CSE 421 Section 2

Graph Search

Graph Modeling

Modeling a Problem

- In order to write an algorithm for a word problem, first we need to translate that word problem into a form that we can interact with more easily.
- Often, that means figuring out how to encode it into data structures and identifying what type of algorithm might work for solving it
- A common form this will take is **graph modeling**, turning the problem into a graph. This will let us use graph algorithms to help us find our solution.

Graph Modeling Steps

1. Ask **"what are my fundamental objects?"** (These are usually your vertices)
2. Ask **"how are they related to each other?"** (These are usually your edges)
   - Be sure you can answer these questions:
     - Are the edges directed or undirected?
     - Are the edges weighted? If so, what are the weights? Are negative edges possible?
     - The prior two usually warrant explicit discussion in a solution. You should also be able to answer, "are there self-loops and multi-edges", though often it doesn't need explicit mention in the solution.
3. Ask **"What am I looking for, is it encoded in the graph?"** Are you looking for a path in the graph? A short(-est) one or long(-est) one or any one? Or maybe an MST or something else?
4. Ask **"How do I find the object from 3?"** If you know how, great! Choose an algorithm you know. If not, can you design an algorithm?
5. If stuck on step 4, you might need to go back to step 1! Sometimes a different graph representation leads to better results, and you'll only realize it around step 3 or 4.

Writing Algorithms Using Existing Algorithms

- Often, a problem can be solved by using an existing algorithm in one of two ways:
    - **Reduction / Calling the existing algorithm** (like a library function) with some additional work before and/or after the call
    - **Modifying the existing algorithm** slightly
- Both are valid approaches, and which one you choose depends on the problem
- Whenever possible, it's a good idea to use ideas that you know work! You don't need to start from scratch to reinvent the wheel

Big-O

Big-O Review

- Big-O lets us analyze the runtime of algorithms as a function of the size of the input, usually denoted as n

- Super important for understanding algorithms and comparing them!
(so, you will be doing this analysis for every algorithm you write)

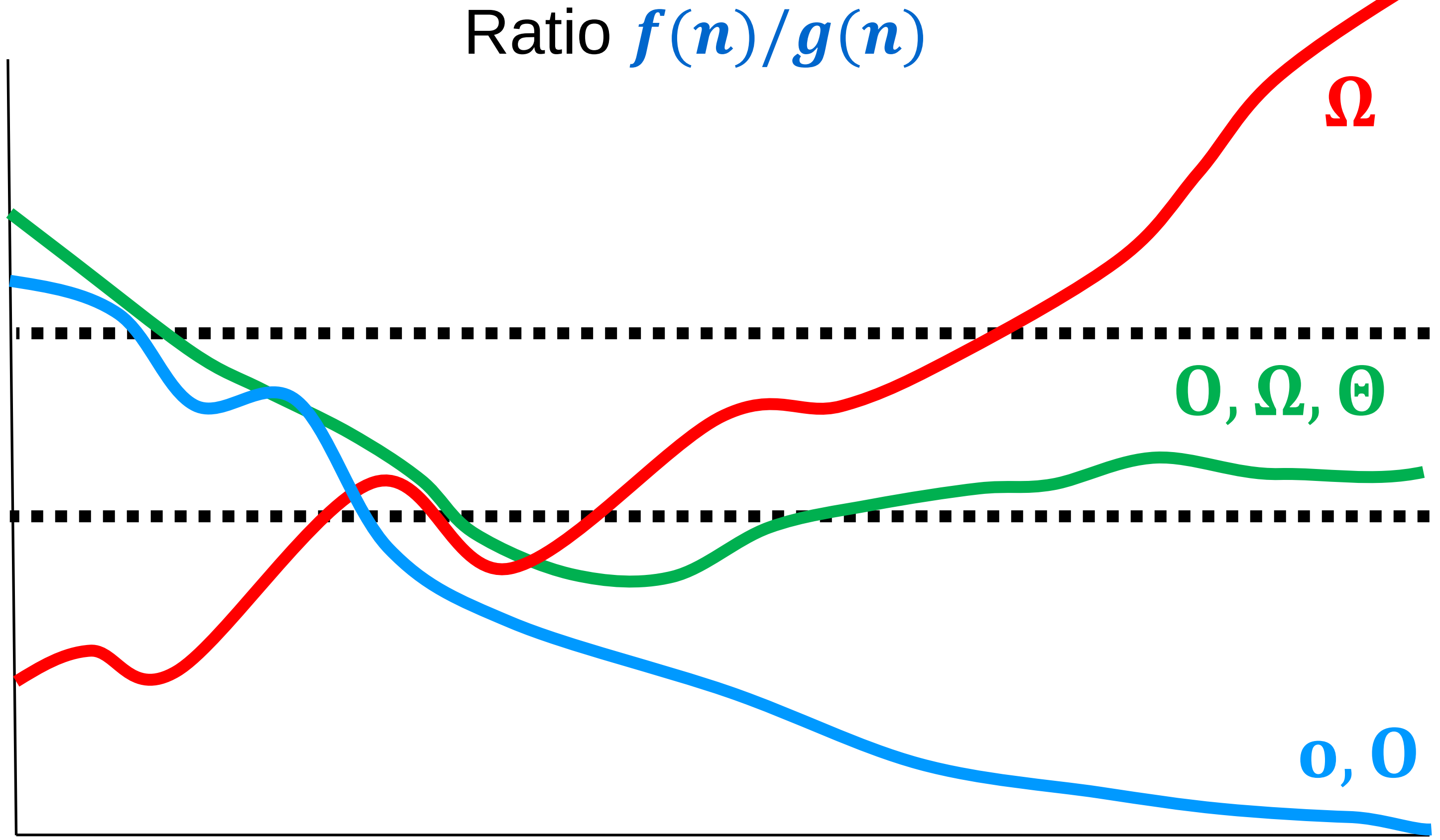
- Given two functions f and g:
  - $f(n)$ is $\boldsymbol{\mathcal{O}(g(n))}$ iff there is a constant $c > 0$ so that $f(n)/g(n)$is eventually always $\leq c$
  - $f(n)$ is $\boldsymbol{o(g(n))}$ iff $\lim_{n\to\infty} f(n)/g(n) = 0.$
  - $f(n)$ is $\boldsymbol{\Omega(g(n))}$ iff there is a constant $c > 0$ so that $f(n)/g(n)$ is eventually always $> c \cdot g(n)$
  - $f(n)$ is $\boldsymbol{\Theta(g(n))}$ iff there are constants $c_1, c_2 > 0$ so that eventually always $c_1 \cdot g(n) < f(n) < c_2 \cdot g(n)$

$\boldsymbol{\mathcal{O}(g(n))}$ is fancy $\leq$

$\boldsymbol{\Omega(g(n))}$ is fancy $\geq$

$\boldsymbol{\Theta(g(n))}$ is fancy $\approx$

Big-O Review



Big-O Tips for Comparing

- We're looking for asymptotic comparison, so just testing values won't necessarily give you a good idea
    - **Exponentials:** 2^n and 3^n are different, which means 2^n and $2^{n/2}$ are different! [constant factors IN EXPONENTS are not constant factors]
    - **Exponentials vs Polynomials:** for all $r > 1$ and all $d > 0$, $n^d = O(r^n)$ [in other words, every exponential grows faster than every polynomial]
    - **Logs vs Polynomials:** $\log^a(n)$ is asymptotically less than n^b for any positive constants a, b
- Key strategy: rewriting functions as $2^{f(n)}$ or $\log(f(n))$ will often make it easier to find the correct order for functions

Problem 1 – Big-O-No

Put these functions in increasing order. That is, if f comes before g in the list, it must be the case that $f(n)$ is $\mathcal{O}(g(n))$. Additionally, if there are any pairs such that $f(n)$ is $\Theta(g(n))$, mark those pairs.

- $2^{\log(n)}$
- $2^{n \log(n)}$
- $\log(\log(n))$
- $2^{\sqrt{n}}$
- $3^{\sqrt{n}}$
- $\log(n)$
- $\log(n^2)$
- $\sqrt{n}$
- $(\log(n))^2$

Hint: A useful trick in these problems is to know that since $\log(\cdot)$ is an increasing function, if $f(n)$ is $\mathcal{O}(g(n))$, then $\log(f(n))$ is $\mathcal{O}(\log(g(n))$. But be careful! Since $\log(\cdot)$ makes functions much smaller it can obscure differences between functions. For example, even though n^3 is less than n^4 , $\log(n^3)$ and $\log(n^4)$ are big-Θ of each other.

Work on this problem with the people around you, and then we'll go over it together!

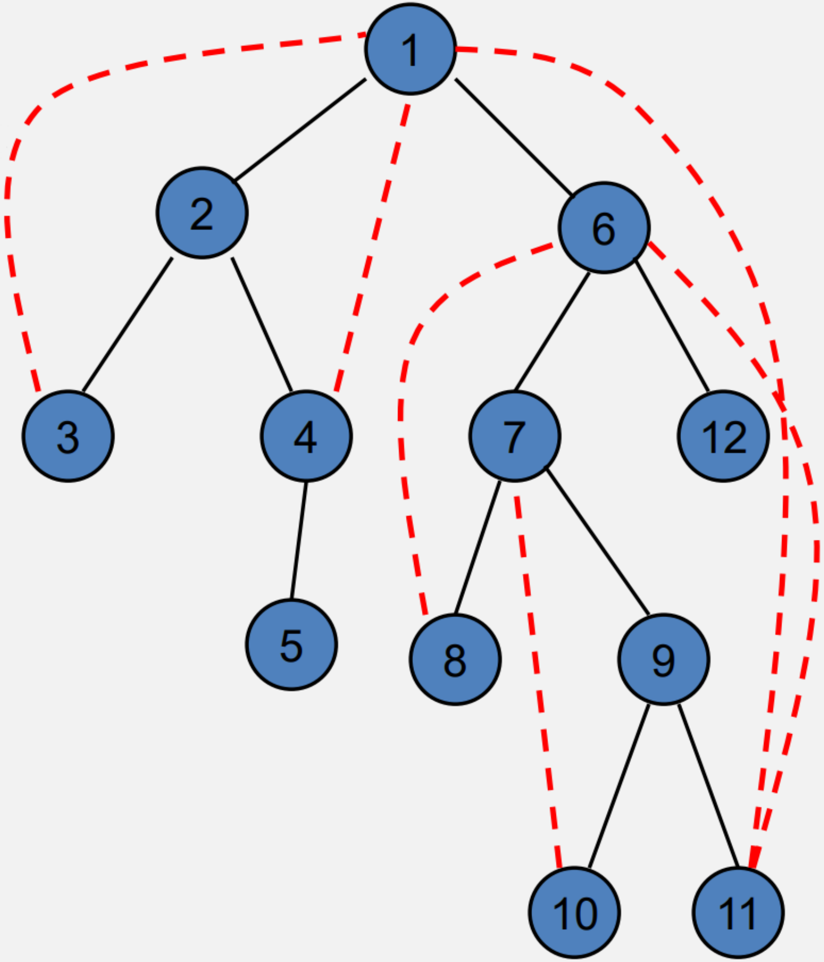
Graph

DFS Review

```
S = {s}
while S is not empty
        u = Pop(S)
        if u is unvisited
                visit u
                foreach v in N(u)
                        Push(S, v)
```

DFS Review

- Each edge goes between vertices on the same branch
- No cross edges

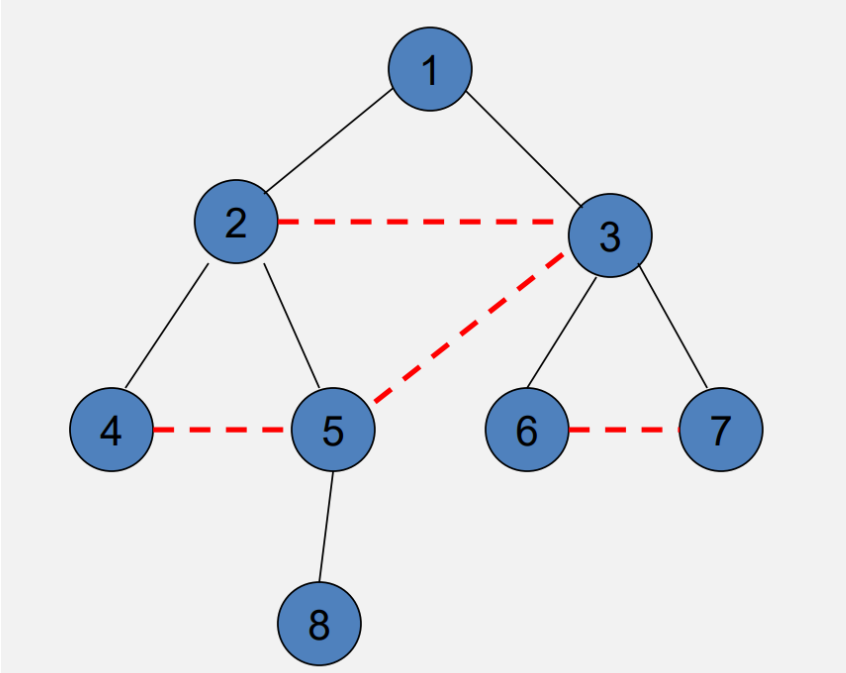


BFS Review

```
S = {s}
while S is not empty
        u = Dequeue(S)
        if u is unvisited
                visit u
                foreach v in N(u)
                        Enqueue(S, v)
```

BFS Review

- All edges go between vertices on the same layer or adjacent layers

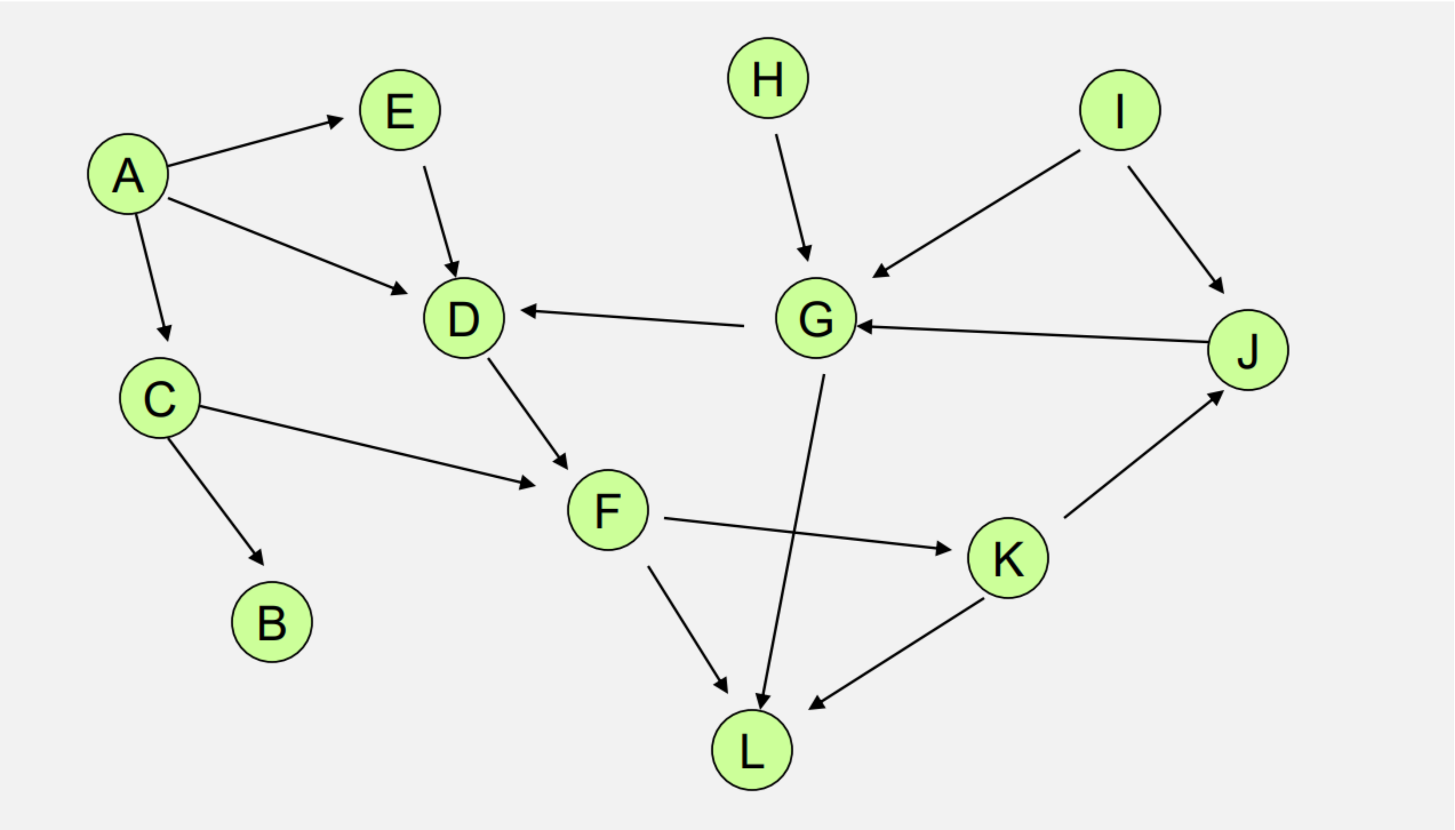


Topological Sort

- Given a set of tasks with precedence constraints, find a linear order of the tasks.
- If a graph has a cycle, there is no topological sort

```
While there exists a vertex v with in-degree 0
        Output vertex v
        Delete the vertex v and all out going edges
```

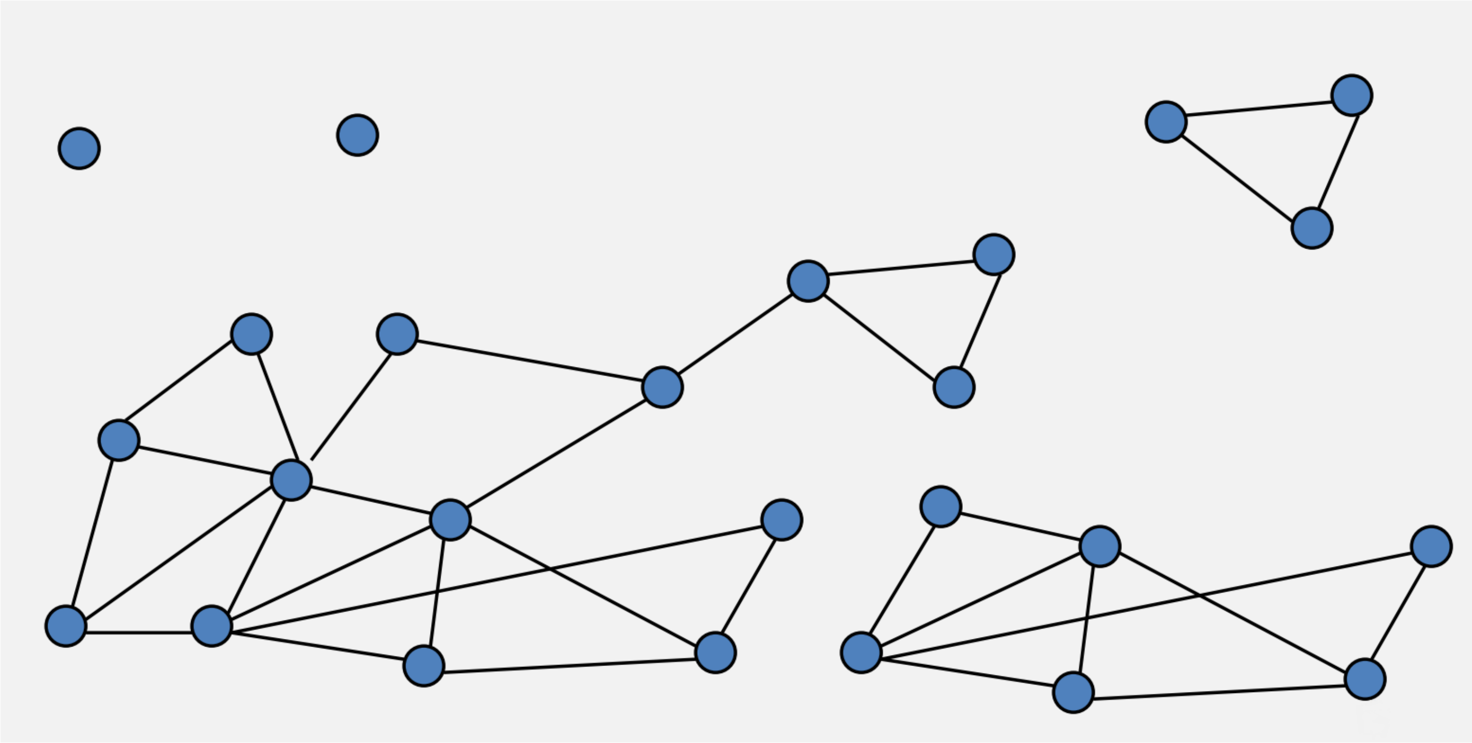
- O(n+m)


A
B
C
D
E
F
G
H
I
J
K
L

Connected Components

- A connected subgraph that is not part of any larger connected subgraph.
- *For undirected graph

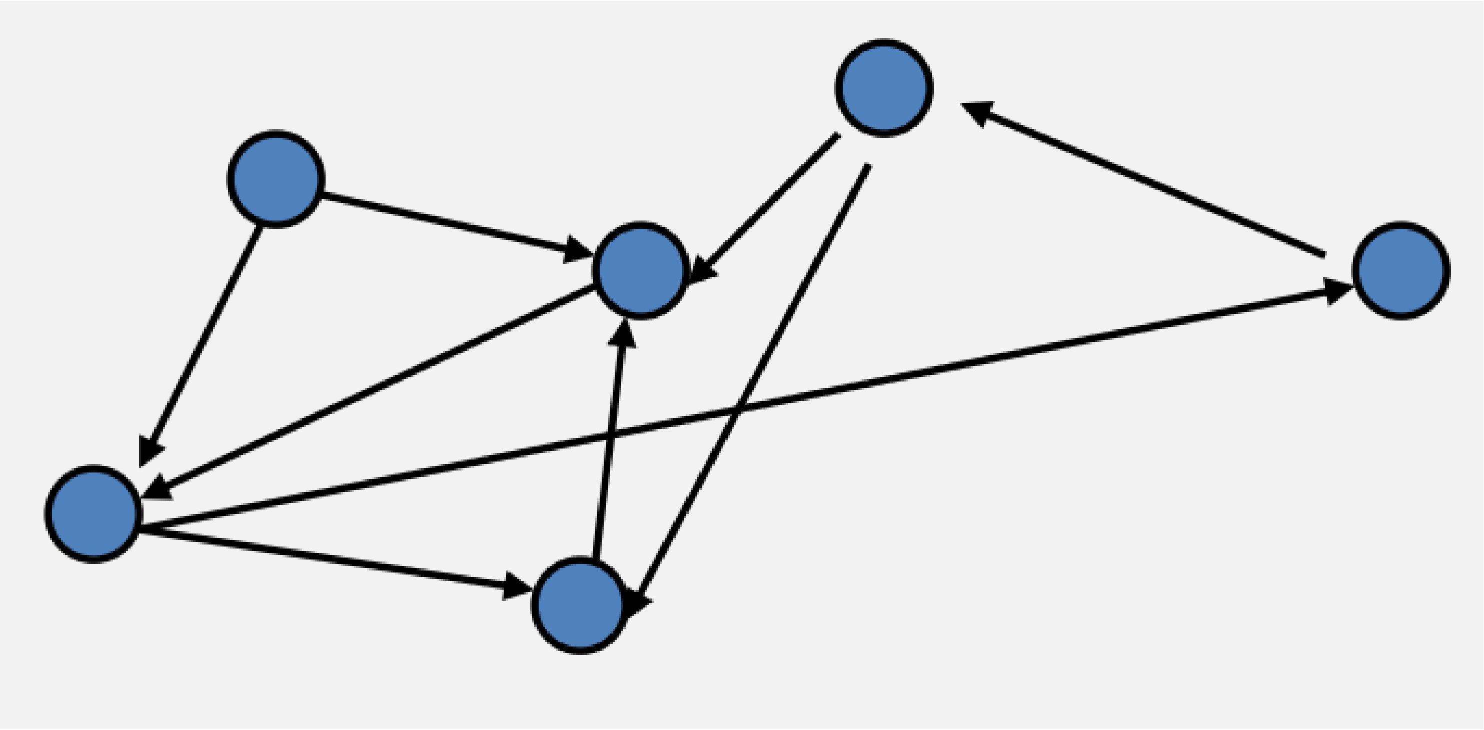
Find all connected components



Strongly Connected Components

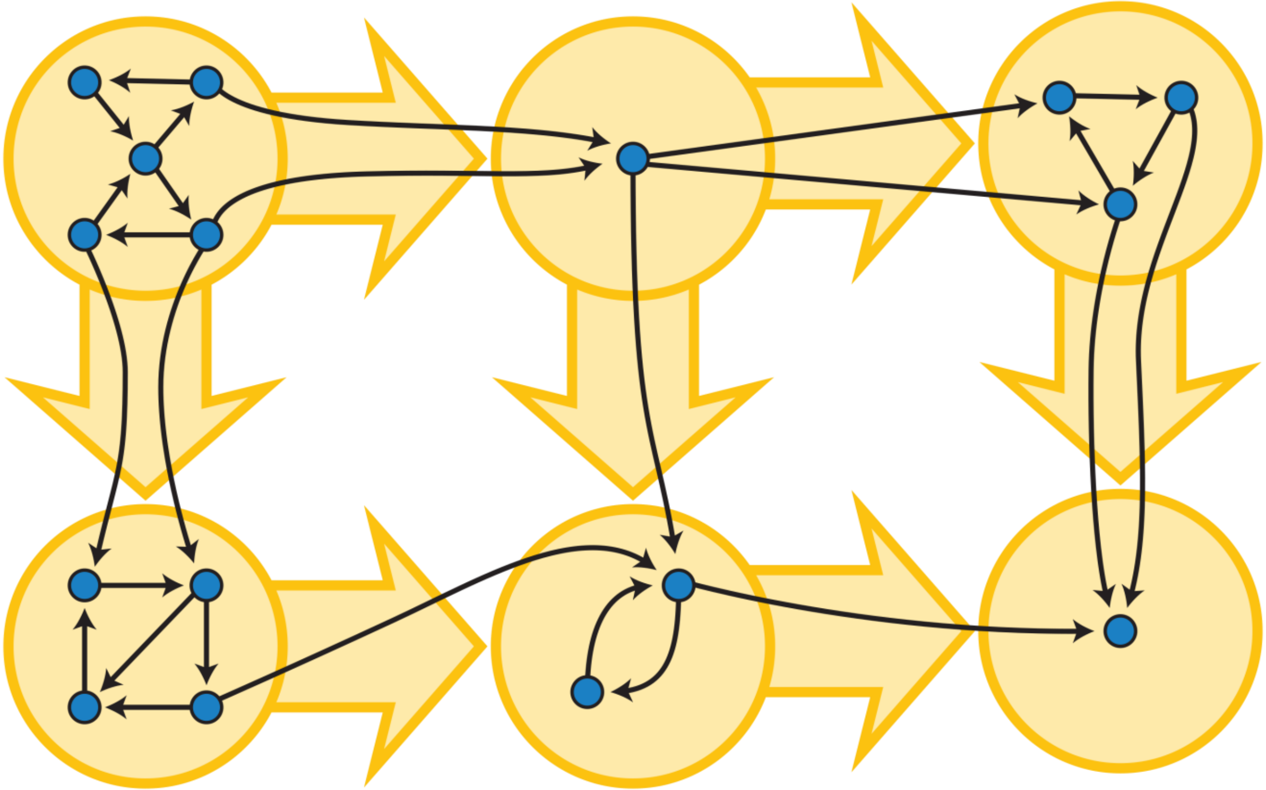
- A subset of the vertices with paths between every pair of vertices
- *For directed graph

Try to separate the graph into SCCs



Fun Fact

- If each strongly connected component is ”contracted” to a single vertex, the resulting graph is a directed acyclic graph (DAG)



That's All, Folks!

Thanks for coming to section this week!
Any questions?