

Algoritmi e Strutture Dati

Scelta della struttura dati

Alberto Montresor

Università di Trento

2019/02/18

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 sorgente unica

- **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 che abbiano tempo di esecuzione migliore.

Panoramica sul problema

Cammini mimimi 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

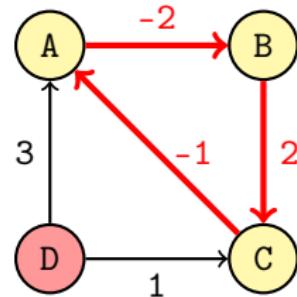
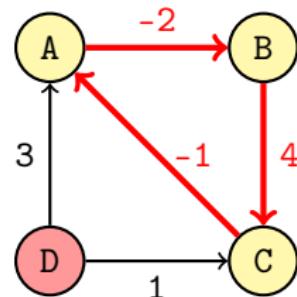
Algoritmi diversi possono funzionare oppure no in caso di alcune categorie speciali di pesi

- Positivi / positivi+negativi
- Reali / interi

Pesi negativi vs grafi con cicli negativi

Esempio: proprietario di un TIR

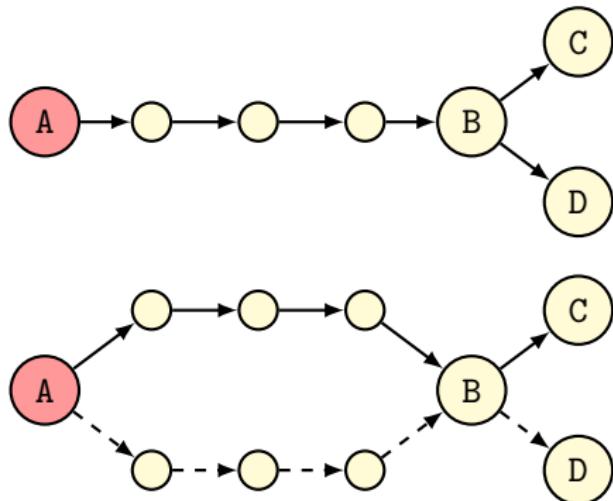
- Viaggare scarico: perdita, peso positivo
- Viaggare carico: profitto, peso negativo



Problema cammini minimi – Sottostruttura ottima

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

... ma non possono convergere in un nodo comune B dopo aver percorso un tratto iniziale distinto



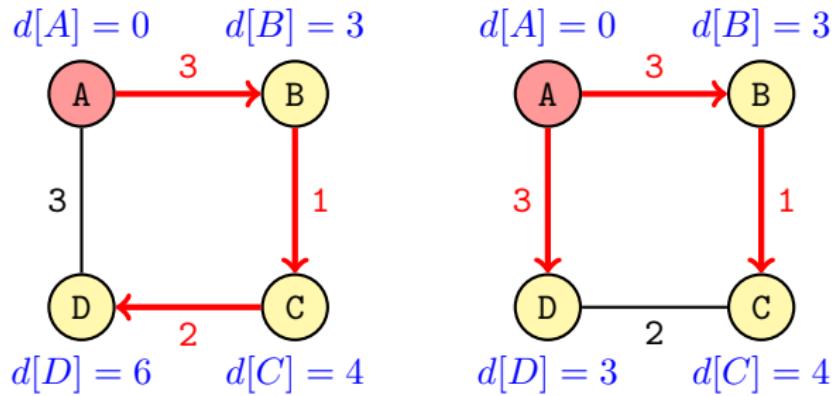
Albero dei cammini minimi

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 .

Soluzione ammissibile

Soluzione ammissibile

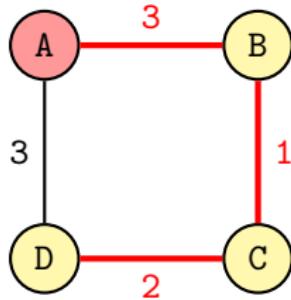
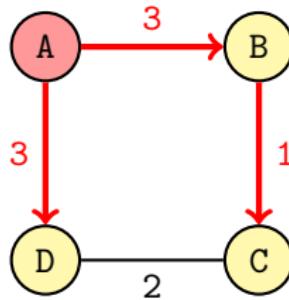
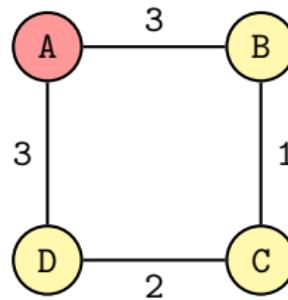
Una soluzione **ammissibile** può essere descritta da un **albero di copertura T** radicato in s e da un **vettore di distanza d** , i cui valori $d[u]$ rappresentano il costo del cammino da s a u in T .



Cammini minimi / alberi di copertura di peso minimo

Questi due alberi di copertura sono identici?

- un **albero dei cammini minimi da singola sorgente A**
- un **albero di copertura di peso minimo**



Rappresentazione 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 p[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:

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

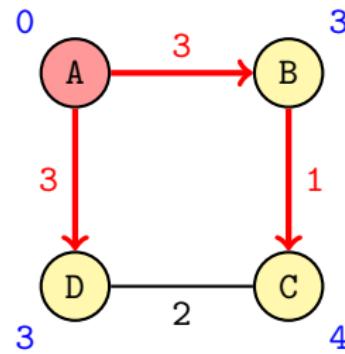
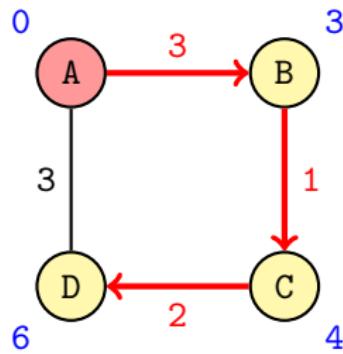
$$d[v] \leq d[u] + w(u, v) \quad \text{per ogni arco } (u, v) \in E$$

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

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

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

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



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 $d[v] \leq d[u] + w(u, v)$, perché 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 il cammino da s a u in T non sia ottimo
- Allora esiste un cammino da s a u con distanza $d'[u] < d[u]$
- Sia $d'[v]$ la distanza da s ad un generico nodo v che appare in tale cammino
- Poiché $d'[s] = d[s] = 0$, ma $d'[u] < d[u]$, esiste un arco (h, k) per cui $d'[h] \geq d'[h]$ 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 $d'[h] + w(h, k) = d'[k]$
- Per ipotesi $d[h] + w(h, k) \geq d[k]$
- Combinando queste due relazioni, si ottiene:

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

che contraddice l'ipotesi

Algoritmo prototipo

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

% Inizializza T ad una foresta di copertura composta da nodi isolati
% Inizializza d con una sovrastima della distanza ($d[s] = 0$,
 $d[x] = +\infty$)
while $\exists(u, v) : d[u] + G.w(u, v) < d[v]$ **do**
 $d[v] = d[u] + w(u, v)$
 % 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 ∈ S
foreach u ∈ G.V() − {s} do

```

$T[u] = \text{nil}$
 $d[u] = +\infty$
 $b[u] = \text{false}$

$T[s] = \text{nil}$

$$d[s] = 0$$

$b[s] = \text{true}$

[...]

Algoritmo generico

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

```

(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
                  if  $d[u] + G.w(u, v) < d[v]$  then
                      if not  $b[v]$  then
                          (3)     $S.\text{add}(v)$ 
                           $b[v] = \text{true}$ 
                      else
                          (4)    % 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: $O(n)$

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

(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**
 [...]

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: $O(n)$

(int[], int[]) 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**
 $\quad \quad \quad \ldots$

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: $O(1)$

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

[...]

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

 if not $b[v]$ then

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

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

 else

 (4) % 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: $O(1)$

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

[...]

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

 if not $b[v]$ then

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

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

 else

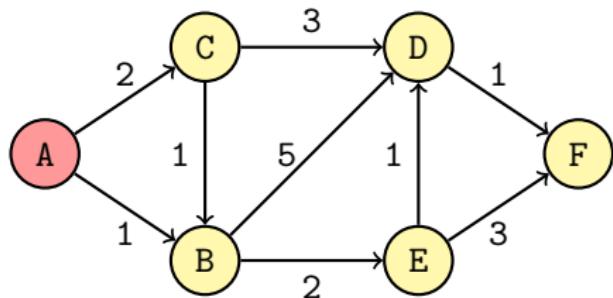
$Q.decrease(v, d[u] + G.w(u, v))$

$T[v] = u$

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

[...]

Dijkstra, 1959 – Coda con priorità basata su vettore



	A	B	C	E	D	F
A	0	0	0	0	0	0
B	∞	1	1	1	1	1
C	∞	2	2	2	2	2
D	∞	∞	6	5	4	4
E	∞	∞	3	3	3	3
F	∞	∞	∞	∞	6	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 lunghezze negative
- 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 ha 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  $Q = \text{PriorityQueue}(); Q.\text{insert}(s, 0)$ 
(2) while not  $Q.\text{isEmpty}()$  do
      $u = Q.\text{deleteMin}()$ 
      $b[u] = \text{false}$ 
     foreach  $v \in G.\text{adj}(u)$  do
         if  $d[u] + G.w(u, v) < d[v]$  then
             if not  $b[v]$  then
                  $Q.\text{insert}(v, d[u] + G.w(u, v))$ 
                  $b[v] = \text{true}$ 
             else
                  $Q.\text{decrease}(v, d[u] + G.w(u, v))$ 
          $T[v] = u$ 
          $d[v] = d[u] + G.w(u, v)$ 
    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  $Q = \text{PriorityQueue}(); Q.\text{insert}(s, 0)$ 
(2) while not  $Q.\text{isEmpty}()$  do
      $u = Q.\text{deleteMin}()$ 
      $b[u] = \text{false}$ 
     foreach  $v \in G.\text{adj}(u)$  do
         if  $d[u] + G.w(u, v) < d[v]$  then
             if not  $b[v]$  then
                  $Q.\text{insert}(v, d[u] + G.w(u, v))$ 
                  $b[v] = \text{true}$ 
             else
                  $Q.\text{decrease}(v, d[u] + G.w(u, v))$ 
          $T[v] = u$ 
          $d[v] = d[u] + G.w(u, v)$ 
(3)
(4)
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)	$O(\log n)$	$O(n)$
(4)	$O(1)^{(*)}$	$O(m)$

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

(*) Costo ammortizzato

shortestPath(GRAPH G , NODE s)

```

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

```

Bellman-Ford, Moore, 1958 – Coda

Storia

- Proposto da Alfonso Shimbel nel 1955
- Pubblicato da Lester Ford, Jr. nel 1956
- Pubblicato da Moore nel 1957
- 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: $O(n)$

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

(1) **QUEUE** $Q = \text{Queue}(); Q.\text{enqueue}(s)$

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

(2) **int** $u = Q.\text{extract}()$

$b[u] = \text{false}$

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 (2): Estrazione

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

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

(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: $O(1)$

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

[...]

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

 if not $b[v]$ then

 Q.enqueue(v)

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

 else

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

$T[v] = u$

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

[...]

Bellman-Ford, Moore, 1958 – Coda

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

- Sezione non necessaria

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

[...]

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

 if not $b[v]$ then

$Q.\text{enqueue}(v)$
 $b[v] = \text{true}$

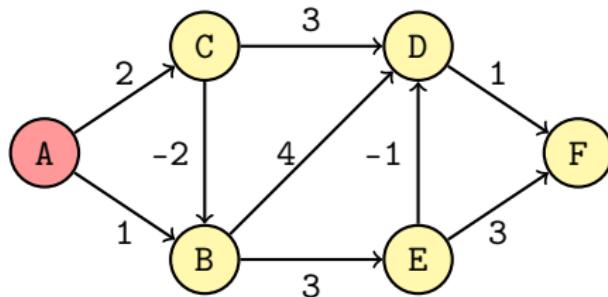
$T[v] = u$

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

(3)

[...]

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
B	∞	1	1	0	0	0	0	0	0	0	0
C	∞	2	2	2	2	2	2	2	2	2	2
D	∞	∞	5	5	5	3	3	3	2	2	2
E	∞	∞	4	4	4	4	3	3	3	3	3
F	∞	∞	∞	∞	6	5	5	4	4	3	3
S	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 di lunghezza al più k
- Al termine della passata $n - 1$, i vettori T e d descrivono i cammini minimi (di lunghezza al più $n - 1$)

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

```

(int[], int[])

shortestPath(GRAPH G, NODE s)

```

```

(1) QUEUE Q = Queue(); Q.enqueue(s)
while not Q.isEmpty() do

```

```

(2)   u = Q.dequeue()
      b[u] = false
      foreach v ∈ G.adj(u) do
        if d[u] + G.w(u, v) < d[v] then
          if not b[v] then
            Q.enqueue(v)
            b[v] = true
          T[v] = u
          d[v] = d[u] + G.w(u, v)

```

```

(3)   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
- E' 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[]  $d = \text{new int}[1 \dots G.n]$            %  $d[u]$  è la distanza da  $s$  a  $u$ 
int[]  $T = \text{new int}[1 \dots G.n]$  %  $T[u]$  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  $S = \text{topsort}(G)$ 
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?

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, dimensioni molto grandi
Bellman-Ford	$O(mn)$	Pesi negativi
	$O(m + n)$	DAG
BFS	$O(m + n)$	Senza pesi

Cammini minimi, sorgente multipla

Possibili soluzioni

Input	Complessità	Approccio
Pesi positivi, grafo denso	$O(n \cdot n^2)$	Applicazione ripetuta dell'algoritmo di Dijkstra
Pesi positivi, 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, sconsigliata
Pesi negativi, grafo denso	$O(n^3)$	Algoritmo di Floyd e Warshall
Pesi negativi, grafo sparso	$O(nm \log n)$	Algoritmo di Johnson per sorgente multipla

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

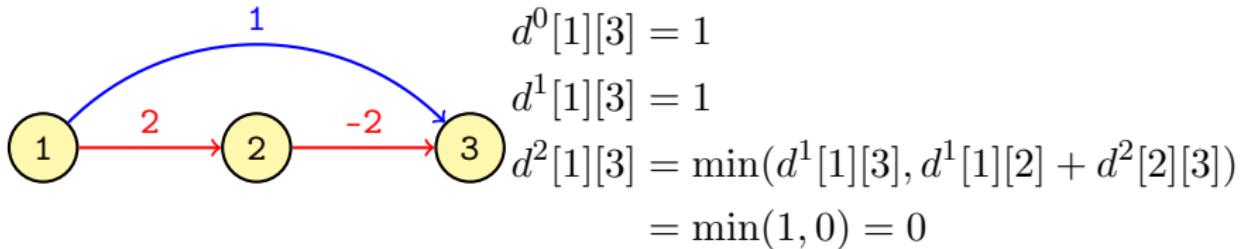
- A cosa corrisponde $d^0[x][y]$?
- A cosa corrisponde $d^n[x][y]$?

Floyd-Warshall, 1962

Formulazione ricorsiva

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

Esempio

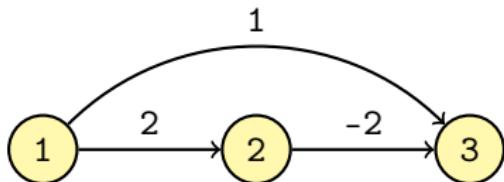


Floyd-Warshall, 1962

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[][] $d = \text{new int}[1 \dots n][1 \dots n]$

int[][] $T = \text{new int}[1 \dots n][1 \dots n]$

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

$d[u][v] = +\infty$

$T[u][v] = \text{nil}$

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

foreach $v \in G.\text{adj}(u)$ **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

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 .

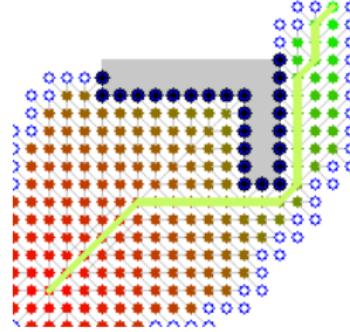
Supponendo di avere il grafo G rappresentato da una matrice di adiacenza M , la matrice M^n rappresenta la matrice di adiacenza di G^* .

Formulazione ricorsiva

$$M^k[x][y] = \begin{cases} M[x][y] & k = 0 \\ M^{k-1}[x][y] \vee M^{k-1}[x][k] \wedge M^{k-1}[k][y] & k > 0 \end{cases}$$

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