Lecture 16: Intro to Graphs

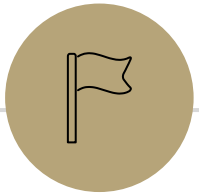
CSE 373: Data Structures and Algorithms

Administrivia

- Midterm is out – due this Friday by 11:59pm NO LATE ASSIGNMENTS
- Clarifications:
  - choose from the given lists of ADTs and data structures
  - Internal array of hash = location of buckets
  - 8.2 is a generic hash table implementation
  - apologies on bad parameters on code modeling, assume all calls get passed max as well

Midterm survey due TONIGHT

- if 90% of class on both - +1 ec point for all
- only at 60% response rate ☹



Introduction to Graphs

Inter-data Relationships

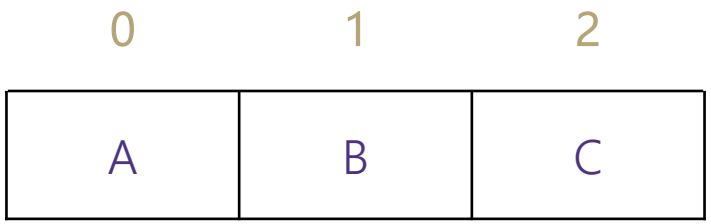
Arrays

Categorically associated

Sometimes ordered

Typically independent

Elements only store pure data, no connection info



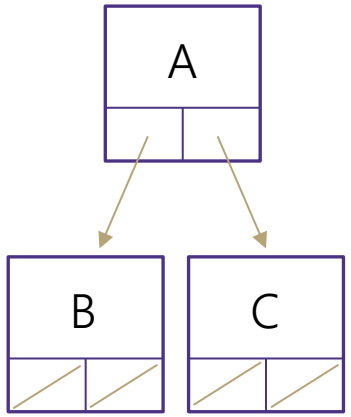
Trees

Directional Relationships

Ordered for easy access

Limited connections

Elements store data and connection info



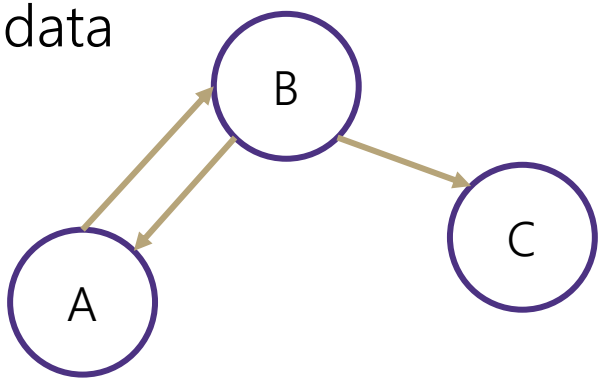
Graphs

Multiple relationship connections

Relationships dictate structure

Connection freedom!

Both elements and connections can store data

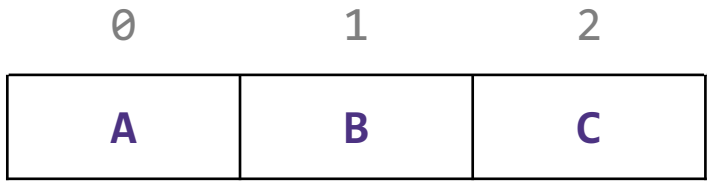


Inter-data Relationships

Arrays

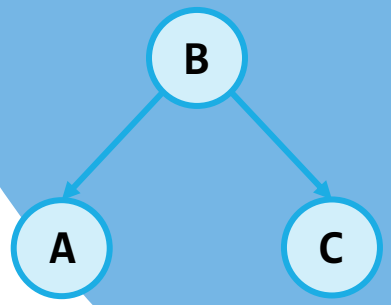
Elements only store pure data, no connection info

Only relationship between data is order



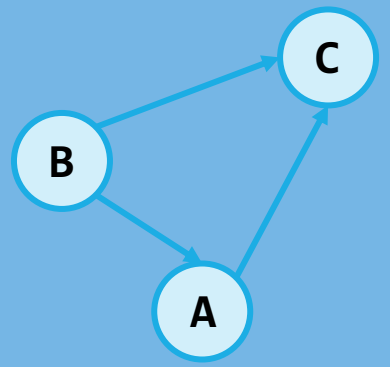
Trees

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



Graphs

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



Graphs

Everything is graphs.

Most things we've studied this quarter can be represented by graphs.

- BSTs are graphs
- Linked lists? Graphs.
- Heaps? Also can be represented as graphs.
- Those trees we drew in the tree method? Graphs.

But it's not just data structures that we've discussed…

- Google Maps database? Graph.
- Facebook? They have a "graph search" team. Because it's a graph
- Gitlab's history of a repository? Graph.
- Those pictures of prerequisites in your program? Graphs.
- Family tree? That's a graph

Applications

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

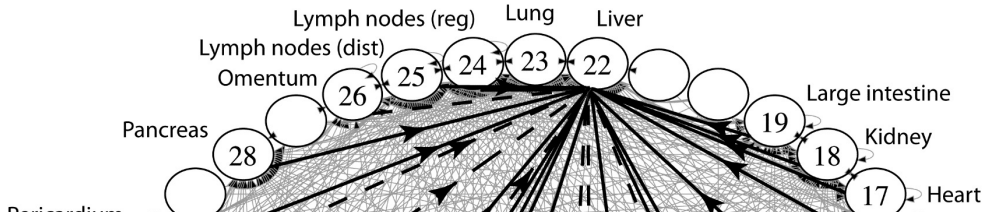
Relationships
- Social media graphs
  - Vertices are accounts, edges are follower relationships
- Code bases
  - Vertices are classes, edges are usage

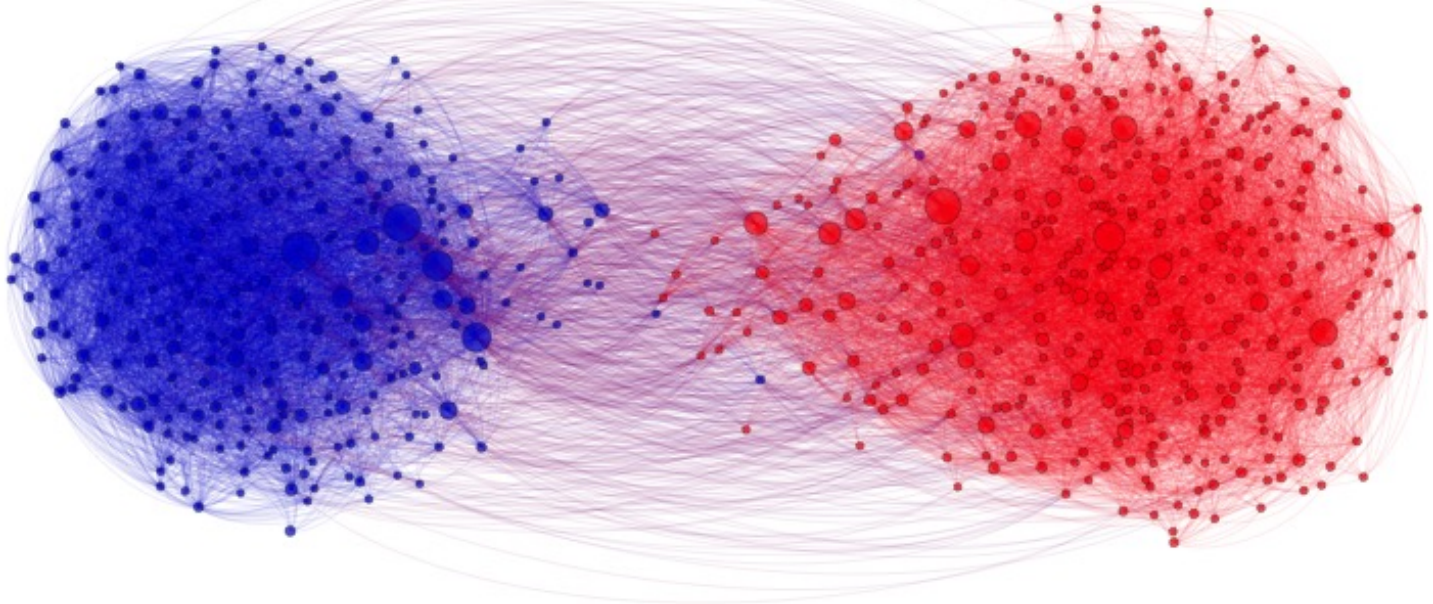
Influence
- Biology
  - Vertices are cancer cell destinations, edges are migration paths

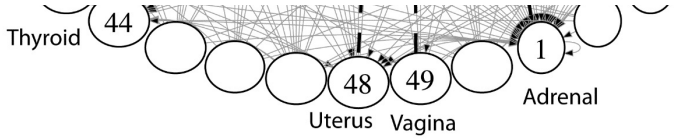
Related topics
- Web Page Ranking
  - Vertices are web pages, edges are hyperlinks
- Wikipedia
  - Vertices are articles, edges are links

SO MANY MORREEEE

www.allthingsgraphed.com







Graph: Formal Definition

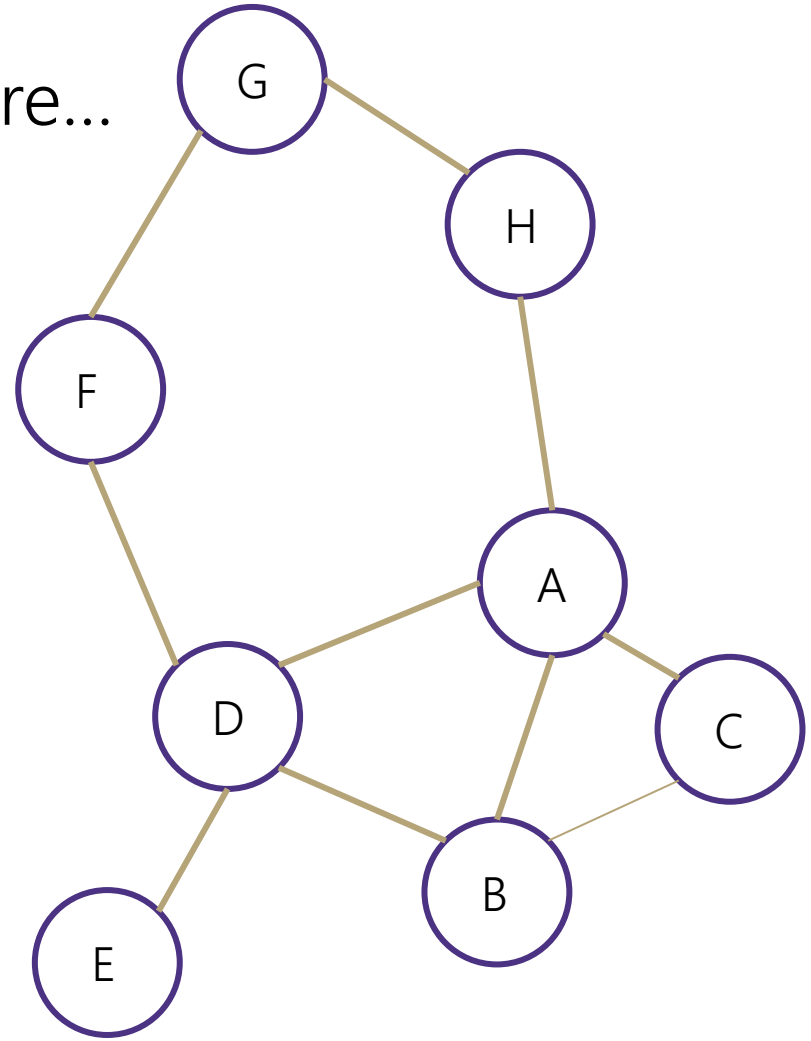
A graph is defined by a pair of sets G = (V, E) where…

- V is a set of vertices
  - A vertex or "node" is a data entity

V = { A, B, C, D, E, F, G, H }

- E is a set of edges
  - An edge is a connection between two vertices

E = { (A, B), (A, C), (A, D), (A, H),
(C, B), (B, D), (D, E), (D, F),
(F, G), (G, H)}



Graph Vocabulary

Graph Direction

- **Undirected graph** – edges have no direction and are two-way

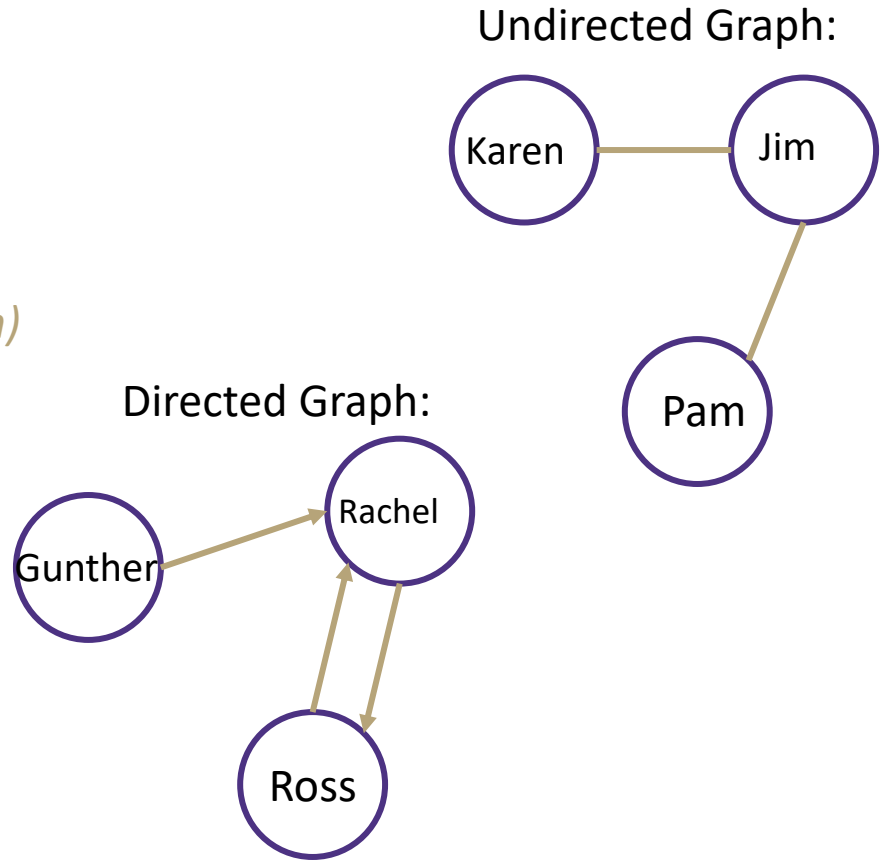
  V = { Karen, Jim, Pam }

  E = { (Jim, Pam), (Jim, Karen) } *inferred (Karen, Jim) and (Pam, Jim)*

- **Directed graphs** – edges have direction and are thus one-way

  V = { Gunther, Rachel, Ross }

  E = { (Gunther, Rachel), (Rachel, Ross), (Ross, Rachel) }



Degree of a Vertex

- **Degree** – the number of edges connected to that vertex

  Karen : 1, Jim : 1, Pam : 1

- **In-degree** – the number of directed edges that point to a vertex

  Gunther : 0, Rachel : 2, Ross : 1

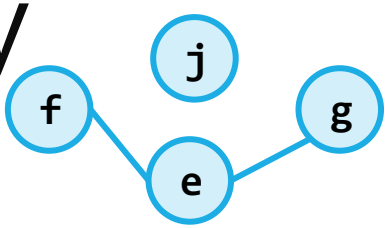
- **Out-degree** – the number of directed edges that start at a vertex

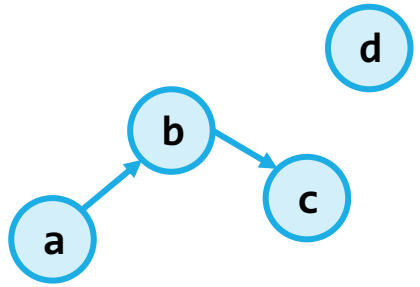
  Gunther : 1, Rachel : 1, Ross : 1

More More Graph Terminology

Two vertices are connected if there is a path between them

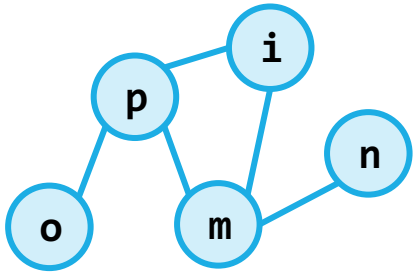
- If all the vertices are connected, we say the graph is connected
- The number of edges leaving a vertex is its degree

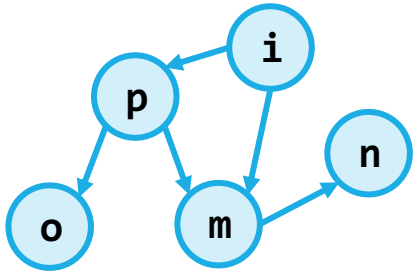




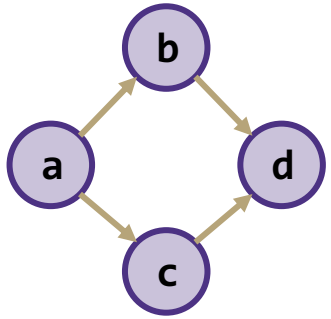
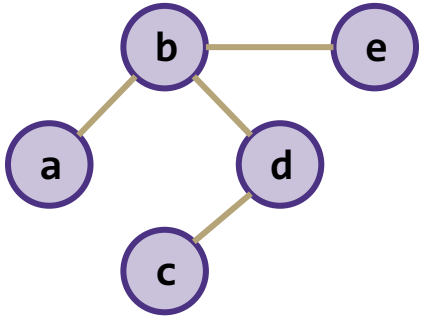
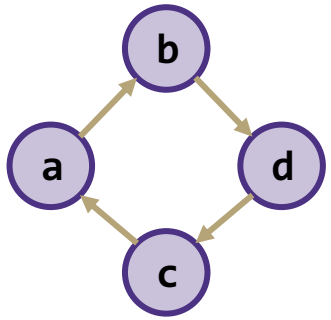
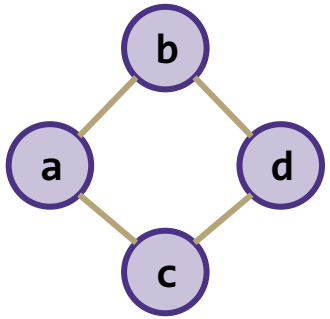
A path is a sequence of vertices connected by edges

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



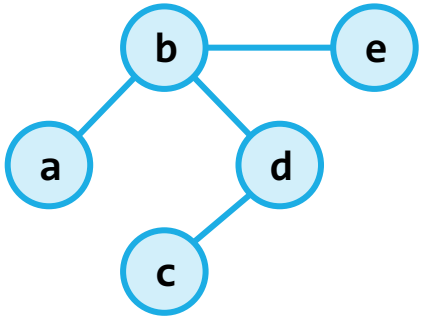


Directed vs Undirected; Acyclic vs Cyclic

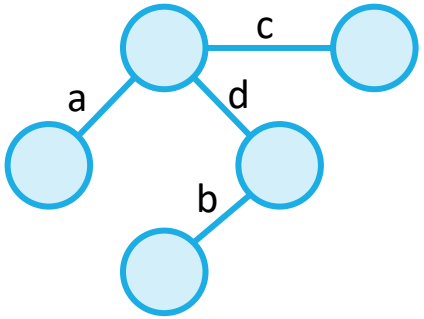
	Directed:	Undirected:
Acyclic:		
Cyclic:		

Labeled and Weighted Graphs

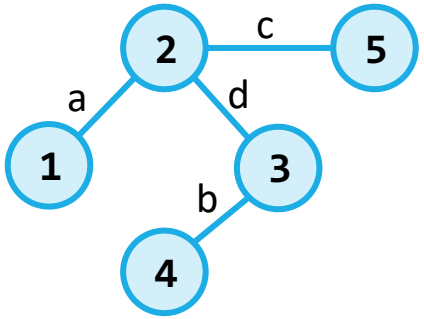
Vertex Labels



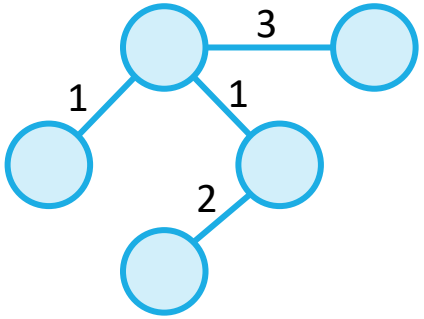
Edge Labels



Vertex & Edge Labels



Numeric Edge Labels
(**Edge Weights**)



Some examples

For each of the following think about what you should choose for vertices and edges.

The internet
- **Vertices**: webpages. **Edges** from a to b if a has a hyperlink to b.
- Directed, since hyperlinks go in one direction

Family tree
- **Vertices**: people. **Edges**: relationships
- Undirected, bidirectional relationships

Input data for the "6 Degrees of Kevin Bacon" game
- **Vertices**: actors. **Edges**: movies
- Undirected, a both actor would need to be in the movie for the edge to be added

Course Prerequisites
- **Vertices**: courses. **Edge**: from a to b if a is a prereq for b.
- Directed, since one course comes before the other

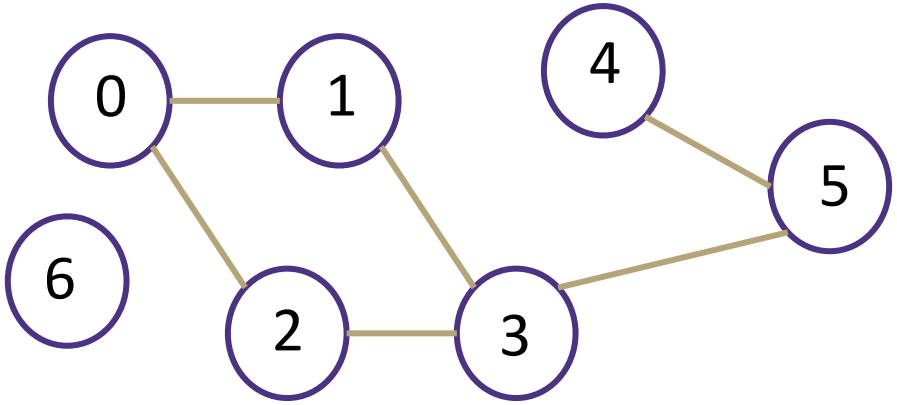
Ways to walk between UW buildings
- **Vertices**: buildings. **Edges**: A street name or walkway that connects 2 buildings
- Undirected, since each route can be walked both ways

Multi-Variable Analysis

- So far, we thought of everything as being in terms of some single argument "n" (sometimes its own parameter, other times a size)
  - But there's no reason we can't do reasoning in terms of multiple inputs!
- Why multi-variable?
  - Remember, algorithmic analysis is just a tool to help us understand code. Sometimes, it helps our understanding more to build a Oh/Omega/Theta bound for multiple factors, rather than handling those factors in case analysis.

- With graphs, we usually do our reasoning in terms of:
  - n (or |V|): total number of vertices (sometimes just call it V)
  - m (or |E|): total number of edges (sometimes just call it E)
  - deg(u): degree of node u (how many outgoing edges it has)

Adjacency Matrix



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

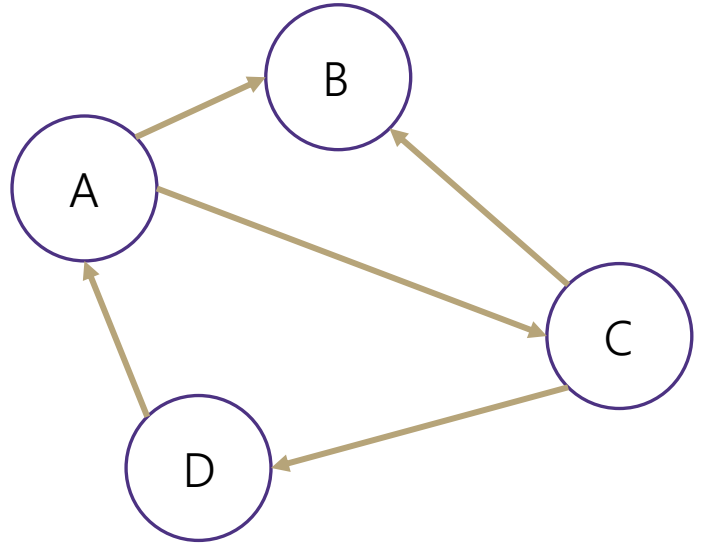
Worst-case Time Complexity (|V| = n, |E| = m):

- Add Edge: $\Theta(1)$
- 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

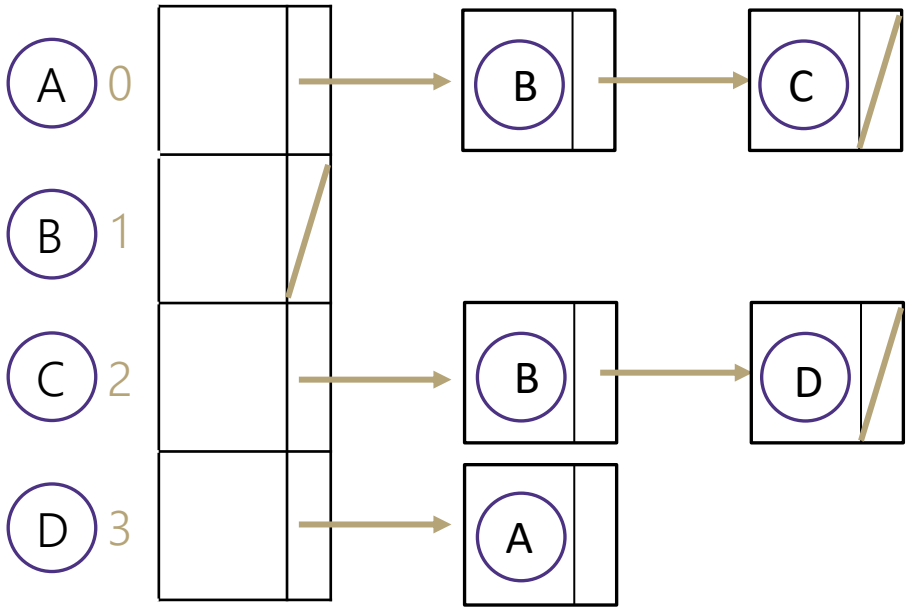
If (x,y) ∈ E then add y to the set associated with the key x

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):

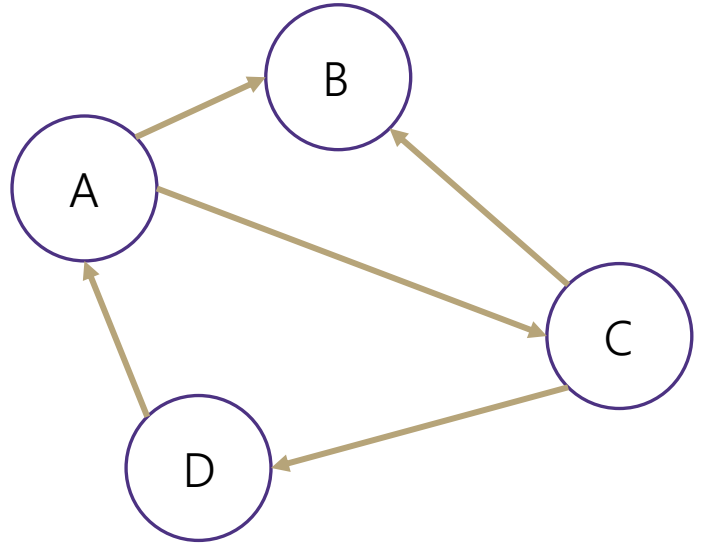
- Add Edge: $\boldsymbol{\Theta(1)}$
- Remove Edge (u,v): $\boldsymbol{\Theta(\deg(u))}$
- Check edge exists from (u,v): $\boldsymbol{\Theta(\deg(u))}$
- Get neighbors of u (out): $\boldsymbol{\Theta(\deg(u))}$
- Get neighbors of u (in): $\boldsymbol{\Theta(n + m)}$

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

Linked Lists



Adjacency List



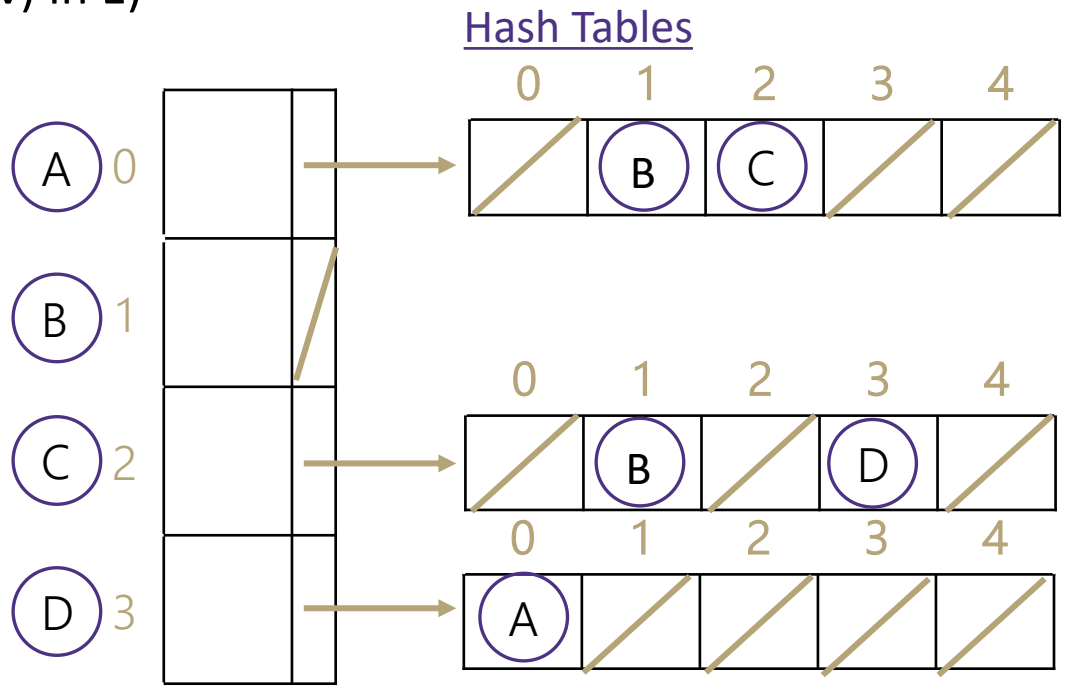
Create a Dictionary of size V from type V to Collection of E

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



Tradeoffs

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

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

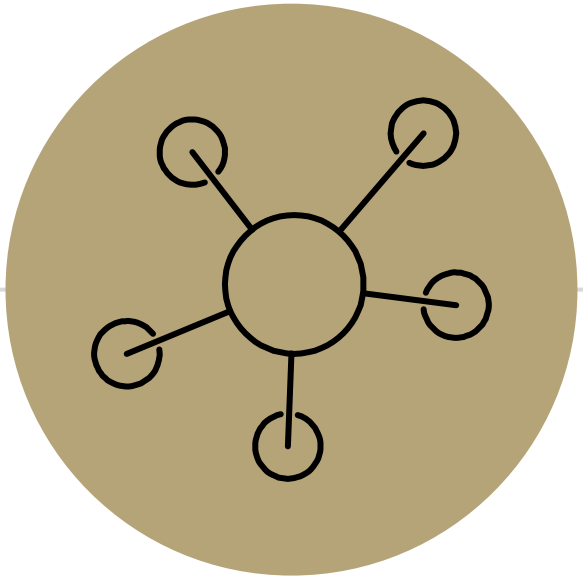
- A hash table still *might* hit a worst-case
- And the linked list might not
  - Graph algorithms often just need to iterate over all the neighbors, so you might get a better guarantee with the linked list.

373: Graph Implementations

For this class, unless we say otherwise, we'll assume the hash tables operations **on graphs** are all $O(1)$.

- Because you can probably control the keys.

Unless we say otherwise, assume we're using the hash table approach.



Questions / clarifications on anything?

relevant ideas for today

- vertices, edges, definitions
- graphs model relationships between real data (you can choose your vertices and edges to
- different graph implementations exist

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)

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:

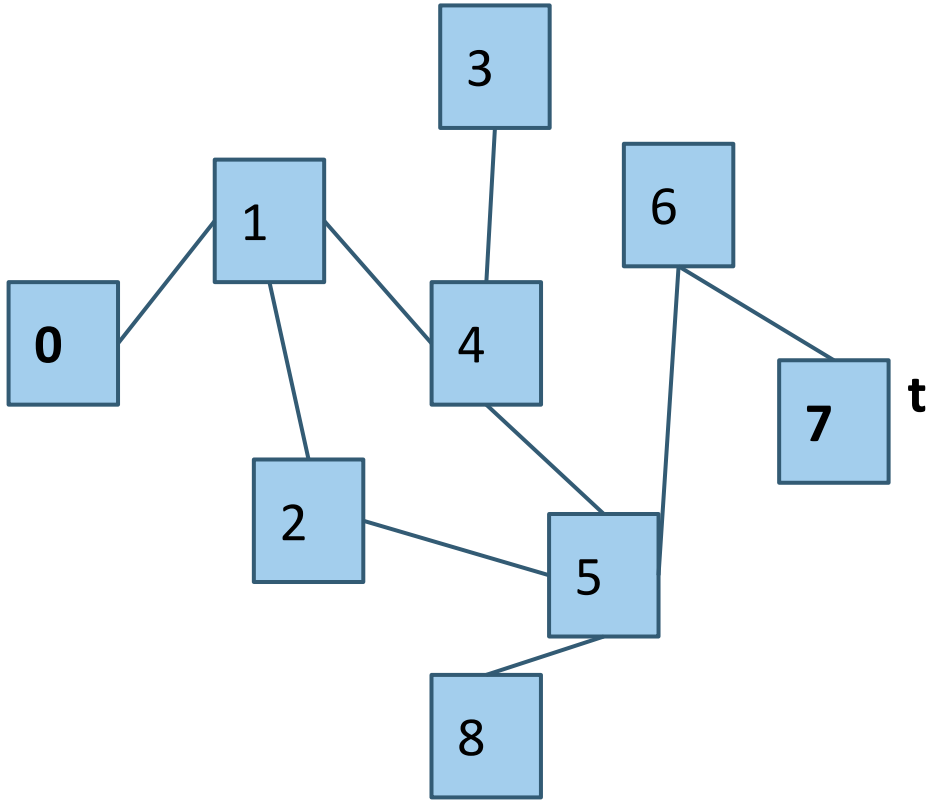
- s-t Path. Is there a path between vertices s and t?
- Connectivity. Is the graph connected?
- Biconnectivity. Is there a vertex whose removal disconnects the graph?
- Shortest s-t Path. What is the shortest path between vertices s and t?
- Cycle Detection. Does the graph contain any cycles?
- Euler 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**?



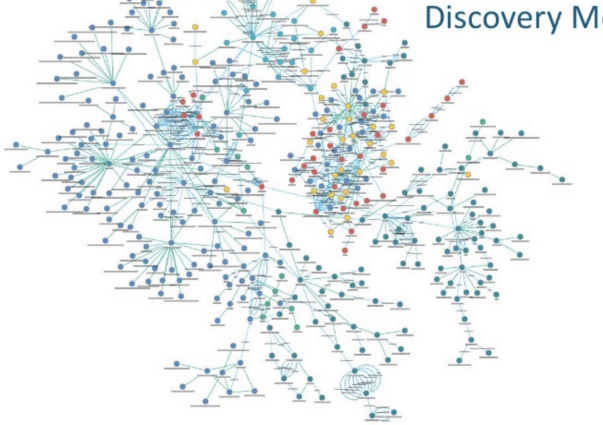
Why does this problem matter? Some possible context:

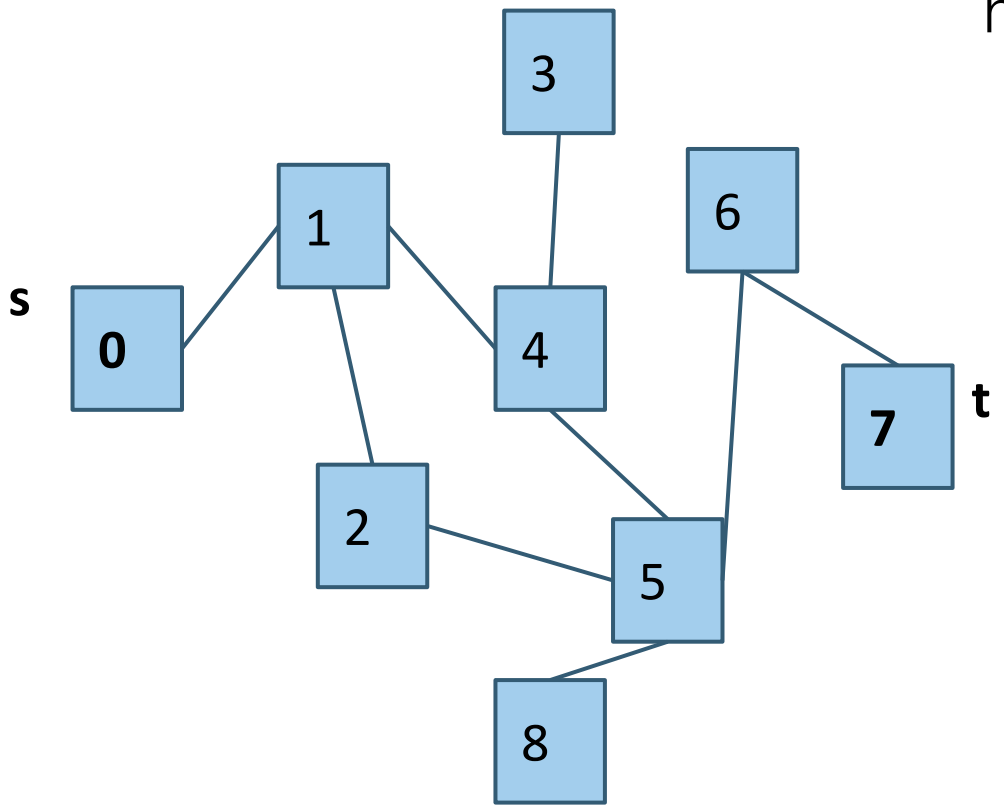
- real life maps and trip planning – can we get from one location (vertex) to another location (vertex) given the current available roads (edges)
- family trees and checking ancestry – are two people (vertices) related by some common ancestor (edges for direct parent/child relationships)
- game states (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**?





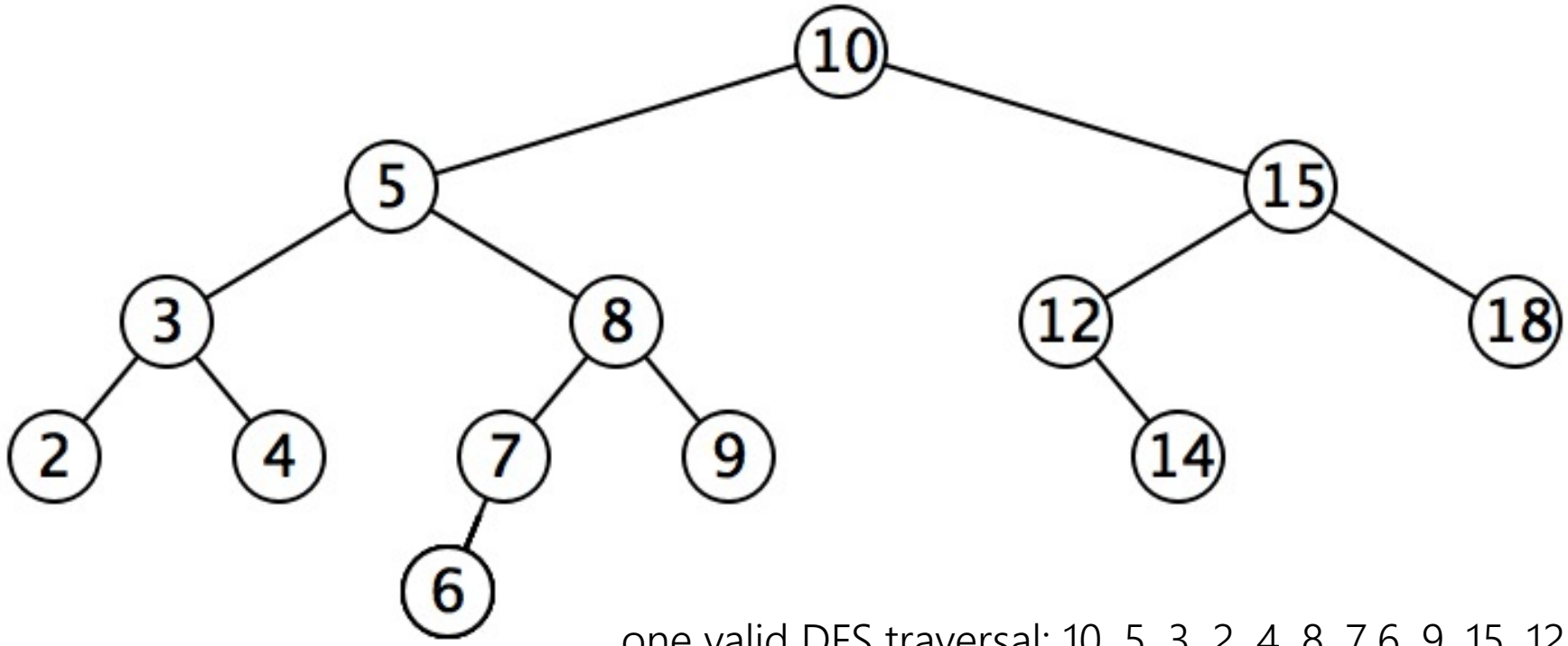
- What's the answer for this graph on the left, and how did we get that answer as humans?
  - We can see there's edges that are visually in between s and t, and we can try out an example path and make sure that by traversing that path you can get from s to t.
  - We know that doesn't scale that well 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 (should feel similar to 143 in the tree context)

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.

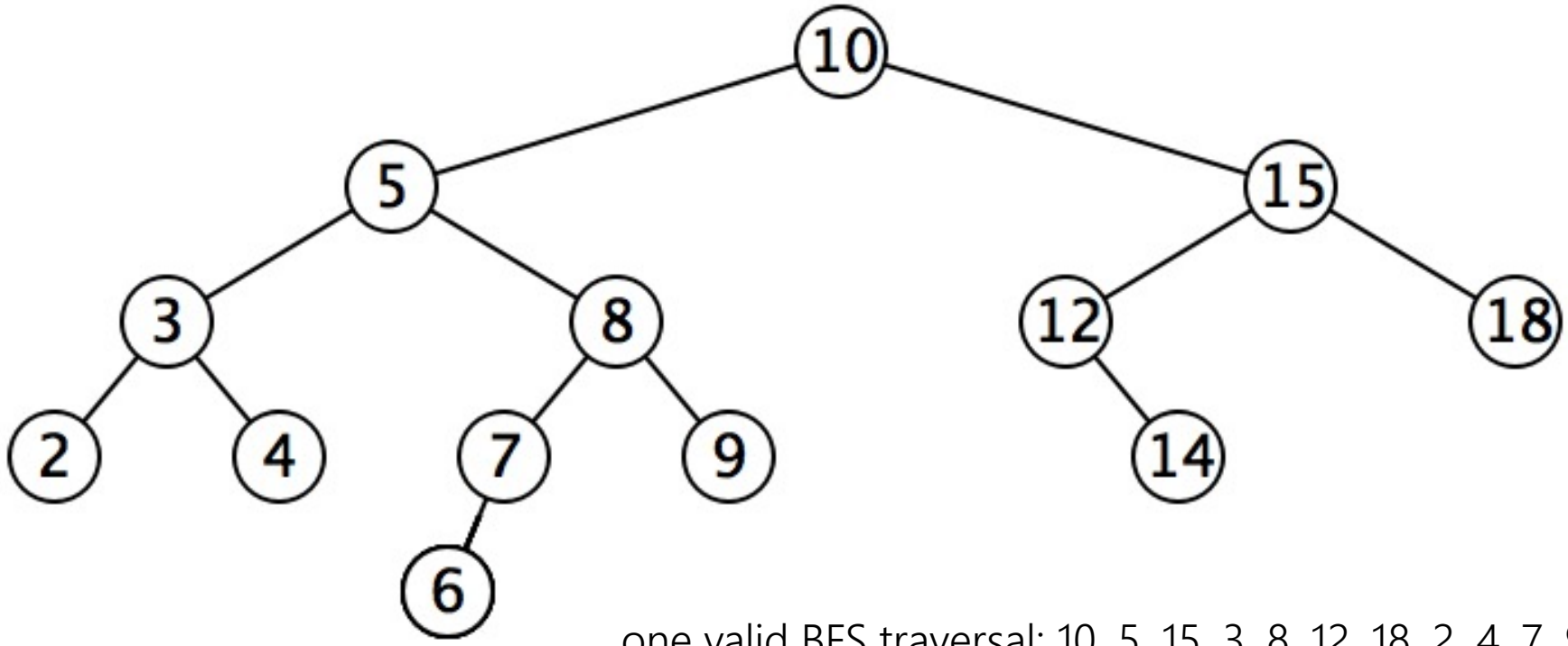
one valid DFS traversal: 10, 5, 3, 2, 4, 8, 7,6, 9, 15, 12, 14, 18

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:

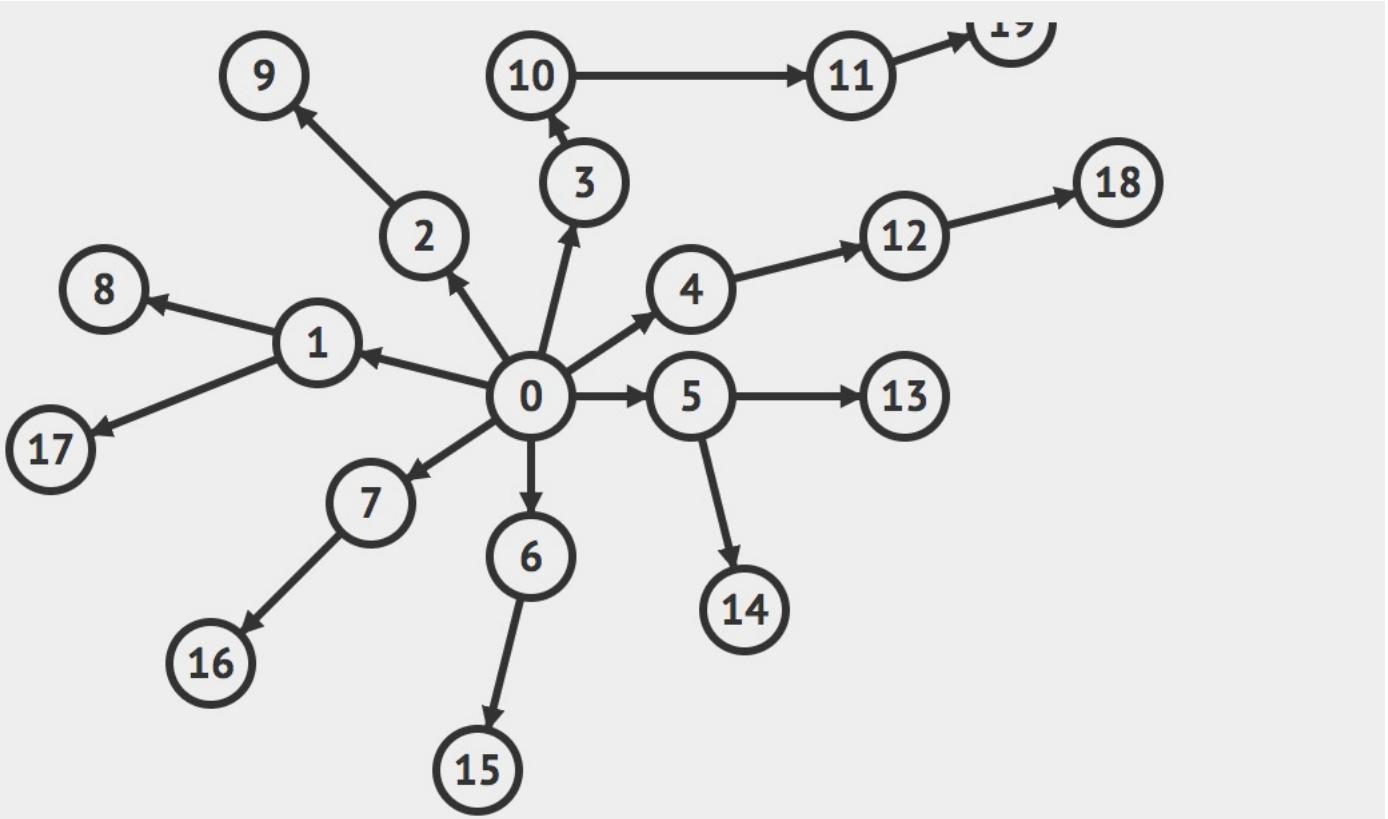
- opposite way of traversing compared to DFS
- a sound wave spreading from a starting point, going outwards in all directions possible.
- mold on a piece of food spreading outwards so that it eventually covers the whole surface



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

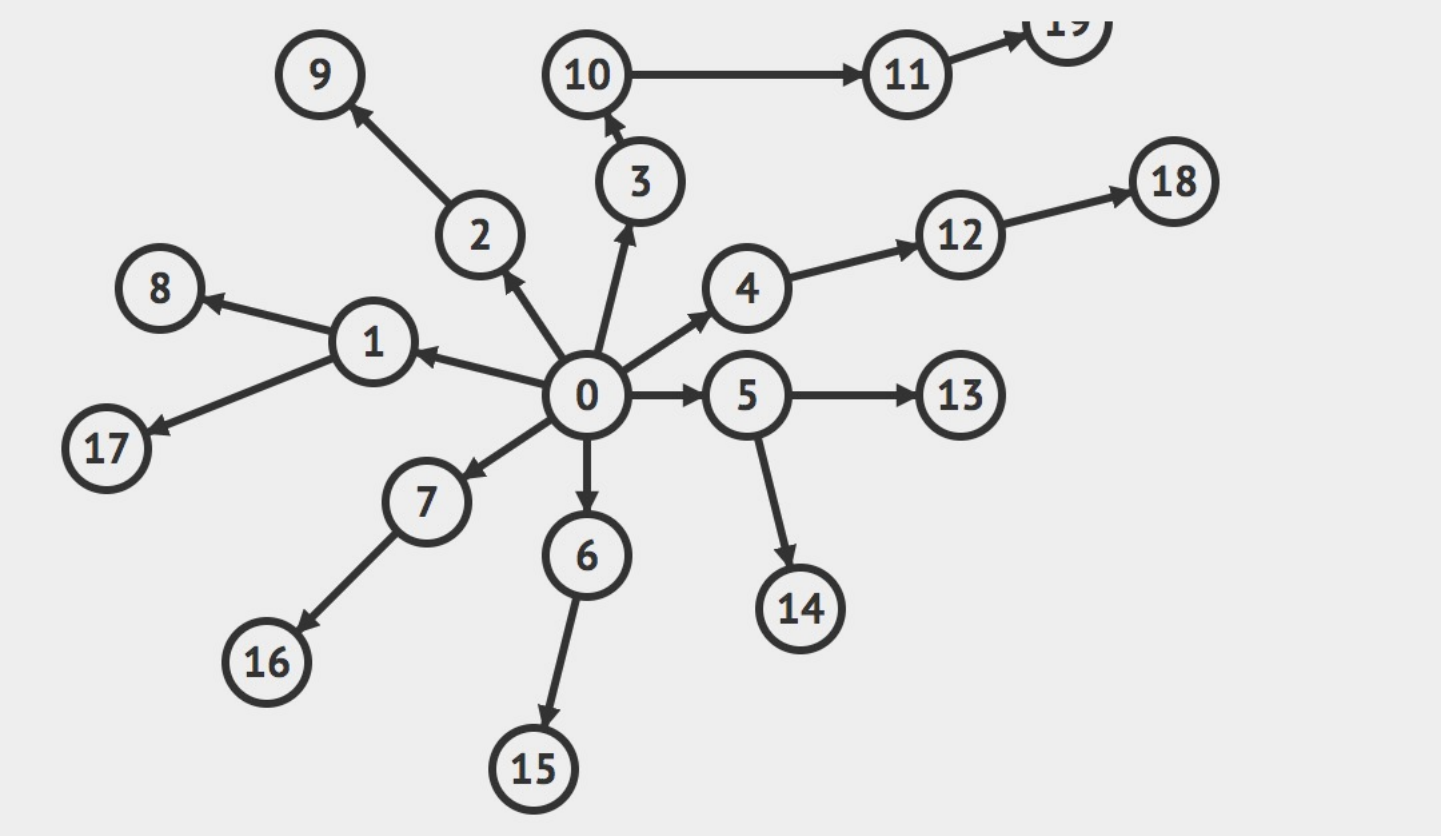
Take 2 minutes and try to come up with two possible traversal orderings <u>starting with the 0 node</u>:

- 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.

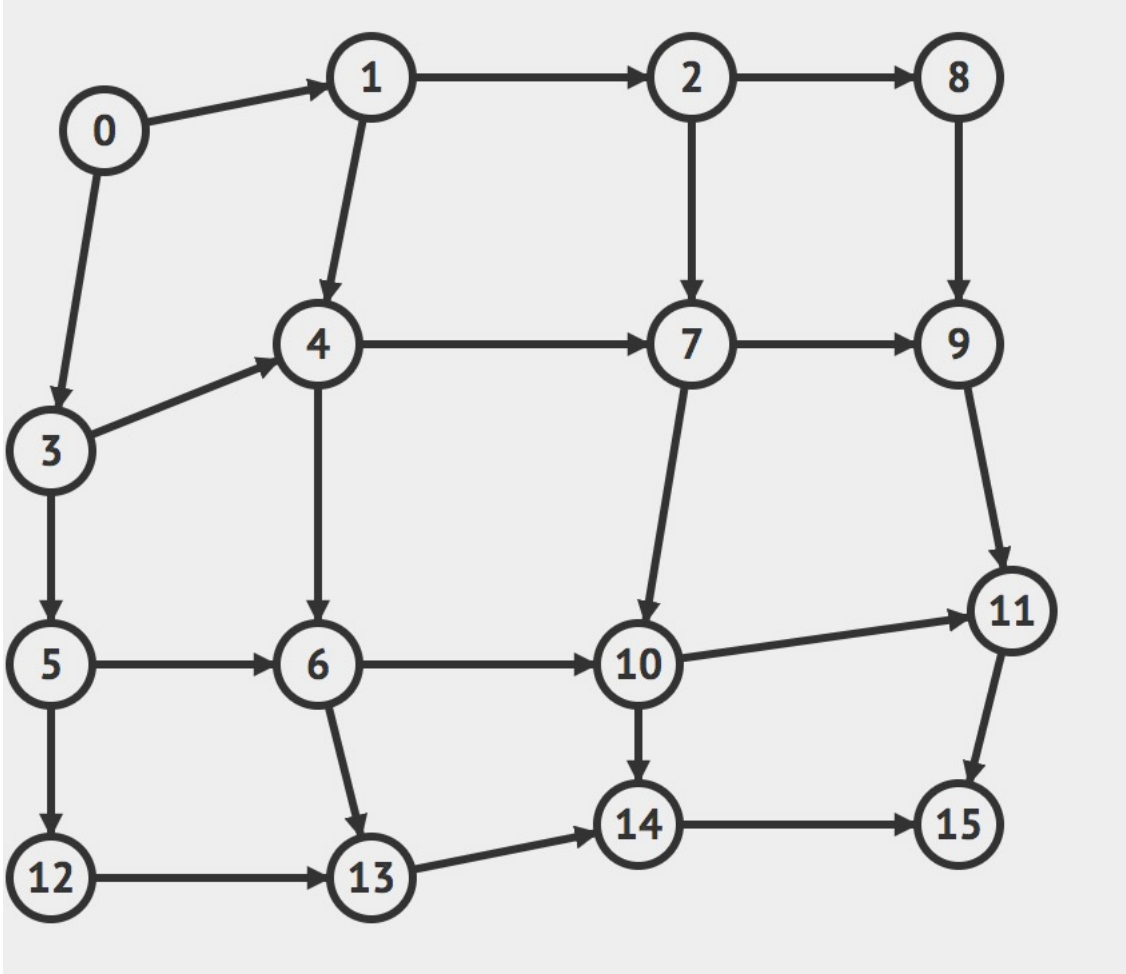


In BFS, you traverse level by level

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

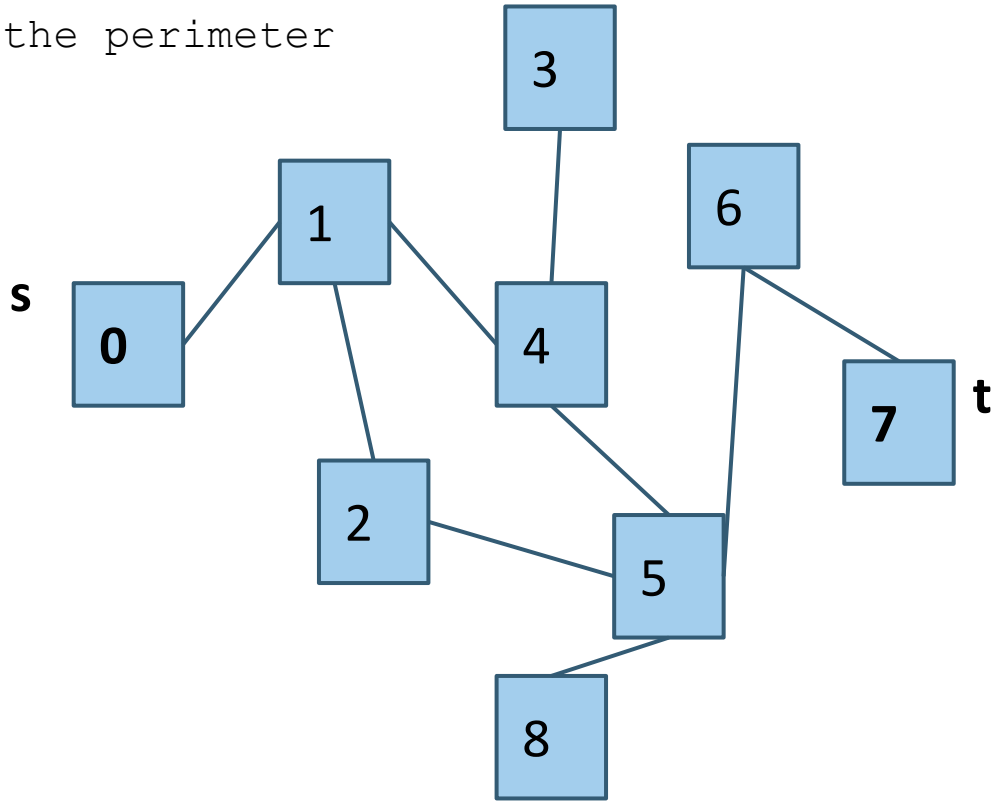


https://visualgo.net/en/dfsbfs

- 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 ☺!

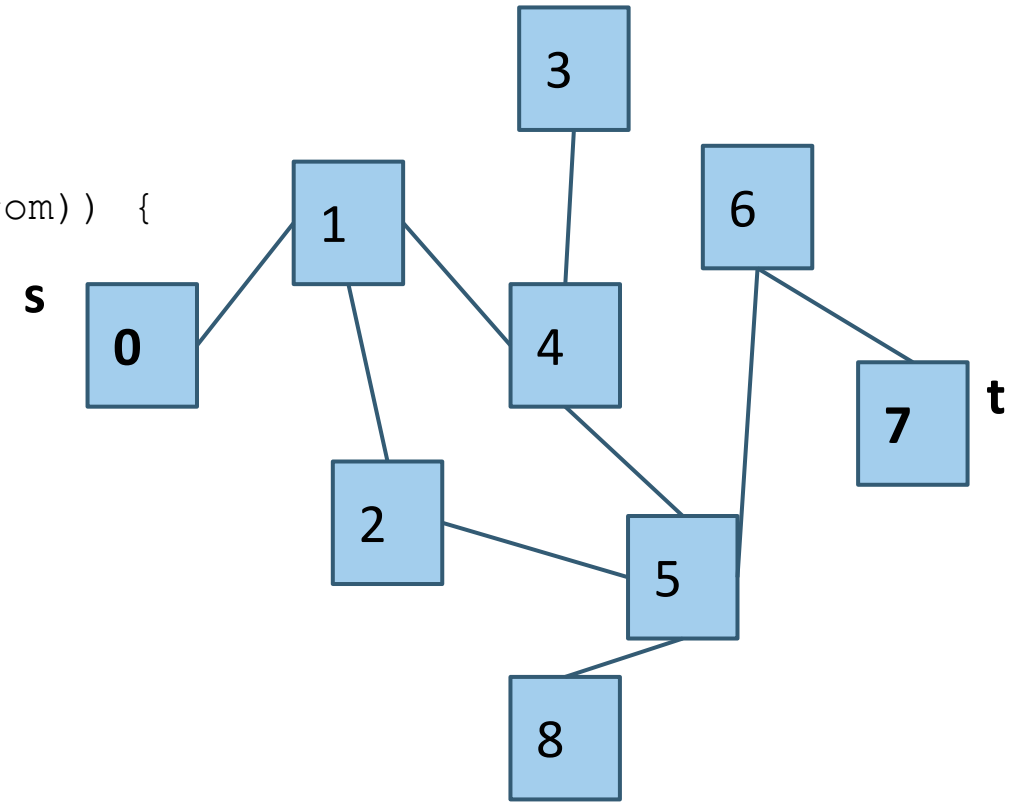
BFS pseudocode

```
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

```
//. . . 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:

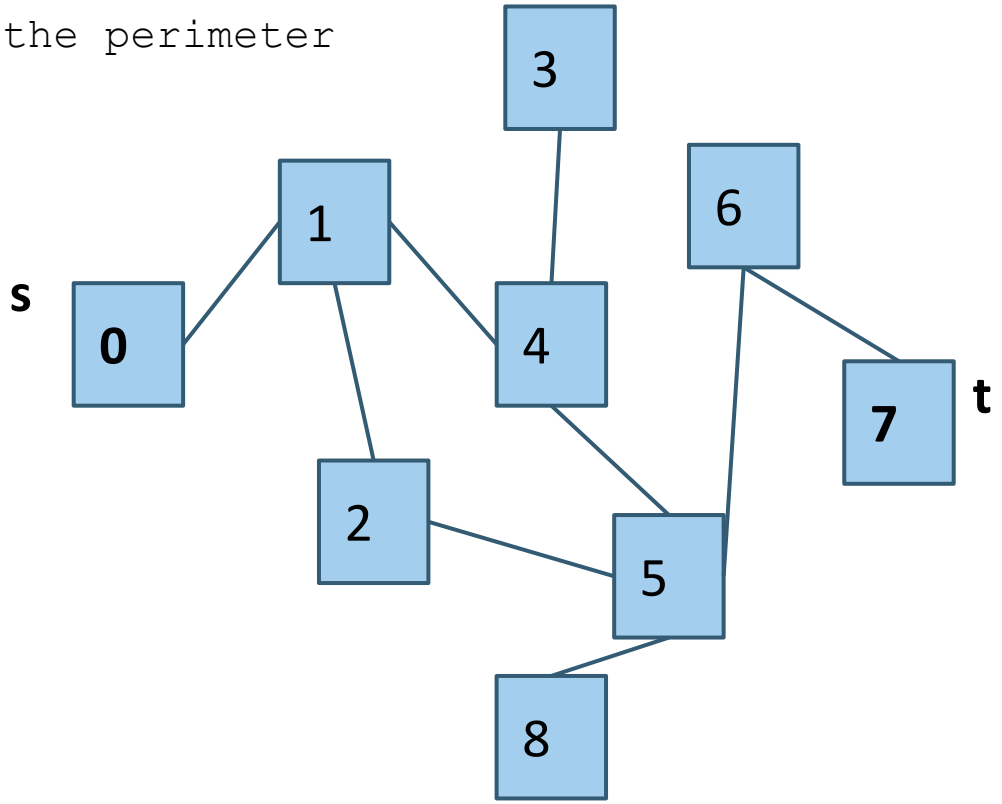
Discovered set:

Expected levels starting the BFS from 0:

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

DFS pseudocode

```
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

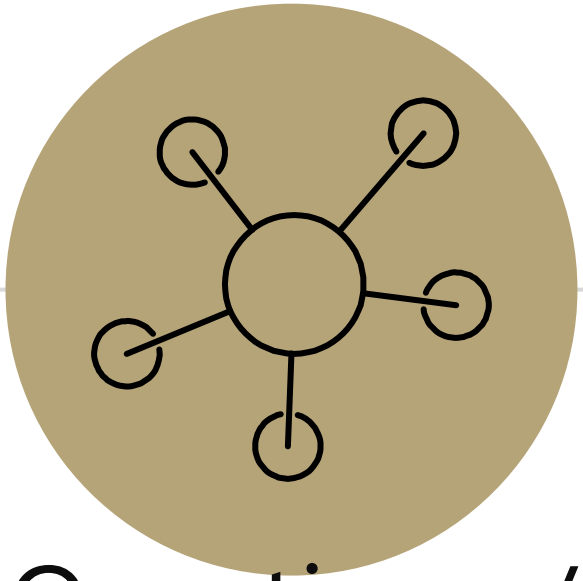
```
//. . . this is the main loop/code for BFS
while (!perimeter.isEmpty()) {
    Vertex from = perimeter.remove();
    for (E edge : graph.outgoingEdgesFrom(from)) {
        Vertex to = edge.to();
        if (!discovered.contains(to)) {
            perimeter.add(to);
            discovered.add(to)
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

```
// 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:

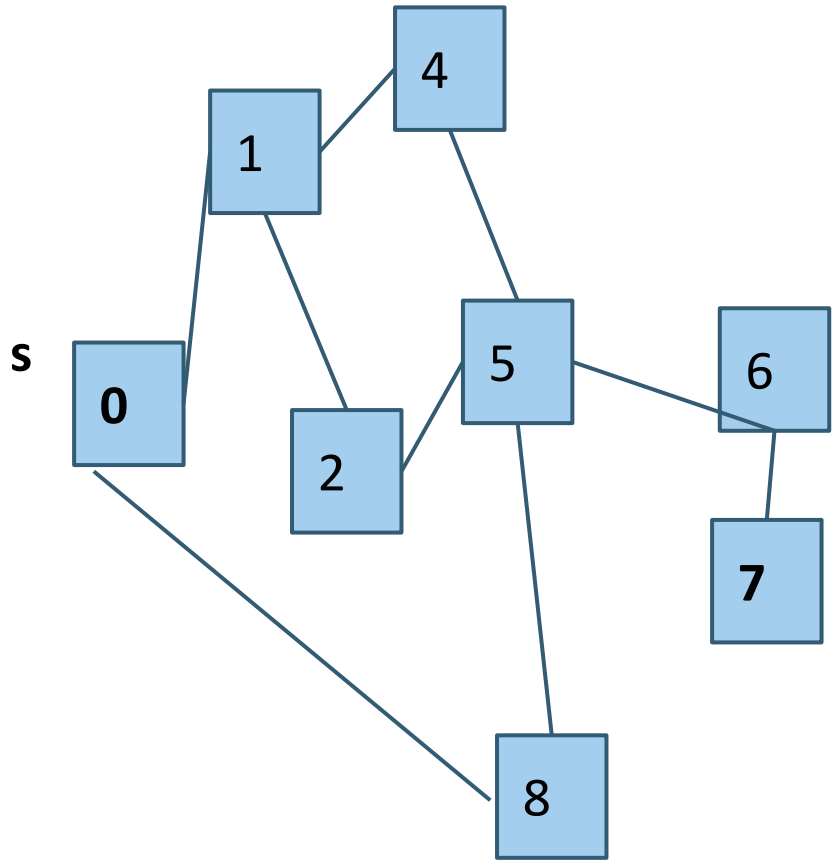
- s-t path problem
- BFS/DFS visually + high-level
- BFS/DFS pseudocode
- modifying BFS/DFS to solve s-t path problem

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)

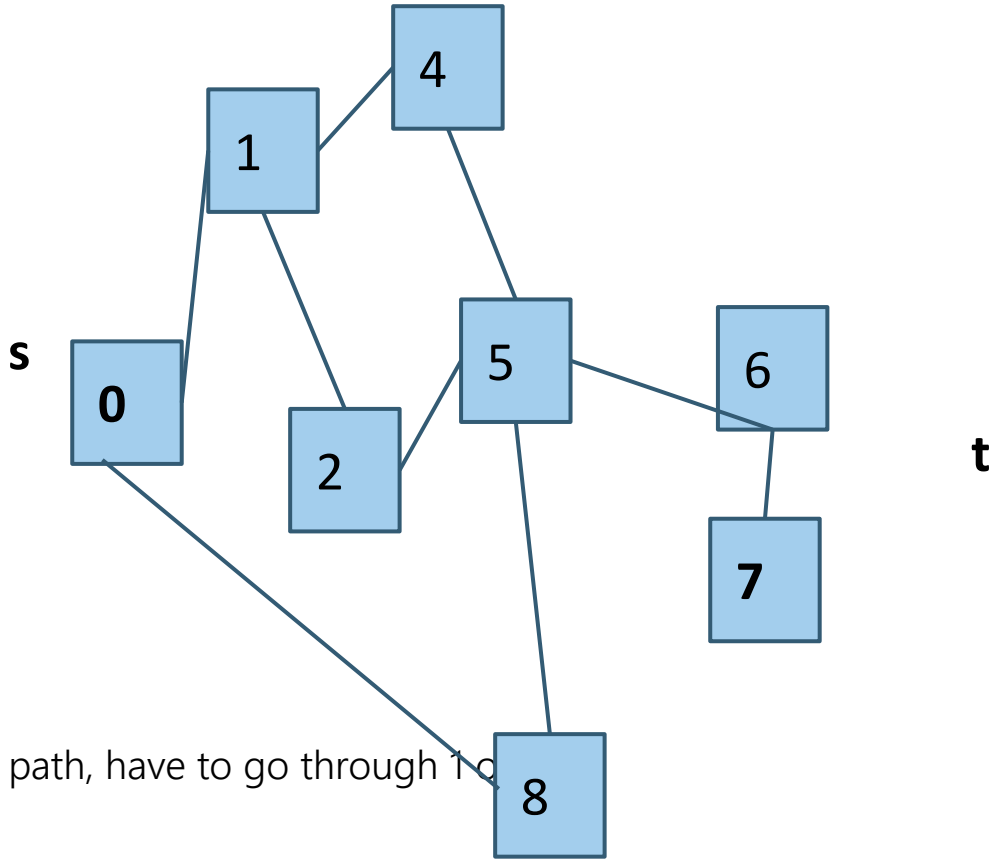
Shortest Path problem (unweighted graph)

- 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:

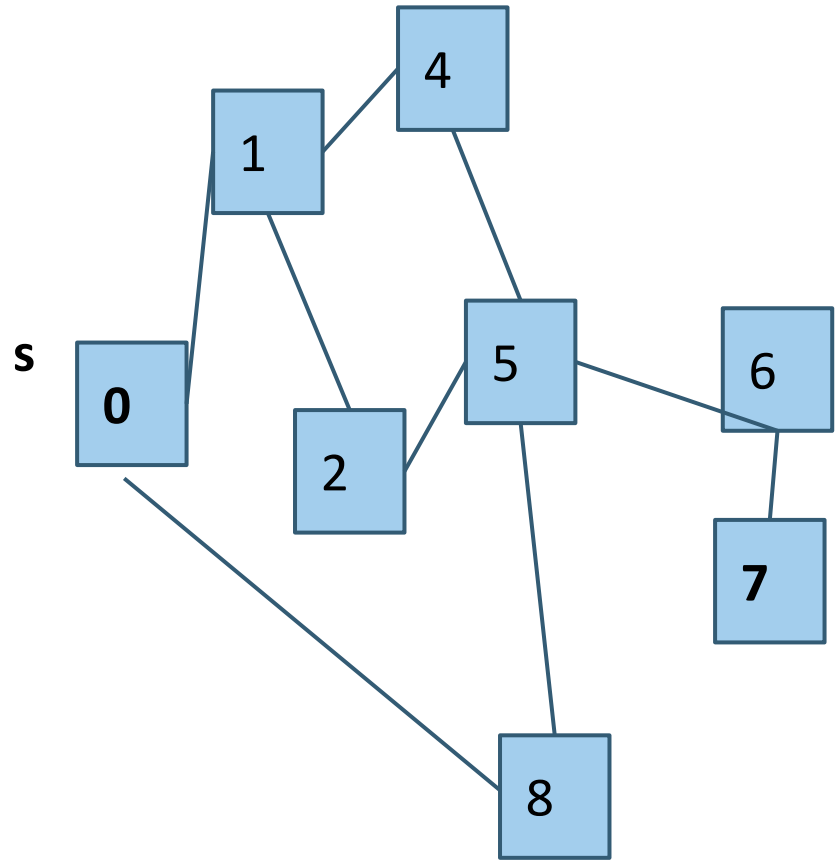
- Every node now will have an associated distance (for convenience)
- Every node V now will have an associated predecessor edge that is the edge that connects V on the shortest path from S to V. The edges that each of the nodes store are the final result.

```
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.

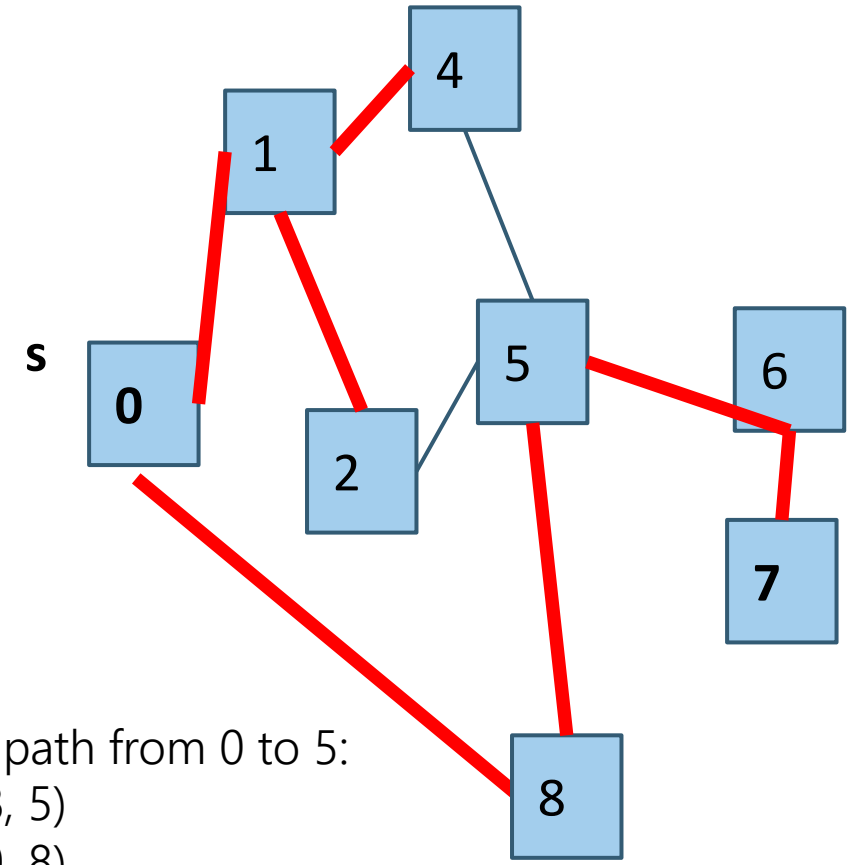
```
perimeter.add(start);
discovered.add(start);
start.distance = 0;
while (!perimeter.isEmpty()) {
    Vertex from = perimeter.remove();
    for (E edge : graph.outgoingEdgesFrom(from)) {
        Vertex to = edge.to();
        if (!discovered.contains(to)) {
            to.distance = from.distance +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