Dynamic Programming I

Summer 2017 • Lecture 08/01

A Few Notes

Homework 5

- Due Friday 8/4 at 11:59 p.m. on Gradescope.

Homework 6

- Released Friday 8/4.

Outline for Today

- Dynamic Programming
  - DP graph algorithms
    - Bellman Ford
    - Floyd Warshall

Bellman-Ford

Bellman-Ford Algorithm

Dijkstra's algorithm solves the single-source shortest path problem in weighted graphs.

- Sometimes it works on graphs with negative edge weights, but sometimes it doesn't work.

Bellman-Ford also solves the SSSP problem in weighted graphs.

- Always works on graphs with negative edge weights (when a solution exists).

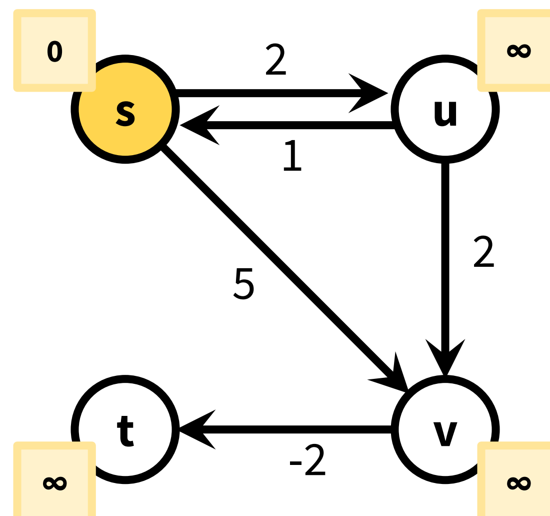
Bellman-Ford Algorithm

We maintain a list $d^{(k)}$ of length n for each k = 0, 1, …, |V|-1.

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

	s	**u**	**v**	**t**
$\mathbf{d^{(0)}}$	**0**	**∞**	**∞**	**∞**
$\mathbf{d^{(1)}}$				
$\mathbf{d^{(2)}}$				
$\mathbf{d^{(3)}}$				

We know k = 0 i.e. shortest paths to each vertex with at most 0 edges in it.



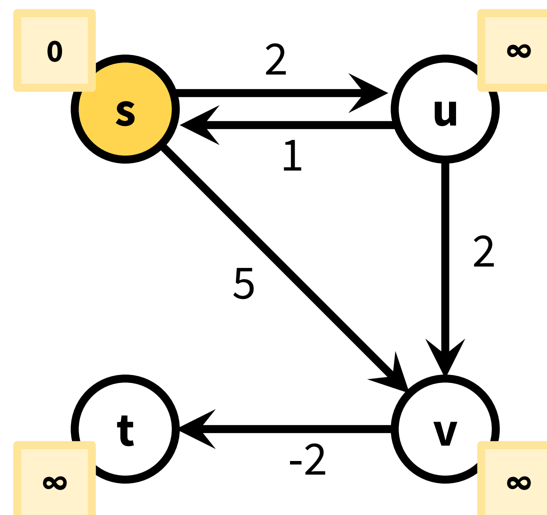
Bellman-Ford Algorithm

We maintain a list $d^{(k)}$ of length n for each k = 0, 1, …, |V|-1.

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

	s	u	v	t
$d^{(0)}$	0	∞	∞	∞
$d^{(1)}$				
$d^{(2)}$				
$d^{(3)}$				

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

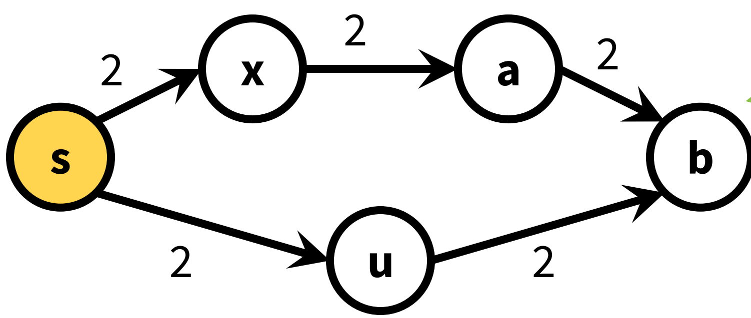


Bellman-Ford Algorithm

How do we use $d^{(k-1)}$ to fill in $d^{(k)}[b]$?

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

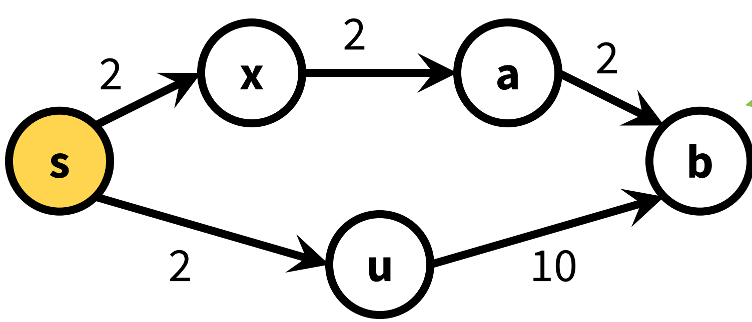
Case 1: the shortest path from s to b with at most k edges actually has at most k - 1 edges.



Suppose k = 3.

$d^{(k)}[b] = d^{(k-1)}[b]$ i.e. the shortest path of at most k - 1 edges is at least as short as any path of at most k edges.

Case 2: the shortest path from s to b with at most k edges really has k edges.



Suppose k = 3.

$d^{(k)}[b] = \min_a\{d^{(k-1)}[a] + w(a, b)\}$ i.e. the shortest path of at most k edges is shorter than any path of at most k - 1 edges.

Bellman-Ford Algorithm

```
algorithm bellman_ford(G):
  d^(k) = [] for k = 0 to |V|-1
  d^(0)[v] = ∞ for all v ≠ s
  d^(0)[s] = 0
  for k = 1 to |V|-1:
    for b in V:
      d^(k)[b] = min{d^(k-1)[b], min_a{d^(k-1)[a] + w(a,b)} }
  return d^(|V|-1)
```

Runtime: $O(|V||E|)$

Bellman-Ford Algorithm

```
algorithm bellman_ford(G):
  d^(k) = [] for k = 0 to |V|-1
  d^(0)[v] = ∞ for all v ≠ s
  d^(0)[s] = 0
  for k = 1 to |V|-1:
    for b in V:
      d^(k)[b] = min{d^(k-1)[b], min_a{d^(k-1)[a] + w(a,b)} }
  return d^(|V|-1)
```

This is a simplification to make the pseudocode nice. In reality, we'd only keep two of them at a time.

Runtime: $O(|V||E|)$

Bellman-Ford Algorithm

```
algorithm bellman_ford(G):
  d^(k) = [] for k = 0 to |V|-1
  d^(0)[v] = ∞ for all v ≠ s
  d^(0)[s] = 0
  for k = 1 to |V|-1:
    for b in V:
      d^(k)[b] = min{d^(k-1)[b], min_a{d^(k-1)[a] + w(a,b)} }
  return d^(|V|-1)
```

This is a simplification to make the pseudocode nice. In reality, we'd only keep two of them at a time.

Minimum over all a such that (a, b) ∈ E.

Runtime: $O(|V||E|)$

Bellman-Ford Algorithm

```
algorithm bellman_ford(G):
  d⁽ᵏ⁾ = [] for k = 0 to |V|-1
  d⁽⁰⁾[v] = ∞ for all v ≠ s
  d⁽⁰⁾[s] = 0
  for k = 1 to |V|-1:
    for b in V:
      d⁽ᵏ⁾[b] = min{d⁽ᵏ⁻¹⁾[b], min_a{d⁽ᵏ⁻¹⁾[a] + w(a,b)} }
  return d⁽|V|-1⁾
```

This is a simplification to make the pseudocode nice. In reality, we'd only keep two of them at a time.

Minimum over all a such that $(a, b) \in E$.

Case 1

Runtime: $O(|V||E|)$

Bellman-Ford Algorithm

```
algorithm bellman_ford(G):
  d^(k) = [] for k = 0 to |V|-1
  d^(0)[v] = ∞ for all v ≠ s
  d^(0)[s] = 0
  for k = 1 to |V|-1:
    for b in V:
      d^(k)[b] = min{d^(k-1)[b], min_a{d^(k-1)[a] + w(a,b)} }
  return d^(|V|-1)
```

$d^{(k)}[b] = \min\{\underbrace{d^{(k-1)}[b]}_{\text{Case 1}}, \underbrace{\min_a\{d^{(k-1)}[a] + w(a,b)\}}_{\text{Case 2}}\}$

This is a simplification to make the pseudocode nice. In reality, we'd only keep two of them at a time.

Minimum over all a such that (a, b) ∈ E.

Runtime: O(|V||E|)

Bellman-Ford Algorithm

```
algorithm bellman_ford(G):
  d^(k) = [] for k = 0 to |V|-1
  d^(0)[v] = ∞ for all v ≠ s
  d^(0)[s] = 0
  for k = 1 to |V|-1:
    for b in V:
      d^(k)[b] = min{d^(k-1)[b], min_a{d^(k-1)[a] + w(a,b)} }
  return d^(|V|-1)
```

This is a simplification to make the pseudocode nice. In reality, we'd only keep two of them at a time.

Minimum over all a such that (a, b) ∈ E.

$$d^{(k)}[b] = \min\{\underbrace{d^{(k-1)}[b]}_{\text{Case 1}},\ \underbrace{\min_a\{d^{(k-1)}[a] + w(a,b)\}}_{\text{Case 2}}\}$$

Runtime: $O(|V||E|)$

Slower than Dijkstra's $O(|E| + |V|\log(|V|))$

Bellman-Ford Algorithm

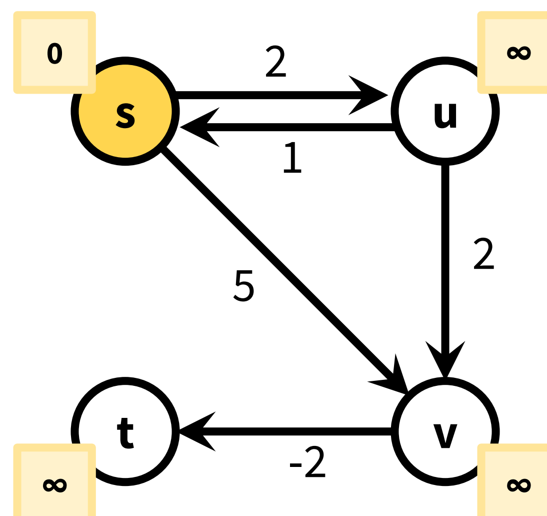
We maintain a list $d^{(k)}$ of length n for each k = 0, 1, …, |V|-1.

```
for k = 1 to |V|-1:
  for b in V:
    d(k)[b] = min{d(k-1)[b], min_a{d(k-1)[a] + w(a,b)} }
```

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

	s	u	v	t
$d^{(0)}$	0	∞	∞	∞
$d^{(1)}$				
$d^{(2)}$				
$d^{(3)}$				

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



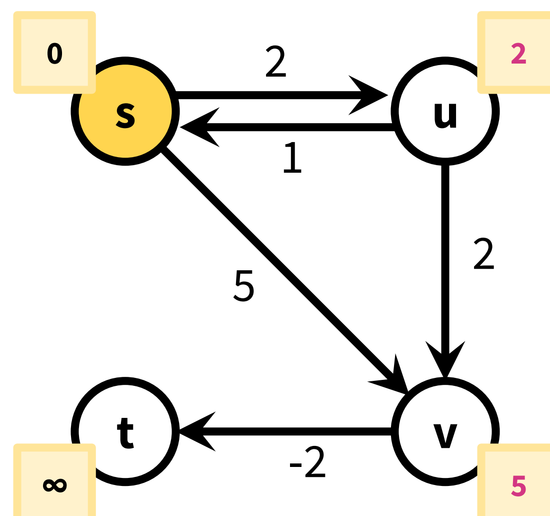
Bellman-Ford Algorithm

We maintain a list $d^{(k)}$ of length n for each $k = 0, 1, \ldots, |V|-1$.

```
for k = 1 to |V|-1:
  for b in V:
    d^(k)[b] = min{d^(k-1)[b], min_a{d^(k-1)[a] + w(a,b)} }
```

	s	u	v	t
$d^{(0)}$	0	∞	∞	∞
$d^{(1)}$	0	2	5	∞
$d^{(2)}$				
$d^{(3)}$				

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

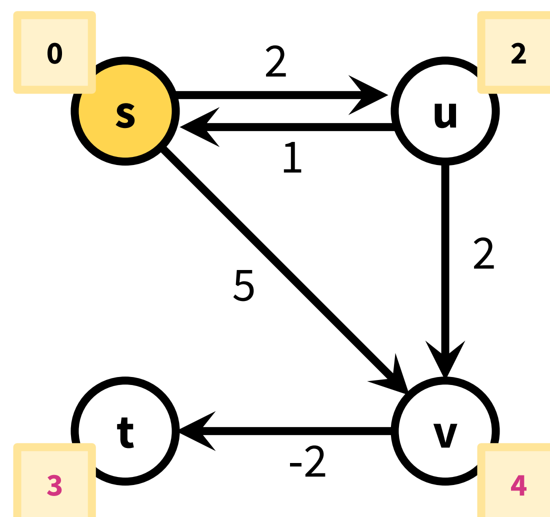


Bellman-Ford Algorithm

We maintain a list $d^{(k)}$ of length n for each k = 0, 1, …, |V|-1.

```
for k = 1 to |V|-1:
  for b in V:
    d^(k)[b] = min{d^(k-1)[b], min_a{d^(k-1)[a] + w(a,b)} }
```

	s	u	v	t
$d^{(0)}$	0	∞	∞	∞
$d^{(1)}$	0	2	5	∞
$d^{(2)}$	0	2	4	3
$d^{(3)}$				

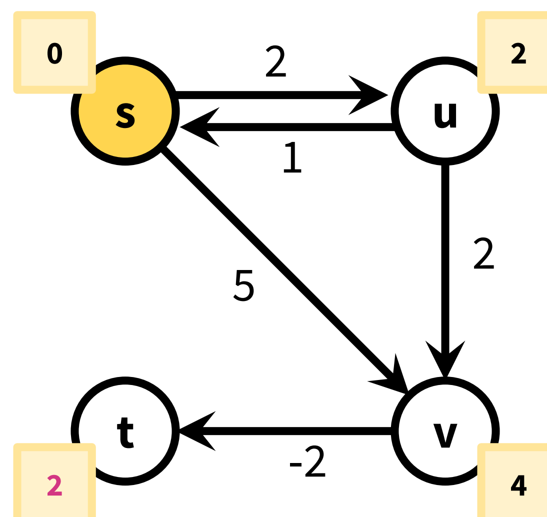


Bellman-Ford Algorithm

We maintain a list $d^{(k)}$ of length n for each k = 0, 1, …, |V|-1.

```
for k = 1 to |V|-1:
  for b in V:
    d^(k)[b] = min{d^(k-1)[b], min_a{d^(k-1)[a] + w(a,b)} }
```

	s	**u**	**v**	**t**
$\mathbf{d^{(0)}}$	0	∞	∞	∞
$\mathbf{d^{(1)}}$	0	2	5	∞
$\mathbf{d^{(2)}}$	0	2	4	3
$\mathbf{d^{(3)}}$	0	2	4	2

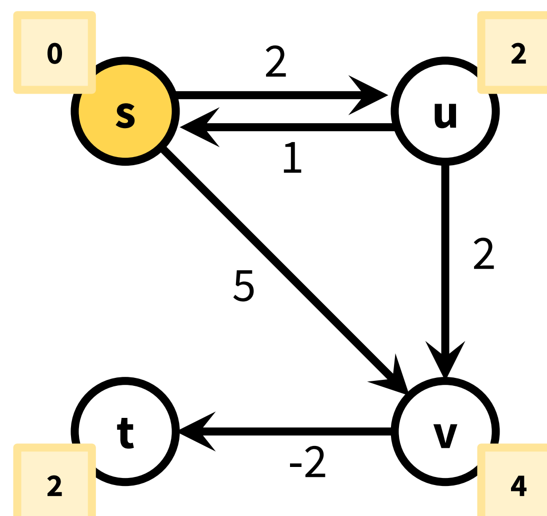


Bellman-Ford Algorithm

We maintain a list $d^{(k)}$ of length n for each k = 0, 1, …, |V|-1.

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

	s	u	v	t
$d^{(0)}$	0	∞	∞	∞
$d^{(1)}$	0	2	5	∞
$d^{(2)}$	0	2	4	3
$d^{(3)}$	0	2	4	2



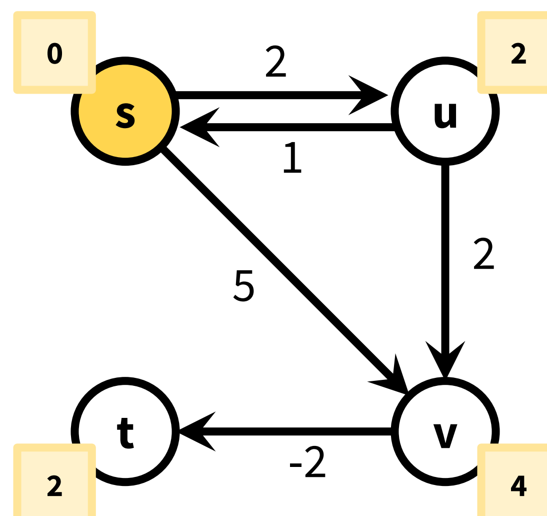
Bellman-Ford Algorithm

We maintain a list $d^{(k)}$ of length n for each k = 0, 1, …, |V|-1.

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

The shortest path from s to t with 1 edge has cost ∞ (no path exists).

	s	u	v	t
$d^{(0)}$	0	∞	∞	∞
$d^{(1)}$	0	2	5	∞
$d^{(2)}$	0	2	4	3
$d^{(3)}$	0	2	4	2



Bellman-Ford Algorithm

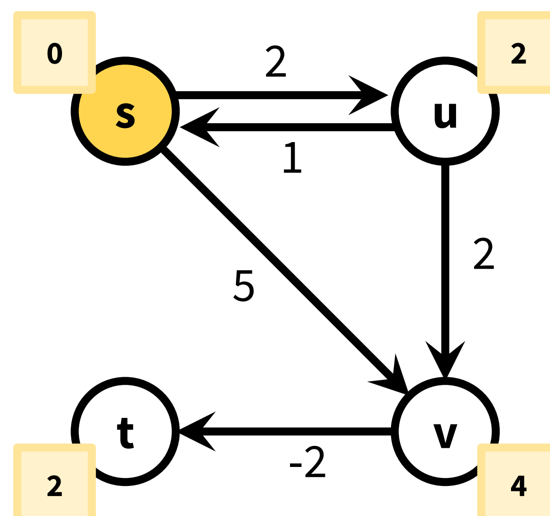
We maintain a list $d^{(k)}$ of length n for each k = 0, 1, …, |V|-1.

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

The shortest path from s to t with 1 edge has cost **∞** (no path exists).

The shortest path from s to t with 2 edges has cost **3** (s-v-t).

	s	u	v	t
$d^{(0)}$	0	∞	∞	∞
$d^{(1)}$	0	2	5	∞
$d^{(2)}$	0	2	4	3
$d^{(3)}$	0	2	4	2



Bellman-Ford Algorithm

We maintain a list $d^{(k)}$ of length n for each $k = 0, 1, \ldots, |V|-1$.

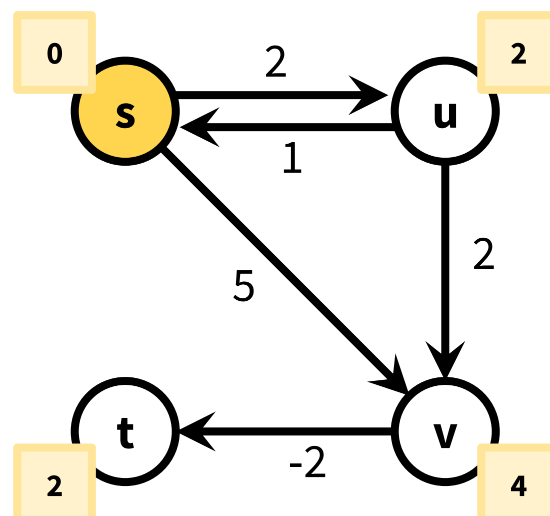
Recall $d^{(k)}[b]$ is the cost of the shortest path from s to b with at most k edges.

The shortest path from s to t with 1 edge has cost **∞** (no path exists).

The shortest path from s to t with 2 edges has cost **3** (s-v-t).

The shortest path from s to t with 3 edges has cost **2** (s-u-v-t).

	s	**u**	**v**	**t**
$\mathbf{d^{(0)}}$	**0**	**∞**	**∞**	**∞**
$\mathbf{d^{(1)}}$	**0**	**2**	**5**	**∞**
$\mathbf{d^{(2)}}$	**0**	**2**	**4**	**3**
$\mathbf{d^{(3)}}$	**0**	**2**	**4**	**2**



BF Proof of Correctness

We need to prove our main argument.

- $d^{(|V|-1)}[b]$ is the cost of the shortest path from s to b with at most |V|-1 edges.

BF Proof of Correctness

Lemma: $d^{(|V|-1)}[b]$ is the cost of the shortest path from s to b with at most |V|-1 edges.

BF Proof of Correctness

Lemma: $d^{(|V|-1)}[b]$ is the cost of the shortest path from s to b with at most $|V|-1$ edges.

Proof: We proceed by induction on k, the number of iterations completed by the algorithm.

BF Proof of Correctness

Lemma: $d^{(|V|-1)}[b]$ is the cost of the shortest path from s to b with at most |V|-1 edges.

Proof: We proceed by induction on k, the number of iterations completed by the algorithm.

For our base case, at the start of iteration k = 1, the shortest path from s to s with 0 edges has cost 0. The path from s to all vertices v ≠ s contains at least 1 edge; there doesn't exist a path from s to v with 0 edges, and this path costs ∞. Therefore, $d^{(0)}$ is correct.

BF Proof of Correctness

Lemma: $d^{(|V|-1)}[b]$ is the cost of the shortest path from s to b with at most |V|-1 edges.

Proof: We proceed by induction on k, the number of iterations completed by the algorithm.

For our base case, at the start of iteration k = 1, the shortest path from s to s with 0 edges has cost 0. The path from s to all vertices v ≠ s contains at least 1 edge; there doesn't exist a path from s to v with 0 edges, and this path costs ∞. Therefore, $d^{(0)}$ is correct.

For our inductive step, assume that at the start of iteration k, $d^{(k-1)}[b]$ is the cost of the shortest path from s to b with at most k - 1 edges. We consider two cases:

BF Proof of Correctness

Lemma: $d^{(|V|-1)}[b]$ is the cost of the shortest path from s to b with at most $|V|-1$ edges.

Proof: We proceed by induction on k, the number of iterations completed by the algorithm.

For our base case, at the start of iteration $k = 1$, the shortest path from s to s with 0 edges has cost 0. The path from s to all vertices $v \neq s$ contains at least 1 edge; there doesn't exist a path from s to v with 0 edges, and this path costs ∞. Therefore, $d^{(0)}$ is correct.

For our inductive step, assume that at the start of iteration k, $d^{(k-1)}[b]$ is the cost of the shortest path from s to b with at most k - 1 edges. We consider two cases:

Case 1: $d^{(k-1)}[b] < \min_a\{d^{(k-1)}[a] + w(a, b)\}$. This corresponds to the case in which the shortest path contains fewer than k edges. Then our algorithm correctly sets $d^{(k)}[b] = d^{(k-1)}[b]$.

BF Proof of Correctness

Lemma: $d^{(|V|-1)}[b]$ is the cost of the shortest path from s to b with at most |V|-1 edges.

Proof: We proceed by induction on k, the number of iterations completed by the algorithm.

For our base case, at the start of iteration k = 1, the shortest path from s to s with 0 edges has cost 0. The path from s to all vertices v ≠ s contains at least 1 edge; there doesn't exist a path from s to v with 0 edges, and this path costs ∞. Therefore, $d^{(0)}$ is correct.

For our inductive step, assume that at the start of iteration k, $d^{(k-1)}[b]$ is the cost of the shortest path from s to b with at most k - 1 edges. We consider two cases:

Case 1: $d^{(k-1)}[b] < \min_a\{d^{(k-1)}[a] + w(a, b)\}$. This corresponds to the case in which the shortest path contains fewer than k edges. Then our algorithm correctly sets $d^{(k)}[b] = d^{(k-1)}[b]$.

Case 2: $d^{(k-1)}[b] \geq \min_a\{d^{(k-1)}[a] + w(a, b)\}$. This corresponds to the case in which the shortest path contains exactly k edges. Then our algorithm correctly sets $d^{(k)}[b] = \min_a\{d^{(k-1)}[a] + w(a, b)\}$, which minimizes the sum of the shortest path with at most k-1 edges to an in-neighbor of b and the weight from a to b.

BF Proof of Correctness

Lemma: $d^{(|V|-1)}[b]$ is the cost of the shortest path from s to b with at most |V|-1 edges.

Proof: We proceed by induction on k, the number of iterations completed by the algorithm.

For our base case, at the start of iteration k = 1, the shortest path from s to s with 0 edges has cost 0. The path from s to all vertices v ≠ s contains at least 1 edge; there doesn't exist a path from s to v with 0 edges, and this path costs ∞. Therefore, $d^{(0)}$ is correct.

For our inductive step, assume that at the start of iteration k, $d^{(k-1)}[b]$ is the cost of the shortest path from s to b with at most k - 1 edges. We consider two cases:

Case 1: $d^{(k-1)}[b] < \min_a\{d^{(k-1)}[a] + w(a, b)\}$. This corresponds to the case in which the shortest path contains fewer than k edges. Then our algorithm correctly sets $d^{(k)}[b] = d^{(k-1)}[b]$.

Case 2: $d^{(k-1)}[b] \geq \min_a\{d^{(k-1)}[a] + w(a, b)\}$. This corresponds to the case in which the shortest path contains exactly k edges. Then our algorithm correctly sets $d^{(k)}[b] = \min_a\{d^{(k-1)}[a] + w(a, b)\}$, which minimizes the sum of the shortest path with at most k-1 edges to an in-neighbor of b and the weight from a to b.

At the start of iteration k = |V|, the algorithm terminates and $d^{(|V|-1)}$ is correct.

BF Proof of Correctness

- We need to prove our main argument.
  - $d^{(|V|-1)}[b]$ is the cost of the shortest path from s to b with at most |V|-1 edges.👌
- What else to do? 🤔

BF Proof of Correctness

We need to prove our main argument.

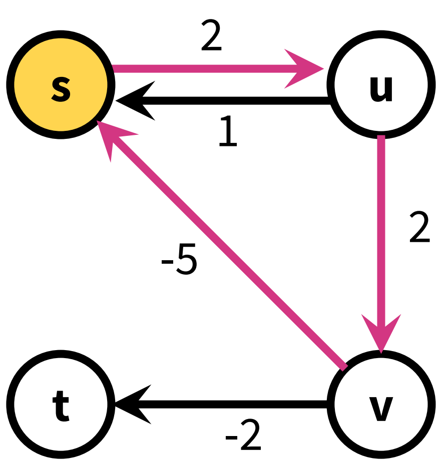
$d^{(|V|-1)}[b]$ is the cost of the shortest path from s to b with at most |V|-1 edges. 👌

What else to do? 🤔

We still need to prove that this argument implies `bellman_ford` is correct i.e. $d^{(|V|-1)}[a]$ = distance(s, a).

To show this, we'll prove that the shortest path with at most |V|-1 edges is the shortest path with any number of edges (if a shortest path exists).

If the graph has a negative cycle, a shortest path might not exist!

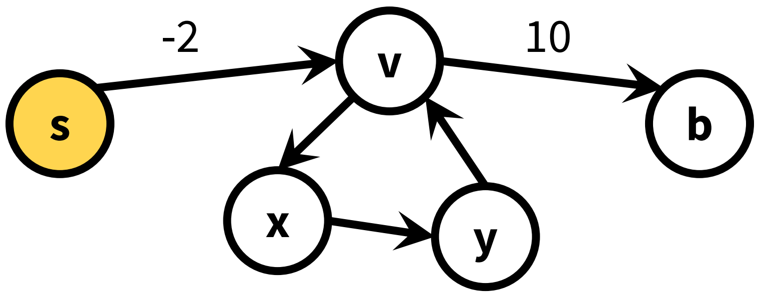


BF Proof of Correctness

But if there's no negative cycle.

There's always a simple shortest path.

A simple path has
no cycles.



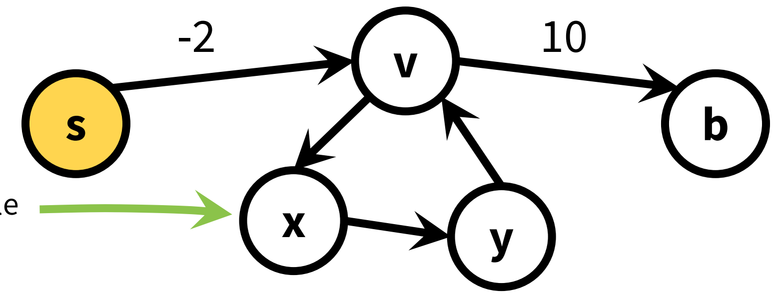
BF Proof of Correctness

But if there's no negative cycle.

There's always a simple shortest path.

A simple path has
no cycles.

How do we know this cycle
doesn't help? 🤔

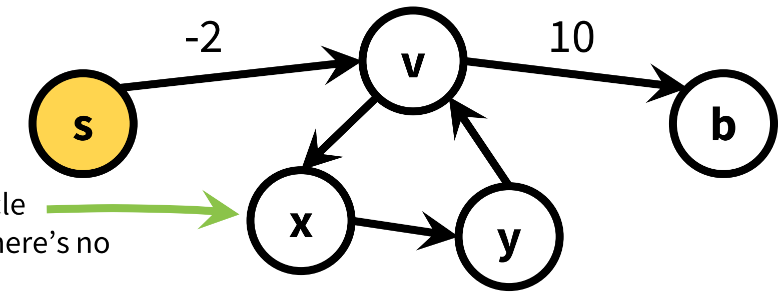


BF Proof of Correctness

But if there's no negative cycle.

There's always a simple shortest path.

A simple path has no cycles.



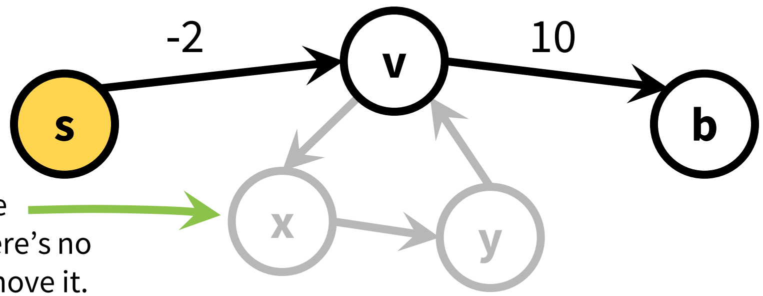
How do we know this cycle doesn't help? 🤔 Since there's no negative cycles!

BF Proof of Correctness

But if there's no negative cycle.

There's always a simple shortest path.

A simple path has no cycles.



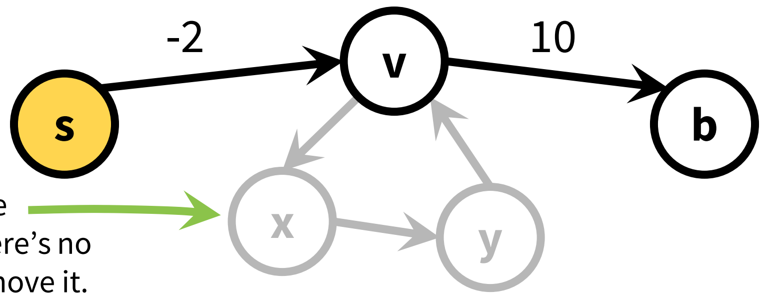
How do we know this cycle doesn't help? 🤔 Since there's no negative cycles! So we remove it.

BF Proof of Correctness

But if there's no negative cycle.

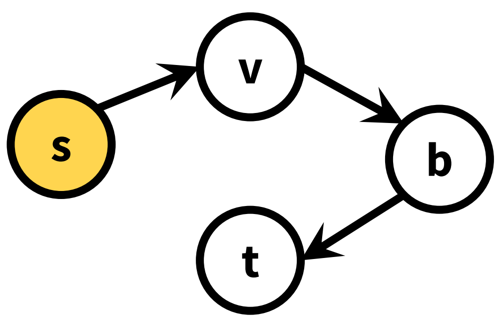
There's always a simple shortest path.

A simple path has
no cycles.



How do we know this cycle
doesn't help? 🤔 Since there's no
negative cycles! So we remove it.

A simple path in a graph with |V| vertices has at most |V|-1 edges in it.



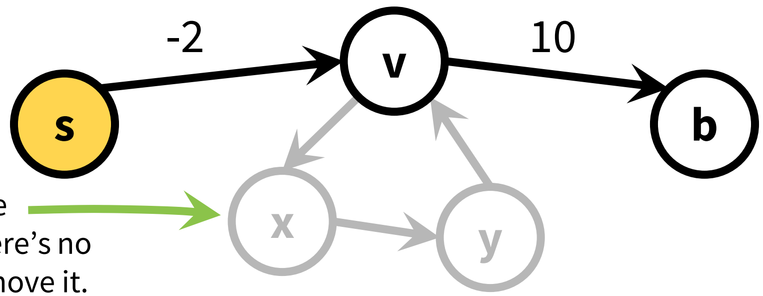
BF Proof of Correctness

But if there's no negative cycle.

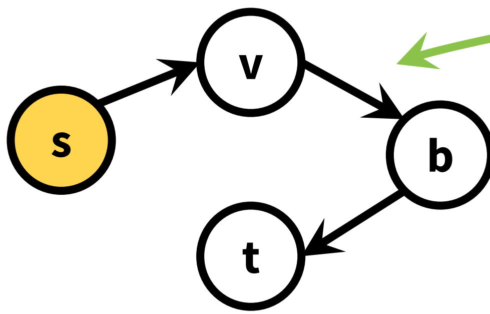
There's always a simple shortest path.

A simple path has no cycles.

How do we know this cycle doesn't help? 🤔 Since there's no negative cycles! So we remove it.



A simple path in a graph with |V| vertices has at most |V|-1 edges in it.



We can't add another edge to this s-t path without making a cycle (an edge from s to b wouldn't be along the path).

BF Proof of Correctness

Theorem: `bellman_ford` is correct as long as the graph has no negative cycles.

Proof:

By our lemma, $d^{(|V|-1)}[b]$ contains the cost of the shortest path from s to b with at most |V|-1 edges. If there are no negative cycles, then the shortest path must be simple, and all simple paths have at most |V|-1 edges. Therefore, the value the algorithm returns, $d^{(|V|-1)}[b]$, is also the cost of the shortest path from s to b with any number of edges.

Bellman-Ford Algorithm

Bellman-Ford gets used in practice.

- e.g. Routing Information Protocol (RIP) uses it. Each router keeps a table of distances to every other router. Periodically, we do a Bellman-Ford update.

Dynamic Programming

Bellman-Ford is an example of **dynamic programming**!

Dynamic programming is an algorithm design paradigm.

- Often it's used to solve optimization problems e.g. **shortest** path.

Dynamic Programming

Elements of dynamic programming

Large problems break up into small problems.

e.g. shortest path with at most k edges.

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

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

Overlapping sub-problems the sub-problems overlap a lot.

e.g. Lots of different entries of $d^{(k)}$ ask for $d^{(k-1)}[a]$.

This means we're save time by solving a sub-problem once and caching the answer.

Dynamic Programming

Two approaches for DP: bottom-up and top-down.

Bottom-up iterates through problems by size and solves the small problems first (Bellman-Ford solves $d^{(0)}$ then $d^{(1)}$ then $d^{(2)}$, etc.)

Top-down recurses to solve smaller problems, which recurse to solve even smaller problems.

How is this different than divide and conquer? **Memoization**, which keeps track of the small problems you've already solved to prevent resolving the same problem more than once.

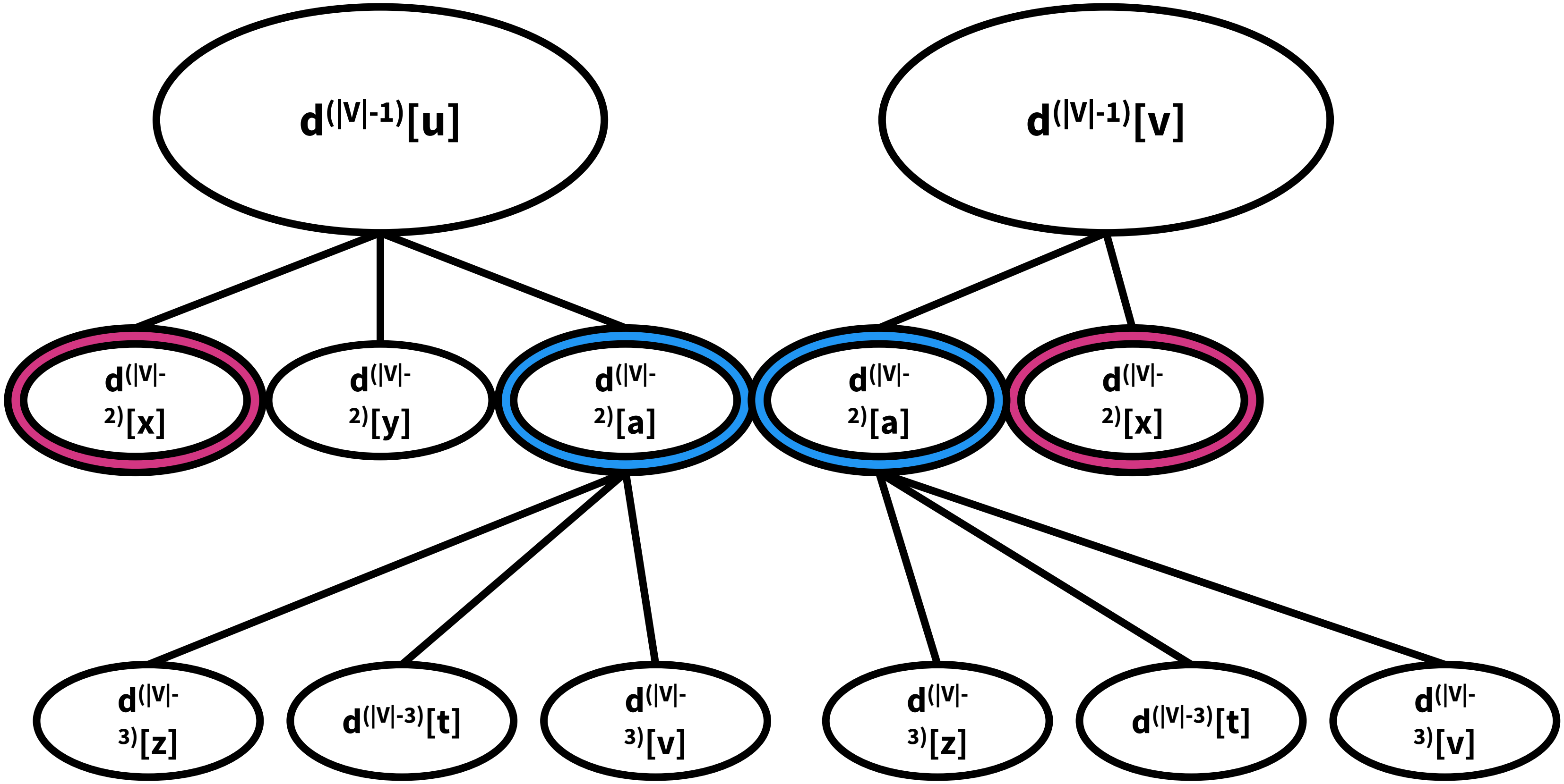
Top-Down BF Algorithm

```
algorithm recursive_bellman_ford(G):
  d(k) = [None] * |V| for k = 0 to |V|-1
  d(0)[v] = ∞ for all v ≠ s
  d(0)[s] = 0
  for b in V:
    recursive_bf_helper(G, b, |V|-1)

algorithm recursive_bf_helper(G, b, k):
  A = {a such that (a, b) in E} ∪ {b}
  for a in A:
    if d(k-1)[a] not None:
      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)} }
```

Runtime: $O(|V||E|)$

Visualization of Top-Down



Floyd-Warshall

Floyd-Warshall Algorithm

Another example of a graph DP algorithm!

The algorithm solves the all-pairs shortest path (**APSP**) problem.

A naive solution

```
for s in V:
  run bellman_ford starting at s
```

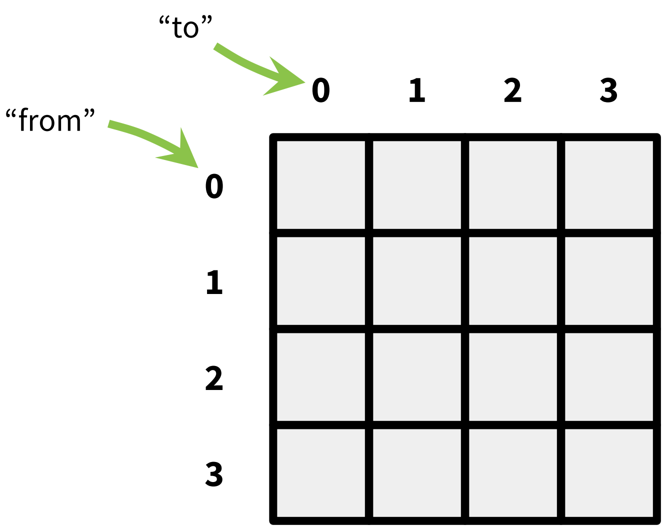
Runtime $O(|V|^2|E|)$

Can we do better?

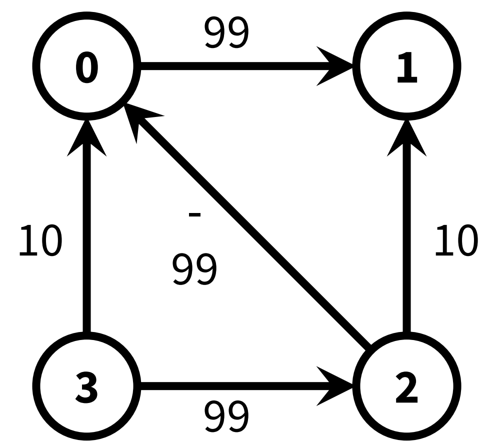
Floyd-Warshall Algorithm

We maintain an $|V| \times |V|$ matrix $D^{(k)}$ for each $k = 0, 1, \ldots, |V|$.

$D^{(k)}[u, v]$ is the cost of the shortest path from u to v, such that all of the internal vertices on the path are in the set of vertices $\{0, \ldots, k\text{-}1\}$.



$D^{(0)}[u, v]$

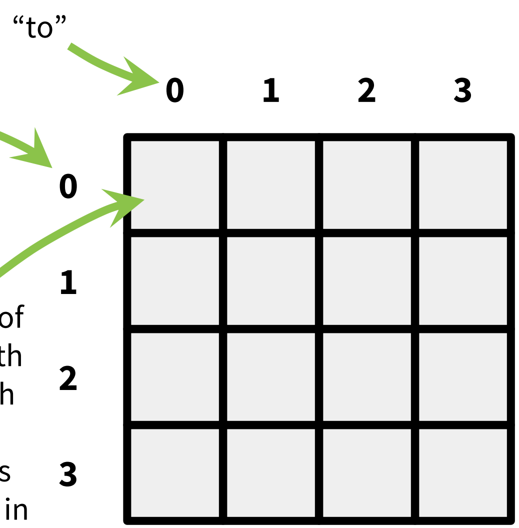


Floyd-Warshall Algorithm

We maintain an $|V| \times |V|$ matrix $D^{(k)}$ for each $k = 0, 1, \ldots, |V|$.

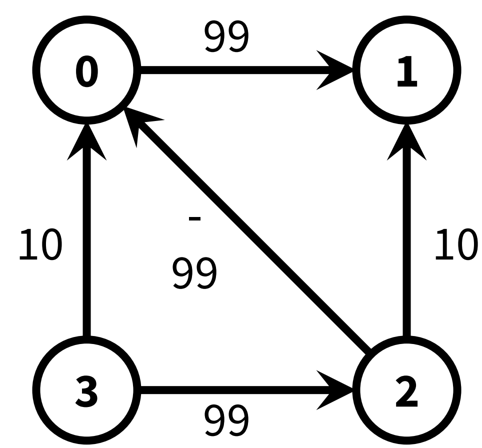
$D^{(k)}[u, v]$ is the cost of the shortest path from u to v, such that all of the internal vertices on the path are in the set of vertices $\{0, \ldots, k\text{-}1\}$.

“from”



This is the cost of the shortest path from 0 to 0, such that all of the internal vertices on the path are in the set of vertices $\{0, \ldots, -1\}$ i.e. the cost of the shortest path from 0 to 0 that passes through no other vertices.

$D^{(0)}[u, v]$



Floyd-Warshall Algorithm

We maintain an $|V| \times |V|$ matrix $D^{(k)}$ for each $k = 0, 1, \ldots, |V|$.

$D^{(k)}[u, v]$ is the cost of the shortest path from u to v, such that all of the internal vertices on the path are in the set of vertices $\{0, \ldots, k\text{-}1\}$.

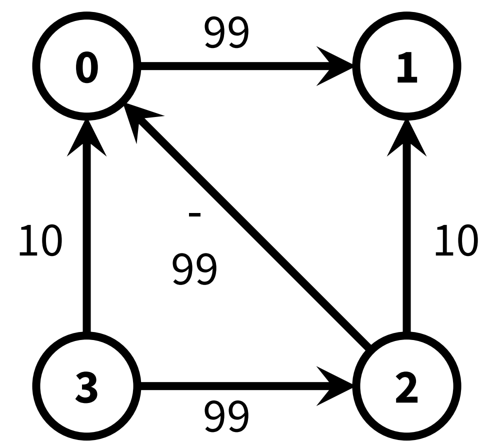
“to”

“from”

	0	**1**	**2**	**3**
0	**0**			
1				
2				
3				

$D^{(0)}[u, v]$

This is the cost of the shortest path from 0 to 0, such that all of the internal vertices on the path are in the set of vertices {0, …, -1} i.e. the cost of the shortest path from 0 to 0 that passes through no other vertices.



Floyd-Warshall Algorithm

We maintain an $|V| \times |V|$ matrix $D^{(k)}$ for each $k = 0, 1, \ldots, |V|$.

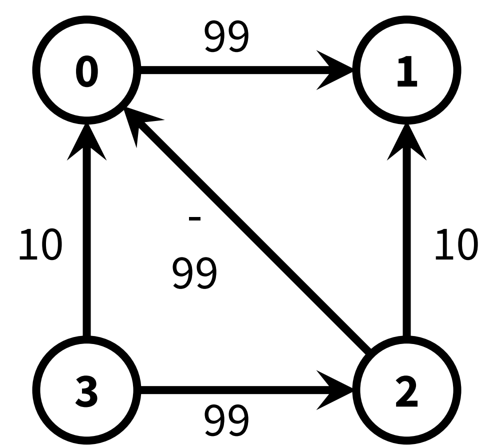
$D^{(k)}[u, v]$ is the cost of the shortest path from u to v, such that all of the internal vertices on the path are in the set of vertices $\{0, \ldots, k\text{-}1\}$.

“to”

“from”

	0	1	2	3
0	0			
1		0		
2			0	
3				0

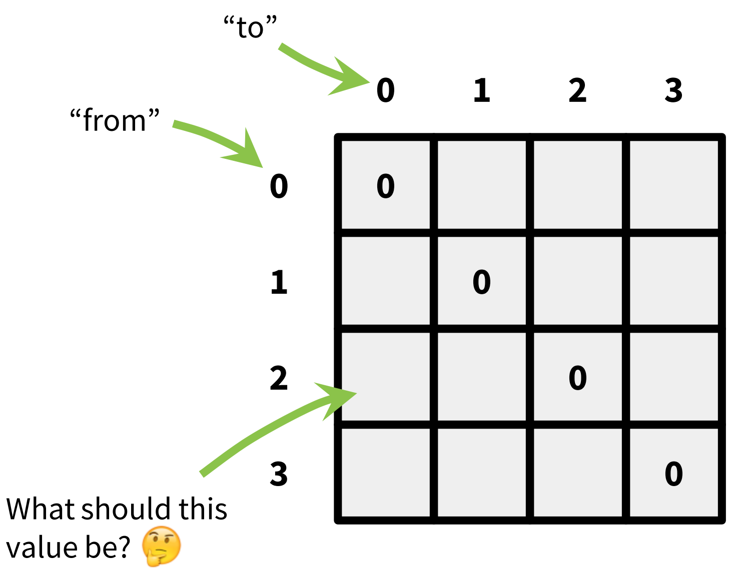
$D^{(0)}[u, v]$



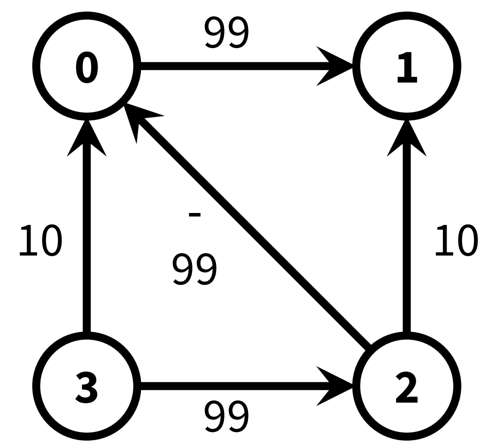
Floyd-Warshall Algorithm

We maintain an $|V| \times |V|$ matrix $D^{(k)}$ for each $k = 0, 1, \ldots, |V|$.

$D^{(k)}[u, v]$ is the cost of the shortest path from u to v, such that all of the internal vertices on the path are in the set of vertices $\{0, \ldots, k-1\}$.



$D^{(0)}[u, v]$



Floyd-Warshall Algorithm

We maintain an |V|×|V| matrix $D^{(k)}$ for each k = 0, 1, …, |V|.

$D^{(k)}[u, v]$ is the cost of the shortest path from u to v, such that all of the internal vertices on the path are in the set of vertices {0, …, k-1}.

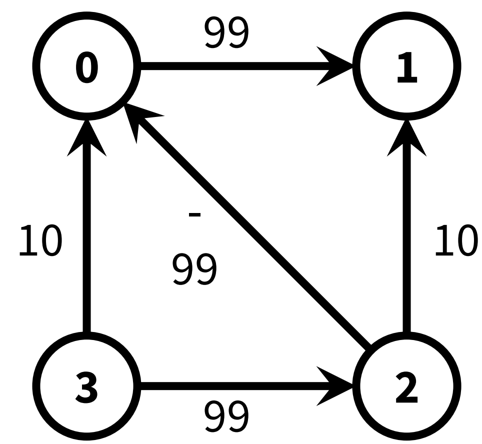
"to"

"from"

	0	1	2	3
0	0			
1		0		
2	-99		0	
3				0

$D^{(0)}[u, v]$

What should this value be? 🤔 -99, since the shortest path from 2 to 0, passing through no other vertices has weight -99.



Floyd-Warshall Algorithm

“to”

“from”

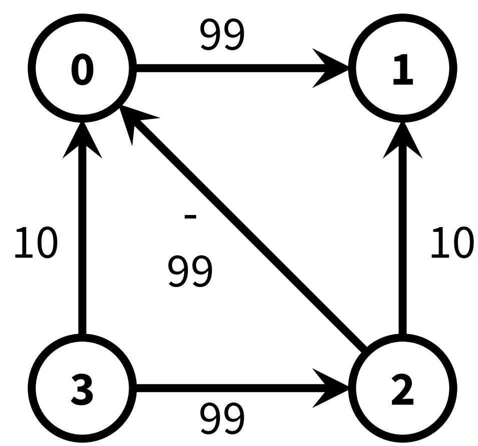
	0	1	2	3
0	0	99	∞	∞
1	∞	0	∞	∞
2	-99	10	0	∞
3	10	∞	99	0

$D^{(0)}[u, v]$

	0	1	2	3
0				
1				
2				
3				

$D^{(1)}[u, v]$

$D^{(k)}[u, v]$ is the cost of the shortest path from u to v, such that all of the internal vertices on the path are in the set of vertices {0, …, k-1}.



Since k = 1, shortest paths are allowed to pass through vertices {0} now. So the we can compare the current cost to the cost of path 3-0-1. $D^{(0)}$ tells us the cost of 3-0 is **10** and the cost of 0-1 is **99**.

Floyd-Warshall Algorithm

“to”

“from”

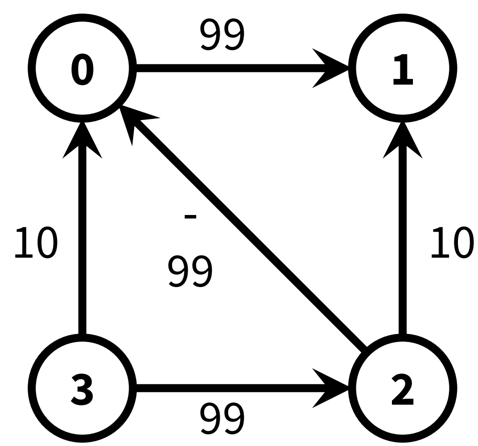
	0	1	2	3
0	0	99	∞	∞
1	∞	0	∞	∞
2	-99	10	0	∞
3	10	∞	99	0

$D^{(0)}[u, v]$

	0	1	2	3
0				
1				
2				
3		109		

$D^{(1)}[u, v]$

$D^{(k)}[u, v]$ is the cost of the shortest path from u to v, such that all of the internal vertices on the path are in the set of vertices {0, …, k-1}.



Since k = 1, shortest paths are allowed to pass through vertices {0} now. So the we can compare the current cost to the cost of path 3-0-1. $D^{(0)}$ tells us the cost of 3-0 is **10** and the cost of 0-1 is **99**. Since the sum of these values is less than ∞, replace it.

Floyd-Warshall Algorithm

“to”

“from”

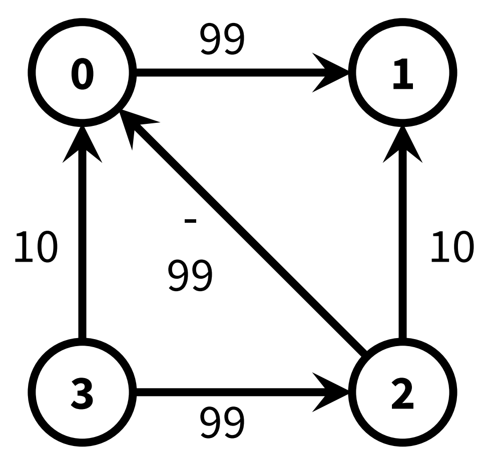
	0	1	2	3
0	0	99	∞	∞
1	∞	0	∞	∞
2	-99	10	0	∞
3	10	∞	99	0

$D^{(0)}[u, v]$

	0	1	2	3
0				
1				
2		0		
3		109		

$D^{(1)}[u, v]$

$D^{(k)}[u, v]$ is the cost of the shortest path from u to v, such that all of the internal vertices on the path are in the set of vertices {0, …, k-1}.



Floyd-Warshall Algorithm

“to”

“from”

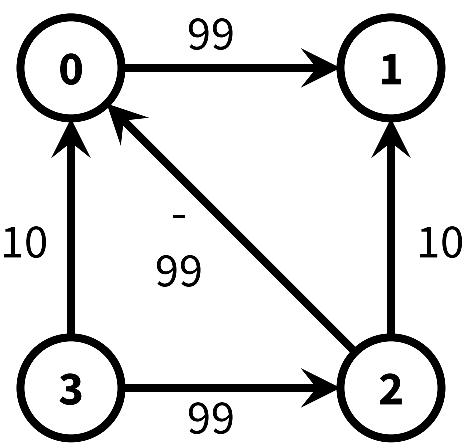
	0	1	2	3
0	0	99	∞	∞
1	∞	0	∞	∞
2	-99	10	0	∞
3	10	∞	99	0

$D^{(0)}[u, v]$

	0	1	2	3
0	0	99	∞	∞
1	∞	0	∞	∞
2	-99	0	0	∞
3	10	109	99	0

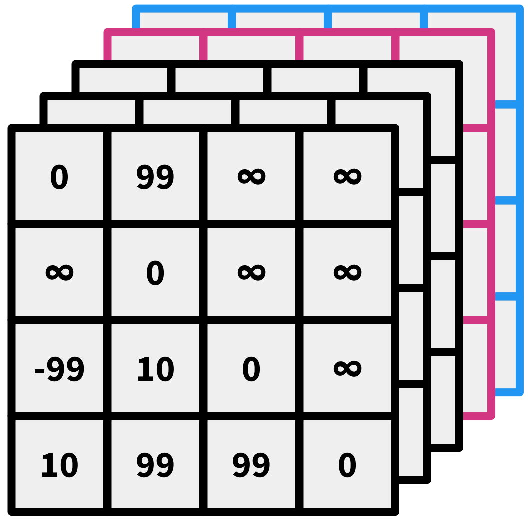
$D^{(1)}[u, v]$

$D^{(k)}[u, v]$ is the cost of the shortest path from u to v, such that all of the internal vertices on the path are in the set of vertices {0, …, k-1}.

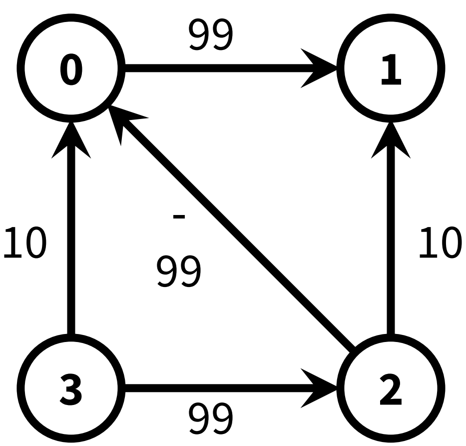


Floyd-Warshall Algorithm

$O(|V|^3)$



$D^{(4)}[u, v]$



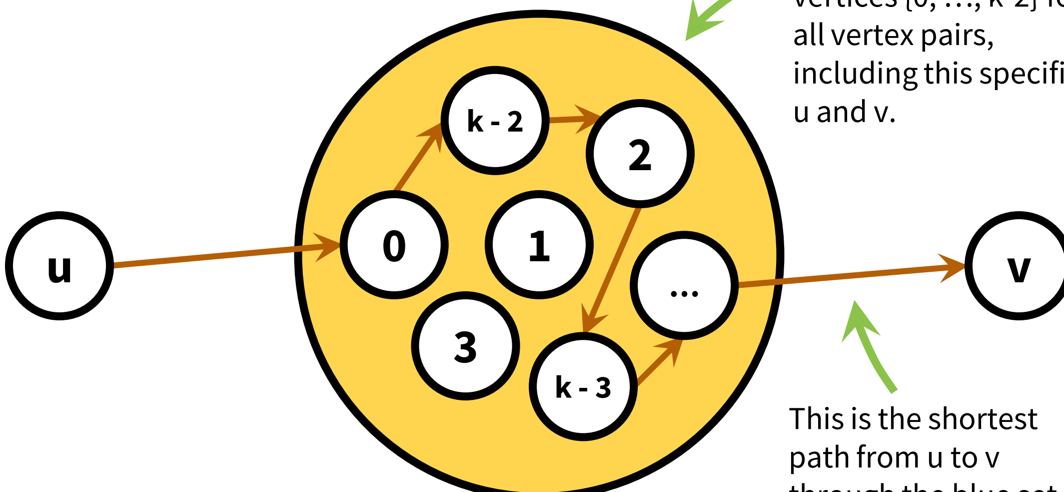
Floyd-Warshall Algorithm

We can represent it more graphically.

$D^{(k)}[u, v]$ is the cost of the shortest path from u to v, such that all of the internal vertices on the path are in the set of vertices {0, …, k-1}.

How might we find $D^{(k)}[u, v]$ using $D^{(k-1)}$?

$D^{(k-1)}$ describes the cost of the shortest path through internal vertices {0, …, k-2} for all vertex pairs, including this specific u and v.



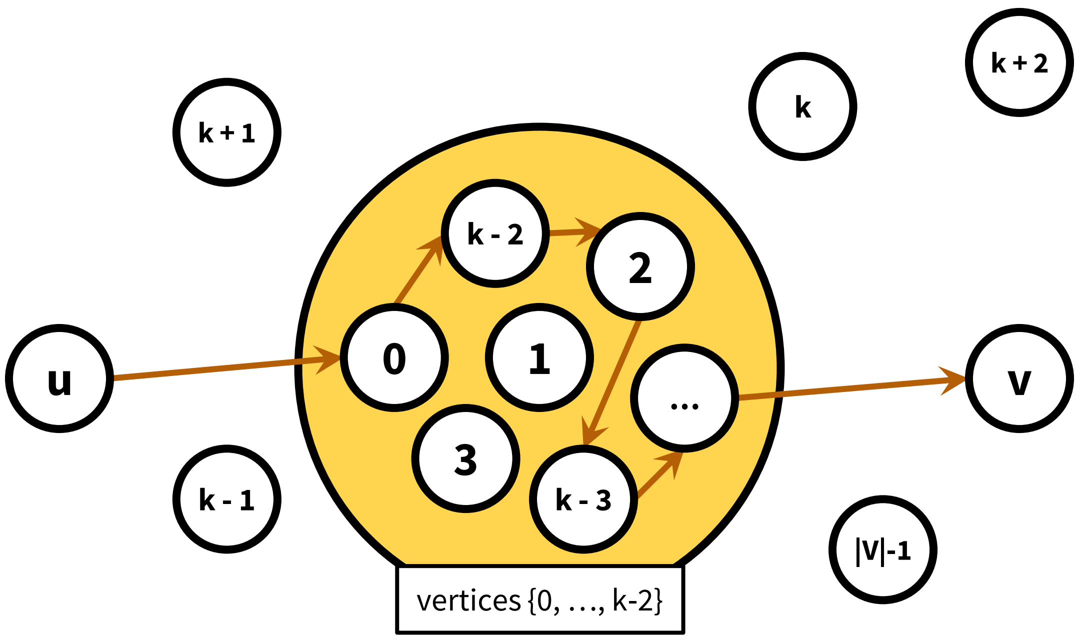
This is the shortest path from u to v through the blue set of vertices; it has weight $D^{(k-1)}[u, v]$.

Floyd-Warshall Algorithm

We can represent it more graphically.

$D^{(k)}[u, v]$ is the cost of the shortest path from u to v, such that all of the internal vertices on the path are in the set of vertices {0, …, k-1}.

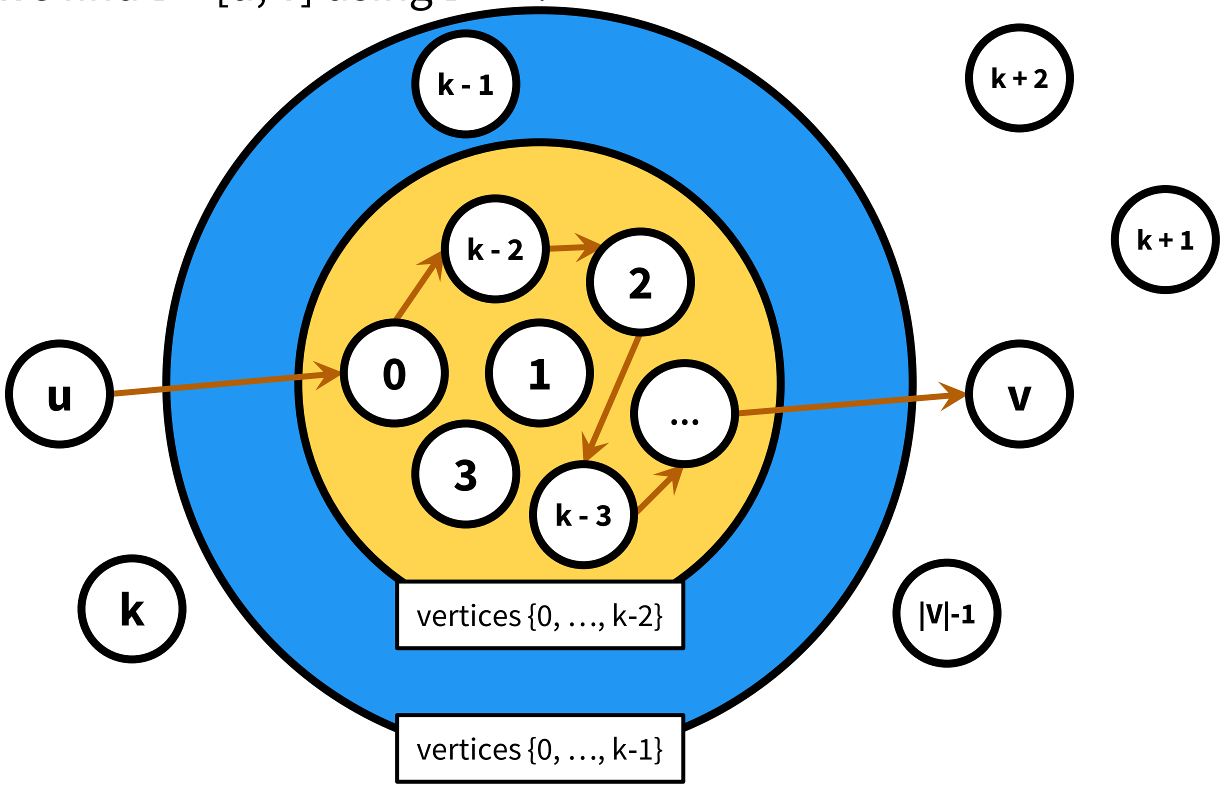
How might we find $D^{(k)}[u, v]$ using $D^{(k-1)}$?



Floyd-Warshall Algorithm

We can represent it more graphically.

- $D^{(k)}[u, v]$ is the cost of the shortest path from u to v, such that all of the internal vertices on the path are in the set of vertices {0, …, k-1}.
- How might we find $D^{(k)}[u, v]$ using $D^{(k-1)}$?



Floyd-Warshall Algorithm

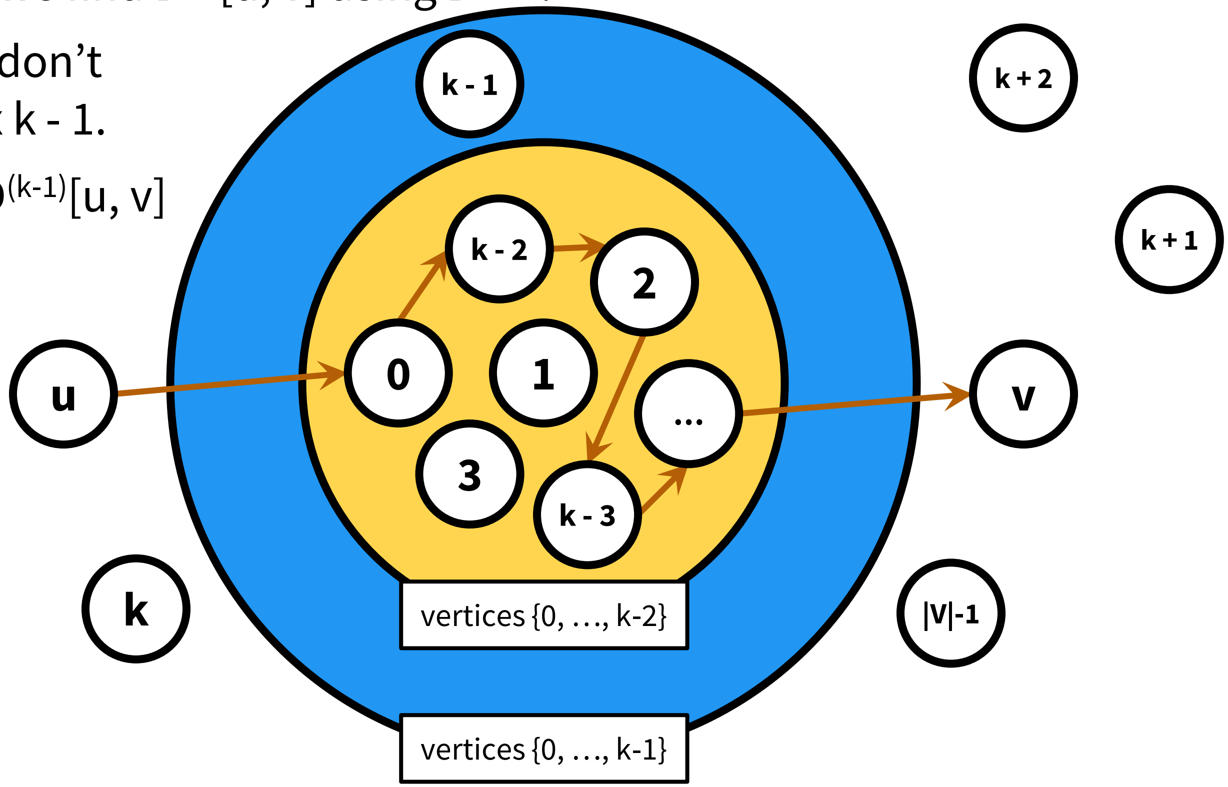
We can represent it more graphically.

$D^{(k)}[u, v]$ is the cost of the shortest path from u to v, such that all of the internal vertices on the path are in the set of vertices {0, …, k-1}.

How might we find $D^{(k)}[u, v]$ using $D^{(k-1)}$?

Case 1: we don't need vertex k - 1.

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



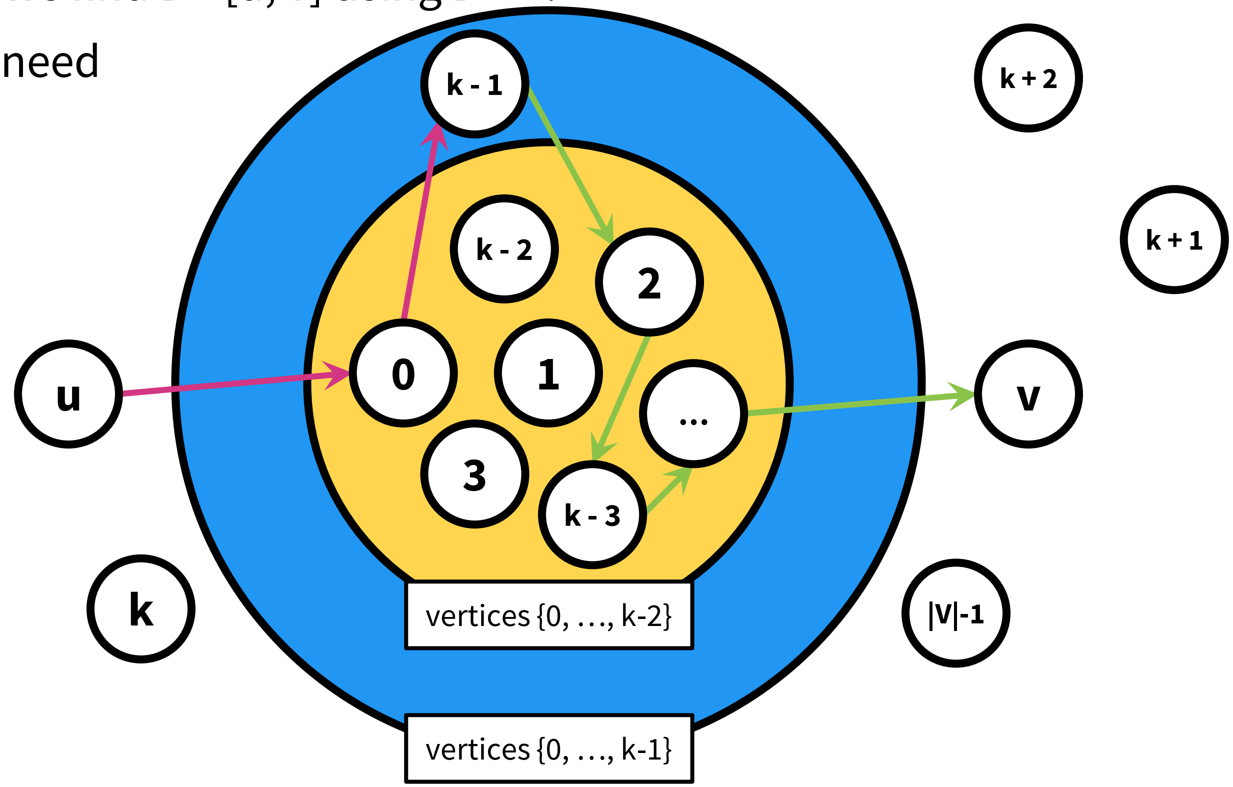
Floyd-Warshall Algorithm

We can represent it more graphically.

$D^{(k)}[u, v]$ is the cost of the shortest path from u to v, such that all of the internal vertices on the path are in the set of vertices {0, …, k-1}.

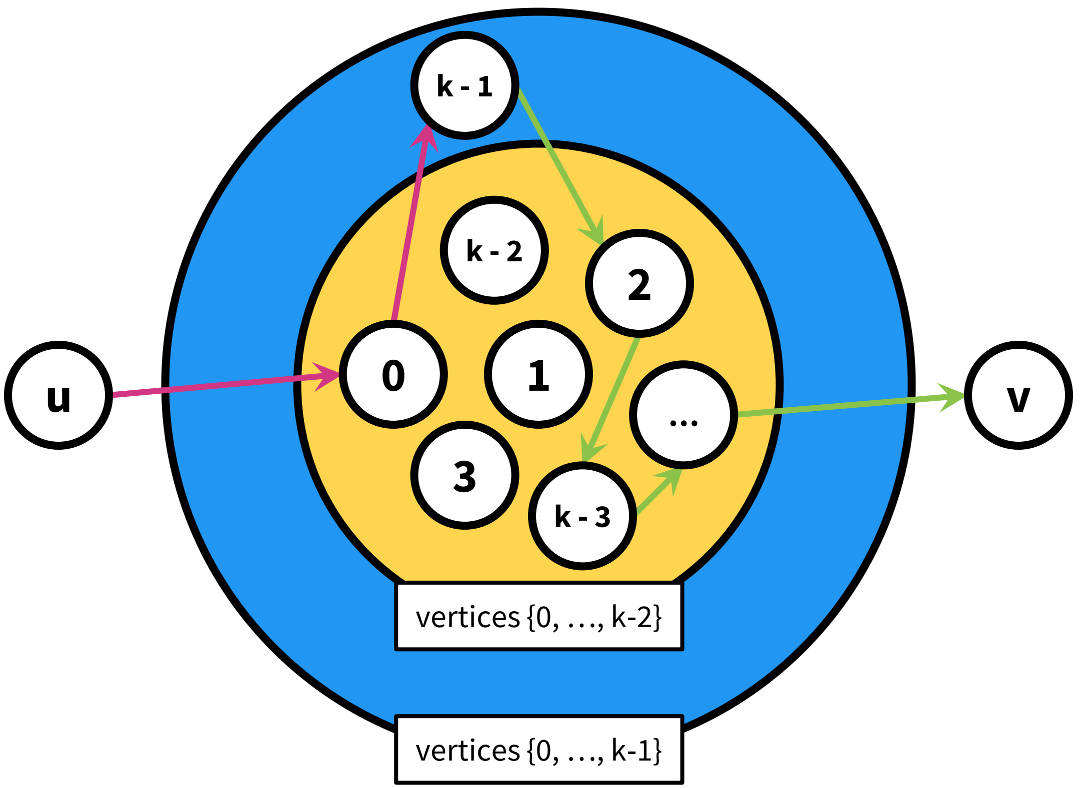
How might we find $D^{(k)}[u, v]$ using $D^{(k-1)}$?

Case 2: we need vertex k - 1.



Floyd-Warshall Algorithm

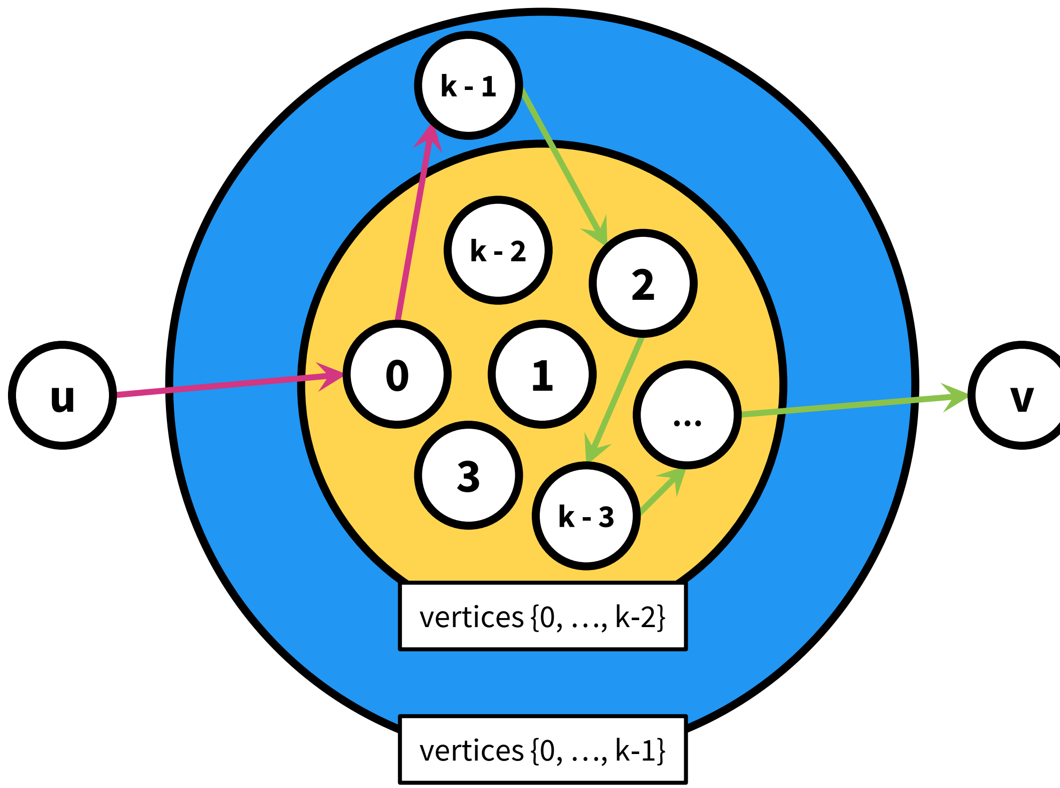
Case 2, cont.: we need vertex k - 1.



Floyd-Warshall Algorithm

Case 2, cont.: we need vertex k - 1.

If there are no negative cycles, then the shortest path from u to v through {0, …, k-1} is simple.

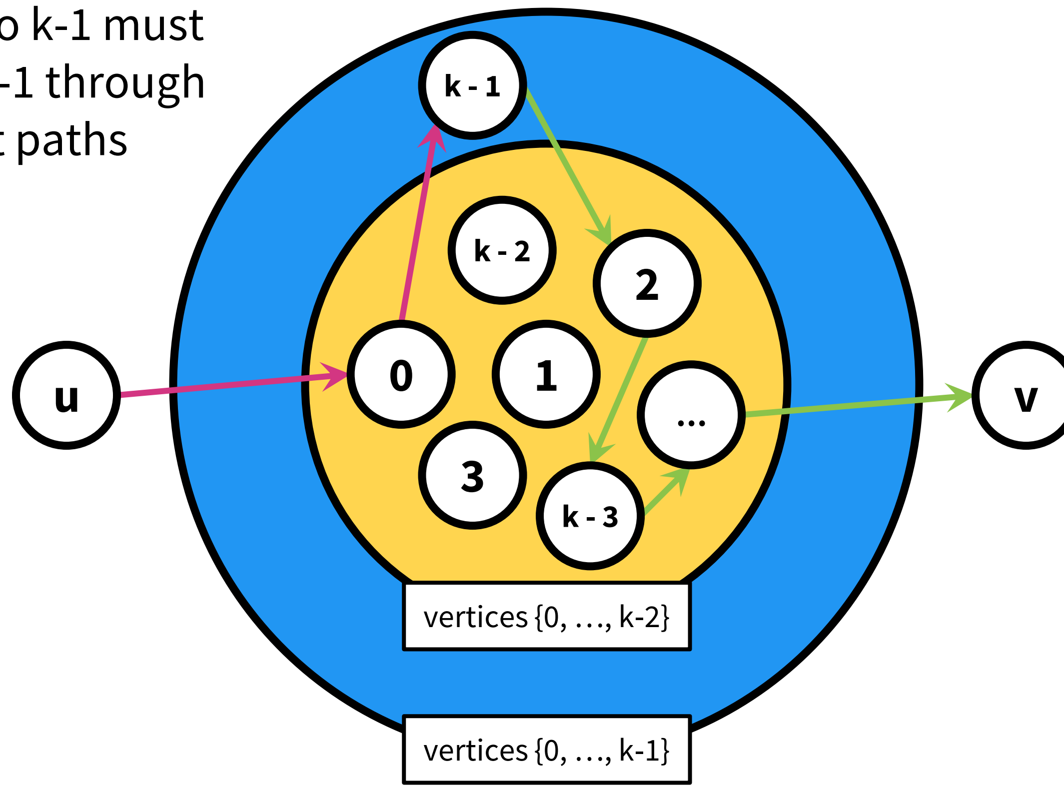


Floyd-Warshall Algorithm

Case 2, cont.: we need vertex k - 1.

If there are no negative cycles, then the shortest path from u to v through {0, …, k-1} is simple.

If the shortest path from u to v needs vertex k - 1, then **the subpath** from u to k-1 must be the shortest path from u to k-1 through {0, …, k-2} (subpaths of shortest paths are shortest paths).



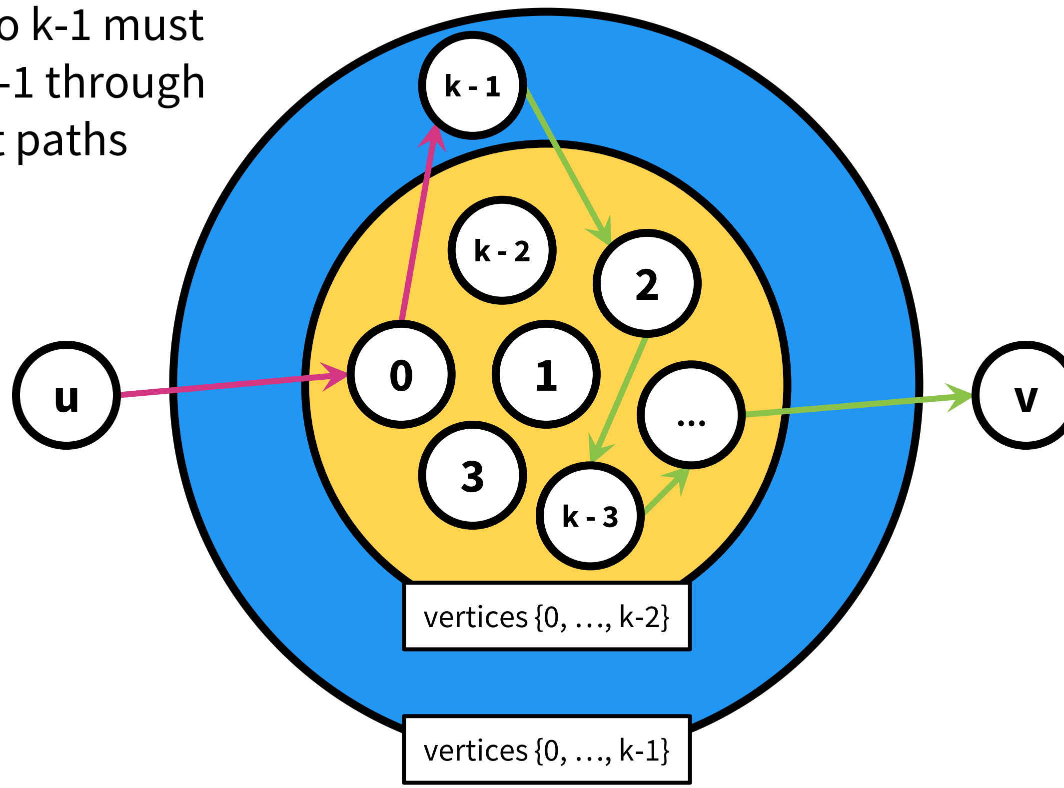
Floyd-Warshall Algorithm

Case 2, cont.: we need vertex k - 1.

If there are no negative cycles, then the shortest path from u to v through {0, …, k-1} is simple.

If the shortest path from u to v needs vertex k - 1, then **the subpath** from u to k-1 must be the shortest path from u to k-1 through {0, …, k-2} (subpaths of shortest paths are shortest paths).

This looks like our inductive hypothesis :)



Floyd-Warshall Algorithm

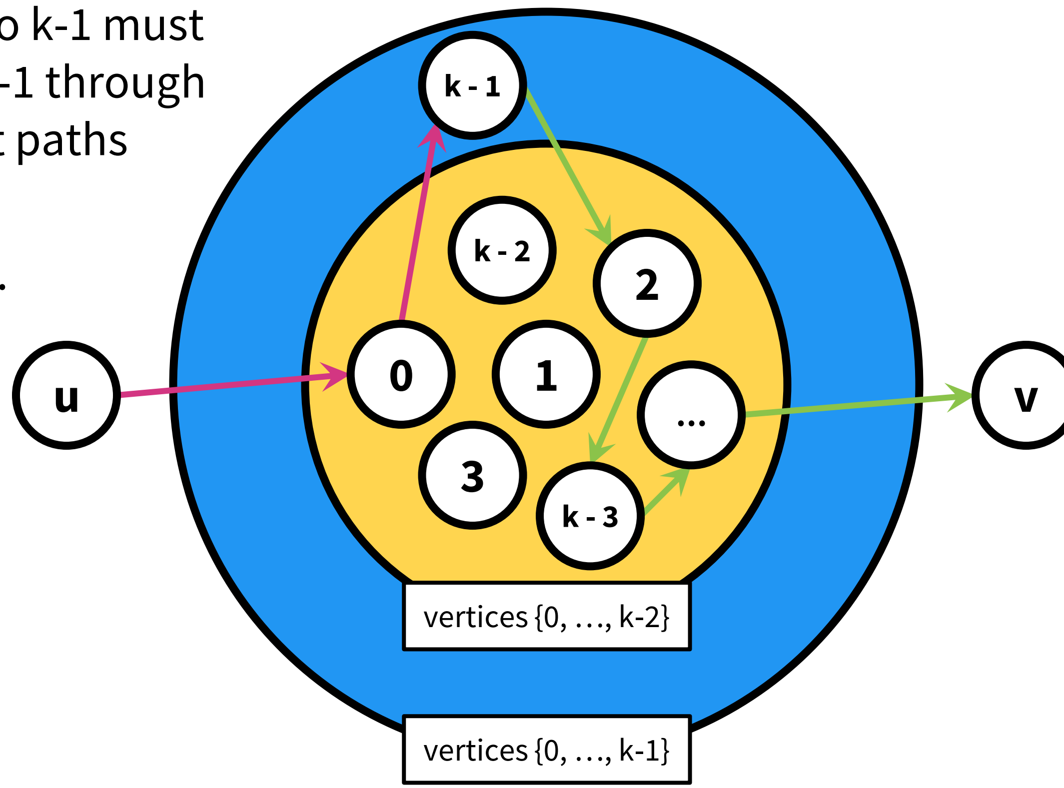
Case 2, cont.: we need vertex k - 1.

If there are no negative cycles, then the shortest path from u to v through {0, …, k-1} is simple.

This looks like our inductive hypothesis :)

If the shortest path from u to v needs vertex k - 1, then **the subpath** from u to k-1 must be the shortest path from u to k-1 through {0, …, k-2} (subpaths of shortest paths are shortest paths).

Same for **the path** from k-1 to v.



Floyd-Warshall Algorithm

Case 2, cont.: we need vertex k - 1.

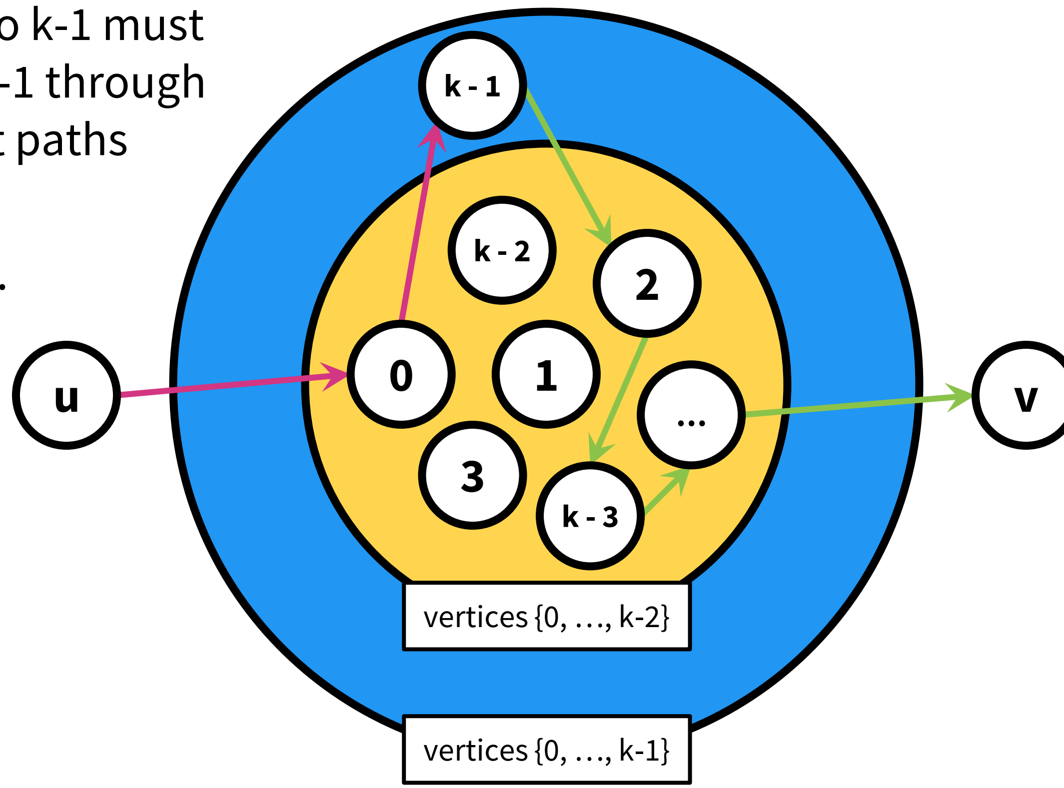
If there are no negative cycles, then the shortest path from u to v through {0, …, k-1} is simple.

This looks like our inductive hypothesis :)

If the shortest path from u to v needs vertex k - 1, then **the subpath** from u to k-1 must be the shortest path from u to k-1 through {0, …, k-2} (subpaths of shortest paths are shortest paths).

Same for **the path** from k-1 to v.

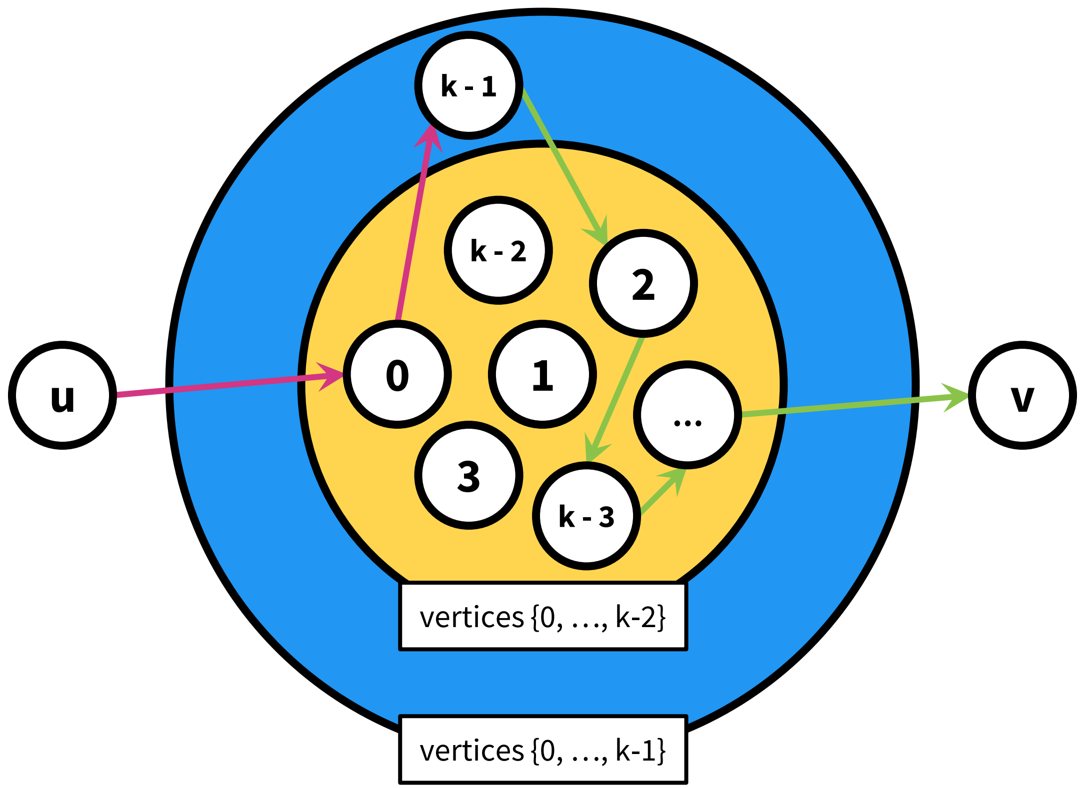
$D^{(k)}[u, v] =$
$D^{(k-1)}[u, k-1] + D^{(k-1)}[k-1, v]$



Floyd-Warshall Algorithm

How might we find $D^{(k)}[u, v]$ using $D^{(k-1)}$?

$D^{(k)}[u, v] = \min\{ \qquad , \qquad \}$

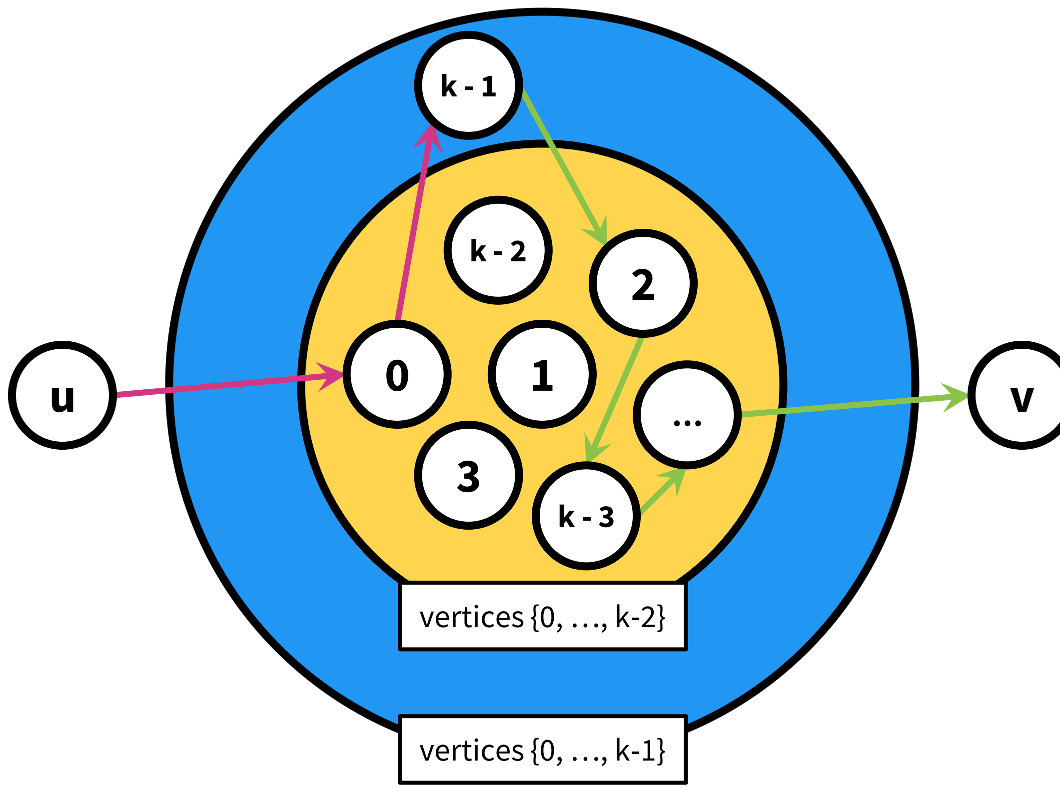


Floyd-Warshall Algorithm

How might we find $D^{(k)}[u, v]$ using $D^{(k-1)}$?

$D^{(k)}[u, v] = \min\{D^{(k-1)}[u, v],\qquad\qquad\qquad\qquad\}$

Case 1

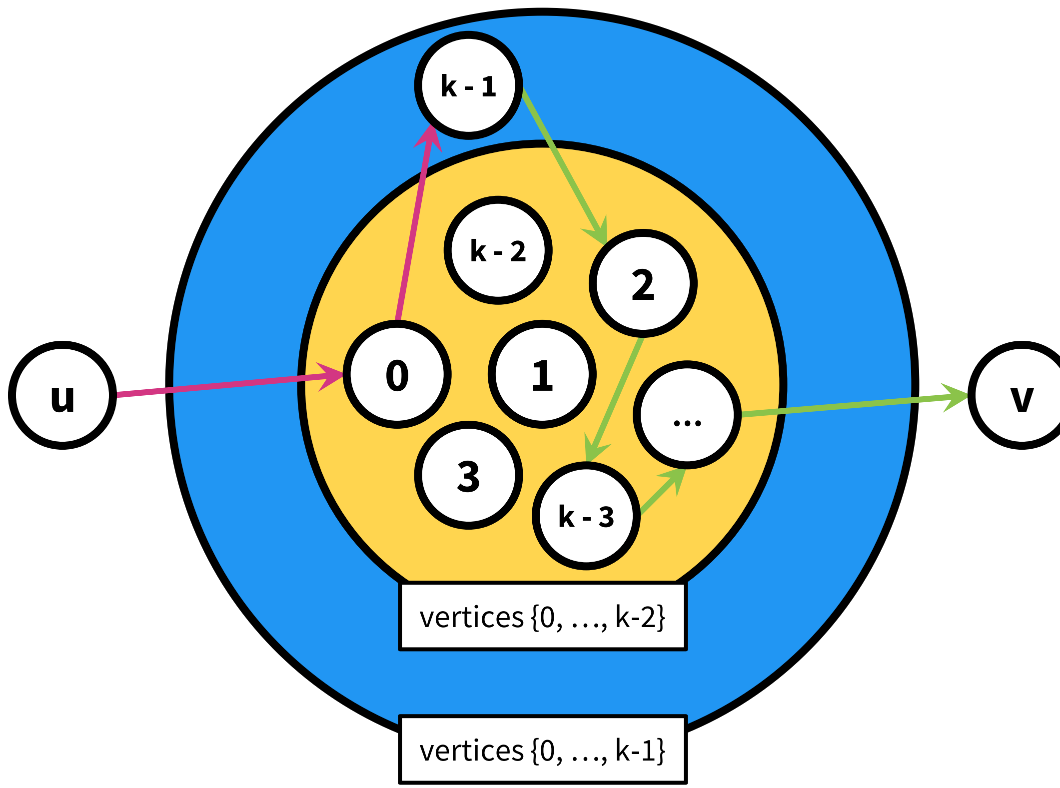


Floyd-Warshall Algorithm

How might we find $D^{(k)}[u, v]$ using $D^{(k-1)}$?

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

Case 1 **Case 2**



Floyd-Warshall Algorithm

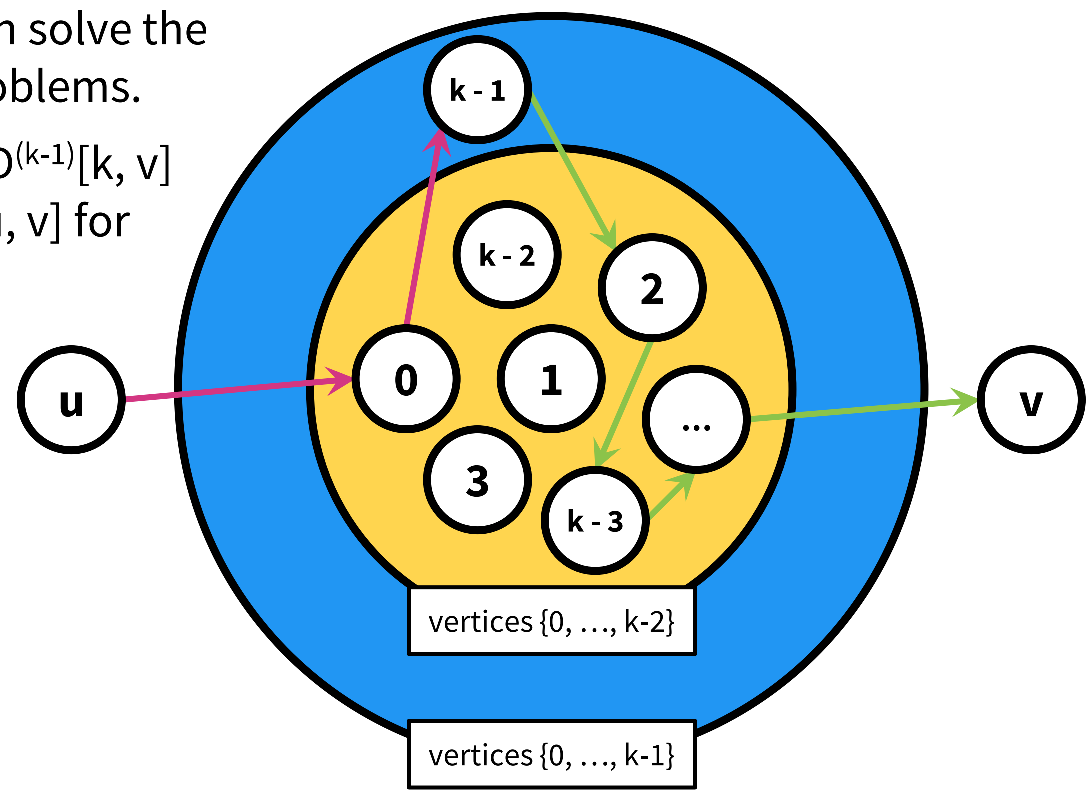
How might we find $D^{(k)}[u, v]$ using $D^{(k-1)}$?

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

Case 1 **Case 2**

Optimal substructure We can solve the big problem using smaller problems.

Overlapping sub-problems $D^{(k-1)}[k, v]$ can be used to compute $D^{(k)}[u, v]$ for lots of different u's.



Floyd-Warshall Algorithm

- Floyd-Warshall can detect negative cycles.
  - If there's a negative cycle, then there's a path from v to v that has cost < 0.
  - How do we check for this condition? 🤔

Floyd-Warshall Algorithm

Floyd-Warshall can detect negative cycles.

- If there's a negative cycle, then there's a path from v to v that has cost < 0.
- How do we check for this condition? 🤔 We can just check $D^{(|V|)}[v, v] < 0$ at the end of the algorithm.

Graph Algorithms

	Dijkstra	Bellman-Ford	Floyd-Warshall
Problem	Single source shortest path	Single source shortest path	All pairs shortest path
Runtime	$O(\lvert E\rvert+\lvert V\rvert\log(\lvert V\rvert))$ **worst-case** with a fibonacci heap	$O(\lvert V\rvert\lvert E\rvert)$ **worst-case**	$O(\lvert V\rvert^3)$ **worst case**
Strengths	---	Works on graphs with negative edge-weights; also can detect negative cycles	Works on graphs with negative edge-weights; also can detect negative cycles
Weaknesses	Might not work on graphs with negative edge-weights	---	---