# CS5800: Algorithms

## Week 6 – Graph Algorithms

Dr. Ryan Rad

Fall 2024


Northeastern
University

# Graph Algorithms

- Group Quiz
- Graph Representation
- $s - t$ connectivity problem
- Depth-First Search (DFS)
- Breadth-First Search (BFS)
- Minimum-weight Spanning Tree (MST)
- Kruskal vs Prim
- Shortest Path vs Minimum Spanning Tree
- Quiz Review

# My Top 3 Career Advancement Strategies

**Value**

Value All Work Experience

- Embrace opportunities in non-profit, volunteer, part-time, and contract work.

**Demonstrate**

Demonstrate Expertise Through Action

- Work on meaningful projects that showcase your skills and interests beyond words.

**Network**

Network Actively

- Attend events and focus on connecting with people to grow your professional network.

# Zoom in & out


Lost in the Moment
Where you need to be!
Obsessed with the Future

# Course Check-in Response

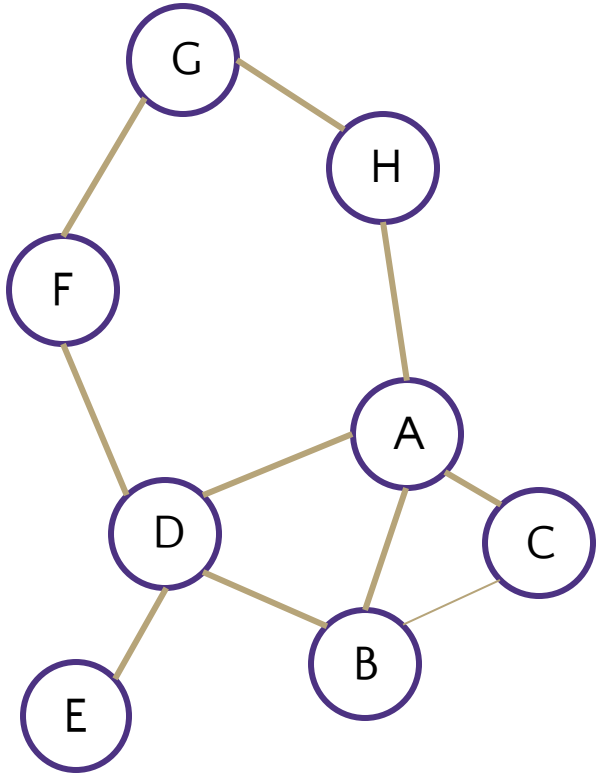
# Graph: Formal Definition

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

- 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: Formal Definition

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

**Labeled**:

- **Weighted**:

**Directed**:

- **Origin, Destination**
- **In-neighbors, out-neighbors**
- **In-degree, out-degree**

**Cyclic**:

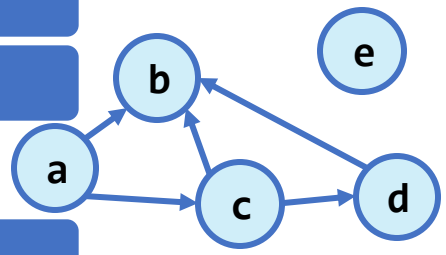
**Simple graph**:

**Path**:

- **Simple path:**
- **Cycle**:

**Self-loop**:

**Parallel edges**:




V: Set of vertices

a

b

c

…

E: Set of edges

( a , b )

( a , c )

( c , d )

…

# Graph: Formal Definition

**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]

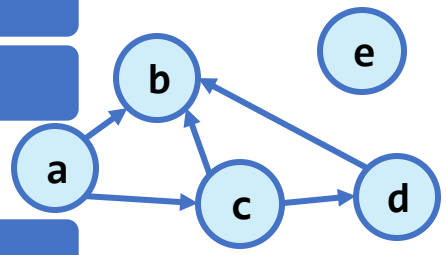
**Simple graph**: No self-loops or parallel edges

**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)

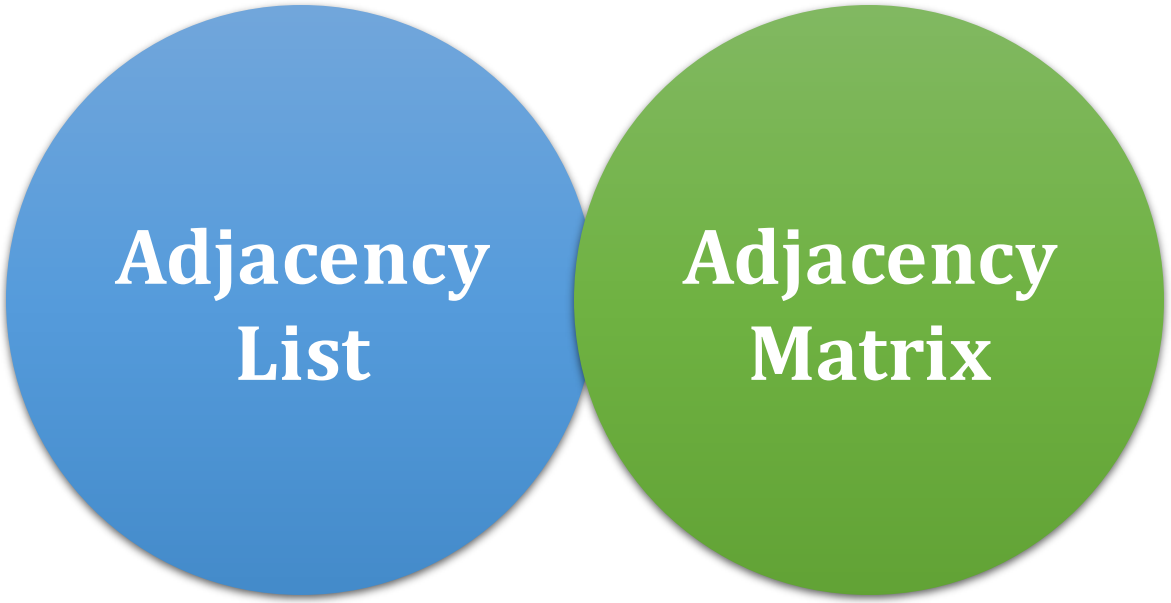
(c, d)

...

# Graph Representation

Given graph $G = (V, E)$. In pseudocode, represent vertex set by $G.V$ and edge set by $G.E$.

- $G$ may be either directed or undirected.
- Two common ways to represent graphs for algorithms:



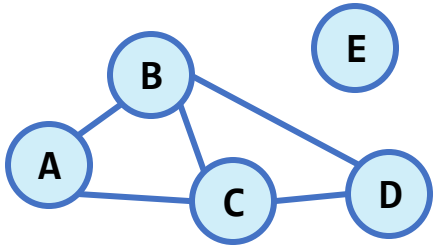
# Adjacency

Array $Adj$ of $|V|$ lists, one per vertex.

Vertex $u$'s list has all vertices $v$ such that $(u, v) \in E$. (Works for both directed and undirected graphs.)

In pseudocode, denote the array as attribute $G.Adj$, so will see notation such as $G.Adj[u]$.

# Adapting for Undirected Graphs




**Adjacency Matrix**

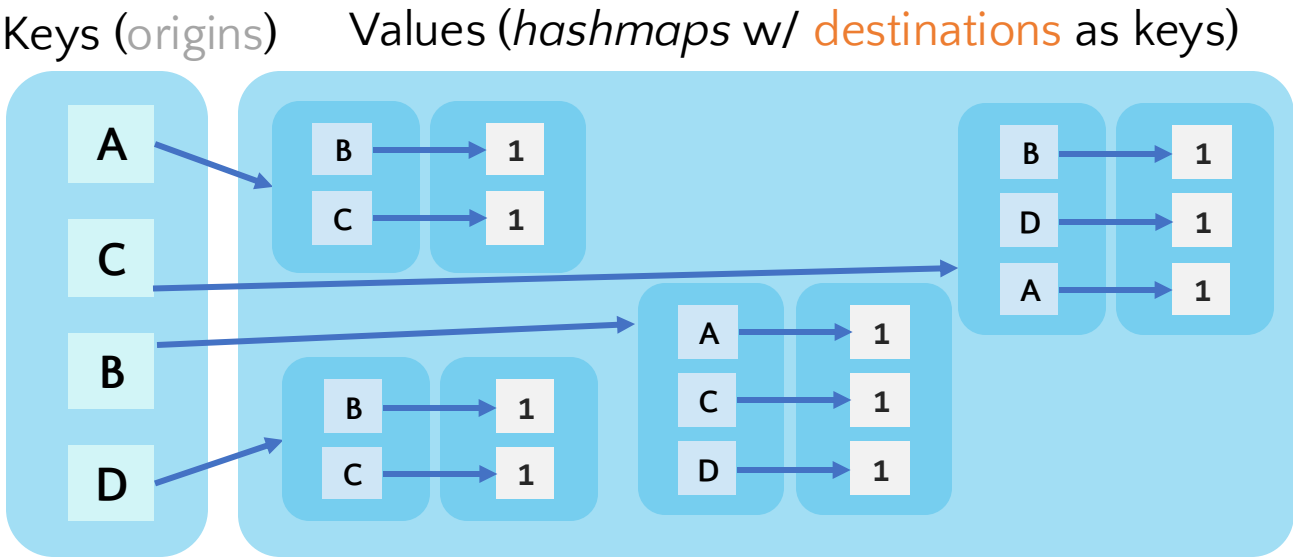
Store each edge as both directions
(makes the matrix symmetrical)

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**

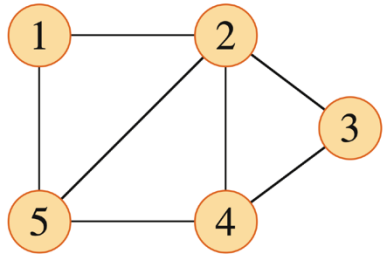
Store each edge as both directions
(doubles the number of entries)



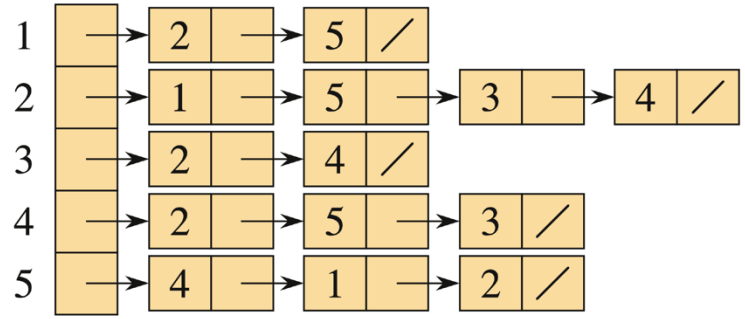

***Abstraction** of the Hash Map! **Buckets not shown**.*

# Adjacency List

- For an undirected graph:




(a)




(b)

If edges have *weights*, can put the weights in the lists.

Weight: $w : E \to \mathbb{R}$

We'll use weights later on for spanning trees and shortest paths.

***Space:***

***Time:*** to list all vertices adjacent to $u$:

***Time:*** to determine whether $(u, v) \in E$:

# Adjacency Matrix

$|V| \times |V|$ matrix $A = (a_{ij})$

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E , \\ 0 & \text{otherwise .} \end{cases}$$

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 | 1 |
| 2 | 1 | 0 | 1 | 1 | 1 |
| 3 | 0 | 1 | 0 | 1 | 0 |
| 4 | 0 | 1 | 1 | 0 | 1 |
| 5 | 1 | 1 | 0 | 1 | 0 |

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 |
| 2 | 0 | 0 | 0 | 1 |
| 3 | 1 | 1 | 0 | 0 |
| 4 | 0 | 0 | 1 | 1 |

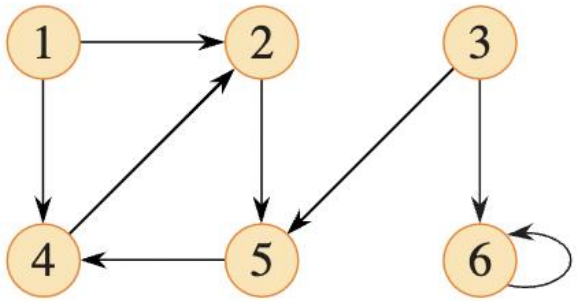
***Space:***

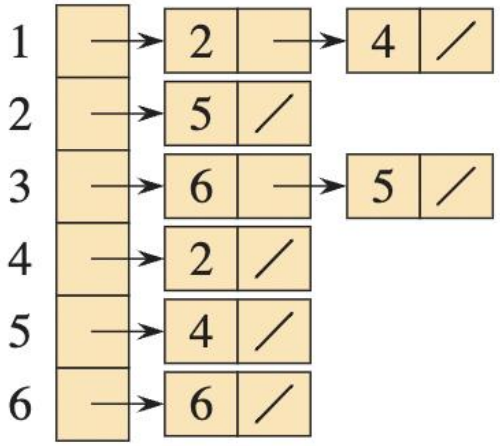
***Time:*** to list all vertices adjacent to $u$:

***Time:*** to determine whether $(u, v) \in E$:

Can store weights instead of bits for weighted graph.

# Directed Graph




| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 1 | 0 |
| 3 | 0 | 0 | 0 | 0 | 1 | 1 |
| 4 | 0 | 1 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 1 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 0 | 1 |

# Let’s discuss!


Mark Zuckerberg, the CEO of Facebook, has hired you to lead the Facebook Algorithms Group. He has asked you to use various graph algorithms to analyze the world's largest social network.

The Facebook Graph has 2.8 billion vertices, with each vertex being a Facebook user. Two vertices are connected provided those two users are "friends".

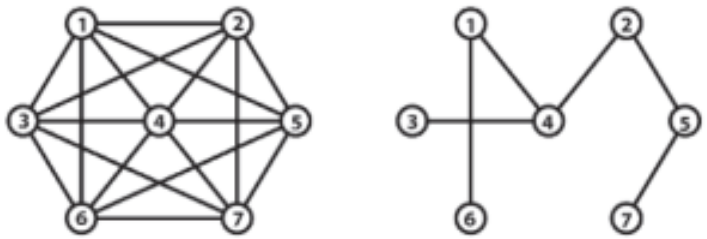
The first decision you need to make is how you want to model the Facebook graph.

**Determine whether you should use an adjacency-list representation or an adjacency-matrix representation.**

# Let's discuss!

- A graph on **n** vertices has a maximum of 1+2+3+…+(n-1) = **n(n-1)/2** edges.
- Let |E| be the total number of edges in this graph.
- The **density** of a graph is defined to be |E| divided by n(n-1)/2.

If this value is close to 1 (the maximum ratio), we say the graph is DENSE.
If this value is close to 0 (the minimum ratio), we say the graph is SPARSE.



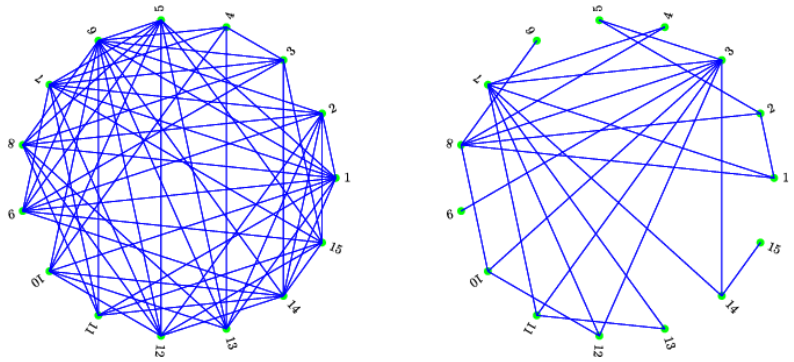

Dense




Sparse

The graph on the left has density 18/21 = 0.857
The graph on the right has density 6/21 = 0.285

# Let's discuss!



*If it's helpful, think of a dense graph has a graph having $O(n^2)$ edges, and a sparse graph as a graph having $O(n)$ edges.*

If a graph is SPARSE, you definitely want to use an **adjacency-list representation**, since every vertex is connected to just a small number of other vertices. There is no need to create an ***n*** by ***n*** matrix storing the set of edges, since the large majority of these entries will be 0.

If a graph is DENSE, you definitely want to use an **adjacency-matrix representation**, since the majority of the entries in the ***n*** by ***n*** matrix will be 1.

You want to be able to quickly check whether an edge appears in the graph. With a matrix, you can do this in O(1) time. With a linked list, this will take O(n) time.

# Let's discuss!

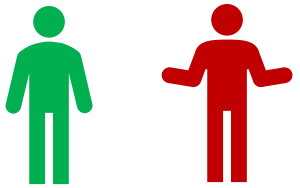
Anyone here on Facebook?

If so, approximately how many friends do you have?

So even if someone has 1,000 friends, that's just a small fraction of the 2.5 billion users on Facebook.
So the Facebook graph is **incredibly sparse**. So you definitely want to use an **adjacency-list representation**.

# Algorithm Design

This week, you will submit a form by specifying your preferences for course project.
Each of could specify who you wish to form a group with
Each of could specify who you wish not to form a group with



First, let's transform the scenario into a graph:

- **What are the vertices?**
- **What are the edges?**
- **Undirected or directed?**
- **Weighted or unweighted?**

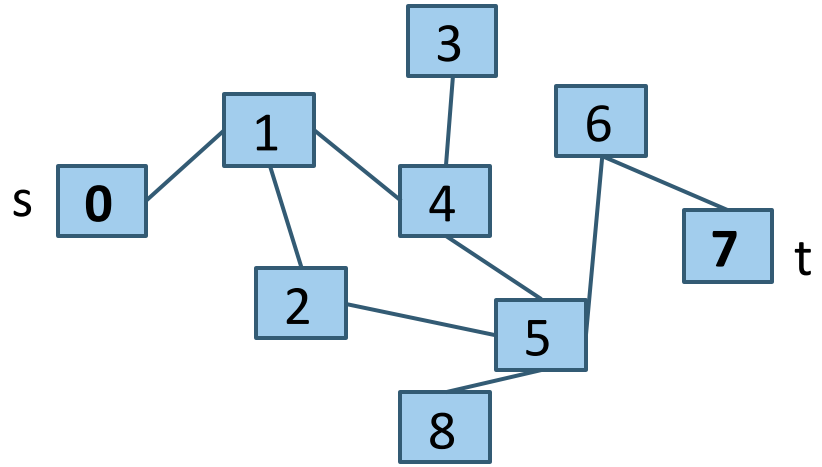
Next, we wanted to form the most optimal groups:

- **What kind of computational problem is this?**

# 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**?

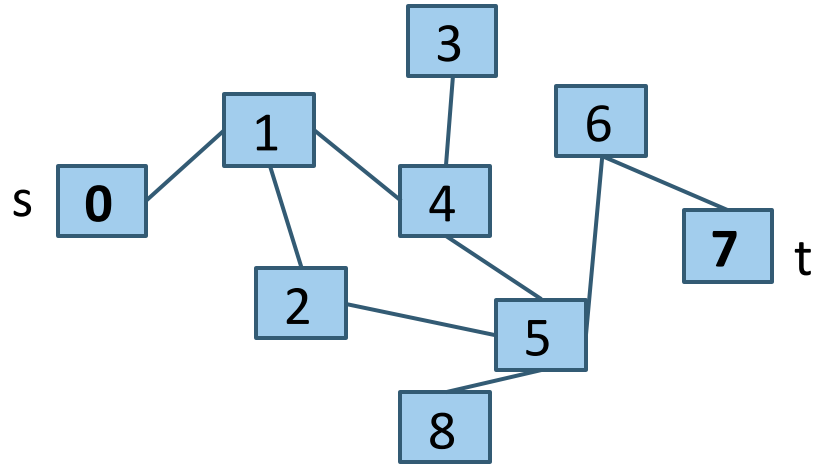



Try to come up with an algorithm for `connected(s, t)`

- 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

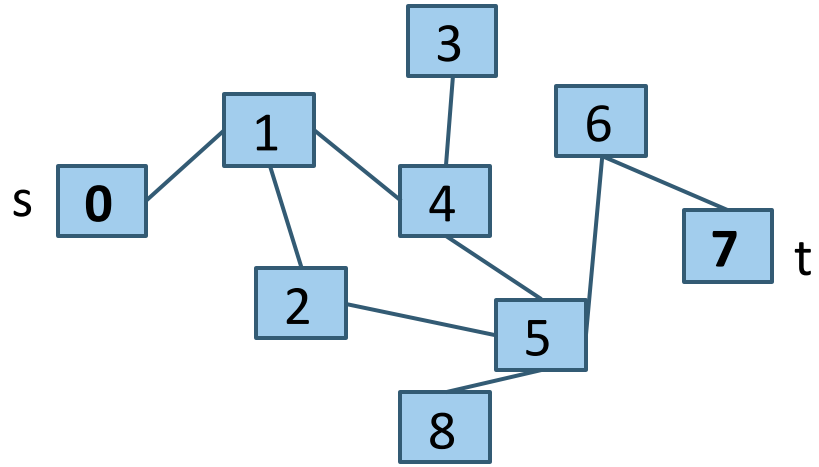
```
connected(Vertex s, Vertex t) {
  if (s == t) {
    return true;
  } else {
    for (Vertex n : s.neighbors) {
      if (connected(n, t)) {
        return true;
      }
    }
    return false;
  }
}
```




**What's wrong with this proposal?**

# 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;
  }
}
```

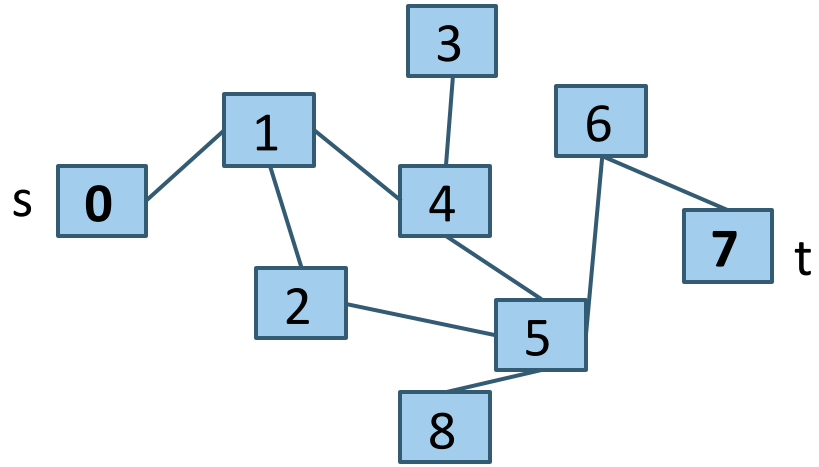



Does 0 == 7? No; if(connected(1, 7) return true;
Does 1 == 7? No; if(connected(0, 7) return true;
Does 0 == 7?

# 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;
  }
}
```



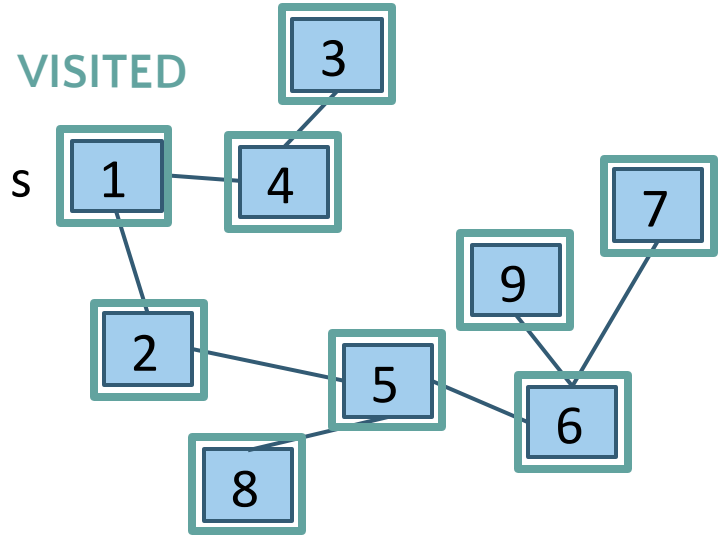

This general approach for crawling through a graph is going to be the basis for a LOT of algorithms!

# 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;
  }
}
```

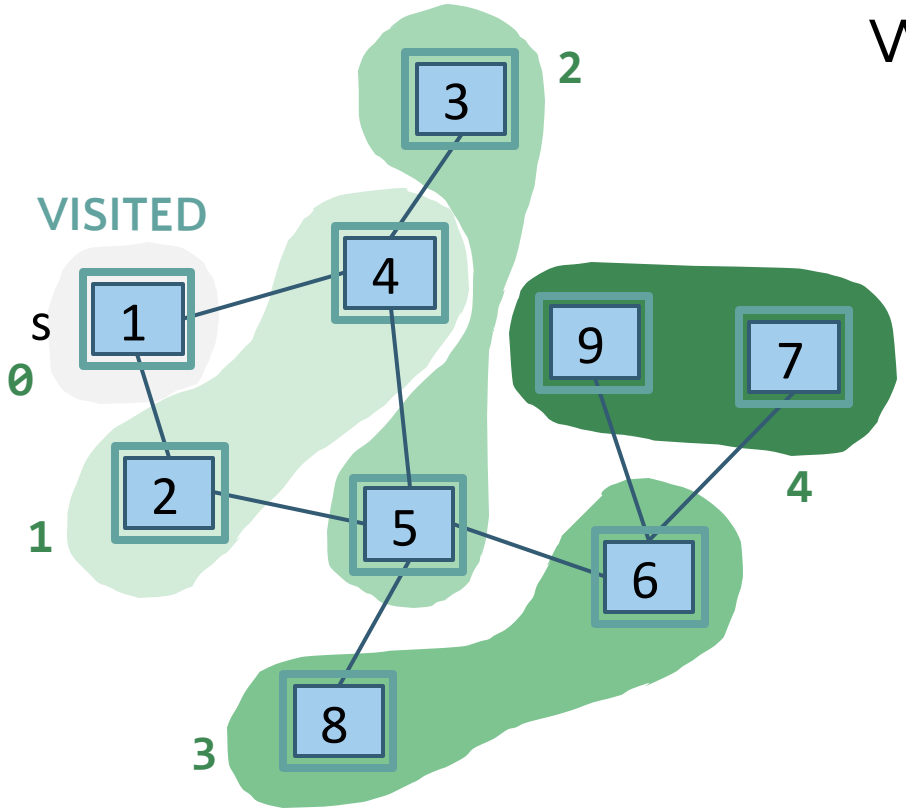
- It will explore one option "all the way down" before coming back to try other options
  - Many possible orderings: {1, 2, 5, 6, 9, 7, 8, 4, 3} or {1, 4, 3, 2, 5, 8, 6, 7, 9} both possible
- We call this approach a **depth-first search (DFS)**



# 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"

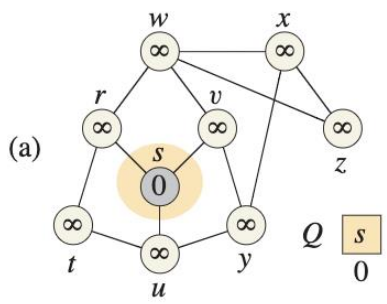
This is our goal, but how do we translate into code?

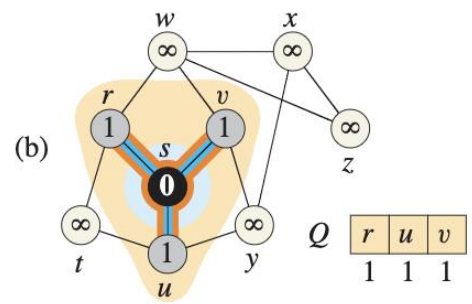
- Key observation: recursive calls interrupted s.neighbors loop to immediately process children
- For BFS, instead we want to *complete* that loop before processing children
- Recursion isn't the answer! Need a data structure to "queue up" children…

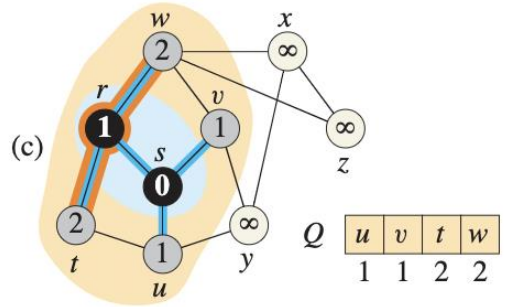
```
for (Vertex n : s.neighbors) {
```

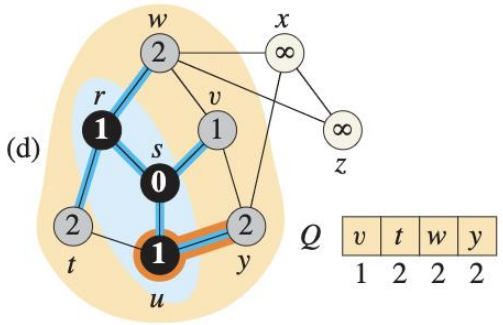
# Breadth First Search (BFS)

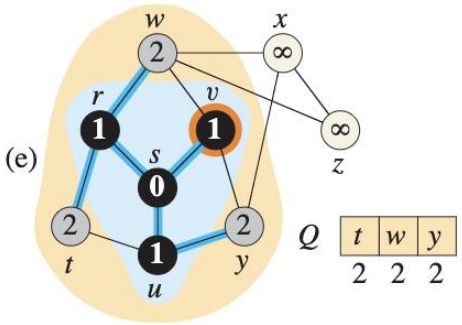
```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2      u.color = WHITE
 3      u.d = ∞
 4      u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11      u = DEQUEUE(Q)
12      for each vertex v in G.Adj[u]    // search the neighbors of u
13          if v.color == WHITE          // is v being discovered now?
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)            // v is now on the frontier
18      u.color = BLACK                  // u is now behind the frontier
```

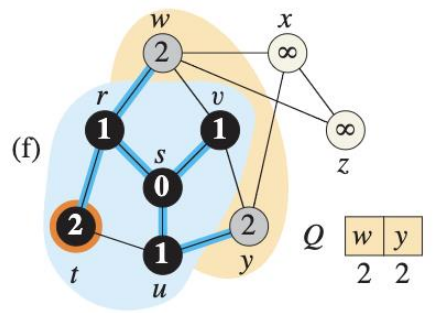


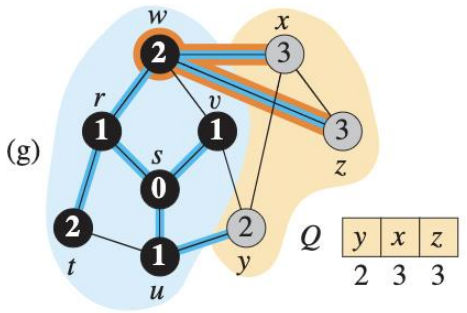



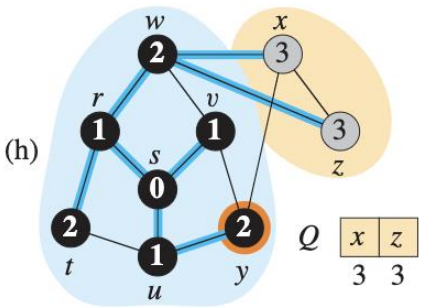



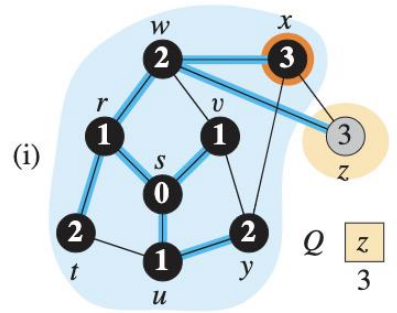



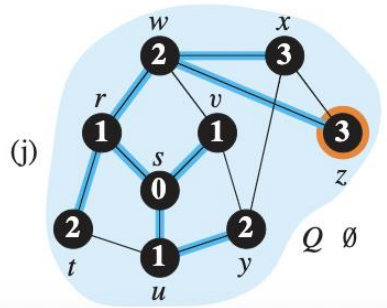



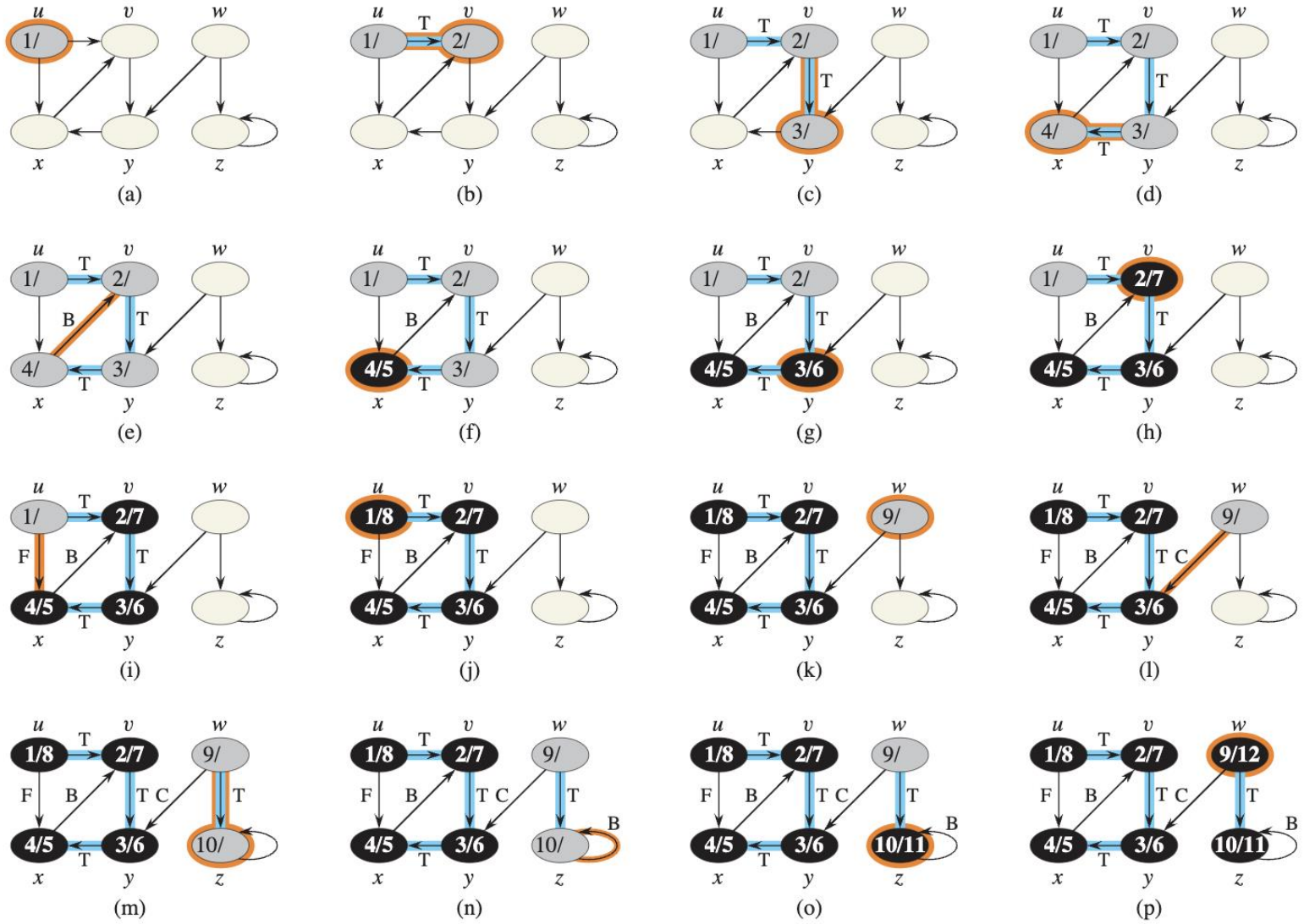



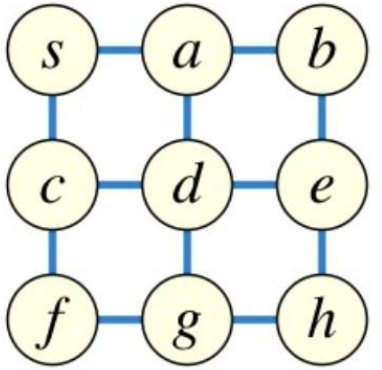
# Depth First Search (DFS)

```
DFS(G)
 1 for each vertex u ∈ G.V
 2     u.color = WHITE
 3     u.π = NIL
 4 time = 0
 5 for each vertex u ∈ G.V
 6     if u.color == WHITE
 7         DFS-VISIT(G, u)

DFS-VISIT(G, u)
 1 time = time + 1                // white vertex u has just been
                                  discovered
 2 u.d = time
 4 for each vertex v in           // explore each edge (u, v)
   G.Adj[u]
 5     if v.color == WHITE
 6         v.π = u
 7         DFS-VISIT(G, v)
 8 time = time + 1
 9 u.f = time
10 u.color = BLACK                // blacken u; it is finished
```



# Question

Consider the following graph with 9 vertices and 12 edges.

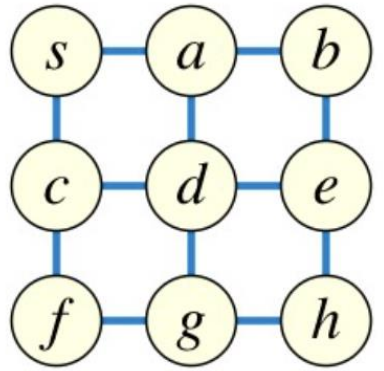


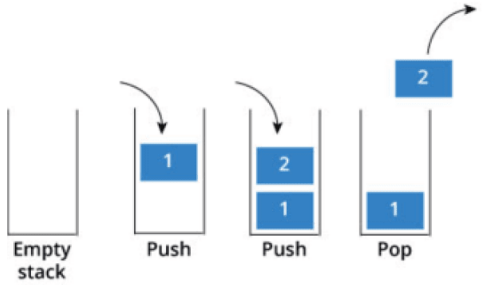

Starting with vertex ***s***, we can use a search algorithm to pass through all nine vertices in this graph. Whenever we have more than one option, we always pick the vertex that appears earlier in the alphabet.
For example, from vertex ***s***, we go to ***a*** instead of ***c***, since the letter ***a*** appears before the letter ***c*** in the alphabet.

a) **Determine the order in which the nine vertices are reached using Depth-First Search (DFS)**
b) **Determine the order in which the nine vertices are reached using Breadth-First Search (BFS)**

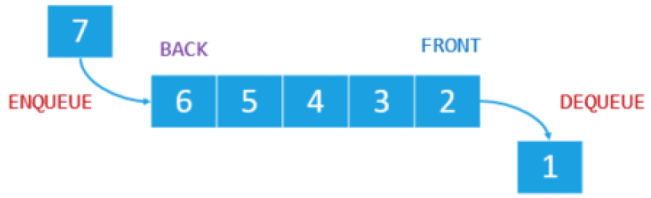
# Reminder

Depth-First Search (DFS) uses a **stack**
Breadth-First Search (BFS) uses a **queue**




Stack




Queue

You all know about queues (e.g. waiting at the grocery store)
You all know about stacks (e.g. when putting your dishes away

# 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 ☺

## *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**!

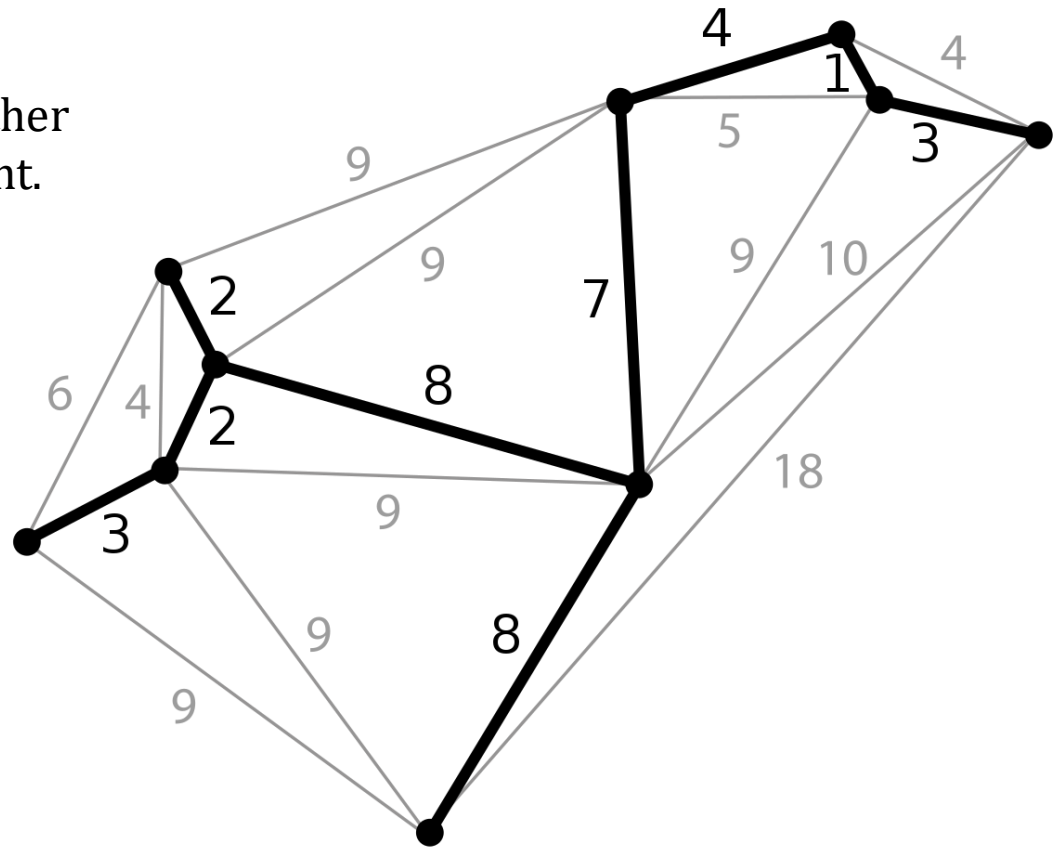
- Note: we're not using any specific properties of BFS here, we just needed a traversal. DFS would also work.

```
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;
}
```

# Growing an MST

**What is a Minimum Spanning Tree?**

A minimum spanning tree (MST) is a subset of the edges of a connected, edge-weighted graph that connects all the vertices together without any cycles and with the minimum possible total edge weight.



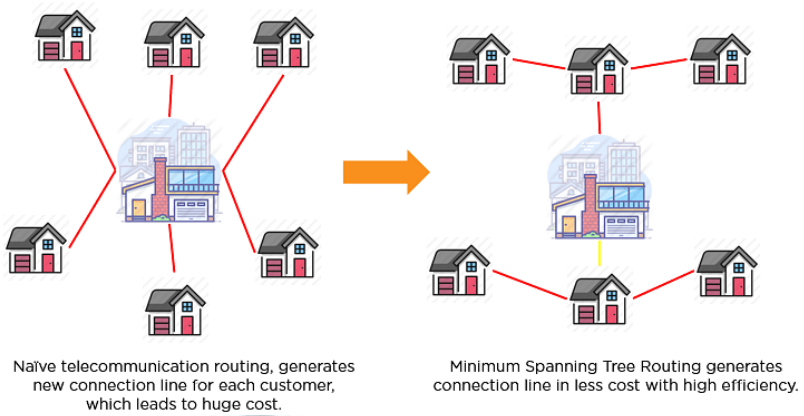
# Minimum-weight Spanning Trees (MST)

**First of all, what's the significances of this topic?**
A group of students at Carleton University (Ottawa, Canada) care deeply about this.  Why?

- https://www.researchgate.net/figure/Carleton-University-Campus-tunnel-network-Ottawa-Canada_fig3_340741033
- https://charlatan.ca/2019/03/carleton-pilots-new-tunnel-navigation-project/

**What are some other examples?**

- Telecommunications companies laying cable to connect every home
- Helping endangered species migrate by creating wildlife corridors
- Creating roads and streets that connect different parts of a community

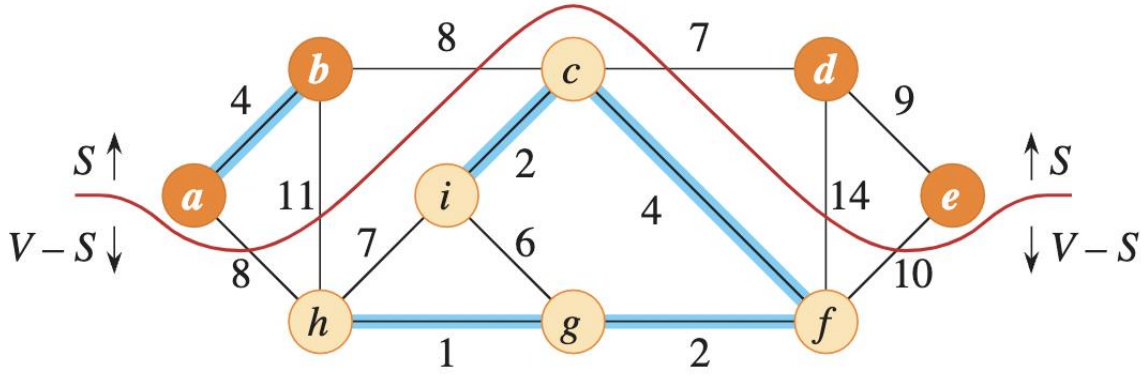



https://www.simplilearn.com/tutorials/data-structure-tutorial/minimum-spanning-tree-algorithm-in-data-structure

# Growing an MST

**Loop Invariant:**
Prior to each iteration, A is a subset of some minimum spanning tree

```
GENERIC-MST(G, w)
1  A = Ø
2  while A does not form a spanning tree
3      find an edge (u, v) that is safe for A
4      A = A ∪ {(u, v)}
5  return A
```

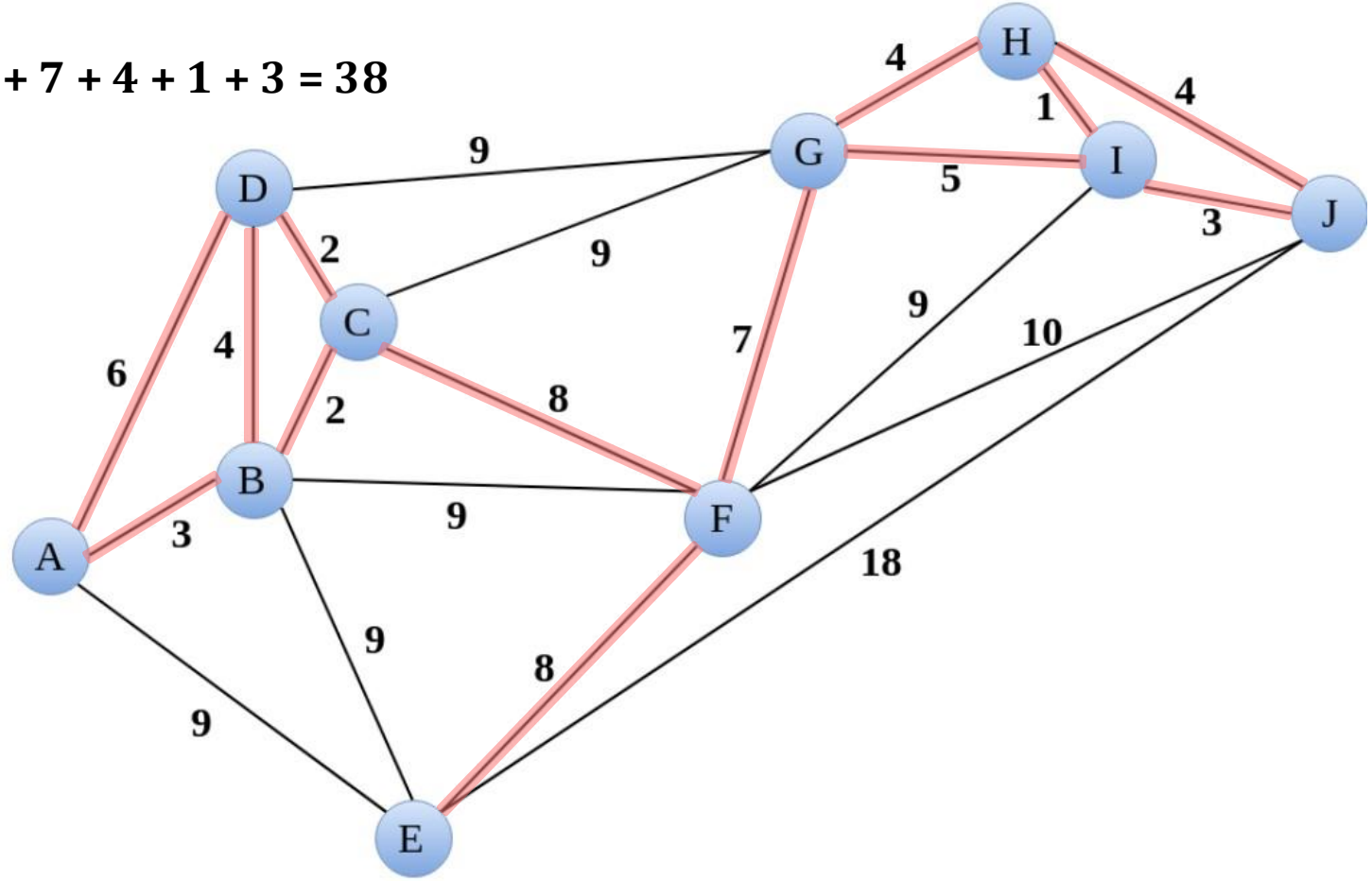



Let $A$ be a subset of $E$ that is included in some minimum spanning tree for $G$, let $(S, V - S)$ be any cut of $G$ that respects $A$, and let $(u\ v)$ be a light edge crossing $(S, V - S)$ . Then, **edge $(u\ v)$ is safe for $A$.**

# Kruskal's Algorithm

**We found MST!**

**3 + 2 + 2 + 8 + 8 + 7 + 4 + 1 + 3 = 38**



# Kruskal's Algorithm

**Kruskal's** algorithm finds a safe edge to add to the growing forest by finding, of all the edges that connect any two trees in the forest, an edge $(\boldsymbol{u}, \boldsymbol{v})$ with the lowest weight.

```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ G.V                          O(V)
3      MAKE-SET(v)
4  create a single list of the edges in G.E         O(E)
5  sort the list of edges into monotonically increasing order by weight w   O(E log E)
6  for each edge (u, v) taken from the sorted list in order
7      if FIND-SET(u) ≠ FIND-SET(v)                 ~O(E)
8          A = A ∪ {(u, v)}
9          UNION(u, v)
10 return A
```

Start with an empty spanning tree.

Pick the edge with smallest weight.

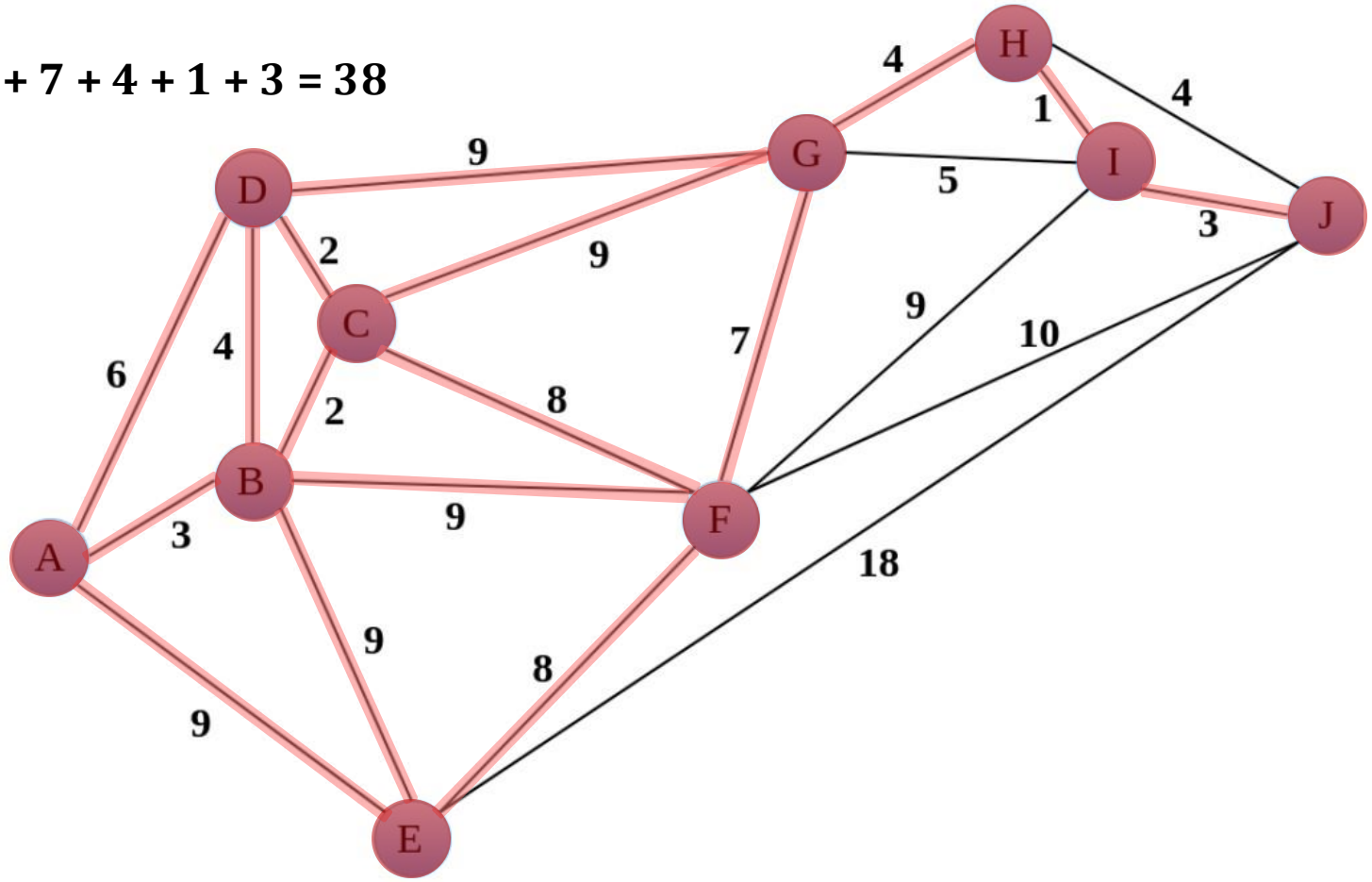
Check to see if it forms a cycle with the spanning tree you've created so far. If it does, discard it. If it doesn't, then add it to your spanning tree.

Repeat

# Prim's Algorithm

**We found MST!**

**3 + 2 + 2 + 8 + 8 + 7 + 4 + 1 + 3 = 38**



# Prim's Algorithm

**Prim's** has the property that the edges in the set A always form a single tree. At each step of the algorithm, the vertices in the tree determine a cut of the graph, and a light edge crossing the cut is added to the tree.

```
MST-PRIM(G, w, r)
1   for each vertex u ∈ G.V
2       u.key = ∞
3       u.π = NIL                                   O(V)
4   r.key = 0
5   Q = ∅
6   for each vertex u ∈ G.V
7       INSERT(Q, u)                                O(V)
8   while Q ≠ ∅
9       u = EXTRACT-MIN(Q)          // add u to the tree      O(v log v)
10      for each vertex v in G.Adj[u]   // update keys of u's non-tree neighbors
11          if v ∈ Q and w(u, v) < v.key
12              v.π = u
13              v.key = w(u, v)
14              DECREASE-KEY(Q, v, w(u, v))         O(E)
```

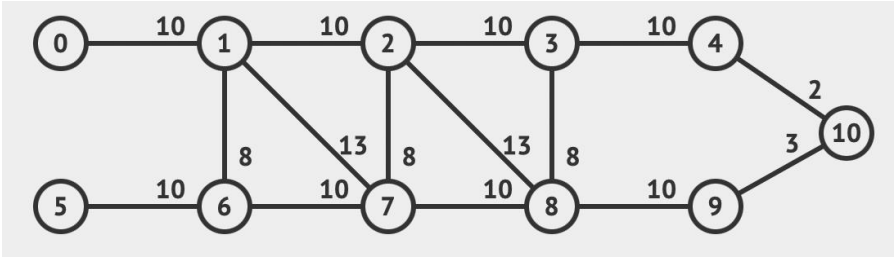
Start with an empty spanning tree, and any vertex v.

Set 1 contains vertex v, and Set 2 contains all other vertices.

At each step, consider all the edges that connect the two sets and pick the edge with minimum weight. Add that edge's other vertex to Set 1.

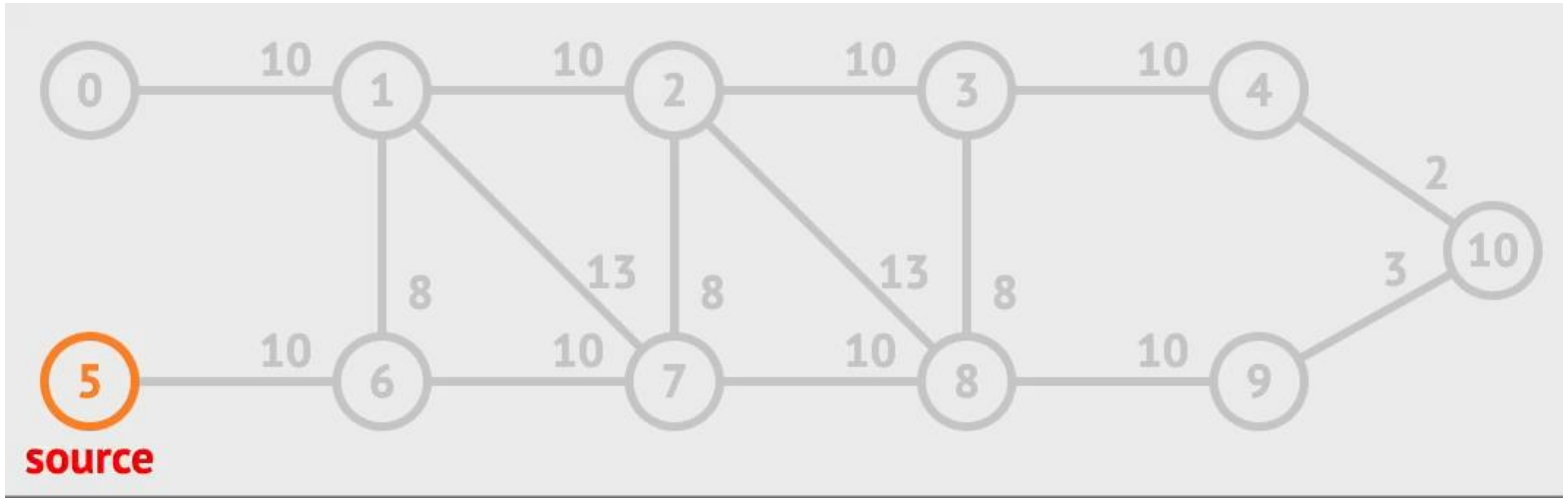
Repeat

# Question




In this question, you will apply Prim's Algorithm starting with vertex 5.
List all edges or show the edges produced by Prim's Algorithm. For example 0-1 means there is a link connecting region 0 and region 1.

**Determine the EXACT order in which these edges are added to form the MST.**



# Breakout Activity

10 Minutes

- If a graph is DENSE, should you use Prim or Kruskal? Why?
- If a graph is SPARSE, should you use Prim or Kruskal? Why?

**Hint:**

**Prim** runs in time **$O(V^2)$** with a simple array implementation, or **$O(E + V \log V)$** with a binary heap

**Kruksal** runs in time **$O(E \log V)$**

If an n-vertex graph is **dense** (i.e., E is close to $V^2$), then Prim's Algorithm is $O(v^2)$ while Kruskal's Algorithm is $O(v^2 \log v)$

If an n-vertex graph is **sparse** (i.e., E is a constant multiple of V), then Prim's Algorithm is $O(v^2)$ or $O(E + V \log V)$ while Kruskal's Algorithm is $O(v \log v)$. For sparse graphs, both algorithms perform similarly, with Kruskal's often having a slight edge.

# Why Kruskal's algorithm often is preferred over Prim's algorithm for sparse graphs?

1. **Asymptotic complexity:**
   - Kruskal's: O(E log E) or O(E log V)
   - Prim's with binary heap: O((V + E) log V)
   - For sparse graphs, E ≈ V, so both are roughly O(V log V)

2. **Constant factors and actual operations:**
   - Kruskal's algorithm sorts all edges once and then processes them sequentially. The sorting is efficient, and the subsequent operations (using a disjoint-set data structure) are very fast in practice.
   - Prim's algorithm requires maintaining a priority queue, with repeated extract-min and decrease-key operations. These operations, even with a binary heap, can be slightly more expensive in practice.

3. **Memory access patterns:**
   - Kruskal's algorithm tends to have better cache performance due to its more straightforward, sequential processing of sorted edges.
   - Prim's algorithm may have more scattered memory access patterns due to priority queue operations, potentially leading to more cache misses.

# Why Kruskal's algorithm often is preferred over Prim's algorithm for sparse graphs?

4. **Easier to implement efficiently**:
   - Kruskal's algorithm is typically simpler to code optimally.

5. **One-time edge processing:**
   - Kruskal examines each edge once after sorting; Prim may revisit edges.

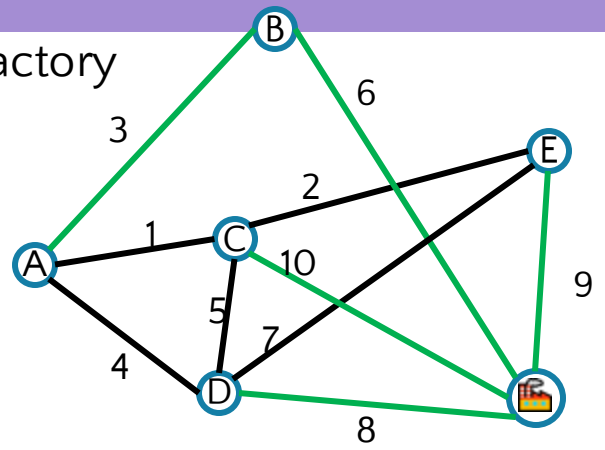
6. **Potential early termination**:
   - Kruskal can stop once enough edges are found; Prim always processes all vertices.

# Shortest Path vs Minimum Spanning

**Shortest Path Problem**

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

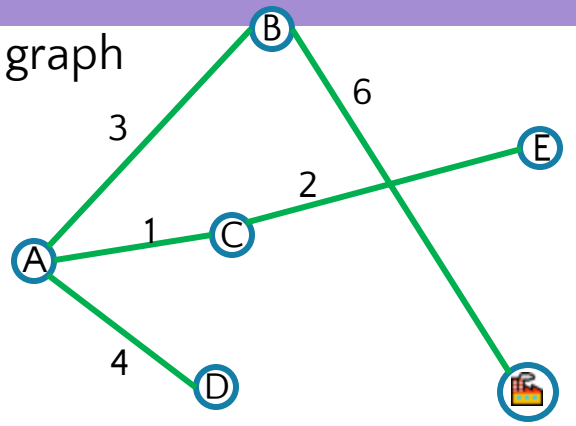
SPT from Factory




**Minimum Spanning Tree Problem**

**Given**: an undirected, weighted graph G
**Find**: A minimum-weight set of edges such that you can get from any vertex of G to any other on only those edges.

MST of the graph




**Shortest Path Tree**
Specific start node (if you have a different start node,
that changes the whole SPT, so there are multiple SPTs for graphs frequently)
Keeps track of total path length.

**Minimum Spanning Tree**
No specific start node, since the goal is just to minimize the edge weights sum. Often only one possible MST that has the minimum sum.
All nodes connected
Keeps track of cheapest edges that maintain connectivity

# 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?

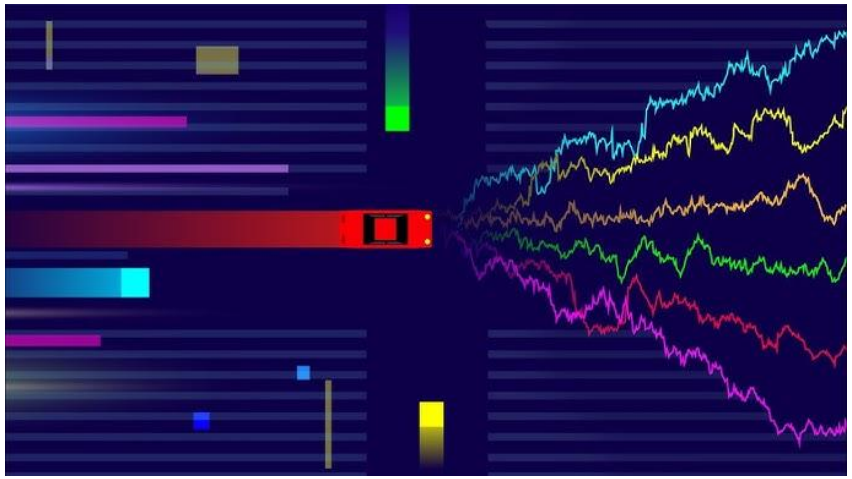
BFS + generate shortest path tree as we go

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

# Up Next: Week 7

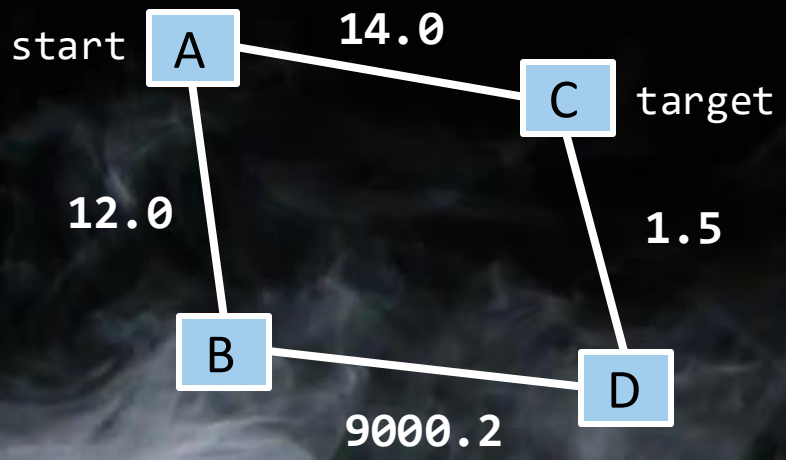
## Linear Programming & Online Algorithm


Introduction to Linear Programming



# *Later* 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?

# Questions?

Northeastern University