Graph Algorithms - III

CSIE 2136 Algorithm Design and Analysis, Fall 2018

https://cool.ntu.edu.tw/courses/61

Hsu-Chun Hsiao

National Taiwan University

Agenda

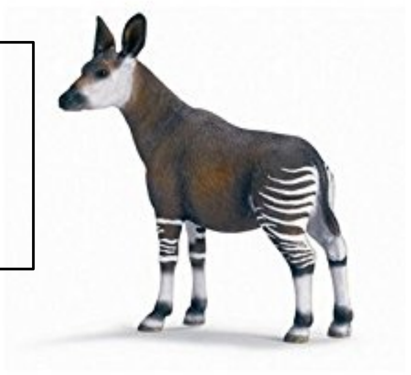
Single-source shortest-path algorithms

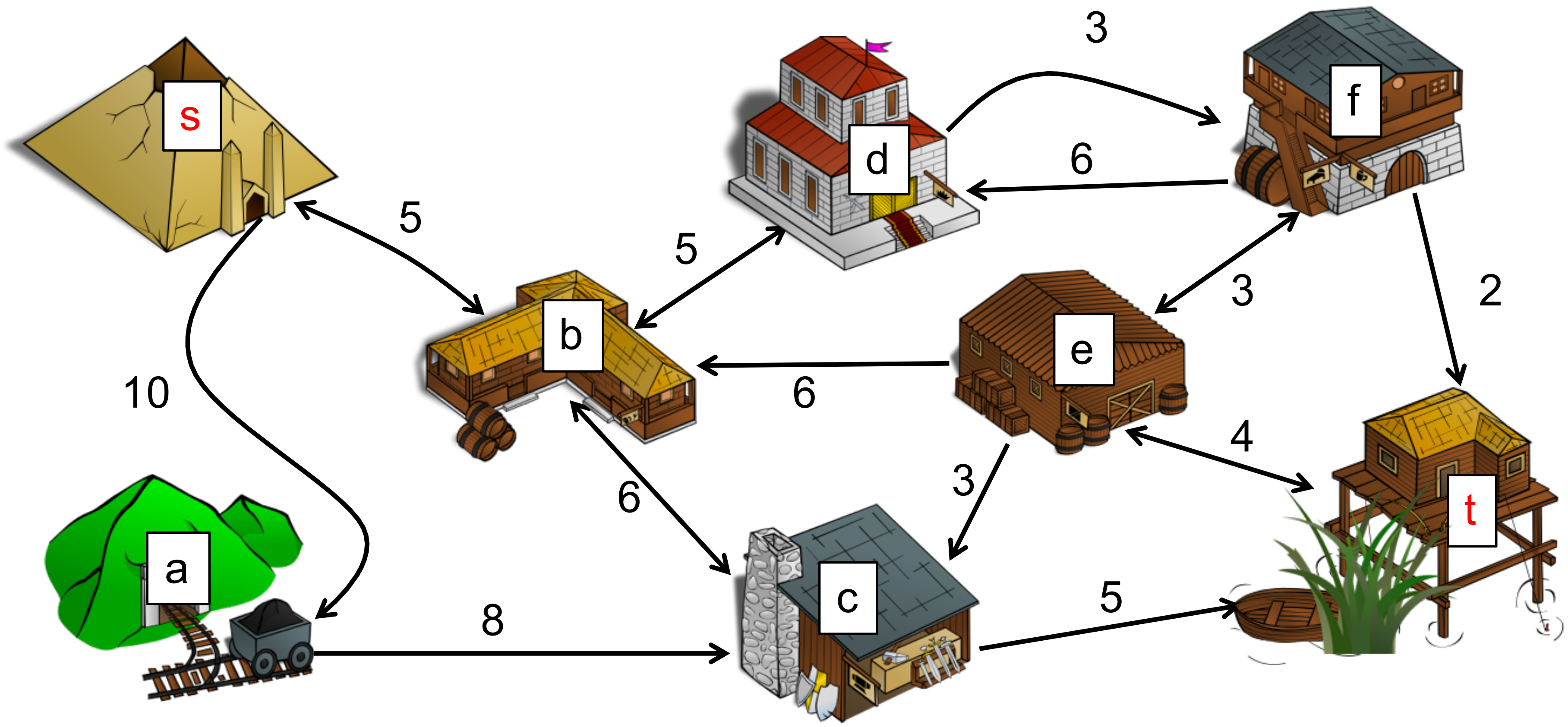
- Dijkstra algorithm
- Bellman-Ford algorithm
- SSSP in DAG

Single-source shortest-path algorithms

Textbook Chapter 24

訓練師在探索神獸級的神奇寶貝時被Okapi 咬傷，每走一公尺就會損血一滴。請找出從金字塔 (s) 到荒野大夫家 (t) 的最短路徑？





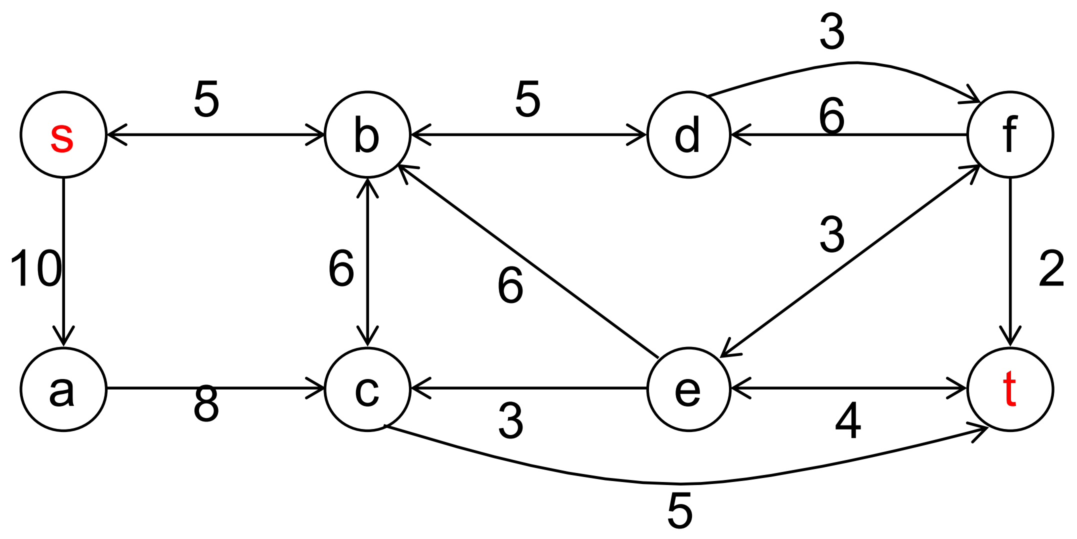
Definitions

Given a weighted, directed graph G = (V, E)

Given a weight function *w* mapping an edge to a weight

- Note that weights are arbitrary numbers, not necessarily distances
- Weight function needs not satisfy triangle inequality (think about airline fares)

Weight of path p = w(p) = sum of weights of edges on p

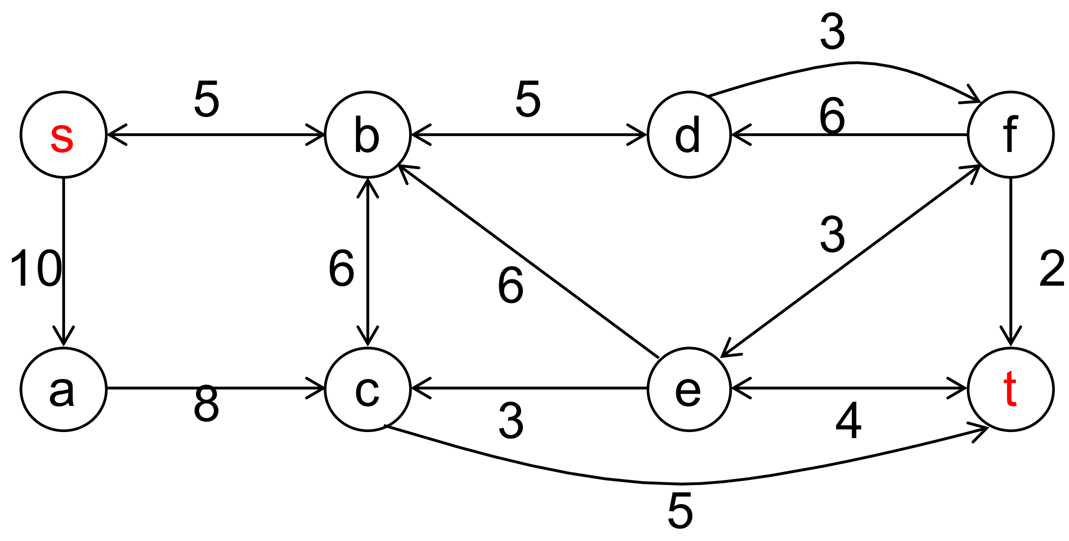


What is the weight of path s->a->c->t?

Definitions

Shortest path weight *δ(s,t)* = minimum weight of path from s to t

A **shortest path** from s to t = any path with weight *δ(s,t)*



δ(s,t) = ?

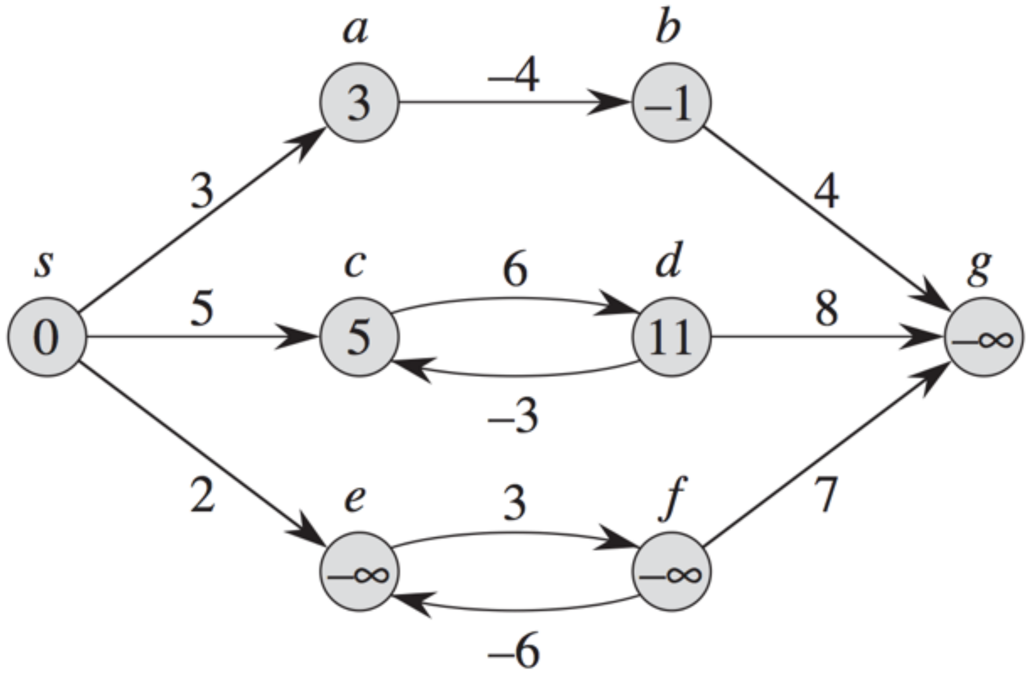
Shortest path from s to t = ?

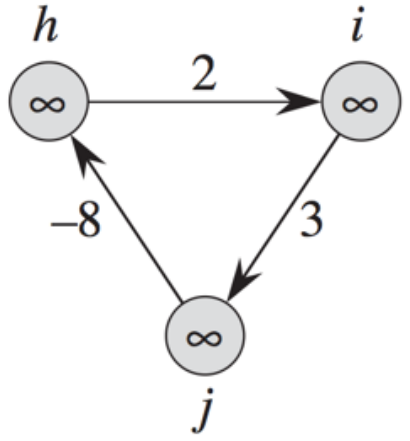
Cycles

Can a shortest path contain a **negative-weight edge**?

Can a shortest path contain a **negative-weight cycle**?

Can a shortest path contain a **cycle**?





Cycles

Can a shortest path contain negative-weight edges?
YES

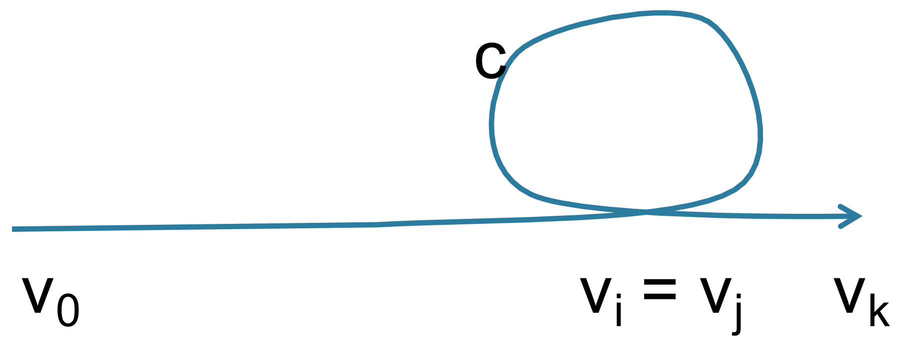
- δ(s,v) remains well defined for all v, if G contains no negative-weight cycles reachable from the source s

Can a shortest path contain negative-weight cycles?
Doesn't make sense

- If there is a negative-weight cycle on some path from s to v, we define δ(s,v) = -∞

Cycles

Can a shortest path contain a positive-weight cycle?



$p = \langle v_0, v_1, \ldots, v_k \rangle$

$c = \langle v_i, v_{i+1}, \ldots, v_j \rangle$

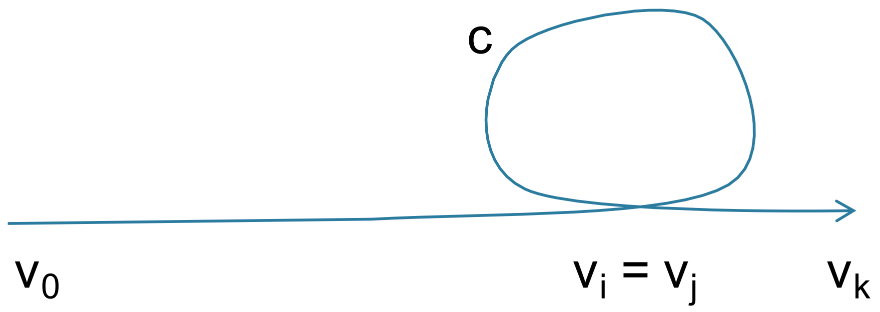
Let $p' = \langle v_0, v_1, \ldots v_i, v_{j+1}, v_{j+2}, \ldots, v_k \rangle$

$w(p') \leq w(p)$ if $w(c) \geq 0$

Cycles

Can a shortest path contain a positive-weight cycle?
NO

- It may contain a zero-weight cycle, but then there must exist a simple path of the same weight.



$p = \langle v_0, v_1, \ldots, v_k \rangle$
$c = \langle v_i, v_{i+1}, \ldots, v_j \rangle$

Let $p' = \langle v_0, v_1, \ldots v_i, v_{j+1}, v_{j+2}, \ldots, v_k \rangle$
$w(p') \leq w(p)$ if $w(c) \geq 0$

Cycles

Hence, we safely assume shortest paths have no cycles

- Define δ(u,v) = ∞ if v is unreachable from u
- Define δ(u,v) = -∞ if there exists a negative cycle on a path from u to v
- No cycle ➔ a shortest path has at most |V| - 1 edges

Variants of shortest-path problems

Our focus of this lecture; it can be applied to solve the following variants

Single-source shortest-path problem: Given a graph G = (V, E) and a *source* vertex s in V, find the minimum cost paths from s to every vertex in V

Single-destination shortest-path problem: Given a graph G = (V, E) and a *destination* vertex t in V, find the minimum cost paths to t from every vertex in V

Single-pair shortest-path problem: Find a shortest path from s to t for *given s and t*

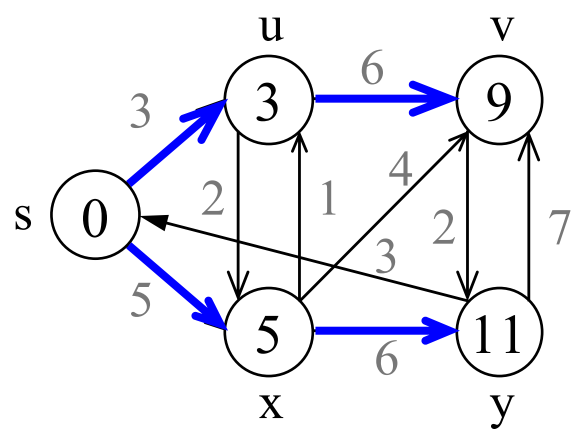
All-pair shortest path problem: Find a shortest path from s to t for *every pair of s and t*

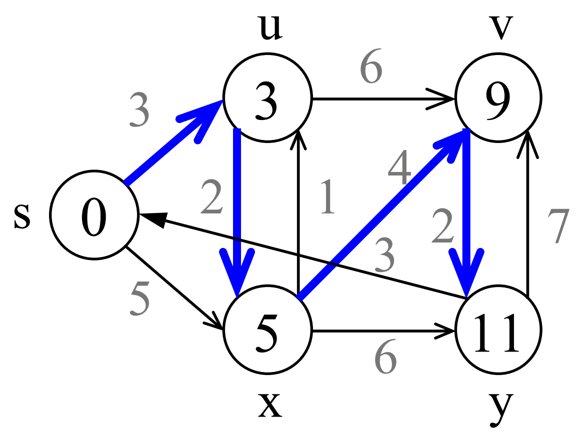
Shortest-paths tree

Let G = (V, E) be a weighted, directed graph with no negative-weight cycles reachable from *s*

A shortest-paths tree G’ = (V’, E’) of *s* is a subgraph of G s.t.:

- V’ is the set of vertices reachable from s in G
- G’ forms a rooted tree with root s
- For all v in V’, the unique simple path from s to v in G’ is a shortest path from s to v in G



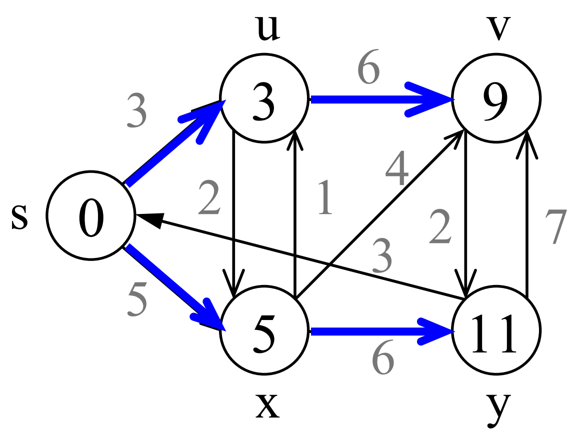


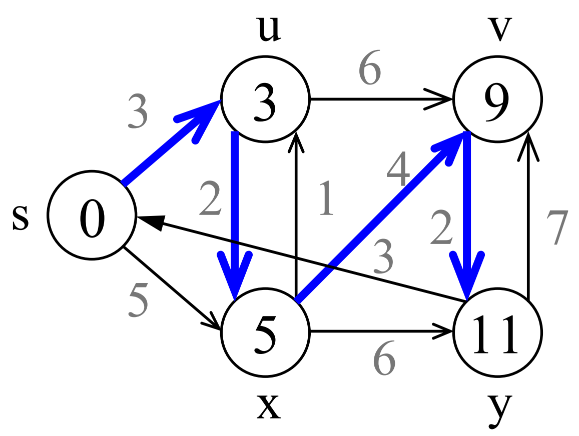
Shortest-paths tree

A shortest-paths tree G' = (V', E') is a subgraph of G s.t.:

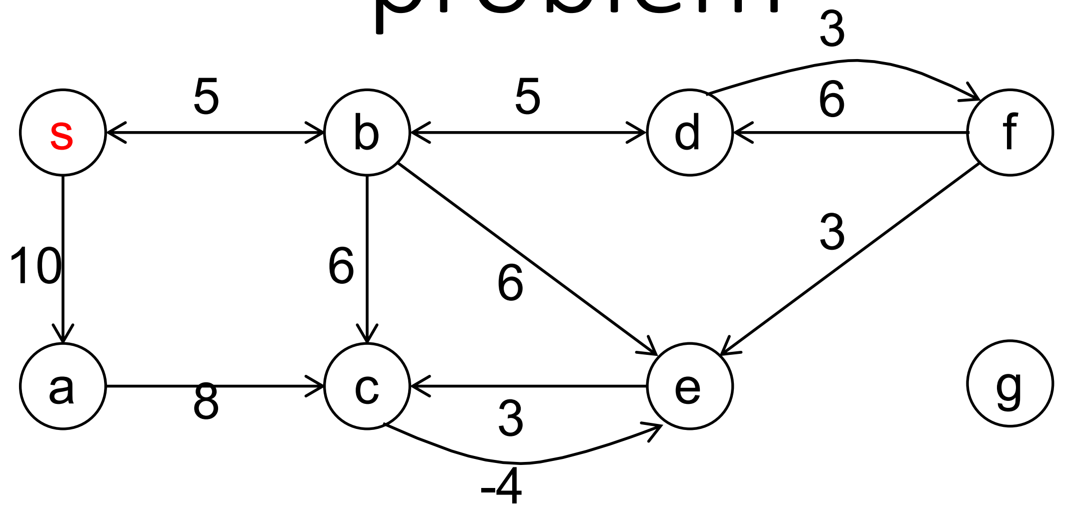
- V' is the set of vertices reachable from s in G
- G' forms a rooted tree with root s
- For all v in V', the unique simple path from s to v in G' is a shortest path from s to v in G

Q: Given G, can there be >1 possible shortest-paths tree? Yes
Q: Given G, can there be >1 possible shortest path weight for a vertex? No
Q: Given G, can we always find one shortest-paths *tree*? Yes. Lemma 24.17





Single-source shortest-path problem

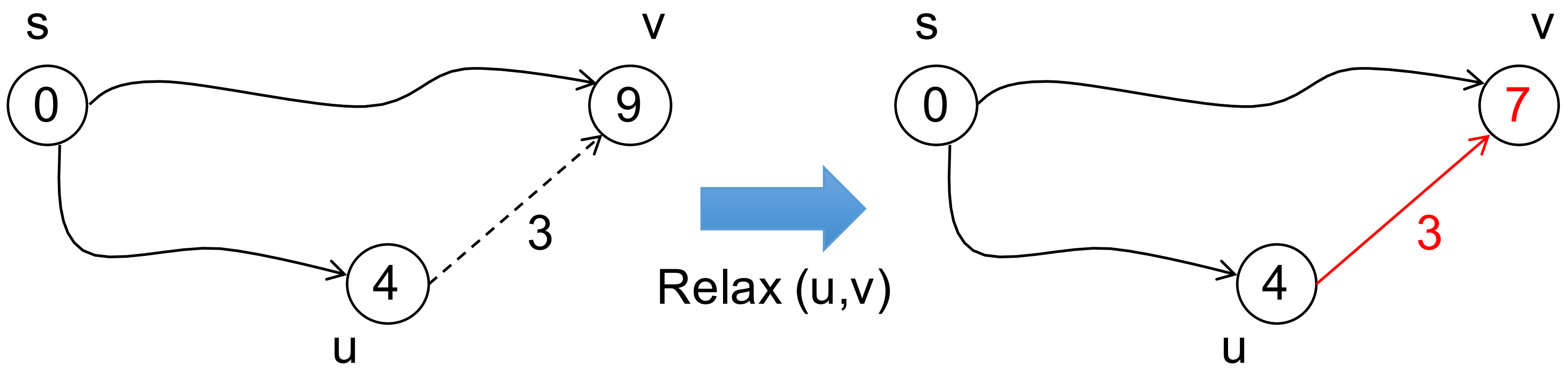


Destination v	Shortest path from s to v	Shortest path weight
a	s a	10
b	s b	5
c	?	
d	s b d	10
e	NIL	-∞
f	?	
g	?	

Edge relaxation

The process of **relaxing** an edge (u,v)
= testing whether the **shortest path weight of v** found so far can be reduced by traveling over u

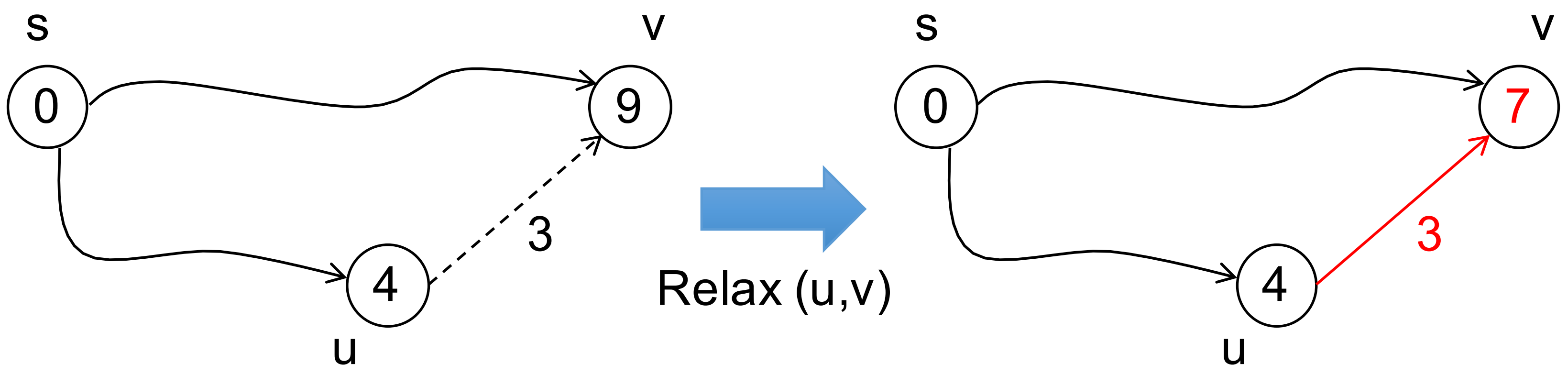
試試看經過u會不會比較好（更短的s-v路徑）



Practice: What if w(u,v) = 10?

Edge relaxation

The process of **relaxing** an edge (u,v) = testing whether the **shortest path weight of v** found so far can be reduced by traveling over u



```
RELAX(u, v, w)
If v.d > u.d + w(u, v)
    v.d = u.d + w(u, v)
    v.π = u
```

v.d = shortest-path estimate

- An **upper bound** on δ(s,v)
- v.d never increases during relaxation

v.π = predecessor attribute

Two classic single source shortest path algorithms

Dijkstra algorithm

- Greedy
- Requiring that all edge weights are **nonnegative**

Bellman-Ford algorithm

- Dynamic programming
- General case, edge weights **may be negative**

- Both on a weighted, directed graph

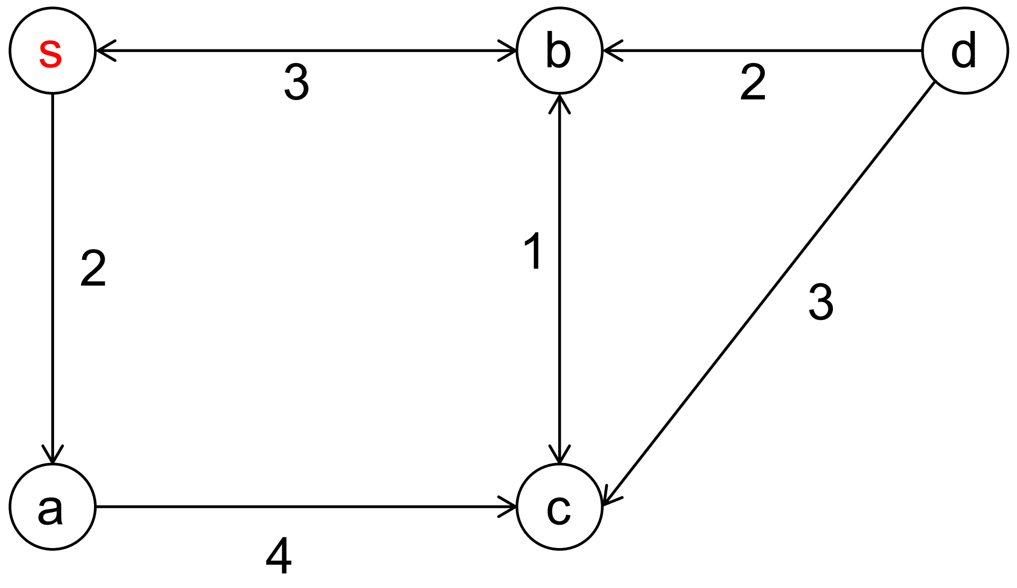
Dijkstra’s algorithm

Textbook Chapter 24.3

Dijkstra's algorithm: intuition

Recall that BFS finds shortest paths on an unweighted graph by expanding the search frontier like ripples.

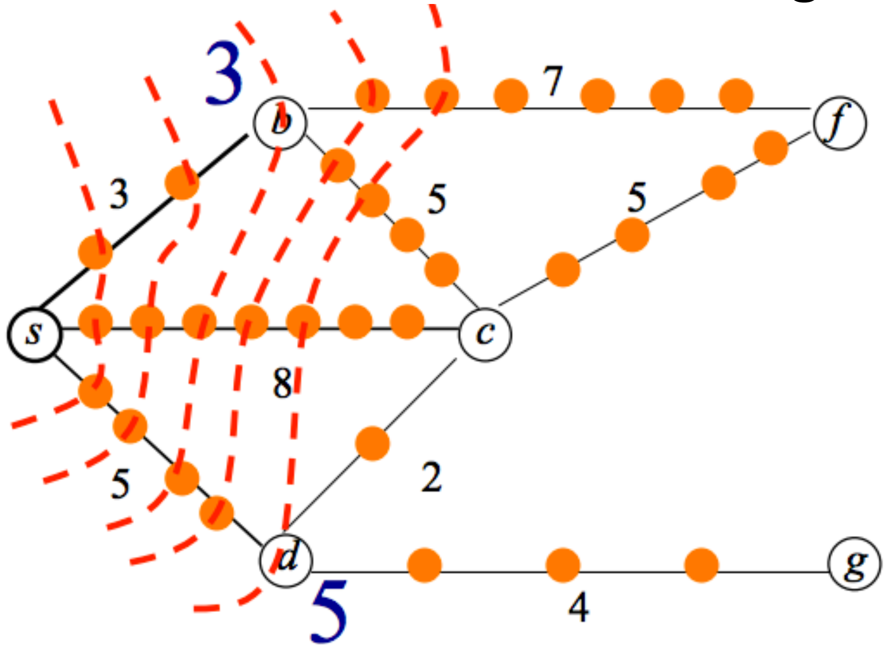
Can we do the same on weighted graph?



Dijkstra's algorithm: intuition

Recall that BFS finds shortest paths on unweighted graph by expanding the search frontier like ripples

Can we do the same on weighted graph?



Dijkstra's algorithm speeds up the process by “skipping” layers that do not intersect with any vertex!

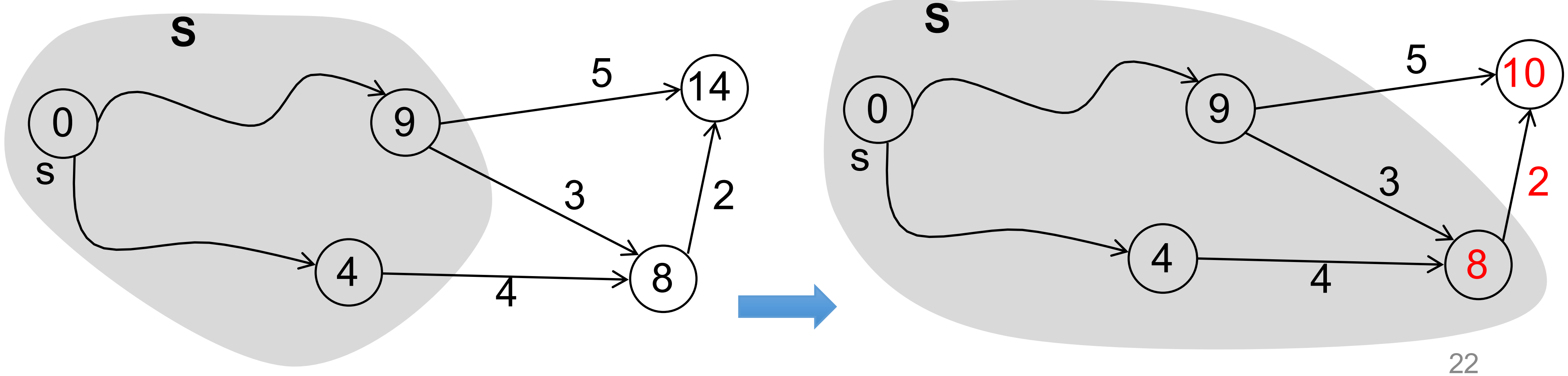
Source: http://courses.csail.mit.edu/6.006/spring11/lectures/lec16.pdf

Dijkstra's algorithm

Dijkstra greedily adds vertices by increasing distance

Maintains a **set of explored vertices** ***S*** whose final shortest-path weights have already been determined

1. Initially, S = {s}, s.d = 0
2. At each step, select unexplored vertex *u* in *V* - *S* with **minimum *u.d***
3. Add *u* to *S*, and **relaxes all edges leaving *u*.** Go back to Step 2.



Implementation of Dijkstra's algorithm

```
DIJKSTRA(G,w,s)
INITIALIZE-SINGLE-SOURCE(G,s)
S= empty
Q= G.v //INSERT
while Q ≠ empty
    u = EXTRACT-MIN(Q)
    S = S∪{u}
    for v in G.adj[u]
        RELAX(u,v,w)
```

```
INITIALIZE-SINGLE-SOURCE(G,s)
for v in G.V
    v.d = ∞
    v.π = NIL
s.d = 0
```

```
RELAX(u, v, w)
if v.d > u.d + w(u, v)
    //DECREASE-KEY
    v.d = u.d + w(u, v)
    v.π = u
```

For u in Q = V-S, u.d is the **shortest-path estimate** (i.e., minimum length over all observed s-u paths so far)

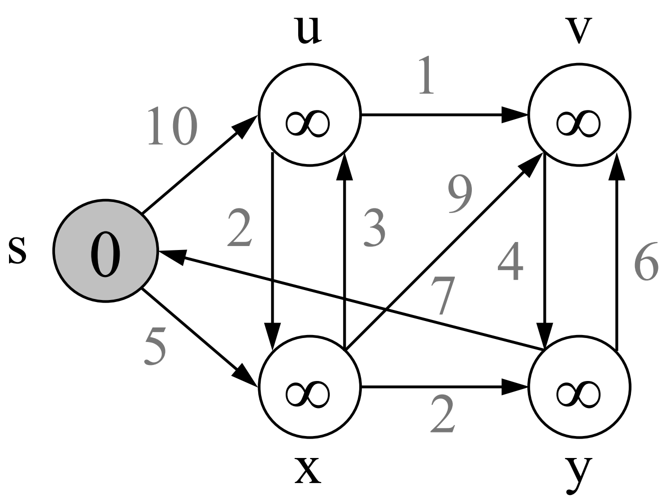
For u in S, u.d = length of the shortest path from s to u

Q is a min-priority queue of vertices, keyed by *d* values

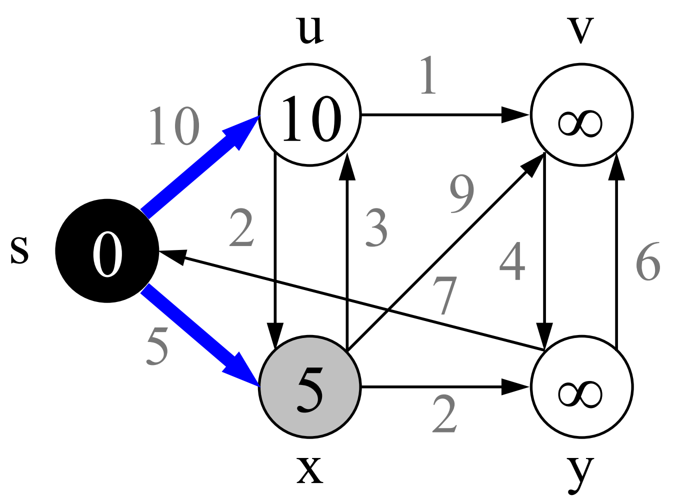
```
DIJKSTRA(G,w,s)
INITIALIZE-SINGLE-SOURCE(G,s)
S=empty
Q= G.v //INSERT
while Q ≠ empty
    u = EXTRACT-MIN(Q)
    S = S∪{u}
    for v in G.adj[u]
        RELAX(u,v,w)
```

Black: in S
White: in Q
Grey: selected
Blue lines: predecessors

Step 0



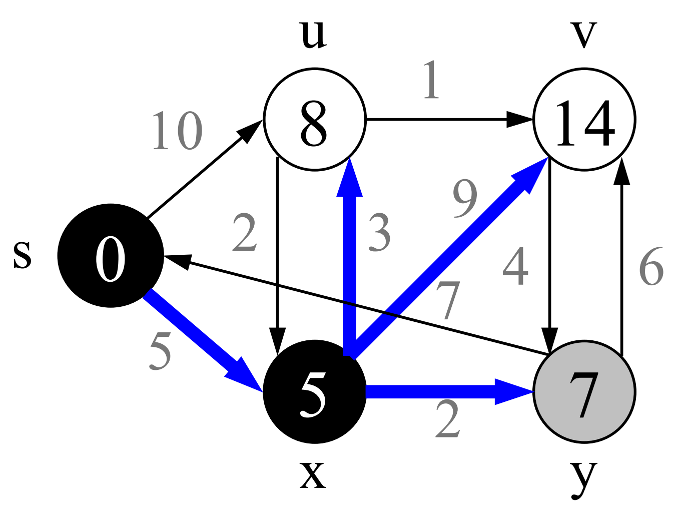
Step 1



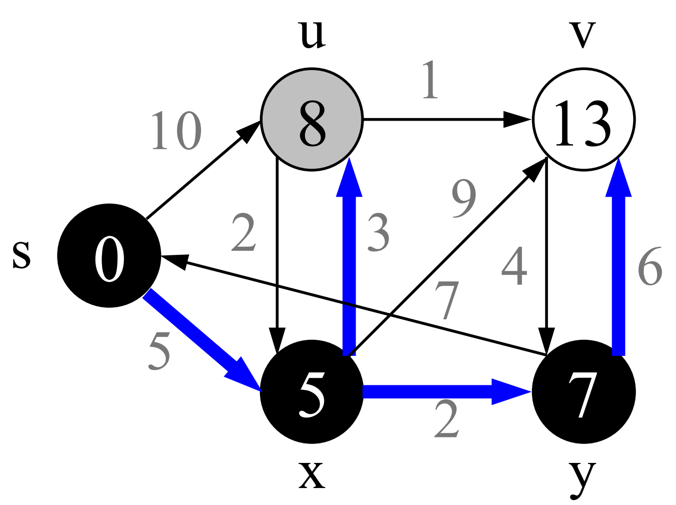
```
DIJKSTRA(G,w,s)
INITIALIZE-SINGLE-SOURCE(G,s)
S=empty
Q= G.v //INSERT
while Q ≠ empty
    u = EXTRACT-MIN(Q)
    S = S∪{u}
    for v in G.adj[u]
        RELAX(u,v,w)
```

Black: in S
White: in Q
Grey: selected
Blue lines: predecessors

Step 2



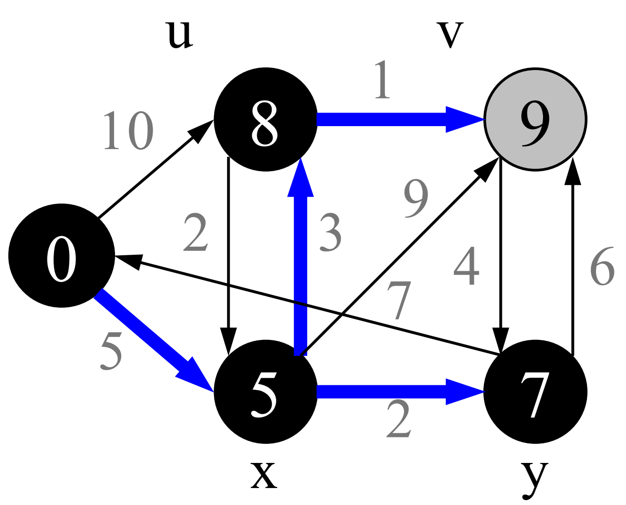
Step 3



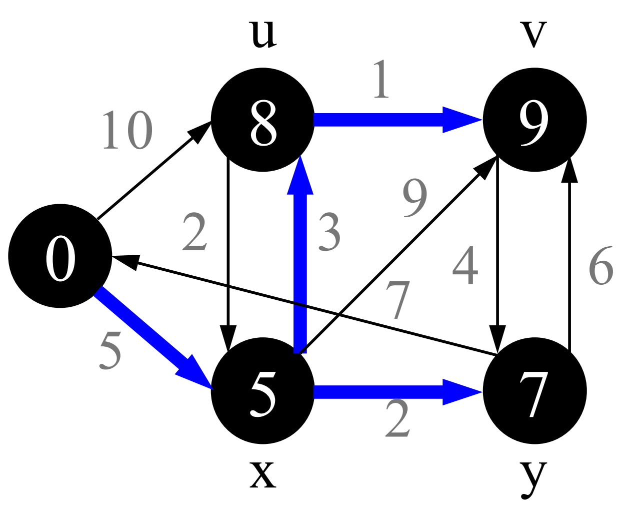
```
DIJKSTRA(G,w,s)
INITIALIZE-SINGLE-SOURCE(G,s)
S=empty
Q= G.v //INSERT
while Q ≠ empty
    u = EXTRACT-MIN(Q)
    S = S∪{u}
    for v in G.adj[u]
        RELAX(u,v,w)
```

Black: in S
White: in Q
Grey: selected
Blue lines: predecessors

Step 4



Step 5



Running time analysis

Q is a min-priority queue of vertices, keyed by *d* values

- # of INSERT = O(V)
- # of EXTRACT-MIN = O(V)
- # of DECREASE-KEY = O(E)

The running time depends on queue implementation

Implementing the min-priority queue using an array indexed by *v*: $O(V^2+E) = O(V^2)$

- INSERT: O(1)
- EXTRACT-MIN: O(V)
- DECREASE-KEY: O(1)

Proof of correctness

S: the set of explored vertices whose final shortest-path weights have already been determined

- Initially, *S = {s}*, *s.d* = 0
- **Invariant:** for all *u* in S, *u.d* = length of the shortest path from s to u
- Note that for *u* in *V-S*, *u.d* = length of *some* path from *s* to *u*

We want to prove that the loop invariant holds throughout the execution of the algorithm.

Loop invariant: for u in S, u.d = δ(s,u)

Proof by induction on the size of S

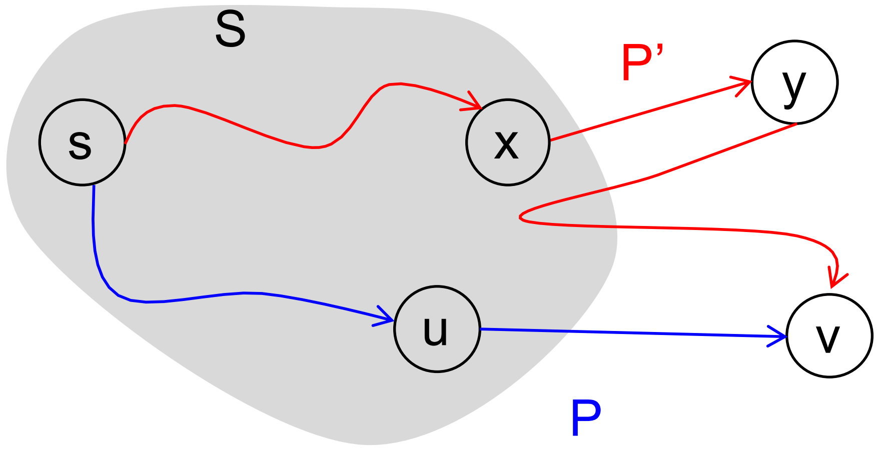
Base case: |S| = 1, correct

Inductive step: Let *v* be the next vertex to be added to S,
u = *v.π*, *P* = shortest path from *s* to *u* + *(u,v)*

=> *v.d* = *w(P)* = ***δ(s,u) + w(u,v)***

Consider any other s-v path P', and Let y be the first vertex on path P' outside S

We want to prove that w(P') ≥ w(P)



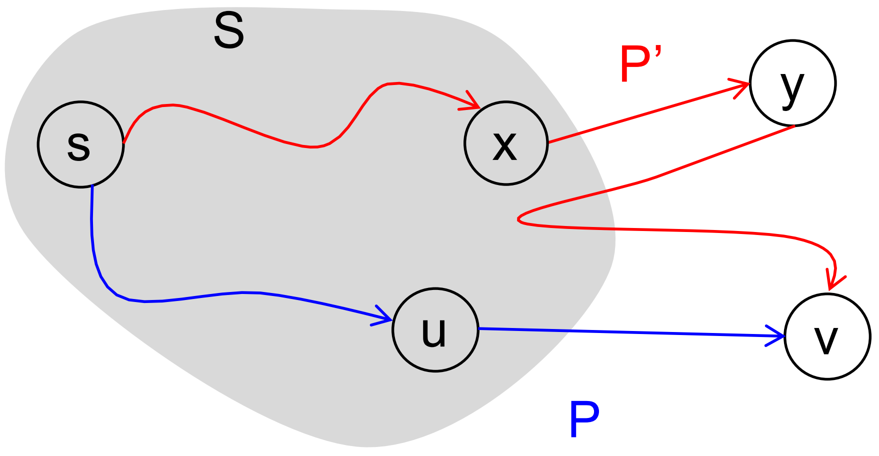
Loop invariant: for u in S, u.d = δ(s,u)

Proof by induction on the size of S (cont'd)

Prove that w(P') ≥ w(P)

1. Because of no negative edges, w(P') ≥ δ(s,x) + w(x,y)
2. By induction hypothesis, δ(s,x) = x.d
3. By construction, y.d ≥ v.d
4. By construction, x.d + w(x,y) ≥ y.d

=> w(P') ≥ δ(s,x) + w(x,y) = x.d + w(x,y) ≥ y.d ≥ v.d = w(P)



Loop invariant: for u in S, u.d = δ(s,u)

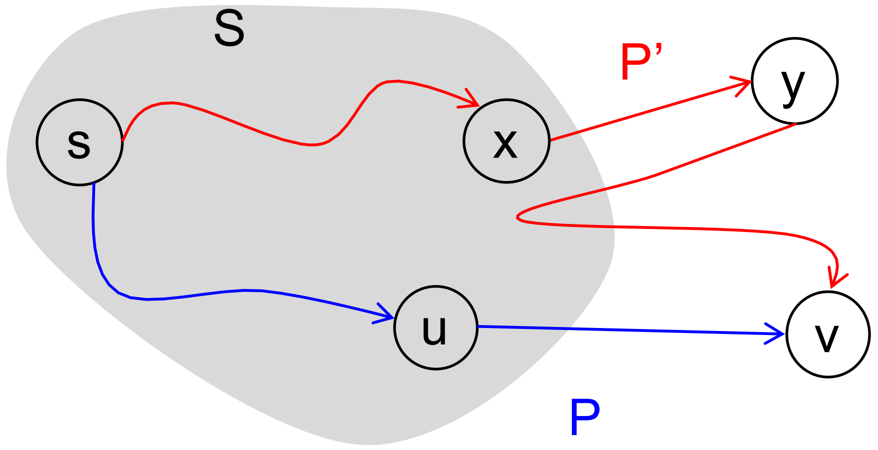
<u>Proof by induction on the size of S (cont'd)</u>

Hence, the greedy choice *v* (and the corresponding path *P* and *u.d*) is at least as good as any other path from *s* to *v*

=> The invariant still holds after adding one more vertex to *S.*

At termination, every vertex is in *S.*

Thus, *u.d = δ(s,v)* for all *u* in *V.*

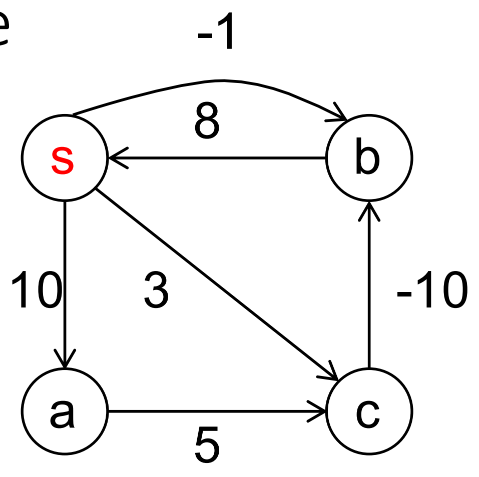


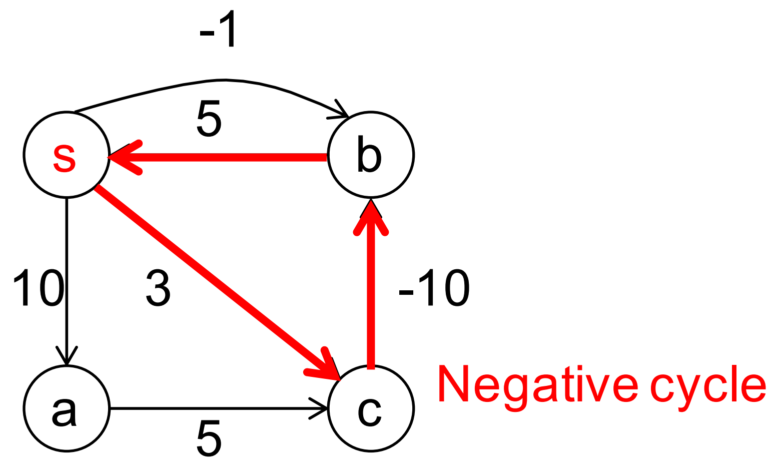
Dijkstra's algorithm may work incorrectly with negative-weight edges.

Let's go back to the proof and see where it breaks!

- This greedy algorithm assumed adding edges always increases path weight, which is not true in case of negative-weight edges

Next topic: a dynamic programming algorithm (Bellman-Ford) either detects negative cycles or returns the shortest-path tree





δ(s,b) = -7
In Dijkstra, *b.d* = -1

δ(s,b) = ??

Priority-first search

See any similarity between BFS, DFS, Prim and Dijkstra?

- They are all greedy algorithms for graph search
- They are each a special case of **priority-first search**

Priority-first search

- Maintain a set of explored vertices *S*
- Grow *S* by exploring **highest-priority edges** with exactly one endpoint leaving S

Priority in each variant

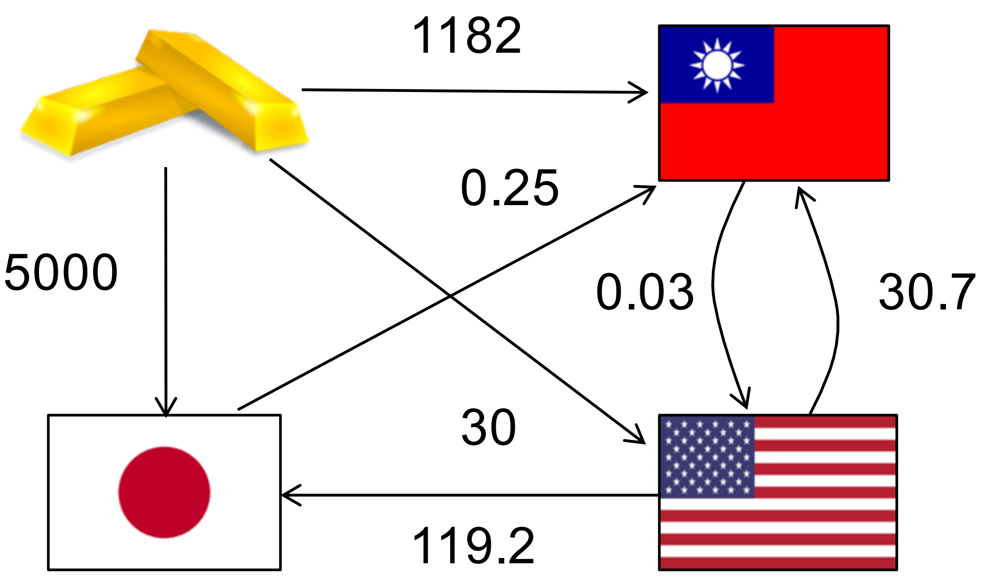
- **BFS:** edges from vertex discovered least recently
- **DFS:** edges from vertex discovered most recently
- **Prim:** edges of minimum weight
- **Dijkstra**: edges to vertex closest to s

Bellman-Ford algorithm

Textbook Chapter 24.1

匯率換算

1公克黃金最多可以換到多少TWD？（假設零手續費）



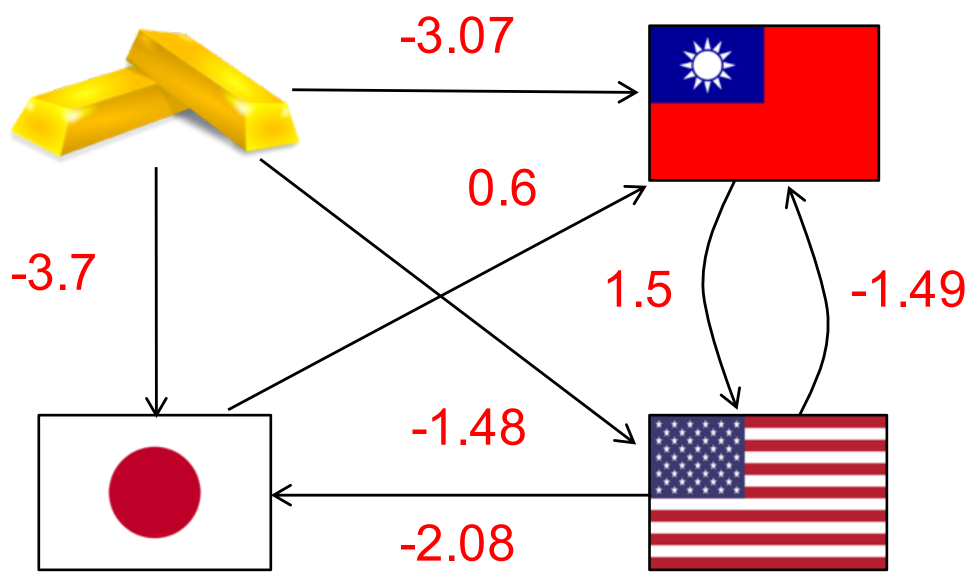
找weight相乘後最大路徑？

如何轉成我們熟悉的最短路徑問題？

匯率換算

1公克黃金最多可以換到多少TWD？（假設零手續費)

Reweighting: w’(e) = - log w(e)

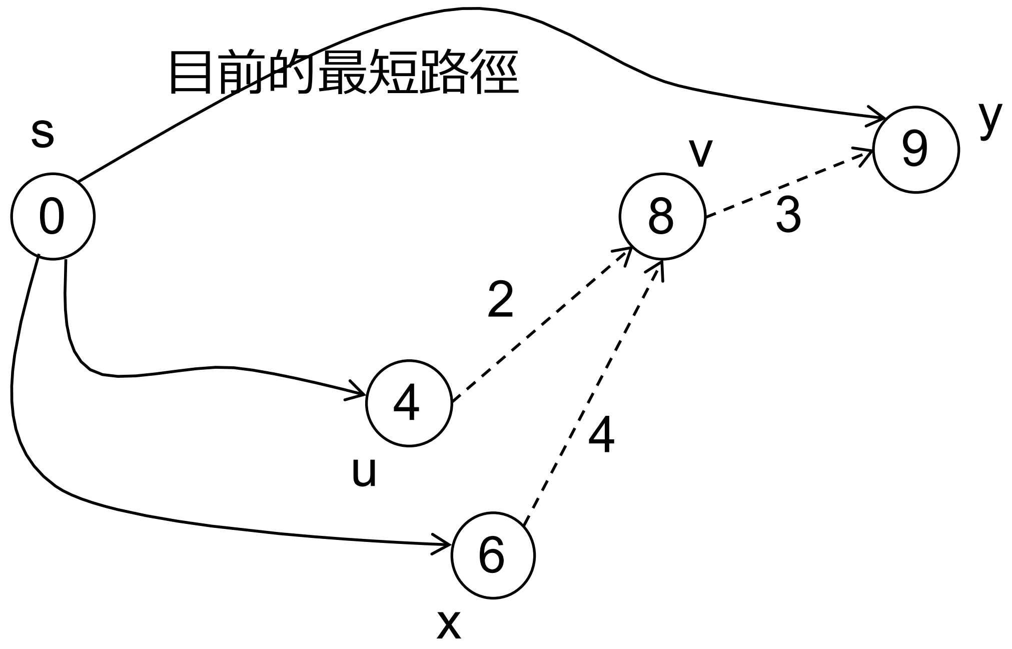


We want to detect the existence of negative cycles (利用匯差賺錢) and find the shortest path (最佳的兌換率)

Bellman-Ford algorithm: intuition

共執行|V|-1回合

- 每一回合中, relax every edge
- 節點v一方面接收從各個“上游”來的最短路徑估計值，試試看改走不同上游會不會比較好，另一方面把自己的估計值傳給“下游”節點



u 和 x 為v的上游
y為v的下游

以v的觀點來看，
relax (u,v), (x,v), (v,y)
順序不重要

Bellman-Ford algorithm: intuition

Bellman-Ford保證在第k回合結束後,**節點v的最短路徑估計值 ≤ 所有邊數至多為k的s-v路徑的最短距離**

=> |V|-1回合結束後，節點v的最短路徑估計值≤所有邊數至多為|V|-1的s-v路徑的最短距離

=>若最短路徑存在，由於最短路徑的邊數不會大於|V|-1，因此Bellman-Ford結束後的確能正確算出最短路徑值

Bellman-Ford algorithm

```
BELLMAN-FORD(G,w,s)
INITIALIZE-SINGLE-SOURCE(G,s)
for i = 1 to |G.V|-1
    for (u,v) in G.E
        RELAX(u,v,w)
for (u,v) in G.E
    if v.d > u.d + w(u,v)
        return FALSE
return TRUE
```

```
INITIALIZE-SINGLE-SOURCE(G,s)
for v in G.V
    v.d = ∞
    v.π = NIL
s.d = 0
```

```
RELAX(u, v, w)
If v.d > u.d + w(u, v)
    //DECREASE-KEY
    v.d = u.d + w(u, v)
    v.π = u
```

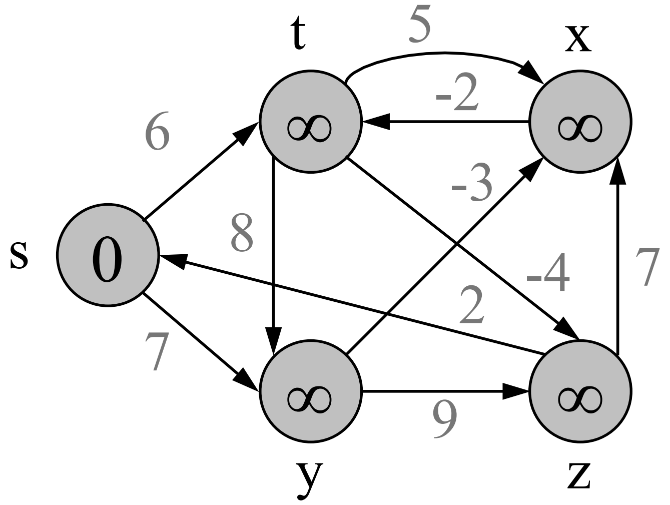
Relax each edge *e*; repeat V-1 times

Detect a negative cycle if any exist

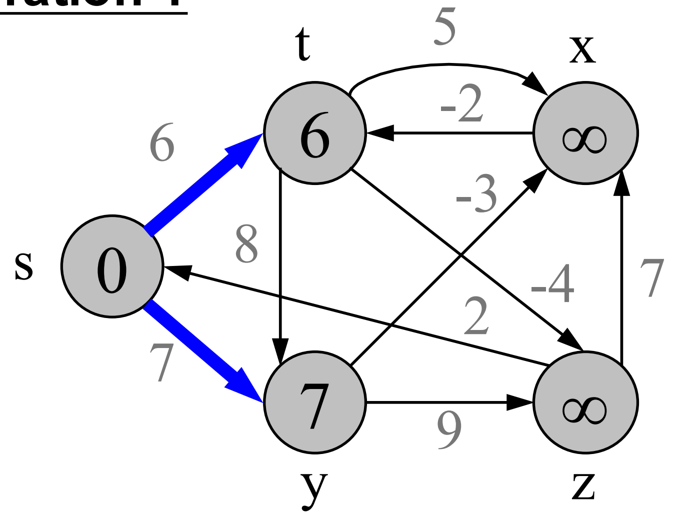
Find shortest simple path if no negative cycle exists

Relaxation sequence in each iteration: (t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)

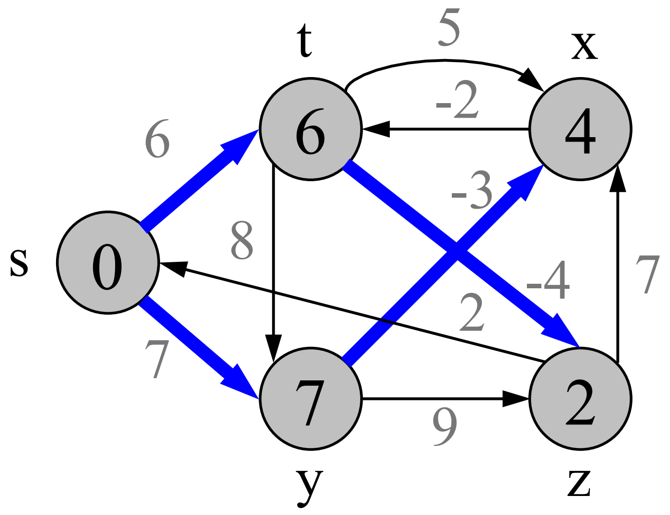
Iteration 0



Iteration 1



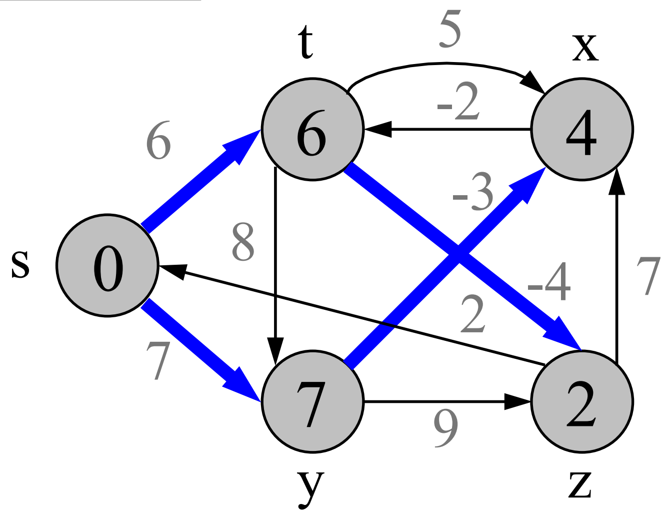
Iteration 2



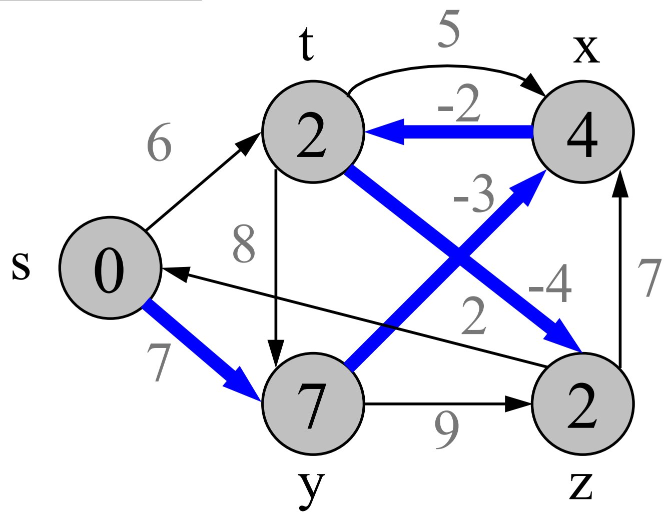
```
BELLMAN-FORD(G,w,s)
INITIALIZE-SINGLE-SOURCE(G,s)
for i = 1 to |G.V|-1
    for (u,v) in G.E
        RELAX(u,v,w)
for (u,v) in G.E
    if v.d > u.d + w(u,v)
        return FALSE
return TRUE
```

Relaxation sequence in each iteration: (t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)

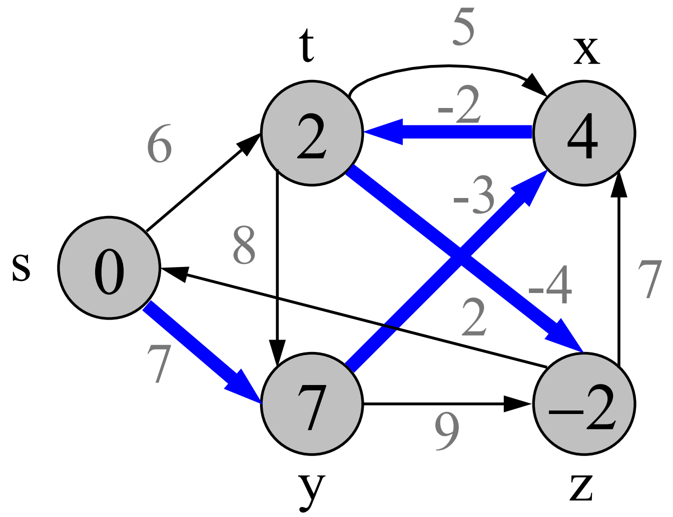
Iteration 2



Iteration 3



Iteration 4



```
BELLMAN-FORD(G,w,s)
INITIALIZE-SINGLE-SOURCE(G,s)
for i = 1 to |G.V|-1
    for (u,v) in G.E
        RELAX(u,v,w)
for (u,v) in G.E
    if v.d > u.d + w(u,v)
        return FALSE
return TRUE
```

Running time analysis

Using adjacency lists,

```
BELLMAN-FORD(G,w,s)
INITIALIZE-SINGLE-SOURCE(G,s)
for i = 1 to |G.V|-1
    for (u,v) in G.E
        RELAX(u,v,w)
for (u,v) in G.E
    if v.d > u.d + w(u,v)
        return FALSE
return TRUE
```

$\Theta(V)$

$\Theta((V\text{-}1)E)$

$\Theta(E)$

Adjacency-list representation = $\Theta(VE)$

Adjacency-matrix representation = ?

- It takes $\Theta(V^2)$ to loop through all edges, thus $\Theta(V^3)$ in total

Correctness of Bellman-Ford

We want to prove the following two statements:

1. Correctly compute δ(s, v) when no negative-weight cycle:
    - After the $|V| - 1$ iterations of relaxation of all edges, it must hold that $v.d = \delta(s, v)$ for all vertices $v \in V$ that are reachable from s
    - For each vertex v in V, there is a path from s to v if and only if the algorithm terminates with $v.d < \infty$
2. Correctly detect the existence of negative cycles
    - Bellman-Ford returns FALSE If G does contain a negative-weight cycle reachable from s

Path-relaxation property

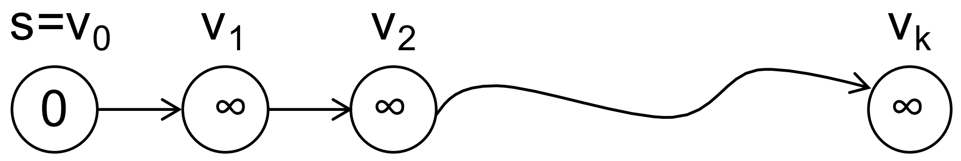
$\boxed{\text{Lemma 24.15}}$

Let $p=\langle v_0,v_1,\ldots,v_k\rangle$ be a shortest path from $s=v_0$ to v_k

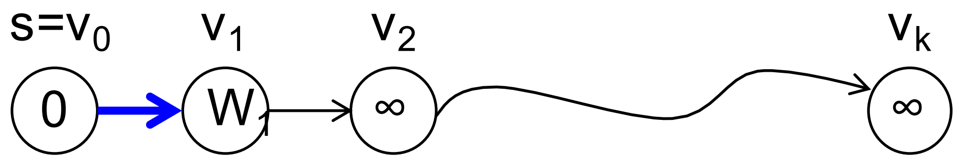
Let $W_i = \Sigma_1^i\, w(v_{i-1}, v_i)$, W_i be the shortest path weight $\delta(s,v_i)$ because of optimal substructure

Path-relaxation property: $v_k.d = \delta(s,v_k)$ after relaxation sequence $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$

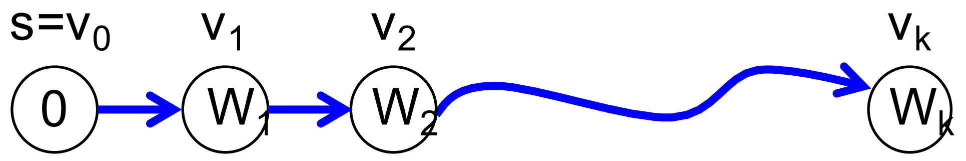
Init



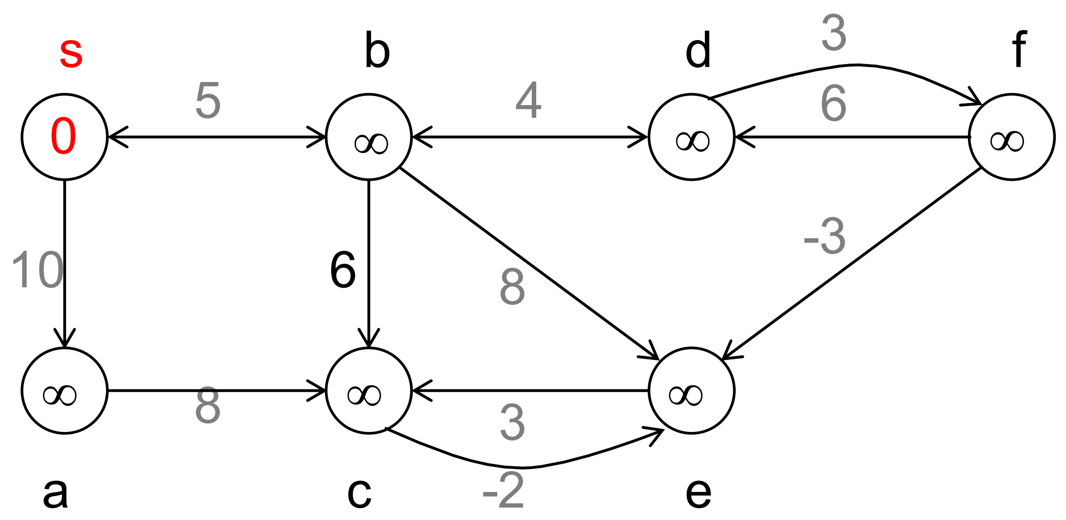
Relax (v_0, v_1)



After relaxation sequence $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$



Note: 此性質對於任何包含這個最短路徑邊的relaxation sequence都成立, e.g., (v_0, v_1), (a, b), (d, c), (v_0, v_1), $(v_1, v_2), \ldots, (v_{k-1}, v_k)$



δ(s,e) = 9

A shortest path from s to e = <s, b, d, f, e>

After relaxing (s, b), (b, d), (d, f), (f, e) in order, what would be the value of e.d?

How about relaxing them in a different order, such as (s, b), (d, f), (b, d), (f, e)?

How about relaxing (s, b), (b, e), (s, a), (b, d), (d, f), (e, c), (f, e)?

Correctness: compute shortest paths

Path-relaxation property: If $p=<v_0,v_1,\ldots,v_k>$ is a shortest path from $s=v_0$ to v_k, then $v_k.d = \delta(s,v_k)$ after relaxation sequence $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$

The relaxation sequence in Bellman Ford must contain all edges in p in order:

- Although the shortest path p from s to v is unknown, we know it has at most V-1 edges if the path exists (i.e., no negative-weight cycles)
- The relaxation sequence must contain all edges in p in order:

(m=|E|)

$$\underbrace{\underbrace{e_1, e_2, \ldots, e_m}_{\text{Must contain 1st edge in p}}; \underbrace{e_1, e_2, \ldots, e_m; \ldots}_{\text{Must contain 2nd edge in p}}\ldots; e_1, e_2, \ldots, e_m}_{\text{Repeated V-1 times, must contain all edges in p in order}}$$

This relaxation sequence contains a shortest path to every vertex!

Correctness: negative cycle detection

Claim: Bellman-Ford returns FALSE If G does contain a negative-weight cycle reachable from s

Proof by contradiction

Suppose Bellman-Ford returns TRUE while G does contain a negative-weight cycle C reachable from s

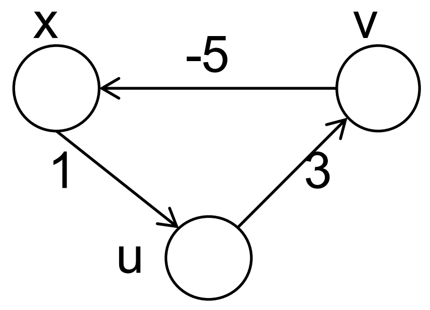
=> $v.d \leq u.d + w(u,v)$ for all (u, v) in C

=> $\Sigma_{v \text{ in } C}\, v.d \leq \Sigma_{u \text{ in } C}\, u.d + \Sigma_{(u,v) \text{ in } C}\, w(u,v)$

=> $0 \leq \Sigma_{(u,v) \text{ in } C}\, w(u,v)$

=> contradiction

Practice: show contradiction using this example

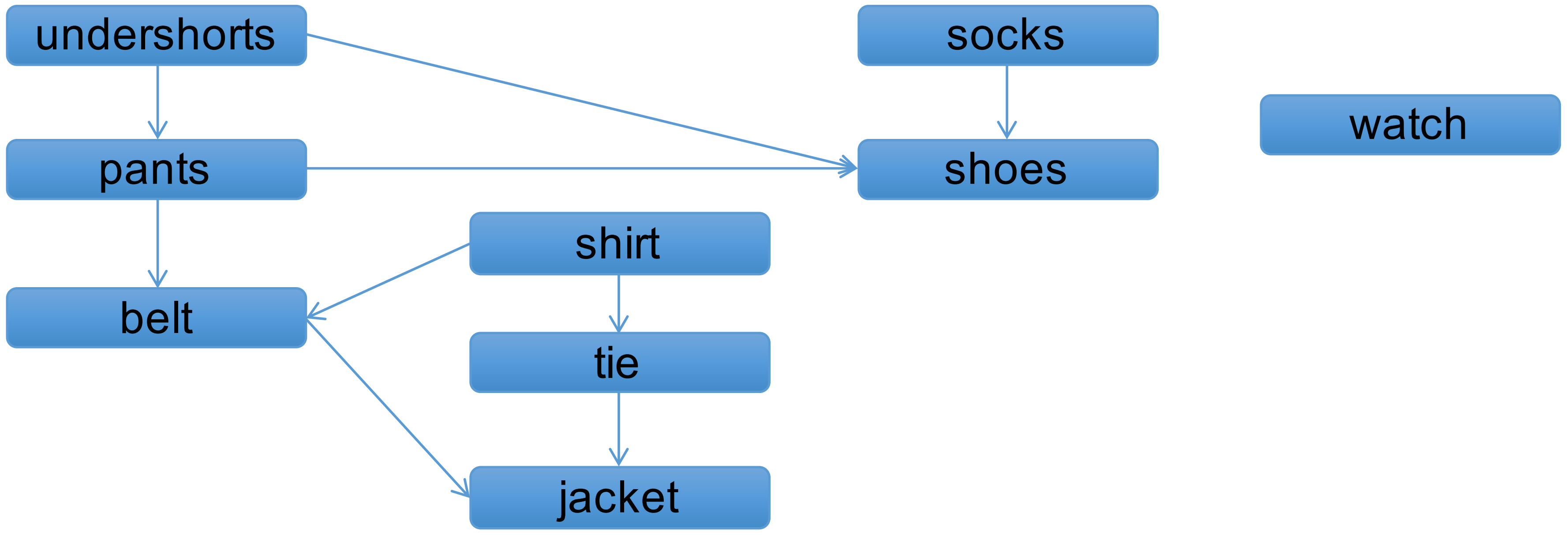


Single-source shortest paths in directed acyclic graphs

Textbook Chapter 24.2

Recall: Directed Acyclic Graphs (DAGs)

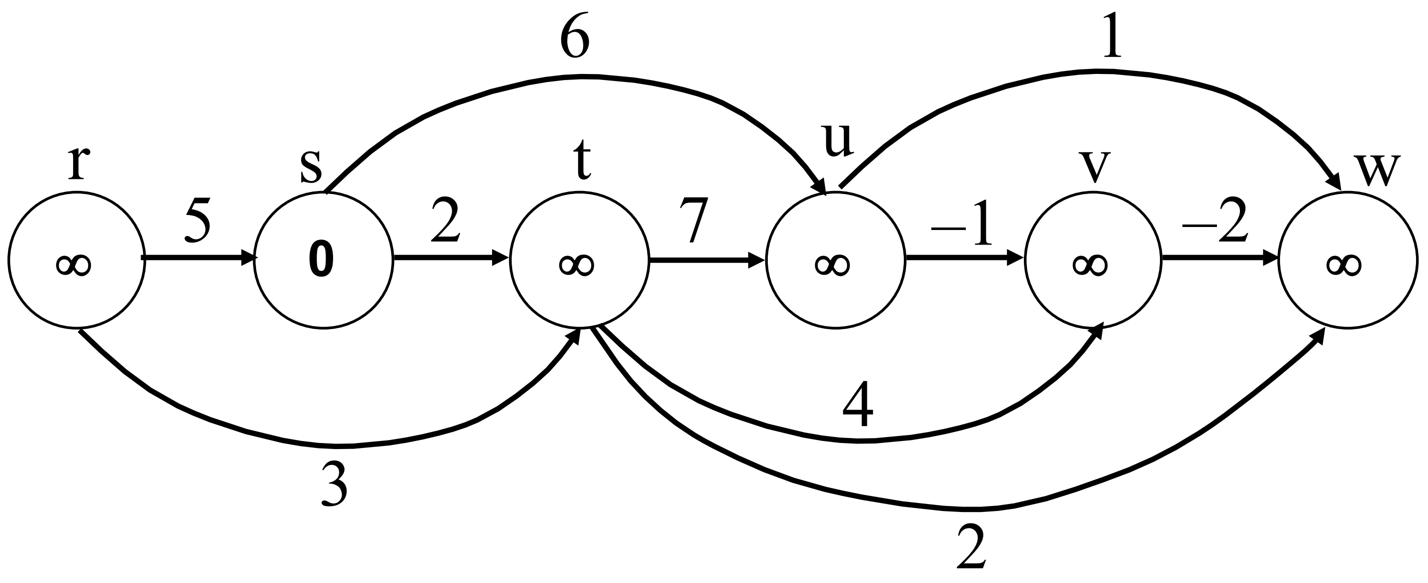
- A DAG is a directed graph with no cycles
- Often used to indicate precedence among events (X must happen before Y)
    - E.g., cooking, taking courses, clothing…



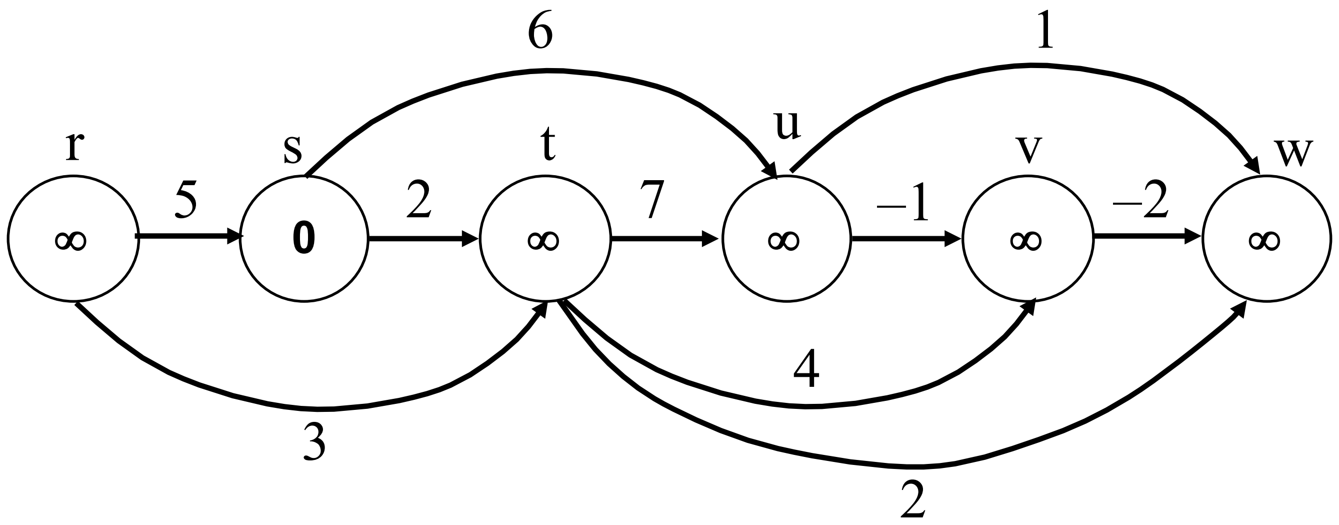
Single-source shortest paths in DAG

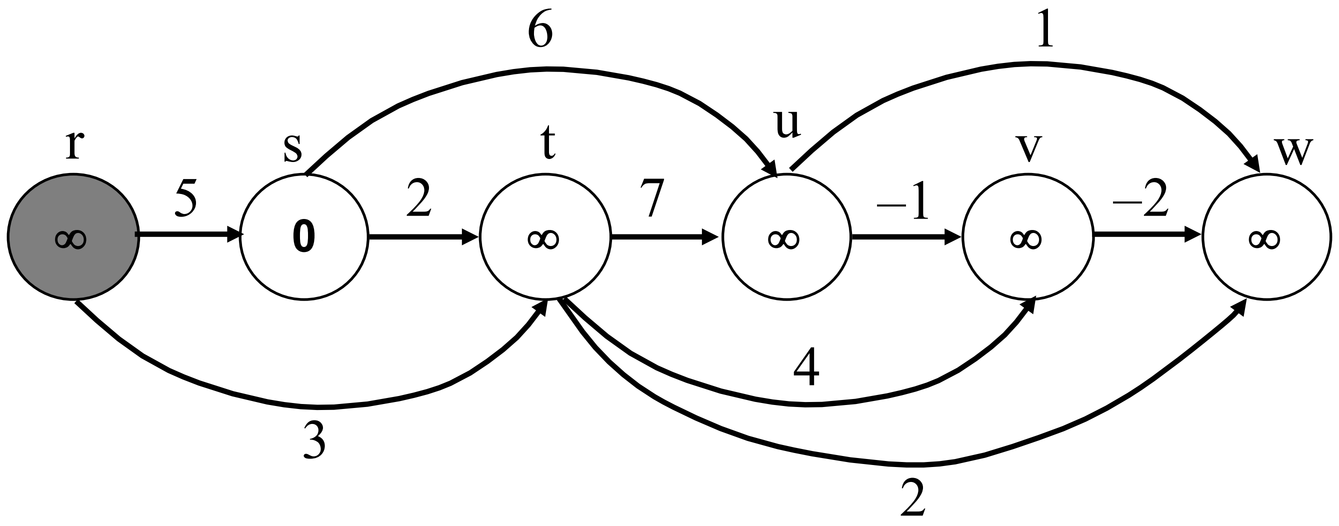
Claim: relaxing the edges in **topologically sorted order** correctly computes the shortest paths

Reason: putting vertices in a topologically sorted order, edges only go from left to right

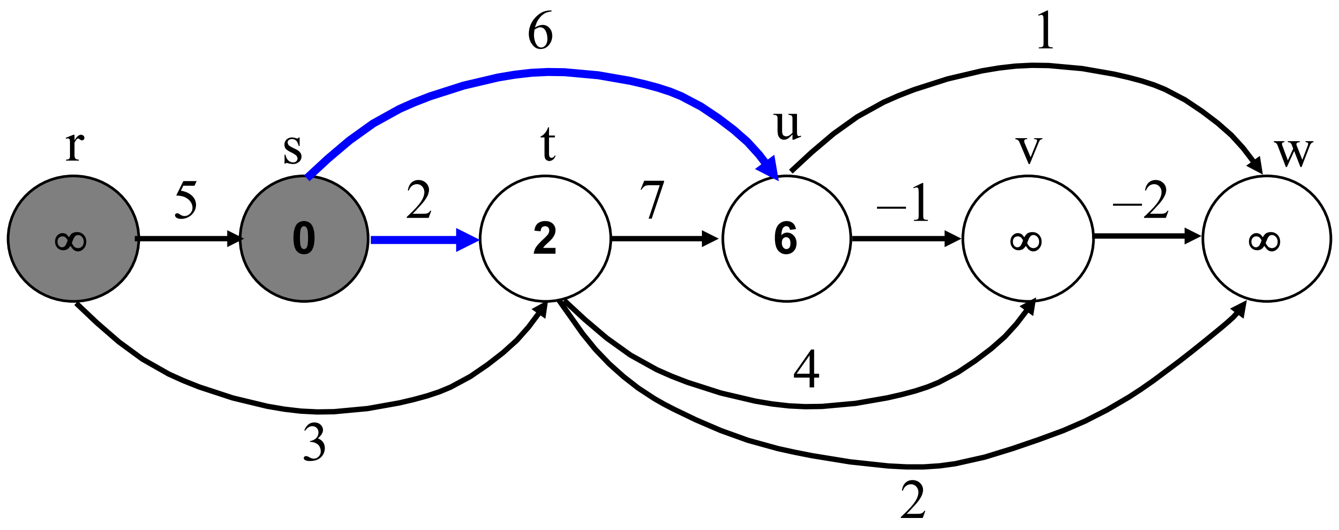


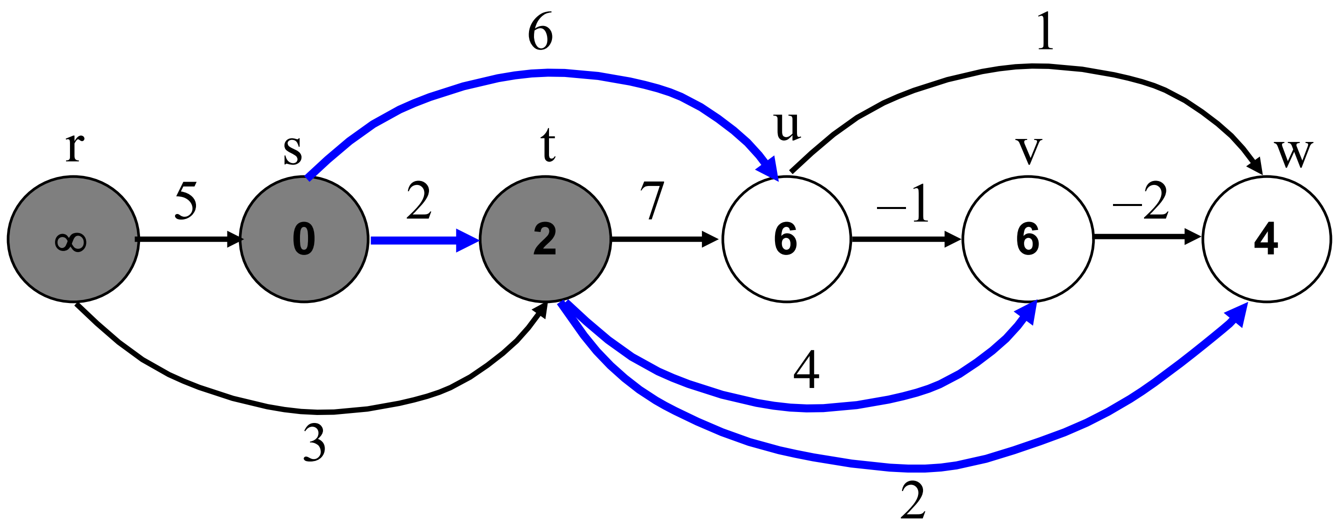
```
DAG-SHORTEST-PATHS(G,w,s)
topologically sort the vertices of G
INITIALIZE-SINGLE-SOURCE(G,s)
for each vertex u, taken in topologically sorted order
    for each vertex v in G.adj[u]
        RELAX(u,v,w)
```



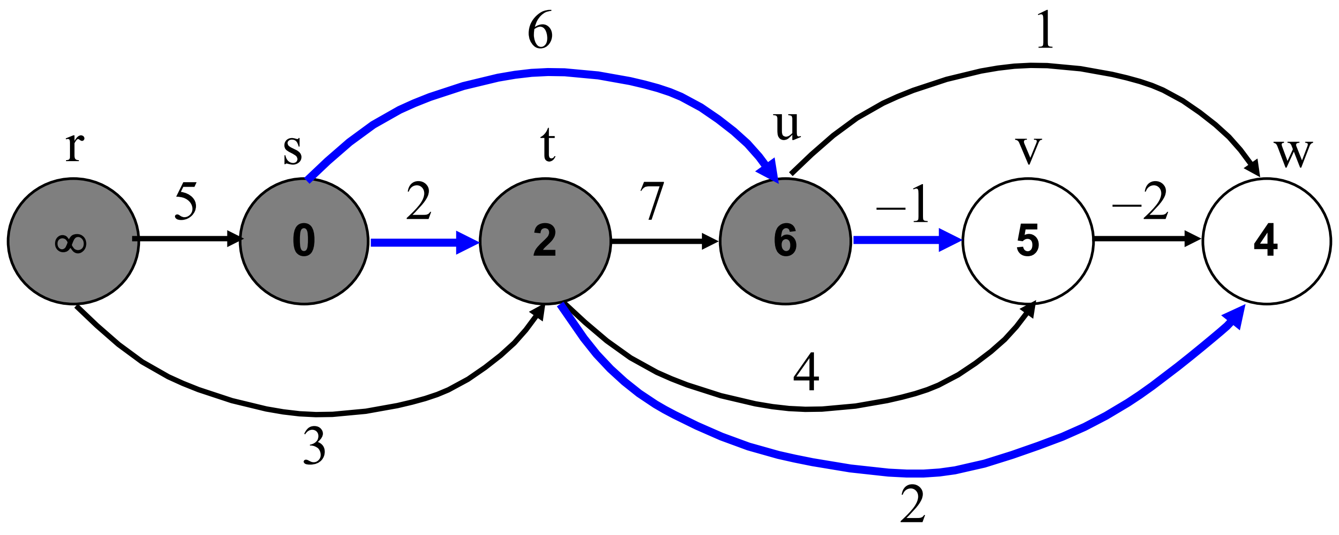


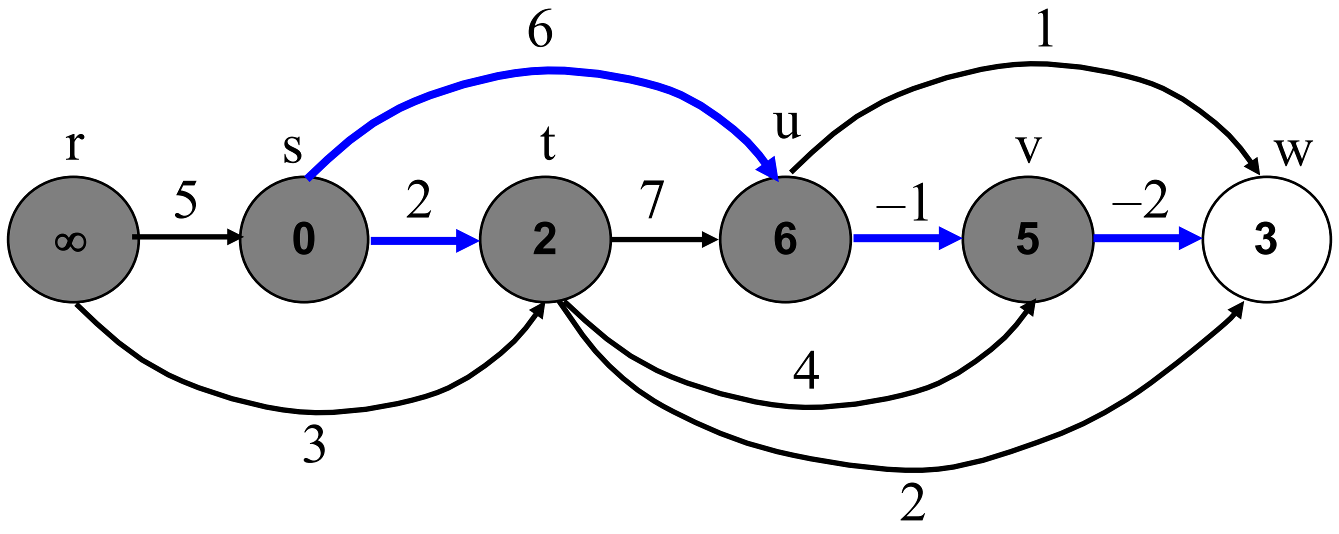
```
DAG-SHORTEST-PATHS(G,w,s)
topologically sort the vertices of G
INITIALIZE-SINGLE-SOURCE(G,s)
for each vertex u, taken in topologically sorted order
    for each vertex v in G.adj[u]
        RELAX(u,v,w)
```





```
DAG-SHORTEST-PATHS(G,w,s)
topologically sort the vertices of G
INITIALIZE-SINGLE-SOURCE(G,s)
for each vertex u, taken in topologically sorted order
    for each vertex v in G.adj[u]
        RELAX(u,v,w)
```

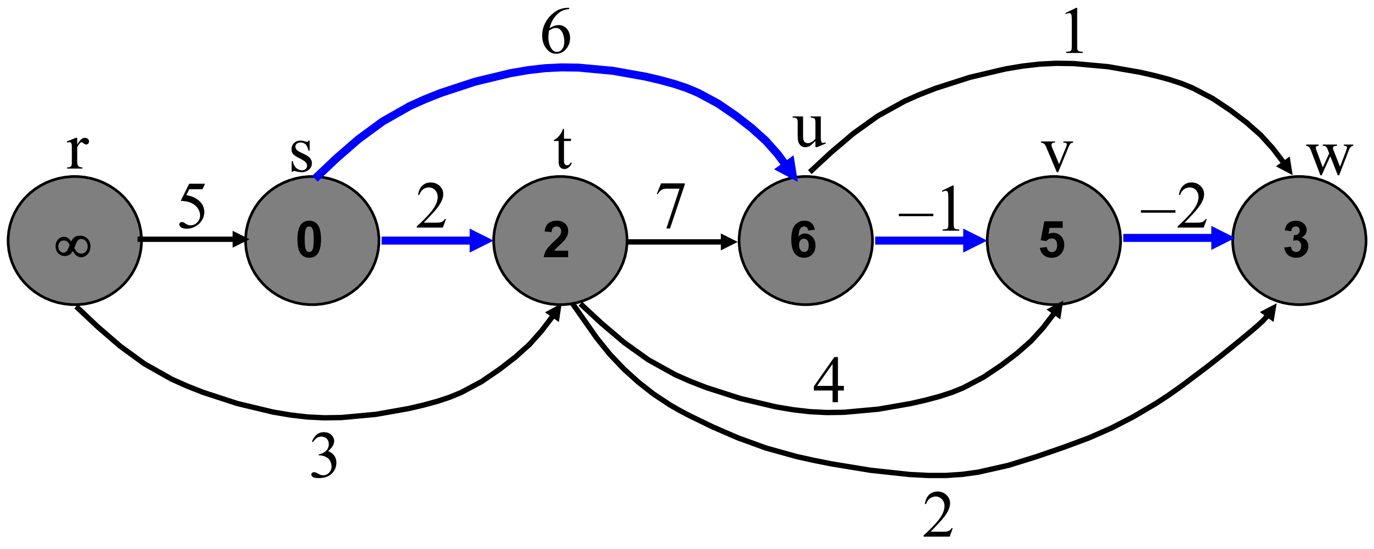




Running time analysis

```
DAG-SHORTEST-PATHS(G,w,s)
topologically sort the vertices of G // Θ(V+E)
INITIALIZE-SINGLE-SOURCE(G,s) // Θ(V)
for each vertex u, taken in topologically sorted order  ⎫
    for each vertex v in G.adj[u]                       ⎬ Θ(V+E)
        RELAX(u,v,w)                                    ⎭
```

=> total running time is Θ(V+ E), same as topological sort



Proof of correctness

Claim: if G = (V, E) is a DAG, then at the termination of `DAG-SHORTEST-PATHS`, v.d = $\delta(s,v)$, for all v

Proof by induction on the position in topological sort order

Inductive hypothesis: if all the vertices before *v* in a topological sort order have been updated, then v.d = $\delta(s,v)$

Base case:

- For all v before s, v.d = ∞ = $\delta(s,v)$
- s.d = 0 = $\delta(s,s)$

Proof of correctness

Claim: if G = (V, E) is a DAG, then at the termination of **`DAG-SHORTEST-PATHS`**, v.d = $\delta(s,v)$, for all v

Proof by induction on the position in topological sort order

Inductive hypothesis: if all the vertices before v in a topological sort order have been updated, then v.d = $\delta(s,v)$

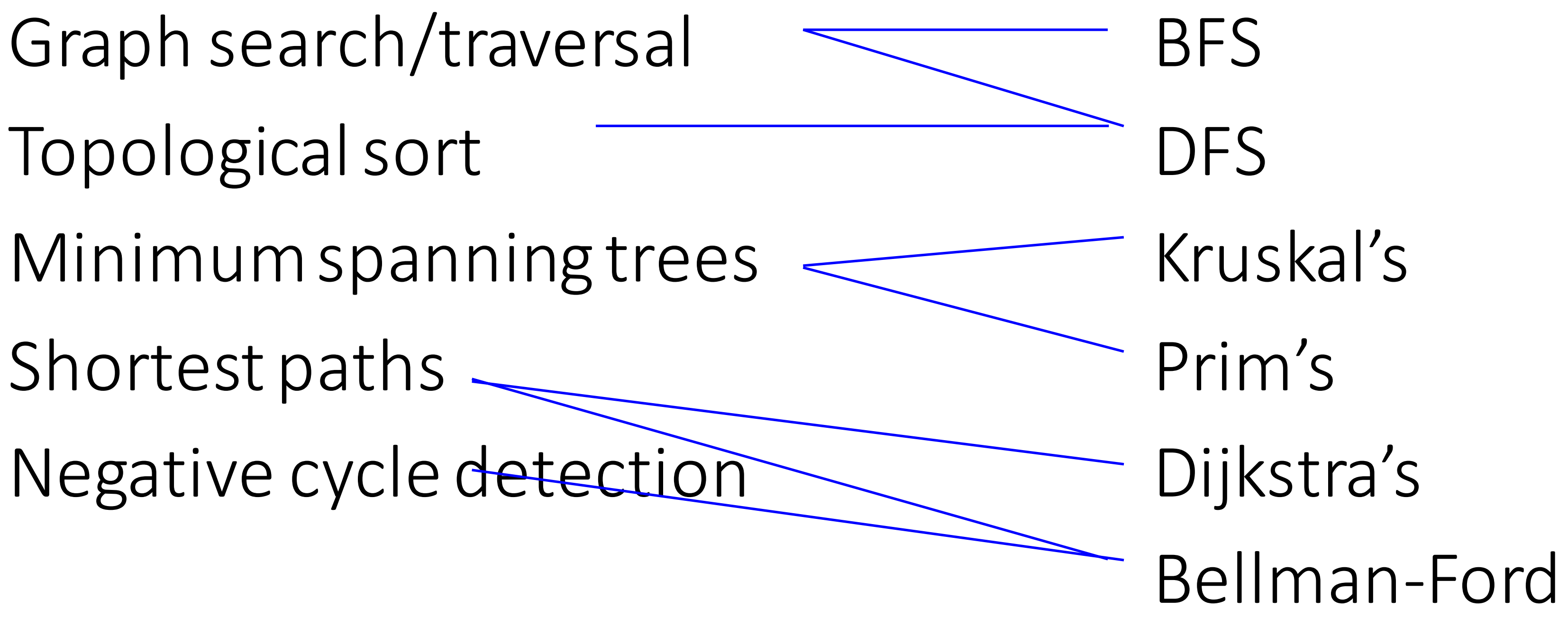
Inductive step:

- Consider a vertex v after s
- By construction, v.d = $\min_{(u,v) \text{ in } E}$ (u.d + w(u,v))
- By inductive hypothesis, u.d + w(u,v) = $\delta(s,u)$ + w(u,v)
- Since some (u,v) must be on the shortest path, by optimal substructure, v.d = $\delta(s,v)$

Summary of single-source shortest-path algorithms

SSSP algorithm	Applicable graph types	Running time
Dijkstra	Nonnegative weights	$\Theta(V^2)$ (array-based)
Topological sort based	DAG	$\Theta(V+E)$
Bellman-Ford	generic	$\Theta(EV)$

Summary of graph algorithms



Application: Internet routing

Vertices = routers, ASes

Edges = network links between routers

Edge weight = delay, bandwidth, cost, hop count, etc.

Link-state (commonly using Dijkstra's algorithm)

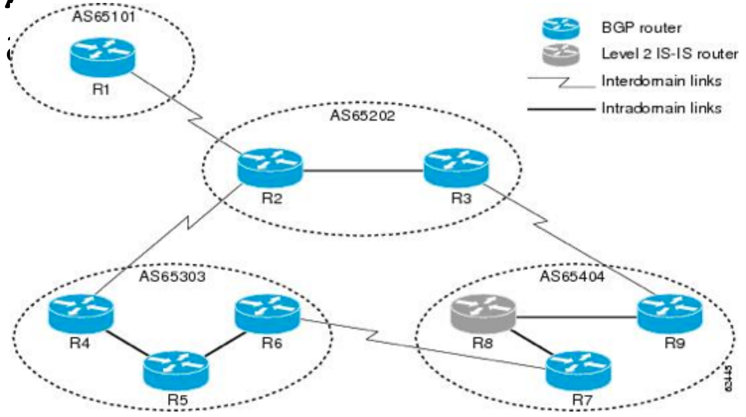
- Nodes flood link state to whole network
- E.g., Open Shortest Path First (OSPF)

Distance-vector (commonly using Bellman-Ford's algorithm)

- Nodes send vectors of destination and distance to neighbors
- E.g., Routing Information Protocol (RIP)

Path-vector (not necessarily shortest paths)

- Nodes advertise the full paths to each destin
- E.g., Border Gateway Routing Protocol (BGP)

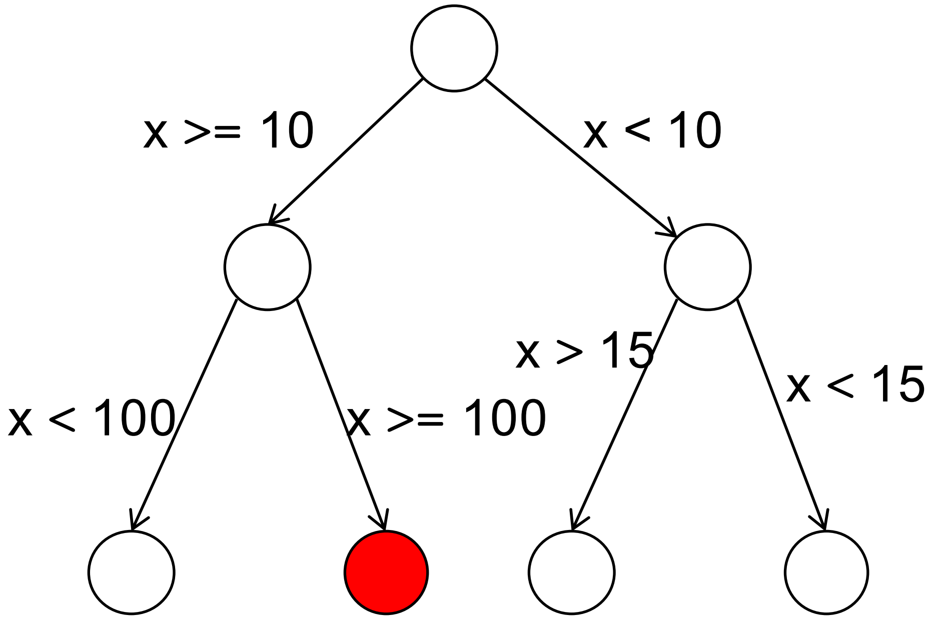


Source: cisco.com

Application: automatic bug finding

Can be seen as a graph search problem on program execution

```
x = int(input())
if x >= 10:
    if x < 100:
        print "Two!"
    else:
        print "Lots!"
else:
    if x > 15:
        print "One!”
    else:
        print ”Three!"
```



How to find the red state?
What should x be to reach the red state?
Is there unreachable state?