Bellman-Ford

CSE 417 Winter 21
Lecture 15

Today

Dynamic Programming on Graphs

We’re building up to “Bellman-Ford” and “Floyd-Warshall”

Two very clever algorithms – we won’t ask you to be as clever.

But they’re standard library functions, so it’s good to know.

And deriving them together is good for practicing DP skills.

Shortest Paths

Shortest Path Problem
Given: A directed graph and a vertex s Find: The length of the shortest path from s to t.

The length of a path is the sum of the edge weights.

Baseline: Dijkstra's Algorithm

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
     }
```

In 373, we said the running time was $O(m \log n + n \log n)$

Can be sped up to $O(m + n \log n)$ by inserting a different heap implementation.

A recurrence

Suppose you have a directed acyclic graph G.

How could you find distances from s?

What’s one step in this problem?

A recurrence

Suppose you have a directed acyclic graph G.

How could you find distances from s?

What's one step in this problem?

Choosing the predecessor, i.e. "the last edge" on a path.

A recurrence

$$dist(v) = \begin{cases} 0 & \text{if } v \text{ is the source} \\ \min_{u:(u,v)\in E} \{dist(u) + weight(u,v)\} & \text{otherwise} \end{cases}$$

Our memoization structure can be the graph itself.

What's an evaluation order? (Remember we're in a DAG!)

A recurrence

$$dist(v) = \begin{cases} 0 & \text{if } v \text{ is the source} \\ \min_{u:(u,v)\in E} \{dist(u) + weight(u,v)\} & \text{otherwise} \end{cases}$$

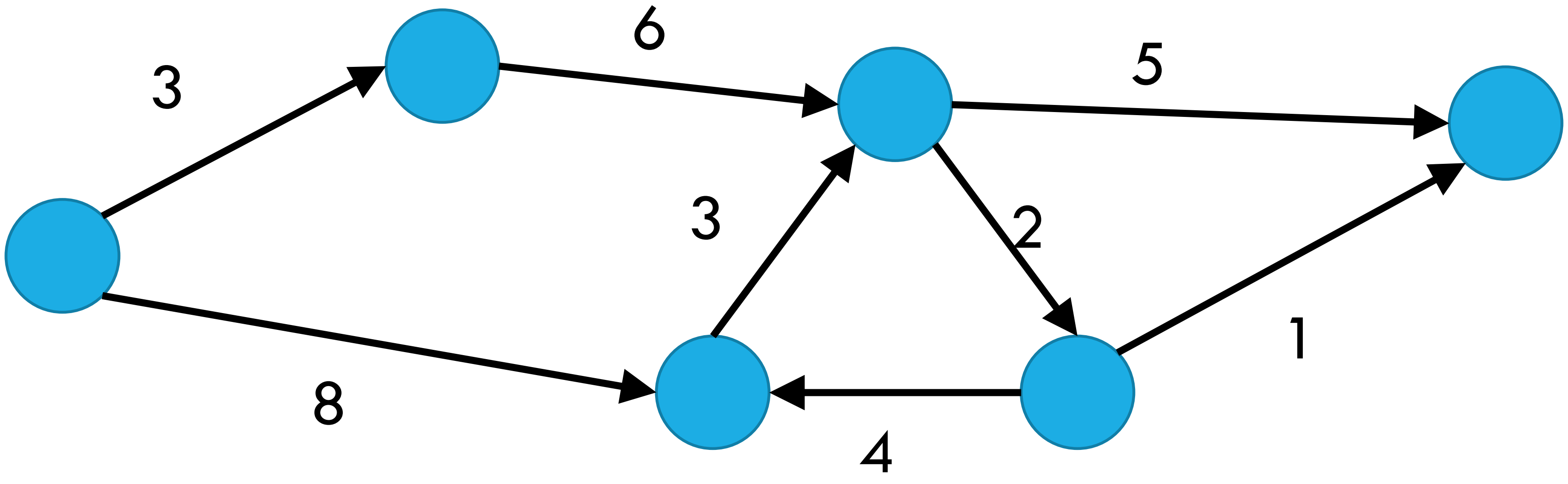
Our memoization structure can be the graph itself.

What's an evaluation order? (Remember we're in a DAG!)

A topological sort! – we need to have distances for all incoming edges calculated.

What about cycles?

$$dist(v) = \begin{cases} 0 & \text{if } v \text{ is the source} \\ \min\limits_{u:(u,v)\in E} \{dist(u) + weight(u,v)\} & \text{otherwise} \end{cases}$$



Cycles

We need some way to “order” the paths.

I.e. we need to be sure we always have **something** to look up.

It doesn’t have to be the perfect distance necessarily…

As long as we’ll realize it and update later

And as long as we can fix it to the true distance eventually.

Ordering

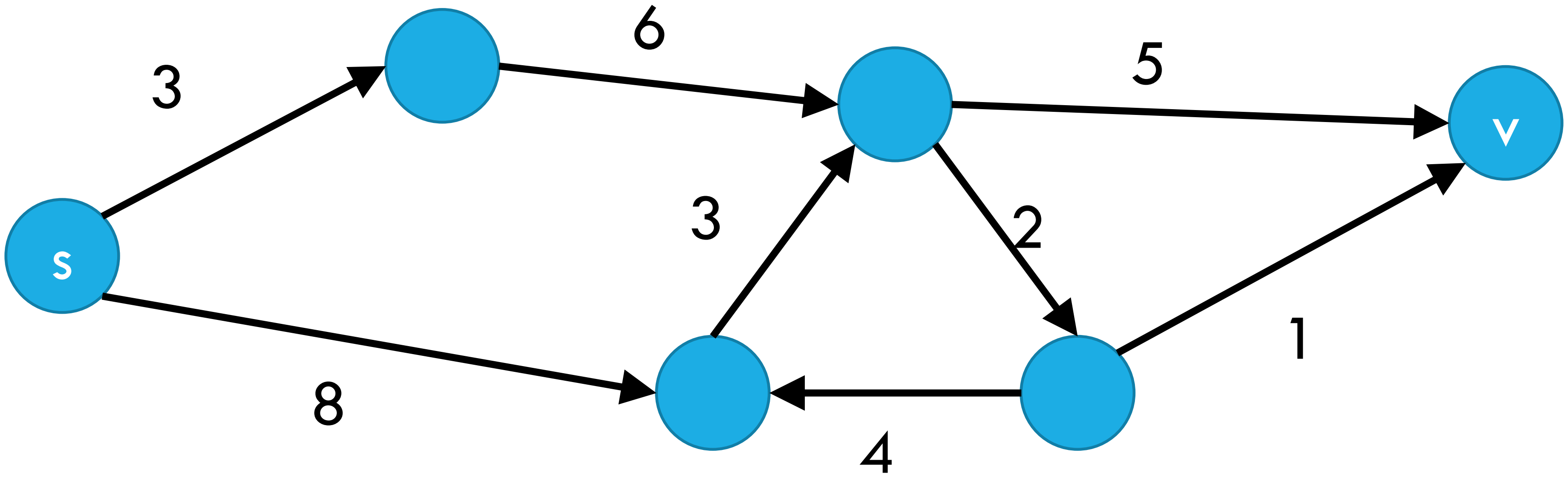
Instead of $dist(v)$, (the true distance) right from the start, we'll let

$dist(v, i)$ to be the length of the shortest path from the source to v that uses at most i edges.

That breaks ties – counting the number of edges required!

$dist(v, i) =$

Distances



$dist(v, 2) = \infty$ (can't get there in 2 hops)

$dist(v, 3) = 14$

$dist(v, 4) = 12$

Ordering

Instead of $dist(v)$, (the true distance) right from the start, we'll let

$dist(v, i)$ to be the length of the shortest path from the source to v that uses at most i edges.

That breaks ties – counting the number of edges required!

$dist(v, i) =$

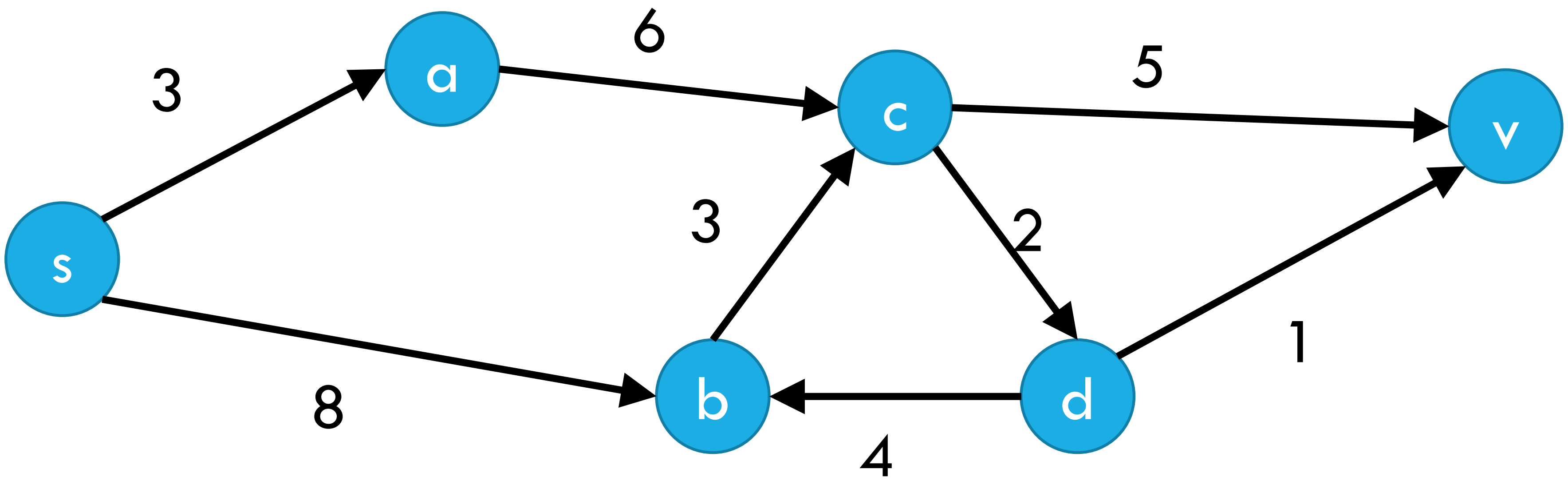
Ordering

Instead of $dist(v)$, we want the

$dist(v, i)$ to be the length of the shortest path from the source to u that uses at most i edges.

$$dist(v,i) = \begin{cases} 0 & \text{if } i = 0 \text{ and } v \text{ is the source} \\ \infty & \text{if } i = 0 \text{ and } v \text{ is not the source} \\ \min\left\{ \min_{u:(u,v)\in E} \{dist(u,i-1)\} + w(u,v)\,, dist(v,i-1) \right\} & \text{o/w} \end{cases}$$

Sample calculation



Vertex\i	0	1	2	3	4	5
S	0	0	0	0	0	0
A	∞	3	3	3	3	3
B	∞	8	8	8	8	8
C	∞	∞	9	9	9	9
D	∞	∞	∞	11	11	11
V	∞	∞	∞	14	12	12

Pseudocode

```
Initialize source.dist[0]=0, u.dist[0]=∞ for others
for(i from 1 to ??)
    for(every vertex v) //what order?
        v.dist[i] = v.dist[i-1]
        for(each incoming edge (u,v))//hmmm
            if(u.dist[i-1]+weight(u,v)<v.dist[i])
                v.dist[i]=u.dist[i-1]+weight(u,v)
            endIf
        endFor
    endFor
endFor
```

$$dist(v,i) = \begin{cases} 0 & \text{if } i = 0 \text{ and } v \text{ is the source} \\ \infty & \text{if } i = 0 \text{ and } v \text{ is not the source} \\ \min\left\{ \min_{u:(u,v)\in E} \{dist(u,i-1)\} + w(u,v)\, , dist(v,i-1) \right\} \end{cases}$$

Pseudocode

```
Initialize source.dist[0]=0, u.dist[0]=∞ for others
for(i from 1 to n-1)
    for(every vertex
        v.dist[i] = v.dist[i-1]
        for(each incoming edge (u,v))//hmmm
            if(u.dist[i-1]+weight(u,v)<v.dist[i])
                v.dist[i]=u.dist[i-1]+weight(u,v)
            endIf
        endFor
    endFor
endFor
```

The shortest path will never need more than $n-1$ edges
(more than that and you've got a cycle)

Pseudocode

Only ever need values from the last iteration

Order doesn't matter!!

```
Initialize sour
for(i from 1 to
    for(every vertex v) //what order?
        v.dist[i] = v.dist[i-1]
        for(each incoming edge (u,v))//hmmm
            if(u.dist[i-1]+weight(u,v)<v.dist[i])
                v.dist[i]=u.dist[i-1]+weight(u,v)
            endIf
        endFor
    endFor
endFor
```

Pseudocode

```
Initialize source.dist[0]=0, u.dist[0]=∞ for others
for(i from 1 to n-1)
    for(every vertex v) //any order
        v.dist[i] = v.dist[i-1]
        for(each incoming edge (u,v))//hmmm
            if(u.dist[i-1]+weight(u,v)<v.dist[i])
                v.dist[i]=u.dist[i-1]+weight(u,v)
            endIf
        endFor
    endFor
endFor
```

Graphs don't usually have easy access to their incoming edges (just the outgoing ones)

Pseudocode

```
Initialize source.dist[0]=0, u.dist[0]=∞ for others
for(i from 1 to n-1)
    for(every vertex v) //any order
        v.dist[i] = v.dist[i-1]
        for(each incoming edge (u,v))//hmmm
            if(u.dist[i-1]+weight(u,v)<v.dist[i])
                v.dist[i]=u.dist[i-1]+weight(u,v)
            endIf
        endFor
    endFor
endFor
```

But the order doesn't matter – as long as we check every edge, the processing order is irrelevant.
So if we only have access to outgoing edges…

Pseudocode

```
Initialize source.dist[0]=0, u.dist[0]=∞ for others
for(i from 1 to n-1)
    set u.dist[i] to u.dist[i-1] for every u
    for(every vertex u) //any order
        for(each outgoing edge (u,v))//better!
            if(u.dist[i-1]+weight(u,v)<v.dist[i])
                v.dist[i]=u.dist[i-1]+weight(u,v)
            endIf
        endFor
    endFor
endFor
```

Pseudocode

```
Initialize source.dist[0]=0, u.dist[0]=∞ for others
for(i from 1 to n-1)
    set u.dist[i] to u.dist[i-1] for every u
    for(every vertex u) //any order
        for(each outgoing edge (u,v))//better!
            if(u.dist[i-1]+weight(u,v)<v.dist[i])
                v.dist[i]=u.dist[i-1]+weight(u,v)
            endIf
        endFor
    endFor
endFor
```

We don’t really need all the different values…
Just the most recent value.

Pseudocode

```
Initialize source.dist=0, u.dist=∞ for others
for(i from 1 to n-1)
    set u.dist[i] to u.dist[i-1] for every u
    for(every vertex u) //any order
        for(each outgoing edge (u,v))//better!
            if(u.dist+weight(u,v)<v.dist)
                v.dist=u.dist+weight(u,v)
            endIf
        endFor
    endFor
endFor
```

We don't really need all the different values…
Just the most recent value.

Pseudocode

```
Initialize source.dist=0, u.dist=∞ for others
for(i from 1 to n-1)
   for(every vertex u) //any order
      for(each outgoing edge (u,v))//better!
         if(u.dist+weight(u,v)<v.dist)
            v.dist=u.dist+weight(u,v)
         endIf
      endFor
   endFor
endFor
```

We don't really need all the different values…
Just the most recent value.

A Caution

We did change the code when we got rid of the indexing

You might have a mix of `dist[i]`,`dist[i+1]`,`dist[i+2]`,... at the same time.

That's ok!

You'll only "override" a value with a better one.

And you'll eventually get to $dist(u, n-1)$

After iteration i, u stores $dist(u, k)$ for some $k \geq i$.

Exit early

If you made it through an entire iteration of the outermost loop and don't update any $dist()$

Then you won't do any more updates in the next iteration either. You can exit early.

More ideas to save constant factors on Wikipedia (or the textbook)

Laundry List of shortest pairs (so far)

Algorithm	Running Time	Special Case	Negative edges?
BFS	$O(m+n)$	ONLY unweighted graphs	X
Simple DP	$O(m+n)$	ONLY for DAGs	X
Dijkstra's	$O(m+n\log n)$		X
Bellman-Ford	$O(mn)$		???

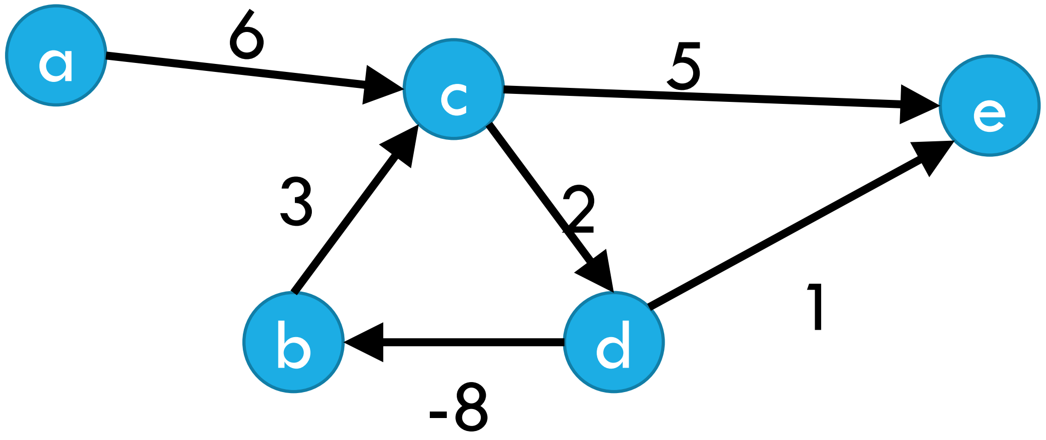
Pseudocode

```
Initialize source.dist=0, u.dist=∞ for others
for(i from 1 to n-1)
   for(every vertex u) //any order
      for(each outgoing edge (u,v))//better!
         if(u.dist+weight(u,v)<v.dist)
            v.dist=u.dist+weight(u,v)
         endIf
      endFor
   endFor
endFor
```

What happens if there's a negative cycle?

Negative Edges

Negative Cycles

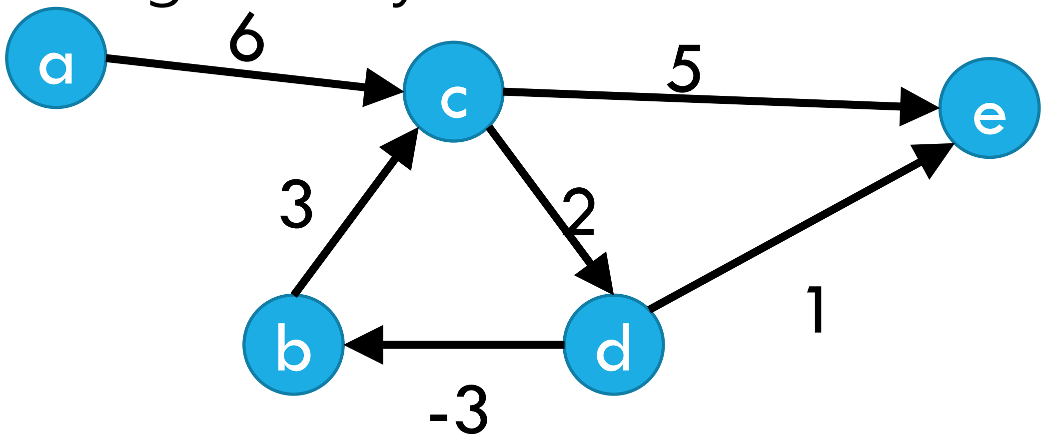


The fastest way from a to e

(i.e. least-weight walk) isn't defined!

No valid answer ($-\infty$)

Negative edges, but only non-negative cycles



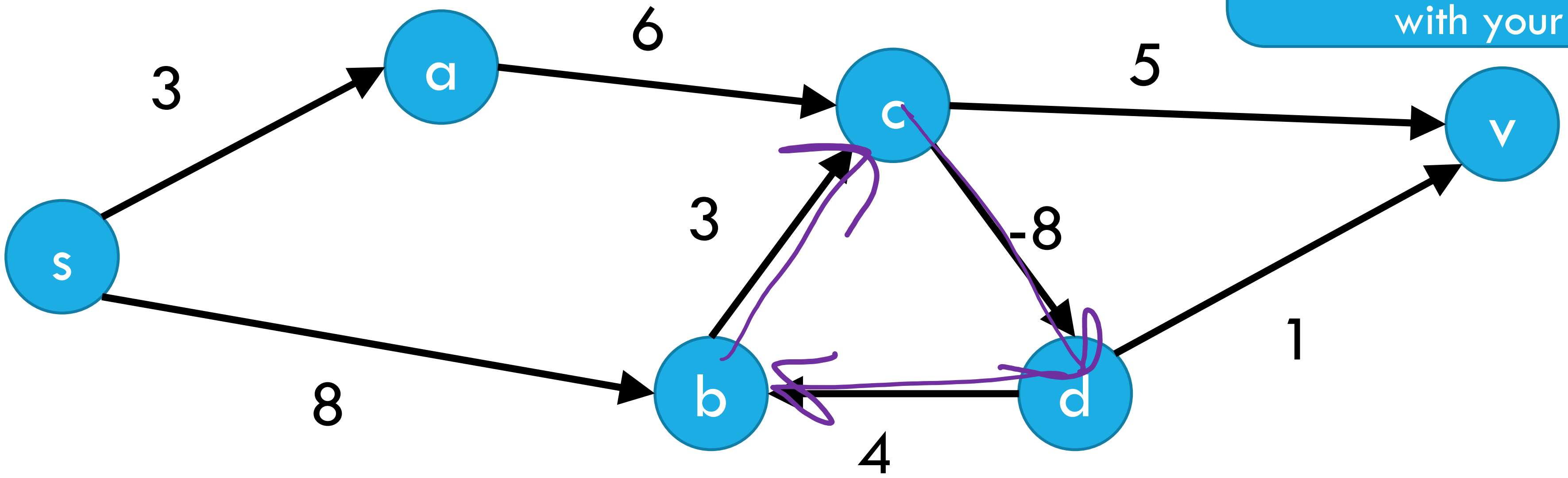
Dijkstra's might fail

But the shortest path IS defined.

There is an answer

Negative Cycle

Fill out the poll everywhere for Activity Credit!
Go to pollev.com/cse417 and login with your UW identity



Vertex\\i	0	1	2	3	4	5	6
S	0	0	0	0	0		
A	∞	3	3	3	3		
B	∞	8	8	8	5		
C	∞	∞	9	9	9		
D	∞	∞	∞	1	1		
V	∞	∞	∞	14	2		

Laundry List of shortest pairs (so far)

Algorithm	Running Time	Special Case only	Negative edges?
BFS	$O(m+n)$	ONLY unweighted graphs	X
Simple DP	$O(m+n)$	ONLY for DAGs	X
Dijkstra's	$O(m+n\log n)$		X
Bellman-Ford	$O(mn)$		Yes!

All Pairs Shortest Paths

All Pairs

For Dijkstra's or Bellman-Ford we got the distances from the source to every vertex.

What if we want the distances from every vertex to every other vertex?

Another Recurrence

$$dist(v) = \begin{cases} 0 & \text{if } v \text{ is the source} \\ \min_{u:(u,v)\in E} \{dist(u) + weight(u,v)\} & \text{otherwise} \end{cases}$$

Another clever way to order paths.

Put the vertices in some (arbitrary) order $1,2, \ldots, n$

Let $dist(u, v, i)$ be the distance from u to v where the only **intermediate** nodes are $1,2, \ldots, i$

Another Recurrence

Put the vertices in some (arbitrary) order $1,2,\ldots,n$

Let $dist(u,v,i)$ be the distance from u to v where the only **intermediate** nodes are $1,2,\ldots,i$

$$\mathrm{dist}(u,v,i)=\begin{cases} weight(u,v) & \text{if } i=0, (u,v) \text{ exists} \\ 0 & \text{if } i=0, u=v \\ \infty & \text{if } i=0, \text{ no edge } (u,v) \\ \min\{\mathrm{dist}(u,i,i-1)+\mathrm{dist}(i,v,i-1), \mathrm{dist}(u,v,i-1)\} & \text{otherwise} \end{cases}$$

Pseudocode

```
dist[][] = new int[n-1][n-1]
for(int i=0; i<n; i++)
   for(int j=0; j<n; j++)
      dist[i][j] = edge(i,j) ? weight(i,j) : ∞
for(int i=0; i<n; i++)
    dist[i][i] = 0
for every vertex r
   for every vertex u
      for every vertex v
         if(dist[u][r] + dist[r][v] < dist[u][v])
              dist[u][v]=dist[u][r] + dist[r][v]
```

"standard" form of the "Floyd-Warshall" algorithm. Similar to Bellman-Ford, you can get rid of the last entry of the recurrence (only need 2D array, not 3D array).

Running Time

$O(n^3)$

How does that compare to Dijkstra's?

Running Time

If you really want all-pairs…

Could run Dijkstra's n times…

$O(mn \log n + n^2 \log n)$

If $m \approx n^2$ then Floyd-Warshall is faster!

Floyd-Warshall also handles negative weight edges.

Ask Robbie after how to detect them.

Takeaways

Some clever dynamic programming on graphs.

Which library to use?

Need just one source?

Dijkstra’s if no negative edge weights.

Bellman-Ford if negative edges.

Need all sources?

Flord-Warshall if negative edges or $m \approx n^2$

Repeated Dijkstra’s otherwise