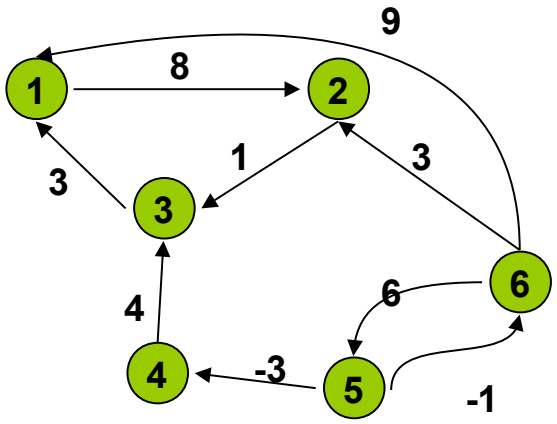
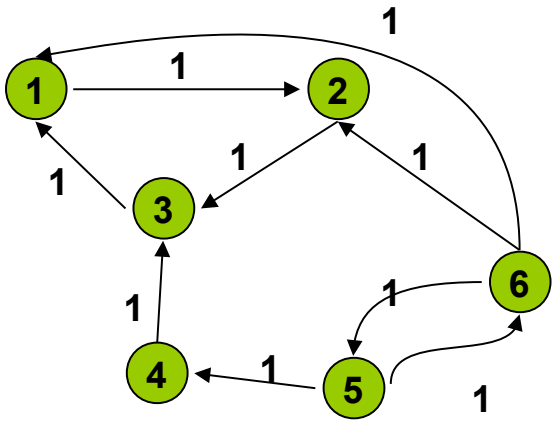
# Cammini minimi con una sorgente

- ✓Problema dei cammini minimi
- ✓Varianti e archi negativi
- ✓Sottostruttura ottima di un cammino minimo
- ✓Algoritmo di Dijkstra
- ✓Complessità dell'algoritmo
- ✓Rappresentazione dei cammini minimi

# Problema dei cammini minimi

Input: un grafo G=(V,E) orientato e pesato, con una funzione peso w: E → R, che associa ad ogni arco in E un peso a valore reale.

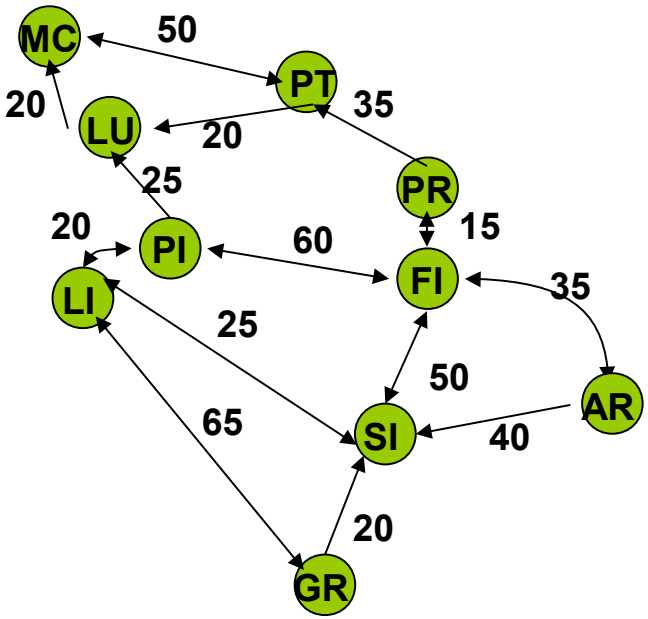


# Problema dei cammini minimi

Il **costo** di un cammino p= <$v_1$, $v_2$, …,$v_k$> è la somma dei pesi degli archi che lo costituiscono.

$$w(p)=\sum_{i=2}^{k} w(v_{i-1},v_i)$$


p1= <PR, FI, SI> =

15 min + 50 min = 65 min

p2 = <PR, FI, AR, SI> =

15 min + 35 min + 40 min =

= 90 min

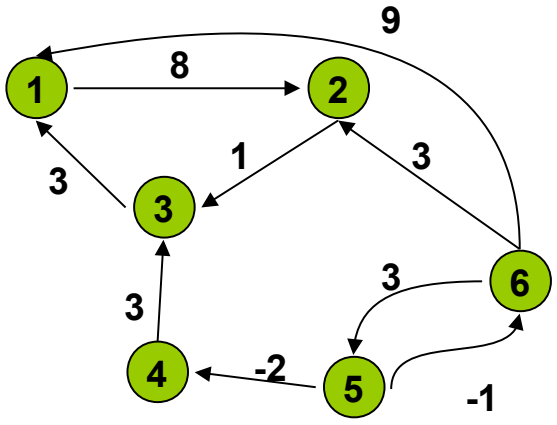
p3 = <PR, FI, PI, LI, GR, SI> = 160 min

# Problema dei cammini minimi

Il **costo di un cammino minimo** dal vertice ***u*** al vertice ***v*** è definito da:

$$\delta(u,v)=\begin{cases}\min\{w(p): u \overset{p}{\rightsquigarrow} v\} & \text{se esiste un cammino da } u \text{ a } v \\ \infty & \text{altrimenti}\end{cases}$$

Un **cammino minimo** dal vertice ***u*** al vertice ***v*** è definito come un qualunque cammino p con peso w(p) = δ(u,v). **Può non essere unico!**




δ(6,1) = 7

$p_1$ = <6, 2, 3, 1 > $w(p_1) = 7$

$p_2$ = <6, 1> $w(p_2) = 9$

$p_3$ = <6, 5, 6, 2, 3, 1 > $w(p_3) = 9$

$p_4$ = <6, 5, 4, 3, 1 > $w(p_4) = 7$

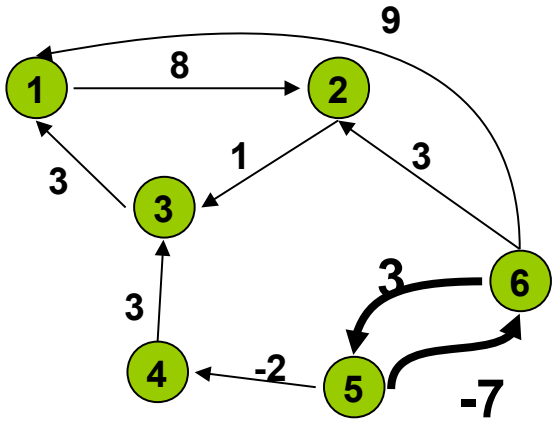
# Vari problemi

- **Problema di cammini minimi con sorgente singola**: si vuole trovare un cammino minimo da un dato vertice sorgente s ad ogni vertice v in V.
- **Problema di cammini minimi con destinazione singola**: si vuole trovare da ogni vertice v in V un cammino minimo ad un dato vertice destinazione t.
- **Problema di cammini minimi tra una coppia**: si vuole trovare un cammino minimo da u a v.
- **Problema di cammini minimi tra tutte le coppie**: determinazione di un cammino minimo per ogni coppia di vertici u e v.

# Archi con pesi negativi

**NOTA**: sono ammessi archi di peso negativo, ma **non** devono esistere nel grafo **cicli di costo negativo**.

Se il costo di un ciclo è negativo, allora tutti i nodi raggiungibile dal ciclo hanno **un cammino minimo di peso infinitamente negativo (-∞).**




Ciclo <6,5> negativo.

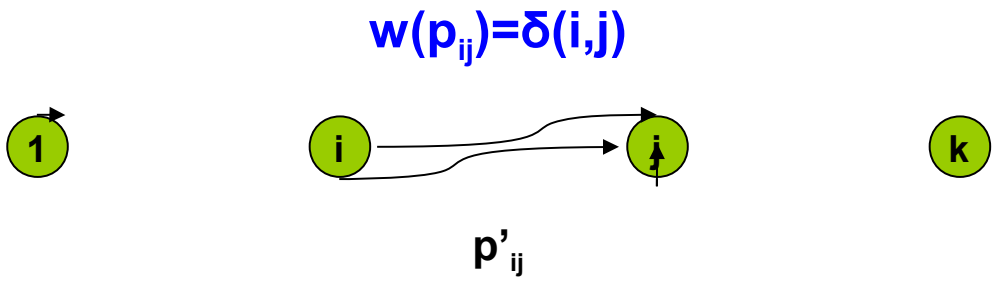
Ogni volta che compio un giro diminuisco il peso del cammino che passa per il ciclo.

**Esempio**: δ(6,1) = -∞

# Sottostruttura ottima di un cammino minimo

**Sottocammini di cammini minimi sono cammini minimi**

Dato un grafo G=(V,E) con funzione peso w:E→ R, sia p= <$v_1$, $v_2$, …,$v_k$> un cammino minimo da $v_1$ a $v_k$. Per ogni *i* e *j* tali che 1 ≤ *i* ≤ *j* ≤ k, si ha che **il sottocammino $p_{ij}$= <$v_i$, $v_{i+1}$, …,$v_j$> è un cammino minimo da *i* a *j***.



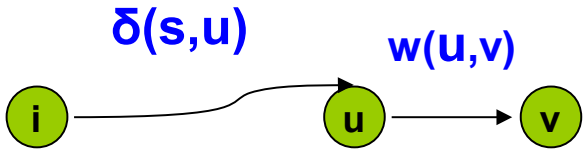

Dato un altro sottocammino $p'_{ij}$ da *i* a *j*, necessariamente

$w(p_{ij}) \leq w(p'_{ij})$, altrimenti il cammino minimo passerebbe per $p'_{ij}$

# Sottostruttura ottima di un cammino minimo

**Di conseguenza:**

Si supponga che **un cammino minimo p** da una sorgente *s* ad un vertice *v* passi per l'arco (*u*,*v*) con peso w(*u*,*v*). Il peso del cammino minimo da *s* a *v* è **δ(*s*,*v*) = δ(*s*,*u*) + w(*u*,*v*).**




cammino minimo tra u e v

**δ(s,v) = δ(s,u) + w(u,v).**

Più **in generale**, se esiste un arco (x,v), allora si ha:

**δ(s,v) ≤ δ(s,x) + w(x,v),** vale l'uguaglianza se l'arco (***x,v***) appartiene ad un cammino minimo da *s a v*.

# Algoritmo di Dijkstra

L'algoritmo di Dijkstra risolve **il problema dei cammini minimi con sorgente singola** *s* su un grafo orientato (o non orientato) e pesato G=(V,E) nel caso in cui **tutti i pesi** degli archi siano **non negativi**.

Ci sono due insiemi:

- **S**: nodi *v* per cui d[v] = δ(s,v), quindi un cammino minimo tra ***s*** e ***v*** è stato determinato.

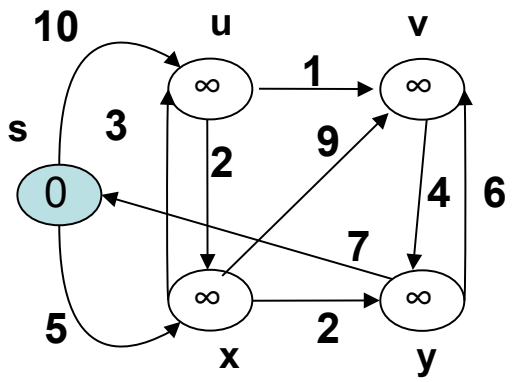
- **Q = V/S**: una coda a priorità dei nodi v, per cui d[v] ha il valore del cammino da *s* con peso minore finora scoperto.

**All'inizio**, S contiene solo ***s***, **d[s]=0**, mentre Q=V / {s} con d[v]=∞.

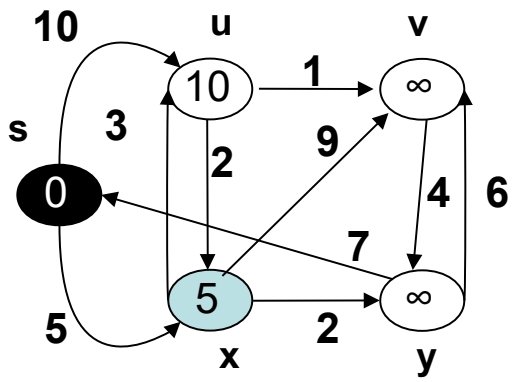
# Algoritmo di Dijkstra

```
DIJKSTRA(G,w,s)
1.    for ogni vertice u in V[G]          // inizializzazione di ogni vertice
2.        do d[u] ← ∞
3.           p[u] ← NIL
4.    d[s] ← 0                            // si comincia dal vertice s
5.    Q ← V[G]                            // coda a priorità
6.    S ← Ø                               // insiemi dei nodi con cammino minimo trovato
7.    while Q≠Ø                           // termina quando la coda Q è vuota
8.        do u ← EXTRACT-MIN(Q)  // prendi il nodo u con d[u] minore
9.           S ← S U {u}                  // inserisci u in S
             for ogni vertice v in Adj[u]    / S
10.          do if d[v] > d[u] + w(u,v)    // Procedura relax(u,v,w)
11.             then d[v] ← d[u] + w[u,w]  // aggiorna cammini minimi
                                              per v adiacente  u
12.             p[v] ← u
13.             Q.decrease.key(v, d[v])    // Aggiorna le distanze in Q
```

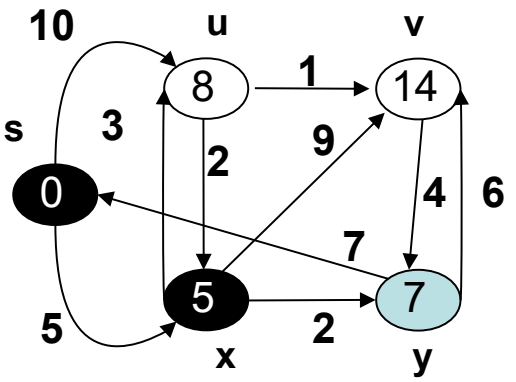
# Esempio



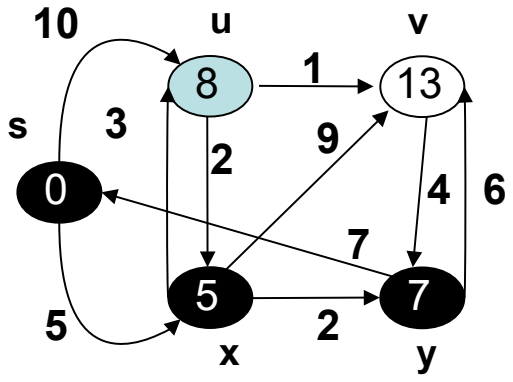

(a)



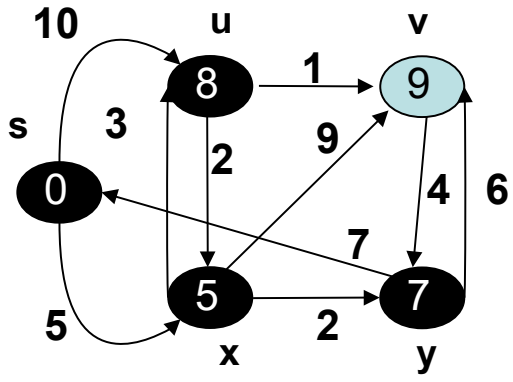

(b)



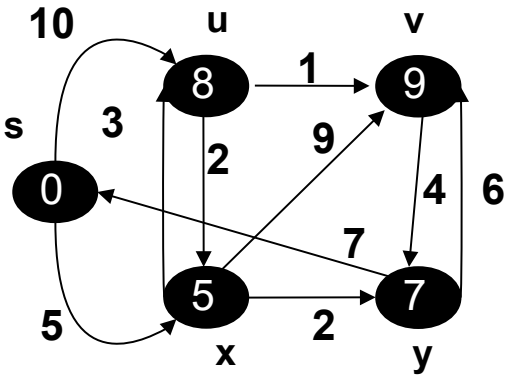

(c)




(d)

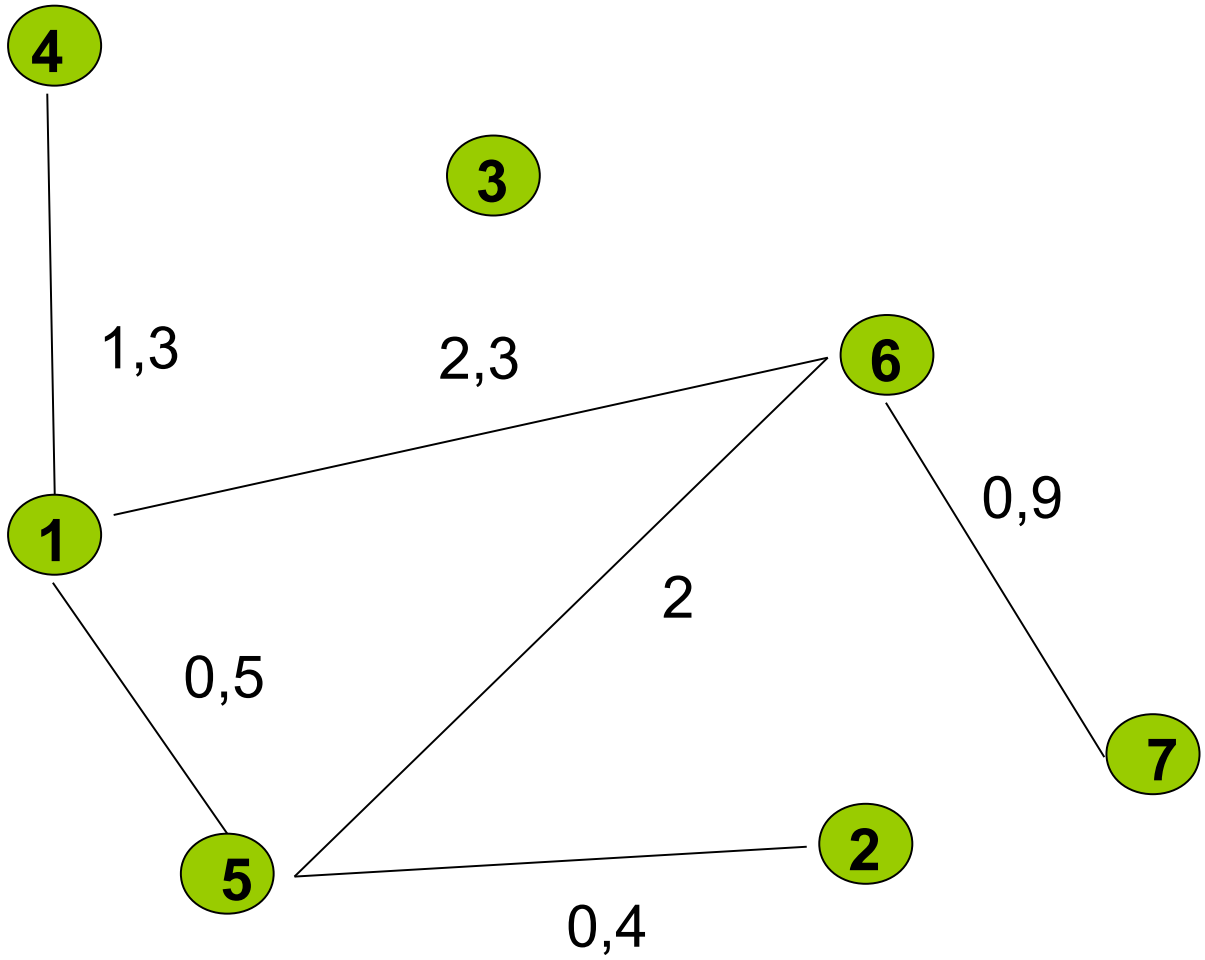



(e)




(f)

# Esempio



# Complessità

Si consideri che la coda con priorità Q come un **array lineare**:

- Inizzializzazione **tempo O(|V|)**
- **EXTRACT-MIN(Q):** ricerca del minimo in Q. Bisogna vedere tutti i valori in Q, **richiede tempo O(|V|)**
  - EXTRACT-MIN(Q) viene eseguito per ogni vertice quindi il tempo totale è **O(|V|x|V|) = O(|V|²)**
- Come in BFS vengono, **si esamina la lista di adiacenza di ogni vertice v,** che entra solo una volta in S. La somma di tutte liste di adiacenze è |E|**,** più il costo delle descrease key totale O(|V||E|)
- **Il tempo totale** dell'algoritmo di Dijkstra è **O(|V||E|+|V|²).**

# Complessità

L'algoritmo di Dijkstra con **binary heap**: l'analisi

1. Inizializzazione di Q
   - Build-Heap che costruisce un heap con |V| elementi in O(V)
2. Analisi del ciclo di **while**
   - V operazioni Extract-Min
     - Ciascuna richiede tempo O(log V)  (totale **O(VlogV)**)
   - V aggiornamenti di S
     - Ciascuna richiede tempo costante
   - Un totale di O(E) iterazioni del ciclo di **for** (linee 10-13) in quanto la lista di adiacenza di ciascun vertice ***u*** viene scandita esattamente una volta (quando ***u*** è inserito in S)
     - **RELAX(u, v, w)** sulla linea 10 :
       - L'aggiornamento di **d[v]** è effettuato eseguendo DECREASEKEY( Q, v , d[u] + w(u, v)) che richiede tempo O(log V). Totale **O(ElogV)**
     - L'aggiornamento di p[v] richiede tempo costante

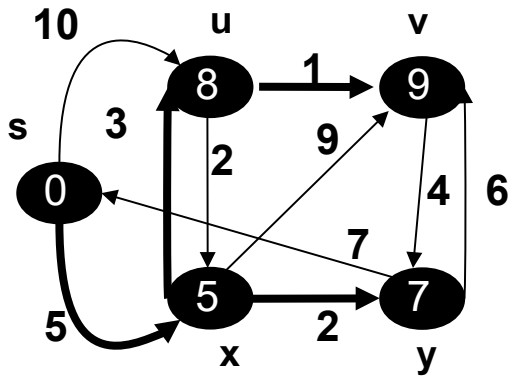
**Tempo totale: O(Vlog V + E log V)**

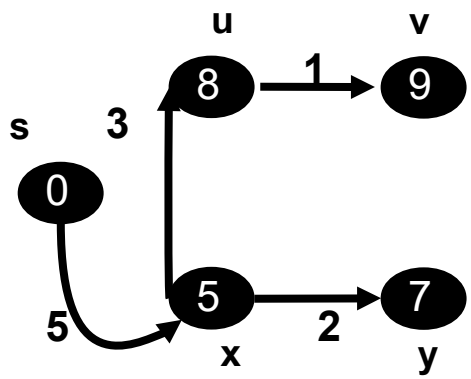
# Rappresentazione dei cammini minimi

Come nella visita in ampiezza (BFS), l'algoritmo Dijkstra definisce un **sottografo dei predecessori $T_p=(V_p,E_p)$.**

L'albero che ne segue contiene i cammini minimi individuati da s ad ogni vertice raggiungibile v.

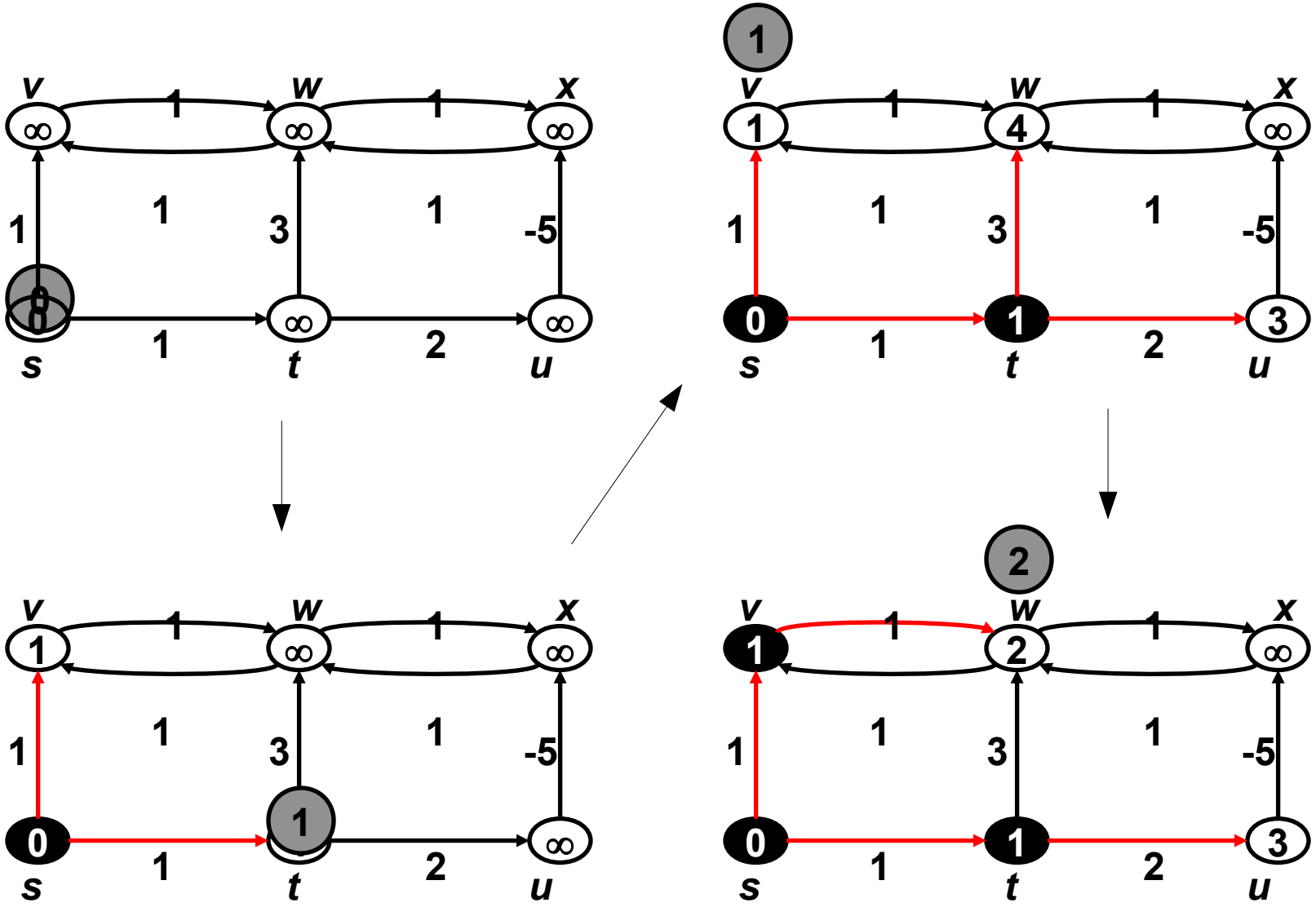
Nota: questo vale solo al termine dell'algoritmo.



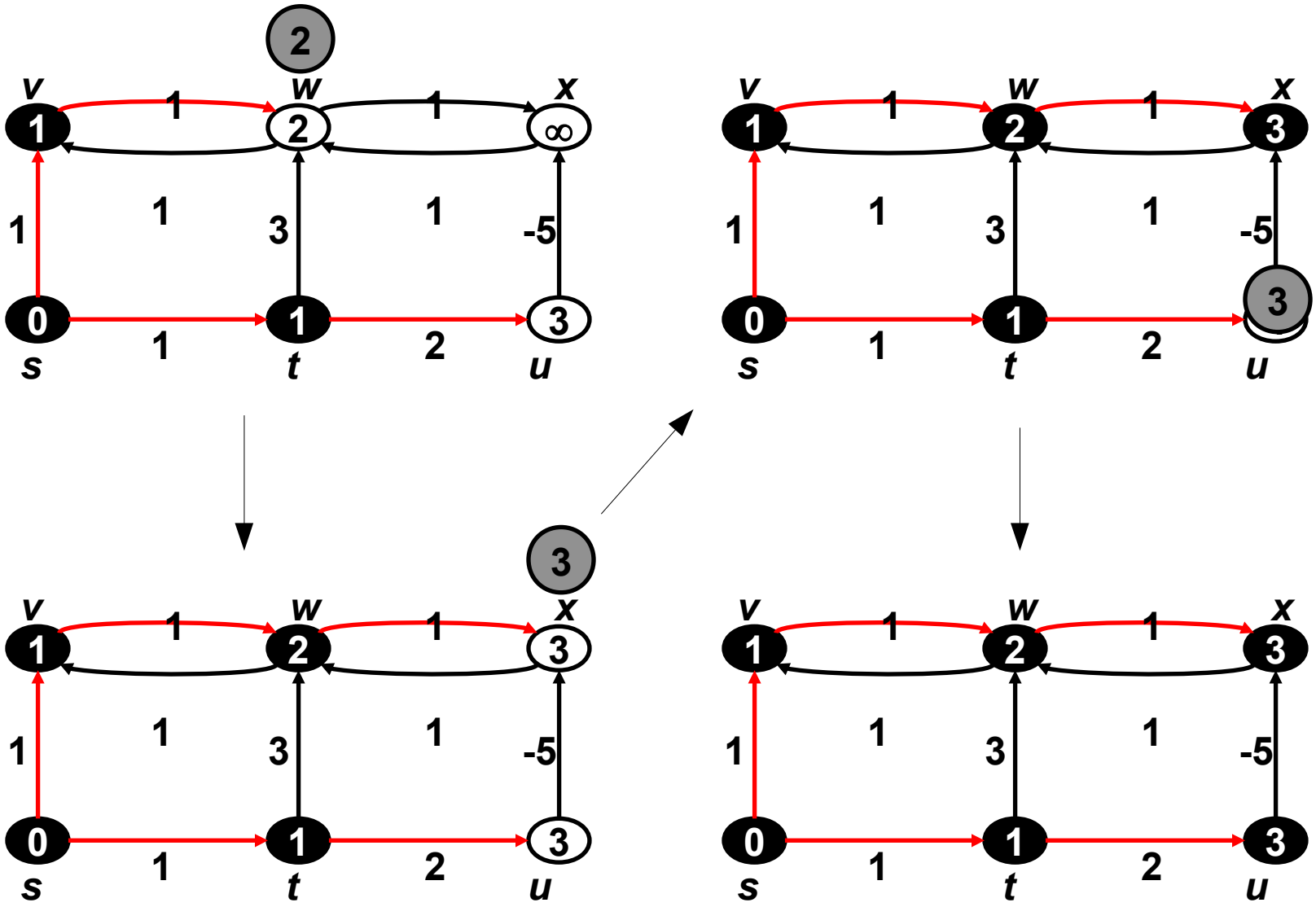




$T_p=(V_p,E_p)$

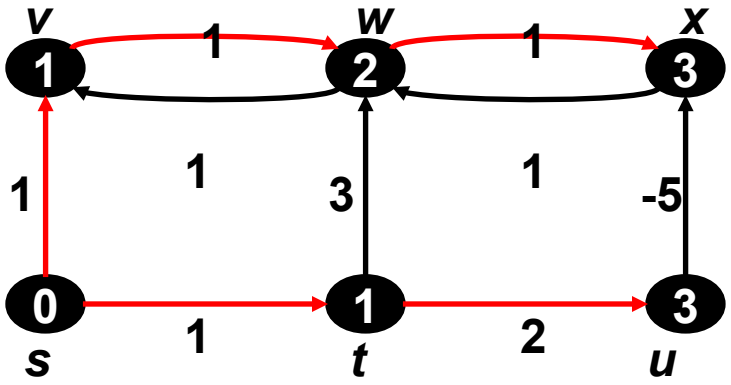
# Problemi Algoritmo Dijkstra



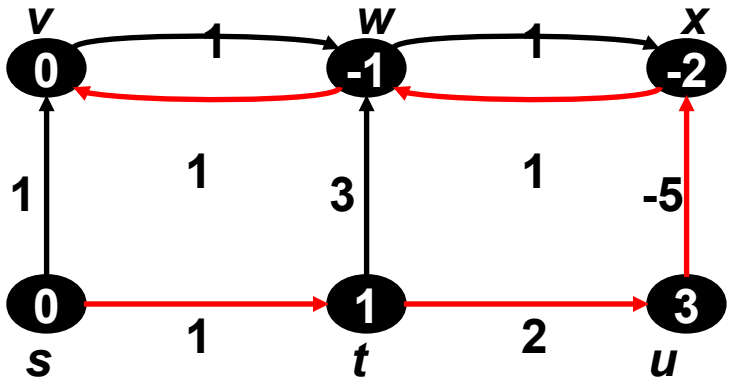
# Problemi Algoritmo Dijkstra



# Problemi Algoritmo Dijkstra




**Soluzione di Dijkstra**




**Soluzione Ottimale**

# Algoritmo di Bellman-Ford

Ricordiamo che per ogni arco $(u,v)$ in E e per ogni vertice $s$ in V, vale:

$$\delta(s,v) \leq \delta(s, u) + w(u,v)$$

- **Tecnica del Rilassamento:**
  - Si parte da stime per eccesso delle distanze $d[y] \geq \delta(s,y)$
  - Si rilassa ogni arco $(u, v)$ aggiornando le stime progressivamente fino a renderle esatte, cioè

    [*Rilassamento* dell'arco $(u, v)$]
    **if** $d[u] + w(u, v) < d[y]$
    **then** $d[y] \leftarrow d[u] + w(u, v)$

# Algoritmo di Bellman-Ford

- Esegue |V|-1 iterazioni
- In ciascuna iterazione rilassa tutti gli archi
- Si prova che dopo la *j*-esima iterazione, i primi *j* rilassamenti hanno sicuramente già raggiunto la stima corretta

- Esegue però molti rilassamenti inutili!

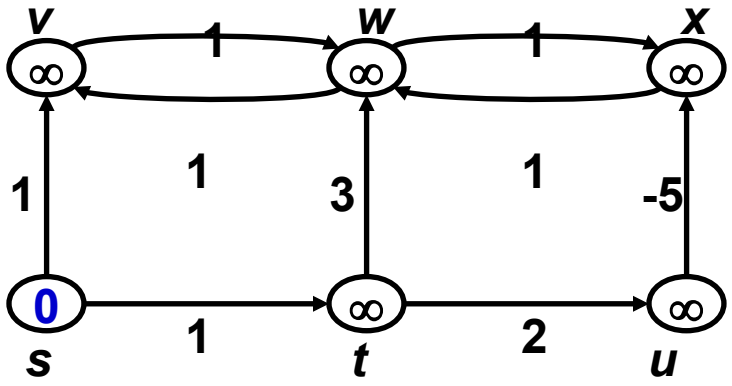
# Algoritmo di Bellman-Ford

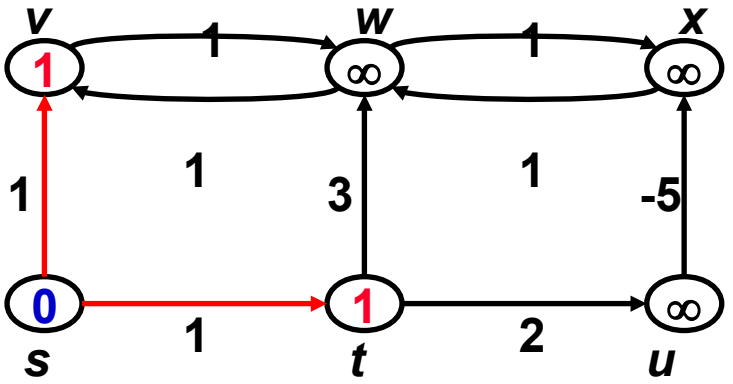
- Inizalmente, $d[s] = 0$ e $d[u] = \infty$ per $u \neq s$
- Vengono fatte $|V| - 1$ passate
- Ad ogni passata, applica il rilassamento *ad ogni arco*

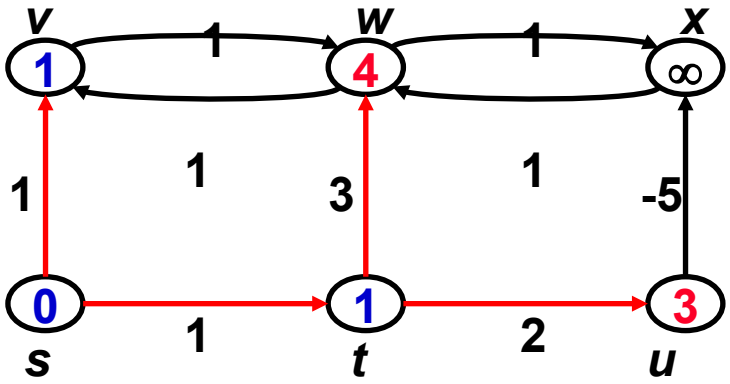
**Tempo = $O(|V||E|)$**

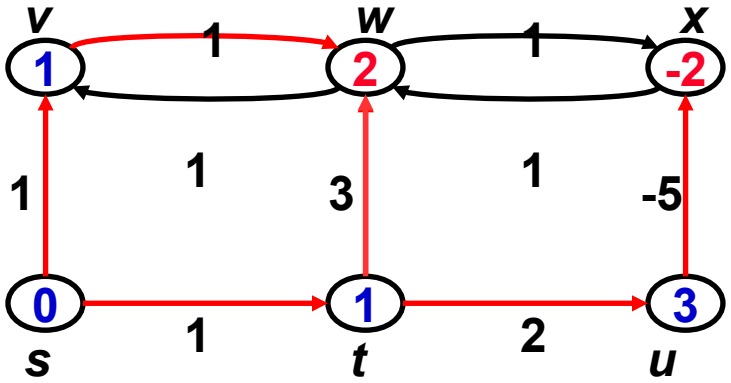
```
Bellman_Ford(G,s)
   Inizializza(G,s,d)
   for i = 1 to |V|-1
       do for each arco (u,v) in E
          do relax(u,v,w)
   for each arco (u,v) in E //controllo
presenza cicli nagativi
       do if d[v] > d[u] + w(u,v)
          then return FALSE
   return TRUE
```

# Esempio Algoritmo Bellman-Ford



# Esempio Algoritmo Bellman-Ford