Lecture 17: BFS, DFS, Dijkstra

CSE 373: Data Structures and Algorithms

Warm Up

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

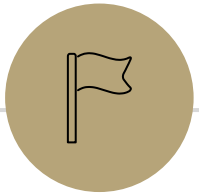
What are the edges? Follows and friendships

Undirected or directed? Directed

Weighted or unweighted? Unweighted

Administrivia

- Q3.1 does not converge to a single value
- Design decisions are left intentionally ambiguous… annoying I know
  - as long as you justify your answer we can give you credit



Introduction to Graphs

Graph: Formal Definition

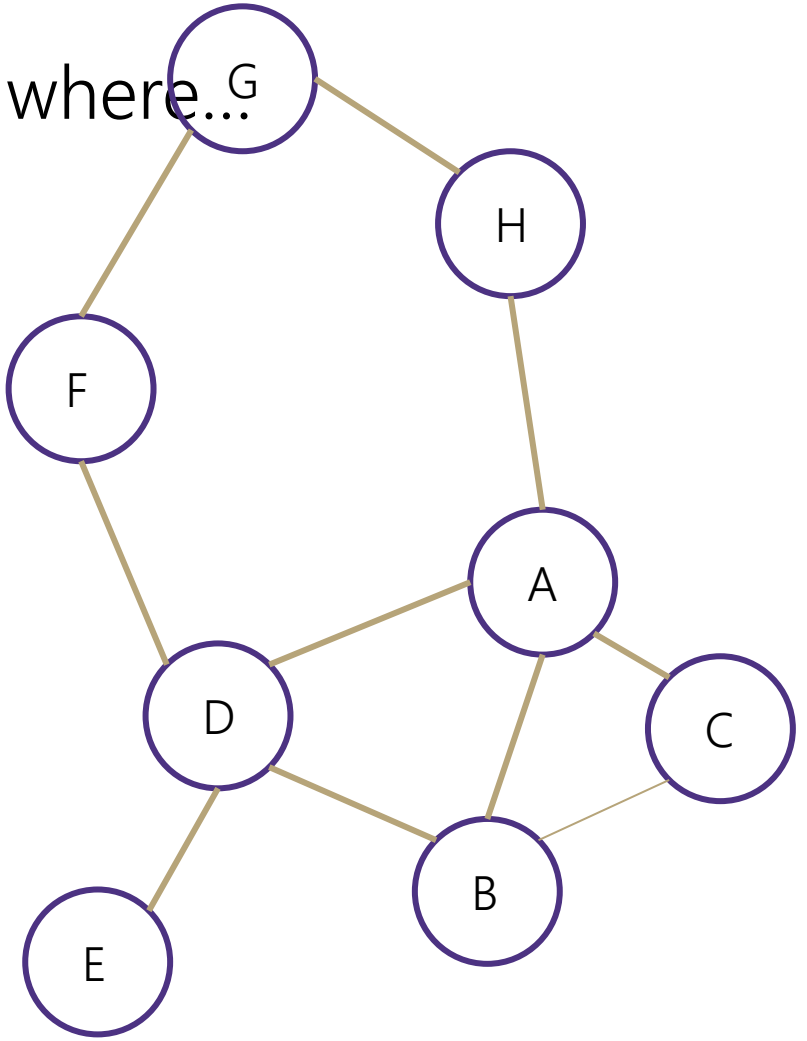
A graph is defined by a pair of sets G = (V, E) where…

- 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)}



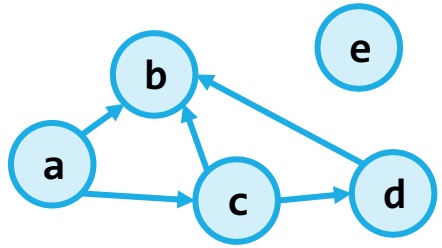
Graph Glossary

Graph: a category of data structures consisting of a set of **vertices** and a set of **edges** (pairs of vertices)

- **Labels**: additional data on vertices, edges, or both
  - **Weighted**: a graph where edges have numeric labels
- **Directed**: the order of edge pairs matters (edges are arrows) [otherwise **undirected**]
  - **Origin** is first in pair, **Destination** is second in pair
  - **In-neighbors** of vertex are vertices that point to it, **out-neighbors** are vertices it points to
  - **In-degree**: number of edges pointing to vertex, **out-degree**: number of edges from vertex
- **Cyclic**: contains at least one cycle [otherwise acyclic]
- **Simple graph**: No self-loops or parallel edges

Path: sequence of vertices reachable by edges

- **Simple path:** no repeated vertices
- **Cycle**: a path that starts and ends at the same vertex



Self-loop: edge from vertex to itself

Parallel edges: two edges between same vertices in directed graph, going opposite directions

V: Set of vertices

a

b

c

…

E: Set of edges

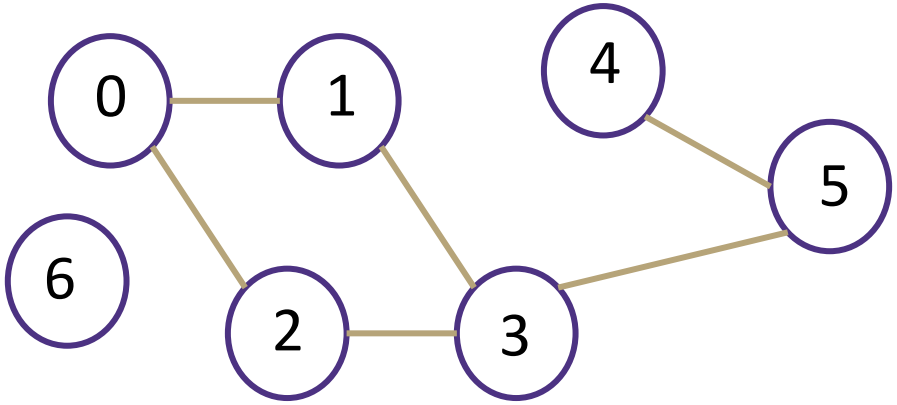
(a, b)

(a, c)

(c, d)

…

Adjacency Matrix



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

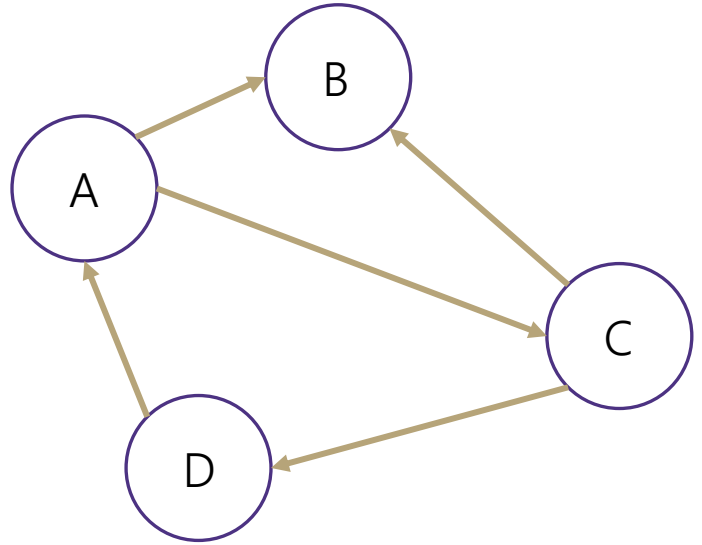
Worst-case Time Complexity (|V| = n, |E| = m):

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

Space Complexity: $\Theta(n^2)$

	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



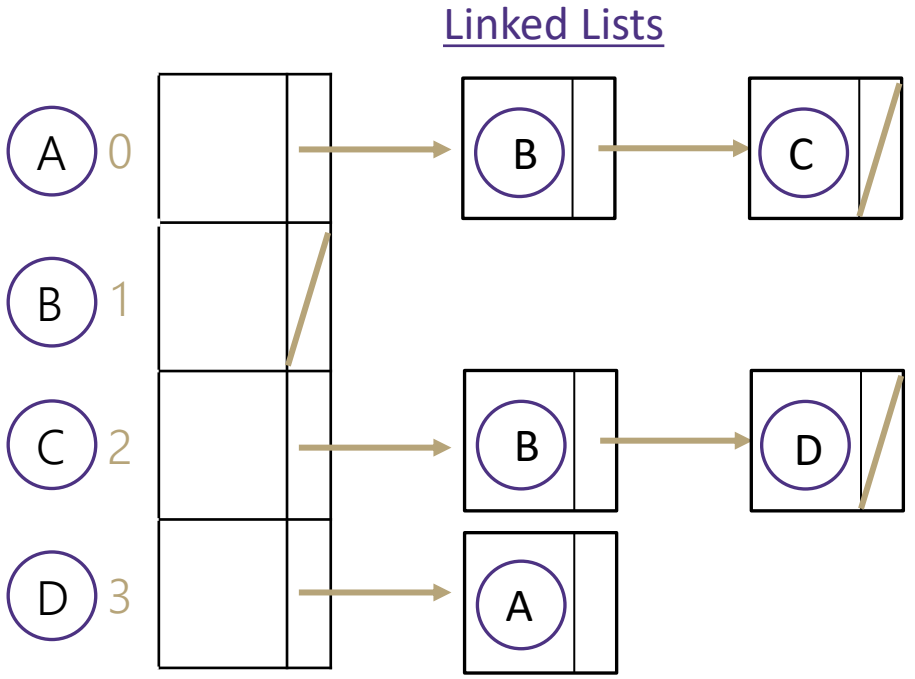
Create a Dictionary of size V from type V to Collection of E

If (x,y) ∈ E then add y to the set associated with the key x

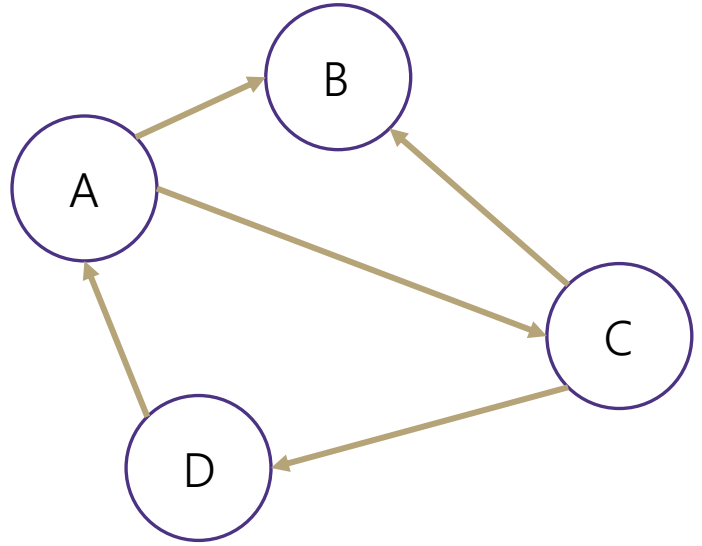
An array where the u^{th} element contains a list of neighbors of u.
Directed graphs: list of out-neighbors (a[u] has v for all (u,v) in E)
Time Complexity (|V| = n, |E| = m):

- Add Edge: $\boldsymbol{\Theta(1)}$
- Remove Edge (u,v): $\boldsymbol{\Theta(\deg(u))}$
- Check edge exists from (u,v): $\boldsymbol{\Theta(\deg(u))}$
- Get neighbors of u (out): $\boldsymbol{\Theta(\deg(u))}$
- Get neighbors of u (in): $\boldsymbol{\Theta(n + m)}$

Space Complexity: $\boldsymbol{\Theta(n + m)}$



Adjacency List



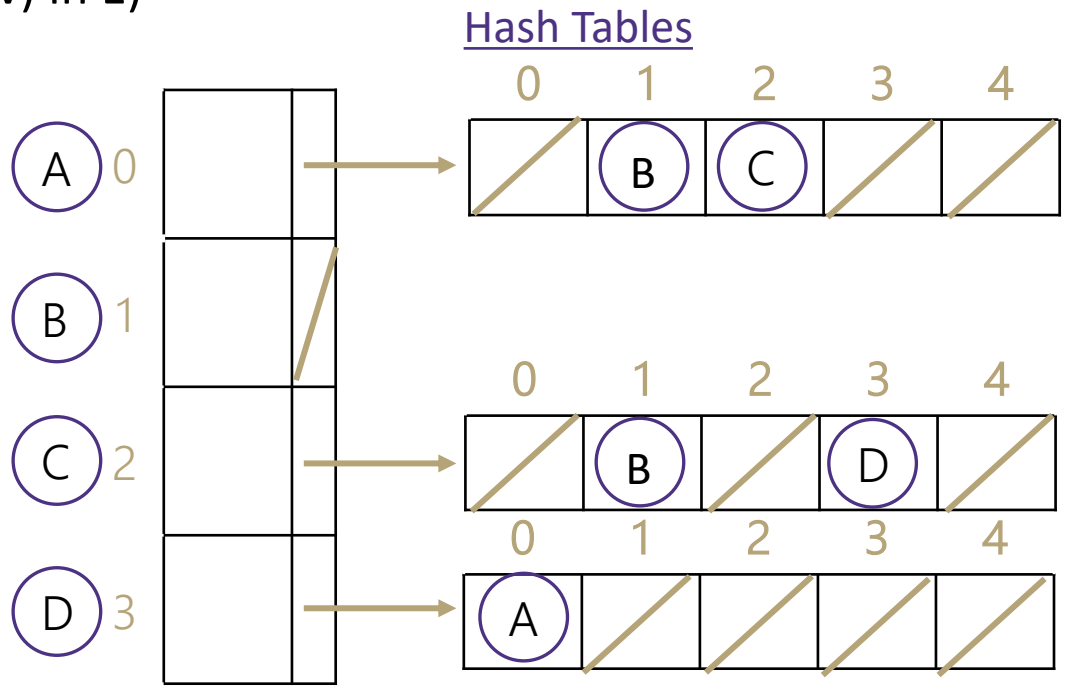
Create a Dictionary of size V from type V to Collection of E

If (x,y) ∈ E then add y to the set associated with the key x

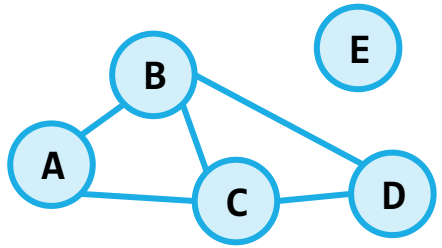
An array where the u^{th} element contains a list of neighbors of u.
Directed graphs: list of out-neighbors (a[u] has v for all (u,v) in E)
Time Complexity (|V| = n, |E| = m):

- Add Edge: $\boldsymbol{\Theta(1)}$
- Remove Edge (u,v): $\boldsymbol{\Theta(1)}$
- Check edge exists from (u,v): $\boldsymbol{\Theta(1)}$
- Get neighbors of u (out): $\boldsymbol{\Theta(\deg(u))}$
- Get neighbors of u (in): $\boldsymbol{\Theta(n)}$

Space Complexity: $\boldsymbol{\Theta(n+m)}$



Adapting for Undirected Graphs



Adjacency Matrix

Store each edge as both directions (makes the matrix symmetrical)

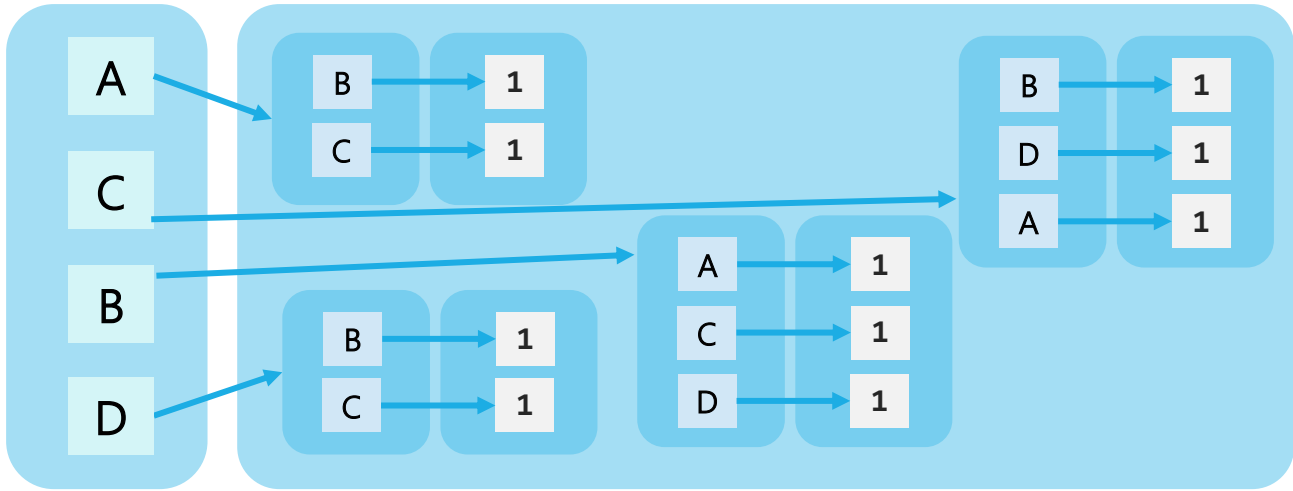
destination

origin	A	B	C	D	E
A	0	1	1	0	0
B	1	0	1	1	0
C	1	1	0	1	0
D	0	1	1	0	0
E	0	0	0	0	0

Adjacency List

Store each edge as both directions (doubles the number of entries)

Keys (origins) Values (*hashmaps* w/ destinations as keys)



Abstraction of the Hash Map! ***Buckets not shown.***

Tradeoffs

Adjacency Matrices take more space, not always faster, why would you use them?

- Checking for an edge is $\Theta(1)$, but finding the neighbors takes $\Theta(n)$ time.
- 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.
- Sometimes the matrix itself is useful ("spectral graph theory")

What's the tradeoff between using linked lists and hash tables for the list of neighbors?

- A hash table still *might* hit a worst-case
- And the linked list might not
  - Graph algorithms often just need to iterate over all the neighbors, so you might get a better guarantee with the linked list.

373: Assumed Graph Implementations

For this class, unless otherwise stated, assume we're using an **adjacency list** with **hash maps**.

- Also unless otherwise stated, assume all graph hash map operations are O(1). This is a pretty reasonable assumption, because for most problems we examine you know the set of vertices ahead of time and can prevent resizing.

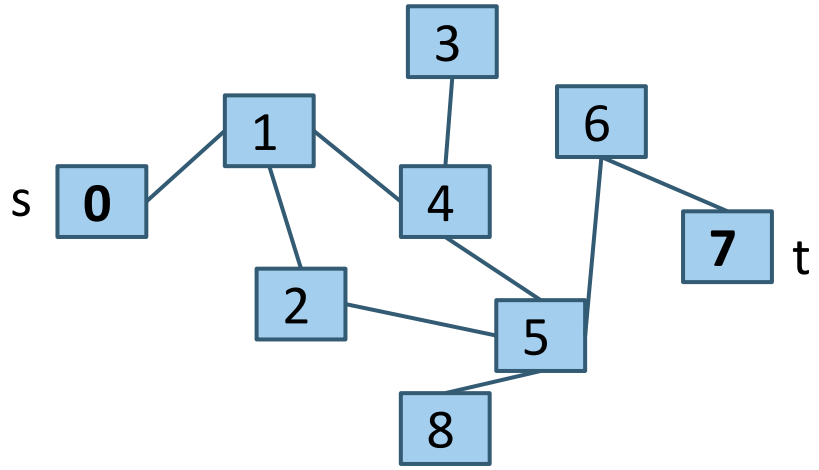
Add Edge	$\Theta(1)$
Remove Edge	$\Theta(1)$
Check if edge (u, v) exists	$\Theta(1)$
Get out-neighbors of u	$\Theta(\deg(u))$
Get in-neighbors of v	$\Theta(n)$
(Space Complexity)	$\Theta(n+m)$

(|V| = n, |E| = m)

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**?

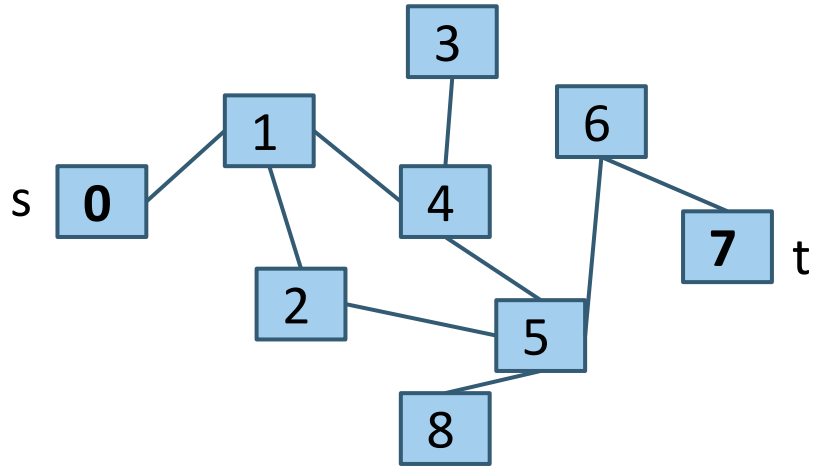


Try to come up with an algorithm for `connected(s, t)`

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

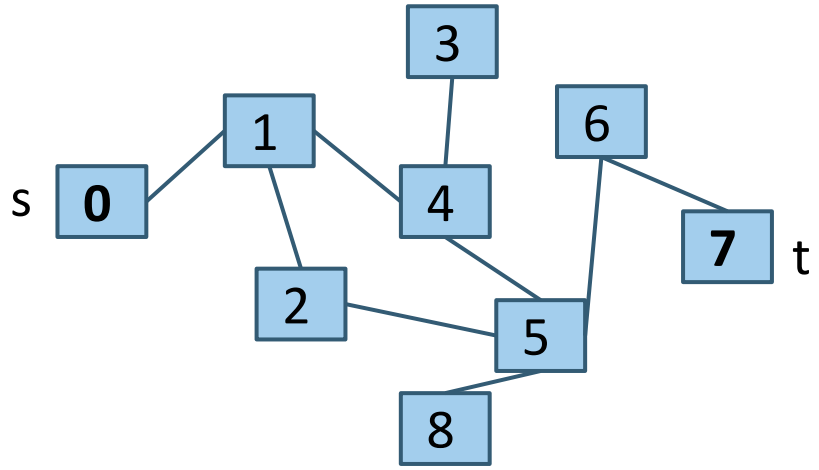
s-t Connectivity Problem: Proposed Solution

```
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;
  }
}
```

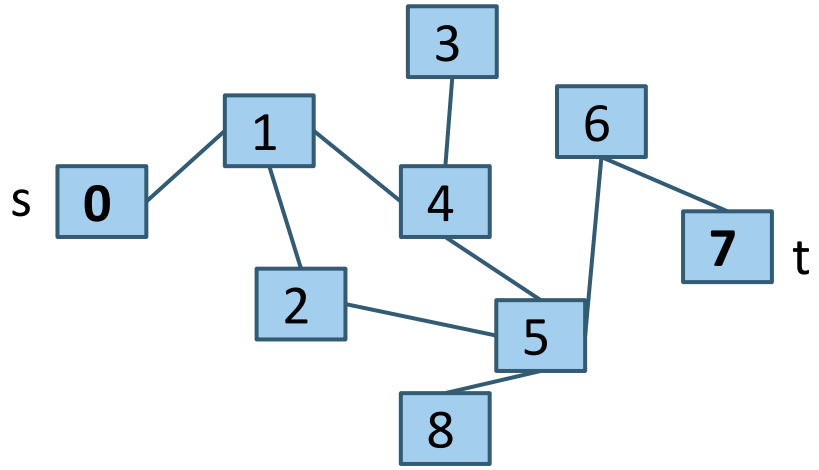


Does 0 == 7? No; if(connected(1, 7) return true;
Does 1 == 7? No; if(connected(0, 7) return true;
Does 0 == 7?

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;
  }
}
```



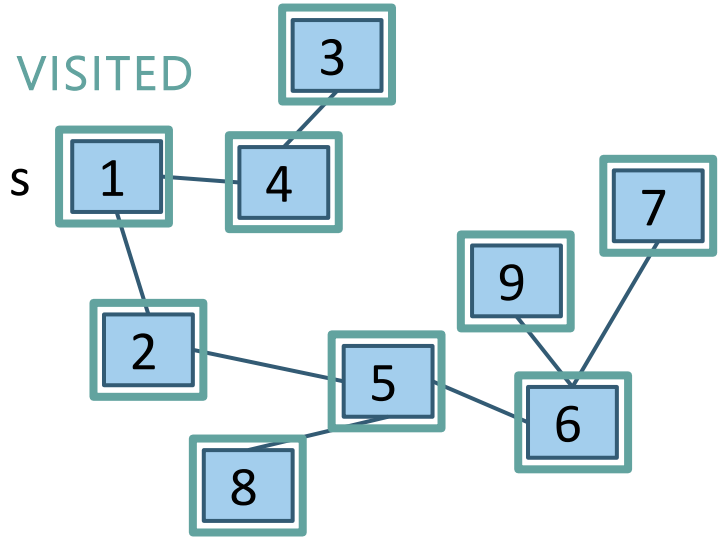
This general approach for crawling through a graph is going to be the basis for a LOT of algorithms!

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;
  }
}
```

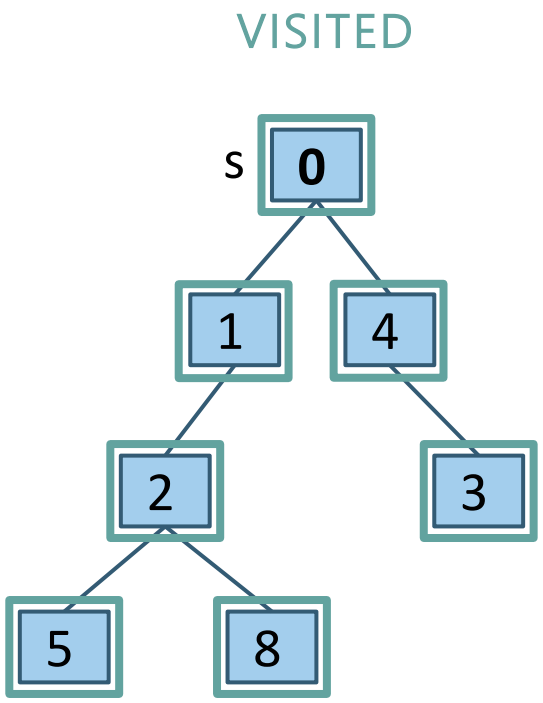
- It will explore one option "all the way down" before coming back to try other options
  - Many possible orderings: {0, 1, 2, 5, 6, 9, 7, 8, 4, 3} or {0, 1, 4, 3, 2, 5, 8, 6, 7, 9} both possible
- We call this approach a depth-first search (DFS)



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;
  }
}
```

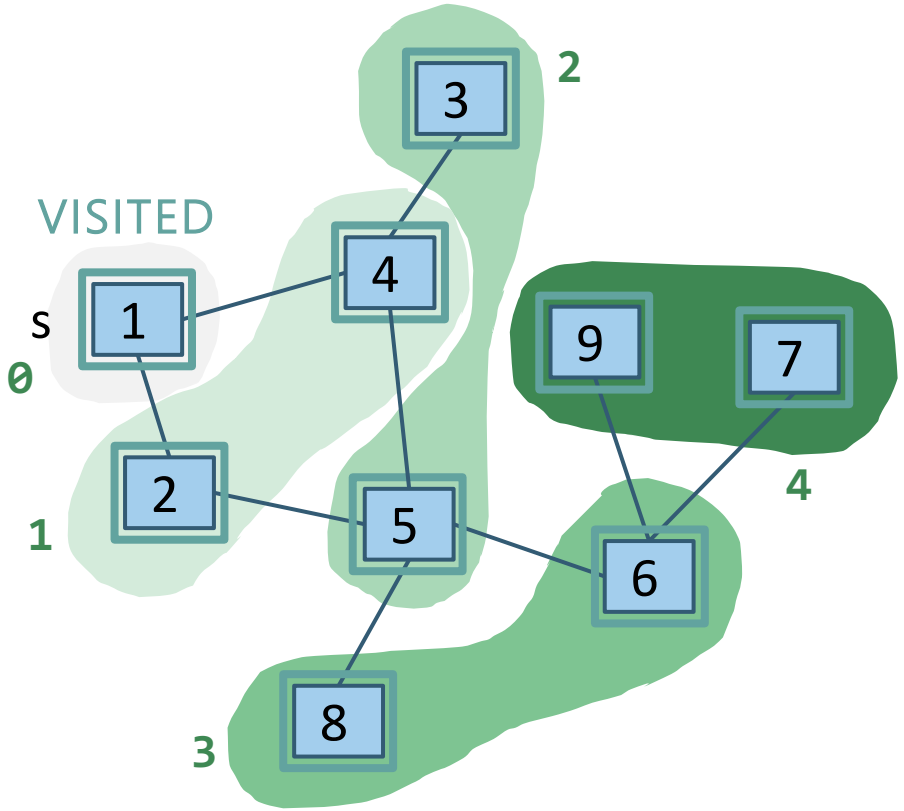


- *CSE 143 Review* traversing a binary tree depth-first has 3 options:
  - Pre-order: visit node before its children
  - In-order: visit node between its children
  - Post-order: visit node after its children
- The difference between these orderings is when we "process" the root – all are DFS!

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

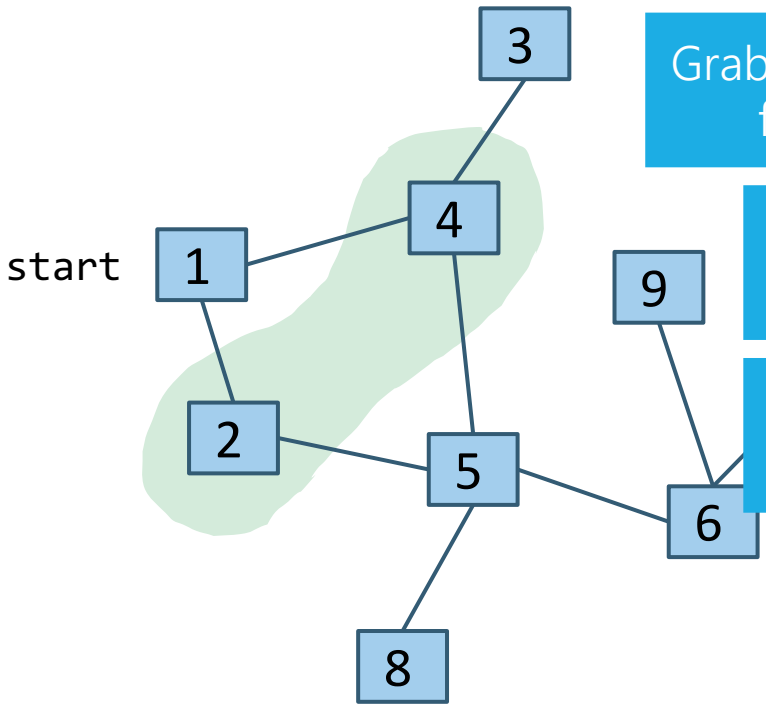
Our extra data structure! Will keep track of "outer edge" of nodes still to explore

Kick off the algorithm by adding start to perimeter

Grab one element at a time from the perimeter

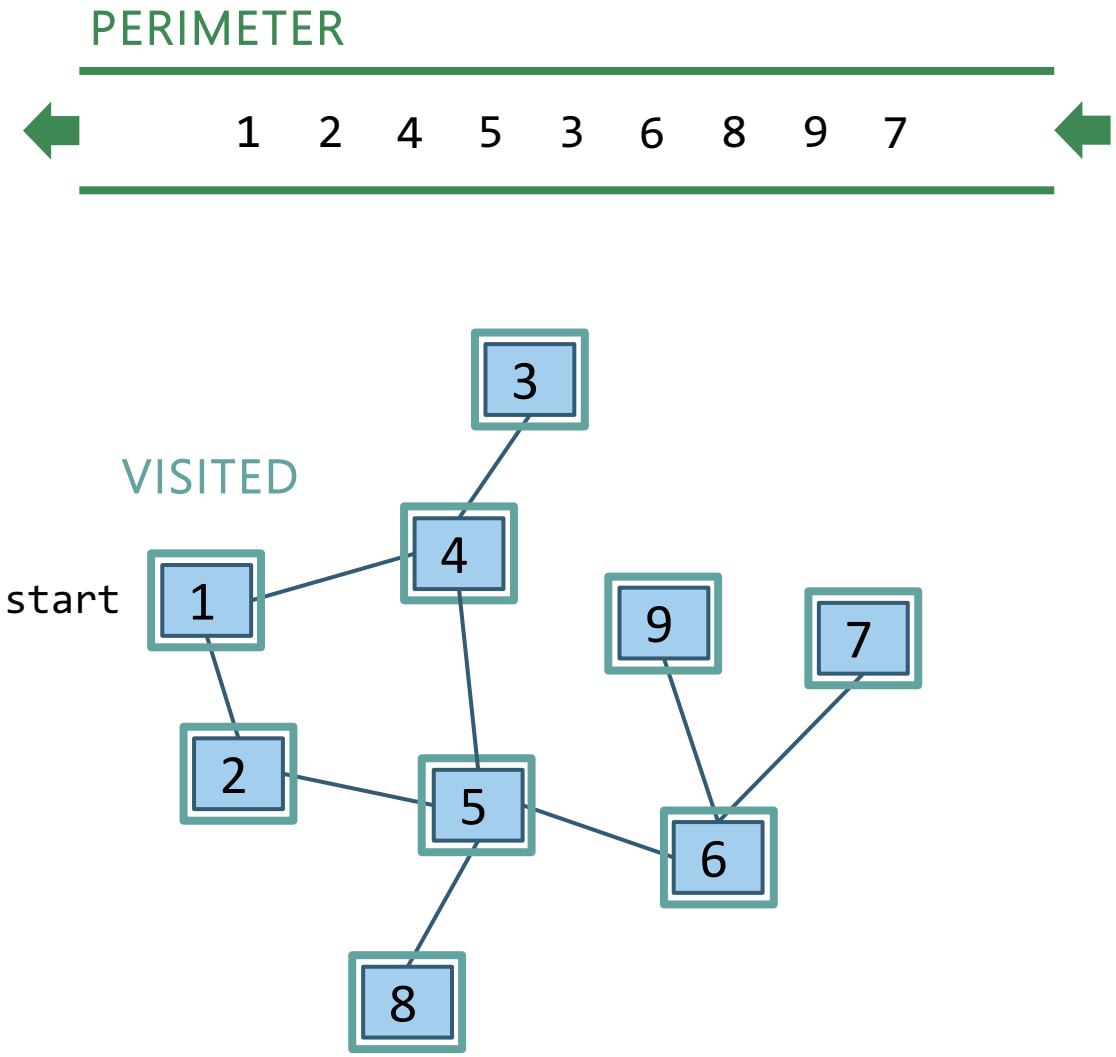
Look at all that element's children

Add new ones to perimeter!



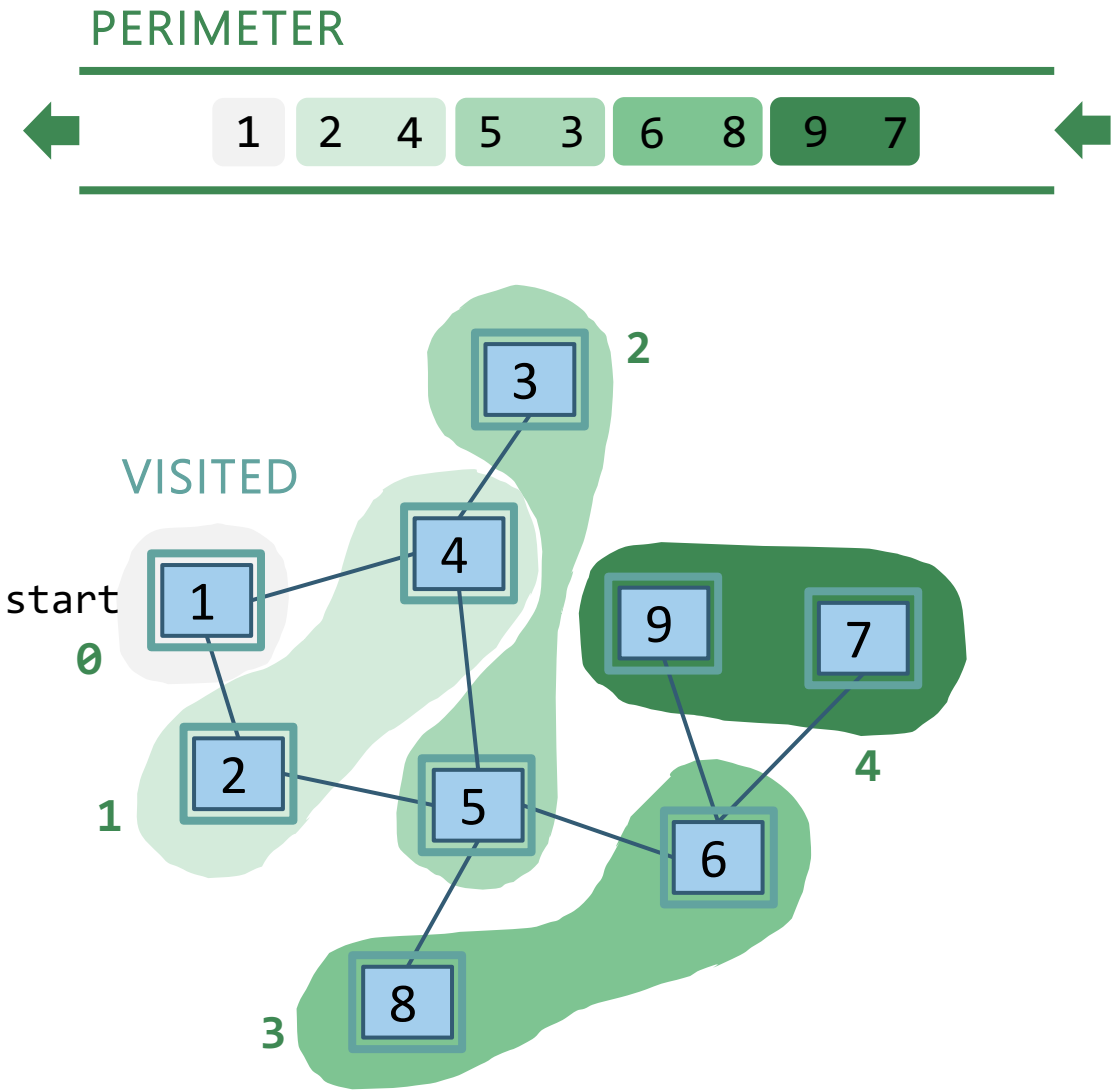

```
bfs(Graph graph, Vertex start) {
  Queue<Vertex> perimeter = new Queue<>();
  Set<Vertex> visited = new Set<>();

  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!

Just change the Queue to a Stack and it becomes DFS!
Now we'll immediately explore the most recent child

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

*there is a bug in this code... can you find it?

```
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

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;
}
```

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 make up 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 make up 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.

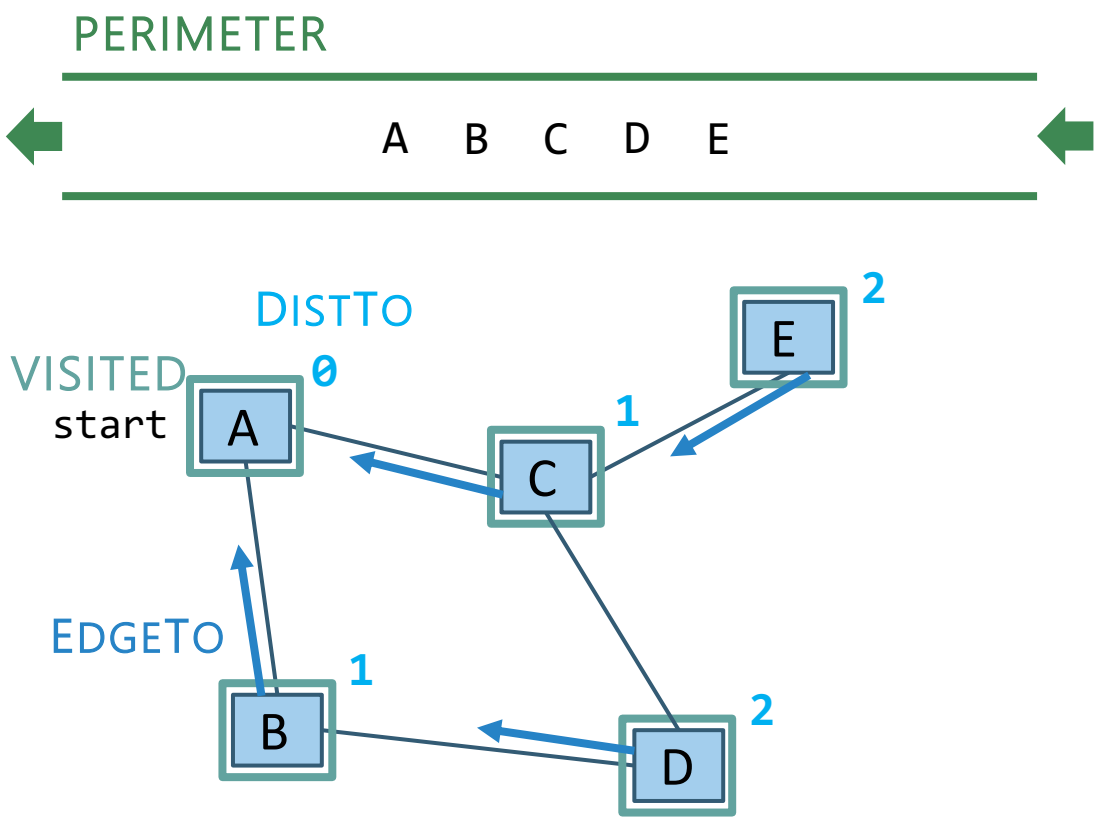
- Sounds like a job for?
  - BFS!

```
...
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



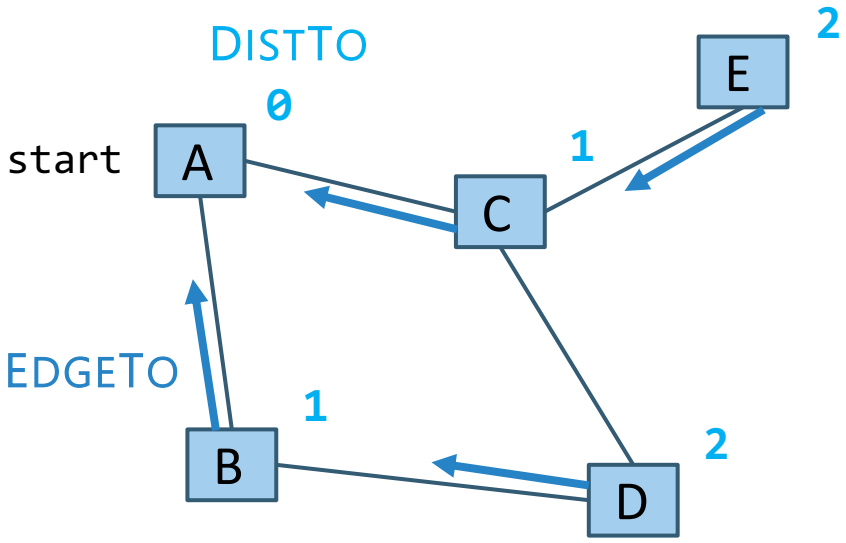
The **edgeTo** map stores backpointers: each vertex remembers what vertex was used to arrive at it!

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

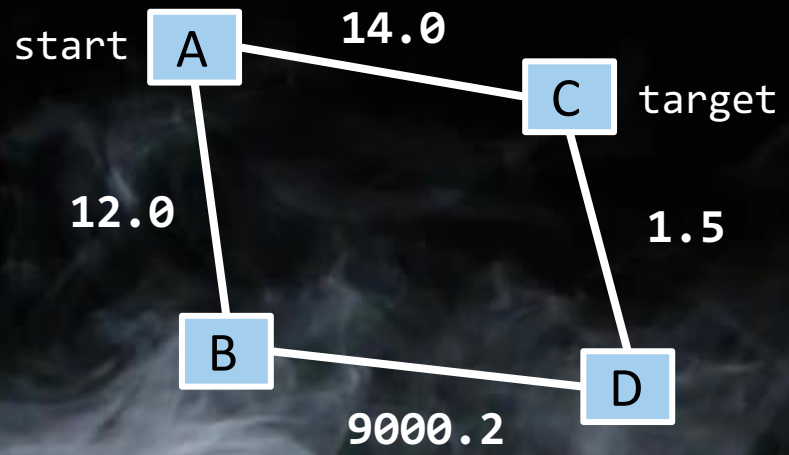
Given source vertex **s** and a target vertex **t**, how long is the shortest path from **s** to **t**? What edges make up that path?

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

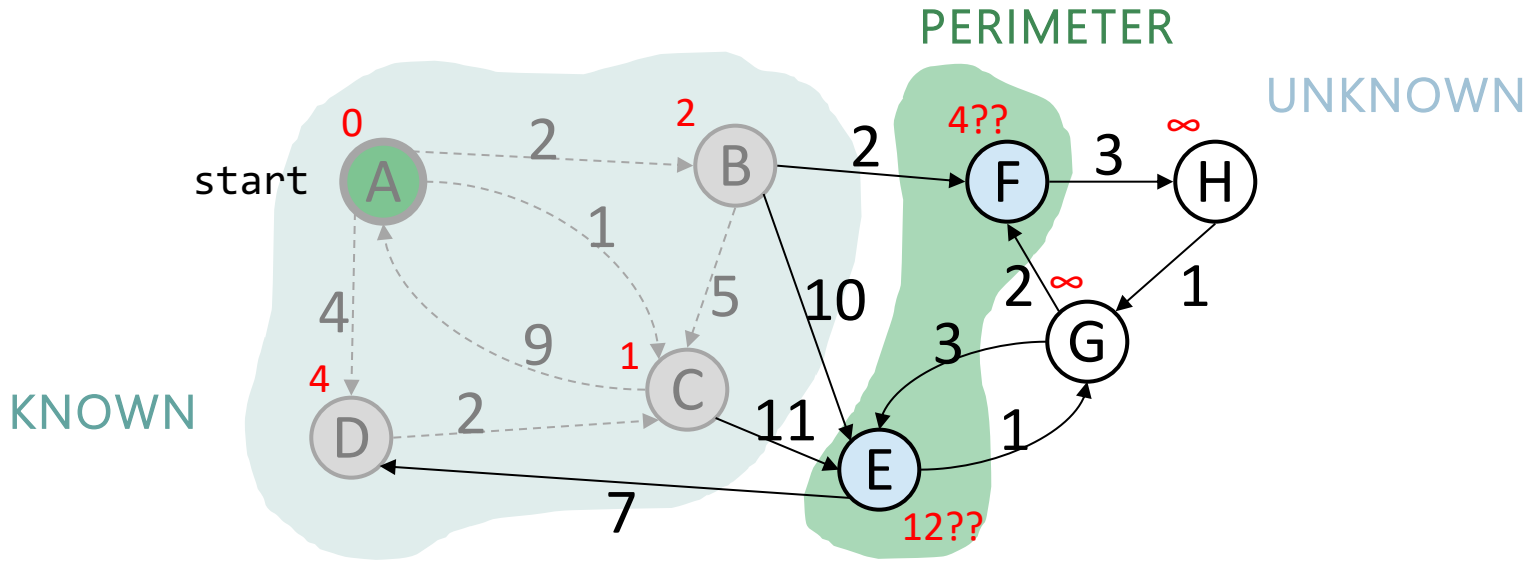
Named after its inventor, Edsger Dijkstra (1930-2002)

- 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



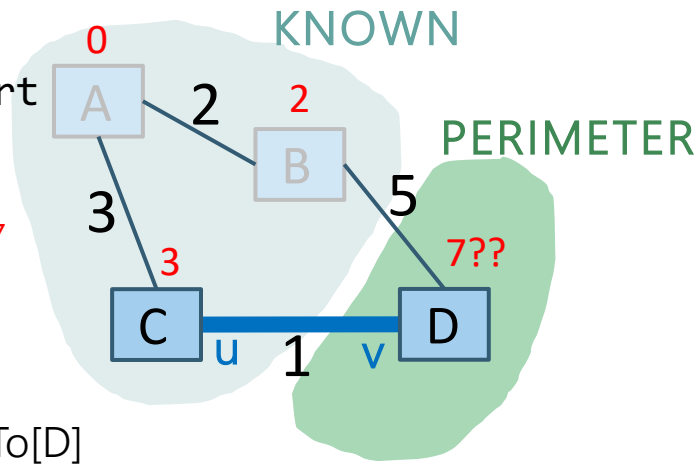
Initialization:

- 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

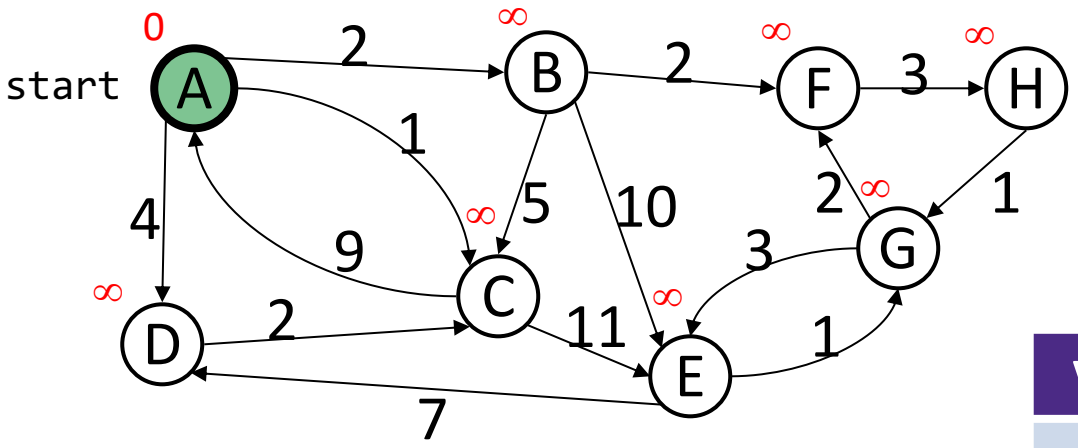
- We call this update relaxing the distance because it only ever shortens the current best path

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)
```

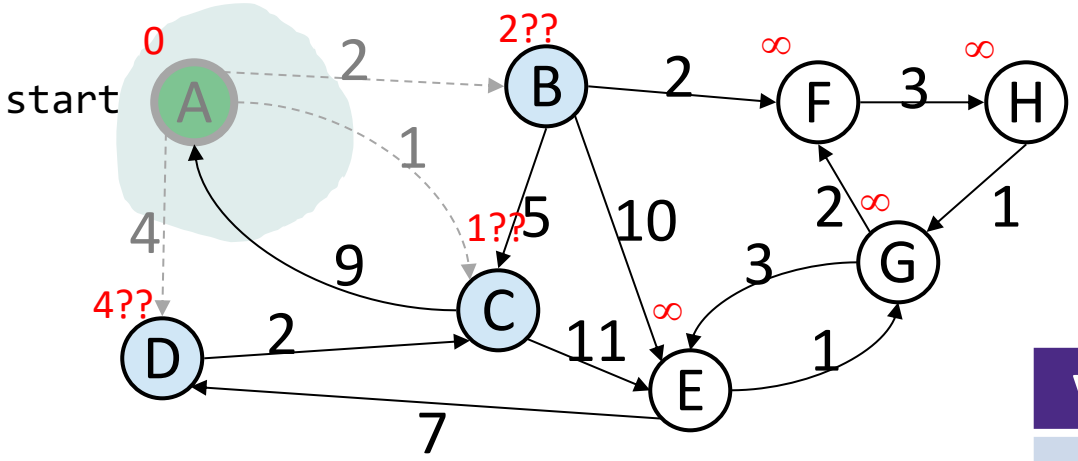
Dijkstra's Algorithm: Example #1



Order Added to Known Set:

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

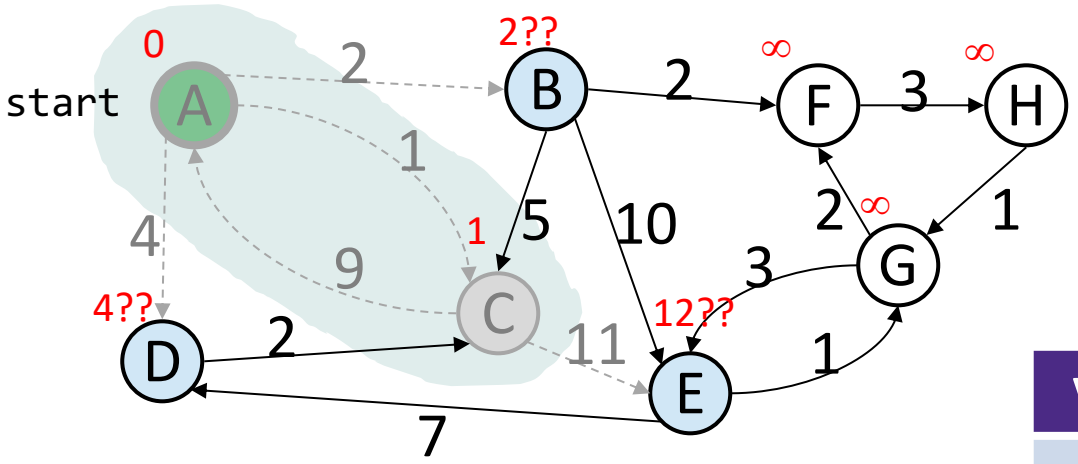
Dijkstra's Algorithm: Example #1



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

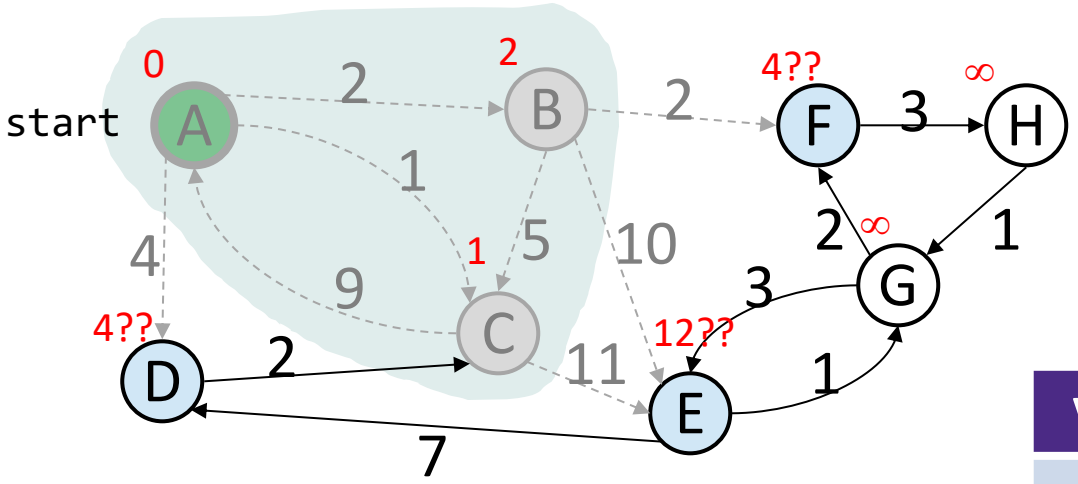
Dijkstra's Algorithm: Example #1



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		∞	

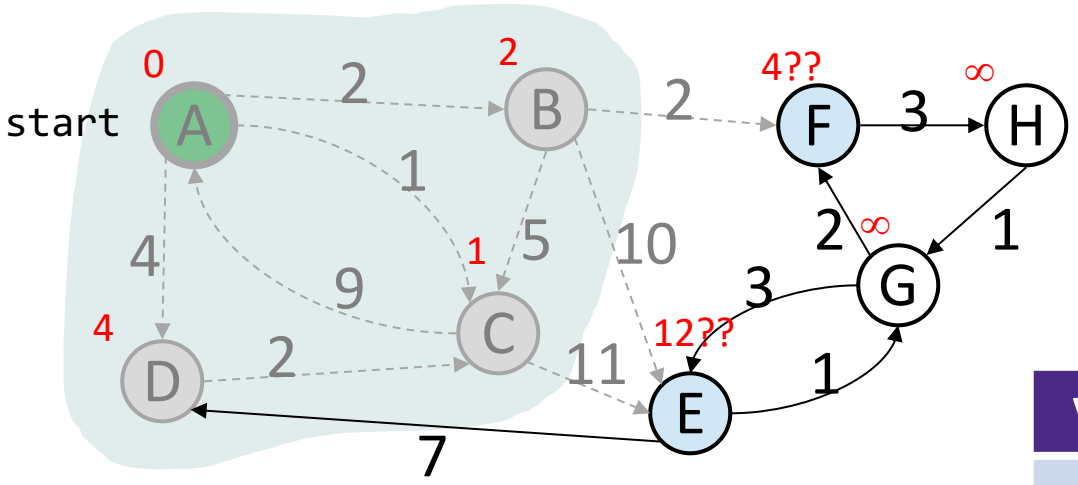
Dijkstra's Algorithm: Example #1



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		∞	

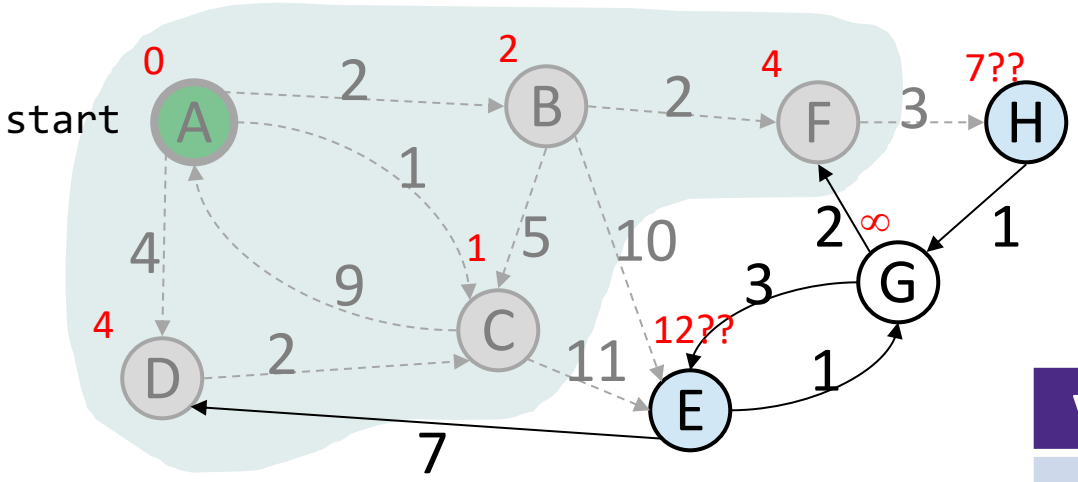
Dijkstra's Algorithm: Example #1



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		∞	

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

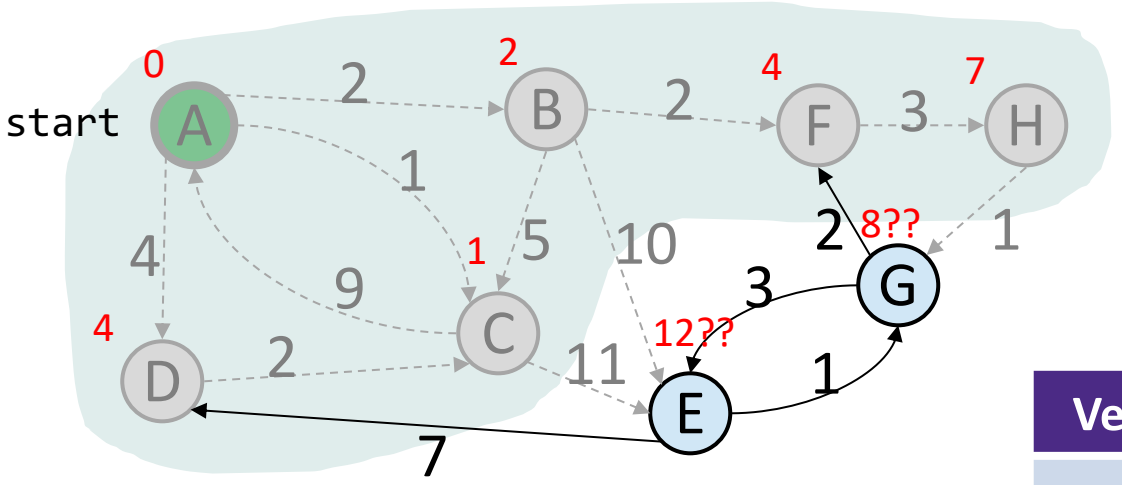
Dijkstra's Algorithm: Example #1



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**

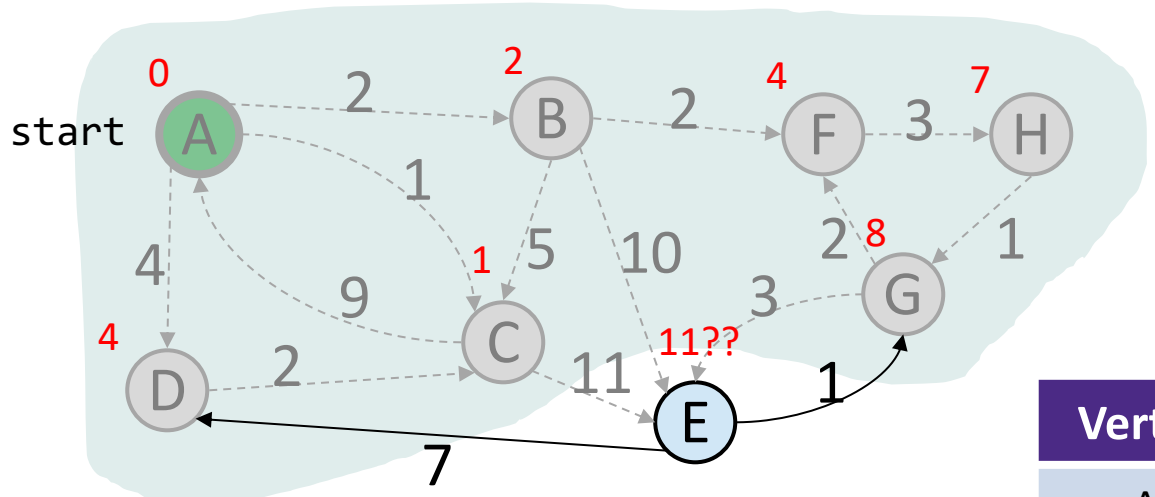
Dijkstra's Algorithm: Example #1



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

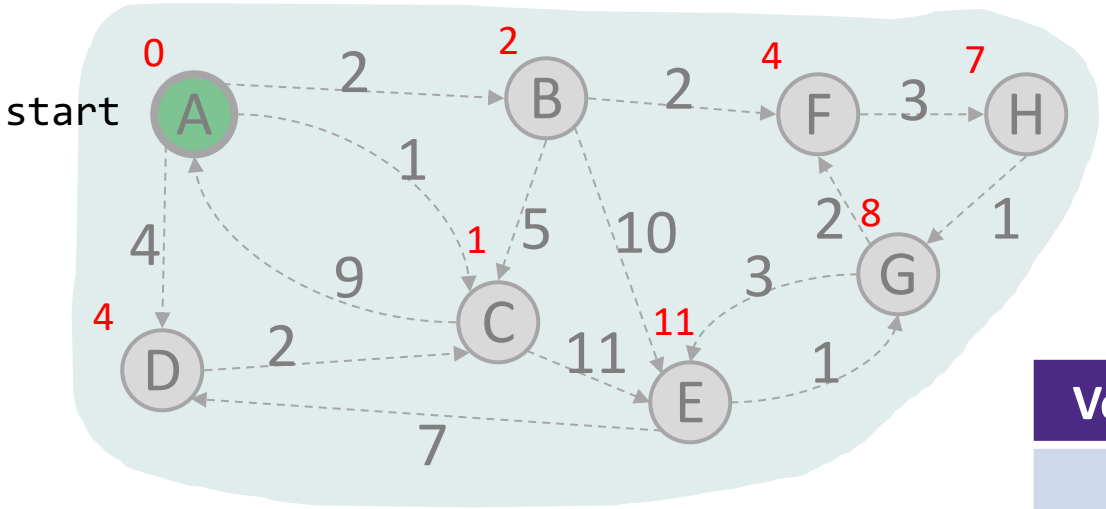
Dijkstra's Algorithm: Example #1



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

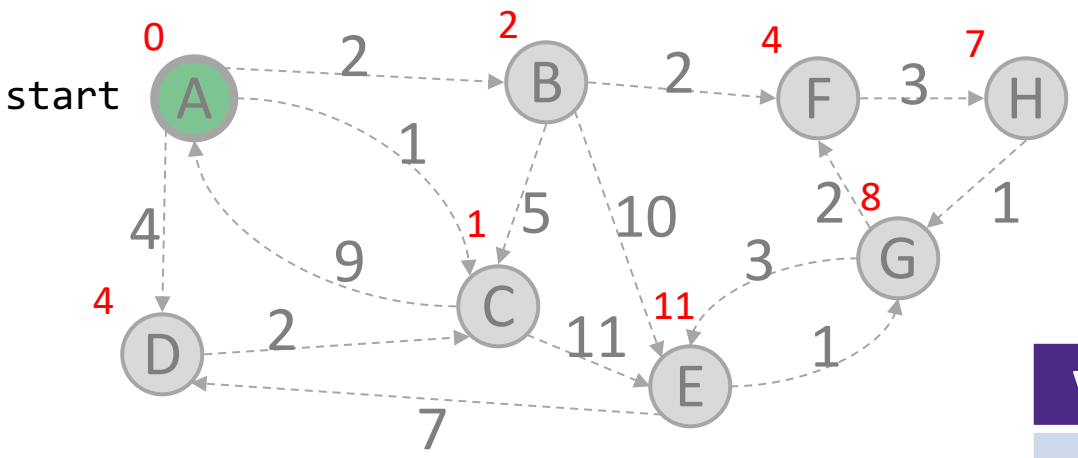
Dijkstra's Algorithm: Example #1



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

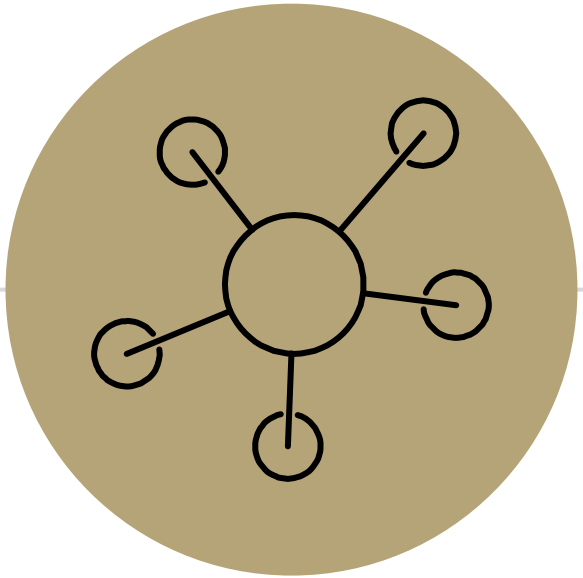
While a vertex is not known, another shorter path might be found!

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**



Appendix

Graph problems

There are lots of interesting questions we can ask about a graph:

- What is the shortest route from S to T?
- What is the longest without cycles?
- Are there cycles?
- Is there a tour (cycle) you can take that only uses each node (station) exactly once?
- Is there a tour (cycle) that uses each edge exactly once?

Graph problems

Some well known graph problems and their common names:

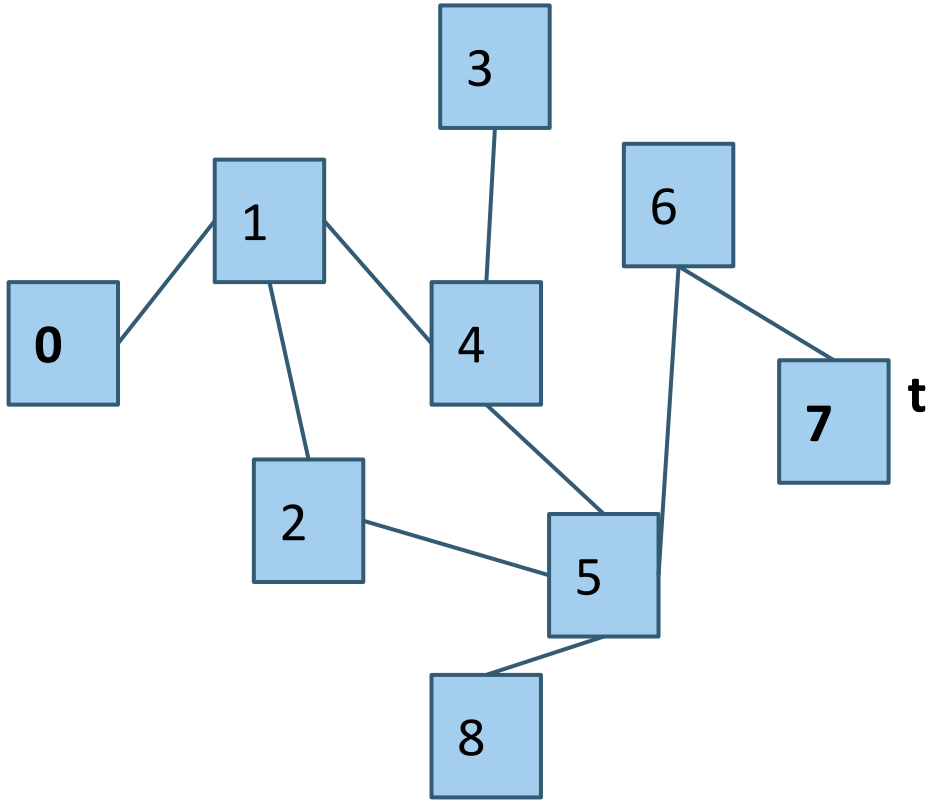
- s-t Path. Is there a path between vertices s and t?
- Connectivity. Is the graph connected?
- Biconnectivity. Is there a vertex whose removal disconnects the graph?
- Shortest s-t Path. What is the shortest path between vertices s and t?
- Cycle Detection. Does the graph contain any cycles?
- Euler Tour. Is there a cycle that uses every edge exactly once?
- Hamilton Tour. Is there a cycle that uses every vertex exactly once?
- Planarity. Can you draw the graph on paper with no crossing edges?
- Isomorphism. Are two graphs the same graph (in disguise)?

Graph problems are among the most mathematically rich areas of CS theory!

s-t path Problem

s-t path problem

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



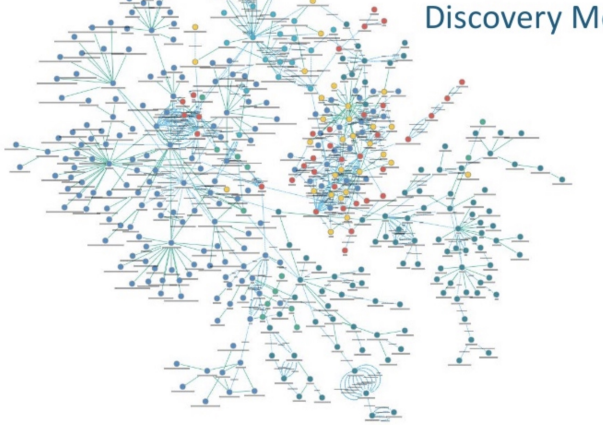
Why does this problem matter? Some possible context:

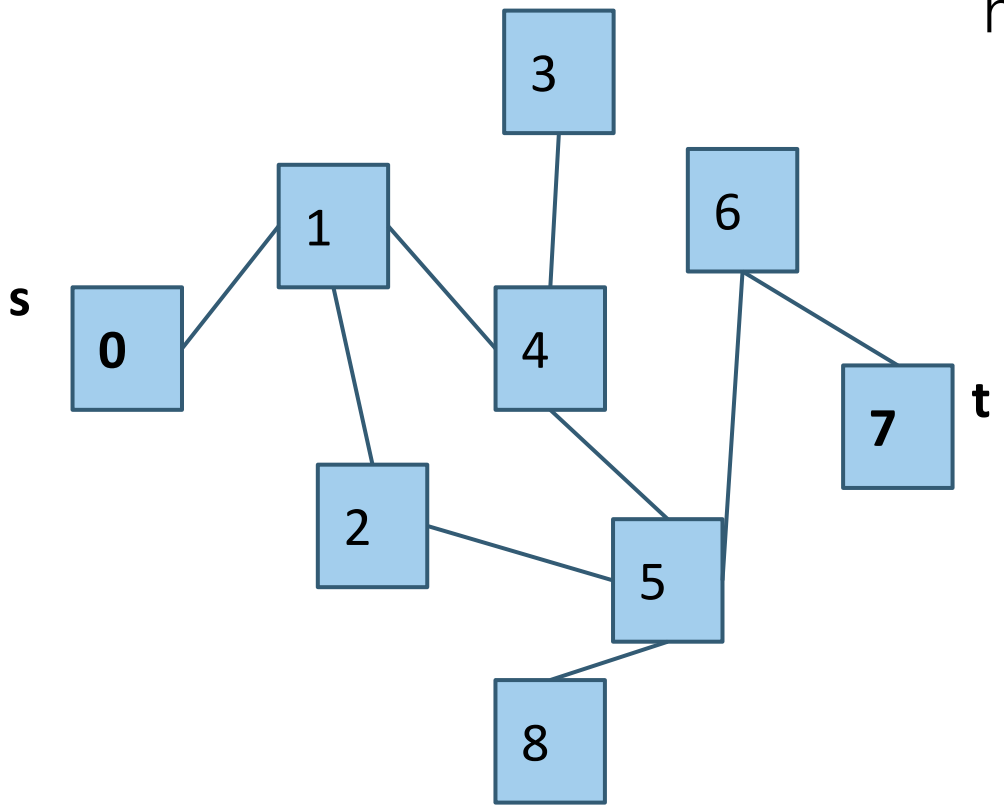
- real life maps and trip planning – can we get from one location (vertex) to another location (vertex) given the current available roads (edges)
- family trees and checking ancestry – are two people (vertices) related by some common ancestor (edges for direct parent/child relationships)
- game states (Artificial Intelligence) can you win the game from the current vertex (think: current board position)? Are there moves (edges) you can take to get to the vertex that represents an already won game?

s-t path Problem

s-t path problem

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





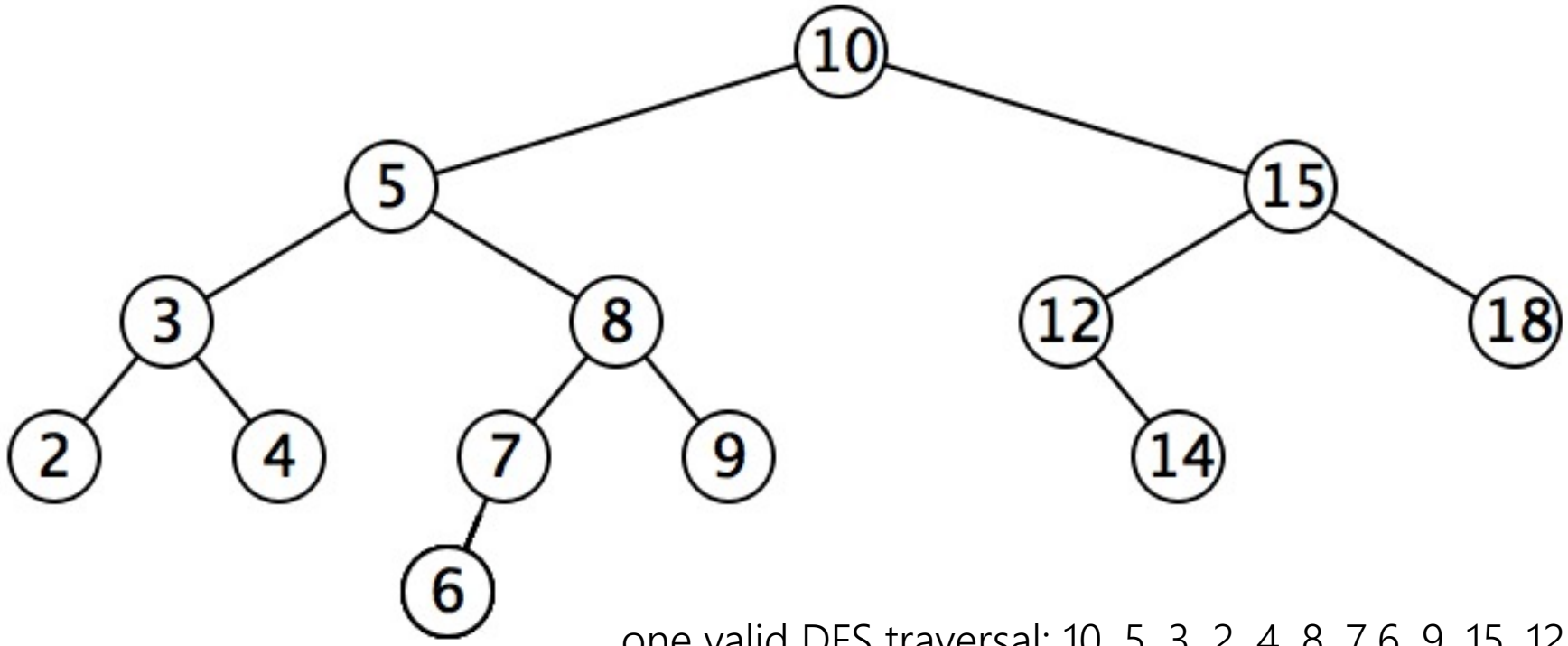
- What's the answer for this graph on the left, and how did we get that answer as humans?
  - We can see there's edges that are visually in between s and t, and we can try out an example path and make sure that by traversing that path you can get from s to t.
  - We know that doesn't scale that well though, so now let's try to define a more algorithmic (comprehensive) way to find these paths. The main idea is: starting from the specified s, try traversing through every single possible path possible that's not redundant to see if it could lead to t.

traversals are really important to solving this problem / problems in general, so slight detour to talk about them, we'll come back to this though

Graph traversals: DFS

Depth First Search - a traversal on graphs (or on trees since those are also graphs) where you traverse "deep nodes" before all the shallow ones

High-level DFS: you go as far as you can down one path till you hit a dead end (no neighbors are still undiscovered or you have no neighbors). Once you hit a dead end, you backtrack / undo until you find some options/edges that you haven't actually tried yet.



Kind of like wandering a maze – if you get stuck at a dead end (since you physically have to go and try it out to know it's a dead end), trace your steps backwards towards your last decision and when you get back there, choose a different option than you did before.

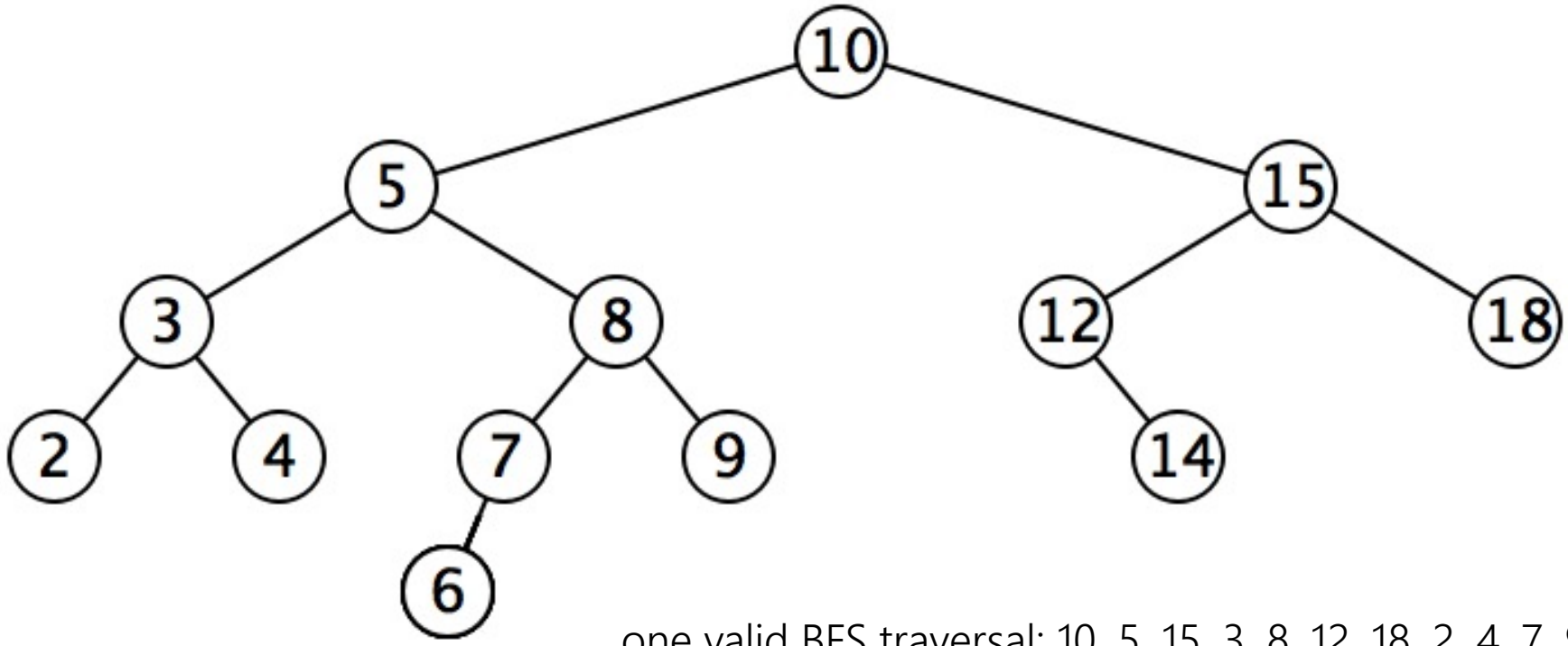
one valid DFS traversal: 10, 5, 3, 2, 4, 8, 7,6, 9, 15, 12, 14, 18

Graph traversals: BFS

Breadth First Search - a traversal on graphs (or on trees since those are also graphs) where you traverse level by level. So in this one we'll get to all the shallow nodes before any "deep nodes".

Intuitive ways to think about BFS:

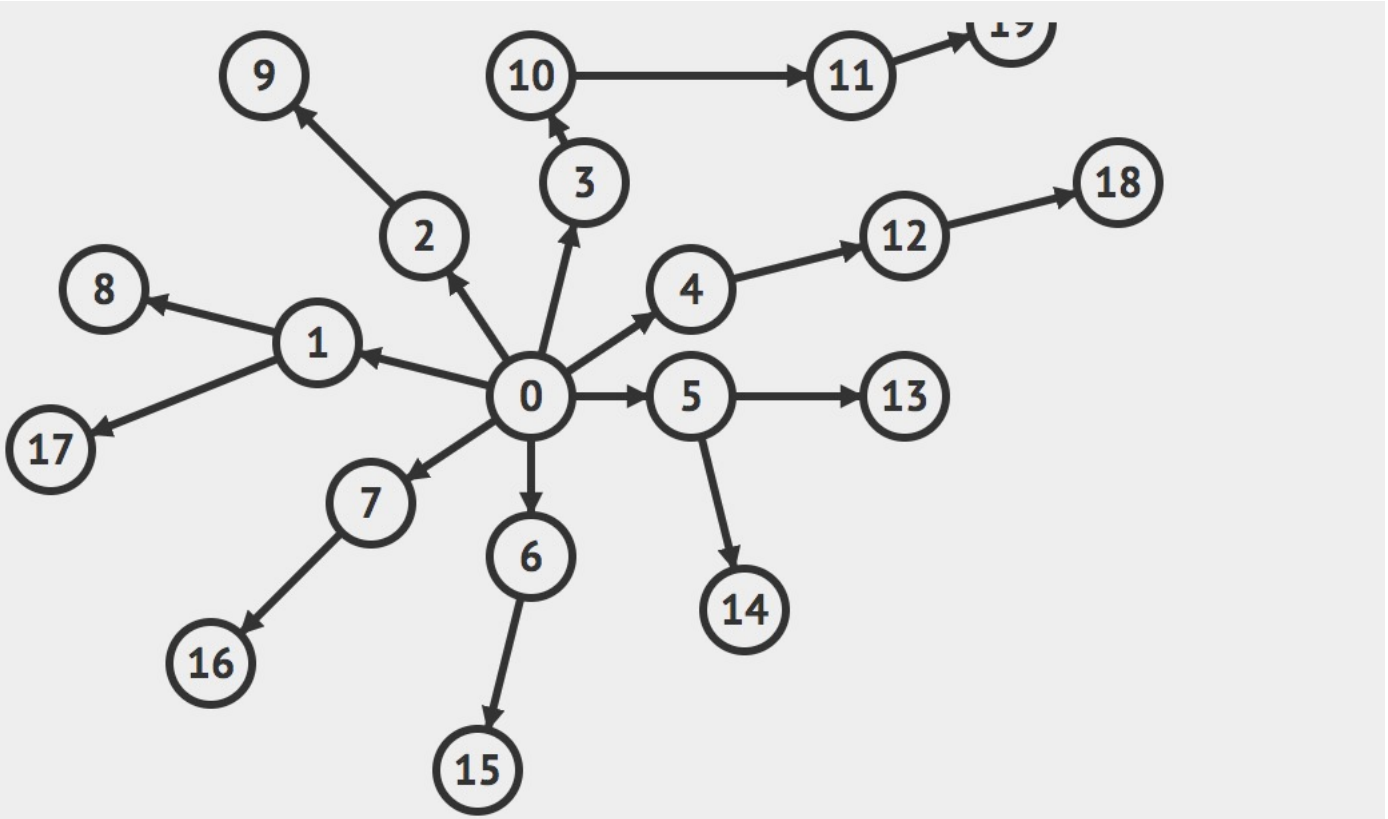
- opposite way of traversing compared to DFS
- a sound wave spreading from a starting point, going outwards in all directions possible.
- mold on a piece of food spreading outwards so that it eventually covers the whole surface



one valid BFS traversal: 10, 5, 15, 3, 8, 12, 18, 2, 4, 7, 9, 14, 6

Graph traversals: BFS and DFS on more graphs

In DFS, you go as far as you can down one path till you hit a dead end (no neighbors are still undiscovered or you have no neighbors). Once you hit a dead end, you backtrack / undo until you find some options/edges that you haven't actually tried yet.



In BFS, you traverse level by level

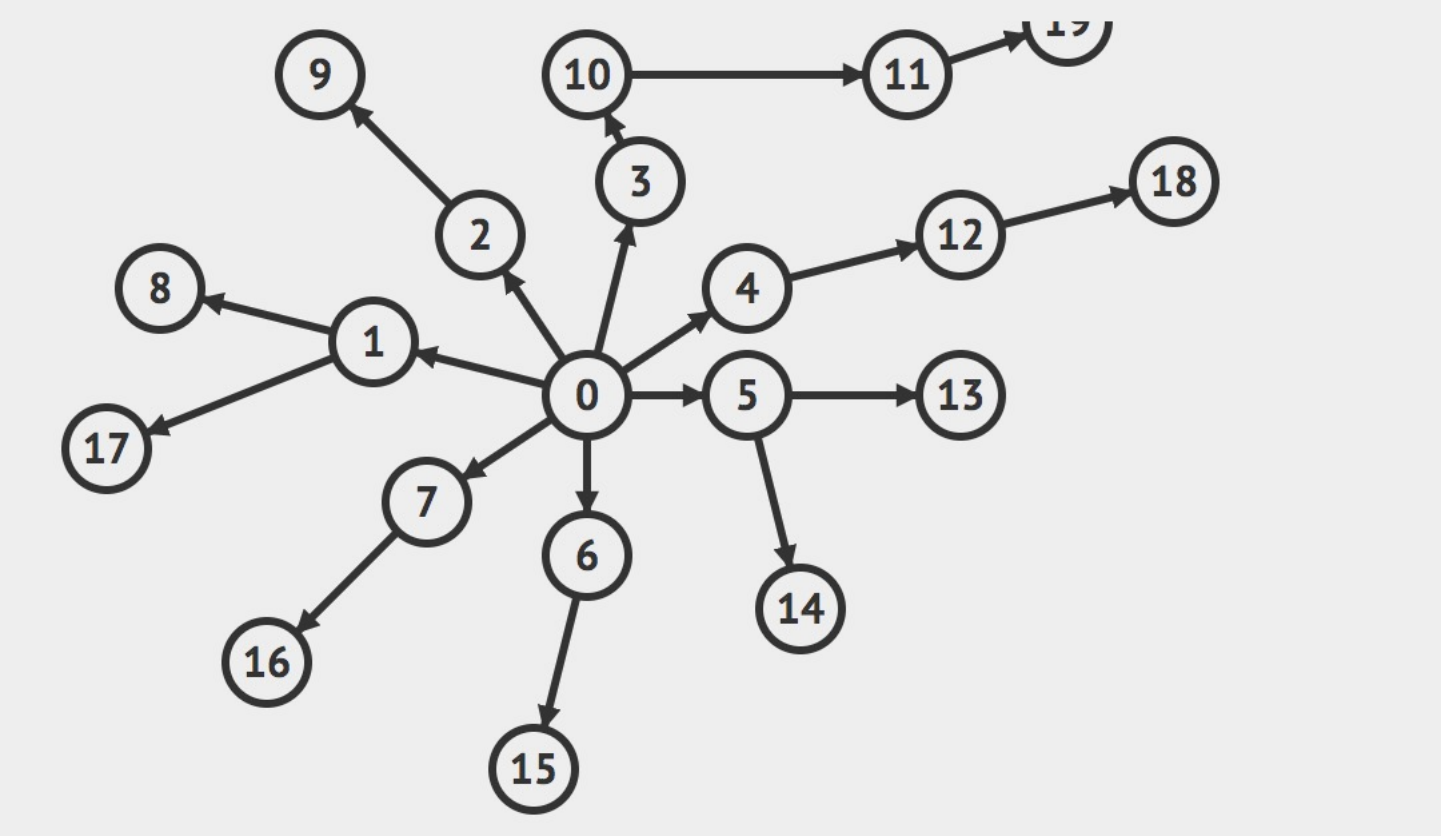
Take 2 minutes and try to come up with two possible traversal orderings <u>starting with the 0 node</u>:

- a BFS ordering (ordering within each layer doesn't matter / any ordering is valid)
- a DFS ordering (ordering which path you choose next at any point doesn't matter / any is valid as long as you haven't explored it before)

@ordering choices will be more stable when we have code in front of us, but not the focus / point of the traversals so don't worry about it

Graph traversals: BFS and DFS on more graphs

In DFS, you go as far as you can down one path till you hit a dead end (no neighbors are still undiscovered or you have no neighbors). Once you hit a dead end, you backtrack / undo until you find some options/edges that you haven't actually tried yet.

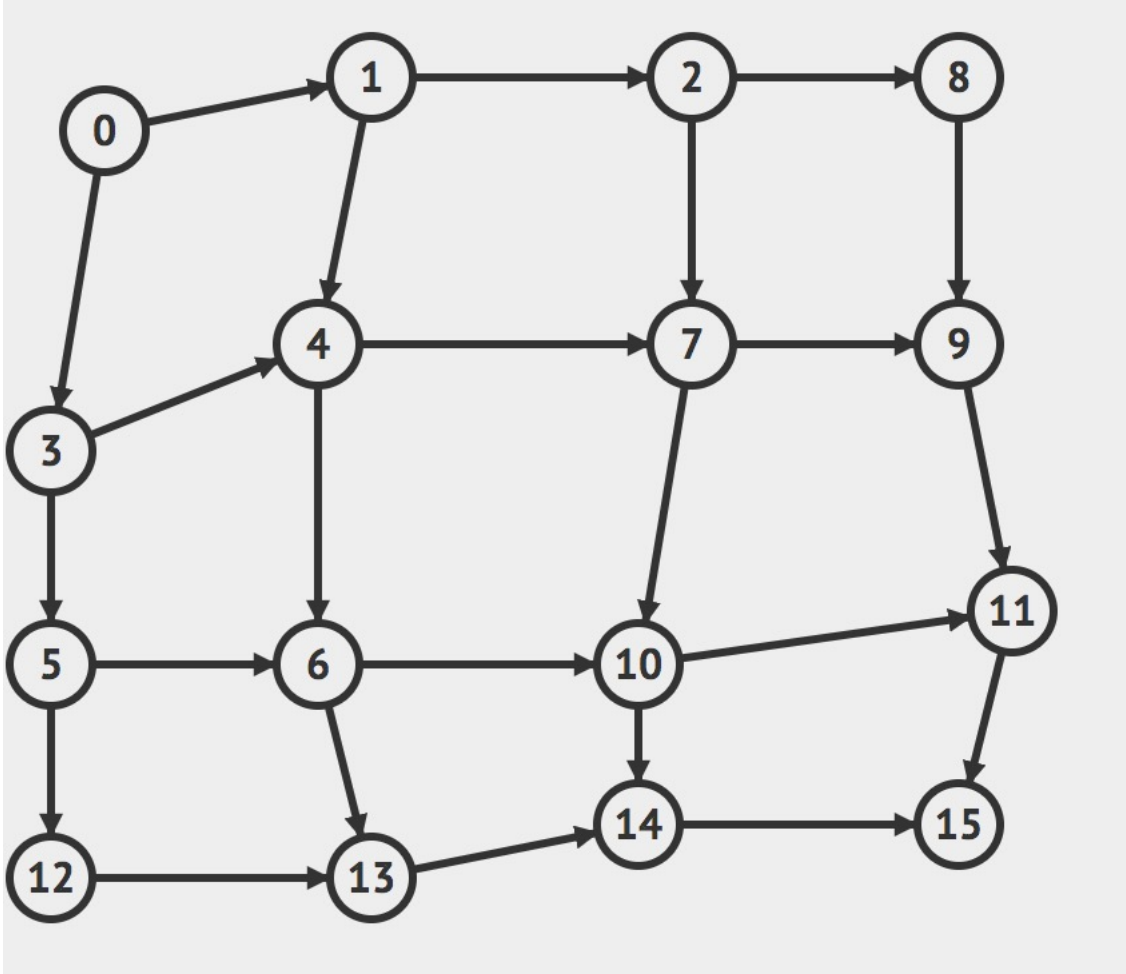


In BFS, you traverse level by level

Take a minute and try to come up with two possible traversal orderings <u>starting with the 0 node</u>:

- a BFS ordering (ordering within each layer doesn't really matter / any ordering is valid)
  - 0, [1, 2, 3, 4, 5, 6, 7], [8, 9, 10, 12, 13, 14, 15, 16, 17], [11, 18], [19]
- a DFS ordering (ordering which path you choose next at any point doesn't matter / any is valid as long as you haven't explored it before)
  - 0, 2, 9, 3, 10, 11, 19, 4, 12, 18, 5, 13, 14, 6, 15, 7, 16, 1,17, 8

Graph traversals: BFS and DFS on more graphs

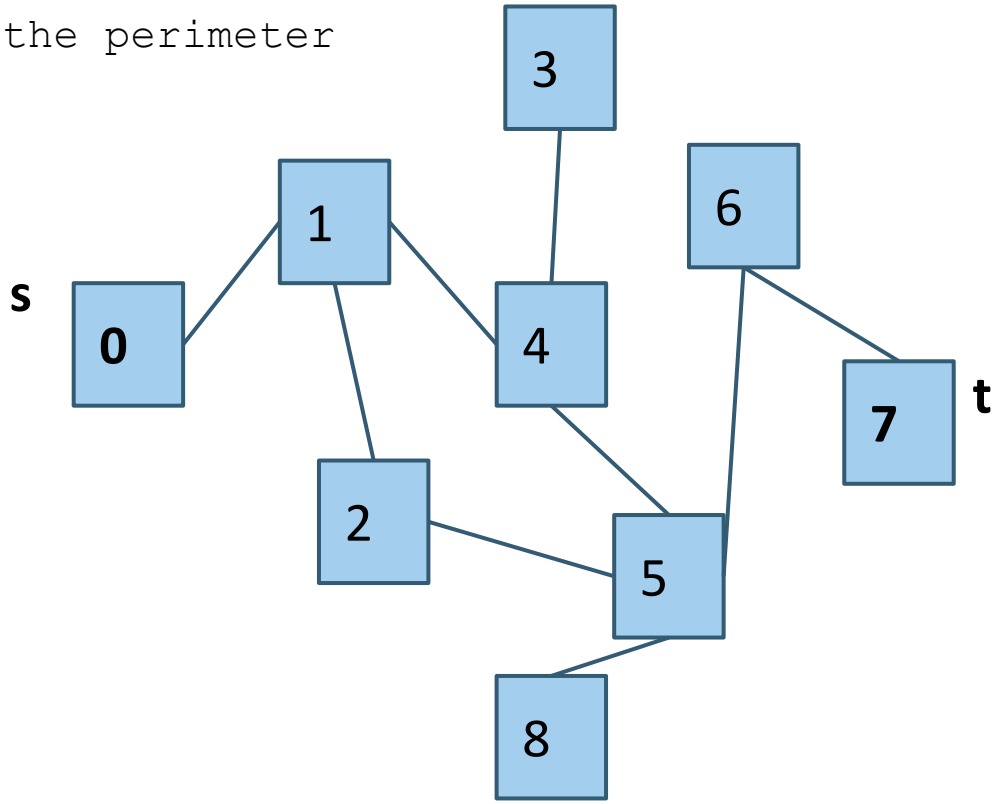


https://visualgo.net/en/dfsbfs

- click on draw graph to create your own graphs and run BFS/DFS on them!
- check out visualgo.net for more really cool interactive visualizations
- or do your own googling – there are a lot of cool visualizations out there ☺!

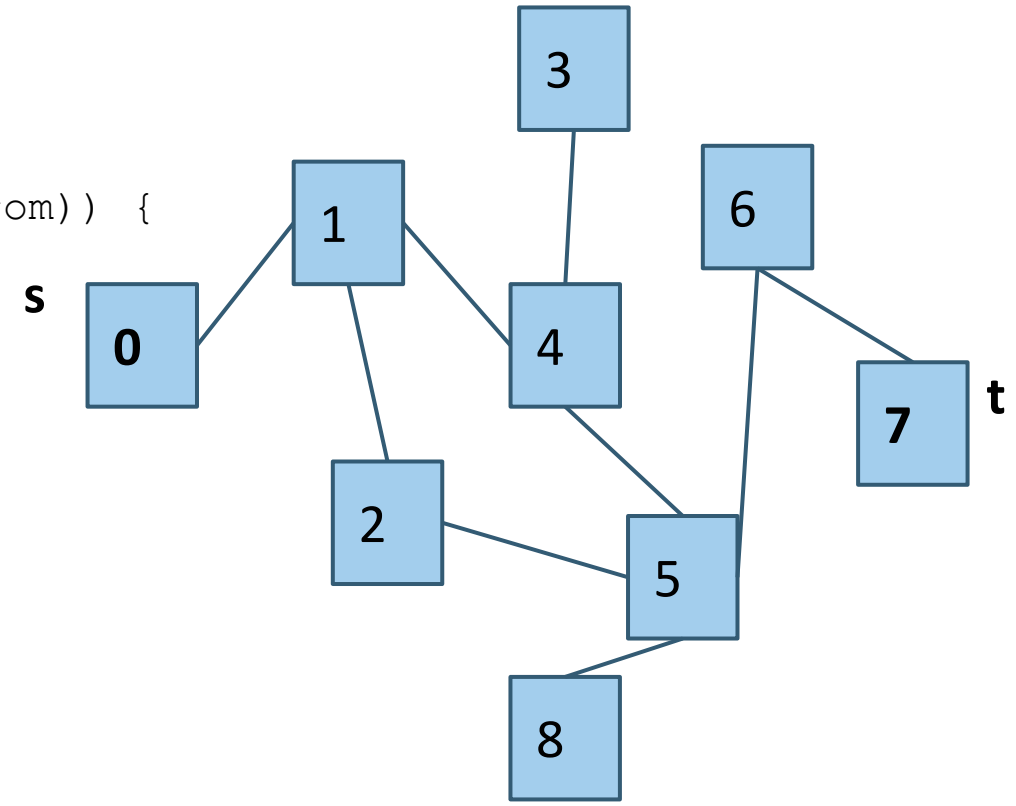
BFS pseudocode (some details not Java specific)

```
bfs(Graph graph, Vertex start) {
    // stores the remaining vertices to visit in the BFS
    Queue<Vertex> perimeter = new Queue<>();

    // stores the set of discovered vertices so we don't revisit them multiple times
    Set<Vertex> discovered = new Set<>();

    // kicking off our starting point by adding it to the perimeter
    perimeter.add(start);
    discovered.add(start);

    while (!perimeter.isEmpty()) {
        Vertex from = perimeter.remove();
        for (E edge : graph.outgoingEdgesFrom(from)) {
            Vertex to = edge.to();
            if (!discovered.contains(to)) {
                perimeter.add(to);
                discovered.add(to)
            }
        }
    }
```



BFS pseudocode (some details not Java specific)

```
// . . . this is the main loop/code for BFS
while (!perimeter.isEmpty()) {
    Vertex from = perimeter.remove();
    for (E edge : graph.outgoingEdgesFrom(from)) {
        Vertex to = edge.to();
        if (!discovered.contains(to)) {
            perimeter.add(to);
            discovered.add(to)
        }
    }
}
```



Perimeter queue:

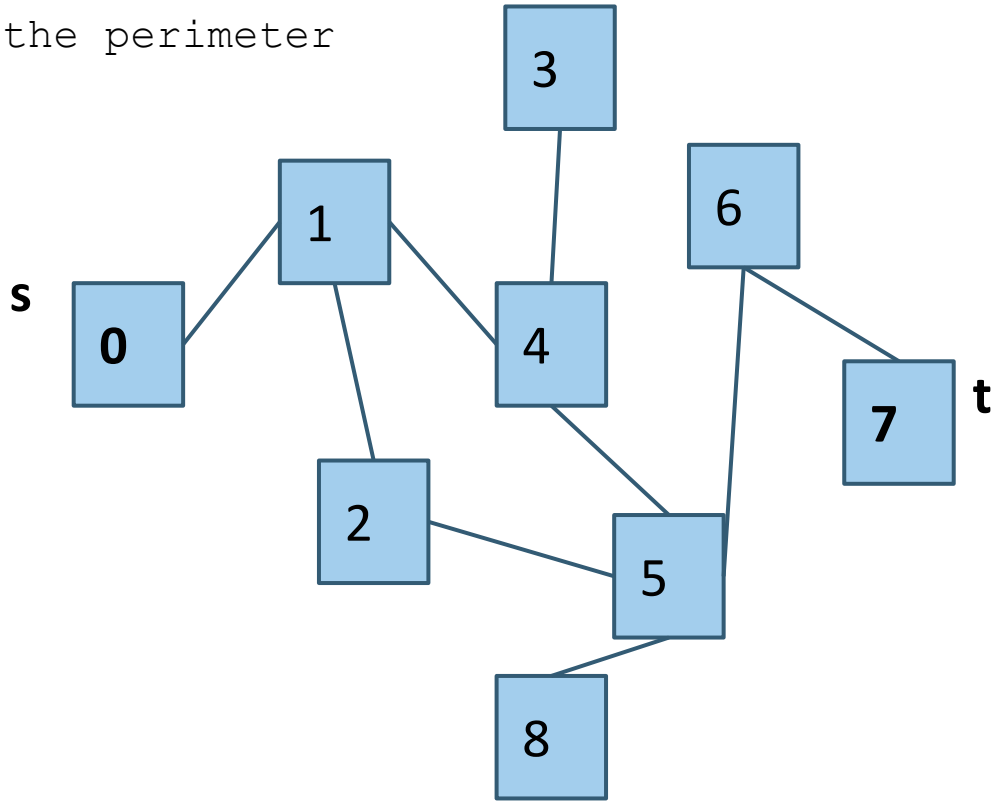
Discovered set:

Expected levels starting the BFS from 0:

- 0
- 1
- 2 4
- 3 5
- 6 8
- 7

DFS pseudocode (some details not Java specific)

```
dfs(Graph graph, Vertex start) {
    // stores the remaining vertices to visit in the DFS
    Stack<Vertex> perimeter = new Stack<>(); //the only change you need to make to do DFS!

    // stores the set of discovered vertices so we don't revisit them multiple times
    Set<Vertex> discovered = new Set<>();

    // kicking off our starting point by adding it to the perimeter
    perimeter.add(start);
    discovered.add(start);

    while (!perimeter.isEmpty()) {
        Vertex from = perimeter.remove();
        for (E edge : graph.outgoingEdgesFrom(from)) {
            Vertex to = edge.to();
            if (!discovered.contains(to)) {
                perimeter.add(to);
                discovered.add(to)
            }
        }
    }
```



Modifying BFS and DFS

BFS and DFS are like the for loops over arrays for graphs. They're super fundamental to so many ideas, but when they're by themselves they don't do anything. Consider the following code:

```
for (int i = 0; i < n; i++) {
    int x = arr[i];
}
```

```
while (!perimeter.isEmpty()) {
    Vertex from = perimeter.remove();
    for (E edge : graph.outgoingEdgesFrom(from)) {
        Vertex to = edge.to();
        if (!discovered.contains(to)) {
            perimeter.add(to, newDist);
            discovered.add(to)
        }
    }
}
```

We actually need to do something with the data for it to be useful!

A lot of times to solve basic graph problems (which show up in technical interviews at this level), and often the answer is that you just need to describe / implement BFS/DFS with a small modification for your specific problem.

Now back to the s-t path problem…

Modifying BFS for the s-t path problem

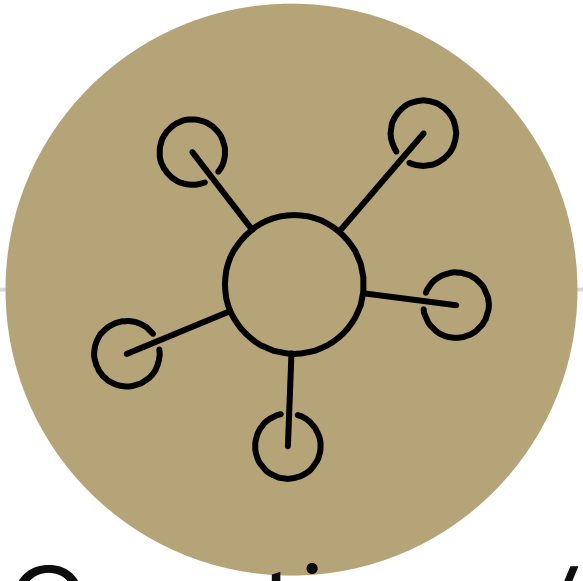
```
//. . . this is the main loop/code for BFS
while (!perimeter.isEmpty()) {
    Vertex from = perimeter.remove();
    for (E edge : graph.outgoingEdgesFrom(from)) {
        Vertex to = edge.to();
        if (!discovered.contains(to)) {
            perimeter.add(to);
            discovered.add(to)
        }
    }
}
```

```
// with modifications to return true if
// there is a path where s can reach t
while (!perimeter.isEmpty()) {
    Vertex from = perimeter.remove();
    if (from == t) {
        return true;
    }
    for (E edge : graph.outgoingEdgesFrom(from)
        Vertex to = edge.to();
        if (!discovered.contains(to)) {
            perimeter.add(to);
            discovered.add(to)
        }
    }
}
return false;
```

Small note: for this s-t problem, we didn't really need the power of BFS in particular, just some way of looping through the graph starting at a particular point and seeing everything it was connected to. So we could have just as easily used DFS.

There are plenty of unique applications of both, however, and we'll cover some of them in this course – for a more comprehensive list, feel free to google or check out resources like:

- https://www.geeksforgeeks.org/applications-of-breadth-first-traversal/
- https://www.geeksforgeeks.org/applications-of-depth-first-search/



Questions / clarifications on anything?

we covered:

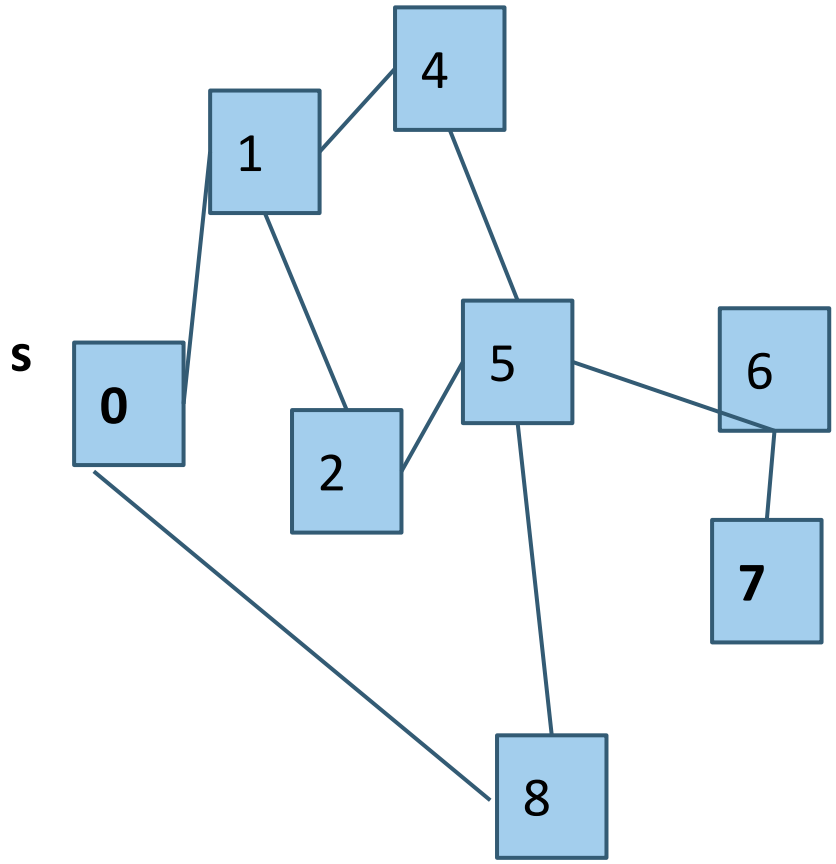
- s-t path problem
- BFS/DFS visually + high-level
- BFS/DFS pseudocode
- modifying BFS/DFS to solve s-t path problem

Roadmap for today

- review Wednesday intro to graphs key points
- graph problems
- s-t path problem
- detour: BFS/DFS
  - visually
  - pseudocode
  - modifications to solve problems (circling back to s-t path)
- shortest path problem (for unweighted graphs)

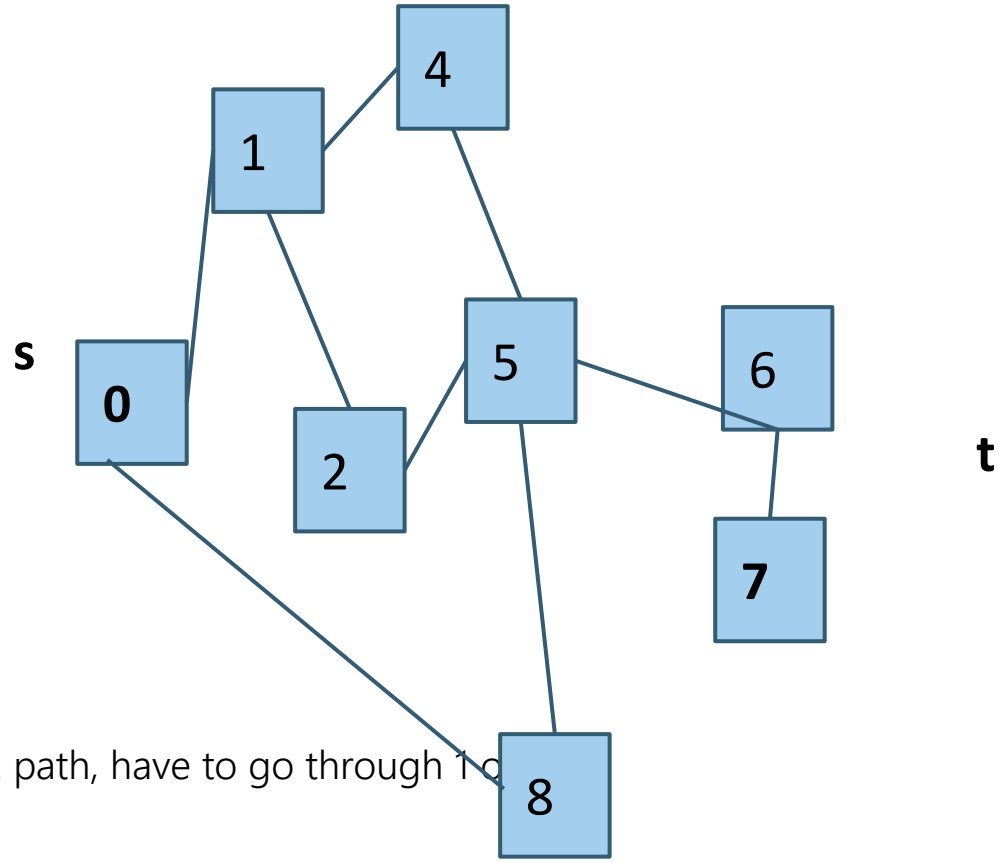
Shortest Path problem (unweighted graph)

- For the graph on the right, find **the shortest path (the path that has the fewest number of edges)** between the 0 node and the 5 node. Describe the path by describing each edge (i.e. (0, 1) edge).
- What's the answer? How did we get that as humans? How do we want to do it comprehensively defined in an algorithm?



Shortest Path problem (unweighted graph)

how do we find a shortest paths?



What's the shortest path from 0 to 0?
- Well….we're already there.

What's the shortest path from 0 to 1 or 8?
- Just go on the edge from 0

From 0 to 4 or 2 or 5?
- Can't get there directly from 0, if we want a length 2 path, have to go through 1 o

From 0 to 6?
- Can't get there directly from 0, if we want a length 3 path, have to go through 5.

Shortest Path problem (unweighted graph) key idea

To find the set of vertices at distance k, just find the set of vertices at distance k-1, and see if any of them have an outgoing edge to an undiscovered vertex. Basically, if we traverse level by level and we're checking all the nodes that show up at each level comprehensively (and only recording the earliest time they show up), when we find our target at level k, we can keep using the edge that led to it from the previous level to justify the shortest path.

Do we already know an algorithm that can help us traverse the graph level by level?

Yes! BFS! Let's modify it to fit our needs.

Changes from traversal BFS:

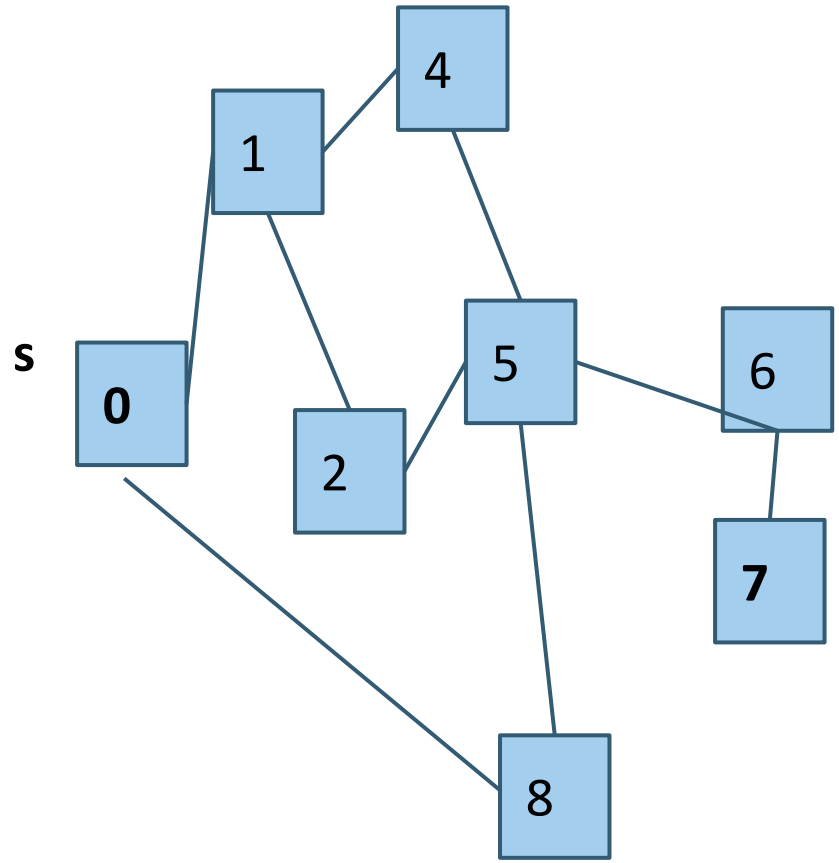
- Every node now will have an associated distance (for convenience)
- Every node V now will have an associated predecessor edge that is the edge that connects V on the shortest path from S to V. The edges that each of the nodes store are the final result.

```
perimeter.add(start);
discovered.add(start);
start.distance = 0;
while (!perimeter.isEmpty()) {
    Vertex from = perimeter.remove();
    for (E edge : graph.outgoingEdgesFrom(from)) {
        Vertex to = edge.to();
        if (!discovered.contains(to)) {
            to.distance = from.distance + 1;
            to.predecessorEdge = edge;
            perimeter.add(to);
            discovered.add(to)
        }
    }
}
```

Unweighted Graphs

Use BFS to find shortest paths in this graph.

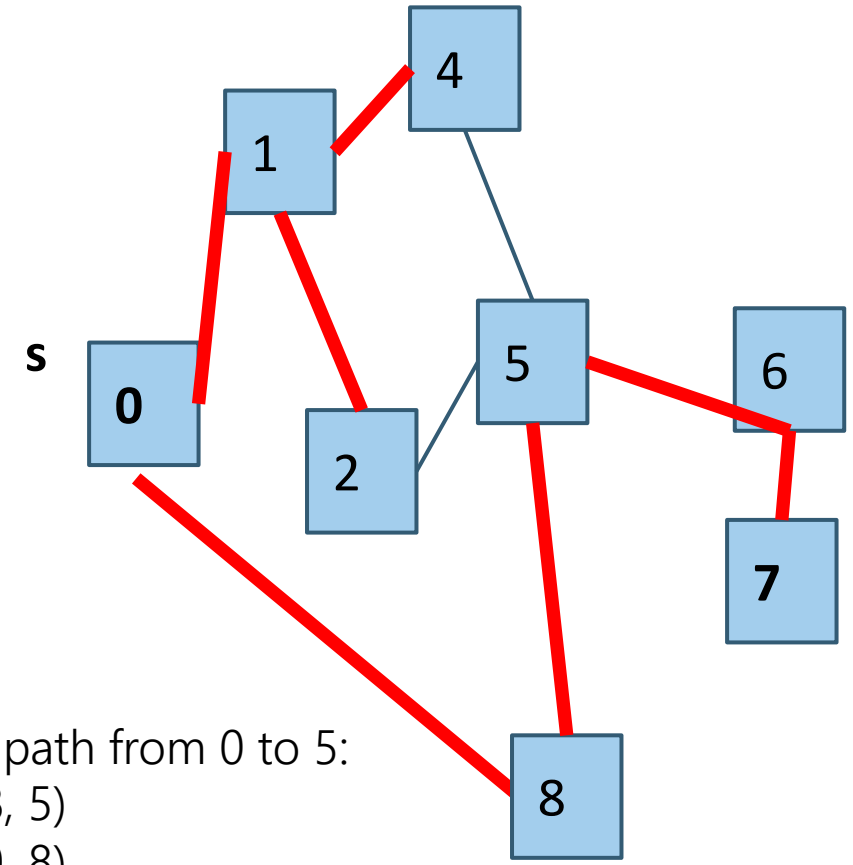
```
perimeter.add(start);
discovered.add(start);
start.distance = 0;
while (!perimeter.isEmpty()) {
    Vertex from = perimeter.remove();
    for (E edge : graph.outgoingEdgesFrom(from)) {
        Vertex to = edge.to();
        if (!discovered.contains(to)) {
            to.distance = from.distance + 1;
            to.predecessorEdge = edge;
            perimeter.add(to);
            discovered.add(to)
        }
    }
}
```



Unweighted Graphs

Use BFS to find shortest paths in this graph.

```
perimeter.add(start);
discovered.add(start);
start.distance = 0;
while (!perimeter.isEmpty()) {
    Vertex from = perimeter.remove();
    for (E edge : graph.outgoingEdgesFrom(from)) {
        Vertex to = edge.to();
        if (!discovered.contains(to)) {
            to.distance = from.distance + 1;
            to.predecessorEdge = edge;
            perimeter.add(to);
            discovered.add(to)
        }
    }
}
```



If trying to recall the best path from 0 to 5:
5's predecessor edge is (8, 5)
8's predecessor edge is (0, 8)
0 was the start vertex

Note: this BFS modification produces these edges, but there's extra work to figure out a specific path from a start / target

What about the target vertex?

Shortest Path Problem

Given: a directed graph G and vertices s,t
Find: the shortest path from s to t.

BFS didn't mention a target vertex…
It actually finds the distance from s to **every** other vertex. The resulting edges are called **the shortest path tree**.

All our shortest path algorithms have this property.
If you only care about one target, you can sometimes stop early (in `bfsShortestPaths`, when the target pops off the queue)

```
Map<V, E> bfsFindShortestPathsEdges(G graph, V start) {
    // stores the edge `E` that connects `V` in the shortest path from s to V
    Map<V, E> edgeToV = empty map

    // stores the shortest path length from `start` to `V`
    Map<V, Double> distToV = empty map


    Queue<V> perimeter = new Queue<>();
    Set<V> discovered = new Set<>();

    // setting up the shortest distance from start to start is just 0 with
    // no edge leading to it
    edgeTo.put(start, null);
    distTo.put(start, 0.0);

    perimeter.add(start);

    while (!perimeter.isEmpty()) {
        V from = perimeter.remove();
        for (E e : graph.outgoingEdgesFrom(from)) {
            V to = e.to();
            if (!discovered.contains(to)) {
                edgeTo.put(to, e);
                distTo.put(to, distTo(from) + 1);
                perimeter.add(to, newDist);
                discovered.add(to)
            }
        }
    }
    return edgeToV;
}
```

This is an alternative way to implement bfsShortestPaths that h an easier time accessing the actual paths / distances by using Maps