Unit 9
Shortest Paths

T.H. Cormen et al., “Introduction to Algorithms”, 3rd ed., Chapters 24, 25

Optimal Substructure of Shortest Paths

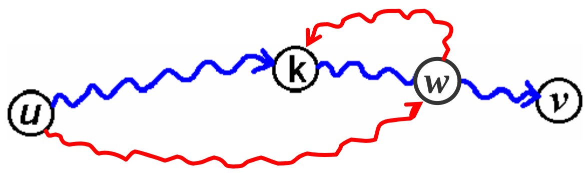
☛ Consider the following problems, and examine if they exhibit optimal substructures:

○ • Find a shortest path on a di-graph.

✗ • Find a longest simple path on a di-graph.

○ • Find a shortest path on a dag (directed acyclic graph).

○ • Find a longest simple path on a dag.



Shortest-Paths Problems

- There are several variants:
  - Single-source (✓)
  - Single-destination
  - Single-pair
  - All-pairs (✓)
- And two types of instances:
  - With only non-negative-weight edges
  - With some negative-weight edges but no negative-weight cycles

Optimal substructure of a shortest path

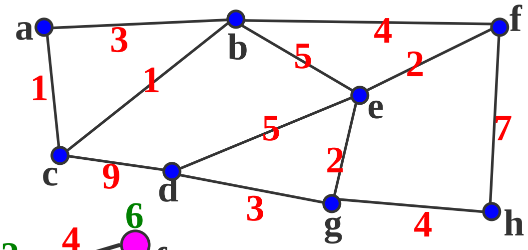
- For a weighted graph with no negative-weight cycles, if path $u_1 \to u_2 \to \ldots \to u_k$ is a shortest path from u_1 to u_k, then:

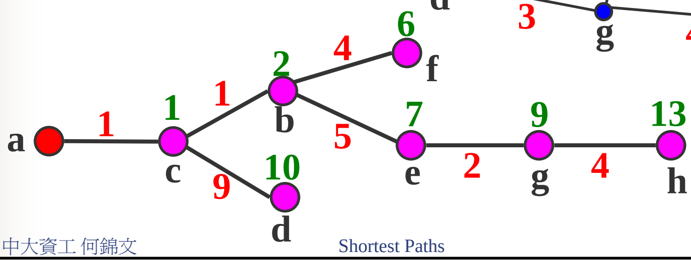
  $u_i \to u_{i+1} \to \ldots \to u_j$ is also a shortest path from u_i to u_j, for any $1 \le i \le j \le k$.
- This is the reason why no algorithms for single-pair shortest-paths problem run asymptotically faster than the best single-source algorithm in the worst case.

Shortest-paths (spanning) trees

- With the optimal substructure of shortest paths, the output of the single-source shortest-paths problem can be represented by a ***shortest-paths (spanning) tree***.

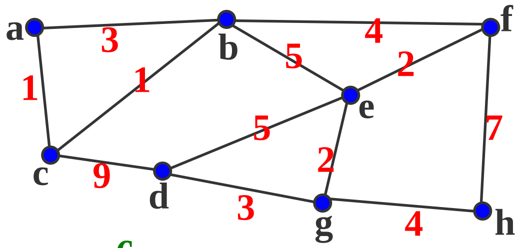
Here, **a** is the source vertex

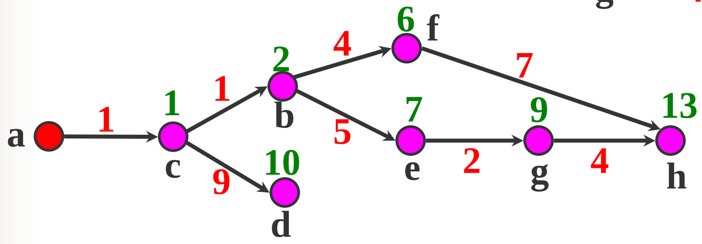




Shortest-paths (spanning) DAG

- All shortest (di)-paths from a given source form a DAG.





The Bellman-Ford algorithm (想法)

- The ***Bellman-Ford algorithm*** solve the single-source shortest-paths problem in the general case in which edge weights may be negative.
- Key observation: a shortest path has at most $|V|-1$ hops.
- The idea: find shortest paths with one hop (from the source vertex) first, and then those with two hops, and so on.

The Bellman-Ford algorithm (pseudo code)

Bellman-Ford(G, s)
Initialize(G, s) // $\pi[v] \leftarrow$NIL, $d[v] \leftarrow \infty$, $\forall$ v; $d[s] \leftarrow 0$
for $i \leftarrow 1$ **to** $|V|-1$ **do**
    for each edge $uv \in E$ **do**
        if $d[v] > d[u] + w(u, v)$
        then $d[v] \leftarrow d[u] + w(u, v)$
            $\pi[v] \leftarrow u$
for each edge $uv \in E$ **do**
    if $d[v] > d[u] + w(u, v)$
    then return False //G has a negative cycle
return True

Relaxation of edge uv

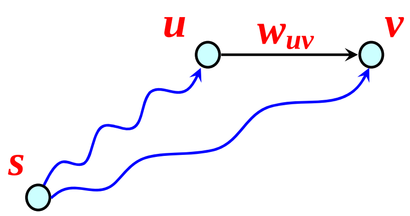
Time: $O(VE)$

The Idea of Relaxation

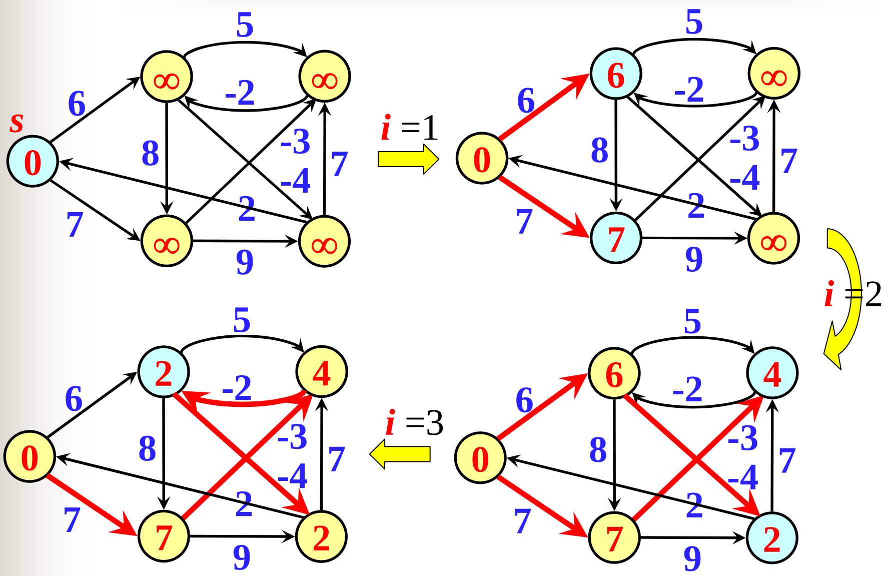
if $d[v] > d[u] + w(u, v)$

then $d[v] \leftarrow d[u] + w(u, v)$

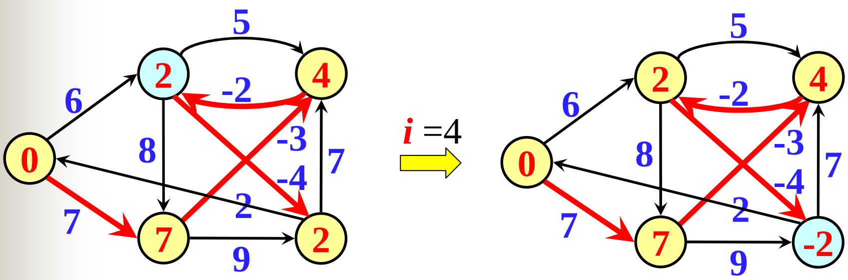
$\pi[v] \leftarrow u$



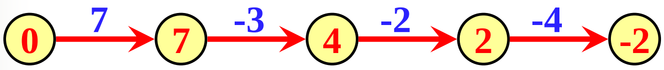
The Bellman-Ford algorithm (例 1/2)



The Bellman-Ford algorithm (例 2/2)



相對的 SPST:



The Bellman-Ford algorithm (實做考量)

- In each iteration of the first loop, is it necessary to do the relaxation for each edge $uv \in E$?
- Similarly, is it necessary to do the checking for each edge $uv \in E$ in the second loop?
- Is it necessary to do the relaxations exactly $|V|-1$ times?
- Can you apply the above observations to the implementation of the Bellman-Ford algorithm?

Single-source shortest paths in DAGs

DAG-Shortest-Paths(G, s) 複習

Topologically sort the vertices of G

Initialize(G, s) // $\pi[v] \leftarrow$NIL, $d[v] \leftarrow \infty$, $\forall\ v$; $d[s] \leftarrow 0$

for each vertex u taken in topological order **do**

  for each vertex $v \in Adj[u]$ **do** // do Relax(u, v)

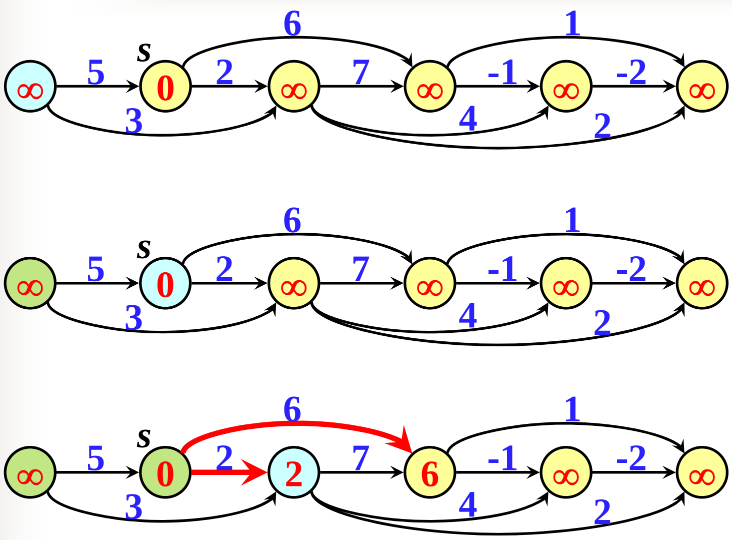
    if $d[v] > d[u] + w(u, v)$

    then $d[v] \leftarrow d[u] + w(u, v)$

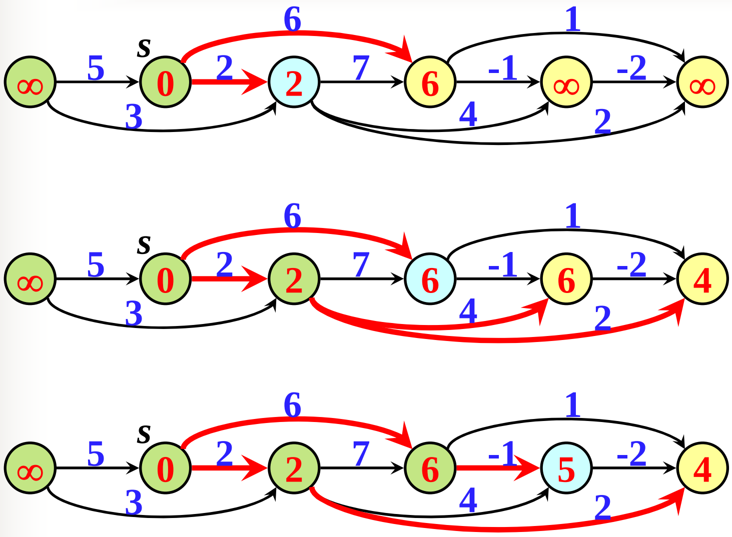
      $\pi[v] \leftarrow u$

Time: $\Theta(V+E)$

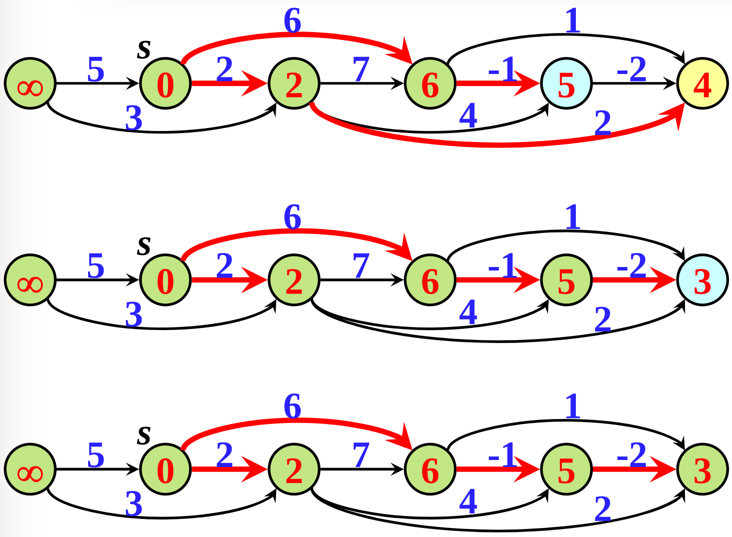
Shortest paths in DAGs (例 1/3)



Shortest paths in DAGs (例 2/3)



Shortest paths in DAGs (例 3/3)



Longest paths in DAGs

- A ***critical path*** of a DAG is a longest path through the DAG.
- We can find a critical path by either
  - Negating the edge weights and running DAG-Shortest-Paths or
  - Running DAG-Shortest-Paths, with the following modification:

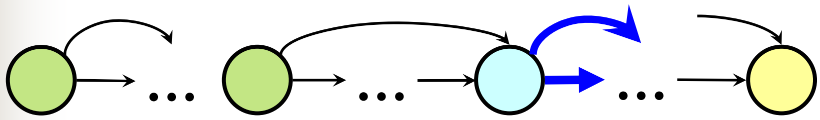
    Replace "∞" by "-∞" in the initialization procedure and ">" by "<" in the relaxation procedure.

DP as Problem Solving in DAGs

- Many DP computations can be viewed as solving some problems in dags.
- For example, rod cutting and LCS, discussed in Unit 5, can be viewed as finding longest paths in dags.
- There are two types of (bottom-up) implementations for DP computation.

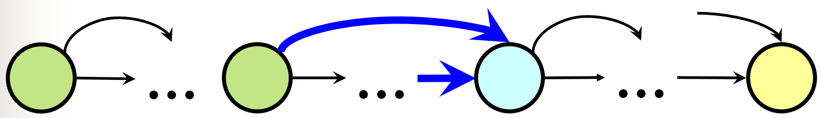
Type I Computation

- 狀態：存在藍點的資料, 在處理前已經正確.
- 計算：將藍點的資料, 沿箭頭送到相關的點（黃）, 並在那些點上執行計算.



Type II Computation

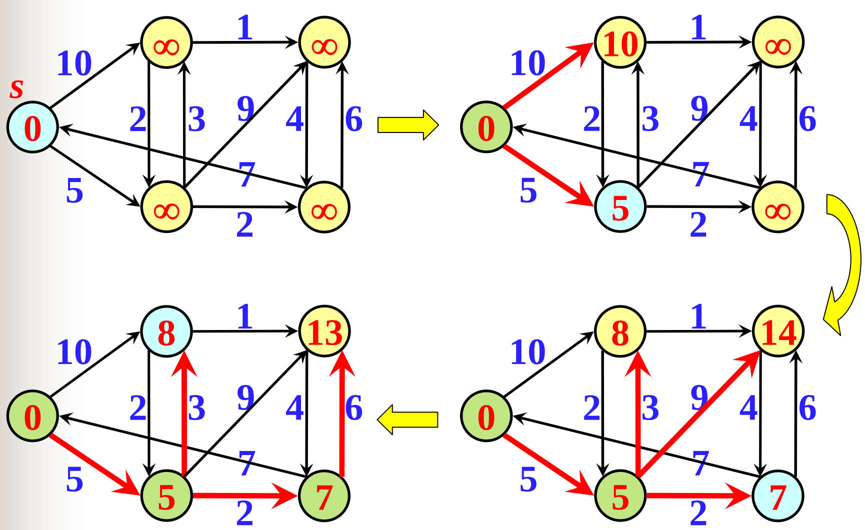
- 狀態：存在藍點的資料, 在處理前不一定正確, 但處理後資料會正確.
- 計算：沿箭頭方向從資料已經正確的點（綠）, 抓取資料到藍點上來計算.



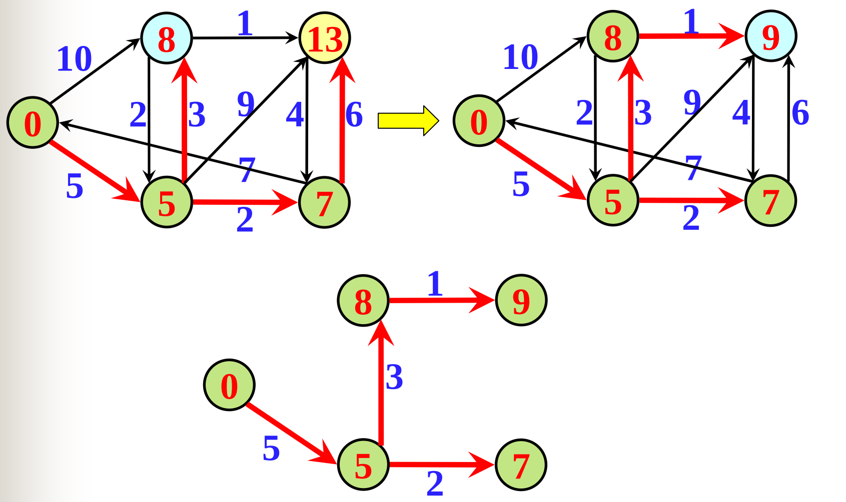
Dijkstra’s algorithm

- Dijkstra’s algorithm solves the single-source shortest-paths problem on a weighted di-graph for the case in which ***all edges weights are nonnegative***.
- Key observation (greedy-choice property): for edges directed from source, an edge with minimal weight must be in a shortest-paths tree.

Dijkstra’s algorithm (例 1/2)



Dijkstra’s algorithm (例 2/2)



Dijkstra’s algorithm (pseudo code)

```
Dijkstra(G, s)
Initialize(G, s) // π[v]←NIL, d[v]←∞, ∀ v; d[s]←0
Q ← V   //Built a priority queue Q for V with d[v] as key
While (Q ≠ ∅) do
   u ← Extract-Min(Q)
   for each v ∈ Adj[u]  do
      if v ∈ Q and d[v] > d[u] + w(u, v) then
         π[v] ← u
         d[v] ← d[u] + w(u, v)
         Change-Priority(Q, v)
```

Dijkstra's algorithm (分析)

- Let $n = |V(G)|$, $m = |E(G)|$.
- Since the implementation of Dijkstra's algorithm is similar to that of Prim's algorithm, the running time of both algorithms are the same:
  - adjacency lists + (binary or ordinary) heap: $O((m+n) \log n) = O(m \log n)$
  - adjacency matrix + unsorted list: $O(n^2)$
  - adjacency lists + Fibonacci heap: $O(n \log n + m)$

Difference constraints (定義＋例)

- Given n events, assign each event i a starting execution time x_i such that these assignments satisfy m given constrains of the form $x_j - x_i \le b_k$ where $1 \le i, j \le n$, and $1 \le k \le m$.
- A special case of the ***linear programming*** (LP) problem.

$$
\begin{aligned}
x_1 - x_2 &\le 0 \\
x_1 - x_5 &\le -1 \\
x_2 - x_5 &\le 1 \\
x_3 - x_1 &\le 5 \\
x_4 - x_1 &\le 4 \\
x_4 - x_3 &\le -1 \\
x_5 - x_3 &\le -3 \\
x_5 - x_4 &\le -3
\end{aligned}
$$

Linear Programming

In general, $n \neq m$.

- Maximize (or minimize)

  $c_1x_1 + c_2x_2 + \cdots + c_nx_n$

  subject to

  $a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \leq$ (or $\geq$ or $=$) b_1

  $a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \leq$ (or $\geq$ or $=$) b_2

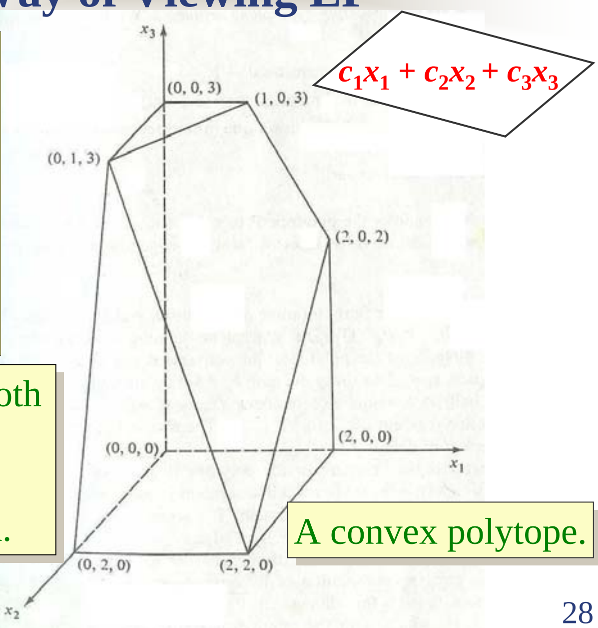
  $\vdots$

  $a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \leq$ (or $\geq$ or $=$) b_m

- Given $A_{m\times n}$, $b_{m\times 1}$, $c_{n\times 1}$, find $x_{n\times 1}$ to max $c^{T}x$ s.t. $Ax \leq b$.
- If all the numbers are required to be integers, the problem is called ***integer linear programming*** (ILP).

A Geometric Way of Viewing LP

$$
\begin{array}{rrrl}
x_1 + & x_2 + & x_3 & \leq 4 \\
x_1 & & & \leq 2 \\
 & & x_3 & \leq 3 \\
 & 3x_2 + & x_3 & \leq 6 \\
x_1 & & & \geq 0 \\
 & x_2 & & \geq 0 \\
 & & x_3 & \geq 0
\end{array}
$$



It can be viewed as both continuous and combinatorial optimization problem.

A convex polytope.

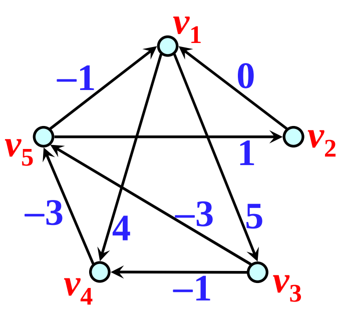
Notes on Linear Programming

- Many problems can be reduced to LP problems.
- A well-known algorithm for LP: ***simplex method***.
- LP can be solved in polynomial time (not by SM).
- In general, ILP are hard problems. There is no known polynomial-time algorithm for ILP, yet no one has ever proved that such an algorithm is not possible.
- Some LP (with integer coefficients) can be proved to have integer solutions: e.g. this problem, single-pair shortest-paths, max-flow, …etc.

Constraint graphs

- Each variable x_i ↔ vertex v_i, each constraint $x_j - x_i \le b_k$ ↔ an edge v_iv_j with weight b_k.

$$
\begin{aligned}
x_1 - x_2 &\le 0 \\
x_1 - x_5 &\le -1 \\
x_2 - x_5 &\le 1 \\
x_3 - x_1 &\le 5 \\
x_4 - x_1 &\le 4 \\
x_4 - x_3 &\le -1 \\
x_5 - x_3 &\le -3 \\
x_5 - x_4 &\le -3
\end{aligned}
$$



Observation 1

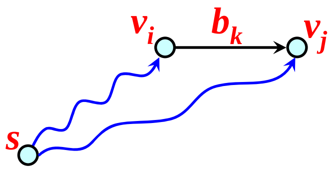
- Consider a cycle in the constraint graph: $v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_t \rightarrow v_1$. It corresponds to constraints:

$$\left.\begin{array}{l} x_2 - x_1 \le b_1 \\ x_3 - x_2 \le b_2 \\ \quad \vdots \\ x_1 - x_t \le b_t \end{array}\right\} \Rightarrow b_1 + b_2 + \cdots + b_t \ge 0$$

- If the given difference constraints instance ***has a solution***, then the corresponding constraint graph ***contains no negative-weight cycles***.
- Is the converse correct?

Observation 2

- Let $\delta(u, v)$ = the shortest-path weight from u to v.
- For a constraint $x_j - x_i \le b_k$ and its corresponding edge, we have:



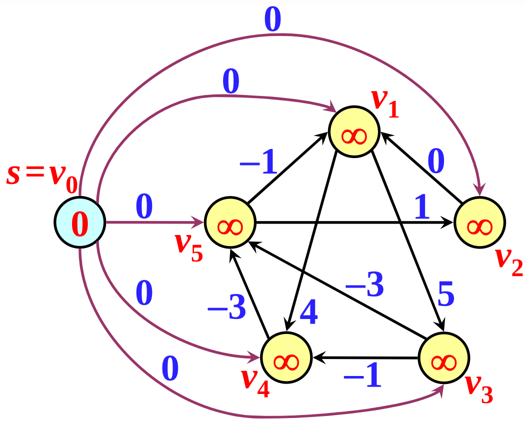
$$\delta(s, v_j) \le \delta(s, v_i) + b_k$$
$$\Downarrow$$
$$\delta(s, v_j) - \delta(s, v_i) \le b_k$$

- Hence, by adding an additional vertex s as the source, and edges $sv_1, sv_2, \ldots, sv_n$ of zero weights, we can solve the problem with the assignments:

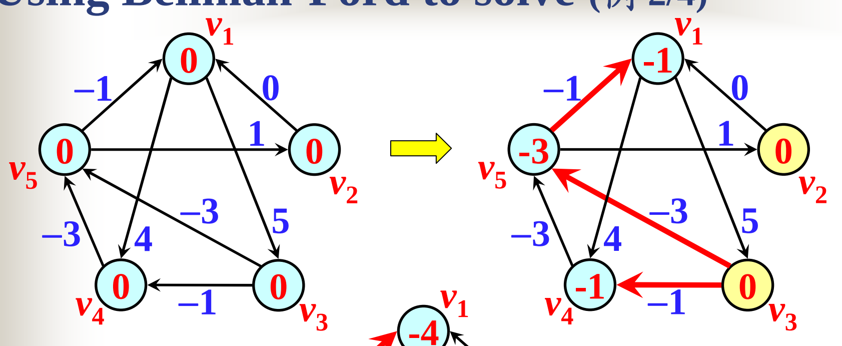
$$x_i \leftarrow \delta(s, v_i), \text{ for } 1 \le i \le n.$$

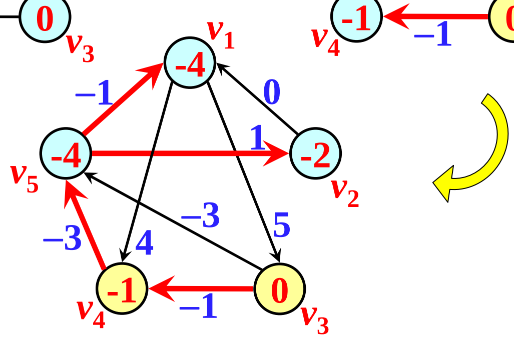
Using Bellman-Ford to solve (例 1/4)

$$
\begin{aligned}
x_1 - x_2 &\le 0 \\
x_1 - x_5 &\le -1 \\
x_2 - x_5 &\le 1 \\
x_3 - x_1 &\le 5 \\
x_4 - x_1 &\le 4 \\
x_4 - x_3 &\le -1 \\
x_5 - x_3 &\le -3 \\
x_5 - x_4 &\le -3
\end{aligned}
$$

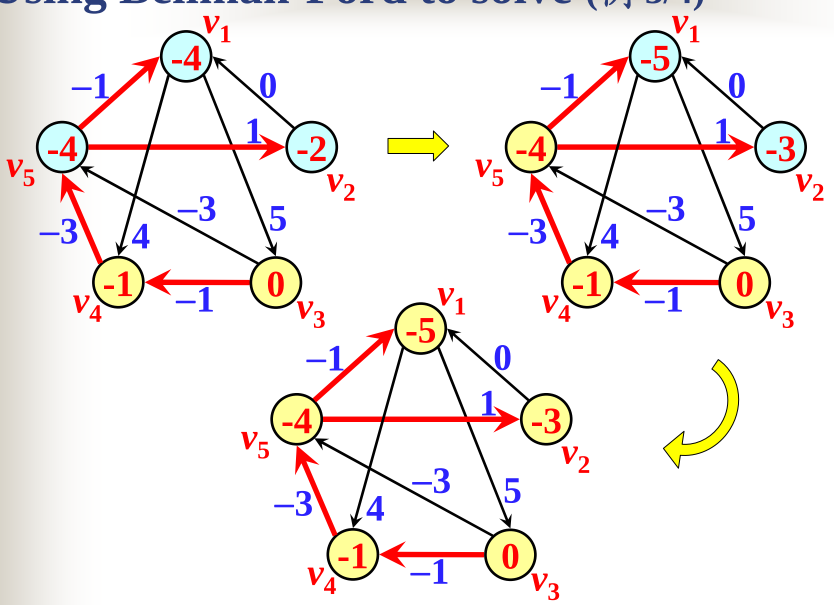


Using Bellman-Ford to solve (例 2/4)



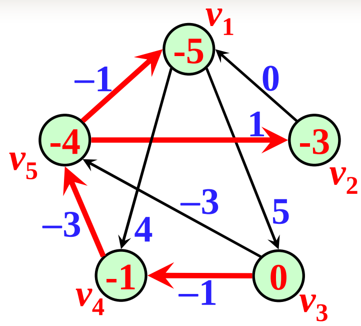


Using Bellman-Ford to solve (例 3/4)



Using Bellman-Ford to solve (例 4/4)

$$
\begin{aligned}
x_1 - x_2 &\le 0 \\
x_1 - x_5 &\le -1 \\
x_2 - x_5 &\le 1 \\
x_3 - x_1 &\le 5 \\
x_4 - x_1 &\le 4 \\
x_4 - x_3 &\le -1 \\
x_5 - x_3 &\le -3 \\
x_5 - x_4 &\le -3
\end{aligned}
$$



$(x_1, x_2, x_3, x_4, x_5)$
$= (-5, -3, 0, -1, -4)$ or
$= (0, 2, 5, 4, 1)$

Note: $(x_1+d, x_2+d, x_3+d, x_4+d, x_5+d)$ is also a solution for any constant d.

All-Pairs Shortest-Path Problem (定義)

- Given a weighted directed graph $G(V, E)$ with a weight function $w : E(G) \to \Re$ (containing no negative-weight cycles), find a shortest path from x to y for every pairs of vertices x and y .
- The problem can be solved by running a single-source shortest-paths algorithm $|V|$ times.
- It is ok for the case that all edge weights ≥ 0.
- For the case that negative-weight edges are allowed, there are several more efficient algorithms.

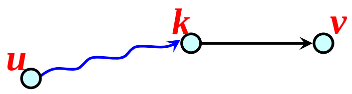
Using matrix multiplication (設計 1/2)

- The algorithm assumes the input graph is given by an adjacency-matrix.
- Key observation: a shortest path has at most $|V|-1$ hops.
- The idea: find shortest paths with one hop first (the input matrix), and then those with two hops, and so on.
- Let $\ell^{(m)}[u,v]$ be the weight of a shortest path from u to v consisting of at most m edges
- Hence, $\ell^{(1)}[u,v] = w[u,v]$.

Using matrix multiplication (設計 2/2)

- We have the following recursive formula:

$$\ell^{(m)}[u,v] = \min(\ell^{(m-1)}[u,v], \min_{1\le k\le n}\{\ell^{(m-1)}[u,k]+w[k,v]\})$$



- The computation is very similar to that of matrix multiplication. The computation sequence :

  compute $L^{(1)}, L^{(2)},\ldots, L^{(n-1)}$

  where $L^{(m)} = L^{(m-1)}\cdot W = W^{m}$

Using matrix multiplication (pseudo-code)

$$\ell^{(m)}[u,v] = \min(\ell^{(m-1)}[u,v], \min_{1\le k\le n}\{\ell^{(m-1)}[u,k]+w[k,v]\})$$

```
Initialization( ); // ℓ[u,v] = w[u,v]
for (m = 1; m < n; m++)
  for (u = 1; u <= n; u++)
    for (v = 1; v <= n; v++)
      for (k = 1; k <= n; k++)
        ℓ[u,v] = min(ℓ[u,v], ℓ[u,k]+w[k,v])
```

Using matrix multiplication (分析)

- A naïve implementation needs ______ time.
- Can be improved to __________ by:

  compute $L^{(1)}, L^{(2)}, L^{(4)} = L^{(2)} \cdot L^{(2)}, L^{(8)} \ldots, L^{(m)}$

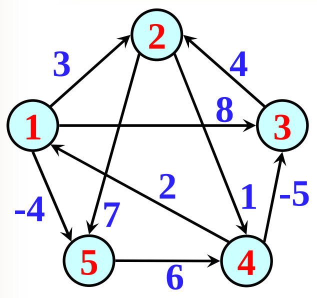
  for some $m = 2^k \geq n-1$.

Note: $L^{(n-1)} = L^{(n)} = L^{(n+1)} = \ldots$

Is the binary operation associative?

If the input has a negative cycle then …

Using matrix multiplication (例)



$$L^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$L^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 2 & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix}$$

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

The Floyd-Warshall algorithm (設計)

- Let $d^{(k)}[u,v]$ be the length of a shortest path from u to v using ***only vertices with indices*** $\leq k$.
- Hence, $d^{(0)}[u,v] = w[u,v]$ and $d^{(n)}[u,v]$ is the desired result for all pair u and v.
- A recursive formula:

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

Pf.: A $d^{(k)}[u,v]$ path either pass thru vertex k or not …

The Floyd-Warshall algorithm (實做)

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

```
Initialization( ); // d[u,v] = w[u,v]
for (k = 1; k <= n; k++)
  for (u = 1; u <= n; u++)
    for (v = 1; v <= n; v++)
        d[u,v] = min(d[u,v], d[u,k]+d[k,v])
```

Time = $O(n^3)$

Space = $O(n^2)$

Note: $d^{(0)}[u,v]$, $d^{(1)}[u,v]$, …, $d^{(n)}[u,v]$ can use the same memory locations.

If the input has a negative cycle then …

The Floyd-Warshall algorithm (例 1/3)

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

$D^{(0)}$

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

$D^{(1)}$

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

The Floyd-Warshall algorithm (例 2/3)

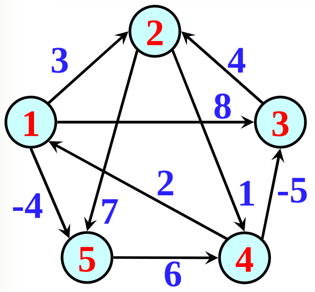
$D^{(1)}$	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	5	-5	0	-2
5	∞	∞	∞	6	0

$D^{(2)}$	1	2	3	4	5
1	0	3	8	4	-4
2	∞	0	∞	1	7
3	∞	4	0	5	11
4	2	5	-5	0	-2
5	∞	∞	∞	6	0

$D^{(3)}$	1	2	3	4	5
1	0	3	8	4	-4
2	∞	0	∞	1	7
3	∞	4	0	5	11
4	2	-1	-5	0	-2
5	∞	∞	∞	6	0

$D^{(4)}$	1	2	3	4	5
1	0	3	-1	4	-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

The Floyd-Warshall algorithm (例 3/3)

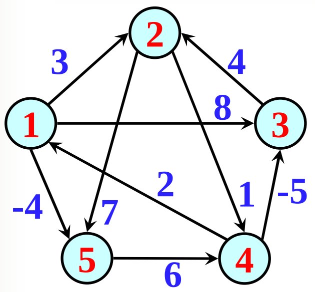


W	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	∞	-5	0	∞
5	∞	∞	∞	6	0

$D^{(4)}$	1	2	3	4	5
1	0	3	-1	4	-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

$D^{(5)}$	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

Constructing a shortest path (例)



D	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

P	1	2	3	4	5
1	0	5	5	5	0
2	4	0	4	0	4
3	4	0	0	2	4
4	0	3	0	0	1
5	4	4	4	0	0

$1 \rightarrow 2$

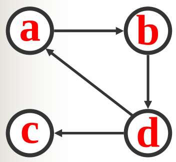
see 5 $1 \rightarrow 5 \rightarrow 2$

see 4 $1 \rightarrow 5 \rightarrow 4 \rightarrow 2$

$1 \rightarrow 5 \rightarrow 4 \rightarrow 3 \rightarrow 2$

Transcritive closure of a di-graph (定義)

- Given a directed graph $G(V, E)$ find a matrix T such that $t[i, j] = 1$ if there is a directed path from vertex i to j; otherwise $t[i, j] = 0$.



A	a	b	c	d
a	1	1	0	0
b	0	1	0	1
c	0	0	1	0
d	1	0	1	1

T	a	b	c	d
a	1	1	1	1
b	1	1	1	1
c	0	0	1	0
d	1	1	1	1

Transitive closure of a di-graph (設計)

- A simple way: assign each edge a weight of 1 and run the Floyd-Warshall algorithm.
- Or use a similar method: Let $t^{(k)}[u,v]$ be 1 if there is a path from u to v using only vertices with indices ≤ k ; otherwise the value is 0.
- $\therefore\ t^{(0)}[u,v] = 1$ if $uv \in E$; 0 otherwise.
- $\therefore T^{(n)}$ is the result we want, and we have:

$$\boxed{t^{(k)}[u,v] = t^{(k-1)}[u,v] \vee (t^{(k-1)}[u,k] \wedge t^{(k-1)}[k,v])}$$

Transitive closure of a di-graph (討論)

- The algorithm needs $O(n^3)$ time.
- There exists a more efficient algorithm with time complexity: $O(nm)$. (How?)
- However, the algorithm is still suitable for ***dense*** graphs.
- Is it possible to solve the problem in $O(n^2)$ time? (If it is possible, the algorithm is optimal.)

All-Pairs Shortest-Path Problem (討論)

- For sparse graphs with non-negative weight edges:
  - Run Dijkstra's algorithm V times
  - Time complexity: $O(V^2 \log V + VE)$
- For dense graphs:
  - Floyd-Warshall algorithm is suitable
  - Time complexity: $O(V^3)$
- How about sparse graphs with negative weight edges?

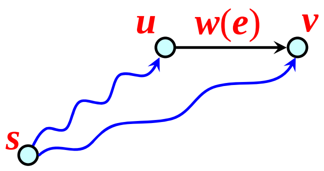
Johnson's algorithm for APSP (idea)

- ***Reweighting*** the weight function from w to w' s.t.:
  - For each edge e, $w'(e) \geq 0$. (RC1)
  - For any path P, P is a shortest path from u to v using w **if and only if** P is a shortest path from u to v using w'. (RC2)
- Then run Dijkstra's algorithm V times.
- Reweighting can be done in $O(VE)$ time.
- Hence, the time complexity of this algorithm is $O(V^2 \log V + VE)$.

How to reweight (RC1)

- For each vertex v, let $h(v) = \delta(s, v)$.
- For any edge $e = uv$, we have:

$$h(u) + w(e) \geq h(v)$$
$$\Downarrow$$
$$w(e) + h(u) - h(v) \geq 0$$



- Hence, by setting $w'(e) = w(e) + h(u) - h(v)$ for each edge $e = uv$, we can see that (RC1) is satisfied.

How to reweight (RC2)

- Let P be a path from v_0 to v_k : $v_0 \rightarrow v_1 \rightarrow \ldots \rightarrow v_k$.
  We can see that: $\boxed{w'(e) = w(e) + h(u) - h(v)}$

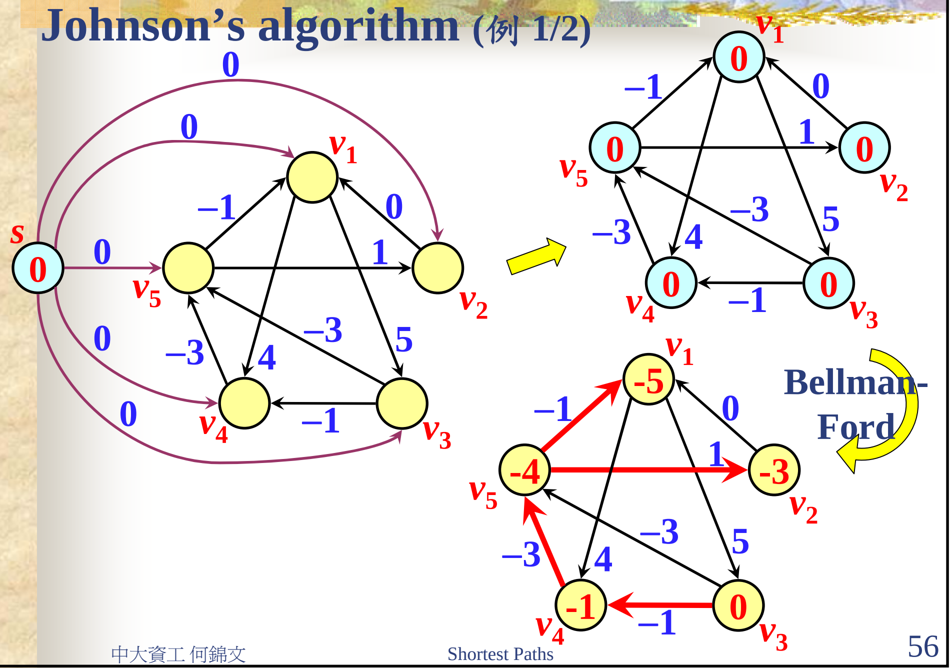
$$w'(P) = \Sigma_{1 \le i \le k}\, w'(v_{i-1}, v_i)$$
$$= \Sigma_{1 \le i \le k}\, [w(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i)\,]$$
$$= w(P) + h(v_0) - h(v_k)$$

- Hence, (RC2) is also satisfied.

Moreover, for any cycle C, $w'(C) = w(C)$

If the input has a negative cycle then …

Johnson's algorithm (例 1/2)



Johnson's algorithm (例 2/2)