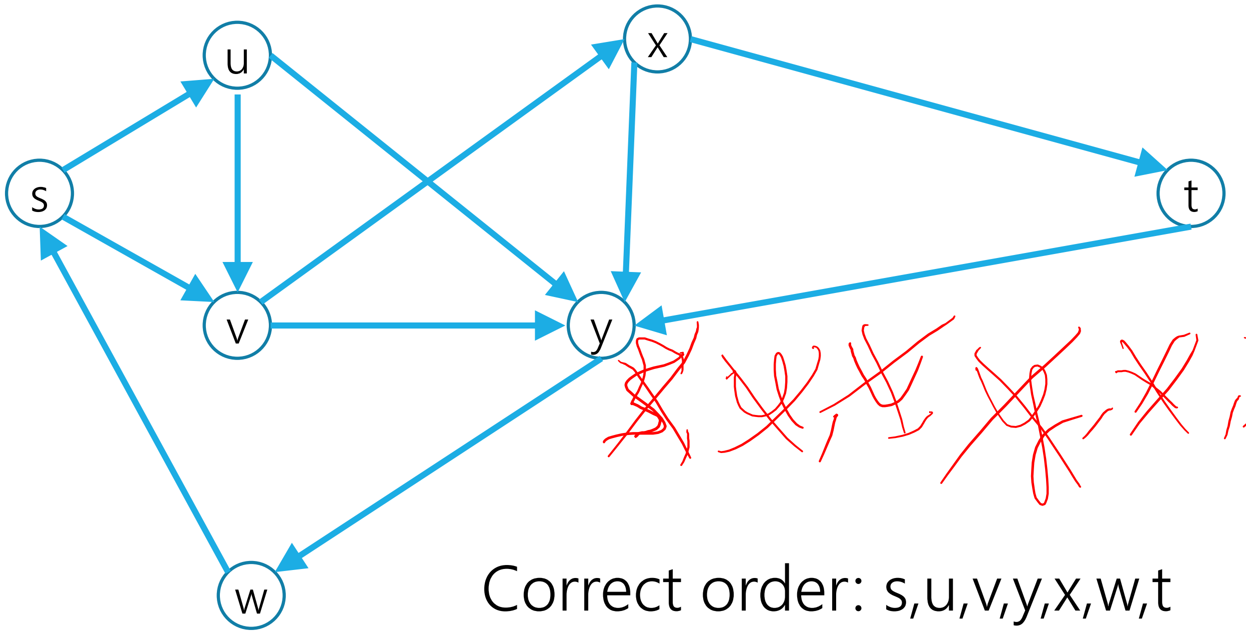
Warm Up

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.

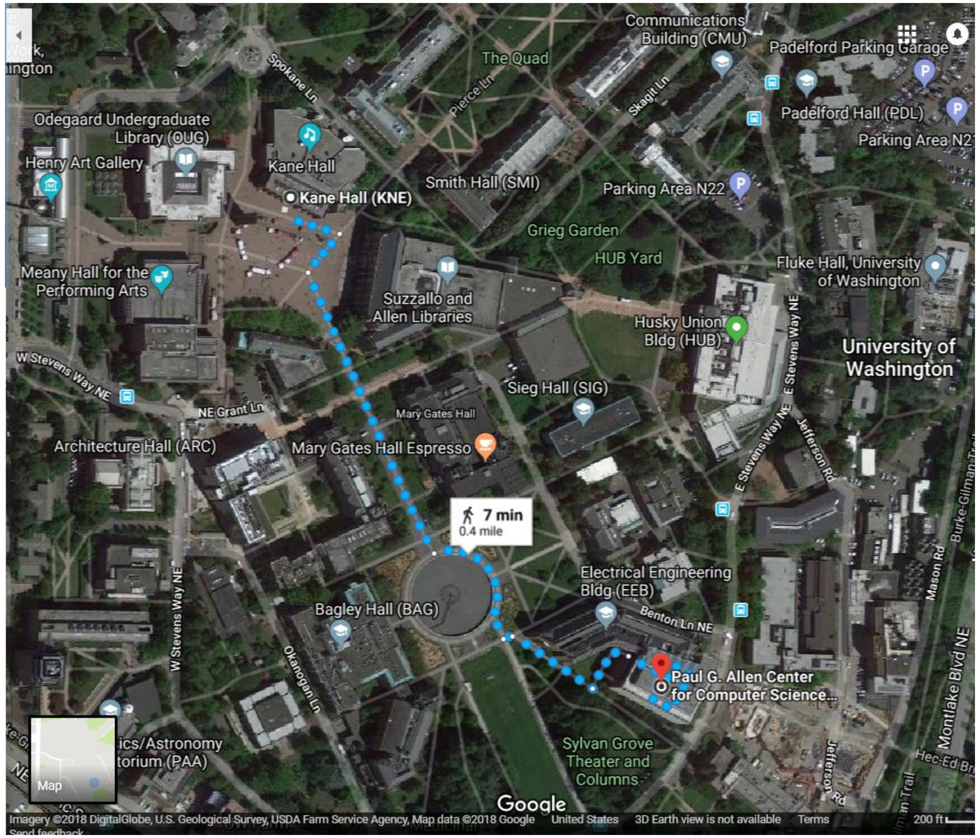



```
bfs(graph)
    toVisit.enqueue(first vertex)
    mark first vertex as visited
    while(toVisit is not empty)
        current = toVisit.dequeue()
        for (V : current.outneighbors())
            if (v is not visited)
                toVisit.enqueue(v)
                mark v as visited
      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

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

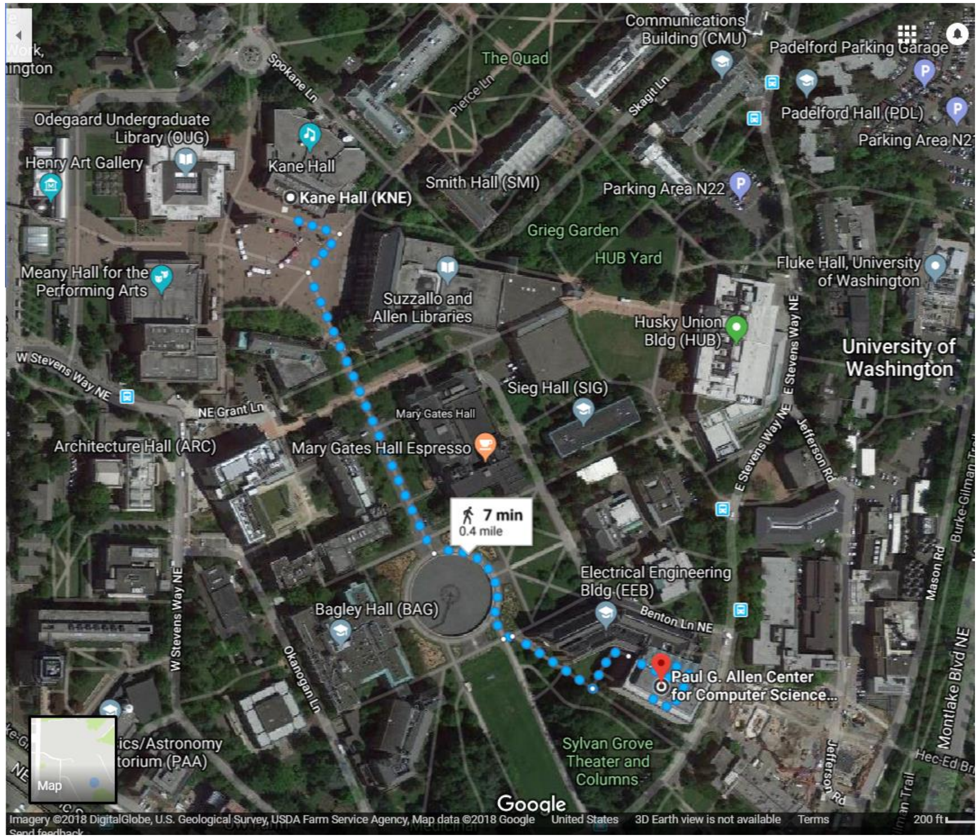
Shortest Paths

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

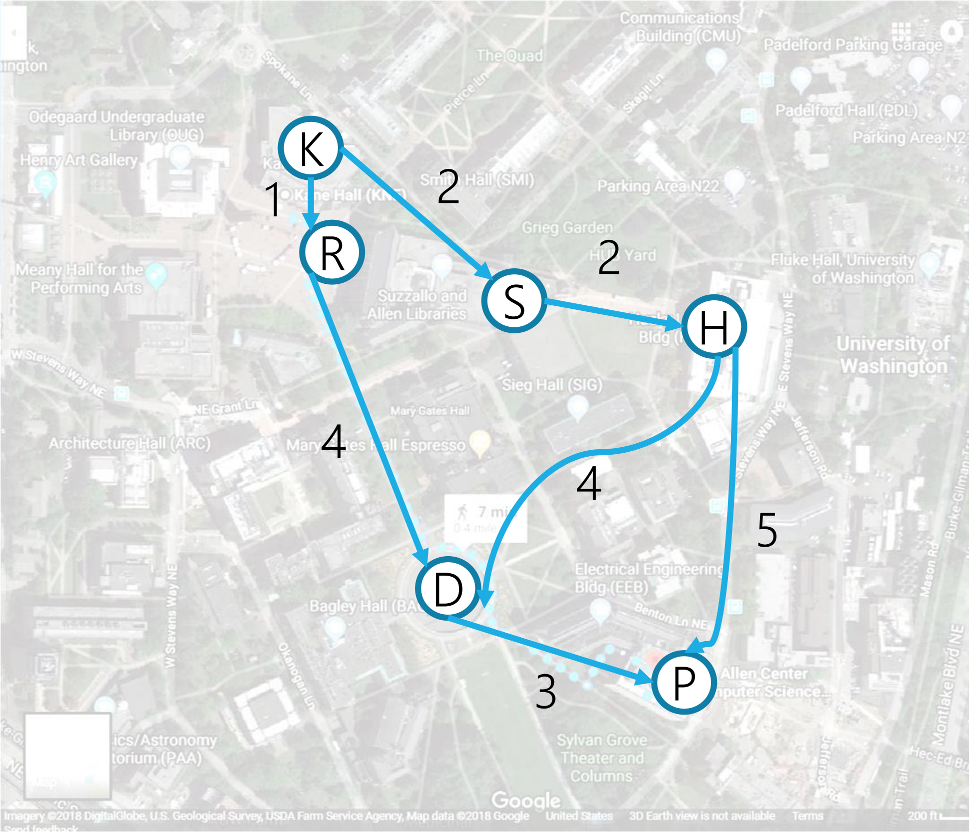


Representing Maps as Graphs

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



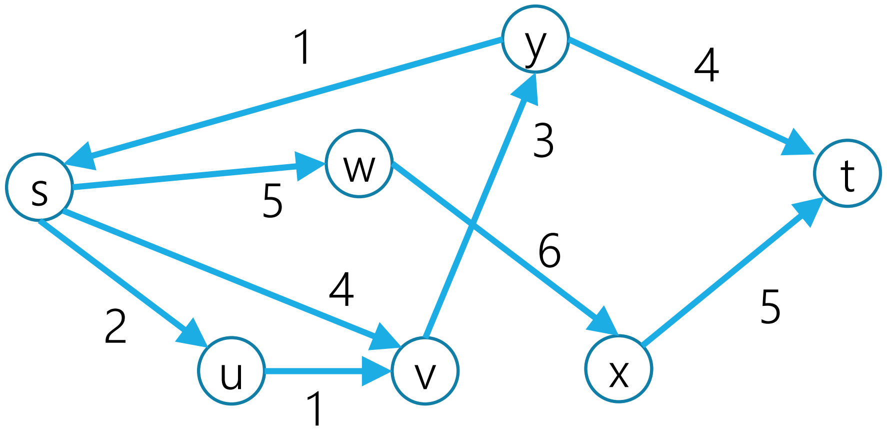
Representing Maps as Graphs



Shortest Paths

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

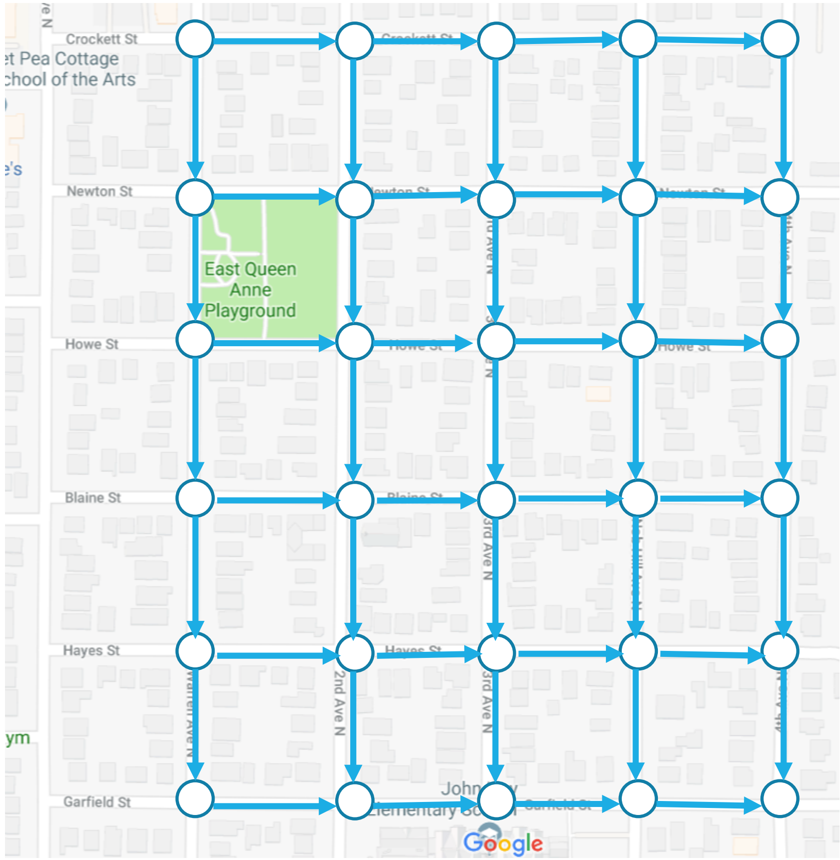
Shortest Path Problem
Given: a directed graph G and vertices s and t
Find: the shortest path from s to t



Unweighted graphs

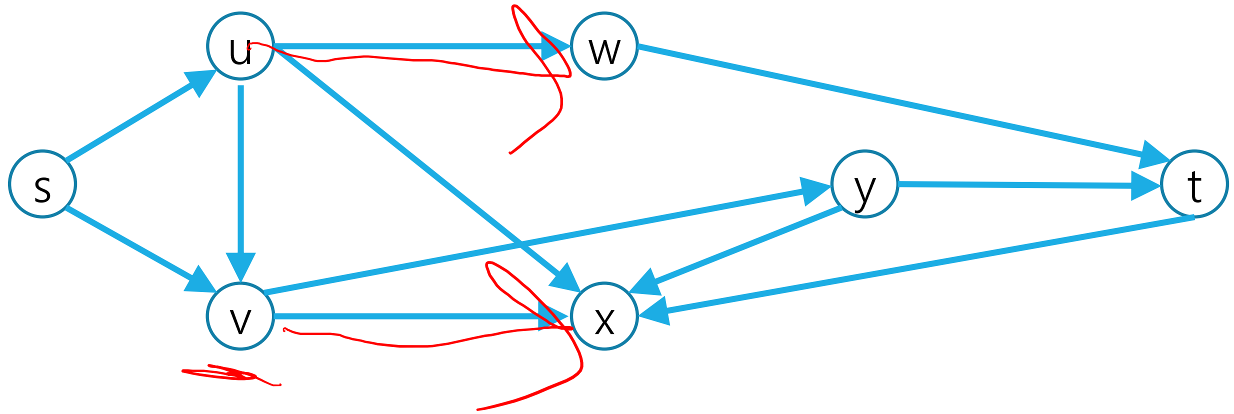
Let’s start with a simpler version: the edges are all the same weight (**unweighted**)

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, if we want a 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 see if any of them have an outgoing edge to an undiscovered vertex.

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

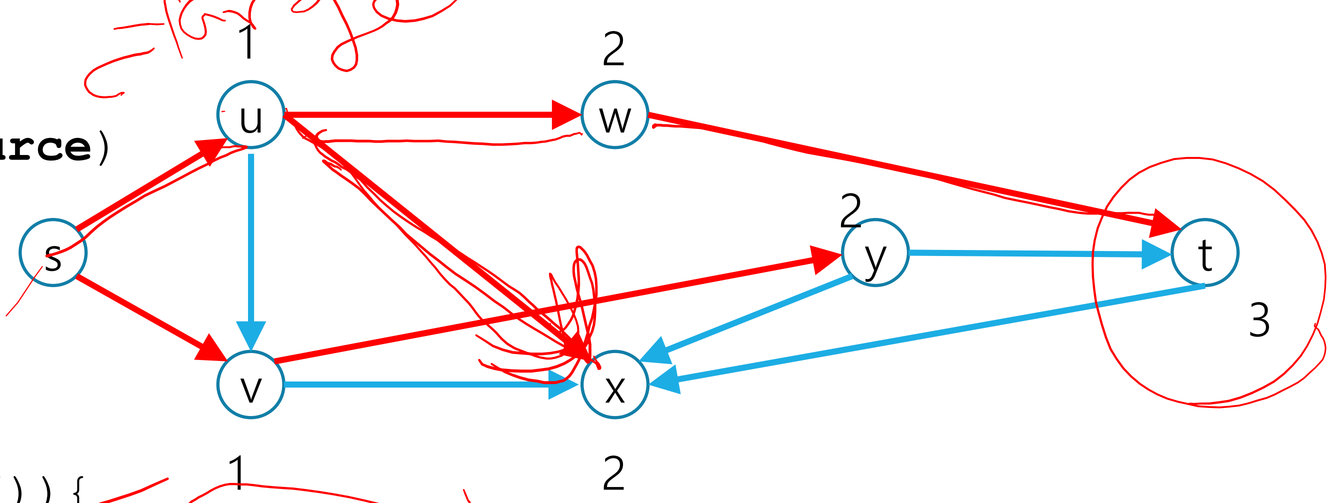
Yes! BFS!

```
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

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

```
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
         }
      }
}
```



suwt

What about the target vertex?

Shortest Path Problem

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

BFS didn't mention a target vertex…
It actually finds the 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.

Weighted Graphs

Each edge should represent the "time" or "distance" from one vertex to another.

Sometimes those aren't uniform, so we put a weight on each edge to record that number.

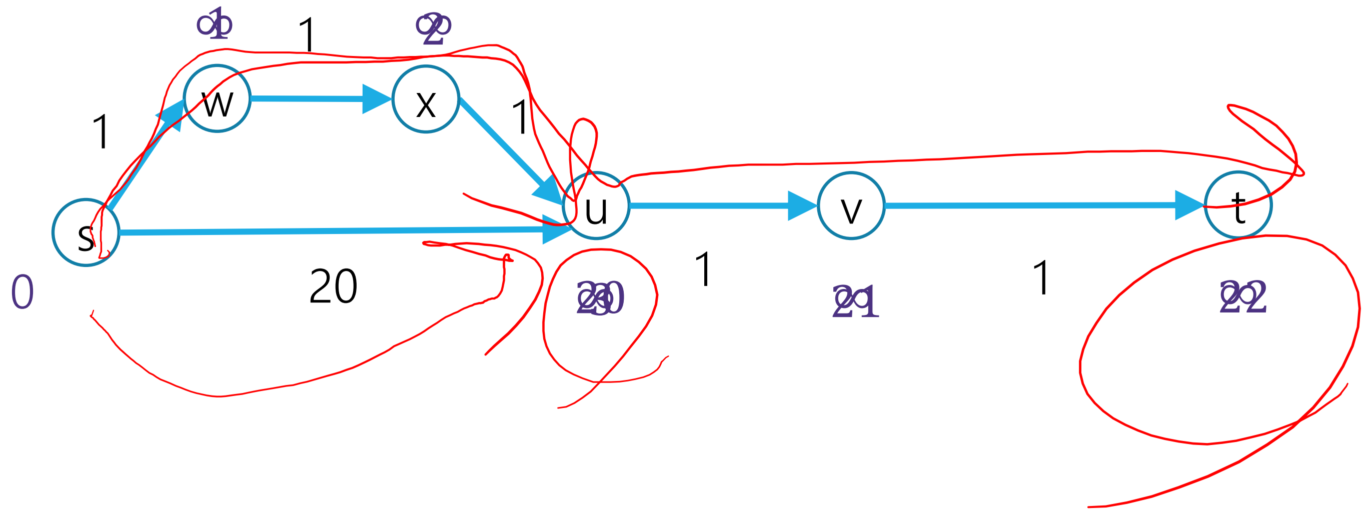
The length of a path in a weighted graph is the sum of the weights along that path.

We'll assume all of the weights are positive

- For GoogleMaps that definitely makes sense.
- Sometimes negative weights make sense. Today's algorithm doesn't work for those graphs
- There are other algorithms that do work.

Weighted Graphs: Take 1

BFS works if the graph is unweighted. Maybe it just works for weighted graphs too?



What went wrong? When we found a shorter path from s to u, we needed to update the distance to v (and anything whose shortest path went through u) but BFS doesn't do that.

Weighted Graphs: Take 2

Reduction (informally)

Using an algorithm for Problem B to solve Problem A.

You already do this all the time.

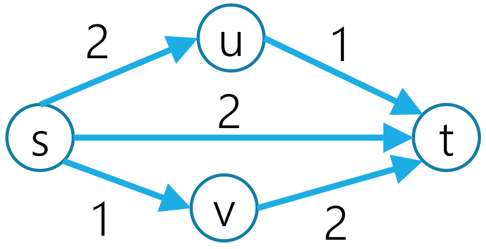
In a previous project, you reduced implementing a hashset to implementing a hashmap.

Any time you use a library, you're reducing your problem to the one the library solves.

Can we reduce finding shortest paths on weighted graphs to finding them on unweighted graphs?

Weighted Graphs: A Reduction

Given a weighted graph, how do we turn it into an unweighted one without messing up the edge lengths?



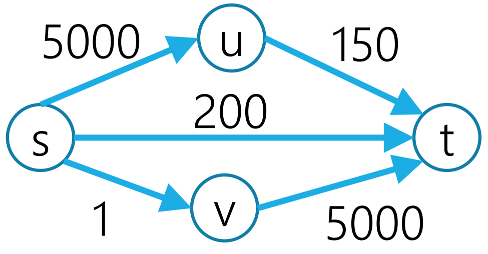
Transform Input

Unweighted Shortest Paths

Transform Output

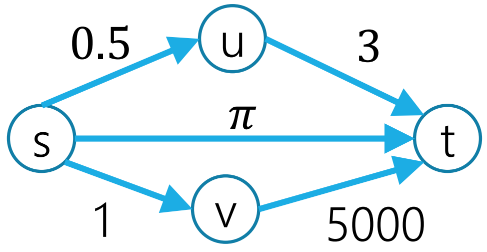
Weighted Graphs: A Reduction

What is the running time of our reduction on this graph?



O(|V|+|E|) of the modified graph, which is…slow.

Does our reduction even work on this graph?



Ummm….

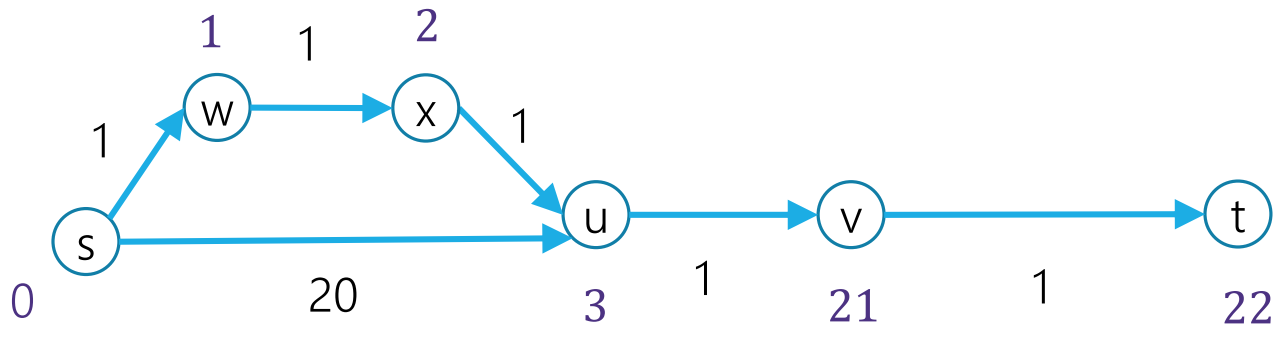
Tl;dr: If your graph's weights are all small positive integers, this reduction might work great. Otherwise we probably need a new idea.

Weighted Graphs: Take 3

So we can't just do a reduction.

Instead let's try to figure out why BFS worked in the unweighted case, and try to make the same thing happen in the weighted case.

Why did BFS work on unweighted graphs? How did we avoid this problem:



When we used a vertex u to update shortest paths we already knew the exact shortest path to u. So we never ran into the update problem

So if we process the vertices in order of distance from s, we have a chance.

Weighted Graphs: Take 3

Goal: Process the vertices in order of distance from s

Idea:

Have a set of vertices that are “known”

- (we know at least one path from s to them).

Record an estimated distance

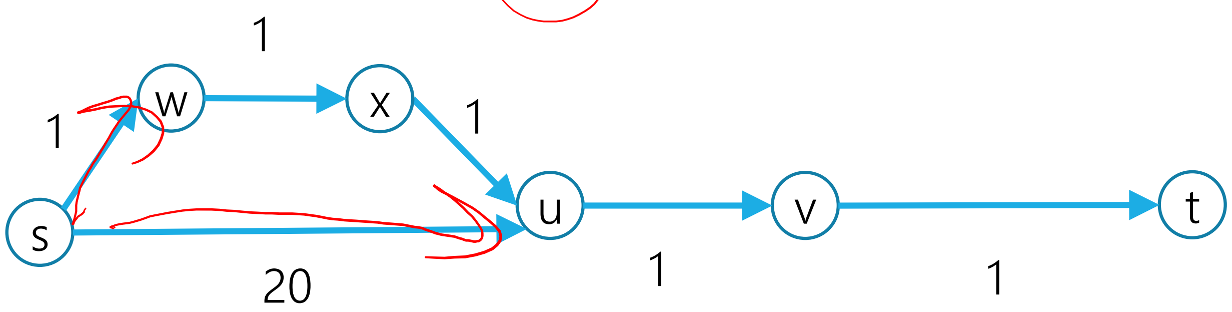
- (the best way we know to get to each vertex).

If we process only the vertex closest in estimated distance, we won’t ever find a shorter path to a processed vertex.

Dijkstra's Algorithm

```
Dijkstra(Graph G, Vertex source)
   initialize distances to ∞
   mark source as distance 0
   mark all vertices unprocessed
   while(there are unprocessed vertices){
      let u be the closest unprocessed vertex
      foreach(edge (u,v) leaving u){
         if(u.dist+w(u,v) < v.dist){
            v.dist = u.dist+w(u,v)
            v.predecessor = u
         }
      }
     mark u as processed
   }
```

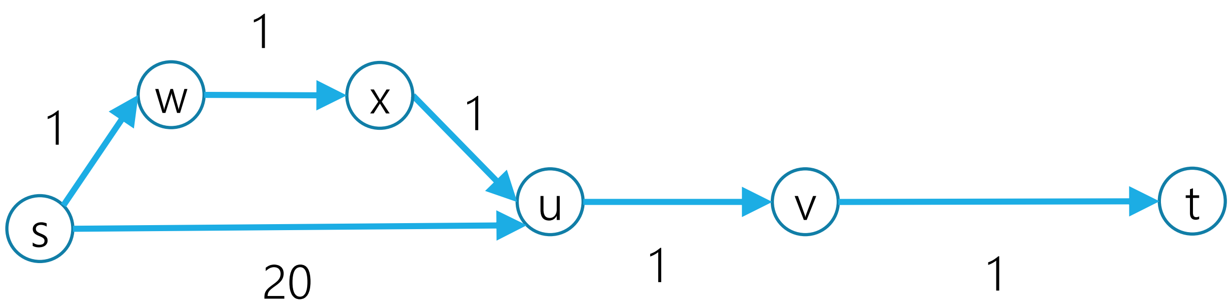
Vertex	Distance	Predecessor	Processed
s	0	--	✓
w	1	s	✓
x	2	w	✓
u	3	x	✓
v	4	u	✓
t	5	v	✓



Dijkstra's Algorithm

```
Dijkstra(Graph G, Vertex source)
   initialize distances to ∞
   mark source as distance 0
   mark all vertices unprocessed
   while(there are unprocessed vertices){
      let u be the closest unprocessed vertex
      foreach(edge (u,v) leaving u){
         if(u.dist+w(u,v) < v.dist){
            v.dist = u.dist+w(u,v)
            v.predecessor = u
         }
      }
      mark u as processed
   }
```

Vertex	Distance	Predecessor	Processed
s	0	--	Yes
w	1	s	Yes
x	2	w	Yes
u	3	x	Yes
v	4	u	Yes
t	5	v	Yes



Implementation Details

One of those lines of pseudocode was a little sketchy

```
> let u be the closest unprocessed vertex
```

What ADT have we talked about that might work here?

Minimum Priority Queues!

Min Priority Queue ADT
state Set of comparable values - Ordered based on "priority"
behavior removeMin() – returns the element with the smallest priority, removes it from the collection peekMin() – find, but do not remove the element with the smallest priority insert(value) – add a new element to the collection

Making Minimum Priority Queues Work

They won't quite work "out of the box".

We don't have an update priority method. Can we add one?

- Percolate up!

To percolate u's entry in the heap up we'll have to get to it.

- Each vertex need pointer to where it appears in the priority queue
- I'm going to ignore this point for the rest of the lecture.

Min Priority Queue ADT
state Set of comparable values - Ordered based on "priority" **behavior** **removeMin()** – returns the element with the smallest priority, removes it from the collection **peekMin()** – find, but do not remove the element with the smallest priority **insert(value)** – add a new element to the collection **DecreasePriority(e, p)** – decreases the priority of element e down to p.

Running Time Analysis

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

```
Dijkstra(Graph G, Vertex source)
   initialize distances to ∞, source.dist to 0
   mark all vertices unprocessed
   initialize MPQ as a Min Priority Queue
   add source at priority 0
   while(MPQ is not empty){
      u = MPQ.removeMin()
      foreach(edge (u,v) leaving u){
         if(u.dist+w(u,v) < v.dist){
            if(v.dist == ∞ ) //if v not in MPQ
               MPQ.insert(v, u.dist+w(u,v))
            else
               MPQ.decreasePriority(v, u.dist+w(u,v))
            v.dist = u.dist+w(u,v)
            v.predecessor = u
         }
      }
      mark u as processed
   }
```

$O(\log|V|)$

$O(\log|V|)$

Another Application of Shortest Paths

Shortest path algorithms are obviously useful for GoogleMaps.

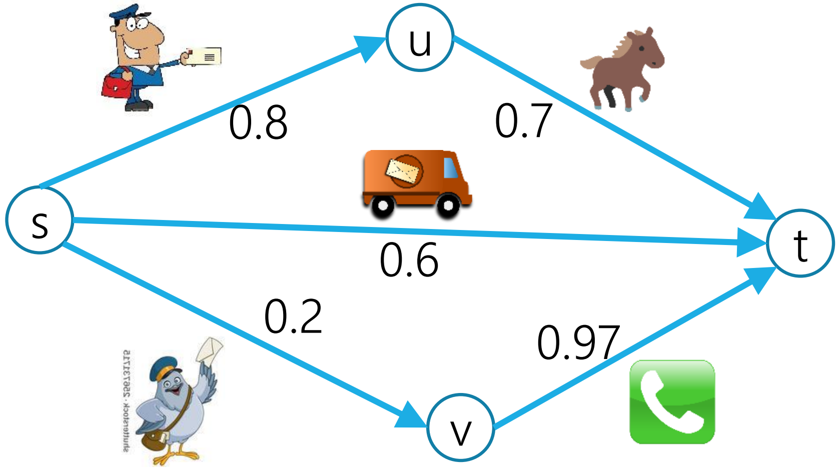
The wonderful thing about graphs is they can encode **arbitrary** relationships among objects.

I don't care if you remember all the details.

I just want you to see that these algorithms have non-obvious applications.

Another Application of Shortest Paths

I have a message I need to get from point s to point t.
But the connections are unreliable.
What path should I send the message along so it has the best chance of arriving?



Maximum Probability Path

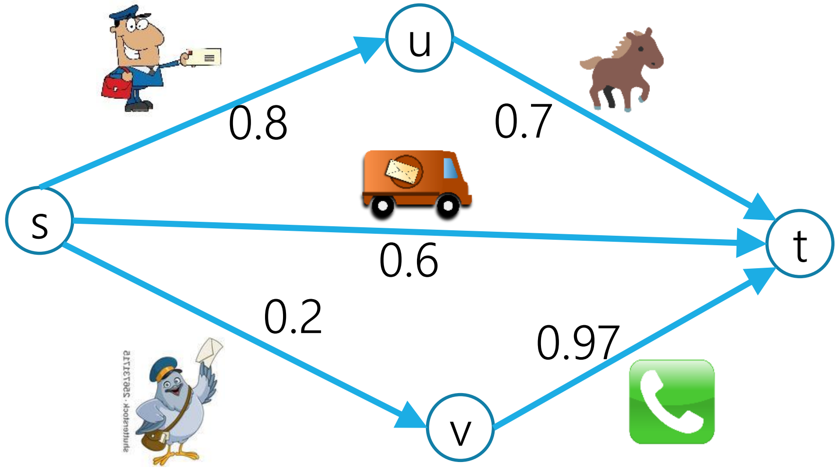
Given: a directed graph G, where each edge weight is the probability of successfully transmitting a message across that edge
Find: the path from s to t with maximum probability of message transmission

Another Application of Shortest Paths

Let each edge's weight be the probability a message is sent successfully across the edge.

What's the probability we get our message all the way across a path?

- It's the product of the edge weights.

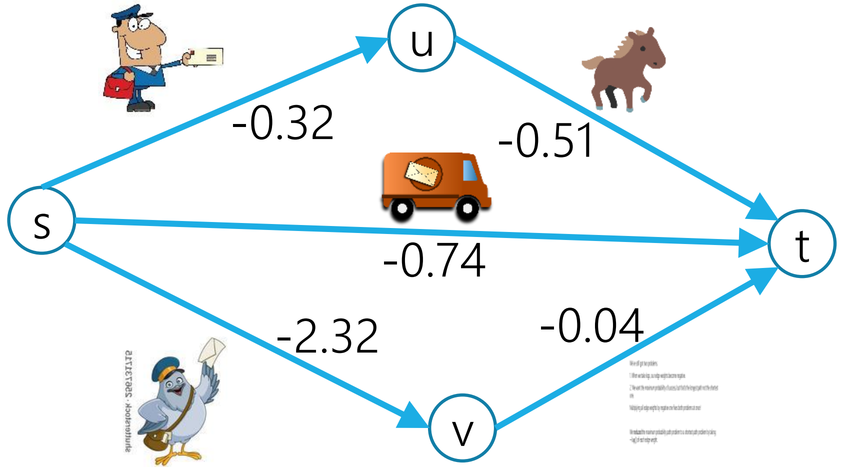


We only know how to handle sums of edge weights.

Is there a way to turn products into sums?

$\log(ab) = \log a + \log b$

Another Application of Shortest Paths



We've still got two problems.

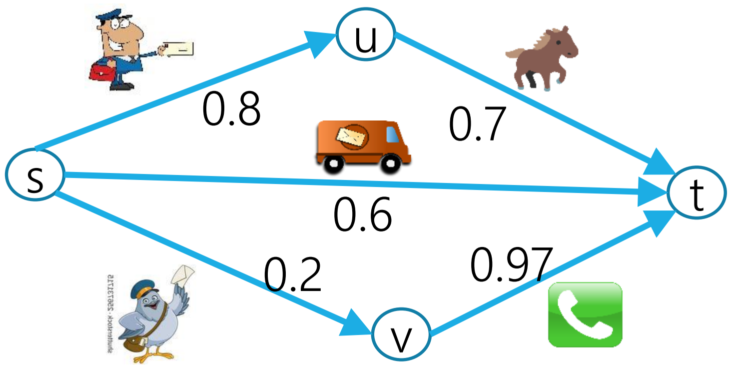
1. When we take logs, our edge weights become negative.

2. We want the *maximum* probability of success, but that's the longest path not the shortest one.

Multiplying all edge weights by negative one fixes both problems at once!

We **reduced** the maximum probability path problem to a shortest path problem by taking $-\log()$ of each edge weight.

Maximum Probability Path Reduction



O(E)

Transform Input

O(V log V + E log V)

Weighted Shortest Paths

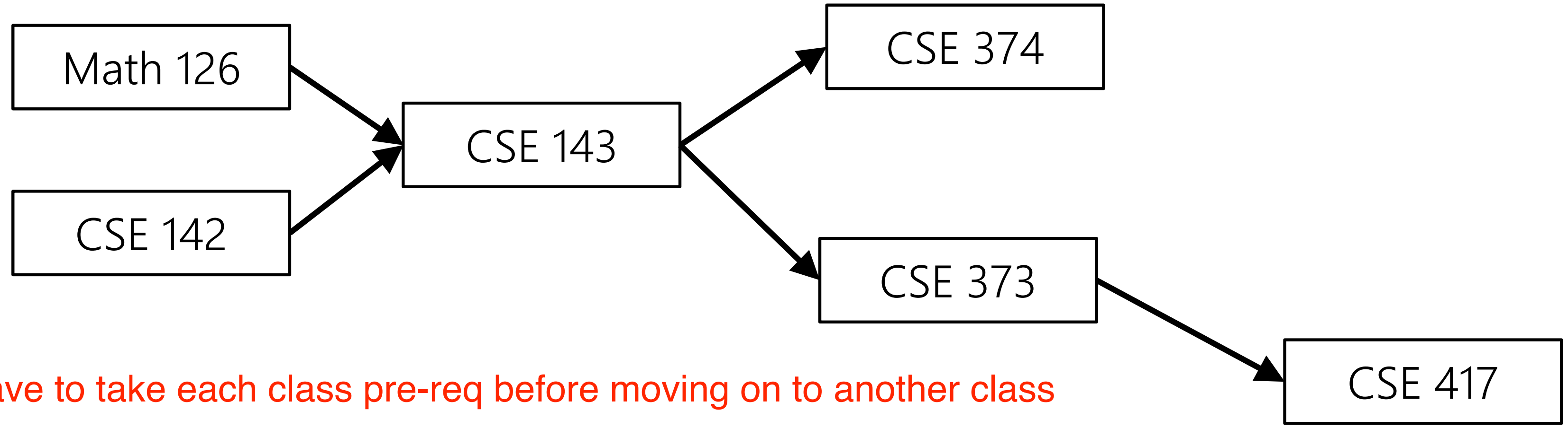
Reduction steps must NOT be more costly than the algorithm… otherwise it kind of ruins the point of the reduction

O(E)

Transform Output

Problem 1: Ordering Dependencies

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



We have to take each class pre-req before moving on to another class

i.e. the later nodes in the graph DEPEND on the ones before them

You can only enter a node if it has no incoming edges (assuming you delete nodes as you take the class)

Problem 1: Ordering Dependencies

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

We can only do things one at a time, can we find an order that **respects dependencies**?

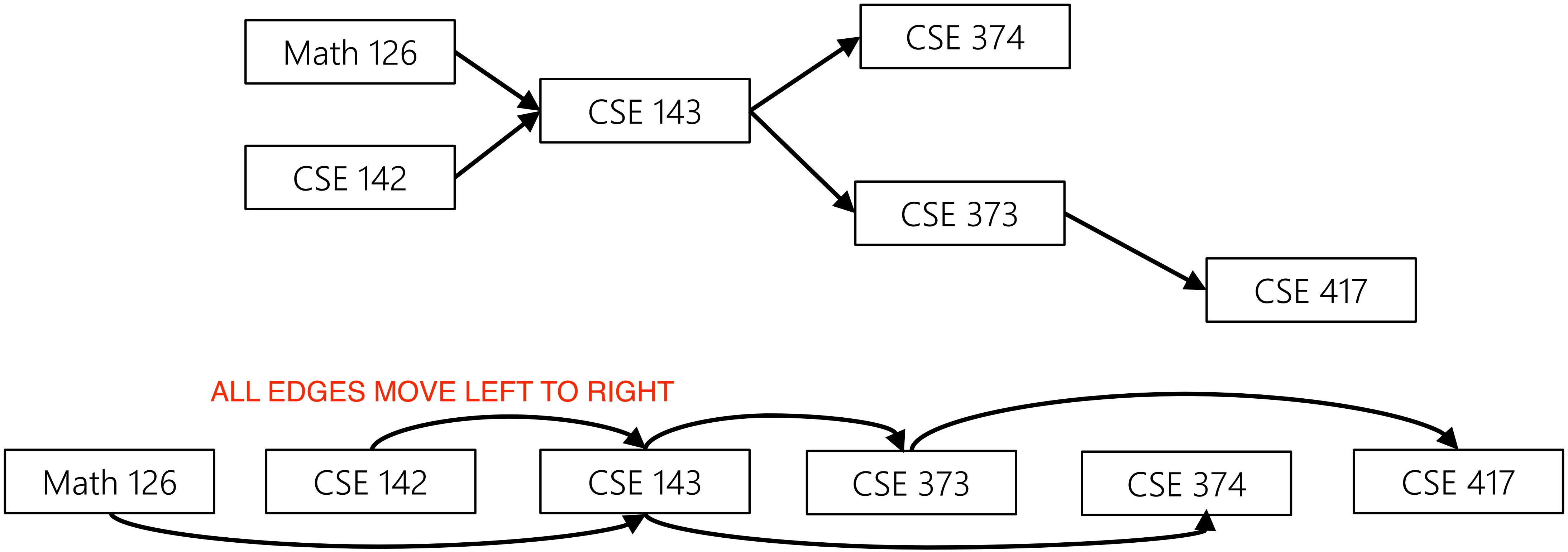
Topological Sort (aka Topological Ordering)

Given: a directed graph G
Find: an ordering of the vertices so all edges go from left to right.

Uses:
Compiling multiple files
Graduating.

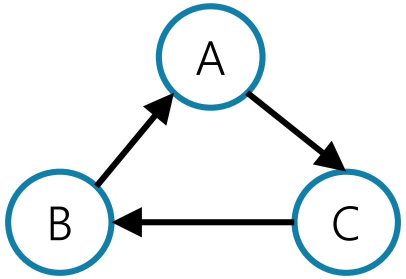
Topological Ordering

A course prerequisite chart and a possible topological ordering.



Can we always order a graph?

Can you topologically order this graph?



NO. graph cannot have cycles

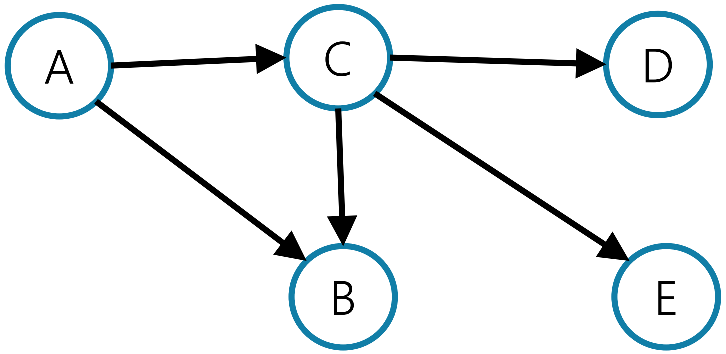
Directed Acyclic Graph (DAG)

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.



Only node with no in-edges = A
remove A

Only node with no in-edges = C
remove C

B,D,E all have no in-edges now
can remove them in any order

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)
        }
    }
```

- modify BFS $O(|V|+|E|)$ time.

choose a data structure with $O(1)$ inserts & removes.

$O(|V|)$

$O(|V|)$

Runs at most once per edge in whole algorithm $O(|E|)$.

Total: $O(|V|+|E|)$.

Strongly Connected Components

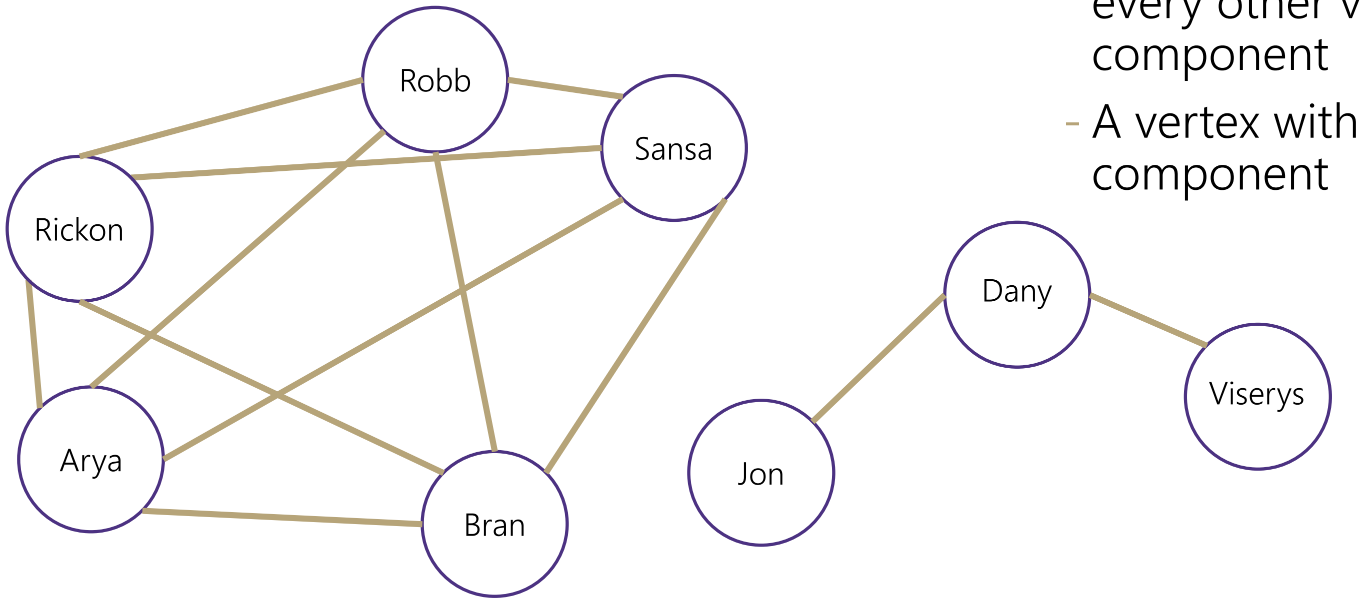
Review: Connected [Undirected] Graphs

Connected graph – a graph where every vertex is connected to every other vertex via some path. It is not required for every vertex to have an edge to every other vertex

There exists some way to get from each vertex to every other vertex

Connected Component – a *subgraph* in which any two vertices are connected via some path, but is connected to no additional vertices in the *supergraph*

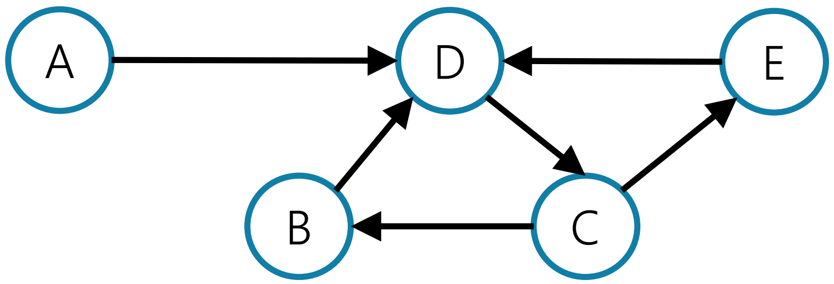
- There exists some way to get from each vertex within the connected component to every other vertex in the connected component
- A vertex with no edges is itself a connected component



Review Strongly Connected Components

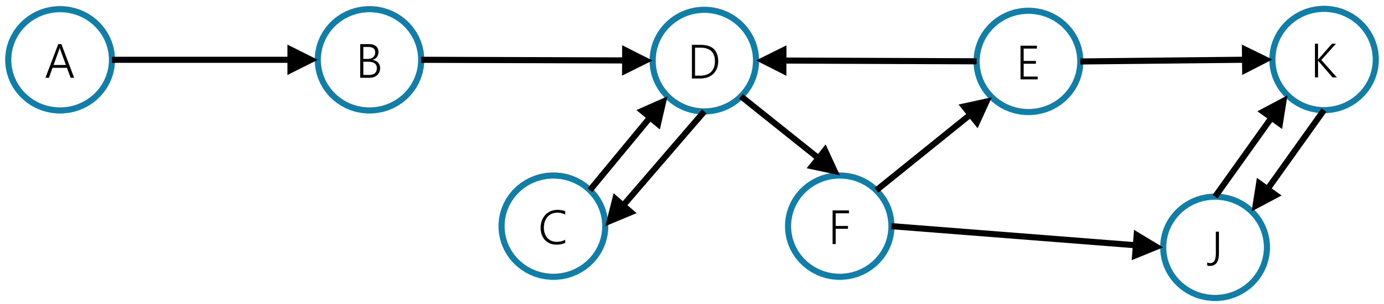
Strongly Connected Component

A subgraph C such that every pair of vertices in C is connected via some path **in both directions,** and there is no other vertex which is connected to every vertex of C in both directions.



Note: the direction of the edges matters!

Strongly Connected Components Problem



{A}, {B}, {C,D,E,F}, {J,K}

Strongly Connected Components Problem

Given: A directed graph G
Find: The strongly connected components of G

SCC Algorithm

Ok. How do we make a computer do this?

You could:

- run a [B/D]FS from every vertex,
- For each vertex record what other vertices it can get to
- and figure it out from there.

But you can do better. There's actually an O(|V|+|E|) algorithm!

I only want you to remember two things about the algorithm:

- It is an application of depth first search.
- It runs in linear time

The problem with running a [B/D]FS from every vertex is you recompute a lot of information.

The time you are popped off the stack in DFS contains a "smart" ordering to do a second DFS where you don't need to recompute that information.

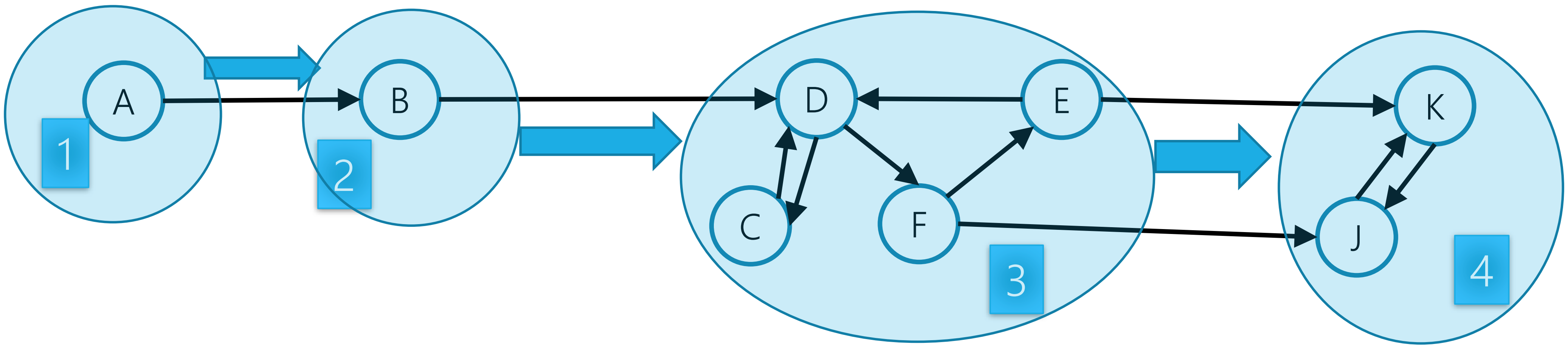
Why Find SCCs?

Graphs are useful because they encode relationships between arbitrary objects.

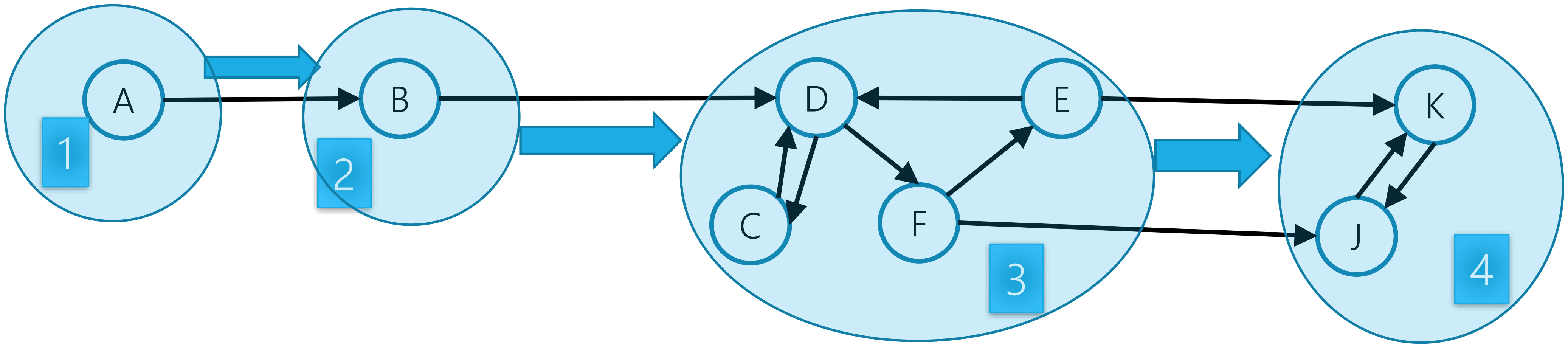
We've found the strongly connected components of G.

Let's build a new graph out of them! Call it H

- Have a vertex for each of the strongly connected components
- Add an edge from component 1 to component 2 if there is an edge from a vertex inside 1 to one inside 2.



Why Find SCCs?



That's awful meta. Why?

This new graph summarizes reachability information of the original graph.

- I can get from A (of G) in 1 to F (of G) in 3 if and only if I can get from 1 to 3 in H.

Why Must H Be a DAG?

H is always a DAG (do you see why?).

Takeaways

Finding SCCs lets you **collapse** your graph to the meta-structure.
If (and only if) your graph is a DAG, you can find a topological sort of your graph.

Both of these algorithms run in linear time.

Just about everything you could want to do with your graph will take at least as long.

You should think of these as **"almost free" preprocessing** of your graph.

- Your other graph algorithms only need to work on
  - topologically sorted graphs and
  - strongly connected graphs.

A Longer Example

The best way to really see why this is useful is to do a bunch of examples.

Take CSE 417 for that. The second best way is to see one example right now…

This problem doesn't *look like* it has anything to do with graphs

- no maps
- no roads
- no social media friendships

Nonetheless, a graph representation is the best one.

I don't expect you to remember this problem.

I just want you to see

- graphs can show up anywhere.
- SCCs and Topological Sort are useful algorithms.

Example Problem: Final Creation

We have a long list of types of problems we might want to put on the final.

- Heap insertion problem, big-O problems, finding closed forms of recurrences, testing…

To try to make you all happy, we might ask for your preferences. Each of you gives us two preferences of the form "I [do/don't] want a [] problem on the final" *

We'll assume you'll be happy if you get at least one of your two preferences.

Final Creation Problem

Given: A list of 2 preferences per student.
Find: A set of questions so every student gets at least one of their preferences (or accurately report no such question set exists).

*This is NOT how Ben is making the final.

Final Creation: Take 1

We have Q kinds of questions and S students.

What if we try every possible combination of questions.

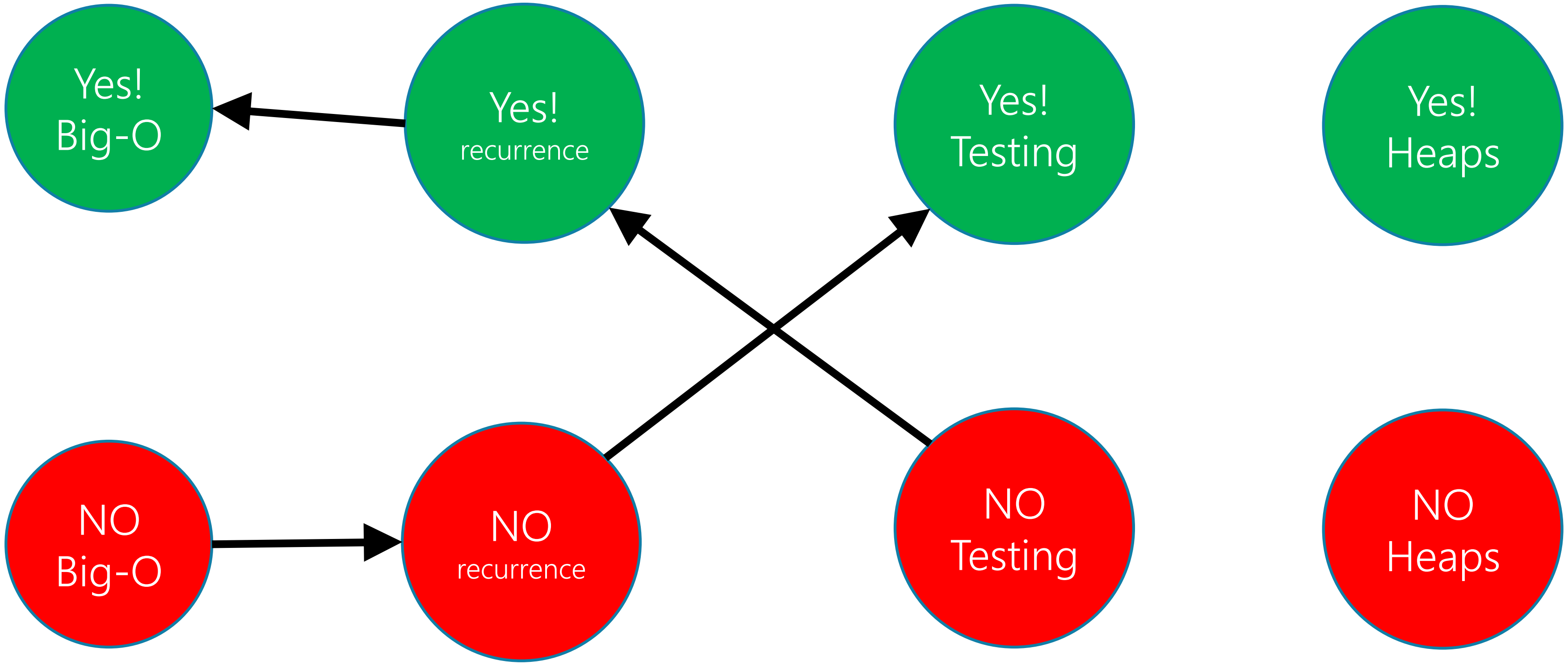
How long does this take? $O(2^Q S)$

If we have a lot of questions, that's **really** slow.

Final Creation: Take 2

Each student introduces new relationships for data:

Let's say your preferences are represented by this table:



Problem	YES	NO
Big-O	X	
Recurrence		X
Testing		
Heaps		

Problem	YES	NO
Big-O		
Recurrence	X	
Testing	X	
Heaps		

If we don't include a big-O proof, can you still be happy?
If we do include a recurrence can you still be happy?

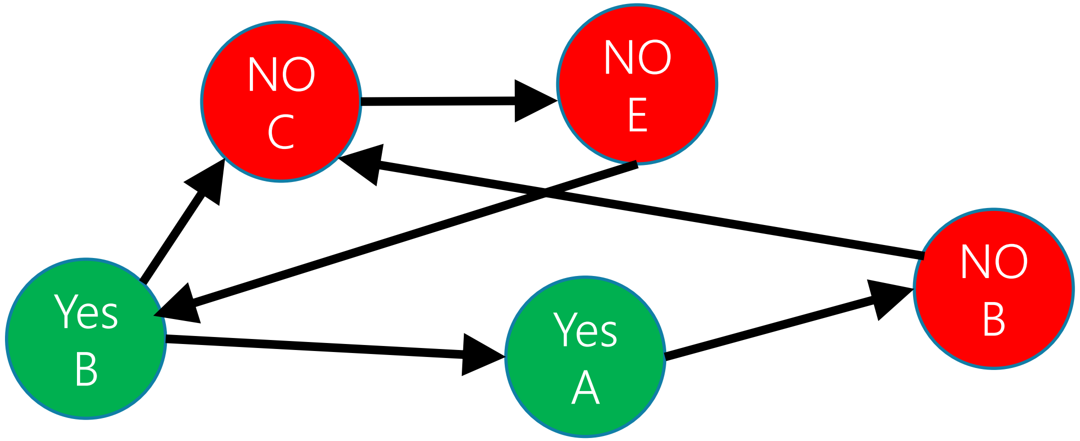
Final Creation: Take 2

Hey we made a graph!

What do the edges mean?

- We need to avoid an edge that goes TRUE THING → FALSE THING

Let's think about a single SCC of the graph.



Can we have a true and false statement in the same SCC?

What happens now that Yes B and NO B are in the same SCC?

Final Creation: SCCs

The vertices of a SCC must either be all true or all false.

Algorithm Step 1: Run SCC on the graph. Check that each question-type-pair are in different SCC.

Now what? Every SCC gets the same value.

- Treat it as a single object!

We want to avoid edges from true things to false things.

- "Trues" seem more useful for us at the end.

Is there some way to start from the end?

YES! Topological Sort

Making the Final

Algorithm:
Make the requirements graph.

Find the SCCs.

If any SCC has including and not including a problem, we can't make the final.

Run topological sort on the graph of SCC.

Starting from the end:

- if everything in a component is unassigned, set them to true, and set their opposites to false.
- Else If one thing in a component is assigned, assign the same value to the rest of the nodes in the component and the opposite value to their opposites.

This works!!

How fast is it?

O(Q + S). That's a HUGE improvement.

Some More Context

The Final Making Problem was a type of "Satisfiability" problem.

We had a bunch of variables (include/exclude this question), and needed to satisfy everything in a list of requirements.

SAT is a general way to encode lots of hard problems.

Because every requirement was "do at least one of these 2" this was a 2-SAT instance.

If we change the 2 into a 3, no one knows an algorithm that runs efficiently.

And finding one (or proving one doesn't exist) has a $1,000,000 prize.

If we get to P vs. NP at the end of the quarter Kasey will tell you more.

Appendix: Strongly Connected Components Algorithm

Efficient SCC

We'd like to find all the vertices in our strongly connected component in time corresponding to the size of the component, not for the whole graph.

We can do that with a DFS (or BFS) as long as we don't leave our connected component.

If we're a "sink" component, that's guaranteed. I.e. a component whose vertex in the meta-graph has no outgoing edges.

How do we find a sink component? We don't have a meta-graph yet (we need to find the components first)

DFS can find a vertex in a source component, i.e. a component whose vertex in the meta-graph has no incoming edges.

- That vertex is the last one to be popped off the stack.

So if we run DFS in the *reversed* graph (where each edge points the opposite direction) we can find a sink component.

Efficient SCC

So from a DFS in the reversed graph, we can use the order vertices are popped off the stack to find a sink component (in the original graph).

Run a DFS from that vertex to find the vertices in that component *in size of that component time*.

Now we can delete the edges coming into that component.

The last remaining vertex popped off the stack is a sink of the remaining graph, and now a DFS from them won't leave the component.

Iterate this process (grab a sink, start DFS, delete edges entering the component).

In total we've run two DFSs. (since we never leave our component in the second DFS).

More information, and pseudocode:

https://en.wikipedia.org/wiki/Kosaraju%27s_algorithm

http://jeffe.cs.illinois.edu/teaching/algorithms/notes/19-dfs.pdf (mathier)