Programmazione dinamica (I parte)

Progettazione di Algoritmi a.a. 2019-20

Matricole congrue a 1

Docente: Annalisa De Bonis

1

Paradigmi della Progettazione degli Algoritmi

- Greedy. Costruisci una soluzione in modo incrementale, ottimizzando (in modo miope) un certo criterio locale.
- Divide-and-conquer. Suddividi il problema in sottoproblemi, risolvi ciascun sottoproblema indipendentemente e combina le soluzioni dei sottoproblemi per formare la soluzione del problema di partenza.
- Programmazione dinamica. Suddividi il problema in un insieme di sottoproblemi che si sovrappongono, cioè che hanno dei sottoproblemi in comune. Costruisci le soluzioni a sottoproblemi via via sempre più grandi **in modo da computare la soluzione di un dato sottoproblema un'unica volta**.
- Nel divide and conquer, se due sottoproblemi condividono uno stesso sottoproblema quest'ultimo viene risolto più volte.

2

Storia della programmazione dinamica

- Bellman. Negli anni '50 è stato il pionere nello studio sistematico della programmazione dinamica.
- Etimologia.
  - Programmazione dinamica = pianificazione nel tempo.

Applicazioni della programmazione dinamica

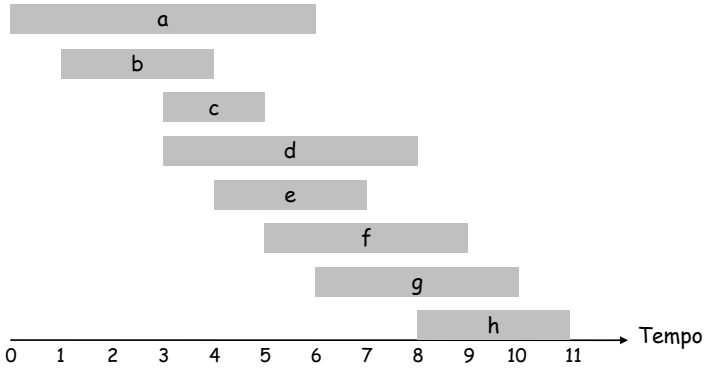
Aree.

- Bioinformatica.
- Teoria dell'informazione
- Ricerca operativa
- Informatica teorica
- Computer graphics
- Sistemi di Intelligenza Artificiale

Interval Scheduling Pesato

Interval scheduling con pesi

- Job j: comincia al tempo s_j, finisce al tempo f_j, ha associato un valore (peso) v_j .
- Due job sono compatibili se non si sovrappongono
- Obiettivo: trovare il sottoinsieme di job compatibili con il massimo peso totale .

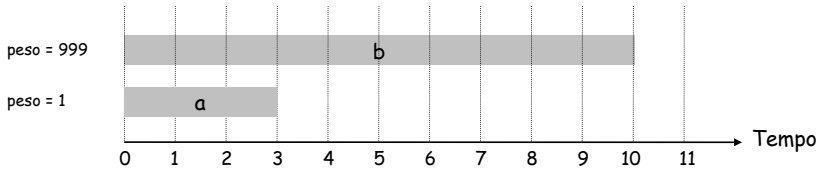


5

Interval scheduling senza pesi

- L'algoritmo greedy Earliest Finish Time funziona quando tutti i pesi sono uguali ad 1.
- Considera i job in ordine non decrescente dei tempi di fine
- Seleziona un job se è compatibile con quelli già selezionati

Osservazione. L'algoritmo greedy Earliest Finish Time può fallire se i pesi dei job sono valori arbitrari.



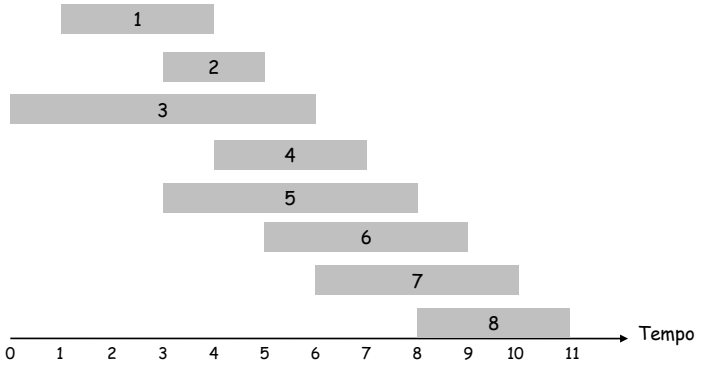
6

Interval Scheduling Pesato

Notazione. Etichettiamo i job in base al tempo di fine :
$f_1 \leq f_2 \leq \ldots \leq f_n$.

Def. p(j) = il più grande indice i < j tale che i è compatibile con j

Ex: p(8) = 5, p(7) = 3, p(2) = 0.



7

Interval Scheduling Pesato: soluzione basata sulla PD

Notazione. OPT(j) = valore della soluzione ottima per l'istanza del problema dell'Interval Scheduling Pesato costituita dalle j richieste con i j tempi di fine più piccoli

Si possono verificare due casi:

- Caso 1: La soluzione ottima per i j job con i tempi di fine piu` piccoli include il job j.
  - In questo caso la soluzione non può usare i job incompatibili { p(j) + 1, p(j) + 2, ..., j - 1 }
  - Deve includere la soluzione ottima al problema dell'Interval Scheduling Pesato **per** i job 1, 2, ..., p(j) (la soluzione non e` formata necessariamente da tutti questi job)

- Caso 2: La soluzione ottima per i j job con i tempi di fine piu` piccoli non contiene il job j.
  - In questo caso la soluzione deve includere la soluzione ottima al problema dell'Interval Scheduling Pesato per i job 1, 2, ..., j-1

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{ v_j + OPT(p(j)),\ OPT(j-1) \} & \text{otherwise} \end{cases}$$

8

Interval Scheduling Pesato: algoritmo ricorsivo **inefficiente**

- Inizialmente Compute-Opt viene invocato con j=n

```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.

Compute p(1), p(2), …, p(n)

Compute-Opt(j) {
   if (j = 0)
      return 0
   else
      return max(vj + Compute-Opt(p(j)), Compute-Opt(j-1))
}
```


9

Interval Scheduling Pesato: algoritmo ricorsivo inefficiente

L'algoritmo computa correttamente OPT(j)

Dim per induzione.

- **Caso base** j=0. Il valore restituito è correttamente 0.
- **Passo Induttivo.** Consideriamo un certo j>0 e supponiamo (ipotesi induttiva) che l'algoritmo produca il valore corretto di OPT(i) per ogni i<j.
- Il valore computato per j dall'algoritmo è

`Compute-Opt(j)= max(vj + Compute-Opt(p(j),Compute-Opt(j-1))`

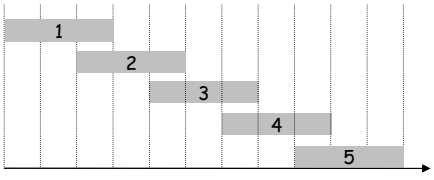
- Siccome per ipotesi induttiva
- `Compute-Opt(p(j))= OPT(p(j))` **e**
- `Compute-Opt(j-1)= OPT(j-1)`

- allora ne consegue che
- `Compute-Opt(j)= max(vj + OPT(p(j)), OPT(j-1))= OPT(j)`

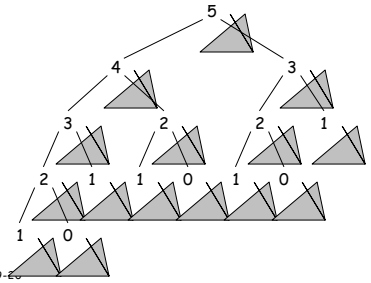
10

Interval Scheduling Pesato: algoritmo ricorsivo inefficiente

- Osservazione. L'algoritmo ricorsivo corrisponde ad un algoritmo di forza bruta perchè ha tempo esponenziale
  - Ciò è dovuto al fatto che
    - ✓ Un gran numero di sottoproblemi sono condivisi da più sottoproblemi
    - ✓ L'algoritmo computa più volte la soluzione ad uno stesso sottoproblema.
- **Esempio**. In questo esempio il numero di chiamate ricorsive cresce come i numeri di Fibonacci.
- N(j)= numero chiamate ricorsive per j. N(j)=N(j-1)+N(j-2)



P(1) = 0, p(2)=0 e p(j) = j-2 per ogni j>1



11

Interval Scheduling Pesato: Memoization

- Osservazione: l'algoritmo ricorsivo precedente computa la soluzione di n+1 sottoproblemi soltanto OPT(0),…,OPT(n). Il motivo dell'inefficienza dell'algoritmo è dovuto al fatto che computa la soluzione ad uno stesso problema più volte.
- Memoization. Consiste nell'immagazzinare le soluzioni di ciascun sottoproblema in un'area di memoria accessibile globalmente.

```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.
Compute p(1), p(2), …, p(n)

for j = 1 to n
   M[j] = empty     ← array globale

M-Compute-Opt(j) {
   if j = 0 Return 0
   if (M[j] is empty)
      M[j] = max(vj + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
   return M[j]
}
```

12

12

Interval Scheduling pesato: Tempo di Esecuzione

Affermazione. La versione "memoized" dell'algoritmo ha tempo di esecuzione $O(n \log n)$.

Fase di inizializzazione: $O(n \log n)$

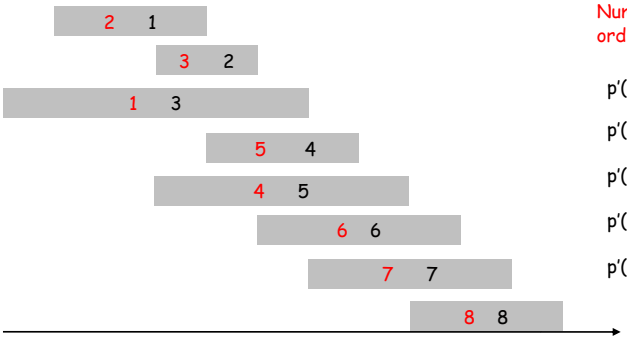
- Ordinamento in base ai tempi di fine: $O(n \log n)$.
- Computazione dei valori $p(\cdot)$: $O(n)$ dopo aver ordinato i job (rispetto ai tempi di inizio e di fine). Siano $a_1,\ldots,a_n$ i job ordinati rispetto ai tempi di inizio e $b_1,\ldots,b_n$ i job ordinati rispetto ai tempi di fine. (si noti che il job con l'i-esimo tempo di inizio non corrisponde necessariamente a quello con l'i-esimo tempo di fine)
  - Si confronta il tempo di fine di b_1 con i tempi di inizio di $a_1,a_2,a_3\ldots$, fino a che non si incontra un job a_j con tempo di inizio $\geq f_1$. Si pone $p'(1)=p'(2)=\ldots=p'(j-1)=0$. Si confronta il tempo di fine di b_2 con i tempi di inizio di $a_j,a_{j+1},a_{j+2}\ldots$, fino a che non si incontra un job a_k con tempo di inizio $\geq f_2$. Si pone $p'(j)=p'(j+1)=p'(j+2)=\ldots=p'(k-1)=1$. Si confronta il tempo di fine di b_3 con i tempi di inizio di $a_k,a_{k+1},a_{k+2}\ldots$, fino a che non si incontra un job a_m con tempo di inizio $\geq f_3$. Si pone $p'(k)=p'(k+1)=p'(k+2)=\ldots=p'(m-1)=2$, e cosi` via.

Continua nel slide successiva

13

Interval Scheduling pesato: Tempo di Esecuzione

- Si noti che in $p'(j)$, j e` l'indice del job nella sequenza $a_1,\ldots,a_n$.
- Per ottenere il corrispondente valore $p(r)$ basta sostituire a j l'indice del job nella sequenza $b_1,\ldots,b_n$. L'associazione tra il job e la sua posizione nella sequenza $a_1,\ldots,a_n$ e quella nella sequenza $b_1,\ldots,b_n$ puo` essere creata quando si ordinano i job.



Numeri neri= indici nella sequenza ordinata $b_1,\ldots,b_n$
Numeri rossi= indici nella sequenza ordinata $a_1,\ldots,a_n$

$p'(1)=p'(3)=p'(4)=0$ $p(3)=p(2)=p(5)=0$

$p'(5)=1$ $p(4)=1$

$p'(6)=2$ $p(6)=2$

$p'(7)=3$ $p(7)=3$

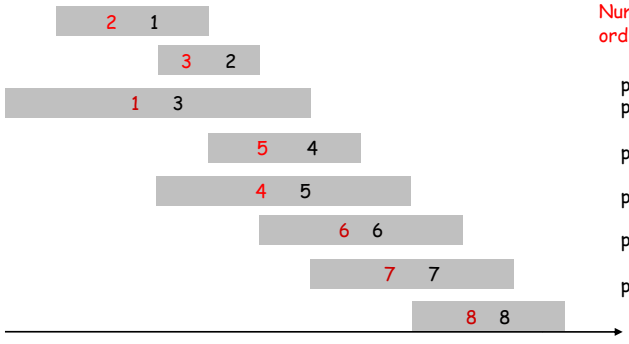
$p'(8)=5$ $p(8)=5$

Continua nel slide successiva

14

Interval Scheduling pesato: Tempo di Esecuzione

- Il tempo per calcolare i valori p(1),…,p(n) (una volta ottenuti i due ordinamenti $a_1,\dots,a_n$ e $b_1,\dots,b_n$) e` O(n) perche` dopo ogni confronto l'algoritmo passa a considerare o il prossimo job nell'ordinamento $b_1,\dots,b_n$ (nel caso di confronto tra due job compatibili) o il prossimo job nell'ordinamento $a_1,\dots,a_n$ (nel caso di confronto tra due job incompatibili).



Numeri neri= indici nella sequenza ordinata $b_1,\dots,b_n$
Numeri rossi= indici nella sequenza ordinata $a_1,\dots,a_n$

p'(1)=p'(2)=p'(3)=p'(4)=0
p(3)=p(1)=p(2)=p(5)=0

p'(5)=1 p(4)=1

p'(6)=2 p(6)=2

p'(7)=3 p(7)=3

p'(8)=5 p(8)=5

Continua nel slide successiva

Interval Scheduling pesato: Tempo di Esecuzione

Affermazione: **`M-Compute-Opt(n)`** richiede O(n)
Dim.

- **`M-Compute-Opt(j)`**: escludendo il tempo per le chiamate ricorsive, ciascuna invocazione prende tempo O(1) e fa una delle seguenti cose
  - (i) restituisce il valore esistente di `M[j]`
  - (ii) riempie l'entrata `M[j]` facendo due chiamate ricorsive

- Per stimare il tempo di esecuzione di **`M-Compute-Opt(j)`** dobbiamo stimare il numero totale di chiamate ricorsive innescate da **`M-Compute-Opt(j)`**
  - Abbiamo bisogno di una misura di come progredisce l'algoritmo
  - Misura di progressione Φ = # numero di entrate non vuote di `M[]`.
  - inizialmente $\Phi = 0$ e durante l'esecuzione si ha sempre $\Phi \leq n$.
  - per far crescere Φ di 1 occorrono al piu` 2 chiamate ricorsive.
  - quindi per far andare Φ da 0 a j, occorrono al piu` 2j chiamate ricorsive per un tempo totale di O(j)

- Il tempo di esecuzione di **`M-Compute-Opt(n)`** e` quindi O(n).

N.B. O(n), una volta ordinati i job in base ai valori di inizio.

Memoization nei linguaggi di programmazione

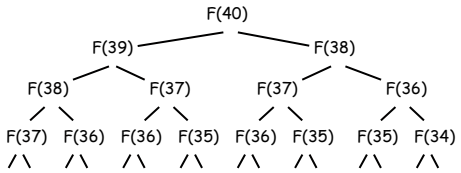
- Automatica. Molti linguaggi di programmazione funzionale, quali il Lisp, prevedono un meccanismo per rendere automatica la memoization

```
(defun F (n)
  (if
    (<= n 1)
    n
    (+ (F (- n 1)) (F (- n 2)))))
```

Lisp (efficiente)

```
static int F(int n) {
   if (n <= 1) return n;
   else return F(n-1) + F(n-2);
}
```

Java (esponenziale



17

Interval Scheduling Pesato: Trovare una soluzione

- Domanda. Gli algoritmi di programmazione dinamica computano il valore ottimo. E se volessimo trovare la soluzione ottima e non solo il suo valore?
- Risposta. Facciamo del post-processing (computazione a posteriori).

```
Run M-Compute-Opt(n)
Run Find-Solution(n)

Find-Solution(j) {
   if (j = 0)
      output nothing
   else if (v_j + M[p(j)] > M[j-1])
      print j
      Find-Solution(p(j))
   else
      Find-Solution(j-1)
}
```


18

Interval Scheduling Pesato: Bottom-Up

Programmazione dinamica bottom-up

Per capire il comportamento dell'algoritmo di programmazione dinamica e` di aiuto formulare una versione iterativa dell'algoritmo.

```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.

Compute p(1), p(2), …, p(n)

Iterative-Compute-Opt {
   M[0] = 0
   for j = 1 to n
      M[j] = max(vj + M[p(j)], M[j-1])
}
```

Correttezza: Con l'induzione su j si puo` dimostrare che ogni entrata M[j] contiene il valore OPT(j)

Tempo di esecuzione: n iterazioni del for, ognuna della quali richiede tempo $O(1) \rightarrow$ tempo totale $O(n)$

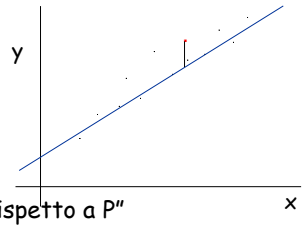
19

19

Segmented Least Squares

- Minimi quadrati.
  - Problema fondazionale in statistica e calcolo numerico.
  - Dato un insieme P di n punti del piano $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$.
  - 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

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

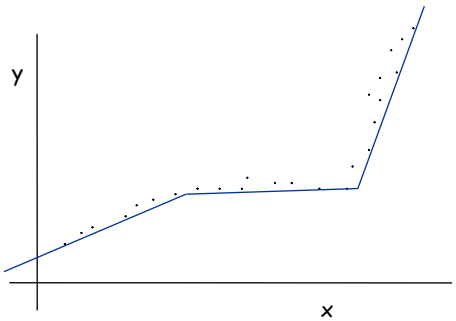
20

20

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.

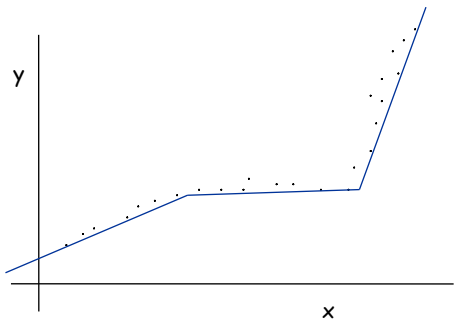


21

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)

22

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) , . . . , (x_n, y_n) con $x_1 < x_2 < \ldots < x_n$, vogliamo partizionare P in un certo numero m di sottoinsiemi $P_1, P_2, \ldots, P_m$ 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 $L_1, L_2, \ldots, L_m$ ottime rispettivamente per $P_1, P_2, \ldots, P_m$ minimizzi la somma delle 2 seguenti quantita:

1) La somma E degli m errori minimi per $P_1, P_2, \ldots, P_m$ (l'errore minimo per il segmento P_i e` ottenuto dalla linea L_i)

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

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) , . . . , (x_n, y_n) con $x_1 < x_2 < \ldots < x_n$, e una costante $C>0$
- Obiettivo: Trovare una partizione $P_1, P_2, \ldots, P_m$ di P tale che

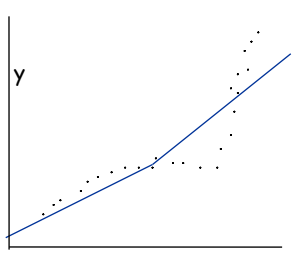
1. ciascun P_i e` costituito da punti contigui lungo l'asse delle ascisse
2. la penalita` E + C m sia la piu` piccola possibile

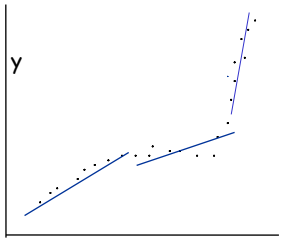
- dove E e` la somma degli m errori minimi per $P_1, P_2, \ldots, P_m$

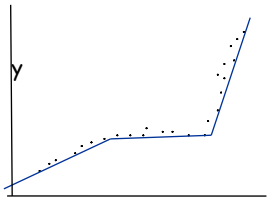
$$E = \sum_{j=1}^{m} \sum_{(x_i, y_i) \in P_j} (y_i - a_j x_i - b_j)^2$$

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

Segmented Least Squares







per ciascuna delle tre partizioni, le linee sono ottenute applicando il metodi dei minimi quadrati ai punti in ciascun segmento della partizione

25

Segmented Least Squares

- Il numero di partizioni in segmenti dei punti in P e` esponenziale → ricerca esaustiva e` inefficiente
- La programmazione dinamica ci permette di progettare un algoritmo efficiente per trovare una partizione di penalita` 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.

26

Approccio basato sulla programmazione dinamica

Notazione

- OPT(j) = costo minimo della penalita` per i punti $p_1, p_2, \ldots, p_j$.
- e(i,j) = minimo errore per l'insieme di punti $\{p_i, p_{i+1}, \ldots, p_j\}$.

$$e(i,j) = \sum_{(x_i, y_i) \in P_j} (y_i - a_j x_i - b_j)^2$$

Per computare OPT(j), osserviamo che

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

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

Segmented Least Squares: Algorithm

```
INPUT: n, p1,…,pn, 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 pi,…, pj

   for j = 1 to n
      M[j] = min 1 ≤ i ≤ 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) e` 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, ..., p_j} and the result of
          Find-Segments(i − 1)
  Endif
```


29

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 sara` suddivisa in gruppi contigui e ciascun gruppo dovra` 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. Piu` **basso** e` il punteggio assegnato dal dispositivo ad un gruppo, migliore e` 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 piu` 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 in≤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

30

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.

31