

# Cammini di costo minimo

Jocelyne Elias

<https://www.unibo.it/sitoweb/jocelyne.elias/>

**Moreno Marzolla**

<https://www.moreno.marzolla.name/>

Dipartimento di Informatica—Scienza e Ingegneria (DISI)  
Università di Bologna

Copyright © 2010—2016, 2020, 2021  
Moreno Marzolla, Università di Bologna, Italy  
<https://www.moreno.marzolla.name/teaching/ASD/>



*This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International (CC BY-SA 4.0) License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/4.0/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.*

# Definizione del problema

- Consideriamo un grafo orientato  $G = (V, E)$  in cui ad ogni arco  $(u, v) \in E$  sia associato un costo  $w(u, v)$
- Il costo di un cammino  $\pi = (v_0, v_1, \dots, v_k)$  che collega il nodo  $v_0$  con  $v_k$  è definito come

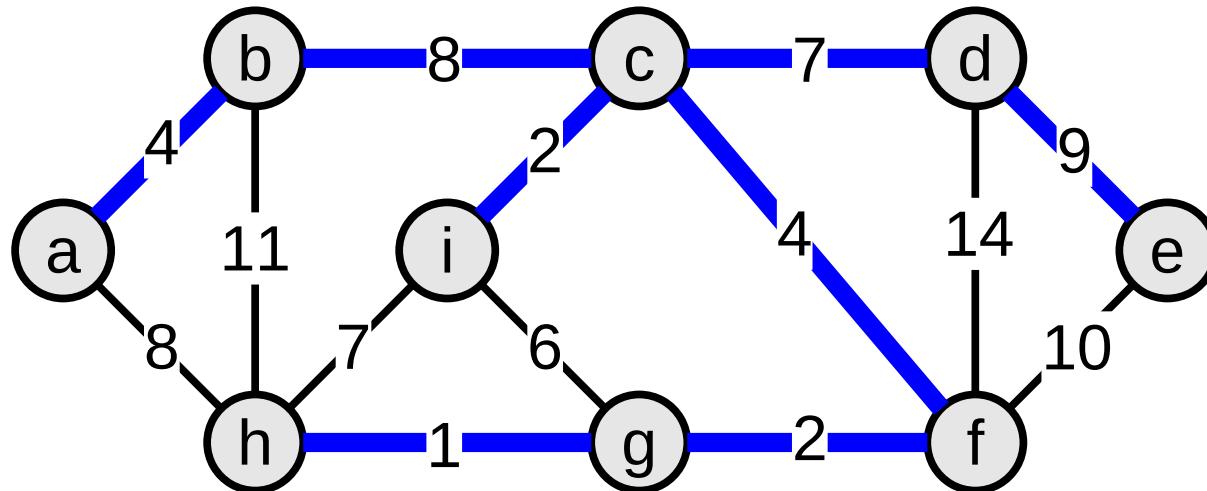
Funzione  
Peso

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

- Data una coppia di nodi  $v_0$  e  $v_k$ , vogliamo trovare (se esiste) il cammino  $\pi_{v_0 v_k}^*$  di costo minimo tra tutti i cammini che vanno da  $v_0$  a  $v_k$

# Osservazione

- Il problema del MST e dei cammini di costo minimo sono due problemi **differenti**
  - Es: il cammino di costo minimo che collega  $h$  e  $i$  è  $(h, i)$  oppure  $(h, g, i)$ , entrambi di peso 7
  - In questo caso il cammino di costo minimo non fa parte del MST



Cammini di Costo Minimo

## Calcolo del costo minimo

Scopo: Trovare il percorso più economico (in termini di somma dei pesi degli archi) tra due nodi specifici (ad esempio da A a F) in un grafo. È un problema punto-a-punto.

## Minimun Spanning Tree (MST)

Scopo: Trovare un insieme di archi che collega tutti i nodi del grafo senza cicli e con il costo totale minimo possibile. È un problema di connettività globale.

# Applicazioni

The screenshot shows a Google Maps interface for driving directions. The starting point is Bologna, Metropolitan City of Bologna, and the destination is Cesena, Province of Forlì-Cesena. The route is highlighted in blue and grey, indicating the fastest route with usual traffic. A callout box for the first segment shows a car icon, 1 h 7 min, and 88.9 km. Another callout box for the second segment shows a bus icon, 1 h 12 min every 60 min. The map also shows other routes like E45, E35, and SS9, and various Italian towns and landmarks. At the bottom left, there's a 'Explore Bologna' section with icons for Restaurants, Hotels, Gas stations, Parking Lots, and More. At the top right, there are 'Lido' and 'Sign in' buttons.

# Applicazioni (oops!)

Cammino più breve tra Haugesund e Trondheim  
secondo Microsoft Autoroute

Nel 2005



Nel 2010



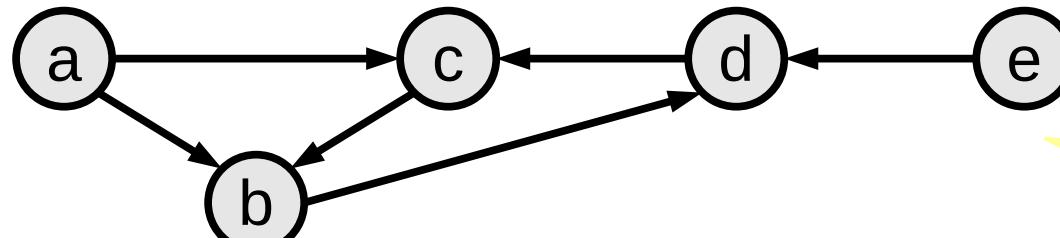
# Diverse formulazioni del problema

1. Cammino di costo minimo fra una singola coppia di nodi  $u$  e  $v$ 
  - Determinare, se esiste, il cammino di costo minimo  $\pi_{uv}^*$  da  $u$  verso  $v$
2. Single-source shortest path
  - Determinare i cammini di costo minimo da un nodo sorgente  $s$  a tutti i nodi raggiungibili da  $s$
3. All-pairs shortest paths
  - Determinare i cammini di costo minimo tra ogni coppia di nodi  $u, v$
  - Non è noto alcun algoritmo in grado di risolvere il problema (1) senza risolvere anche (2) nel caso peggiore

# Osservazione

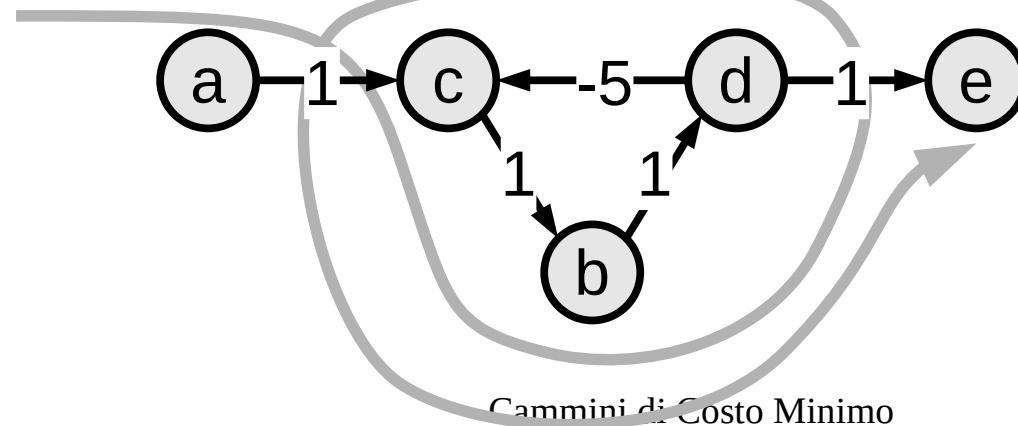
- In quali situazioni **non** esiste un cammino di costo minimo?

- Quando la destinazione non è raggiungibile



Non esiste alcun cammino che connette **a** con **e**

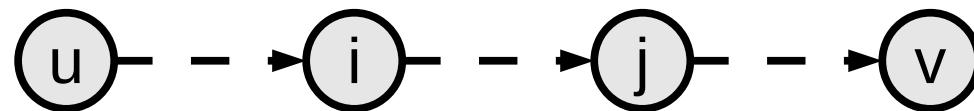
- Quando ci sono **cicli di costo negativo**



È sempre possibile trovare un cammino di costo inferiore che connette **a** con **e**

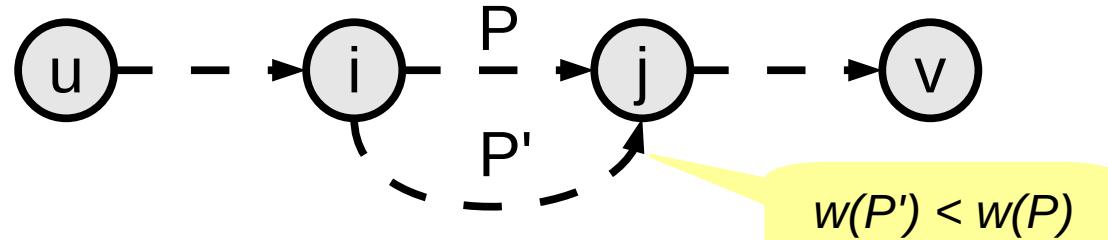
# Proprietà (sottostruttura ottima)

- Sia  $G = (V, E)$  un grafo orientato con funzione costo  $w$ ; allora ogni sotto-cammino di un cammino di costo minimo in  $G$  è a sua volta un cammino di costo minimo
- Dimostrazione
  - Consideriamo un cammino minimo  $\pi_{uv}^*$  da  $u$  a  $v$
  - Siano  $i$  e  $j$  due nodi intermedi
  - Dimostriamo che il sotto-cammino di  $\pi_{uv}^*$  che collega  $i$  e  $j$  è un cammino minimo tra  $i$  e  $j$

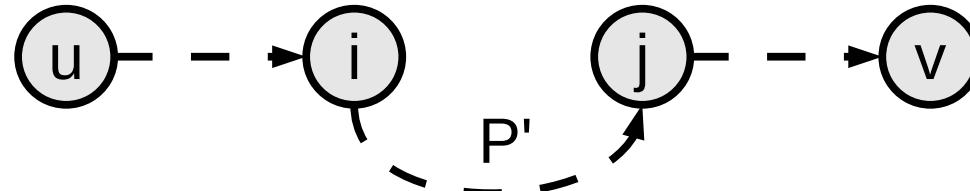


# Proprietà (sottostruttura ottima)

- Supponiamo per assurdo che esista un cammino  $P'$  tra  $i$  e  $j$  di costo strettamente inferiore a  $P$



- Ma allora potremmo costruire un cammino tra  $u$  e  $v$  di costo inferiore a  $\pi_{uv}^*$ , il che è assurdo perché avevamo fatto l'ipotesi che  $\pi_{uv}^*$  fosse il cammino di costo minimo

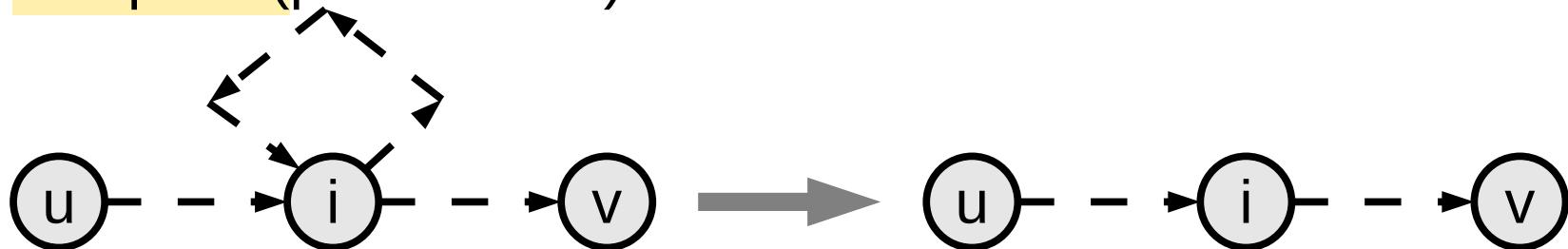


Cammini di Costo Minimo

# Esistenza

- Sia  $G = (V, E)$  un grafo orientato con funzione peso  $w$ . Se non ci sono cicli negativi, allora fra ogni coppia di vertici connessi in  $G$  esiste sempre un cammino semplice di costo minimo
- Dimostrazione

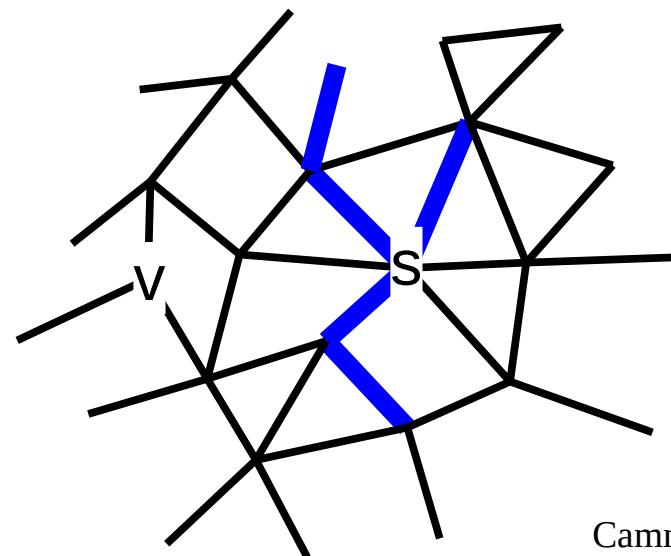
- Possiamo sempre trasformare un cammino in un cammino semplice (privo di cicli)



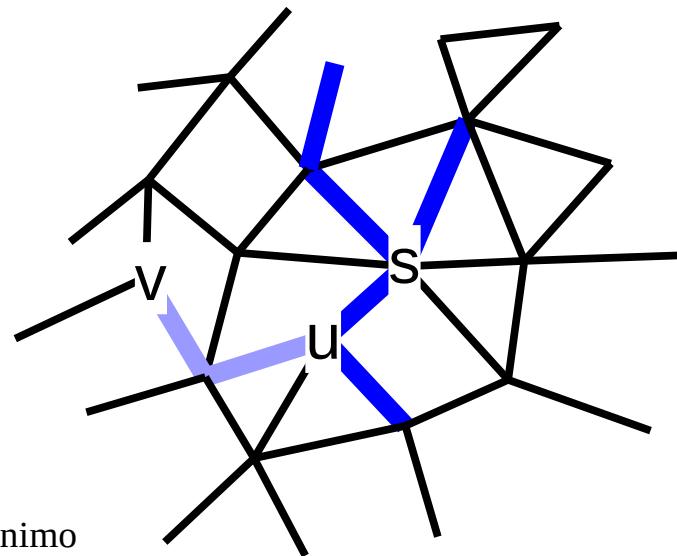
- Ogni volta che si rimuove un ciclo, il costo diminuisce (o resta uguale), perché per ipotesi non ci sono cicli negativi
  - Il numero di cammini è finito, esiste il minimo

# Albero dei cammini di costo minimo

- Sia  $s$  un nodo di un grafo orientato pesato  $G = (V, E)$ . Allora esiste un albero  $T$  che contiene i nodi raggiungibili da  $s$  tale che ogni cammino in  $T$  sia un cammino di costo minimo
  - Grazie alla proprietà di sottostruttura ottima, è sempre possibile far “crescere” un albero parziale  $T'$  fino a includere tutti i vertici raggiungibili



Cammini di Costo Minimo



# Distanza tra vertici in un grafo

- Sia  $G = (V, E)$  un grafo orientato con funzione costo  $w$ . La distanza  $d_{xy}$  tra  $x$  e  $y$  in  $G$  è il costo di un cammino di costo minimo che li connette;  $+\infty$  se tale cammino non esiste

$$d_{xy} = \begin{cases} w(\pi_{xy}^*) & \text{se esiste un cammino } \pi_{xy}^* \text{ di costo minimo} \\ +\infty & \text{altrimenti} \end{cases}$$

- **Nota:**  $d_{vv} = 0$  per ogni vertice  $v$
- **Nota:** Vale la diseguaglianza triangolare

$$d_{xz} \leq d_{xy} + d_{yz}$$

Se la diseguaglianza triangolare fosse violata (come nell'esempio del pedaggio da 100), questa certezza verrebbe meno. L'algoritmo di Dijkstra marcarebbe  $C$  con un costo di 8 attraverso  $B$ , ma poi scoprirebbe l'arco diretto  $A \rightarrow C$  con costo 100 e lo scarterebbe perché più costoso. Funzionerebbe comunque bene.

# Condizione di Bellman

- Per ogni arco  $(u, v)$  e per ogni vertice  $s$ , vale la seguente diseguaglianza

$$d_{sv} \leq d_{su} + w(u, v)$$

- Dimostrazione
  - Dalla diseguaglianza triangolare si ha

$$d_{sv} \leq d_{su} + d_{uv}$$

- Ma risulta anche

$$d_{uv} \leq w(u, v)$$



la distanza minima tra  $u$  e  $v$   
non può essere maggiore del  
costo dell'arco  $(u, v)$

da cui la tesi

# Trovare cammini di costo minimo

- Dalla condizione di Bellman

$$d_{sv} \leq d_{su} + w(u, v)$$

si può dedurre che l'arco  $(u, v)$  fa parte del cammino di costo minimo  $\pi_{sv}^*$  se e solo se

$$d_{sv} = d_{su} + w(u, v)$$

# Tecnica del *rilassamento*

- Supponiamo di mantenere una stima  $D_{sv} \geq d_{sv}$  della lunghezza del cammino di costo minimo tra  $s$  e  $v$
- Effettuiamo dei passi di “rilassamento”, riducendo progressivamente la stima finché si ha  $D_{sv} = d_{sv}$

```
if ( $D_{su} + w(u, v) < D_{sv}$ ) then  $D_{sv} \leftarrow D_{su} + w(u, v)$ 
```

$u \rightarrow$  un nodo di cui abbiamo appena migliorato la stima della distanza  $s$ .

$D_{sv} \rightarrow$  La migliore stima corrente della distanza da  $s$  a  $v$

$d_{sv} \rightarrow$  La vera distanza minima che vogliamo raggiungere

# Algoritmo di Bellman e Ford

- Consideriamo un cammino  $\pi_{sv_k^*} = (s, v_1, \dots, v_k)$  di costo minimo inizialmente ignoto

- Sappiamo che

$$d_{sv_k} = d_{sv_{k-1}} + w(v_{k-1}, v_k)$$

da cui partendo da  $D_{ss} = 0$ , potremmo effettuare i passi di rilassamento seguenti

$$\begin{aligned} D_{sv_1} &\leftarrow D_{ss} + w(s, v_1) \\ D_{sv_2} &\leftarrow D_{sv_1} + w(v_1, v_2) \\ &\vdots \\ D_{sv_k} &\leftarrow D_{sv_{k-1}} + w(v_{k-1}, v_k) \end{aligned}$$

# Algoritmo di Bellman e Ford

- Problema: noi non conosciamo gli archi del cammino minimo  $\pi_{svk}^*$  né il loro ordine, quindi non possiamo fare il rilassamento nell'ordine corretto
- Però se eseguiamo per ogni arco  $(u, v)$

```
if (Dsu + w(u, v) < Dsv) then Dsv ← Dsu + w(u, v)
```

sicuramente includeremo anche il primo passo di rilassamento “corretto”

$$D_{sv_1} \leftarrow D_{ss} + w(s, v_1)$$

# Algoritmo di Bellman e Ford

- Ad ogni passo consideriamo tutti gli  $m$  archi del grafo  $(u, v)$  ed effettuiamo il passo di rilassamento

```
if  $(D_{su} + w(u, v) < D_{sv})$  then  $D_{sv} \leftarrow D_{su} + w(u, v)$ 
```

- Dopo  $n - 1$  iterazioni (tante quanti sono i possibili vertici di destinazione dei cammini che partono da  $s$ ) siamo sicuri di aver calcolato tutti i valori  $D_{sv_k}$  corretti

# Algoritmo di Bellman e Ford

*single-source shortest path*

```
double[1..n] BellmanFord(Grafo G=(V,E,w), int s)
    int n ← G.numNodi();
    int pred[1..n], v, u;
    double D[1..n];
    for v ← 1 to n do      inizializza tutte le
        D[v] ← +∞ ;       distanze a "infinito" e
        pred[v] ← -1;      i predecessori a "non definito"
    endfor
    D[s] ← 0; → distanza del nodo sorgente, da se stesso (0)
    for int i ← 1 to n - 1 do      Fase di rilassamento
        for each (u,v) in E do
            if ( D[u] + w(u,v) < D[v] ) then
                D[v] ← D[u] + w(u,v);
                pred[v] ← u;
            endif
        endfor
    endfor
    // eventuale controllo per cicli negativi (vedi seguito)
    return D;
```

I nodi del grafo sono identificati dagli interi 1, ... n

$D[v]$  = (stima della) distanza del nodo  $v$  dalla sorgente  $s$

$\text{pred}[v]$  = predecessore del nodo  $v$  sul cammino di costo minimo che collega  $s$  con  $v$

- Costo  $O(nm)$

numero di archi  $m=|E|$   
numero di nodi  $n=|V|$

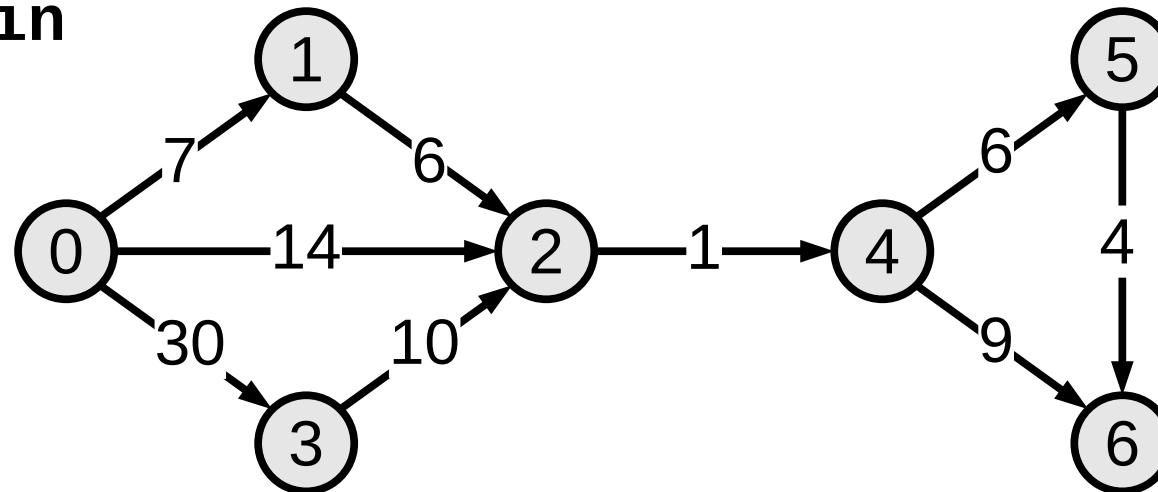
# Algoritmo di Bellman e Ford

- L'algoritmo di Bellman e Ford determina i cammini di costo minimo anche in presenza di archi con peso negativo
  - Però non devono esistere cicli di peso negativo
  - Il controllo seguente, da fare alla fine, determina se esistono cicli negativi

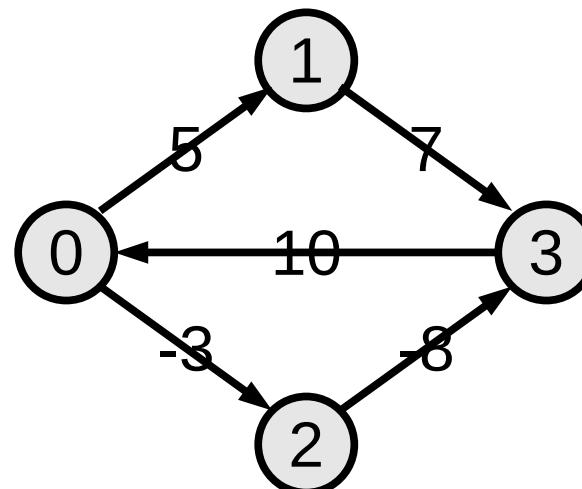
```
// eventuale controllo per cicli negativi
for each (u,v) in E do
    if ( D[u] + w(u,v) < D[v] ) then
        error "Il grafo contiene cicli negativi"
    endif
endfor
```

# Esempi

**simple.in**



**negative-cycle.in**



Cammini di Costo Minimo

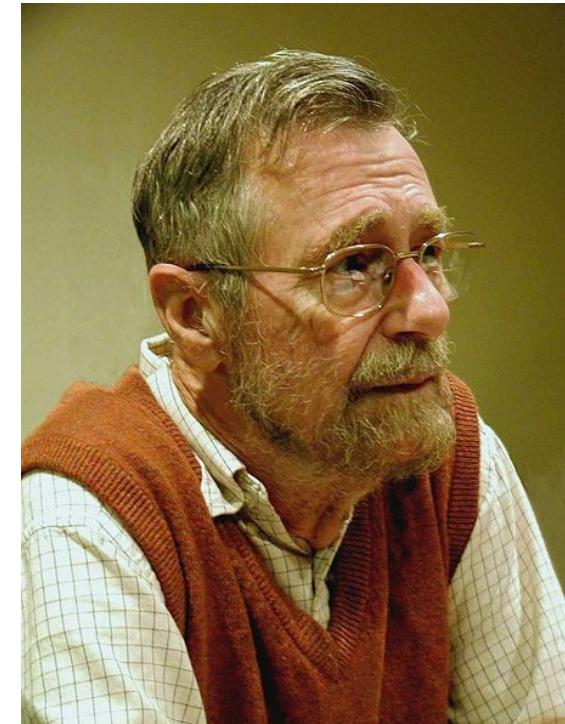
22

# Algoritmo di Dijkstra

## *Single-Source Shortest Path*

- Algoritmo più efficiente di quello di Bellman-Ford per determinare i cammini di costo minimo da singola sorgente **nel caso in cui tutti gli archi abbiano costo  $\geq 0$**

non gestisce costo  $< 0$



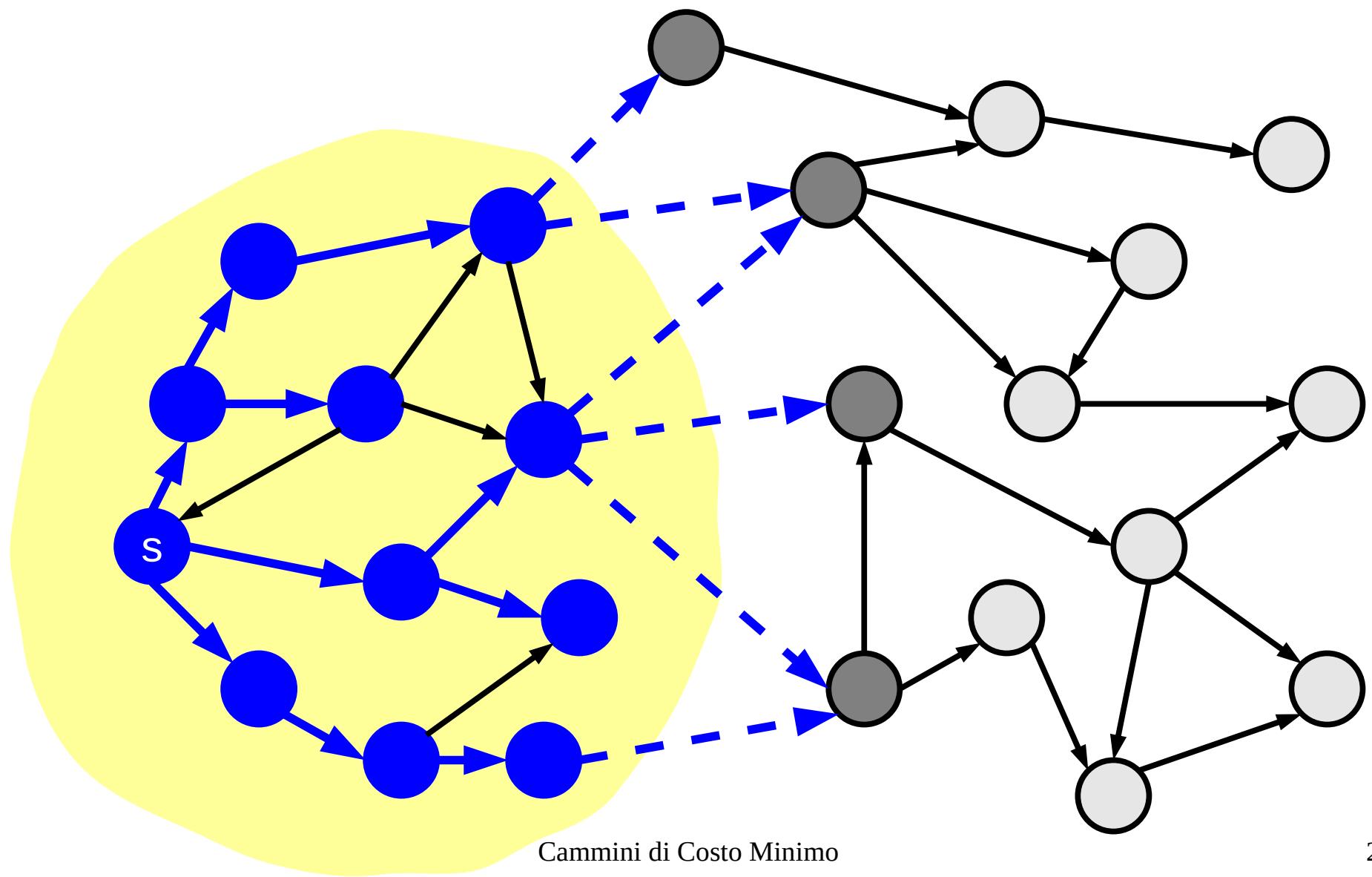
Edsger W. Dijkstra, (1930—2002)  
[http://en.wikipedia.org/wiki/Edsger\\_W.\\_Dijkstra](http://en.wikipedia.org/wiki/Edsger_W._Dijkstra)

# Lemma (Dijkstra)

- Sia  $G = (V, E)$  un grafo orientato con funzione costo  $w$ 
  - I costi degli archi devono essere  $\geq 0$ .
- Sia  $T$  una parte dell'albero dei cammini di costo minimo radicato in  $s$ 
  - $T$  rappresenta porzioni di cammini di costo minimo che partono da  $s$

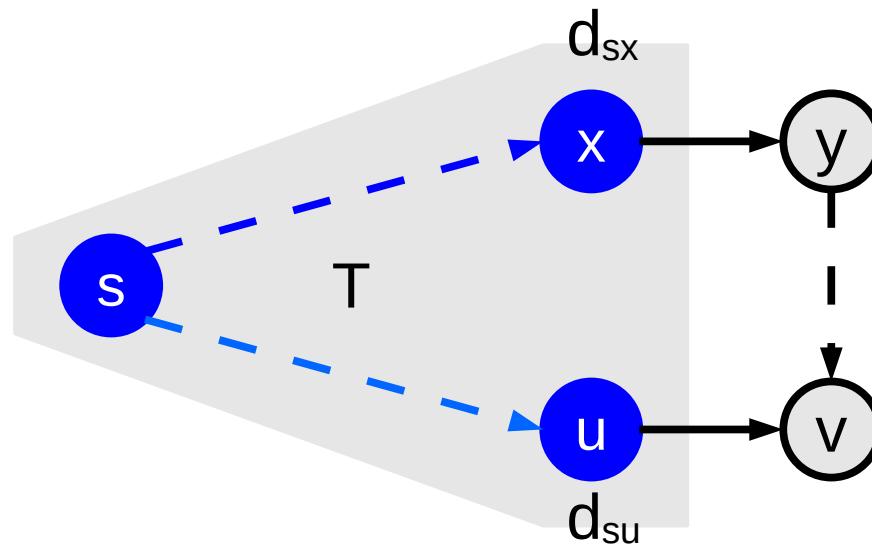
Allora l'arco  $(u, v)$  con  $u \in V(T)$  e  $v \notin V(T)$  che minimizza la quantità  $d_{su} + w(u, v)$  appartiene ad un cammino minimo da  $s$  a  $v$

# Lemma (Dijkstra)



# Dimostrazione

- Supponiamo per assurdo che  $(u,v)$  non appartenga ad un cammino di costo minimo tra  $s$  e  $v$ 
  - quindi  $d_{su} + w(u,v) > d_{sv}$
- Quindi deve esistere  $\pi_{sv}^*$  che porta da  $s$  in  $v$  senza passare per  $(u,v)$  con costo inferiore a  $d_{su} + w(u,v)$

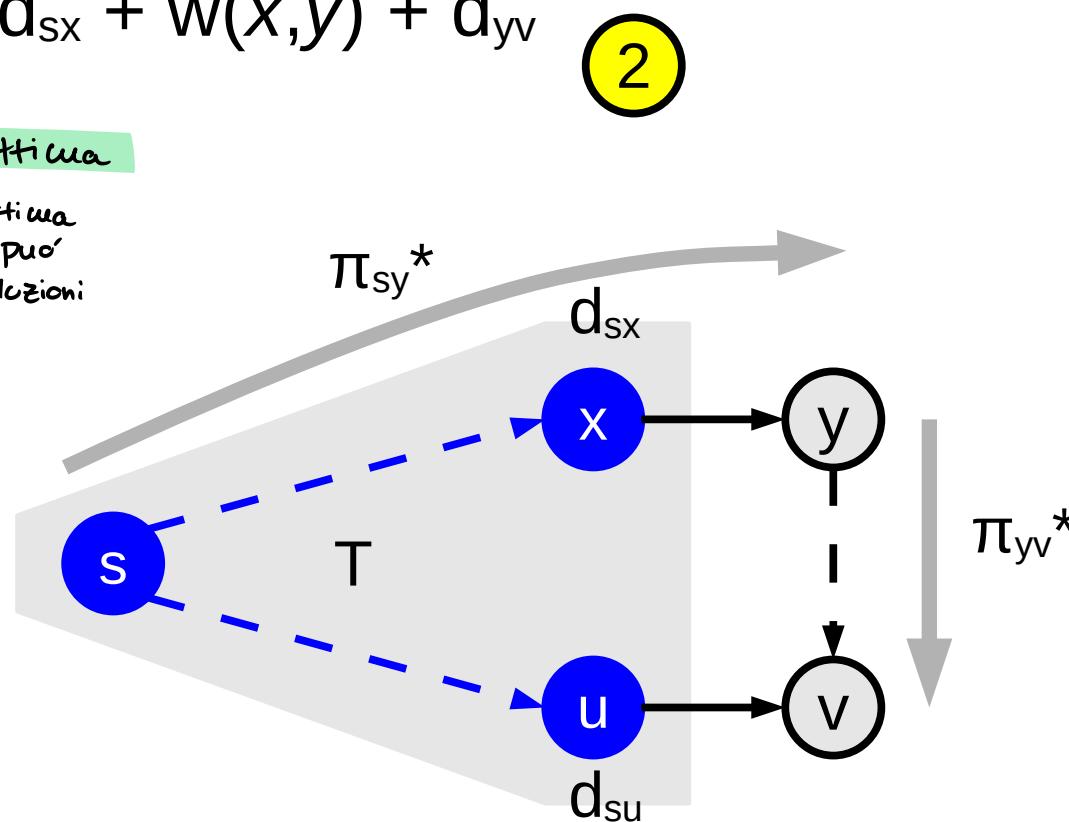


# Dimostrazione

- Per il teorema di sottostruttura ottima, il cammino  $\pi_{sv}^*$  si scomponе in  $\pi_{sy}^*$  e  $\pi_{yv}^*$
- Quindi  $d_{sv} = d_{sx} + w(x,y) + d_{yv}$

## Teorema di sottostruttura ottima

Un problema ha sottostruttura ottima se la soluzione ottima del problema puo' essere costruita a partire dalle soluzioni ottime dei suoi sottoproblemi.

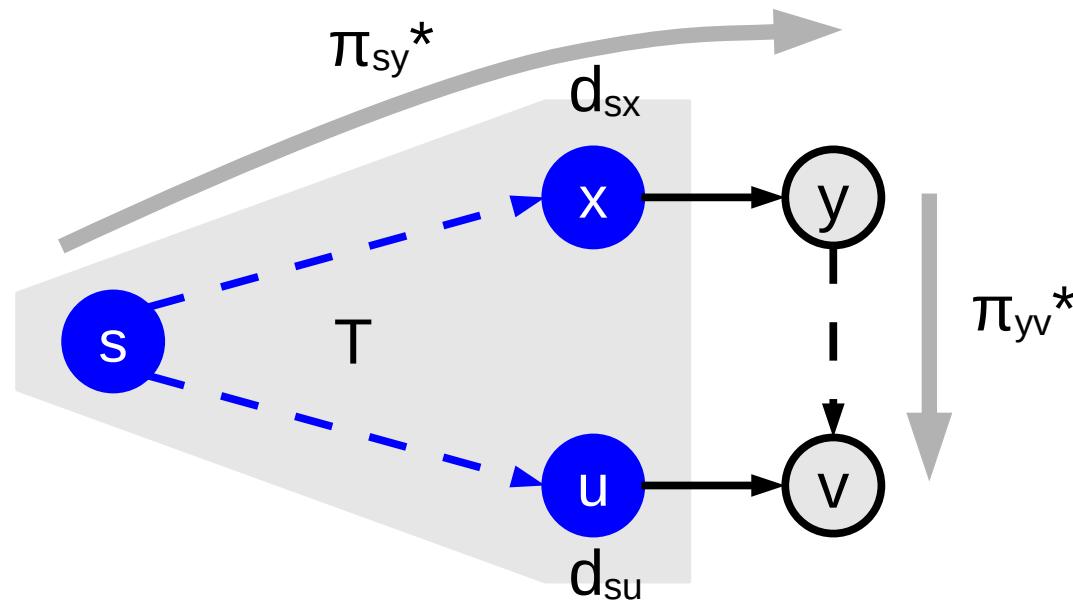


Cammini di Costo Minimo

# Dimostrazione

- Per ipotesi (lemma di Dijkstra), l'arco  $(u,v)$  è quello che, tra tutti gli archi che collegano un vertice in  $T$  con uno non ancora in  $T$ , minimizza la somma  $d_{su} + w(u,v)$
- In particolare:  $d_{su} + \underline{\underline{w(u,v)}} \leq d_{sx} + \underline{\underline{w(x,y)}}$

3

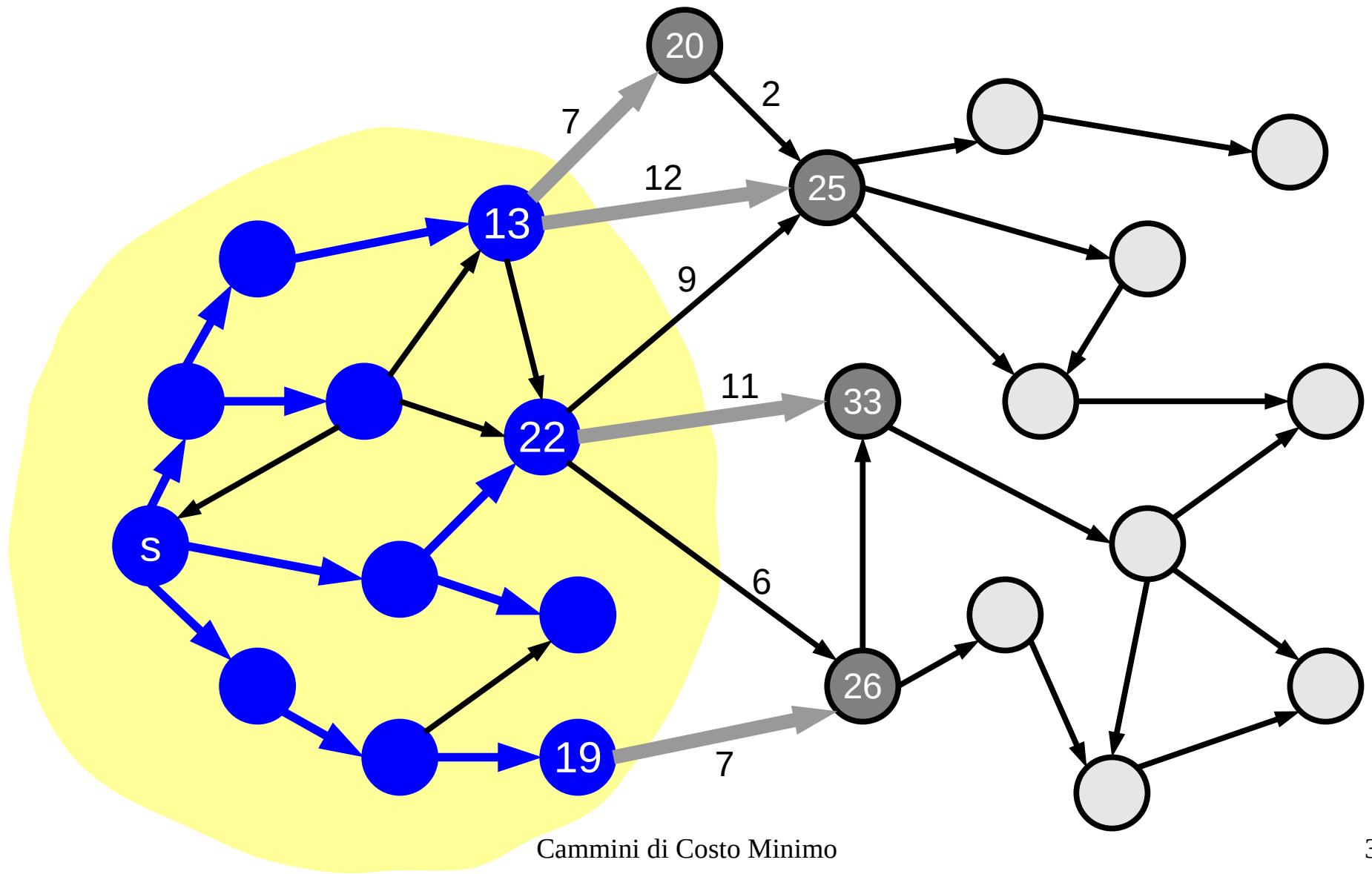


# Riassumiamo

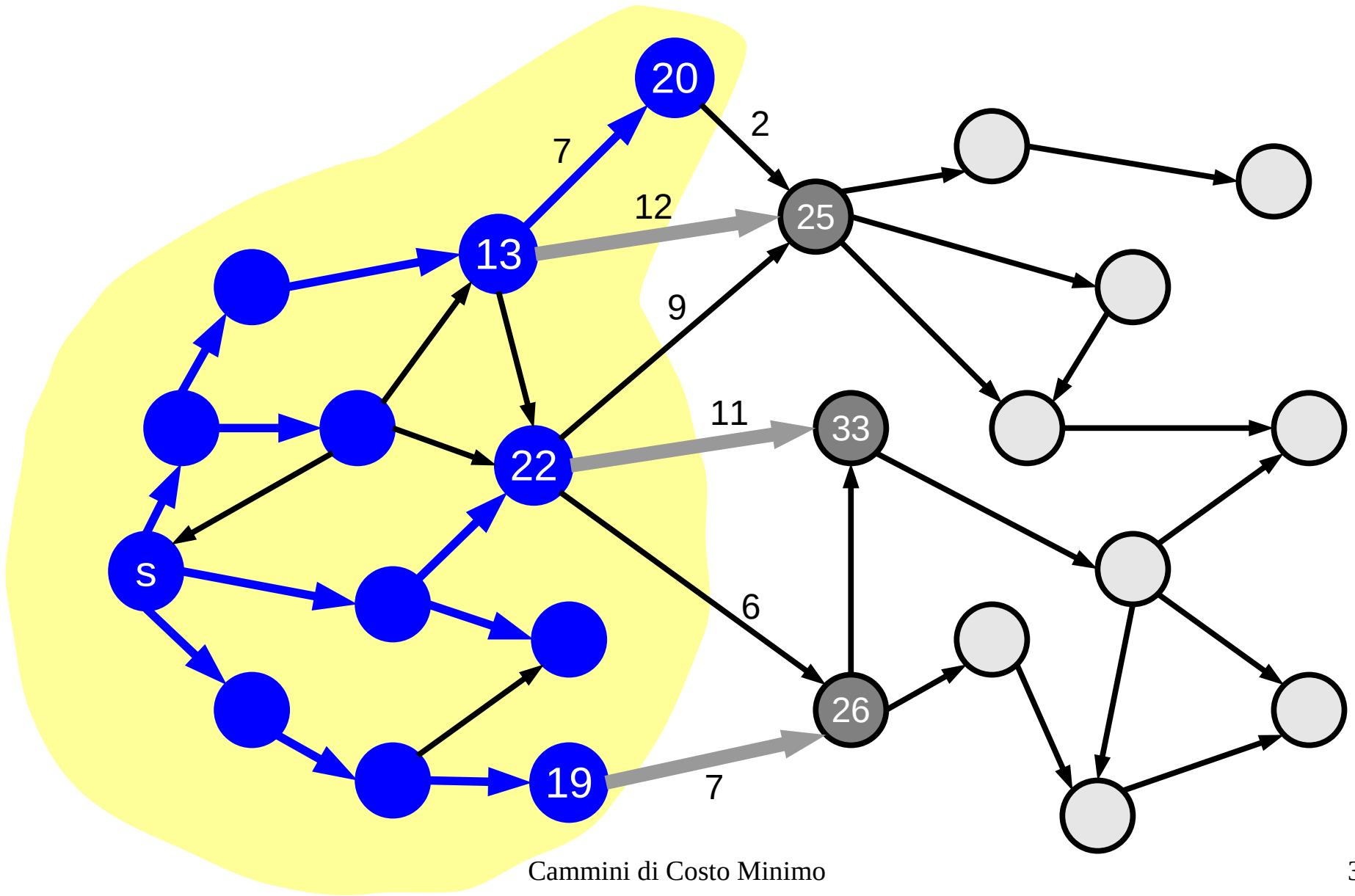
- Da (1) abbiamo  $d_{su} + w(u,v) > d_{sv}$
- Da (2) abbiamo  $d_{sv} = d_{sx} + w(x,y) + d_{yv}$
- Da (3) abbiamo  $d_{su} + w(u,v) \leq d_{sx} + w(x,y)$
- Combinando (1) (2) e (3) otteniamo

$$\begin{array}{lcl} d_{su} + w(u, v) & > & d_{sx} + w(x, y) + d_{yv} & \text{da (1) e (2)} \\ & \nearrow & & \\ & \geq & d_{sx} + w(x, y) & \\ & \geq & d_{su} + w(u, v) & \text{da (3)} \end{array}$$

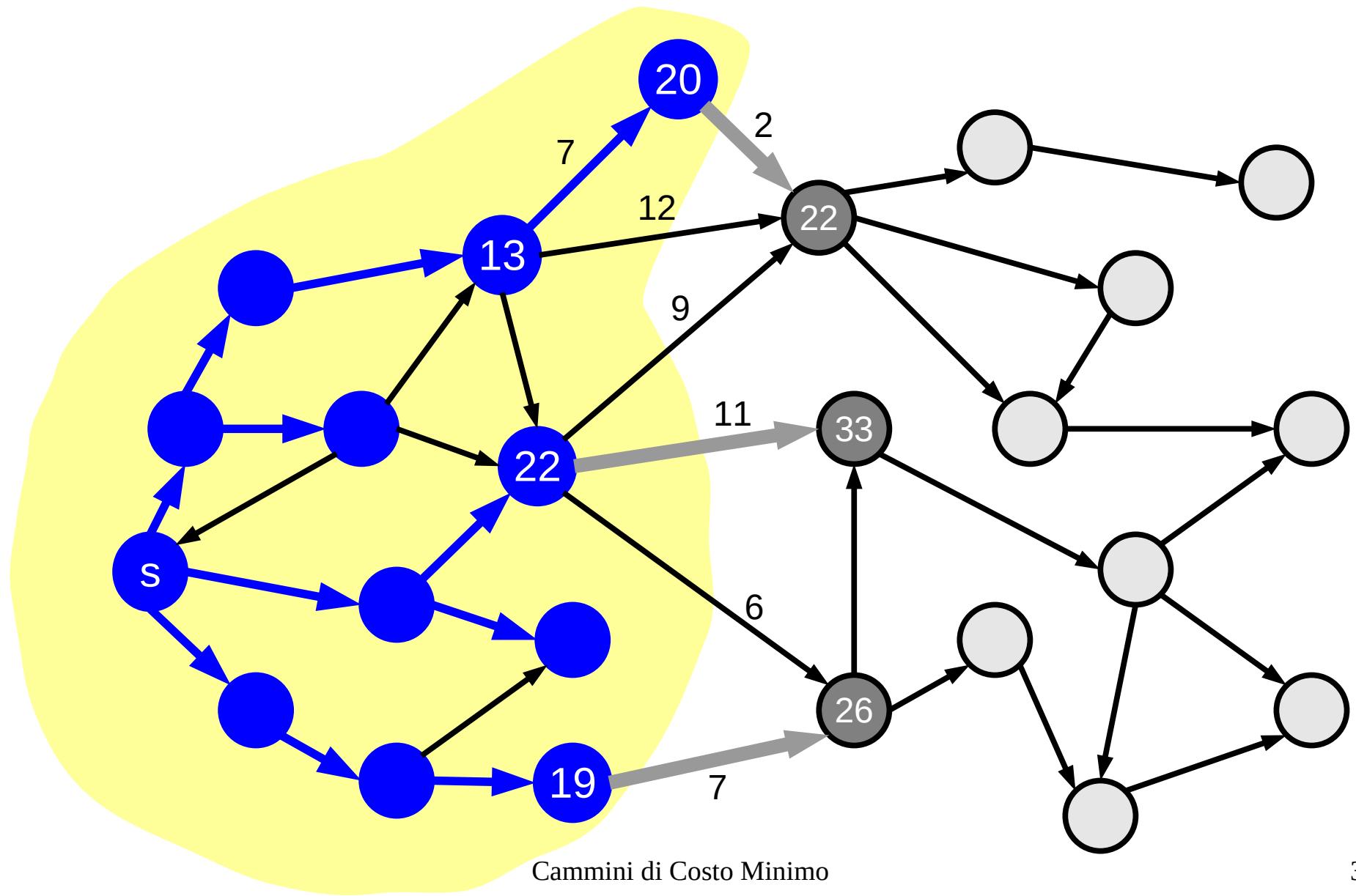
# Esempio



# Esempio



# Esempio



L'inizializzazione è identica a quella di Bellman-Ford

# Algoritmo di Dijkstra generico

```
double[1..n] DijkstraGenerico(Grafo G=(V,E,w), int s)
    int n ← G.numNodi();
    int pred[1..n], u, v;
    double D[1..n];
    for v ← 1 to n do
        D[v] ← +∞ ;
        pred[v] ← -1;
    endfor
    D[s] ← 0;           • Visitiemo un nodo per volta • T→insieme dei nodi non ancora visitati
    while (non ho visitato tutti i nodi raggiungibili da s) do
        Trova l'arco (u,v) incidente su T con D[u] + w(u,v) minimo
        D[v] ← D[u] + w(u,v);
        pred[v] ← u;
    endfor
    return D;
```

• Inizialmente è pieno  
• Sceglie il nodo 'u' in T  
che ha la stima di  
distanza D[u] cui incuba  
(rimuove 'u' da T)

python

Copy Download

```
double[1..n] DijkstraGenerico(Grafo G=(V,E,w), int s):
    # INIZIALIZZAZIONE: Identica a Bellman-Ford.
    int n = G.numNodi();
    int pred[1..n]; # Array per ricostruire i percorsi
    double D[1..n]; # Array delle distanze minime stimate

    for v = 1 to n do:
        D[v] = +∞; # All'inizio, nessun nodo è raggiungibile
        pred[v] = -1; # Nessun predecessore
    endfor

    D[s] = 0; # La distanza dalla sorgente a sé stessa è zero

    # Creiamo un insieme (che chiameremo "T") dei nodi non ancora visitati.
    # Inizialmente, T contiene tutti i nodi.

    # CICLO PRINCIPALE: Visiteremo un nodo per volta.
    while (T non è vuoto) do: # Finché ci sono nodi non visitati...

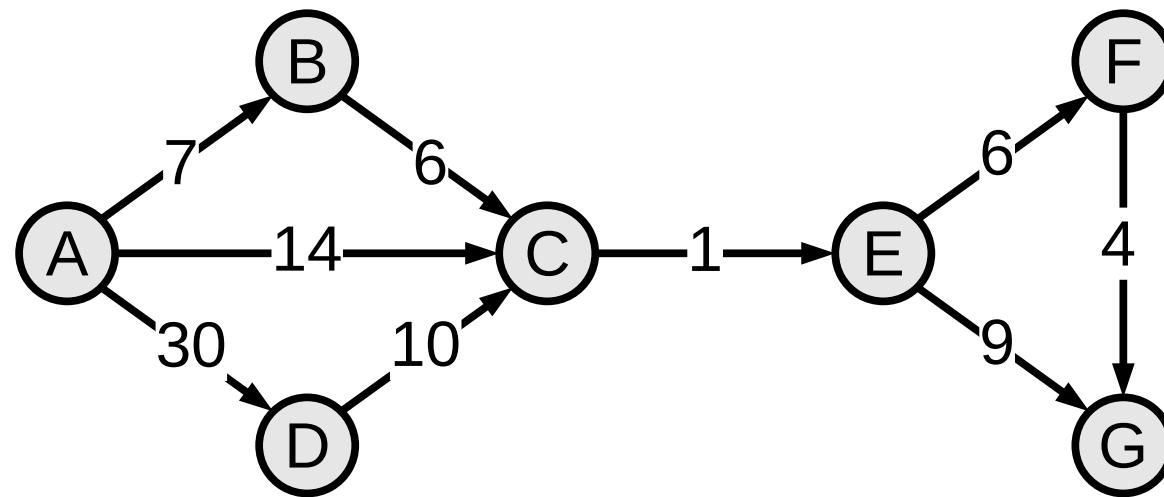
        # (PASSO CRUCIALE 1) Scegli il nodo 'u' in T che ha la stima di distanza D[u] MINIMA.
        u = nodo in T con D[u] minimo;

        # Rimuovi 'u' da T. A questo punto, D[u] è la distanza minima definitiva per 'u'.
        T.rimuovi(u);

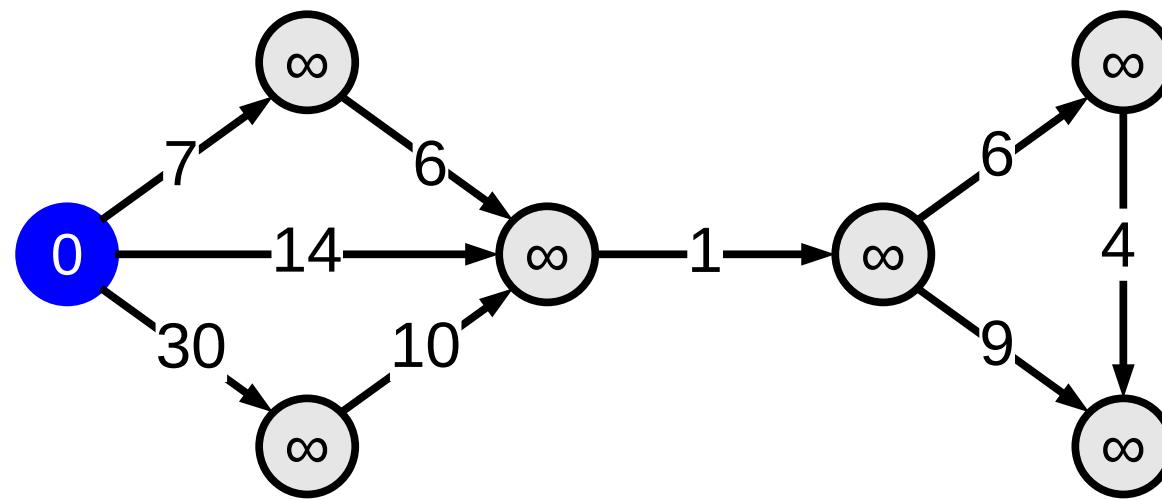
        # (PASSO CRUCIALE 2) Esamina tutti i nodi vicini a 'u' che non sono ancora stati visitati (cioè ancora in T).
        for each (u, v) in E tale che v è in T do:
            # Questo è il "rilassamento" dell'arco (u, v).
            if ( D[u] + w(u, v) < D[v] ) then:
                D[v] = D[u] + w(u, v); # Aggiorna la stima di distanza per v
                pred[v] = u;           # Segna 'u' come predecessore di 'v' sul cammino minimo
            endif
        endfor
    endwhile

    return D; # Restituisce l'array delle distanze minime definitive.
```

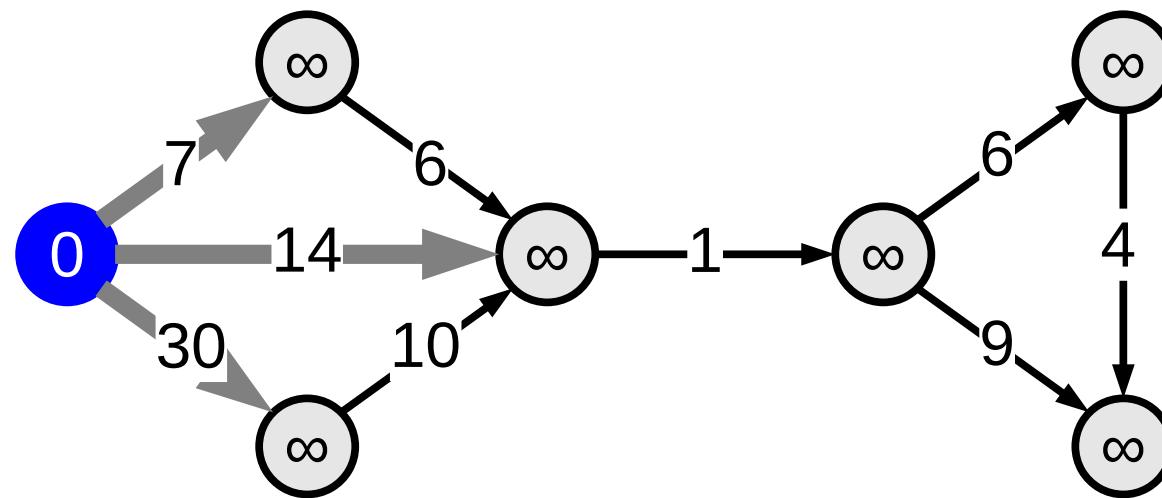
# Esempio



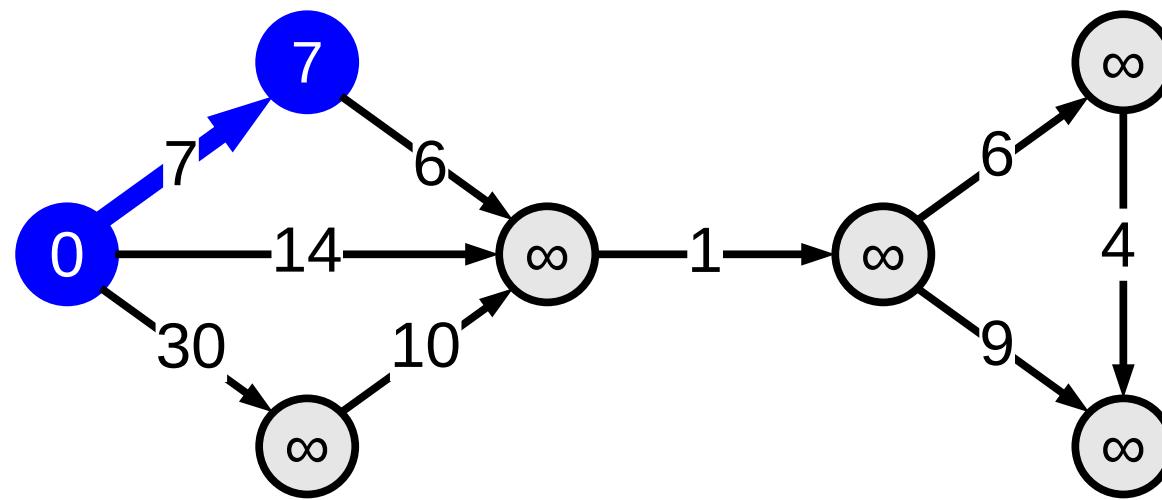
# Esempio



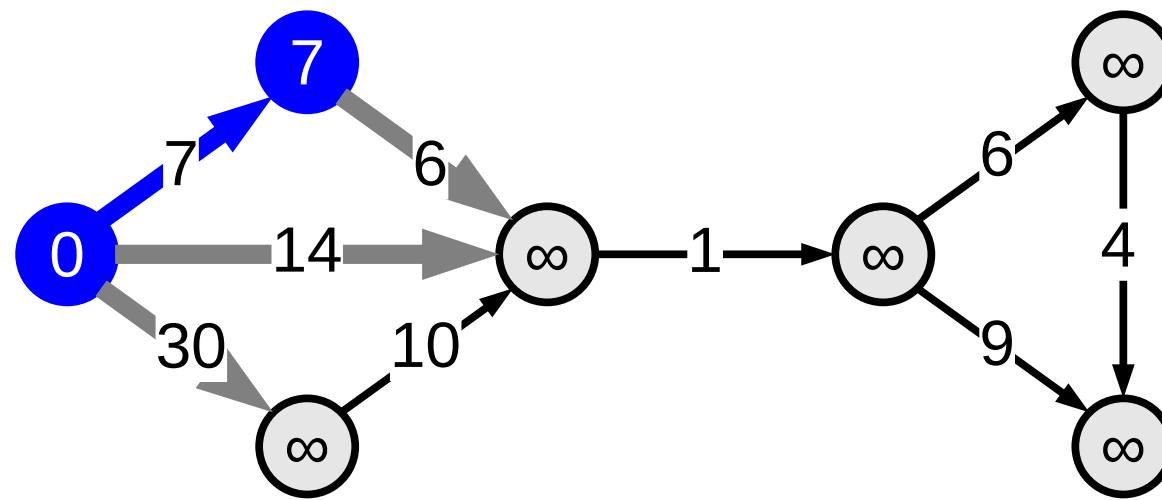
# Esempio



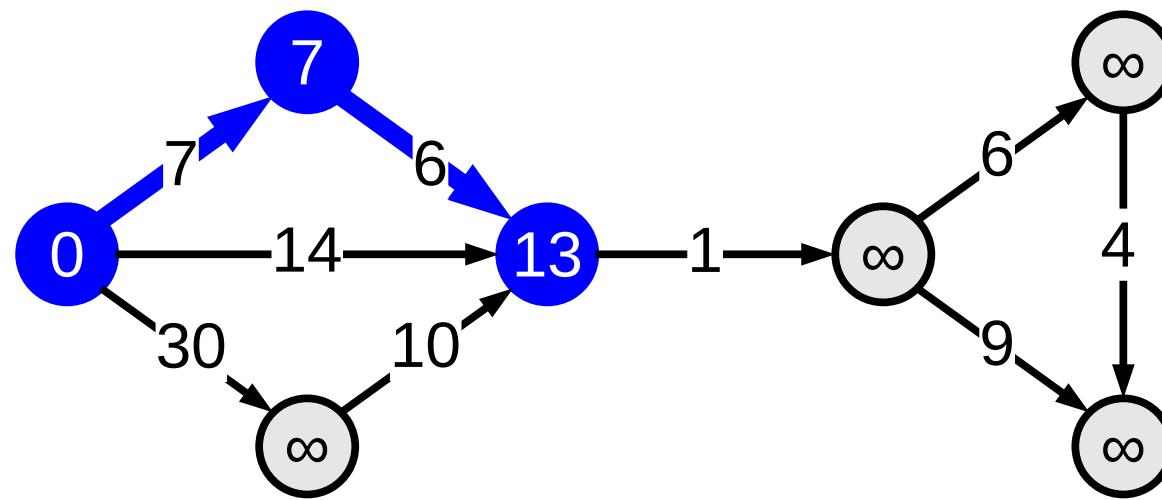
# Esempio



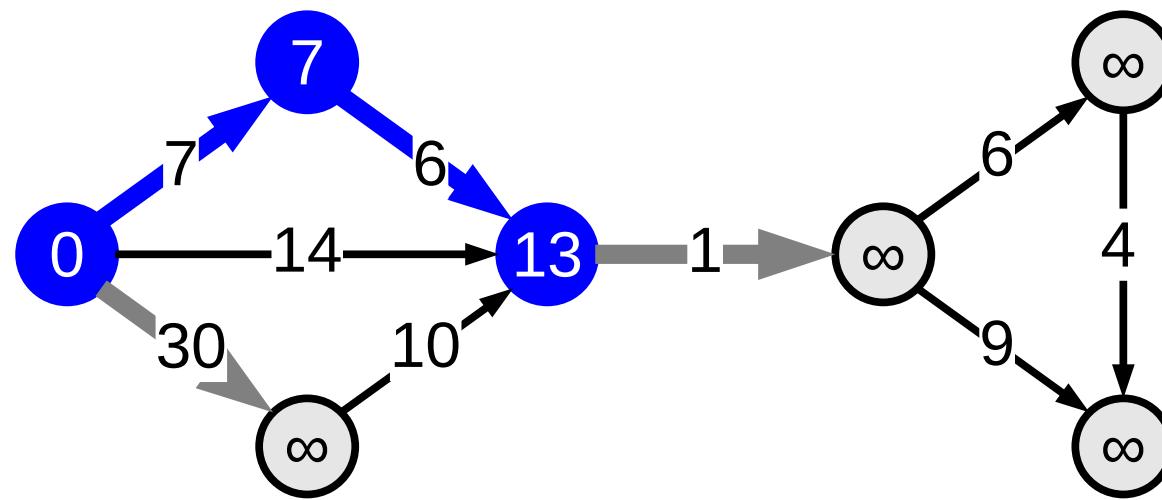
# Esempio



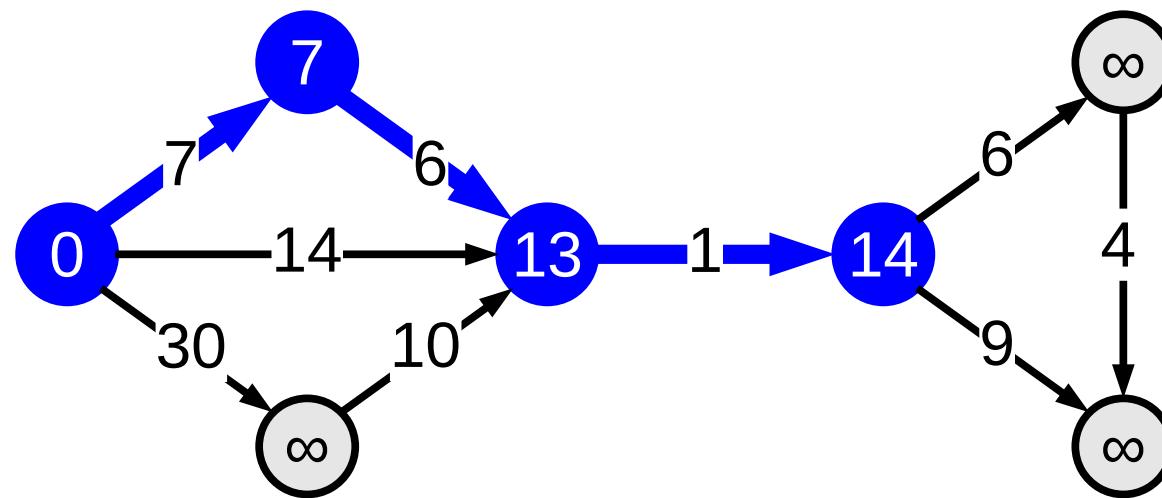
# Esempio



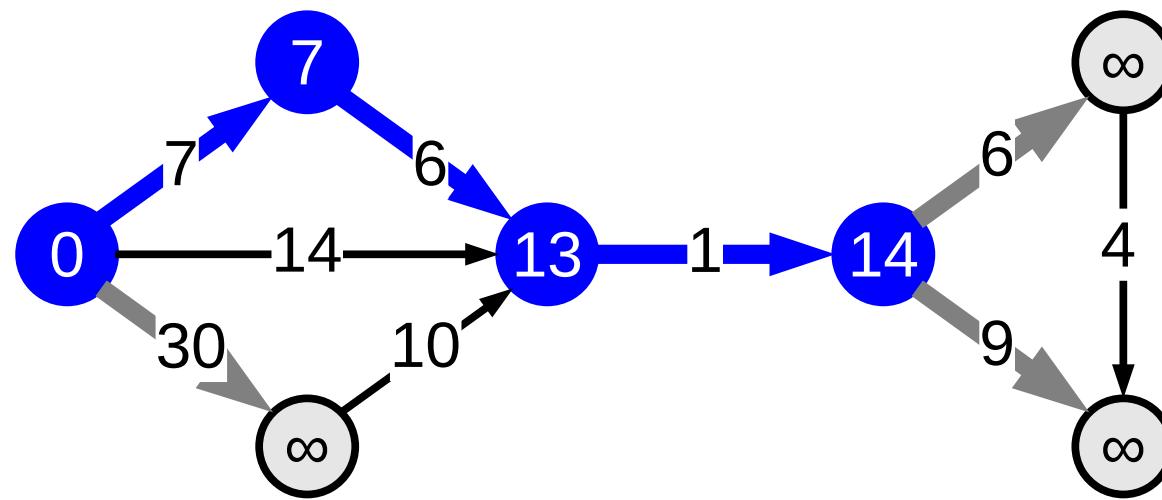
# Esempio



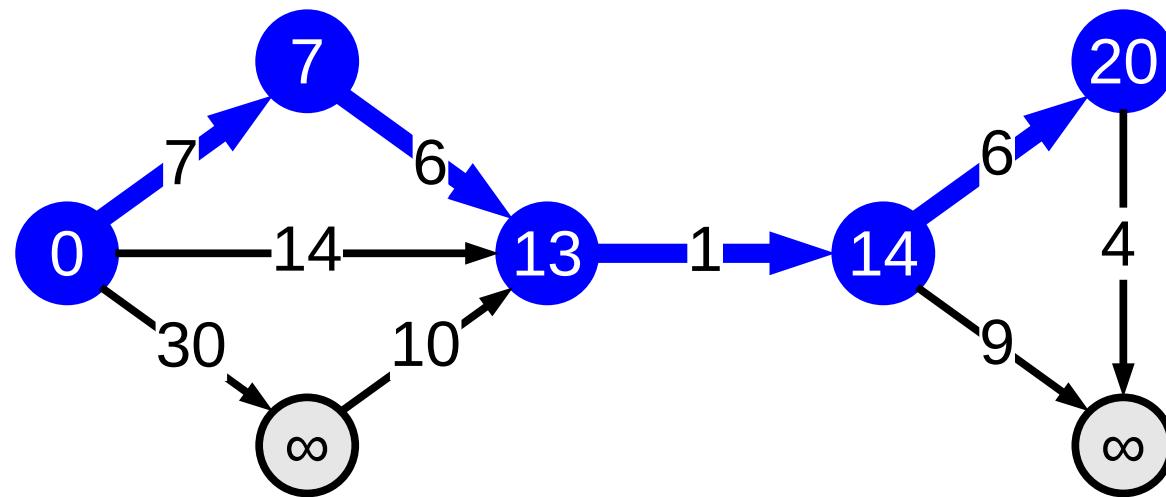
# Esempio



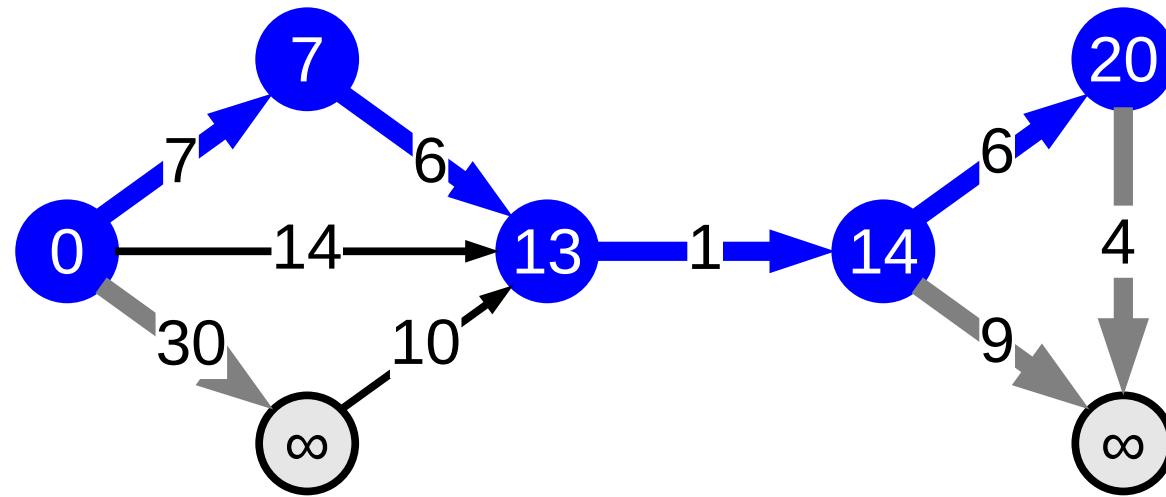
# Esempio



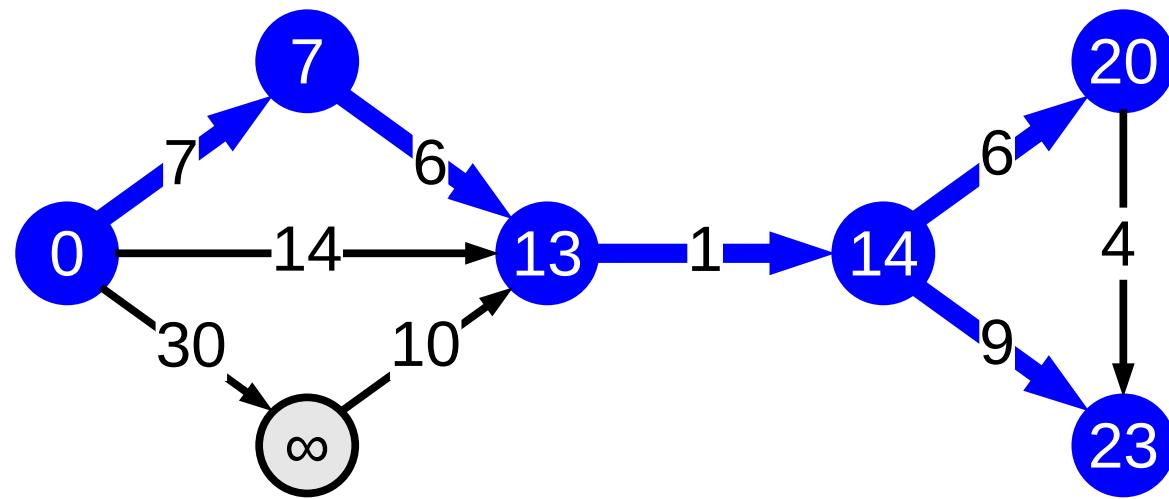
# Esempio



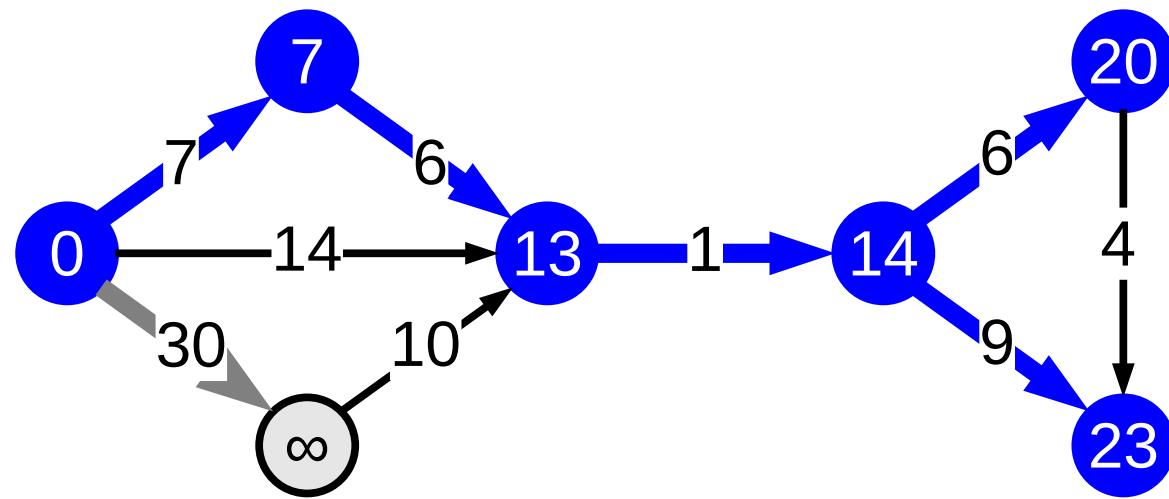
# Esempio



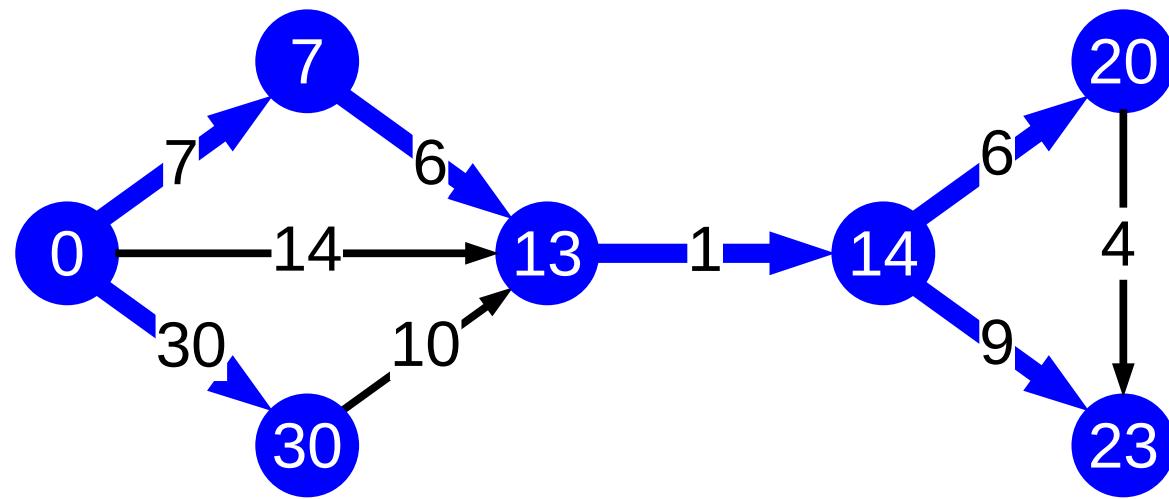
# Esempio



# Esempio



# Esempio



```

double[1..n] Dijkstra(Grafo G=(V,E,w), int s)
    int n ← G.numNodi();
    int pred[1..n], v, u;
    double D[1..n];
    boolean added[1..n];
    CodaPriorita<int, double> Q; • Min-Heap: la priorità di un nodo è la sua stima di
    INITIALIZATION distanza  $D(v)$ 
    for v ← 1 to n do
        pred[v] ← -1; added[v] ← false;
        if (v == s) D[v] ← 0 else D[v] ← +∞ endif
        Q.insert(v, d[v]);
    endfor
    while (not Q.isEmpty()) do
        u ← Q.find(); Estrae il minimo: Prende il nodo
        Q.deleteMin(); 'u' con la distanza minima della
        added[u] ← true; (distanza definitiva) Trova e rimuovi il nodo con
        for each v adiacente a u do distanza minima
            if (not added[v] and D[u] + w(u,v) < D[v]) then Controllo se il vicino 'v' non
                D[v] ← D[u] + w(u,v); è stato ancora processato
                Q.decreaseKey(v, D[v]); e se passare per 'u' offre
                pred[v] ← u; un percorso migliore per 'v'
            endif Somiglia all'algoritmo di Prim
        endfor (MST), ma priorità diversa
    endwhile
    return D;

```

nodo con  
priorità  
minima

Rilassa gli  
archi: Esclusa  
tutti i vicini di 'u'  
che non sono ancora  
definitivi

**Estrae il minimo:** Prende il nodo  
'u' con la distanza minima della  
coda → Questo nodo adesso ha la sua distanza minima definitiva

**Aggiorna la**  
**Stima:** Trovato  
un percorso  
migliore per 'v'

↳ Aggiorna la  
coda: La priorità  
di 'v' è diminuita

→ la funzione aggiorna la priorità di 'v' e ricondiziona l'heap

**Rendi  $D[u]+w(u,v)$  la nuova**  
**distanza di v da s**

## Importanza della coda di priorità

Il cuore di Dijkstra è questo passaggio ripetuto continuamente:

"Tra tutti i nodi non ancora processati, prendi quello con distanza minima"

**Senza Coda di priorità:** Per trovare questo nodo minimo si dovrebbe scansionare tutto l'array  $D[]$  delle distanze ogni volta. Questo ha un costo  $O(n)$  per nodo che si estrae. Dato che si estrae  $n$  nodi, il costo totale diventa  $O(n^2)$  (cattivo).

**Con coda di priorità:** La coda di priorità (min-heap) è una struttura dati progettata proprio per fare questo in coda efficiente, permettendo di:

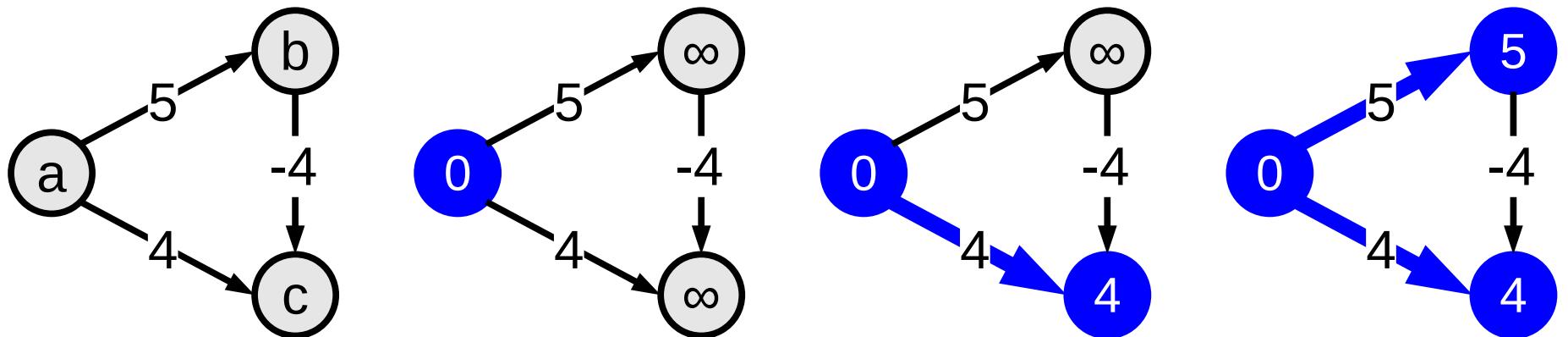
- Trovare il nodo con la priorità minima (distanza minima) in tempo  $O(1)$
- Ricavare (estrazione) quel nodo e riordinare la coda in tempo  $O(\log n)$

# Analisi dell'algoritmo di Dijkstra

- L'inizializzazione ha costo  $O(n)$
- **find()** ha costo  $O(1)$ ; **insert()** e **deleteMin()** hanno costo  $O(\log n)$ 
  - Sono eseguite al più  $n$  volte
  - Un nodo estratto dalla coda di priorità non viene più reinserito
- Le operazioni **insert()** e **decreaseKey()** hanno costo  $O(\log n)$ 
  - Sono eseguite al più  $m$  volte
  - Una volta per ogni arco
- Totale:  $O((n+m) \log n) = O(m \log n)$  se tutti i nodi sono raggiungibili dalla sorgente

# Osservazione

- Perché l'algoritmo di Dijkstra funzioni correttamente è essenziale che i pesi degli archi siano tutti  $\geq 0$
- Esempio di funzionamento errato

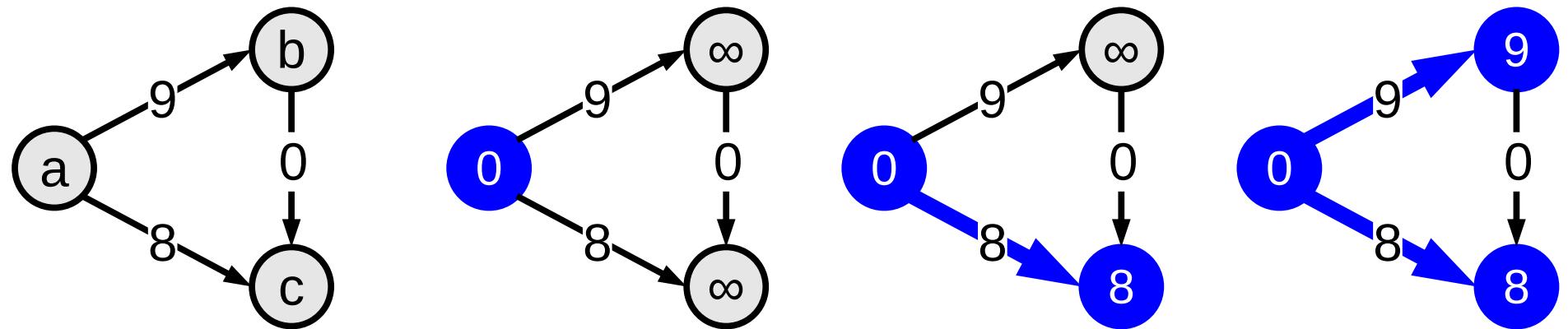


- Il cammino minimo da  $a \rightarrow c$  non è  $(a,c)$  ma  $(a,b,c)$  che ha costo 1

# Domanda

- Sia  $G = (V, E)$  un grafo orientato pesato, anche con pesi negativi
- Supponiamo che in  $G$  non esistano cicli negativi.
- Supponiamo di incrementare i pesi di tutti gli archi di una costante  $C$  in modo che tutti i pesi diventino non negativi.
- L'algoritmo di Dijkstra applicato ai nuovi pesi restituisce l'albero dei cammini minimi anche per i pesi originali?

# Risposta: NO



# Algoritmo di Floyd e Warshall

*all-pair shortest paths*

- Basato sulla programmazione dinamica
  - Si può applicare a grafi orientati con costi arbitrari (anche negativi), purché non ci siano cicli negativi
- Sia  $V = \{1, 2, \dots, n\}$
- Sia  $D_{uv}^k$  la distanza minima dal nodo  $u$  al nodo  $v$ , nell'ipotesi in cui gli eventuali nodi intermedi possano appartenere esclusivamente all'insieme  $\{1, \dots, k\}$
- La soluzione al nostro problema è  $D_{uv}^n$  per ogni coppia di nodi  $u$  e  $v$

Floyd-Warshall calcola in un colpo solo la distanza minima tra ogni coppia di nodi  $(u, v)$  del grafo. Molto utile quando bisogna conoscere le distanze tra molti punti

### Idea Fondamentale

L'algoritmo si basa su un'idea di programmazione dinamica. L'osservazione chiave è: "Il cammino minimo da  $u$  a  $v$  può essere migliorato se percorso al percorso di passare attraverso un nodo intermedio  $k$ ".

L'algoritmo costruisce progressivamente la soluzione, considerando un nodo alla volta come potenziale nodo intermedio.

### La Definizione Ricorsiva

Sia  $D[k][u][v]$  la lunghezza del cammino minimo dal nodo  $u$  al nodo  $v$  quando siamo autorizzati a utilizzare solo i nodi solo i nodi  $\{1, 2, \dots, k\}$  come nodi intermedi.

La genialità dell'algoritmo è nella relazione di ricorrenza che lega  $D[k][u][v]$  a  $D[k-1][u][v]$ :

$$D[k][u][v] = \min(D[k-1][u][v], D[k-1][u][k] + D[k-1][k][v])$$

Spiegazione:

$D[k-1][u][v] \rightarrow$  è la migliore distanza conosciuta da  $u$  a  $v$  senza usare  $k$  come intermedio (usando solo i nodi da 1 a  $k-1$ )

$D[k-1][u][k] + D[k-1][k][v] \rightarrow$  è la distanza da  $u$  a  $k$  (senza usare  $k$  come intermedio) più la distanza da  $k$  a  $v$  (senza usare  $k$  come intermedio). In pratica, è la distanza del percorso che passa attraverso  $k$ .

L'algoritmo sceglie il minimo tra queste due opzioni. Se passare per il nuovo nodo  $k$  offre un percorso più breve, aggiorna la sua stessa

# Inizializzazione

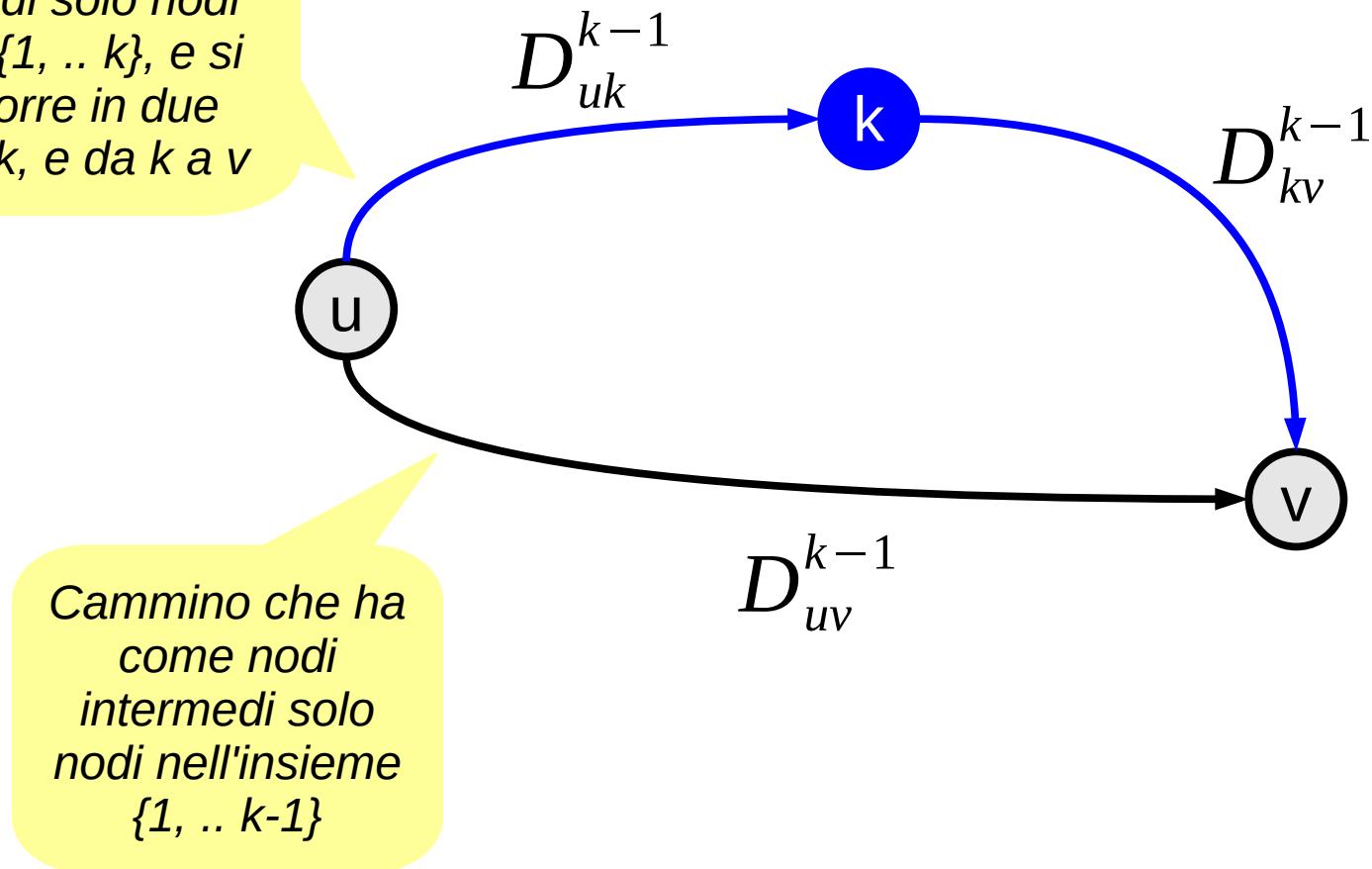
- $D_{uv}^0$  è la distanza minima tra  $u$  e  $v$  nell'ipotesi di non poter passare per alcun nodo intermedio
- Posso calcolare  $D_{uv}^0$  come

$E \rightarrow$  insieme degli archi

$$D_{uv}^0 = \begin{cases} 0 & \text{se } u=v \\ w(u,v) & \text{se } (u,v) \in E \\ \infty & \text{se } (u,v) \notin E \end{cases} \quad (\text{ciclo})$$

# Caso generale

*Il cammino blu ha come nodi intermedi solo nodi nell'insieme  $\{1, \dots k\}$ , e si può scomporre in due parti: da  $u$  a  $k$ , e da  $k$  a  $v$*



# Caso generale

- Per andare da  $u$  a  $v$  usando solo nodi intermedi in  $\{1, \dots, k\}$  ho due possibilità
  - Non passo per il nodo  $k$ . La distanza in tal caso è  $D_{uv}^{k-1}$  ✓
  - Passo per il nodo  $k$ . Per la proprietà di sottostruttura ottima, la distanza in tal caso è  $D_{uk}^{k-1} + D_{kv}^{k-1}$
- Quindi

$$D_{uv}^k = \min \left\{ D_{uv}^{k-1}, D_{uk}^{k-1} + D_{kv}^{k-1} \right\}$$

# Algoritmo di Floyd e Warshall

```

double[1..n,1..n] FloydWarshall( G=(V,E,w) )
    int n ← G.numNodi();
    double D[1..n, 1..n, 0..n]; int u, v, k;
    for u ← 1 to n do
        for v ← 1 to n do
            if (u == v) then D[u,v,0] ← 0;
            elseif ((u,v) ∈ E) then D[u,v,0] ← w(u,v); → se c'è un arco diretto usa il peso
            else D[u,v,0] ← + ∞;
            endif
        endfor
    endfor
    for k ← 1 to n do → permette al nodo 'k' di essere un intermedio
        for u ← 1 to n do
            for v ← 1 to n do
                D[u,v,k] ← D[u,v,k-1]; → inizialmente, si assume che il migliore percorso non usi
                if (D[u,v,k-1] + D[k,v,k-1] < D[u,v,k-1]) then → 'k' come intermedio
                    D[u,v,k] ← D[u,k,k-1] + D[k,v,k-1];
                endif
            endfor
        endfor
    endfor
    // eventuale controllo per cicli negativi (vedi seguito)
    return D[1..n, 1..n, n];

```

matrice 3D  
non 2D

Versione 2D

$$D[u][v] = \min(D[u][v], D[u][k] + D[k][v])$$

Versione 3D

$$D[u][v][k] = \min(D[u,v,k-1], D[u,k,k-1] + D[k,v,k-1])$$

$$D_{uv}^k = \min \{ D_{uv}^{k-1}, D_{uk}^{k-1} + D_{kv}^{k-1} \}$$

Fase  
iterativa

→ permette al nodo 'k' di essere un intermedio

inizialmente, si assume che il migliore percorso non usi  
'k' come intermedio

→ verifica se passare per  
'k' migliora il percorso  
• Percorso candidato è da  
'u' a 'k' (usando intermedio ≤ k-1)  
+ da 'k' a 'v' (usando intermedio ≤ k-1)

↓  
se sì  
aggiorna la  
distanza minima  
del percorso u→v

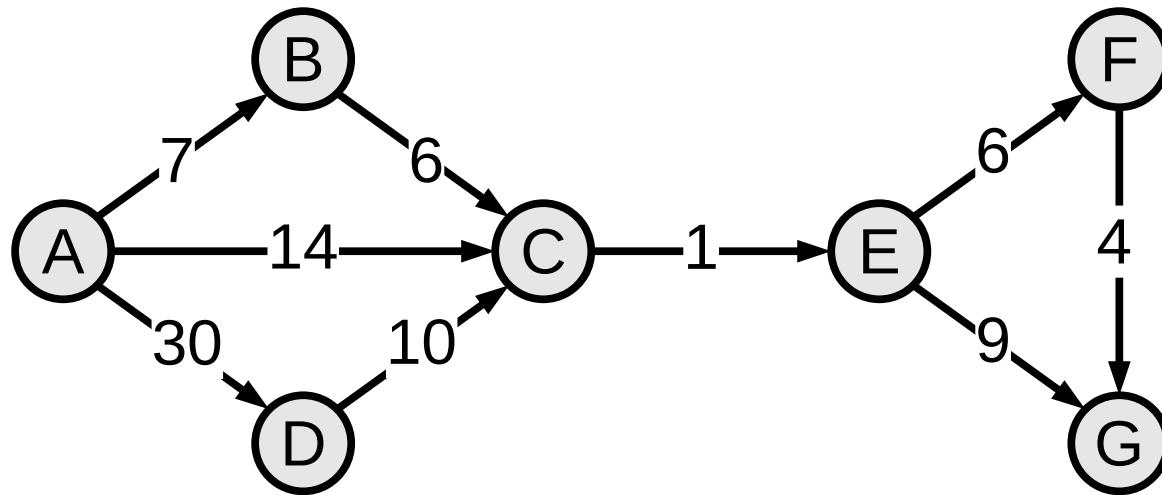
- Costo: tempo  $O(n^3)$ , spazio  $O(n^3)$  → 3 cicli for annidati  
Cammini di Costo Minimo

# Individuare cicli negativi

- L'algoritmo di Floyd e Warshall funziona anche se sono presenti archi di peso negativo
- Al termine dell'algoritmo, se  $D[u, u, n] < 0$  per qualche  $u$ , allora il nodo  $u$  fa parte di un ciclo negativo

```
// eventuale controllo per cicli negativi
for u  $\leftarrow$  1 to n do
    if ( D[u,u,n] < 0 ) then
        error "Il grafo contiene cicli negativi"
    endif
endfor
```

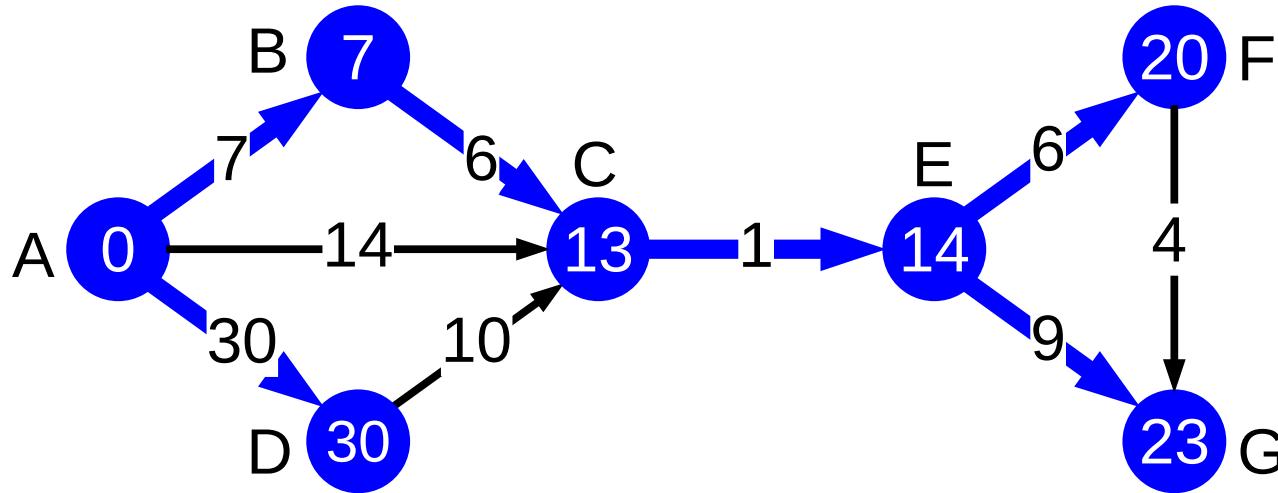
# Esempio



$D[u, v, \theta] =$

	<b>A</b>	<b>B</b>	<b>C</b>	<b>D</b>	<b>E</b>	<b>F</b>	<b>G</b>
<b>A</b>	0	7	14	30	Inf	Inf	Inf
<b>B</b>	Inf	0	6	Inf	Inf	Inf	Inf
<b>C</b>	Inf	Inf	0	Inf	1	Inf	Inf
<b>D</b>	Inf	Inf	10	0	Inf	Inf	Inf
<b>E</b>	Inf	Inf	Inf	Inf	0	6	9
<b>F</b>	Inf	Inf	Inf	Inf	Inf	0	4
<b>G</b>	Inf	Inf	Inf	Inf	Inf	Inf	0

# Esempio



$D[u, v, n] =$

	<b>A</b>	<b>B</b>	<b>C</b>	<b>D</b>	<b>E</b>	<b>F</b>	<b>G</b>
<b>A</b>	0	7	13	30	14	20	23
<b>B</b>	Inf	0	6	Inf	7	13	16
<b>C</b>	Inf	Inf	0	Inf	1	7	10
<b>D</b>	Inf	Inf	10	0	11	17	20
<b>E</b>	Inf	Inf	Inf	Inf	0	6	9
<b>F</b>	Inf	Inf	Inf	Inf	Inf	0	4
<b>G</b>	Inf	Inf	Inf	Inf	Inf	Inf	0

# Ottimizzazione

- Si può dimostrare che l'algoritmo di Floyd e Warshall funziona correttamente anche usando una matrice bidimensionale  $D[u, v]$  di  $n \times n$  elementi
- Per ricostruire i cammini di costo minimo possiamo usare una matrice dei successori  $next[u, v]$  di  $n \times n$  elementi
  - $next[u, v]$  è l'indice del secondo nodo attraversato dal cammino di costo minimo che va da  $u$  a  $v$  (il primo nodo di tale cammino è  $u$ , l'ultimo è  $v$ )

Matrice Next → La matrice next permette di non solo sapere quanto è lungo il cammino minimo, ma anche qual è il cammino minimo

```

double[1..n,1..n] FloydWarshall2( G=(V,E,w) )
    int n ← G.numNodi();
    double D[1..n, 1..n];
    int u, v, k, next[1..n, 1..n];
    for u ← 1 to n do
        for v ← 1 to n do
            if (u == v) then
                D[u,v] ← 0;
                next[u,v] ← -1; → fine del percorso (nessun successore)
            elseif ((u,v) ∈ E) then
                D[u,v] ← w(u,v);
                next[u,v] ← v; → il prossimo nodo da v per andare a v' e' v (percorso diretto)
            else
                D[u,v] ← + ∞;      ] Nessun cammino diretto noto
                next[u,v] ← -1;    • Nessun successore
            endif
        endfor
    endfor
    for k ← 1 to n do
        for u ← 1 to n do
            for v ← 1 to n do
                if (D[u,k] + D[k,v] < D[u,v]) then
                    D[u,v] ← D[u,k] + D[k,v]; → Aggiorna la distanza minima
                    next[u,v] ← next[u,k]; → Aggiorna il percorso: da u va prima verso k
                endif
            endfor
        endfor
    endfor
endfor
return D;

```

izazione

*Verifica se il percorso u->k è migliore del percorso diretto u-k*

*Matrice next()*

- Se  $u==v$ ,  $\text{next}[u,v]=-1$  (sei arrivato)
- Se  $\text{next}[u,v]=-1$  con  $u \neq v$  (nessun prece-

*Il primo passo per andare da u a v è lo stesso primo passo che garantisce per andare da u a k*

Verifica se il percorso  $u \rightarrow k \rightarrow v$  è migliore del percorso diretto  $u \rightarrow v$

### La distanza minima

caso: da una prima verso 4

Il Primo passo per andare da u a v  
e' lo stesso primo passo che garesti per andare da u a v

Matrice next()

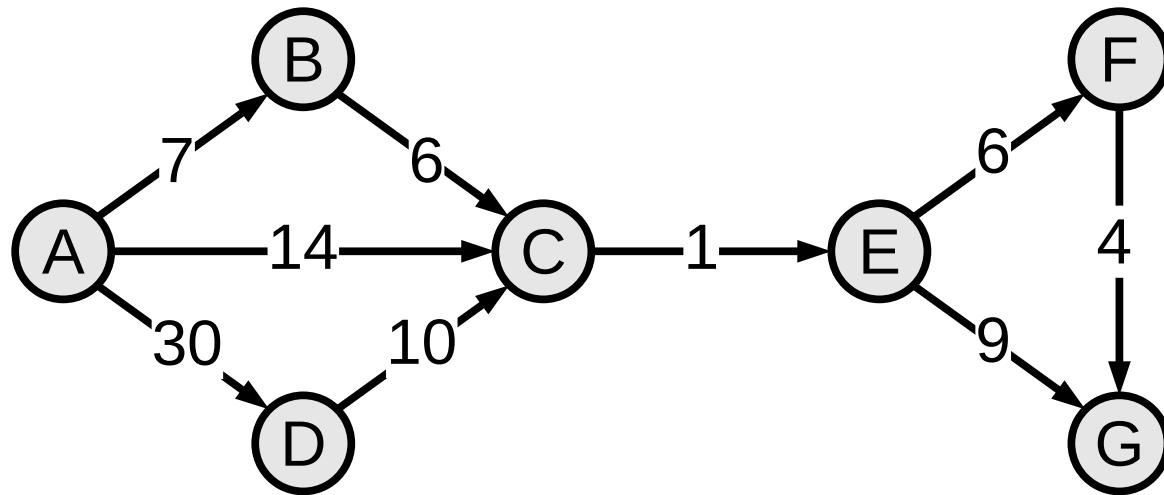
- Se  $u == v$ ,  $\text{next}[u, v] = -1$  (sei arrivato a destinazione)
  - Se  $\text{next}[u, v] = -1$  con  $u \neq v$  (nessun percorso noto)

# Stampa dei cammini

- Al termine dell'algoritmo di Floyd e Warshall, la procedura seguente stampa i nodi del cammino di costo minimo che va dal nodo  $u$  al nodo  $v$  in ordine di attraversamento

```
PrintPath( int u, int v, int next[1..n, 1..n] )
  if ( u != v and next[u,v] < 0 ) then
    errore "u e v non sono connessi";
  else
    print u;
    while ( u != v ) do
      u ← next[u,v];
      print u;
    endwhile;
  endif
```

# Esempio



`next[u, v] =`

	<b>A</b>	<b>B</b>	<b>C</b>	<b>D</b>	<b>E</b>	<b>F</b>	<b>G</b>
<b>A</b>	-1	B	B	D	B	B	B
<b>B</b>	-1	-1	C	-1	C	C	C
<b>C</b>	-1	-1	-1	-1	E	E	E
<b>D</b>	-1	-1	C	-1	C	C	C
<b>E</b>	-1	-1	-1	-1	-1	F	G
<b>F</b>	-1	-1	-1	-1	-1	-1	G
<b>G</b>	-1	-1	-1	-1	-1	-1	-1

Per rendere la matrice più comprensibile abbiamo usato i nomi dei nodi anziché gli indici