LEC 15

CSE 373

BFS, DFS, Shortest Paths

BEFORE WE START

If we model UW course prerequisites with a graph, what would it look like?

- Courses represented as Vertices or Edges?
- Undirected or Directed Graph?
- Cyclic or Acyclic?
- Weighted or Unweighted?
- Are there edge labels?
- Are there Parallel Edges?
- Are there Self Loops?

pollev.com/uwcse373

Instructor: Aaron Johnston

TAs: Timothy Akintilo, Brian Chan, Joyce Elauria, Eric Fan, Farrell Fileas, Melissa Hovik, Leona Kazi, Keanu Vestil, Siddharth Vaidyanathan, Howard Xiao

Animation-Heavy Slides Ahead!

Hello, PDF slide reader! How’s your day been?

This is a heads up that there’s an unusually high number of animations in this slide deck. If you’d like to watch DFS and BFS run step-by-step (& the state of the BFS queue at each step), consider downloading the PPTX files from our course website!

- Aaron

Announcements

- P3 due in 1 week on Wednesday, 8/05
  - Remember `changePriority` and `contains` are a chance for you to **extend** what we've seen in lecture – go beyond what's in the slides
  - Just get working first, then add extra data structure for that wonderful sub-linear efficiency!
- EX3 published this Friday, 7/31
  - Will focus on the graph problems we talk about this week
- Still a number of 1:1 slots available this week if you're interested in talking about applying this material after this class!
  - Jobs, internships, research, future classes, etc.

Learning Objectives

After this lecture, you should be able to...

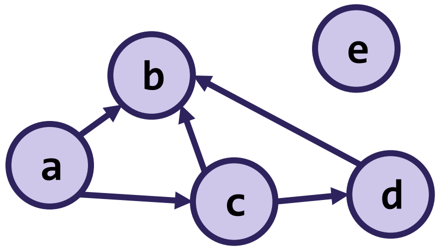
1. Review Compare various graph implementations (Adjacency List/Adjacency Matrix) and choose appropriately for a specific graph
2. Implement iterative BFS and DFS, and synthesize solutions to graph problems by modifying those algorithms
3. Describe the s-t Connectivity Problem, write code to solve it, and explain why we mark nodes as visited
4. Describe the Shortest Paths Problem, write code to solve it, and explain how we could use a shortest path tree to come up with the result

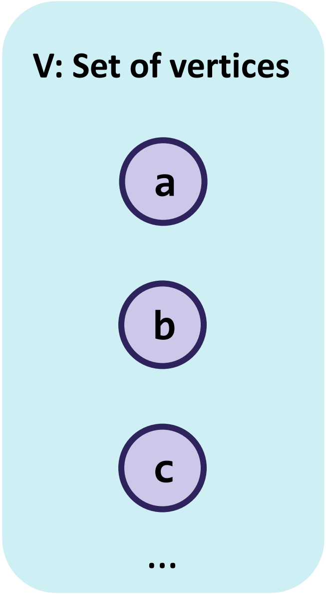
Lecture Outline

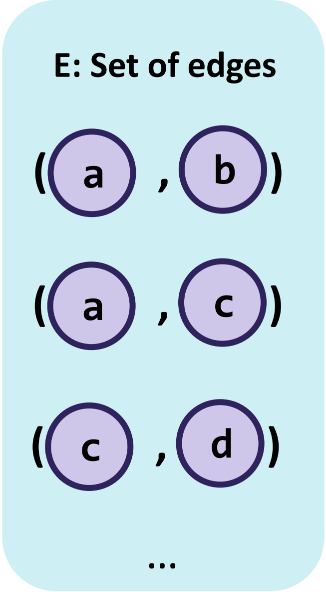
- *Review* Graph Implementations
- s-t Connectivity Problem
- BFS and DFS
- Shortest Paths Problem

Review 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





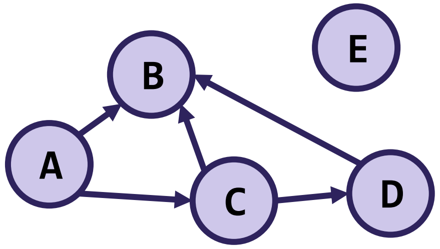


Review Adjacency Matrix

- A 2D array of with a cell for every *possible* edge
  - A row for each vertex (representing origins)
  - A column for each vertex (representing destinations)
  - The edges that exist in the graph have 1's in their cell

Add Edge	$\Theta(1)$
Remove Edge	$\Theta(1)$
Check if edge (u, v) exists	$\Theta(1)$
Get out-neighbors of u	$\Theta(n)$
Get in-neighbors of v	$\Theta(n)$
(Space Complexity)	$\Theta(n^2)$

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



destination

origin

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

Checking if an edge exists:

We want to look up `(u, v)` → Find row corresponding to u (origin) → Find column corresponding to v (destination)

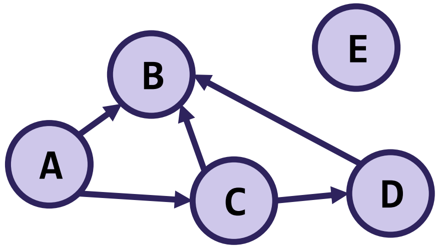
$\Theta(1)$ $\Theta(1)$

Review Adjacency List: Linked Lists

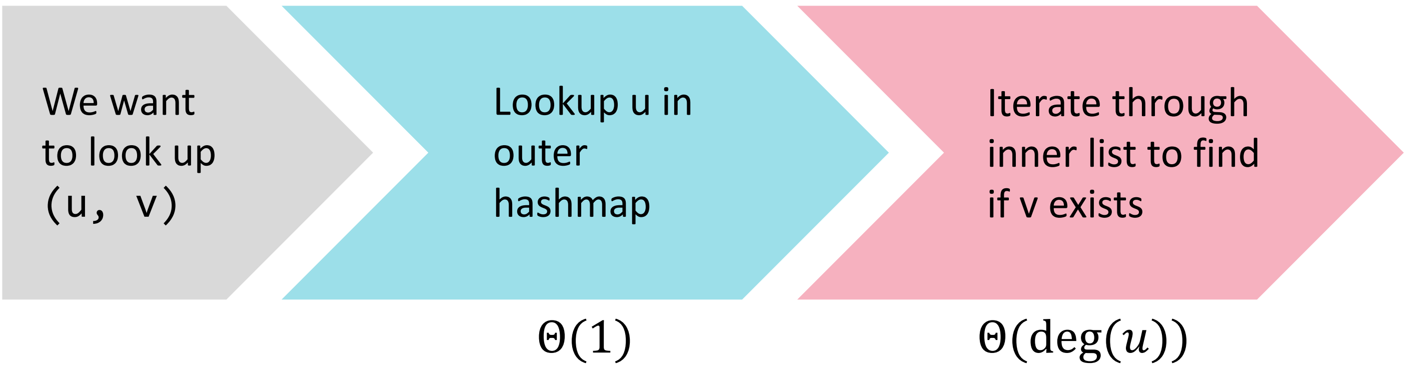
- Outer hash map, containing inner linked lists
  - Each key in the hash map is a vertex (representing origins)
  - Each value in the hash map is a linked list of vertices (representing destinations of edges from that origin)

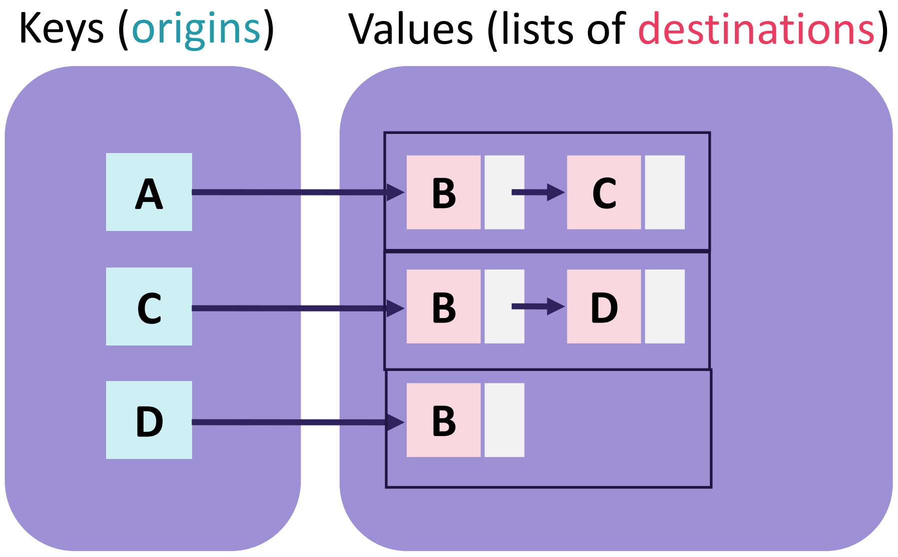
Add Edge	$\Theta(1)$
Remove Edge	$\Theta(\deg(u))$
Check if edge (u, v) exists	$\Theta(\deg(u))$
Get out-neighbors of u	$\Theta(\deg(u))$
Get in-neighbors of v	$\Theta(n+m)$
(Space Complexity)	$\Theta(n+m)$

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



Checking if an edge exists:





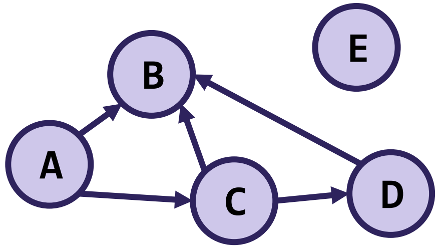
***Abstraction** of the Hash Map! **Buckets not shown**.*

Review Adjacency List: Hash Maps

- Outer hash map, containing inner hash maps
  - Each key in the outer hash map is a vertex (representing origins)
  - Each value is an inner hash map of vertices (representing destinations of edges from that origin)
  - Just presence of key in the inner hash map means that edge exists, but if you wanted to store labels on edges, you would put them as the values of the inner hash map

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)

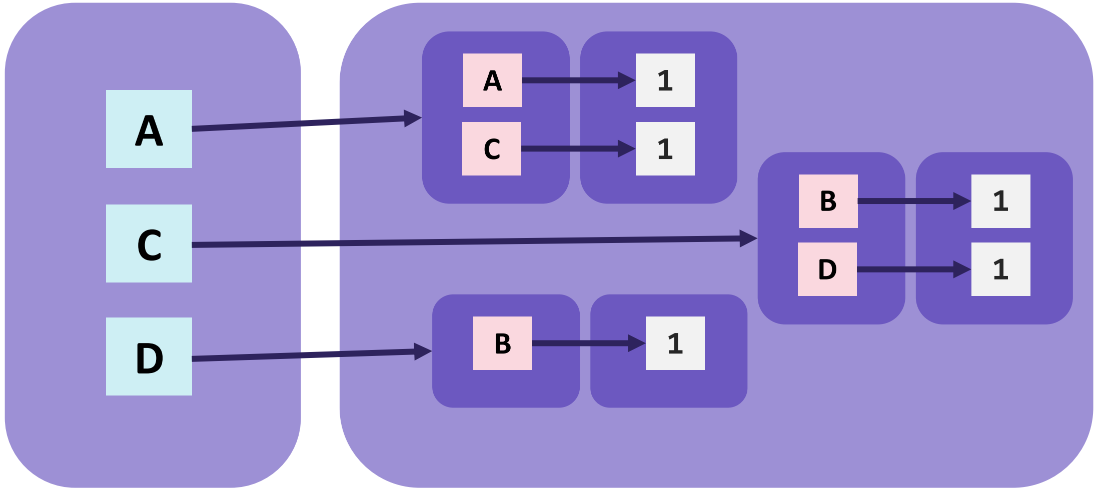


Checking if an edge exists:

We want to look up (u, v) → Lookup u in outer hashmap → Lookup v in inner hashmap

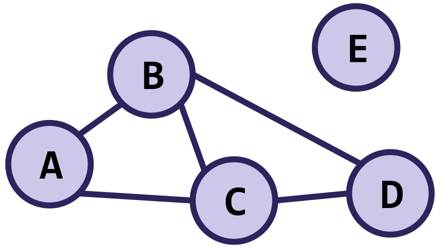
$\Theta(1)$ $\Theta(1)$

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



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

Adapting for Undirected Graphs



Adjacency Matrix

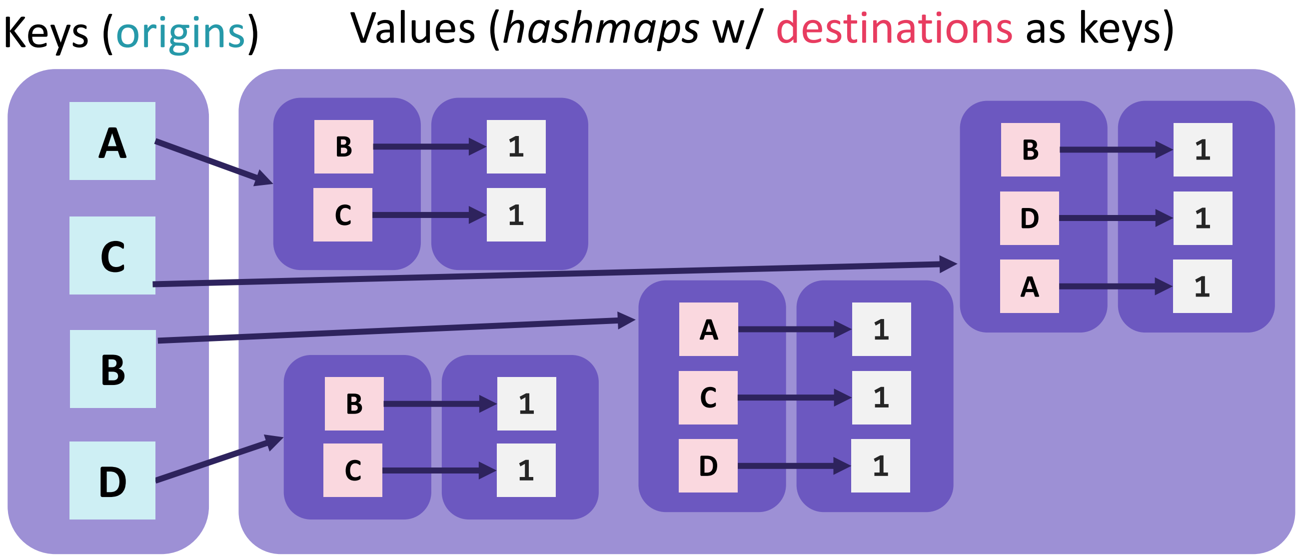
Store each edge as both directions (makes the matrix symmetrical)

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)



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

Tradeoffs

- Adjacency Matrices take more space, and have slower $\Theta()$ bounds, 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.
  - 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)

Lecture Outline

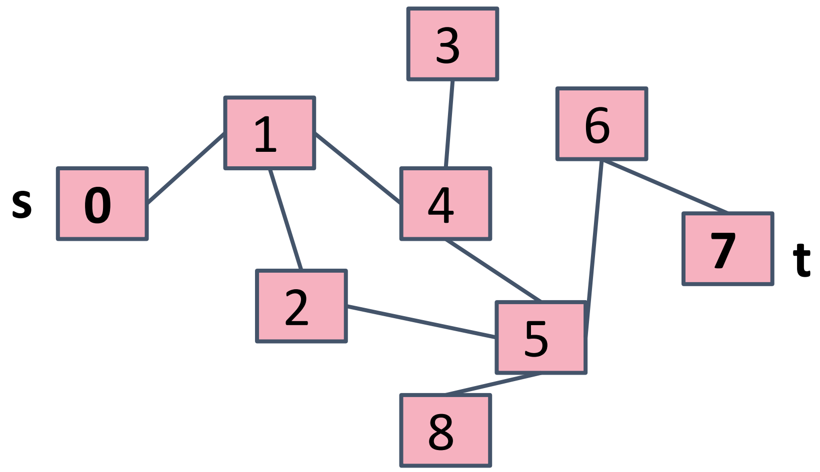
- ***Review*** Graph Implementations
- **s-t Connectivity Problem**
- BFS and DFS
- Shortest Paths Problem

s-t Connectivity Problem

s-t Connectivity Problem

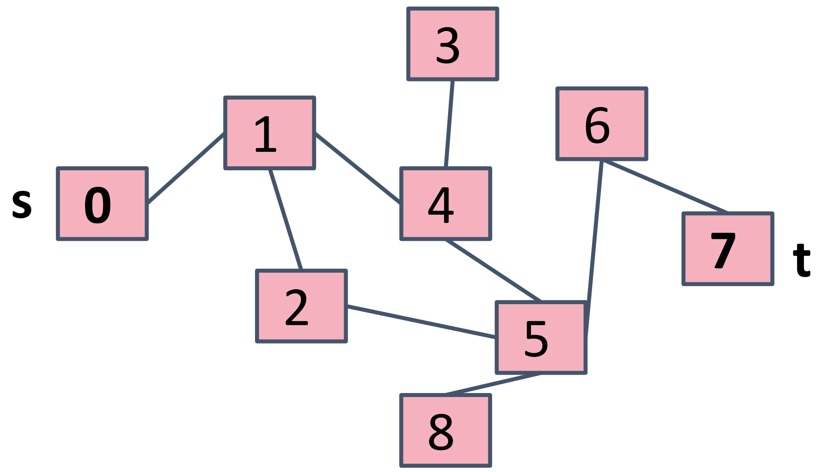
Given source vertex **s** and a target vertex **t**, does there exist a path between **s** and **t**?

- 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

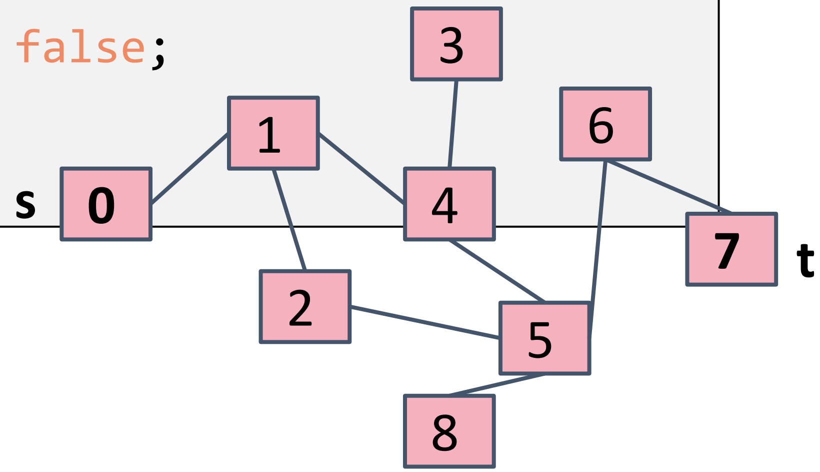
```
connected(Vertex s, Vertex t) {
  if (s == t) {
    return true;
  } else {
    for (Vertex n : s.neighbors) {
      if (connected(n, t)) {
        return true;
      }
    }
    return false;
  }
}
```

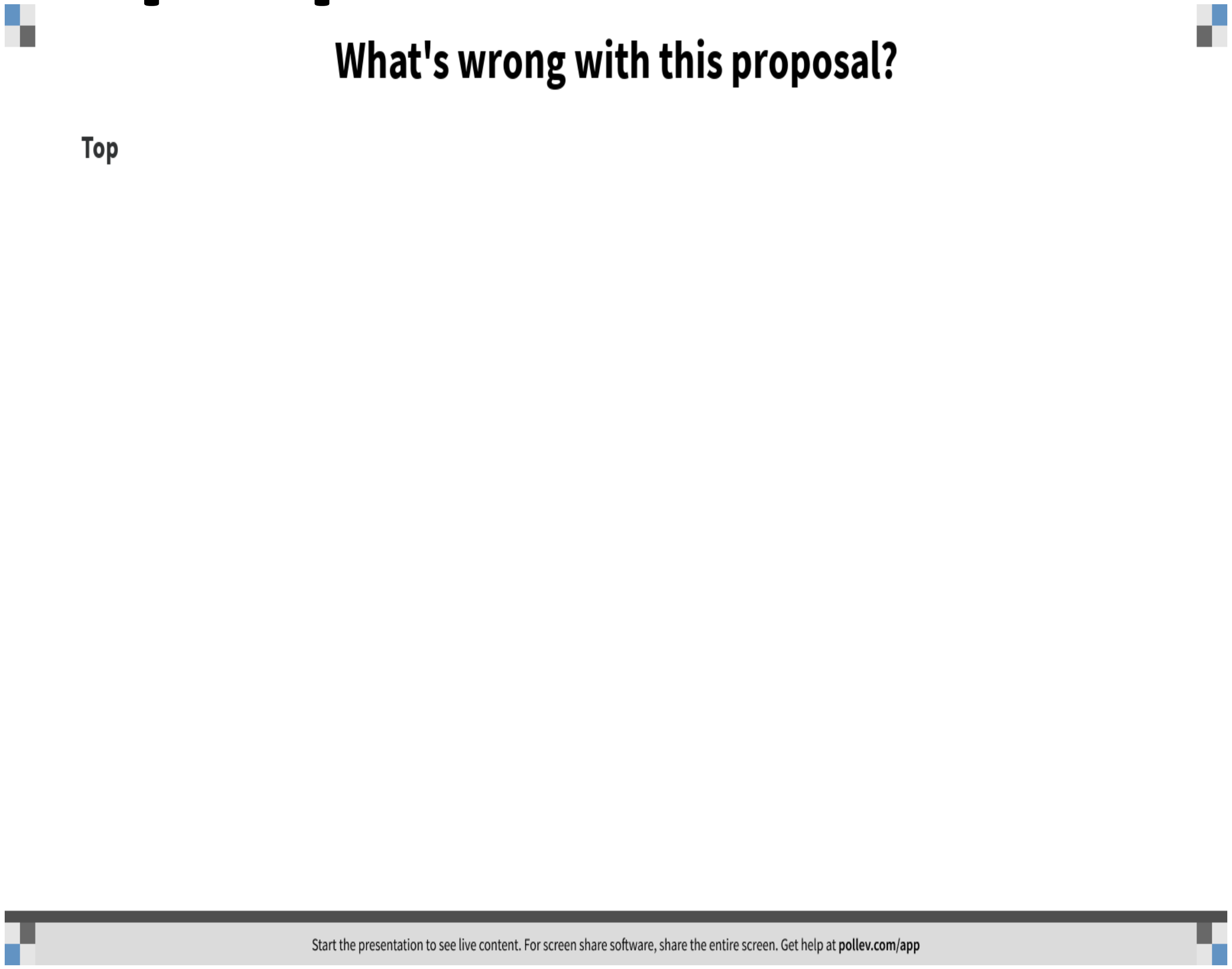


Poll Everywhere pollev.com/uwcse373

What’s wrong with this proposal?

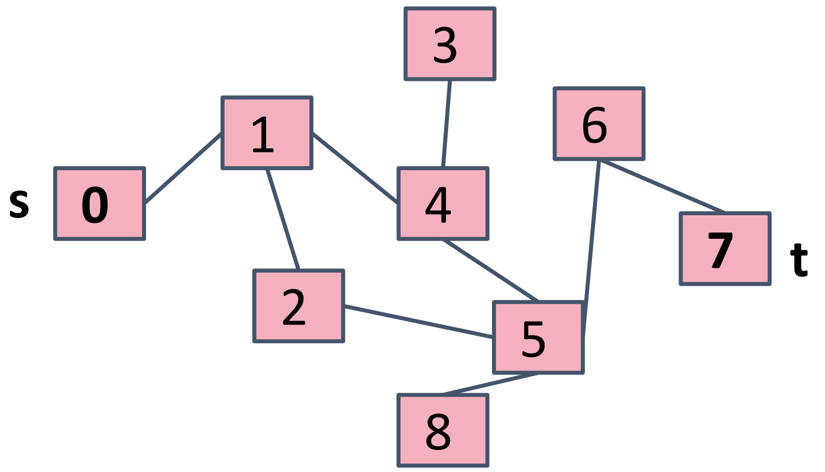
```
connected(Vertex s, Vertex t) {
  if (s == t) {
    return true;
  } else {
    for (Vertex n : s.neighbors) {
      if (connected(n, t)) {
        return true;
      }
    }
    return false;
  }
}
```





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;
  }
}
```

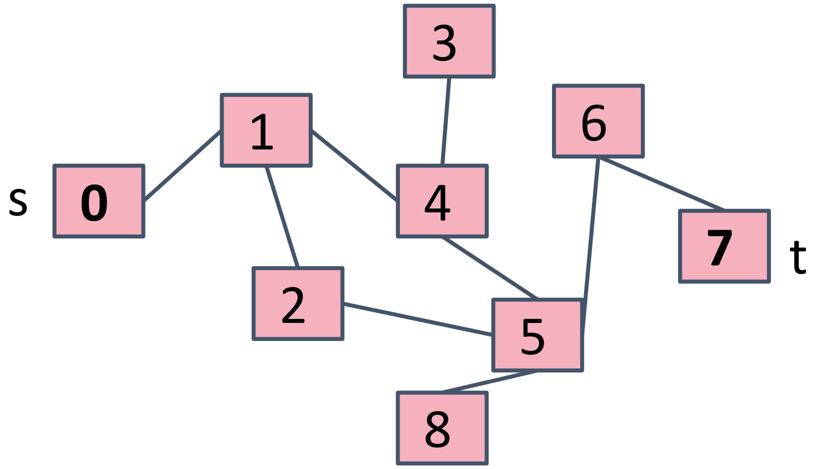


Does 0 == 7? No; if(connected(1, 7) return true;
Does 1 == 7? No; if(connected(0, 7) return true;
Does 0 == 7?

s-t Connectivity Problem: Better Solution

- Solution: Mark each node as visited!

```
connected(Vertex s, Vertex t) {
  Set<Vertex> visited;  // assume global
  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;
  }
}
```



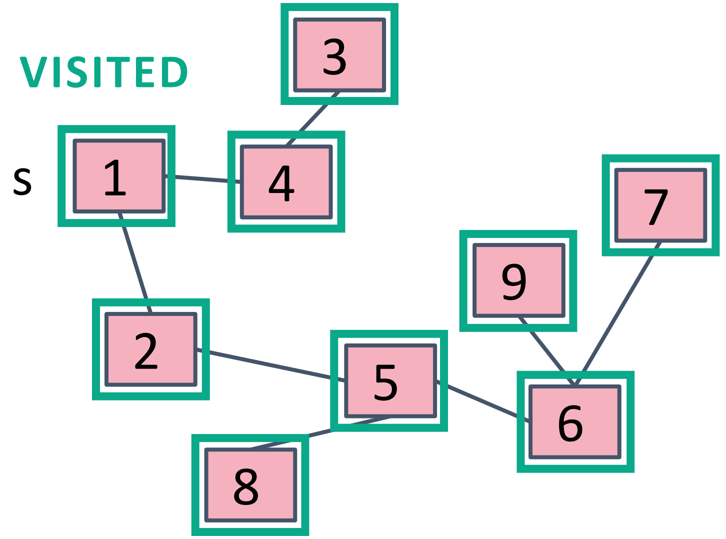
This general approach for crawling through a graph is going to be the basis for a LOT of algorithms!

Recursive Depth-First Search (DFS)

- What order does this algorithm use to visit nodes?
  - Assume order of `s.neighbors` is arbitrary!

```
connected(Vertex s, Vertex t) {
  Set<Vertex> visited; // assume global
  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;
  }
}
```

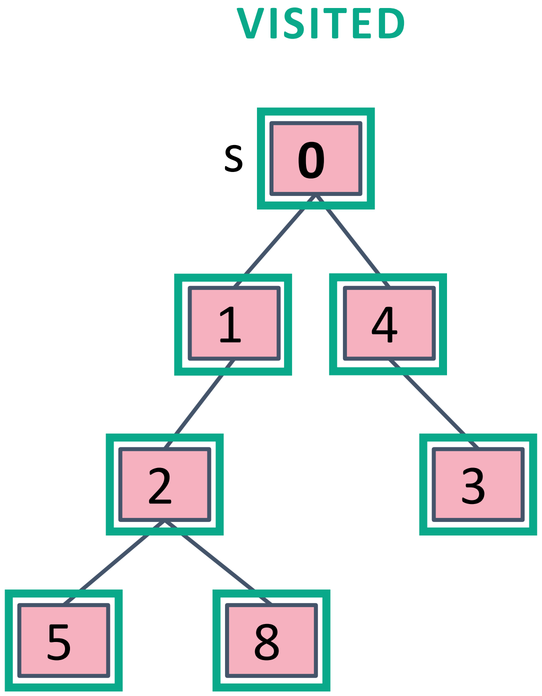
- It will explore one option "all the way down" before coming back to try other options
  - Many possible orderings: {0, 1, 2, 5, 6, 9, 7, 8, 4, 3} or {0, 1, 4, 3, 2, 5, 8, 6, 7, 9} both possible
- We call this approach a **depth-first search (DFS)**



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

```
connected(Vertex s, Vertex t) {
  Set<Vertex> visited; // assume global
  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;
  }
}
```



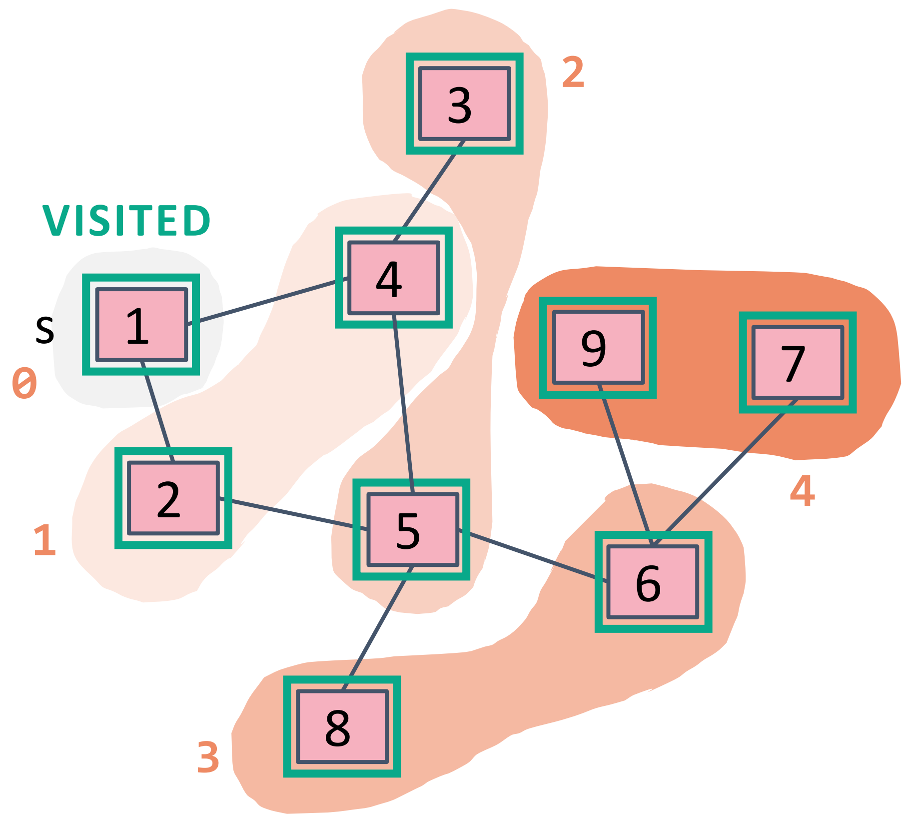
- ***CSE 143 Review*** traversing a binary tree depth-first has 3 options:
  - Pre-order: visit node before its children
  - In-order: visit node between its children
  - Post-order: visit node after its children
- The difference between these orderings is **when we "process" the root** – all are DFS!

Lecture Outline

- *Review* Graph Implementations
- s-t Connectivity Problem
- **BFS and DFS**
- Shortest Paths Problem

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

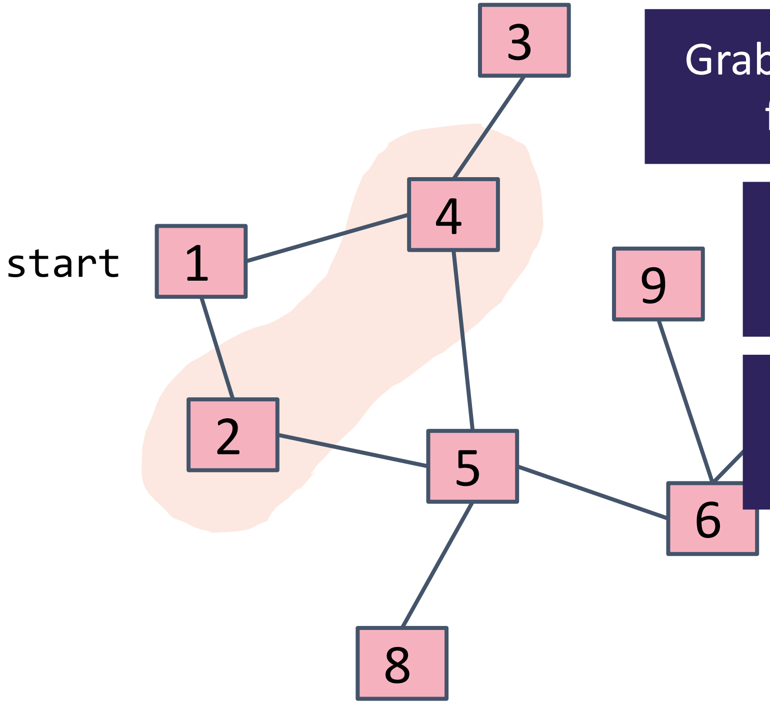
Our extra data structure! Will keep track of “outer edge” of nodes still to explore

Kick off the algorithm by adding start to perimeter

Grab one element at a time from the perimeter

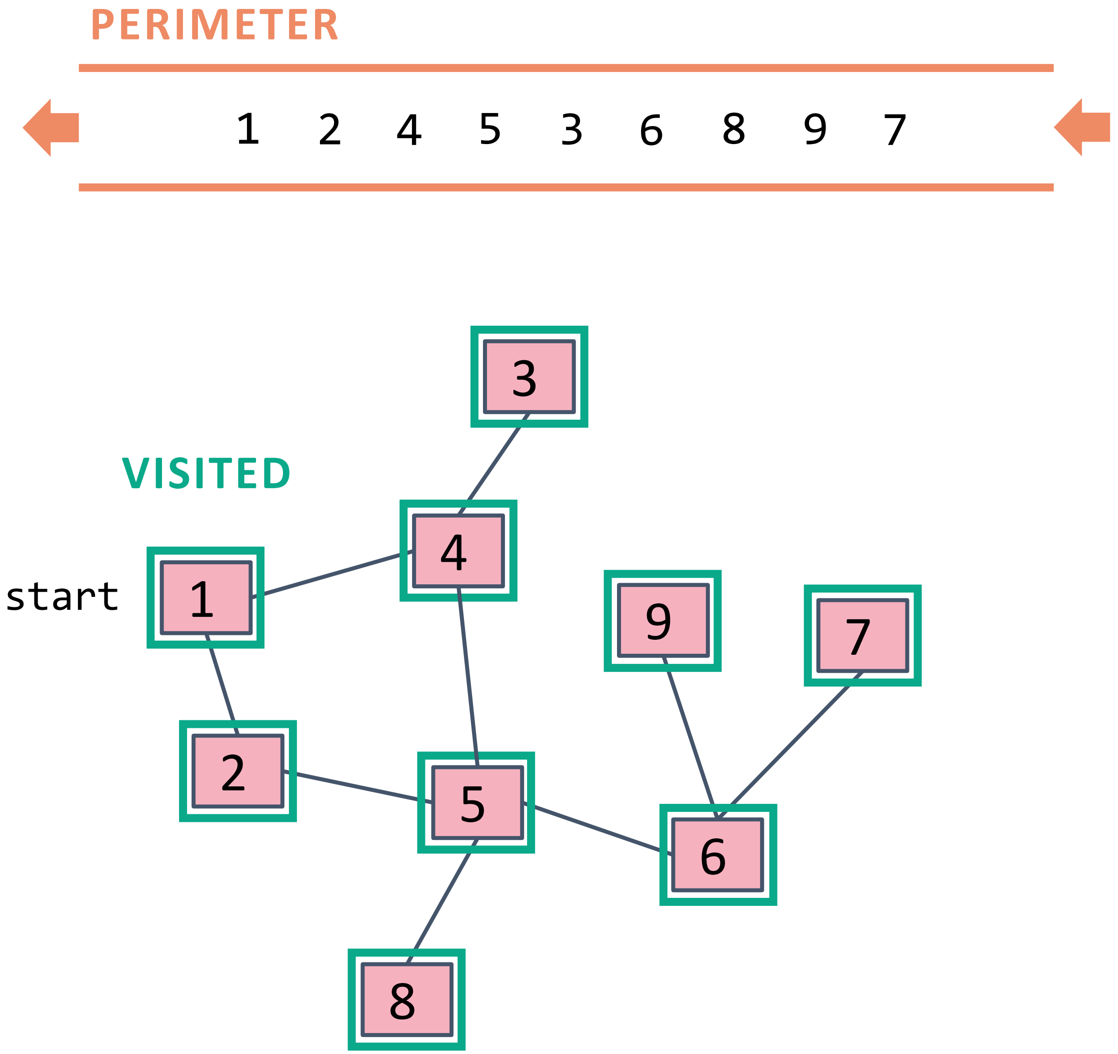
Look at all that element’s children

Add new ones to perimeter!



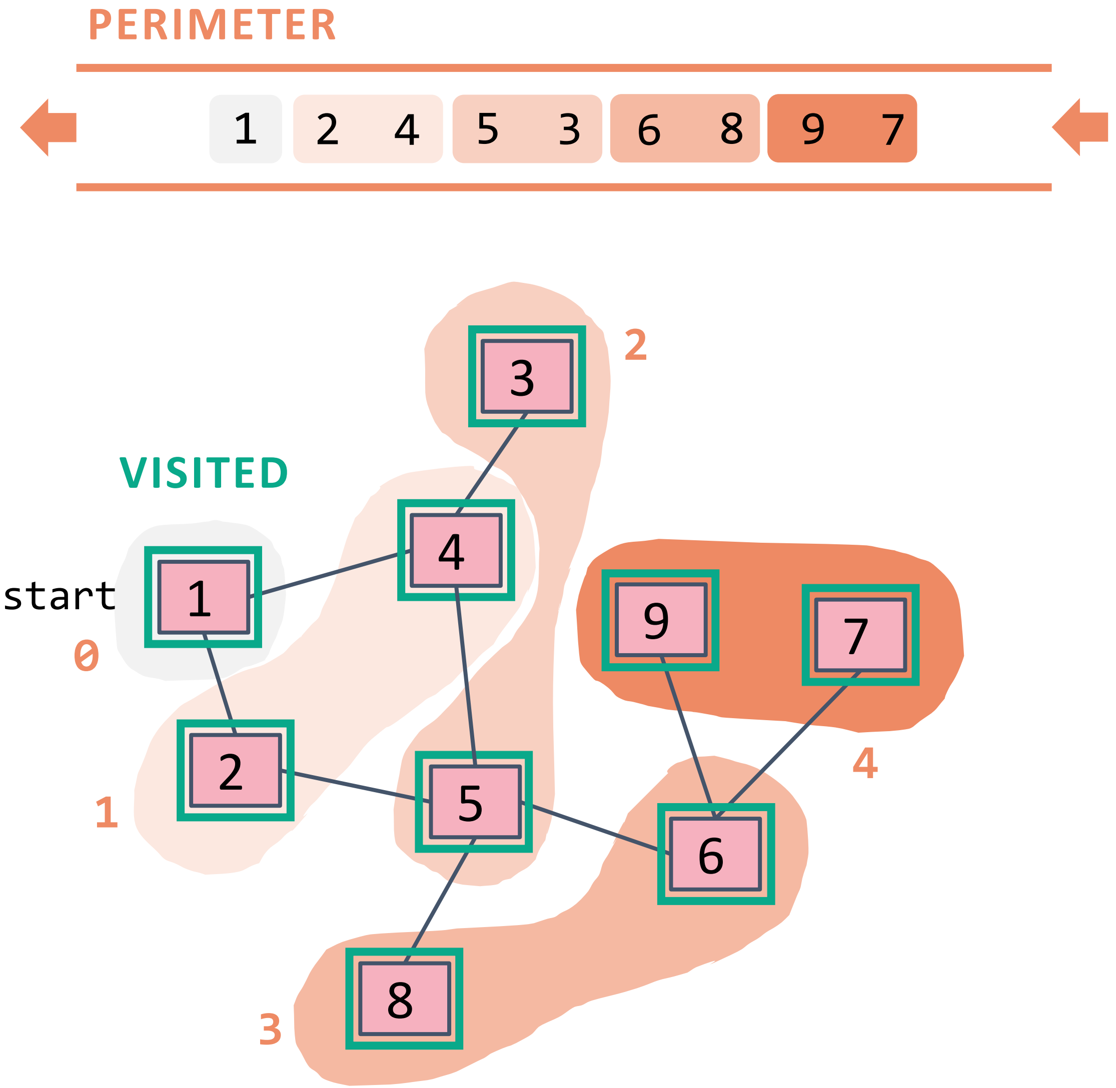

```
bfs(Graph graph, Vertex start) {
  Queue<Vertex> perimeter = new Queue<>();
  Set<Vertex> visited = new Set<>();

  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);
    }
  }
}
}
```

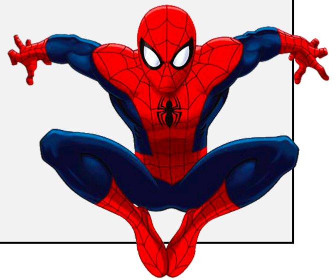
BFS's Evil Twin: DFS!

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);
      }
    }
  }
}
```

I think this is Spiderman's evil twin (?)
I've never seen the movies and… there's only so many Spiderman wikis I can justify reading during lecture prep

```
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**!

DFS

(Recursive)

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

BFS

(Iterative)

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

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.

```
bfs(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;
}
```

Lecture Outline

- *Review* Graph Implementations
- s-t Connectivity Problem
- BFS and DFS
- **Shortest Paths Problem**

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?

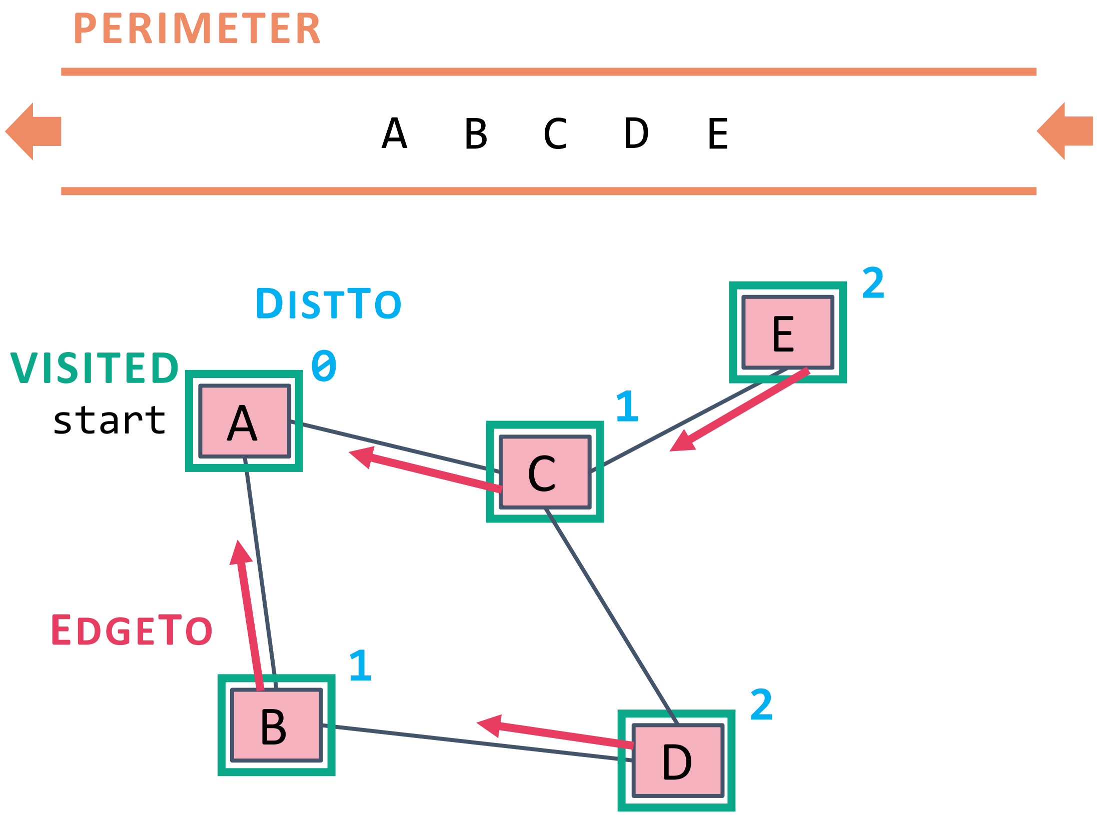
- This is a little harder, but still totally doable! We just need a way to keep track of how far each node is from the start.
  - Sounds like a job for?
    - BFS!

```
  ...
  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

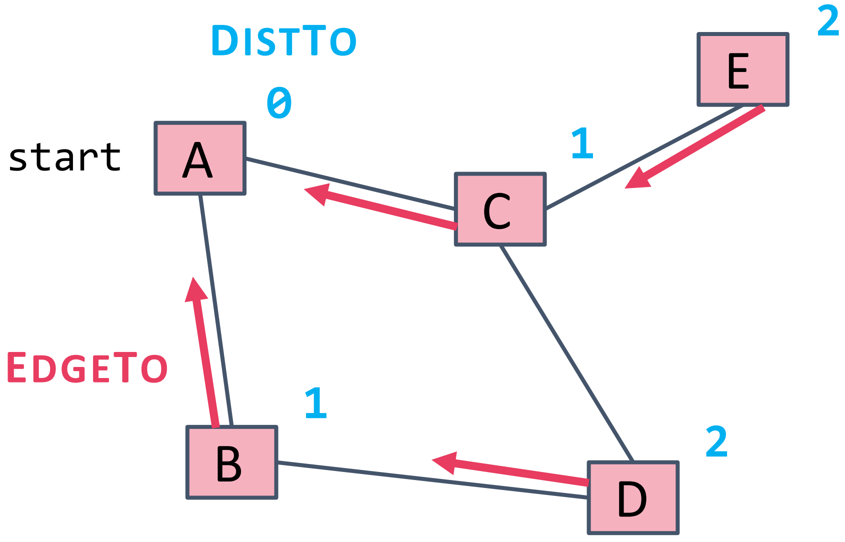



```
  ...
  Map<Vertex, Edge> edgeTo = ...
  Map<Vertex, Double> distTo = ...

  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 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

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

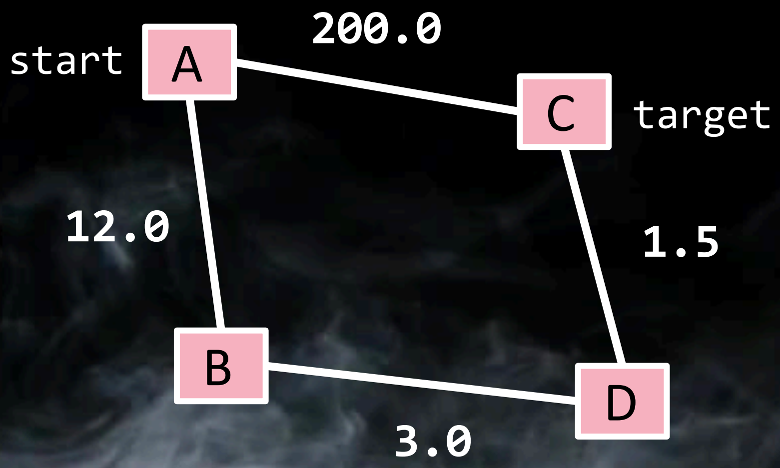
Given source vertex **s** and a target vertex **t**, how long is the shortest path from **s** to **t**? What edges make up that path?

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?