

TECNICO SUPERIORE WEB DEVELOPER FULL STACK

The background features a collage of various technology-related icons and text overlays. At the top left is a smartphone icon with a gear inside. To its right is a laptop icon with a gear inside. Below these are icons for 'TESTING FEATURES' (a smartphone with a gear), 'WEB ANALYTICS' (a magnifying glass over a bar chart), 'RESPONSIVE DESIGN' (a smartphone with binary code), 'CONTENT MANAGEMENT' (a smartphone with three gears), and 'MIGRATION' (a smartphone with three gears). The word 'GS' is visible at the top center. The overall theme is digital development and technology.

#13 - Grafi: Cammini minimi

Introduzione

Gli algoritmi che abbiamo visto sono tutti applicati a grafi non pesati, ossia dove ogni arco ha lo stesso “peso” degli altri. I cammini quindi si differenziano solo per il numero di archi da attraversare per arrivare dalla sorgente alla destinazione.

BFS, lo pseudocodice

```
distance(GRAPH G, NODE r, int[] distance)
  QUEUE Q = Queue()
  Q.enqueue(r)
  foreach u ∈ G.V() – {r} do
    [ distance[u] = ∞
    distance[r] = 0
    while not Q.isEmpty() do
      NODE u = Q.dequeue()
      foreach v ∈ G.adj(u) do
        if distance[v] == ∞ then
          [ distance[v] = distance[u] + 1
            Q.enqueue(v)
```

BFS, complessità computazionale

- Ognuno degli n nodi viene inserito nella coda al massimo una volta.
- Ogni volta che un nodo viene estratto, tutti i suoi archi vengono analizzati una volta sola.
- Il numero di archi analizzati è quindi

$$m = \sum_{u \in V} d_{out}(u)$$

dove d_{out} è l'out-degree del nodo u .

- La complessità è quindi $O(m+n)$

E se i grafi fossero pesati?

Cos'è un grafo pesato? Formalmente, è un grafo orientato $G=(V,E)$, insieme ad una **funzione di peso** $w:E\rightarrow R$ che associa ad ogni arco del grafo un valore numerico chiamato "peso" dell'arco.

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

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

L'obiettivo è trovare un cammino dalla sorgente s verso ogni nodo u il cui costo sia minimo, ovvero più piccolo o uguale del costo di qualunque altro cammino da s a u (CM da sorgente unica).

Pesi

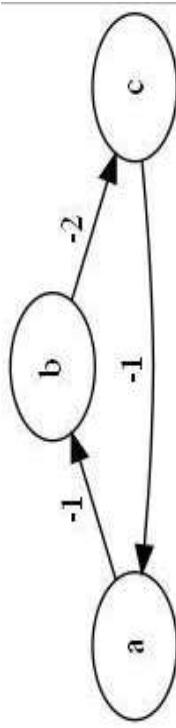
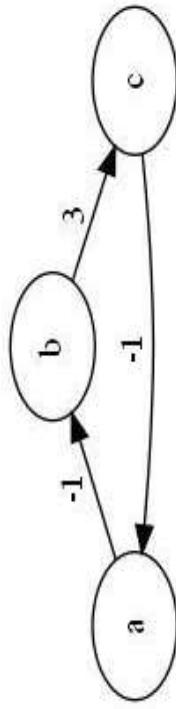
6

Il primo problema è definire quali sono i pesi “accettabili”: potremmo averne di diverse tipologie e gli algoritmi possono avere un funzionamento diverso a seconda del tipo di peso:

- Positivi VS positivi e negativi
- Reali VS interi
- Pesi negativi (OK)

VS

Cicli negativi (NO)



Cosa vogliamo ottenere?

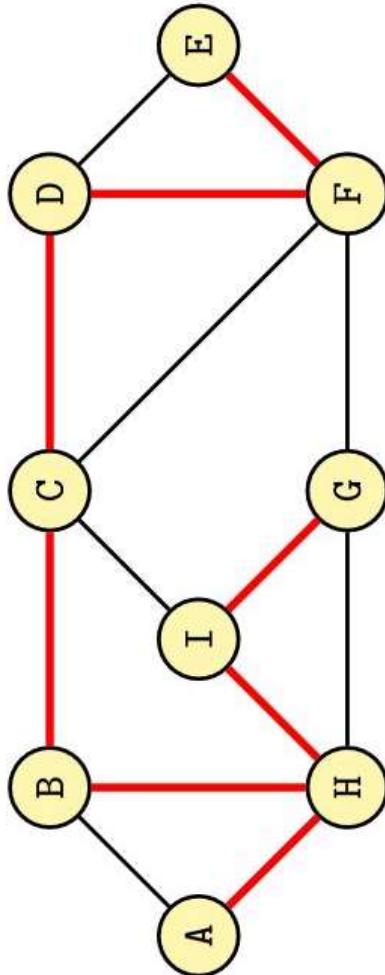
Una soluzione al nostro problema è descritta da un albero di copertura dei cammini minimi T radicato in s e da un vettore di distanza d , i cui valori $d[u]$ rappresentano il costo del cammino da s a u in T .

Ma cos'è un albero di copertura dei cammini minimi?

Albero di copertura dei cammini minimi

Un albero di copertura è un sottografo $T=(V, E_T)$ tale che T è un albero, $E_T \subseteq E$, e T contiene tutti i vertici di G .

In un albero di copertura dei cammini minimi, il cammino dalla sorgente s ad ogni foglia u corrisponde al cammino minimo nel grafo G da s a u .



Rappresentazione

Per rappresentare il vettore delle distanze d [] usiamo un normale vettore.

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

Cammini minimi: l'idea

1. Si inizializza l'albero T ad una foresta di copertura composta da nodi isolati;
2. Si inizializza il vettore $d[\cdot]$ con una sovrastima della distanza $(d[s] = 0, d[x] = \infty)$;
3. Si visita il grafo e, se esiste un arco (u, v) tale per cui la distanza attuale dalla sorgente a v è maggiore della distanza da s a $u +$ il peso dell'arco (u, v) , allora il cammino minimo da s a v passa per u ;
4. Imposta u come padre di v nel vettore dei padri;
5. Se al termine dell'esecuzione qualche nodo mantiene una distanza infinita, esso non è raggiungibile.

Cammini minimi: inizializzazione

11

```
(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] = nil
    d[u] = +∞
    b[u] = false
T[s] = nil
d[s] = 0
b[s] = true
[...]
```

Cammini minimi, quante versioni?

In breve: molte.

In "meno breve": ne vedremo due un po' più in dettaglio, più qualche cenno veloce ad un'altra.

- Dijkstra, 1959
- Johnson, 1977
- Bellman-Ford-Moore, 1958
- Fredman-Tarjan, 1987

Cammini minimi: lo pseudocodice generale

```
(int[], int[]) shortestPath(GRAPH G, NODE s)
(1) DATASTRUCTURE S = DataStructure(); S.add(s)
    while not S.isEmpty() do
        (2) int u = S.extract()
            b[u] = false
            foreach v ∈ G.adj(u) do
                if d[u] + G.w(u, v) < d[v] then
                    (3) if not b[v] then
                        S.add(v)
                        b[v] = 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

Nella versione originale veniva utilizzato per trovare la distanza minima fra due nodi.

- Funziona (bene) solo con **pesi positivi**
- Utilizzato in **protocolli di rete** come IS-IS e OSPF
- Utilizzava il concetto di *coda con priorità*

Coda con priorità?

15

Una coda con priorità (Priority Queue) è una struttura dati astratta, simile ad una coda, in cui ogni elemento inserito possiede una sua “priorità”, ossia un indicatore di precedenza.

L'estrazione quindi non restituisce l'elemento “in testa”, ma quello con il valore di precedenza più basso/alto.

- **Min-priority queue:** estrazione per valori crescenti di priorità;
- **Max-priority queue:** estrazione per valori decrescenti di priorità.

Dijkstra - 1: Inizializzazione

16

Usa una coda con priorità basata su un vettore di dimensione n , dove la priorità è la distanza da s (un nodo vicino ha priorità più bassa di uno lontano, e quindi viene scelto prima). Le priorità vengono inizializzate a $+\infty$ per tutti i nodi tranne $s=0$.

```
(int[], int[]) shortestPath(GRAPH G, NODE s)
(1) PRIORITYQUEUE Q = PriorityQueue(); Q.insert(s, 0)
    while not Q.isEmpty() do
        u = Q.deleteMin()
        b[u] = false
        foreach v ∈ G.adj(u) do
            if d[u] + G.w(u, v) < d[v] then
                [...]
return (T, d)
```

Dijkstra - 2: Estrazione del minimo

L'estrazione del minimo avviene tramite ricerca lineare, cercando il minimo all'interno del vettore. Una volta trovato, si "cancella" la sua priorità (ad es. inserendo -1, a significare "questo nodo non è da considerare").

```
(int[], int[]) shortestPath(GRAPH G, NODE s)
_____
(1) PRIORITYQUEUE Q = PriorityQueue(); Q.insert(s, 0)
    while not Q.isEmpty() do
        (2)   u = Q.deleteMin()
              b[u] = false
              foreach v ∈ G.adj(u) do
                  if d[u] + G.w(u, v) < d[v] then
                      [...]
    _____
    return (T, d)
```

Dijkstra - 3: Inserimento in coda

Inserisco un elemento in coda registrando la priorità (cioè la distanza da s) nella posizione v -esima del vettore.

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

Dijkstra - 4: Aggiornamento della priorità

Si aggiorna la priorità nella posizione v -esima

(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

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

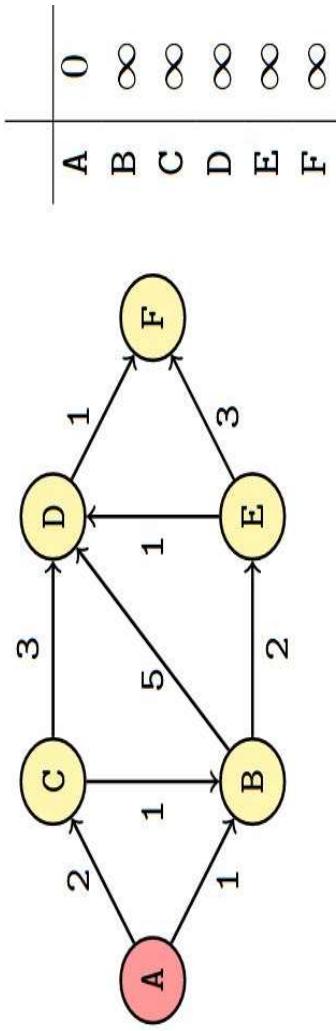
$T[v] = u$

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

[...]

Esempio di funzionamento

20



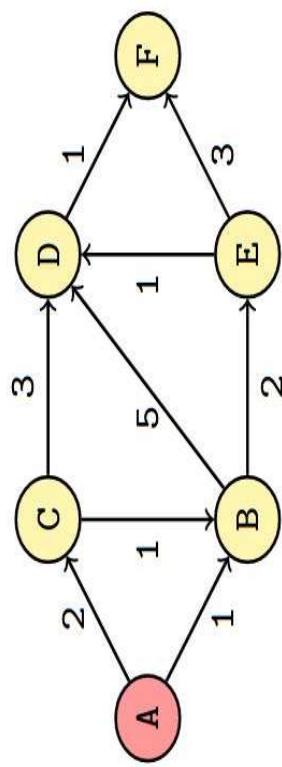
Ogni colonna contiene lo stato del vettore dall'inizio di ogni ripetizione del ciclo while not Q.isEmpty()

Ogni riga rappresenta l'evoluzione dello stato dell'elemento d [v]

La legenda delle colonne rappresenta il nodo che viene estratto **nivello**
Innovation and knowledge
ITS DIGITAL ACADEMY
MARIO VOLPATO

Esempio di funzionamento

21



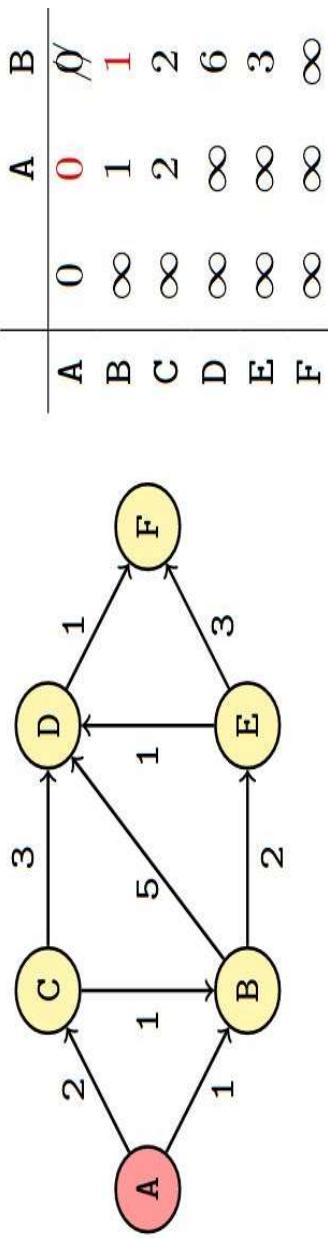
A	0	0
B	∞	1
C	∞	2
D	∞	∞
E	∞	∞
F	∞	∞

Ogni colonna contiene lo stato del vettore dall'inizio di ogni ripetizione del ciclo while not Q.isEmpty()

Ogni riga rappresenta l'evoluzione dello stato dell'elemento d [v]

Esempio di funzionamento

22

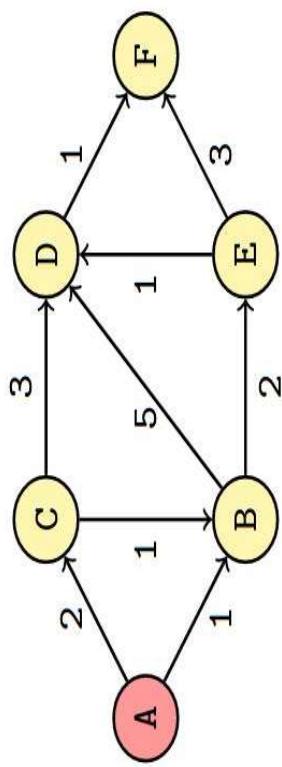


Ogni colonna contiene lo stato del vettore dall'inizio di ogni ripetizione del ciclo while not Q.isEmpty()

Ogni riga rappresenta l'evoluzione dello stato dell'elemento d [v]

La legenda delle colonne rappresenta il nodo che viene estratto **n i u k <>**
ITS DIGITAL
ACADEMY
Mario Tricarico

Esempio di funzionamento



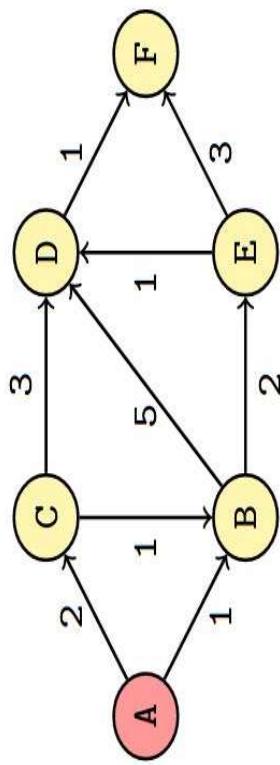
	A	B	C
A	0	0	0
B	∞	1	1
C	∞	2	2
D	∞	∞	6
E	∞	∞	3
F	∞	∞	∞

Ogni colonna contiene lo stato del vettore dall'inizio di ogni ripetizione del ciclo while not Q.isEmpty()

Ogni riga rappresenta l'evoluzione dello stato dell'elemento d [v]

Esempio di funzionamento

24



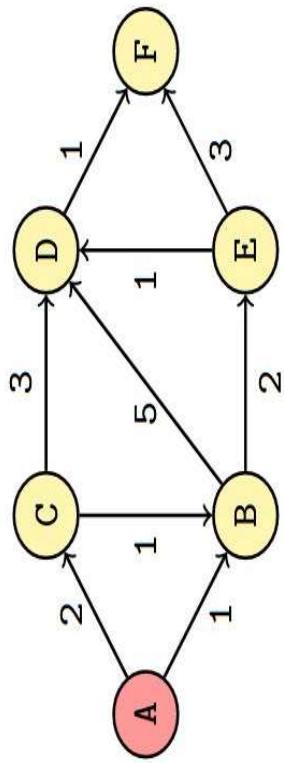
	A	B	C	E
A	0	0	0	0
B	∞	1	1	1
C	∞	2	2	2
D	∞	∞	6	4
E	∞	∞	3	3
F	∞	∞	∞	6

Ogni colonna contiene lo stato del vettore dall'inizio di ogni ripetizione del ciclo while not Q.isEmpty()

Ogni riga rappresenta l'evoluzione dello stato dell'elemento d [v]

Esempio di funzionamento

25



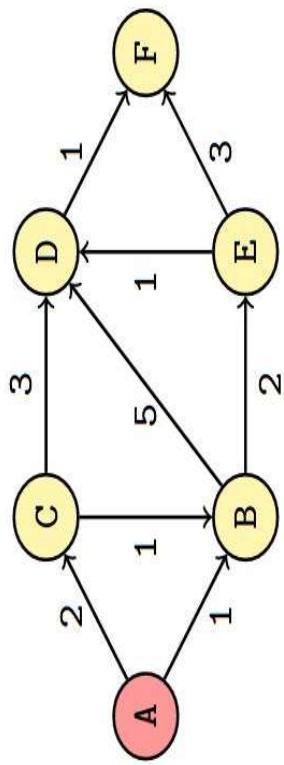
	A	B	C	D	E	F
A	0	0	0	0	0	0
B	∞	1	∞	∞	∞	∞
C	∞	2	2	∞	∞	∞
D	∞	∞	6	5	3	3
E	∞	∞	3	5	∞	∞
F	∞	∞	3	4	∞	6

Ogni colonna contiene lo stato del vettore dall'inizio di ogni ripetizione del ciclo while not Q.isEmpty()

Ogni riga rappresenta l'evoluzione dello stato dell'elemento d [v]

Esempio di funzionamento

26



	A	B	C	E	D	F
A	0	0	0	0	0	0
B	∞	1	1	\cancel{A}	\cancel{A}	\cancel{A}
C	∞	2	2	\cancel{B}	\cancel{B}	\cancel{B}
D	∞	∞	6	5	4	\cancel{D}
E	∞	∞	3	3	3	\cancel{E}
F	∞	∞	∞	∞	6	5

Ogni colonna contiene lo stato del vettore dall'inizio di ogni ripetizione del ciclo while not Q.isEmpty()

Ogni riga rappresenta l'evoluzione dello stato dell'elemento d [v]

Dijkstra - Costo computazionale

27

Riga	Costo	Rip.
(1)	$O(n)$	1
(2)	$O(n)$	$O(n)$
(3)	$O(1)$	$O(n)$
(4)	$O(1)$	$O(m)$

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  $(T, d)$ 

```

Dijkstra - Costo computazionale

28

Riga	Costo	Rip.
(1)	$O(n)$	1
(2)	$O(n)$	$O(n)$
(3)	$O(1)$	$O(n)$
(4)	$O(1)$	$O(m)$

shortestPath(GRAPH G, NODE s)

```

(1) PRIORITYQUEUE Q = PriorityQueue(); Q.insert(s, 0)
(2) while not Q.isEmpty() do
    u = Q.deleteMin()
    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.insert(v, d[u] + G.w(u, v))
                b[v] = true
            else
                Q.decrease(v, d[u] +
                            G.w(u, v))
        T[v] = u
        d[v] = d[u] + G.w(u, v)
    return (T, d)

```

Evoluzione della Priority Queue

Nel 1964 viene sviluppata la struttura dati Heap, una struttura estremamente specializzata che definisce un albero binario tale che il valore memorizzato in ogni nodo è maggiore (max-heap) o minore (min-heap) dei valori memorizzati nei suoi figli.

La radice, quindi, contiene sempre il valore minimo o massimo.

Operazione	Costo
insert()	$O(\log n)$
deleteMin()	$O(\log n)$
min()	$\Theta(1)$
decrease()	$O(\log n)$

Johnson, 1977 - min-heap applicato a Dijkstra

Nel 1977, Johnson sostituisce la priority queue basata su vettore presente nell'algoritmo di Dijkstra con una priority queue basata su min-heap.

Johnson - Costo computazionale

31

Riga	Costo	Rip.
(1)	$O(n)$	1
(2)	$O(\log n)$	$O(n)$
(3)	$O(\log n)$	$O(n)$
(4)	$O(\log n)$	$O(m)$

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  $(T, d)$ 

```

Johnson, considerazione

La complessità $O(m \log n)$ dipende dal valore di m .

- Se il grafo è sparso, $m = O(n)$, quindi siamo sull'ordine di $O(n \log n)$, che è migliore di $O(n^2)$ di Dijkstra “puro”.
- Se il grafo è denso, $m = O(n^2)$, quindi il costo complessivo diventa $O(n^2 \log n)$, che è peggiore di Dijkstra.

In base alla situazione si sceglie la versione migliore.

Fredman-Tarjan, 1987 – Heap di Fibonacci

Nel 1987 Fredman e Tarjan inventano una struttura dati complicatissima, chiamata Heap di Fibonacci.

Non ho idea di come funzioni, ma permette di effettuare l'operazione di aggiornamento in tempo costante $O(1)$ invece di $O(\log n)$, portando il costo a $O(m + \log n)$.

Da usare solo per grafi enormi (tipo quelli di Facebook o Instagram), altrimenti non vale la pena.

Bellman-Ford-Moore, 1958

L’algoritmo di Bellman-Ford-Moore (spesso chiamato anche solo “Bellman-Ford”) è un algoritmo per trovare i cammini minimi su un grafo pesato.

- Cambia la struttura dati di riferimento: non più una priority queue ma una semplice coda;
- Funziona anche con archi con pesi negativi.

Bellman-Ford-Moore - 1: Inizializzazione

Viene creata una nuova coda e inserita la sorgente.

```
(int[], int[]) shortestPath(GRAPH G, NODE s)
(1) QUEUE Q = Queue(); Q.enqueue(s)
    while not Q.isEmpty() do
        int u = Q.dequeue()
        b[u] = false
        foreach v in G.adj(u) do
            if d[u] + G.w(u, v) < d[v] then
                [...]
return (T, d)
```

Bellman-Ford-Moore - 2: Estrazione del minimo

L'estrazione del minimo è semplicemente l'operazione dequeue della coda.

```
(int[], int[]) shortestPath(GRAPH G, NODE s)
(1) QUEUE Q = Queue(); Q.enqueue(s)
    while not Q.isEmpty() do
        (2)   int u = Q.dequeue()
              b[u] = false
              foreach v ∈ G.adj(u) do
                  if d[u] + G.w(u, v) < d[v] then
                      [...]
    return (T, d)
```

Bellman-Ford-Moore - 3: Inserimento in coda

37

Inserisco un elemento in coda quando mi accorgo che c'è stato un miglioramento. Questa operazione potrebbe avvenire più volte.

(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] = true  
  else  
    [...]  
(3)  [...] % Azione da svolgere nel caso  $v$  sia già presente in  $S$   
(4)  [...]  
    T[v] = u  
    d[v] =  $d[u] + G.w(u, v)$   
  [...]
```

Bellman-Ford-Moore - 4: Aggiornamento

Sezione non necessaria, perché non c'è una priorità da aggiornare.

(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

(3)

(4) % 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 - 4: Aggiornamento

Sezione non necessaria, perché non c'è una priorità da aggiornare.

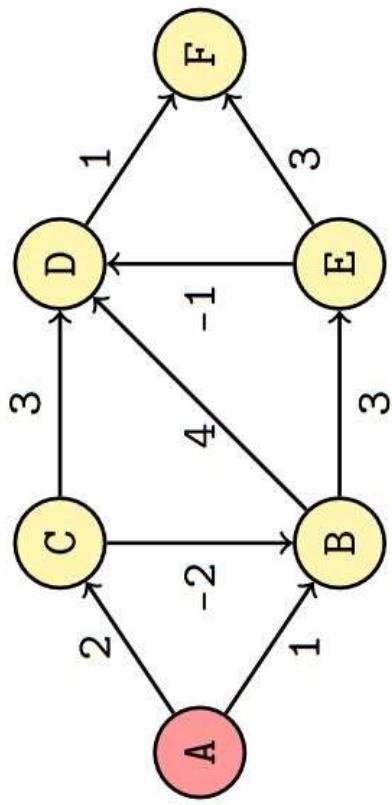
(int[], int[]) shortestPath(GRAPH G, NODE s)

```
[...]  
if  $d[u] + G.w(u, v) < d[v]$  then  
  if not  $b[v]$  then  
    Q.enqueue(v)  
    b[v] = true  
  T[v] = u  
  d[v] =  $d[u] + G.w(u, v)$   
[...]
```

(3)

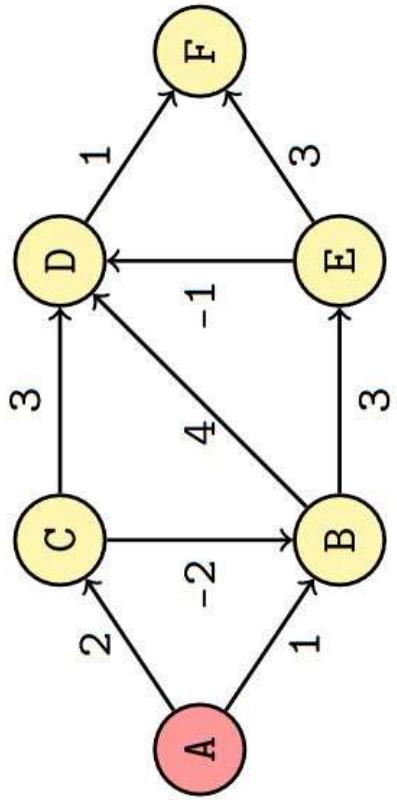
Esempio di esecuzione

Inizialmente, in S (la coda) c'è solo A.



	A	B	C	D	E	F	S	A
A	0							
B		∞						
C			∞					
D				∞				
E					∞			
F						∞		
S							∞	

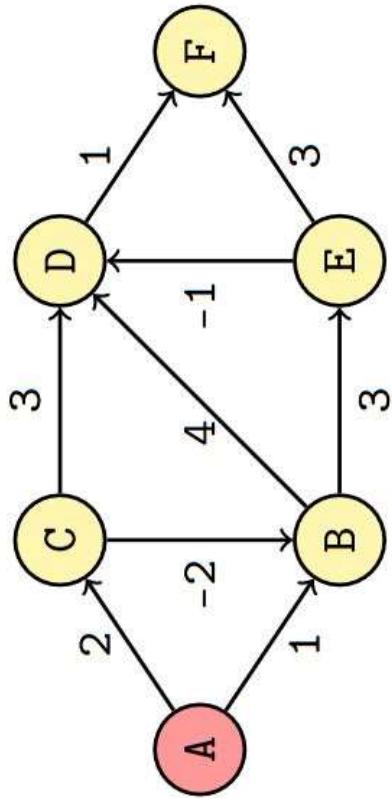
Esempio di esecuzione



Estraggo A, e aggiorno B e C
di conseguenza.

	A	
A	0	0
B	∞	1
C	∞	2
D	∞	∞
E	∞	∞
F	∞	∞

Esempio di esecuzione

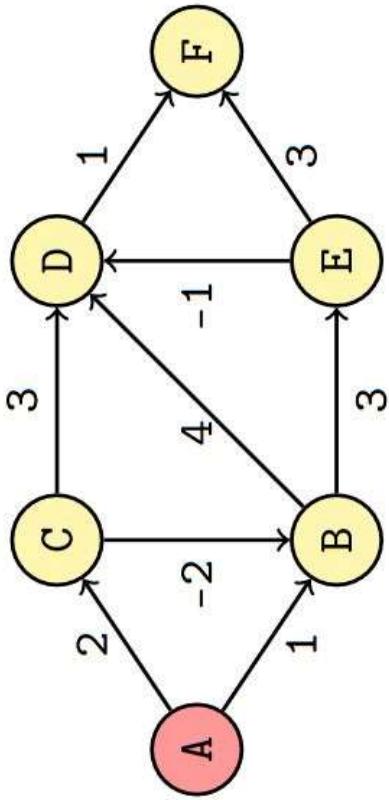


Estraggo B e poi C,
aggiornando i valori di D ed E.

B viene aggiornato, perché
passare per C costa meno, e
viene rimesso in coda.

	A	B	C
A	0	0	0
B	∞	1	0
C	∞	2	2
D	∞	5	5
E	∞	4	4
F	∞	∞	∞

Esempio di esecuzione

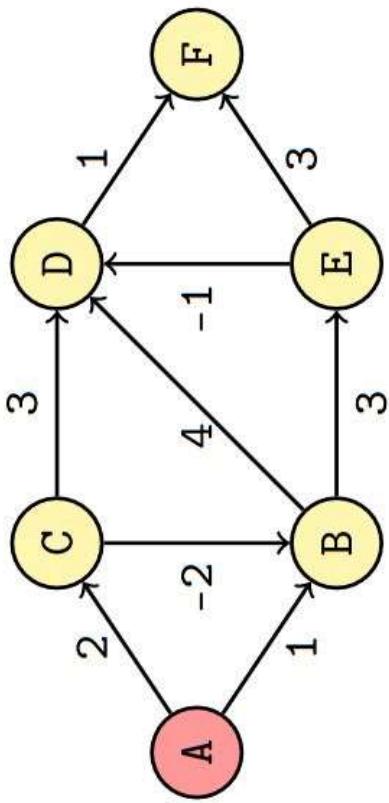


Estraggo D, E e B.

Vengono aggiornati D, E ed F.

	A	B	C	D	E	F
A	0	0	0	0	0	0
B	∞	1	0	0	0	0
C	∞	2	2	2	2	2
D	∞	5	5	5	3	3
E	∞	4	4	4	4	3
F	∞	∞	∞	6	6	6

Esempio di esecuzione



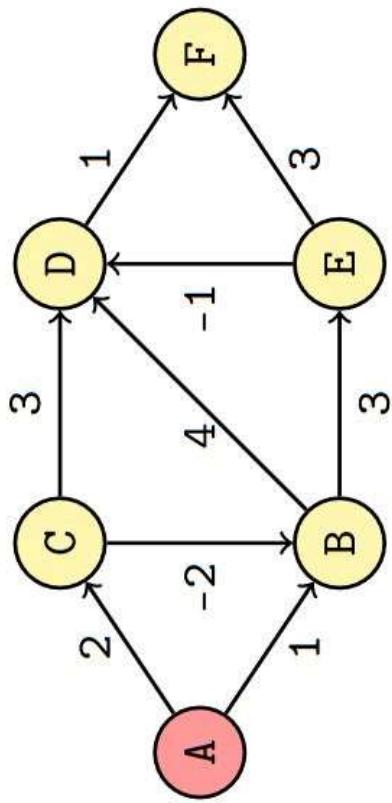
Estraggo F, D e E.

Vengono aggiornati F e D.

	A	B	C	D	E	B	F	D	E
A	0	0	0	0	0	0	0	0	0
B	∞	1	0	0	0	0	0	0	0
C	∞	2	2	2	2	2	2	2	2
D	∞	5	5	5	3	3	3	3	2
E	∞	4	4	4	4	3	3	3	3
F	∞	∞	∞	6	6	6	6	4	4

Esempio di esecuzione

45



Estraggo D e aggiorno F.

Estraggo F e non aggiorno altro.

	A	B	C	D	E	F	B	E	D	E	D	F
A	0	0	0	0	0	0	0	0	0	0	0	0
B	∞	1	0	0	0	0	0	0	0	0	0	0
C	∞	2	2	2	2	2	2	2	2	2	2	2
D	∞	5	5	5	3	3	3	3	2	2	2	2
E	∞	4	4	4	3	3	3	3	3	3	3	3
F	∞	∞	∞	6	6	6	6	4	4	3	3	3
S	A	BC	CDE	DEB	EBF	BFD	FDE	DE	E	D	F	

Bellman-Ford-Moore: correttezza

L’algoritmo è organizzato in “passate”, ossia serie di estrazioni dalla coda.

- Al termine della passata k , i vettori T e δ descrivono i cammini minimi di lunghezza al più k
- Al termine della passata $n-1$, i vettori T e δ descrivono i cammini minimi dalla sorgente verso tutte le destinazioni (la lunghezza è al più $n-1$)

Bellman-Ford-Moore - Costo computazionale

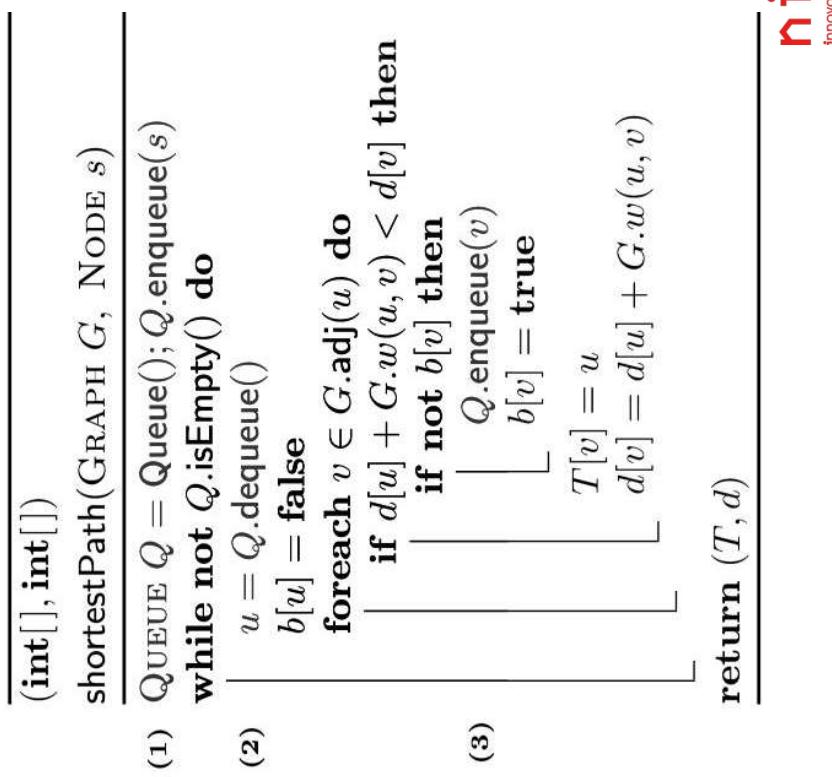
47

Riga	Costo	Rip.
(1)	$O(1)$	1
(2)	$O(1)$	$O(n^2)$
(3)	$O(1)$	$O(nm)$

Ogni nodo viene inserito

almeno una volta, ma

potenzialmente potrei dover
reinserire ogni nodo tutte le
volte.



Bellman-Ford-Moore - Costo computazionale

48

Costo totale: sarebbe il max tra

$O(nm)$ e $O(n^2)$.

Però, se il grafo è连通 (caso
pessimo), m è più grande di n e
quindi $O(nm)$

```
(int[], int[])
shortestPath(GRAPH G, NODE s)
_____
(1) QUEUE Q = Queue(); Q.enqueue(s)
_____
(2)   while not Q.isEmpty() do
        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)
```

Riassuntone finale

49

Algoritmo	Costo	Note
<i>BFS</i>	$O(m+n)$	Grafi senza pesi
<i>Algoritmo nei DAG</i>	$O(m+n)$	Grafi DAG
<i>Dijkstra</i>	$O(n^2)$	Pesi positivi, grafi densi
<i>Johnson</i>	$O(m \log n)$	Pesi positivi, grafi sparsi
<i>Fredman-Tarjan</i>	$O(m + \log n)$	Pesi positivi, grafi densi, dimensioni molto grandi
<i>Bellman-Ford-Moore</i>	$O(mn)$	Anche pesi negativi

Conclusione

Abbiamo visto una panoramica dei più importanti algoritmi per la ricerca dei cammini minimi. Ma si usano davvero?

Sì, nella stragrande maggioranza dei casi (nelle reti, nell'analisi delle reti dei social network, ecc.).

E nella minoranza?

Ci sono però algoritmi specializzati ad esempio per le reti stradali: applicare Dijkstra o Bellman-Ford su Google Maps significa cercare di capire se la strada migliore tra Thiene e Vicenza passa per Berlino.

Quello che succede è che questi algoritmi pre-calcolano alcuni percorsi e poi utilizzano queste informazioni per evitare di espandersi all'infinito nel grafo stradale.

E nella minoranza?

A*, è un algoritmo che utilizza delle tecniche euristiche per velocizzare la ricerca, cercando prima percorsi nel grafo che portano “più probabilmente” alla destinazione, piuttosto che espandersi in tutte le direzioni.

<https://www.youtube.com/watch?v=GC-nBgi9r0U>