

Programmazione dinamica (IV parte)

Progettazione di Algoritmi a.a. 2021-22

Matricole congrue a 1

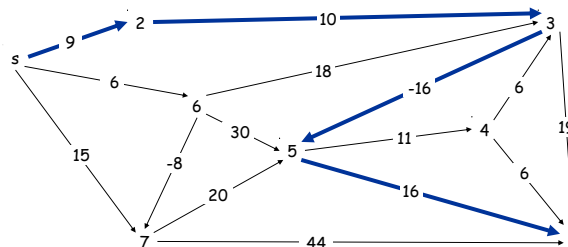
Docente: Annalisa De Bonis

58

58

Cammini minimi

- **Problema del percorso più corto.** Dato un grafo direzionato $G = (V, E)$, con pesi degli archi c_{vw} , trovare il percorso più corto da s a t .
- **Esempio.** I nodi rappresentano agenti finanziari e c_{vw} è il costo (eventualmente <0) di una transazione che consiste nel comprare dall'agente v e vendere immediatamente a w .



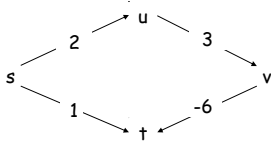
Progettazione di Algoritmi A.A. 2021-22
A. De Bonis

59

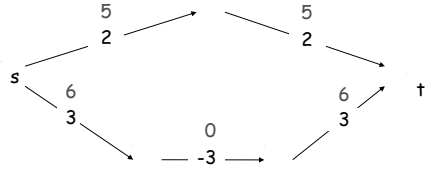
59

Cammini minimi in presenza di archi con costo negativo

- **Dijkstra.** Può fallire se ci sono archi di costo negativo



- **Re-weighting.** Aggiungere una costante positiva ai pesi degli archi potrebbe non funzionare.

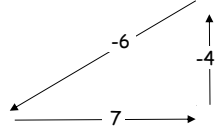
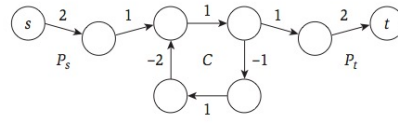


Progettazione di Algoritmi A.A. 2021-22
A. De Bonis

60

Cammini minimi in presenza di archi con costo negativo

- **Ciclo di costo negativo.**

Progettazione di Algoritmi A.A. 2021-22
A. De Bonis

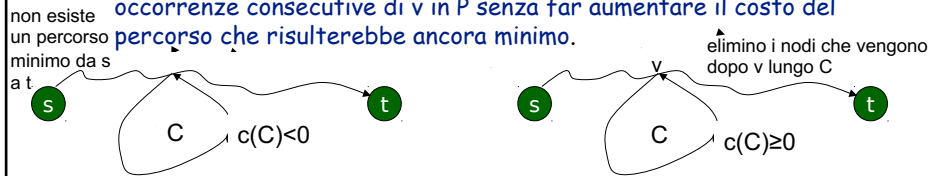
61

Cammini minimi in presenza di archi con costo negativo

Osservazione. Se qualche percorso da s a t contiene un ciclo di costo negativo allora non esiste un percorso minimo da s a t . In caso contrario esiste un percorso minimo da s a t che è semplice (nessun nodo compare due volte sul percorso).

- **Dim.** Se esiste un percorso P da s a t con un ciclo C di costo negativo $-c$ allora ogni volta che attraversiamo il ciclo riduciamo il costo del percorso di un valore pari a c . Ciò rende impossibile definire il costo del percorso minimo perché dato un percorso riusciamo sempre a trovarne uno di costo minore attraversando il ciclo C (osservazione questa che avevamo già fatto in precedenti lezioni).

Supponiamo ora che nessun percorso da s a t contenga cicli negativi e sia P un percorso minimo da s a t (ovviamente P è privo di cicli di costo negativo). Supponiamo che un certo vertice v appaia almeno due volte in P . C'è quindi in P un ciclo che contiene v e che per ipotesi deve avere costo **non negativo**. In questo caso potremmo rimuovere le porzioni di P tra due occorrenze consecutive di v in P senza far aumentare il costo del percorso che risulterebbe ancora minimo.



62

Cammini minimi: Programmazione dinamica

Sia t la destinazione a cui si vuole arrivare.

Def. $OPT(i, v)$ = lunghezza del cammino più corto P per andare da v a t che consiste di al più i archi

Per computare $OPT(i, v)$ quando $i > 0$ e $v \neq t$, osserviamo che

- il percorso ottimo P deve contenere almeno un arco (che ha come origine v).
- se (v, w) è il primo arco di P allora P è formato da (v, w) e da percorso più corto da w a t di al più $i-1$ archi.
- siccome non sappiamo quale sia il primo arco di P allora computiamo $OPT(i, v) = \min_{(v,w) \in E} \{OPT(i-1, w) + c_{vw}\}$

La formula di ricorrenza è quindi:

$$OPT(i, v) = \begin{cases} 0 & \text{se } v=t \\ \infty & \text{se } i=0 \text{ e } v \neq t \\ \min_{(v,w) \in E} \{OPT(i-1, w) + c_{vw}\} & \text{altrimenti} \end{cases}$$

Progettazione di Algoritmi A.A. 2021-22
A. De Bonis

63

63

Cammini minimi: Programmazione dinamica

$$OPT(i,v) = \begin{cases} 0 & \text{se } v=t \\ \infty & \text{se } i=0 \text{ e } v \neq t \\ \min_{(v,w) \in E} \{OPT(i-1, w) + c_{vw}\} & \text{altrimenti} \end{cases}$$

Notiamo che nel secondo caso usiamo ∞ per indicare che attraversando 0 archi non è possibile raggiungere t .

Dove si usa l'osservazione di prima sul fatto che in assenza di cicli negativi il percorso minimo è semplice?

Ecco dove...

Affermazione. Se non ci sono cicli di costo negativo allora $OPT(n-1, v)$ = lunghezza del percorso più corto da v a t .

Dim. Dall'osservazione precedente se non ci sono cicli negativi allora esiste un percorso di costo minimo da v a t che è semplice e di conseguenza contiene al più $n-1$ archi

$$OPT(i,v) = \begin{cases} 0 & \text{se } v=t \\ \infty & \text{se } i=0 \text{ e } v \neq t \\ \min_{(v,w) \in E} \{OPT(i-1, w) + c_{vw}\} & \text{altrimenti} \end{cases}$$

A. De Bonis

64

64

Algoritmo di Bellman-Ford per i cammini minimi

```
Shortest-Path(G, t) {
  foreach node v ∈ V
    M[0, v] ← ∞
    S[0, v] ← ∅ // ∅ indica che non ci sono percorsi
                //da v a t di al più 0 archi
  for i = 0 to n-1
    M[i, t] ← 0
    S[i, t] ← t //t indica che non ci sono successori di t
                //lungo il percorso ottimo da t a t
  for i = 1 to n-1
    foreach node v ∈ V
      M[i, v] ← ∞, S[i, v] ← ∅
      foreach edge (v, w) ∈ E
        if M[i-1, w] + cvw < M[i, v]
          M[i, v] ← M[i-1, w] + cvw
          S[i, v] ← w //serve per ricostruire i
                    //percorsi minimi verso t
}
```

- Assumiamo che per ogni v esista un percorso da v a $t \rightarrow n=O(m)$
- Analisi. Tempo $\Theta(mn)$, spazio $\Theta(n^2)$.
- $S[i,v]$: Memorizza il successore di v lungo il percorso minimo per andare da v a t attraversando al più i archi

Progettazione di Algoritmi A.A. 2021-22

A. De Bonis

65

65

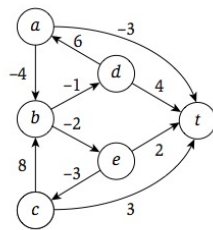
Algoritmo di Bellman-Ford per i cammini minimi

Osservazione

- L'algoritmo di Bellman-Ford di fatto calcola le lunghezze dei cammini minimi da v a t per ogni v (risolve Single Destination Shortest Paths)
- Queste lunghezze sono contenute nella riga $n-1$
- L'algoritmo puo` essere scritto in modo che prenda in input un vertice sorgente s ed un vertice destinazione t ma il contenuto della tabella M dipende solo da t .
- In altri termini, una volta costruita la tabella per un certo t , possiamo ottenere la lunghezza del percorso piu` corto da un qualsiasi nodo v al nodo t andando a leggere l'entrata $M[n-1,v]$

66

Bellman-Ford: esempio



Come agguino la cella $M[i,j]$ e la cella $S[i,j]$, per $i > 1$?
Inizialmente pongo $M[i,j] = \infty$ e $S[i,j] = \emptyset$.
Nella riga $i-1$ esamino tutte le entrate $M[i-1,k]$ per cui esiste l'arco (j,k) e per ciascuna di queste entrate computo $M[i-1,k] + c_{jk}$. Se questo valore è piu` piccolo di $M[i,j]$, pongo $M[i,j] = M[i-1,k] + c_{jk}$ e $S[i,j] = k$.

	t	a	b	c	d	e
0	0	∞	∞	∞	∞	∞
1	0	-3	∞	3	4	2
2	0	-3	0	3	3	0
3	0	-4	-2	3	3	0
4	0	-6	-2	3	2	0
5	0	-6	-2	3	0	0

	t	a	b	c	d	e
0	t	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset
1	t	t	\emptyset	t	t	t
2	t	t	e	t	a	c
3	t	b	e	t	a	c
4	t	b	e	t	a	c
5	t	b	e	t	a	c

67

Algoritmo che produce il cammino minimo

```

FindPath(i,v):
  if S[i,v]= ∅
    output "No path"
    return
  if v= t
    output t
    return
  output v
  FindPath(i-1,S[i,v])
            
```

prima volta invocato con $i=n-1$ e v uguale al nodo per il quale vogliamo computare il cammino minimo fino a t

tempo $O(n)$ perche'

- se ignoriamo il tempo per la chiamata ricorsiva al suo interno, il tempo di ciascuna chiamata e' $O(1)$
- vengono effettuate al piu' $n-1$ chiamate

68

Bellman-Ford: esempio

	t	a	b	c	d	e
0	t	∅	∅	∅	∅	∅
1	t	t	∅	t	t	t
2	t	t	e	t	a	c
3	t	b	e	t	a	c
4	t	b	e	t	a	c
5	t	b	e	t	a	c

Supponiamo di voler conoscere il percorso minimo tra a e t

Invoco FindPath(5,a)

output **a** e effettua ricorsione con $i=4$ e $v=S[5,a]=b$

output **b** e effettua ricorsione con $i=3$ e $v= S[4,b]=e$

output **e** e effettua ricorsione con $i=2$ e $v= S[2,e]=c$

output **c** e effettua ricorsione con $i=1$ e $v= S[1,c]=t$

output **t** ed esci

Il percorso minimo da a verso t e' **a,b,e,c,t**

```

FindPath(i,v):
  if S[i,v]= ∅
    output "No path"
    return
  if v= t
    output t
    return
  output v
  FindPath(i-1,S[i,v])
            
```

```

graph TD
    a((a)) -- 6 --> d((d))
    a -- -4 --> b((b))
    d -- -1 --> b
    d -- 4 --> t((t))
    b -- -2 --> t
    c((c)) -- 8 --> b
    c -- -3 --> e((e))
    e -- 2 --> t
    c -- 3 --> t
    a -- -3 --> t
            
```

69

Miglioramento dell'algoritmo

- Usiamo un array unidimensionale M :
 - $M[v]$ = percorso da v a t piu` corto che abbiamo trovato fino a questo momento
 - Per ogni $i=1,\dots,n-1$ computiamo cosi` $M[v]$ per ogni v :

$$M[v] = \min(M[v], \min_{(v,w) \in E} (c_{vw} + M[w]))$$
1. computiamo per ogni arco (v,w) uscente da v la distanza $c_{v,w} + M[w]$ che rappresenta la lunghezza del percorso piu` corto **computato fino a quel momento** per andare da v a t passando per l'arco (v,w)
 2. tra tutte le lunghezze computate in 1, prendiamo quella piu` piccola e se questa e` minore di $M[v]$ aggiorniamo $M[v]$

Progettazione di Algoritmi A.A. 2021-22
A. De Bonis

70

70

Miglioramento dell'algoritmo

1. computiamo per ogni arco (v,w) uscente da v la distanza $c_{v,w} + M[w]$.
 2. tra tutte le lunghezze computate in 1. prendiamo quella piu` piccola e se questa e` minore di $M[v]$ aggiorniamo $M[v]$
- l'algoritmo che vedremo calcola le distanze al punto 1 considerando solo quegli archi (v,w) per cui si ha che $M[w]$ ha cambiato valore all'iterazione precedente
 - di fatto l'algoritmo scandisce tutti i nodi w del grafo e ogni volta che ne incontra uno il cui valore $M[w]$ e` cambiato all'iterazione precedente va ad esaminare tutti gli archi (v,w) entranti in w . Per ciascuno di questi archi calcola la distanza $c_{v,w} + M[w]$ e se questa e` minore di $M[v]$, pone $M[v] = c_{v,w} + M[w]$
 - si noti che alla fine l'algoritmo avra` esaminato per ogni nodo v tutti gli archi (v,w) per cui si ha che $M[w]$ ha cambiato valore all'iterazione precedente

Progettazione di Algoritmi A.A. 2021-22
A. De Bonis

71

71

Computazione del cammino minimo

- Per ogni vertice v memorizziamo in $S[v]$ il successore di v , cioè il primo nodo che segue v lungo il percorso da v a t di costo $M[v]$.
- $S[v]$ viene aggiornato ogni volta che $M[v]$ viene aggiornato. Se $M[v]$ viene posto uguale a $c_{vw}+M[w]$ allora si pone $S[v]=w$.

72

Implementazione efficiente di Bellman-Ford

Alcune osservazioni sull'algoritmo

- Risparmio in termini di spazio: può essere ottenuto osservando che non è necessario portarsi dietro tutta la matrice M perché 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 perché 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 più 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 \rightarrow spazio $O(n)$

73

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

          MA[v]=M[v] //salvo M[v] nell'array di appoggio
}
```

Progettazione di Algoritmi A.A. 2021-22
A. De Bonis

74

74

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. 2021-22
A. De Bonis

75

75

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. 2021-22
A. De Bonis

76

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. 2021-22
A. De Bonis

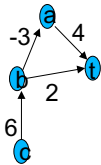
77

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**.



- Esempio, Consideriamo il grafo qui di fianco.
- Iterazione $i=1$: supponiamo di esaminare i nodi w in questo ordine t, a, b, c . 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 ∞ 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. 2021-22
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 gia` in quella stessa iterazione.

Progettazione di Algoritmi A.A. 2021-22
A. De Bonis

79

79

Miglioramento dell'algoritmo

Teorema. Durante l'algoritmo **Push-Based-Shortest-Path**, $M[v]$ e' 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 e' più grande della lunghezza del percorso minimo da v a t che usa al più i archi**

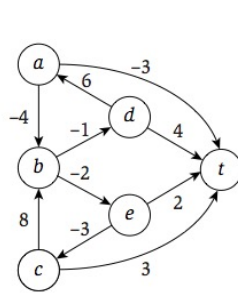
- Non usare un array di appoggio in pratica accelera i tempi per ottenere i percorsi più corti fino a t formati da al più $n-1$ archi (che sono quelli che ci interessa ottenere).
- **Nulla cambia per quanto riguarda l'analisi asintotica dell'algoritmo**
- **Conseguenze sullo spazio usato da Push-Based-Shortest-Path**
 - Memoria: $O(n)$.
 - Tempo:
 - il tempo e' sempre $O(nm)$ nel caso pessimo pero' 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. 2021-22
A. De Bonis

80

80

Miglioramento dell'algoritmo: un esempio



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

	t	a	b	c	d	e	
M	0	∞	∞	∞	∞	∞	inizializzazione
S	t	ϕ	ϕ	ϕ	ϕ	ϕ	
M	0	-3	∞	3	4	2	i=1
S	t	t	ϕ	t	t	t	w=t
M	0	-3	∞	3	3	2	i=1
S	t	t	ϕ	t	a	t	w=a
M	0	-3	∞	3	3	2	i=1
S	t	t	ϕ	t	a	t	w=b
M	0	-3	∞	3	3	0	i=1
S	t	t	ϕ	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

Progettazione di Algoritmi A.A. 2021-22
A. De Bonis

81

81

Miglioramento dell'algoritmo: un esempio

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

	t	a	b	c	d	e	
M	0	-3	-2	3	3	0	fine iteraz.
S	t	t	e	t	a	c	i=1
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

Progettazione di Algoritmi A.A. 2021-22
A. De Bonis

82

82

Miglioramento dell'algoritmo: un esempio

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

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

Progettazione di Algoritmi A.A. 2021-22
A. De Bonis

83

83

Miglioramento dell'algoritmo: un esempio

```

graph TD
    a((a)) -- -4 --> b((b))
    a((a)) -- -3 --> d((d))
    b((b)) -- -1 --> d((d))
    b((b)) -- -2 --> e((e))
    c((c)) -- 8 --> b((b))
    c((c)) -- -3 --> e((e))
    d((d)) -- 4 --> t((t))
    e((e)) -- 2 --> t((t))
    e((e)) -- 3 --> c((c))
    
```

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

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

Progettazione di Algoritmi A.A. 2021-22
A. De Bonis

84

Segmented Least Squares

- **Minimi quadrati.**
 - Problema fondamentale in statistica e calcolo numerico.
 - Dato un insieme P di n punti del piano $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$.
 - Trovare una linea L di equazione $y = ax + b$ che minimizza la somma degli errori quadratici.

$$Error(L,P) = \sum_{i=1}^n (y_i - ax_i - b)^2$$

- Chiameremo questa quantita' "Errore di L rispetto a P"
- Chiameremo "Errore minimo per P", il minimo valore di Error(L,P) su tutte le possibili linee L
- **Soluzione.** Analisi \Rightarrow il **minimo errore** per un dato insieme P di punti si ottiene usando la linea di equazione $y = ax + b$ con a e b dati da

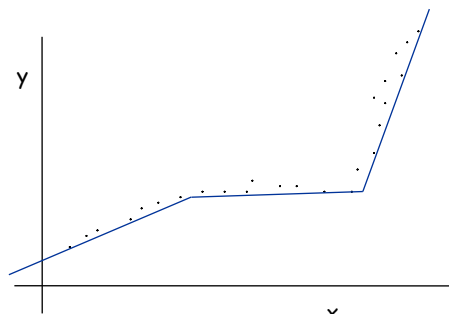
$$a = \frac{n \sum_i x_i y_i - (\sum_i x_i) (\sum_i y_i)}{n \sum_i x_i^2 - (\sum_i x_i)^2}, \quad b = \frac{\sum_i y_i - a \sum_i x_i}{n}$$

85

Segmented Least Squares

- L'errore minimo per alcuni insiemi input di punti puo` essere molto alto a causa del fatto che i punti potrebbero essere disposti in modo da non poter essere ben approssimati usando un'unica linea.

Esempio: i punti in figura non possono essere ben approssimati usando un'unica linea. Se pero` usiamo tre linee riusciamo a ridurre di molto l'errore.



Progettazione di Algoritmi A.A. 2021-22
A. De Bonis

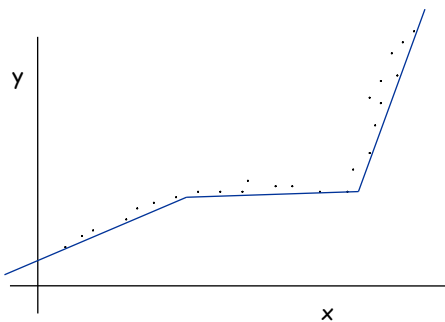
86

86

Segmented Least Squares

Segmented least squares.

- In generale per ridurre l'errore avremo bisogno di una sequenza di linee intorno alle quali si distribuiscono sottoinsiemi di punti di P .
- Ovviamente se ci fosse concesso di usare un numero arbitrariamente grande di segmenti potremmo ridurre a zero l'errore:
 - Potremmo usare una linea per ogni coppia di punti consecutivi.
- **Domanda.** Qual e` la misura da ottimizzare se vogliamo trovare un giusto compromesso tra accuratezza della soluzione e parsimonia nel numero di linee usate?



Il problema e` un caso particolare del problema del change detection che trova applicazione in data mining e nella statistica: data una sequenza di punti, vogliamo identificare alcuni punti della sequenza in cui avvengono delle variazioni significative (in questo caso quando si passa da un'approssimazione lineare ad un'altra)

Progettazione di Algoritmi A.A. 2021-22
A. De Bonis

87

87

Segmented Least Squares

Formulazione del problema Segmented Least Squares.

- Dato un insieme P di n punti nel piano $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ con $x_1 < x_2 < \dots < x_n$, **vogliamo partizionare P in un certo numero m di sottoinsiemi P_1, P_2, \dots, P_m in modo tale che**
- Ciascun P_i e' costituito da punti contigui lungo l'asse delle ascisse
 - P_i viene chiamato segmento
- La sequenza di linee L_1, L_2, \dots, L_m **ottime** rispettivamente per P_1, P_2, \dots, P_m minimizzi la **somma delle 2 seguenti quantita:**
 - 1) La somma **E** degli m errori minimi per P_1, P_2, \dots, P_m (l'errore minimo per il segmento P_i e' ottenuto dalla linea L_i)

$$E = \text{Error}(L_1, L_2, \dots, L_m; P_1, P_2, \dots, P_m) = \sum_{j=1}^m \sum_{(x_i, y_i) \in P_j} (y_i - a_j x_i - b_j)^2$$

- 2) Il numero **m** di linee (pesato per una certa costante input **C**>0)

La quantita' da minimizzare e' quindi **E + Cm** (penalita').

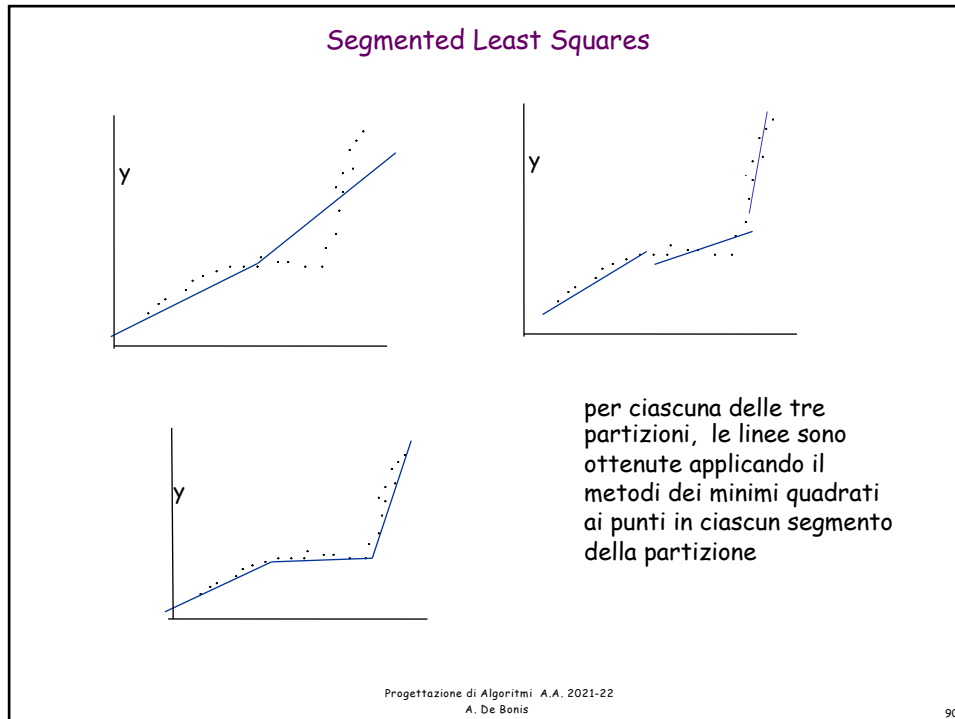
Segmented Least Squares

Formulazione del problema Segmented Least Squares.

- **Input:** insieme P di n punti nel piano $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ con $x_1 < x_2 < \dots < x_n$, e una costante **C**>0
- **Obiettivo:** Trovare una partizione P_1, P_2, \dots, P_m di P tale che
 1. ciascun P_i e' costituito da punti contigui lungo l'asse delle ascisse
 2. la penalita' **E + Cm** sia la piu' piccola possibile
- dove E e' la somma degli m errori minimi per P_1, P_2, \dots, P_m

$$E = \sum_{j=1}^m \sum_{(x_i, y_i) \in P_j} (y_i - a_j x_i - b_j)^2$$

Per ogni j nella sommatoria a_j e b_j sono ottenuti applicando il metodo dei minimi quadrati ai punti di P_j



90

Segmented Least Squares

- Il numero di partizioni in segmenti dei punti in P è esponenziale \rightarrow ricerca esaustiva e inefficiente
- La programmazione dinamica ci permette di progettare un algoritmo efficiente per trovare una partizione di penalità minima
- A differenza del problema dell'Interval Scheduling Pesato in cui utilizzavamo una ricorrenza basata su due possibili scelte, per questo problema utilizzeremo una ricorrenza basata su un numero polinomiale di scelte.

Progettazione di Algoritmi A.A. 2021-22
A. De Bonis

91

Approccio basato sulla programmazione dinamica

Notazione

- Sia p_j un qualsiasi punto input,
 $OPT(j)$ = costo minimo della penalità per i punti p_1, p_2, \dots, p_j .
- Siano p_i e p_j due dei punti input, con $i \leq j$,
 $e(i, j)$ = minimo errore per l'insieme di punti $\{p_i, p_{i+1}, \dots, p_j\}$.

$$e(i, j) = \sum_{k=i}^j (y_k - a_{ij}x_k - b_{ij})^2$$

dove a_{ij} e b_{ij} sono ottenuti applicando il metodo dei minimi quadrati ai punti p_1, p_2, \dots, p_j

Per computare $OPT(j)$, osserviamo che

- se l'ultimo segmento nella partizione di $\{p_1, p_2, \dots, p_j\}$ è costituito dai punti p_i, p_{i+1}, \dots, p_j per un certo i , allora
- **penalità** = $OPT(i-1) + e(i, j) + C$.
- Il valore della **penalità** cambia in base alla scelta di i
- Il valore $OPT(j)$ è ottenuto in corrispondenza dell'indice i che minimizza $OPT(i-1) + e(i, j) + C$.

Progettazione di Algoritmi A.A. 2021-22
A. De Bonis

92

92

Approccio basato sulla programmazione dinamica

- Da quanto detto nella slide precedente, si ottiene la seguente formula per $OPT(j)$:

$$OPT(j) = \begin{cases} 0 & \text{se } j=0 \\ \min_{1 \leq i \leq j} \{ e(i, j) + C + OPT(i-1) \} & \text{altrimenti} \end{cases}$$

Progettazione di Algoritmi A.A. 2021-22
A. De Bonis

93

93

Segmented Least Squares: Algorithm

```

INPUT:  $n, p_1, \dots, p_n, c$ 

Segmented-Least-Squares() {
   $M[0] = 0$ 
  for  $j = 1$  to  $n$ 
    for  $i = 1$  to  $j$ 
      compute the least square error  $e(i, j)$  for
      the segment  $p_i, \dots, p_j$ 

  for  $j = 1$  to  $n$ 
     $M[j] = \min_{1 \leq i \leq j} (e(i, j) + C + M[i-1])$ 

  return  $M[n]$ 
}

```

Tempo di esecuzione. $O(n^3)$.

- Collo di bottiglia = dobbiamo computare il valore $e(i, j)$ per $O(n^2)$ coppie i, j . Usando la formula per computare la minima somma degli errori quadratici, ciascun $e(i, j)$ è computato in tempo $O(n)$

Algoritmo che produce la partizione

```

Find-Segments( $j$ )
  If  $j = 0$  then
    Output nothing
  Else
    Find an  $i$  that minimizes  $e_{i,j} + C + M[i-1]$ 
    Output the segment  $\{p_i, \dots, p_j\}$  and the result of
      Find-Segments( $i-1$ )
  Endif

```

Esercizio

- Per il saggio di fine anno gli alunni di una scuola saranno disposti in fila secondo un ordine prestabilito e **non modificabile**. La fila sarà suddivisa in gruppi contigui e ciascun gruppo dovrà intonare una parte dell'inno della scuola. Il maestro di canto ha inventato un dispositivo che permette di valutare come si fondono le voci di un gruppo tra di loro. Più **basso** è il punteggio assegnato dal dispositivo ad un gruppo, migliore è l'armonia delle voci. Il maestro vuole ripartire la fila di alunni in gruppi contigui in modo da ottenere entrambi i seguenti obiettivi
 - la somma dei punteggi dei gruppi sia la più piccola possibile
 - il numero totale di gruppi non sia troppo grande per evitare che l'inno debba essere suddiviso in parti troppo piccole. Ogni gruppo fa aumentare il costo della soluzione di un valore costante $g > 0$.

NB: Il dispositivo computa per ogni coppia di posizioni i e j con $i < j$, il valore $f(i,j)$ dove $f(i,j)$ = punteggio assegnato al gruppo che parte dall'alunno in posizione i e termina con l'alunno in posizione j .

continua nella slide
successiva

96

Esercizio

- Si formuli il suddetto problema sotto forma di problema computazionale specificando in cosa consistono un'istanza del problema (input) e una soluzione del problema (output). Occorre definire una funzione costo di cui occorre ottimizzare il valore.
- Si fornisca una formula ricorsiva per il calcolo del valore della soluzione ottima del problema basata sul principio della programmazione dinamica. **Si spieghi in modo chiaro come si ottiene la suddetta formula.**
- Si scriva lo pseudocodice dell'algoritmo che trova il valore della soluzione ottima per il problema.

97