

# Algoritmi e Strutture Dati

## Scelta della struttura dati

Alberto Montresor

Università di Trento

2024/08/11

This work is licensed under a Creative Commons  
Attribution-ShareAlike 4.0 International License.



# Sommario

- 1 Introduzione
- 2 Cammini minimi, sorgente singola
  - Dijkstra
  - Johnson
  - Fredman-Tarjan
  - Bellman-Ford-Moore
  - Casi speciali – DAG
- 3 Cammini minimi, sorgente multipla
  - Floyd-Warshall
  - Chiusura transitiva
- 4 Conclusione

# Problema cammini minimi

## Input

- Grafo orientato  $G = (V, E)$
- Un nodo sorgente  $s$
- Una funzione di peso  $w : E \rightarrow R$

## Definizione

Dato un cammino  $p = \langle v_1, v_2, \dots, v_k \rangle$  con  $k > 1$ , il costo del cammino è dato da

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

## Output

Trovare un cammino da  $s$  ad  $u$ , per ogni nodo  $u \in V$ , il cui costo sia minimo, ovvero più piccolo o uguale del costo di qualunque altro cammino da  $s$  a  $u$ .

# Panoramica sul problema

## Cammini minimi da singola sorgente

- **Input:** Grafo pesato, nodo radice  $s$
- **Output:** i cammini minimi che vanno da  $s$  a tutti gli altri nodi

## Cammino minimo tra una coppia di vertici

- **Input:** Grafo pesato, una coppia di vertici  $s, d$
- **Output:** un cammino minimo fra  $s$  e  $d$
- Si risolve il primo problema e si estrae il cammino richiesto  
Non si conoscono algoritmi più efficienti

# Panoramica sul problema

## Cammini minimi tra tutte le coppie di vertici

- **Input:** Grafo pesato
- **Output:** i cammini minimi fra tutte le coppie di vertici.
- Soluzione basata su programmazione dinamica

# Pesi

## Tipologie di pesi

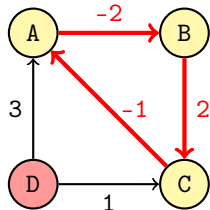
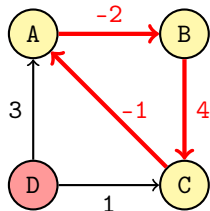
Algoritmi diversi possono funzionare oppure no per alcune categorie speciali di pesi

- Positivi / positivi+negativi
- Reali / interi

## Esempio: proprietario di un TIR

- Viaggiare scarico: perdita, peso positivo
- Viaggiare carico: profitto, peso negativo

Domanda: perché i cicli negativi sono un problema?



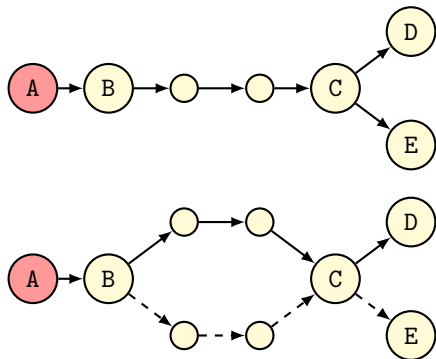
# Sommario

- 1 Introduzione
- 2 Cammini minimi, sorgente singola
  - Dijkstra
  - Johnson
  - Fredman-Tarjan
  - Bellman-Ford-Moore
  - Casi speciali – DAG
- 3 Cammini minimi, sorgente multipla
  - Floyd-Warshall
  - Chiusura transitiva
- 4 Conclusione

## Problema cammini minimi – Sottostruttura ottima

Si noti che due cammini minimi possono avere un tratto in comune  $A \rightsquigarrow C \dots$

$\dots$  ma non possono convergere in un nodo comune  $C$  dopo aver percorso un tratto distinto



### Albero dei cammini minimi

L'**albero dei cammini minimi** è un albero di copertura radicato in  $s$  avente un cammino da  $s$  a tutti i nodi raggiungibili da  $s$ .

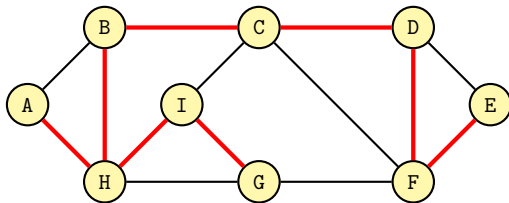


# Albero di copertura

## Albero di copertura (Spanning tree)

Dato un grafo  $G = (V, E)$  non orientato e connesso, un albero di copertura di  $G$  è un sottografo  $T = (V, E_T)$  tale che

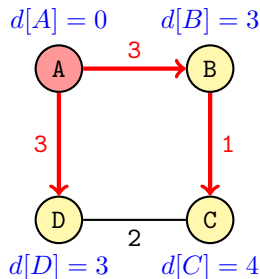
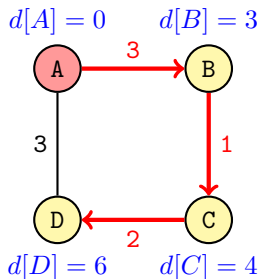
- $T$  è un albero
- $E_T \subseteq E$
- $T$  contiene tutti i vertici di  $G$



# Soluzione ammissibile

## Soluzione ammissibile

Una soluzione **ammissibile** può essere descritta da un **albero di copertura**  $T$  radicato in  $s$  e da un **vettore di distanza**  $d$ , i cui valori  $d[u]$  rappresentano il costo del cammino da  $s$  a  $u$  in  $T$ .



## Rappresentazione dell'albero

Per rappresentare l'albero, utilizziamo la rappresentazione basata su vettore dei padri, così come abbiamo fatto con le visite in ampiezza/profondità.

---

```
printPath(NODE s, NODE d, NODE[] T)
```

---

```

if s == d then
  | print s
else if T[d] == nil then
  | print "error"
else
  | printPath(s, T[d], T)
  | print d

```

---

# Teorema di Bellman

## Teorema di Bellman

Una soluzione ammissibile  $T$  è **ottima** se e solo se:

$$d[v] = d[u] + w(u, v)$$

per ogni arco  $(u, v) \in T$

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

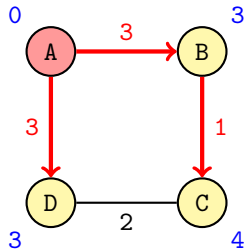
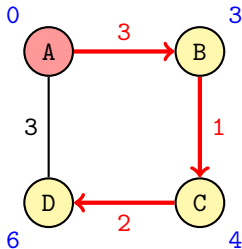
per ogni arco  $(u, v) \in E$

$$d[B] = d[A] + w(A, B) \quad d[C] = d[B] + w(B, C)$$

$$d[D] = d[C] + w(C, D) \quad d[D] > d[A] + w(A, D)$$

$$d[B] = d[A] + w(A, B) \quad d[C] = d[B] + w(B, C)$$

$$d[D] = d[A] + w(A, D) \quad d[D] \leq d[C] + w(C, D)$$



# Dimostrazione

## Teorema di Bellman - Parte 1

Se  $T$  è una soluzione ottima, allora valgono le condizioni di Bellman:

$$d[v] = d[u] + w(u, v) \quad \text{per ogni arco } (u, v) \in T$$

$$d[v] \leq d[u] + w(u, v) \quad \text{per ogni arco } (u, v) \in E$$

Sia  $T$  una soluzione ottima e sia  $(u, v) \in E$ .

- Se  $(u, v) \in T$ , allora  $d[v] = d[u] + w(u, v)$
- Se  $(u, v) \notin T$ , allora  $d[v] \leq d[u] + w(u, v)$ , perchè altrimenti esisterebbe nel grafo  $G$  un cammino da  $s$  a  $v$  più corto di quello in  $T$ , assurdo.

# Dimostrazione

## Teorema di Bellman - Parte 2

Se valgono le condizioni di Bellman:

$$d[v] = d[u] + w(u, v) \quad \text{per ogni arco } (u, v) \in T$$

$$d[v] \leq d[u] + w(u, v) \quad \text{per ogni arco } (u, v) \in E$$

allora  $T$  è una soluzione ottima.

- Supponiamo per assurdo che  $T$  non sia ottimo
- Allora esiste un cammino  $C$  da  $s$  ad un nodo  $u$  in  $T$  non ottimo
- Allora esiste un albero ottimo  $T'$ , in cui il cammino  $C'$  da  $s$  a  $u$  ha distanza  $d'[u] < d[u]$
- Sia  $d'$  il vettore delle distanze associato a  $T'$

# Dimostrazione

## Teorema di Bellman - Parte 2

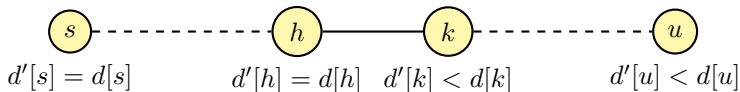
Se valgono le condizioni di Bellman:

$$d[v] = d[u] + w(u, v) \quad \text{per ogni arco } (u, v) \in T$$

$$d[v] \leq d[u] + w(u, v) \quad \text{per ogni arco } (u, v) \in E$$

allora  $T$  è una soluzione ottima.

- Poichè  $d'[s] = d[s] = 0$ , ma  $d'[u] < d[u]$ , esiste un arco  $(h, k)$  in  $C'$  tale che:
  - $d'[h] = d[h]$  e
  - $d'[k] < d[k]$



# Dimostrazione

## Teorema di Bellman - Parte 2

Se valgono le condizioni di Bellman:

$$d[v] = d[u] + w(u, v) \quad \text{per ogni arco } (u, v) \in T$$

$$d[v] \leq d[u] + w(u, v) \quad \text{per ogni arco } (u, v) \in E$$

allora  $T$  è una soluzione ottima.

- Per costruzione:  $d'[h] = d[h]$
- Per costruzione:  $d'[k] = d'[h] + w(h, k)$
- Per ipotesi:  $d[k] \leq d[h] + w(h, k)$
- Combinando queste due relazioni, si ottiene:

$$d'[k] = d'[h] + w(h, k) = d[h] + w(h, k) \geq d[k]$$

Quindi  $d'[k] \geq d[k]$ , il che contraddice  $d'[k] < d[k]$



## Algoritmo prototipo – Rilassamento

---

```
(int[], int[]) prototipoCamminiMinimi(GRAPH  $G$ , NODE  $s$ )
```

---

```
% Inizializza  $T$  ad una foresta di copertura composta da nodi isolati
```

```
% Inizializza  $d$  con sovrastima della distanza ( $d[s] = 0$ ,  $d[x] = +\infty$ )
```

```
while  $\exists(u, v) : d[u] + G.w(u, v) < d[v]$  do
```

```
     $d[v] = d[u] + G.w(u, v)$   
    % Sostituisci il padre di  $v$  in  $T$  con  $u$ 
```

```
return ( $T, d$ )
```

---

### Note

- Se al termine dell'esecuzione qualche nodo mantiene una distanza infinita, esso non è raggiungibile
- Come implementare la condizione  $\exists$ ?

## Algoritmo generico

---

```
(int[], int[]) shortestPath(GRAPH  $G$ , NODE  $s$ )
```

---

```
int[]  $d$  = new int[1... $G.n$ ]           %  $d[u]$  è la distanza da  $s$  a  $u$ 
int[]  $T$  = new int[1... $G.n$ ]         %  $T[u]$  è il padre di  $u$  nell'albero  $T$ 
boolean[]  $b$  = new boolean[1... $G.n$ ] %  $b[u]$  è true se  $u \in S$ 
foreach  $u \in G.V() - \{s\}$  do
    [
         $T[u] = \mathbf{nil}$ 
         $d[u] = +\infty$ 
         $b[u] = \mathbf{false}$ 
    ]
 $T[s] = \mathbf{nil}$ 
 $d[s] = 0$ 
 $b[s] = \mathbf{true}$ 
[...]
```

---

# Algoritmo generico

---

shortestPath(GRAPH  $G$ , NODE  $s$ ) – Corpo principale

---

```

(1) DATASTRUCTURE  $S = \text{DataStructure}()$ ;  $S.\text{add}(s)$ 
   while not  $S.\text{isEmpty}()$  do
(2)     int  $u = S.\text{extract}()$ 
        $b[u] = \text{false}$ 
       foreach  $v \in G.\text{adj}(u)$  do
(3)         if  $d[u] + G.w(u, v) < d[v]$  then
           if not  $b[v]$  then
(4)              $S.\text{add}(v)$ 
                  $b[v] = \text{true}$ 
           else
               % Azione da svolgere nel caso  $v$  sia già presente in  $S$ 
                  $T[v] = u$ 
                  $d[v] = d[u] + G.w(u, v)$ 
       return  $(T, d)$ 

```

---

# Dijkstra, 1959

## Storia

- Sviluppato da Edsger W. Dijkstra nel 1956, pubblicato nel 1959
- Nella versione originale:
  - Veniva utilizzata per trovare la distanza minima fra due nodi
  - Utilizzava il concetto di coda con priorità
  - Tenete conto però che gli heap sono stati proposti nel 1964

## Note

- Funziona (bene) solo con pesi positivi
- Utilizzato in protocolli di rete come IS-IS e OSPF

# Dijkstra, 1959 – Coda con priorità basata su vettore

## Linea (1): Inizializzazione

- Viene creato un vettore di dimensione  $n$
- L'indice  $u$  rappresenta il nodo  $u$ -esimo
- Le priorità vengono inizializzate ad  $+\infty$
- La priorità di  $s$  è posta uguale a 0
- Costo computazionale:  $O(n)$

---

shortestPath(GRAPH  $G$ , NODE  $s$ ) – Corpo principale

---

- ```
(1) PRIORITYQUEUE  $Q = \text{PriorityQueue}(); Q.\text{insert}(s, 0)$ 
while not  $Q.\text{isEmpty}()$  do
(2)   int  $u = Q.\text{deleteMin}()$ 
       $b[u] = \text{false}$ 
      foreach  $v \in G.\text{adj}(u)$  do
        if  $d[u] + G.w(u, v) < d[v]$  then
           $\lfloor \dots$ 
```

# Dijkstra, 1959 – Coda con priorità basata su vettore

## Linea (2): Estrazione minimo

- Si ricerca il minimo all'interno del vettore
- Una volta trovato, si "cancella" la sua priorità
- Costo computazionale:  $O(n)$

---

shortestPath(GRAPH  $G$ , NODE  $s$ ) – Corpo principale

---

(1) PRIORITYQUEUE  $Q = \text{PriorityQueue}(); Q.\text{insert}(s, 0)$

**while not**  $Q.\text{isEmpty}()$  **do**

(2)  $\left[ \begin{array}{l} u = Q.\text{deleteMin}() \\ b[u] = \text{false} \\ \text{foreach } v \in G.\text{adj}(u) \text{ do} \\ \quad \left[ \begin{array}{l} \text{if } d[u] + G.w(u, v) < d[v] \text{ then} \\ \quad \left[ \dots \right] \end{array} \right. \end{array} \right.$

**return**  $(T, d)$

---

# Dijkstra, 1959 – Coda con priorità basata su vettore

## Linea (3): Inserimento in coda

- Si registra la priorità nella posizione  $v$ -esima
- Costo computazionale:  $O(1)$

---

shortestPath(GRAPH  $G$ , NODE  $s$ ) – Corpo principale

---

[...]

**if**  $d[u] + G.w(u, v) < d[v]$  **then**

**if not**  $b[v]$  **then**

$Q.insert(v, d[u] + G.w(u, v))$

$b[v] = \mathbf{true}$

**else**

        % Azione da svolgere nel caso  $v$  sia già presente in  $S$

$T[v] = u$

$d[v] = d[u] + G.w(u, v)$

[...]

---

# Dijkstra, 1959 – Coda con priorità basata su vettore

## Linea (4): Aggiornamento priorità

- Si aggiorna la priorità nella posizione  $v$ -esima
- Costo computazionale:  $O(1)$

---

shortestPath(GRAPH  $G$ , NODE  $s$ ) – Corpo principale

---

[...]

**if**  $d[u] + G.w(u, v) < d[v]$  **then**

**if not**  $b[v]$  **then**

$Q.insert(v, d[u] + G.w(u, v))$

$b[v] = \mathbf{true}$

**else**

$Q.decrease(v, d[u] + G.w(u, v))$

$T[v] = u$

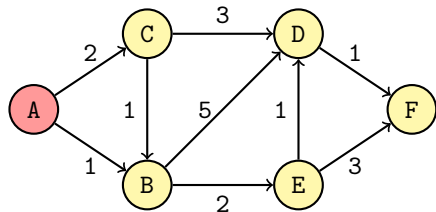
$d[v] = d[u] + G.w(u, v)$

[...]

---



## Dijkstra, 1959 – Coda con priorità basata su vettore



|   |   | A | B            | C            | E            | D            | F            |
|---|---|---|--------------|--------------|--------------|--------------|--------------|
| A | 0 | 0 | <del>∞</del> | <del>∞</del> | <del>∞</del> | <del>∞</del> | <del>∞</del> |
| B | ∞ | 1 | 1            | <del>1</del> | <del>1</del> | <del>1</del> | <del>1</del> |
| C | ∞ | 2 | 2            | 2            | <del>2</del> | <del>2</del> | <del>2</del> |
| D | ∞ | ∞ | 6            | 5            | 4            | 4            | <del>4</del> |
| E | ∞ | ∞ | 3            | 3            | 3            | <del>3</del> | <del>3</del> |
| F | ∞ | ∞ | ∞            | ∞            | 6            | 5            | 5            |

## Spiegazione

- Ogni colonna contiene lo stato del vettore  $d$  all'inizio di ogni ripetizione del ciclo **while not**  $Q.isEmpty()$
- Ogni riga  $v$  rappresenta l'evoluzione dello stato dell'elemento  $d[v]$
- La legenda delle colonne rappresenta il nodo che viene estratto

# Dijkstra

## Correttezza per pesi positivi

- Ogni nodo viene estratto una e una sola volta
- Al momento dell'estrazione la sua distanza è minima

Per induzione sul numero  $k$  di nodi estratti

- Caso base: vero perchè  $d[s] = 0$  e non ci sono pesi negativi
- Ipotesi induttiva: vero per i primi  $k - 1$  nodi
- Passo induttivo: quando viene estratto il  $k$ -esimo nodo  $u$ :
  - La sua distanza  $d[u]$  dipende dai  $k - 1$  nodi già estratti
  - Non può dipendere dai nodi ancora da estrarre, che hanno distanza  $\geq d[u]$
  - Quindi  $d[u]$  è minimo e  $u$  non verrà più re-inserito, perchè non ci sono distanze negative

## Dijkstra, 1959 – Coda con priorità basata su vettore

## Costo computazionale

| Riga | Costo  | Ripet. |
|------|--------|--------|
| (1)  | $O(n)$ | 1      |
| (2)  | $O(n)$ | $O(n)$ |
| (3)  | $O(1)$ | $O(n)$ |
| (4)  | $O(1)$ | $O(m)$ |

Costo totale:  $O(n^2)$

---

**shortestPath**(GRAPH  $G$ , NODE  $s$ )
 

---

```

[...]
```

(1) **PRIORITYQUEUE**  $Q = \text{PriorityQueue}(); Q.\text{insert}(s, 0)$   
**while not**  $Q.\text{isEmpty}()$  **do**

(2)      $u = Q.\text{deleteMin}()$   
            $b[u] = \text{false}$   
           **foreach**  $v \in G.\text{adj}(u)$  **do**

(3)             **if**  $d[u] + G.w(u, v) < d[v]$  **then**  
                   **if not**  $b[v]$  **then**  
                        $Q.\text{insert}(v, d[u] + G.w(u, v))$   
                        $b[v] = \text{true}$   
                   **else**  
                        $Q.\text{decrease}(v, d[u] + G.w(u, v))$

(4)              $T[v] = u$   
                    $d[v] = d[u] + G.w(u, v)$

**return**  $(T, d)$

---

## Johnson, 1977 – Coda con priorità basata su heap binario

## Costo computazionale

| Riga | Costo       | Ripet. |
|------|-------------|--------|
| (1)  | $O(n)$      | 1      |
| (2)  | $O(\log n)$ | $O(n)$ |
| (3)  | $O(\log n)$ | $O(n)$ |
| (4)  | $O(\log n)$ | $O(m)$ |

Costo totale:  $O(m \log n)$

Heap binario introdotto nel  
1964

---

**shortestPath(GRAPH  $G$ , NODE  $s$ )**


---

```

[...]
```

(1) **PRIORITYQUEUE**  $Q = \text{PriorityQueue}(); Q.\text{insert}(s, 0)$

**while not**  $Q.\text{isEmpty}()$  **do**

(2)      $u = Q.\text{deleteMin}()$   
            $b[u] = \text{false}$   
           **foreach**  $v \in G.\text{adj}(u)$  **do**

**if**  $d[u] + G.w(u, v) < d[v]$  **then**

(3)             **if not**  $b[v]$  **then**  
                        $Q.\text{insert}(v, d[u] + G.w(u, v))$   
                        $b[v] = \text{true}$

**else**  
                        $Q.\text{decrease}(v, d[u] + G.w(u, v))$

(4)              $T[v] = u$   
                    $d[v] = d[u] + G.w(u, v)$

**return**  $(T, d)$

---

## Fredman-Tarjan, 1987 – Heap di Fibonacci

## Costo computazionale

| Riga | Costo        | Ripet. |
|------|--------------|--------|
| (1)  | $O(n)$       | 1      |
| (2)  | $O(\log n)$  | $O(n)$ |
| (3)  | $O(1)^{(*)}$ | $O(n)$ |
| (4)  | $O(1)^{(*)}$ | $O(m)$ |

Costo:  $O(m + n \log n)$

(\*) Costo ammortizzato

---

**shortestPath(GRAPH  $G$ , NODE  $s$ )**


---

```

[...]
```

(1) **PRIORITYQUEUE**  $Q = \text{PriorityQueue}(); Q.\text{insert}(s, 0)$

**while not**  $Q.\text{isEmpty}()$  **do**

(2)      $u = Q.\text{deleteMin}()$   
            $b[u] = \text{false}$   
           **foreach**  $v \in G.\text{adj}(u)$  **do**

**if**  $d[u] + G.w(u, v) < d[v]$  **then**

(3)             **if not**  $b[v]$  **then**  
                        $Q.\text{insert}(v, d[u] + G.w(u, v))$   
                        $b[v] = \text{true}$

**else**

(4)                     $Q.\text{decrease}(v, d[u] + G.w(u, v))$

$T[v] = u$   
                        $d[v] = d[u] + G.w(u, v)$

**return**  $(T, d)$

---

# Bellman-Ford-Moore, 1958 – Coda

## Storia

- Proposto da **Alfonso Shimbel** nel 1955
- Pubblicato da **Lester Ford, Jr.** nel 1956
- Pubblicato da **Edward F. Moore** nel 1958
  - **Shortest path faster algorithm**
  - La versione che vediamo qui
- Pubblicato da **Richard Bellman** nel 1958
- Noto come Bellman-Ford, o Bellman-Ford-Moore

## Note

- Computazionalmente più pesante di Dijkstra
- Funziona anche con archi di peso negativo

# Bellman-Ford-Moore, 1958 – Coda

## Linea (1): Inizializzazione

- Viene creata una coda di dimensione  $n$
- Costo computazionale:  $O(1)$

---

`shortestPath(GRAPH  $G$ , NODE  $s$ )` – Corpo principale

---

```
(1) QUEUE  $Q = \text{Queue}(); Q.\text{enqueue}(s)$   
while not  $Q.\text{isEmpty}()$  do  
(2)    $u = Q.\text{dequeue}()$   
       $b[u] = \text{false}$   
      foreach  $v \in G.\text{adj}(u)$  do  
        if  $d[u] + G.w(u, v) < d[v]$  then  
           $\lfloor \quad \lfloor \dots$   
return  $(T, d)$ 
```

---

# Bellman-Ford-Moore, 1958 – Coda

## Linea (2): Estrazione

- Viene estratto il prossimo elemento della coda
- Costo computazionale:  $O(1)$

---

`shortestPath(GRAPH  $G$ , NODE  $s$ )` – Corpo principale

---

```
(1) QUEUE  $Q = \text{Queue}(); Q.\text{enqueue}(s)$ 
    while not  $Q.\text{isEmpty}()$  do
(2)    $u = Q.\text{dequeue}()$ 
       $b[u] = \text{false}$ 
      foreach  $v \in G.\text{adj}(u)$  do
        if  $d[u] + G.w(u, v) < d[v]$  then
          [ ... ]
    return  $(T, d)$ 
```

---



# Bellman-Ford-Moore, 1958 – Coda

## Linea (3): Inserimento in coda

- Si inserisce l'indice  $v$  in coda
- Costo computazionale:  $O(1)$

---

shortestPath(GRAPH  $G$ , NODE  $s$ ) – Corpo principale

---

[...]

**if**  $d[u] + G.w(u, v) < d[v]$  **then**

**if not**  $b[v]$  **then**

$Q.enqueue(v)$

$b[v] = \mathbf{true}$

**else**

        % Azione da svolgere nel caso  $v$  sia già presente in  $S$

$T[v] = u$

$d[v] = d[u] + G.w(u, v)$

[...]

# Bellman-Ford-Moore, 1958 – Coda

**Linea (4): Azione nel caso  $v$  sia già presente in  $S$**

- Sezione non necessaria

---

shortestPath(GRAPH  $G$ , NODE  $s$ ) – Corpo principale

---

[...]

**if**  $d[u] + G.w(u, v) < d[v]$  **then**

**if not**  $b[v]$  **then**

$Q.enqueue(v)$

$b[v] = \mathbf{true}$

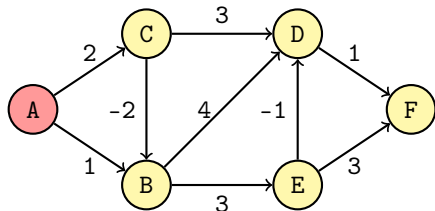
$T[v] = u$

$d[v] = d[u] + G.w(u, v)$

[...]

---

## Bellman-Ford-Moore, 1958 – Coda



- La prima riga contiene l'elemento estratto dalla coda
- L'ultima riga contiene lo stato della coda

|   |          | A        | B        | C        | D   | E   | B   | F  | D | E | D | F |
|---|----------|----------|----------|----------|-----|-----|-----|----|---|---|---|---|
| A | 0        | 0        | 0        | 0        | 0   | 0   | 0   | 0  | 0 | 0 | 0 | 0 |
| B | $\infty$ | 1        | 1        | 0        | 0   | 0   | 0   | 0  | 0 | 0 | 0 | 0 |
| C | $\infty$ | 2        | 2        | 2        | 2   | 2   | 2   | 2  | 2 | 2 | 2 | 2 |
| D | $\infty$ | $\infty$ | 5        | 5        | 5   | 3   | 3   | 3  | 3 | 2 | 2 | 2 |
| E | $\infty$ | $\infty$ | 4        | 4        | 4   | 4   | 3   | 3  | 3 | 3 | 3 | 3 |
| F | $\infty$ | $\infty$ | $\infty$ | $\infty$ | 6   | 6   | 6   | 6  | 4 | 4 | 3 | 3 |
| Q | A        | BC       | CDE      | DEB      | EBF | BFD | FDE | DE | E | D | F |   |

# Bellman-Ford-Moore, 1958 – Coda

## Passata - definizione ricorsiva

- Per  $k = 0$ , la zeresima passata consiste nell'estrazione del nodo  $s$  dalla coda  $S$ ;
- Per  $k > 0$ , la  $k$ -esima passata consiste nell'estrazione di tutti i nodi presenti in  $S$  al termine della passata  $k - 1$ -esima.

## Correttezza – intuizione

- Al termine della passata  $k$ , i vettori  $T$  e  $d$  descrivono i cammini minimi composti da al più  $k$  archi
- Al termine della passata  $n - 1$ , i vettori  $T$  e  $d$  descrivono i cammini minimi (composti da al più  $n - 1$  archi)

## Bellman-Ford-Moore, 1958 – Coda

## Costo computazionale

| Riga | Costo  | Ripet.   |
|------|--------|----------|
| (1)  | $O(1)$ | 1        |
| (2)  | $O(1)$ | $O(n^2)$ |
| (3)  | $O(1)$ | $O(nm)$  |

Costo:  $O(nm)$

Ogni nodo può essere inserito ed estratto al massimo  $n - 1$  volte

---

shortestPath(GRAPH  $G$ , NODE  $s$ ) – Corpo principale

---

```

(1) QUEUE  $Q = \text{Queue}(); Q.\text{enqueue}(s)$ 
while not  $Q.\text{isEmpty}()$  do
(2)    $u = Q.\text{dequeue}()$ 
       $b[u] = \text{false}$ 
      foreach  $v \in G.\text{adj}(u)$  do
        if  $d[u] + G.w(u, v) < d[v]$  then
          if not  $b[v]$  then
            (3)    $Q.\text{enqueue}(v)$ 
                   $b[v] = \text{true}$ 
                   $T[v] = u$ 
                   $d[v] = d[u] + G.w(u, v)$ 
      return  $(T, d)$ 

```

---

# Cammini minimi su DAG

## Osservazione

I cammini minimi in un DAG sono sempre ben definiti; anche in presenza di pesi negativi, non esistono cicli negativi

Come risolvere il problema?

# Cammini minimi su DAG

## Osservazione

I cammini minimi in un DAG sono sempre ben definiti; anche in presenza di pesi negativi, non esistono cicli negativi

## Come risolvere il problema?

È possibile rilassare gli archi in ordine topologico, una volta sola. Non essendoci cicli, non c'è modo di tornare su un nodo già visitato e abbassare il valore del suo campo  $d$

## Algoritmo

- Si utilizza l'ordinamento topologico

# Cammini minimi su DAG

---

```
(int[], int[]) shortestPath(GRAPH  $G$ , NODE  $s$ )
```

---

```
int[]  $d$  = new int[1... $G.n$ ]           %  $d[u]$  è la distanza da  $s$  a  $u$ 
```

```
int[]  $T$  = new int[1... $G.n$ ] %  $T[u]$  è il padre di  $u$  nell'albero  $T$ 
```

```
foreach  $u \in G.V() - \{s\}$  do
```

```
   $T[u] = \text{nil}; d[u] = +\infty$ 
```

```
 $T[s] = \text{nil}; d[s] = 0$ 
```

```
STACK  $S$  = topsort( $G$ )
```

```
while not  $S$ .isEmpty() do
```

```
   $u = S.pop()$ 
```

```
  foreach  $v \in G.adj(u)$  do
```

```
    if  $d[u] + G.w(u, v) < d[v]$  then
```

```
       $T[v] = u$ 
```

```
       $d[v] = d[u] + G.w(u, v)$ 
```

```
return ( $T, d$ )
```

---



# Riassunto

Complessità: quale preferire?

|                                            |                          |                                                        |
|--------------------------------------------|--------------------------|--------------------------------------------------------|
| BFS                                        | $O(m + n)$               | Senza pesi                                             |
| Dijkstra                                   | $O(n^2)$                 | Pesi positivi, grafi densi                             |
| Johnson                                    | $O(m \log n)$            | Pesi positivi, grafi sparsi                            |
| Fredman-Tarjan                             | $O(m + n \log n)$        | Pesi positivi, grafi densi,<br>dimensioni molto grandi |
| Bellman-Ford                               | $O(mn)$                  | Pesi negativi                                          |
|                                            | $O(m + n)$               | DAG                                                    |
| Bernstein, Nanongkai<br>Wulf-Nilsen (2022) | $O(m \log^8 \nu \log W)$ | Pesi negativi, interi                                  |

# Sommario

- 1 Introduzione
- 2 Cammini minimi, sorgente singola
  - Dijkstra
  - Johnson
  - Fredman-Tarjan
  - Bellman-Ford-Moore
  - Casi speciali – DAG
- 3 Cammini minimi, sorgente multipla
  - Floyd-Warshall
  - Chiusura transitiva
- 4 Conclusione

# Cammini minimi, sorgente multipla

## Possibili soluzioni

| Input                          | Complessità             | Approccio                                                  |
|--------------------------------|-------------------------|------------------------------------------------------------|
| Pesi positivi,<br>grafo denso  | $O(n \cdot n^2)$        | Applicazione ripetuta dell'algoritmo di Dijkstra           |
| Pesi positivi,<br>grafo sparso | $O(n \cdot (m \log n))$ | Applicazione ripetuta dell'algoritmo di Johnson            |
| Pesi negativi                  | $O(n \cdot nm)$         | Applicazione ripetuta di Bellman-Ford, <b>sconsigliata</b> |
| Pesi negativi,<br>grafo denso  | $O(n^3)$                | Algoritmo di <b>Floyd e Warshall</b>                       |
| Pesi negativi,<br>grafo sparso | $O(nm \log n)$          | Algoritmo di <b>Johnson per sorgente multipla</b>          |

# Floyd-Warshall, 1962

## Cammini minimi $k$ -vincolati

Sia  $k$  un valore in  $\{0, \dots, n\}$ . Diciamo che un cammino  $p_{xy}^k$  è un **cammino minimo  $k$ -vincolato** fra  $x$  e  $y$  se esso ha il costo minimo fra tutti i cammini fra  $x$  e  $y$  che non passano per nessun vertice in  $v_{k+1}, \dots, v_n$  ( $x$  e  $y$  sono esclusi dal vincolo).

## Note

Assumiamo (come abbiamo sempre fatto) che esista un ordinamento fra i nodi del grafo  $v_1, v_2, \dots, v_n$ .

## Domande

- A cosa corrisponde  $p_{xy}^0$ ?
- A cosa corrisponde  $p_{xy}^n$ ?

# Floyd-Warshall, 1962

## Distanza $k$ -vincolata

Denotiamo con  $d^k[x][y]$  il costo totale del cammino minimo  $k$ -vincolato fra  $x$  e  $y$ , se esiste.

$$d^k[x][y] = \begin{cases} w(p_{xy}^k) & \text{se esiste } p_{xy}^k \\ +\infty & \text{altrimenti} \end{cases}$$

## Domande

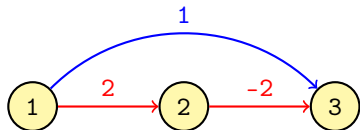
- A cosa corrisponde  $d^0[x][y]$ ?
- A cosa corrisponde  $d^n[x][y]$ ?

# Floyd-Warshall, 1962

## Formulazione ricorsiva

$$d^k[x][y]) = \begin{cases} w(x, y) & k = 0 \\ & k > 0 \end{cases}$$

## Esempio



$$d^0[1][3] =$$

$$d^1[1][3] =$$

$$d^2[1][3] =$$

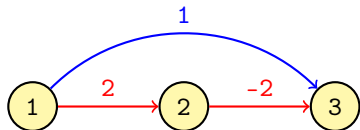
$$=$$

# Floyd-Warshall, 1962

## Formulazione ricorsiva

$$d^k[x][y] = \begin{cases} w(x, y) & k = 0 \\ \min(d^{k-1}[x][y], d^{k-1}[x][k] + d^{k-1}[k][y]) & k > 0 \end{cases}$$

## Esempio



$$d^0[1][3] = 1$$

$$d^1[1][3] = 1$$

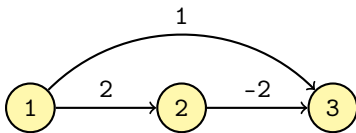
$$\begin{aligned} d^2[1][3] &= \min(d^1[1][3], d^1[1][2] + d^1[2][3]) \\ &= \min(1, 0) = 0 \end{aligned}$$

# Floyd-Warshall, 1962

## Matrice dei padri

Oltre a definire la matrice  $d$ , calcoliamo una matrice  $T$  dove  $T[x][y]$  rappresenta il predecessore di  $y$  nel cammino più breve da  $x$  a  $y$ .

## Esempio



$$T[1][2] = 1$$

$$T[2][3] = 2$$

$$T[1][3] = 2$$



# Floyd-Warshall, programmazione dinamica

---

```
(int[][], int[][]) floydWarshall(GRAPH G)
```

---

```
int[][] d = new int[1...n][1...n]
```

```
int[][] T = new int[1...n][1...n]
```

```
foreach  $u, v \in G.V()$  do
```

```
┌  $d[u][v] = +\infty$ 
```

```
└  $T[u][v] = \text{nil}$ 
```

```
foreach  $u \in G.V()$  do
```

```
┌ foreach  $v \in G.adj(u)$  do
```

```
└  $d[u][v] = G.w(u, v)$ 
```

```
└  $T[u][v] = u$ 
```

---

# Floyd-Warshall, programmazione dinamica

---

```
(int[][], int[][]) floydWarshall(GRAPH G)
```

---

```
[...]
```

```
for  $k = 1$  to  $G.n$  do
```

```
    foreach  $u \in G.V()$  do
```

```
        foreach  $v \in G.V()$  do
```

```
            if  $d[u][k] + d[k][v] < d[u][v]$  then
```

```
                 $d[u][v] = d[u][k] + d[k][v]$ 
```

```
                 $T[u][v] = T[k][v]$ 
```

```
return ( $d, T$ )
```

---

# Chiusura transitiva (Algoritmo di Warshall)

## Chiusura transitiva

La chiusura transitiva  $G^* = (V, E^*)$  di un grafo  $G = (V, E)$  è il grafo orientato tale che  $(u, v) \in E^*$  se e solo esiste un cammino da  $u$  a  $v$  in  $G$ .

Supponendo di avere il grafo  $G$  rappresentato da una matrice di adiacenza  $M$ , la matrice  $M^n$  rappresenta la matrice di adiacenza di  $G^*$ .

## Formulazione ricorsiva

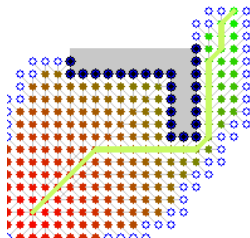
$$M^k[x][y] = \begin{cases} M[x][y] & k = 0 \\ M^{k-1}[x][y] \text{ or } (M^{k-1}[x][k] \text{ and } M^{k-1}[k][y]) & k > 0 \end{cases}$$

# Sommario

- 1 Introduzione
- 2 Cammini minimi, sorgente singola
  - Dijkstra
  - Johnson
  - Fredman-Tarjan
  - Bellman-Ford-Moore
  - Casi speciali – DAG
- 3 Cammini minimi, sorgente multipla
  - Floyd-Warshall
  - Chiusura transitiva
- 4 Conclusione

# Conclusione

- Abbiamo visto una panoramica dei più importanti algoritmi per la ricerca dei cammini minimi
- Ulteriori possibilità:
  - A\*, un algoritmo che utilizza euristiche per velocizzare la ricerca
  - Algoritmi specializzati per reti stradali



[https://en.wikipedia.org/wiki/A\\*\\_search\\_algorithm#/media/File:Astar\\_progress\\_animation.gif](https://en.wikipedia.org/wiki/A*_search_algorithm#/media/File:Astar_progress_animation.gif)