All Pairs Shortest Paths

Algorithms we have learned

Graph type	Algorithm	Running Time		
		Binary heap	Linear array	Fibo. heap
Unweighted graph	BFS	O(V+E)		O(V)
Non-negative edge weight graph	Dijsktra	O(E+VlogV)		O(VlogV)
General graph	Bellman-Ford	O(VE)		
DAG	Bellman-Ford	O(V+E)		O(V)

All Pairs Shortest Paths (APSP)

- given : directed graph $G = (V, E)$,
  weight function $w : \mathrm{E} \rightarrow \mathrm{R}$, $|\mathrm{V}| = n$

- goal : create an $n \times n$ matrix $D = (d_{ij})$ of shortest path distances
  i.e., $d_{ij} = \delta (v_i , v_j)$

- trivial solution : run a shortest path algorithm (SSP) algorithm n times, one for each vertex as the source.

All-pair shortest paths

- How about using the previous algorithms to solve all-pair shortest path problem ?
  - Unweighted graph: run BFS |V| times → O(VE)
  - Non-negative graph: run Dijkstra |V| times → $O(VE+V^2 \lg V)$
  - General case: run Bellman-Ford |V| times → $O(V^2E)$

- When handling general cases, the time complexity is at most $O(V^4)$……….
- We can do better!

All Pairs Shortest Paths (APSP)

- all edge weights are nonnegative : use Dijkstra's algorithm

  PQ: priority queue implementation method

  - PQ = linear array : $O(V^3 + VE) = O(V^3)$
  - PQ = binary heap : $O(V^2\log V + EV\log V) = O(V^3\log V)$ for dense graphs
    - **better only for sparse graphs**
  - PQ = fibonacci heap : $O(V^2\log V + EV) = O(V^3)$ for dense graphs
    - **better only for sparse graphs**

- negative edge weights : use Bellman-Ford algorithm

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

Adjacency Matrix Representation of Graphs

- ▶ $n \times n$ matrix $W = (w_{ij})$ of edge weights :

$$w_{ij} = \begin{cases} w(v_i, v_j) & \text{if } (v_i, v_j) \in E \\ \infty & \text{if } (v_i, v_j) \notin E \end{cases}$$

- ▶ assume $w_{ii} = 0$ for all $v_i \in V$, because
  - no negative weight cycle

    $\Rightarrow$ shortest path to itself has no edge,

    i.e., $\delta(v_i, v_i) = 0$

Dynamic Programming

(1) Characterize the structure of an optimal solution.

(2) Recursively define the value of an optimal solution.

(3) Compute the value of an optimal solution in a bottom-up manner.

(4) Construct an optimal solution from information constructed in (3).

Shortest Paths and Matrix Multiplication

Assumption : negative edge weights may be present, but no negative weight cycles.

(1) Structure of a Shortest Path :

- Consider a shortest path p_{ij}^{m} from v_i to v_j such that $|p_{ij}^{m}| \le m$ (最多通過m個邊)
  - ▶ i.e., path p_{ij}^{m} has at most m edges.
- no negative-weight cycle $\Rightarrow$ all shortest paths are simple
  $\Rightarrow$ m is finite $\Rightarrow m \le n - 1$
- $i = j \Rightarrow |p_{ii}| = 0$ & $\omega(p_{ii}) = 0$
- $i \ne j \Rightarrow$ decompose path p_{ij}^{m} into p_{ik}^{m-1} & $v_k \rightarrow v_j$, where $|p_{ik}^{m-1}| \le m - 1$
  - ▶ p_{ik}^{m-1} should be a shortest path from v_i to v_k by optimal substructure property. 最佳化原理：所有最短路徑的子路徑均為最短路徑
  - ▶ Therefore, $\delta(v_i, v_j) = \delta(v_i, v_k) + w_{kj}$

Shortest Paths and Matrix Multiplication

(2) A Recursive Solution to All Pairs Shortest Paths Problem :

- d_{ij}^{m} = minimum weight of any path from v_i to v_j that contains at most "m" edges.

- $m = 0$: There exist a shortest path from v_i to v_j with no edges $\leftrightarrow$ $i = j$.

  ▶ $$d_{ij}^{0} = \begin{cases} 0 & \text{if } i = j \\ \infty & \text{if } i \neq j \end{cases}$$

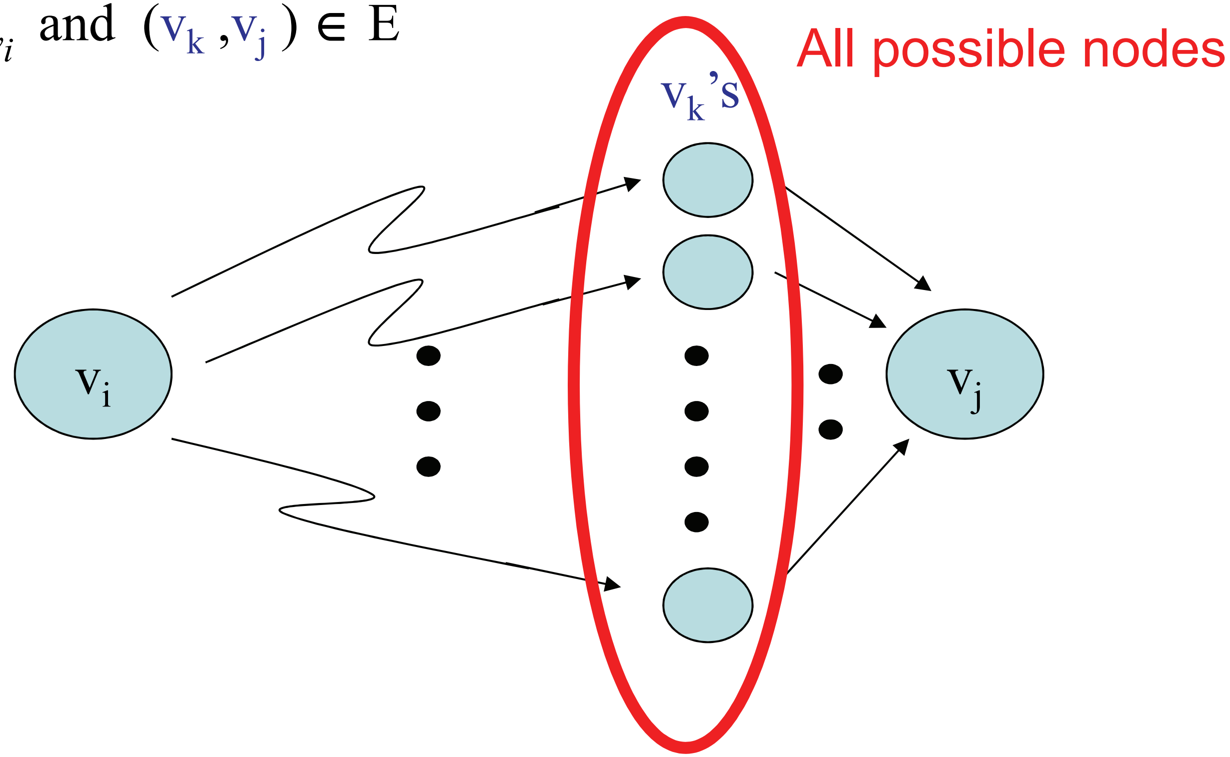
- $m \geq 1$: $d_{ij}^{m} = \min \{d_{ij}^{m-1}, \min_{1 \leq k \leq n \wedge k \neq j} \{d_{ik}^{m-1} + w_{kj}\}\}$

  $= \min_{1 \leq k \leq n} \{d_{ik}^{m-1} + w_{kj}\}$ for all $v_k \in V$,

  since $w_{jj} = 0$ for all $v_j \in V$.

Shortest Paths and Matrix Multiplication

- to consider all possible shortest paths with $\leq m$ edges from v_i to v_j
  - ▶ consider shortest path with $\leq m - 1$ edges, from v_i to v_k, where $v_k \in R_{v_i}$ and $(v_k, v_j) \in E$



- note : $\delta(v_i, v_j) = d_{ij}^{n-1} = d_{ij}^{n} = d_{ij}^{n+1}$, since $m \leq n - 1 = |V| - 1$

Shortest Paths and Matrix Multiplication

(3) Computing the shortest-path weights bottom-up :

- given $W = D^1$, compute a series of matrices $D^2, D^3, \ldots, D^{n-1}$, where $D^m = (d_{ij}^{m})$ for $m = 1, 2, \ldots, n-1$
  ► final matrix D^{n-1} contains actual shortest path weights, i.e., $d_{ij}^{n-1} = \delta(v_i, v_j)$

- SLOW-APSP(W)

```
SLOW-APSP( W )
    D^1 ← W
    for m ← 2 to n-1 do
        D^m ← EXTEND( D^(m-1) , W )
    return D^(n-1)
```

Shortest Paths and Matrix Multiplication

EXTEND **(D , W)**

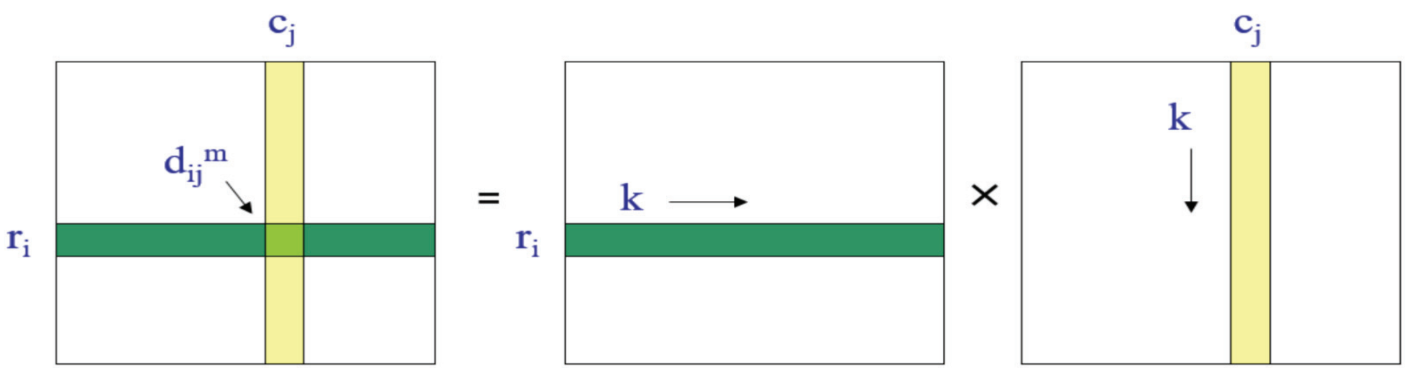
▶ $D = (d_{ij})$ is an $n \times n$ matrix

```
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 + ω_kj}
return D
```

MATRIX-MULT **(A , B)**

▶ $C = (c_{ij})$ is an $n \times n$ result matrix

```
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 x b_kj
return C
```



Shortest Paths and Matrix Multiplication

- **relation to matrix multiplication** $C = A \times B$: $\mathbf{c}_{ij} = \sum_{1 \le k \le n} \mathbf{a}_{ik} \times \mathbf{b}_{kj}$,
  - ▶ $D^{m-1} \leftrightarrow A$ & $\mathbf{W} \leftrightarrow B$ & $D^{m} \leftrightarrow C$
    "min" ↔ "+" & "+" ↔ "×" & "∞" ↔ "0"

  $\boxed{d_{ij} \leftarrow \mathbf{min}\{d_{ij}, d_{ik} + \omega_{kj}\}}$ ⟺ $\boxed{c_{ij} \leftarrow c_{ij} + a_{ik} \times b_{kj}}$

- Thus, we compute the sequence of matrix products

  $D^1 = D^0 \times W = W$; note D^0 = identity matrix, i.e., $d_{ij}^{\,0} = \begin{cases} 0 & \text{if } i = j \\ \infty & \text{if } i \neq j \end{cases}$

  $D^2 = D^1 \times W = W^2$

  $D^3 = D^2 \times W = W^3$

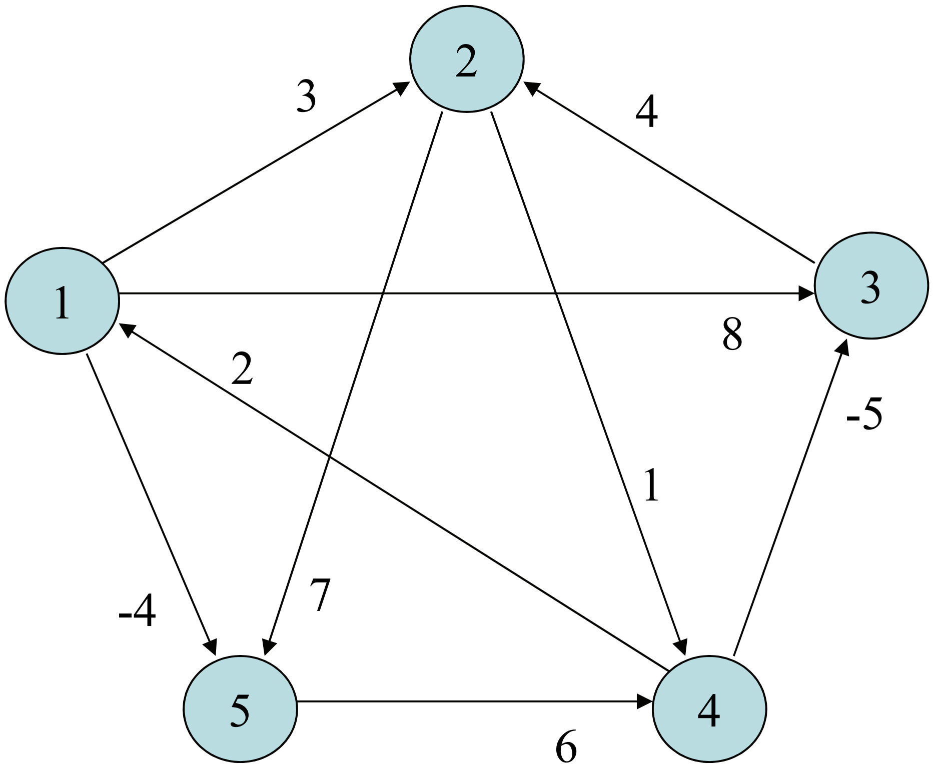
  $\vdots$

  $D^{n-1} = D^{n-2} \times W = W^{n-1}$

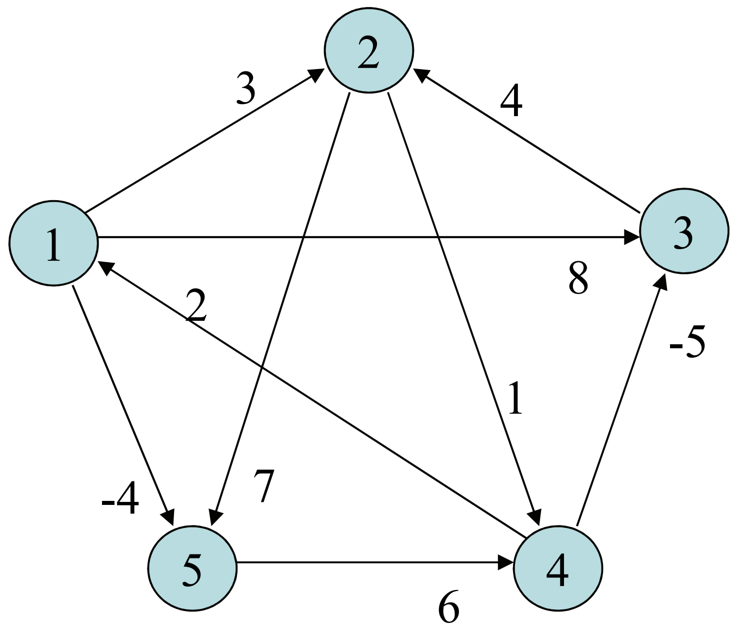
- running time : $\Theta(n^4) = \Theta(V^4)$
  - ▶ each matrix product : $\Theta(n^3)$
  - ▶ number of matrix products : n-1

Shortest Paths and Matrix Multiplication

- Example



Shortest Paths and Matrix Multiplication



	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	∞	-5	0	∞
5	∞	∞	∞	6	0

$D^1 = D^0 W$

Shortest Paths and Matrix Multiplication

	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	∞	-5	0	∞
5	∞	∞	∞	6	0

	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	∞	-5	0	∞
5	∞	∞	∞	6	0

$$\boxed{d_{ij} \leftarrow \mathbf{min}\{d_{ij}, d_{ik} + \omega_{kj}\}}$$

	1	2	3	4	5
1	0	3	8	2	-4
2	3	0	-4	1	7
3	∞	4	0	5	11
4	2	-1	-5	0	-2
5	8	∞	1	6	0

$D^2 = D^1 W$

Shortest Paths and Matrix Multiplication

	1	2	3	4	5
1	0	3	8	2	-4
2	3	0	-4	1	7
3	∞	4	0	5	11
4	2	-1	-5	0	-2
5	8	∞	1	6	0

	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	∞	-5	0	∞
5	∞	∞	∞	6	0

	1	2	3	4	5
1	0	3	-3	2	-4
2	3	0	-4	1	-1
3	7	4	0	5	11
4	2	-1	-5	0	-2
5	8	5	1	6	0

$$\boxed{d_{ij} \leftarrow \mathbf{min}\{d_{ij}, d_{ik} + \omega_{kj}\}}$$

$$D^2W = D^3$$

Shortest Paths and Matrix Multiplication

	1	2	3	4	5
1	0	3	-3	2	-4
2	3	0	-4	1	-1
3	7	4	0	5	11
4	2	-1	-5	0	-2
5	8	5	1	6	0

	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	∞	-5	0	∞
5	∞	∞	∞	6	0

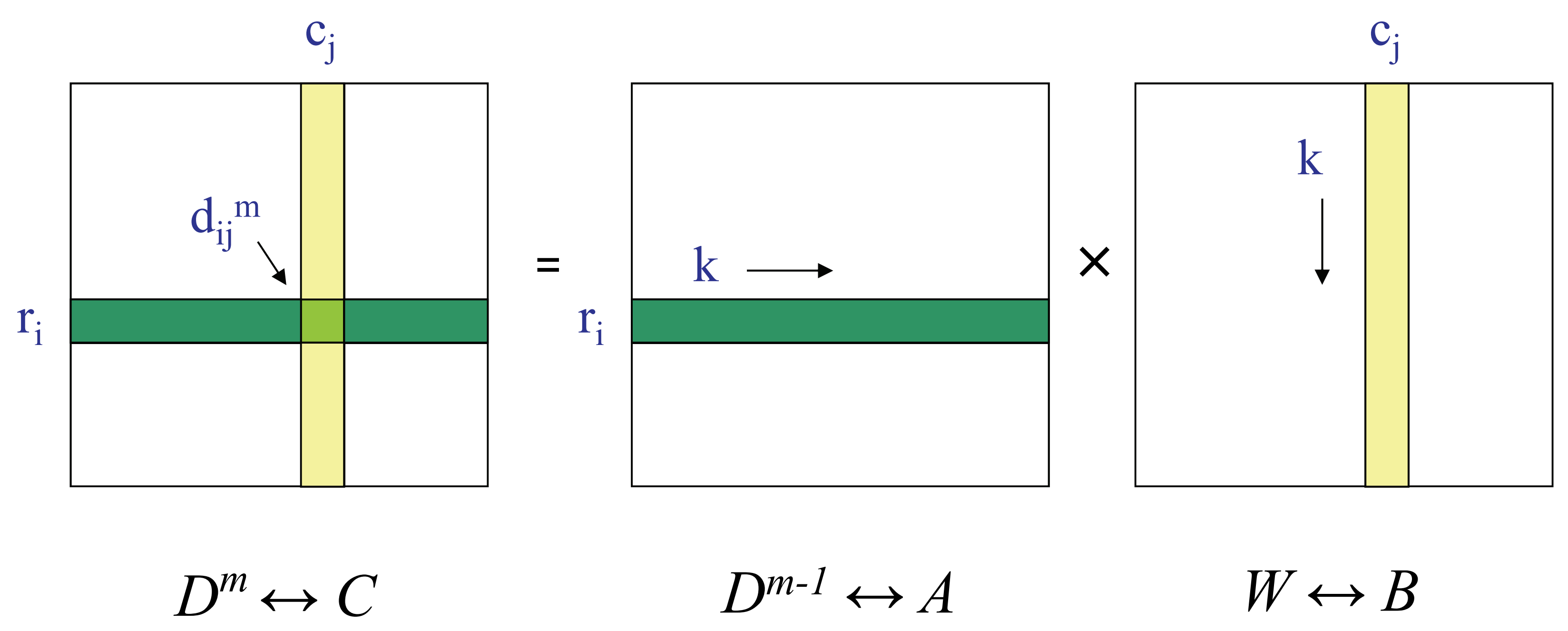
	1	2	3	4	5
1	0	1	-3	2	-4
2	3	0	-4	1	-1
3	7	4	0	5	3
4	2	-1	-5	0	-2
5	8	5	1	6	0

$$\boxed{d_{ij} \leftarrow \mathbf{min}\{d_{ij}, d_{ik} + \omega_{kj}\}}$$

$$D^3 W = D^4$$

SSSP and Matrix-Vector Multiplication

- relation of APSP to one step of matrix multiplication



SSSP and Matrix-Vector Multiplication

- $d_{ij}^{\,n-1}$ at row r_i and column c_j of product matrix $= \delta(v_i{=}s, v_j)$ for $j = 1, 2, 3, \ldots, n$
- row r_i of the product matrix = solution to single-source shortest path problem for $s = v_i$.

► r_i of C = matrix B multiplied by r_i of A

$$\Rightarrow D_i^{\,m} = D_i^{\,m-1} \times W$$

SSP and Matrix-Vector Multiplication

- let $D_i^0 = d^0$, where $d_j^0 = \begin{cases} 0 & \text{if } i = j \\ \infty & \text{otherwise} \end{cases}$
- we compute a sequence of n-1 "matrix-vector" products

$$d_i^1 = d_i^0 \times W$$
$$d_i^2 = d_i^1 \times W$$
$$d_i^3 = d_i^2 \times W$$
$$\vdots$$
$$d_i^{n-1} = d_i^{n-2} \times W$$

SSP and Matrix-Vector Multiplication

- this sequence of matrix-vector products
  - ► same as Bellman-Ford algorithm.
  - ► vector $d_i^m \Rightarrow$ d values of Bellman-Ford algorithm after m-th relaxation pass.
  - ► $d_i^m \leftarrow d_i^{m-1} \times W$

    $\Rightarrow$ *m-th* relaxation pass over all edges.

SSP and Matrix-Vector Multiplication

BELLMAN-FORD (G , v_i)

▶ perform RELAX (u , v) for

▶ every edge (u , v) ∈ E

for $j \leftarrow 1$ to n do

  for $k \leftarrow 1$ to n do

    RELAX (v_k , v_j)

RELAX (u , v)

$d_v = \min \{ d_v , d_u + \omega_{uv} \}$

EXTEND (d_i , W)

▶ d_i is an n-vector

for $j \leftarrow 1$ to n do

  $d_j \leftarrow \infty$

  for $k \leftarrow 1$ to n do

    $d_j \leftarrow \min \{ d_j , d_k + \omega_{kj} \}$

Improving Running Time Through Repeated Squaring

- idea : goal is not to compute all D^m matrices
  - ▶ we are interested only in matrix D^{n-1}
- recall : no negative-weight cycles $\Rightarrow D^m = D^{n-1}$ for all $m \geq n-1$
- we can compute D^{n-1} with only $\lceil \lg(n-1) \rceil$ matrix products as

$$
\begin{aligned}
D^1 &= W \\
D^2 &= W^2 = W \times W \\
D^4 &= W^4 = W^2 \times W^2 \\
D^8 &= W^8 = W^4 \times W^4 \\
&\vdots
\end{aligned}
$$

$$D^{2^{\lceil \lg(n-1) \rceil}} = W^{2^{\lceil \lg(n-1) \rceil}} = W^{2^{\lceil \lg(n-1) \rceil - 1}} \times W^{2^{\lceil \lg(n-1) \rceil - 1}}$$

- This technique is called repeated squaring.

Improving Running Time Through Repeated Squaring

- FASTER-APSP (W)

  $D^1 \leftarrow W$

  $m \leftarrow 1$

  while $m < n\text{-}1$ do

      $D^{2m} \leftarrow$ EXTEND (D^m , D^m)

      $m \leftarrow 2m$

  return D^m

- final iteration computes D^{2m} for some $n\text{-}1 \leq 2m \leq 2n\text{-}2 \Rightarrow D^{2m} = D^{n-1}$

- running time : $\Theta(n^3 \lg n) = \Theta(V^3 \lg V)$

  - each matrix product : $\Theta(n^3)$
  - # of matrix products : $\lceil \lg(n\text{-}1) \rceil$
  - simple code, no complex data structures, small hidden constants in Θ-notation.

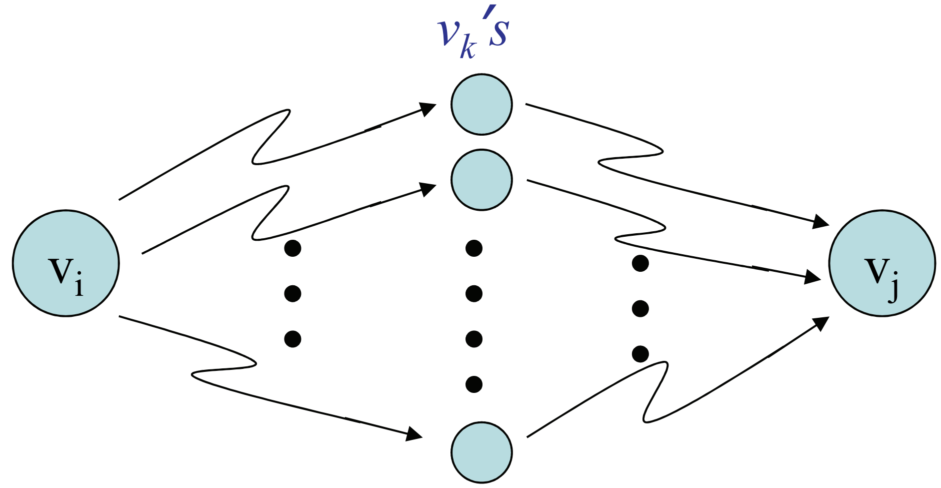
Idea Behind Repeated Squaring

- decompose p_{ij}^{2m} as p_{ik}^{m} & p_{kj}^{m}, where

  p_{ij}^{2m} : $v_i \rightsquigarrow v_j$

  p_{ik}^{m} : $v_i \rightsquigarrow v_k$

  p_{kj}^{m} : $v_k \rightsquigarrow v_j$



Floyd-Warshall Algorithm

- assumption : negative-weight edges, but no negative-weight cycles

  (1) The Structure of a Shortest Path :

- Definition : intermediate vertex of a path $p = < v_1 , v_2 , v_3 , \ldots , v_k >$

  ▶ any vertex of p other than v_1 or v_k .

- p_{ij}^{m} : a shortest path from v_i to v_j with all intermediate vertices from $V_m = \{ v_1 , v_2 , \ldots , v_m \}$

- rcursive relationship between p_{ij}^{m} and p_{ij}^{m-1}

  ▶ depends on whether v_m is an intermediate vertex of p_{ij}^{m}

  - case 1: v_m is not an intermediate vertex of p_{ij}^{m}

    $\Rightarrow$ all intermediate vertices of p_{ij}^{m} are in V_{m-1}

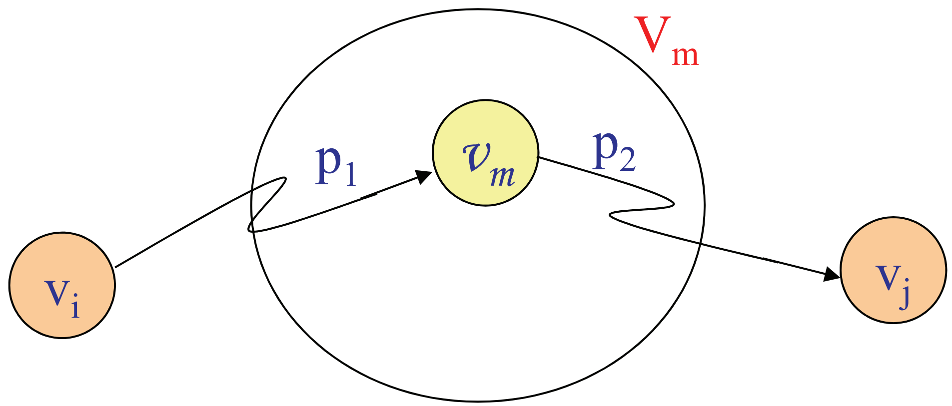
    $\Rightarrow p_{ij}^{m} = p_{ij}^{m-1}$

Floyd-Warshall Algorithm

- case 2 : v_m is an intermediate vertex of p_{ij}^{m}
  - decompose path as $v_i \leadsto v_m \leadsto v_j$

    $\Rightarrow p_1 : v_i \leadsto v_m$ & $p_2 : v_m \leadsto v_j$
  - by opt. structure property both p_1 & p_2 are shortest paths.
  - v_m is not an intermediate vertex of p_1 & p_2

    $\Rightarrow p_1 = p_{im}^{m-1}$ & $p_2 = p_{mj}^{m-1}$



Floyd-Warshall Algorithm

(2) A Recursive Solution to APSP Problem :

- $d_{ij}^{m} = \omega(p_{ij})$: weight of a shortest path from v_i to v_j with all intermediate vertices from

  $$V_m = \{ v_1 , v_2 , \dots , v_m \}.$$

- note : $d_{ij}^{n} = \delta(v_i , v_j)$ since $V_n = V$
  - ► i.e., all vertices are considered for being intermediate vertices of p_{ij}^{n} .

Floyd-Warshall Algorithm

- compute d_{ij}^{m} in terms of d_{ij}^{k} with smaller $k < m$
- $m = 0$: V_0 = empty set

  $\Rightarrow$ path from v_i to v_j with no intermediate vertex.

  i.e., v_i to v_j paths with at most one edge

  $\Rightarrow d_{ij}^{0} = \omega_{ij}$
- $m \geq 1$: $d_{ij}^{m} = \min \{d_{ij}^{m-1}, d_{im}^{m-1} + d_{mj}^{m-1}\}$

Floyd-Warshall Algorithm

(3) Computing Shortest Path Weights Bottom Up :

```
FLOYD-WARSHALL( W )
    ► D^0, D^1, ... , D^n are n x n  matrices
    for m ← 1 to n do
        for i ← 1 to n do
            for j ← 1 to n do
                d_ij^m ← min {d_ij^(m-1) , d_im^(m-1) + d_mj^(m-1) }
    return D^n
```

$$d_{ij}^{m} \leftarrow \min \{ d_{ij}^{m-1} , d_{im}^{m-1} + d_{mj}^{m-1} \}$$

Floyd-Warshall Algorithm

FLOYD-WARSHALL (W)

▶ D is an n x n matrix

D ← W

for m ← 1 to n do

  for i ← 1 to n do

    for j ← 1 to n do

      if $d_{ij} > d_{im} + d_{mj}$ then

        $d_{ij} \leftarrow d_{im} + d_{mj}$

        $\Pi_{ij} \leftarrow \Pi_{kj}$

return D

Floyd-Warshall Algorithm

- maintaining *n* D matrices can be avoided by dropping all superscripts.
  - *m-th* iteration of outermost for-loop

    begins with $D = D^{m-1}$

    ends with $D = D^{m}$
  - computation of d_{ij}^{m} depends on d_{im}^{m-1} and d_{mj}^{m-1} .

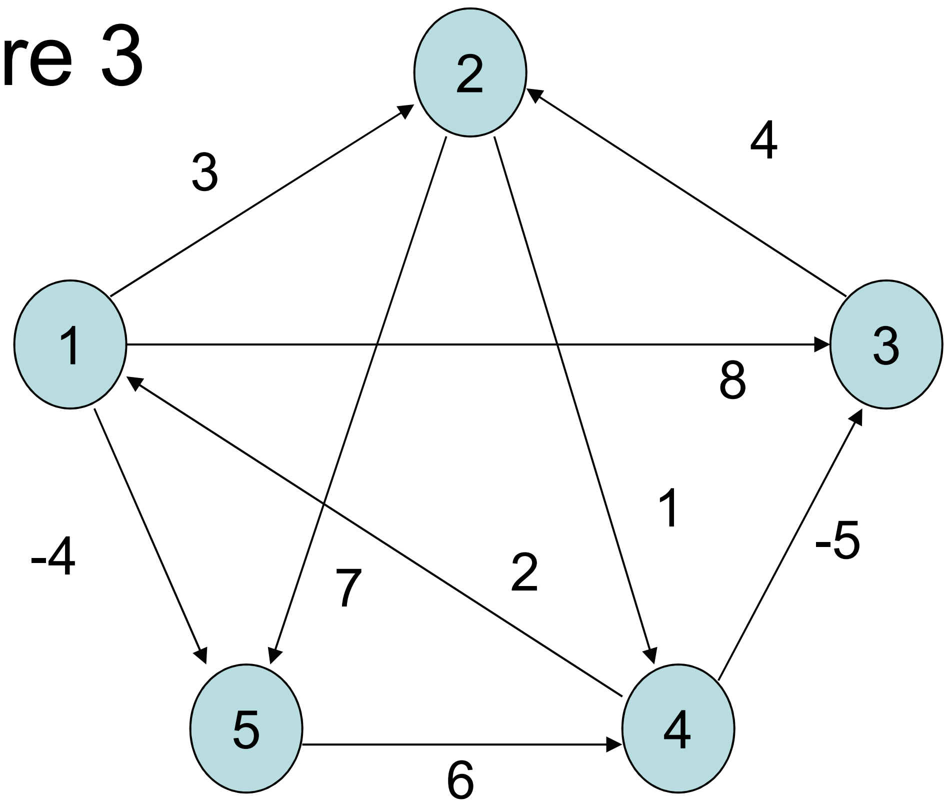
    no problem if d_{im} & d_{mj} are already updated to d_{im}^{m} & d_{mj}^{m}
    since $d_{im}^{m} = d_{im}^{m-1}$ & $d_{mj}^{m} = d_{mj}^{m-1}$.

- running time : $\Theta(n^3) = \Theta(V^3)$

  simple code, no complex data structures, small hidden constants

Example:

- Figure 3



Path Reconstruction

- Before you run Floyd's, you initialize your distance matrix $\boldsymbol{D}$ and path matrix $\boldsymbol{\Pi}$ to indicate the use of no immediate vertices.
  - (Thus, you are only allowed to traverse direct paths between vertices.)
- Then, at each step of Floyd's, you essentially find out whether or not using vertex k will *improve* an estimate between the distances between vertex i and vertex j.

Path Reconstruction

- If it ***does improve*** the estimate here's what you need to record:
  1) record the new shortest path weight between i and j
     - **We don't need to change our path and we do not update the path matrix**
  2) record the fact that the shortest path between i and j goes through k
     - **We want to store the last vertex from the shortest path from vertex k to vertex j. *This will NOT necessarily be k, but rather, it will be path[k][j].***

This gives us the following update to our algorithm:

```
if (D[i][k]+D[k][j] < D[i][j]) { // Update is necessary to use k as intermediate vertex
    D[i][j] = D[i][k]+D[k][j];
    Π[i][j] = Π[k][j];
}
```

Path Reconstruction

- Now, the once this path matrix is computed, we have all the information necessary to reconstruct the path.
  - Consider the following path matrix (indexed from 1 to 5 instead of 0 to 4):

NIL	**3**	**4**	**5**	**1**
4	**NIL**	**4**	**2**	**1**
4	**3**	**NIL**	**2**	**1**
4	**3**	**4**	**NIL**	**1**
4	**3**	**4**	**5**	**NIL**

- Reconstruct the path from vertex1 to vertex 2:
  - First look at path[1][2] = 3. This signifies that on the path from 1 to 2, 3 is the last vertex visited before 2.
    - Thus, the path is now, 1…3->2.
  - Now, look at path[1][3], this stores a 4. Thus, we find the last vertex visited on the path from 1 to 3 is 4.
  - So, our path now looks like 1…4->3->2. So, we must now look at path[1][4]. This stores a 5,
  - thus, we know our path is 1…5->4->3->2. When we finally look at path[1][5], we find 1,
  - which means our path really is 1->5->4->3->2.

$$D(0)=\begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \prod(0)=\begin{pmatrix} NIL & 1 & 1 & NIL & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & NIL & NIL \\ 4 & NIL & 4 & NIL & NIL \\ NIL & NIL & NIL & 5 & NIL \end{pmatrix}$$

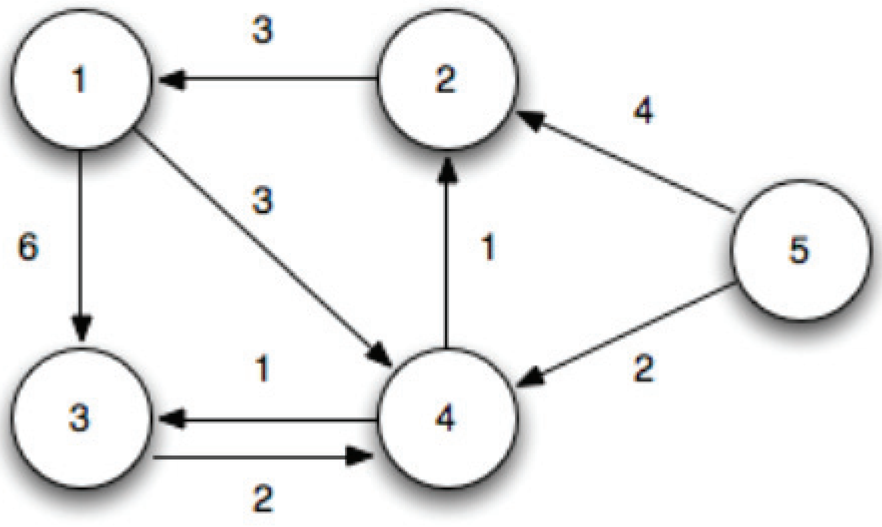
$$D(1)=\begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \prod(1)=\begin{pmatrix} NIL & 1 & 1 & NIL & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & NIL & NIL \\ 4 & 1 & 4 & NIL & 1 \\ NIL & NIL & NIL & 5 & NIL \end{pmatrix}$$

$$D(2)=\begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \prod(2)=\begin{pmatrix} NIL & 1 & 1 & 2 & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & 2 & 2 \\ 4 & 1 & 4 & NIL & 1 \\ NIL & NIL & NIL & 5 & NIL \end{pmatrix}$$

$$D(3)=\begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \prod(3)=\begin{pmatrix} NIL & 1 & 1 & 2 & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & 2 & 2 \\ 4 & 3 & 4 & NIL & 1 \\ NIL & NIL & NIL & 5 & NIL \end{pmatrix}$$

$$D^{(4)}=\begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad \Pi^{(4)}=\begin{pmatrix} NIL & 1 & 4 & 2 & 1 \\ 4 & NIL & 4 & 2 & 1 \\ 4 & 3 & NIL & 2 & 1 \\ 4 & 3 & 4 & NIL & 1 \\ 4 & 3 & 4 & 5 & NIL \end{pmatrix}$$

$$D^{(5)}=\begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad \Pi^{(5)}=\begin{pmatrix} NIL & 3 & 4 & 5 & 1 \\ 4 & NIL & 4 & 2 & 1 \\ 4 & 3 & NIL & 2 & 1 \\ 4 & 3 & 4 & NIL & 1 \\ 4 & 3 & 4 & 5 & NIL \end{pmatrix}$$



Initialization: $(k = 0)$

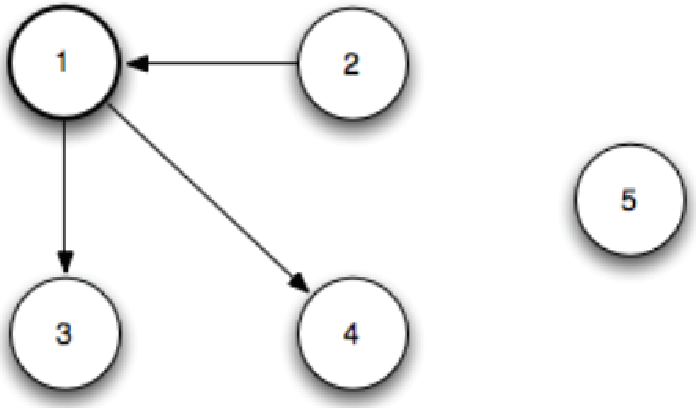
D

	1	2	3	4	5
1	0	∞	6	3	∞
2	3	0	∞	∞	∞
3	∞	∞	0	2	∞
4	∞	1	1	0	∞
5	∞	4	∞	2	0

Π

	1	2	3	4	5
1	/	/	1	1	/
2	2	/	/	/	/
3	/	/	/	3	/
4	/	4	4	/	/
5	/	5	/	5	/

Iteration 1: ($k = 1$) Shorter paths from $2 \rightsquigarrow 3$ and $2 \rightsquigarrow 4$ are found through vertex 1



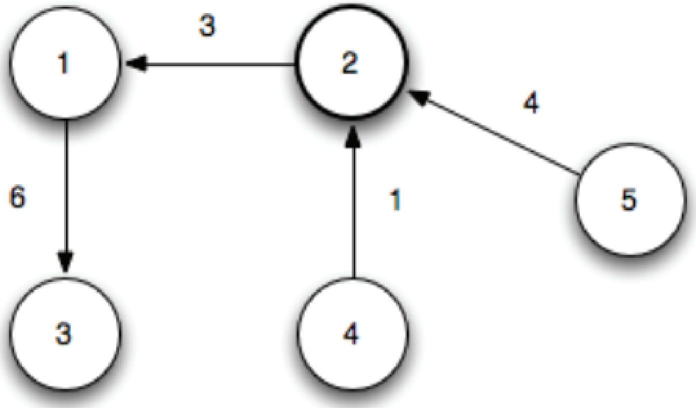
D

	1	2	3	4	5
1	0	∞	6	3	∞
2	3	0	**9**	**6**	∞
3	∞	∞	0	2	∞
4	∞	1	1	0	∞
5	∞	4	∞	2	0

Π

	1	2	3	4	5
1	/	/	1	1	/
2	2	/	**1**	**1**	/
3	/	/	/	3	/
4	/	4	4	/	/
5	/	5	/	5	/

Iteration 2: ($k = 2$) Shorter paths from $4 \rightsquigarrow 1$, $5 \rightsquigarrow 1$, and $5 \rightsquigarrow 3$ are found through vertex 2



D

	1	2	3	4	5
1	0	∞	6	3	∞
2	3	0	9	6	∞
3	∞	∞	0	2	∞
4	**4**	1	1	0	∞
5	**7**	4	**13**	2	0

Π

	1	2	3	4	5
1	/	/	1	1	/
2	2	/	1	1	/
3	/	/	/	3	/
4	**2**	4	4	/	/
5	**2**	5	**2**	5	/

Iteration 3: ($k = 3$) No shorter paths are found through vertex 3

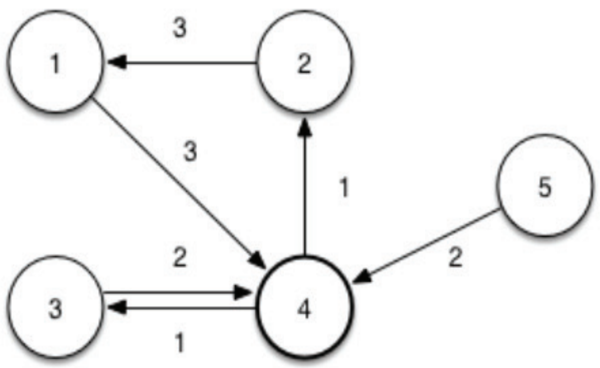
D

	1	2	3	4	5
1	0	∞	6	3	∞
2	3	0	9	6	∞
3	∞	∞	0	2	∞
4	4	1	1	0	∞
5	7	4	13	2	0

Π

	1	2	3	4	5
1	/	/	1	1	/
2	2	/	1	1	/
3	/	/	/	3	/
4	2	4	4	/	/
5	2	5	2	5	/

Iteration 4: ($k = 4$) Shorter paths from 1 ⤳ 2, 1 ⤳ 3, 2 ⤳ 3, 3 ⤳ 1, 3 ⤳ 2, 5 ⤳ 1, 5 ⤳ 2, 5 ⤳ 3, and 5 ⤳ 4 are found through vertex 4



D

	1	2	3	4	5
1	0	**4**	**4**	3	∞
2	3	0	**7**	6	∞
3	**6**	**3**	0	2	∞
4	4	1	1	0	∞
5	**6**	**3**	**3**	2	0

Π

	1	2	3	4	5
1	/	**4**	**4**	1	/
2	2	/	**4**	1	/
3	**2**	**4**	/	3	/
4	2	4	4	/	/
5	2	**4**	**4**	5	/

Iteration 5: ($k = 5$) No shorter paths are found through vertex 5

D

	1	2	3	4	5
1	0	4	4	3	∞
2	3	0	7	6	∞
3	6	3	0	2	∞
4	4	1	1	0	∞
5	6	3	3	2	0

Π

	1	2	3	4	5
1	/	4	4	1	/
2	2	/	4	1	/
3	2	4	/	3	/
4	2	4	4	/	/
5	2	4	4	5	/

The final shortest paths for all pairs is given by

D

	1	2	3	4	5
1	0	4	4	3	∞
2	3	0	7	6	∞
3	6	3	0	2	∞
4	4	1	1	0	∞
5	6	3	3	2	0

Π

	1	2	3	4	5
1	/	4	4	1	/
2	2	/	4	1	/
3	2	4	/	3	/
4	2	4	4	/	/
5	2	4	4	5	/

- ***What is Binary Relation?***

A binary relation R from the set S to the set T is a subset of $S \times T$, $R \subseteq S \times T$. If $S = T$, we say that the relation is a binary relation on S.

- ***Properties of Binary Relation***

Let R be a binary relation on S. Then R is

Reflexive: iff $(\forall x), \ (x \in S \rightarrow xRx)$

Symmetric: iff $(\forall x)(\forall y), \ (x \in S \wedge y \in S \wedge xRy \rightarrow yRx)$

Anti-symmetric: iff $(\forall x)(\forall y), (x \in S \wedge y \in S \wedge xRy \wedge yRx$
$\rightarrow x = y)$

Transitive: iff $(\forall x)(\forall y)(\forall z), (x \in S \wedge y \in S \wedge z \in S$
$\wedge xRy \wedge yRz \rightarrow xRz)$

Some binary relations don't have these properties.

• *Closures of Binary Relation*

A binary relation R on a set *S* may not have a particular property such as reflexivity, symmetry, or transitivity. However, it may be possible to extend the relation so that it does have the property.

Extending R means finding a larger subset of $S \times S$ that contains R and which has the desired property. The closure of a relation on *S* with respect to a property is the smallest such extension that has the desired property.

Commonly used Closures:

transitive closure

reflexive closure

symmetric closure

- ***Transitive Closure of Binary Relation***

A relation R^t is the transitive closure of a binary relation R if and only if:

(1) R^t is transitive,
(2) $R \subseteq R^t$, and
(3) for any relation S, if $R \subseteq S$ and S is transitive, then $R^t \subseteq S$, that is, R^t is the smallest relation that satisfies (1) and (2).

- ***How to find Transitive Closure?***

We need to add the minimum number of tuples to R, giving us R^t, such that if (a,b) is in R^t and (b,c) is in R^t, then (a,c) is in R^t.

$$R^t = R \cup \Delta$$

$$(a,b) \in R^t \wedge (b,c) \in R^t \rightarrow (a,c) \in R^t$$

Example of Transitive Closure:

Let $S = \{1, 2, 3\}$.

$R = \{(1,1), (1,2), (1,3), (2,3), (3,1)\}$.

$(2,3) \in R \wedge (3,1) \in R \rightarrow (2,1) \in R^t$

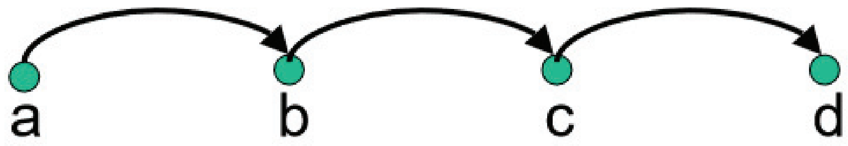
$(3,1) \in R \wedge (1,2) \in R \rightarrow (3,2) \in R^t$

$(3,1) \in R \wedge (1,3) \in R \rightarrow (3,3) \in R^t$

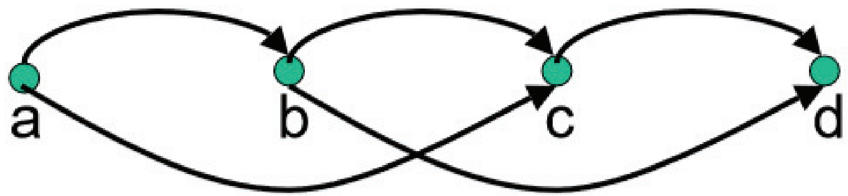
$(2,1) \in R^t \wedge (1,2) \in R \rightarrow (2,2) \in R^t$ (*Must be done iteratively)

So, $R^t = R \cup \{(2,1), (3,2), (3,3), (2,2)\}$

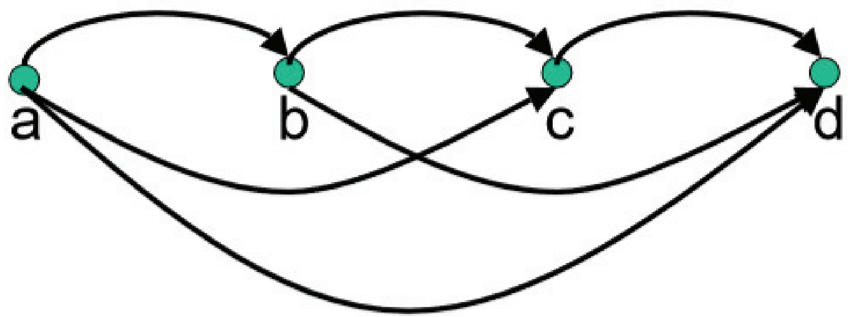
Graphical Construction of Transitive Closure



Original Relation



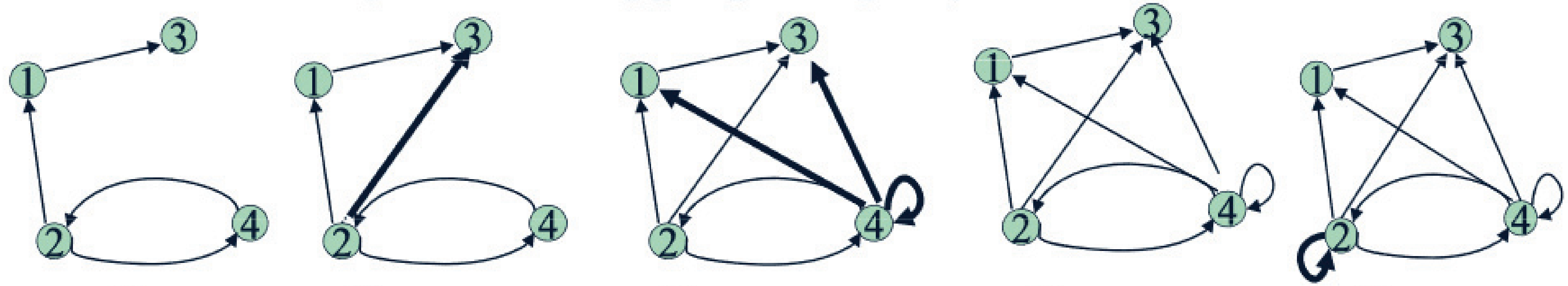
Original Relation plus 2-jumps



Original Relation plus 2-jumps, 3-jump

Warshall's Algorithm

- Main idea: a path exists between two vertices i, j, iff
  - there is an edge from i to j; or
  - there is a path from i to j going through vertex 1; or
  - there is a path from i to j going through vertex 1 and/or 2; or
  - there is a path from i to j going through vertex 1, 2, and/or 3; or
  - ...
  - there is a path from i to j going through any of the other vertices



$$R_0 = \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{matrix} \qquad R_1 = \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & \mathbf{1} & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{matrix} \qquad R_2 = \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ \mathbf{1} & 1 & \mathbf{1} & \mathbf{1} \end{matrix} \qquad R_3 = \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{matrix} \qquad R_4 = \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & \mathbf{1} & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{matrix}$$

Transitive Closure of a Directed Graph

- $G' = (V, E')$: transitive closure of $G = (V, E)$, where
  - ► $E' = \{ (v_i, v_j)$: there exists a path from v_i to v_j in $G \}$

- trivial solution : assign W such that

$$\omega_{ij} = \begin{cases} 1 \text{ if } (v_i, v_j) \in E \\ \infty \quad \text{otherwise} \end{cases}$$

  - ► run Floyd-Warshall algorithm on W
  - ► $d_{ij}^{\,n} < n \Rightarrow$ there exists a path from v_i to v_j ,
    i.e., $(v_i, v_j) \in E'$
  - ► $d_{ij}^{\,n} = \infty \Rightarrow$ no path from v_i to v_i ,
    i.e., $(v_i, v_j) \notin E'$
  - ► running time : $\Theta(n^3) = \Theta(V^3)$

Transitive Closure of a Directed Graph

- Better $\Theta(V^3)$ algorithm : saves time and space.
  - ▶ W = adjacency matrix : $\omega_{ij} = \begin{cases} 1 & \text{if } i = j \text{ or } (v_i, v_j) \in E \\ 0 & \text{otherwise} \end{cases}$
  - ▶ run Floyd-Warshall algorithm by replacing "min" → "$\vee$" & "+" → "$\wedge$"
- define $t_{ij}^{m} = \begin{cases} 1 \text{ if } \exists \text{ a path from } v_i \text{ to } v_j \text{ with all intermediate vertices from } V_m \\ 0 \text{ otherwise} \end{cases}$
  - ▶ $t_{ij}^{n} = 1 \Rightarrow (v_i, v_j) \in E'$ & $t_{ij}^{n} = 0 \Rightarrow (v_i, v_j) \notin E'$
- recursive definition for $t_{ij}^{m} = t_{ij}^{m-1} \vee (t_{im}^{m-1} \wedge t_{mj}^{m-1})$ with $t_{ij}^{0} = \omega_{ij}$

Transitive Closure of a Directed Graph

```
T-CLOSURE (G )
  ▶ T = ( t_ij ) is an n x n boolean matrix
    for i ← 1 to n do
      for j ← 1 to n do
        if  i = j  or ( v_i , v_j ) ∈ E  then
            t_ij ← 1
        else
            t_ij ← 0
    for  m ← 1 to n do
      for i ← 1 to n do
        for j ← 1 to n do
            t_ij ← t_ij ∨ ( t_im ∧ t_mj )
```

Johnson’s all-pairs algorithm

- Johnson’s演算法可用於計算All pairs shortest path問題。
- 在邊的數量不多的時候，如$|E|=O(|V|\log|V|)$時，能有比Warshall-Floyd演算法較佳的效能。
- 其輸入需求是利用Adjacency list表示的圖。

Johnson's Algorithm for Sparse Graphs

(1) Preserving shortest paths by edge reweighting (重新調整權重) :

- L1 : given $G = (V, E)$ with $\omega : E \rightarrow R$
  - ▶ let $h : V \rightarrow R$ be any weighting function (real) on the vertex set
  - ▶ define $\hat{\omega}(\omega, h) : E \rightarrow R$ as $\hat{\omega}(u, v) = \omega(u, v) + h(u) - h(v)$
  - ▶ let $p_{0k} = < v_0, v_1, \ldots, v_k >$ be a path from v_0 to v_k

(a) $\hat{\omega}(p_{0k}) = \omega(p_{0k}) + h(v_0) - h(v_k)$

(b) $\omega(p_{0k}) = \delta(v_0, v_k)$ in $(G, \omega) \Leftrightarrow \hat{\omega}(p_{0k}) = \hat{\delta}(v_0, v_k)$ in $(G, \hat{\omega})$

(c) (G, ω) has a neg-wgt cycle $\Leftrightarrow$ $(G, \hat{\omega})$ has a neg-wgt cycle

Observation

$$\hat{w}(\mathbf{p_{ok}}) = w(v_0, v_1) + h(v_0) - \cancel{h(v_1)}$$

$$+ w(v_1, v_2) + \cancel{h(v_1)} - \cancel{h(v_2)} + \ldots$$

$$+ w(v_{k-2}, v_{k-1}) + \cancel{h(v_{k-2})} - \cancel{h(v_{k-1})}$$

$$+ w(v_{k\_1}, v_k) + \cancel{h(v_{k-1})} - h(v_k)$$

$$= w(v_0, v_1) + w(v_1, v_2) + \ldots + w(v_{k-2}, v_{k-1}) + w(v_{k\_1}, v_k) + h(v_0) - h(v_k)$$

$$= w(\mathbf{p}) + h(v_0) - h(v_k)$$

Under the new weighting scheme, weight of every path between v_0 and v_k is incremented by constant amount (decremented if the constant is negeative).

So shortest paths remain the same under the new weights.

Johnson's Algorithm for Sparse Graphs

- proof (a): $\hat{\omega}(p_{0k}) = \sum_{1 \le i \le k} \hat{\omega}(v_{i-1}, v_i)$
  $= \sum_{1 \le i \le k} (\omega(v_{i-1}, v_i) + h(v_0) - h(v_k))$
  $= \sum_{1 \le i \le k} \omega(v_{i-1}, v_i) + \sum_{1 \le i \le k} (h(v_0) - h(v_k))$
  $= \omega(p_{0k}) + h(v_0) - h(v_k)$

- proof (b): ($\Rightarrow$) show $\omega(p_{0k}) = \delta(v_0, v_k) \Rightarrow \hat{\omega}(p_{0k}) = \hat{\delta}(v_0, v_k)$ by contradiction.
  - ▶ Suppose that a $\exists$ shorter path p_{0k}' from v_0 to v_k in $(G, \hat{\omega})$, then $\hat{\omega}(p_{0k}') < \hat{\omega}(p_{0k})$

- due to (a) we have
  - $\omega(p_{0k}') + h(v_0) - h(v_k) = \hat{\omega}(p_{0k}') < \hat{\omega}(p_{0k}) = \omega(p_{0k}) + h(v_0) - h(v_k)$
    $\omega(p_{0k}') + h(v_0) - h(v_k) < \omega(p_{0k}) + h(v_0) - h(v_k)$
    $\omega(p_{0k}') < \omega(p_{0k}) \Rightarrow$ contradicts that p_{0k} is a shortest path in (G, ω)

Johnson's Algorithm for Sparse Graphs

- proof (b): (<=) similar

- proof (c): ($\Leftrightarrow$) consider a cycle $c = < v_0 , v_1 , \ldots , v_k = v_0 >$.

  Due to (a)

  ► $\hat{\omega}(c) = \sum_{1 \le i \le k} \hat{\omega}(v_{i-1}, v_i) = \omega(c) + h(v_0) - h(v_k)$

  $= \omega(c) + h(v_0) - h(v_0) = \omega(c)$ since $v_k = v_0$

  ► $\hat{\omega}(c) = \omega(c)$.

QED

Still need to ensure that weight of every edge is nonnegative

$w_{new}(u,v) = w(u,v) + h(u) - h(v) \ge 0$

Johnson's Algorithm for Sparse Graphs

(2) Producing nonnegative edge weights by reweighting :

- given (G, ω) with $G = (V, E)$ and $\omega : E \rightarrow R$
  construct a new graph (G', ω') with $G' = (V', E')$ and

  add a node s to the existing network,
  and add an edge from s to every node

  $\omega' = E' \rightarrow R$
  - ▶ $V' = V \cup \{ s \}$ for some new vertex $s \notin V$
  - ▶ $E' = E \cup \{ (s, v) : v \in V \}$
  - ▶ $\omega'(u,v) = \omega(u,v) \quad \forall (u,v) \in E$ and $\omega'(s,v) = 0 , \quad \forall v \in V$

  •Weight of new edges are 0

- vertex s has no incoming edges $\Rightarrow s \notin R_v$ for any v in V
  - ▶ no shortest paths from $u \neq s$ to v in G' contains vertex s
  - ▶ (G', ω') has no neg-wgt cycle $\Leftrightarrow$ (G, ω) has no neg-wgt cycle

Johnson's Algorithm for Sparse Graphs

- suppose that G and G' have no neg-wgt cycle

- L2 : if we define $h(v) = \delta(s, v) \quad \forall v \in V$ in G' and $\hat{\omega}$ according to L1.
  - ▶ we will have $\hat{\omega}(u,v) = \omega(u,v) + h(u) - h(v) \geq 0 \ \forall v \in V$

proof : for every edge $(u, v) \in E$

$$\delta(s, v) \leq \delta(s, u) + \omega(u, v) \text{ in } G' \text{ due to triangle inequality}$$

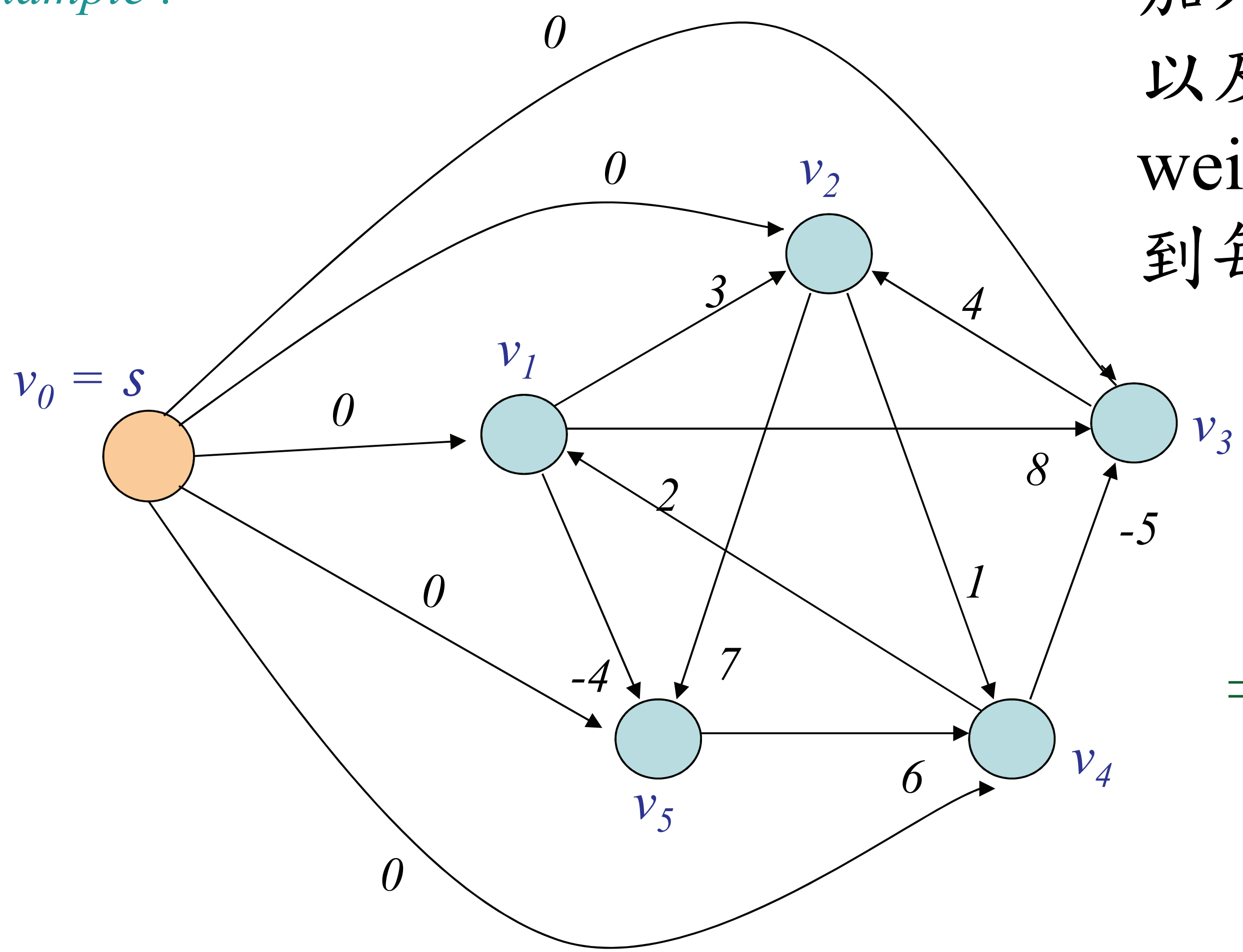
$$h(v) \leq h(u) + \omega(u, v) \Rightarrow 0 \leq \omega(u, v) + h(u) - h(v) = \hat{\omega}(u, v)$$

Johnsons Algorithm

- Start with the original graph
- Add the new vertex s and the new edges with 0 weight to all other vertices
- Run Bellman-ford with source s, and original weights to compute shortest path weights p(s,v) to every vertex v.
  - Can we run Dijkstra instead?
- Compute the new weights for the original edges:
  - $w_{new}(u, v) = w(u, v) + p(s, u) - p(s, v)$
  - Can get rid of the new vertex and edges at this point
- Run Dijkstra to compute the shortest paths

Johnson’s Algorithm for Sparse Graphs

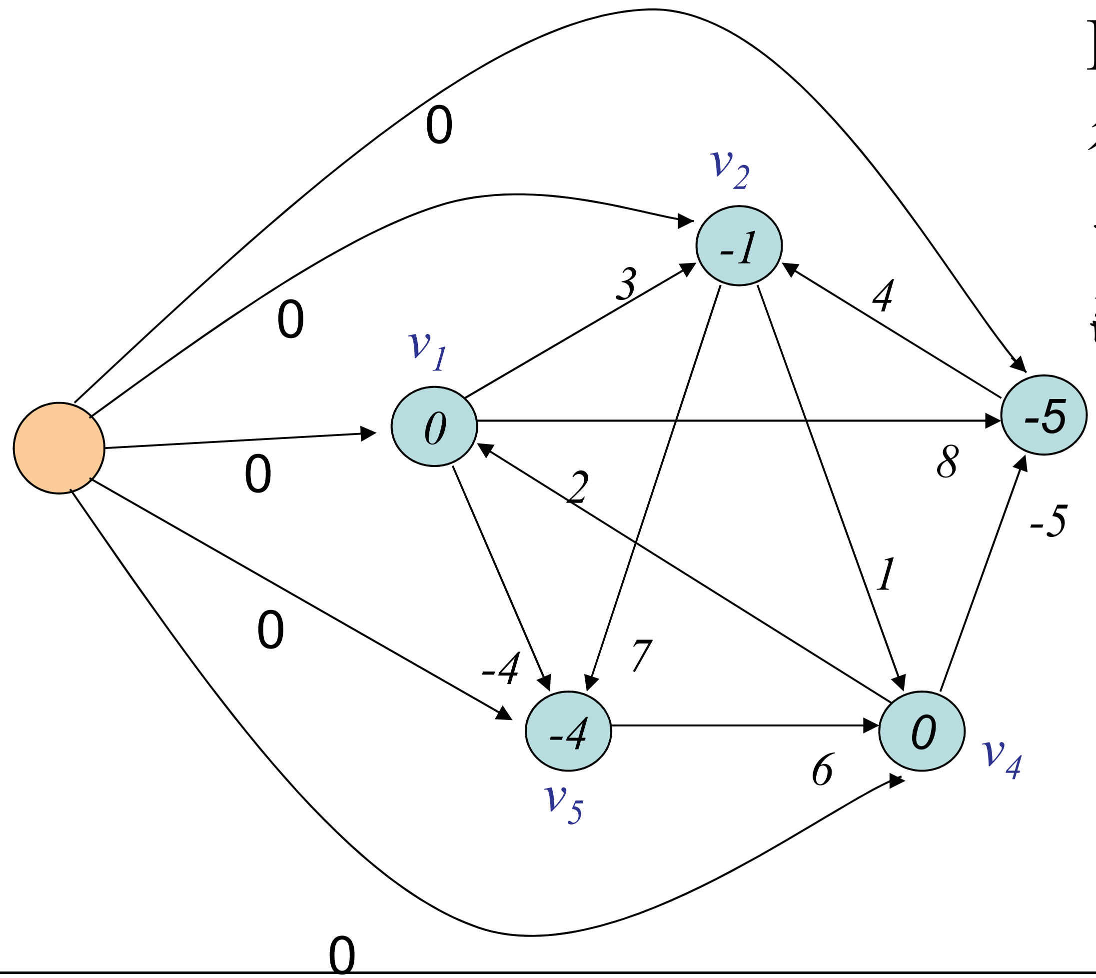
example :



加入一個點s，以及自s拉一條weight為0的邊到每一點。

$\Rightarrow (G', \omega')$

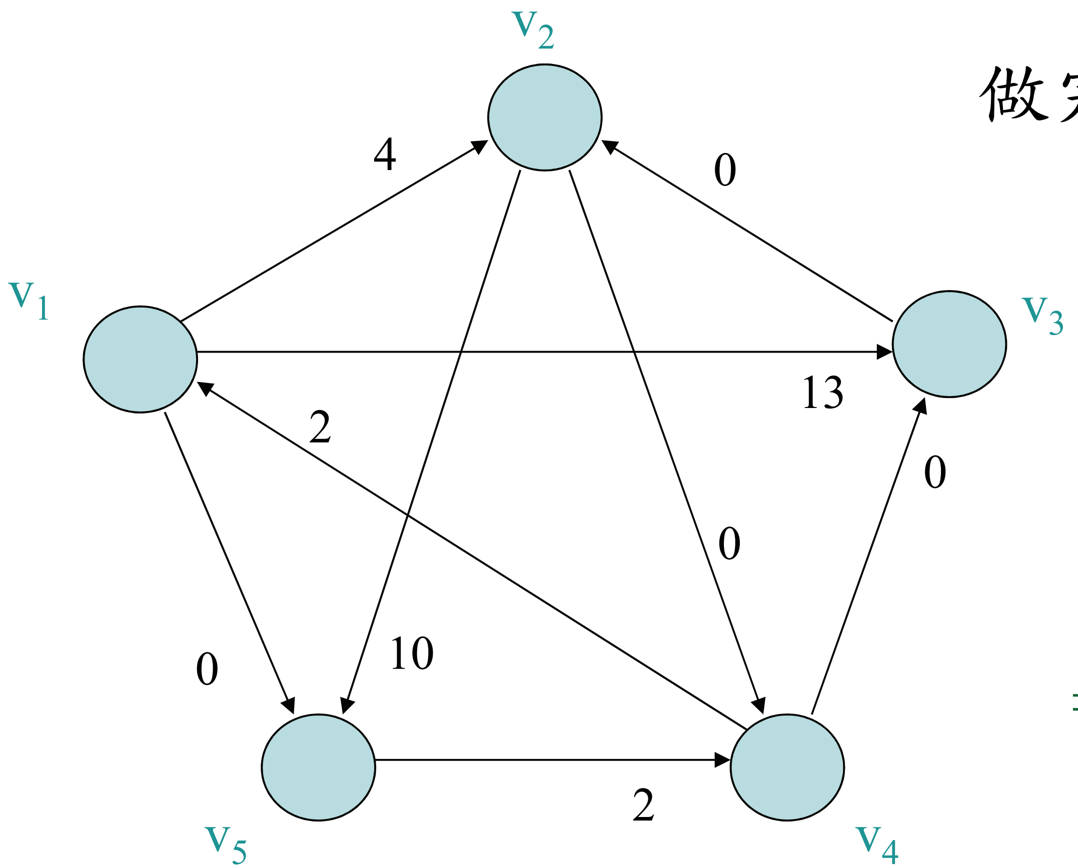
Johnson's Algorithm for Sparse Graphs
Edge Reweighting



執行Bellman-Ford演算法，得到自s出發每一點的最短距離。

$\Rightarrow (G', \omega')$ with $h(v)$

Johnson's Algorithm for Sparse Graphs
Edge Reweighting



做完reweighting

$\Rightarrow (G, \hat{\omega})$

Johnson's Algorithm for Sparse Graphs

Computing All-Pairs Shortest Paths

- adjacency list representation of G.

- returns $n \times n$ matrix $D = (d_{ij})$ where

  $d_{ij} = \delta_{ij}$,

  or reports the existence of a neg-wgt cycle.

Johnson's Algorithm for Sparse Graphs

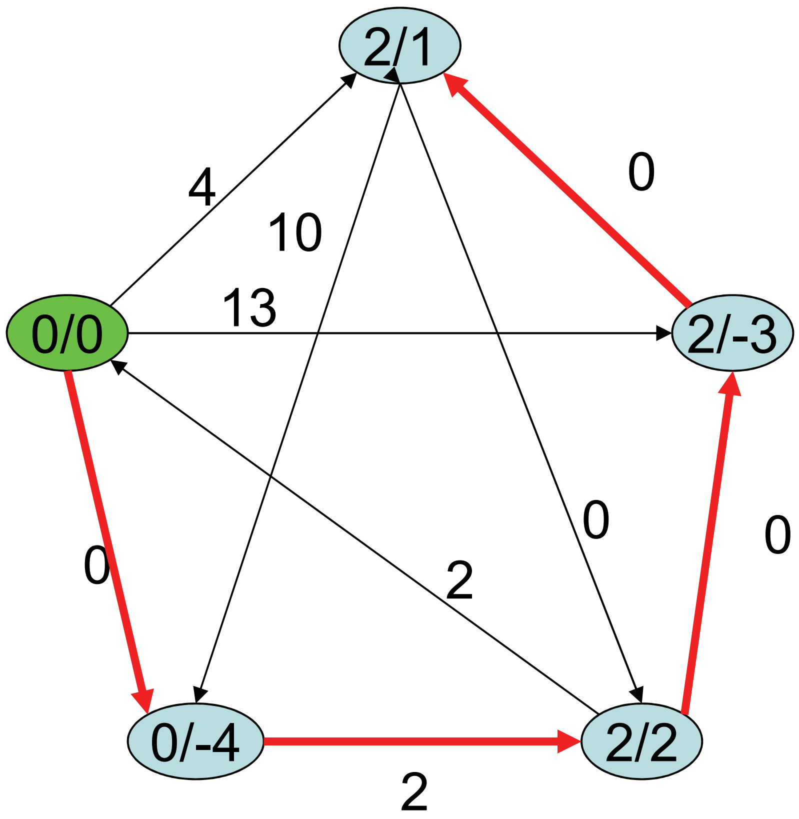
- JOHNSON(G,ω)

```
    ▶ D=(d_ij) is an nxn matrix
    ▶ construct ( G' = (V', E') , ω' )  s.t. V' = V ∪ {s};  E' = E U { (s,v) : ∀ v ∈ V }
    ▶ ω'(u,v) = ω(u,v),  ∀(u,v) ∈ E   &   ω'(s,v) = 0   ∀v ∈ V
    if BELLMAN-FORD(G', ω', s) = FALSE then
        return "negative-weight cycle"
    else
        for each vertex v ∈ V'- {s} = V do
            h[v] ← d'[v] ▶ d'[v] = δ'(s,v) computed by BELLMAN-FORD(G', ω', s)
        for each edge (u,v) ∈ E do
            ω̂(u,v) ← ω(u,v) + h[u] – h[v]      ▶ edge reweighting
        for each vertex u ∈ V do
            run DIJKSTRA(G, ω̂, u) to compute d̂[v] = δ̂(u,v)  for all v in V ∈ (G,ω̂)
            for each vertex v ∈ V do
                d_uv = d̂[v] – ( h[u] – h[v] )
     return D
```

Johnson's Algorithm for Sparse Graphs

- running time : $O(V^2 \lg V + EV)$
  - ► edge reweighting
    - BELLMAN-FORD(G', ω', s) : $O(EV)$
    - computing $\hat{\omega}$ values : $O(E)$
  - ► $|V|$ runs of DIJKSTRA : $|V| \times O(V \lg V + EV)$
    - $= O(V^2 \lg V + EV)$;
    - PQ = fibonacci heap

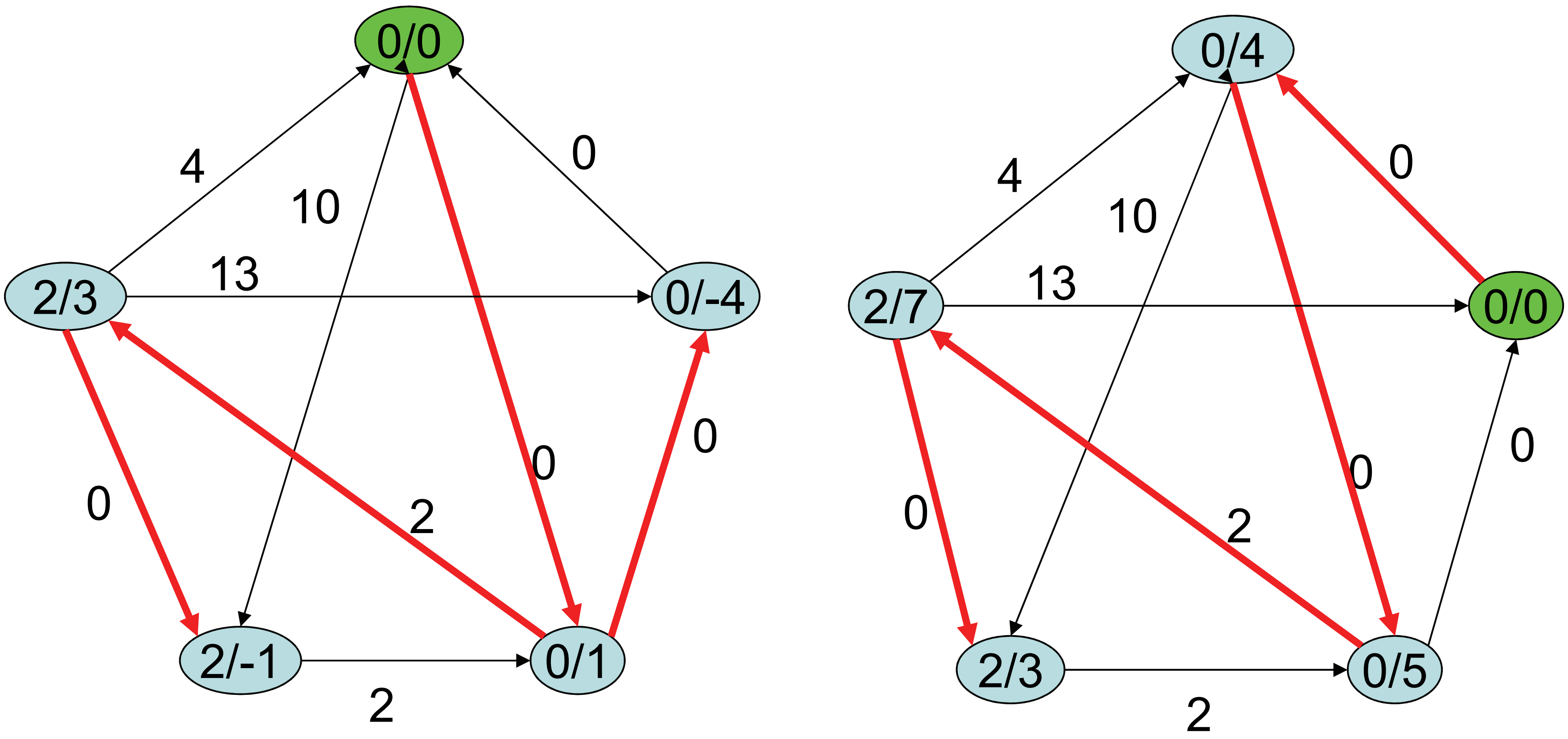
Johnson’s algorithm 範例



紅線部分是Shortest-paths tree。

點的數字a/b代表自出發點(綠色點)出發，到達該點的最短路徑(Reweighting後的圖/原圖)。

Johnson’s algorithm 範例



Johnson's algorithm 範例