

## Programmazione dinamica (V parte)

Progettazione di Algoritmi a.a. 2023-24

Matricole congrue a 1

Docente: Annalisa De Bonis

71

71

### Implementazione efficiente di Bellman-Ford

Alcune osservazioni sull'algoritmo

- Risparmio in termini di spazio: puo` essere ottenuto osservando che non e` necessario portarsi dietro tutta la matrice  $M$  perche' nell'algoritmo di fatto ogni volta che si riempie una nuova riga di  $M$  si fa uso solo dei valori della riga precedente
  - per riempire la riga  $i$  si usano solo i valori presenti della riga  $i-1$
  - quindi perche' portarsi dietro anche le altre righe?
- Un primo immediato miglioramento lo si ottiene andando a modificare la prima versione dell'algoritmo in modo che
  1. usi un array unidimensionale  $M$
  2. ad ogni iterazione del for piu` esterno vada ad aggiornare ciascun valore  $M[v]$  allo stesso modo in cui prima computava i valori  $M[i,v]$ .
    - Per far questo invece di utilizzare i valori  $M[i-1,v]$  utilizzerà i valori  $M[v]$  computati all'iterazione precedente che saranno stati salvati in un array di appoggio

Con questa modifica usiamo 2 array unidimensionali per computare le lunghezze dei percorsi e un array  $S$  per tenere traccia dei successori -> spazio  $O(n)$

Progettazione di Algoritmi A.A. 2023-24  
A. De Bonis

72

72

### Algoritmo di Bellman-Ford : I miglioramento

MA: array di appoggio

```
Improved-Shortest-Path_1(G, t) {
  foreach node v ∈ V
    M[v] ← ∞
    MA[v] ← ∞
    S[v] ← ∅ // ∅ indica che non ci sono percorsi
              //da v a t di al piu` 0 archi

  M[t] ← 0
  MA[t] ← 0
  S[t] ← t //t indica che non ci sono successori
           //lungo il percorso ottimo da t a t

  for i = 1 to n-1
    foreach node v ∈ V
      foreach edge (v, w) ∈ E
        if MA[w] + cvw < M[v]
          M[v] ← MA[w] + cvw
          S[v] ← w //serve per ricostruire i
                  //percorsi minimi verso t
      foreach node v ∈ V
        MA[v]=M[v] //salvo M[v] nell'array di appoggio
}
```

Progettazione di Algoritmi A.A. 2023-24  
A. De Bonis

73

73

### Implementazione efficiente di Bellman-Ford

Alcune osservazioni sull'algoritmo

- Un ulteriore miglioramento basato sulla seguente osservazione:
- Se in una certa iterazione  $i$  del for esterno il valore di  $MA[w]$  è lo stesso dell'iterazione precedente ( $M[w]$  non è stato aggiornato nel corso dell'iterazione  $i-1$ ) allora i valori  $MA[w] + c_{vw}$  computati nell'iterazione  $i$  sono esattamente gli stessi computati nell'iterazione  $i-1$ .
- Questa osservazione dà l'idea per un secondo miglioramento dell'algoritmo: quando in una certa iterazione  $i$  del for esterno, l'algoritmo calcola  $M[v]$  va a considerare solo quei nodi  $w$  per cui esiste l'arco  $(v,w)$  e tali che  $M[w]$  è stato modificato durante l'iterazione  $i-1$ .
- L'algoritmo nella slide successiva realizza questa idea in questo modo: scandisce ciascun nodo  $w$  del grafo e controlla se il valore di  $M[w]$  è cambiato nell'iterazione precedente e solo in questo caso esamina gli archi  $(v,w)$  entranti in  $v$  e per ciascuno di questi archi computa  $MA[w] + c_{vw}$ 
  - Cio` equivale a scandire tutti i nodi  $v$  e a controllare per ogni arco  $(v,w)$  uscente da  $v$  se  $M[w]$  è cambiato nell'iterazione precedente prima di calcolare  $MA[w] + c_{vw}$

Progettazione di Algoritmi A.A. 2023-24  
A. De Bonis

74

74

### Algoritmo di Bellman-Ford : II miglioramento

MA: array di appoggio

```
Improved-Shortest-Path_2(G, t) {
  foreach node v ∈ V
    M[v] ← ∞
    MA[v] ← ∞
    S[v] ← ∅ // ∅ indica che non ci sono percorsi
              //da v a t di al piu' 0 archi

  M[t] ← 0
  MA[t] ← 0
  S[t] ← t //t indica che non ci sono successori
           //lungo il percorso ottimo da t a t

  for i = 1 to n-1
    foreach node w ∈ V
      if M[w] has been updated in iteration i-1
        foreach edge (v, w) ∈ E
          if MA[w] + cvw < M[v]
            M[v] ← MA[w] + cvw
            S[v] ← w //serve per ricostruire i
                    //percorsi minimi verso t

    foreach node v ∈ V
      MA[v]=M[v]//salvo M[v]nell'array di appoggio
}
```

Progettazione di Algoritmi A.A. 2023-24  
A. De Bonis

75

### Implementazione efficiente di Bellman-Ford

Alcune osservazioni sull'algoritmo

- Torniamo per un momento al fatto che un miglioramento dell'algoritmo consiste nell'usare un array unidimensionale  $M$ .
- Abbiamo detto che per far ciò l'algoritmo può usare un array di appoggio che memorizza i valori di  $M$  computati dall'iterazione precedente del for esterno.
- **Domanda:** cosa accade se non utilizziamo un array di appoggio?
- Consideriamo l'iterazione  $i$  del for esterno.
- Se non utilizziamo un array di appoggio, quando calcoliamo  $M[w] + c_{vw}$ , siamo costretti ad usare i valori  $M[w]$  presenti in  $M$ .
  - Quando calcoliamo  $M[w] + c_{vw}$ , il valore  $M[w]$  potrebbe essere uguale al valore computato nell'iterazione  $i-1$  o potrebbe già essere stato aggiornato nell'iterazione  $i$  (anche più di una volta).
  - Nel caso  $M[w]$  sia stato già modificato nell'iterazione  $i$  allora  $M[w]$  conterrà la lunghezza di un percorso più corto rispetto al valore di  $M[w]$  computato nell'iterazione precedente.
    - Di conseguenza  $M[v]$  potrebbe essere aggiornato con un valore più piccolo di quello che si sarebbe ottenuto utilizzando il valore di  $M[w]$  computato nell'iterazione precedente.

Progettazione di Algoritmi A.A. 2023-24  
A. De Bonis

76

76

### Implementazione efficiente di Bellman-Ford

Alcune osservazioni sull'algoritmo

- Conseguenze dell'osservazione nella slide precedente:
  - Dopo ogni iterazione  $i$ ,  $M[v]$  **potrebbe** contenere la lunghezza di un percorso per andare da  $v$  a  $t$  formato da **piu` di  $i$  archi**.
  - La lunghezza di  $M[v]$  e` sicuramente non piu` grande della lunghezza del percorso piu` corto per andare da  $v$  a  $t$  formato da **al massimo  $i$  archi**.

Progettazione di Algoritmi A.A. 2023-24  
A. De Bonis

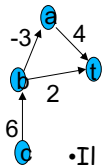
77

77

### Implementazione efficiente di Bellman-Ford

Alcune osservazioni sull'algoritmo

- Conseguenze dell'osservazione nella slide precedente:
  - Esempio, Consideriamo il grafo qui di fianco.
  - supponiamo di esaminare i nodi  $w$  in questo ordine  $t, a, b, c$
  - Iterazione  $i=1$ :
    - Quando esaminiamo  $w=t$ , poniamo  $M[a]=4$  e  $M[b]=2$ .
    - Quando si esamina  $w=a$  si ha  $M[a]=4$  e di conseguenza  $M[b]$  diventa 1 (lunghezza del percorso  $b, a, t$ ).
    - Quando poi esaminiamo  $b$ ,  $M[c]$  da  $\infty$  che era, diventa 7 (lunghezza di  $c, b, a, t$ ).
  - Nell'implementazione con array di appoggio, alla fine della prima iterazione avremmo avuto  $M[b]=2$  e  $M[c] = \infty$
- Il terzo e ultimo miglioramento consiste nel modificare `Improved-Shortest-Path_2` in modo che non usi l'array di appoggio. In questo modo si ottiene l'algoritmo `Push-Based-Shortest-Path`.



Progettazione di Algoritmi A.A. 2023-24  
A. De Bonis

78

78

## Implementazione efficiente di Bellman-Ford

```

Push-Based-Shortest-Path(G, s, t) {
  foreach node v ∈ V {
    M[v] ← ∞
    S[v] ← φ //nel libro si chiama first[v]
  }

  M[t] = 0, S[t]=t
  for i = 1 to n-1 {
    foreach node w ∈ V {
      if (M[w] has been updated in previous iteration) {
        foreach node v such that (v, w) ∈ E {
          if (M[v] > M[w] + cvw) {
            M[v] ← M[w] + cvw
            S[v] ← w
          }
        }
      }
    }
    if no M[v] value changed in this iteration i
      return M[s]
  }
  return M[s]
}

```

Le osservazioni viste nelle slide precedenti ci portano a questa versione dell'algoritmo

NB: in una certa iterazione del for esterno quando si calcola una distanza  $M[w]+c_{vw}$  potrebbe accadere che  $M[w]$  sia stata aggiornata già in quella stessa iterazione.

Progettazione di Algoritmi A.A. 2023-24  
A. De Bonis

79

79

## Miglioramento dell'algoritmo

**Teorema.** Durante l'algoritmo `Push-Based-Shortest-Path`,  $M[v]$  è la lunghezza di un certo percorso da  $v$  a  $t$ , e dopo  $i$  round di aggiornamenti (dopo  $i$  iterazioni del for esterno) il valore di  $M[v]$  **non è più grande della lunghezza del percorso minimo da  $v$  a  $t$  che usa al più  $i$  archi**

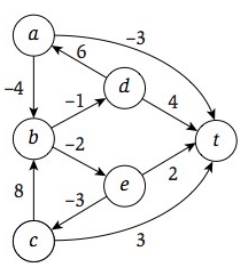
- Non usare un array di appoggio in pratica accelera i tempi per ottenere i percorsi più corti fino a  $t$  formati da al più  $n-1$  archi (che sono quelli che ci interessa ottenere).
- **Nulla cambia per quanto riguarda l'analisi asintotica dell'algoritmo**
- **Conseguenze sullo spazio usato da `Push-Based-Shortest-Path`**
  - Memoria:  $O(n)$ .
- Tempo:
  - il tempo è sempre  $O(nm)$  nel caso pessimo però in pratica l'algoritmo si comporta meglio.
    - Possiamo interrompere le iterazioni non appena accade che durante una certa iterazione i nessun valore  $M[v]$  cambia

Progettazione di Algoritmi A.A. 2023-24  
A. De Bonis

80

80

### Miglioramento dell'algoritmo: un esempio



	t	a	b	c	d	e	
M	0	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	inizializ-
S	t	$\phi$	$\phi$	$\phi$	$\phi$	$\phi$	zazione
M	0	-3	$\infty$	3	4	2	i=1
S	t	t	$\phi$	t	t	t	w=t
M	0	-3	$\infty$	3	3	2	i=1
S	t	t	$\phi$	t	a	t	w=a
M	0	-3	$\infty$	3	3	2	i=1
S	t	t	$\phi$	t	a	t	w=b
M	0	-3	$\infty$	3	3	0	i=1
S	t	t	$\phi$	t	a	c	w=c
M	0	-3	2	3	3	0	i=1
S	t	t	d	t	a	c	w=d
M	0	-3	-2	3	3	0	i=1
S	t	t	e	t	a	c	w=e

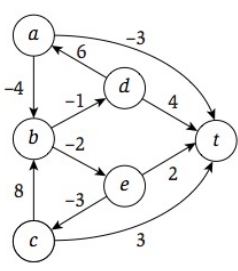
per ogni nodo w esaminato nel secondo foreach le celle verdi sono quelle che cambiano valore

le celle arancioni sono quelle il cui valore non e' cambiato nell'i-esima iterazione

Progettazione di Algoritmi A.A. 2023-24  
A. De Bonis

81

### Miglioramento dell'algoritmo: un esempio



	t	a	b	c	d	e	
M	0	-3	-2	3	3	0	fine
S	t	t	e	t	a	c	iteraz.
M	0	-3	-2	3	3	0	i=2
S	t	t	e	t	a	c	w=a
M	0	-6	-2	3	3	0	i=2
S	t	b	e	t	a	c	w=b
M	0	-6	-2	3	3	0	i=2
S	t	b	e	t	a	c	w=c
M	0	-6	-2	3	3	0	i=2
S	t	b	e	t	a	c	w=d
M	0	-6	-2	3	3	0	i=2
S	t	b	e	t	a	c	w=e

per ogni nodo w esaminato nel secondo foreach le celle verdi sono quelle che cambiano valore

le celle arancioni sono quelle il cui valore non e' cambiato nell'i-esima iterazione

Progettazione di Algoritmi A.A. 2023-24  
A. De Bonis

82

### Miglioramento dell'algoritmo: un esempio

	t	a	b	c	d	e	
M	0	-6	-2	3	3	0	fine iteraz. i=2
S	t	b	e	t	a	c	
M	0	-6	-2	3	0	0	i=3 w=a
S	t	b	e	t	a	c	

per ogni nodo w esaminato nel secondo foreach le celle verdi sono quelle che cambiano valore

le celle arancioni sono quelle il cui valore non e' cambiato nell'i-esima iterazione

Progettazione di Algoritmi A.A. 2023-24  
A. De Bonis

83

### Miglioramento dell'algoritmo: un esempio

	t	a	b	c	d	e	
M	0	-6	-2	3	0	0	fine iteraz. i=3
S	t	b	e	t	a	c	
M	0	-6	-2	3	0	0	i=4 w=d
S	t	b	e	t	a	c	

le celle arancioni sono quelle il cui valore non e' cambiato nell'i-esima iterazione

Progettazione di Algoritmi A.A. 2023-24  
A. De Bonis

84

### Moltiplicazione di una catena di matrici

- **Input:** una sequenza di  $n$  matrici  $A_1, A_2, A_3, \dots, A_n$ , *compatibili due a due rispetto al prodotto*
  - Due matrici  $A$  e  $B$  sono compatibili rispetto al prodotto se il numero di colonne di  $A$  è uguale al numero di righe di  $B$
- **Obiettivo:** vogliamo calcolare il prodotto delle  $n$  matrici in modo da **minimizzare il numero di moltiplicazioni**.
  - Data una matrice  $m \times n$   $A$  e una matrice  $n \times p$   $B$  la matrice  $A \times B$  è una matrice  $m \times p$  e per calcolare ciascuna delle  $mp$  entrate di  $A \times B$  abbiamo bisogno di moltiplicare una riga di  $A$  per una colonna di  $B \rightarrow n$  moltiplicazioni scalari per ciascuna entrata di  $A \times B \rightarrow mnp$  moltiplicazioni scalari.
  - La moltiplicazione tra matrici è associativa per cui possiamo scegliere l'ordine in cui moltiplichiamo le matrici parentesizzando opportunamente la catena di matrici

85

### Moltiplicazione di una catena di matrici

- **Consideriamo le tre matrici**
  - $A: 100 \times 1$             vettore colonna
  - $B: 1 \times 100$             vettore riga
  - $C: 100 \times 1$             vettore colonna
- **Numero di moltiplicazioni per diverse parentesizzazioni:**
  - $((A \cdot B) \cdot C) \rightarrow (100 \times 1 \times 100) + (100 \times 100 \times 1) = 20000$ 
    - prima  $100 \times 1 \times 100$  moltiplicazioni per  $A \cdot B$  e poi  $100 \times 100 \times 1$  moltiplicazioni per  $(A \cdot B) \cdot C$
  - $(A \cdot (B \cdot C)) \rightarrow (1 \times 100 \times 1) + (100 \times 1 \times 1) = 200$ 
    - prima  $1 \times 100 \times 1$  moltiplicazioni per  $B \cdot C$  e poi  $100 \times 1 \times 1$  moltiplicazioni per  $A \cdot (B \cdot C)$

86



### Moltiplicazione di una catena di matrici

- Per  $i < j$ , una parentesizzazione  $P(i, \dots, j)$  del prodotto  $A_i \cdot A_{i+1} \cdot \dots \cdot A_j$  consiste nel prodotto di due parentesizzazioni  $(P(i, \dots, k) \cdot P(k+1, \dots, j))$ 
  - $P(i, \dots, k)$  e  $P(k+1, \dots, j)$  sono le due parentesizzazioni a livello piu' esterno
- Sia  $A(i \dots j) = A_i \cdot A_{i+1} \cdot \dots \cdot A_j$  una sottosequenza della catena di moltiplicazioni  $A_1 \cdot A_2 \cdot A_3 \cdot \dots \cdot A_n$ .
- Supponiamo che  $P(i \dots j)$  sia una parentesizzazione ottima di  $A(i \dots j)$  e siano  $P(i \dots k)$  e  $P(k+1 \dots j)$  le parentesizzazioni a livello piu' esterno in  $P(i \dots j)$ .
- Sottostruttura ottimale: se  $P(i \dots j)$  è una parentesizzazione ottima di  $A(i \dots j)$  allora le due parentesizzazioni  $P(i \dots k)$  e  $P(k+1 \dots j)$  sono ottime per le sottosequenze  $A(i \dots k)$  e  $A(k+1 \dots j)$  rispettivamente.

87

### Moltiplicazione di una catena di matrici

$OPT(i, j)$ : minimo numero di moltiplicazioni scalari per calcolare il prodotto  $A(i \dots j)$

Caso  $i = j$ . In questo caso (base)  $OPT(i, j) = 0$

Caso  $i < j$ .

- Supponiamo di sapere che la parentesizzazione ottima per  $A(i, \dots, j)$  sia formata a livello piu' esterno dal prodotto delle parentesizzazioni di  $A(i, \dots, k)$  e  $A(k+1, \dots, j)$ . Per la sottostruttura ottimale si ha:
 
$$OPT(i, j) = OPT(i, k) + OPT(k+1, j) + c_{i-1} \cdot c_k \cdot c_j$$
  - $c_{i-1}$  = numero di righe della matrice  $A_i$
  - $c_i$  = numero di colonne della matrice  $A_i$
  - la matrice  $(A_i \cdot \dots \cdot A_k)$  ha dimensioni  $c_{i-1} \times c_k$
  - la matrice  $(A_{k+1} \cdot \dots \cdot A_j)$  ha dimensioni  $c_k \times c_j$

→ costo per moltiplicare la matrice  $(A_i \cdot \dots \cdot A_k)$  con  $(A_{k+1} \cdot \dots \cdot A_j)$  è  $c_{i-1} \cdot c_k \cdot c_j$
- Siccome non conosciamo il valore di  $k$  nella soluzione ottima, computiamo il valore  $OPT(i, k) + OPT(k+1, j) + c_{i-1} \cdot c_k \cdot c_j$  per ogni  $k$  tra  $i$  e  $j-1$  e scegliamo il piu' piccolo di questi valori.

88

### Moltiplicazione di una catena di matrici

Dai due casi precedenti si ha la seguente formula di ricorrenza:

$$\text{OPT}(i,j)=0 \text{ se } i=j$$

$$\text{OPT}(i,j)=\min_{i \leq k < j} \{ \text{OPT}(i,k) + \text{OPT}(k+1,j) + c_{i-1} \cdot c_k \cdot c_j \} \quad \text{se } i < j$$

89

### Esercizio

MoltiplicazioneMatrici (c) // c array t.c. c[i]= c<sub>i</sub>= #colonne A<sub>i</sub> = #righe A<sub>i+1</sub>

n=length(c)

for i=1 to n

  M[i, i]=0

for lung=2 to n //ogni iterazione calcola M[i,j] ed m[i,j] per

  //i,j tali che i<j e j-i=lung

  for i=1 to n-lung+1

    j=i+lung-1

    M[i, j]=∞

    for k=i to j-1

      M = M[i, k]+ M[k+1, j] + c[i-1] c[k] c[j]

      if M < M[i, j] then M[i, j]=M

  return M[1, n]

$O(n^3)$

vengono calcolati M[i,j]  
per ogni (i,j), con i<j, in questo ordine:

(1,2),(2,3), ..., (n-1,n)

(1,3),(2,4),..., (n-2,n)

...

(1,n-1),(2,n)

(1,n)

90