ساختمان داده و الگوریتم ها (CE203)

جلسه بیستم:
برنامه نویسی پویا

سجاد شیرعلی شهرضا
پاییز 1400
شنبه، 20 آذر 1400

Based on Stanford CS161 Slides from Summer 2020 by Karey Shi

اطلاع رسانی

- بخش مرتبط کتاب برای این جلسه: 15.3
- یادآوری مهلت ارسال تمرین سوم: 8 صبح روز چهارشنبه 24 آذر 1400
- امتحانک سوم: دوشنبه هفته آینده، 29 آذر 1400، در وقت کلاس
- قرار دادن نظرسنجی چهارم : مهلت ارسال: 8 صبح روز سه شنبه 30 آذر 1400
- نمرات تمرین دوم و امتحان میان ترم اعلام شده است
    - در صورت داشتن سوال در مورد نمرات، از طریق ایمیل با من مکاتبه کنید.

چکیده ای از نظر سنجی سوم

- تا اینجای ترم، ارائه درس (به طور کلی) را چطور ارزیابی میکنید؟
    - 14%: عالی. از این بهتر نمیشود. همینطور ادامه دهید.
    - 58%: قابل قبول. هرچند میتوان بهتر کرد، اما ادامه همین روال هم قابل قبول است.
    - 29%: نیازمند بهبود. نیاز به تغییراتی است تا کیفیت ارائه درس بهتر شود.
    - 0%: افتضاح! واقعا باید برای این درس فکری کرد.
- تا اینجای ترم، محتوای مطالب ارائه شده (اسلایدها) را چطور ارزیابی میکنید؟
    - 19%: عالی. از این بهتر نمیشود. همینطور ادامه دهید.
    - 61%: قابل قبول. هرچند میتوان بهتر کرد، اما ادامه همین روال هم قابل قبول است.
    - 22%: نیازمند بهبود. نیاز به تغییراتی است تا کیفیت ارائه درس بهتر شود.
    - 0%: افتضاح! واقعا باید برای این درس فکری کرد.

چکیده ای از نظر سنجی سوم

- تا اینجای ترم، نحوه ارائه مطالب در جلسات درس را چطور ارزیابی می کنید؟
  - 29%: عالی. از این بهتر نمیشود. همینطور ادامه دهید.
  - 58%: قابل قبول. هرچند میتوان بهتر کرد، اما ادامه همین روال هم قابل قبول است.
  - 11%: نیازمند بهبود. نیاز به تغییراتی است تا کیفیت ارائه درس بهتر شود.
  - 3%: افتضاح! واقعا باید برای این درس فکری کرد.
- تا اینجای ترم، نظر شما در مورد تمرینهای خواسته شده از شما چطور است؟
  - 11%: کم. انتظار تمرینهای بیشتر و یا سختتری را داشتم تا بیشتر به یادگیری من کمک کنند.
  - 56%: مناسب. به نظر حجم و دشواری تمرینها مناسب است.
  - 35%: زیاد. حجم تمرینها خیلی زیاد است و زمان زیادی از من می گیرند.

مقدمه ای بر برنامه نویسی پویا

یک روش طراحی الگوریتم

DYNAMIC PROGRAMMING

Dynamic programming (DP) is an algorithm design paradigm.
It’s often used to solve optimization problems (e.g. *shortest* path).

DYNAMIC PROGRAMMING

Dynamic programming (DP) is an algorithm design paradigm.
It’s often used to solve optimization problems (e.g. *shortest* path).

We’ll see two examples of DP today:
Bellman-Ford and Floyd-Warshall algorithms.
We will go over some DP practice problems in depth next week.

But first, an overview of DP!

DYNAMIC PROGRAMMING

Elements of dynamic programming:

DYNAMIC PROGRAMMING

Elements of dynamic programming:

Optimal substructure: the optimal solution of a problem can be expressed in terms of optimal solutions to smaller sub-problems.

DYNAMIC PROGRAMMING

Elements of dynamic programming:

Optimal substructure: the optimal solution of a problem can be expressed in terms of optimal solutions to smaller sub-problems.

e.g. $\mathbf{d^{(k)}[b] = \min\{\ d^{(k-1)}[b], \min_a\{d^{(k-1)}[a] + w(a, b)\}\ \}}$

DYNAMIC PROGRAMMING

Elements of dynamic programming:

Optimal substructure: the optimal solution of a problem can be expressed in terms of optimal solutions to smaller sub-problems.

e.g. $\mathbf{d^{(k)}[b] = \min\{ d^{(k-1)}[b], \min_a\{d^{(k-1)}[a] + w(a, b)\} \}}$

Overlapping sub-problems: the subproblems overlap a lot!

This means we can save time by solving a sub-problem once & cache the answer.

(this is sometimes called “memoization”)

DYNAMIC PROGRAMMING

Elements of dynamic programming:

Optimal substructure: the optimal solution of a problem can be expressed in terms of optimal solutions to smaller sub-problems.

e.g. $\mathbf{d^{(k)}[b] = \min\{ d^{(k-1)}[b], \min_a\{d^{(k-1)}[a] + w(a, b)\} \}}$

Overlapping sub-problems: the subproblems overlap a lot!

This means we can save time by solving a sub-problem once & cache the answer.

(this is sometimes called "memoization")

e.g. **Lots of different entries in the row $d^{(k)}$ may ask for $d^{(k-1)}[v]$**

DYNAMIC PROGRAMMING

Two approaches for DP

(2 different ways to think about and/or implement DP algorithms)

DYNAMIC PROGRAMMING

Two approaches for DP

(2 different ways to think about and/or implement DP algorithms)

Bottom-up: iterates through problems by size and solves the small problems first (kind of like taking care of base cases first & building up).

DYNAMIC PROGRAMMING

Two approaches for DP

(2 different ways to think about and/or implement DP algorithms)

Bottom-up: iterates through problems by size and solves the small problems first (kind of like taking care of base cases first & building up).

e.g. **Bellman-Ford (as we will see shortly!) computes $d^{(0)}$, then $d^{(1)}$, then $d^{(2)}$, etc.**

DYNAMIC PROGRAMMING

Two approaches for DP

(2 different ways to think about and/or implement DP algorithms)

Bottom-up: iterates through problems by size and solves the small problems first (kind of like taking care of base cases first & building up).
e.g. **Bellman-Ford (as we will see shortly!) computes $d^{(0)}$, then $d^{(1)}$, then $d^{(2)}$, etc.**

Top-down: instead uses recursive calls to solve smaller problems, while using memoization/caching to keep track of small problems that you've already computed answers for (simply fetch the answer instead of re-solving that problem and waste computational effort)

DYNAMIC PROGRAMMING

Two approaches for DP

(2 different ways to think about and/or implement DP algorithms)

Bottom-up: iterates through problems by size and solves the small problems first (kind of like taking care of base cases first & building up).
e.g. **Bellman-Ford (as we will see shortly!) computes $d^{(0)}$, then $d^{(1)}$, then $d^{(2)}$, etc.**

Top-down: instead uses recursive calls to solve smaller problems, while using memoization/caching to keep track of small problems that you've already computed answers for (simply fetch the answer instead of re-solving that problem and waste computational effort)
We will see a way later to implement **Bellman-Ford** using a top-down approach.

DYNAMIC PROGRAMMING

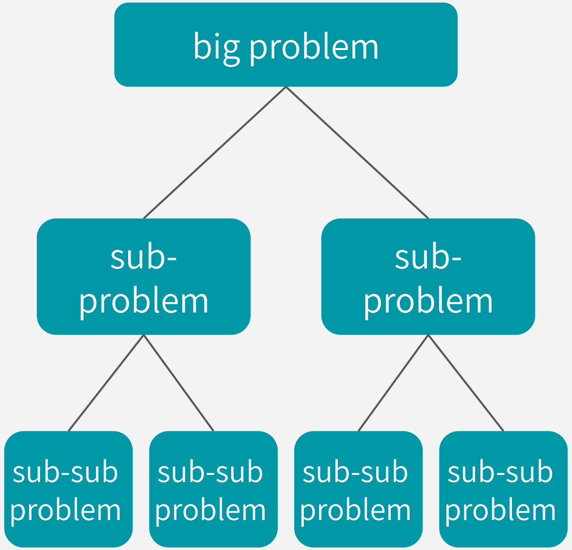
Why "dynamic programming"?

Richard Bellman invented the term in the 1950's. He was working for the RAND corporation at the time, which was employed by the Air Force, and government projects needed flashy non-mathematical non-researchy names to get funded and approved.

"It's impossible to use the word dynamic in a pejorative sense…
I thought dynamic programming was a good name.
It was something not even a Congressman could object to."

DIVIDE & CONQUER vs DP

DIVIDE-AND-CONQUER



big problem

sub-problem

sub-problem

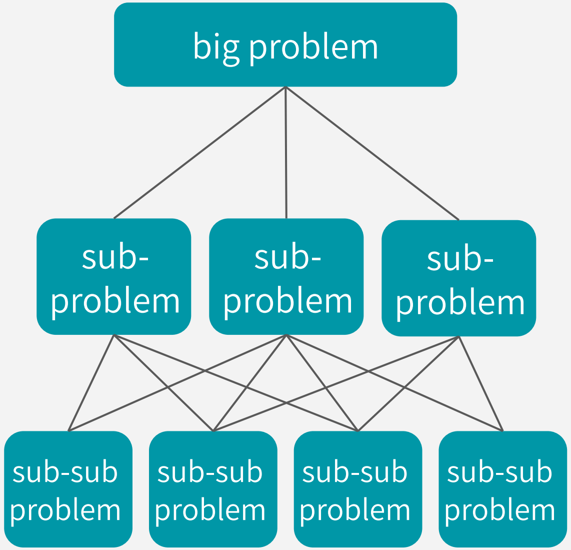
sub-sub problem

sub-sub problem

sub-sub problem

sub-sub problem

DYNAMIC PROGRAMMING



big problem

sub-problem

sub-problem

sub-problem

sub-sub problem

sub-sub problem

sub-sub problem

sub-sub problem

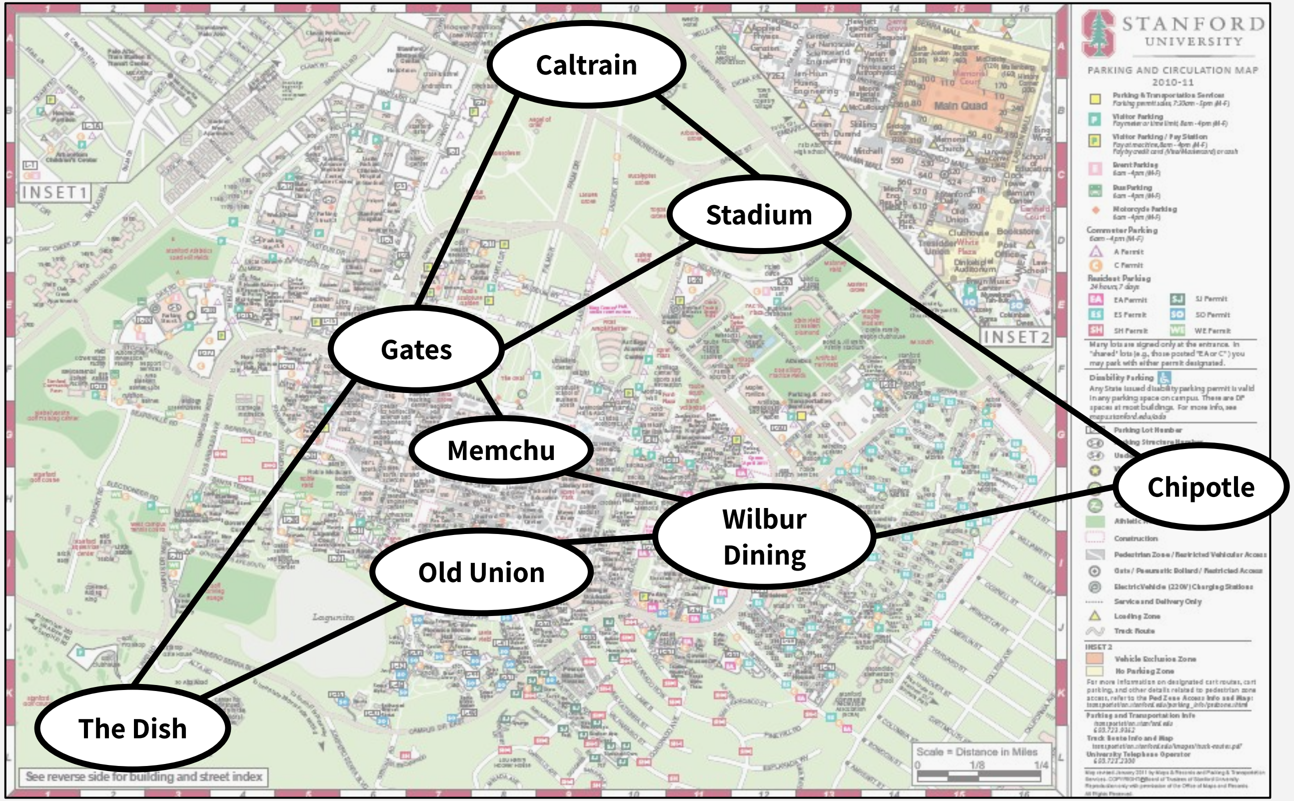
سوال؟

یافتن کوتاه ترین مسیر در گراف

تعریف مسئله

SHORTEST PATHS IN WEIGHTED GRAPHS

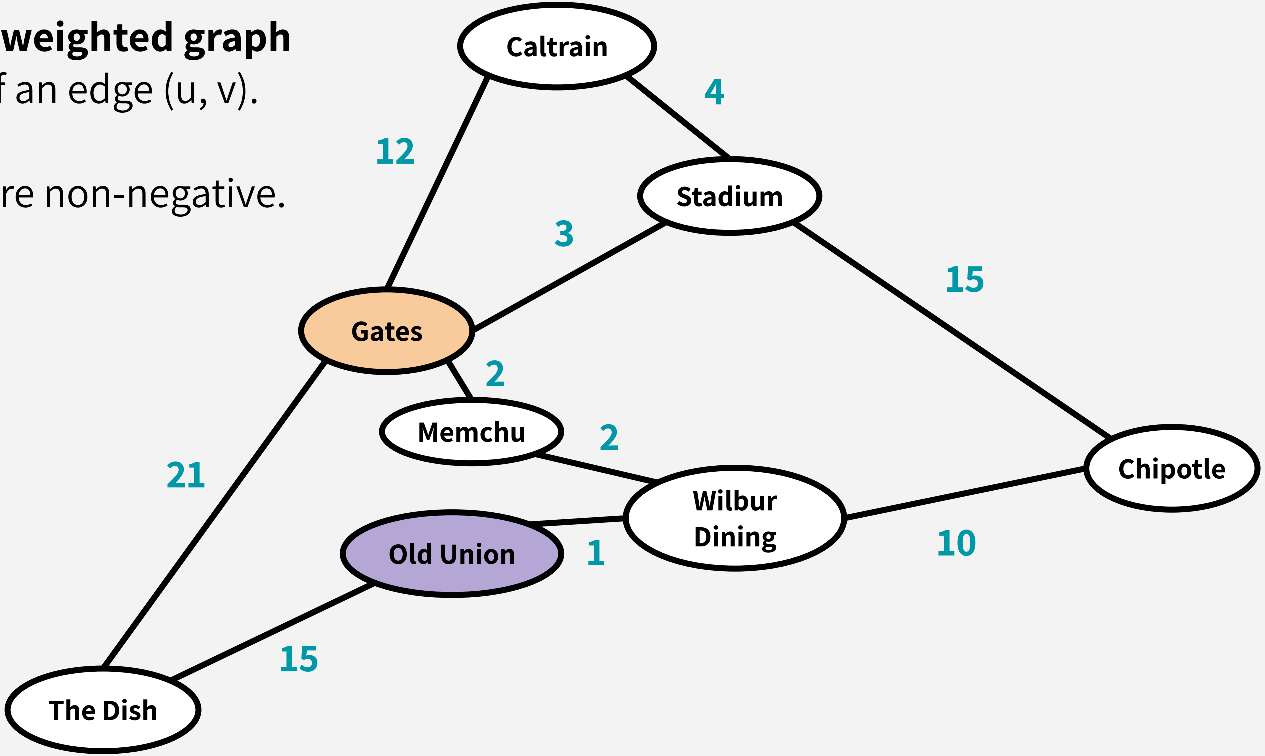
Suppose you only know your way around campus via certain landmarks



SHORTEST PATHS IN WEIGHTED GRAPHS

We can represent this as a **weighted graph** where **w(u,v)** = weight of an edge (u, v).

For today, these weights are non-negative.



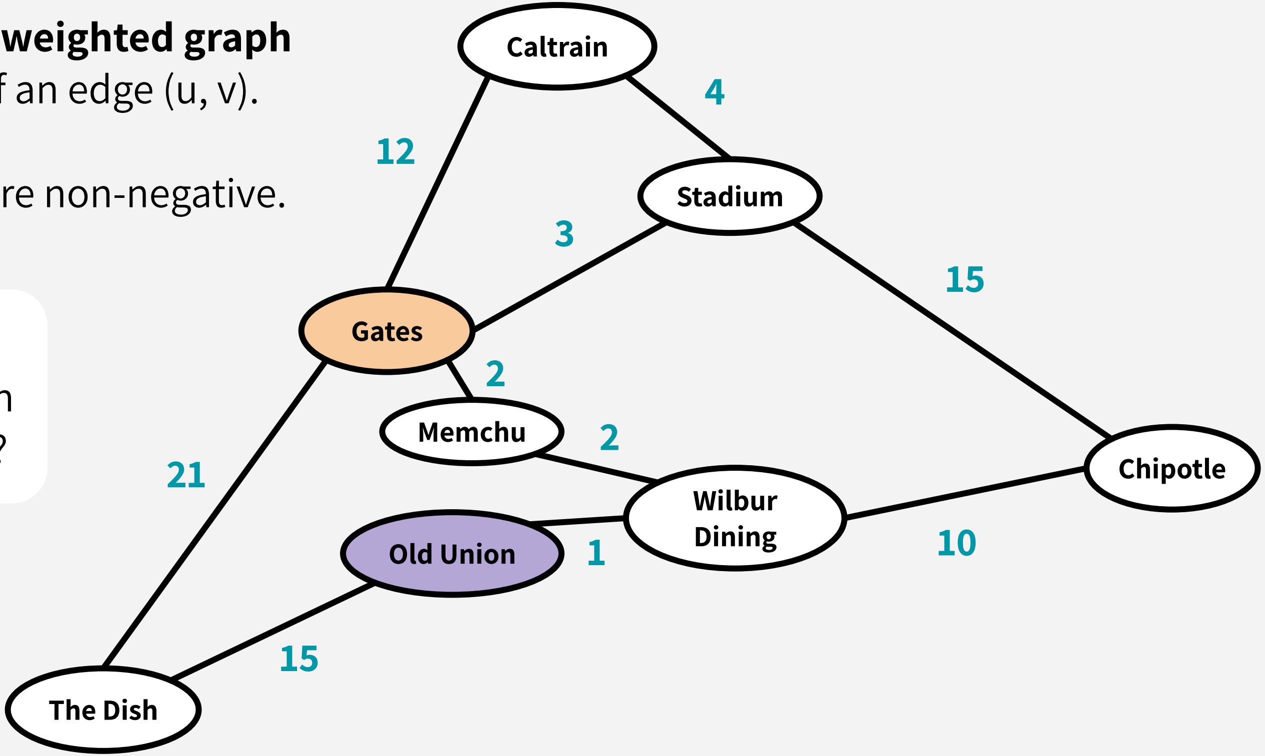
Note: All graphs are directed, but to save the trouble of drawing double arrows everywhere:

SHORTEST PATHS IN WEIGHTED GRAPHS

We can represent this as a **weighted graph** where **w(u,v)** = weight of an edge (u, v).

For today, these weights are non-negative.

What if we wanted to compute the shortest path from **Gates** to **Old Union**?



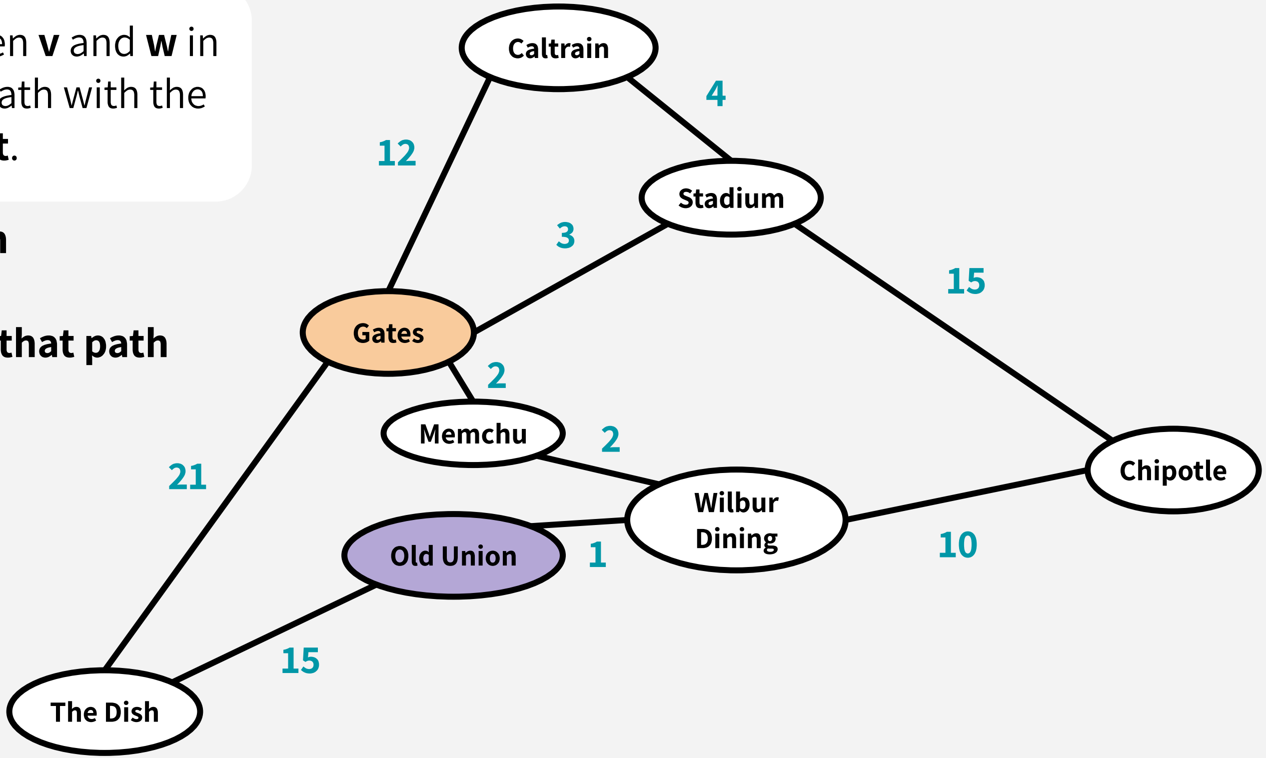
SHORTEST PATHS IN WEIGHTED GRAPHS

The **shortest path** between **v** and **w** in a weighted graph is the path with the **minimum cost**.

Cost of a path

=

sum of weights along that path



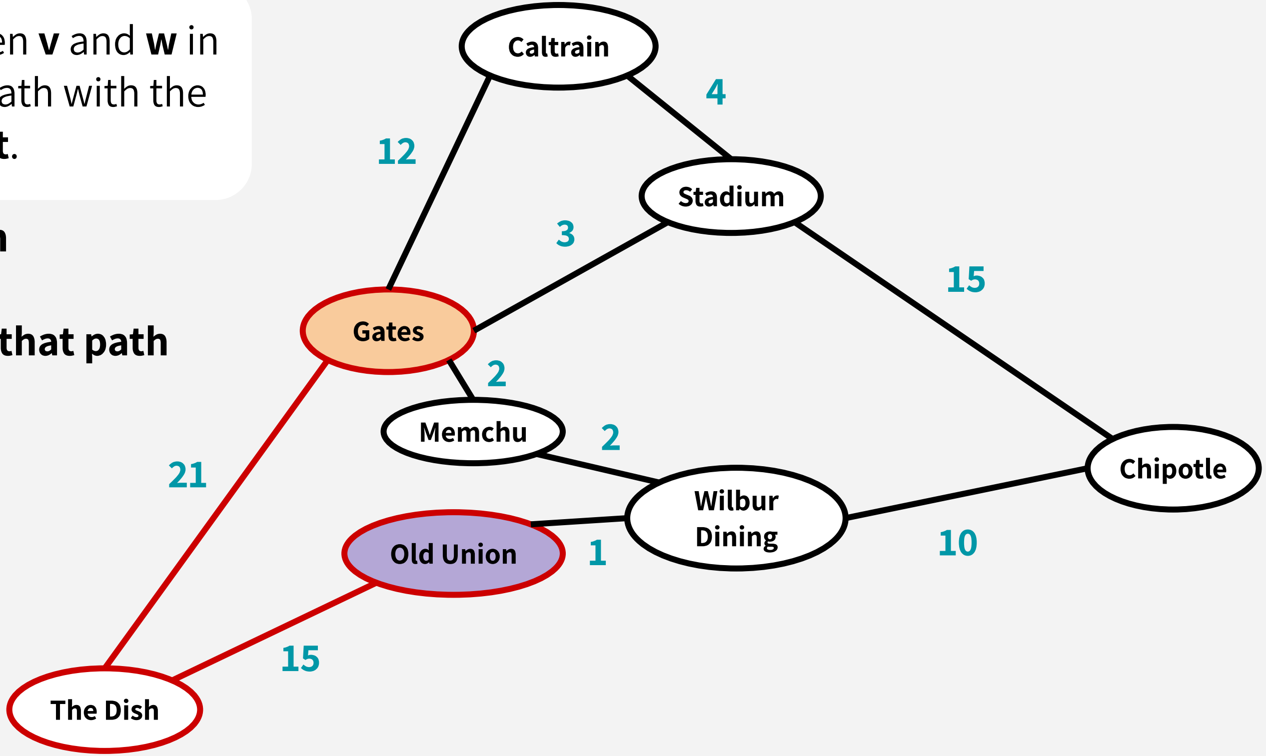
SHORTEST PATHS IN WEIGHTED GRAPHS

The **shortest path** between **v** and **w** in a weighted graph is the path with the **minimum cost**.

Cost of a path

=

sum of weights along that path



This path from Gates to Old Union has cost 36.

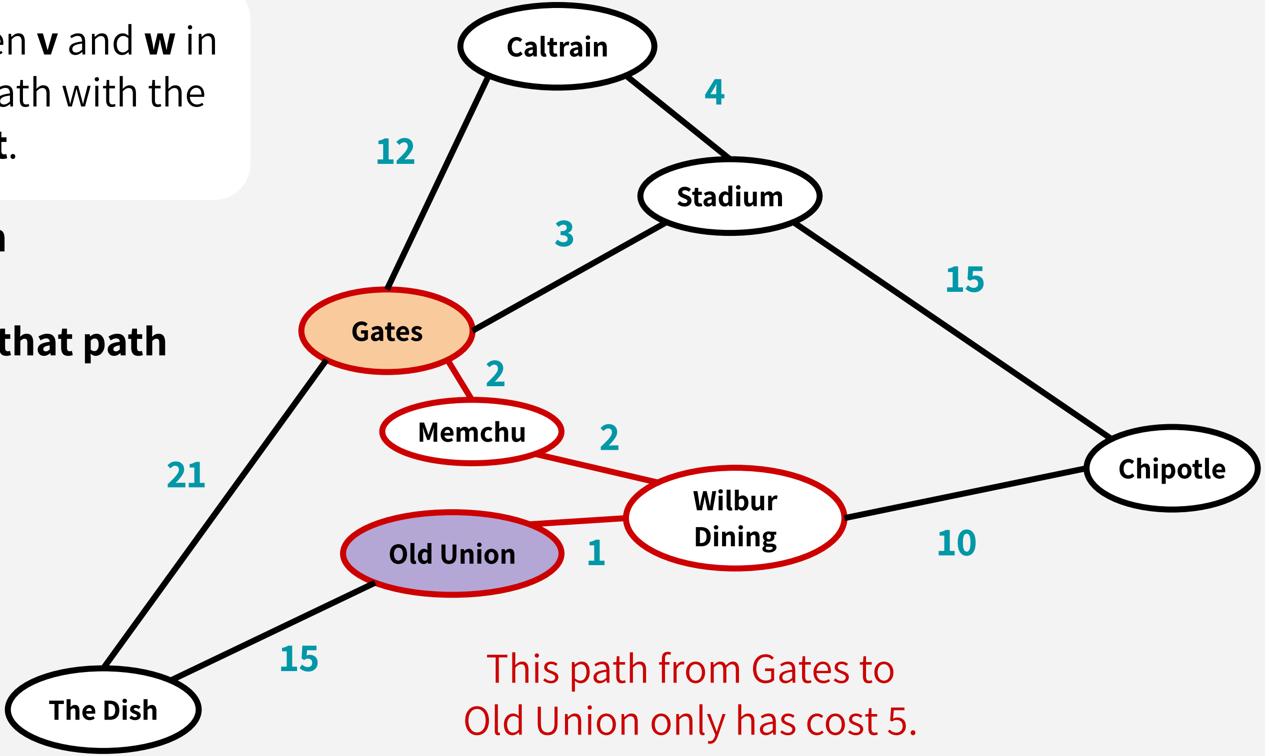
SHORTEST PATHS IN WEIGHTED GRAPHS

The **shortest path** between **v** and **w** in a weighted graph is the path with the **minimum cost**.

Cost of a path

=

sum of weights along that path

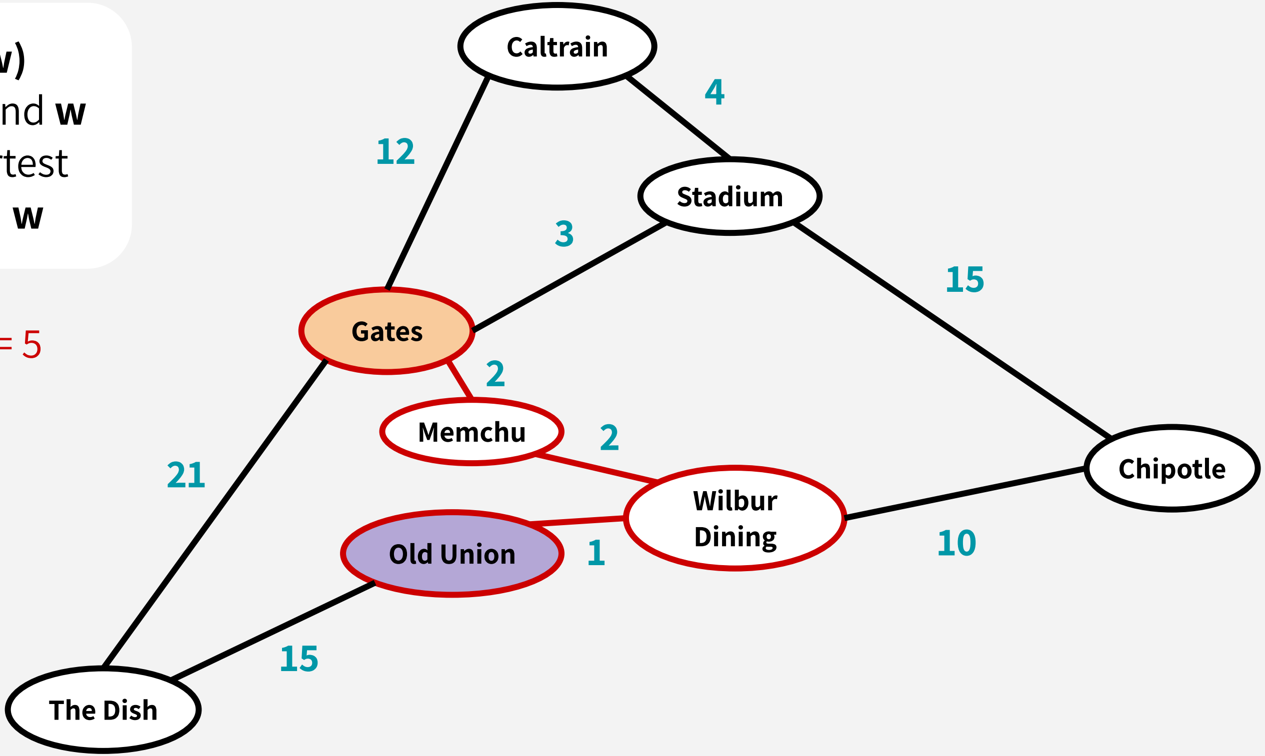


This path from Gates to Old Union only has cost 5.

SHORTEST PATHS IN WEIGHTED GRAPHS

The **distance d(v,w)** between 2 vertices **v** and **w** is the cost of the shortest path between **v** and **w**

d(Gates, Old Union) = 5



SINGLE-SOURCE SHORTEST PATH

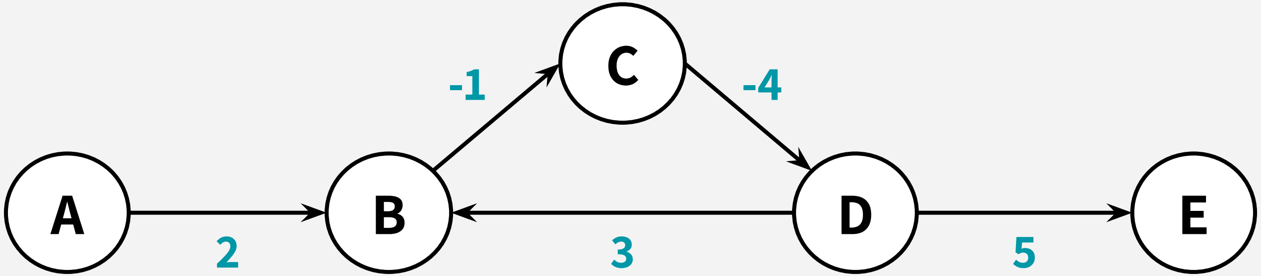
Applications:

Finding the shortest/most efficient path from point A to point B via bike, walking, Uber, Lyft, train, etc.
(Edge weights could be time, money, hassle, effort)

Finding the shortest path to send packets from my computer to some desired server using the Internet
(Edge weights could be link length, traffic, etc.)

NEGATIVE CYCLES

If negative cycles exist in the graph, we'll say *no solution exists*. Why?

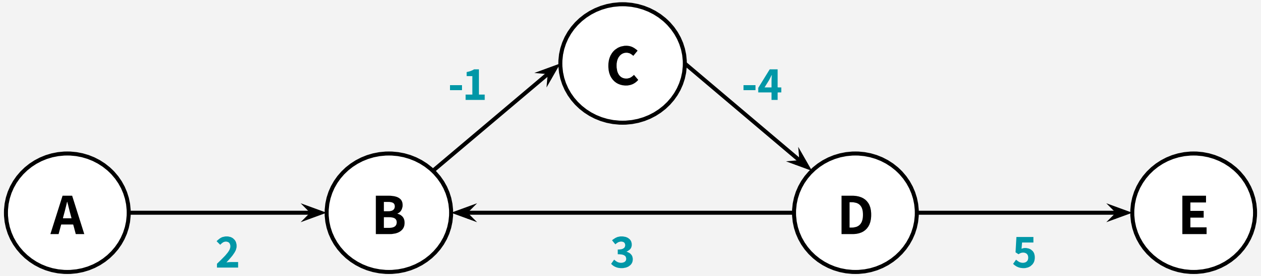


What's the shortest path from A to E?

Is it: A → B → C → D → E? Cost = 2 - 1 - 4 + 5 = **2**.

NEGATIVE CYCLES

If negative cycles exist in the graph, we'll say *no solution exists*. Why?



What's the shortest path from A to E?

Is it: A → B → C → D → E? Cost = 2 - 1 - 4 + 5 = **2**.

Or is it: A → B → C → D → B → C → D → B → C → D → B → C → D → B → C → D → E ?

Basically, shortest paths aren't defined if there are negative cycles!

سوال؟

الگوریتم بلمن-فورد

پیدا کردن کوتاه ترین مسیر از یک راس به تمام رئوس

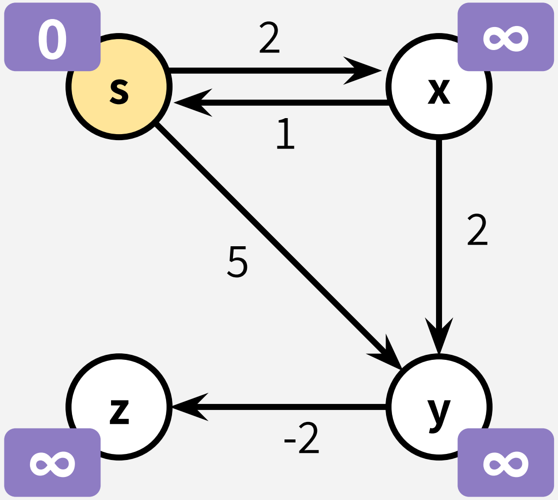
BELLMAN-FORD

We maintain a list $\mathbf{d}^{(k)}$ of length n, for each k = 0, 1, …, n–1.

$\mathbf{d}^{(k)}[\mathbf{b}]$ = the cost of the shortest path from s to b *with at most k edges.*

	s	**x**	**y**	**z**
$\mathbf{d}^{(0)}$	**0**	**∞**	**∞**	**∞**
$\mathbf{d}^{(1)}$				
$\mathbf{d}^{(2)}$				
$\mathbf{d}^{(3)}$				

We know how to fill in $\mathbf{d}^{(0)}$ -- the costs of shortest paths to each vertex with at most **k = 0** edges in it



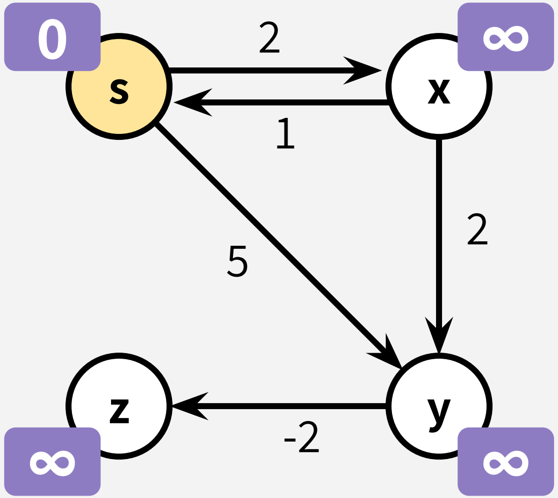
BELLMAN-FORD

We maintain a list $\mathbf{d}^{(k)}$ of length n, for each k = 0, 1, …, n–1.

$\mathbf{d}^{(k)}[\mathbf{b}]$ = the cost of the shortest path from s to b *with at most k edges.*

	s	x	y	z
$\mathbf{d}^{(0)}$	0	∞	∞	∞
$\mathbf{d}^{(1)}$				
$\mathbf{d}^{(2)}$				
$\mathbf{d}^{(3)}$				

We will use table $\mathbf{d}^{(0)}$ to fill in $\mathbf{d}^{(1)}$. More generally, we will use table $\mathbf{d}^{(k-1)}$ to fill in $\mathbf{d}^{(k)}$.



BELLMAN-FORD

We maintain a list $\mathbf{d}^{(k)}$ of length n, for each k = 0, 1, …, n–1.

$\mathbf{d}^{(k)}\mathbf{[b]}$ = the cost of the shortest path from s to b *with at most k edges.*

How do we use $\mathbf{d}^{(0)}$ to update $\mathbf{d}^{(1)}\mathbf{[b]}$?

BELLMAN-FORD

We maintain a list $\mathbf{d}^{(k)}$ of length n, for each k = 0, 1, …, n–1.

$\mathbf{d}^{(k)}[\mathbf{b}]$ = the cost of the shortest path from s to b *with at most k edges.*

How do we use $\mathbf{d}^{(0)}$ to update $\mathbf{d}^{(1)}[\mathbf{b}]$?

Case 1: the shortest path from s to b with at most k edges could be one with at most k–1 edges! In other words, allowing k edges is not going to change anything. Then:

$$\mathbf{d}^{(k)}[\mathbf{b}] = \mathbf{d}^{(k-1)}[\mathbf{b}]$$

BELLMAN-FORD

We maintain a list $\mathbf{d}^{(k)}$ of length n, for each k = 0, 1, …, n–1.

$\mathbf{d}^{(k)}[\mathbf{b}]$ = the cost of the shortest path from s to b *with at most k edges.*

How do we use $\mathbf{d}^{(0)}$ to update $\mathbf{d}^{(1)}[\mathbf{b}]$?

Case 1: the shortest path from s to b with at most k edges could be one with at most k–1 edges! In other words, allowing k edges is not going to change anything. Then:

$$\mathbf{d}^{(k)}[\mathbf{b}] = \mathbf{d}^{(k-1)}[\mathbf{b}]$$

Case 2: the shortest path from s to b with at most k edges could be one with exactly k edges!

BELLMAN-FORD

We maintain a list $\mathbf{d}^{(k)}$ of length n, for each k = 0, 1, …, n–1.

$\mathbf{d}^{(k)}[\mathbf{b}]$ = the cost of the shortest path from s to b *with at most k edges.*

How do we use $\mathbf{d}^{(0)}$ to update $\mathbf{d}^{(1)}[\mathbf{b}]$?

Case 1: the shortest path from s to b with at most k edges could be one with at most k–1 edges! In other words, allowing k edges is not going to change anything. Then:

$$\mathbf{d}^{(k)}[\mathbf{b}] = \mathbf{d}^{(k-1)}[\mathbf{b}]$$

Case 2: the shortest path from s to b with at most k edges could be one with exactly k edges! I.e. this length-k shortest path is [length k–1 shortest path to some incoming neighbor a] + w(a,b). Which of b's incoming neighbors will offer this shortest path? Let's check them all:

$$\mathbf{d}^{(k)}[\mathbf{b}] = \mathbf{min}_{\text{a in b's incoming neighbors}}\{ \mathbf{d}^{(k-1)}[\mathbf{a}] + \mathbf{w(a,b)} \}$$

BELLMAN-FORD PSEUDOCODE

```
BELLMAN_FORD(G,s):
    $d^{(k)}$ = [] for k = 0, ..., n-1
    $d^{(0)}$[v] = ∞ for all v in V (except s)
    $d^{(0)}$[s] = 0
    for k = 1, ..., n-1:
        for b in V:
            $d^{(k)}[b] \leftarrow \min\{ d^{(k-1)}[b], \min_a\{d^{(k-1)}[a] + w(a,b)\} \}$
    return $d^{(n-1)}$
```

BELLMAN-FORD PSEUDOCODE

```
BELLMAN_FORD(G,s):
    $d^{(k)}$ = [] for k = 0, ..., n-1
    $d^{(0)}$[v] = ∞ for all v in V (except s)
    $d^{(0)}$[s] = 0
    for k = 1, ..., n-1:
        for b in V:
            $d^{(k)}[b] \leftarrow \min\{ d^{(k-1)}[b], \min_a\{d^{(k-1)}[a] + w(a,b)\} \}$
    return $d^{(n-1)}$
```

Keeping all n–1 rows is a simplification to make the pseudocode straightforward. In practice, we'd only keep 2 of them at a time!

BELLMAN-FORD PSEUDOCODE

```
BELLMAN_FORD(G,s):
    $d^{(k)}$ = [] for k = 0, ..., n-1
    $d^{(0)}$[v] = ∞ for all v in V (except s)
    $d^{(0)}$[s] = 0
    for k = 1, ..., n-1:
        for b in V:
            $d^{(k)}[b] \leftarrow \min\{ \underbrace{d^{(k-1)}[b]}_{\text{CASE 1}}, \underbrace{\min_a\{d^{(k-1)}[a] + w(a,b)\}}_{\text{CASE 2}} \}$
    return $d^{(n-1)}$
```

Keeping all n–1 rows is a simplification to make the pseudocode straightforward. In practice, we'd only keep 2 of them at a time!

BELLMAN-FORD PSEUDOCODE

```
BELLMAN_FORD(G,s):
    $d^{(k)}$ = [] for k = 0, ..., n-1
    $d^{(0)}$[v] = ∞ for all v in V (except s)
    $d^{(0)}$[s] = 0
    for k = 1, ..., n-1:
        for b in V:
            $d^{(k)}[b] \leftarrow \min\{ d^{(k-1)}[b], \min_a\{d^{(k-1)}[a] + w(a,b)\} \}$
    return $d^{(n-1)}$
```

CASE 1

CASE 2

Keeping all n–1 rows is a simplification to make the pseudocode straightforward. In practice, we'd only keep 2 of them at a time!

Take the minimum over all incoming neighbors a (i.e. all a s.t. (a, b) ∈ E)

This takes O(deg(b))!!!

BELLMAN-FORD PSEUDOCODE

```
BELLMAN_FORD(G,s):
    $d^{(k)}$ = [] for k = 0, ..., n-1
    $d^{(0)}$[v] = ∞ for all v in V (except s)
    $d^{(0)}$[s] = 0
    for k = 1, ..., n-1:
        for b in V:
            $d^{(k)}$[b] ← min{ $d^{(k-1)}$[b], $\min_a$ {$d^{(k-1)}$[a] + w(a,b)} }
    return $d^{(n-1)}$
```

CASE 1

CASE 2

Keeping all n–1 rows is a simplification to make the pseudocode straightforward. In practice, we'd only keep 2 of them at a time!

Take the minimum over all incoming neighbors a (i.e. all a s.t. (a, b) ∈ E)

This takes O(deg(b))!!!

Runtime: O(m·n)

BELLMAN-FORD

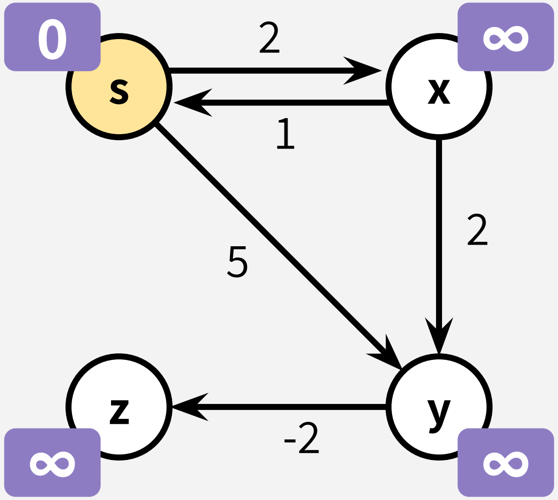
We store a list $\mathbf{d}^{(k)}$ of length n, for each k = 0, 1, …, n–1.

$\mathbf{d}^{(k)}[\mathbf{b}]$ = cost of shortest path from s to b *w/ at most k edges.*

```
for k = 1, ..., n-1:
    for b in V:
        $d^{(k)}[b] \leftarrow \min\{d^{(k-1)}[b], \min_a\{d^{(k-1)}[a] + w(a,b)\}\}$
```

We will use table $\mathbf{d}^{(0)}$ to fill in $\mathbf{d}^{(1)}$. More generally, we will use table $\mathbf{d}^{(k-1)}$ to fill in $\mathbf{d}^{(k)}$.

	s	**x**	**y**	**z**
$\mathbf{d}^{(0)}$	**0**	**∞**	**∞**	**∞**
$\mathbf{d}^{(1)}$				
$\mathbf{d}^{(2)}$				
$\mathbf{d}^{(3)}$				



BELLMAN-FORD

We store a list $\mathbf{d}^{(k)}$ of length n, for each k = 0, 1, …, n–1.

$\mathbf{d}^{(k)}[\mathbf{b}]$ = cost of shortest path from s to b *w/ at most k edges.*

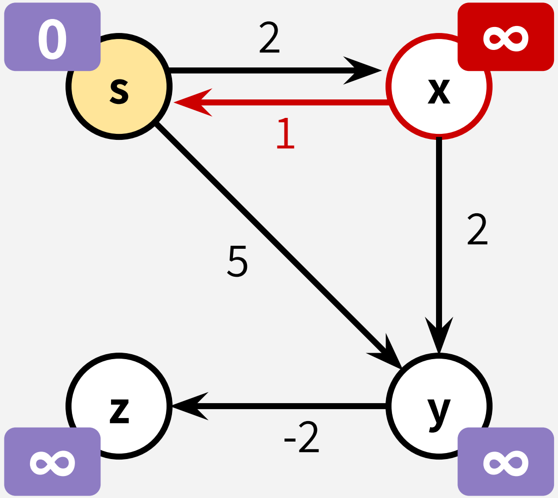
```
for k = 1, ..., n-1:
    for b in V:
        $d^{(k)}[b] \leftarrow \min\{d^{(k-1)}[b], \min_a\{d^{(k-1)}[a] + w(a,b)\}\}$
```

	s	**x**	**y**	**z**
$\mathbf{d}^{(0)}$	0	∞	∞	∞
$\mathbf{d}^{(1)}$	0			
$\mathbf{d}^{(2)}$				
$\mathbf{d}^{(3)}$				

This is min of:

$d^{(0)}[s]$

$d^{(0)}[\mathbf{x}] + w(\mathbf{x}, s)$



BELLMAN-FORD

We store a list $\mathbf{d}^{(k)}$ of length n, for each k = 0, 1, …, n–1.

$\mathbf{d}^{(k)}[\mathbf{b}]$ = cost of shortest path from s to b *w/ at most k edges.*

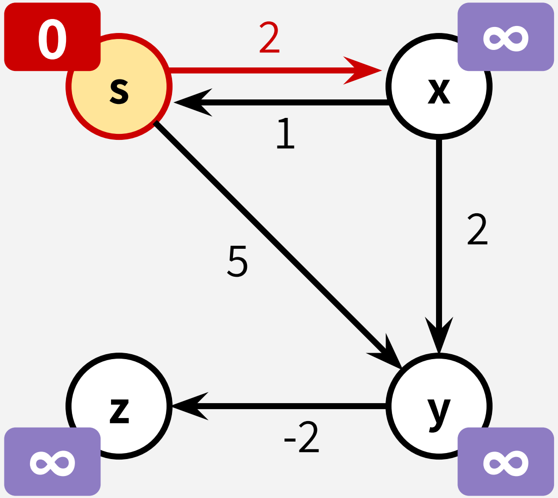
```
for k = 1, ..., n-1:
    for b in V:
        $d^{(k)}[b] \leftarrow \min\{d^{(k-1)}[b], \min_a\{d^{(k-1)}[a] + w(a,b)\}\}$
```

	s	x	y	z
$\mathbf{d}^{(0)}$	0	∞	∞	∞
$\mathbf{d}^{(1)}$	0	2		
$\mathbf{d}^{(2)}$				
$\mathbf{d}^{(3)}$				

This is min of:

$d^{(0)}[x]$

$d^{(0)}[\mathbf{s}] + w(\mathbf{s}, x)$



BELLMAN-FORD

We store a list **d**$^{(k)}$ of length n, for each k = 0, 1, …, n–1.

d$^{(k)}$**[b]** = cost of shortest path from s to b *w/ at most k edges.*

```
for k = 1, ..., n-1:
    for b in V:
        $d^{(k)}[b] \leftarrow \min\{d^{(k-1)}[b],\ \min_a\{d^{(k-1)}[a] + w(a,b)\}\}$
```

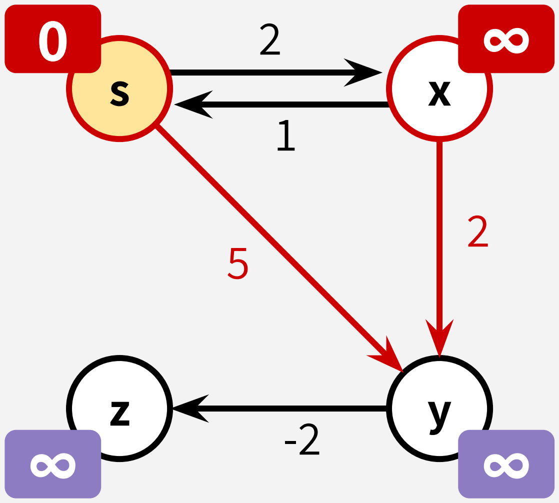
	s	**x**	**y**	**z**
d$^{(0)}$	0	∞	∞	∞
d$^{(1)}$	0	2	5	
d$^{(2)}$				
d$^{(3)}$				

This is min of:

$d^{(0)}[y]$

$d^{(0)}[\mathbf{s}] + w(\mathbf{s}, y)$

$d^{(0)}[\mathbf{x}] + w(\mathbf{x}, y)$



BELLMAN-FORD

We store a list $\mathbf{d}^{(k)}$ of length n, for each k = 0, 1, …, n−1.

$\mathbf{d}^{(k)}[\mathbf{b}]$ = cost of shortest path from s to b *w/ at most k edges.*

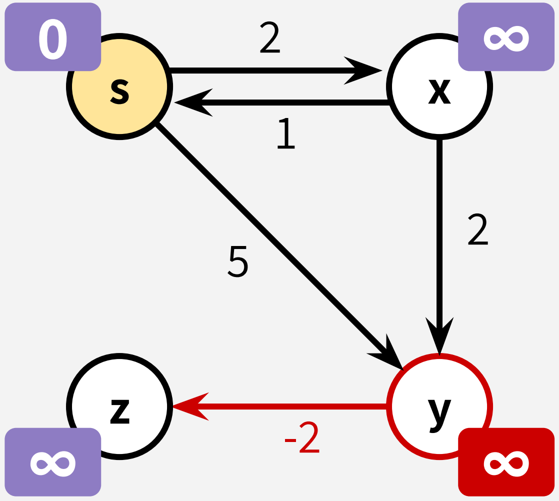
```
for k = 1, ..., n-1:
    for b in V:
        $d^{(k)}[b] \leftarrow \min\{d^{(k-1)}[b],\ \min_a\{d^{(k-1)}[a] + w(a,b)\}\}$
```

	s	x	y	z
$\mathbf{d}^{(0)}$	0	∞	∞	∞
$\mathbf{d}^{(1)}$	0	2	5	∞
$\mathbf{d}^{(2)}$				
$\mathbf{d}^{(3)}$				

This is min of:

$d^{(0)}[z]$

$d^{(0)}[\mathbf{y}] + w(\mathbf{y}, z)$



BELLMAN-FORD

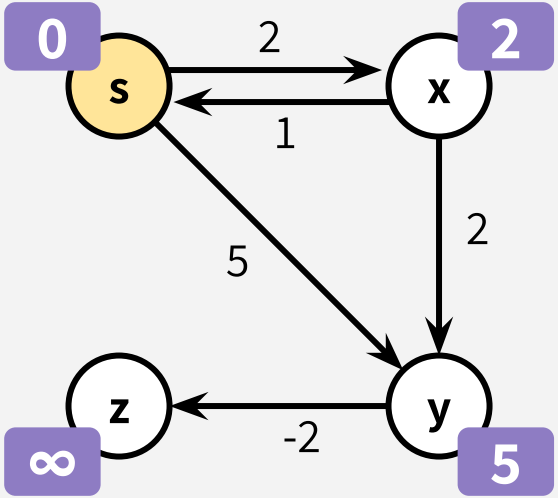
We store a list $\mathbf{d}^{(k)}$ of length n, for each k = 0, 1, …, n–1.

$\mathbf{d}^{(k)}[\mathbf{b}]$ = cost of shortest path from s to b *w/ at most k edges.*

```
for k = 1, ..., n-1:
    for b in V:
        $d^{(k)}[b] \leftarrow \min\{d^{(k-1)}[b], \min_a\{d^{(k-1)}[a] + w(a,b)\}\}$
```

	s	x	y	z
$\mathbf{d}^{(0)}$	0	∞	∞	∞
$\mathbf{d}^{(1)}$	0	2	5	∞
$\mathbf{d}^{(2)}$				
$\mathbf{d}^{(3)}$				

Now, fill in $\mathbf{d}^{(2)}$!!!



BELLMAN-FORD

We store a list $\mathbf{d}^{(k)}$ of length n, for each k = 0, 1, …, n–1.

$\mathbf{d}^{(k)}[\mathbf{b}]$ = cost of shortest path from s to b *w/ at most k edges.*

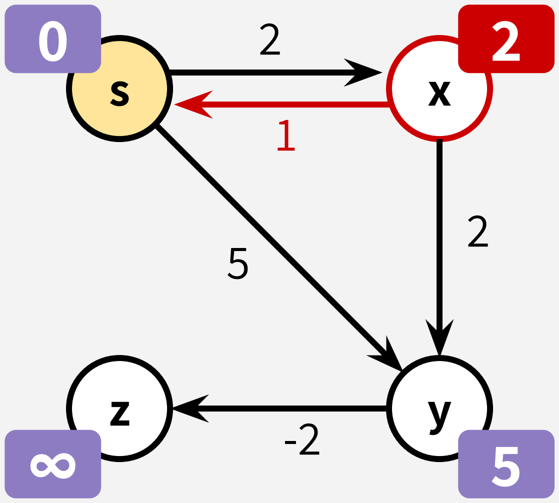
```
for k = 1, ..., n-1:
    for b in V:
        $d^{(k)}[b] \leftarrow \min\{d^{(k-1)}[b],\ \min_a\{d^{(k-1)}[a] + w(a,b)\}\}$
```

	s	**x**	**y**	**z**
$\mathbf{d}^{(0)}$	0	∞	∞	∞
$\mathbf{d}^{(1)}$	0	2	5	∞
$\mathbf{d}^{(2)}$	0			
$\mathbf{d}^{(3)}$				

This is min of:

$d^{(1)}[s]$

$d^{(1)}[\mathbf{x}] + w(\mathbf{x}, s)$



BELLMAN-FORD

We store a list $\mathbf{d^{(k)}}$ of length n, for each k = 0, 1, …, n–1.

$\mathbf{d^{(k)}[b]}$ = cost of shortest path from s to b *w/ at most k edges.*

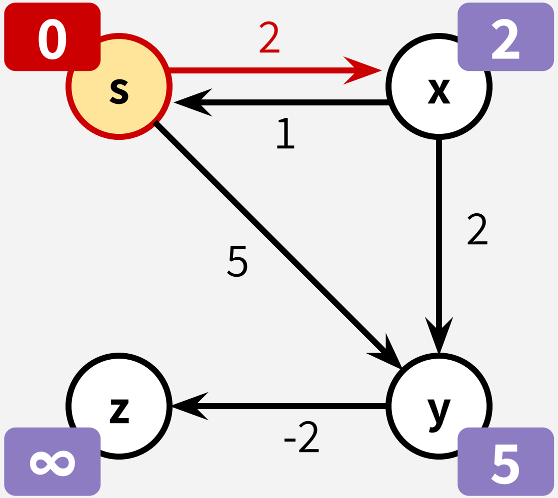
```
for k = 1, ..., n-1:
    for b in V:
        $d^{(k)}[b] \leftarrow \min\{d^{(k-1)}[b], \min_a\{d^{(k-1)}[a] + w(a,b)\}\}$
```

	s	x	y	z
$\mathbf{d^{(0)}}$	0	∞	∞	∞
$\mathbf{d^{(1)}}$	0	2	5	∞
$\mathbf{d^{(2)}}$	0	2		
$\mathbf{d^{(3)}}$				

This is min of:

$d^{(1)}[x]$

$d^{(1)}[\mathbf{s}] + w(\mathbf{s}, x)$



BELLMAN-FORD

We store a list $\mathbf{d}^{(k)}$ of length n, for each k = 0, 1, …, n–1.

$\mathbf{d}^{(k)}[\mathbf{b}]$ = cost of shortest path from s to b *w/ at most k edges.*

```
for k = 1, ..., n-1:
    for b in V:
        $d^{(k)}[b] \leftarrow \min\{d^{(k-1)}[b], \min_a\{d^{(k-1)}[a] + w(a,b)\}\}$
```

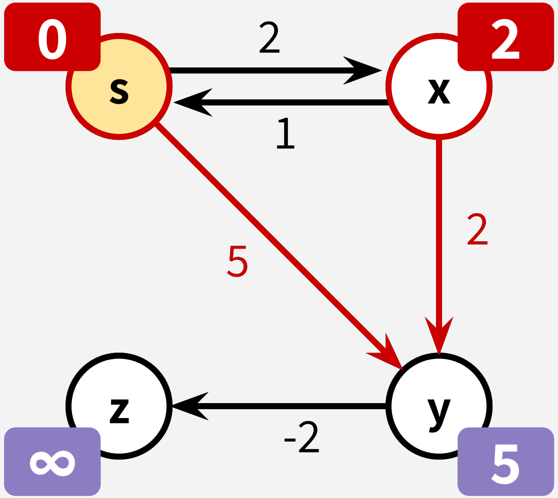
	s	x	y	z
$d^{(0)}$	0	∞	∞	∞
$d^{(1)}$	0	2	5	∞
$d^{(2)}$	0	2	4	
$d^{(3)}$				

This is min of:

$d^{(1)}[y]$

$d^{(1)}[\mathbf{s}] + w(\mathbf{s}, y)$

$d^{(1)}[\mathbf{x}] + w(\mathbf{x}, y)$



BELLMAN-FORD

We store a list $\mathbf{d}^{(k)}$ of length n, for each k = 0, 1, …, n–1.

$\mathbf{d}^{(k)}[\mathbf{b}]$ = cost of shortest path from s to b *w/ at most k edges.*

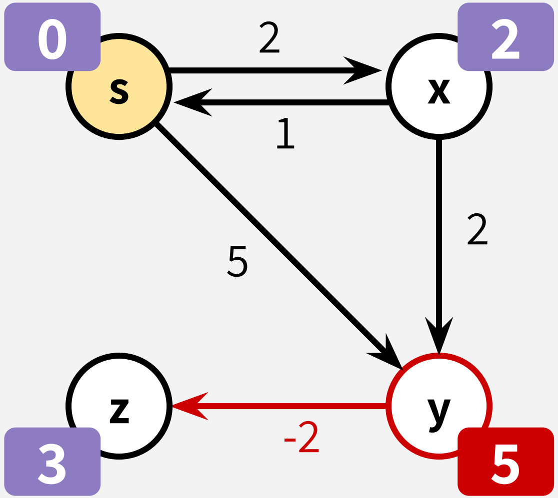
```
for k = 1, ..., n-1:
    for b in V:
        $d^{(k)}[b] \leftarrow \min\{d^{(k-1)}[b], \min_a\{d^{(k-1)}[a] + w(a,b)\}\}$
```

	s	x	y	z
$\mathbf{d}^{(0)}$	0	∞	∞	∞
$\mathbf{d}^{(1)}$	0	2	5	∞
$\mathbf{d}^{(2)}$	0	2	4	3
$\mathbf{d}^{(3)}$				

This is min of:

$d^{(1)}[z]$

$d^{(1)}[\mathbf{y}] + w(\mathbf{y}, z)$



BELLMAN-FORD

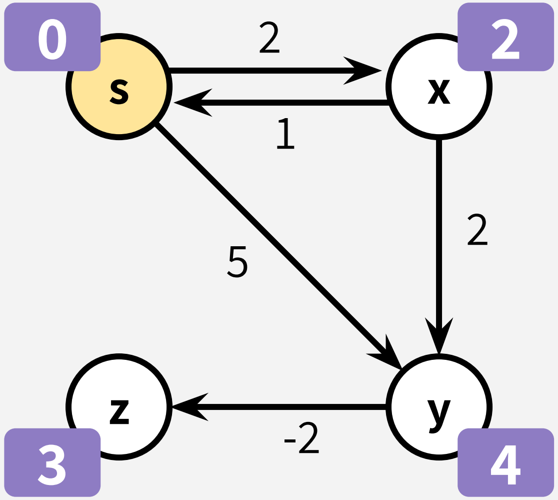
We store a list $\mathbf{d}^{(k)}$ of length n, for each k = 0, 1, …, n–1.

$\mathbf{d}^{(k)}$**[b]** = cost of shortest path from s to b *w/ at most k edges.*

```
for k = 1, ..., n-1:
    for b in V:
        $d^{(k)}[b] \leftarrow \min\{d^{(k-1)}[b], \min_a\{d^{(k-1)}[a] + w(a,b)\}\}$
```

	s	x	y	z
$\mathbf{d}^{(0)}$	0	∞	∞	∞
$\mathbf{d}^{(1)}$	0	2	5	∞
$\mathbf{d}^{(2)}$	0	2	4	3
$\mathbf{d}^{(3)}$				

Now, fill in $\mathbf{d}^{(3)}$!!!



BELLMAN-FORD

We store a list **$d^{(k)}$** of length n, for each k = 0, 1, …, n–1.

$d^{(k)}[b]$ = cost of shortest path from s to b *w/ at most k edges.*

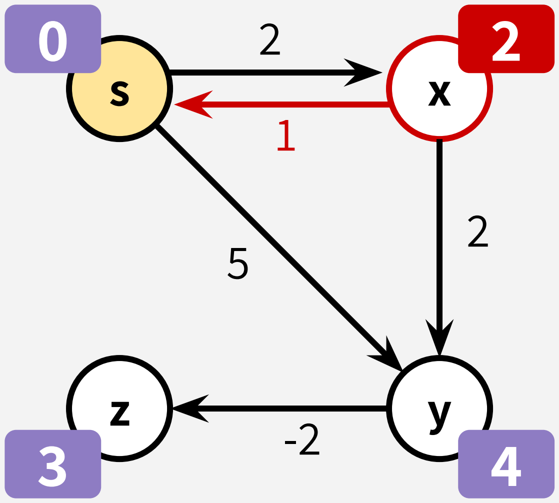
```
for k = 1, ..., n-1:
    for b in V:
        $d^{(k)}[b] \leftarrow \min\{d^{(k-1)}[b],\ \min_a\{d^{(k-1)}[a] + w(a,b)\}\}$
```

	s	x	y	z
$d^{(0)}$	0	∞	∞	∞
$d^{(1)}$	0	2	5	∞
$d^{(2)}$	0	2	4	3
$d^{(3)}$	0			

This is min of:

$d^{(2)}[s]$

$d^{(2)}[\mathbf{x}] + w(\mathbf{x}, s)$



BELLMAN-FORD

We store a list $\mathbf{d}^{(k)}$ of length n, for each k = 0, 1, …, n–1.

$\mathbf{d}^{(k)}[\mathbf{b}]$ = cost of shortest path from s to b *w/ at most k edges.*

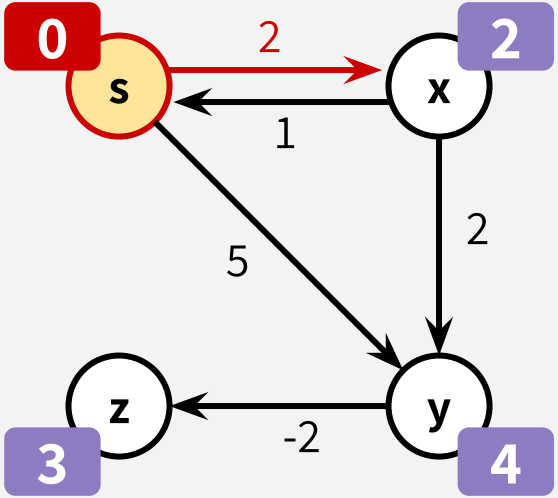
```
for k = 1, ..., n-1:
    for b in V:
        $d^{(k)}[b] \leftarrow \min\{d^{(k-1)}[b], \min_a\{d^{(k-1)}[a] + w(a,b)\}\}$
```

	s	x	y	z
$\mathbf{d}^{(0)}$	0	∞	∞	∞
$\mathbf{d}^{(1)}$	0	2	5	∞
$\mathbf{d}^{(2)}$	0	2	4	3
$\mathbf{d}^{(3)}$	0	2		

This is min of:

$d^{(2)}[x]$

$d^{(2)}[\mathbf{s}] + w(\mathbf{s}, x)$



BELLMAN-FORD

We store a list $\mathbf{d}^{(k)}$ of length n, for each k = 0, 1, …, n–1.

$\mathbf{d}^{(k)}[\mathbf{b}]$ = cost of shortest path from s to b *w/ at most k edges.*

```
for k = 1, ..., n-1:
    for b in V:
        $d^{(k)}[b] \leftarrow \min\{d^{(k-1)}[b], \min_a\{d^{(k-1)}[a] + w(a,b)\}\}$
```

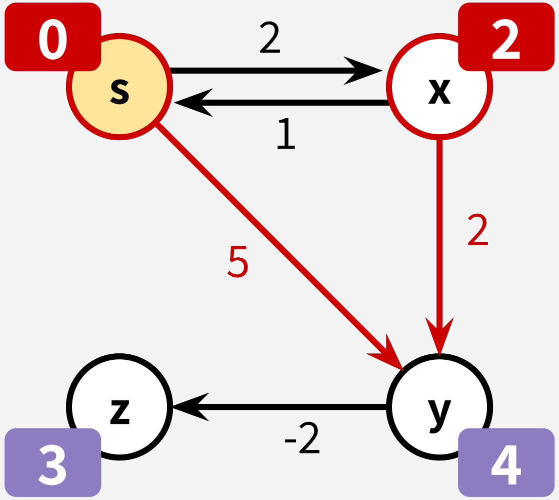
	s	x	y	z
$d^{(0)}$	0	∞	∞	∞
$d^{(1)}$	0	2	5	∞
$d^{(2)}$	0	2	4	3
$d^{(3)}$	0	2	4	

This is min of:

$d^{(2)}[y]$

$d^{(2)}[\mathbf{s}] + w(\mathbf{s}, y)$

$d^{(2)}[\mathbf{x}] + w(\mathbf{x}, y)$



BELLMAN-FORD

We store a list $\mathbf{d}^{(k)}$ of length n, for each k = 0, 1, …, n–1.

$\mathbf{d}^{(k)}[\mathbf{b}]$ = cost of shortest path from s to b *w/ at most k edges.*

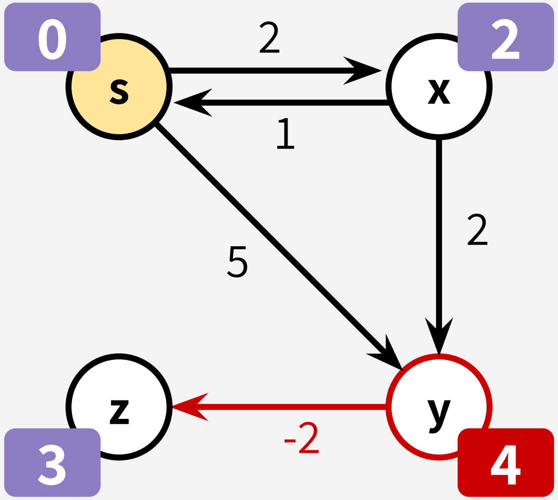
```
for k = 1, ..., n-1:
    for b in V:
        $d^{(k)}[b] \leftarrow \min\{d^{(k-1)}[b], \min_a\{d^{(k-1)}[a] + w(a,b)\}\}$
```

	s	x	y	z
$\mathbf{d}^{(0)}$	0	∞	∞	∞
$\mathbf{d}^{(1)}$	0	2	5	∞
$\mathbf{d}^{(2)}$	0	2	4	3
$\mathbf{d}^{(3)}$	0	2	4	2

This is min of:

$d^{(2)}[z]$

$d^{(2)}[\mathbf{y}] + w(\mathbf{y}, z)$



BELLMAN-FORD

We store a list $\mathbf{d}^{(k)}$ of length n, for each k = 0, 1, …, n–1.

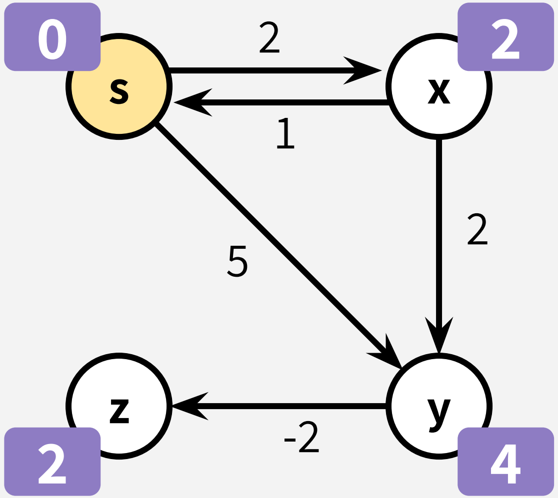
$\mathbf{d}^{(k)}[\mathbf{b}]$ = cost of shortest path from s to b *w/ at most k edges.*

```
for k = 1, ..., n-1:
    for b in V:
        $d^{(k)}[b]$ ← min{$d^{(k-1)}[b]$, $\min_a\{d^{(k-1)}[a] + w(a,b)\}$}
```

	s	x	y	z
$d^{(0)}$	0	∞	∞	∞
$d^{(1)}$	0	2	5	∞
$d^{(2)}$	0	2	4	3
$d^{(3)}$	0	2	4	2

We're done!

We can double check the entry for **z** in each $d^{(i)}$:



BELLMAN-FORD

We store a list $\mathbf{d}^{(k)}$ of length n, for each k = 0, 1, …, n–1.

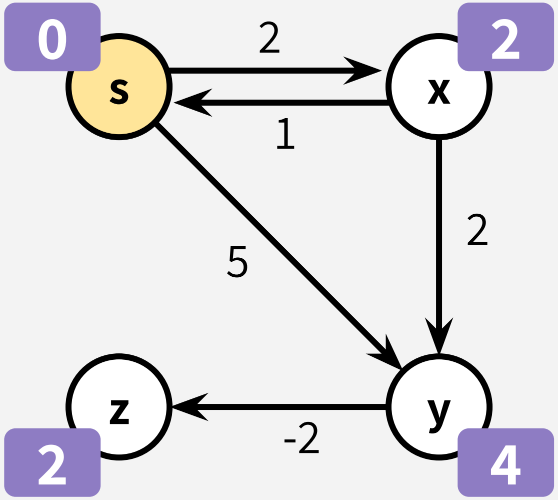
$\mathbf{d}^{(k)}[\mathbf{b}]$ = cost of shortest path from s to b *w/ at most k edges.*

```
for k = 1, ..., n-1:
    for b in V:
        $d^{(k)}[b] \leftarrow \min\{d^{(k-1)}[b], \min_a\{d^{(k-1)}[a] + w(a,b)\}\}$
```

Just to double check:

cost of shortest path from s to z with **1** edge = **∞**

	s	x	y	z
$\mathbf{d}^{(0)}$	0	∞	∞	∞
$\mathbf{d}^{(1)}$	0	2	5	∞
$\mathbf{d}^{(2)}$	0	2	4	3
$\mathbf{d}^{(3)}$	0	2	4	2



BELLMAN-FORD

We store a list $\mathbf{d}^{(k)}$ of length n, for each k = 0, 1, …, n–1.

$\mathbf{d}^{(k)}\mathbf{[b]}$ = cost of shortest path from s to b *w/ at most k edges.*

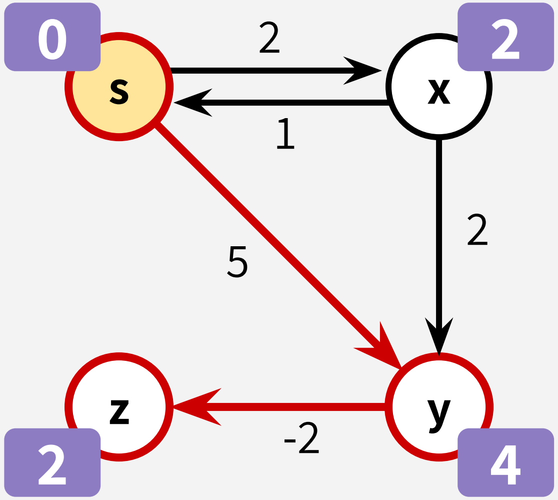
```
for k = 1, ..., n-1:
    for b in V:
        $d^{(k)}[b] \leftarrow \min\{d^{(k-1)}[b], \min_a\{d^{(k-1)}[a] + w(a,b)\}\}$
```

Just to double check:

cost of shortest path from s to z with **1** edge = **∞**

cost of shortest path from s to z with **2** edges = **3**

	s	x	y	z
$d^{(0)}$	0	∞	∞	∞
$d^{(1)}$	0	2	5	∞
$d^{(2)}$	0	2	4	3
$d^{(3)}$	0	2	4	2



BELLMAN-FORD

We store a list $\mathbf{d}^{(k)}$ of length n, for each k = 0, 1, …, n–1.

$\mathbf{d}^{(k)}[\mathbf{b}]$ = cost of shortest path from s to b *w/ at most k edges.*

```
for k = 1, ..., n-1:
    for b in V:
        $d^{(k)}[b] \leftarrow \min\{d^{(k-1)}[b], \min_a\{d^{(k-1)}[a] + w(a,b)\}\}$
```

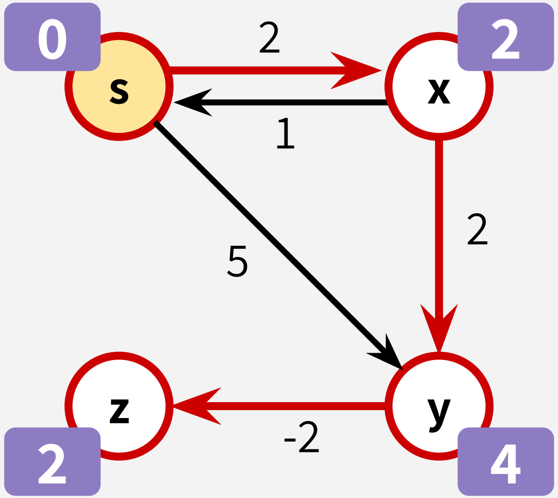
Just to double check:

	s	x	y	z
$d^{(0)}$	0	∞	∞	∞
$d^{(1)}$	0	2	5	∞
$d^{(2)}$	0	2	4	3
$d^{(3)}$	0	2	4	2

cost of shortest path from s to z with **1** edge = **∞**

cost of shortest path from s to z with **2** edges = **3**

cost of shortest path from s to z with **3** edges = **2**



سوال؟

پیاده سازی دیگری از الگوریتم بلمن-فورد

پیدا کردن کوتاه ترین مسیر از یک راس به تمام رئوس

DYNAMIC PROGRAMMING

Two approaches for DP

(2 different ways to think about and/or implement DP algorithms)

Bottom-up: iterates through problems by size and solves the small problems first (kind of like taking care of base cases first & building up).
e.g. **Bellman-Ford (as we will see shortly!) computes $d^{(0)}$, then $d^{(1)}$, then $d^{(2)}$, etc.**

Top-down: instead uses recursive calls to solve smaller problems, while using memoization/caching to keep track of small problems that you've already computed answers for (simply fetch the answer instead of re-solving that problem and waste computational effort)
We will see a way later to implement **Bellman-Ford** using a top-down approach.

TOP-DOWN BELLMAN-FORD

```
RECURSIVE_BELLMAN_FORD(G,s):
    $d^{(k)}$ = [None] * n for k = 0, ..., n-1
    $d^{(0)}$[v] = ∞ for all v in V (except s)
    $d^{(0)}$[s] = 0
    for b in V:
        $d^{(n-1)}$[b] ← RECURSIVE_BF_HELPER(G, b, n-1)

RECURSIVE_BF_HELPER(G, b, k):
    A = {a such that (a,b) in E} ∪ {b}    // b's in-neighbors
    for a in A:
        if $d^{(k-1)}$[a] is None:                    // not yet solved
            $d^{(k-1)}$[a] ← RECURSIVE_BF_HELPER(G, a, k-1)
    return min{ $d^{(k-1)}$[b], $\min_a\{d^{(k-1)}[a] + w(a,b)\}$ }
```

TOP-DOWN BELLMAN-FORD

```
RECURSIVE_BELLMAN_FORD(G,s):
    $d^{(k)}$ = [None] * n for k = 0, ..., n-1
    $d^{(0)}$[v] = ∞ for all v in V (except s)
    $d^{(0)}$[s] = 0
    for b in V:
        $d^{(n-1)}$[b] ← RECURSIVE_BF_HELPER(G, b, n-1)

RECURSIVE_BF_HELPER(G, b, k):
    A = {a such that (a,b) in E} ∪ {b}    // b's in-neighbors
    for a in A:
        if $d^{(k-1)}$[a] is None:                 // not yet solved
            $d^{(k-1)}$[a] ← RECURSIVE_BF_HELPTER(G, a, k-1)
    return min{ $d^{(k-1)}$[b], $\min_a\{d^{(k-1)}[a] + w(a,b)\}$ }
```

Think of this as a table/cache that holds the computed answers of our subproblems.

if the answer to this subproblem hasn't been computed yet, then we'll first solve it! It immediately gets saved in our cache, so we won't ever solve it twice.

TOP-DOWN BELLMAN-FORD

```
RECURSIVE_BELLMAN_FORD(G,s):
    $d^{(k)}$ = [None] * n for k = 0, ..., n-1
    $d^{(0)}$[v] = ∞ for all v in V (except s)
    $d^{(0)}$[s] = 0
    for b in V:
        $d^{(n-1)}$[b] ← RECURSIVE_BF_HELPER(G, b, n-1)

RECURSIVE_BF_HELPER(G, b, k):
    A = {a such that (a,b) in E} ∪ {b}    // b's in-neighbors
    for a in A:
        if $d^{(k-1)}$[a] is None:                  // not yet solved
            $d^{(k-1)}$[a] ← RECURSIVE_BF_HELPTER(G, a, k-1)
    return min{ $d^{(k-1)}$[b], $\min_a$\{$d^{(k-1)}$[a] + w(a,b)} }
```

Think of this as a table/cache that holds the computed answers of our subproblems.

if the answer to this subproblem hasn't been computed yet, then we'll first solve it! It immediately gets saved in our cache, so we won't ever solve it twice.

Runtime: O(m·n)

الگوریتم فلوید-وارشال

پیدا کردن کوتاه ترین مسیر بین هر دو راس

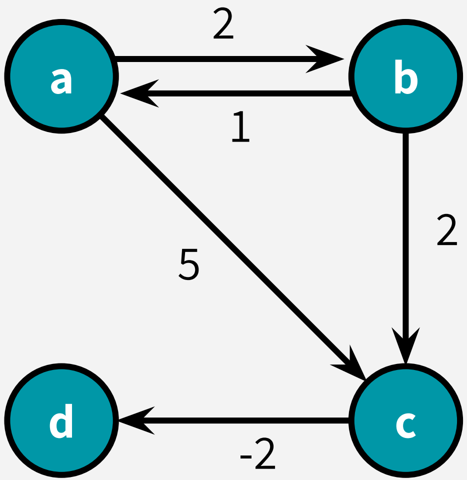
ALL-PAIRS SHORTEST PATHS (APSP)

Find the shortest paths from **v** to **w** for ALL pairs **v**, **w** of vertices in the graph (not just shortest paths from a special single source **s**)

Destination

Source

	a	**b**	**c**	**d**
a	**0**	**2**	**4**	**2**
b	**1**	**0**	**2**	**0**
c	**∞**	**∞**	**0**	**-2**
d	**∞**	**∞**	**∞**	**0**



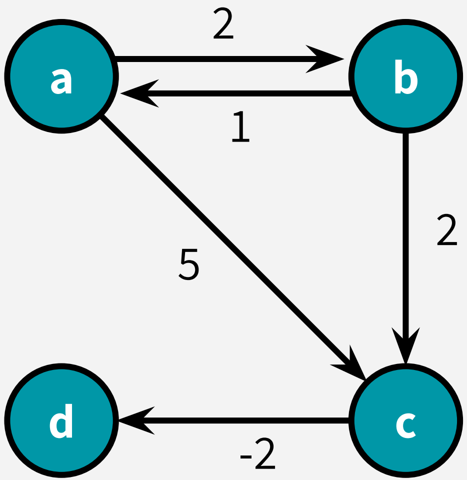
ALL-PAIRS SHORTEST PATHS (APSP)

Find the shortest paths from **v** to **w** for ALL pairs **v**, **w** of vertices in the graph (not just shortest paths from a special single source **s**)

Destination

Source

	a	b	c	d
a	0	2	4	2
b	1	0	2	0
c	∞	∞	0	-2
d	∞	∞	∞	0



What's a naive algorithm?

ALL-PAIRS SHORTEST PATHS (APSP)

Find the shortest paths from **v** to **w** for ALL pairs **v, w** of vertices in the graph

Naive algorithm (if we want to handle negative edge weights):

```
For all s in G:
    Run Bellman-Ford on G starting at s
```

Runtime: $O(n \cdot mn)$ = $\mathbf{O(mn^2)}$… this may be as bad as n^4 if $m = n^2$

Can we do better?

What's a naive algorithm?

FLOYD-WARSHALL: A DP APPROACH

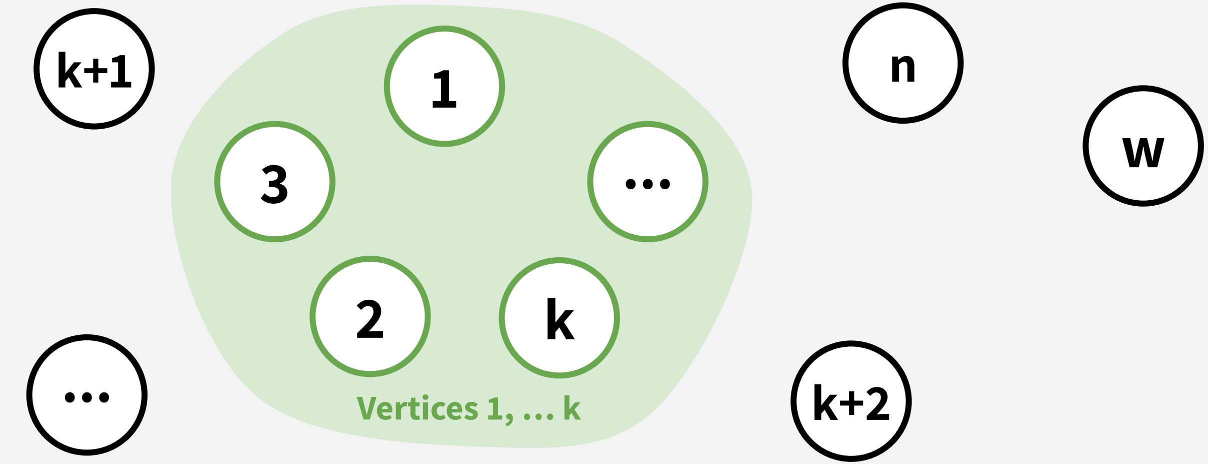
We need to define the optimal substructure: Figure out what your subproblems are, and how you'll express an optimal solution in terms of optimal solutions to subproblems.

FLOYD-WARSHALL: A DP APPROACH

We need to define the optimal substructure: Figure out what your subproblems are, and how you'll express an optimal solution in terms of optimal solutions to subproblems.

Subproblem(k): for all pairs **v**, **w**, find the cost of the shortest path from **v** to **w** so that all the internal vertices on that path are in {1, …, k}

Let $\mathbf{D}^{(k)}[\mathbf{v}, \mathbf{w}]$ be the solution to Subproblem(k)



Assign each vertex a number in {1, 2…, n}

(This diagram omits many edges and nodes, so it's not a full example)

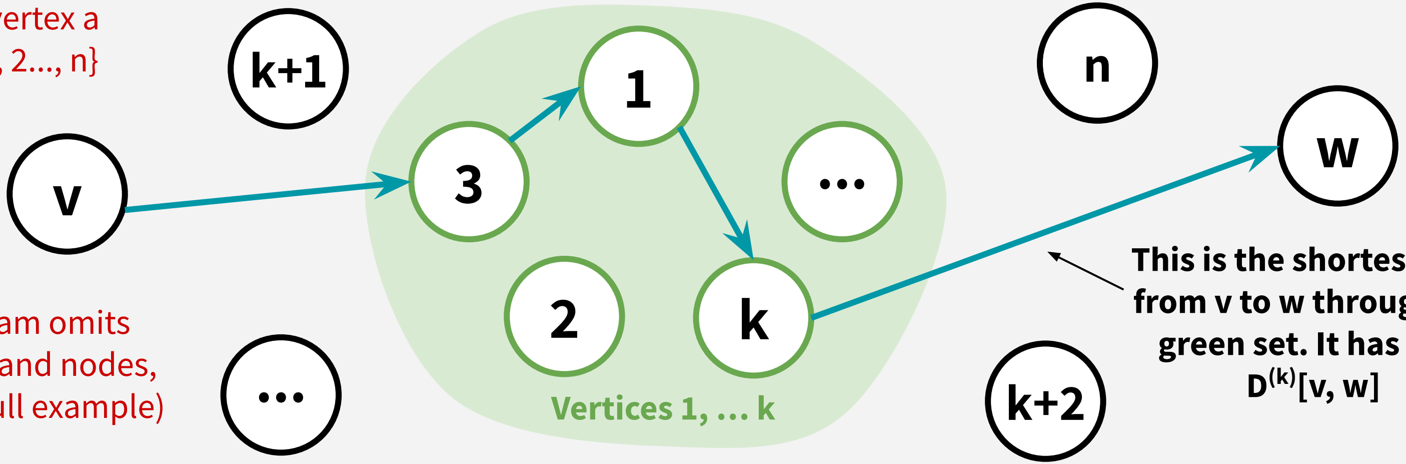
FLOYD-WARSHALL: A DP APPROACH

We need to define the optimal substructure: Figure out what your subproblems are, and how you'll express an optimal solution in terms of optimal solutions to subproblems.

Subproblem(k): for all pairs **v**, **w**, find the cost of the shortest path from **v** to **w** so that all the internal vertices on that path are in {1, …, k}

Let $\mathbf{D^{(k)}[v, w]}$ be the solution to Subproblem(k)

Assign each vertex a number in {1, 2…, n}



(This diagram omits many edges and nodes, so it's not a full example)

This is the shortest path from v to w through the green set. It has cost $D^{(k)}[v, w]$

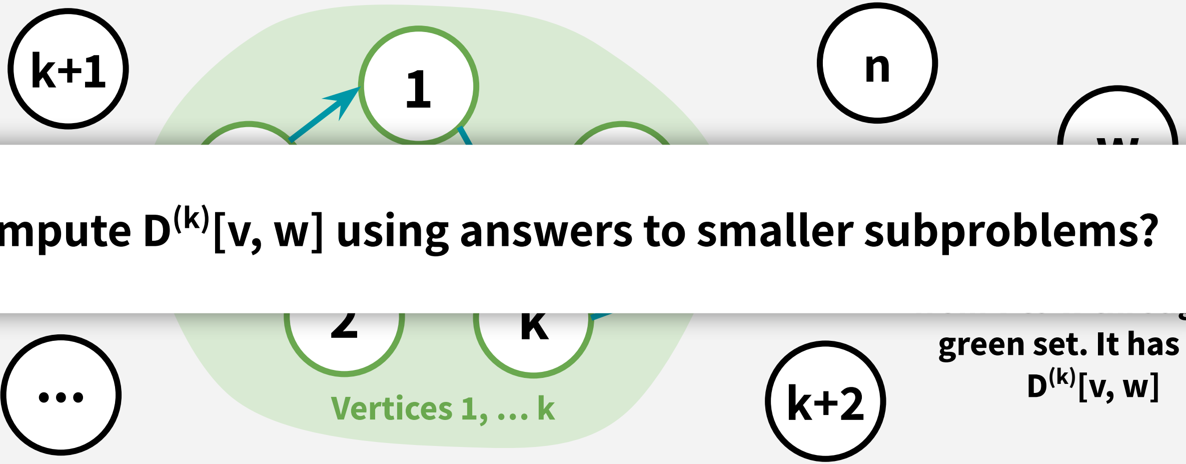
FLOYD-WARSHALL: A DP APPROACH

We need to define the optimal substructure: Figure out what your subproblems are, and how you'll express an optimal solution in terms of optimal solutions to subproblems.

Subproblem(k): for all pairs **v**, **w**, find the cost of the shortest path from **v** to **w** so that all the internal vertices on that path are in {1, …, k}

Let **$D^{(k)}$[v, w]** be the solution to Subproblem(k)

Assign each vertex a number in {1, 2..., n}



How do I compute $D^{(k)}$[v, w] using answers to smaller subproblems?

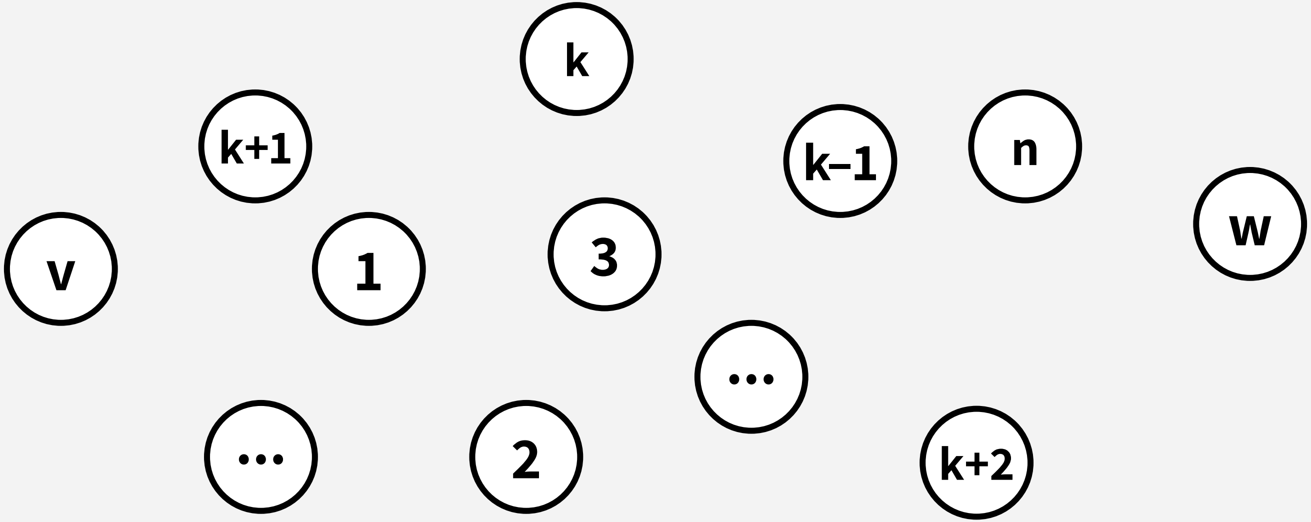
ath
the
green set. It has cost $D^{(k)}$[v, w]

(This diagram omits many edges and nodes, so it's not a full example)

FLOYD-WARSHALL: A DP APPROACH

$D^{(k)}$[v, w] is the cost of the shortest path from **v** to **w**, s.t. all of the internal vertices on the path are in the set of vertices {1, …, k}.

Two cases to consider: vertex k *is not* included in that path, or it *is.*

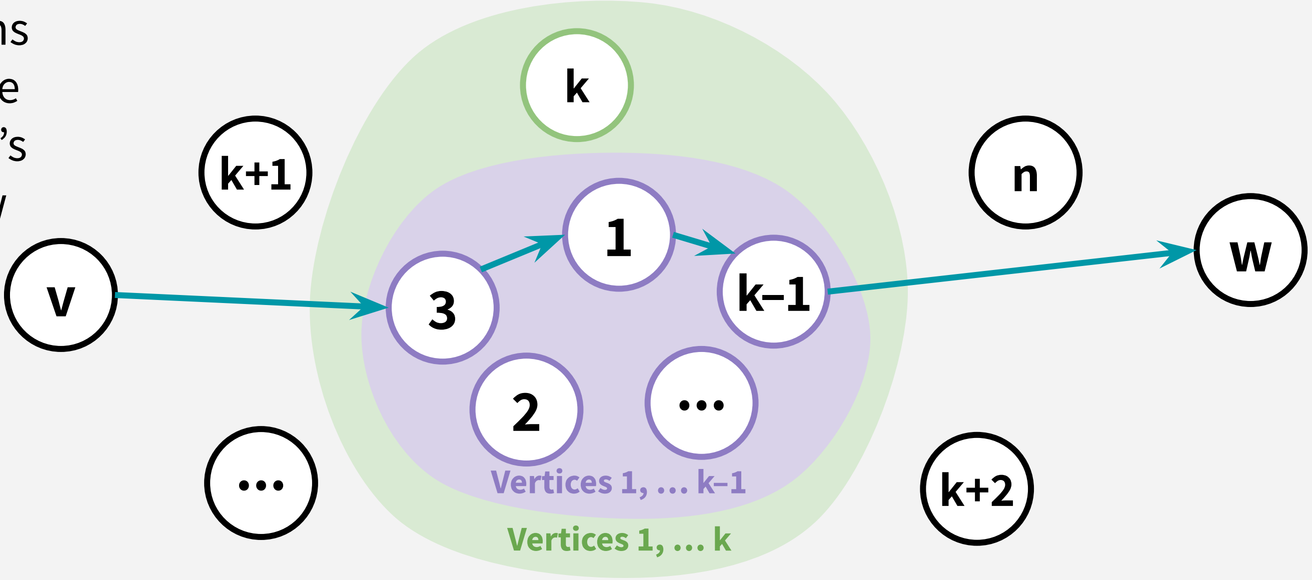


FLOYD-WARSHALL: A DP APPROACH

$D^{(k)}$**[v, w]** = cost of the shortest path from **v** to **w**, s.t. all the internal vertices on the path are in the set of vertices {1,…, k}.

CASE 1: We don't need vertex k! So, $D^{(k)}[v, w] = D^{(k-1)}[v, w]$

In this case, this means that **this path** was the shortest before *and* it's still the shortest now

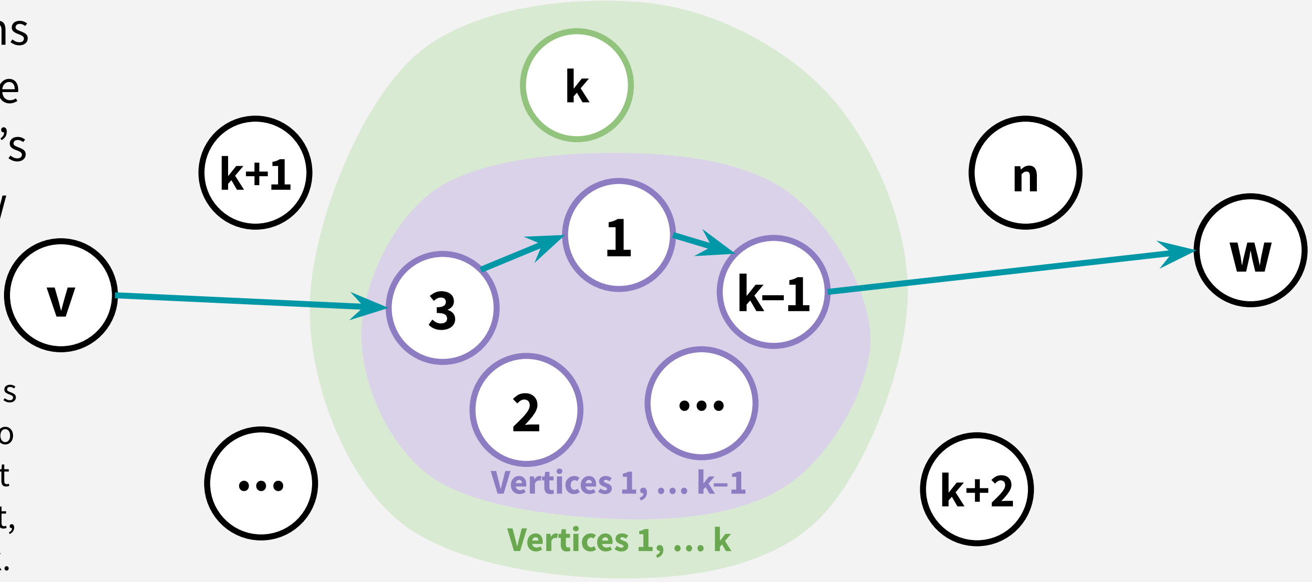


FLOYD-WARSHALL: A DP APPROACH

$D^{(k)}[v, w]$ = cost of the shortest path from **v** to **w**, s.t. all the internal vertices on the path are in the set of vertices {1,…, k}.

CASE 1: We don't need vertex k! So, $D^{(k)}[v, w] = D^{(k-1)}[v, w]$

In this case, this means that **this path** was the shortest before *and* it's still the shortest now



In other words, allowing paths to go through k (in addition to nodes 1, …, k–1) now doesn't change the shortest path cost, since it doesn't need to use k.

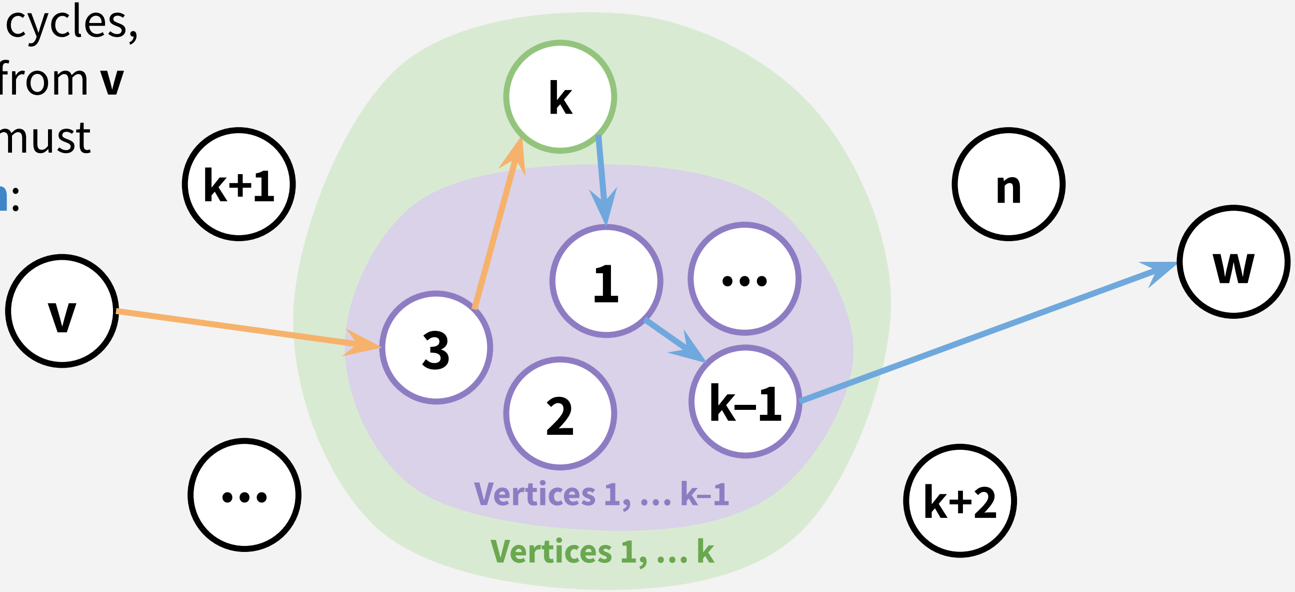
FLOYD-WARSHALL: A DP APPROACH

$D^{(k)}[v, w]$ = cost of the shortest path from **v** to **w**, s.t. all the internal vertices on the path are in the set of vertices {1,…, k}.

CASE 2: We need vertex k! So, $D^{(k)}[v, w] = D^{(k-1)}[v, k] + D^{(k-1)}[k, w]$

If there are no negative cycles, then the shortest path from **v** to **w** is *simple*, and it must look like **this path**:

(we also know that neither of these subpaths contains nodes greater than k–1.)



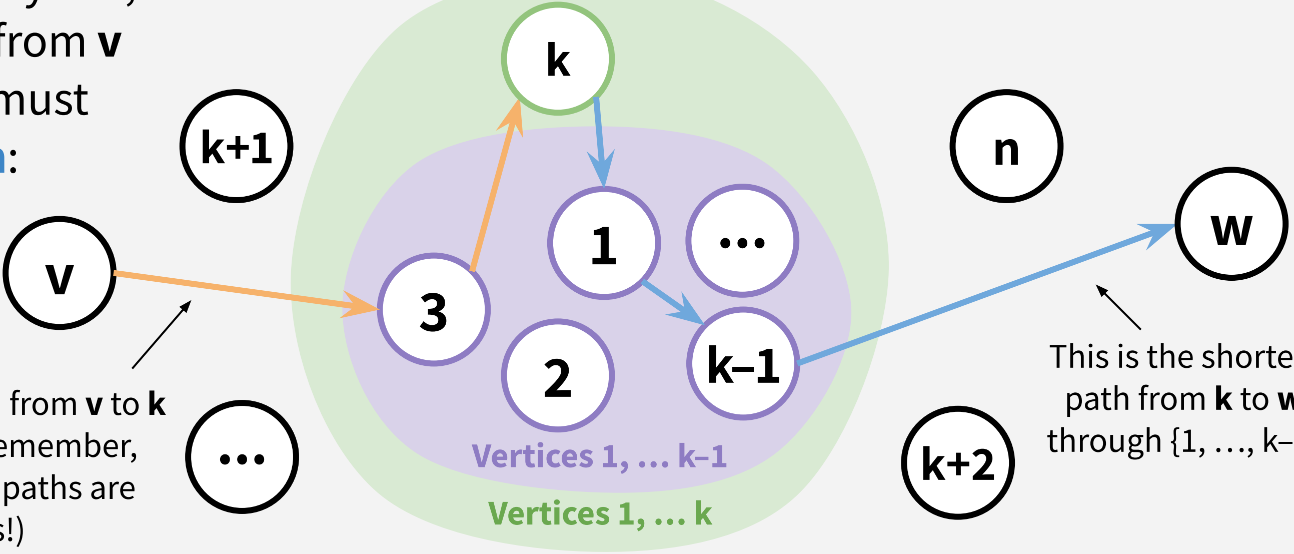
FLOYD-WARSHALL: A DP APPROACH

$D^{(k)}[v, w]$ = cost of the shortest path from **v** to **w**, s.t. all the internal vertices on the path are in the set of vertices {1,…, k}.

CASE 2: We need vertex k! So, $D^{(k)}[v, w] = D^{(k-1)}[v, k] + D^{(k-1)}[k, w]$

If there are no negative cycles, then the shortest path from **v** to **w** is *simple*, and it must look like **this path**:

(we also know that neither of these subpaths contains nodes greater than k–1.)



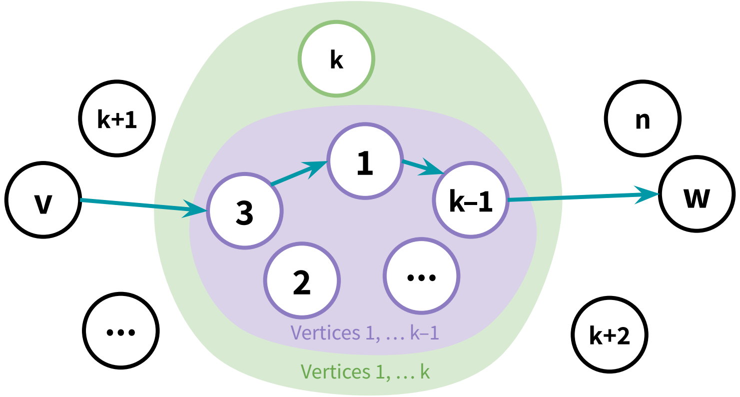
This is the shortest path from **v** to **k** through {1, …, k–1} (remember, sub-paths of shortest paths are shortest paths!)

This is the shortest path from **k** to **w** through {1, …, k–1}

FLOYD-WARSHALL: A DP APPROACH

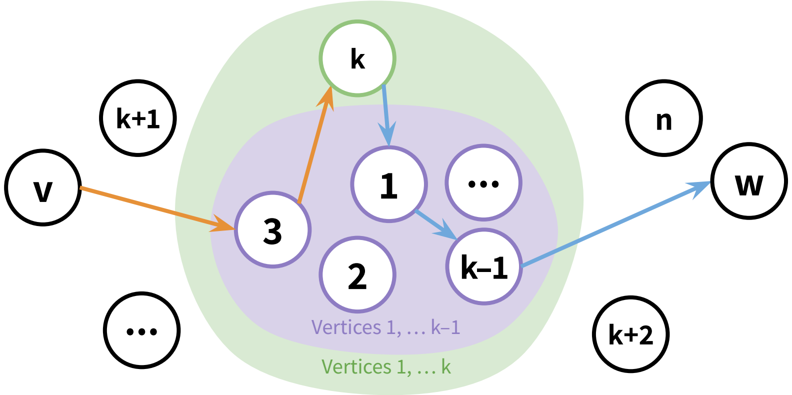
How do we find $D^{(k)}[v, w]$ using $D^{(k-1)}$? Choose the minimum of these 2 cases:

CASE 1: We don't need vertex **k**



$$D^{(k)}[v, w] = D^{(k-1)}[v, w]$$

CASE 2: We need vertex **k**



$$D^{(k)}[v, w] = D^{(k-1)}[v, k] + D^{(k-1)}[k, w]$$

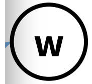
FLOYD-WARSHALL: A DP APPROACH

How do we find $D^{(k)}[v, w]$ using $D^{(k-1)}$? Choose the minimum of these 2 cases:

This is our optimal substructure: We know what our subproblems are (finding costs of shortest paths through a restricted set of vertices), and we know how to express our optimal solution in terms of these subproblem results (get the minimum of these two cases).

These subproblems are also overlapping: Memoization/caching can be useful here! For example, $D^{(k-1)}[k, w]$ can be used to help compute $D^{(k)}[v, w]$ for a lot of different starting points as v!

Now that we've settled this, we can write the algorithm!



$D^{(k)}[v, w] = D^{(k-1)}[v, w]$

$D^{(k)}[v, w]= D^{(k-1)}[v, k] + D^{(k-1)}[k, w]$

FLOYD-WARSHALL: A DP APPROACH

```
FLOYD_WARSHALL(G):
    Initialize n x n arrays $D^{(k)}$ for k = 0,...,n
        $D^{(k)}$[v,v] = 0 for all v, for all k
        $D^{(k)}$[v,w] = ∞ for all v ≠ w, for all k
        $D^{(0)}$[v,w] = weight(v,w) for all (v,w) in E
    for k = 1,...,n:
        for pairs v,w in $V^2$:
            $D^{(k)}$[v,w] = min{ $D^{(k-1)}$[v,w], $D^{(k-1)}$[v,k] + $D^{(k-1)}$[k,w] }
    return $D^{(n)}$
```

FLOYD-WARSHALL: A DP APPROACH

```
**FLOYD_WARSHALL**(G):
    Initialize n x n arrays $D^{(k)}$ for k = 0,...,n
        $D^{(k)}[v,v] = 0$ for all v, for all k
        $D^{(k)}[v,w] = \infty$ for all $v \neq w$, for all k
        $D^{(0)}[v,w]$ = weight(v,w) for all (v,w) in E
    for k = 1,...,n:
        for pairs v,w in $V^2$:
            $D^{(k)}[v,w] = \min\{ D^{(k-1)}[v,w], D^{(k-1)}[v,k] + D^{(k-1)}[k,w] \}$
    return $D^{(n)}$
```

Keeping all these n x n arrays would be a waste of space. In practice, only need to store 2!

Take the minimum over our two cases!

FLOYD-WARSHALL: A DP APPROACH

```
FLOYD_WARSHALL(G):
    Initialize n x n arrays $D^{(k)}$ for k = 0,...,n
        $D^{(k)}[v,v] = 0$ for all v, for all k
        $D^{(k)}[v,w] = \infty$ for all $v \neq w$, for all k
        $D^{(0)}[v,w]$ = weight(v,w) for all (v,w) in E
    for k = 1,...,n:
        for pairs v,w in $V^2$:
            $D^{(k)}[v,w] = \min\{ D^{(k-1)}[v,w], D^{(k-1)}[v,k] + D^{(k-1)}[k,w] \}$
    return $D^{(n)}$
```

Keeping all these n x n arrays would be a waste of space. In practice, only need to store 2!

Take the minimum over our two cases!

Runtime: $O(n^3)$

(Better than running Bellman-Ford n times!)

WHAT ABOUT NEGATIVE CYCLES?

Negative cycle means there's some **v**
s.t. there is a path from **v** to **v** that has cost < 0

WHAT ABOUT NEGATIVE CYCLES?

Negative cycle means there's some **v**
s.t. there is a path from **v** to **v** that has cost < 0

```
FLOYD_WARSHALL(G):
    Initialize n x n arrays $D^{(k)}$ for k = 0,...,n
        $D^{(k)}$[v,v] = 0 for all v, for all k
        $D^{(k)}$[v,w] = ∞ for all v ≠ w, for all k
        $D^{(k)}$[v,w] = weight(v,w) for all (v,w) in E
    for k = 1,...,n:
        for pairs v,w in $V^2$:
            $D^{(k)}$[v,w] = min{ $D^{(k-1)}$[v,w], $D^{(k-1)}$[v,k] + $D^{(k-1)}$[k,w]
}
    for v in V:
        if $D^{(n)}$[v,v] < 0:
            return “NEGATIVE CYCLE!”
    return $D^{(n)}$
```

SHORTEST-PATH ALGORITHMS

$n = |V|$
$m = |E|$

BFS	**DFS**	**DIJKSTRA**	**BELLMAN-FORD**	**FLOYD-WARSHALL**
$O(m+n)$	$O(m+n)$	$O(m+n\log n)$*	$O(mn)$	$O(n^3)$
Unweighted (or weights don't matter)	Unweighted (or weights don't matter)	Weighted (weights must be *non-negative*)	Weighted (can handle *negative* weights)	Weighted (can handle *negative* weights)
Single source shortest path Test bipartiteness Find connected components	Path finding (s,t) Toposort (DAG!!) Find SCC's Find connected components	***Single source shortest paths:*** Compute shortest path from a source s to all other nodes	***Single source shortest paths:*** Compute shortest path from source s to all other nodes Detect negative cycles	***All pairs shortest paths:*** Compute shortest path between every pair of nodes (v,w)