All-Pairs Shortest Paths

[Erickson, chapter 9]

- Recap: Algorithms for Single-Source Shortest Path
- All-Pairs Shortest Paths
  - Algorithm 1: Lots of Single Sources
  - Algorithm 2: Basic Dynamic Programming
  - Algorithm 3: Divide & Conquer Dynamic Programming
  - Algorithm 4: Floyd-Warshall's Algorithm

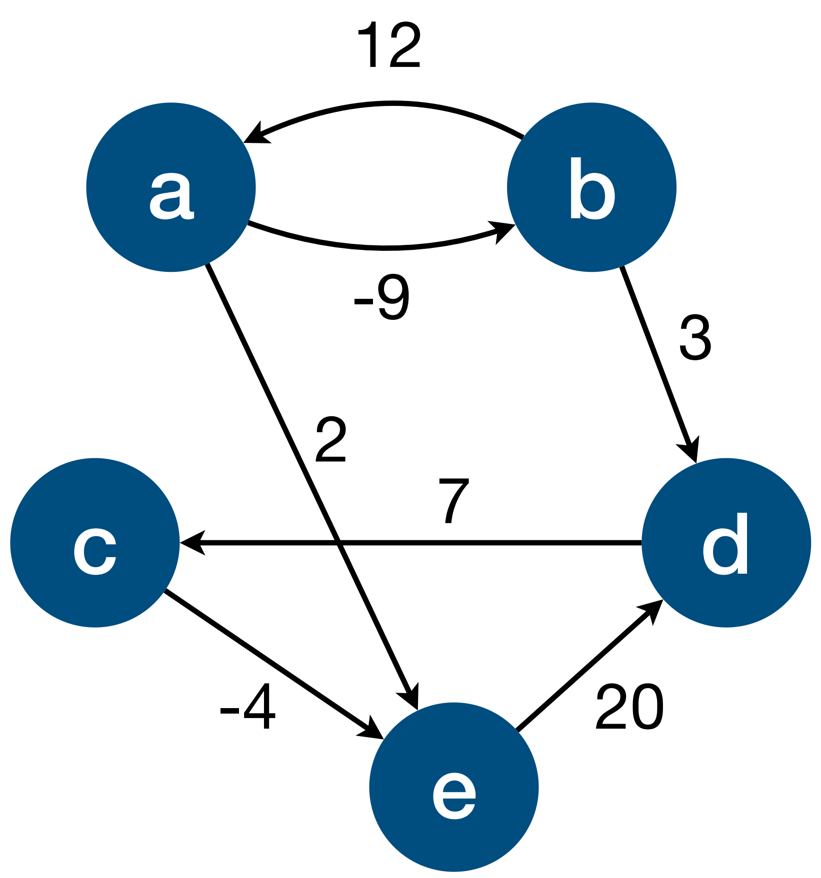
Holger Dell

All-Pairs Shortest Paths
[Erickson, chapter 9]

- Recap: Algorithms for Single-Source Shortest Path
- All-Pairs Shortest Paths
  - Algorithm 1: Lots of Single Sources
  - Algorithm 2: Basic Dynamic Programming
  - Algorithm 3: Divide & Conquer Dynamic Programming
  - Algorithm 4: Floyd-Warshall's Algorithm

Representation of Directed, Weighted Graphs

Recap



Drawing

	a	b	c	d	e
a	0	-9	0	0	2
b	12	0	0	3	0
c	0	0	0	0	-4
d	0	0	7	0	0
e	0	0	0	20	0

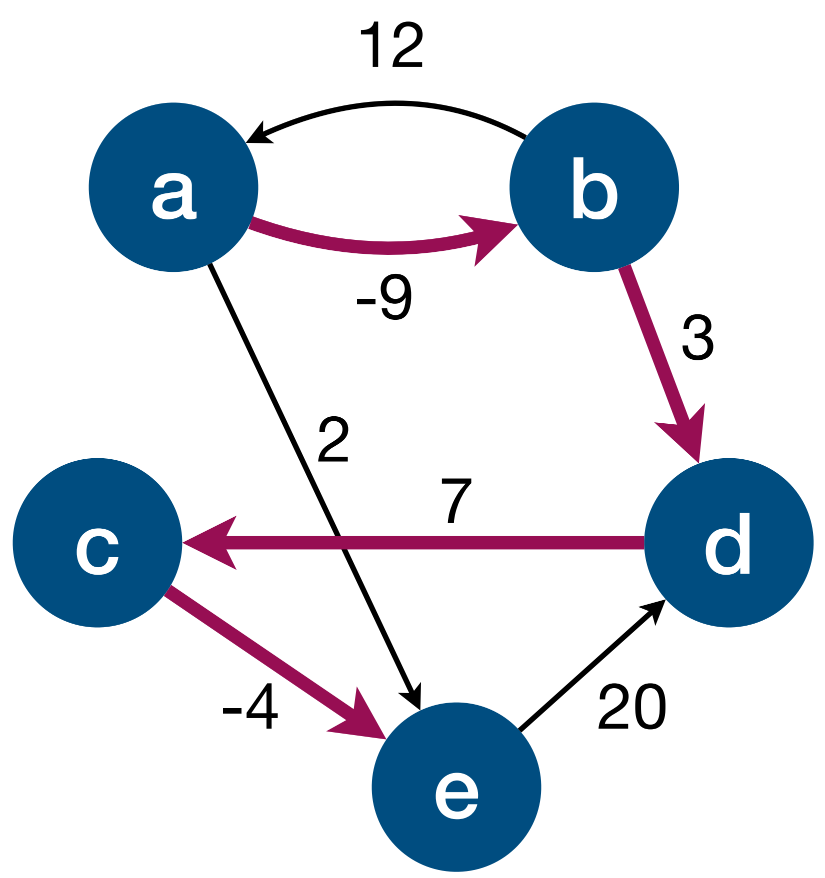
Weighted Adjacency Matrix

```
a b -9
a e 2
b a 12
b d 3
c e -4
d c 7
e d 20
```

Edge List

Single-Source Shortest Paths (SSSP)

Goal: Compute the shortest paths from s to all vertices



Path from a to e of length -3

Shortest paths from a:
(0, -9, 1, -6, -3)

Shortest paths from a:
dist(a, a) = 0
dist(a, b) = -9
dist(a, c) = 1
dist(a, d) = -6
dist(a, e) = -3

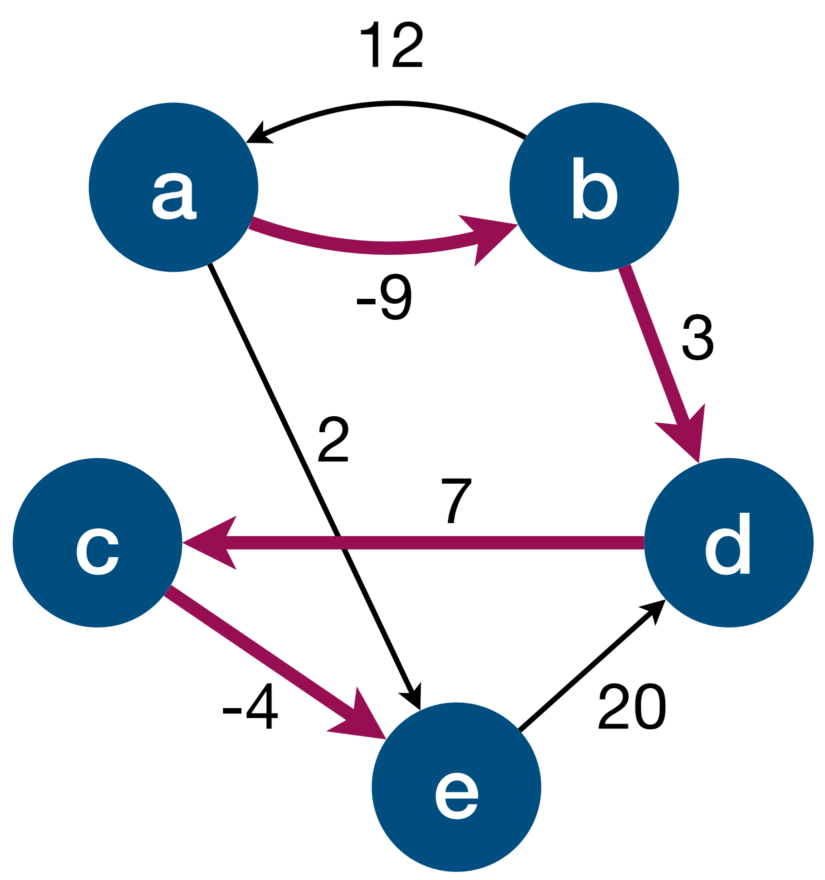
The ***distance*** **dist(s, v)** is the length of a shortest path from s to v.

weights
edges
source
vertices

Goal of `SSSP(V, E, w, s)`:

Compute the vector (dist(s,v) : for all v)

Single-Source Shortest Paths (SSSP)
Algorithms



Path from a to e of length -3

Shortest paths from a:
(0, -9, 1, -6, -3)

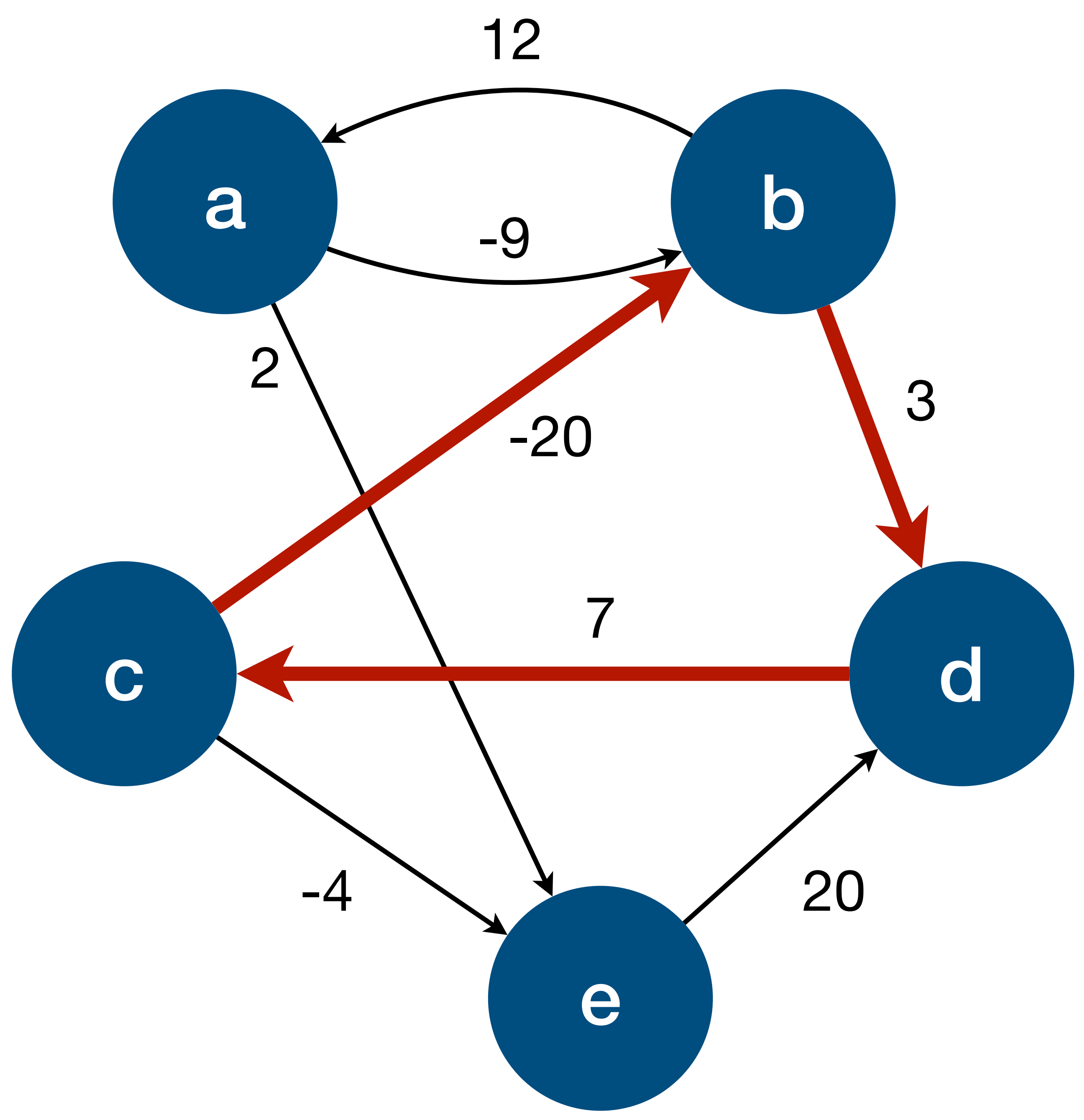
- **Breadth-first search (BFS):**
  - only works if all weights are 1!
  - Time $O(V + E)$

Here we write:
V = number of vertices
E = number of edges

- **Dijkstra's algorithm:**
  - works if all weights are ≥0.
  - Time $O(E \log V)$ slower than BFS!
- **Bellman-Ford's algorithm:**
  - works even if there are negative weights!
  - Time $O(EV)$ slower than Dijkstra!

Single-Source Shortest Paths (SSSP)

The Issue of Negative Cycles



- **bdcb** is a negative cycle!
- The path **abdce** has weight -3.
- The path **abdcbdce** has weight -13.
- The path **abdcbdcbdce** has weight -23.
- ...
- There is no ***shortest*** path from **a** to **e**.

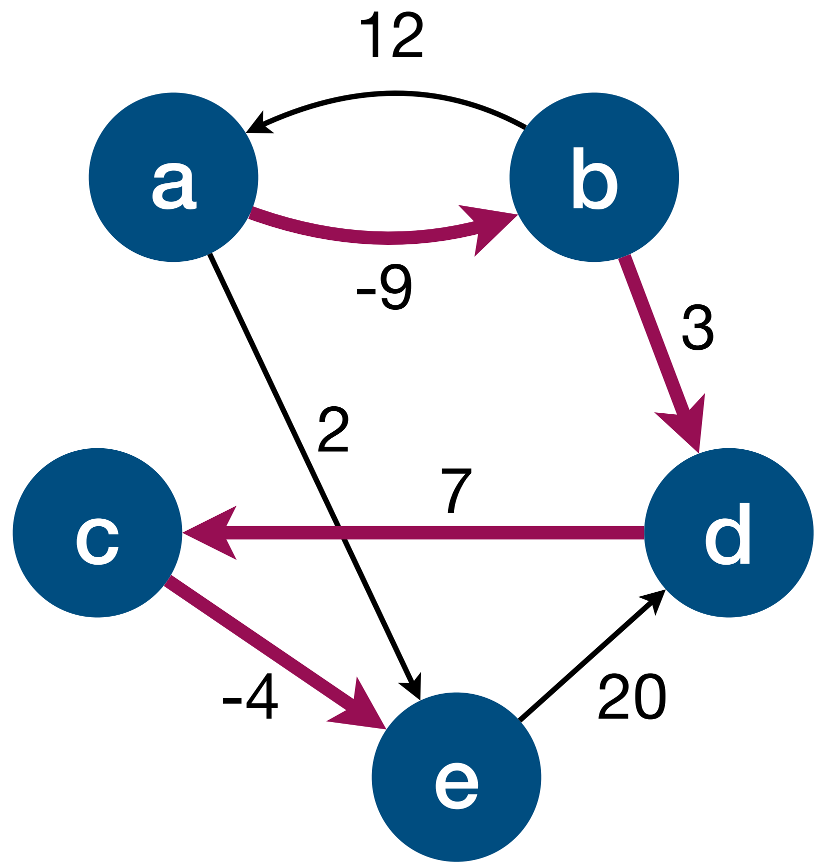
Beware: The concept of "shortest paths"
only makes sense if there is no negative cycle!

All-Pairs Shortest Paths
[Erickson, chapter 9]

- Recap: Algorithms for Single-Source Shortest Path
- All-Pairs Shortest Paths
  - Algorithm 1: Lots of Single Sources
  - Algorithm 2: Basic Dynamic Programming
  - Algorithm 3: Divide & Conquer Dynamic Programming
  - Algorithm 4: Floyd-Warshall's Algorithm

All-Pairs Shortest Paths (APSP)

Goal: Compute shortest paths between ***all*** vertices



Distances can be arranged in a matrix:

	a	b	c	d	e
a	0	-9	1	-6	-3
b	12	0	10	3	6
c	∞	∞	0	16	-4
d	∞	∞	7	0	3
e	∞	∞	27	20	0

Examples:
- **dist(e,c)=27**
- **dist(c,b)=∞**

The goal of APSP is to compute this matrix.

Goal of `APSP(V, E, w)`:

Compute the matrix (dist(u,v) : for all vertices u and v)

Algorithm 1: Lots of Single Sources

[Erickson, chapter 9.2]

- How would we solve APSP with what we already know?

```
ObviousAPSP(V, E, w):
  for every vertex s:
    dist[s, .] = SSSP(V, E, w, s)
```

- **Running Time?**

Algorithm	Weights	Time
V times **BFS**	none	$O(V \cdot E) = O(V^3)$
V times **Dijkstra**	non-negative	$O(V \cdot (E \log V)) = O(V^3 \log V)$
V times **Bellman-Ford**	no negative cycles	$O(V \cdot (EV)) = O(V^4)$

Challenge: Can we achieve $O(V^3)$ when the graph has negative weights?

All-Pairs Shortest Paths
[Erickson, chapter 9]

- Recap: Algorithms for Single-Source Shortest Path
- All-Pairs Shortest Paths
  - Algorithm 1: Lots of Single Sources
  - Algorithm 2: Basic Dynamic Programming
  - Algorithm 3: Divide & Conquer Dynamic Programming
  - Algorithm 4: Floyd-Warshall's Algorithm

Dynamic Programming
Recap from [Erickson, chapter 3.4]

Designing algorithms using **Dynamic Programming** takes two main steps:

1. Solve the problem using a recursive algorithm
   (This step requires creativity.)

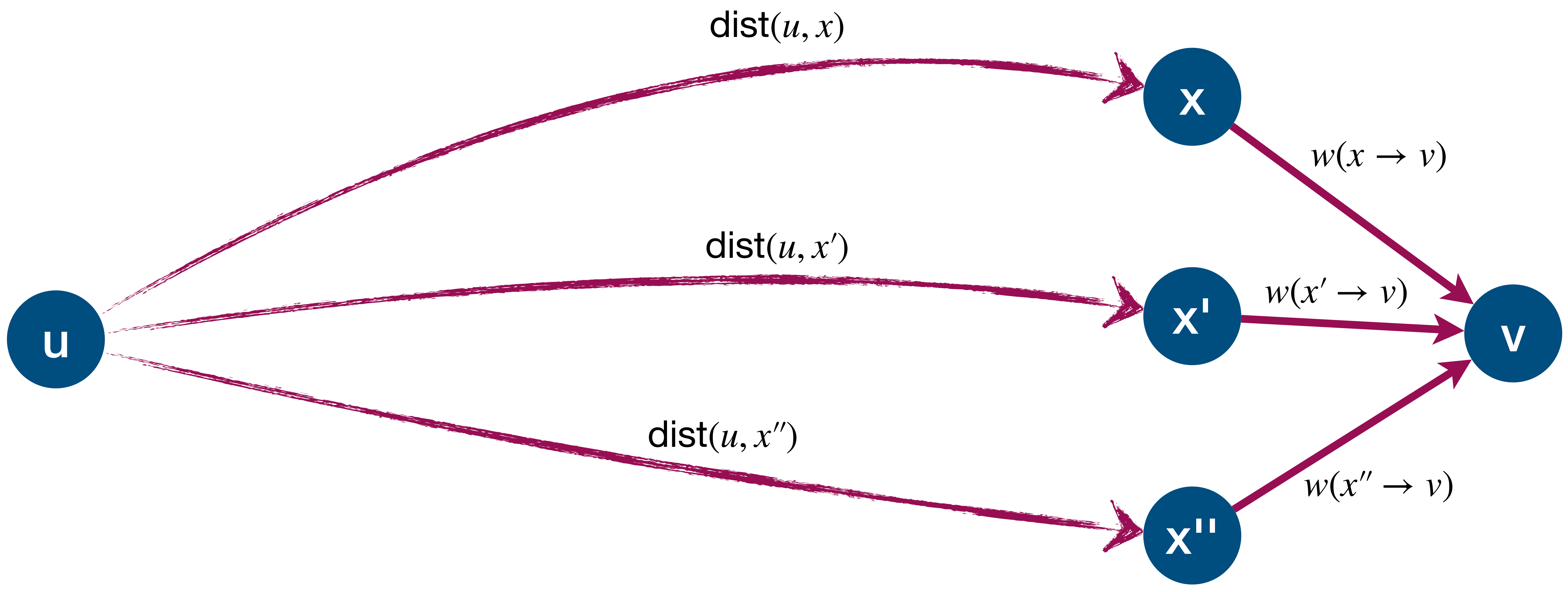
2. Turn the recursive algorithm into a bottom-up iterative algorithm
   (This step is quite mechanical and requires little creativity.)

Challenge. How can we formulate APSP recursively?

Recursive Invariant for Distances

[Erickson, chapter 9.5]

$$\text{dist}(u, v) = \begin{cases} 0 & \text{if } u = v \\ \min_{x \to v} \left(\text{dist}(u, x) + w(x \to v) \right) & \text{otherwise} \end{cases}$$



Recursive Invariant for Distances

[Erickson, chapter 9.5]

$$\text{dist}(u,v) = \begin{cases} 0 & \text{if } u = v \\ \min_{x \to v} \left(\text{dist}(u,x) + w(x \to v) \right) & \text{otherwise} \end{cases}$$

- $\text{dist}(u, v)$ satisfies this recursive invariant
- **Issue.** If the graph has cycles, the recursion never bottoms out!

Refined Recursive Invariant

[Erickson, chapter 9.5]

dist(u, v, l) = length of shortest path from u to v with at most l edges.

$$\text{dist}(u,v,l) = \begin{cases} 0 & \text{if } l = 0 \text{ and } u = v \\ \infty & \text{if } l = 0 \text{ and } u \neq v \\ \min \left\{ \begin{array}{l} \text{dist}(u,v,l-1), \\ \min_{x \to v} \big(\text{dist}(u,x,l-1) + w(x \to v)\big) \end{array} \right\} & \text{otherwise} \end{cases}$$

- This recursive algorithm terminates!

Algorithm 2: Dynamic Programming
[Erickson, chapter 9.5]

- Dynamic Programming.
  Instead of recomputing dist(u, v, l) each time recursively,
  we store it in a table `dist[u,v,l]`:

`dist[*,*,0]` → `dist[*,*,1]` → `dist[*,*,2]` ⋯→ `dist[*,*,l-1]`

base cases can
be initialized
directly.

Algorithm 2: Dynamic Programming
[Erickson, chapter 9.5]

```
ShimbelAPSP(V, E, w):
  initialize dist[u,u,0] = 0 for all vertices u.
  initialize dist[u,v,0] = ∞ for all vertices u and v with u≠v.

  for l from 1 to V-1:
    for all vertices u:
      for all vertices v with u≠v:
        set dist[u,v,l] = min{ dist[u,v,l-1],
                               dist[u,x,l-1] + w(x→v) : for all edges x→v }
```

- Running time: $O(V^4)$
- Space: $O(V^3)$
- We only ever need `dist[*,*,l-1]` to compute `dist[*,*,l]`, so we can simplify the algorithm and use space $O(V^2)$.

Algorithm 2: Dynamic Programming
[Erickson, chapter 9.5]

```
ShimbelAPSP(V, E, w):
  initialize dist[u,u,0] = 0 for all vertices u.
  initialize dist[u,v,0] = ∞ for all vertices u and v with u≠v.

  for l from 1 to V-1:
    for all vertices u:
      for all vertices v with u≠v:
        set dist[u,v,l] = min{ dist[u,v,l-1],
                               dist[u,x,l-1] + w(x→v) : for all edges x→v }
```

- Running time: $O(V^4)$
- Space: $O(V^3)$
- We only ever need `dist[*,*,l-1]` to compute `dist[*,*,l]`, so we can simplify the algorithm and use space $O(V^2)$.

Algorithm 2: Dynamic Programming
[Erickson, chapter 9.5]

```
AllPairsBellmanFord(V, E, w):
  initialize dist[u,u  ] = 0 for all vertices u.
  initialize dist[u,v  ] = ∞ for all vertices u and v with u≠v.

  for l from 1 to V-1:
    for all vertices u:
      for all vertices v with u≠v:
        set dist[u,v  ] = min{ dist[u,v    ],
                               dist[u,x    ] + w(x→v) : for all edges x→v }
```

- Running time: $O(V^4)$
- Space: $O(V^3)$
- We only ever need `dist[⋆,⋆,l-1]` to compute `dist[⋆,⋆,l]`, so we can simplify the algorithm and use space $O(V^2)$.

Algorithm 2: Dynamic Programming
[Erickson, chapter 9.5]

```
AllPairsBellmanFord(V, E, w):
  initialize dist[u,u] = 0 for all vertices u.
  initialize dist[u,v] = ∞ for all vertices u and v with u≠v.

  for l from 1 to V-1:
    for all vertices u:
      for all vertices v with u≠v:
        set dist[u,v] = min{ dist[u,v],
                             dist[u,x] + w(x→v) : for all edges x→v }
```

- Running time: $O(V^4)$
- Space: $O(V^2)$

All-Pairs Shortest Paths

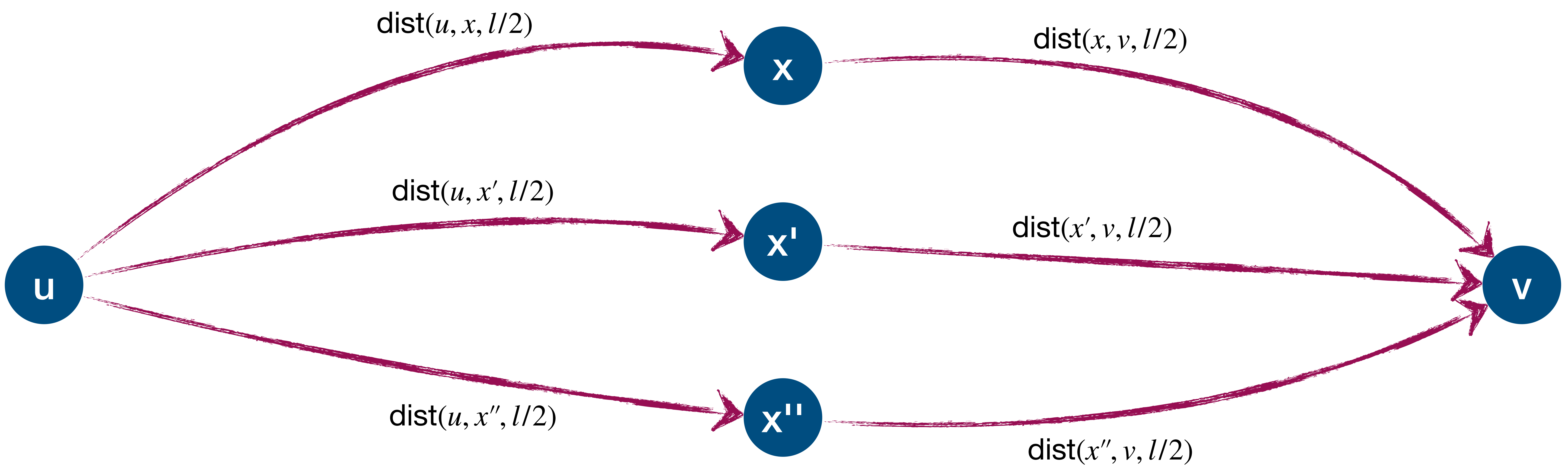
[Erickson, chapter 9]

- Recap: Algorithms for Single-Source Shortest Path
- All-Pairs Shortest Paths
  - Algorithm 1: Lots of Single Sources
  - Algorithm 2: Basic Dynamic Programming
  - Algorithm 3: Divide & Conquer Dynamic Programming
  - Algorithm 4: Floyd-Warshall's Algorithm

Divide and Conquer Recursion

[Erickson, chapter 9.6]

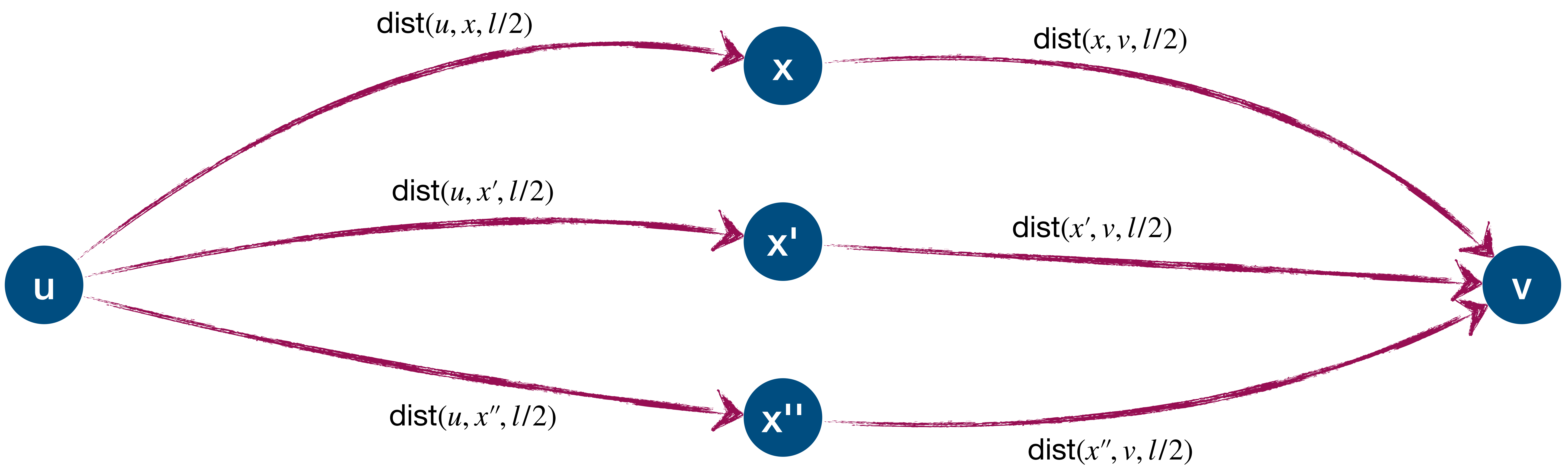
dist(u, v, l) = length of shortest path from u to v with at most l edges.



Divide and Conquer Recursion

[Erickson, chapter 9.6]

$$\text{dist}(u, v, l) = \begin{cases} w(u \to v) & \text{if } l = 1 \\ \min_x \left(\text{dist}(u, x, l/2) + \text{dist}(x, v, l/2)\right) & \text{otherwise} \end{cases}$$



Faster Dynamic Programming using Divide and Conquer

[Erickson, chapter 9.6]

$$\text{dist}(u, v, l) = \begin{cases} w(u \to v) & \text{if } l = 1 \\ \min_x \left(\text{dist}(u, x, l/2) + \text{dist}(x, v, l/2) \right) & \text{otherwise} \end{cases}$$

- This recursive algorithm can be turned into an iterative program.
- The refined program runs in time $O(V^3 \log V)$ and uses space $O(V^2)$.

All-Pairs Shortest Paths
[Erickson, chapter 9]

- Recap: Algorithms for Single-Source Shortest Path
- All-Pairs Shortest Paths
  - Algorithm 1: Lots of Single Sources
  - Algorithm 2: Basic Dynamic Programming
  - Algorithm 3: Divide & Conquer Dynamic Programming
  - Algorithm 4: Floyd-Warshall's Algorithm

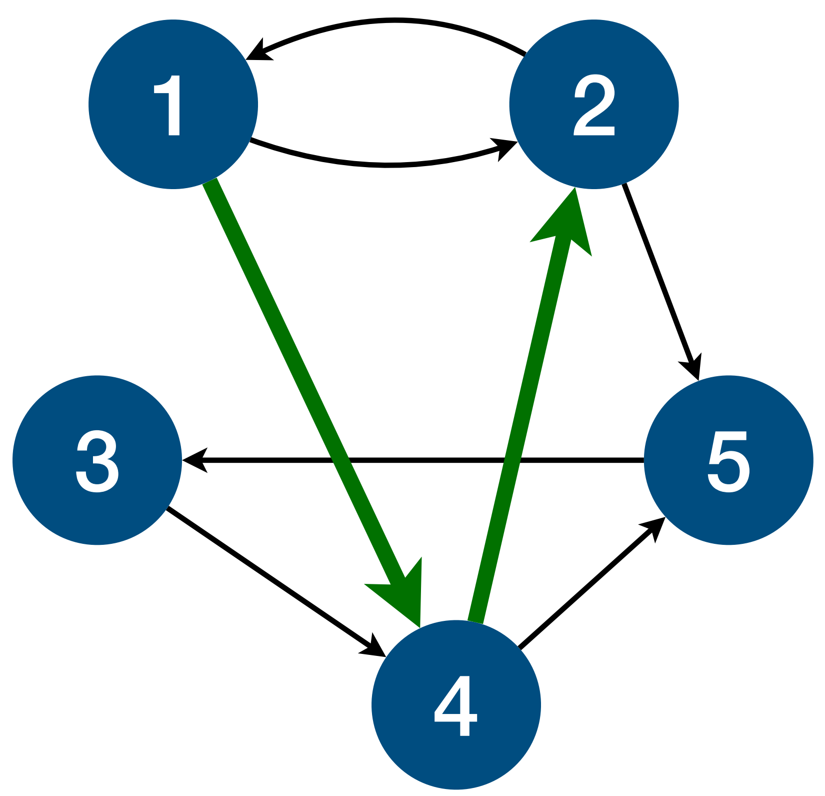
Floyd-Warshall's Recursion

[Erickson, chapter 9.8]

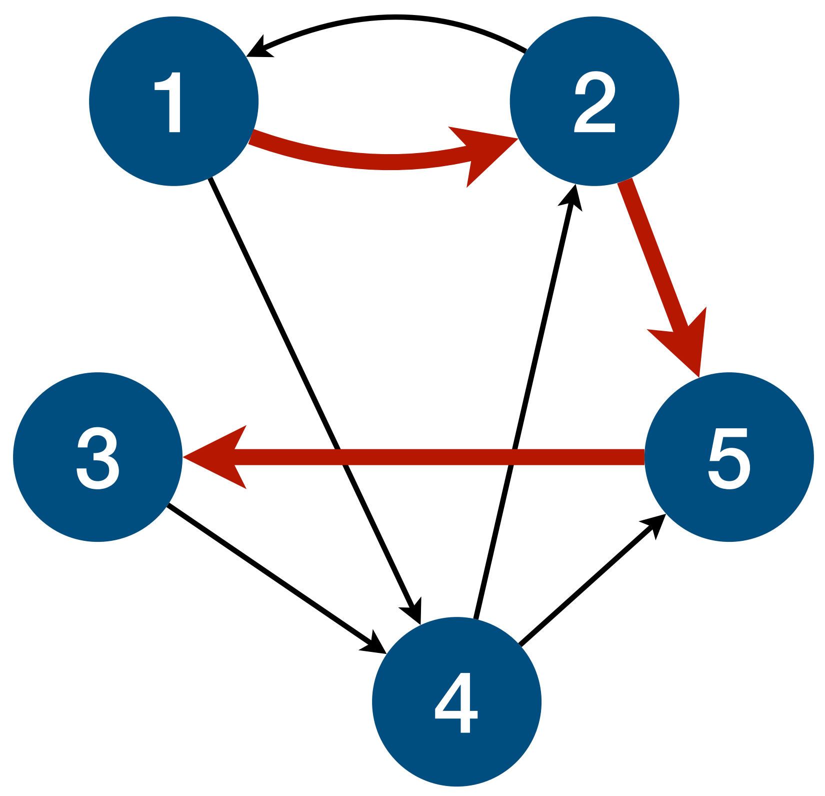
- Let us label the vertices with the numbers 1, 2, 3, ..., V.

$\pi(u, v, r)$ = length of shortest path from u to v
that may only use intermediate vertices numbered $\leq r$.

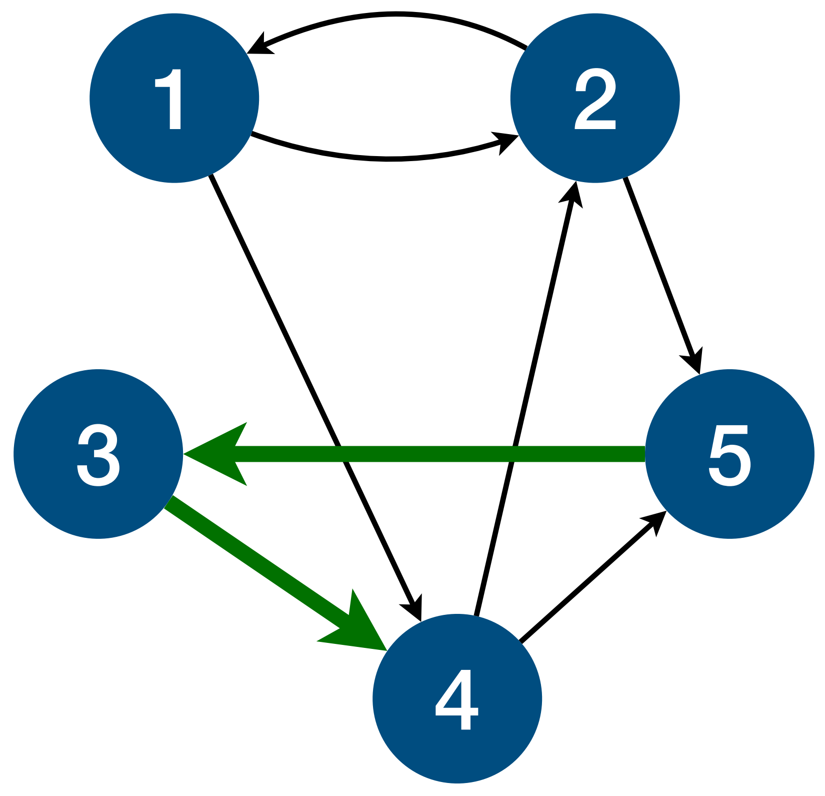
- Examples:



valid path for $\pi(1,2,4)$



invalid path for $\pi(1,3,4)$



valid path for $\pi(5,4,4)$

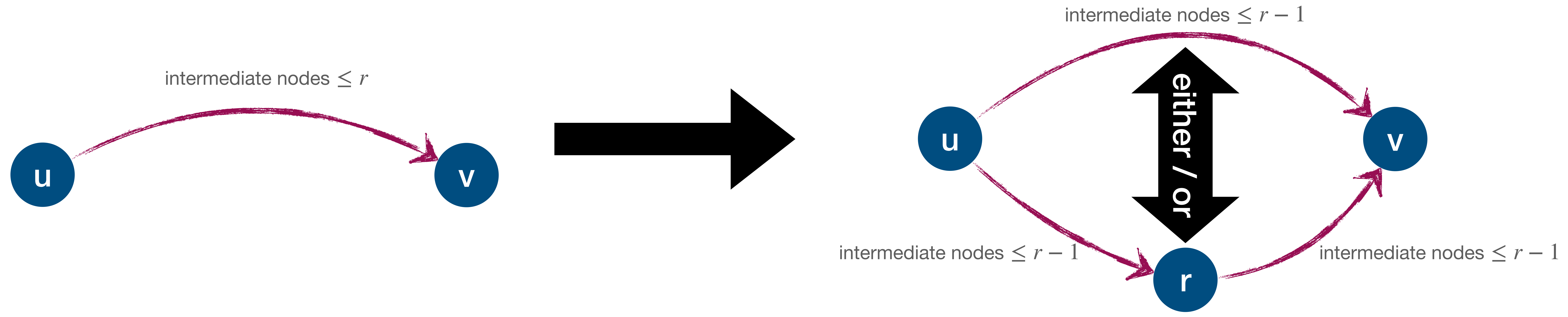
Floyd-Warshall's Recursion

[Erickson, chapter 9.8]

- Let us label the vertices with the numbers 1, 2, 3, ..., V.

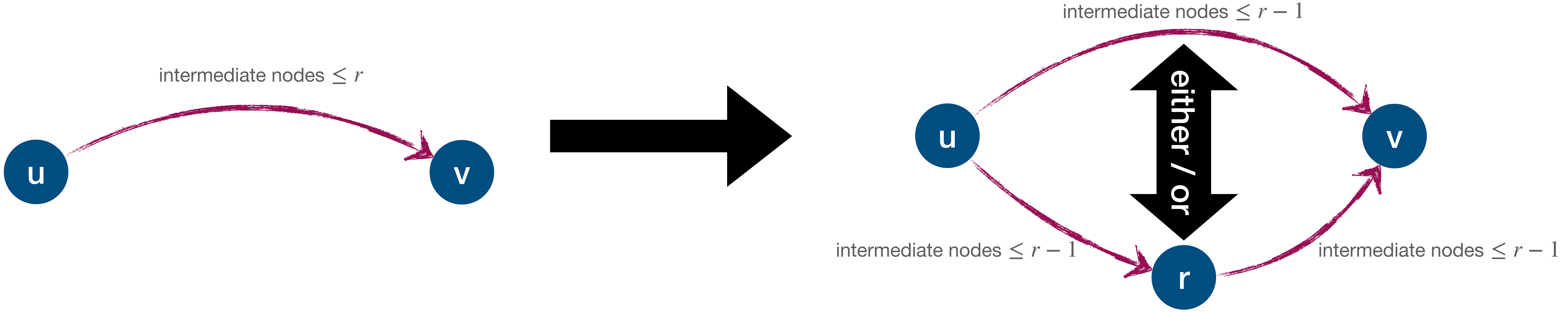
$\pi(u, v, r)$ = length of shortest path from u to v
that may only use intermediate vertices numbered $\leq r$.

- Idea: Shortest valid path for $\pi(u, v, r)$ **either goes through r or not.**



Floyd-Warshall's Recursion
[Erickson, chapter 9.8]

$$\text{dist}(u, v, r) = \begin{cases} w(u \to v) & \text{if } r = 0 \\ \min\big(\text{dist}(u, v, r-1),\ \ \text{dist}(u, r, r-1) + \text{dist}(r, v, r-1)\big) & \text{otherwise} \end{cases}$$



Algorithm 3: Floyd-Warshall's Algorithm
[Erickson, chapter 9.8]

$$\text{dist}(u, v, r) = \begin{cases} w(u \to v) & \text{if } r = 0 \\ \min\left(\text{dist}(u, v, r-1),\ \ \text{dist}(u, r, r-1) + \text{dist}(r, v, r-1)\right) & \text{otherwise} \end{cases}$$

- Dynamic Progamming. Turn this recursive function into an iterative program.
- The final program runs in time $O(V^3)$.

Overview of APSP Algorithms

Algorithm	Weights	Time
V times **BFS**	none	$O(V^3)$
V times **Dijkstra**	non-negative	$O(V^3 \log V)$
V times **Bellman-Ford**	no negative cycles	$O(V^4)$
Basic Dynamic Programming	no negative cycles	$O(V^4)$
Divide and Conquer	no negative cycles	$O(V^3 \log V)$
Floyd-Warshall's Algorithm	no negative cycles	$O(V^3)$

- Floyd-Warshall's algorithm is from 1951-1962.
- Is there a faster algorithm? Humanity does not know yet.
- **APSP-Conjecture:** There is no $O(V^{2.99999})$ time algorithm for APSP.

All-Pairs Shortest Paths

[Erickson, chapter 9]

- Recap: Algorithms for Single-Source Shortest Path
- All-Pairs Shortest Paths
  - Algorithm 1: Lots of Single Sources
  - Algorithm 2: Basic Dynamic Programming
  - Algorithm 3: Divide & Conquer Dynamic Programming
  - Algorithm 4: Floyd-Warshall's Algorithm