

# **INTRODUCTION TO GRAPHS**

**SAYED AHMAD SAHIM 26-11.2017**

# GRAPHS

- **Subject of our next couple weeks of the course**

# GRAPHS

- **Subject of our next couple weeks of the course**
  - New theoretical framework
- **Many different problems and algorithms stem from graph theory**

# GRAPHS

- **Subject of our next couple weeks of the course**
  - New theoretical framework
- **Many different problems and algorithms stem from graph theory**
- **Allows us to conceptualize new types of problems**

# GRAPHS

- **A graph is composed of two things**
  - A set of vertices
  - A set of edges (which are ordered vertex tuples)

# GRAPHS

- **A graph is composed of two things**
  - A set of vertices
  - A set of edges (which are ordered vertex tuples)
- **Trees are types of graphs**

# GRAPHS

- **A graph is composed of two things**
  - A set of vertices
  - A set of edges (which are ordered vertex tuples)
- **Trees are types of graphs**
  - Each of the nodes is a vertex

# GRAPHS

- **A graph is composed of two things**
  - A set of vertices
  - A set of edges (which are ordered vertex tuples)
- **Trees are types of graphs**
  - Each of the nodes is a vertex
  - Each pointer from parent to child is an edge



# GRAPHS

- **A graph is composed of two things**
  - A set of vertices
  - A set of edges (which are ordered vertex tuples)
- **Trees are types of graphs**
  - Each of the nodes is a vertex
  - Each pointer from parent to child is an edge
- **Represented as  $G(V,E)$  to indicate that  $V$  is the set of vertices and  $E$  is the set of edges**

# GRAPHS

- **Trees are acyclic graphs**

# GRAPHS

- **Trees are acyclic graphs**
- **Graphs can be traversed**

# GRAPHS

- **Trees are acyclic graphs**
- **Graphs can be traversed**
  - Breadth-first
  - Depth-first

# GRAPHS

- **Trees are acyclic graphs**
- **Graphs can be traversed**
  - Breadth-first
  - Depth-first
- **Edges can also have weights**

# GRAPHS

- **Trees are acyclic graphs**
- **Graphs can be traversed**
  - Breadth-first
  - Depth-first
- **Edges can also have weights**
  - Path finding on a map

# GRAPHS

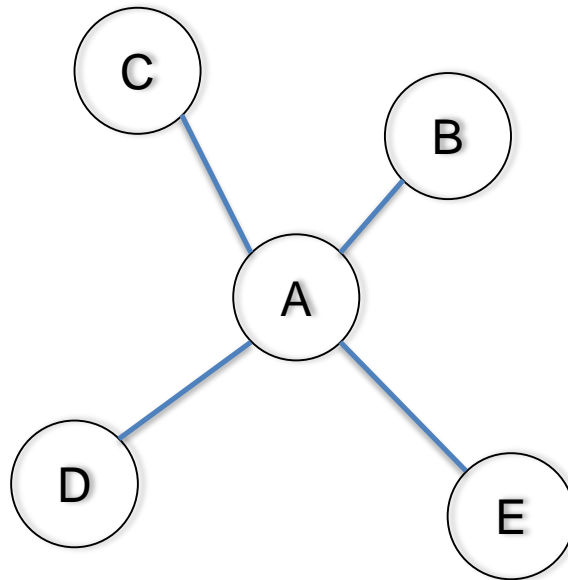
- **Trees are acyclic graphs**
- **Graphs can be traversed**
  - Breadth-first
  - Depth-first
- **Edges can also have weights**
  - Path finding on a map
  - Route-optimization problems

# GRAPHS

- **Graphs are not an ADT**
  - There is no “functions” that a graph supports
  - Rather, graphs are a theoretical framework for understanding certain types of problems.
  - Travelling salesman, path finding, resource allocating

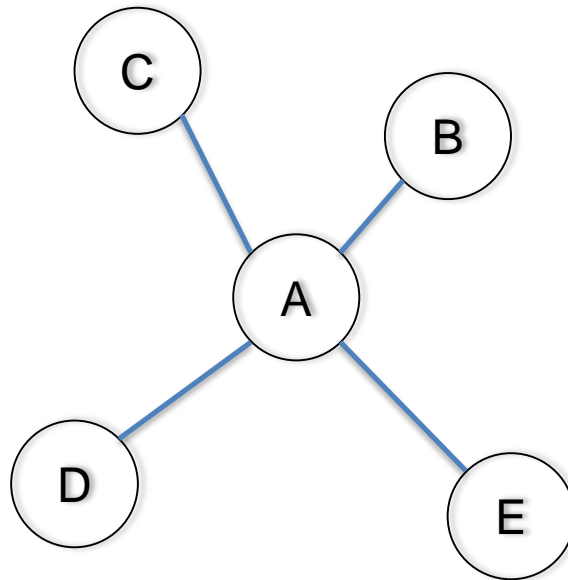


# GRAPHS



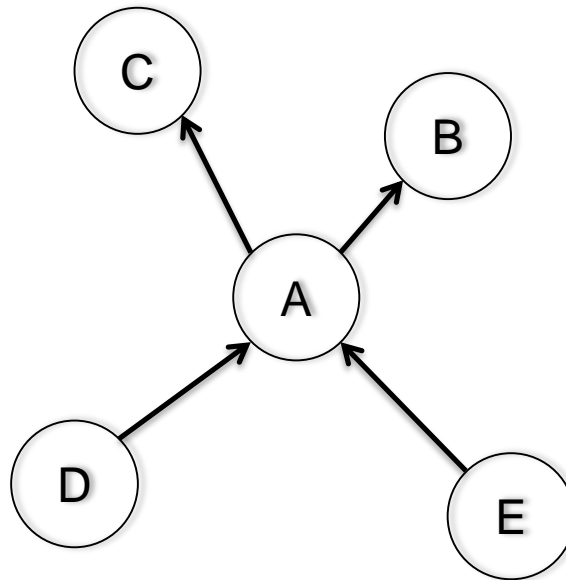
- What this graphs vertices and edges?

# GRAPHS



- What this graphs vertices and edges?
  - $V = \{A, B, C, D, E\}$
  - $E = \{(A,B), (A,C), (A,D), (A,E)\}$

# GRAPHS



- What this graphs vertices and edges?
  - $V = \{A, B, C, D, E\}$
  - $E = \{(A,B) , (A,C), (D,A), (E,A)\}$

# ANALYZING GRAPHS

- In graphs, there are two important variables,  $|V|$  and  $|E|$ 
  - Our analysis can now have two inputs
  - Before, our input size was  $n$ , now we use  $|V|$  and  $|E|$

# ANALYZING GRAPHS

- In graphs, there are two important variables,  $|V|$  and  $|E|$ 
  - Our analysis can now have two inputs
  - Before, our input size was  $n$ , now we use  $|V|$  and  $|E|$
  - What is the maximum size of  $|E|$ ?

# ANALYZING GRAPHS

- In graphs, there are two important variables,  $|V|$  and  $|E|$ 
  - Our analysis can now have two inputs
  - Before, our input size was  $n$ , now we use  $|V|$  and  $|E|$
  - What is the maximum size of  $|E|$ ?
    - For any vertices  $a, b$ , there can exist at most one edge  $(a, b)$

# ANALYZING GRAPHS

- In graphs, there are two important variables,  $|V|$  and  $|E|$ 
  - Our analysis can now have two inputs
  - Before, our input size was  $n$ , now we use  $|V|$  and  $|E|$
  - What is the maximum size of  $|E|$ ?
    - For any vertices  $a, b$ , there can exist at most one edge  $(a, b)$
    - $A$  can equal  $B$  (this is a self loop)
    - There can be  $(b, a)$  -- directed

# ANALYZING GRAPHS

- In graphs, there are two important variables,  $|V|$  and  $|E|$ 
  - Our analysis can now have two inputs
  - Before, our input size was  $n$ , now we use  $|V|$  and  $|E|$
  - What is the maximum size of  $|E|$ ?  $O(|V|^2)$ 
    - For any vertices  $a, b$ , there can exist at most one edge  $(a, b)$
    - $A$  can equal  $B$  (this is a self loop)
    - There can be  $(b, a)$  -- directed



# GRAPHS

- **Paths and Cycles**

- A path: a set of edges connecting two vertices where all of the edges are connected and neither edges nor vertices are repeated
- A cycle: a path that starts and ends on the same

# GRAPHS

- In paths and cycles, the sets of edges must pass from one vertex to another, i.e. each edge must share a vertex with some other edge.

# GRAPHS

- In paths and cycles, the sets of edges must pass from one vertex to another, i.e. each edge must share a vertex with some other edge.
  - $(A,B) (B,C)$  is a path from A to C, while
  - $(A,B) (D,C)$  is not.

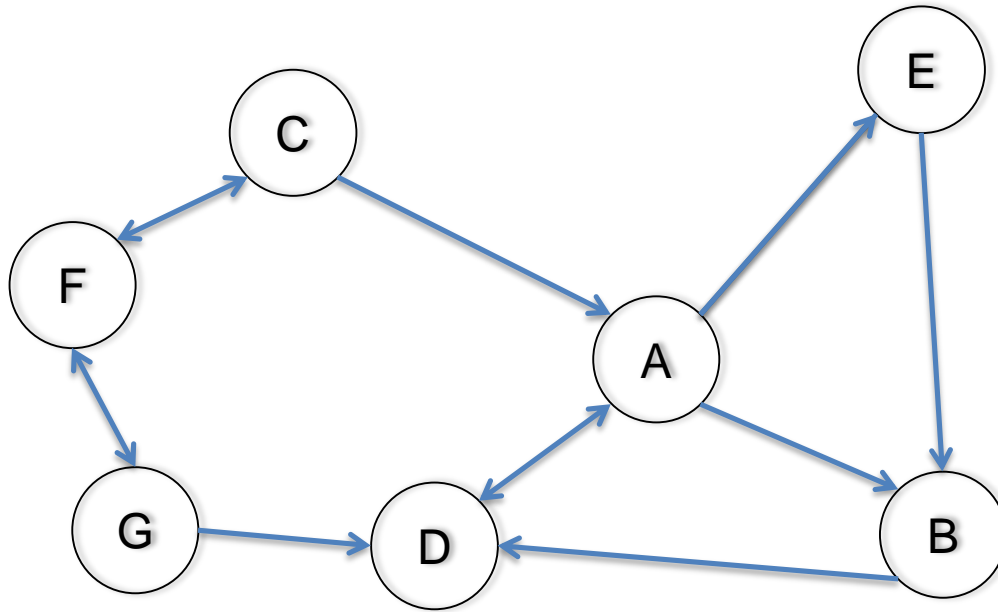
# GRAPHS

- In paths and cycles, the sets of edges must pass from one vertex to another, i.e. each edge must share a vertex with some other edge.
  - $(A,B) (B,C)$  is a path from A to C, while
  - $(A,B) (D,C)$  is not.
  - There is no way to get from B to D

# GRAPHS

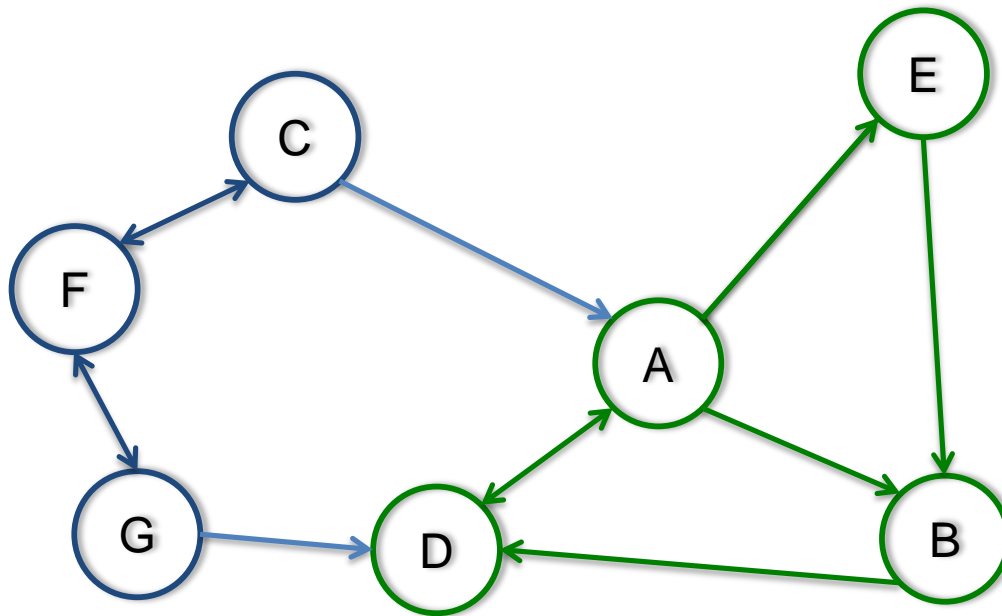
- **Graphs can be either directed or undirected**
  - Undirected graph, if  $(A,B)$  is in the set of edges,  $(B,A)$  must be in the set of edges
  - Directed graphs, both can be in the set of edges, but those graphs have different connectivity
- **We call a graph *connected* if there is a path between every pair of vertices**

# GRAPHS



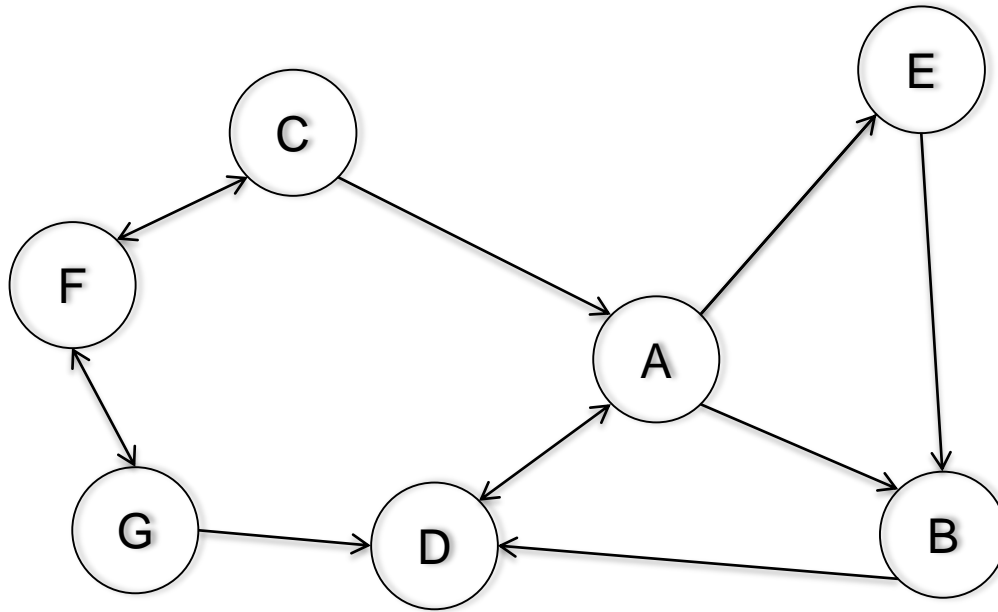
- **Is this graph connected?**
  - Is there a path between every pair of vertices?

# GRAPHS



- **Is this graph connected?**
  - There's no way to get from the green graph to the red

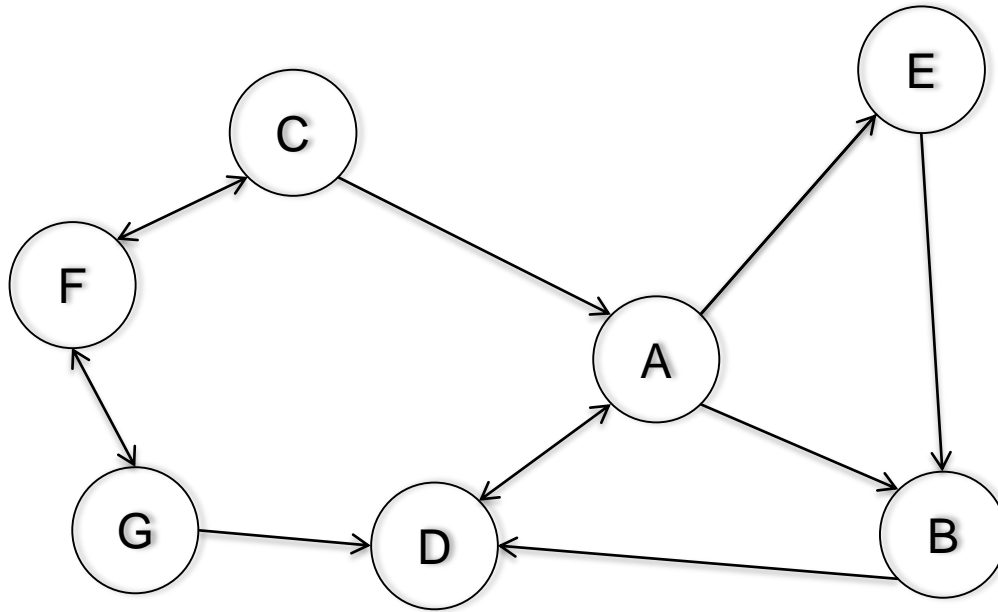
# GRAPHS



- **Does this graph have a cycle?**
  - How many does it have?



# GRAPHS



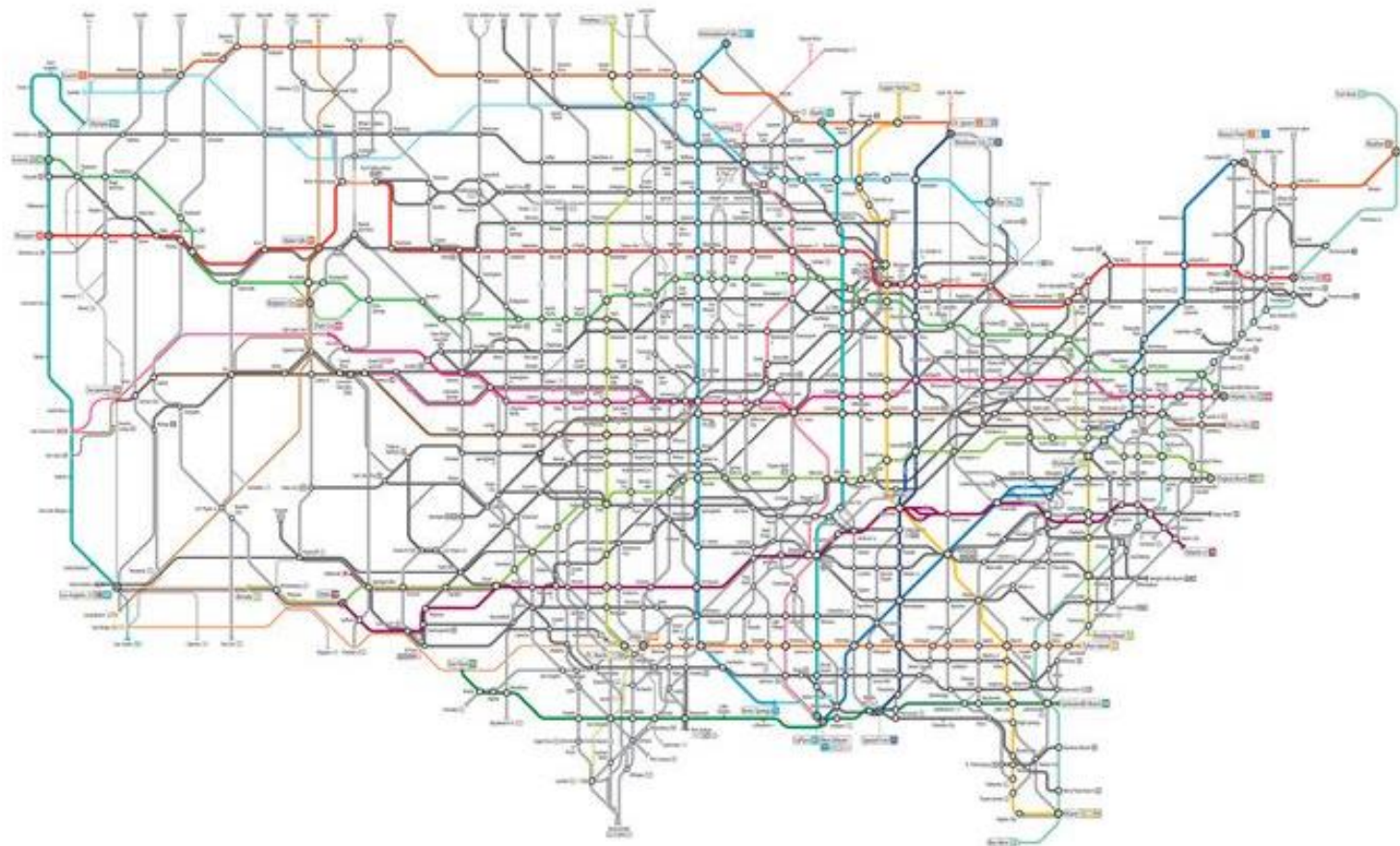
- Does this graph have a cycle?
  - $\{(A,E),(E,B),(B,D),(D,A)\}$
  - $\{(A,B),(B,D),(D,A)\}$

# GRAPHS

- **Paths and cycles can not have repeated vertices or edges**
  - A path that can repeat vertices or edges is called a walk
  - A path that can repeat vertices but not edges is called a trail
  - A circuit is a trail that starts and ends at the same vertex

# GRAPHS

- **Edges can have weights**
  - This becomes important when we consider path finding algorithms
  - Usually, we consider the weights to be the costs of using a particular edge.
  - In a graph representation of the US interstate system, the I-90 edge between Seattle and Spokane may have weight 270 for miles or 4 for hours, depending on what we want to minimize!



Orange: Primary  
 Red: Secondary  
 Blue: Interstate  
 Green: State  
 Yellow: County  
 Purple: Local

Blue: Interstate  
 Green: State  
 Yellow: County  
 Purple: Local

Blue: Interstate  
 Green: State  
 Yellow: County  
 Purple: Local

Blue: Interstate  
 Green: State  
 Yellow: County  
 Purple: Local

Blue: Interstate  
 Green: State  
 Yellow: County  
 Purple: Local

Blue: Interstate  
 Green: State  
 Yellow: County  
 Purple: Local

Blue: Interstate  
 Green: State  
 Yellow: County  
 Purple: Local

Blue: Interstate  
 Green: State  
 Yellow: County  
 Purple: Local

Blue: Interstate  
 Green: State  
 Yellow: County  
 Purple: Local

Blue: Interstate  
 Green: State  
 Yellow: County  
 Purple: Local

Blue: Interstate  
 Green: State  
 Yellow: County  
 Purple: Local

Blue: Interstate  
 Green: State  
 Yellow: County  
 Purple: Local

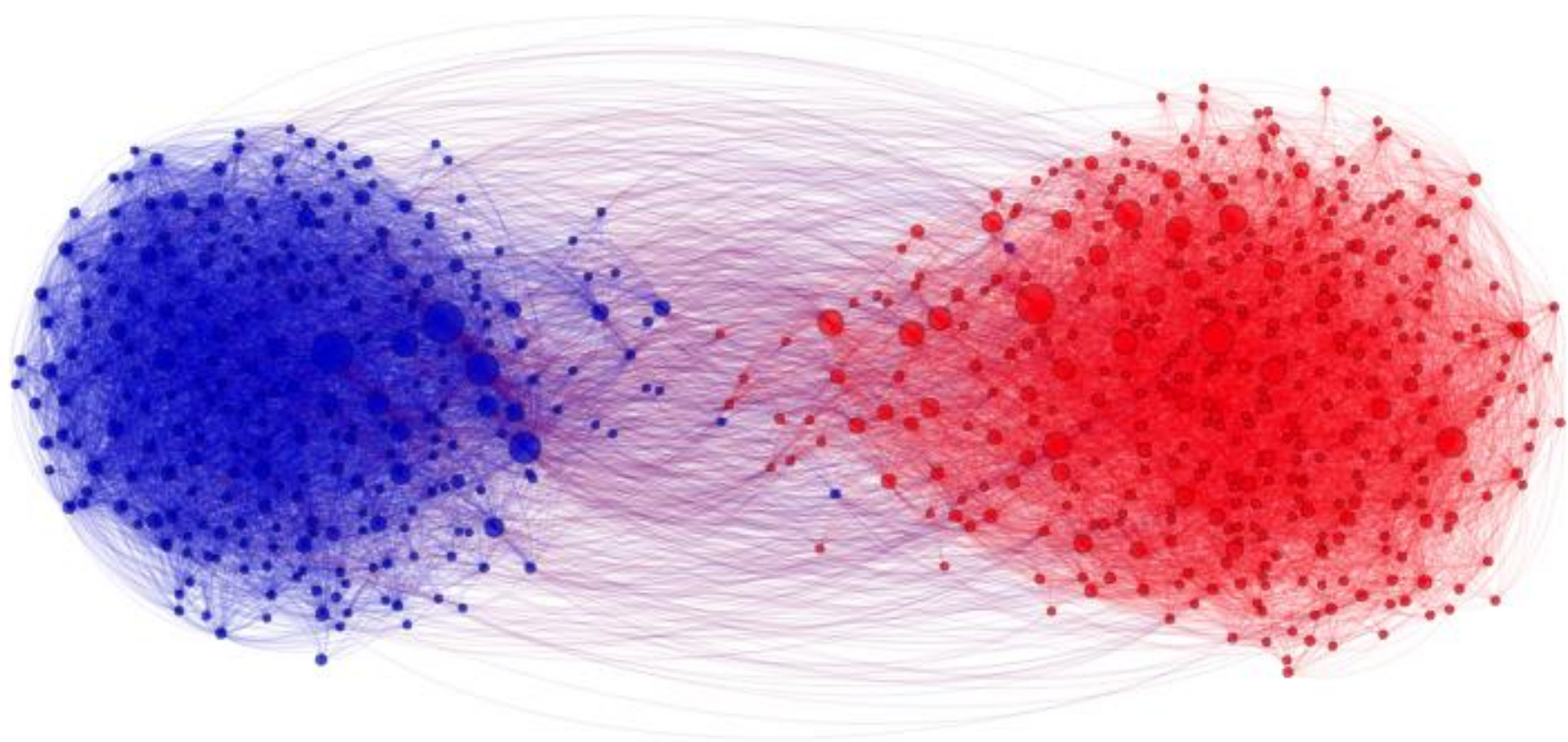
Blue: Interstate  
 Green: State  
 Yellow: County  
 Purple: Local

# GRAPHS

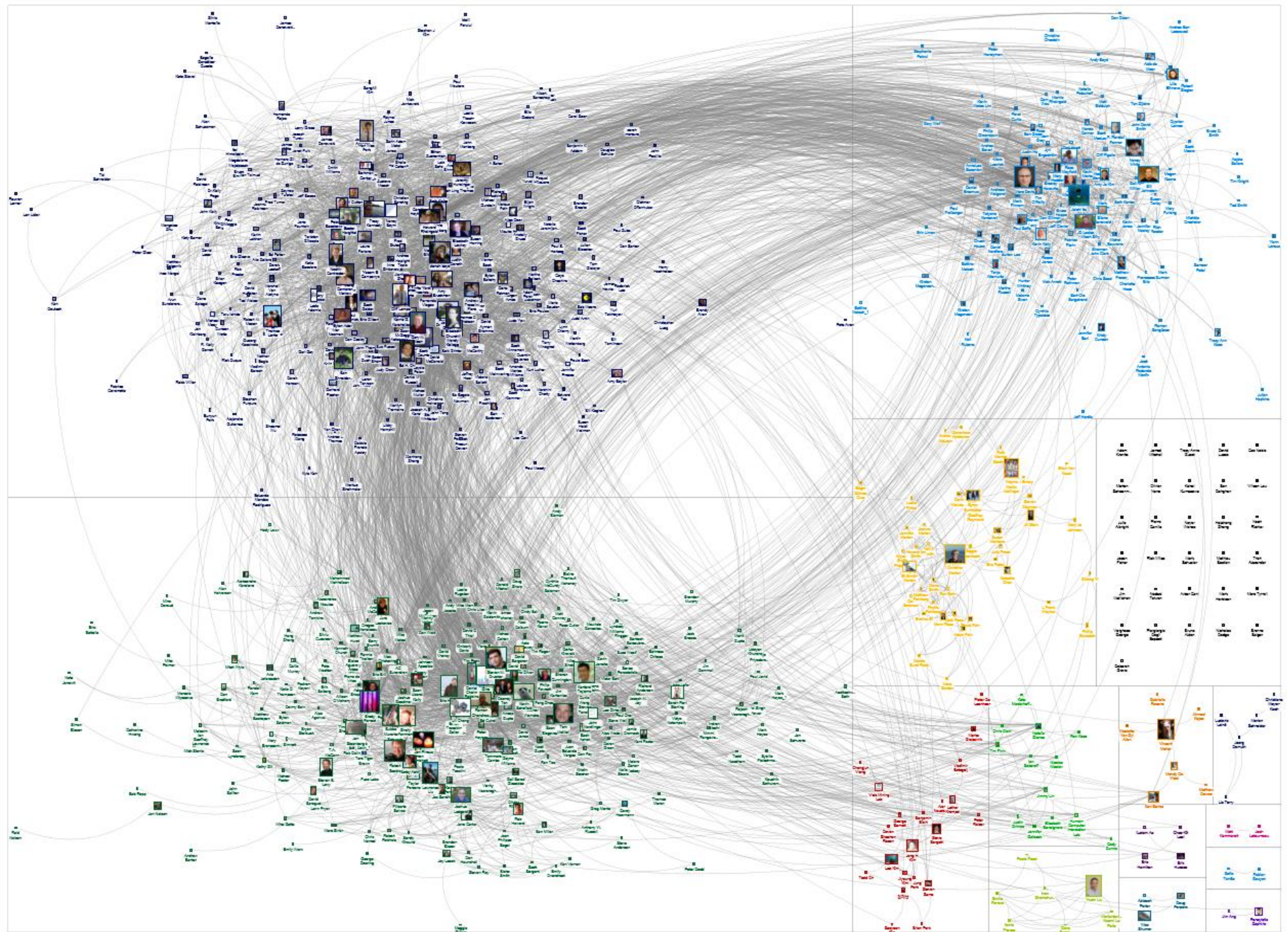
- **When we consider graphs, we determine them to be either dense or sparse**
  - Dense graphs are very connected, each vertex is connected to a fraction of the total vertices
  - Sparse graphs are less connected and can be more clustered, each vertex is connected to some constant number of vertices

# GRAPHS

- Is data often dense?









# GRAPHS

- **When graphs are small, it is difficult to distinguish between the two**
  - If we represent Facebook as a graph, where users are vertices and “friendships” are edges, what can we say about the graph?
    - Directed?
    - Connected?
    - Cyclic?
    - Sparse/Dense?

# GRAPHS

- **When graphs are small, it is difficult to distinguish between the two**
  - If we represent Facebook as a graph, where users are vertices and “friendships” are edges, what can we say about the graph?
    - Directed? **No, (A,B) means (B,A)**
    - Connected? **Maybe not!**
    - Cyclic? **Yes, mutual friends**
    - Sparse/Dense? **Sparse! 338 average!**

# GRAPHS

- **This “value” is called the degree of the vertex**
  - If you have 338 friends, then that vertex has degree 338.
- **In directed graph, we separate this into in-degree and out-degree**
  - Consider Twitter, where friendship isn't symmetric. The number of followers you have is your in-degree and the number of people you follow is your out degree

# REPRESENTATION

- **How do we represent graphs on a computer?**
  - Two main approaches

# REPRESENTATION

- **How do we represent graphs on a computer?**
  - Two main approaches
    - Adjacency List
    - Adjacency Matrix

# ADJACENCY LIST

- If  $(u,v)$  is an edge, then we say  $v$  is ***adjacent*** to  $u$ .
- If we want to store these edges then,
  - For each vertex, we maintain a list of all edges coming *out* of that vertex

# ADJACENCY LIST

- If  $(u,v)$  is an edge, then we say  $v$  is ***adjacent*** to  $u$ .
- If we want to store these edges then,
  - For each vertex, we maintain a list of all edges coming *out* of that vertex
- The number of elements coming out of the vertex is called the *out-degree*
- The number of elements coming into the vertex is the *in-degree*

# ADJACENCY LIST

- How much memory does this consume?



# ADJACENCY LIST

- How much memory does this consume?
  - $O(|V|+|E|)$ : There must be a list for each vertex, plus each edge must be in a list

# ADJACENCY LIST

- How much memory does this consume?
  - $O(|V|+|E|)$ : There must be a list for each vertex, plus each edge must be in a list
- What is a problem with the adjacency list?

# ADJACENCY LIST

- How much memory does this consume?
  - $O(|V|+|E|)$ : There must be a list for each vertex, plus each edge must be in a list
- What is a problem with the adjacency list?
  - Can't retrieve edges pointing **to** that vertex

# ADJACENCY LIST

- How much memory does this consume?
  - $O(|V|+|E|)$ : There must be a list for each vertex, plus each edge must be in a list
- What is a problem with the adjacency list?
  - Can't retrieve edges pointing **to** that vertex
  - Two-lists possible

# ADJACENCY MATRIX

- Imagine a two dimensional  $|V| \times |V|$  matrix.
- Let the rows be source vertices, and let the rows be destination vertices

# ADJACENCY MATRIX

- Imagine a two dimensional  $|V| \times |V|$  matrix.
- Let the rows be source vertices, and let the rows be destination vertices
  - If the edge  $(u,v)$  is in the graph, then  $\text{matrix}[u][v]$  is set to true
  - Alternatively, we can set  $\text{matrix}[u][v]$  to be the weight of the edge

# ADJACENCY MATRIX

- Imagine a two dimensional  $|V| \times |V|$  matrix.
- Let the rows be source vertices, and let the rows be destination vertices
  - If the edge  $(u,v)$  is in the graph, then  $\text{matrix}[u][v]$  is set to true
  - Alternatively, we can set  $\text{matrix}[u][v]$  to be the weight of the edge
- What is the memory consumption?

# ADJACENCY MATRIX

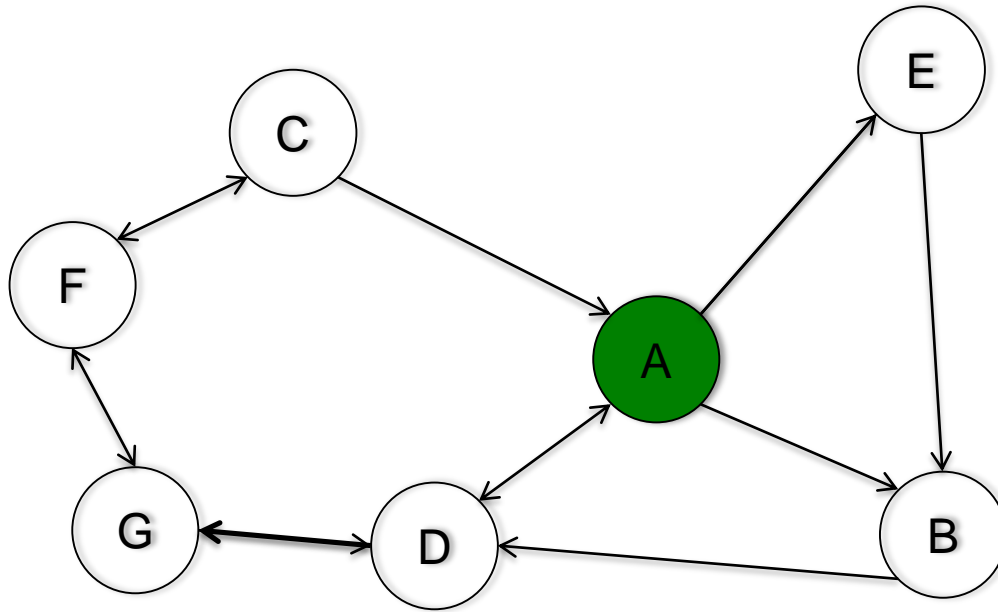
- Imagine a two dimensional  $|V| \times |V|$  matrix.
- Let the rows be source vertices, and let the rows be destination vertices
  - If the edge  $(u,v)$  is in the graph, then  $\text{matrix}[u][v]$  is set to true
  - Alternatively, we can set  $\text{matrix}[u][v]$  to be the weight of the edge
- What is the memory consumption?
  - $O(|V|^2)$ , but it implicitly stores in and out vertices
  - If the graph is dense, then this is more efficient



# TRAVERSALS

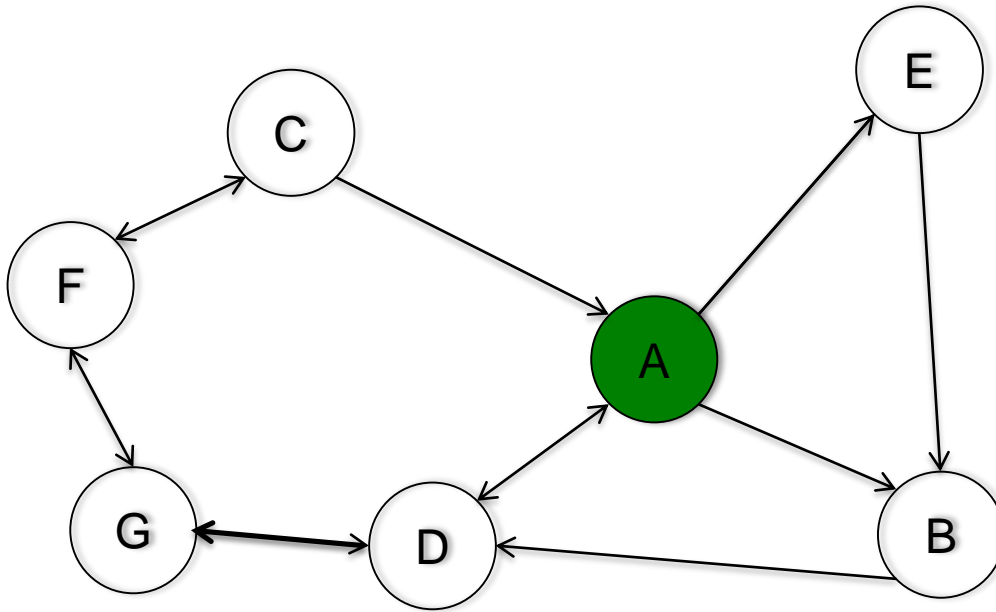
- **Since graphs are abstractions similar to trees, we can also perform traversals.**
  - If a graph is connected, i.e. there is a path between all pairs of vertices, then a traversal can output all nodes if you do it cleverly

# TRAVERSAL



- **Depth-first search (prev graph with (D,G) added to make it connected**
  - Traverse the tree with DFS, if there are multiple nodes to choose from, go alphabetically. Start at A.

# TRAVERSAL

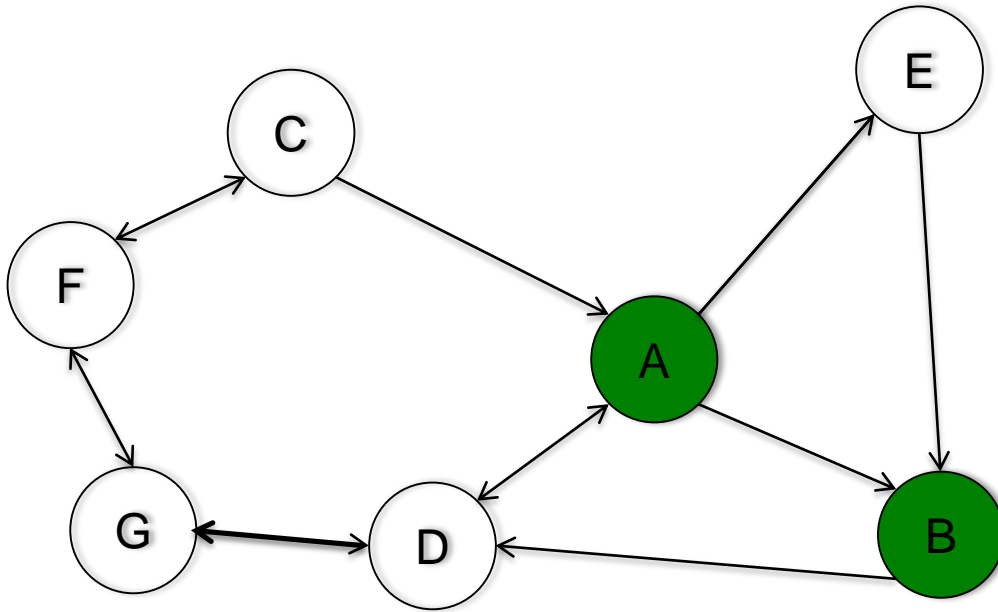


Output: A

Current Node: A

Out-vertices: B, D, E

# TRAVERSAL

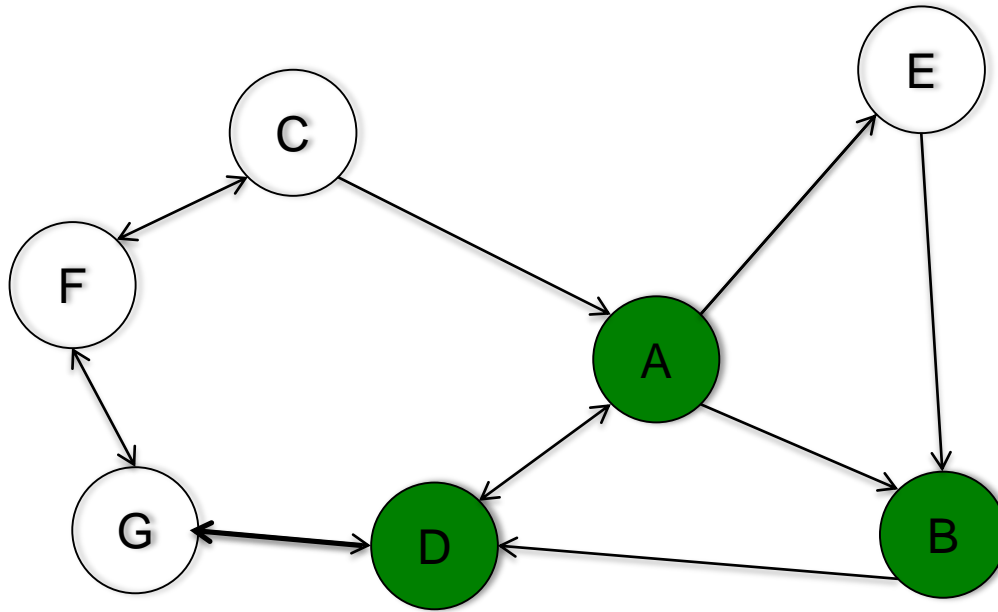


Output: A,B

Current Node: B

Out-vertices: D

# TRAVERSAL

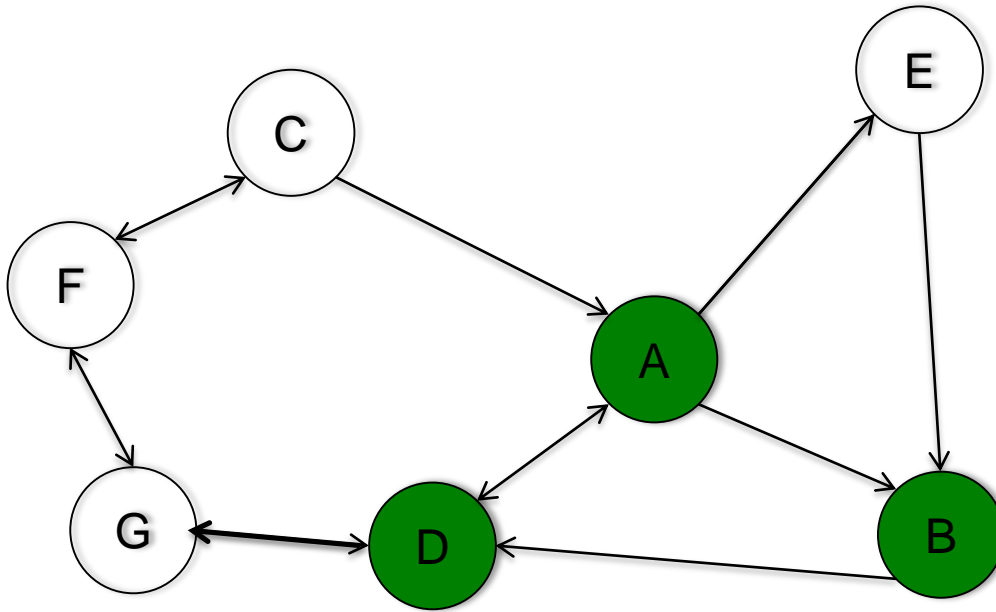


Output: A,B, D

Current Node: D

Out-vertices: A,G

# TRAVERSAL

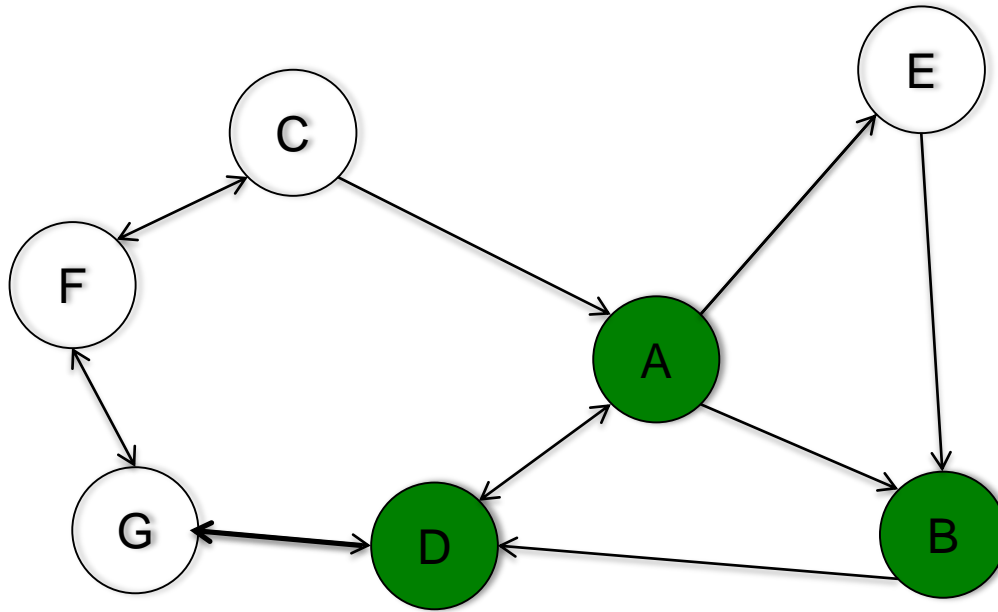


Output: A,B, D, A

Current Node: A

Out-vertices: B,D,E

# TRAVERSAL



Output: A,B, D, A

**Oh, no! We have repeated output!**

Current Node: A

Out-vertices: B,D,E

# TRAVERSAL

- **Depth first search needs to check which nodes have been output or else it can get stuck in loops.**
  - This increases the runtime and memory constraints of the traversal
- **In a connected graph, a BFS will print all nodes, but it will repeat if there are cycles and may not terminate**



# TRAVERSAL

- **As an aside, in-order, pre-order and post-order traversals only make sense in binary trees, so they aren't important for graphs. However, we do need some way to order our out-vertices (left and right in BST).**

# TERMINOLOGY

- **Know the following terms**
  - Vertices and Edges
  - Directed v. Undirected
  - In-degree and out-degree
  - Connected
  - Weighted v. unweighted
  - Cyclic v. acyclic
  - DAG: Directed Acyclic Graph

# TRAVERSALS

- **For an arbitrary graph and starting node  $v$ , find all nodes *reachable* from  $v$ .**
  - There exists a path from  $v$
  - Doing something or “processing” each node
  - Determines if an undirected graph is connected?  
If a traversal goes through all vertices, then it is connected
- **Basic idea**
  - Traverse through the nodes like a tree
  - Mark the nodes as visited to prevent cycles and from processing the same node twice

# ABSTRACT IDEA IN PSEUDOCODE

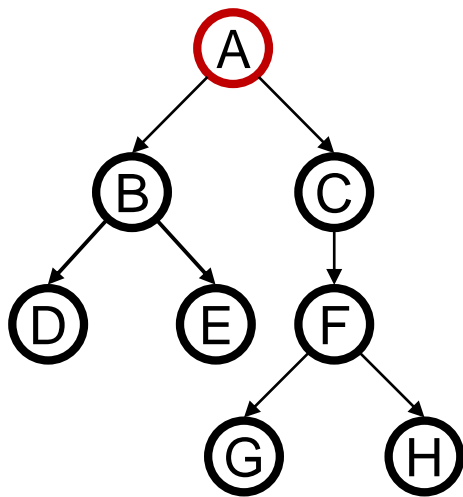
```
void traverseGraph(Node start) {  
    Set pending = emptySet()  
    pending.add(start)  
    mark start as visited  
    while (pending is not empty) {  
        next = pending.remove()  
        for each node u adjacent to next  
            if (u is not marked visited) {  
                mark u  
                pending.add(u)  
            }  
    }  
}
```

# **RUNTIME AND OPTIONS**

- **Assuming we can add and remove from our “pending” DS in  $O(1)$  time, the entire traversal is  $O(|E|)$**
- **Our traversal order depends on what we use for our pending DS.**
  - **Stack : DFS**
  - **Queue: BFS**
- **These are the main traversal techniques in CS, but there are others!**

# EXAMPLE: TREES

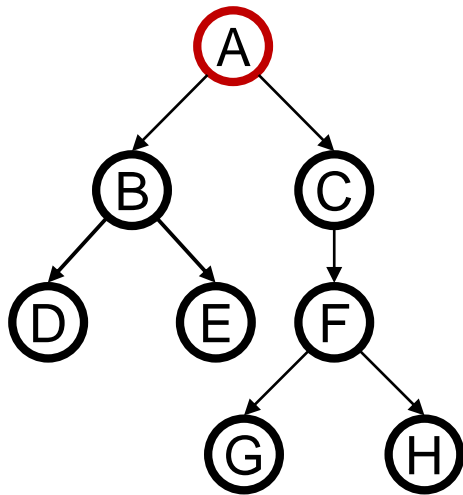
A tree is a graph and make DFS and BFS are easier to “see”



```
DFS(Node start) {  
    mark and process start  
    for each node u adjacent to start  
        if u is not marked  
            DFS(u)  
}
```

- A, B, D, E, C, F, G, H
- Exactly what we called a “pre-order traversal” for trees
  - The marking is because we support arbitrary graphs and we want to process each node exactly once

# EXAMPLE: TREES



```
DFS2(Node start) {  
    initialize stack s to hold start  
    mark start as visited  
    while (s is not empty) {  
        next = s.pop() // and "process"  
        for each node u adjacent to next  
            if (u is not marked)  
                mark u and push onto s  
    }  
}
```

- A, C, F, H, G, B, E, D
- A different but perfectly fine depth traversal

# COMPARISON

**Breadth-first always finds shortest length paths, i.e., “optimal solutions”**

- Better for “what is the shortest path from  $x$  to  $y$ ”

**But depth-first can use less space in finding a path**

- If *longest path* in the graph is  $p$  and highest out-degree is  $d$  then DFS stack never has more than  $d \cdot p$  elements
- But a queue for BFS may hold  $O(|V|)$  nodes

**A third approach (useful in Artificial Intelligence)**

- *Iterative deepening (IDFS)*:
  - Try DFS but disallow recursion more than  $\kappa$  levels deep
  - If that fails, increment  $\kappa$  and start the entire search over
- Like BFS, finds shortest paths. Like DFS, less space.



# TRAVERSAL

PAGE 3

DEPARTMENT	COURSE	DESCRIPTION	PREREQS
COMPUTER SCIENCE	CPSC 432	INTERMEDIATE COMPILER DESIGN, WITH A FOCUS ON DEPENDENCY RESOLUTION.	CPSC 432

# TRAVERSAL

PAGE 3

DEPARTMENT	COURSE	DESCRIPTION	PREREQS
COMPUTER SCIENCE	CPSC 432	INTERMEDIATE COMPILER DESIGN, WITH A FOCUS ON DEPENDENCY RESOLUTION.	CPSC 432

- It's never too late to start your xkcd addiction

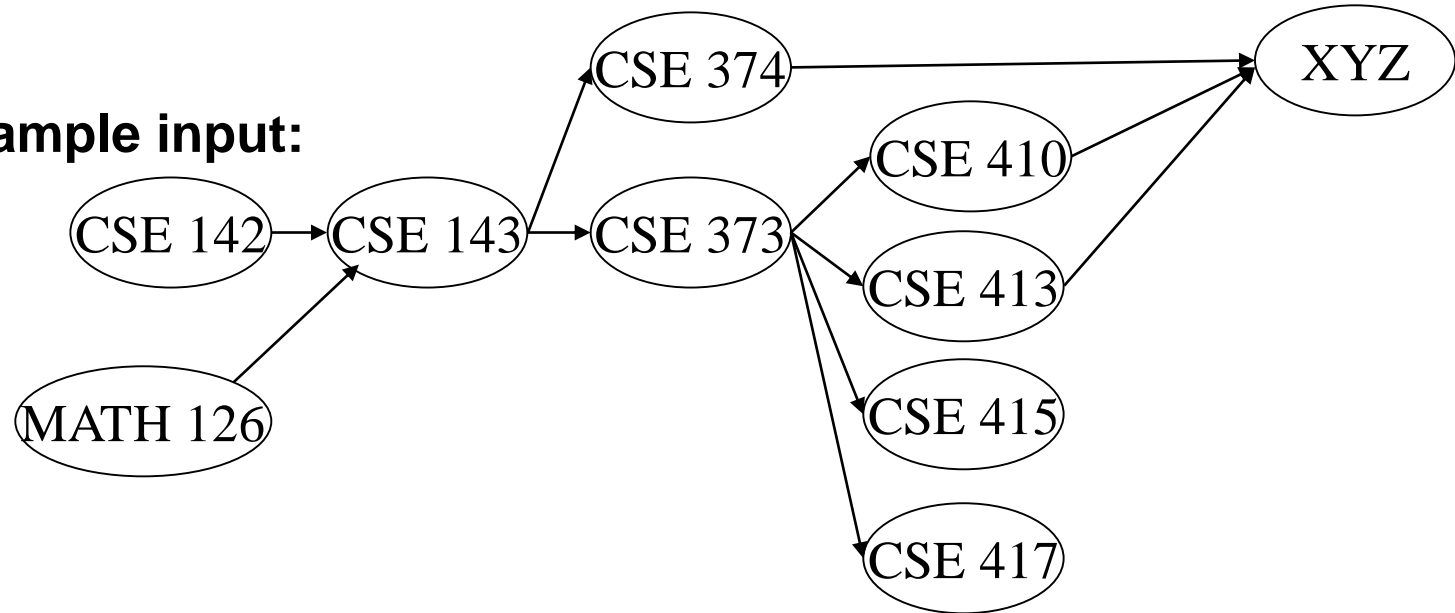
# TRAVERSAL

- **Topological ordering**
  - One final ordering for graphs
  - Ordering with a focus on dependency resolutions
- **Example, consider a graph where courses are vertices and prerequisites are edges.**
- **A topological ordering is any valid class order**

# TOPOLOGICAL SORT

**Problem:** Given a DAG  $G = (V, E)$ , output all vertices in an order such that no vertex appears before another vertex that has an edge to it

**Example input:**



**One example output:**

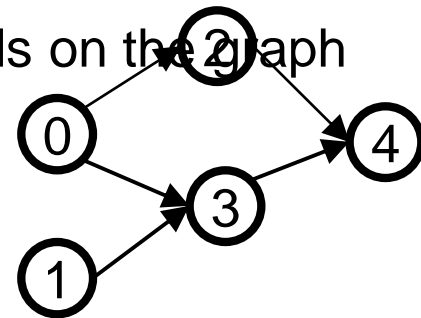
# QUESTIONS AND COMMENTS

**Why do we perform topological sorts only on DAGs?**

- Because a cycle means there is no correct answer

**Is there always a unique answer?**

- No, there can be 1 or more answers; depends on the graph
- Graph with 5 topological orders:



**Do some DAGs have exactly 1 answer?**

- Yes, including all lists

**Terminology:** A DAG represents a **partial order** and a topological sort produces a **total order** that is consistent with it

# USES OF TOPOLOGICAL SORT

**Figuring out how to graduate**

**Computing an order in which to recompute cells in a spreadsheet**

**Determining an order to compile files using a Makefile**

**In general, taking a dependency graph and finding an order of execution**

**...**

# TOPOLOGICAL SORT

## 1. Label (“mark”) each vertex with its in-degree

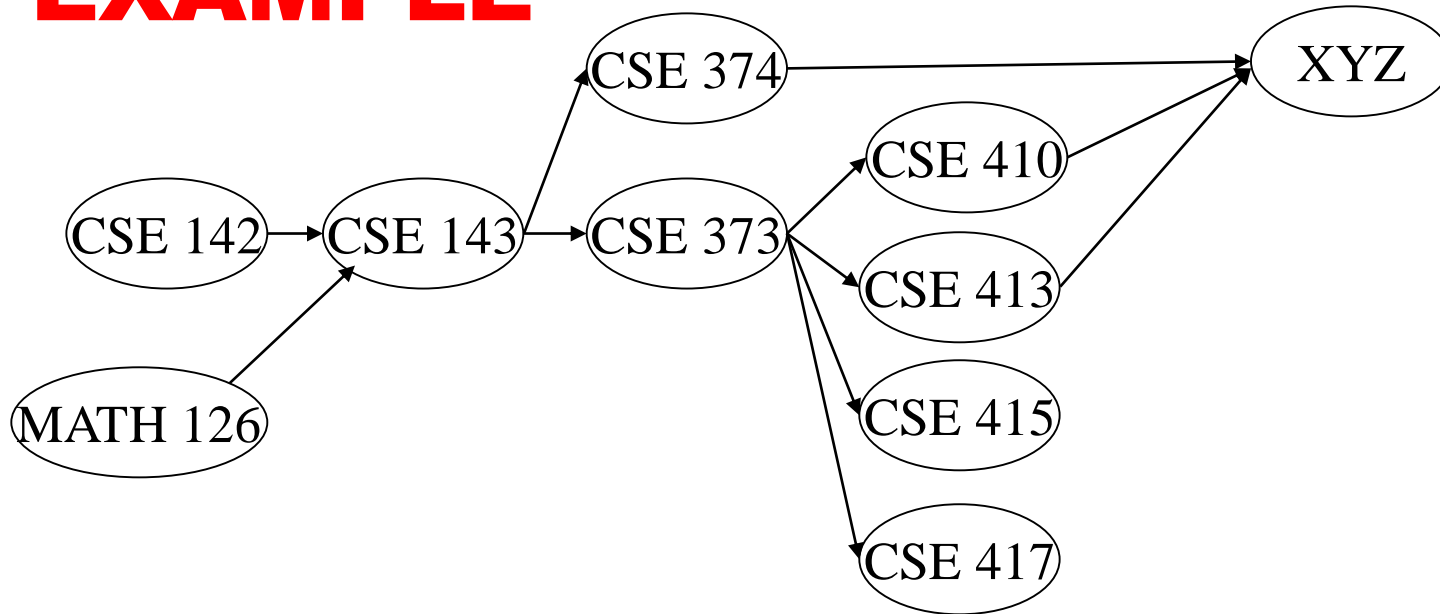
- Think “write in a field in the vertex”
- Could also do this via a data structure (e.g., array) on the side

## 1. While there are vertices not yet output:

- a) Choose a vertex  $v$  with labeled with in-degree of 0
- b) Output  $v$  and *conceptually* remove it from the graph
- c) For each vertex  $u$  adjacent to  $v$  (i.e.  $u$  such that  $(v,u)$  in  $\mathbf{E}$ ), decrement the in-degree of  $u$

# EXAMPLE

Output:



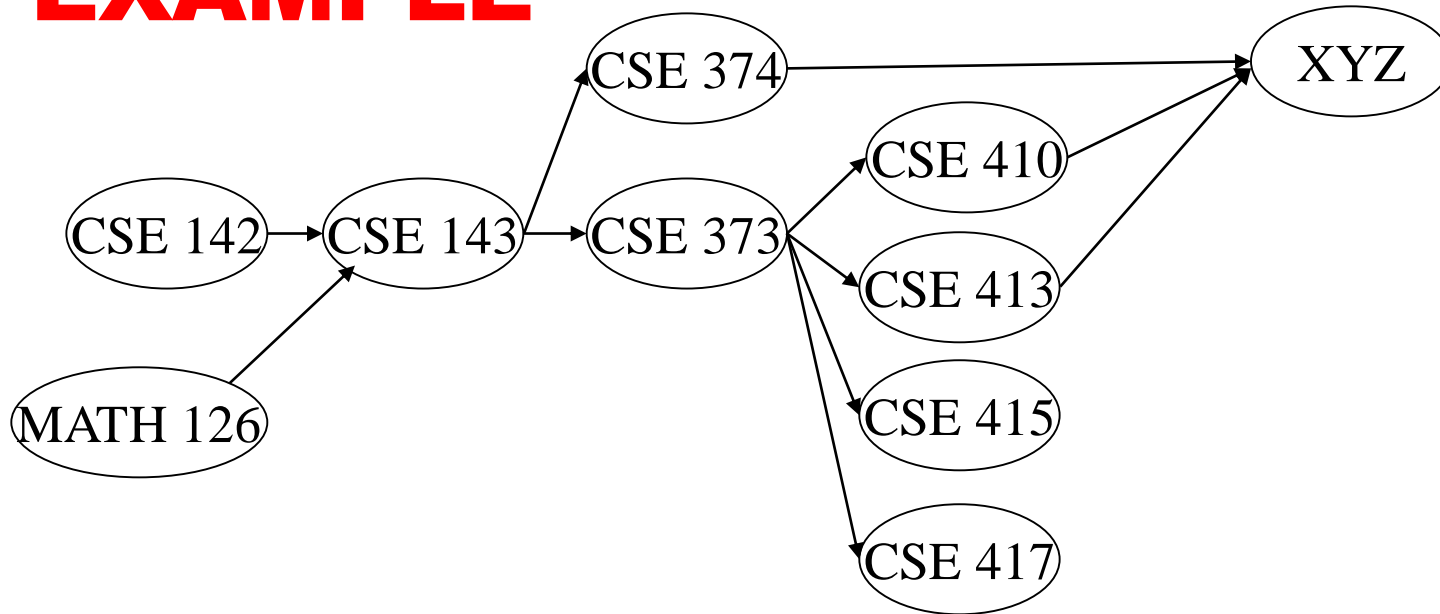
Node: 126 142 143 374 373 410 413 415 417 XYZ

Removed?

In-degree: 0 0 2 1 1 1 1 1 1 3



# EXAMPLE



Output:

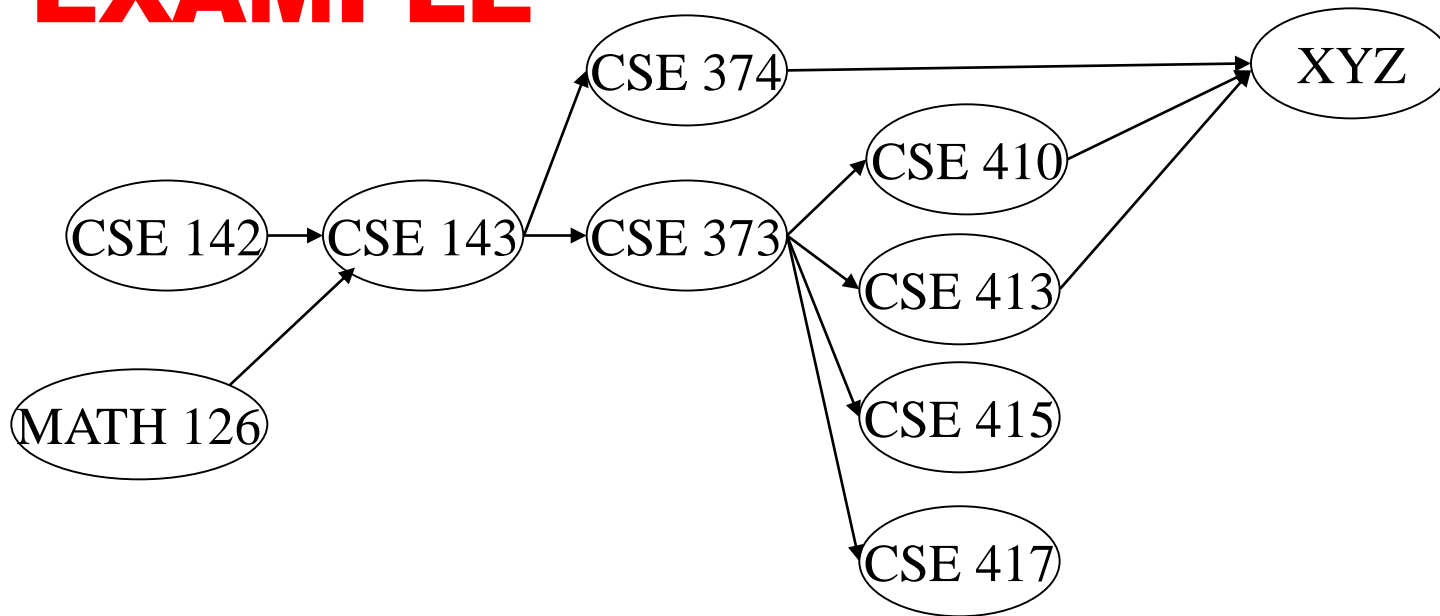
126

Node: 126 142 143 374 373 410 413 415 417 XYZ

Removed? x

In-degree: 0 0 ~~2~~ 1 1 1 1 1 1 3  
1

# EXAMPLE



Output:

126

142

Node: 126 142 143 374 373 410 413 415 417 XYZ

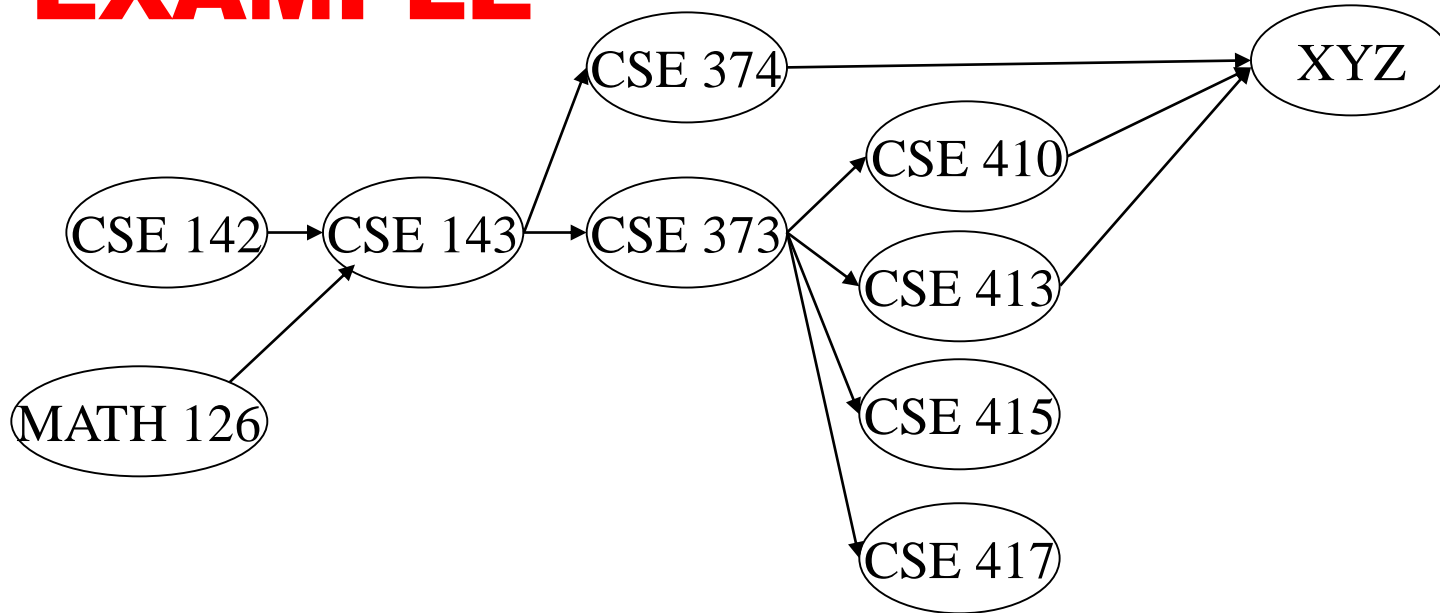
Removed? x x

In-degree: 0 0 ~~2~~ 1 1 1 1 1 1 3

~~1~~

~~0~~

# EXAMPLE



Output:

126

142

143

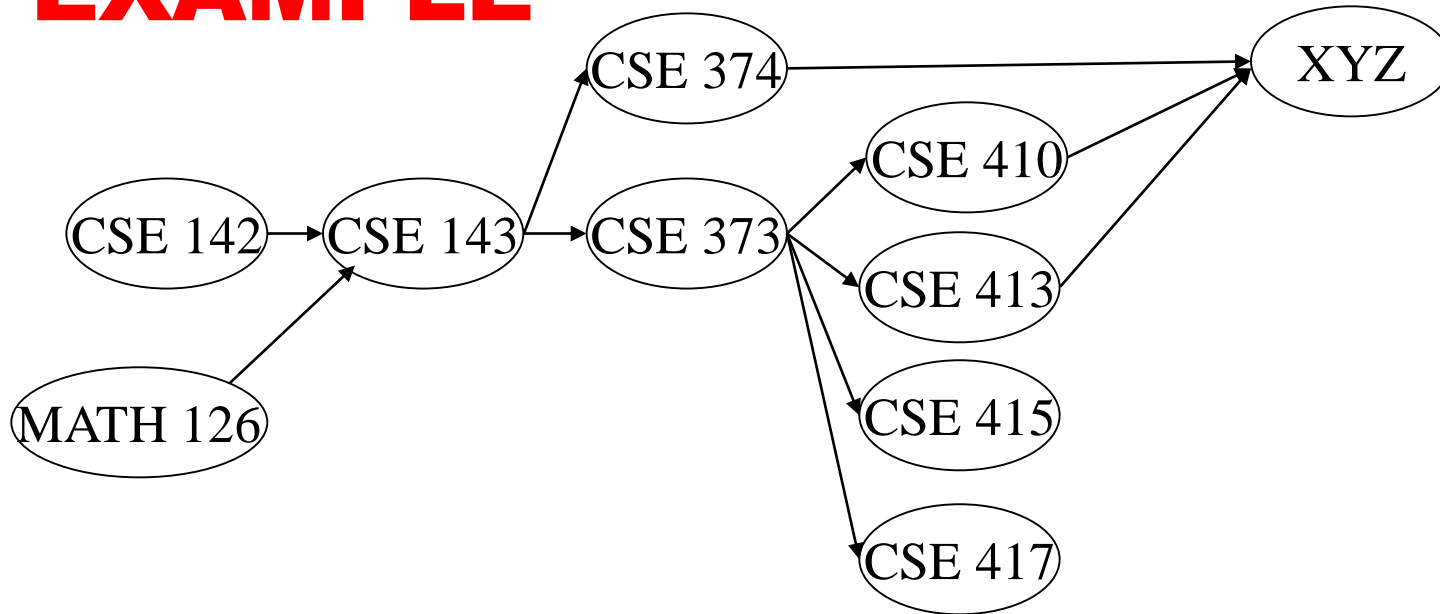
Node: 126 142 143 374 373 410 413 415 417 XYZ

Removed? x x **x**

In-degree: 0 0 ~~2~~ ~~1~~ ~~1~~ 1 1 1 1 3

~~1~~ ~~0~~ ~~0~~

# EXAMPLE



Output:

126

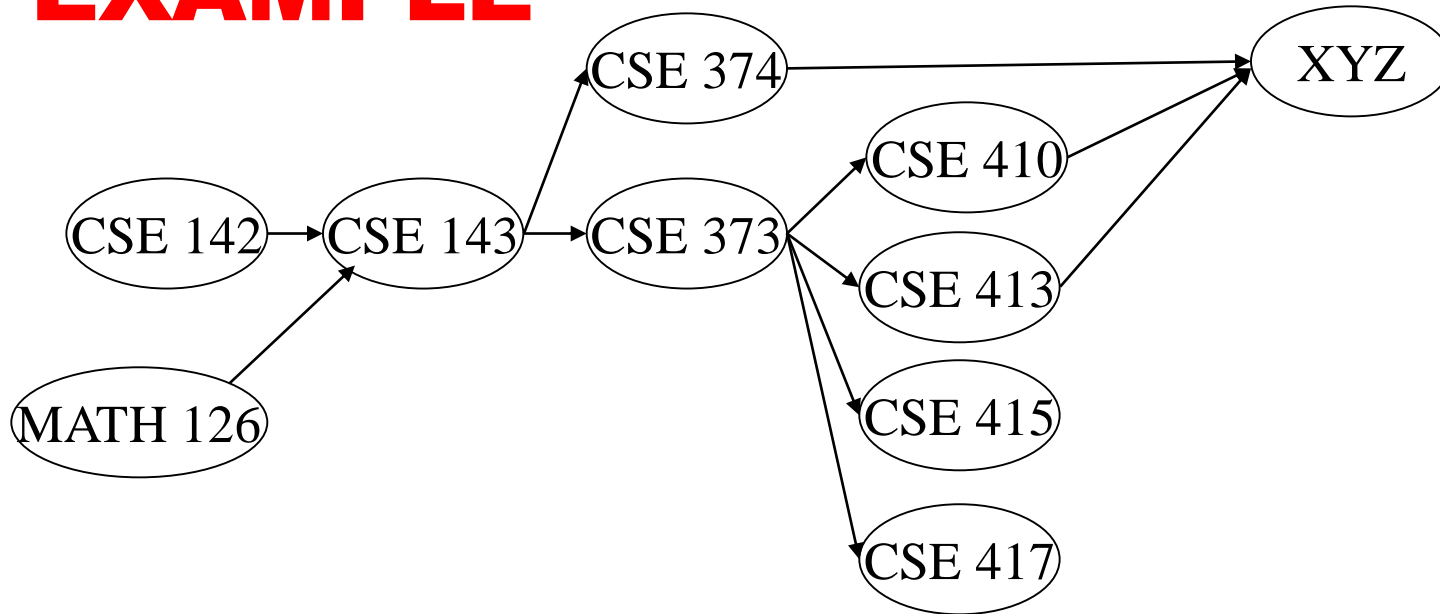
142

143

374

Node:	126	142	143	374	373	410	413	415	417	XYZ
Removed?	x	x	x	x						
In-degree:	0	0	<del>2</del>	<del>1</del>	<del>1</del>	1	1	1	1	<del>3</del>
			<del>1</del>	<del>0</del>	<del>0</del>					<del>2</del>
			0							

# EXAMPLE



Output:

126

142

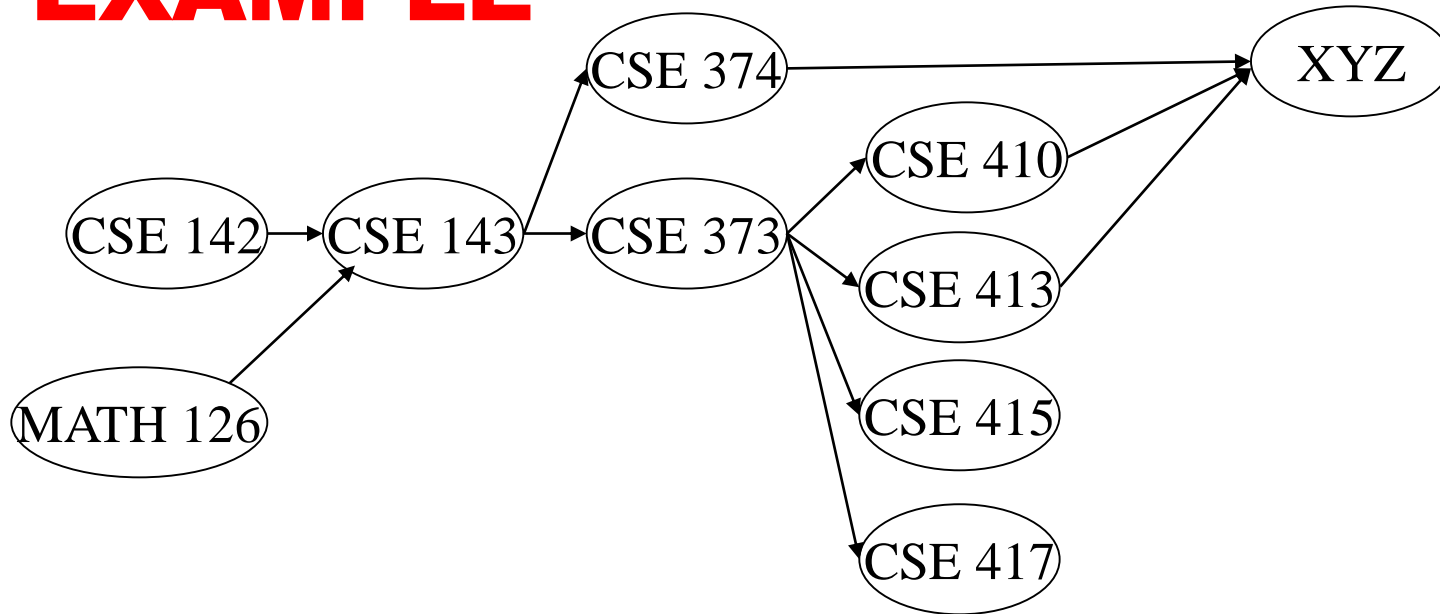
143

374

373

Node:	126	142	143	374	373	410	413	415	417	XYZ
Removed?	x	x	x	x	x					
In-degree:	0	0	<del>2</del>	<del>1</del>	<del>1</del>	<del>1</del>	<del>1</del>	<del>1</del>	<del>1</del>	<del>3</del>
			<del>1</del>	0	0	0	0	0	0	2
			0							

# EXAMPLE



Output:

126

142

143

374

373

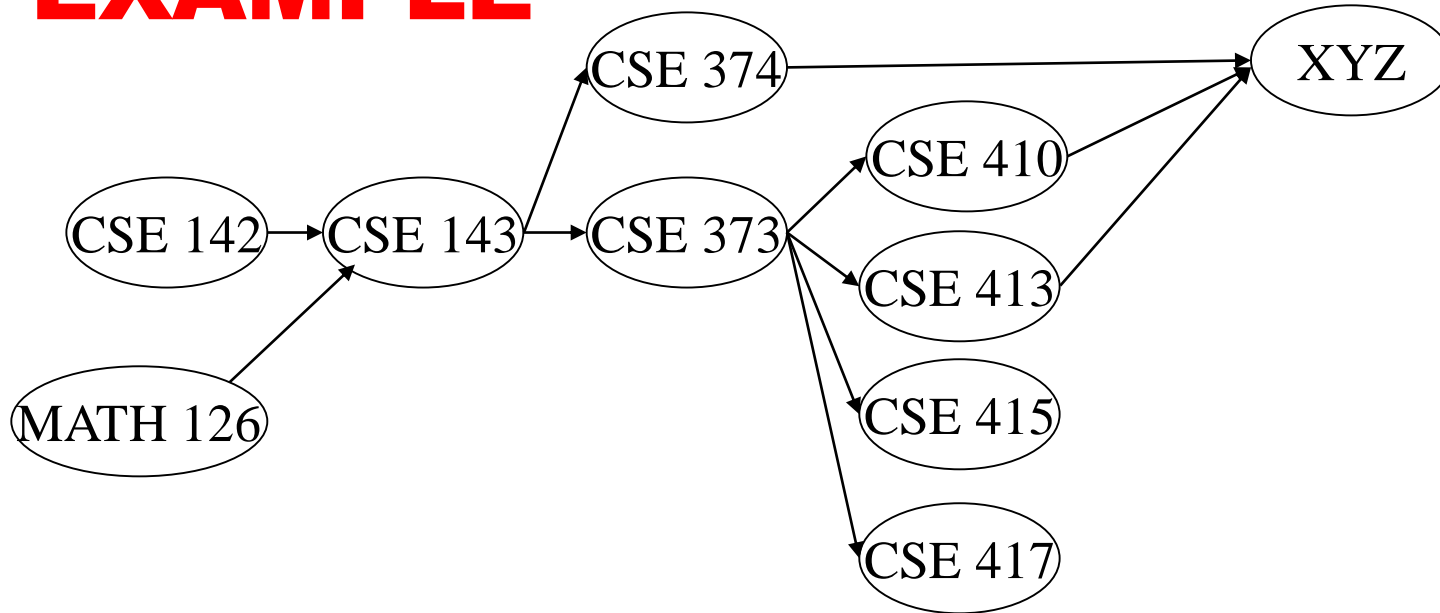
417

Node: 126 142 143 374 373 410 413 415 417 XYZ

Removed? x x x x x x x x

In-degree: 0 0 ~~2~~ ~~1~~ ~~1~~ ~~1~~ ~~1~~ ~~1~~ ~~1~~ ~~3~~  
~~1~~ 0 0 0 0 0 0 0 2  
 0

# EXAMPLE



Output:

126

142

143

374

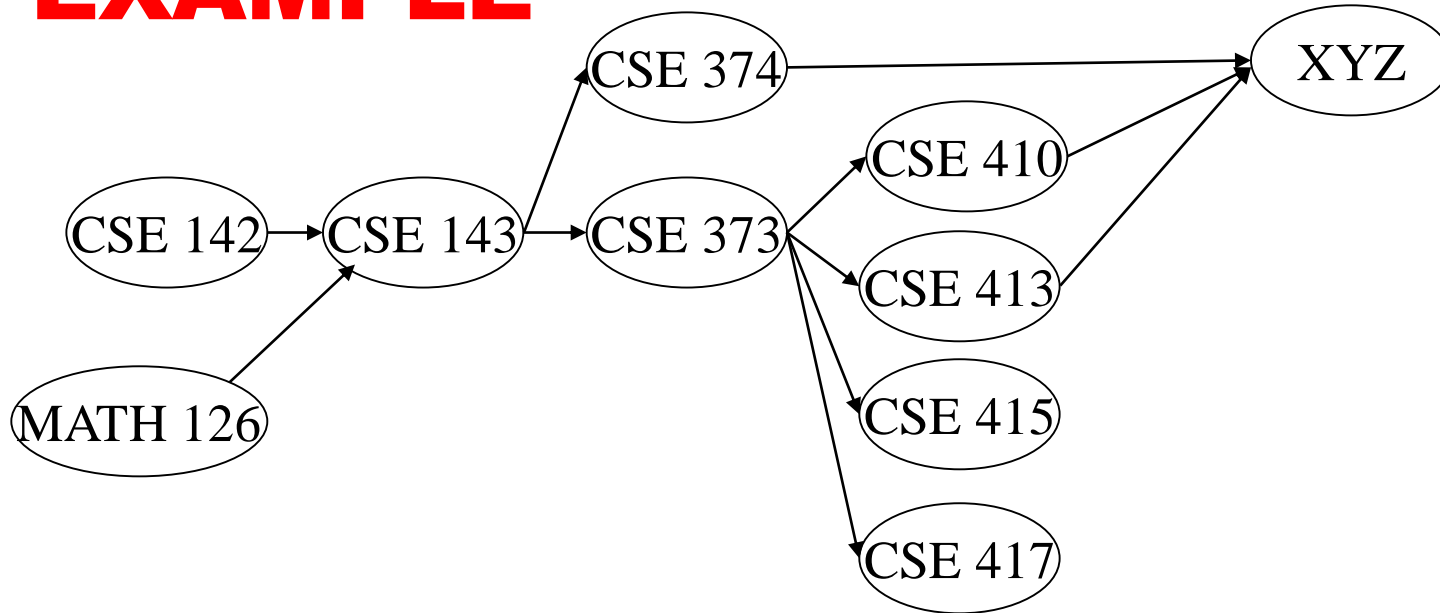
373

417

410

Node:	126	142	143	374	373	410	413	415	417	XYZ
Removed?	x	x	x	x	x	x			x	
In-degree:	0	0	<del>2</del>	<del>1</del>	<del>1</del>	<del>1</del>	<del>1</del>	<del>1</del>	<del>1</del>	<del>3</del>
			<del>1</del>	0	0	0	0	0	0	<del>2</del>
			0							1

# EXAMPLE



Output:

126

142

143

374

373

417

410

413

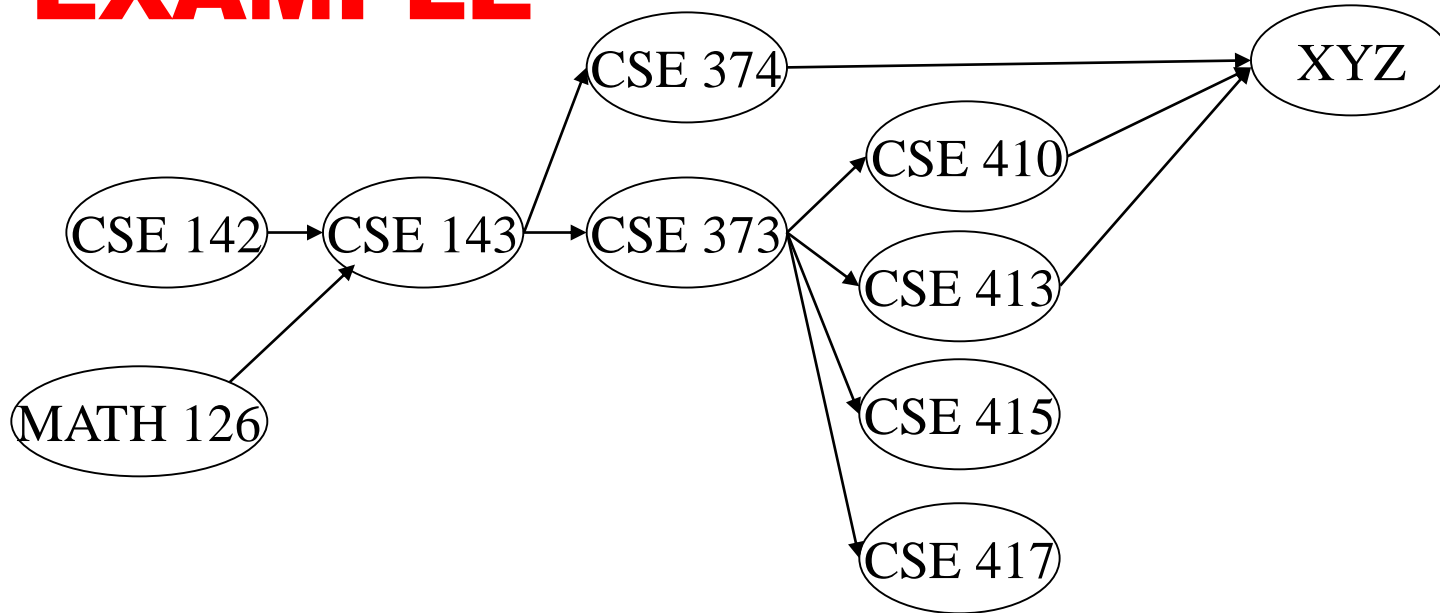
Node: 126 142 143 374 373 410 413 415 417 XYZ

Removed? x x x x x x **x** x

In-degree: 0 0 ~~2~~ ~~1~~ ~~1~~ ~~1~~ ~~1~~ ~~1~~ ~~1~~ ~~3~~  
~~1~~ 0 0 0 0 0 0 2  
0 ~~1~~ 0



# EXAMPLE



Output:

126

142

143

374

373

417

410

413

XYZ

Node: 126 142 143 374 373 410 413 415 417 XYZ

Removed? x x x x x x x x x

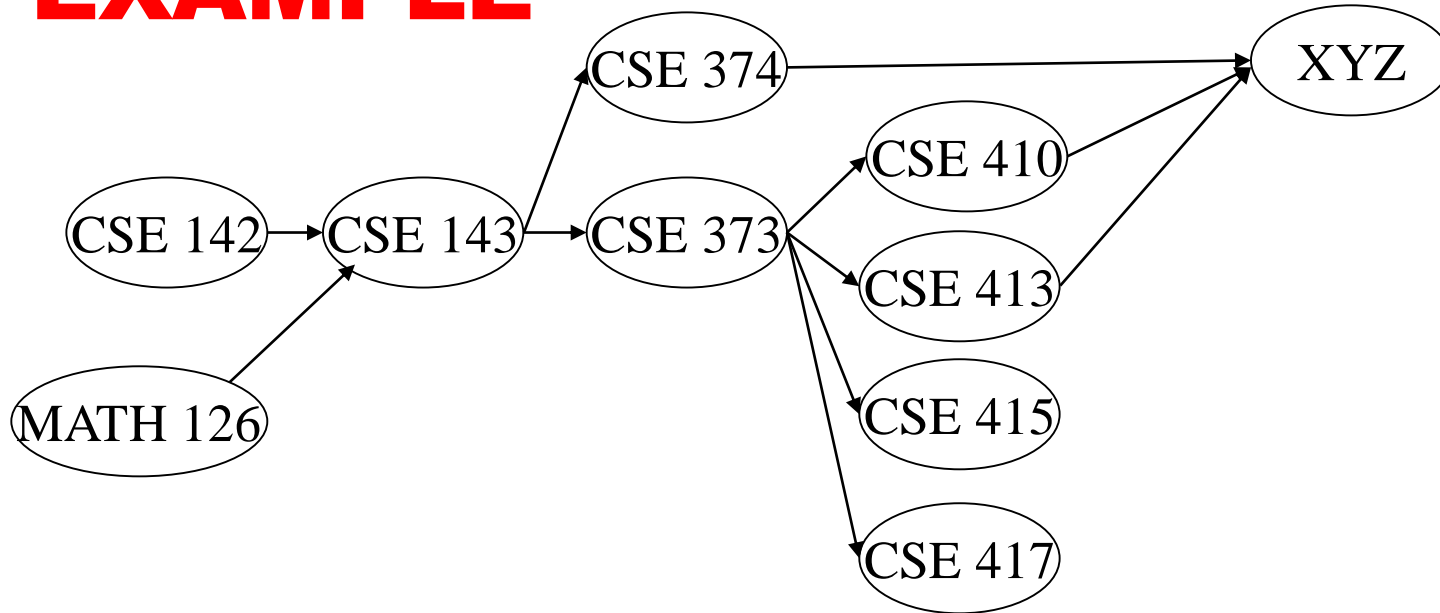
In-degree: 0 0 ~~2~~ ~~1~~ ~~1~~ ~~1~~ ~~1~~ ~~1~~ ~~1~~ ~~3~~

~~1~~ 0 0 0 0 0 0 0 2

0 ~~1~~

0

# EXAMPLE



Output:

126

142

143

374

373

417

410

413

XYZ

415

Node:	126	142	143	374	373	410	413	415	417	XYZ
Removed?	x	x	x	x	x	x	x	x	x	x
In-degree:	0	0	<del>2</del>	<del>1</del>	<del>1</del>	<del>1</del>	<del>1</del>	<del>1</del>	<del>1</del>	<del>3</del>
			<del>1</del>	0	0	0	0	0	0	<del>2</del>
			0							<del>1</del>
										0

# NOTICE

**Needed a vertex with in-degree 0 to start**

- Will always have at least 1 because no cycles

**Ties among vertices with in-degrees of 0 can be broken arbitrarily**

- Can be more than one correct answer, by definition, depending on the graph

# IMPLEMENTATION

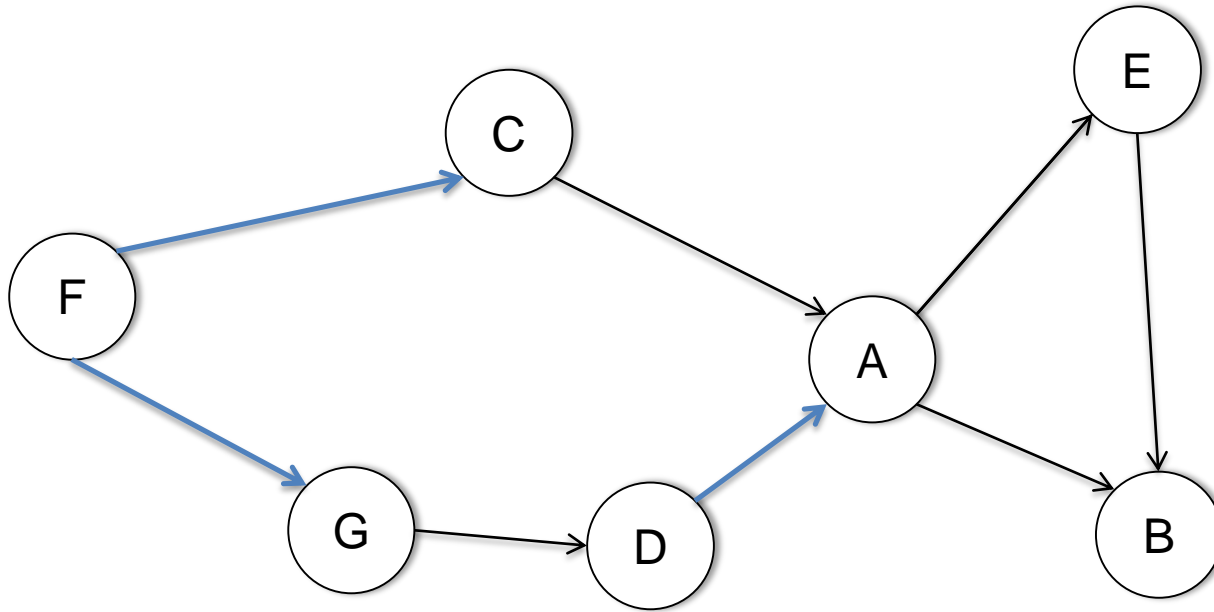
**The trick is to avoid searching for a zero-degree node every time!**

- Keep the “pending” zero-degree nodes in a list, stack, queue, bag, table, or something
- Order we process them affects output but not correctness or efficiency provided add/remove are both  $O(1)$

**Using a queue:**

1. **Label each vertex with its in-degree, enqueue 0-degree nodes**
2. **While queue is not empty**
  - a)  $v = \text{dequeue}()$
  - b) Output  $v$  and remove it from the graph

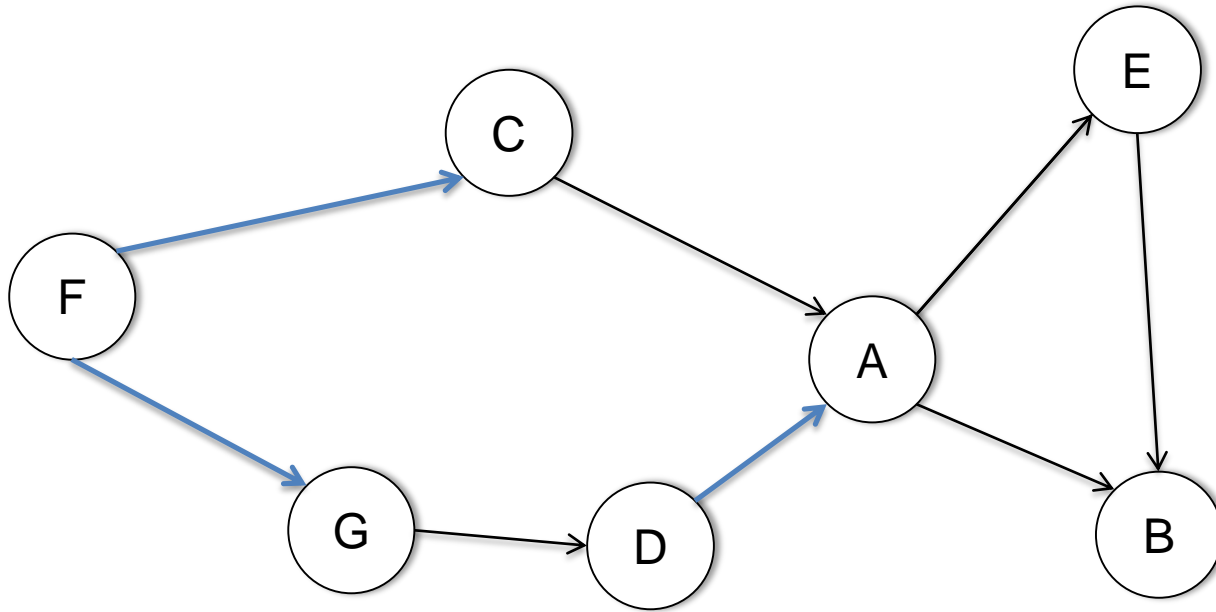
# TRAVERSAL



**Start with the nodes that have in-degree 0 (no prereqs)**

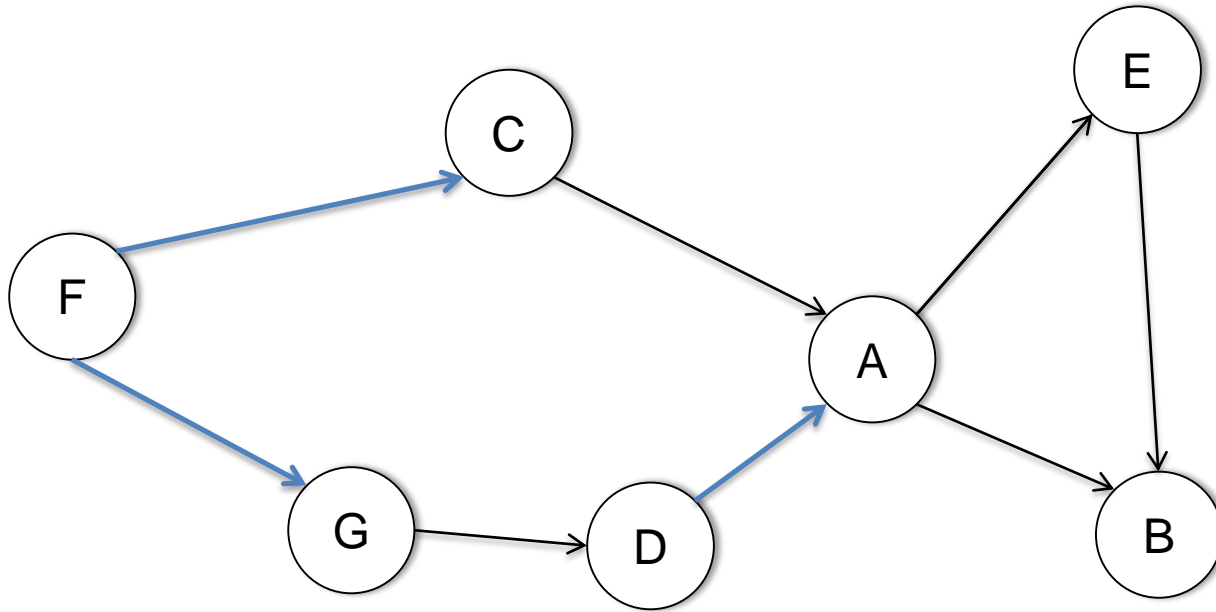
**Then eliminate that vertex (print it out) and eliminate its out edges.**

# TRAVERSAL



**What is a valid topological sort of this graph?**

# TRAVERSAL



**What is a valid topological sort of this graph?**

F,C,G,D,A,E,B

F,G,D,C,A,E,B

F,G,C,D,A,E,B

**Are these all the valid solutions?**

# TOPOLOGICAL SORT

- What use does this traversal have?



# TOPOLOGICAL SORT

- What use does this traversal have?
  - Good for dependency resolution

# TOPOLOGICAL SORT

- **What use does this traversal have?**
  - Good for dependency resolution
  - Can also be used for cycle detection

# TOPOLOGICAL SORT

- **What use does this traversal have?**
  - Good for dependency resolution
  - Can also be used for cycle detection
- **How could we find cycles in an undirected graph?**

# TOPOLOGICAL SORT

- **What use does this traversal have?**
  - Good for dependency resolution
  - Can also be used for cycle detection
- **How could we find cycles in an undirected graph?**
  - Any traversal that visits a node more than once has a cycle.

# GRAPH PROBLEMS

- When thinking about graphs, it is important to understand what the graph represents

# GRAPH PROBLEMS

- When thinking about graphs, it is important to understand what the graph represents
  - Topological sort:

# GRAPH PROBLEMS

- **When thinking about graphs, it is important to understand what the graph represents**
  - Topological sort:
    - Programs and dependencies
    - Courses and prereqs

# GRAPH PROBLEMS

- **When thinking about graphs, it is important to understand what the graph represents**
  - Topological sort:
    - Programs and dependencies
    - Courses and prereqs
  - What the vertices and edges are impact what the “solution” is?





# GRAPH PROBLEMS

- What type of problem could we want to solve with a graph of US cities and the freeway distance between them

# GRAPH PROBLEMS

- **What type of problem could we want to solve with a graph of US cities and the freeway distance between them**
  - Same as a lot of network problems

# GRAPH PROBLEMS

- **What type of problem could we want to solve with a graph of US cities and the freeway distance between them**
  - Same as a lot of network problems
  - “Traffic” networks

# GRAPH PROBLEMS

- **What type of problem could we want to solve with a graph of US cities and the freeway distance between them**
  - Same as a lot of network problems
  - “Traffic” networks
  - What do our edges represent?

# SINGLE SOURCE SHORTEST PATH

- Given an undirected, *unweighted* graph  $G(V,E)$  and a start vertex  $A$ , find the shortest path to all connected vertices

# SINGLE SOURCE SHORTEST PATH

- Given an undirected, *unweighted* graph  $G(V,E)$  and a start vertex  $A$ , find the shortest path to all connected vertices
  - If a graph is unweighted you can treat all of their weights as 1

# SINGLE SOURCE SHORTEST PATH

- Given an undirected, *unweighted* graph  $G(V,E)$  and a start vertex  $A$ , find the shortest path to all connected vertices
  - If a graph is unweighted you can treat all of their weights as 1
  - Do a BFS traversal of the tree and keep track of paths!



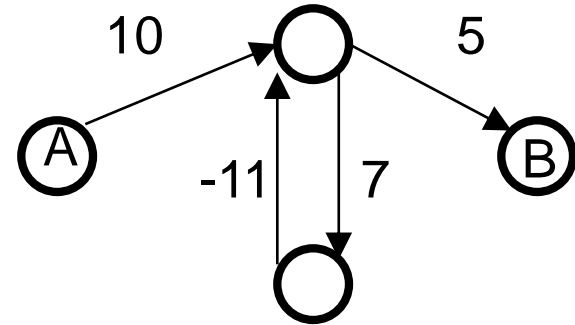
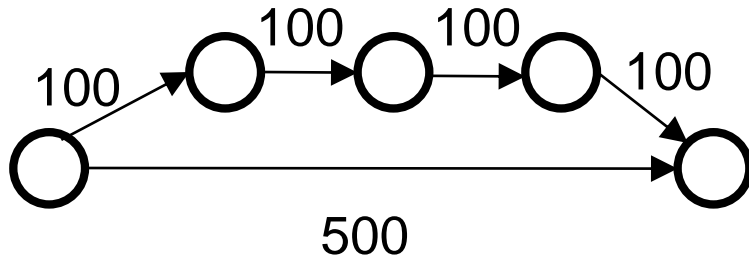
# SINGLE SOURCE SHORTEST PATH

- Given an undirected, *unweighted* graph  $G(V,E)$  and a start vertex  $A$ , find the shortest path to all connected vertices
  - If a graph is unweighted you can treat all of their weights as 1
  - Do a BFS traversal of the tree and keep track of paths!
  - Path-keeping is non-trivial, we'll talk about it on Wednesday

# SINGLE SOURCE SHORTEST PATH

- Given an undirected, *unweighted* graph  $G(V,E)$  and a start vertex  $A$ , find the shortest path to all connected vertices
  - If a graph is unweighted you can treat all of their weights as 1
  - Do a BFS traversal of the tree and keep track of paths!
  - Path-keeping is non-trivial, we'll talk about it on Wednesday
  - What if the graph has weights?

# PATH-FINDING



**Why BFS won't work: Shortest path may not have the fewest edges**

- Annoying when this happens with costs of flights

We will assume there are no negative weights

- *Problem is ill-defined* if there are negative-cost cycles
- Wednesday's *algorithm is wrong* if edges can be negative
  - There are other, slower (but not terrible) algorithms

# **NEXT CLASS**

- **Dijkstra's algorithm**

# **NEXT CLASS**

- **Dijkstra's algorithm**
- **P3 checkpoint 2**