# Chapter 25
# All-Pairs Shortest Paths

# Single Source Shortest Paths

- Unweighted graph:
  - BFS - O(V+E)
- Non-negative edge weights:
  - Dijkstra – O(VlogV+E) Fib Heap
- Negative edge weights:
  - Bellman Ford – O(VE)

# All-Pairs Shortest Paths

**Application:** Computing distance table for a road atlas.

| | Atlanta | Chicago | Detroit | … |
|---|---|---|---|---|
| Atlanta | - | 650 | 520 | |
| Chicago | 650 | - | 210 | |
| Detroit | 520 | 210 | - | |
| ⋮ | | | | |

**One Approach:** Run single-source SP algorithm |V| times.

# All-Pairs Shortest Paths

**One Approach:** Run single-source SP algorithm |V| times.

**Nonnegative Edges:** Use Dijkstra.

Time complexity:

$O(V^3)$ with linear array

$O(VE\lg V)$ with binary heap

$O(V^2\lg V + VE)$ with Fibonacci heap

**Negative Edges:** Use Bellman-Ford.

Time Complexity:

$O(V^2E) = O(V^4)$ for dense graphs

Here we can improve!

**Three algorithms in this chapter: Dynamic Programming**

"Repeated Squaring": $O(V^3\lg V)$

Floyd-Warshall: $O(V^3)$

Johnson's: $O(V^2\lg V + VE)$

negative edges allowed, but no negative cycles

# "Repeated Squaring" Algorithm

A dynamic-programming algorithm.

Assume input graph is given by an adjacency matrix.

$W = (w_{ij})$

Let $d_{ij}^{(m)}$ = minimum weight of any path from vertex i to vertex j, containing at most **m** edges.

$$d_{ij}^{(0)} = \begin{cases} 0 & \textbf{if } i = j \\ \infty & \textbf{if } i \neq j \end{cases}$$

dij is the shortest path from i to j using <= m edges

$$\begin{aligned} d_{ij}^{(m)} &= \min(d_{ij}^{(m-1)}, \min\{d_{ik}^{(m-1)} + w_{kj}\}) \\ &= \min_{1 \le k \le n}\{d_{ik}^{(m-1)} + w_{kj}\}, \text{ since } w_{jj} = 0. \end{aligned}$$

Assuming no negative-weight cycles:

$$\delta(i,j) = d_{ij}^{(n-1)} = d_{ij}^{(n)} = d_{ij}^{(n+1)} = \ldots$$

# "Repeated Squaring"

So, given W, we can simply compute a series of matrices $D^{(1)}$, $D^{(2)}$, …, $D^{(n-1)}$ where:

$D^{(1)} = W$

$D^{(m)} = (d_{ij}^{(m)})$

[We'll improve on this shortly.]

```
n := rows[W];
D^(1) := W;
for m := 2 to n – 1 do
    D^(m) := Extend-SP(D^(m-1), W)
end for
return D^(n-1)
```

```
Extend-SP(D, W)
    n := rows[D];
    for i := 1 to n do
        for j :=1 to n do
            d´_ij := ∞;
            for k := 1 to n do
                d´_ij := min(d´_ij, d_ik + w_kj)
            end for
        end for
    end for
    return D´
```

# "Repeated Squaring" and Matrix Mult.

Running time is $O(V^4)$.

Note the similarity to matrix multiplication:

```
Matrix-Multiply(A, B)
    n := rows[A];
    for i := 1 to n do
        for j :=1 to n do
            c_ij := 0;
            for k := 1 to n do
                c_ij := c_ij + a_ik·b_kj
            end for
        end for
    end for
    return C
```

```
Extend-SP(D, W)
    n := rows[D];
    for i := 1 to n do
        for j :=1 to n do
            d´_ij := ∞;
            for k := 1 to n do
                d´_ij := min(d´_ij, d_ik + w_kj)
end for
        end for
    end for
    return D´
```

# Improving the Running Time

Can improve time to $O(V^3 \lg V)$ by computing "products" as follows:

$$D^{(1)} = W$$
$$D^{(2)} = W^2 = W \cdot W$$
$$D^{(4)} = W^4 = W^2 \cdot W^2$$
$$D^{(8)} = W^8 = W^4 \cdot W^4$$
$$\vdots$$
$$D^{(2^{\lceil \lg(n-1) \rceil})} = W^{(2^{\lceil \lg(n-1) \rceil})} = W^{2^{\lceil \lg(n-1) \rceil - 1}} \cdot W^{2^{\lceil \lg(n-1) \rceil - 1}}$$
$$D^{(n-1)} = D^{(2^{\lceil \lg(n-1) \rceil})}$$

Called **repeated squaring**.

Can modify algorithm to use only two matrices.

```
n := rows[W];
D^(1) := W;
m := 1;
while n – 1 > m do
      D^(2m) := Extend-SP(D^(m), D^(m));
      m := 2m
return D^(m)
```

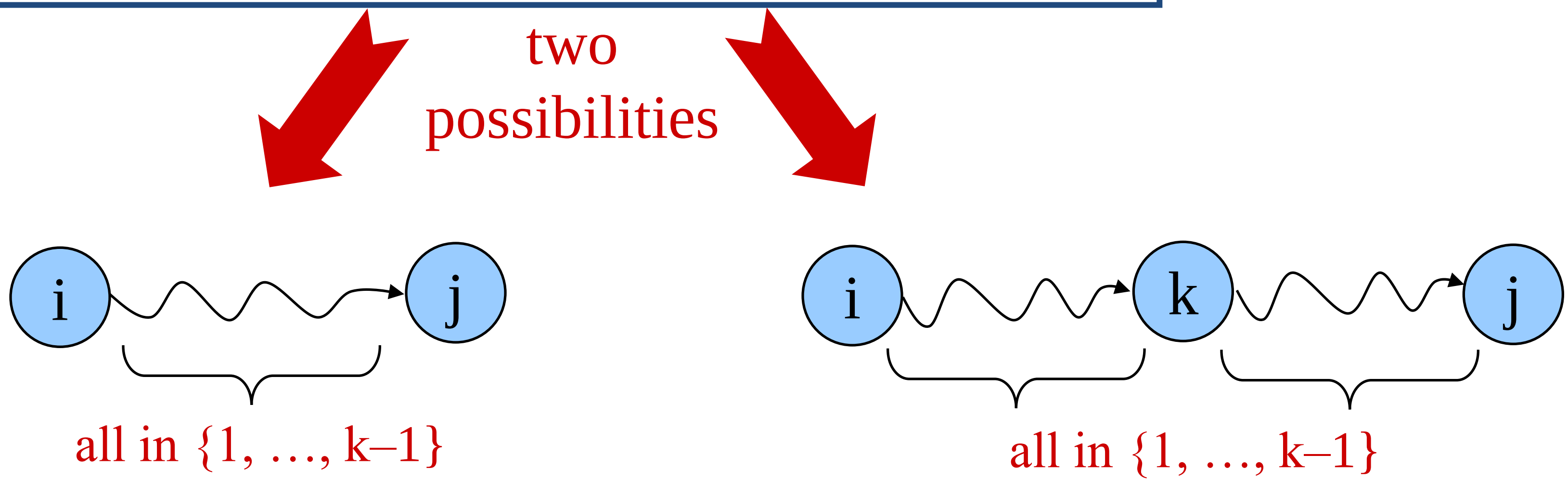
Can I detect negative cycles here?

# Floyd-Warshall Algorithm

- Also dynamic programming, but with different recurrence.

- Let $d_{ij}^{(k)}$ = weight of SP from vertex i to vertex j with all intermediate vertices in the set $\{1, 2, \ldots, k\}$.

$$
d_{ij}^{(k)} = \begin{cases} w_{ij} & \textbf{if } k = 0 \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \textbf{if } k \geq 1 \end{cases}
$$



# Floyd Warshall

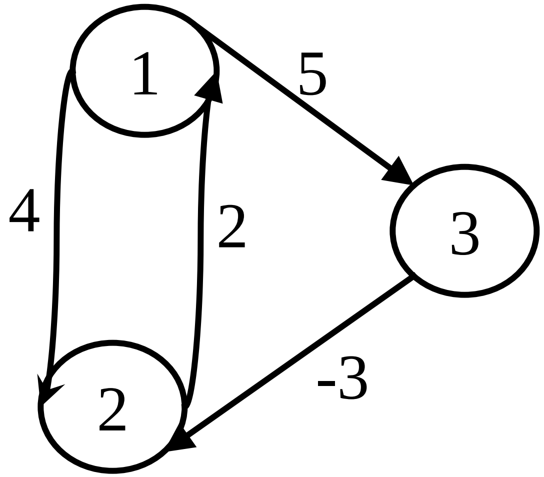
- $\delta(i,j) = d_{ij}^{(n)}$.
- So, want to compute $D^{(n)} = (d_{ij}^{(n)})$

```
n := rows[D];
D^(0) := W;
for k := 1 to n do
for i :=1 to n do
    for j := 1 to n do
        d_ij^(k) := min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
    end for
end for
end for
return D^(n)
```

Can reduce space from $O(V^3)$ to $O(V^2)$ — see Exercise 25.2-4.
Can also modify to compute predecessor matrix.

# Example

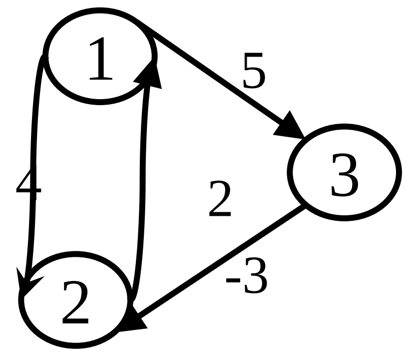
```
n := rows[D];
D(0) := W;
for k := 1 to n do
  for i :=1 to n do
    for j := 1 to n do
      d_ij(k) := min(d_ij(k-1), d_ik(k-1) + d_kj(k-1))
return D(n)
```




$W = D^0 =$

| | 1 | 2 | 3 |
|---|---|---|---|
| 1 | 0 | 4 | 5 |
| 2 | 2 | 0 | ∞ |
| 3 | ∞ | -3 | 0 |

n := rows[D];
$D^{(0)}$ := W;
**for** k := 1 **to** n **do**
  **for** i :=1 **to** n **do**
    **for** j := 1 **to** n **do**
      $d_{ij}^{(k)} := \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$
**return** $D^{(n)}$




$D^0 =$

| | 1 | 2 | 3 |
|---|---|---|---|
| 1 | 0 | 4 | 5 |
| 2 | 2 | 0 | ∞ |
| 3 | ∞ | -3 | 0 |

k = 1
Vertex 1 can be intermediate node

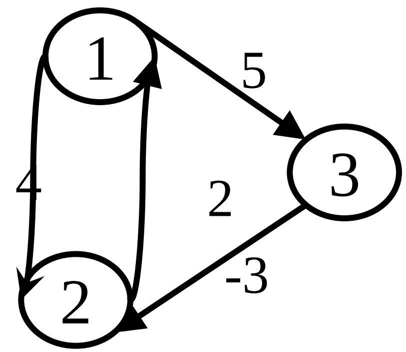
$D^1 =$

| | 1 | 2 | 3 |
|---|---|---|---|
| 1 | 0 | 4 | 5 |
| 2 | 2 | 0 | 7 |
| 3 | ∞ | -3 | 0 |

$D^1[2,3] = \min( D^0[2,3], D^0[2,1]+D^0[1,3] )$
$= \min (\infty, 7) = 7$

$D^1[3,2] = \min( D^0[3,2], D^0[3,1]+D^0[1,2] )$
$= \min (-3,\infty) = -3$

n := rows[D];
$D^{(0)}$ := W;
**for** k := 1 **to** n **do**
  **for** i :=1 **to** n **do**
    **for** j := 1 **to** n **do**
      $d_{ij}^{(k)} := \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$
**return** $D^{(n)}$




$$D^1 = \begin{array}{c|ccc} & 1 & 2 & 3 \\ \hline 1 & 0 & 4 & 5 \\ 2 & 2 & 0 & 7 \\ 3 & \infty & -3 & 0 \end{array}$$

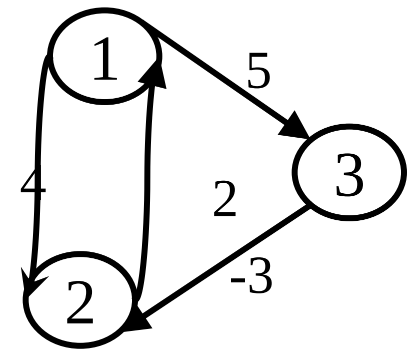
k = 2
Vertices 1, 2 can be intermediate

$$D^2 = \begin{array}{c|ccc} & 1 & 2 & 3 \\ \hline 1 & 0 & 4 & 5 \\ 2 & 2 & 0 & 7 \\ 3 & -1 & -3 & 0 \end{array}$$

$D^2[1,3] = \min( D^1[1,3], D^1[1,2]+D^1[2,3] )$
$= \min (5, 4+7) = 5$

$D^2[3,1] = \min( D^1[3,1], D^1[3,2]+D^1[2,1] )$
$= \min (\propto, -3+2) = -1$

```
n := rows[D];
D^(0) := W;
for k := 1 to n do
  for i :=1 to n do
    for j := 1 to n do
      d_ij^(k) := min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
return D^(n)
```




$$D^2 = \begin{array}{c|ccc} & 1 & 2 & 3 \\ \hline 1 & 0 & 4 & 5 \\ 2 & 2 & 0 & 7 \\ 3 & -1 & -3 & 0 \end{array}$$

k = 3
Vertices 1, 2, 3 can be intermediate

$$D^3[1,2] = \min(D^2[1,2],\ D^2[1,3]+D^2[3,2])$$
$$= \min(4,\ 5+(-3)) = 2$$

$$D^3 = \begin{array}{c|ccc} & 1 & 2 & 3 \\ \hline 1 & 0 & 2 & 5 \\ 2 & 2 & 0 & 7 \\ 3 & -1 & -3 & 0 \end{array}$$

$$D^3[2,1] = \min(D^2[2,1],\ D^2[2,3]+D^2[3,1])$$
$$= \min(2,\ 7+(-1)) = 2$$

# Predecessor Matrix

Let $\pi_{ij}^{(k)}$ = predecessor of vertex j on SP from vertex i with all intermediate vertices in {1, 2, …, k}.

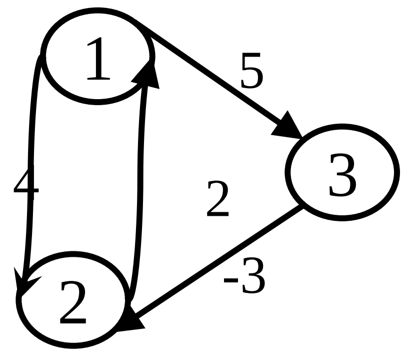
$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \textbf{if } i = j \text{ or } w_{ij} = \infty \\ i & \textbf{otherwise} \end{cases}$$

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \textbf{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} & \textbf{otherwise} \end{cases}$$

**Exercise:** Add computation of $\Pi$ matrix to the algorithm.

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \textbf{if } i = j \text{ or } w_{ij} = \infty \\ i & \textbf{otherwise} \end{cases}$$

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \textbf{if } d_{ij}^{(k-1)} \le d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} & \textbf{otherwise} \end{cases}$$




$D^0 =$

| | 1 | 2 | 3 |
|---|---|---|---|
| 1 | 0 | 4 | 5 |
| 2 | 2 | 0 | ∞ |
| 3 | ∞ | -3 | 0 |

k = 1
Vertex 1 can be intermediate node

$\Pi^{(0)} =$

| | 1 | 2 | 3 |
|---|---|---|---|
| 1 | NIL | 1 | 1 |
| 2 | 2 | NIL | NIL |
| 3 | NIL | 3 | NIL |

$D^1 =$

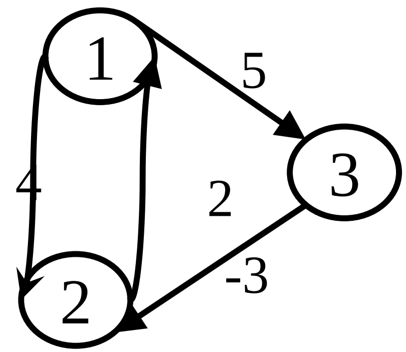
| | 1 | 2 | 3 |
|---|---|---|---|
| 1 | 0 | 4 | 5 |
| 2 | 2 | 0 | 7 |
| 3 | ∞ | -3 | 0 |

$\Pi^{(1)} =$

| | 1 | 2 | 3 |
|---|---|---|---|
| 1 | NIL | 1 | 1 |
| 2 | 2 | NIL | 1 |
| 3 | NIL | 3 | NIL |

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \textbf{if } i = j \text{ or } w_{ij} = \infty \\ i & \textbf{otherwise} \end{cases}$$

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \textbf{if } d_{ij}^{(k-1)} \le d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} & \textbf{otherwise} \end{cases}$$




$$D^1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{ccc} 1 & 2 & 3 \\ \hline 0 & 4 & 5 \\ 2 & 0 & 7 \\ \infty & -3 & 0 \end{array}$$

k = 2
Vertices 1, 2 can be intermediate

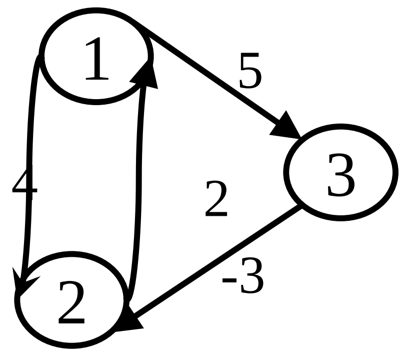
$$\Pi^{(1)} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{ccc} 1 & 2 & 3 \\ \hline \text{NIL} & 1 & 1 \\ 2 & \text{NIL} & 1 \\ \text{NIL} & 3 & \text{NIL} \end{array}$$

$$D^2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{ccc} 1 & 2 & 3 \\ \hline 0 & 4 & 5 \\ 2 & 0 & 7 \\ -1 & -3 & 0 \end{array}$$

$$\Pi^{(2)} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{ccc} 1 & 2 & 3 \\ \hline \text{NIL} & 1 & 1 \\ 2 & \text{NIL} & 1 \\ 2 & 3 & \text{NIL} \end{array}$$

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \textbf{if } i = j \text{ or } w_{ij} = \infty \\ i & \textbf{otherwise} \end{cases}$$

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \textbf{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} & \textbf{otherwise} \end{cases}$$




k = 3
Vertices 1, 2, 3 can be intermediate

$D^2 =$

| | 1 | 2 | 3 |
|---|---|---|---|
| 1 | 0 | 4 | 5 |
| 2 | 2 | 0 | 7 |
| 3 | -1 | -3 | 0 |

$\Pi^{(2)} =$

| | 1 | 2 | 3 |
|---|---|---|---|
| 1 | NIL | 1 | 1 |
| 2 | 2 | NIL | 1 |
| 3 | 2 | 3 | NIL |

$D^3 =$

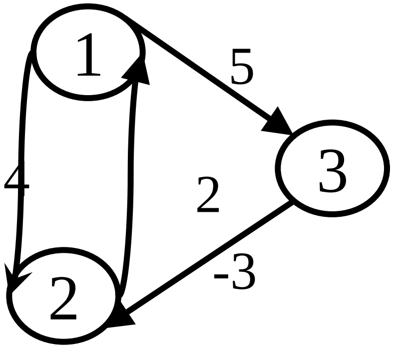
| | 1 | 2 | 3 |
|---|---|---|---|
| 1 | 0 | 2 | 5 |
| 2 | 2 | 0 | 7 |
| 3 | -1 | -3 | 0 |

$\Pi^{(3)} =$

| | 1 | 2 | 3 |
|---|---|---|---|
| 1 | NIL | 3 | 1 |
| 2 | 2 | NIL | 1 |
| 3 | 2 | 3 | NIL |

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \textbf{if } i = j \text{ or } w_{ij} = \infty \\ i & \textbf{otherwise} \end{cases}$$

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \textbf{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} & \textbf{otherwise} \end{cases}$$




k = 3
Vertices 1, 2, 3 can be intermediate

$$D^2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{ccc} 1 & 2 & 3 \\ \hline 0 & 4 & 5 \\ 2 & 0 & 7 \\ -1 & -3 & 0 \end{array}$$

$$\Pi^{(2)} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{ccc} 1 & 2 & 3 \\ \hline \text{NIL} & 1 & 1 \\ 2 & \text{NIL} & 1 \\ 2 & 3 & \text{NIL} \end{array}$$

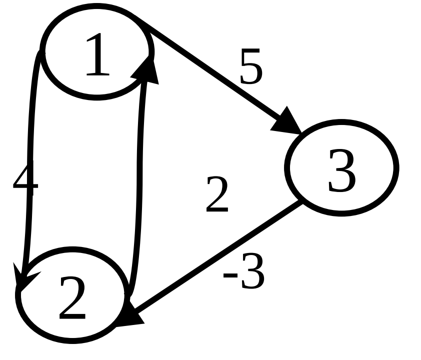
$$D^3 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{ccc} 1 & 2 & 3 \\ \hline 0 & 2 & 5 \\ 2 & 0 & 7 \\ -1 & -3 & 0 \end{array}$$

$$\Pi^{(3)} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{ccc} 1 & 2 & 3 \\ \hline \text{NIL} & 3 & 1 \\ 2 & \text{NIL} & 1 \\ 2 & 3 & \text{NIL} \end{array}$$

# Printing intermediate nodes

```
Path(q, r)
    if (P[ q, r ]!=0)
        Path(q, P[q, r])
        println( "v"+ P[q, r])
        path(P[q, r], r)
        return;
    //no intermediate nodes
    else return
```

$$\Pi^{(3)} = \begin{array}{c|c|c|c|} & 1 & 2 & 3 \\ \hline 1 & \text{NIL} & 3 & 1 \\ \hline 2 & 2 & \text{NIL} & 1 \\ \hline 3 & 2 & 3 & \text{NIL} \\ \hline \end{array}$$



# Johnson's Algorithm

- Makes clever use of Bellman-Ford and Dijkstra to do All-Pairs-Shortest-Paths efficiently on sparse graphs.
- An $O(V^2 \lg V + VE)$ algorithm
- Motivation:
  - By running Dijkstra $|V|$ times, we could do APSP in time $O(V^2 \lg V + VE \lg V)$ or $O(V^2 \lg V + VE)$ (Fib. Dijkstra).
  - This beats $O(V^3)$ (Floyd-Warshall) when the graph is sparse.
- Problem: negative edge weights.

# The Basic Idea

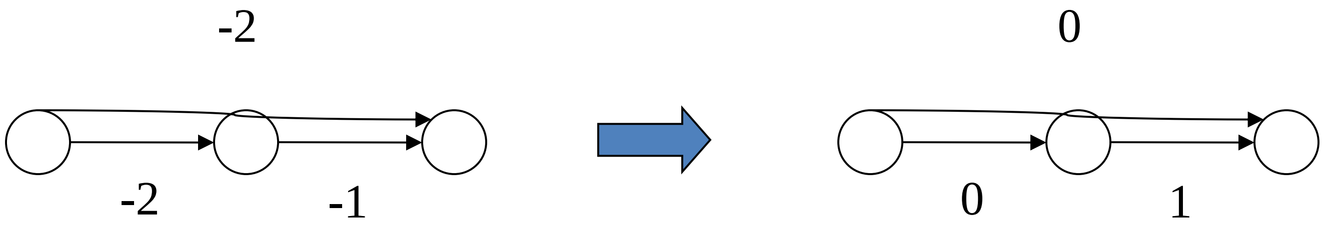
- Reweight the edges so that:
    1. No edge weight is negative.
    2. Shortest paths are preserved. (A shortest path in the original graph is still one in the new, reweighted graph.)
- An obvious attempt: subtract the minimum weight from all the edge weights. E.g. if the minimum weight is -2:

- -2 - -2 = 0
- 3 - -2 = 5

- etc.

# Counterexample

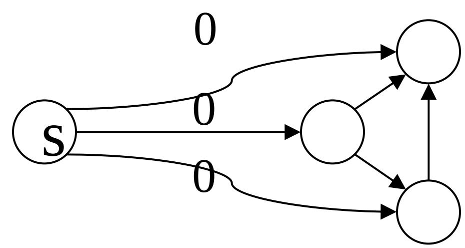
- Subtracting the minimum weight from every weight doesn't work.
- Consider:




- Paths with more edges are unfairly penalized.

# Johnson's Insight

- Add a vertex *s* to the original graph G, with edges of weight 0 to each vertex in G:




- Assign new weights ŵ to each edge as follows:

$$\hat{w}(u, v) = w(u, v) + \delta(s, u) - \delta(s, v)$$

# A General Result about Reweighting

**Define:** $\hat{w}(u,v) = w(u,v) + h(u) - h(v)$, where $h: V \to \Re$.

**Lemma 25.1:** Let $p = \langle v_0, v_1, \ldots, v_k \rangle$. Then, **(i)** $w(p) = \delta(v_0, v_k)$ iff $\hat{w}(p) = \hat{\delta}(v_0, v_k)$. **(ii)** G has a negative-weight cycle using w iff G has a negative-weight cycle using $\hat{w}$.

**Proof of (i):**

$$\hat{w}(p)$$

$$= \sum_{i=1}^{k} \hat{w}(v_{i-1}, v_i)$$

$$= \sum_{i=1}^{k} (w(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i))$$

$$= \sum_{i=1}^{k} w(v_{i-1}, v_i) + h(v_0) - h(v_k)$$

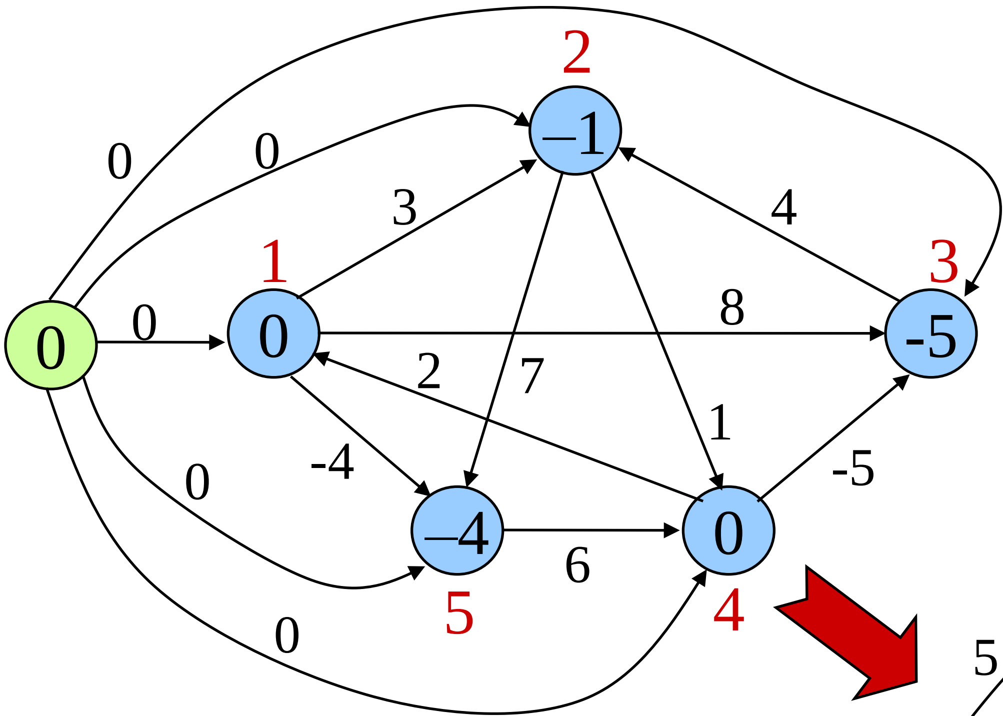
$$= w(p) + h(v_0) - h(v_k)$$

**Proof of (ii):**

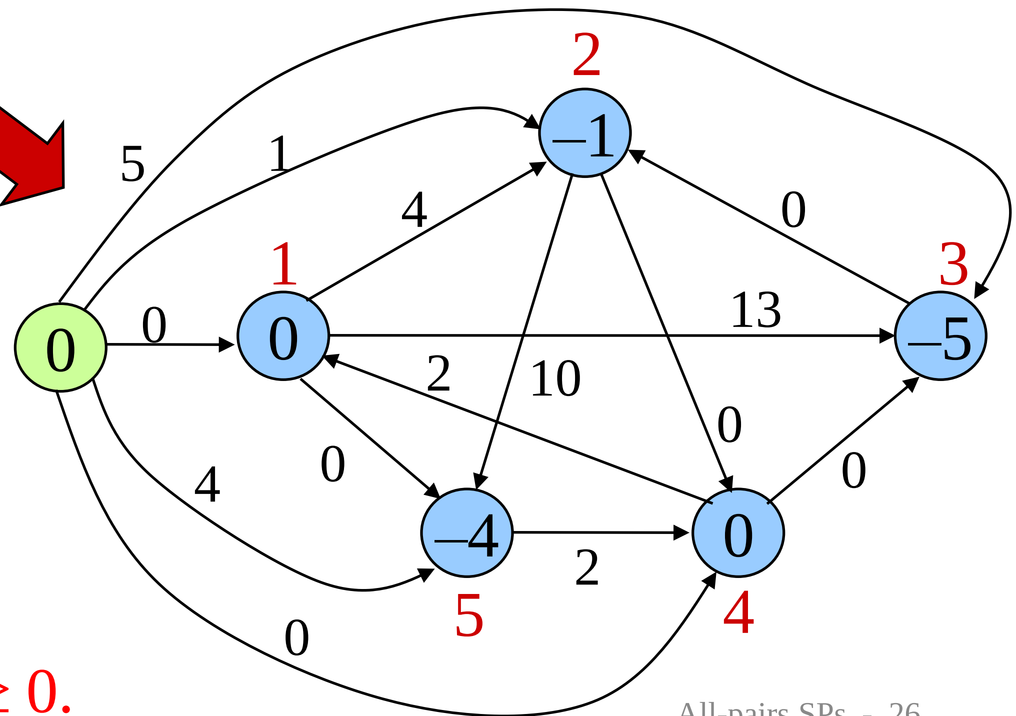
Consider any cycle $c = \langle v_0, v_1, \ldots, v_k \rangle$ where $v_k = v_0$.

$$\hat{w}(c) = w(c) + h(v_0) - h(v_k)$$
$$= w(c).$$

# Reweighting in Johnson's Algorithm




Want to define h s.t.
$\hat{w}(u,v) \ge 0$.




1. $h(v) = \delta(s, v) \ \forall v \in V'$
(computed by Bellman-Ford)

By Lemma 24.10, $\forall\ (u,v) \in E'$:
$h(v) \le h(u) + w(u,v)$.

2. $\hat{w}(u,v) = w(u,v) + h(u) - h(v) \ge 0$.

# Code for Johnson's Algorithm

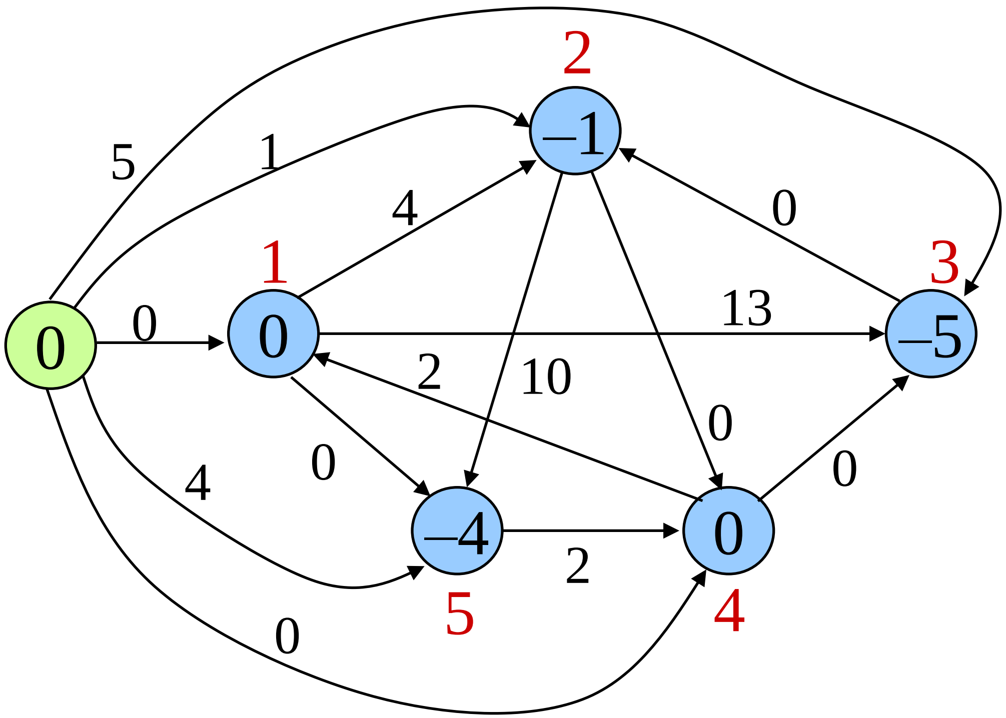
Compute $G'$, where $V[G'] = V[G] \cup \{s\}$, $E[G'] = E[G] \cup \{(s,v): v \in V[G]\}$;
**if** Bellman-Ford($G'$, $w$, $s$) = false **then**
negative-weight cycle
**else**
**for** each $v \in V[G']$ **do**
    set $h(v)$ to $\delta(s, v)$ computed by Bellman-Ford
**end for**
**for** each $(u,v) \in E[G']$ **do**
    $\hat{w}(u,v) := w(u,v) + h(u) - h(v)$
**end for**;
**for** each $u \in V[G]$ **do**
    run Dijkstra($G$, $\hat{w}$, $u$) to compute $\hat{\delta}(u, v)$ for all $v \in V[G]$;
    **for** each $v \in V[G]$ **do**
        $d_{uv} := \hat{\delta}(u, v) + h(v) - h(u)$
    **end for**
**end for**
**end if**

Running time is $O(V^2 \lg V + VE)$.

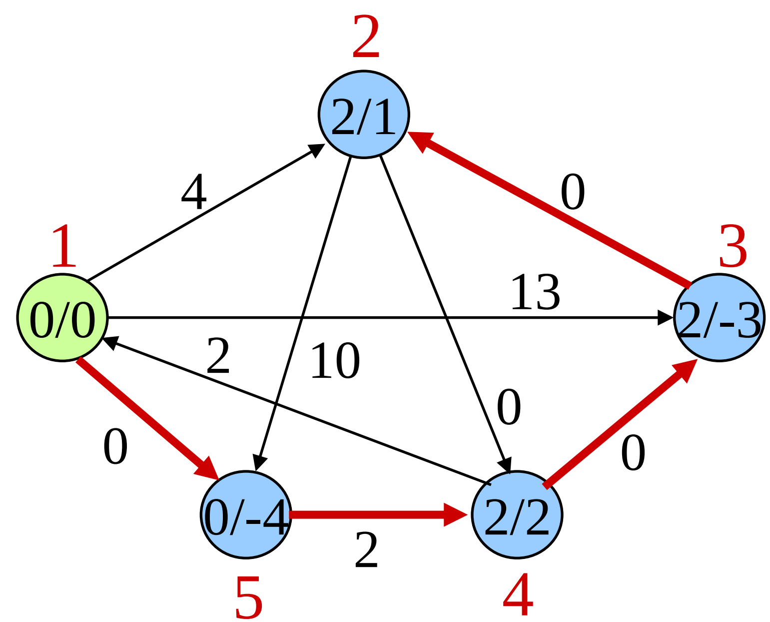
# Example

For each vertex, $\hat{\delta}/\delta$

$$d_{uv} := \hat{\delta}(u, v) + h(v) - h(u)$$



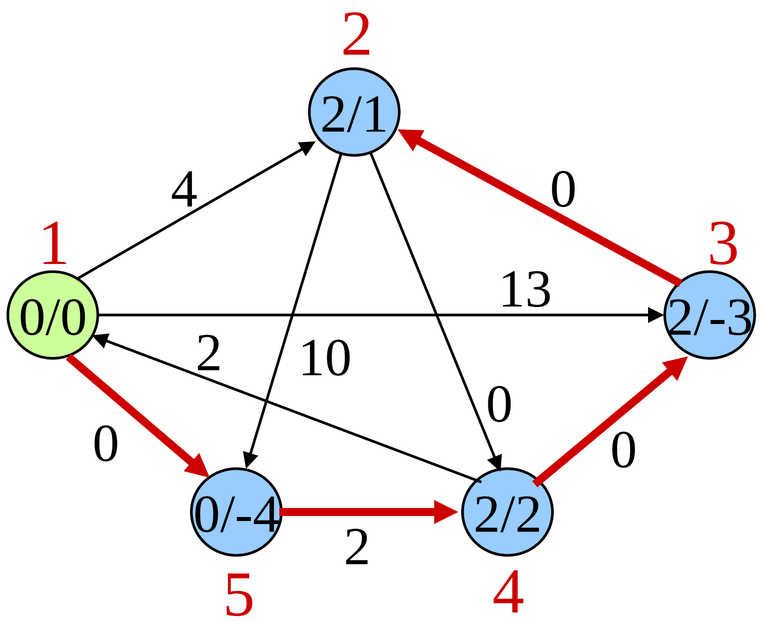

G'



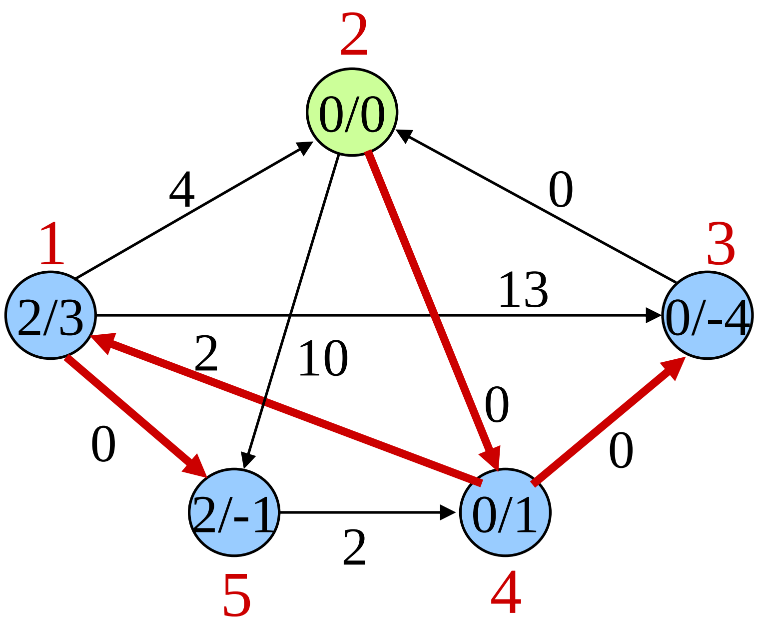

G

# Run Dijkstra

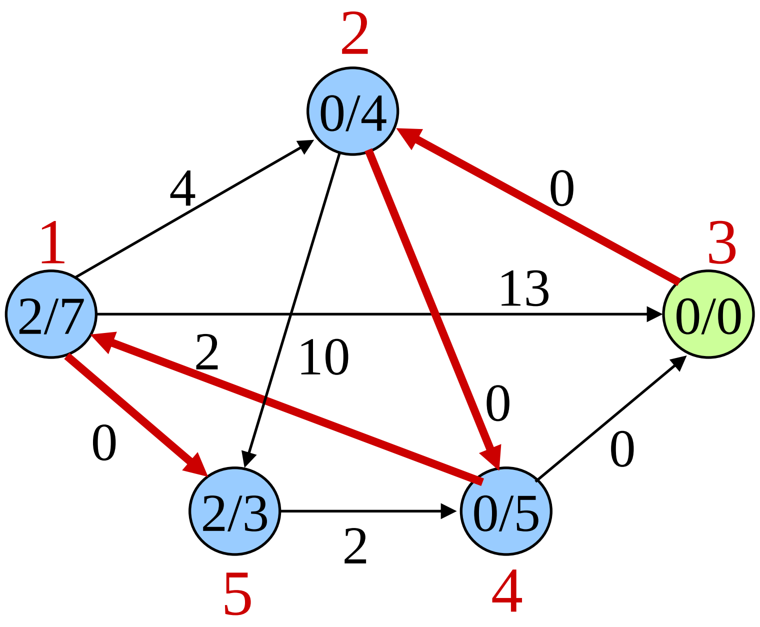
```
Dijkstra(G)
   for each v ∈ V
      d[v] = ∞;
   d[s] = 0; S = ∅; Q = V;
   while (Q ≠ ∅)
      u = ExtractMin(Q);
      S = S U {u};
      for each v ∈ u->Adj[]
         if (d[v] > d[u]+w(u,v))
            d[v] = d[u]+w(u,v);
```



# Example

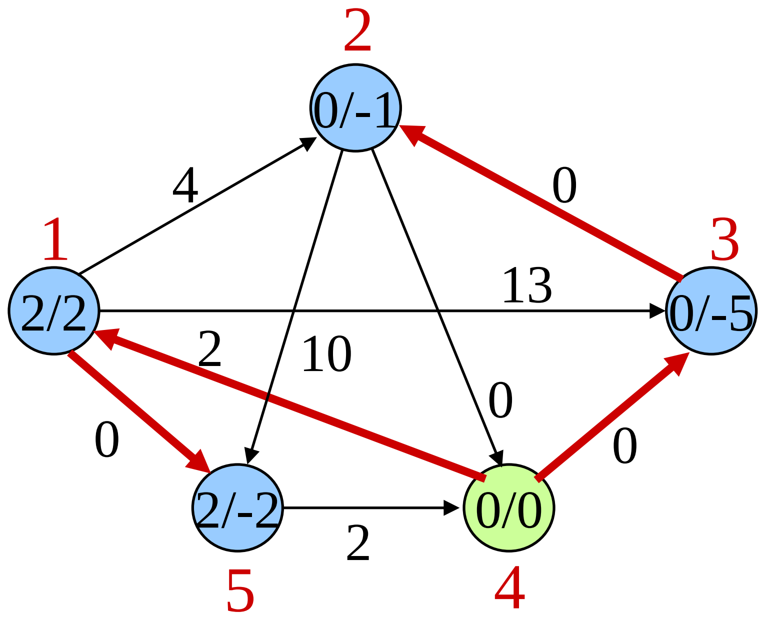


2
0/0
4
0
1
3
2/3
13
0/-4
2
10
0
0
0
2/-1
0/1
2
5
4

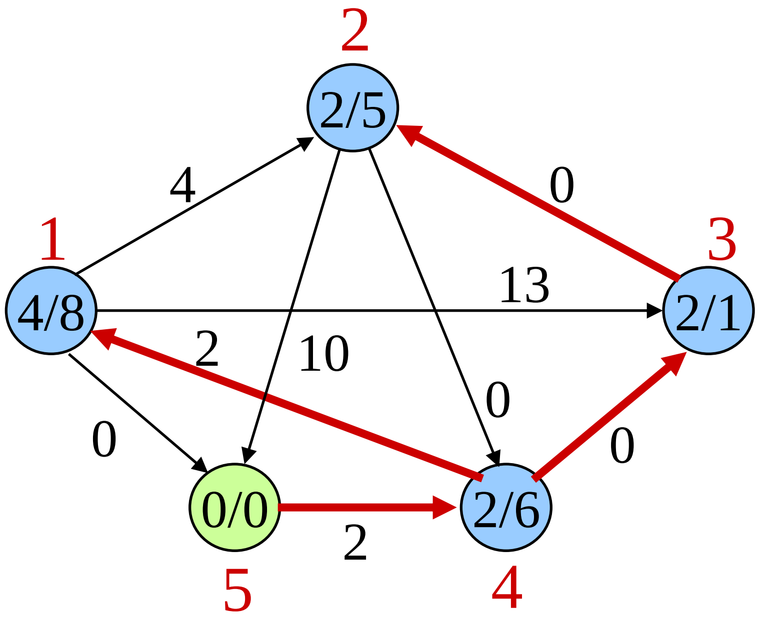



2
0/4
4
0
1
3
2/7
13
0/0
2
10
0
0
0
2/3
0/5
2
5
4

# Example



2
0/-1
4
0
1
3
2/2
13
0/-5
2
10
0
0
0
2/-2
0/0
2
5
4

2
2/5
4
0
1
3
4/8
13
2/1
2
10
0
0
0
0/0
2/6
2
5
4

# Summary

- Dynamic-programming algorithm
  - $O(V^4)$
- Connection to matrix-multiplication
  - Improved version (repeated squaring) : $O(V^3 \log V)$
  - Floyd-Warshall: $O(V^3)$ and very simple to implement;
- Johnson's algorithm: $O(V^2 \lg V + VE)$
  - Runs Bellman Ford (detects negative cycles)
  - Reweighting: modify graph to make all edge-weights non-negative
  - run Dijkstra's algorithm $|V|$ times