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

Definizione del problema

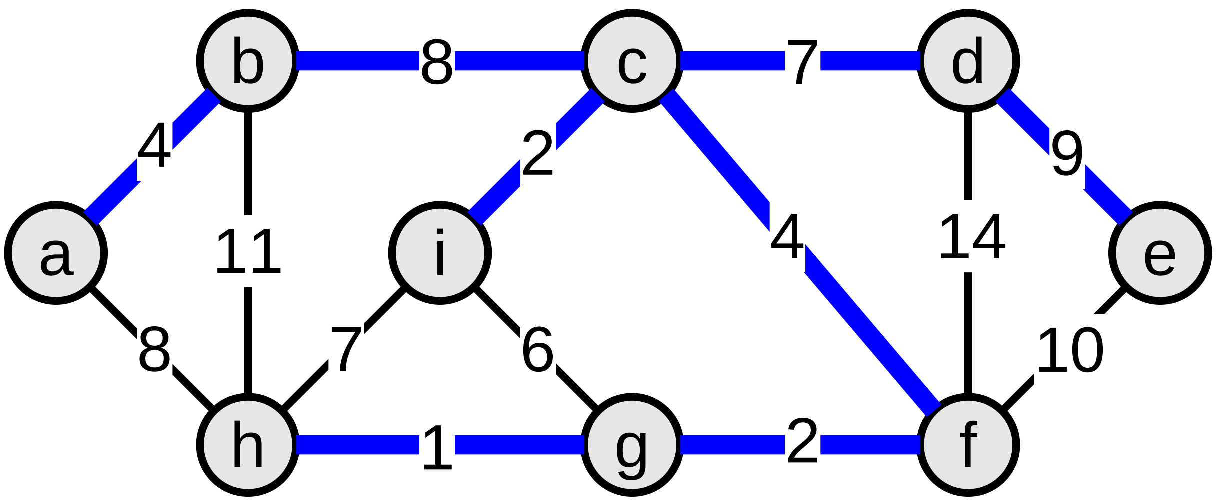
- Consideriamo un grafo orientato $G = (V, E)$ in cui ad ogni arco $(u, v) \in E$ sia associato un costo $w(u, v)$
- Il costo di un cammino $\pi = (v_0, v_1, \ldots v_k)$ che collega il nodo v_0 con v_k è definito come

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

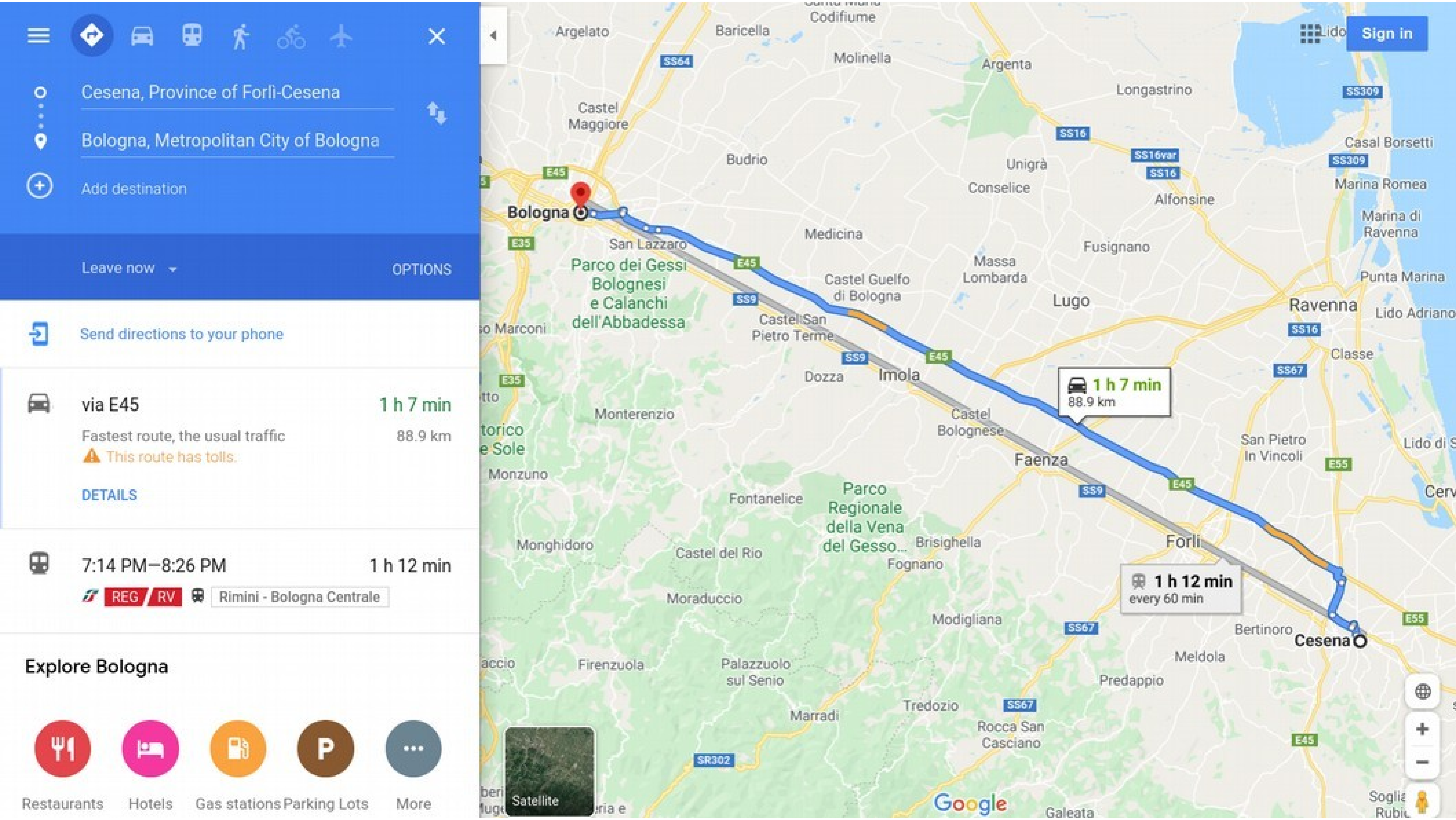
- Data una coppia di nodi v_0 e v_k, vogliamo trovare (se esiste) il cammino $\pi_{v0\,vk}^*$ di costo minimo tra tutti i cammini che vanno da v_0 a v_k

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



Applicazioni



Applicazioni (oops!)

Cammino più breve tra Haugesund e Trondheim secondo Microsoft Autoroute

Nel 2005

Nel 2010

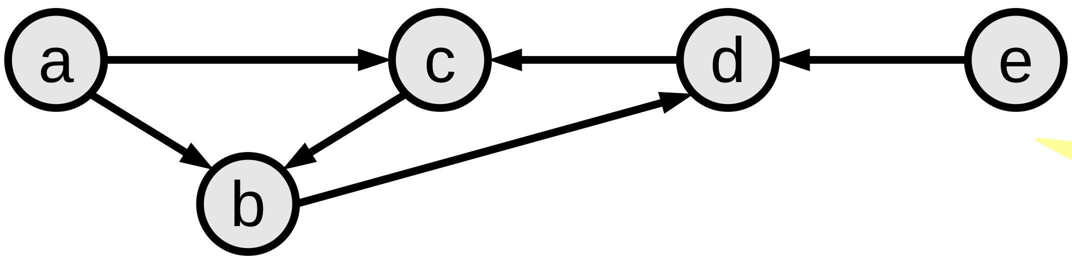
Diverse formulazioni del problema

1. Cammino di costo minimo fra una singola coppia di nodi *u* e *v*
   - Determinare, se esiste, il cammino di costo minimo π_{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

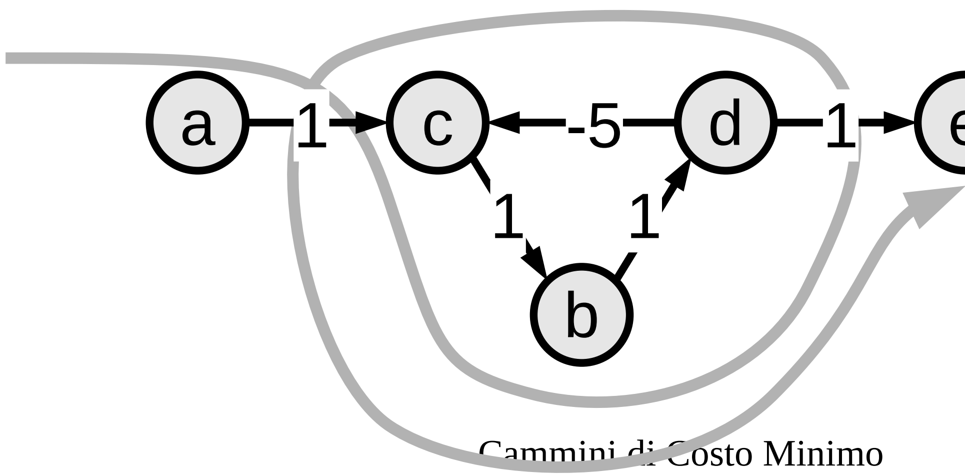
Osservazione

- 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***

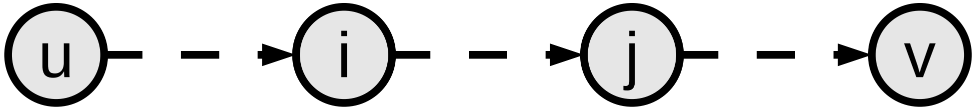
  - Quando ci sono cicli di costo negativo



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

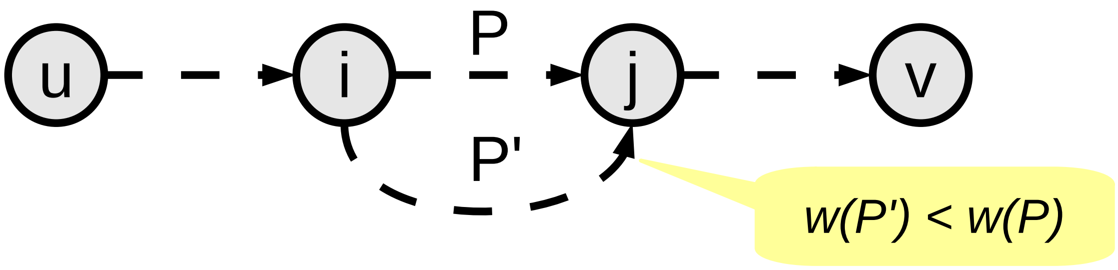
Proprietà (sottostruttura ottima)

- 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 π_{uv}^* da u a v
  - Siano i e j due nodi intermedi
  - Dimostriamo che il sotto-cammino di π_{uv}^* che collega i e j è un cammino minimo tra i e j

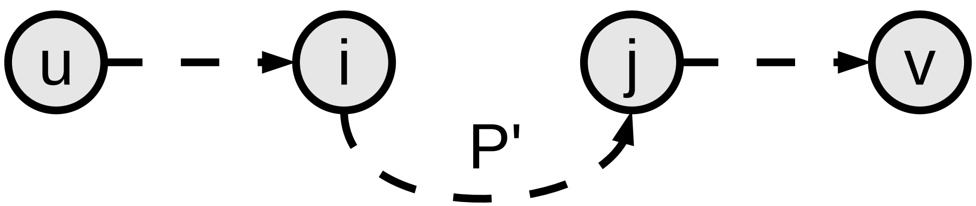


Proprietà
(sottostruttura ottima)

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

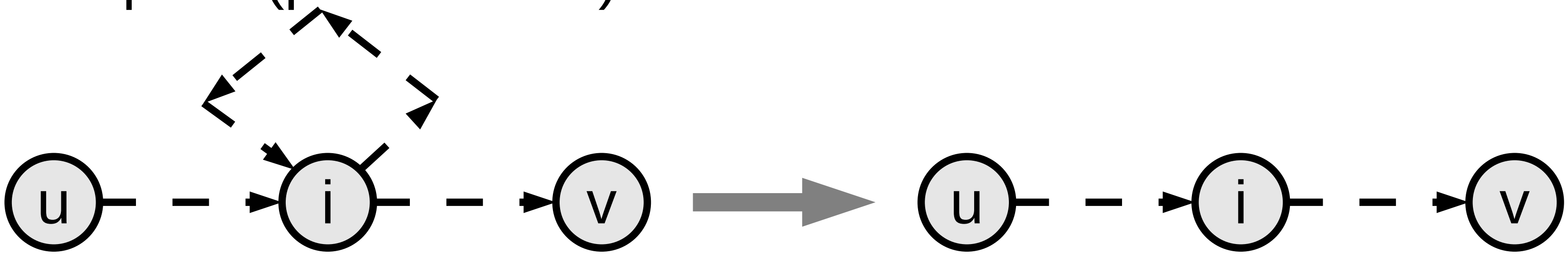


- Ma allora potremmo costruire un cammino tra *u* e *v* di costo inferiore a π_{uv}*, il che è assurdo perché avevamo fatto l'ipotesi che π_{uv}* fosse il cammino di costo minimo



Esistenza

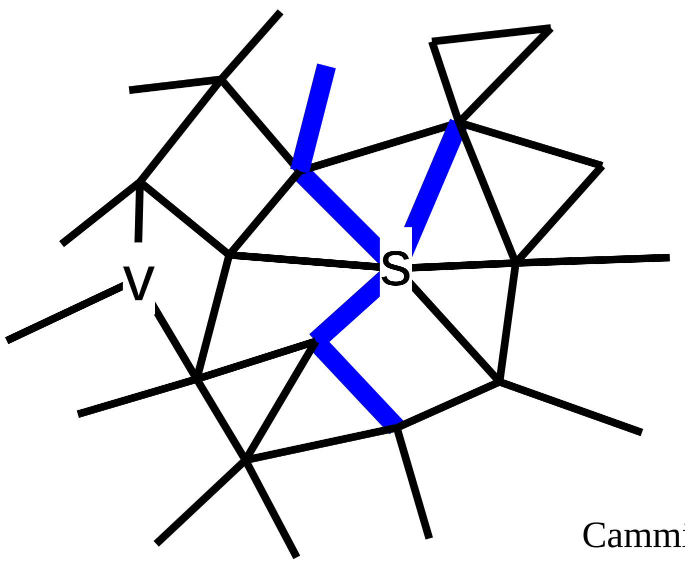
- Sia $G = (V, E)$ un grafo orientato con funzione peso w. Se non ci sono cicli negativi, allora fra ogni coppia di vertici connessi in G esiste sempre un cammino semplice di costo minimo
- Dimostrazione
  - Possiamo sempre trasformare un cammino in un cammino semplice (privo di cicli)

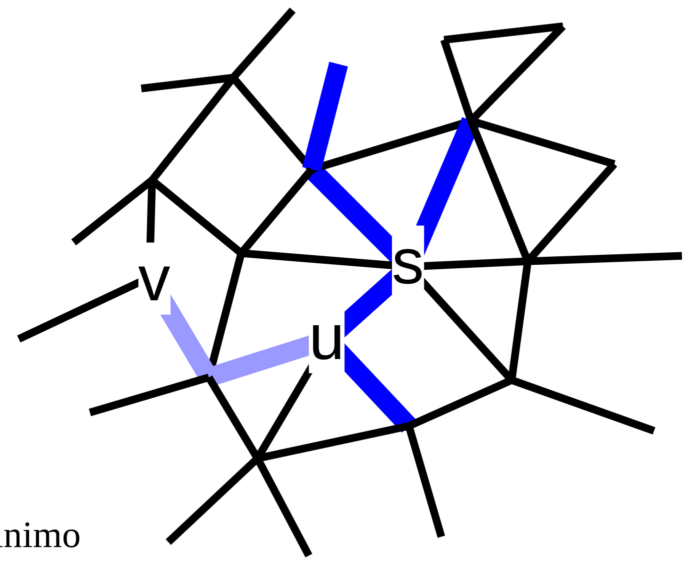


  - Ogni volta che si rimuove un ciclo, il costo diminuisce (o resta uguale), perché per ipotesi non ci sono cicli negativi
  - Il numero di cammini è finito, esiste il minimo

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





Distanza tra vertici in un grafo

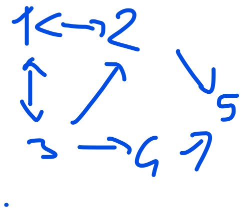
- Sia $G = (V, E)$ un grafo orientato con funzione costo w. La distanza d_{xy} tra x e y in G è il costo di un cammino di costo minimo che li connette; $+\infty$ se tale cammino non esiste

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

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

$$\boxed{d_{xz} \le d_{xy} + d_{yz}}$$

Prendiamo un grafo semplice composto da 5 nodi, con tutti i pesi dei cammini pari a 2. d32 <=d31+w(1,2). La mia sorgente è 3 e voglio andare a 2.



Condizione di Bellman

s= 3
v = 2
mi = 1
Quindi:
2 <=2+2
Per definizione il cammino d32 è il costo minimo per arrivare al punto 2 dal punto 3

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

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

Immaginiamo adesso che i cammini tra 1 e 3 e tra 1 e 2 abbiano peso 0.5
Applichiamo la formula:
duv<=w(u,v)
u = 3, v=2 w(u,v)
duv = d32 = 0.5 +0.5 = 1
Con W indico l'arco mentre con duv indico la distanza tra due nodi (cammino). In questo caso 1<=2

Il costo complessivo di arrivare da v a s, è dato dal peso dell'arco che mi collega u a v

- Dimostrazione
  - Dalla disuguaglianza triangolare si ha

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

  - Ma risulta anche

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

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

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

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

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

Tecnica del *rilassamento*

- Supponiamo di mantenere una <u>stima</u> $D_{sv} \geq d_{sv}$ 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 (Dsu + w(u,v) < Dsv) then Dsv ← Dsu + w(u,v)
```

Algoritmo di Bellman e Ford

- Consideriamo un cammino $\pi_{s\,vk}^* = (s, v_1, \ldots v_k)$ di costo minimo <u>inizialmente ignoto</u>
- Sappiamo che $$\boxed{d_{sv_k} = d_{sv_{k-1}} + w(v_{k-1}, v_k)}$$

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

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

Algoritmo di Bellman e Ford

- Problema: noi non conosciamo gli archi del cammino minimo $\pi_{s\,vk}$* 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"

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

Algoritmo di Bellman e Ford

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

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

- 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 $D_{s\,v_k}$ 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
        D[v] ← +∞ ;
        pred[v] ← -1;
    endfor
    D[s] ← 0;
    for int i ← 1 to n - 1 do
        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;
```

tecnica del rilassamento

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

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

pred[v] = predecessore del nodo v sul cammino di costo minimo che collega s con v

- Costo $O(nm)$

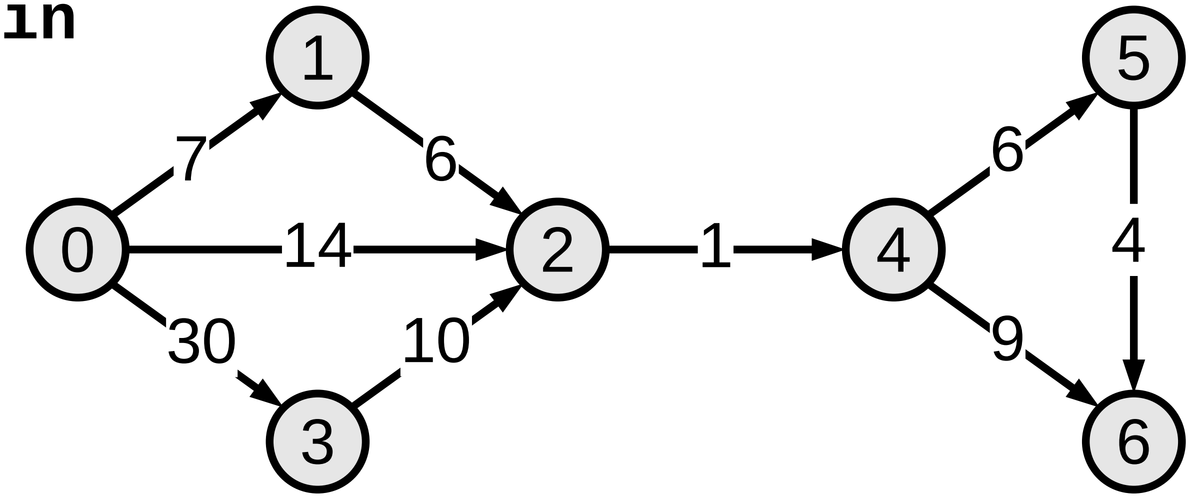
Algoritmo di Bellman e Ford

- 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

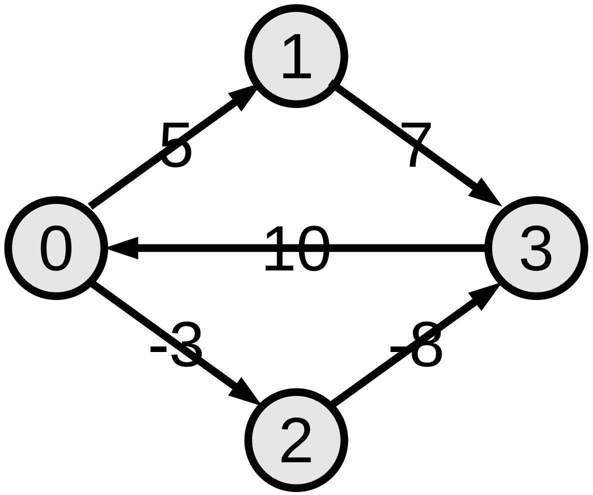
```
// eventuale controllo per cicli negativi
for each (u,v) in E do
    if ( D[u] + w(u,v) < D[v] ) then
        error "Il grafo contiene cicli negativi"
    endif
endfor
```

Esempi

`simple.in`

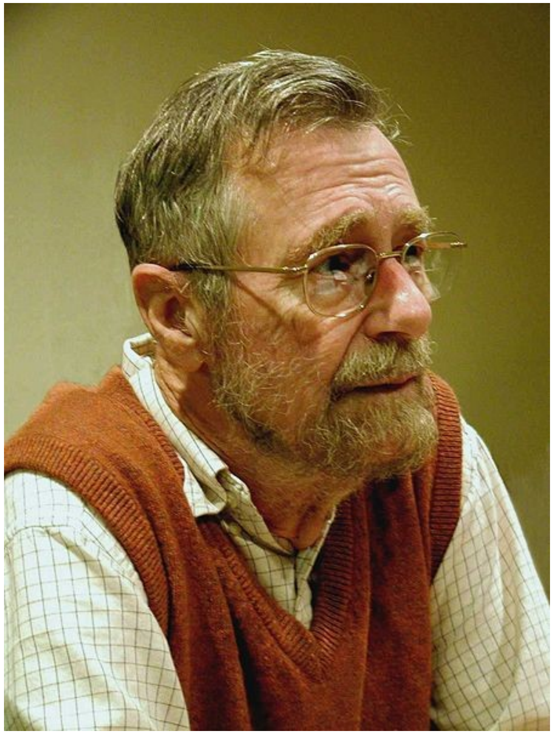


`negative-cycle.in`



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 ≥ 0



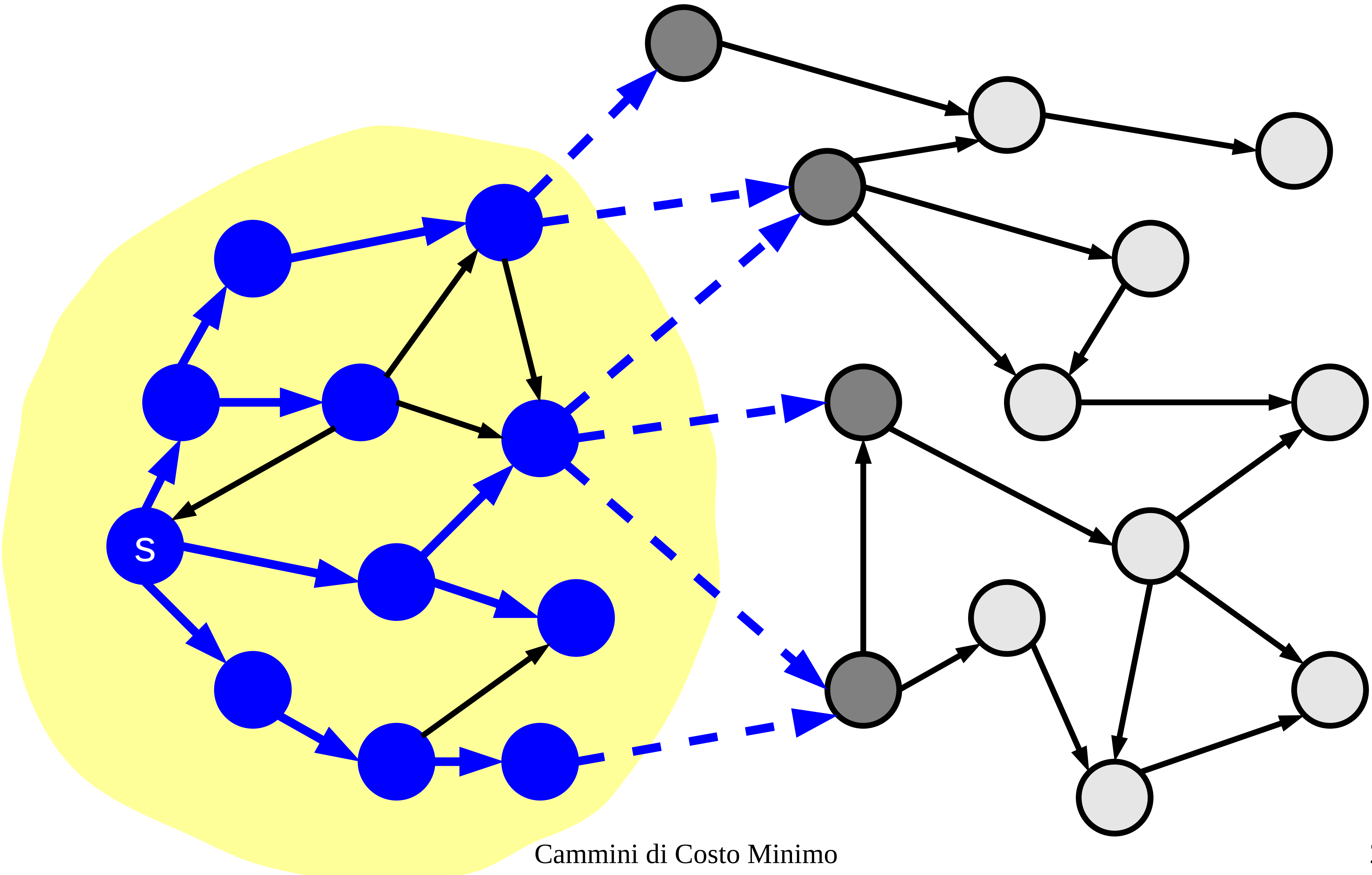
Edsger W. Dijkstra, (1930—2002)
http://en.wikipedia.org/wiki/Edsger_W._Dijkstra

Lemma (Dijkstra)

- Sia $G = (V, E)$ un grafo orientato con funzione costo w
  - I costi degli archi devono essere ≥ 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

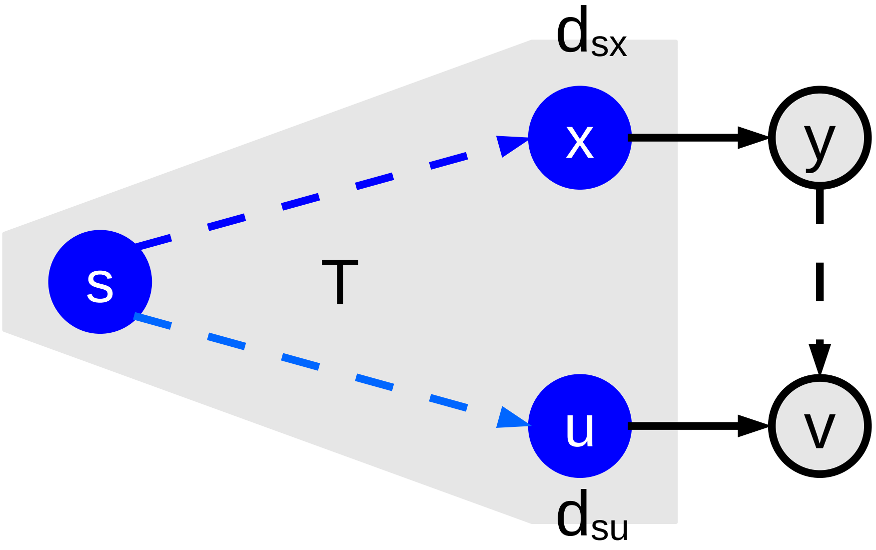
Allora l'arco (u, v) con $u \in V(T)$ e $v \notin V(T)$ che minimizza la quantità $d_{su} + w(u, v)$ appartiene ad un cammino minimo da s a v

Lemma (Dijkstra)



Dimostrazione

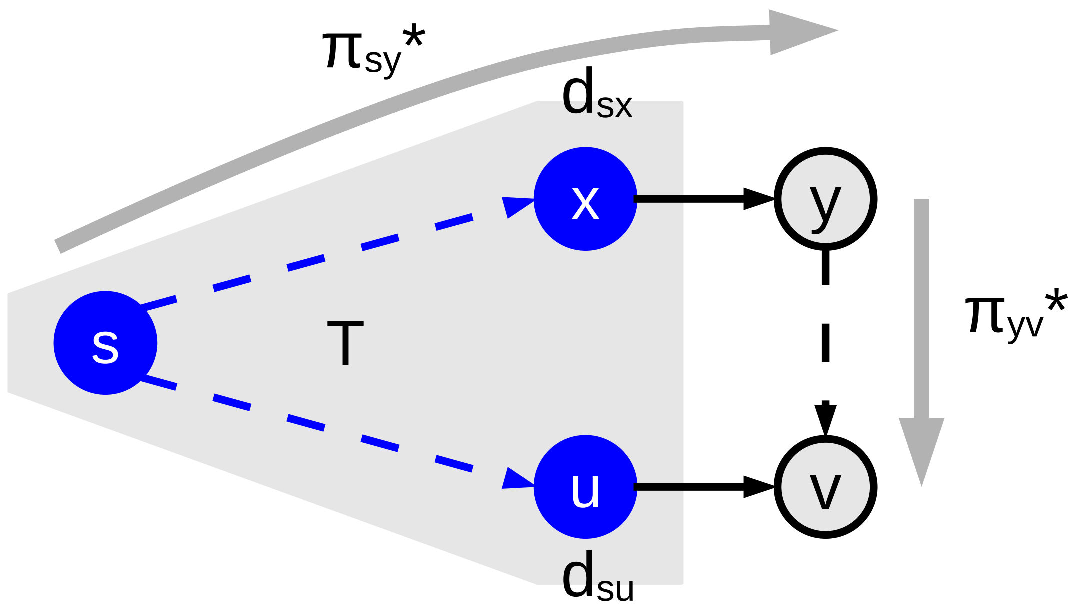
- Supponiamo per assurdo che (u,v) *non appartenga* ad un cammino di costo minimo tra s e v
  - quindi $d_{su} + w(u,v) > d_{sv}$ 1
- Quindi deve esistere π_{sv}^* che porta da s in v senza passare per (u,v) con costo inferiore a $d_{su} + w(u,v)$



Dimostrazione

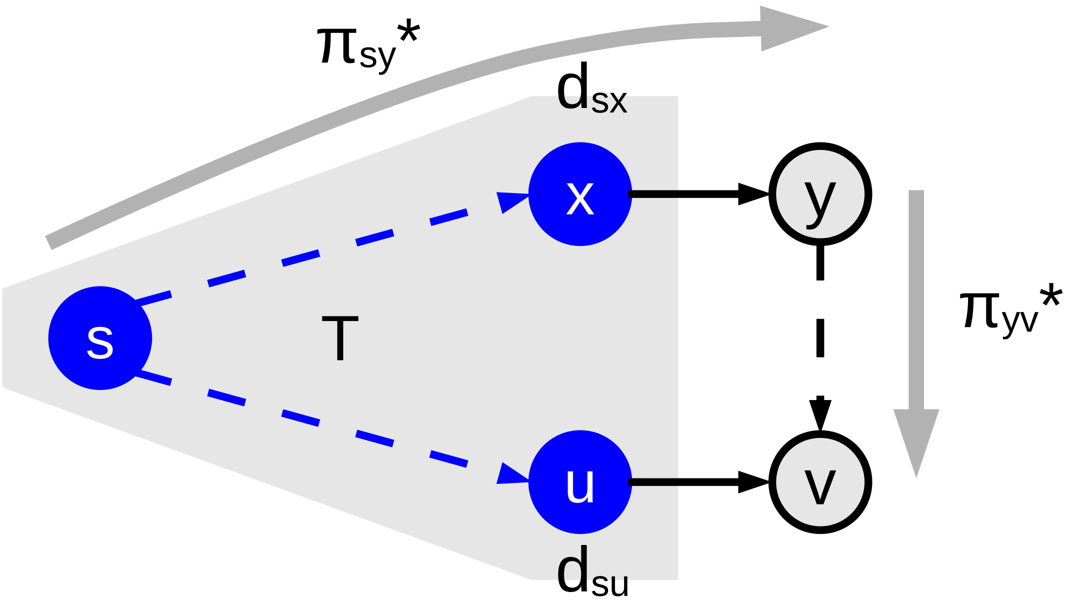
- Per il teorema di sottostruttura ottima, il cammino π_{sv}* si scompone in π_{sy}* e π_{yv}*
- Quindi $d_{sv} = d_{sx} + w(x,y) + d_{yv}$

2



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: $d_{su} + w(u,v) \leq d_{sx} + w(x,y)$ 3

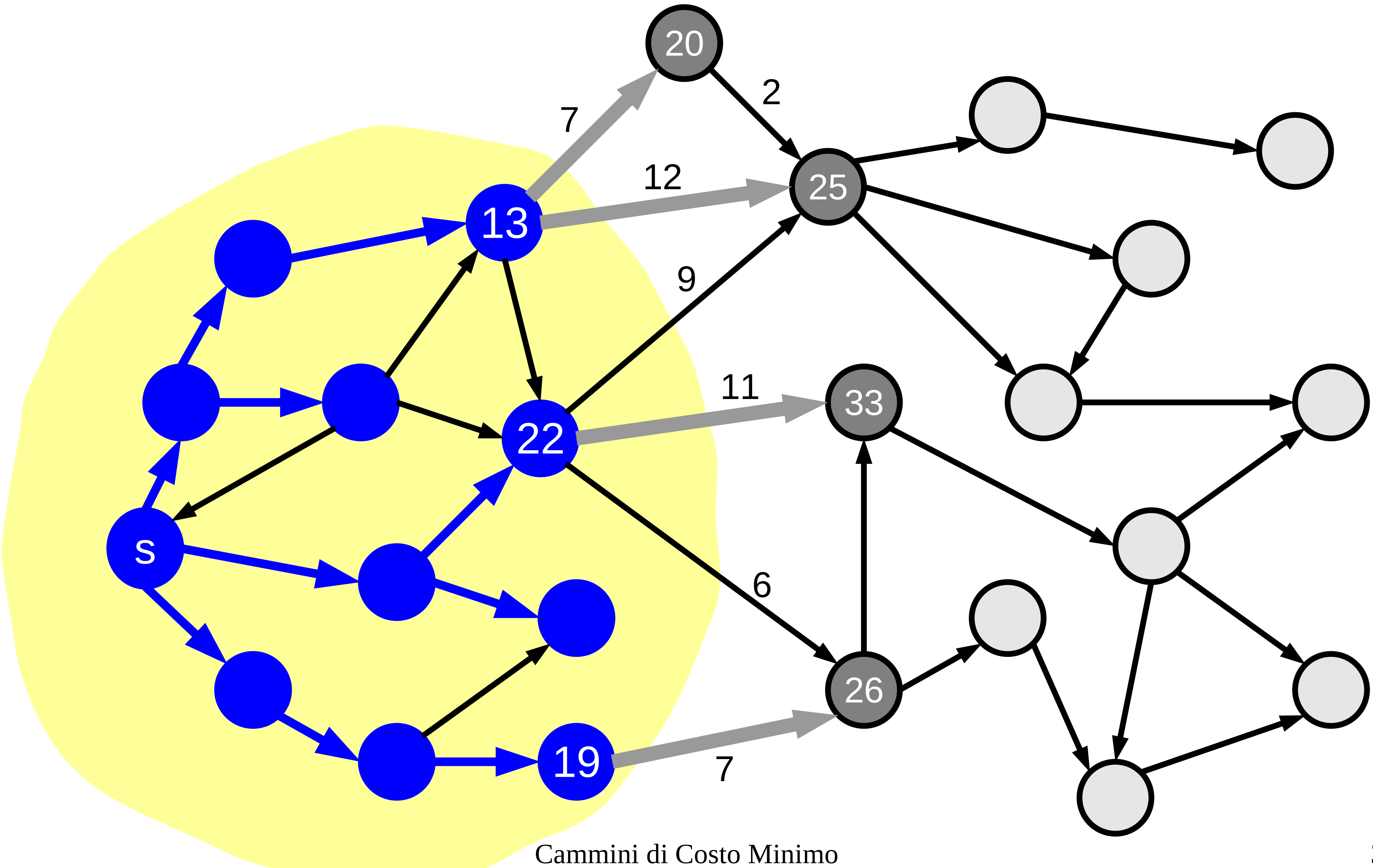


Riassumiamo

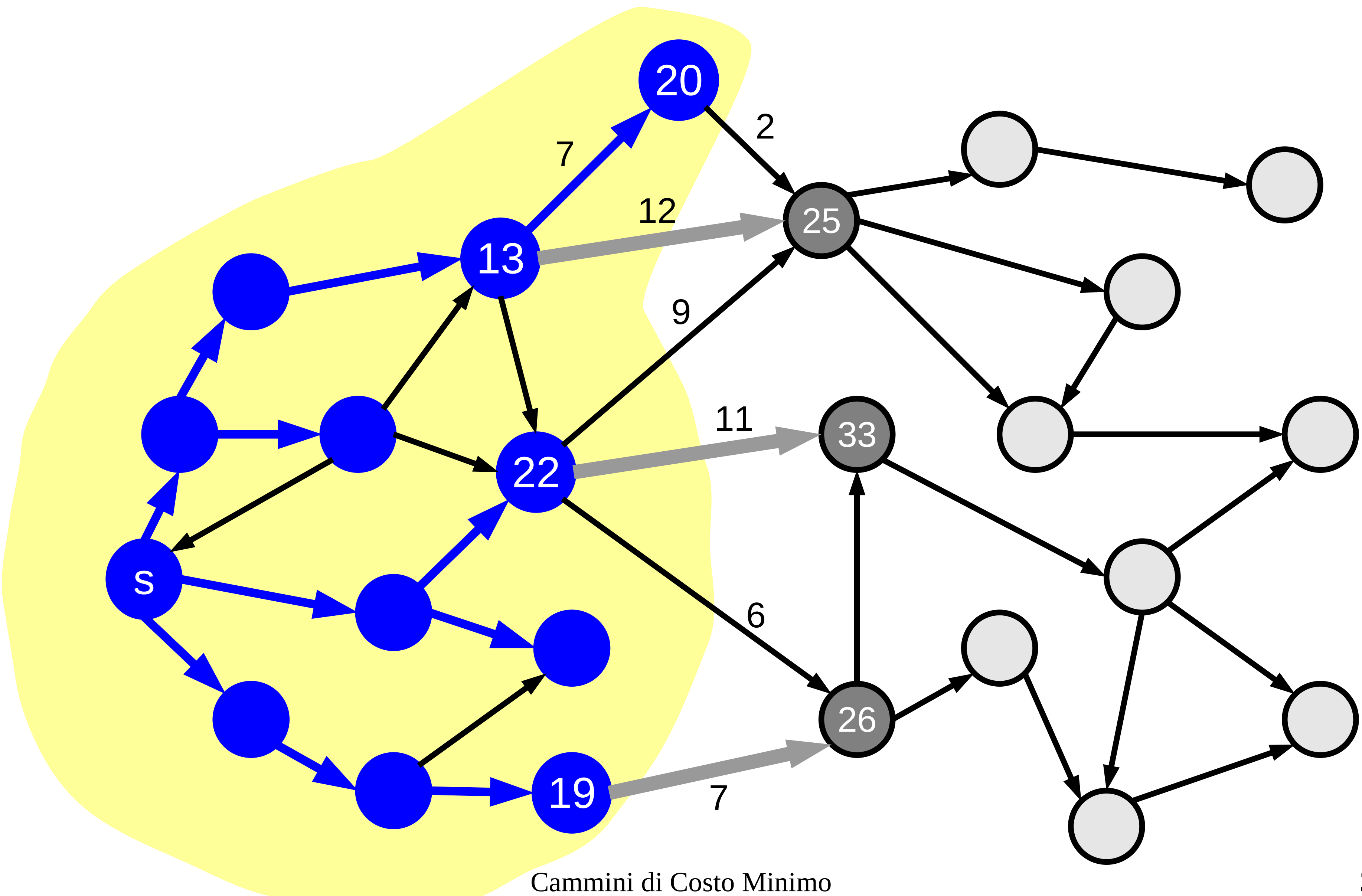
- Da (1) abbiamo $d_{su} + w(u,v) > d_{sv}$
- Da (2) abbiamo $d_{sv} = d_{sx} + w(x,y) + d_{yv}$
- Da (3) abbiamo $d_{su} + w(u,v) \leq d_{sx} + w(x,y)$
- Combinando (1) (2) e (3) otteniamo

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

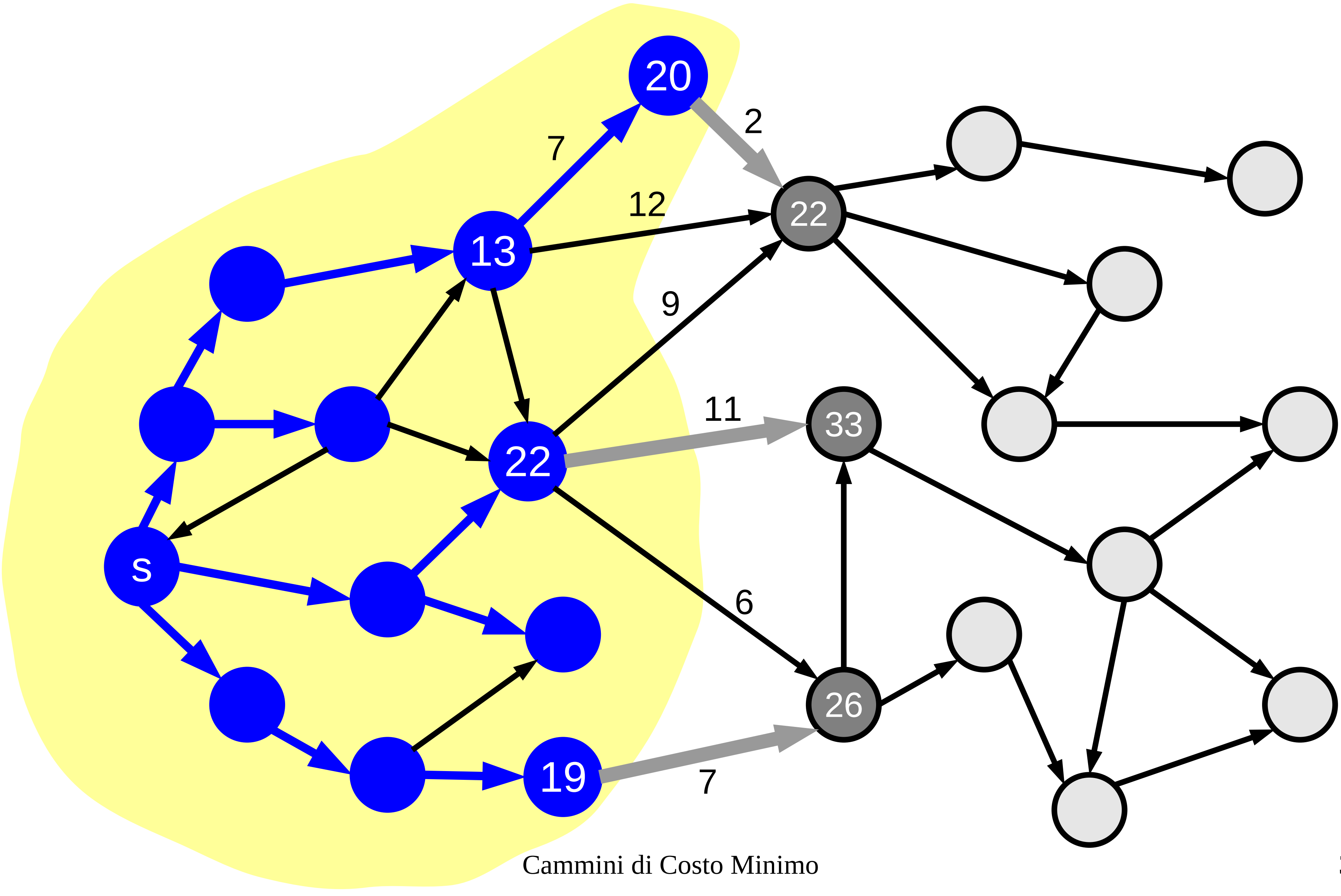
Esempio



Esempio



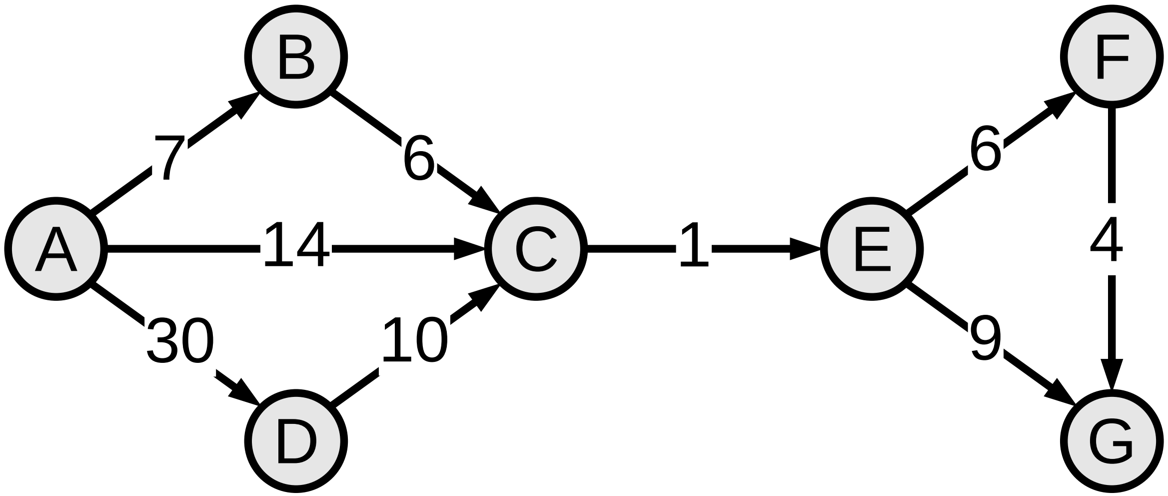
Esempio



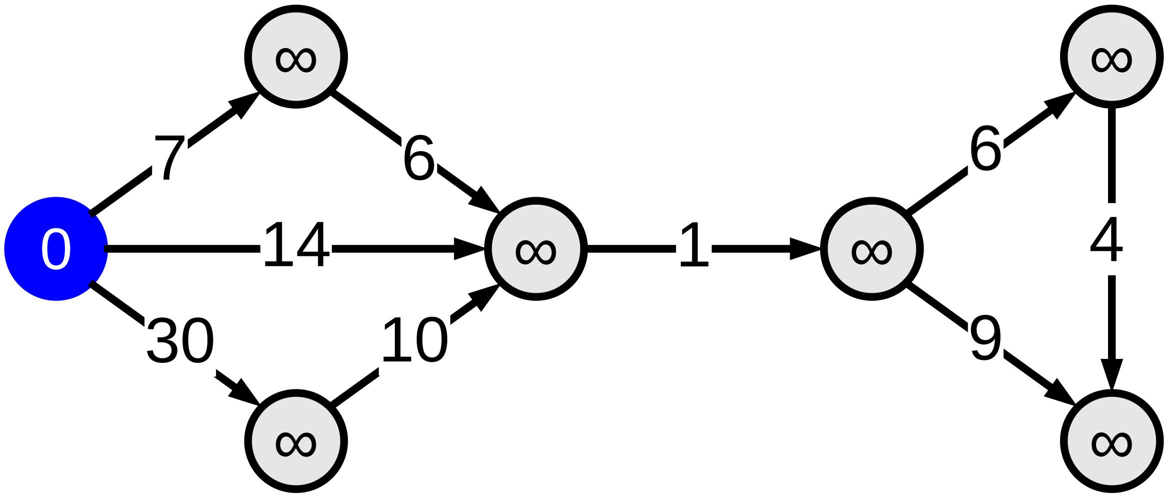
Algoritmo di Dijkstra generico

```
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;
    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;
```

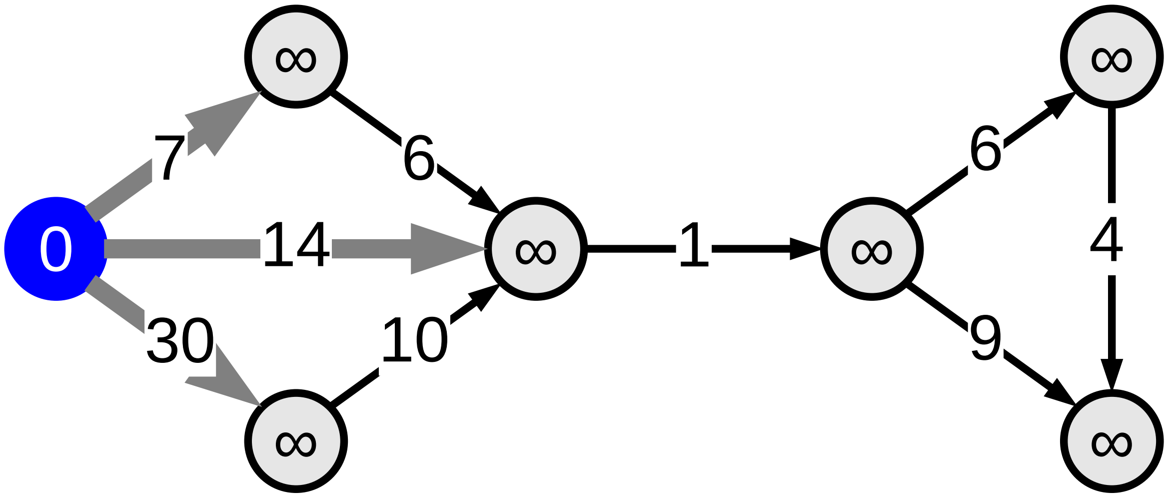
Esempio



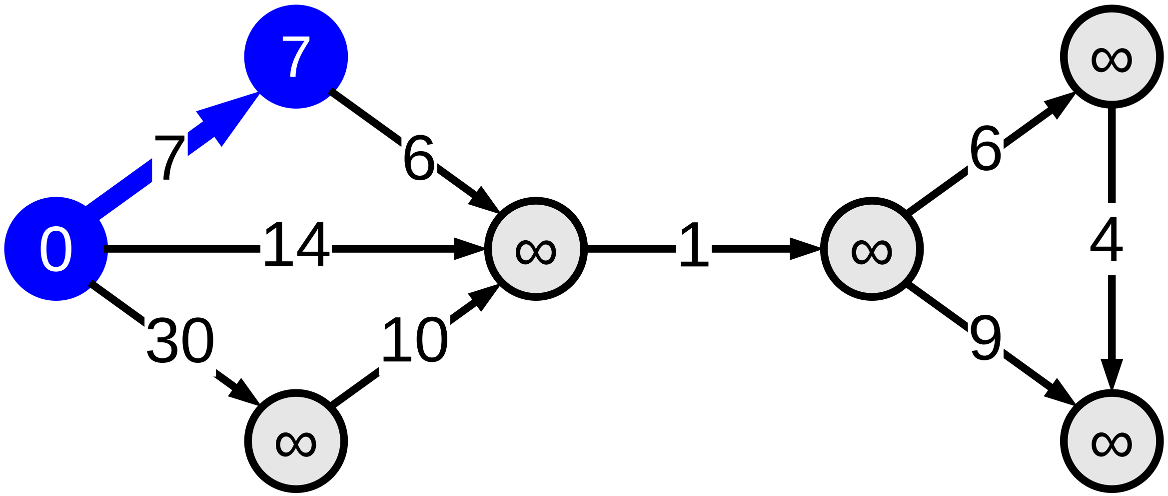
Esempio



Esempio

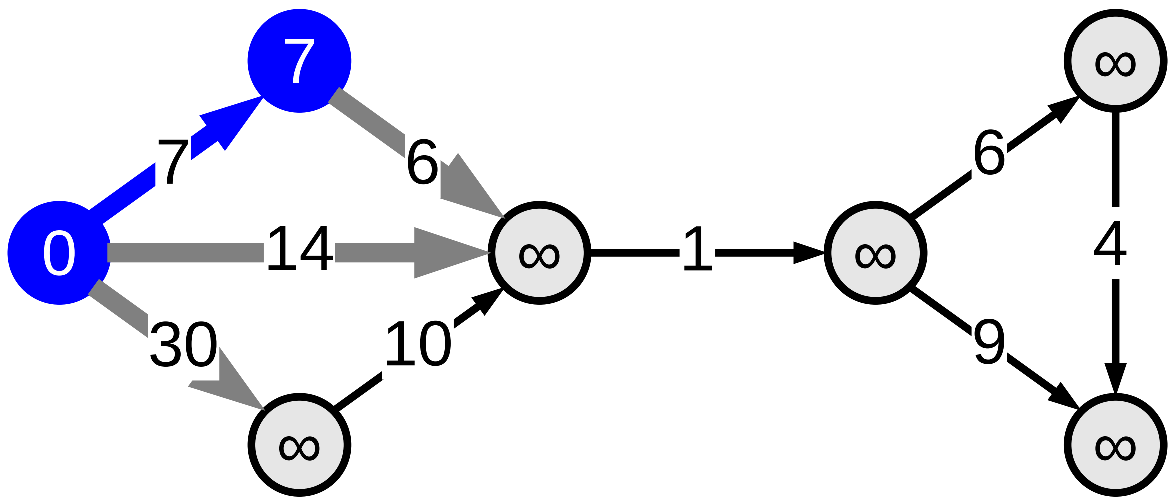


Esempio

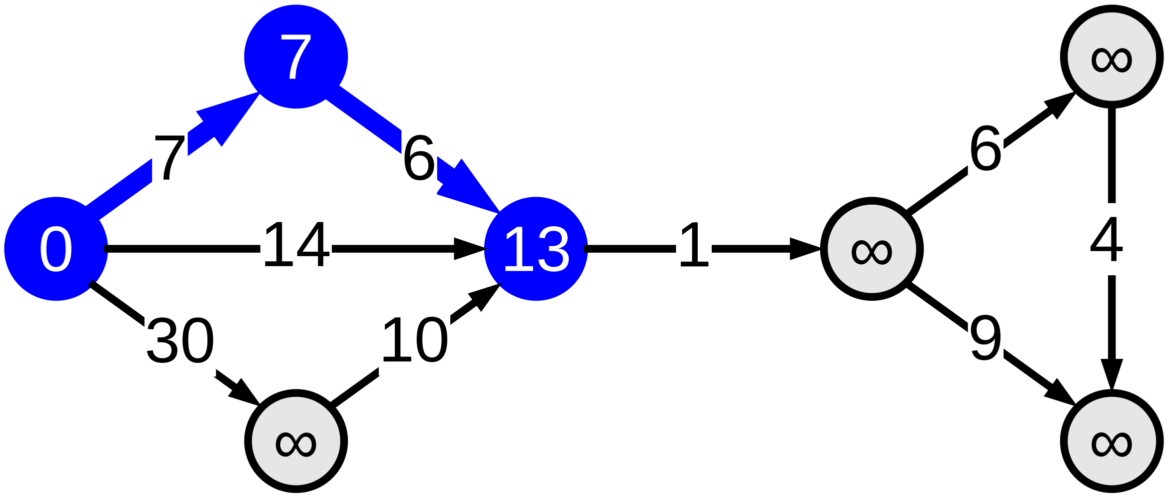


dB = 7
dA =0
dB+W(B,C) = 13
dA+W(A,C) = 14
W(A,D) = 30
Seleziono il monore

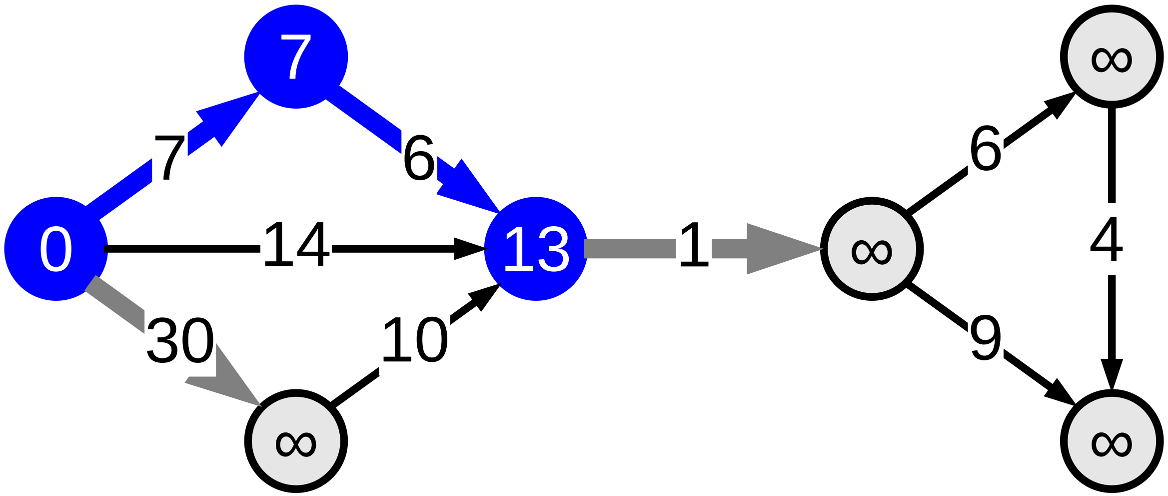
Esempio



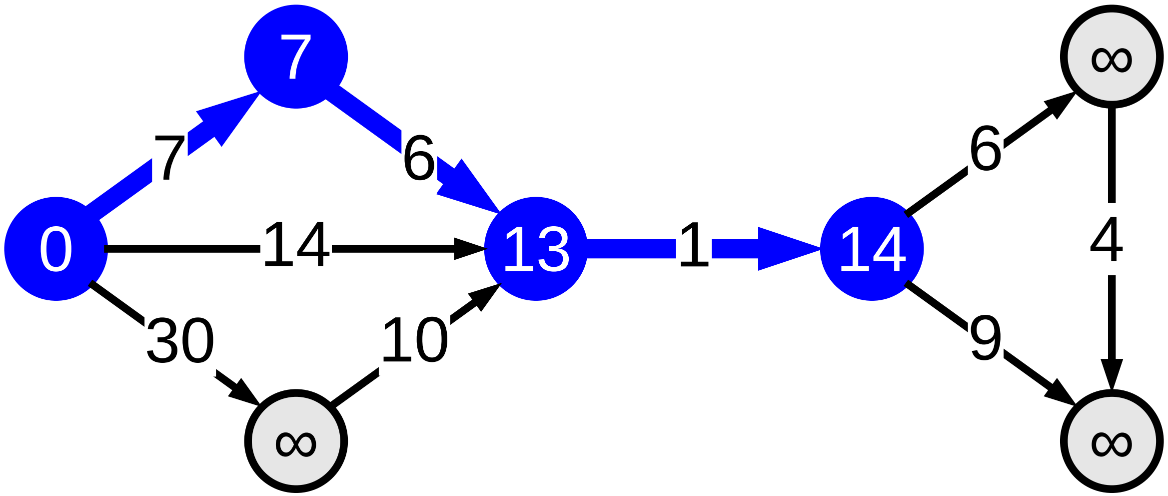
Esempio



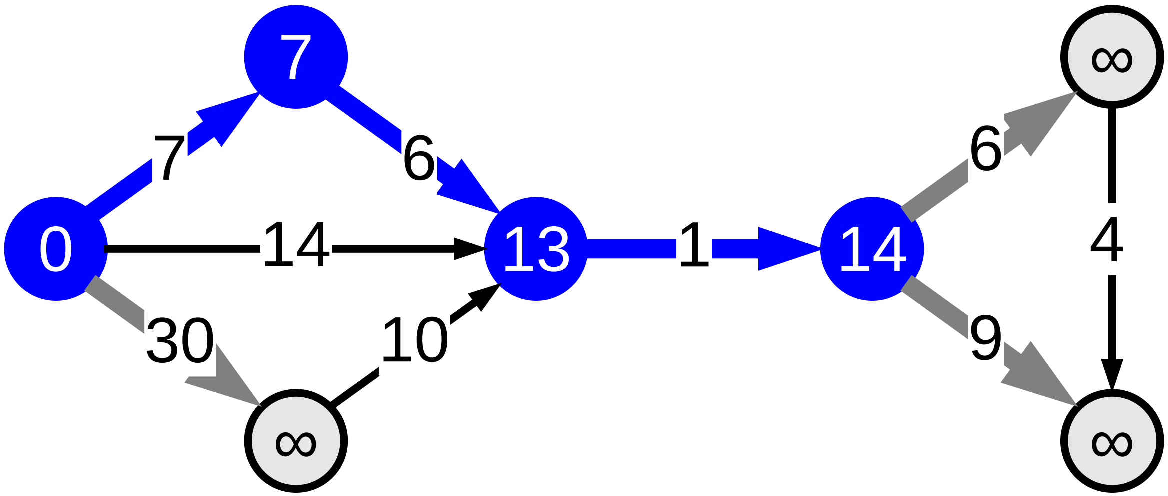
Esempio



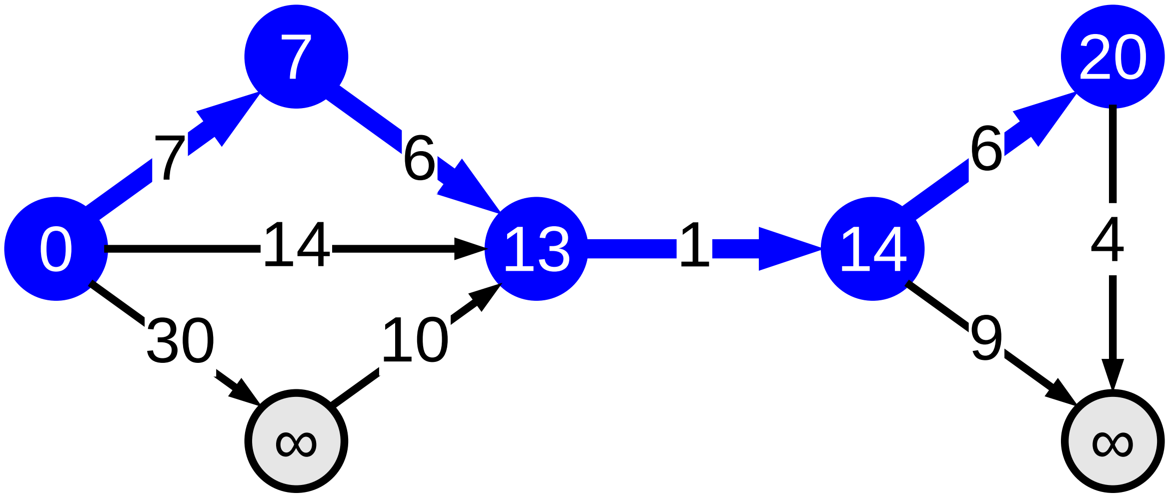
Esempio



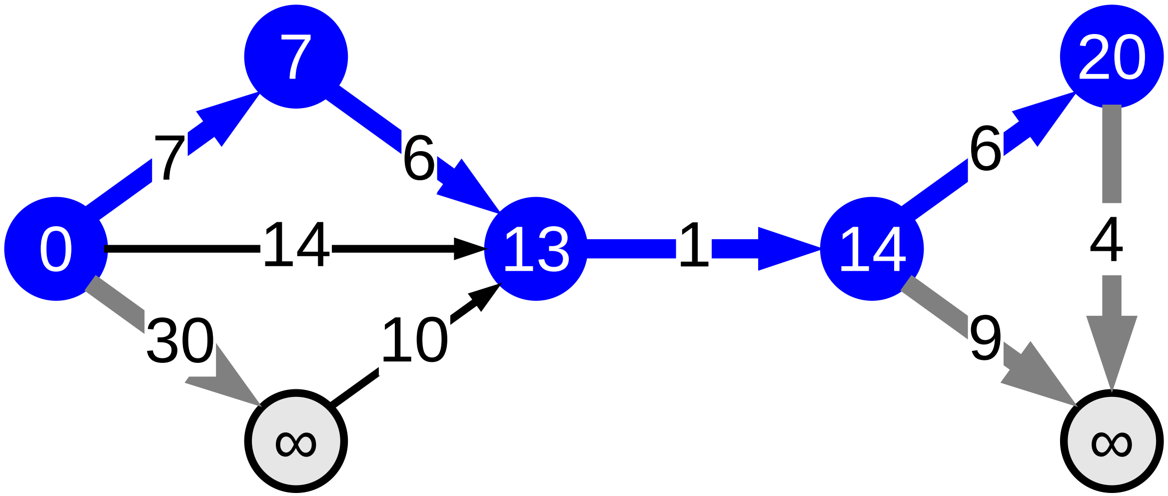
Esempio



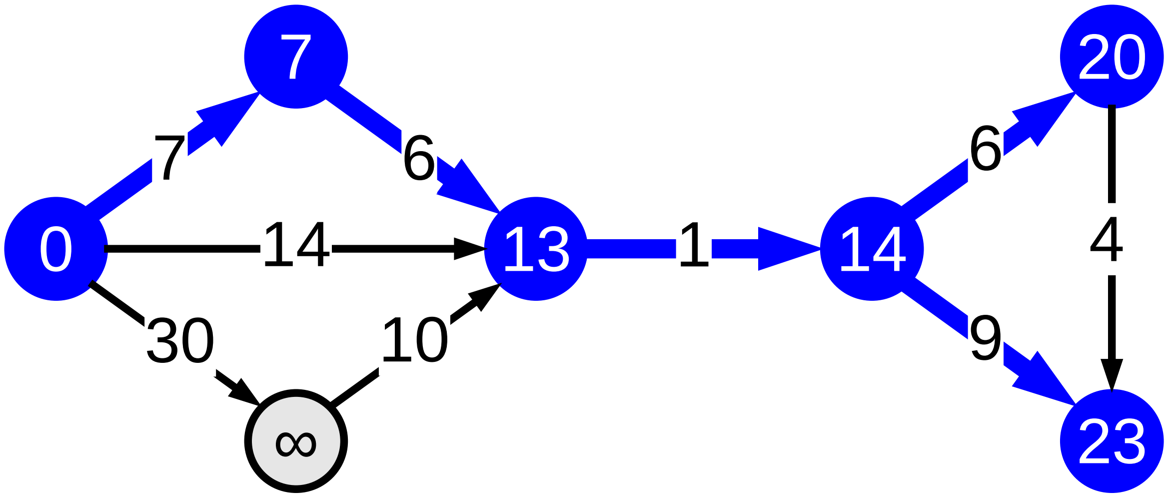
Esempio



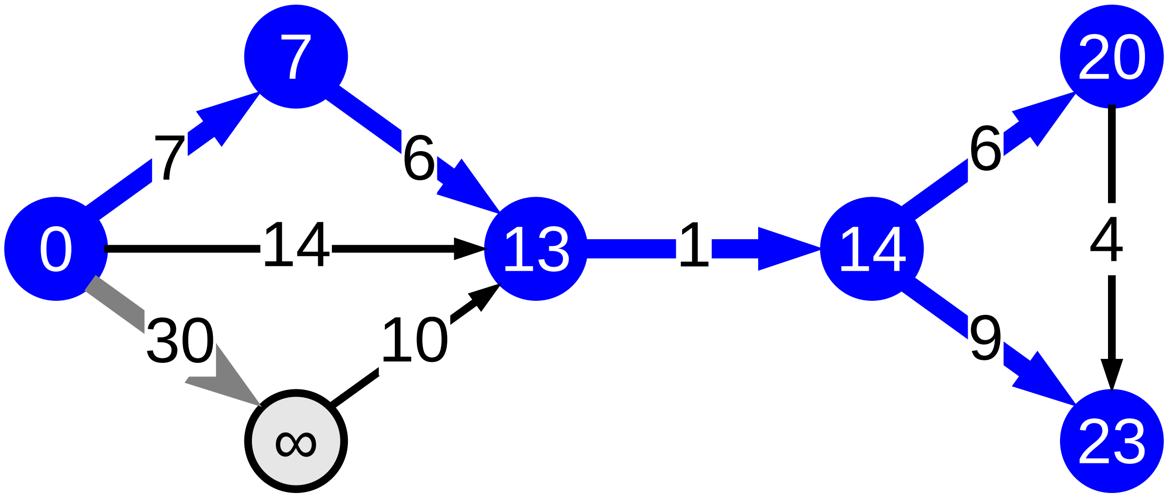
Esempio



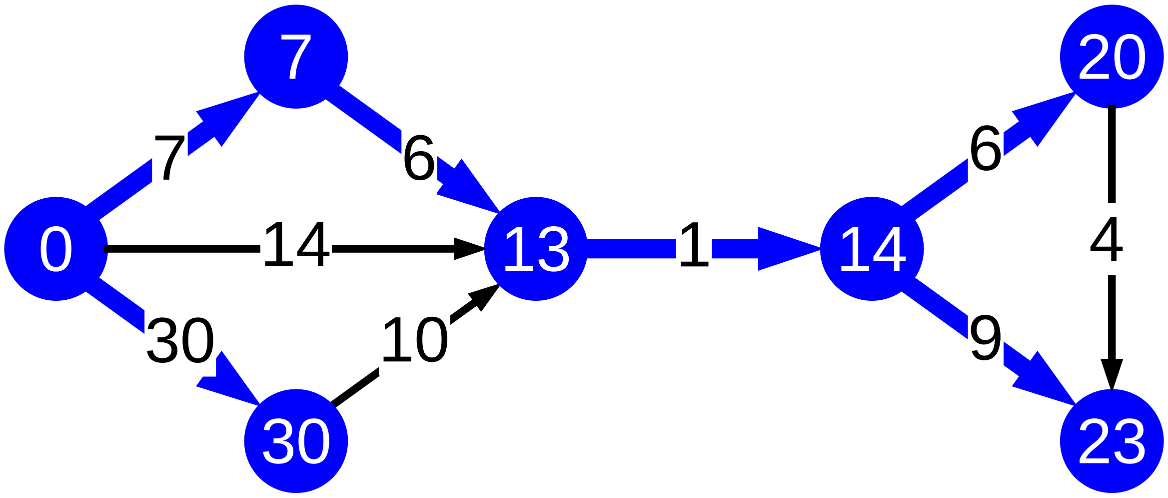
Esempio



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;
    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();
        Q.deleteMin();
        added[u] ← true;
        for each v adiacente a u do
            if (not added[v] and D[u] + w(u,v) < D[v]) then
                D[v] ← D[u] + w(u,v);
                Q.decreaseKey(v, D[v]);
                pred[v] ← u;
            endif
        endfor
    endwhile
    return D;
```

Trova e rimuovi il nodo con distanza minima

Somiglia all'algoritmo di Prim (MST), ma priorità diversa

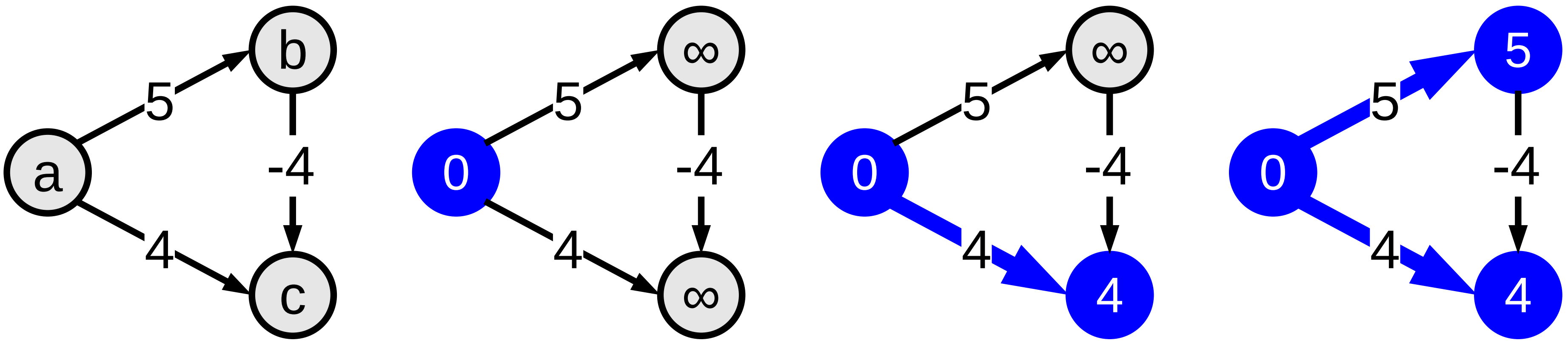
Rendi D[u]+w(u,v) la nuova distanza di v da s

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 ≥ 0
- Esempio di funzionamento errato

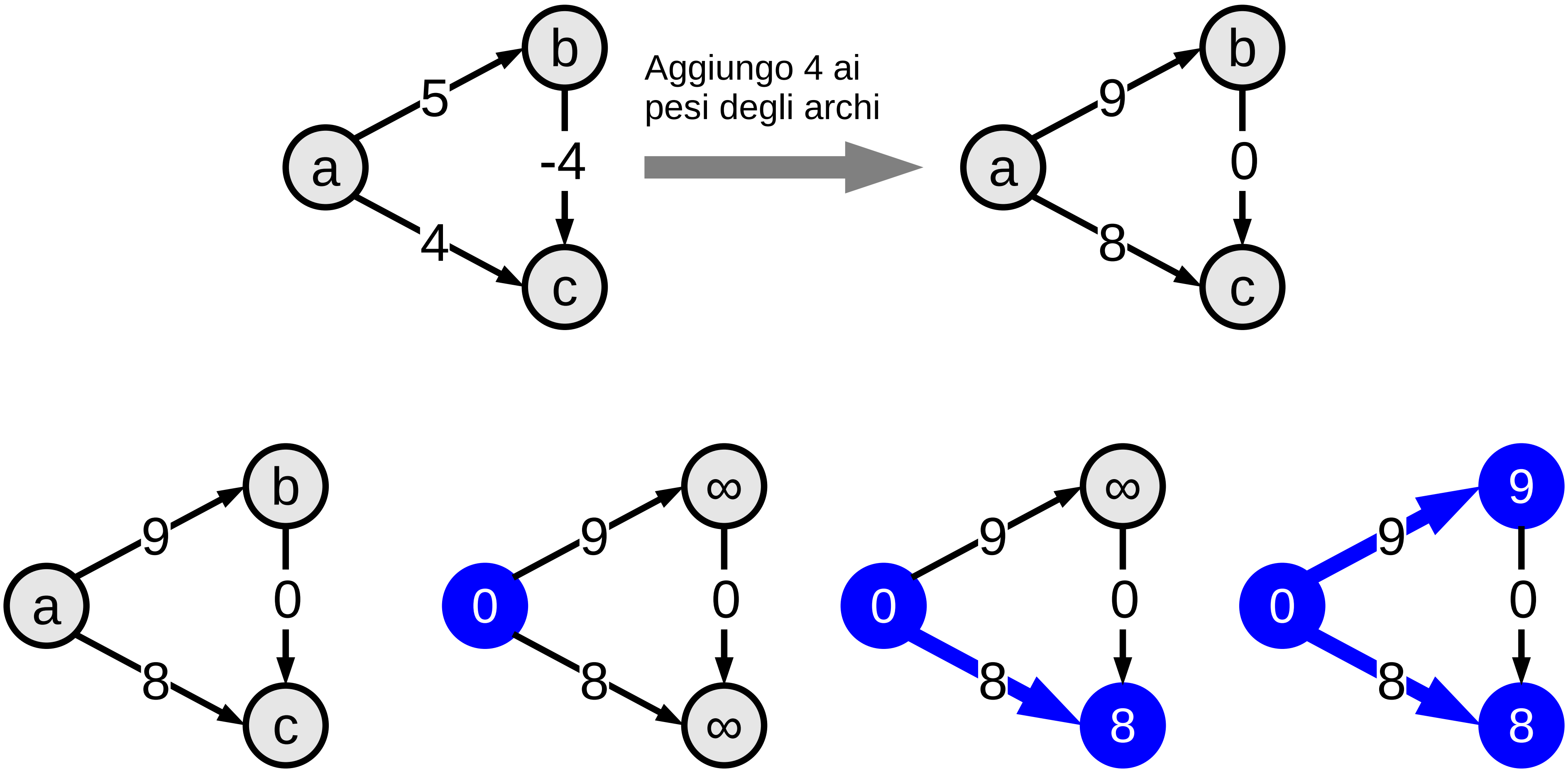


  - Il cammino minimo da $a \rightarrow c$ non è (a,c) ma (a,b,c) che ha costo 1

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

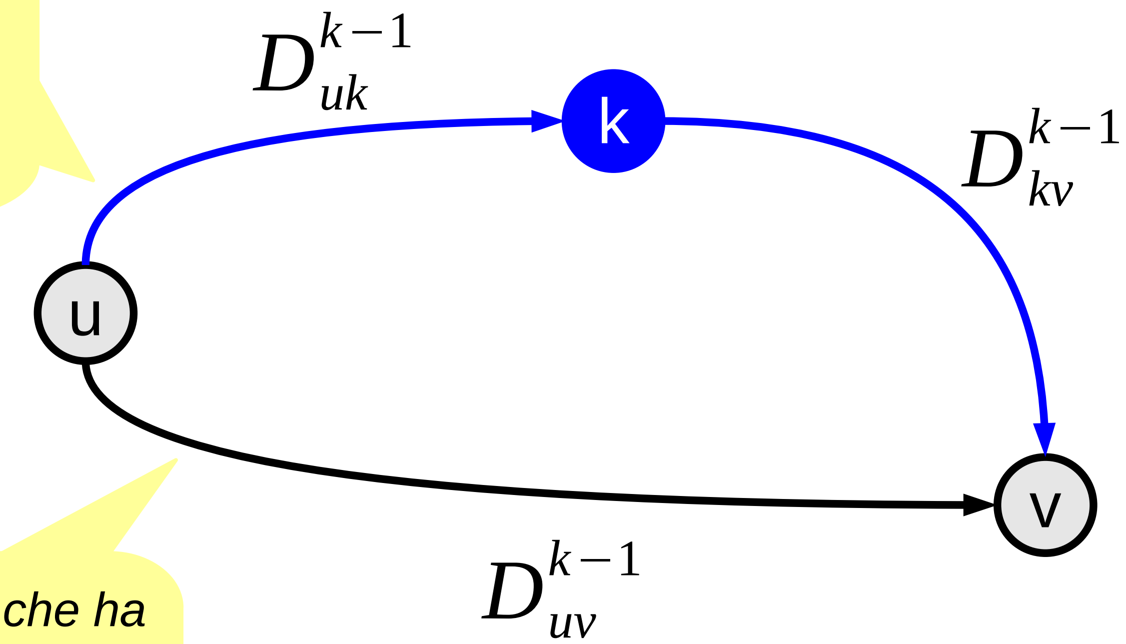
Inizializzazione

- 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

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

Caso generale

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



Cammino che ha come nodi intermedi solo nodi nell'insieme {1, .. k-1}

Caso generale

- Per andare da *u* a *v* usando solo nodi intermedi in {1, … *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 $D_{u\,k}^{k-1} + D_{k\,v}^{k-1}$
- Quindi

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

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);
            else D[u,v,0] ← + ∞;
            endif
        endfor
    endfor
    for k ← 1 to n do
        for u ← 1 to n do
            for v ← 1 to n do
                D[u,v,k] ← D[u,v,k-1];
                if (D[u,v,k-1] + D[u,v,k-1] < D[u,v,k] ) then
                    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];
```

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

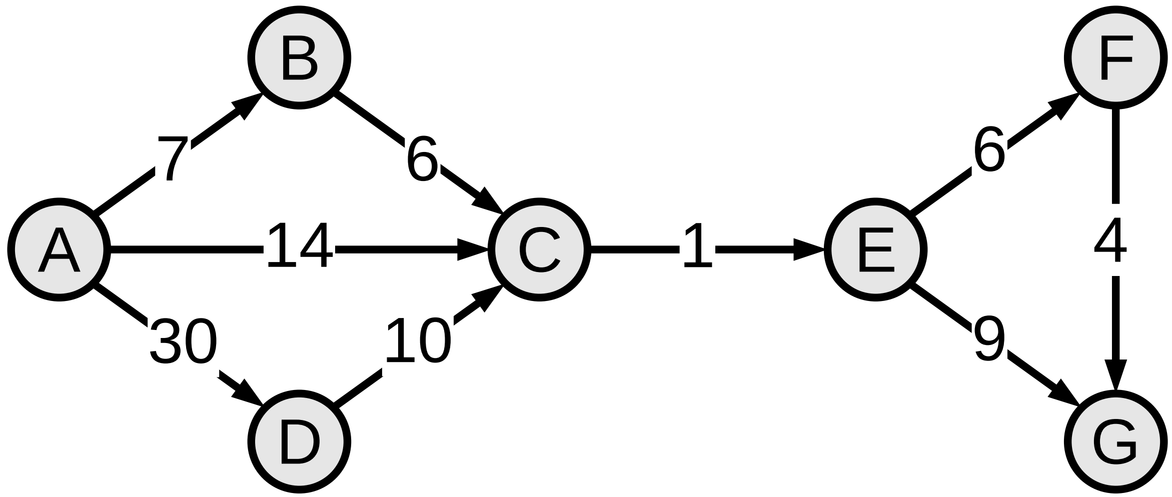
- Costo: tempo O(n^3), spazio O(n^3)

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 ← 1 to n do
    if ( D[u,u,n] < 0 ) then
        error “Il grafo contiene cicli negativi”
    endif
endfor
```

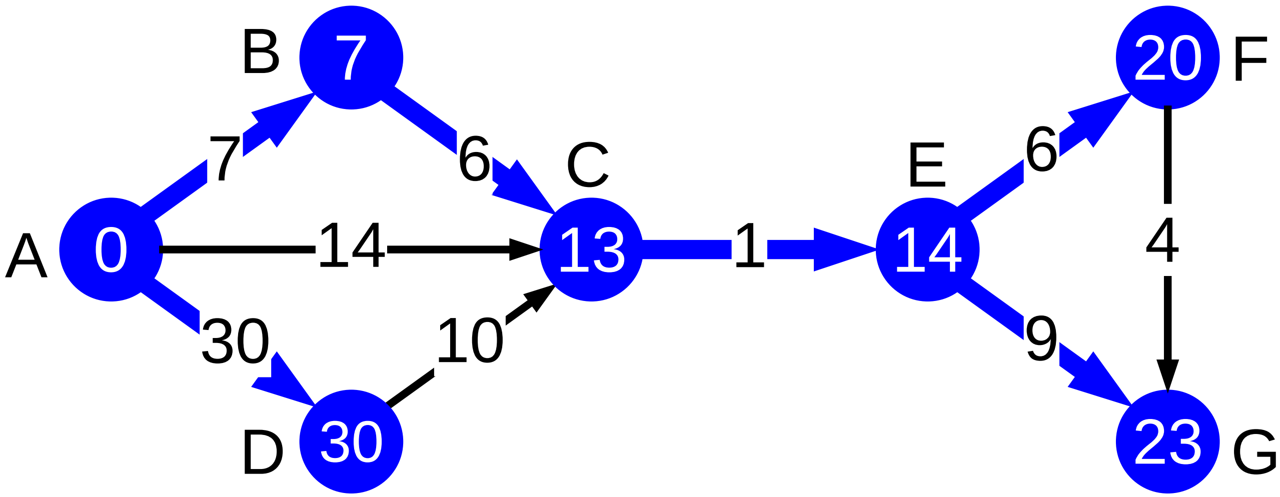
Esempio



D[u,v,0] =

	A	***B***	***C***	***D***	***E***	***F***	***G***
A	0	7	14	30	Inf	Inf	Inf
B	Inf	0	6	Inf	Inf	Inf	Inf
C	Inf	Inf	0	Inf	1	Inf	Inf
D	Inf	Inf	10	0	Inf	Inf	Inf
E	Inf	Inf	Inf	Inf	0	6	9
F	Inf	Inf	Inf	Inf	Inf	0	4
G	Inf	Inf	Inf	Inf	Inf	Inf	0

Esempio




```
D[u,v,n] =
        A      B      C      D      E      F      G
A       0      7     13     30     14     20     23
B     Inf      0      6    Inf      7     13     16
C     Inf    Inf      0    Inf      1      7     10
D     Inf    Inf     10      0     11     17     20
E     Inf    Inf    Inf    Inf      0      6      9
F     Inf    Inf    Inf    Inf    Inf      0      4
G     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)

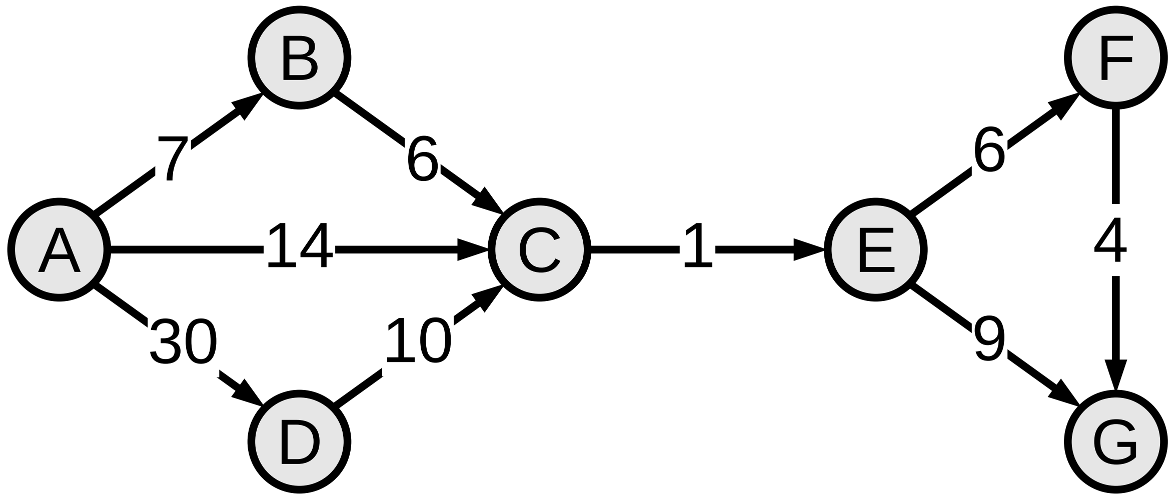
```
double[1..n,1..n] FloydWarshall2( G=(V,E,w) )
    int n ← G.numNodi();
    double D[1..n, 1..n];
    int u, v, k, next[1..n, 1..n];
    for u ← 1 to n do
        for v ← 1 to n do
            if (u == v) then
                D[u,v] ← 0;
                next[u,v] ← -1;
            elseif ((u,v) ∈ E) then
                D[u,v] ← w(u,v);
                next[u,v] ← v;
            else
                D[u,v] ← + ∞;
                next[u,v] ← -1;
            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];
                    next[u,v] ← next[u,k];
                endif
            endfor
        endfor
    endfor
    return D;
```

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] =
```

	A	B	C	D	E	F	G
A	-1	B	B	D	B	B	B
B	-1	-1	C	-1	C	C	C
C	-1	-1	-1	-1	E	E	E
D	-1	-1	C	-1	C	C	C
E	-1	-1	-1	-1	-1	F	G
F	-1	-1	-1	-1	-1	-1	G
G	-1	-1	-1	-1	-1	-1	-1

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