Lecture 17: Shortest Paths

CSE 373: Data Structures and Algorithms

Administrivia

Project 4 partner form due tonight.

Project 3 due Wednesday

Exercise 4 is out.

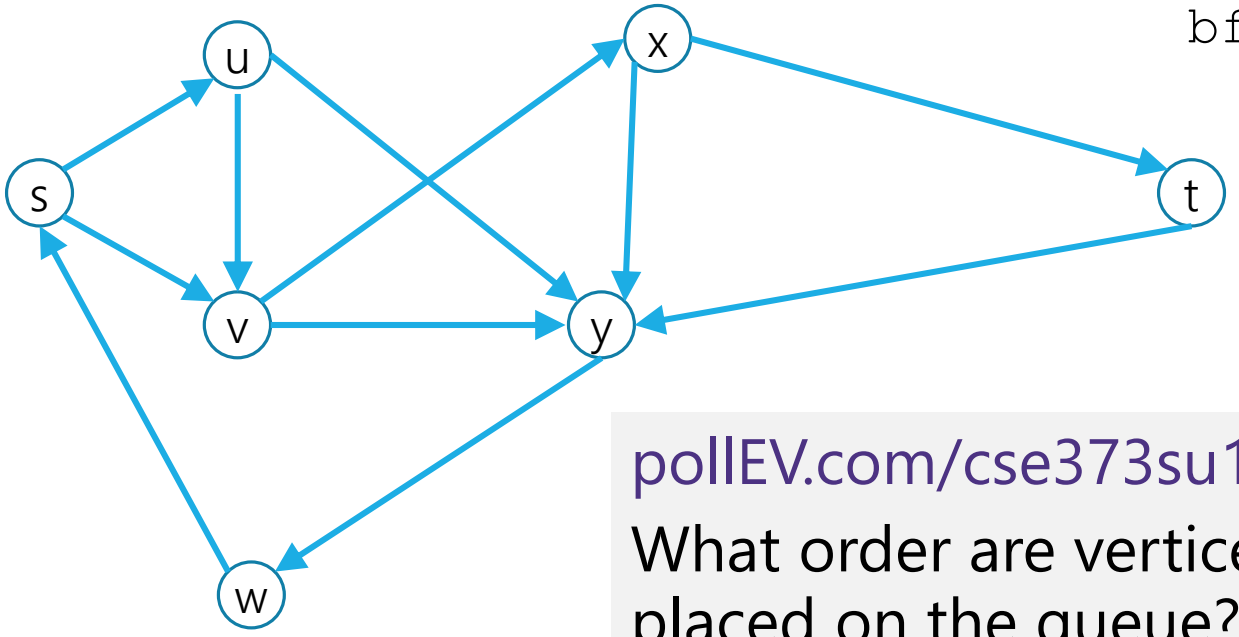
Warm Up

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

pollEV.com/cse373su19

What order are vertices placed on the queue?

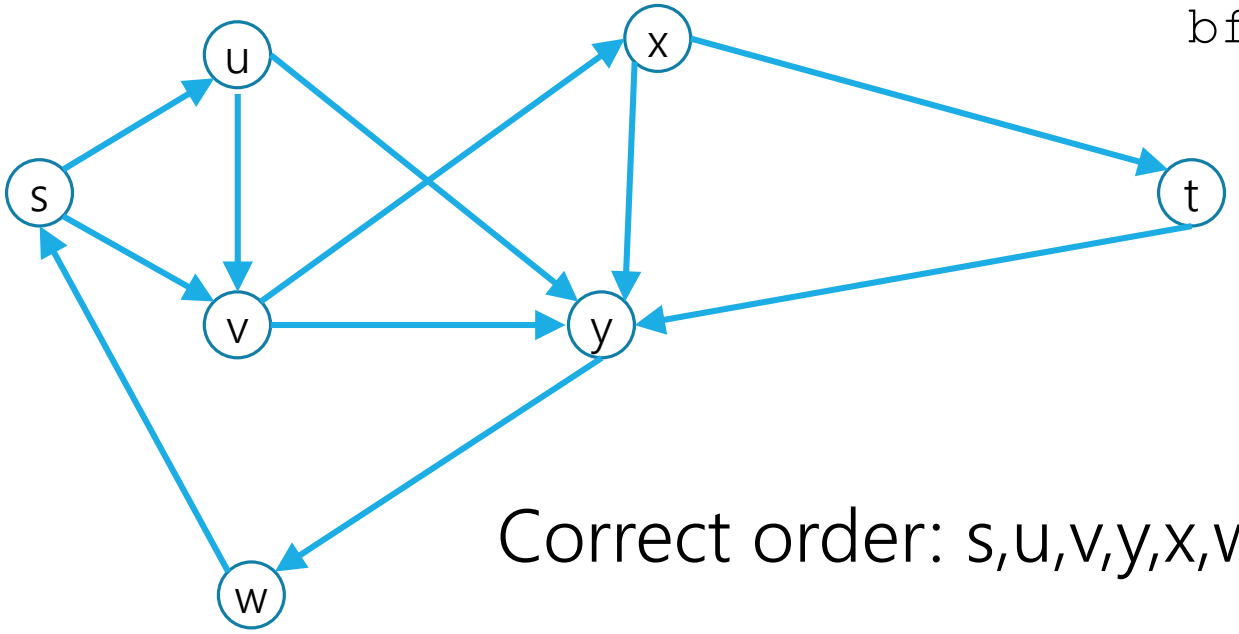
Warm Up

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.

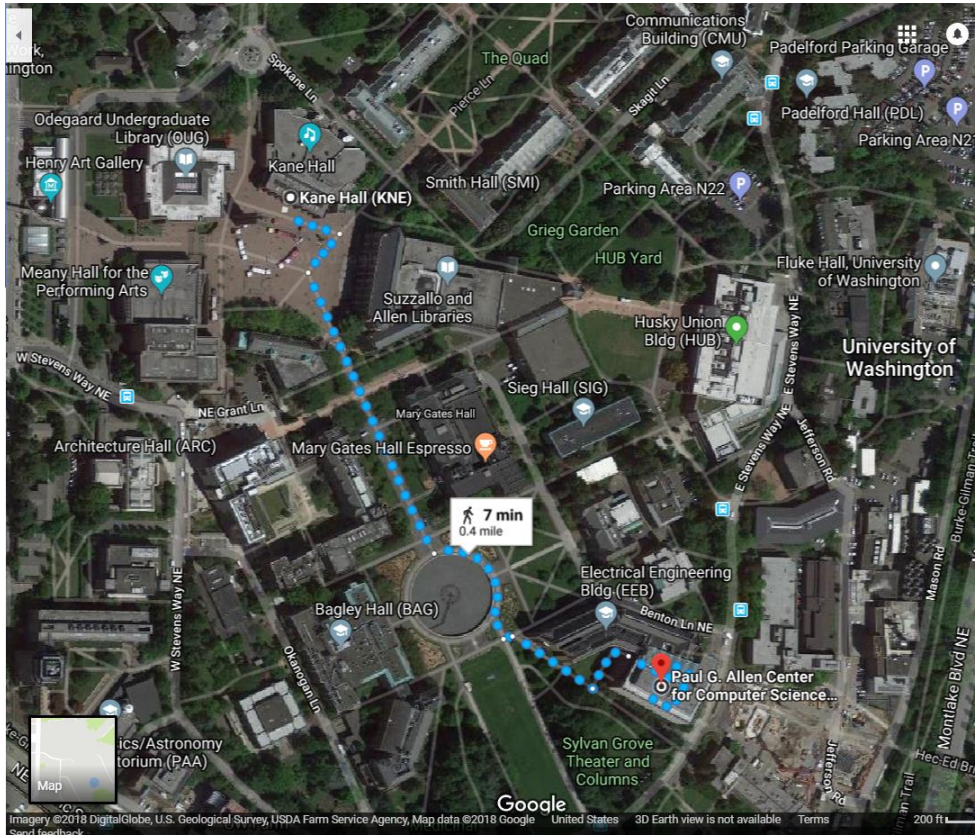


Correct order: s,u,v,y,x,w,t

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

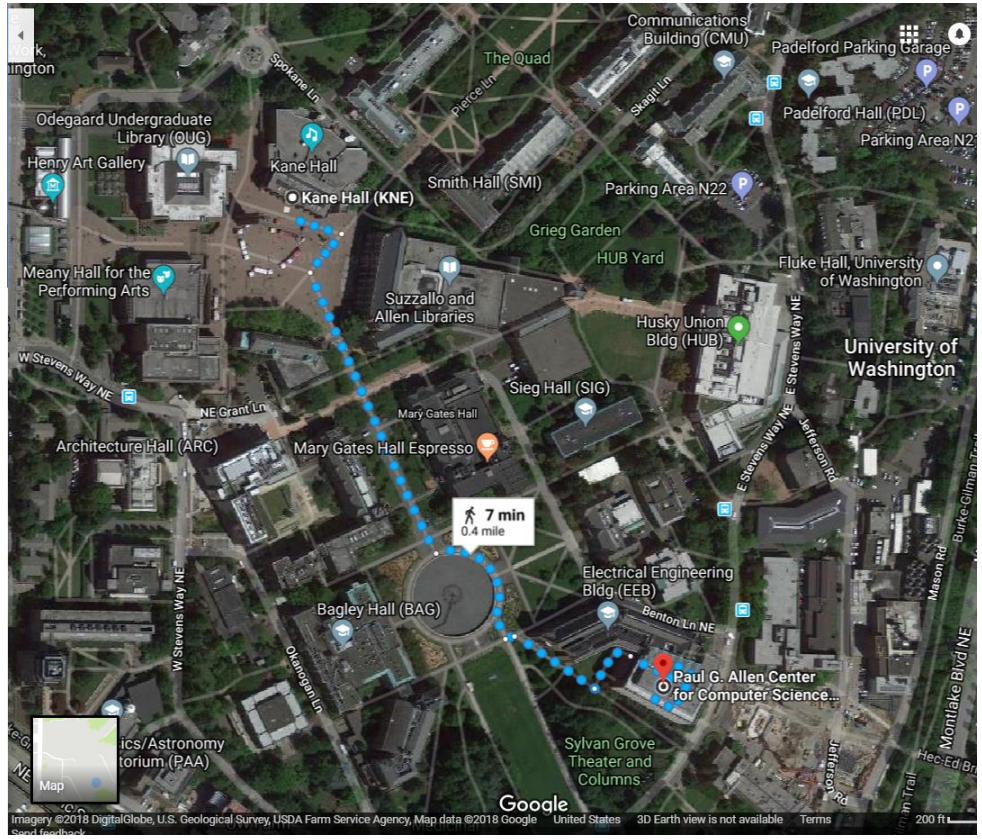
Shortest Paths

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

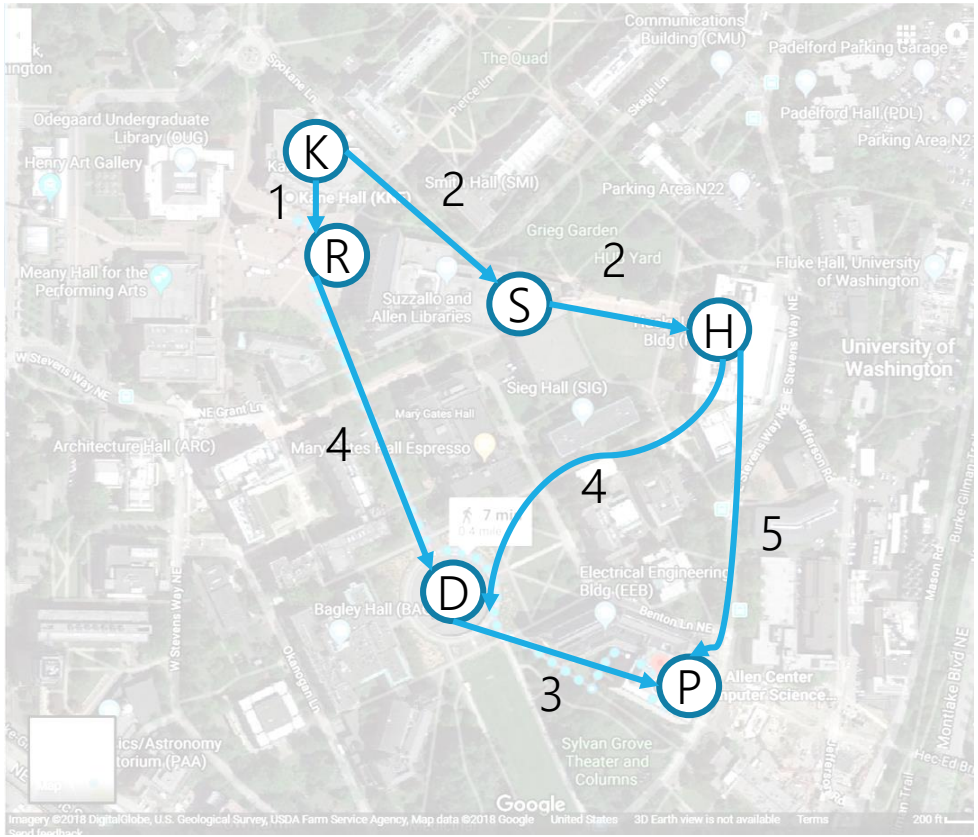


Representing Maps as Graphs

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



Representing Maps as Graphs



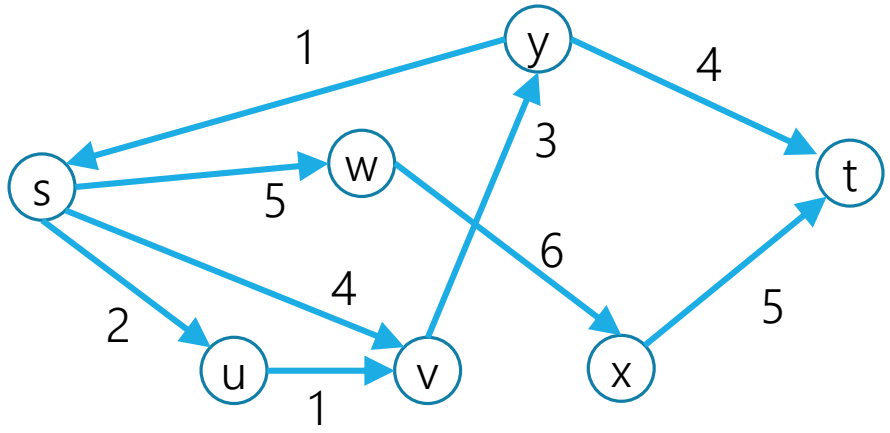
Shortest Paths

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

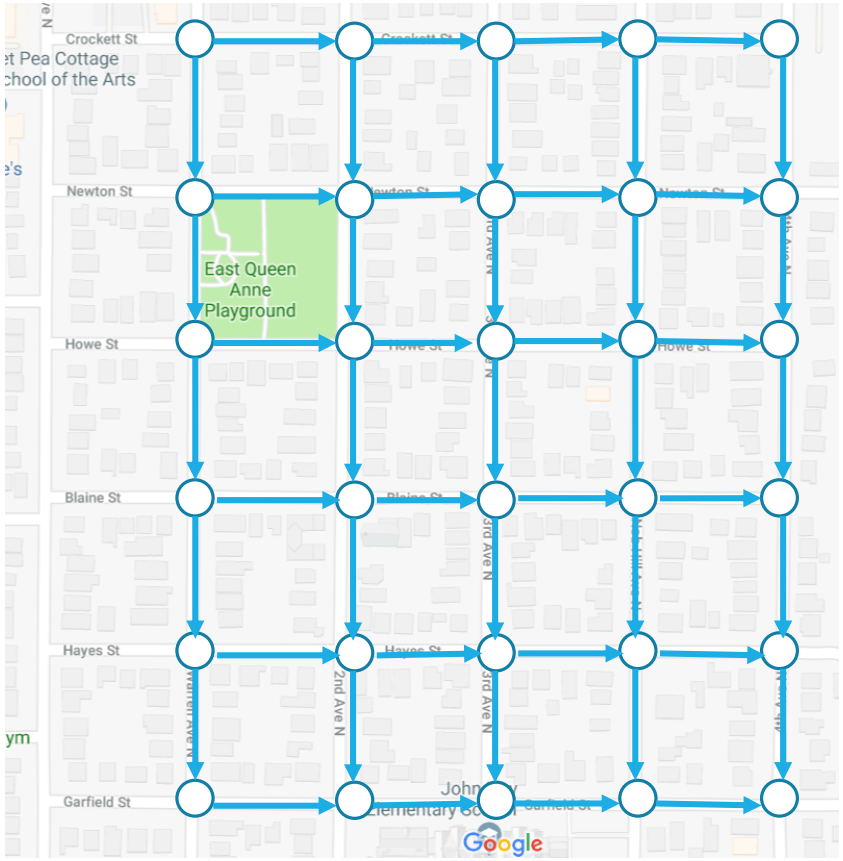
Find: the shortest path from s to t



Unweighted graphs

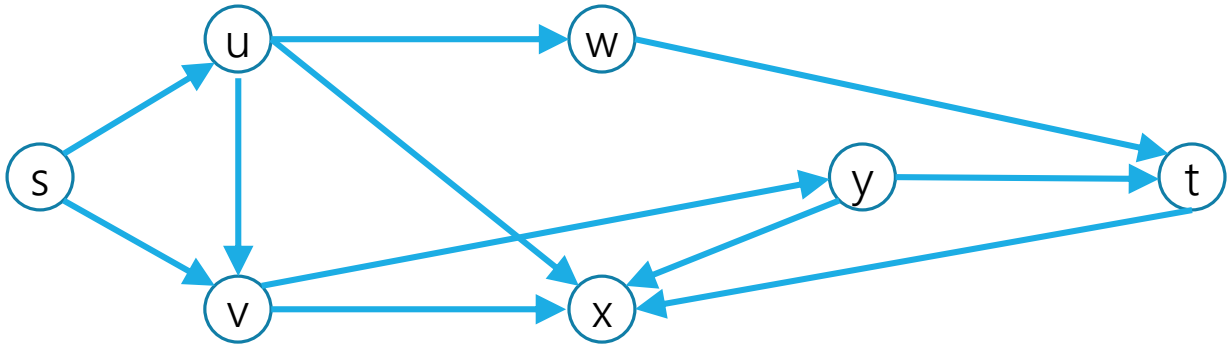
Let's start with a simpler version: the edges are all the same weight (**unweighted**)

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?

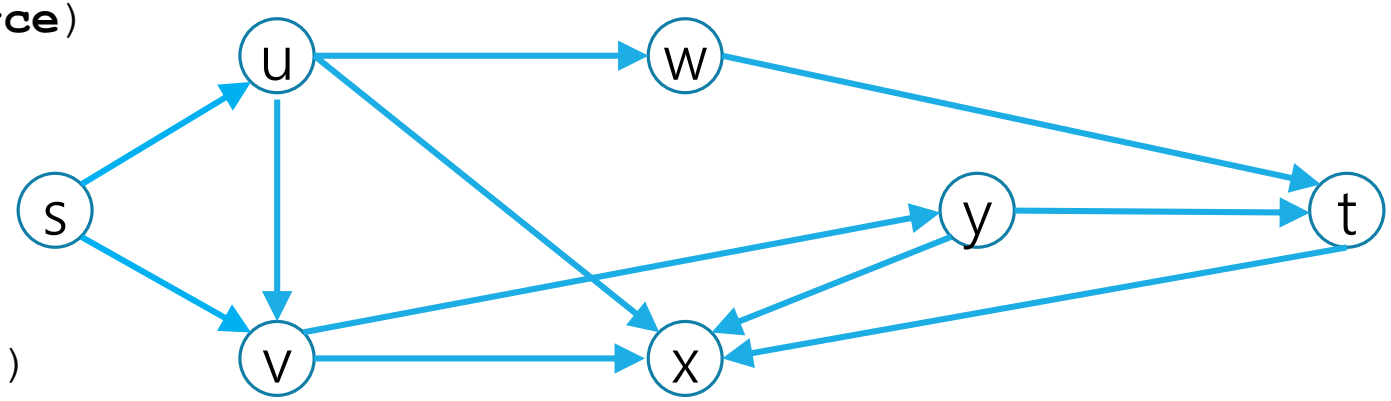
Yes! BFS!

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

Unweighted Graphs

Use BFS to find shortest paths in this graph.

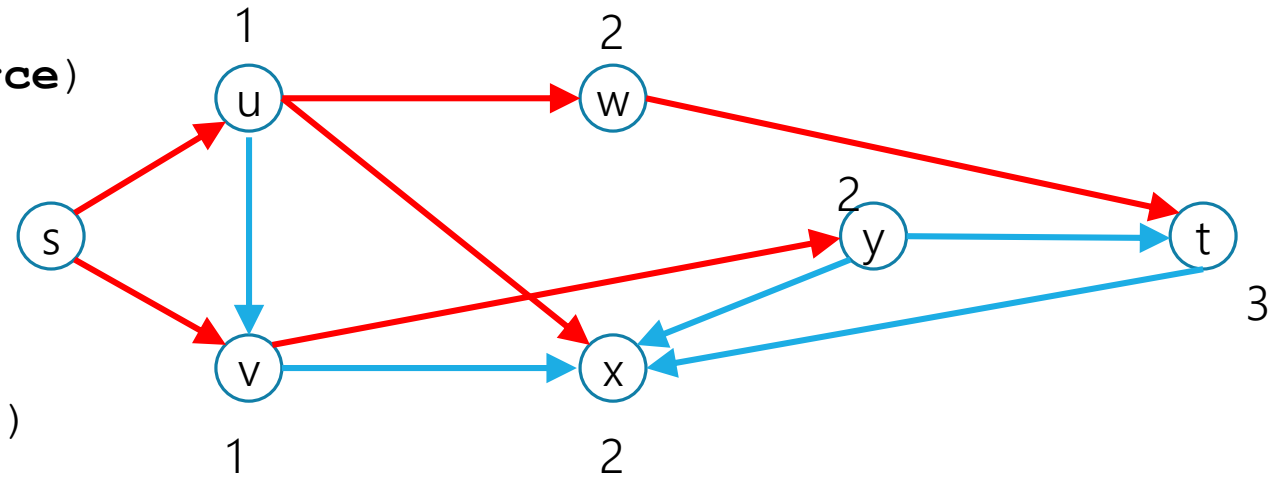
```
bfsShortestPaths(graph G, vertex source)
    toVisit.enqueue(source)
    mark source as seen
    source.dist = 0
    while(toVisit is not empty){
        current = toVisit.dequeue()
        for (v : current.outNeighbors())
         {
            if (v is not seen){
                v.distance = current.distance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



Unweighted Graphs

Use BFS to find shortest paths in this graph.

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



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 distance from s to **every** other vertex.

All our shortest path algorithms have this property.
If you only care about one target, you can sometimes stop early (in `bfsShortestPaths`, when the target pops off 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.

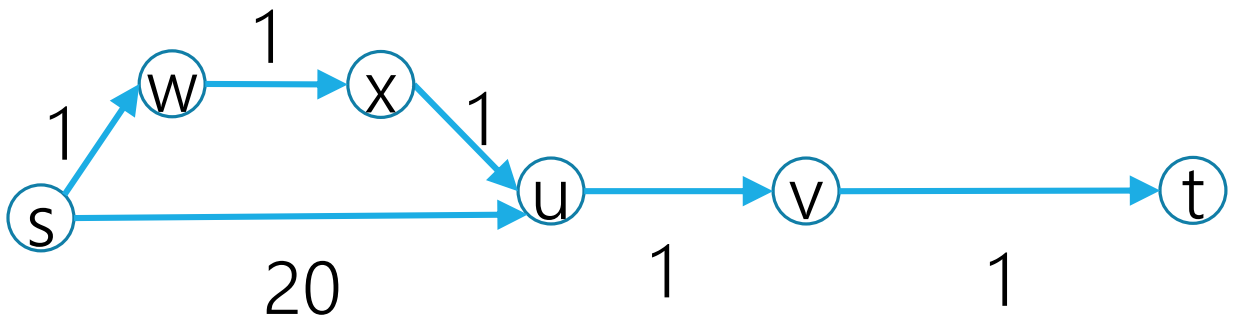
We'll assume all of the weights are positive

- 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

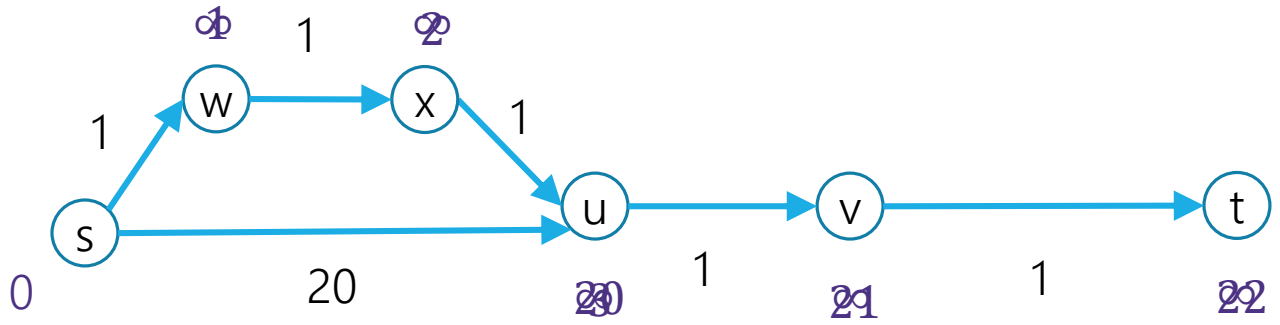
BFS works if the graph is unweighted.

Maybe it just works for weighted graphs too?



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.

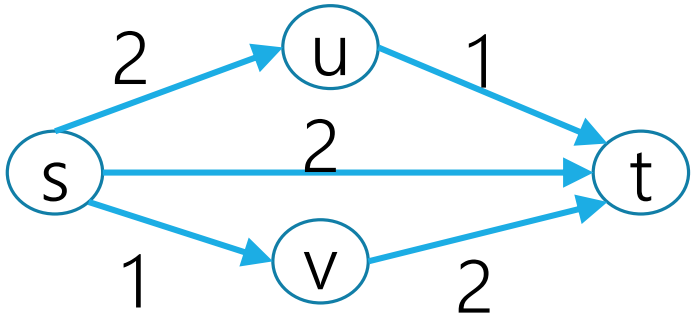
Any time you use a library, you're reducing your problem to the one the library solves.

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

Weighted Graphs: Take 2

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

Weighted Graphs: A Reduction



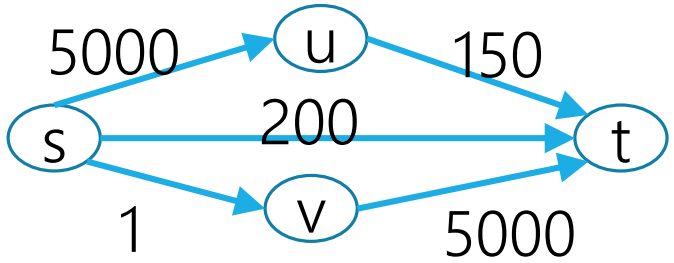
Transform Input

Unweighted Shortest Paths

Transform Output

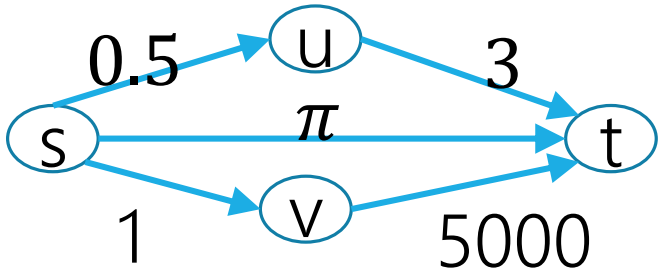
Weighted Graphs: A Reduction

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….

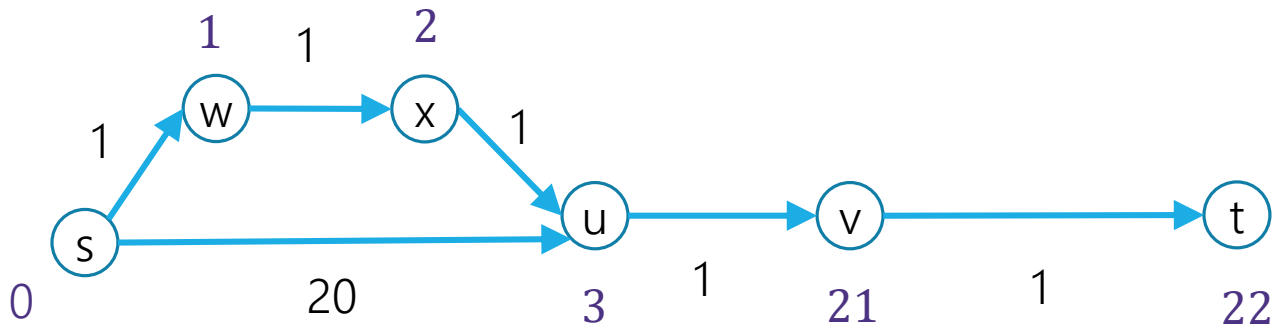
tl;dr: If your graph's weights are all small positive integers, this reduction might work great.
Otherwise we probably need a new idea.

Weighted Graphs: Take 3

So we can't just do a reduction.

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

How did we avoid this problem:



Weighted Graphs: Take 3

In BFS 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

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).

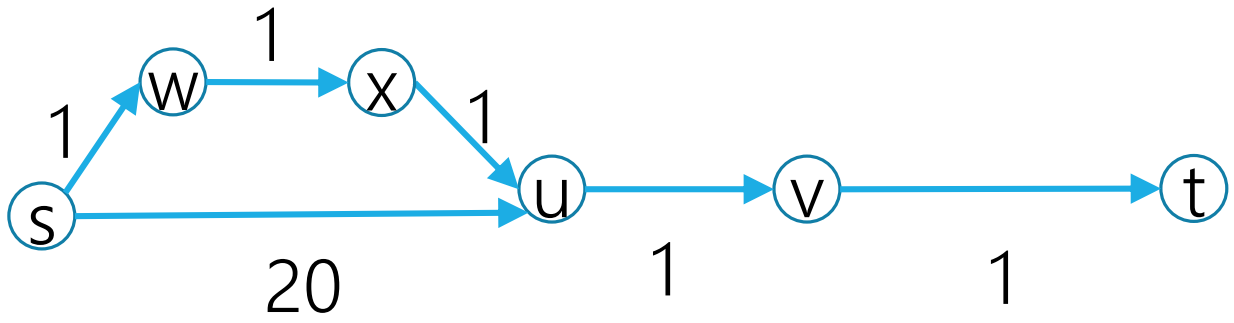
Record an estimated distance
- (the best way we know to get to each vertex).

If we process only the vertex closest in estimated distance, we won't ever find a shorter path to a processed vertex.
- This statement is the key to proving correctness.
- It's nice if you want to practice induction/understand the algorithm better.

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+weight(u,v) < v.dist){
                v.dist = u.dist+weight(u,v)
                v.predecessor = u
            }
        }
       mark u as processed
    }
```

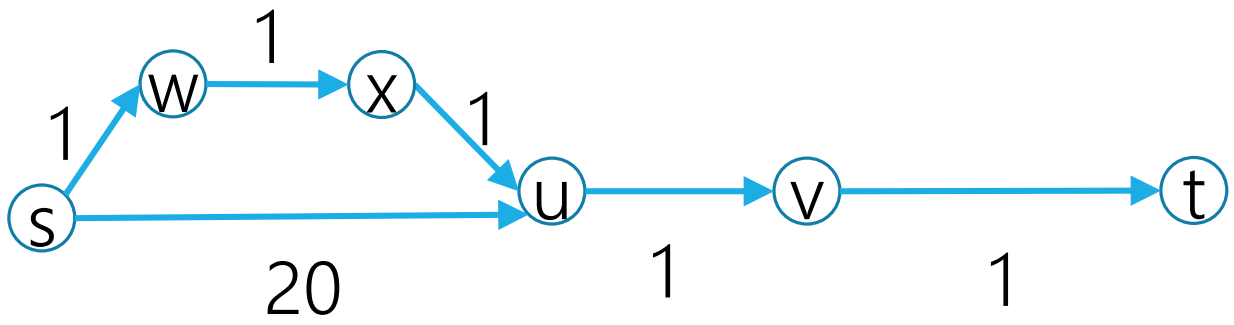
Vertex	Distance	Predecessor	Processed
s			
w			
x			
u			
v			
t			



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+weight(u,v) < v.dist){
                v.dist = u.dist+weight(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	~~20~~ 3	~~s~~ x	Yes
v	4	u	Yes
t	5	v	Yes

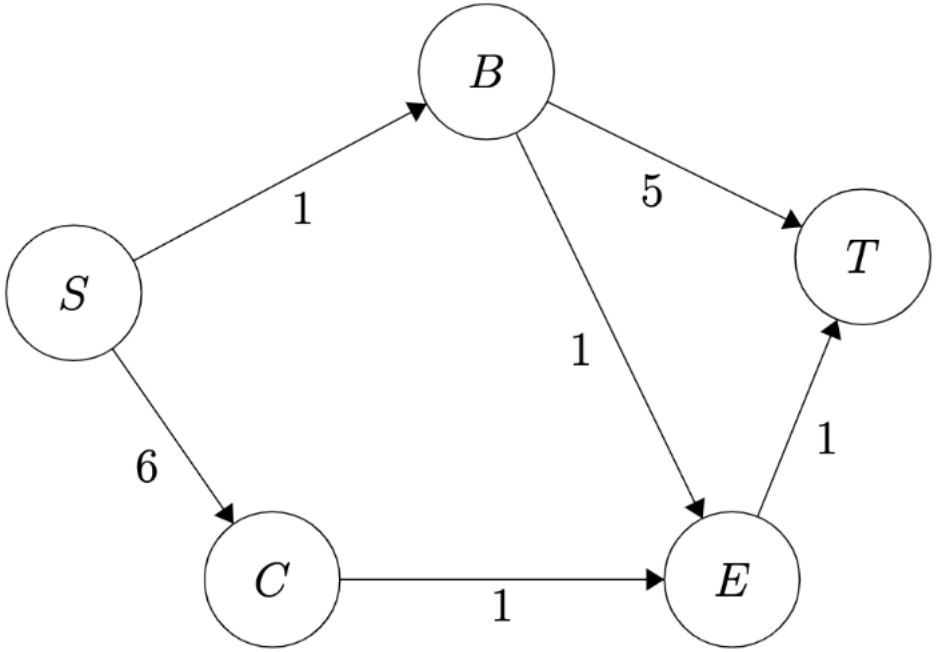


Dijkstra's Run Through

Vertex	Distance	Predecessor	Processed
S			
C			
B			
T			
E			

Pseudocode

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

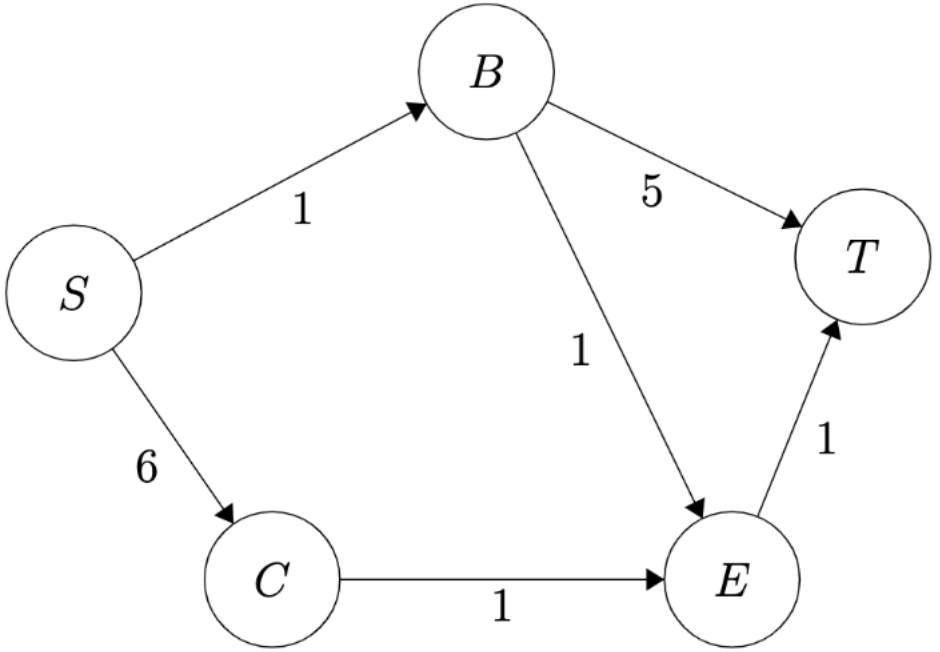


Dijkstra's Run Through

Pseudocode

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

Vertex	Distance	Predecessor	Processed
S	0		No
C	∞		No
B	∞		No
T	∞		No
E	∞		No

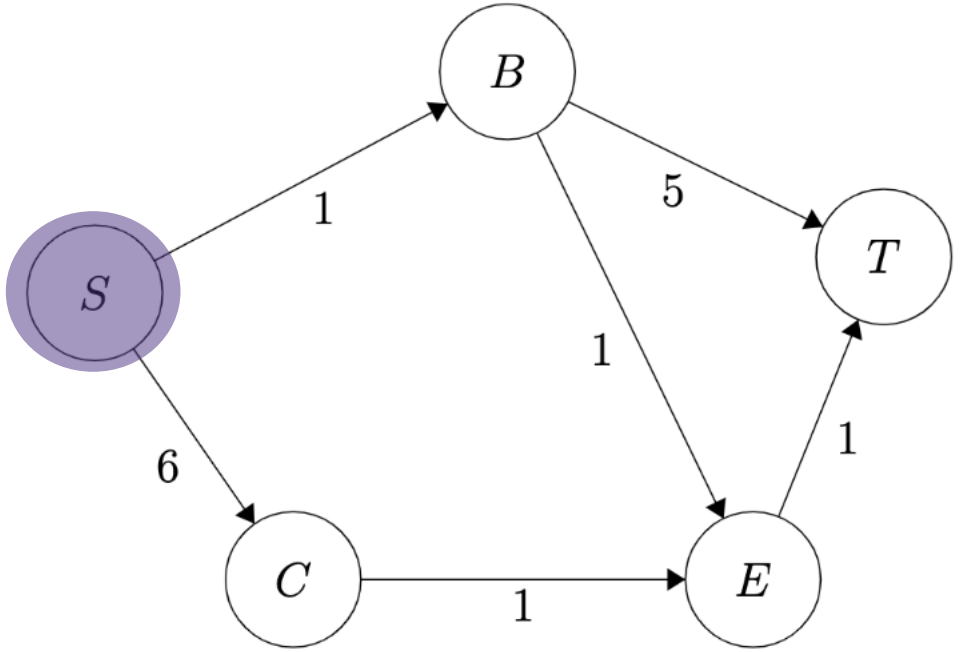


Dijkstra's Run Through

Pseudocode

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

Vertex	Distance	Predecessor	Processed
S	0	--	No
C	6	S	No
B	1	S	No
T	∞		No
E	∞		No

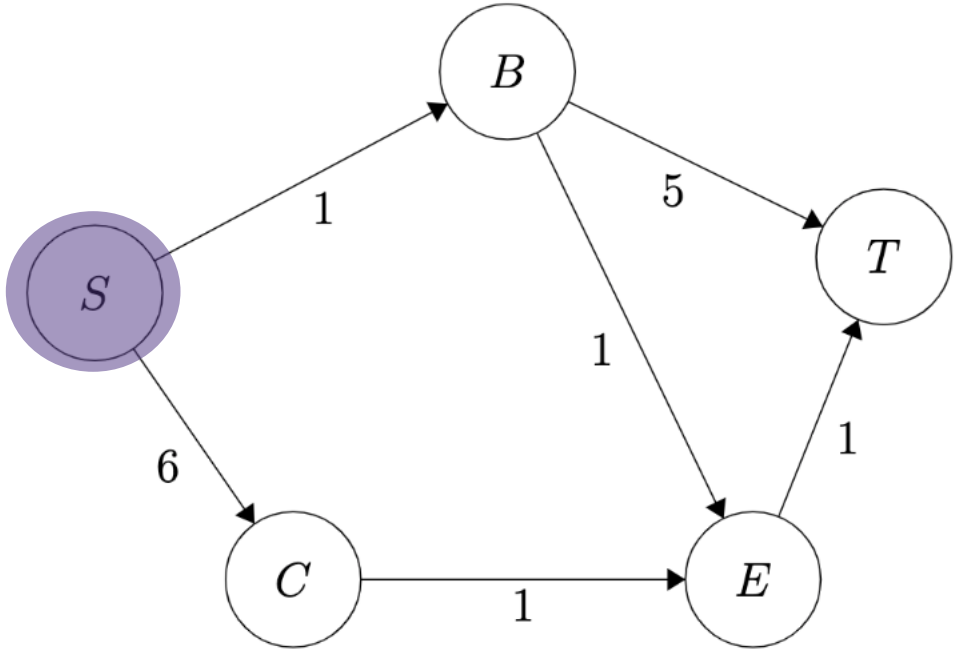


Dijkstra's Run Through

Pseudocode

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

Vertex	Distance	Predecessor	Processed
S	0	--	Yes
C	6	S	No
B	1	S	No
T	∞		No
E	∞		No

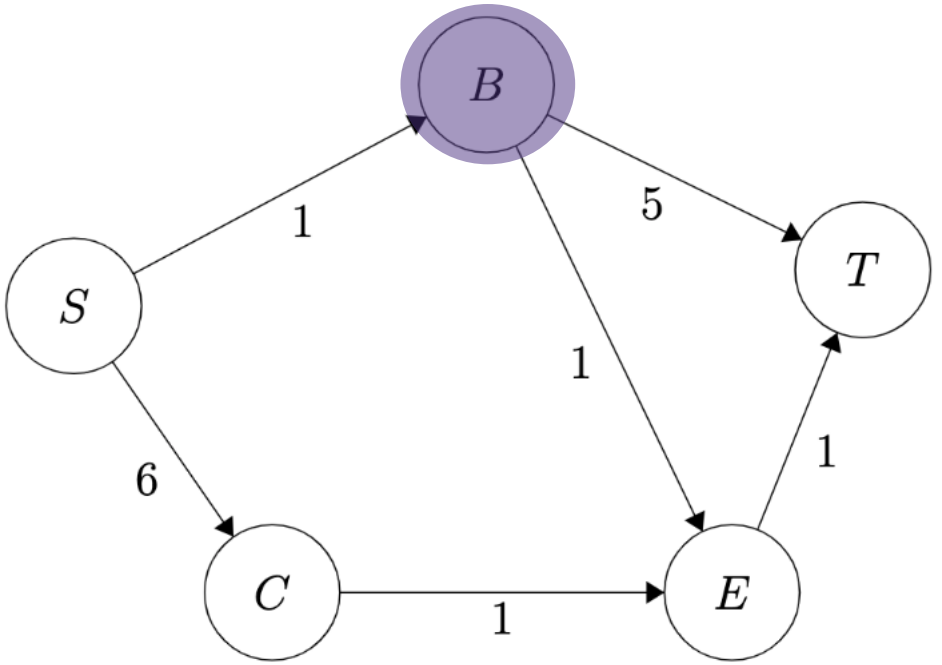


Dijkstra's Run Through

Pseudocode

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

Vertex	Distance	Predecessor	Processed
S	0	--	Yes
C	6	S	No
B	1	S	Yes
T	6	B	No
E	2	B	No

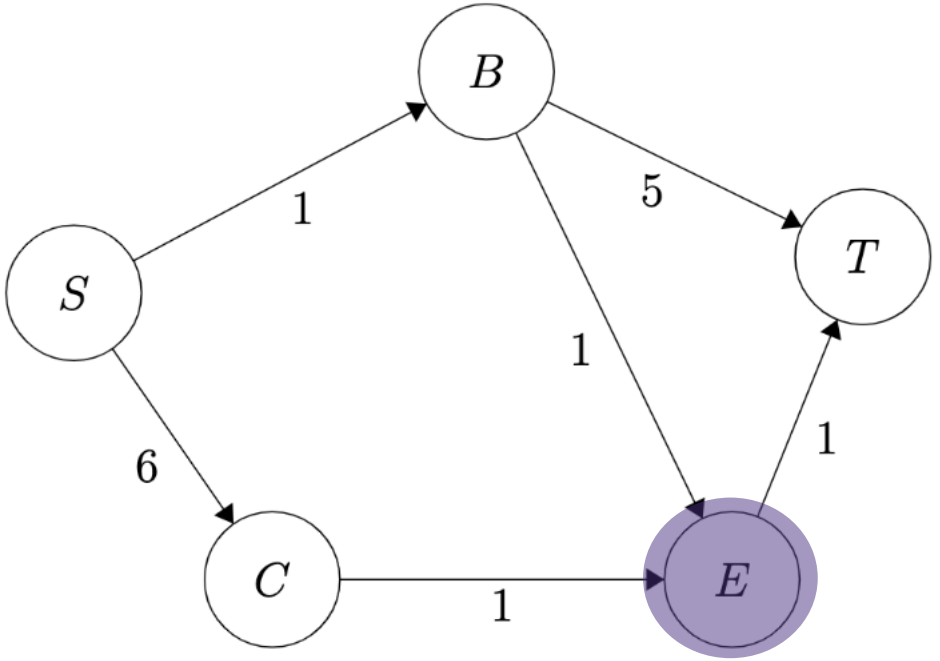


Dijkstra's Run Through

Pseudocode

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

Vertex	Distance	Predecessor	Processed
S	0	--	Yes
C	6	S	No
B	1	S	Yes
T	~~6~~ 3	E	No
E	2	B	Yes

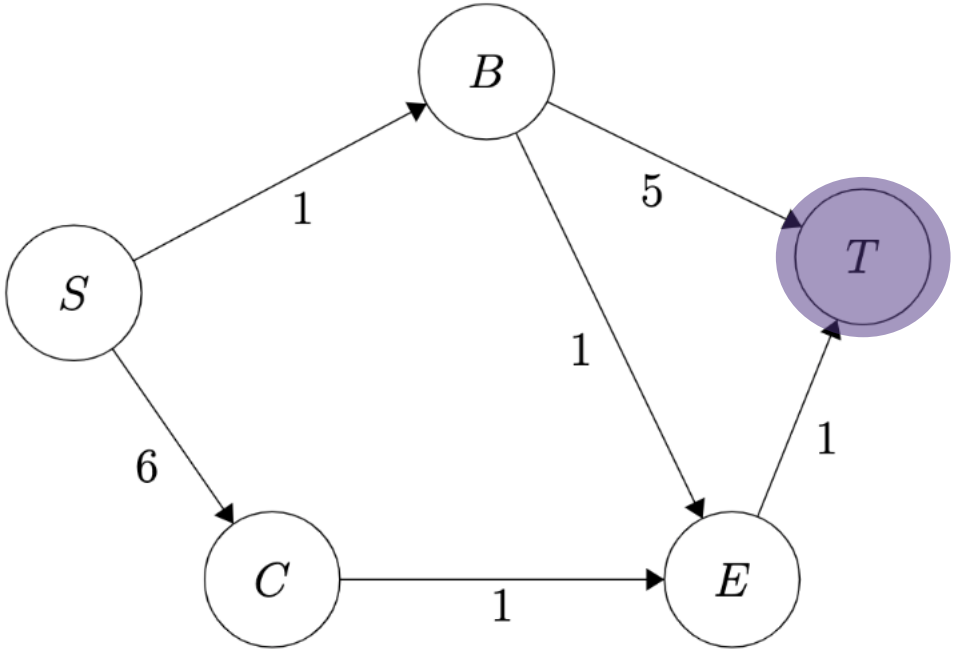


Dijkstra's Run Through

Pseudocode

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

Vertex	Distance	Predecessor	Processed
S	0	--	Yes
C	6	S	No
B	1	S	Yes
T	~~6~~ 3	E	Yes
E	2	B	Yes



Dijkstra's Pseudocode

```
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+weight(u,v) < v.dist){
            v.dist = u.dist+weight(u,v)
            v.predecessor = u
         }
      }
      mark u as processed
   }
```

Huh?

Min Priority Queue ADT

state

Set of comparable values - Ordered by "priority"

behavior

peek() – find the element with the smallest priority

insert(value) – add new element to collection

removeMin() – returns and removes element with the smallest priority

Dijkstra's Pseudocode

```
Dijkstra(Graph G, Vertex source)
   initialize distances to ∞
   mark source as distance 0
   mark all vertices unprocessed
   initialize MPQ as a Min Priority Queue, add source
   while(there are unprocessed vertices){
      u = MPQ.removeMin();
      foreach(edge (u,v) leaving u){
         if(u.dist+weight(u,v) < v.dist){
            v.dist = u.dist+weight(u,v)
            v.predecessor = u
         }
      }
      mark u as processed
   }
```

How?

Min Priority Queue ADT

state

Set of comparable values - Ordered by "priority"

behavior

peek() – find the element with the smallest priority

insert(value) – add new element to collection

removeMin() – returns and removes element with the smallest priority

```
Dijkstra(Graph G, Vertex source)
   initialize distances to ∞
   mark source as distance 0
   mark all vertices unprocessed

   initialize MPQ as a Min Priority Queue
   add source
   while(there are unprocessed vertices){
      u = MPQ.removeMin();
      foreach(edge (u,v) leaving u){


          if(u.dist+weight(u,v) < v.dist){
                v.dist = u.dist+weight(u,v)
                v.predecessor = u
         }
      }
      mark u as processed
   }
```

```
Dijkstra(Graph G, Vertex source)
   for (Vertex v : G.getVertices())
       { v.dist = INFINITY; }
   G.getVertex(source).dist = 0;

   initialize MPQ as a Min Priority Queue
   add source
    while(MPQ is not empty){
      u = MPQ.removeMin();
      for (Edge e : u.getEdges(u)){
         oldDist = v.dist;
         newDist = u.dist+weight(u,v)
         if(newDist < oldDist){
            v.dist = newDist
            v.predecessor = u
            if(oldDist == INFINITY) { MPQ.insert(v) }
            else { MPQ.updatePriority(v, newDist) }
         }
      }
   }
```

Dijkstra's Runtime

Just like when we analyzed BFS, don't just work inside out; try to figure out how many times each line will be executed.

```
Dijkstra(Graph G, Vertex source)
   for (Vertex v : G.getVertices()) { v.dist = INFINITY; }
   G.getVertex(source).dist = 0;
   initialize MPQ as a Min Priority Queue, add source
   while(MPQ is not empty){
      u = MPQ.removeMin();
      for (Edge e : u.getEdges(u)){
         oldDist = v.dist; newDist = u.dist+weight(u,v)
         if(newDist < oldDist){
            v.dist = newDist
            v.predecessor = u
            if(oldDist == INFINITY) { MPQ.insert(v) }
            else { MPQ.updatePriority(v, newDist) }
         }
      }
   }
```

+logV

+logV

This actually doesn't run m times for every iteration of the outer loop. It actually will run m times in total; if every vertex is only removed from the priority queue (processed) once, then we examine each edge once. Each line inside this foreach gets multiplied by a single E instead of E * V.

Tight O Bound = O(n log n + m log n)

More Dijkstra's Implementation

The details of the implementation depend on what data structures you have available.

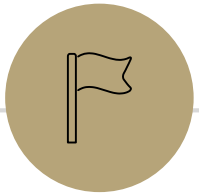
Your implementation in the programming project will be different in a few spots.

Our running time is $\Theta(E \log V + V \log V)$ i.e. $\Theta(m \log n + n \log n)$.

If you go to Wikipedia right now, they say it's $O(E + V \log V)$

They're using a Fibonacci heap instead of a binary heap.

$\Theta(E \log V + V \log V)$ is the right running time for this class.



Optional Content
More Graph Applications

Another Application of Shortest Paths

Shortest path algorithms are obviously useful for GoogleMaps.

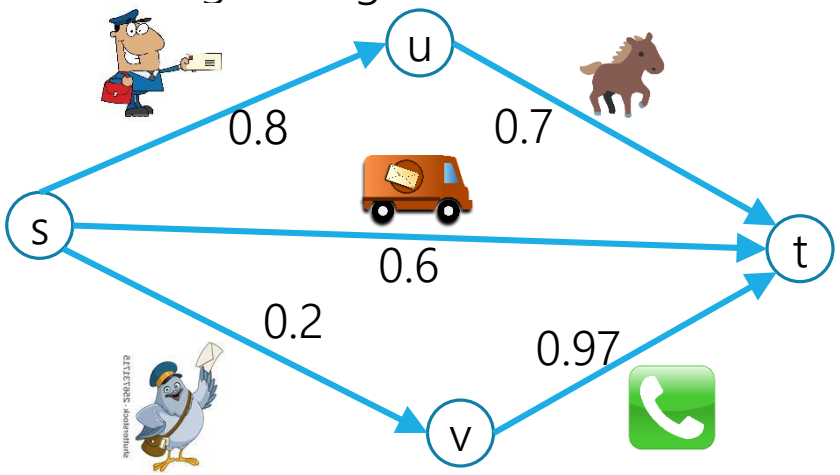
The wonderful thing about graphs is they can encode **arbitrary** relationships among objects.

We won't test you on this application of Dijkstra's

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

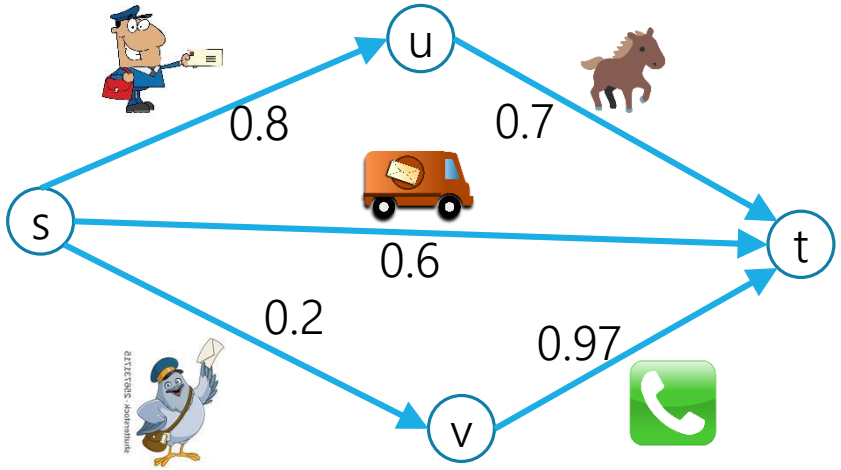
Given: a directed graph G, where each edge weight is the probability of successfully transmitting a message across that edge
Find: the path from s to t with maximum probability of message transmission

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.

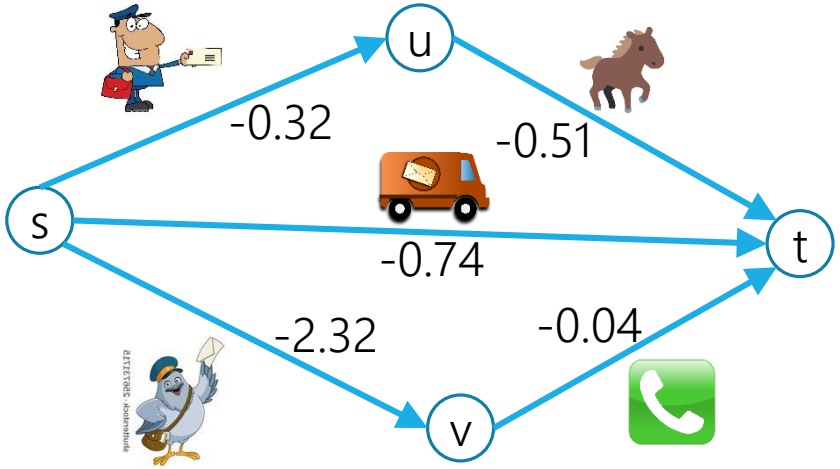


We only know how to handle sums of edge weights.

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.

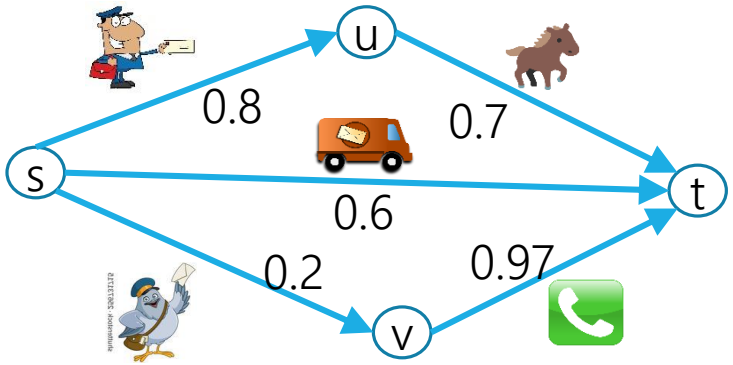
1. When we take logs, our edge weights become negative.

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



Transform Input

Weighted Shortest Paths

Transform Output