Graph DP Mid-Quarter Summary

CSE 421 22AU
Lecture 17

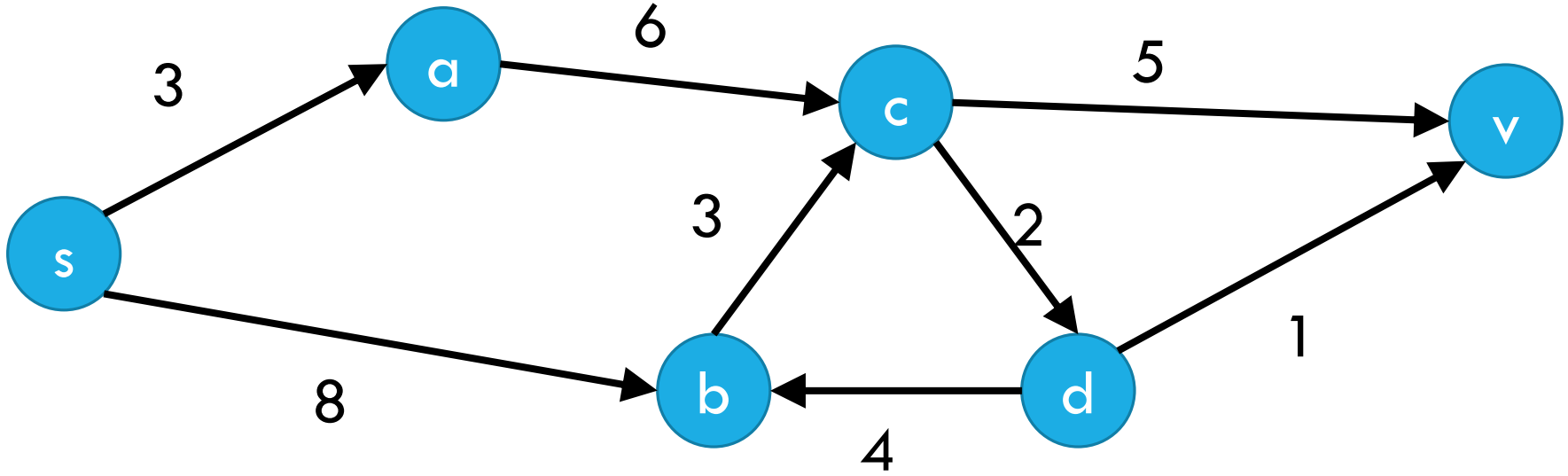
Ordering

Instead of $dist(v)$, we want

$dist(v, i)$ to be the length of the shortest path from the source to v 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

```
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
```

Only ever need values from the previous iteration
Order doesn't matter!!

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 "overwrite" 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 a 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	$O(mn)$		???

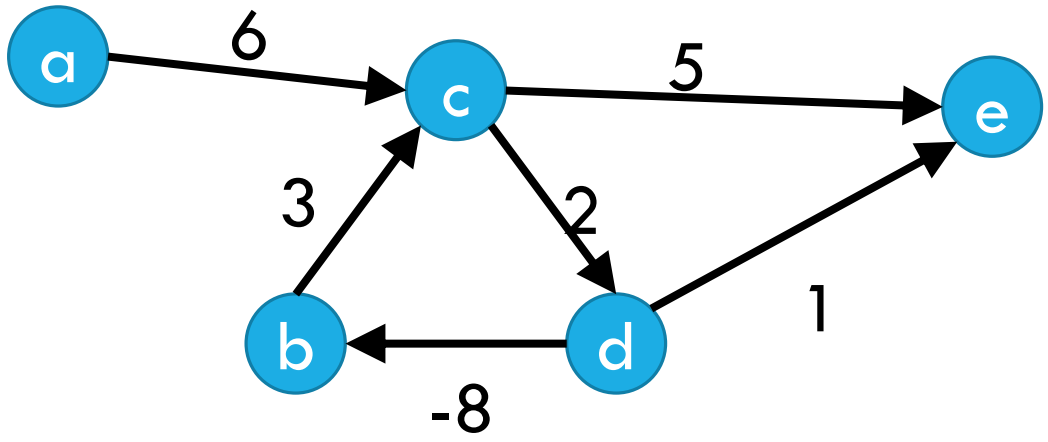
Pseudocode

```
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

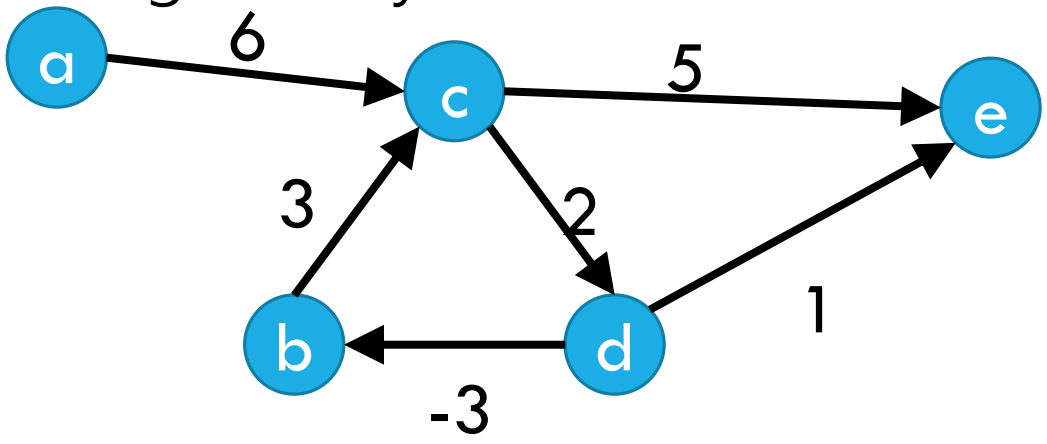


The fastest way from a to e

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

No valid answer ($-\infty$)

Negative edges, but only non-negative cycles



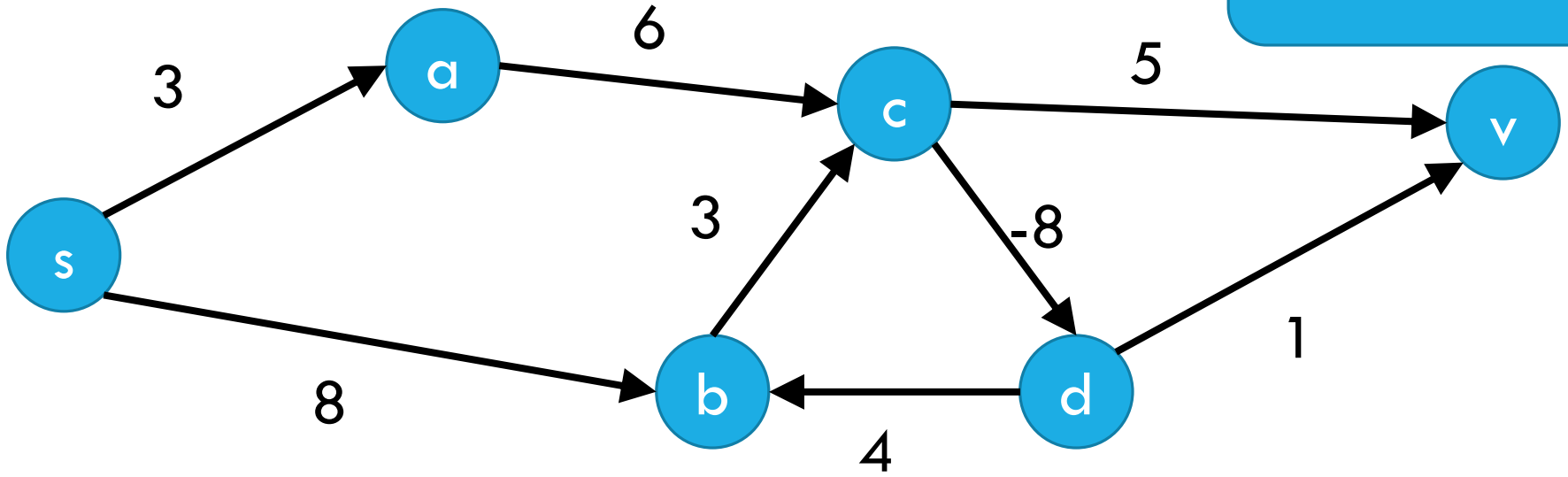
Dijkstra's might fail

But the shortest path IS defined.

There is an answer

Negative Cycle

Pollev.com/Robbie



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		

Negative Cycles

If you have a negative length edge: Dijkstra's might or might not give you the right answer.

And it can't even tell you if there's a negative cycle (i.e. whether some of the answers are supposed to be negative infinity)

For Bellman-Ford:

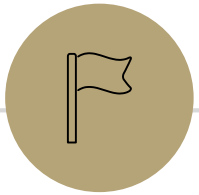
Run one extra iteration of the main loop– if any value changes, you have a negative length cycle. Some of the values you calculated are wrong.

Run a BFS from the vertex that just changed. Anything you can find should have $-\infty$ as the distance. (anything else has the correct [finite] value).

If the extra iteration doesn't change values, no negative length cycle.

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?

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?

Why? Most commonly pre-computation.

Imagine you're google maps – you could run Dijkstra's every time anyone anywhere asks for directions…

Or store how to get between transit hubs and only use Dijkstra's locally.

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 $1,2,\ldots,i$

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

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.

If $dist(u, u) < 0$ then there's a negative weight cycle.

Takeaways

Some clever dynamic programming on graphs.

Which library to use (at least asymptotically)?

Need just one source?

Dijkstra's if no negative edge weights.

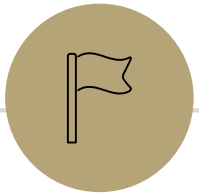
Bellman-Ford if negative edges.

Need all sources?

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

Repeated Dijkstra's otherwise

These are all asymptotics! For any "real-world" problem prefer running actual code to see which is faster.



DP Context

DP history

So…why is it called “dynamic programming?”

“programming” is an old-timey meaning of the word.

It means “scheduling”

Like a conference has a “program” of who speaks where when.
Or a television executive decides on the nightly programming (what show airs when).

DP history

So…*dynamic*?

The phrase “dynamic programming” was popularized by Richard Bellman.

He was a researcher, funded by the U.S. military….

But the Secretary of Defense [as Bellman tells it] hated research. And hated math even more.

So Bellman needed a description of his research that everyone would approve of.

DP history

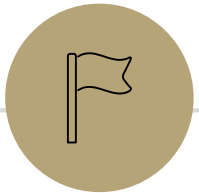
Dynamic

Is actually an accurate adjective – what we think is the best option (include/exclude) can change over time.

Even better

“It’s impossible to use the word ‘dynamic’ in a pejorative sense”

“It was something not even a Congressman could object to.”



Techniques So Far

What have we seen so far?

Stable Matchings

Graph Search

- BFS/DFS
- Graph modeling

Greedy Algorithms

Divide and Conquer

Dynamic Programming

Stable Matchings

Modeling matters!

It's better to be a proposer than a chooser!

Algorithms can be used to prove 'non-computational' facts

Stable Matchings always exist is easiest to prove by saying "here's how to find one."

Reductions

Sometimes there's a clever way to use an existing library (we'll need these a lot later in the quarter).

Graph Search

BFS and DFS search through a graph differently

So you can adapt them to solve different problems!

Use libraries

Finding SCCs and Topological sorting are “almost free” preprocessing

2-Coloring can be performed in linear time.

Greedy

Code is easy; proofs are hard.

Generating examples is **extremely** important.

To frame your thinking for proofs

Greedy stays ahead

Exchange argument

Structural result

Divide and Conquer

Trust the recursion.

Don’t be afraid to change what the recursive call gives you!

Add extra parameters!

State in English what the recursive call gives you.

In your “combine” step, make sure you’re beating baseline!

Dynamic Programming

Focus on solving the problem recursively; everything else is (mostly) formulaic once you've done that.

Write exactly the problem you're solving in English.

It's better to get down a "guess" at the problem and then see where you get stuck.

Don't be afraid to add a second recurrence or extra parameters.

Don't try to cleverly figure out which option is best. Try them all.

The magic of recursion tells you which is best for a particular situation.

How To Approach Problems

In section, we've made you follow these steps:

1. Read the problem carefully (make sure you know what problem you're actually solving)

2. Make some sample inputs/outputs

3. Set a "baseline."

4. Then try to generate the algorithm.

It's hard to take the time to do these in an exam, but at least make sure you do #1. Solving the wrong problem is not good for test-taking.