Lecture 17: Graph Traversals

CSE 373: Data Structures and Algorithms

Warm Up

Slido Event #1608638
https://app.sli.do/event/bqTzak53Lu9msB67Q72djQ

How would you transform the following scenario into a graph?

You are creating a graph representing a brand-new social media network. Each profile has both the option to “friend” another user or to “follow” another user. When ”friend” is selected the other profile is asked for permission, and if given the two profiles will link to one another. If “follow” is selected then no permission is asked, but the recipient will not connect to the follower. Answer the following questions about the graph design:

What are the vertices? **Profiles**

What are the edges? **Follows and friendships**

Undirected or directed edges? **Directed**

Weighted or unweighted edges? **Unweighted**

Warm Up

How would you transform the following scenario into a graph?

You are going to a music festival and are trying to plan the perfect schedule so you can catch all of your favorite artists. You know what time each act starts and how long it will be, you can only get into an act if you get there before it starts and you can only leave an act after it ends.

Answer the following questions about the graph design:

What are the vertices? **Artists (aka each act/performance)**

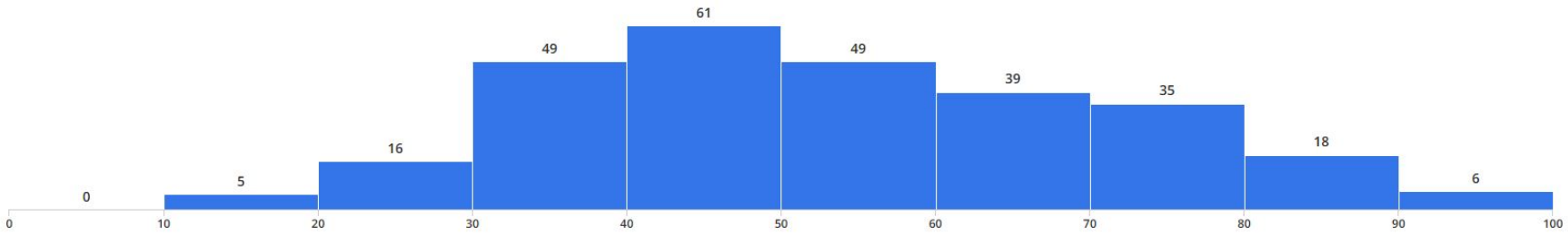
What are the edges? **Possible choices for the next act**

Undirected or directed? **Directed**

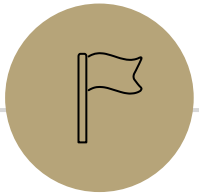
Weighted or unweighted? **Unweighted**

Announcements

- Midterm Grades released!
  - You will see the points you got per question but not the specific rubric items applied
  - Instead you can see the rubric we used and some hints for how to get full credit in this document so you can assess your own answers
  - The resubmission will be entirely online through Gradescope
    - You have the choice per question to submit a new answer, if you do not want to resubmit simply enter "skip"
    - This is a "no risk" resubmission, we will take the greater of your two scores per question to make your resubmission grade
    - Your final midterm grade will be the average of your two scores
    - Please put all questions about the resubmission as a private EdBoard post
    - You can use the internet, your notes, conceptual conversations with one another and with the course staff to formulate your new answers
    - We will open both the paper and the digital midterm submissions for regrade requests after all grades have been released



Minimum	Median	Maximum	Mean	Std Dev
12.0	51.5	96.0	53.3	17.93



Graph Traversals

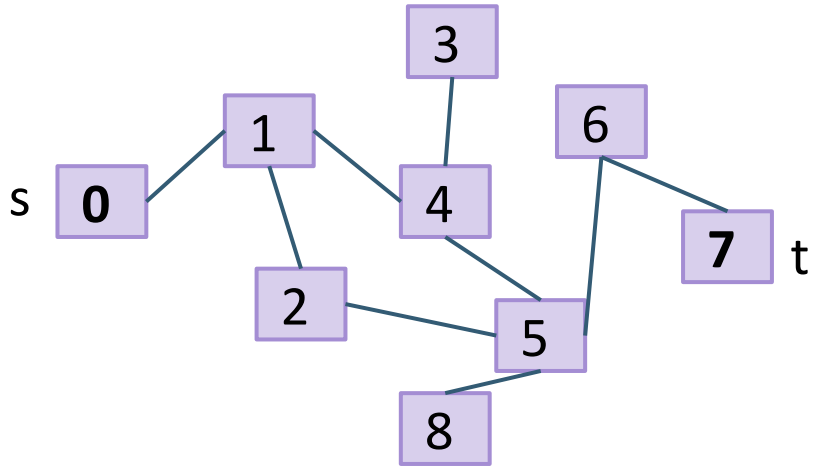
Topological Sort

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**?

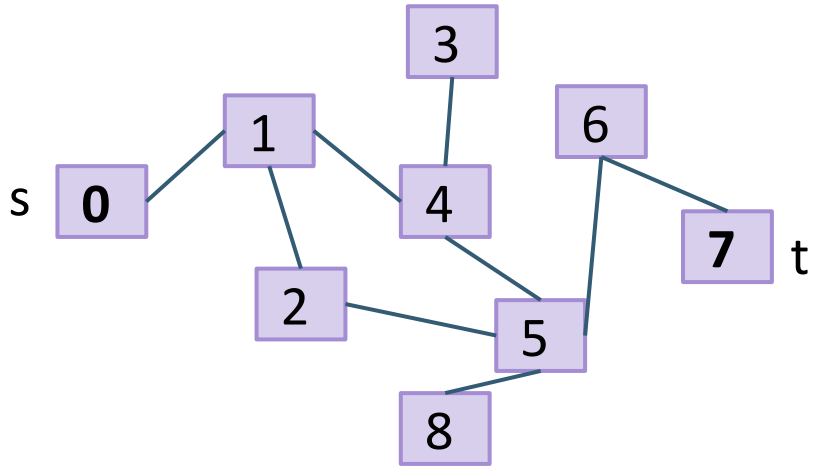


Try to come up with an algorithm for `connected(s, t)`

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

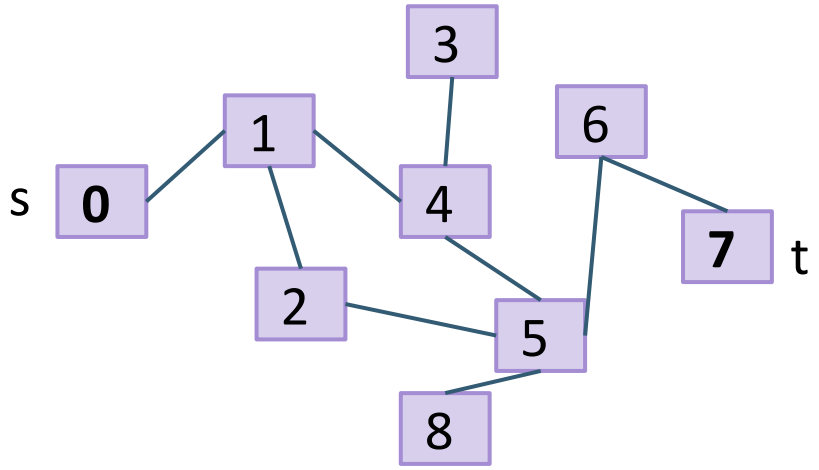
s-t Connectivity Problem: Proposed Solution

```
connected(Vertex s, Vertex t) {
  if (s == t) {
    return true;
  } else {
    for (Vertex n : s.neighbors) {
      if (connected(n, t)) {
        return true;
      }
    }
    return false;
  }
}
```



What's wrong with this proposal?

```
connected(Vertex s, Vertex t) {
  if (s == t) {
    return true;
  } else {
    for (Vertex n : s.neighbors) {
      if (connected(n, t)) {
        return true;
      }
    }
    return false;
  }
}
```

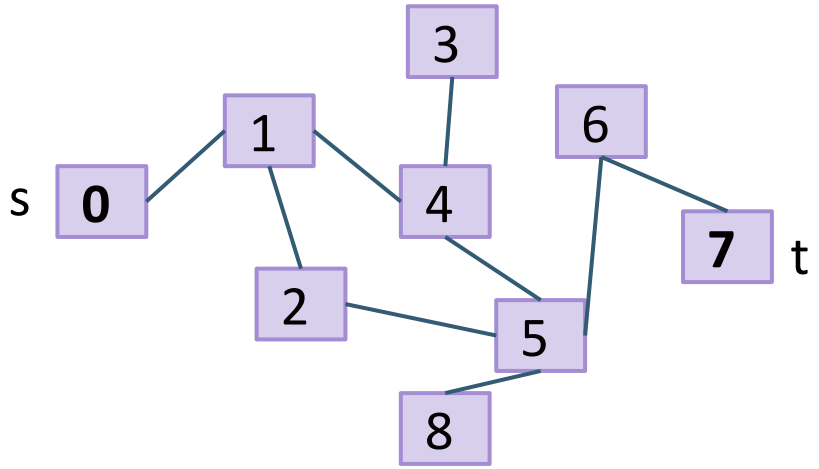


Does 0 == 7? No; if(connected(1, 7) return true;
Does 1 == 7? No; if(connected(0, 7) return true;
Does 0 == 7?

s-t Connectivity Problem: Better Solution

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!

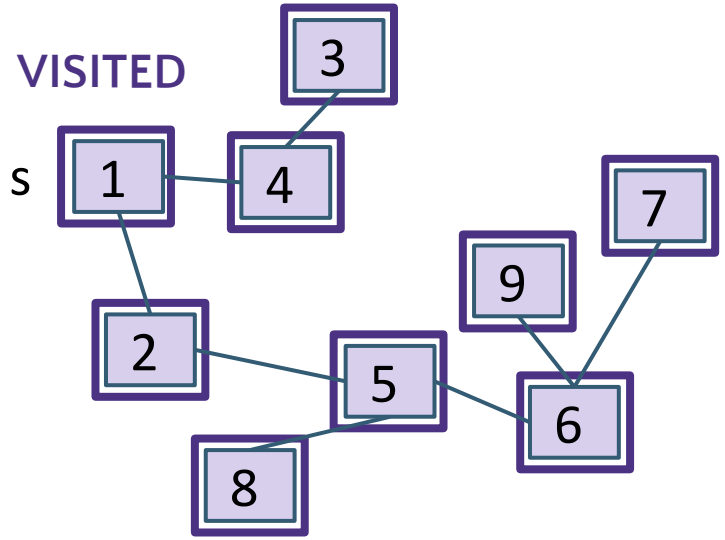
Recursive Depth-First Search (DFS)

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;
  }
}
```

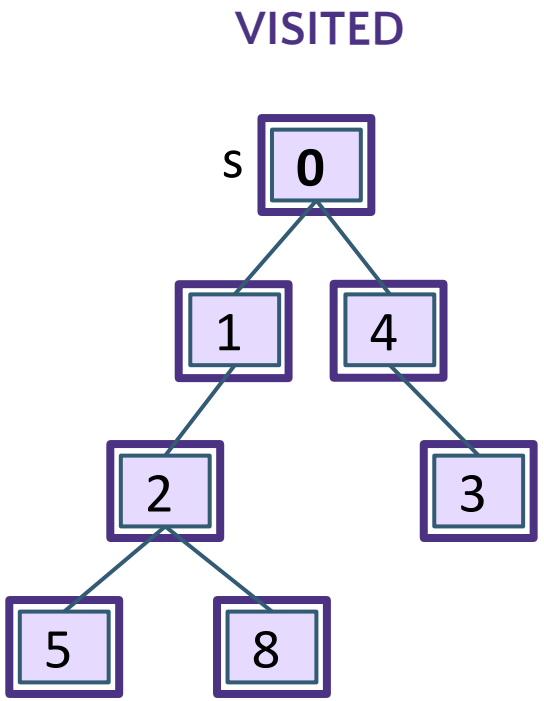
- It will explore one option "all the way down" before coming back to try other options
  - Many possible orderings: {0, 1, 2, 5, 6, 9, 7, 8, 4, 3} or {0, 1, 4, 3, 2, 5, 8, 6, 7, 9} both possible
- We call this approach a **depth-first search (DFS)**



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;
  }
}
```



CSE 143 Review traversing a binary tree depth-first has 3 options:

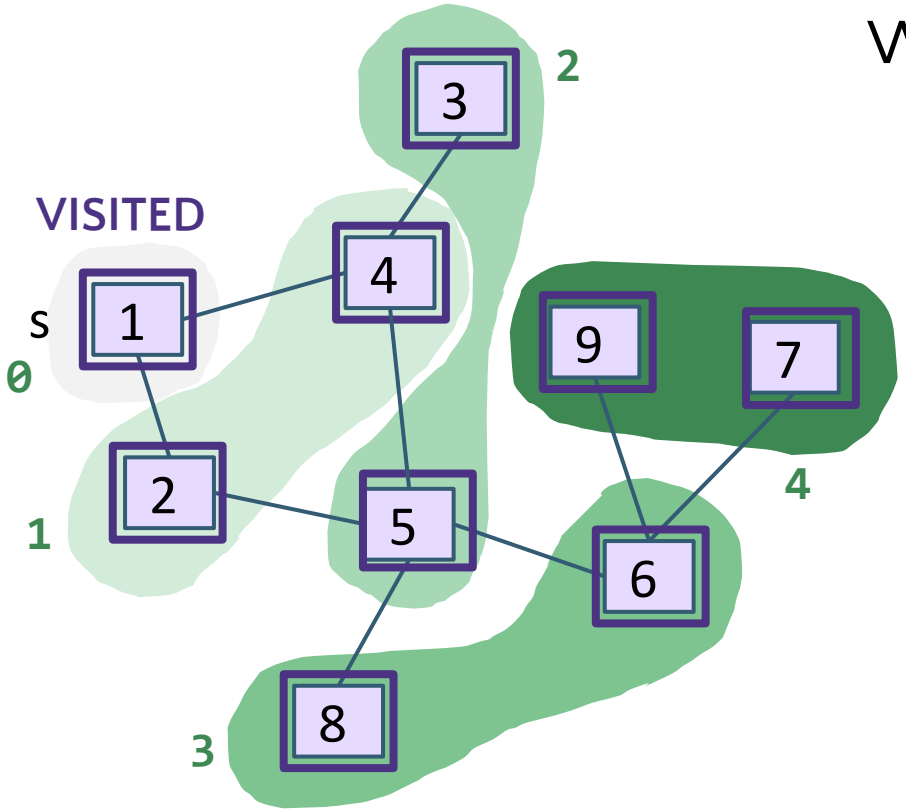
- Pre-order: visit node before its children
- In-order: visit node between its children
- Post-order: visit node after its children

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

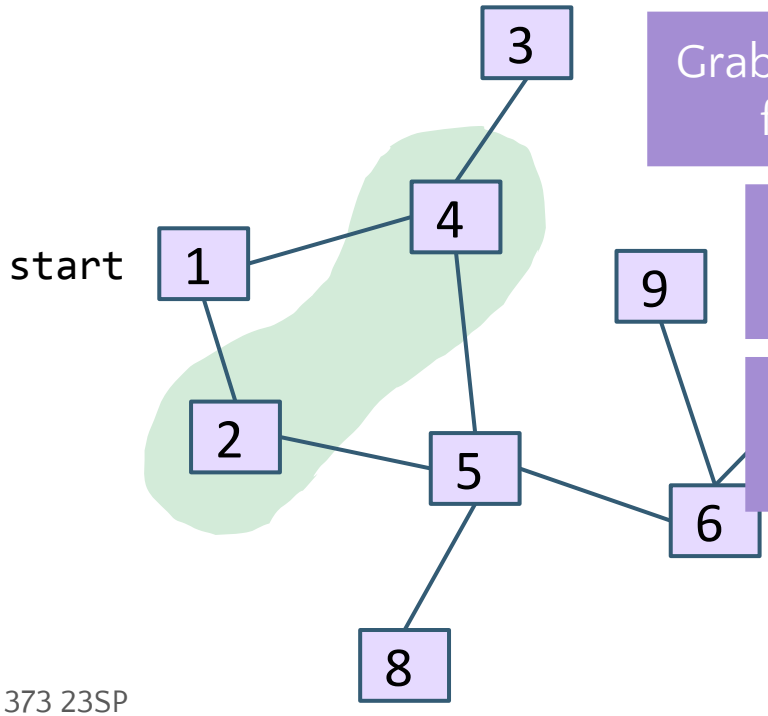
Our extra data structure! Will keep track of "**outer edge**" of nodes still to explore

Kick off the algorithm by adding start to perimeter

Grab one element at a time from the perimeter

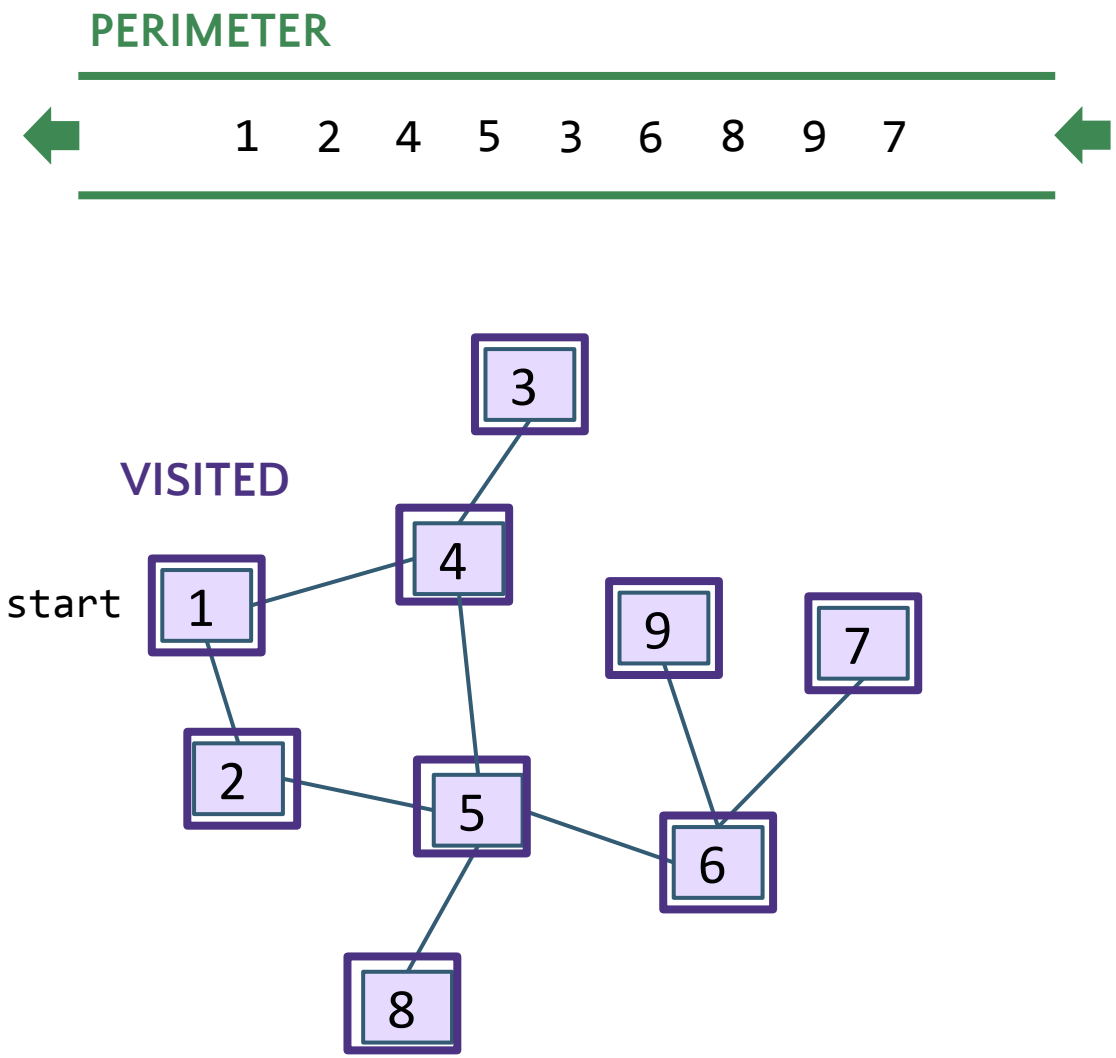
Look at all that element's children

Add new ones to perimeter!



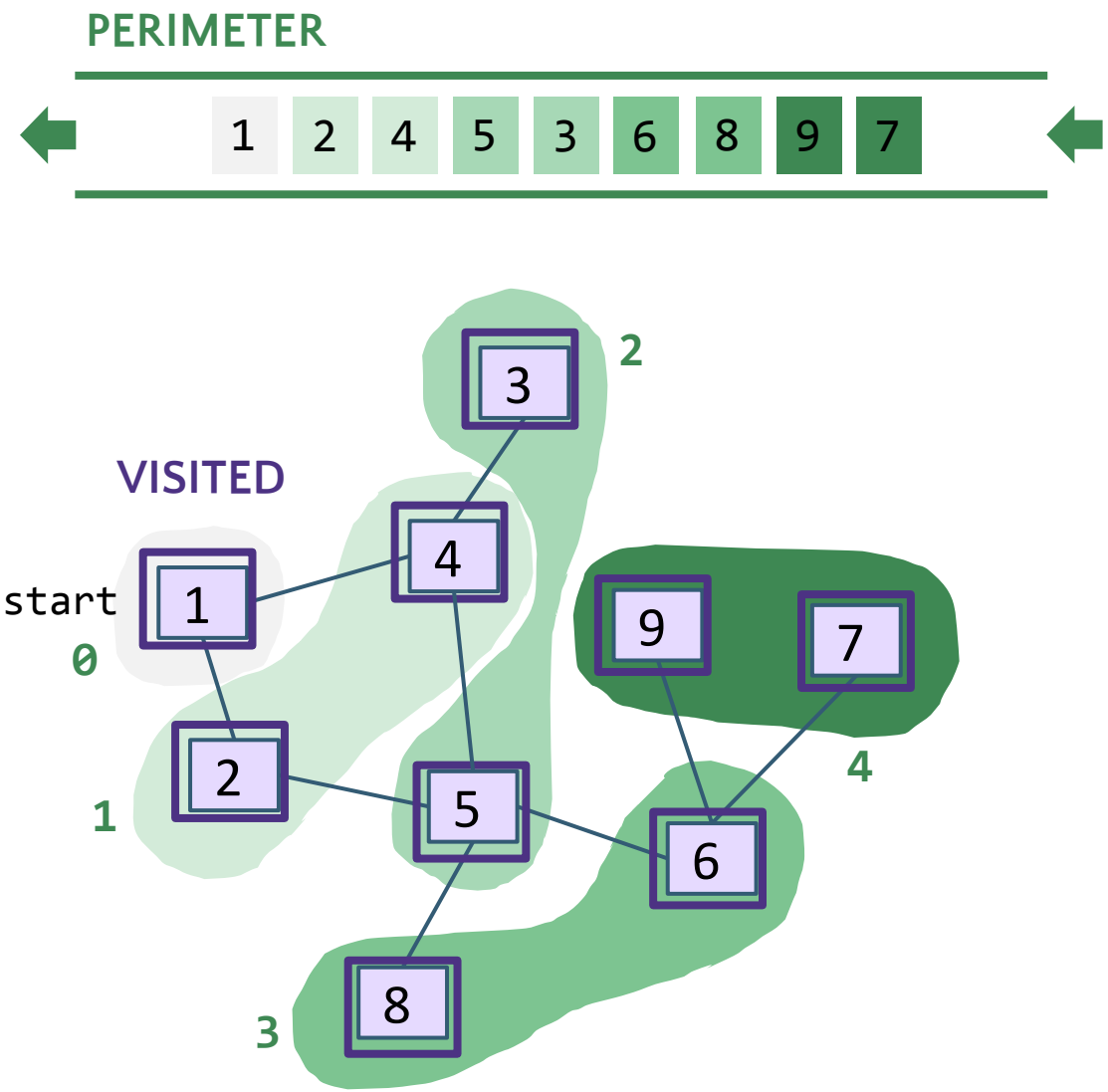

```
bfs(Graph graph, Vertex start) {
   Queue<Vertex> perimeter = new Queue<>();
   Set<Vertex> visited = new Set<>();

  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?



- Properties of a queue exactly what gives us this incredibly cool behavior
- As long as we explore an entire layer before moving on (and we will, with a queue) the next layer will be fully built up and waiting for us by the time we finish!
- Keep going until `perimeter` is empty

```
  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's Evil Twin: DFS!

```
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<>();

  dd(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

We've seen two approaches for ordering a graph traversal

BFS and DFS are just techniques for iterating! (think: for loop over an array)

- Need to add code that actually processes something to solve a problem
- A *lot* of interview problems on graphs can be solved with **modifications on top of BFS or DFS**! Very worth being comfortable with the pseudocode ☺️

Let's Practice Now!

DFS

(Iterative)

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

DFS

(Recursive)

← Be careful using this – on huge graphs, might overflow the call stack

Using BFS for the s-t Connectivity Problem

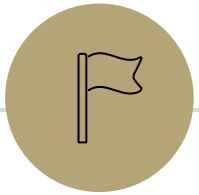
s-t Connectivity Problem

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

A **topological sort** of a directed acyclic graph G is an ordering of the nodes, where for every edge in the graph, the origin appears before the destination in the ordering

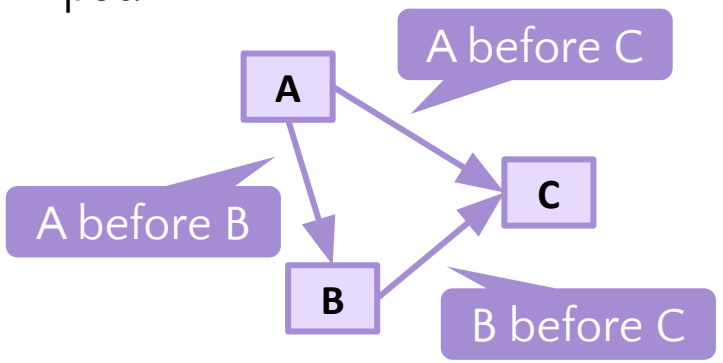
Intuition: a "dependency graph"

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

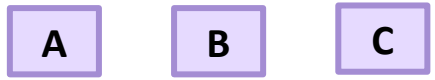
Applications:

- Graduating
- Compiling multiple Java files
- Multi-job Workflows

Input:



Topological Sort:



With original edges for reference:

Ordering Dependencies

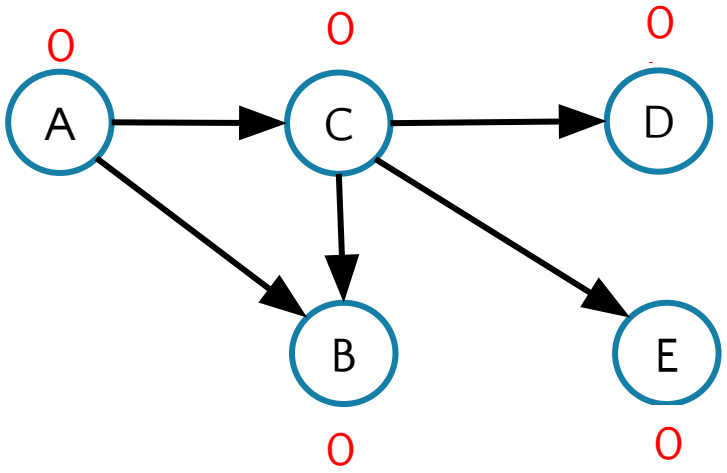
- Given a **Directed Acyclic Graph(DAG)** G, where we have an edge from u to v if u must happen before v.
- We can only do things one at a time, can we find an order that **respects dependencies**?

Topological Sort (aka Topological Ordering)

Given: a directed graph G
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) .

Ordering a DAG

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

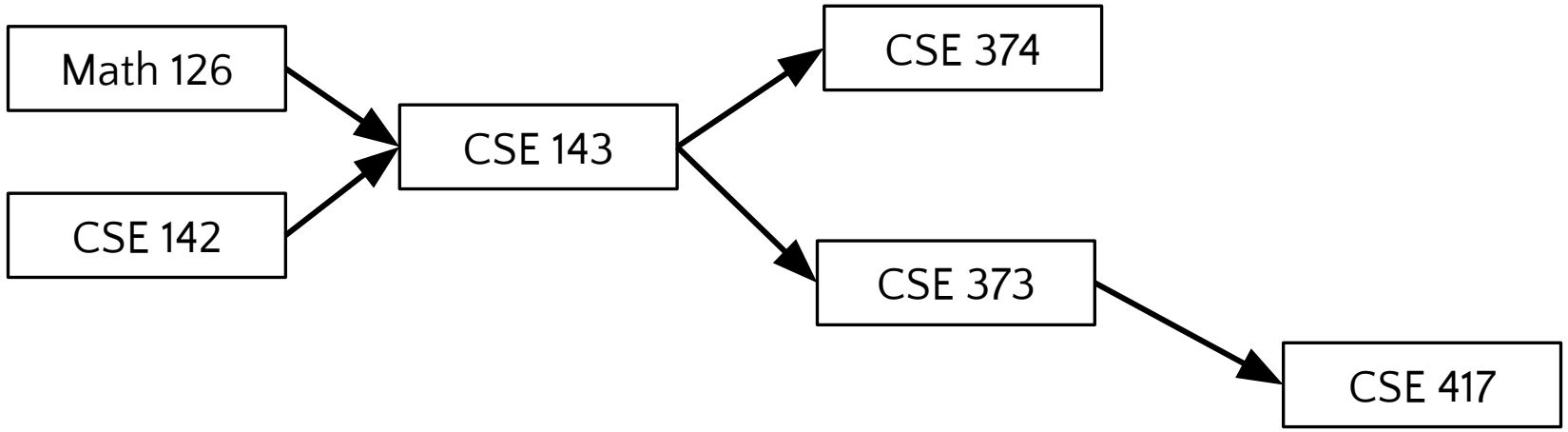


A C B D E

If a vertex doesn't have any edges going into it, we can add it to the ordering.
More generally, if the only incoming edges are from vertices already in the ordering, it's safe to add.

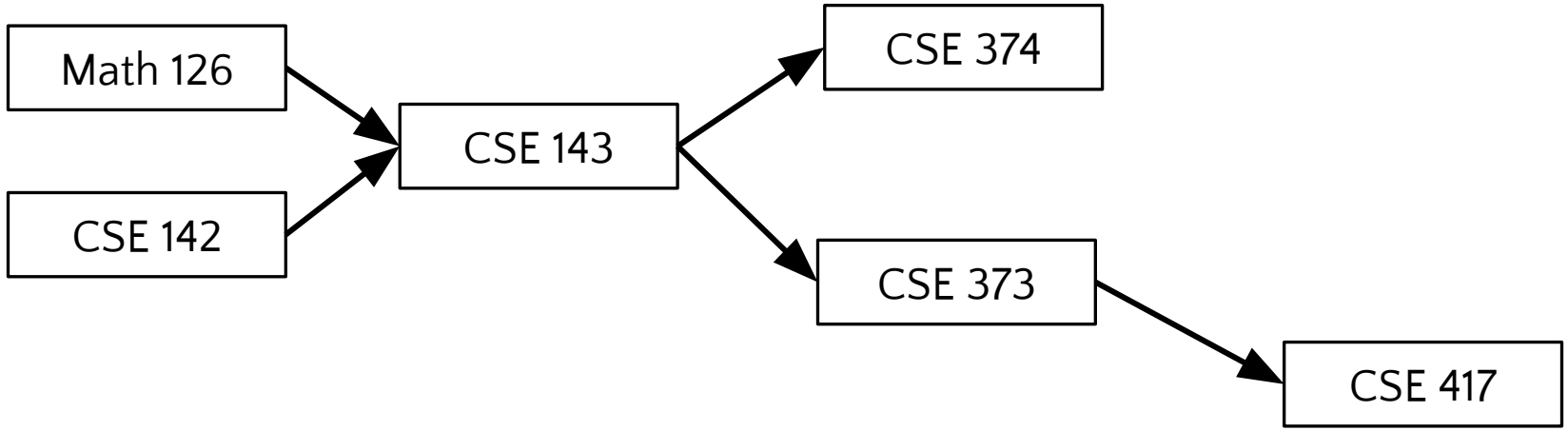
Problem 1: Ordering Dependencies

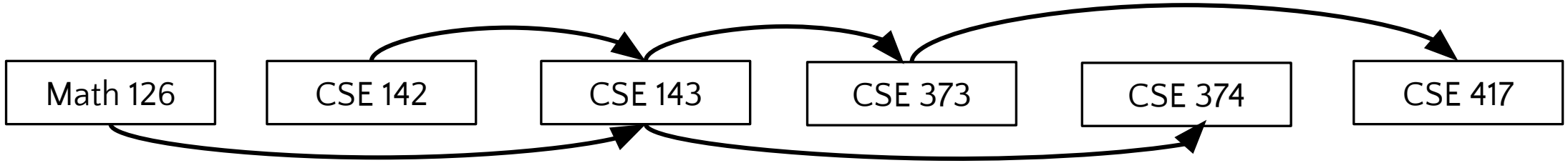
Today's (first) problem: Given a bunch of courses with prerequisites, find an order to take the courses in.



Topological Ordering

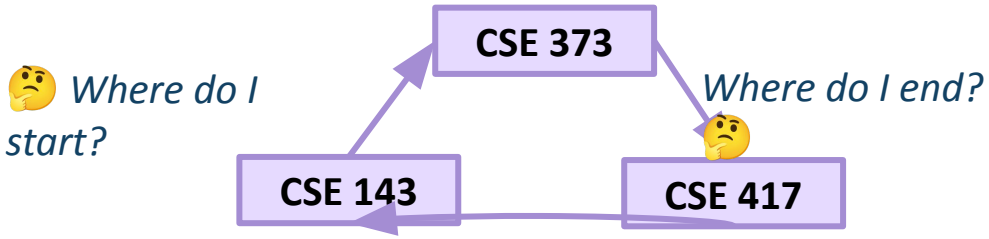
A course prerequisite chart and a possible topological ordering.





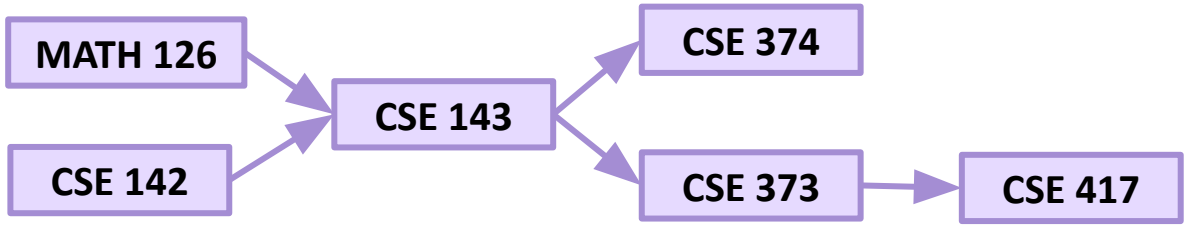
Can We Always Topo Sort a Graph?

Can you topologically sort this graph?



No 😭

What's the difference between this graph and our first graph?



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

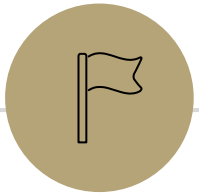
```
toposort(Graph graph) {
  Queue<Vertex> perimeter = new Queue<>();
  Set<Vertex> visited = new Set<>();
  Map<Vertex, Integer> indegree = countInDegree(graph);

  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?

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.

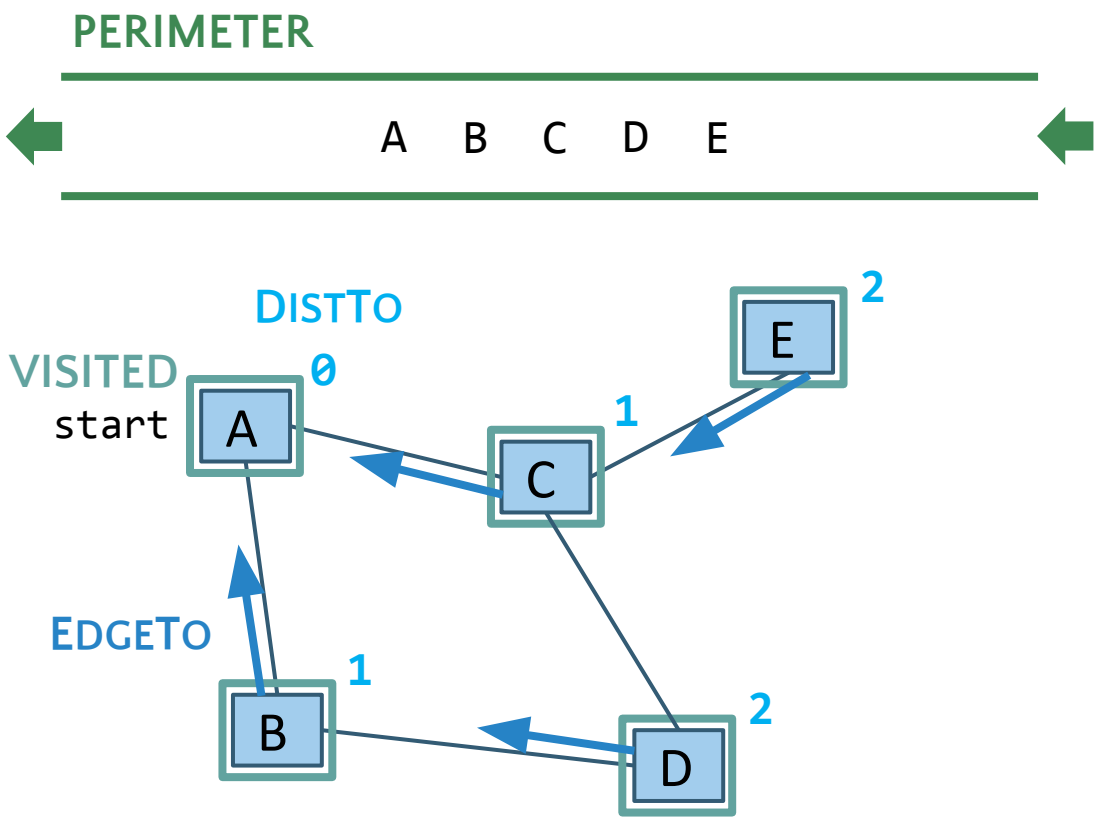
- Sounds like a job for?
  - BFS!

```
...
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



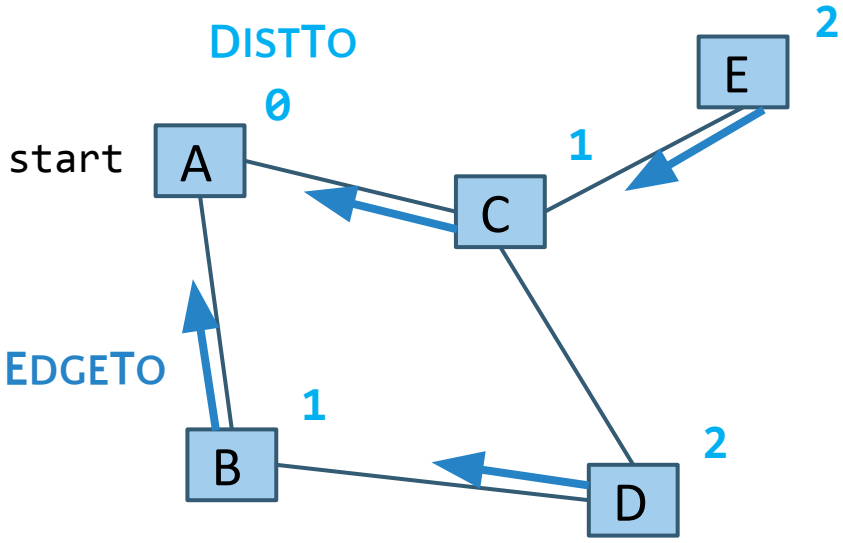
The **edgeTo** map stores **backpointers**: each vertex remembers what vertex was used to arrive at it!

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.

EASY

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

MEDIUM

(Unweighted) Shortest Path Problem

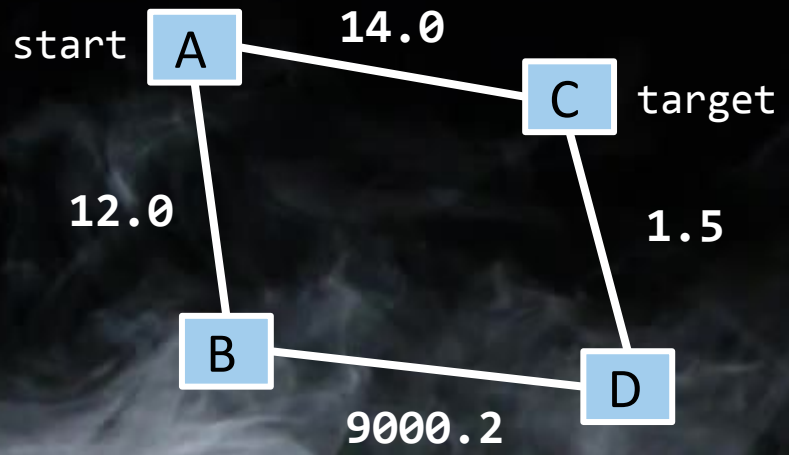
Given source vertex **s** and a target vertex **t**, how long is the shortest path from **s** to **t**? What edges make up that path?

BFS + generate shortest path tree as we go

What about the Shortest Path Problem on a *weighted* graph?

Next Stop Weighted Shortest Paths

HARDER (FOR NOW)

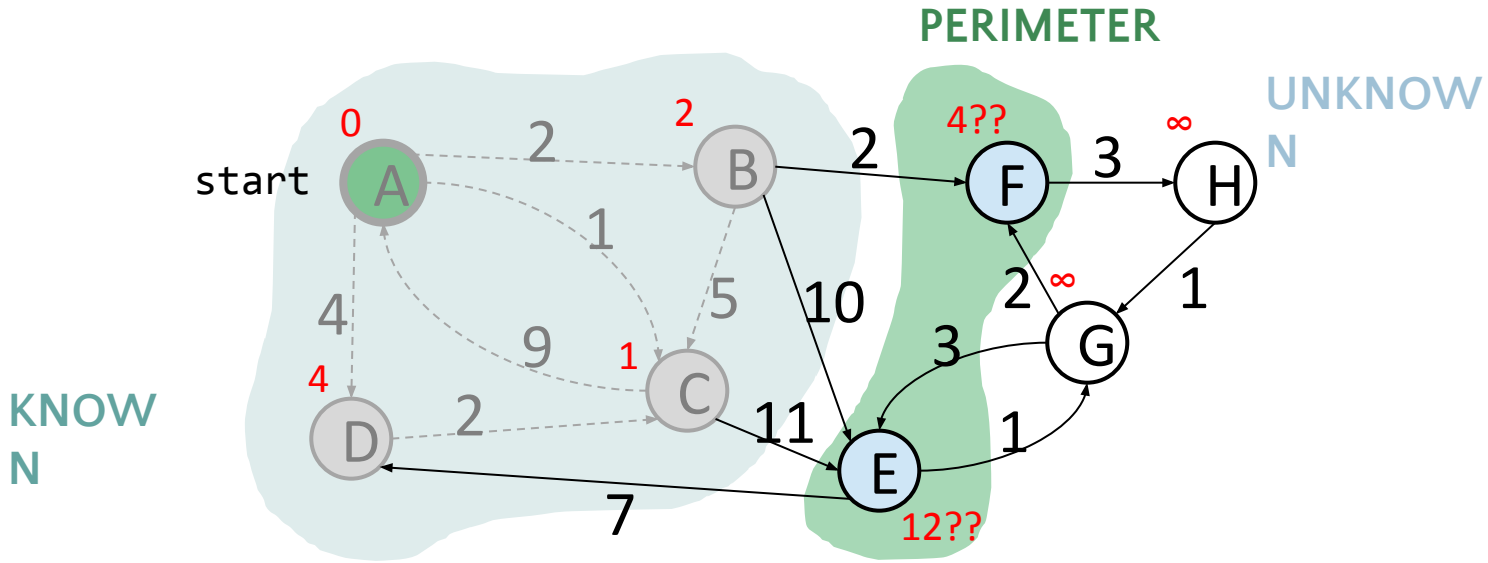


- Suppose we want to find shortest path from A to C, using weight of each edge as "distance"
- Is BFS going to give us the right result here?

Dijkstra's Algorithm

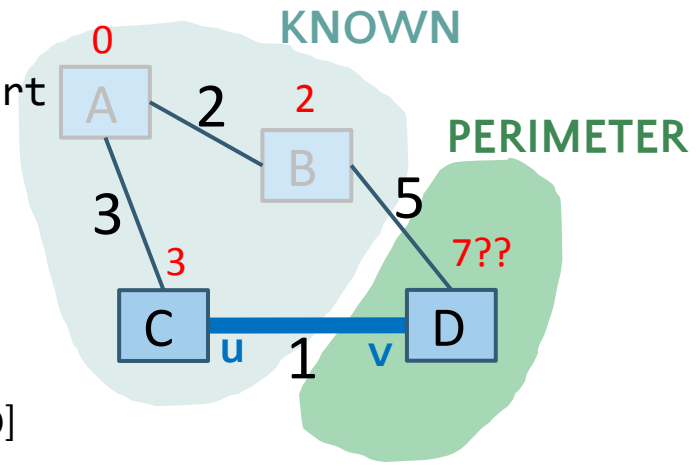
- Named after its inventor, Edsger Dijkstra (1930–2002)
  - Truly one of the "founders" of computer science
  - 1972 Turing Award
  - This algorithm is just *one* of his many contributions!
  - Example quote: "Computer science is no more about computers than astronomy is about telescopes"
- The idea: 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)

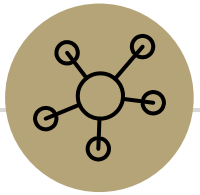
While a vertex is not known, another shorter path might be found

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

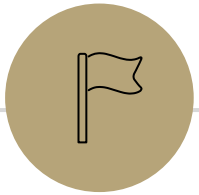
Going through closest vertices first lets us confidently say no shorter path will be found once known

- 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)
```



Questions?



That’s all!