



# Graph Search

CSE 332 Spring 25  
Lecture 16

# What do we do with graphs

So many things!

-That's why we said graphs are more general than a single ADT---they don't have a standard set of operations.

As a starting point--how could we process the entire graph? Examine every vertex and every edge?

Called a "search" of the graph or a "traversal"

Two algorithms (with different purposes)

# BFS

Start somewhere...

For every vertex (in some order)

Do whatever you want on that vertex

- Sometimes, record some information, store something in there, etc.
- At least, we'll mark it as having been "visited"

You need to process all of its neighbors...store them in some data structure to process them. Then back to the top of the loop.

If you use a Queue for your storage structure, you get BFS.

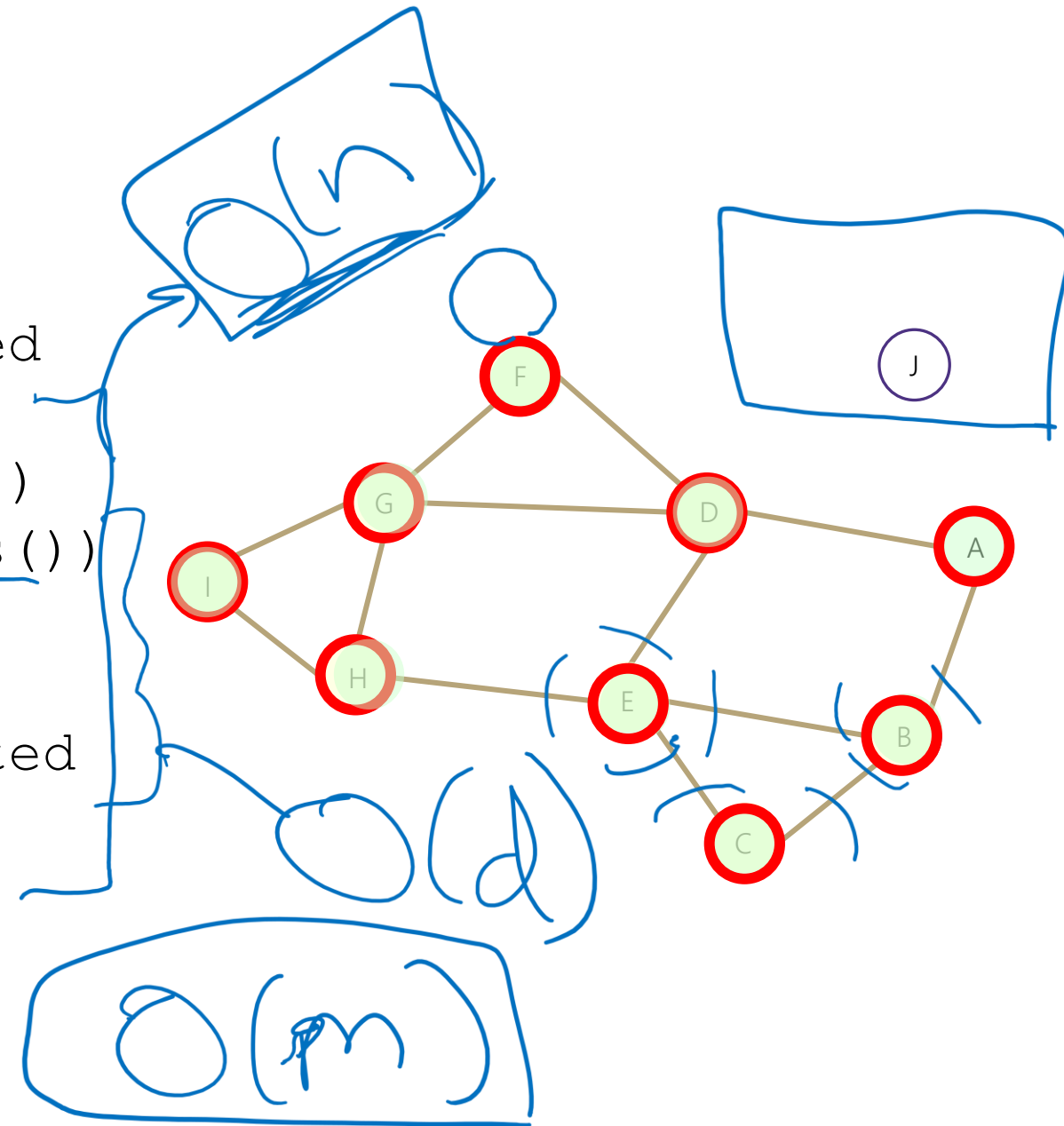
# Breadth First Search

```
search(graph)
  toVisit.enqueue(first vertex)
  mark first vertex as visited
  while(toVisit is not empty)
    current = toVisit.dequeue()
    for (V : current.neighbors())
      if (v is not visited)
        toVisit.enqueue(v)
        mark v as visited
    finished.add(current)
```

Current node: I

Queue: B D E C F G H I

Finished: A B D E C F G H I



# Breadth First Search

search(graph)

toVisit.enqueue(first vertex)

mark first vertex as visited

while(toVisit is not empty)

current = toVisit.dequeue()

for (V : current.neighbors())

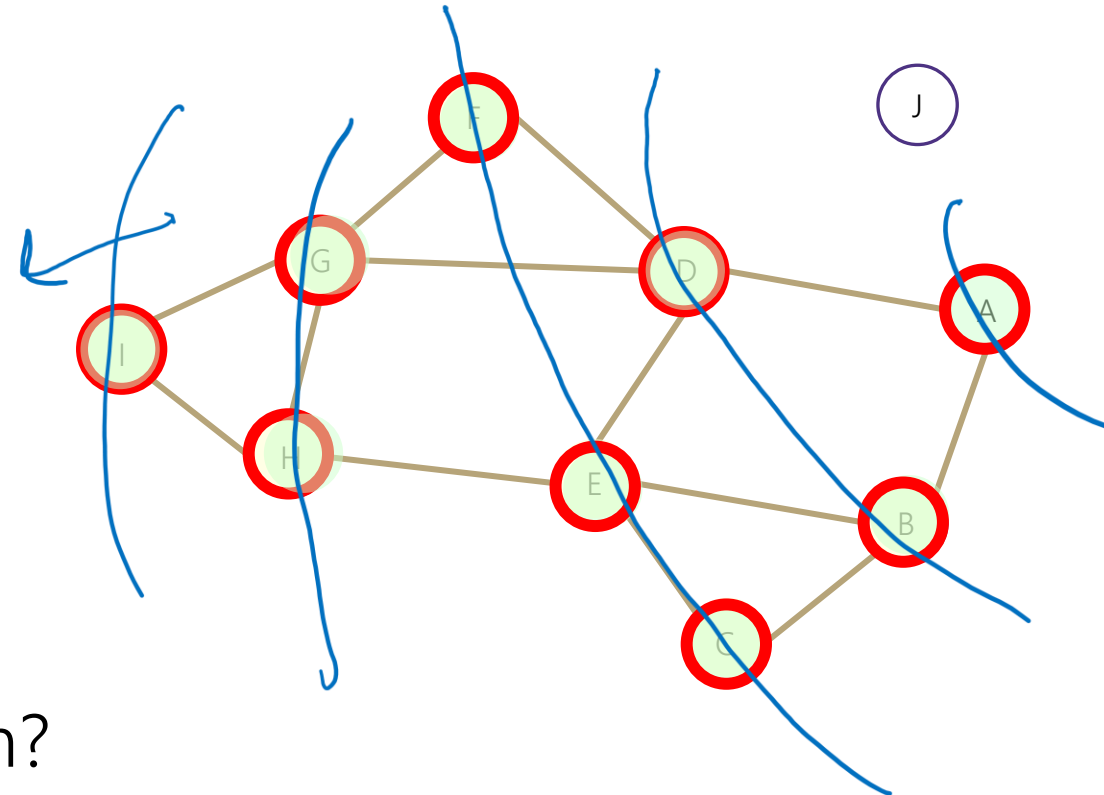
if (v is not visited)

toVisit.enqueue(v)

mark v as visited

finished.add(current)

$$O(n+m)$$



What's the running time of this algorithm?

We visit each vertex at most twice, and each edge at most once:

$$O(|V| + |E|)$$

# Depth First Search (DFS)

BFS uses a queue to order which vertex we move to next

Gives us a growing "frontier" movement across graph

Can you move in a different pattern? What if you used a stack instead?

```
bfs(graph)
  toVisit.enqueue(first vertex)
  mark first vertex as visited
  while(toVisit is not empty)
    current = toVisit.dequeue()
    for (V : current.neighbors())
      if (v is not visited)
        toVisit.enqueue(v)
        mark v as visited
  finished.add(current)
```

```
dfs(graph, curr)
  mark curr as visited
  for(v : curr.neighbors()) {
    if(v is not visited) {
      dfs(graph, v)
    }
  }
  mark curr as "done"
```

# Depth First Search

```
dfs(graph, curr)
```

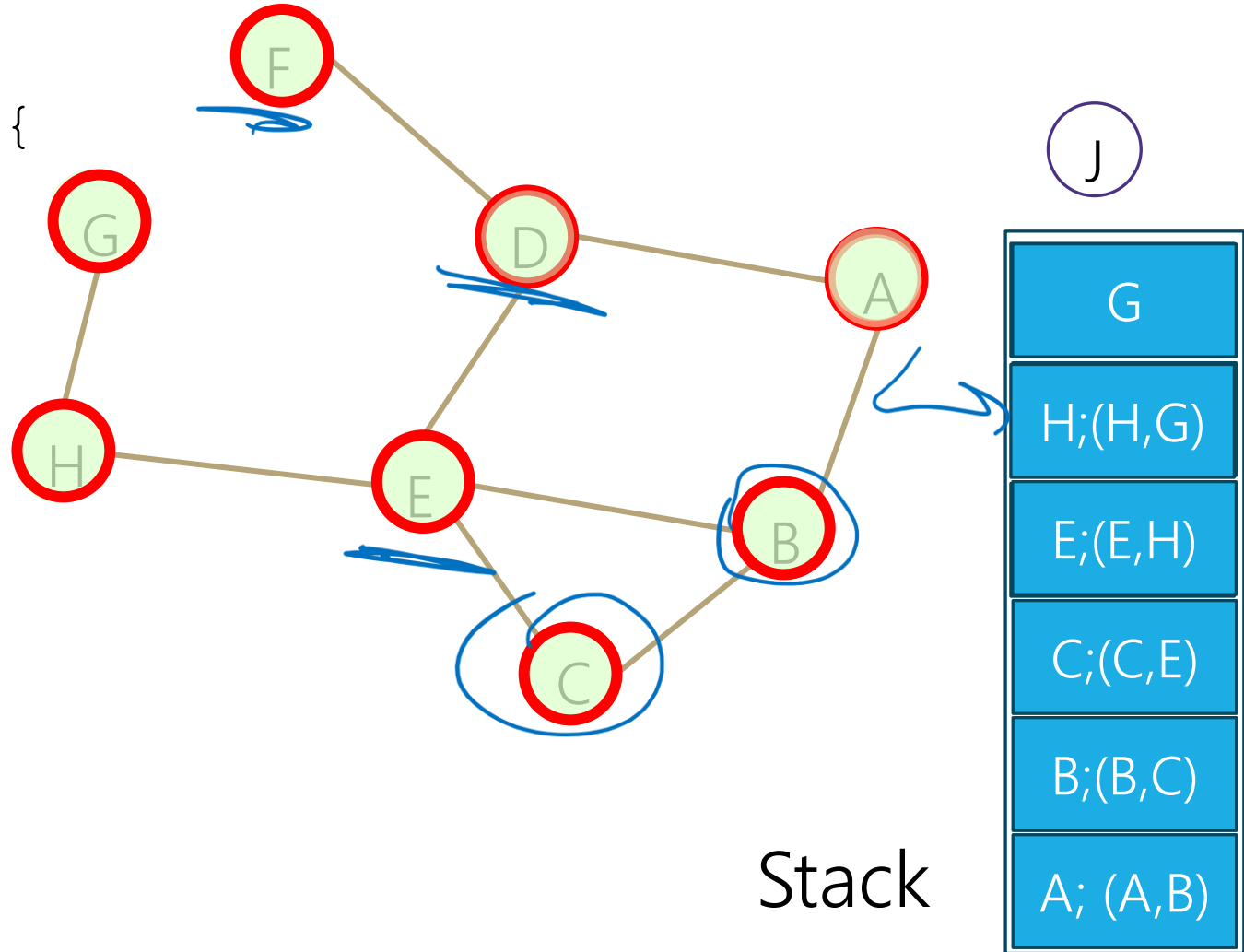
```
mark curr as visited
```

```
for (v : curr.neighbors()) {
```

```
  if (v is not visited) {  
    dfs(graph, v)
```

```
  }
```

```
mark curr as "done"
```



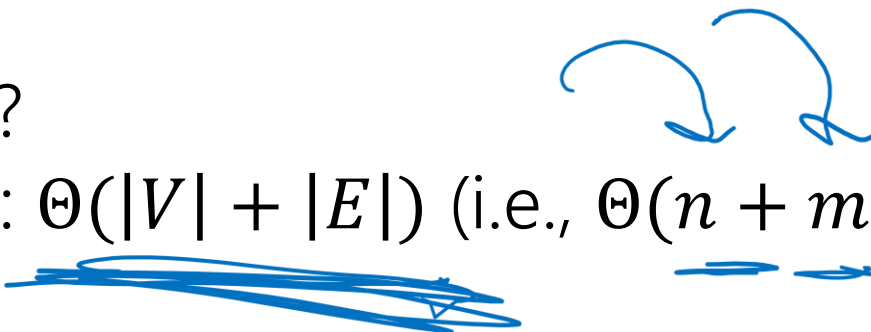
Finished: F D G H E C B A

Stack

# DFS

Running time?

-Same as BFS:  $\Theta(|V| + |E|)$  (i.e.,  $\Theta(n + m)$ )



You can rewrite DFS to be an iterative method (that explicitly uses a stack data structure). Use that in place of the call stack.

Getting the details right is actually pretty annoying/subtle.

Next: Using BFS, DFS and other algorithms to solve problems!



# DFS for applications

Applications for DFS (and BFS) are often:

Run [D/B]FS, and do some extra bookkeeping.

- Many applications work (easily) with only one ordering.

For DFS, it's common to classify based on "start" and "finish" times

When vertices go on the (call) stack, and when they come off.

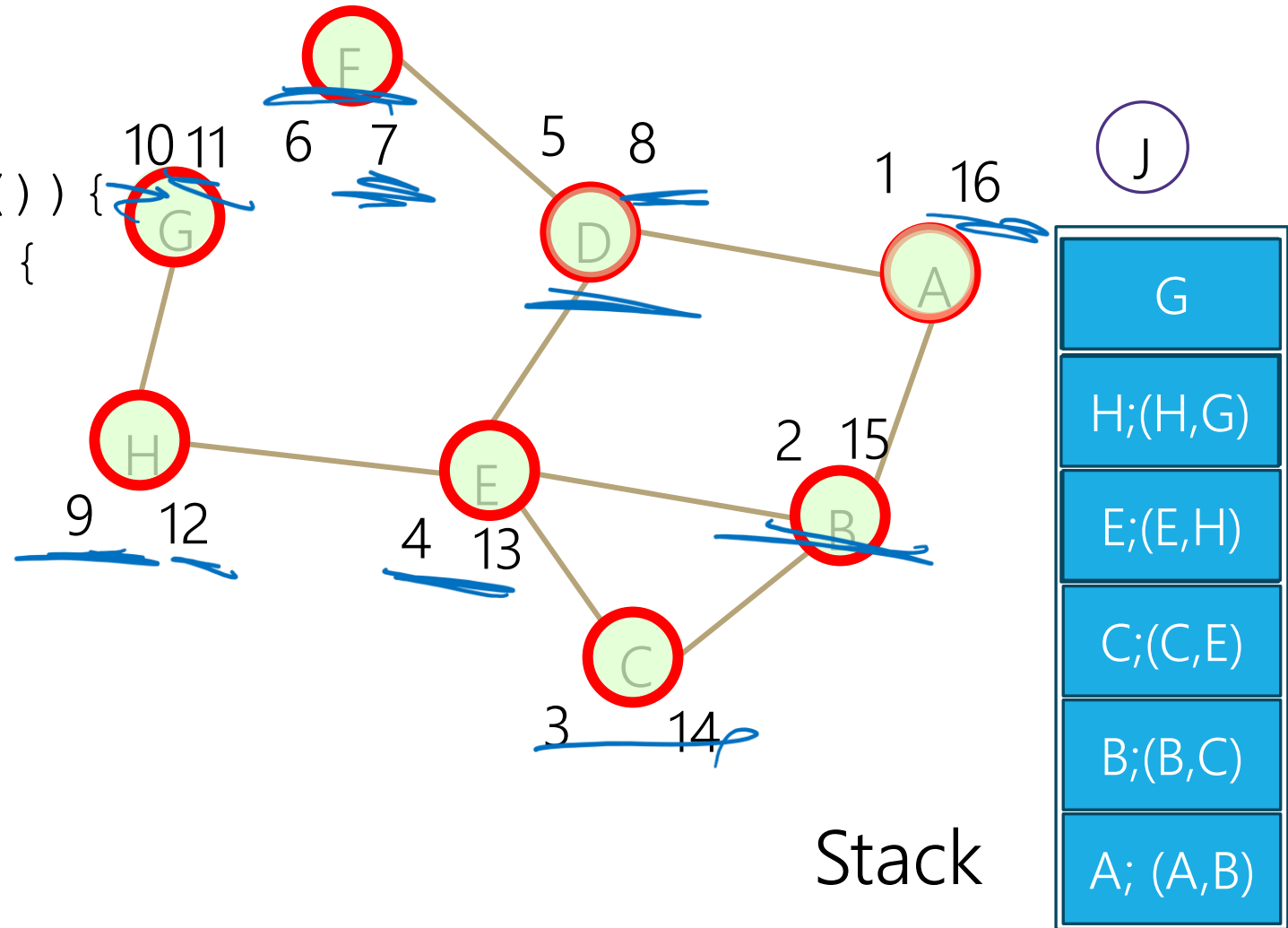
# Depth First Search

```
dfs(graph, curr)
  mark curr as visited
  record curr.start
  for (v : curr.neighbors()) {
    if (v is not visited) {
      dfs(graph, v)
    }
  }
  mark curr as "done"
  record curr.end
```

Finished: F D G H E C B A

(A,B), (B,C), (C,E) cause a new vertex to go on the stack.

(E,B) goes "back" to an edge that's already on the stack, but not finished.



# What do we use DFS for?

DFS can be used to detect cycles

How? What happened when we went around a cycle---we had an edge "back" to a vertex we'd discovered already (but that we hadn't finished processing).

What happened between "steps" 4 and 5?

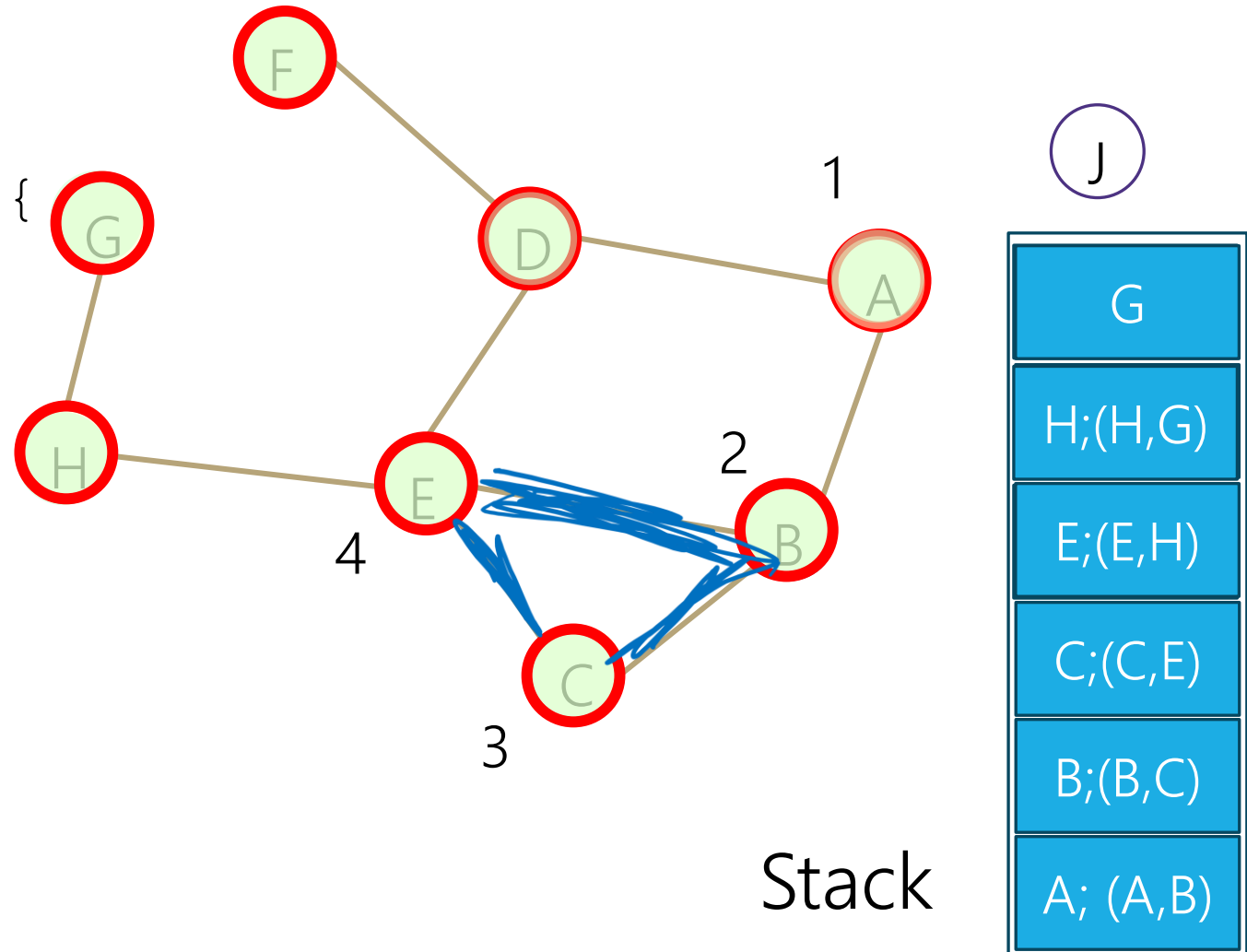
# Depth First Search

```
dfs(graph, curr)
  mark curr as visited
  record curr.start
  for(v : curr.neighbors()) {
    if(v is not visited) {
      dfs(graph, v)
    }
  }
  mark curr as "done"
  record curr.end
```

Finished: F D G H E C B A

(A,B), (B,C), (C,E) cause a new vertex to go on the stack.

(E,B) goes "back" to an edge that's already on the stack, but not finished.



# What do we use DFS for?

DFS can be used to detect cycles

How? What happened when we went around a cycle---we had an edge "back" to a vertex we'd discovered already (but that we hadn't finished processing).

The details are different depending on if your graph is directed or undirected. Your exercise will use directed graphs, so let's look at that.

# Edge Classification (DFS, directed graphs)

Edge type	Definition	When is $(u, v)$ that edge type?
Tree	Edges forming the DFS tree (or forest).	$v$ was not seen before we processed $(u, v)$ .
Forward	From ancestor to descendant in tree.	$u$ and $v$ have been seen, and $u.start < v.start < v.end < u.end$
Back	From descendant to ancestor in tree.	$u$ and $v$ have been seen, and $v.start < u.start < u.end < v.end$
Cross	Edges going between vertices without an ancestor relationship.	$u$ and $v$ have <del>not</del> been seen, and $v.start < v.end < u.start < u.end$

# Try it Yourself!

DFSWrapper (G)

counter = 0

For each vertex u of G

    If u is not "seen"

        DFS(u)

    End If

End For

DFS(u)

Mark u as "seen"

u.start = counter++

For each edge (u,v) //leaving u

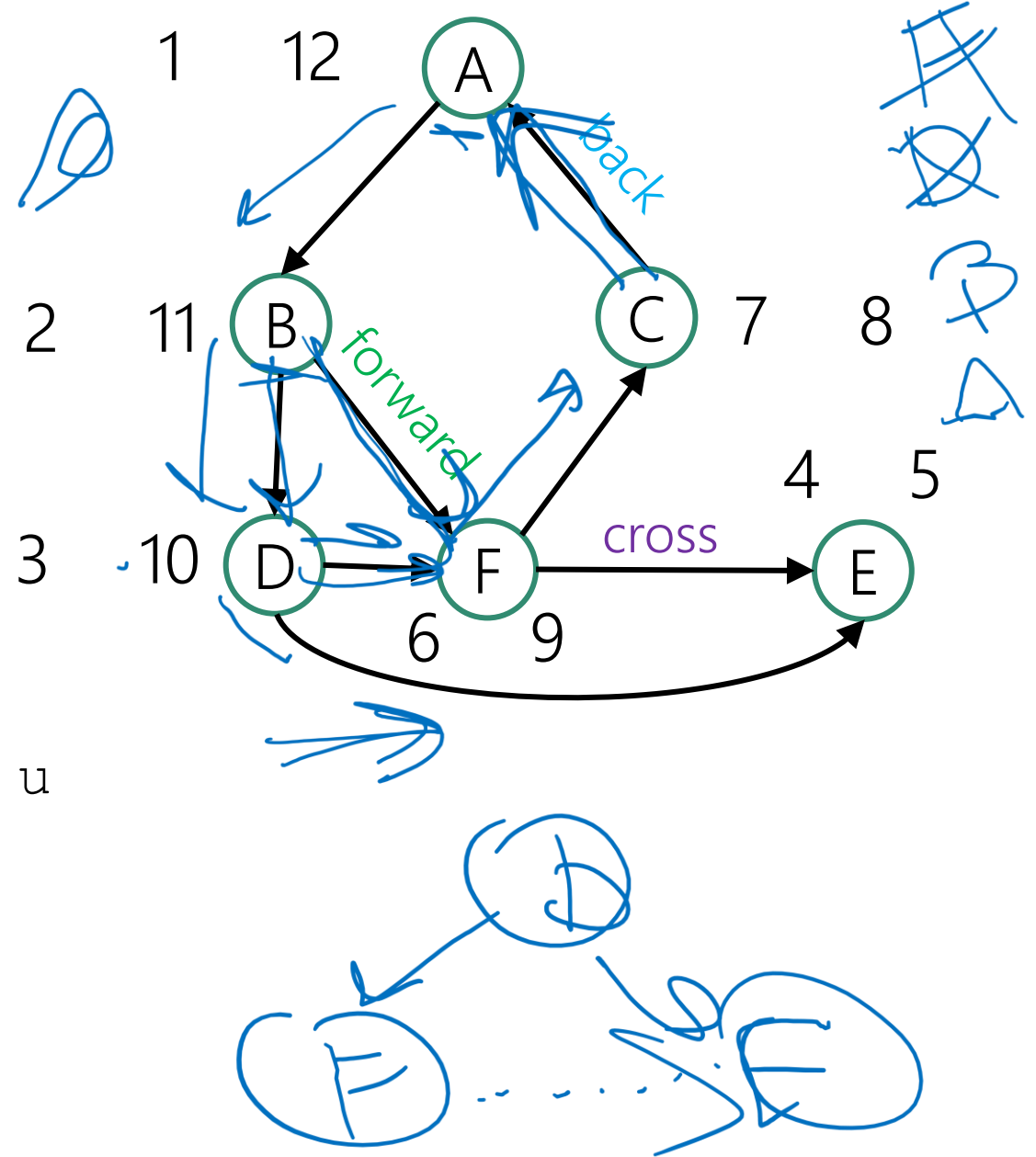
    If v is not "seen"

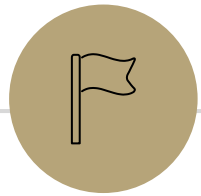
        DFS(v)

    End If

End For

u.end = counter++



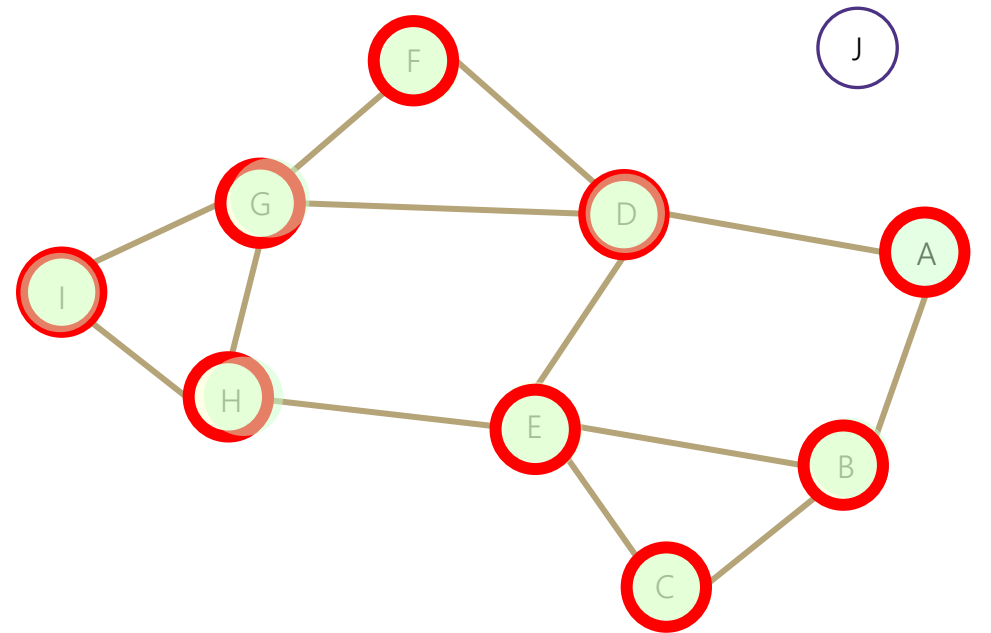


# Another Search Application



# Breadth First Search

```
search(graph)
  toVisit.enqueue(first vertex)
  mark first vertex as visited
  while(toVisit is not empty)
    current = toVisit.dequeue()
    for (V : current.neighbors())
      if (v is not visited)
        toVisit.enqueue(v)
        mark v as visited
    finished.add(current)
```



What's the running time of this algorithm?

We visit each vertex at most twice, and each edge at most once:  $O(|V| + |E|)$

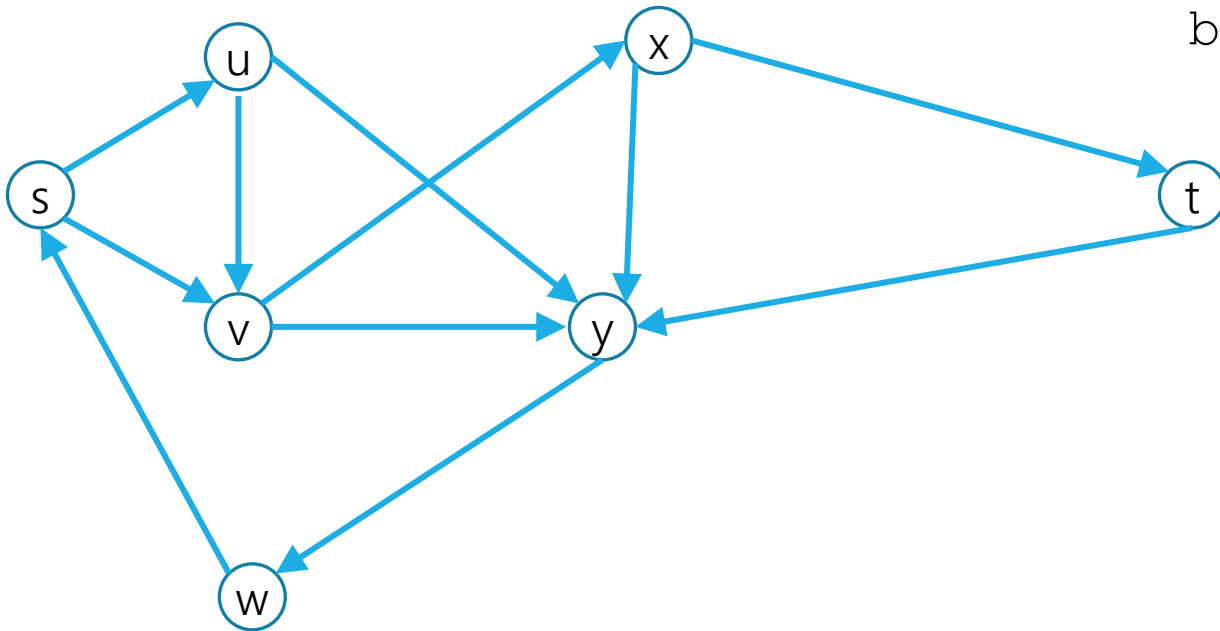
# Extra Practice

Run Breadth First Search on this graph starting from s.

What order are vertices placed on the queue?

When processing a vertex insert neighbors in alphabetical order.

In a directed graph, BFS only follows an edge in the direction it points.



```
bfs (graph)
```

```
  toVisit.enqueue (first vertex)  
  mark first vertex as visited  
  while (toVisit is not empty)  
    current = toVisit.dequeue ()  
    for (V : current.outneighbors ())  
      if (v is not visited)  
        toVisit.enqueue (v)  
        mark v as visited  
  finished.add (current)
```

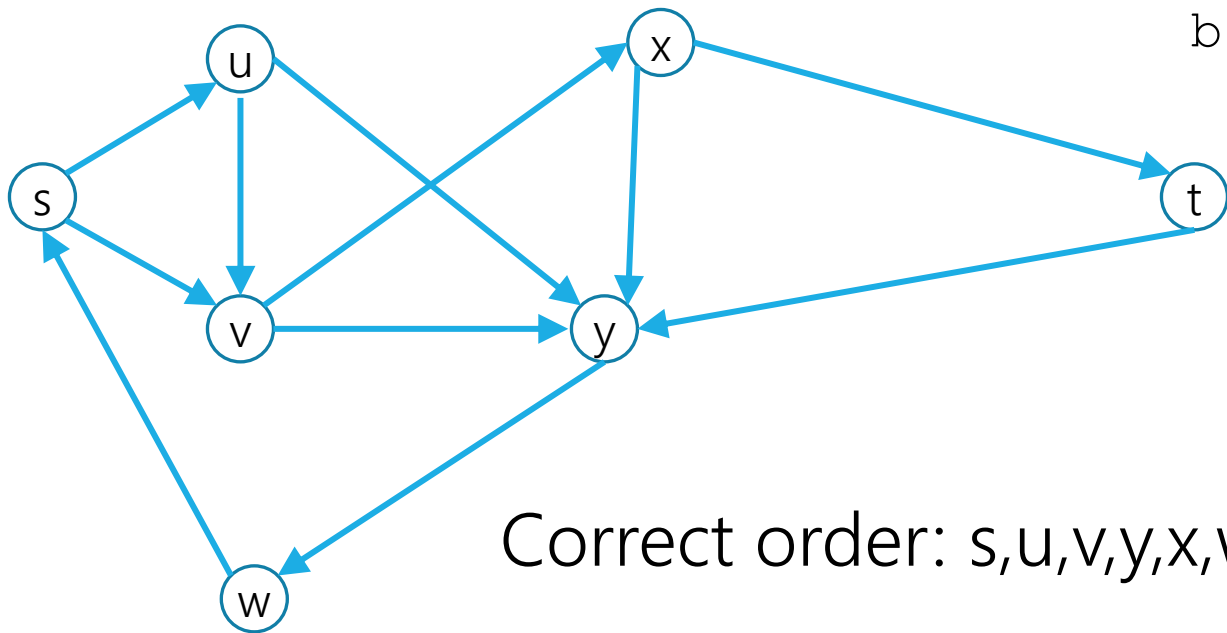
# Extra Practice

Run Breadth First Search on this graph starting from s.

What order are vertices placed on the queue?

When processing a vertex insert neighbors in alphabetical order.

In a directed graph, BFS only follows an edge in the direction it points.



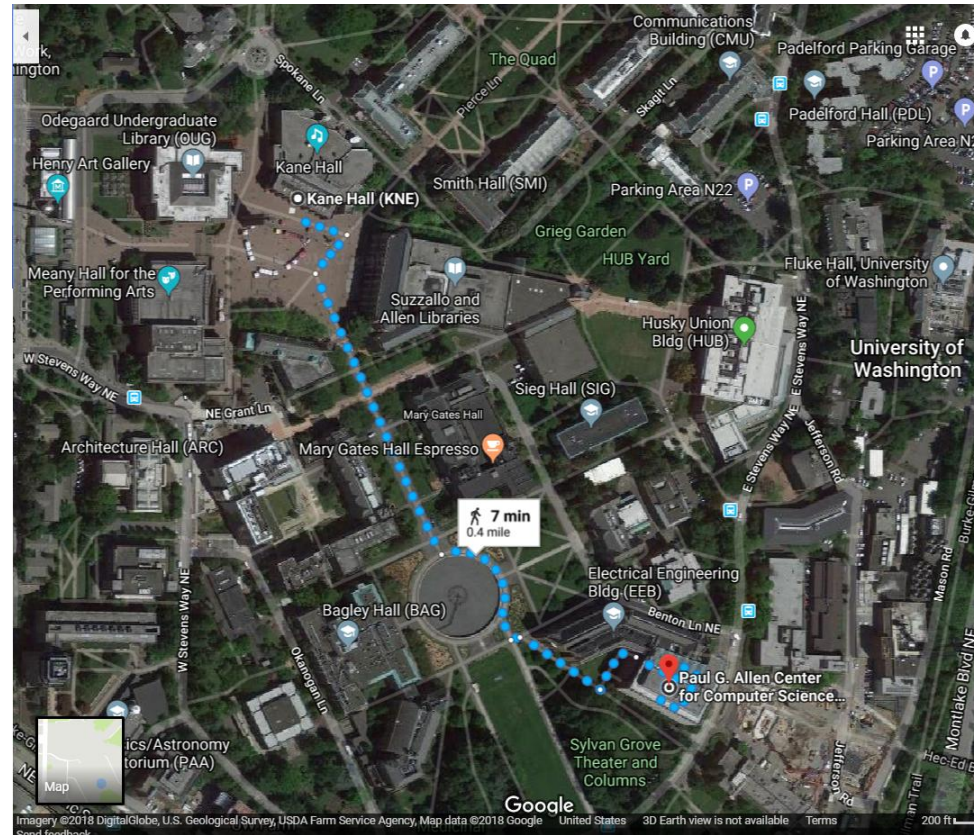
Correct order: s,u,v,y,x,w,t

```
bfs (graph)
```

```
toVisit.enqueue (first vertex)
mark first vertex as visited
while (toVisit is not empty)
  current = toVisit.dequeue ()
  for (V : current.outneighbors ())
    if (v is not visited)
      toVisit.enqueue (v)
      mark v as visited
  finished.add (current)
```

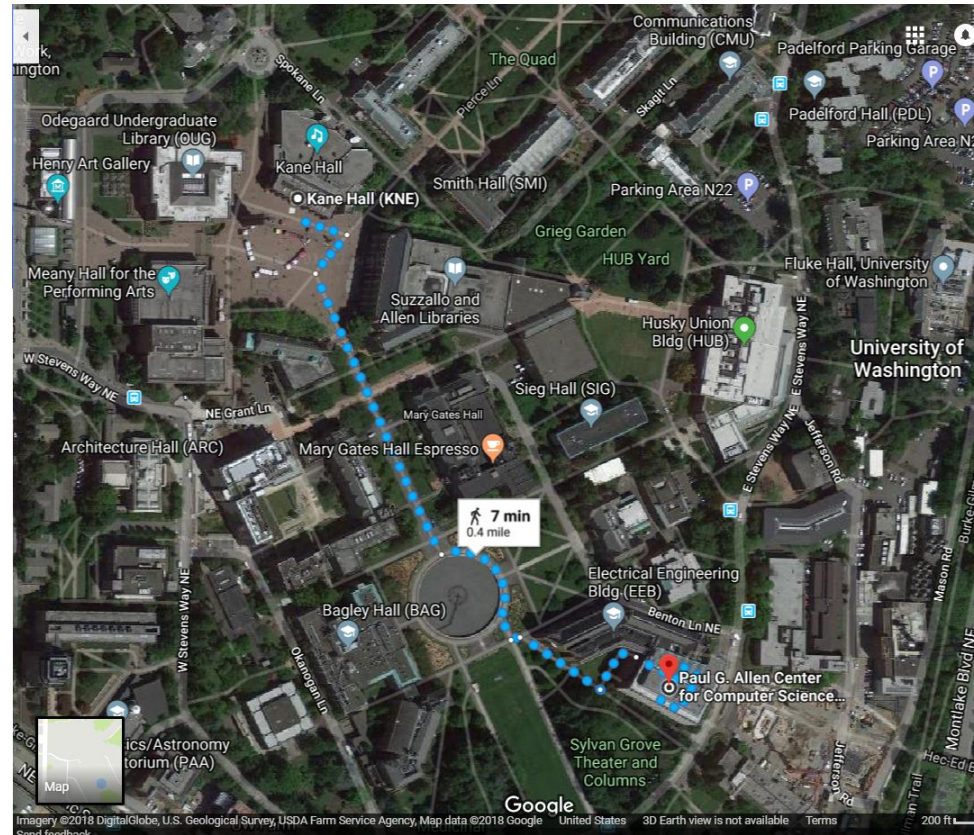
# Shortest Paths

How does Google Maps figure out this is the fastest way to get from Kane Hall to CSE?

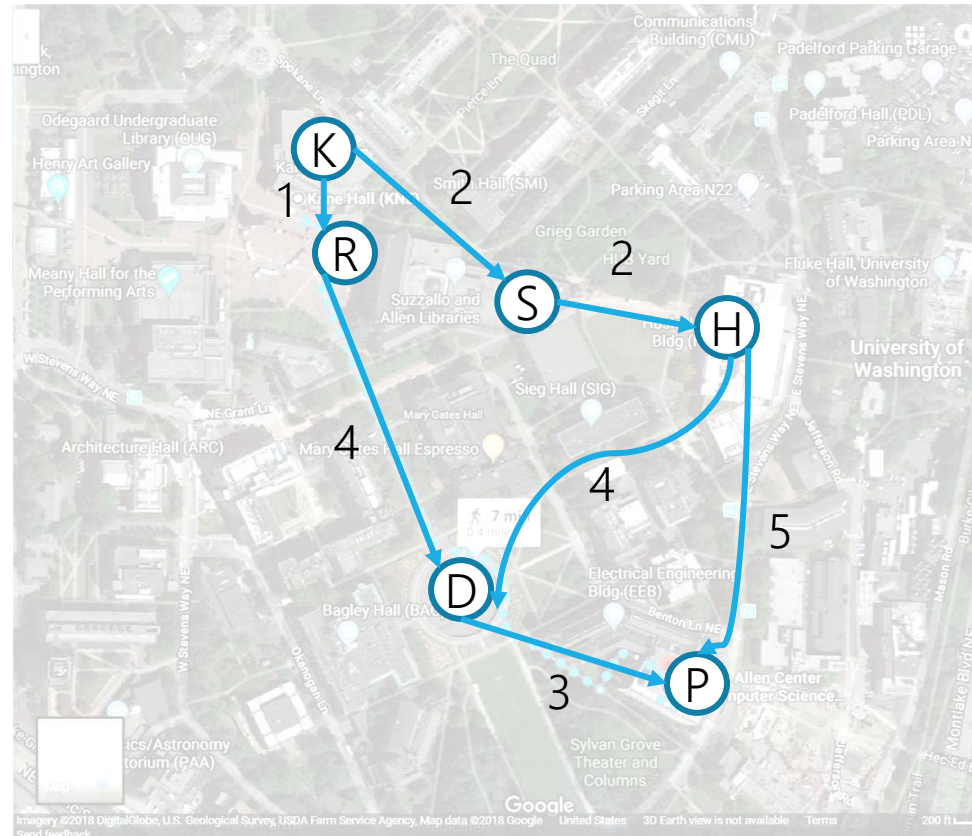


# Representing Maps as Graphs

How do we represent a map as a graph? What are the vertices and edges?



# Representing Maps as Graphs

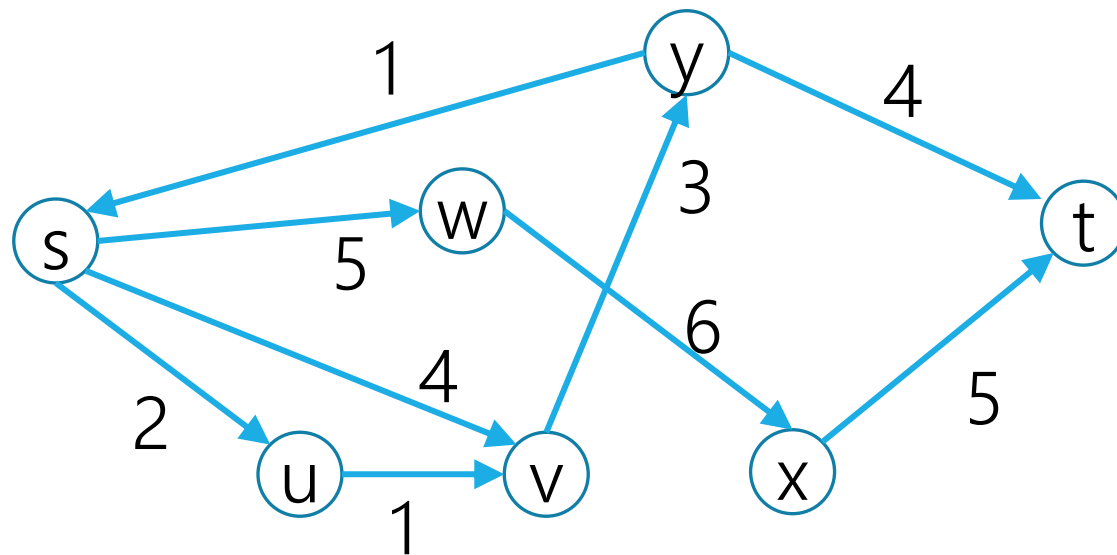


# Shortest Paths

The **length** of a path is the sum of the edge weights on that path.

## Shortest Path Problem

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

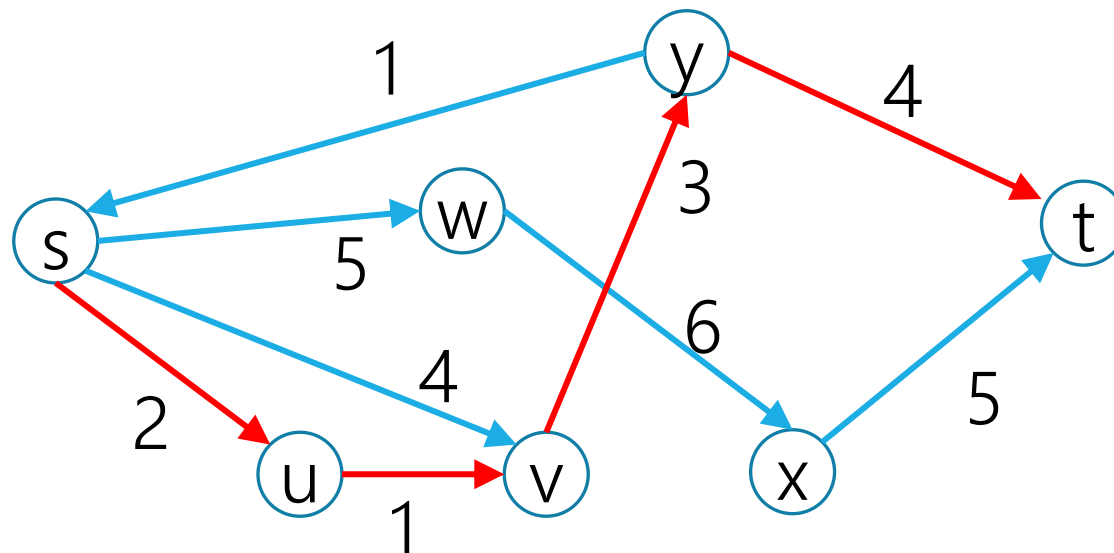


# Shortest Paths

The **length** of a path is the sum of the edge weights on that path.

## Shortest Path Problem

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

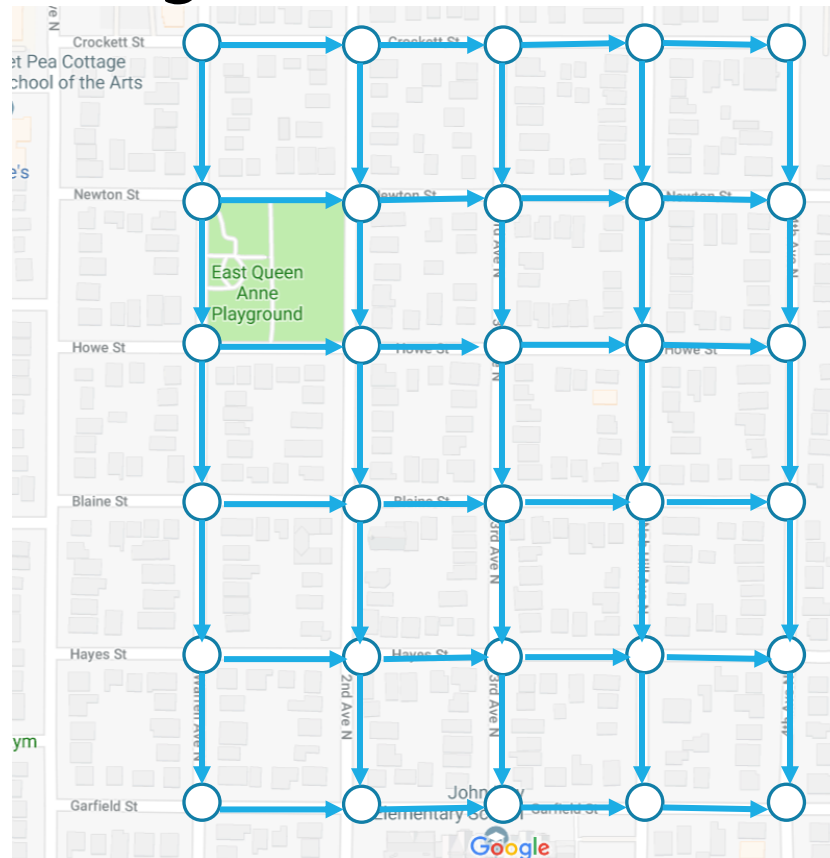




# Unweighted Graphs

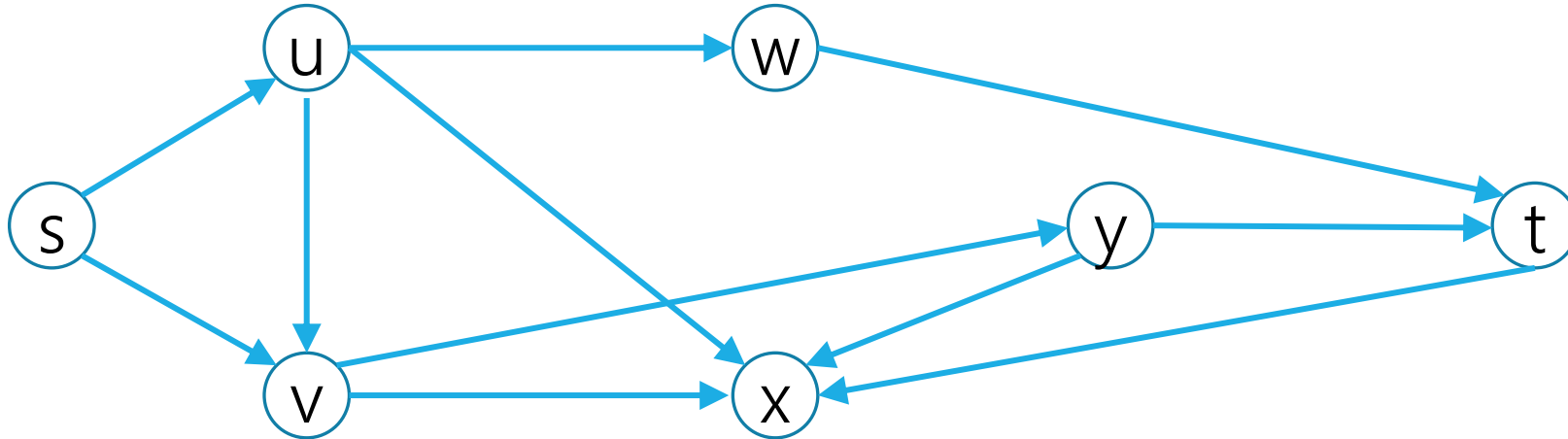
Let's start with a simpler version: the edges are all the same weight

If the graph is **unweighted**, how do we find a shortest paths?



# Unweighted Graphs

If the graph is unweighted, how do we find a shortest paths?



What's the shortest path from s to s?

Well....we're already there.

What's the shortest path from s to u or v?

Just go on the edge from s

From s to w,x, or y?

Can't get there directly from s, for length 2 path, have to go through u or v.

# Unweighted Graphs: Key Idea

To find the set of vertices at distance  $k$ , just find the set of vertices at distance  $k-1$ , and check for outgoing edge to an undiscovered vertex.

Do we already know an algorithm that does something like that?

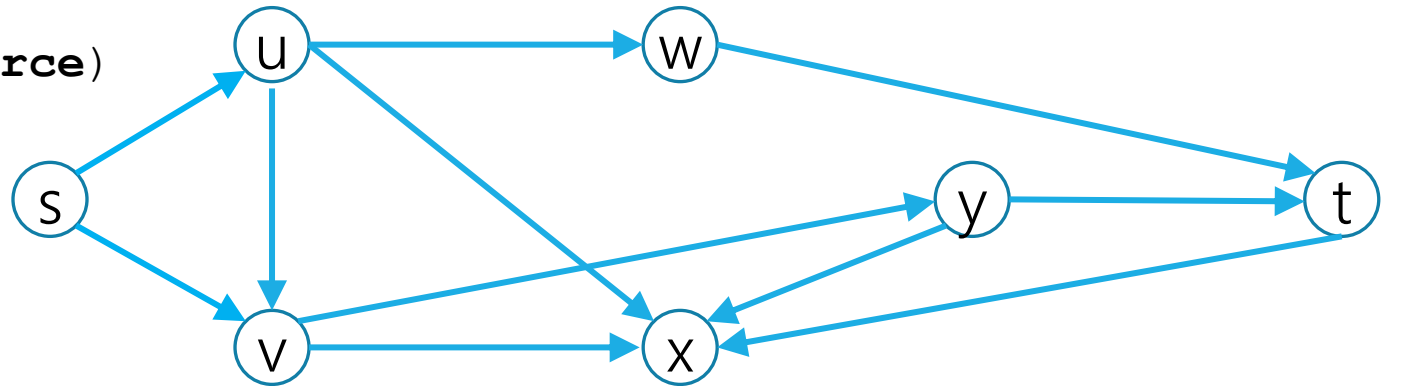
Yes! BFS!

```
    bfsShortestPaths(graph G, vertex source)
        toVisit.enqueue(source)
        source.dist = 0
        mark source as visited
        while(toVisit is not empty){
            current = toVisit.dequeue()
            for (v : current.outNeighbors()){
                if (v is not yet visited){
                    v.distance = current.distance + 1
                    v.predecessor = current
                    toVisit.enqueue(v)
                    mark v as visited
                }
            }
        }
    }
```

# Unweighted Graphs

Use BFS to find shortest paths in this graph.

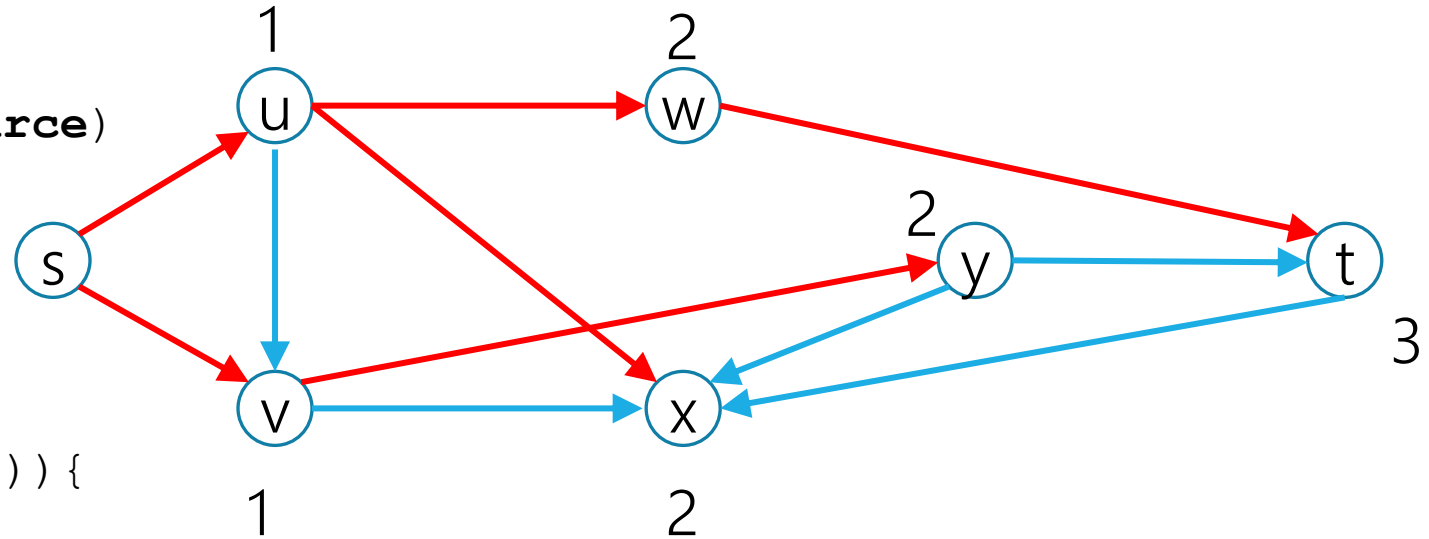
```
bfsShortestPaths(graph G, vertex source)  
  toVisit.enqueue(source)  
  source.dist = 0  
  mark source as visited  
  while(toVisit is not empty){  
    current = toVisit.dequeue()  
    for (v : current.outNeighbors()){  
      if (v is not yet visited){  
        v.distance = current.distance + 1  
        v.predecessor = current  
        toVisit.enqueue(v)  
        mark v as visited  
      }  
    }  
  }  
}
```



# Unweighted Graphs

Use BFS to find shortest paths in this graph.

```
bfsShortestPaths(graph G, vertex source)
  toVisit.enqueue(source)
  source.dist = 0
  mark source as visited
  while(toVisit is not empty){
    current = toVisit.dequeue()
    for (v : current.outNeighbors()){
      if (v is not yet visited){
        v.distance = current.distance + 1
        v.predecessor = current
        toVisit.enqueue(v)
        mark v as visited
      }
    }
  }
}
```



# What about the target vertex?

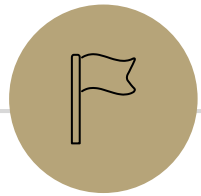
## Shortest Path Problem

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

BFS didn't mention a target vertex...

It actually finds the shortest path from  $s$  to every other vertex.

If you know your target, you can stop the algorithm early, when the target is removed from the queue.

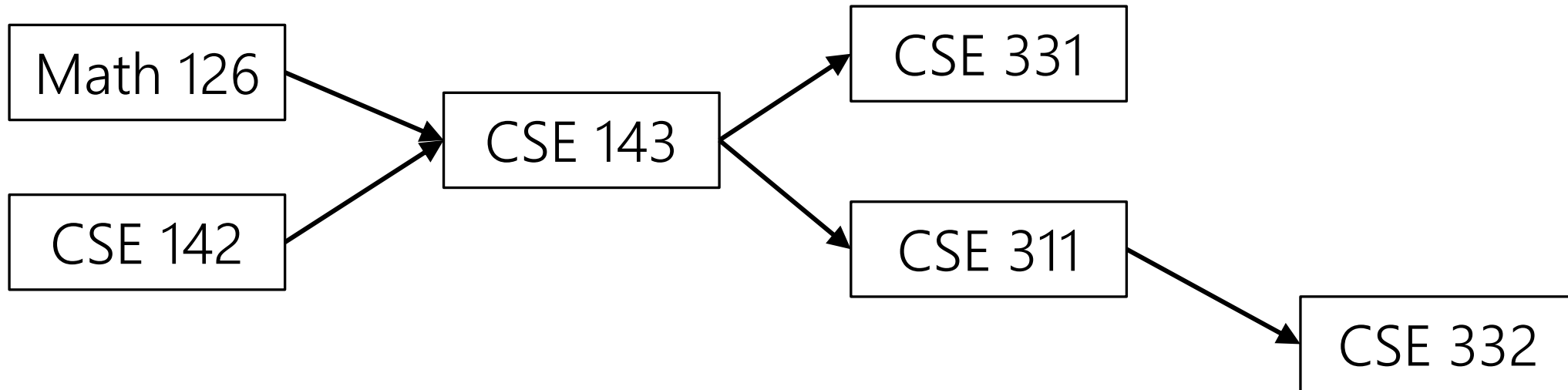


## One more Graph Algorithm

---

# Problem 1: Ordering Dependencies

Today's next problem: Given a bunch of courses with prerequisites, find an order to take the courses in.





# Problem 1: Ordering Dependencies

Given a directed graph  $G$ , where we have an edge from  $u$  to  $v$  if  $u$  must happen before  $v$ .

Can we find an order that **respects dependencies**?

## Topological Sort (aka Topological Ordering)

Given: a directed graph  $G$

Find: an ordering of the vertices so all edges go from left to right.

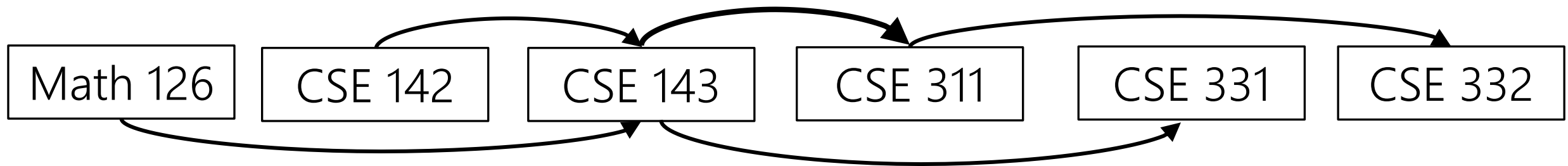
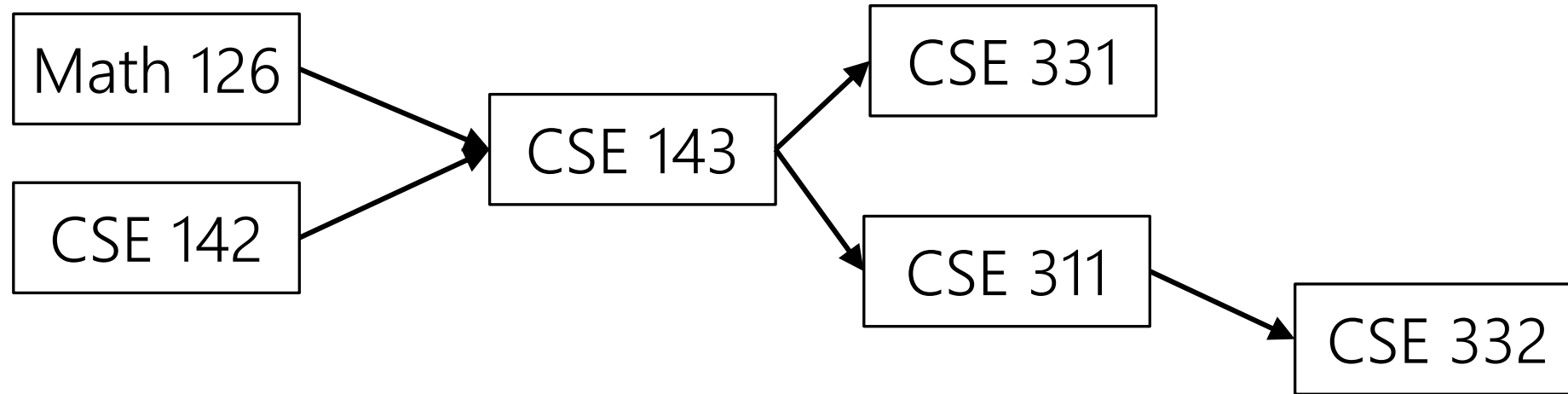
Uses:

Compiling multiple files

Graduating.

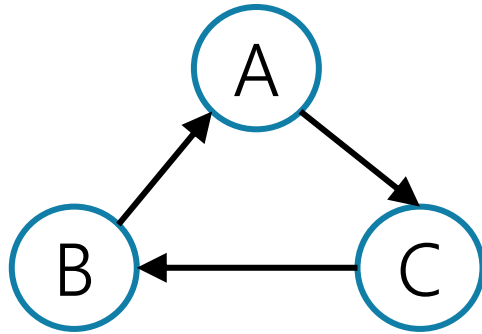
# Topological Ordering

A course prerequisite chart and a possible topological ordering.



# Can we always order a graph?

Can you topologically order this graph?



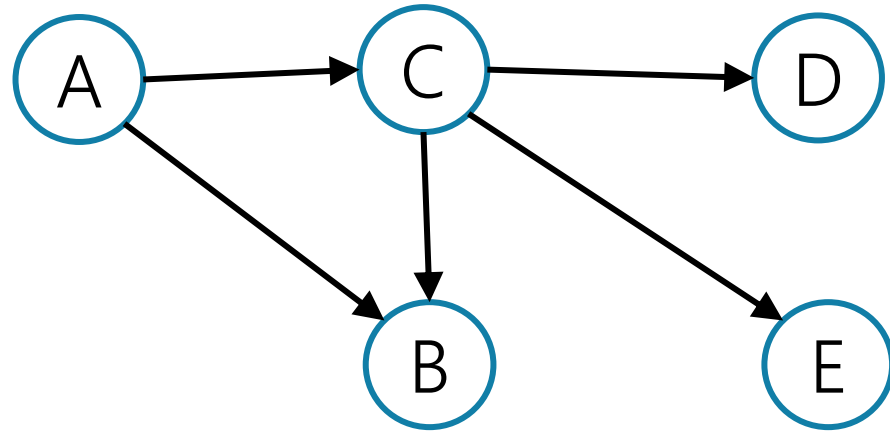
## Directed Acyclic Graph (DAG)

A directed graph without any cycles.

A graph has a topological ordering if and only if it is a DAG.

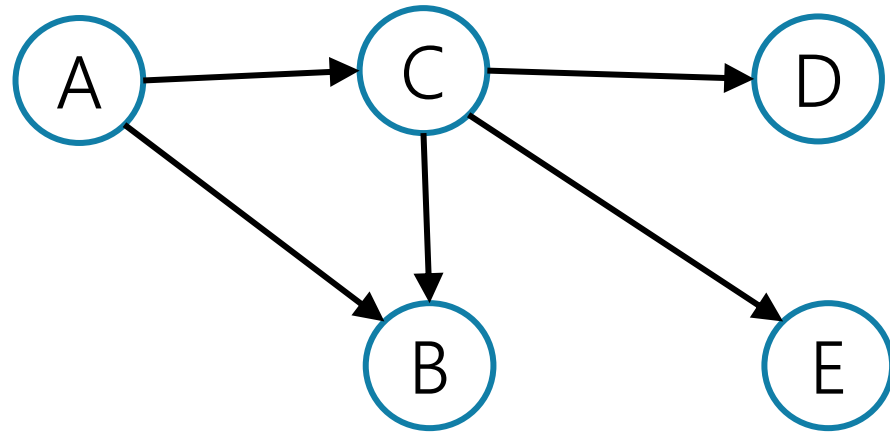
# Ordering a DAG

Does this graph have a topological ordering? If so find one.



# Ordering a DAG

Does this graph have a topological ordering? If so find one.



If a vertex doesn't have any edges going into it, we can add it to the ordering.

More generally, if the only incoming edges are from vertices already in the ordering, it's safe to add.

# How Do We Find a Topological Ordering?

```
TopologicalSort(Graph G, Vertex source)
    count how many incoming edges each vertex has
    Collection toProcess = new Collection()
    foreach(Vertex v in G){
        if(v.edgesRemaining == 0)
            toProcess.insert(v)
    }
    topOrder = new List()
    while(toProcess is not empty){
        u = toProcess.remove()
        topOrder.insert(u)
        foreach(edge (u,v) leaving u){
            v.edgesRemaining--
            if(v.edgesRemaining == 0)
                toProcess.insert(v)
        }
    }
}
```

# What's the running time?

```
TopologicalSort(Graph G, Vertex source)
    count how many incoming edges each vertex has
    Collection toProcess = new Collection()
    foreach(Vertex v in G){
        if(v.edgesRemaining == 0)
            toProcess.insert(v)
    }
    topOrder = new List()
    while(toProcess is not empty){
        u = toProcess.remove()
        topOrder.insert(u)
        foreach(edge (u,v) leaving u)
            v.edgesRemaining--
            if(v.edgesRemaining == 0)
                toProcess.insert(v)
    }
}
```

Running Time:  $O(|V| + |E|)$

# Finding a Topological Ordering

Instead of counting incoming edges, you can actually modify DFS to find you one (think about why).

But the "count incoming edges" is a bit easier to understand (for me 😊 )



# What we've seen so far

Either BFS or DFS traverse a graph (if you're careful about disconnected graphs).

BFS goes steadily through the graph

- Useful for computing shortest paths in unweighted graphs

DFS explores deep into the graph as long as the thing is new

- Useful for finding cycles
- And for other applications...

Next time: Another algorithm for finding shortest paths, but when you have a weighted graph.