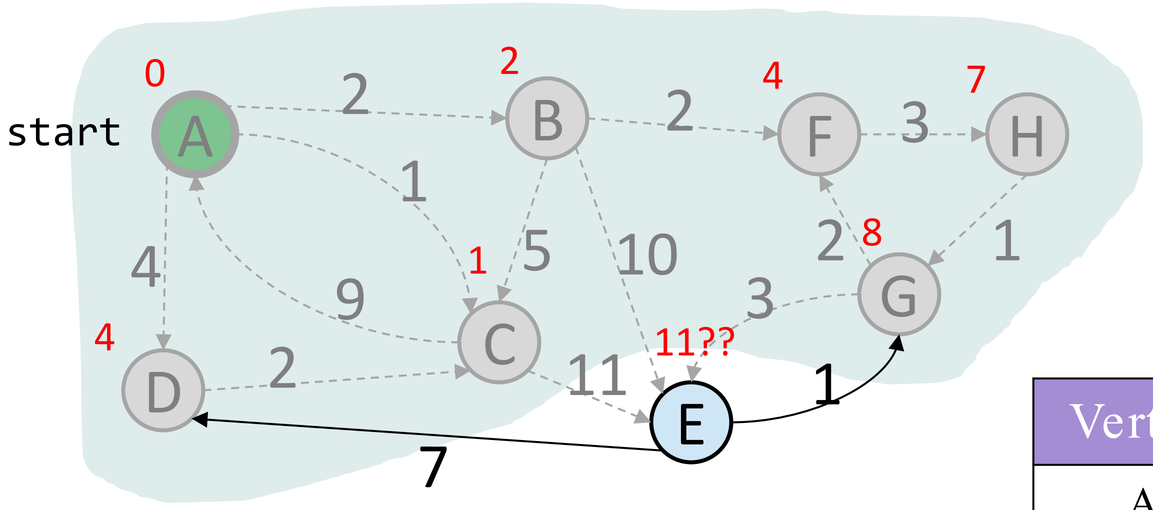
Dijkstra's Algorithm: Example #1



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

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

Dijkstra’s Algorithm: Example #1

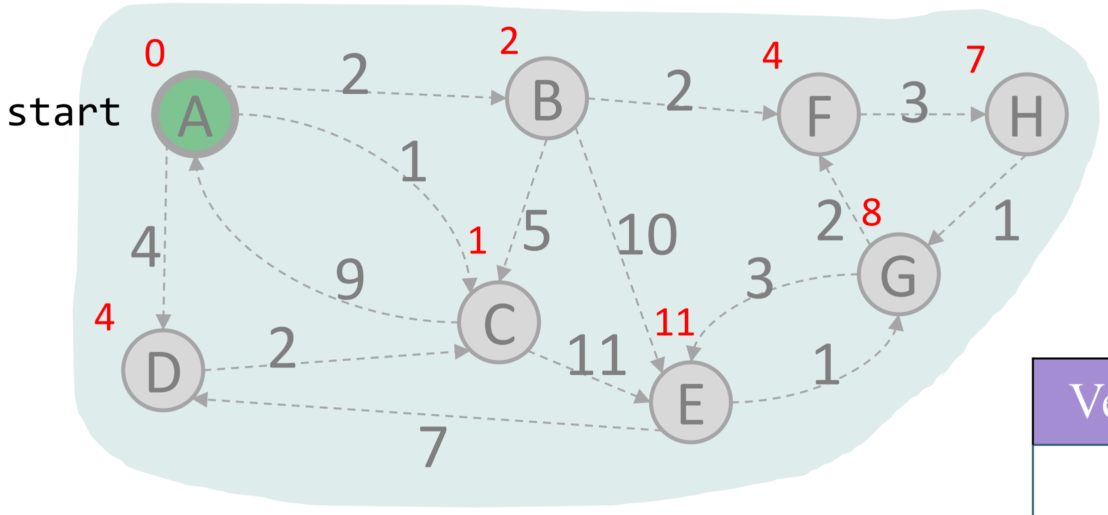


Vertex	Known?	distTo	edgeTo
A	Y	0	/
B	Y	2	A
C	Y	1	A
D	Y	4	A
E		≤ 11	G
F	Y	4	B
G	Y	8	H
H	Y	7	F

<u>Order Added to Known Set:</u>

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

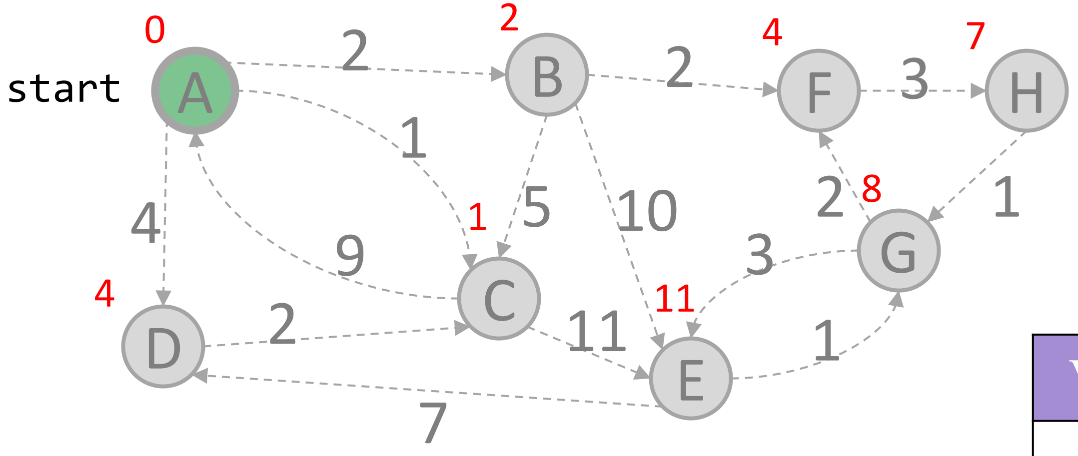
Dijkstra's Algorithm: Example #1



Vertex	Known?	distTo	edgeTo
A	Y	0	/
B	Y	2	A
C	Y	1	A
D	Y	4	A
E	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

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

<u>Order Added to Known Set:</u>

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

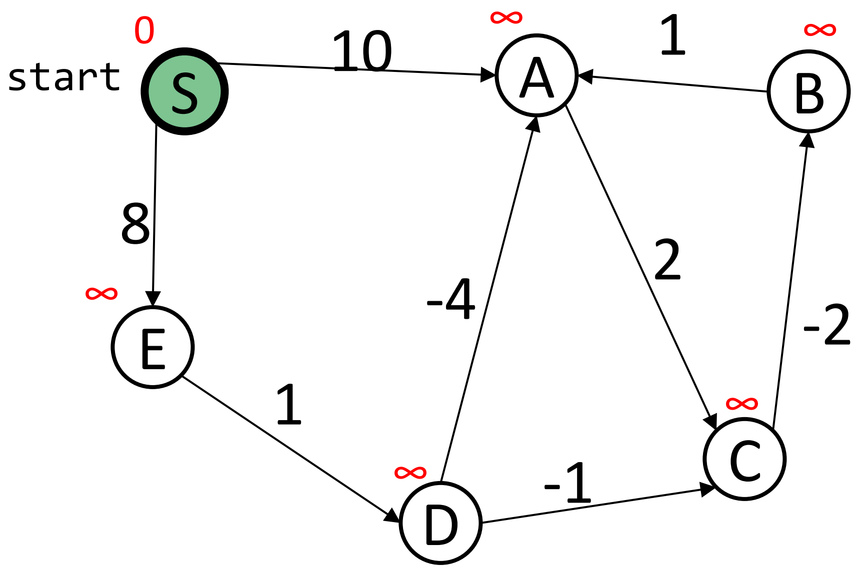
Bellman-Ford Shortest Path

- 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

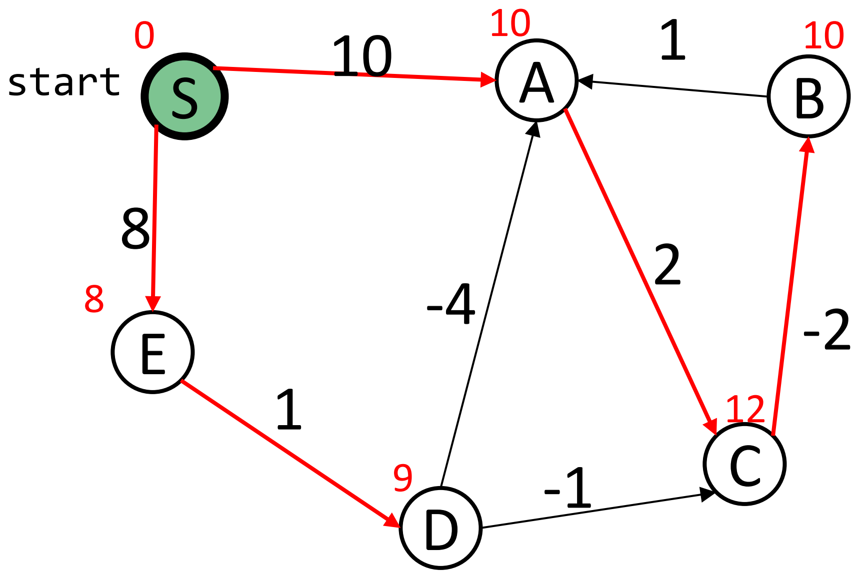
Bellman-Ford Example



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

[Example Walk Through Video (5min)]

Bellman-Ford Example

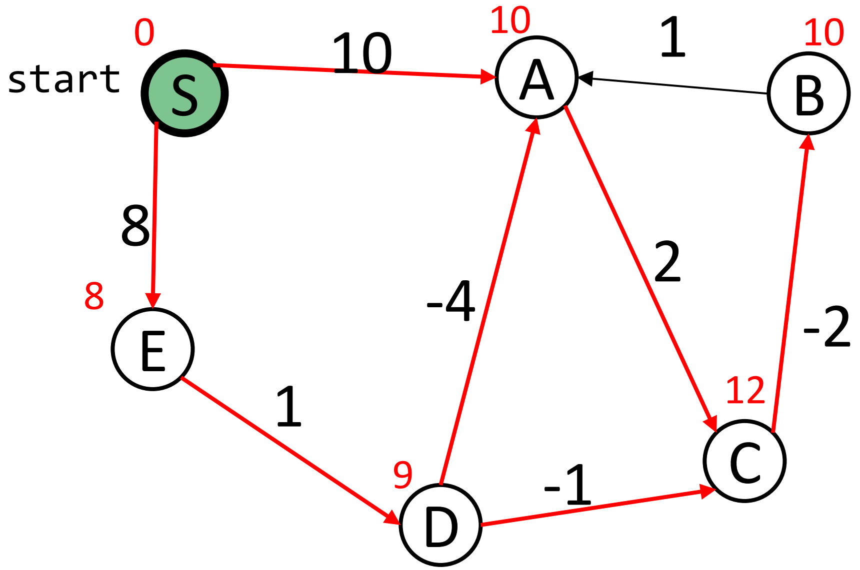


Iteration 1- for each Vertex's outgoing edge, does that give us a shorter way to get to a new vertex?

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



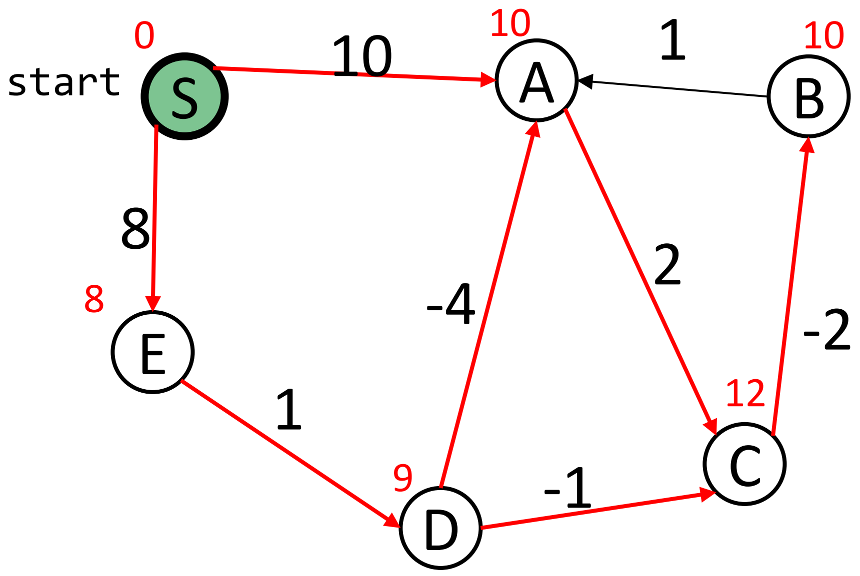
Iteration 2 - re-examining outgoing edges, can we improve the distance to any given Vertex?

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!

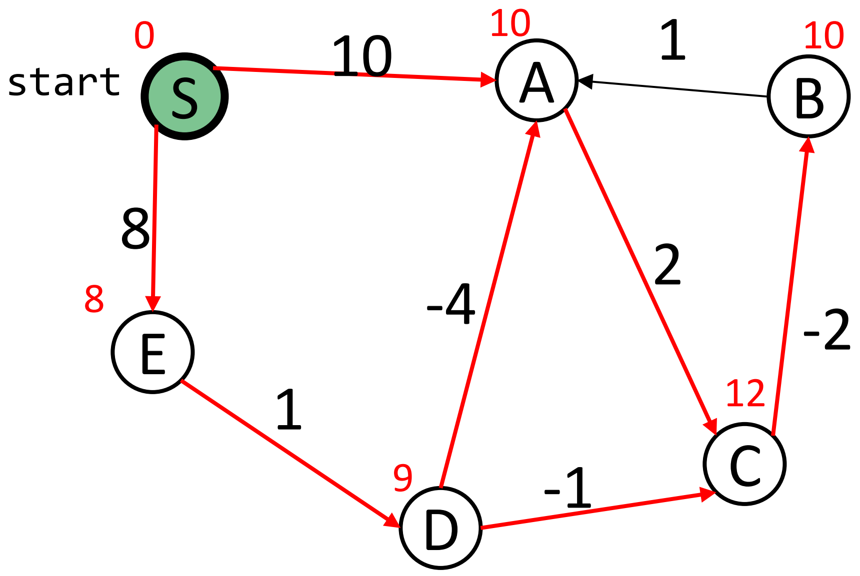
Vertex	distTo	edgeTo
S	0	-
A	5	D
B	~~10~~ 5	C
C	~~8~~ 7	A
D	9	E
E	8	A

* 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)