CSE 373: Data Structures and Algorithms

Shortest Paths

Autumn 2018

Shrirang (Shri) Mare
shri@cs.washington.edu

Thanks to Kasey Champion, Ben Jones, Adam Blank, Michael Lee, Evan McCarty, Robbie Weber, Whitaker Brand, Zora Fung, Stuart Reges, Justin Hsia, Ruth Anderson, and many others for sample slides and materials ...

Four classes of graph problem

.. that can be solved efficiently (in polynomial time)

1. Shortest path – find a shortest path between two vertices in a graph
2. Minimum spanning tree – find subset of edges with minimum total weights
3. Matching – find set of edges without common vertices
4. Maximum flow – find the maximum flow from a source vertex to a sink vertex

A wide array of graph problems that can be solved in polynomial time are variants of these above problems.

In this class, we'll cover the first two problems – shortest path and minimum spanning tree

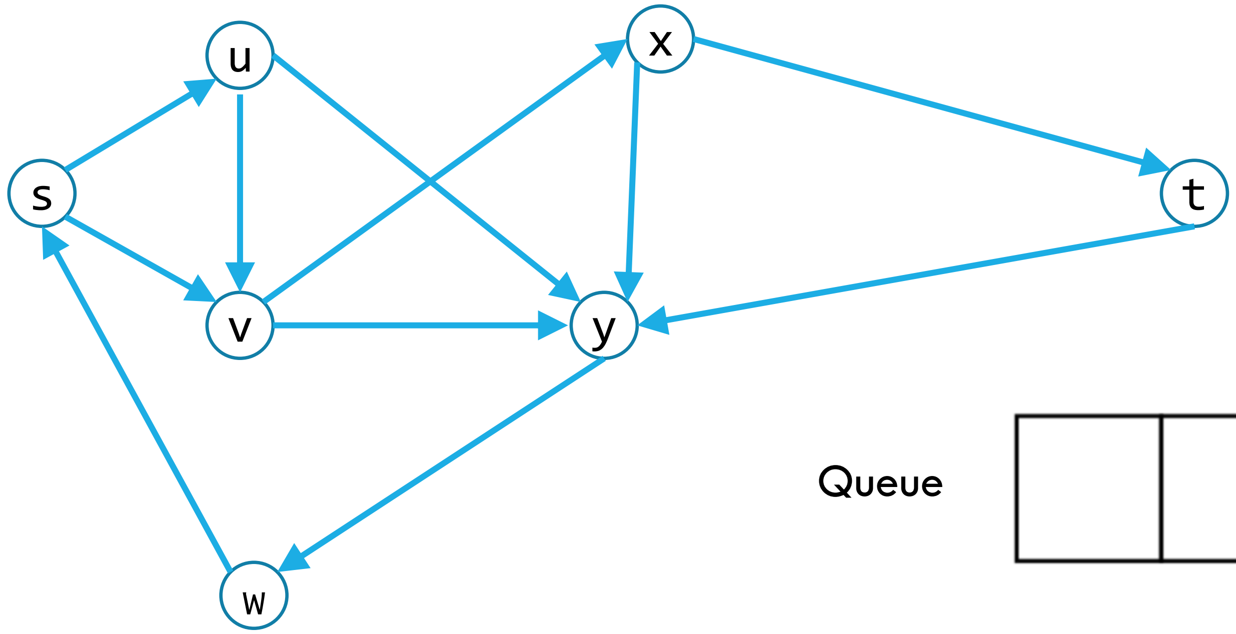
BFS on a directed graph

Run Breadth First Search on this graph starting from s.

What order are vertices placed on the queue?

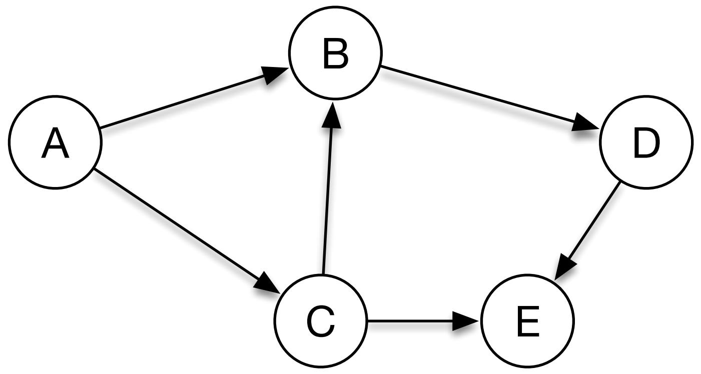
When processing a vertex insert neighbors in alphabetical order.

```
function search(Graph G)
    mark all vertices as unknown
    toVisit.enqueue(first vertex)
    mark first vertex as known
    while toVisit is not empty do
        current = toVisit.dequeue()
        for v : current.outNeighbors() do
            if v is unknown then
                toVisit.enqueue(v)
                mark v as known
            end if
        end for
        visited.add(current)
    end while
end function
```



Queue

Q1: Do a BFS no this graph, starting at node A



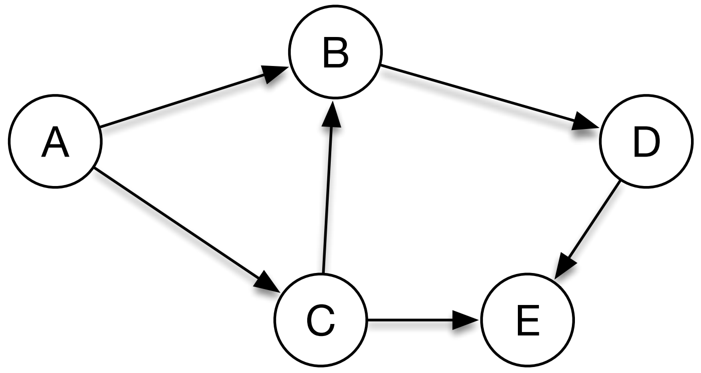
Add nodes to queue in alphabetical order

Queue

Add nodes to queue in reverse alphabetical order

Queue

Q2: Do a DFS no this graph, starting at node A



Add nodes to stack in alphabetical order

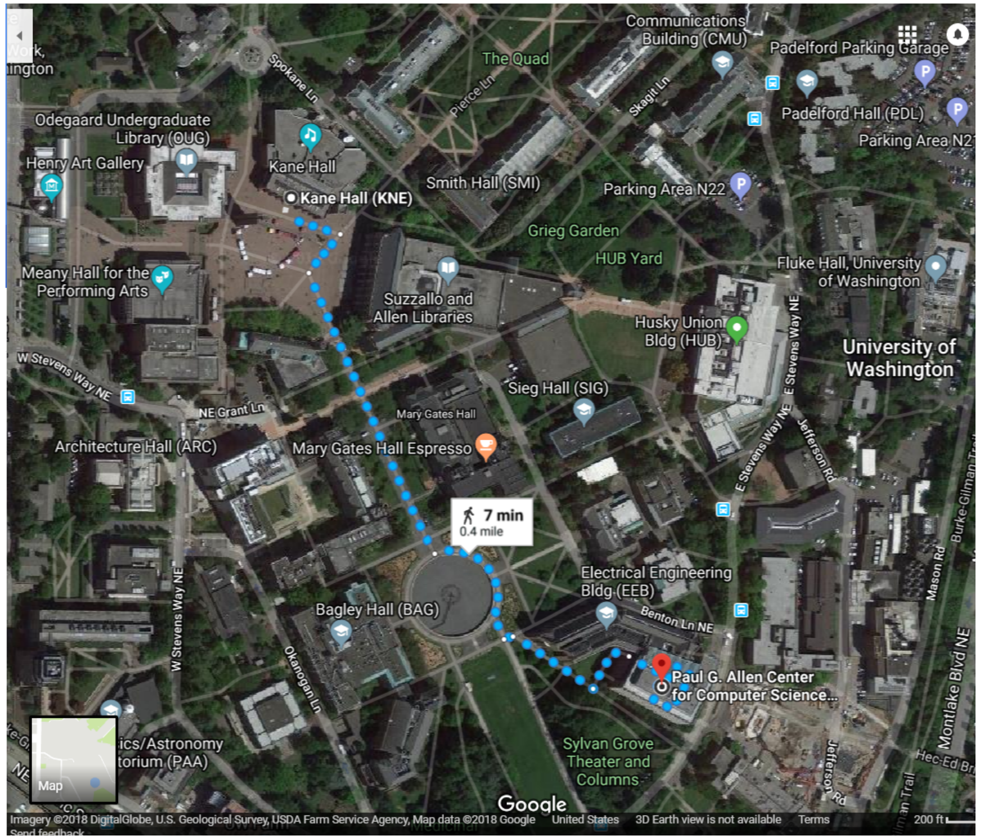
Stack					

Add nodes to stack in reverse alphabetical order

Stack					

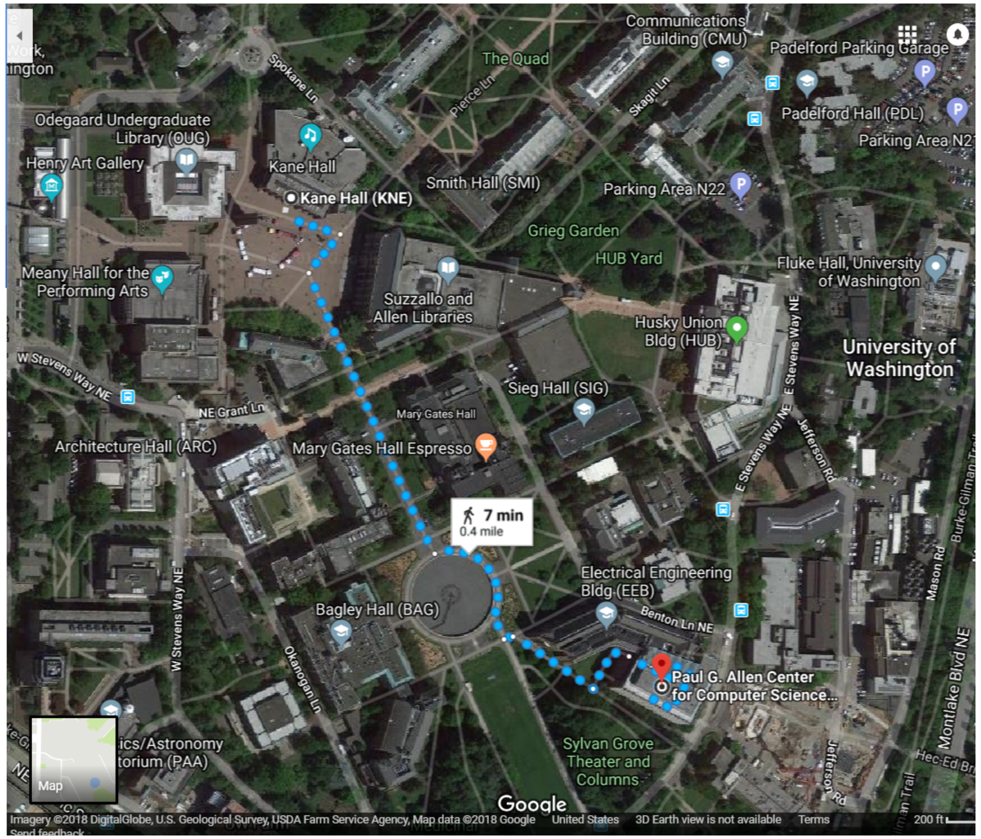
Shortest paths

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

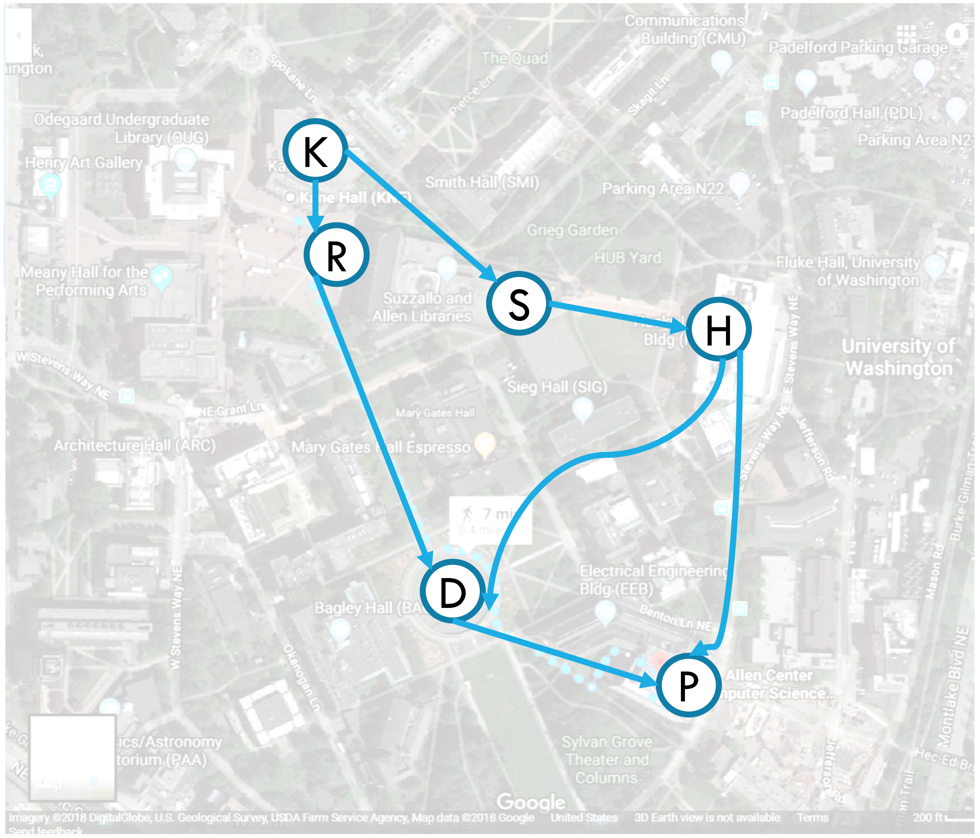

Kane Hall (KNE)
Paul G. Allen Center for Computer Science...
7 min
0.4 mile
Odegaard Undergraduate Library (OUG)
Suzzallo and Allen Libraries
Mary Gates Hall Espresso
Bagley Hall (BAG)
Electrical Engineering Bldg (EEB)
Husky Union Bldg (HUB)
Sieg Hall (SIG)
University of Washington
Google

Representing maps as graphs

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

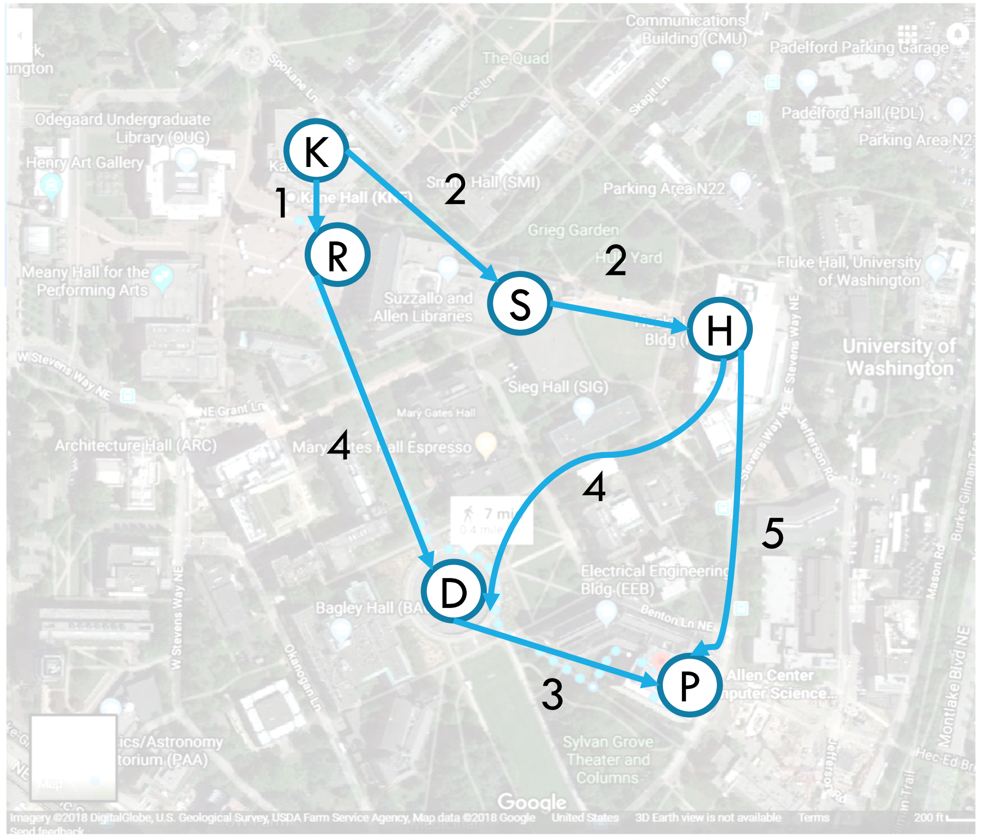


Representing maps as graphs



Question:
Does this graph correctly model all the shown paths?

Representing maps as graphs



Question:
Does this graph correctly model all the shown paths?

Not quite.
Some paths are longer than others.
So we use "weights" to capture that additional information.

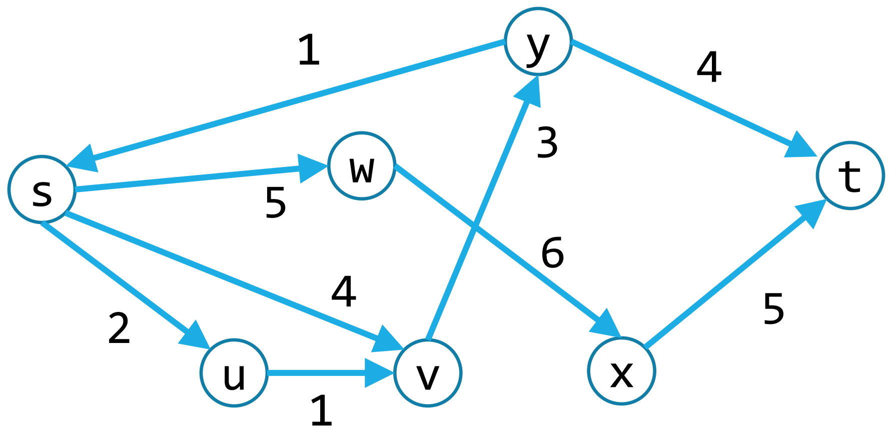
Shortest paths

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

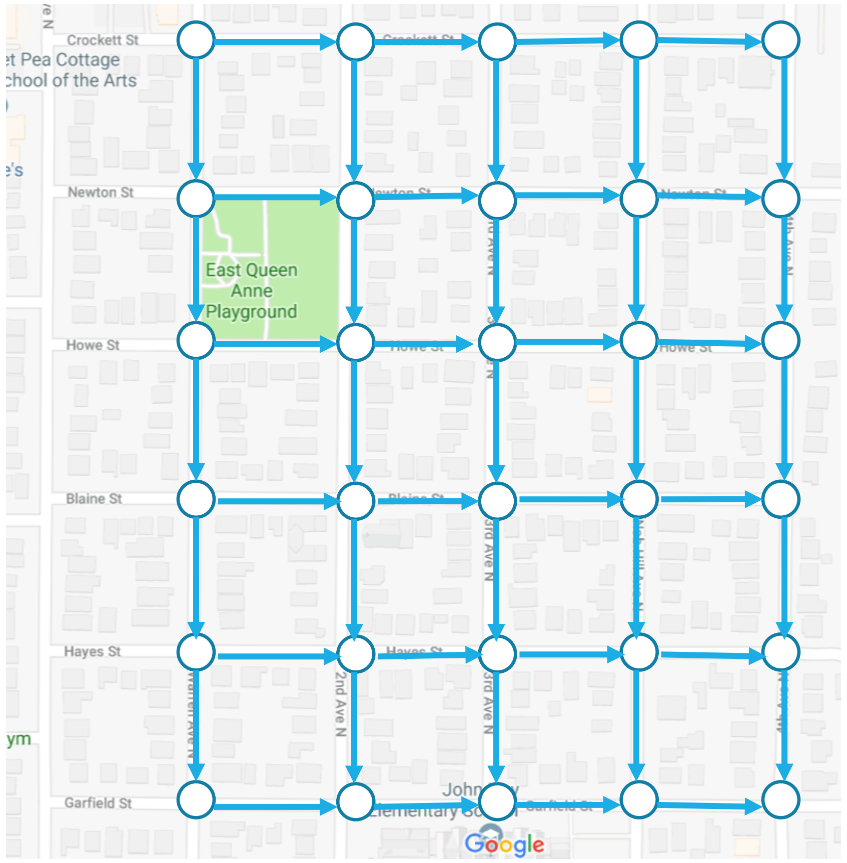
Find: the shortest path from s to t



Unweighted graphs

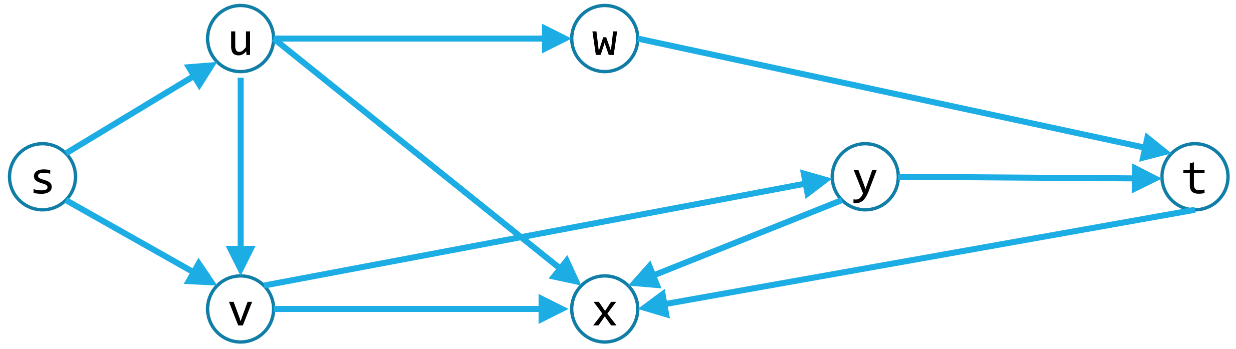
Let’s start with a simpler version: the edges are all the same weight (**unweighted**)

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 s to 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?

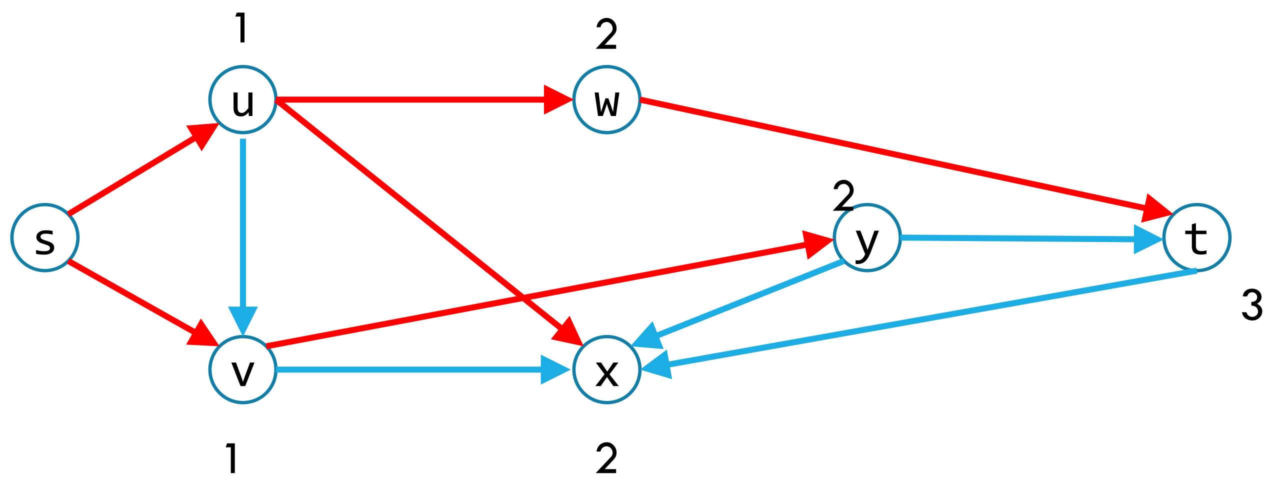
Yes! BFS!

```
function bfsShortestPaths(Graph G, Vertex source)
    toVisit.enqueue(source)
    mark source as known
    source.dist = 0
    while toVisit is not empty do
        current = toVisit.dequeue()
        for v : current.outNeighbors() do
            if v is unknown then
                v.distance = current.distance + 1
                v.predecessor = current
                toVisit.enqueue(v)
                mark v as known
            end if
        end for
    end while
end function
```

Unweighted graphs

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

```
function bfsShortestPaths(Graph G, Vertex source)
    toVisit.enqueue(source)
    mark source as known
    source.dist = 0
    while toVisit is not empty do
        current = toVisit.dequeue()
        for v : current.outNeighbors() do
            if v is unknown then
                v.distance = current.distance + 1
                v.predecessor = current
                toVisit.enqueue(v)
                mark v as known
            end if
        end for
    end while
end function
```



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.

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.

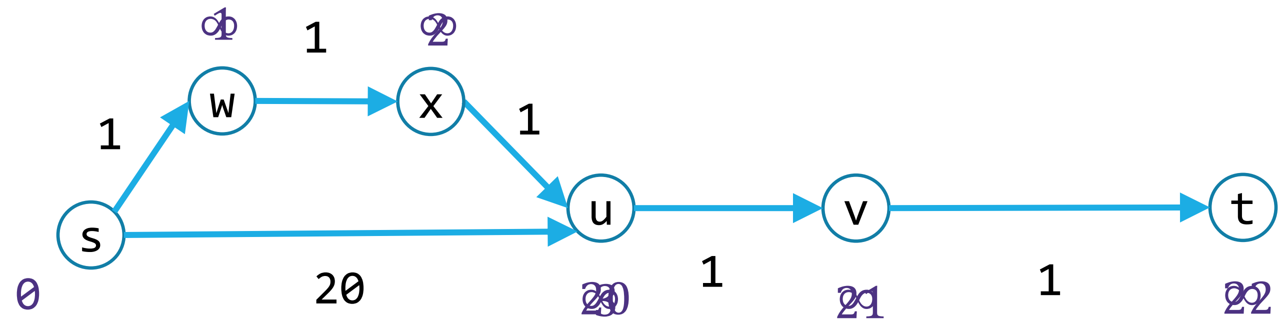
The length of a path in a weighted graph is the sum of the weights along that path.

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.

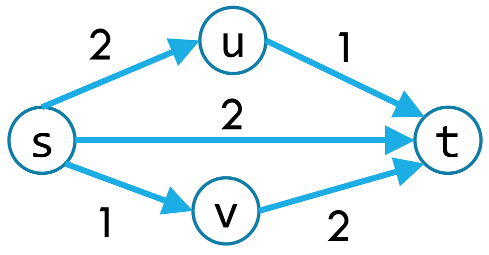
In project 2, you reduced implementing a hashset to implementing a hashmap.

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?



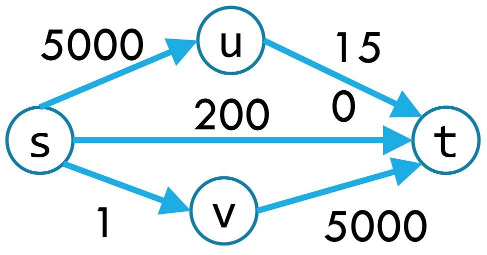
Transform Input

Unweighted Shortest Paths

Transform Output

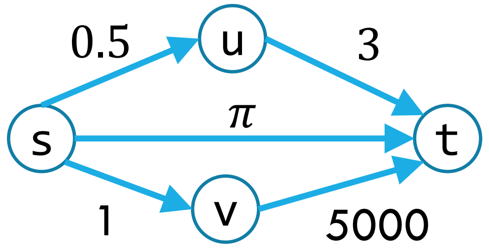
Weighted graphs: A Reduction

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….

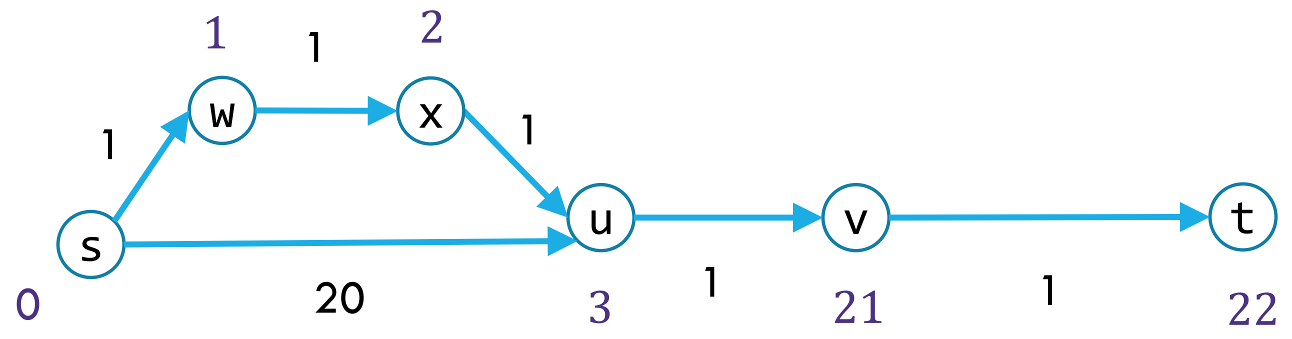
Tl;dr: If your graph's weights are all small positive integers, this reduction might work great.
Otherwise we probably need a new idea.

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).

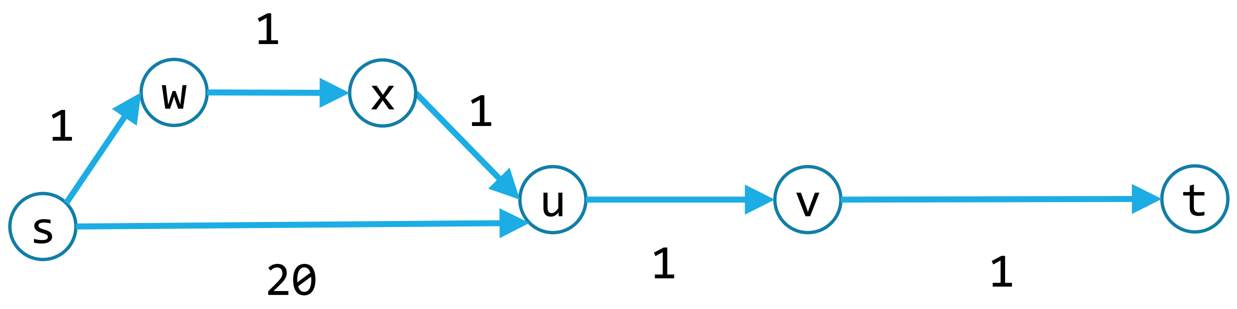
Record an estimated distance
- (the best way we know to get to each vertex).

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

```
1:  function Dijkstra(Graph G, Vertex source)
2:      initialize distances to $\infty$
3:      mark source as distance 0
4:      mark all vertices unprocessed
5:      while there are unprocessed vertices do
6:          let u be the closest unprocessed vertex
7:          for each edge (u, v) leaving u do
8:              if $u.dist + w(u,v) < v.dist$ then
9:                  $v.dist = u.dist + w(u,v)$
10:                 $v.predecessor = u$
11:             end if
12:         end for
13:         mark u as processed
14:     end while
15: end function
```

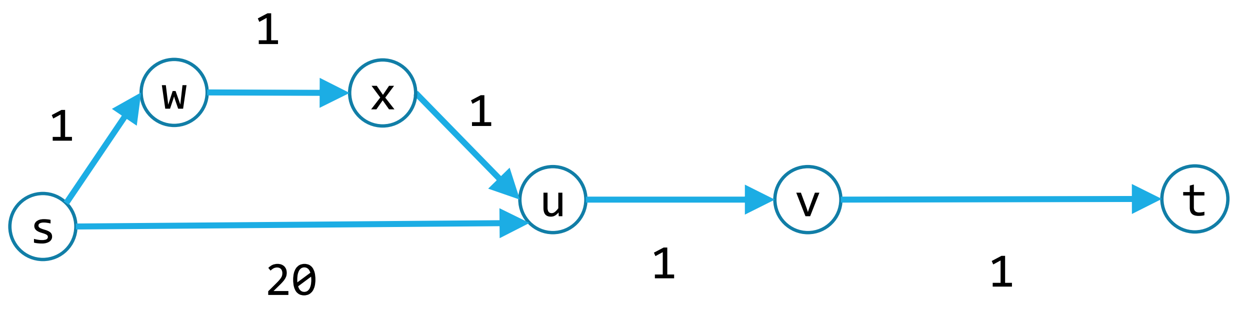
Vertex	Distance	Predecessor	Processed
s			
w			
x			
u			
v			
t			



Dijkstra’s algorithm

```
function Dijkstra(Graph G, Vertex source)
    initialize distances to $\infty$
    mark source as distance 0
    mark all vertices unprocessed
    while there are unprocessed vertices do
        let u be the closest unprocessed vertex
        for each edge (u, v) leaving u do
            if u.dist + w(u,v) < v.dist then
                v.dist = u.dist + w(u,v)
                v.predecessor = u
            end if
        end for
        mark u as processed
    end while
end function
```

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


$$\begin{tikzcd} & w \arrow[r, "1"] & x \arrow[dr, "1"] & & & \\ s \arrow[ur, "1"] \arrow[rrr, "20"'] & & & u \arrow[r, "1"'] & v \arrow[r, "1"'] & t \end{tikzcd}$$

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 (MPQ) work

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

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

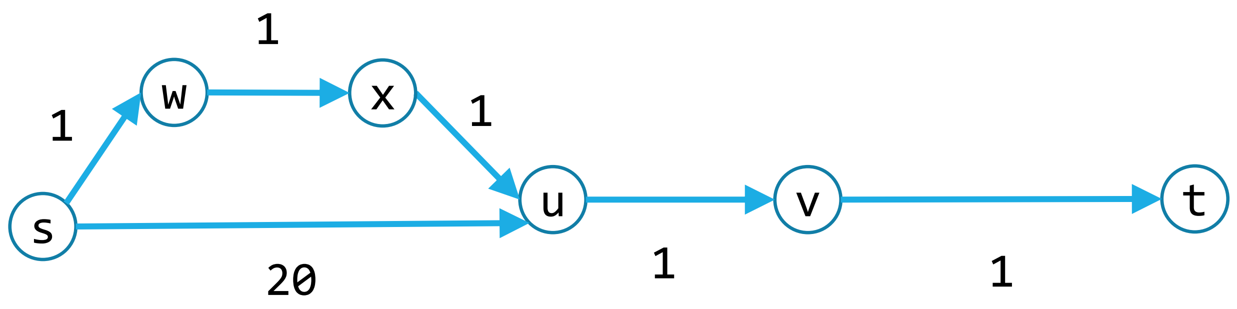
peekMin() – find, but do not remove the element with the smallest priority

insert(value) – add a new element to the collection

Dijkstra's algorithm

```
1:  function Dijkstra(Graph G, Vertex source)
2:      initialize distances to $\infty$
3:      mark source as distance 0
4:      mark all vertices unprocessed
5:      while there are unprocessed vertices do
6:          let u be the closest unprocessed vertex
7:          for each edge (u, v) leaving u do
8:              if $u.dist + w(u,v) < v.dist$ then
9:                  $v.dist = u.dist + w(u,v)$
10:                 $v.predecessor = u$
11:             end if
12:         end for
13:         mark u as processed
14:     end while
15: end function
```

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



Making minimum priority queues (MPQ) 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 <u>smallest</u> priority, removes it from the collection

peekMin() – find, but do not remove the element with the smallest <u>priority</u>

insert(value) – add a new element to the collection

DecreasePriority(e, p) – decreases the priority of element e down to p.

Running time analysis

```
function Dijkstra(Graph G, Vertex source)                    ▷ with MPQ
    initialize distances to $\infty$, source.dist = 0
    mark all vertices unprocessed
    initialize MPQ as a min priority queue; add source with priority 0
    while MPQ is not empty do
        u = MPQ.getMin()
        for each edge (u,v) leaving u do
            if u.dist + w(u,v) < v.dist then
                if v.dist == $\infty$ then
                    MPQ.insert(v, u.dist + w(u, v))
                else
                    MPQ.decreasePriority(v, u.dist + w(u,v))
                end if
                v.dist = u.dist + w(u,v)
                v.predecessor = u
            end if
        end for
        mark u as processed
    end while
end function
```

History of shortest path algorithms

Algorithm/Author	Time complexity	Year
Ford	$O(\|V\|^2\|E\|)$	1956
Bellman-Ford. Shimbel	$O(\|V\|\|E\|)$	1958
Dijkstra's algorithm with binary heap	$\mathbf{O(\|E\| \log\|V\| + \|V\| \log \|V\|)}$	**1959**
Dijkstra's algorithm with Fibonacci heap	$O(\|E\| + \|V\| \log \|V\|)$	1984
Gabow's algorithm	$O(\|E\| + \|V\|\sqrt{\log \|V\|})$	1990
Thorup	$O(\|E\| + \|V\| \log\log \|V\|)$	2004

History of shortest path algorithms. In this class, from this table, you are expected to know only Dijkstra's algorithm with binary heap and its time complexity. You are not expected to know the other algorithms or their time complexities.

Other applications 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 this problem

I don’t care if you remember how we apply shortest paths.

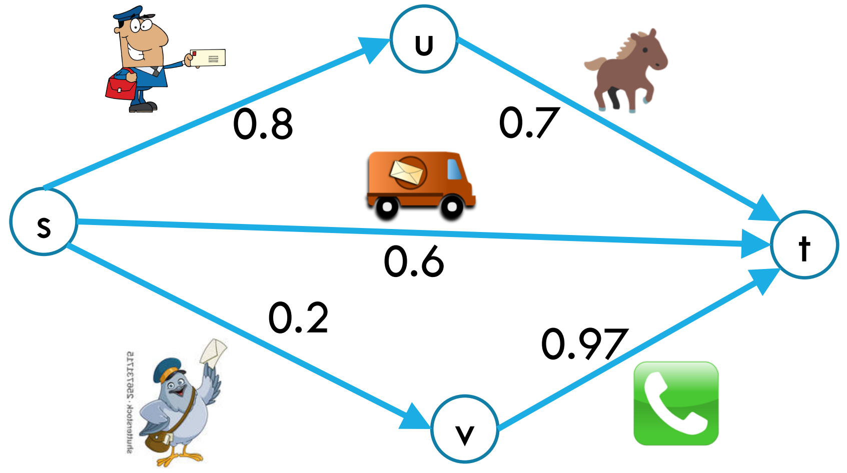
I just want you to see that these algorithms have non-obvious applications.

Other applications: Maximum probability path

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

Other applications of shortest paths

Robot navigation

Urban traffic planning

Tramp steamer problem

Optimal pipelining of VLSI chips

Operator scheduling

Subroutine in higher level algorithms

Exploiting arbitrage opportunities in currency exchanges

Open shortest path first routing protocol for IP

Optimal truck routing through given traffic congestion