Algoritmi greedy
VI parte

Progettazione di Algoritmi a.a. 2022-23
Matricole congrue a 1
Docente: Annalisa De Bonis

151

151

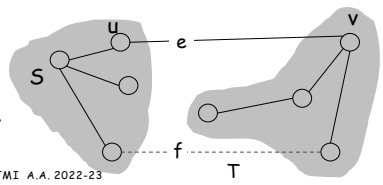
Proprietà del ciclo

- Proprietà del ciclo . Sia C un ciclo e sia e=(u,v) un arco di costo massimo tra quelli appartenenti a C. Esiste un minimo albero ricoprente che non contiene l'arco e.

Dim. (tecnica dello scambio)

- Sia T un MST che contiene l'arco e. Dimostriamo che possiamo sostituire e con un altro arco di C in modo da ottenere ancora uno MST.
- Se rimuoviamo l 'arco e da T disconnettiamo T in due alberi uno contenente u e l'altro contenente v. Chiamiamo S l'insieme dei nodi dell'albero che contiene u.
- Il ciclo C contiene due percorsi per andare da u a v. Un percorso è costituito dall'arco e=(u,v) mentre l'altro va da u a v attraverso gli archi di C diversi da (u,v). Tra questi archi deve essercene uno che attraversa il taglio [S,V-S] altrimenti non sarebbe possibile andare da u che sta in S a v che sta in V-S. Sia f questo arco.

Se al posto di e inseriamo in T l'arco f, otteniamo un albero ricoprente T' di costo $c(T')=c(T)- c_e+c_f$
Siccome $c_f \leq c_e$ allora $c(T') \leq c(T)$. Siccome T è per ipotesi uno MST allora deve essere $c(T') = c(T)$ → T' è anch'esso un MST.



152

Correttezza dell'algoritmo Inverti-Cancella

L'algoritmo Inverti-Cancella produce un MST.

Dim.

Sia R_i l'insieme di archi rimossi fino ad un certo passo i da Inverti-Cancella. Dimostriamo per induzione che per ogni i esiste un MST che non contiene nessuno degli archi in R_i. La base per i=0 e` banalmente verificata visto che non è stato cancellato ancora niente.

Passo induttivo. Supponiamo che per i<=j esista un MST che non contiene nessuno degli archi di R_i e dimostriamo che cio` vale anche R_{j+1}. Se al passo j+1 l'algoritmo non cancella l'arco e_{j+1} allora il passo induttivo e dimostrato banalmente perche' $R_{j+1}=R_j$. Se viene cancellato l'arco e_{j+1} allora vuol dire che l'arco e_{j+1} si trova su un ciclo C **del grafo ottentuto rimuovendo da G gli archi di R_j** (altrimenti la sua rimozione avrebbe disconnesso u e v). Chiamiamo G_j il grafo ottenuto cancellando da G gli archi di R_j.

- Dal momento che gli archi vengono esaminati in ordine non crescente di costo, l'arco e_{j+1} ha costo massimo tra gli archi sul ciclo C di G_j
- La proprietà del ciclo implica che esiste un MST T di G_j che non contiene e_{j+1}
- Siccome G_j non contiene alcun arco di R_j allora T è un albero ricoprente di G di costo minimo tra quelli che non contengono archi di $R_j \cup \{e_{j+1}\} = R_{j+1}$

continua

153

153

Correttezza dell'algoritmo Inverti-Cancella

- L'ipotesi induttiva ci dice che esiste almeno un MST T' di G che non contiene nessun arco di R_j .
- Siccome T' è un MST di G allora c(T')<=c(T)
- Ovviamente T' è un albero ricoprente anche di G_j dal momento che contiene solo archi di G_j
- Siccome T è un MST di G_j allora c(T)<=c(T').
- Si ha quindi c(T)=c(T') e cioe` che T è u MST anche per G.
- Inoltre T non contiene nessun arco di R_j e non contiene l'arco e_{j+1} . Abbiamo quindi dimostrato che esiste un MST T di G che non contiene gli archi di $R_j \cup \{e_{j+1}\} = R_{j+1}$ e questo conclude il passo induttivo.

continua

154

154

Correttezza degli algoritmi quando i costi non sono distinti

- Abbiamo dimostrato che per ogni passo di Inverti-Cancellla esiste un MST di G che non contiene nessuno degli archi rimossi fino a quel passo.
- Cio` vale anche alla fine (al passo j=m) quando l'algoritmo resta con n-1 archi e restituisce l'albero formato da quegli n-1 archi.
- Infatti dire che esiste un MST che non contiene nessun arco di R_m equivale a dire che gli archi non cancellati formano un MST.

155

Clustering

- Clustering. Dato un insieme U di n oggetti $p_1, \ldots, p_n$, vogliamo classificarli in gruppi coerenti
- Esempi: foto, documenti, microorganismi.

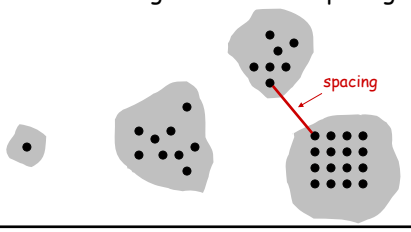
- Funzione distanza. Associa **ad ogni coppia di oggetti** un valore numerico che indica la vicinanza dei due oggetti
  - Questa funzione dipende dai criteri in base ai quali stabiliamo che due oggetti sono simili o appartengono ad una stessa categoria.
  - Esempio: numero di anni dal momento in cui due specie hanno cominciato ad evolversi in modo diverso

Problema. Dividere i punti in cluster (gruppi) in modo che punti in cluster distinti siano distanti tra di loro.
- Classificazione di documenti per la ricerca sul Web.
- Ricerca di somiglianze nei database di immagini mediche
- Classificazione di oggetti celesti in stelle, quasar, galassie.

156

Clustering con Massimo Spacing

- k-clustering. Partizione dell'insieme U in k sottoinsiemi non vuoti (cluster).
- Funzione distanza. Soddisfa le seguenti proprietà
  - $d(p_i, p_j) = 0$ se e solo se $p_i = p_j$
  - $d(p_i, p_j) \geq 0$
  - $d(p_i, p_j) = d(p_j, p_i)$
- Spacing. Distanza più piccola tra due oggetti in cluster differenti
- Problema del clustering con massimo spacing.
- Input: un intero k, un insieme U, una funzione distanza sull'insieme delle coppie di elementi diU.
- Output: un k-clustering con massimo spacing.



k = 4

157

157

Algoritmo greedy per il clustering

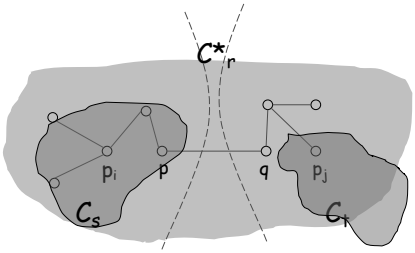
- Algoritmo basato sul single-link k-clustering.
  - Costruisce un grafo sull'insieme di vertici U in modo che alla fine abbia k componenti connesse. Ogni componente connessa corrisponderà ad un cluster.
  - Inizialmente il grafo non contiene archi per cui ogni vertice u è in un cluster che contiene solo u.
  - Ad ogni passo trova i due oggetti x e y più vicini e tali che x e y sono in cluster distinti. Aggiunge un arco tra x e y.
  - Va avanti fino a che ha aggiunto n-k archi: a quel punto ci sono esattamente k cluster (ogni arco riduce di 1 il numero di cluster)
- Osservazione. Questa procedura corrisponde ad eseguire l'algoritmo di Kruskal su un grafo **completo** in cui i costi degli archi rappresentano la distanza tra due oggetti (costo dell'arco (u,v) = d(u,v)). L'unica differenza è che l'algoritmo si ferma prima di inserire i k-1 archi più costosi dello MST.
- NB: Corrisponde a cancellare i k-1 archi più costosi da un MST

158

158

Algoritmo greedy per il clustering: Analisi

- Teorema. Sia C^* il clustering $C^*_1, \ldots, C^*_k$ ottenuto cancellando i k-1 archi più costosi da un MST T del grafo completo in cui ogni arco e=(u,v) ha costo $c_e = d(u,v)$. C^* è un k-clustering con massimo spacing.
- Dim. Sia C un clustering $C_1, \ldots, C_k$ diverso da C^*
  - Sia d^* lo spacing di C^*. La distanza d^* corrisponde al costo del (k-1)-esimo arco più costoso dello MST T (il meno costoso tra quelli cancellati dallo MST T)
  - Facciamo vedere che lo spacing di C non è maggiore di d^*
  - Siccome C e C^* sono diversi allora devono esistere due oggetti p_i e p_j che si trovano nello stesso cluster in C^* e in cluster differenti in C. Chiamiamo rispettivamente C^*_r il cluster di C^* che contiene p_i e p_j e C_s e C_t i due cluster di C contenenti p_i e p_j , rispettivamente.



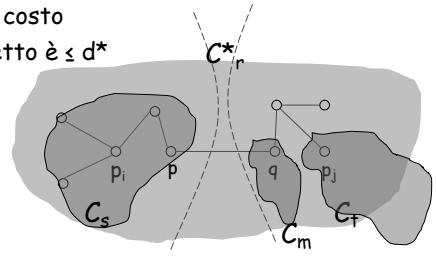
159

159

Algoritmo greedy per il clustering: Analisi

- Sia P il percorso tra p_i e p_j che passa esclusivamente per nodi di C^*_r (cioe` attraverso archi selezionati da Kruskal nei primi n-k passi) e sia q il primo vertice di P che non appartiene a C_s. Indichiamo con C_m la componente in cui si trova q.
- Sia p il predecessore di q lungo P. Il nodo p è in C_s in quanto q è il primo nodo incontrato lungo il percorso che non sta in C_s
- Tutti gli archi sul percorso P e quindi anche (p,q) hanno costo $\leq d^*$ in quanto sono stati scelti da Kruskal nei primi n-k passi.
- Lo spacing di C è minore o uguale della distanza tra i punti piu` vicini di C_s e C_m e quindi è anche minore del costo dell'arco (p,q) che per quanto detto è $\leq d^*$

Abbiamo quindi dimostrato che un qualsiasi clustering ha spacing minore o uguale di d^* per cui il clustering trovato dal nostro algorimo massimizza lo spacing.



160

160

Problema del caching offline ottimale

- Caching. Una cache è un tipo di memoria a cui si può accedere molto velocemente. Una cache permette accessi più veloci rispetto alla memoria principale ma ha dimensioni molto più piccole.

- Possiamo pensare ad una cache come ad un posto in cui possiamo tenere a portata di mano le cose che ci servono ma che è di dimensione limitata per cui dobbiamo riflettere bene su cosa mettervi e su cosa togliere per evitare che ci serva qualcosa che non abbiamo a portata di mano.

  - Cache hit: elemento già presente nella cache quando richiesto.
  - Cache miss: elemento non presente nella cache quando richiesto: occorre portare l'elemento richiesto nella cache e se la cache è piena occorre espellere dalla cache alcuni elementi per fare posto a quelli richiesti.

Problema del caching offline ottimale

Caching. Formalizziamo il problema come segue:

- Memoria centrale contenente un insieme U di n elementi
- Cache con capacità di memorizzare k elementi.
- Sequenza di m richieste di elementi $d_1, d_2, \ldots, d_m$ di U fornita in input in modo offline (tutte le richieste vengono rese note all'inizio). Non molto realistico!
- Assumiamo che inizialmente la cache sia piena, cioè contenga k elementi

Def. Un **eviction schedule ridotto** è uno scheduling degli elementi da espellere, cioè una sequenza che indica quale elemento espellere **quando c'è bisogno di far posto ad un elemento richiesto** che non è in cache.

Un eviction schedule non ridotto è uno scheduling che può decidere di inserire in cache un elemento che non è stato richiesto

Problema del caching offline ottimale

Un eviction schedule ridotto inserisce in cache un elemento solo nel momento in cui è richiesto e se non è presente già in cache al momento della richiesta.

Osservazione. In un eviction schedule ridotto il numero di inserimenti in cache è uguale al numero di cache miss.

Obiettivo. Un eviction schedule ridotto che minimizzi il numero di inserimenti (o equivalentemente di cache miss).

.

Eviction Schedule ridotto

a	a	b	c
a	a	x	c
c	a	d	c
d	a	d	b
a	a	c	b
b	a	x	b
c	a	c	b
a	a	c	b
a	a	c	b

Uno schedule non ridotto

a	a	b	c
a	a	b	c
c	a	b	c
d	a	d	c
a	a	d	c
b	a	d	b
c	a	c	b
a	a	c	b
a	a	c	b

Uno schedule ridotto

caching offline ottimale: Farthest-In-Future

Farthest-in-future. Quando viene richiesto un elemento che non è presente in cache, espelli dalla cache l'elemento che sarà richiesto più in là nel tempo o che non sarà più richiesto.

Cache in questo momento: a b c d e f

Richieste future: g a b c e d a b b a c d e a f a d e f g h ...

cache miss

Espelli questa

Teorema. [Belady, 1960s] Farthest-in-future è uno schedule (ridotto) ottimo.

Dim. La tesi del teorema è intuitiva ma la dimostrazione è sottile.

Problema del caching offline ottimale

Esempio.

Cache di dimensione k = 2,
Inizialmente la cache contiene ab,
Le richieste sono a, b, c, b, c, a, a, b.
Usiamo farthest-in-future:
Quando arriva la prima richiesta di c viene espulso a perchè a verrà richiesto più in là nel tempo rispetto a b.
Quando arriva la seconda richiesta di a viene espulso c perchè c non viene più richiesto

Scheduling ottimo: 2 cache miss.

richieste	cache	
a	a	b
b	a	b
c	c	b
b	c	b
c	c	b
a	a	b
a	a	b
b	a	b

Implementazione dell'algoritmo di Belady

- Siano $d_1, d_2, \ldots, d_m$ le richieste in ordine dei tempo di arrivo
- Per ogni elemento d nella sequenza delle m richieste, l'algoritmo mantiene la lista L[d] contenente le posizioni in cui d appare nella sequenza. Ad esempio, se le richieste sono a, b, c, b, c, a, a, b allora L[a]=<1,6,7>, L[b]=<2,4,8>, L[c]=<3,5>.
- L'algoritmo mantiene inoltre una coda a priorita` Q.
  - Per ogni elemento d in cache, la coda a priorita` Q contiene un'entrata (k,d), dove la chiave k è un intero che indica il punto della sequenza in cui verra` richiesto nuovamente l'elemento d. Se k=m+1 allora vuol dire che d non verra` piu` richiesto,

167

Implementazione dell'algoritmo di Belady

- Ogni volta arriva una nuova richiesta, l'algoritmo si comporta come segue.
- Se l'elemento richiesto d_j non è in cache, l'algoritmo
  - estrae da Q l'entrata (h,d) con chiave h piu` grande
  - espelle d dalla cache e inserisce. d_j **in cache**.
  - rimuove il primo elemento di $L[d_j]$ in modo che in testa venga a trovarsi la prossima posizione della sequenza in cui verra richiesto d_j.
  - se $L[d_j]$ non è vuota, inserisce in Q l'entrata (p,d_j) dove p è l'intero che si trova ora in testa a $L[d_j]$; altrimenti inserisce in Q l'entrata $(m+1,d_j)$
- Se l'elemento richiesto d_j è in cache, l'algoritmo
  - rimuove il primo elemento di $L[d_j]$ in modo che in testa venga a trovarsi la prossima posizione della sequenza in cui verra richiesto d_j.
  - se $L[d_j]$ non è vuota, rimpiazza la chiave di d_j in Q con p, dove p è l'elemento che si trova ora in testa a $L[d_j]$; altrimenti rimpiazza questa chiave con m+1.

168

```
Input: requests d1,d2,…,dm arranged in ascending order of arrival time
For each element d, let L[d] the list of positions j s.t.dj=d;
initially L[d]=∅
Let Q be a priority queue //entries associated with elements in cache
for j = 1 to m {                                                    ┐ O(m+k log k )
if(list L[dj] is empty and dj is in the cache)                      │ k = dimensione
      insert (j,dj) in Q     //j is the key                         │ cache
append j to list L[dj]                                              ┘
}
for j = 1 to m {                                                    ┐
   if (dj is in the cache){                                         │
        remove first element from L[dj]                             │
        if(L[dj] is empty)                                          │ O(m log k )
          replace key of dj with m+1 in Q                           │ k = dimensione
        else                                                        │ cache
          {p ← first element of L[dj]                               │
           replace key of dj with p in Q }                          │
      }                                                             │
   else{       //dj needs to be brought into the cache             │
       (h,dh) ← ExtractMax(Q)                                       │
        evict dh from the cache and bring dj to the cache           │
        remove first element from L[dj]                             │
        if(L[dj] is NOT empty) {                                    │
            p ← first element of L[dj]                              │
            insert (p,dj ) in Q  }                                  │
        else insert (m+1,dj) in Q                                   ┘
  }
```

169

Implementazione dell'algoritmo di Belady

- Per ogni elemento d_j della sequenza di richieste, il primo for inizializza la lista $L[d_j]$ e se d_j è gia` presente in cache inserisce d_j in Q con chiave uguale alla prima posizione in cui d_j appare nella sequenza delle richieste.
  - Ad esempio se inizialmente a e b sono in cache e la sequenza delle richieste è a, b, c, b, c, a, a, b allora dopo il primo for Q contiene le entrate (a,1) (b,2)
- Nella j-esima iterazione del secondo for, l'if-else gestisce i due seguenti casi.
  - d_j è presente in cache. In questo viene rimosso l'intero in testa a $L[d_j]$ e viene aggiornata la chiave di d_j con l'intero che si trova ora in testa a $L[d_j]$, sempre che $L[d_j]$ non sia vuota . Se $L[d_j]$ è vuota allora la chiave di d_j viene sostituita con m+1.
  - d_j non è presente in cache. In questo caso viene estratta da Q l'entrata (h, d_h) con chiave massima e h viene espulso dalla cache. Viene poi rimosso l'intero che si trova in testa a $L[d_j]$. Se dopo questa rimozione $L[d_j]$ non è vuota allora viene inserita in Q l'entrata (p, d_j), dove p indica l'intero che ora si trova in testa a $L[d_j]$. Se $L[d_j]$ è vuota allora in Q viene inserita l'entrata $(m+1, d_j)$

170

170

Analisi dell'algoritmo di Belady

L'algoritmo nella slide precedente richiede tempo O(m log k) se

- Ad ogni elemento è associato un flag che è true se e solo l'elemento è in cache
- Usiamo un heap binario come coda a priorità
  - assumiamo che l'heap supporti l'operazione changeKey che consente di modificare la chiave di un'entrata arbitraria dell'heap e l'operazione di remove che consente di cancellare un'entrata arbitaria. Queste operazioni possono essere implementata in modo da richiedere tempo O(log k).
- Consideriamo costante il tempo per espellere e inserire ciascun elemento in cache

171

171

Farthest-In-Future: ottimalità

La dimostrazione dell'ottimalità si basa sui seguenti fatti che andremo a dimostrare

1. Ogni schedule può essere trasformato in uno schedule ridotto senza aumentare il numero di inserimenti
2. Ogni schedule ridotto può essere trasformato nello schedule FF senza aumentare il numero di cache miss

- Per la 1 possiamo trasformare uno schedule ottimo S in uno schedule ridotto S' senza aumentare il numero di inserimenti
- Per la 2 possiamo trasformare S' nello schedule FF senza aumentare il numero di cache miss (= numero inserimenti)

→ FF va incontro allo stesso numero di inserimenti dell'algoritmo ottimo ed è quindi anch'esso ottimo

172

172

Farthest-In-Future: ottimalità

1. Un qualsiasi eviction schedule S può essere trasformato in un eviction schedule ridotto S' senza aumentare il numero di inserimenti nella cache.

Dim. Costruiamo S' a partire da S come segue:

- Caso 1. Al tempo t, S porta in cache un elemento x non richiesto. Possono verificarsi i seguenti sottocasi:
  - Caso 1a. al tempo t non viene richiesto alcun elemento oppure viene richiesto un elemento d (diverso da x) e l'elemento d è nella cache di S'. In questo caso S' non fa niente → 1 inserimento in piu` per S e al piu` 1 elemento in comune **in meno** nelle due cache (1 elemento in comune in meno nel caso in cui x non è nella cache di S' ed S espelle un elemento contenuto anche nella cache di S'; altrimenti non cambia niente)
  - Caso 1b. al tempo t viene richiesto un elemento d (diverso da x) e l'elemento d richiesto non è nella cache di S'. In questo caso S' inserisce d nella sua cache e se l'elemento q espulso da S è presente anche nella cache di S' allora espelle q altrimenti espelle un elemento tra quelli non presenti nella cache di S (deve per forza esistere un tale elemento altrimenti la cache di S' avrebbe meno di k elementi)→ differenza tra numero di inserimenti di S ed S' invariato e differenza tra contenuto delle due cache invariata se S' espelle q e x non è nella cache di S' oppure 1 elemento in comune in piu` se S' espelle q e x è nella cache di S' o se S' espelle un elemento che non è nella cache di S.

Farthest-In-Future: ottimalità

- Caso 2. Al tempo t, S porta in cache un elemento d richiesto Possono verificarsi i seguenti sottocasi
  - Caso 2a. d è gia` nella cache di S'. In questo caso S' non fa niente→ 1 inserimento in piu` per S e differenza tra contenuto delle due cache invariata se S espelle un elemento presente nella cache di S oppure 1 elemento in comune in piu` tra le due cache se S espelle un elemento che non è nella cache di S'
  - Caso 2b. d non è nella cache di S' . In questo caso, S' inserisce anch'esso d in cache e se l'elemento q espulso da S è presente anche nella sua cache allora espelle q altrimenti espelle un elemento tra quelli non presenti nella cache di S (deve per forza esistere un tale elemento altrimenti la cache di S' avrebbe meno di k elementi) → differenza tra numero di inserimenti di S ed S' invariato e differenza tra contenuto delle due cache invariata se S' espelle q oppure 1 elemento in comune in piu` tra le due cache se S' espelle un elemento che non è nella cache di S.

Farthest-In-Future: ottimalità

Ora consideriamo i casi in cui S non inserisce niente al tempo t. Ovviamente in questi casi l'elemento richiesto d è gia` nella cache di S al tempo t.
Caso 3: Al tempo t, S non inserisce niente e l'elemento d richiesto è gia nella cache di S'. In questo caso S' non fa niente. → differenza numero inserimenti invariata e differenza tra contenuto delle due cache invariata

Fino ad ora i casi considerati vedono o 1 inserimento in piu` per S o lasciano inalterata la differenza tra il numero di inserimenti dei due scheduling ma puo` accadere che…

- Caso 4. Al tempo t, S non inserisce niente e l'elemento richiesto d non è nella cache di S'. In questo caso S' inserisce in cache d ed espelle un elemento tra quelli non presenti nella cache di S (tale elemento deve esistere perche' la cache di S' contiene gia` d e quindi le due cache hanno al piu` k-1 elementi in comune) → 1 inserimento in piu` per S' e 1 elemento in comune in piu` tra le cache di S ed S'

Il caso 4 è l'unico caso in cui S' effettua un inserimento ed S non fa niente. Nella prossima slide facciamo vedere che se ad un certo tempo t si verifica il caso 4 allora fino a quel momento S' ha fatto almeno un inserimento in meno rispetto a S→ Il numero di inserimenti di S' non supera mai quello di S.

Farthest-In-Future: ottimalità

- Indichiamo con cS e cS' il contenuto della cache di S e di S', rispettivamente.
  a. Affinche' si verifichi il caso 4 è necessario che la cache di S abbia un elemento non presente in quella di S', cioe` $|cS-cS'|>0$.
  b. L'unico caso che puo` far aumentare il numero di elementi di cS-cS' è il caso 1a.
  c. Ogni volta che si verifica il caso 4 il numero di elementi in cS-cS' diminuisce di uno.
- La a, b, e c insieme → il numero di passi in cui si verifica il caso 4 è minore o uguale del numero di passi in cui si verifica che si verifica il caso 1a. →il numero volte in cui S' effettua un inserimento ed S non fa niente è minore o uguale del numero di volte in cui S effettua un inserimento ed S' non fa niente

Farthest-In-Future: ottimalità

Teorema. Sia S uno **scheduling ridotto** che fa le stesse scelte dello scheduling S_{FF} di farthest-in-future per i primi j elementi, per un certo $j \geq 0$. E' possibile costruire uno scheduling ridotto S' che fa le stesse scelte di S_{FF} per i primi j+1 elementi e determina un numero di cache miss non maggiore di quello determinato da S.

Dim.

Produciamo S' nel seguente modo.

- Consideriamo la (j+1)-esima richiesta e sia $d = d_{j+1}$ l'elemento richiesto,
- Siccome S e S_{FF} hanno fatto le stesse scelte fino alla richiesta j-esima, quando arriva la richiesta (j+1)-esima il contenuto della cache per i due scheduling è lo stesso.
  - Caso 1: d è già nella cache. In questo caso sia S_{FF} che S non fanno niente perché entrambi sono ridotti.
  - Caso 2: d non è nella cache ed S espelle lo stesso elemento espulso da S_{FF} .
- In questi due casi basta porre S'=S visto che S ed S_{FF} hanno lo stesso comportamento anche per la (j+1)-esima richiesta.

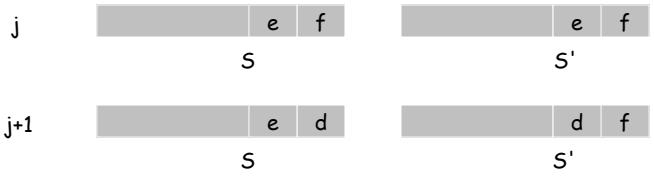
Continua nella prossima slide

177

177

Farthest-In-Future: ottimalità

- Caso 3: d non è nella cache e S_{FF} espelle e mentre S espelle $f \neq e$.
  - Costruiamo S' a partire da S modificando la (j+1)-esima scelta di S in modo che S' espella e invece di f.



  - ora S' ha lo stesso comportamento di S_{FF} per le prime j+1 richieste. Occorre dimostrare che S' riesce ad effettuare successivamente delle scelte che non determinano un numero di cache miss maggiore di quello di S.

Continua nella prossima slide

178

178

Farthest-In-Future: ottimalità

- Dopo la (j+1)-esima richiesta facciamo fare ad S' le stesse scelte di S fino a che, ad un certo tempo j', accade per la prima volta che non è possibile che S ed S' facciano la stessa scelta.
- A questo punto S' deve fare necessariamente una scelta diversa da quella di S. Facciamo però in modo che la scelta di S' renda il contenuto della cache di S' identico a quello della cache di S.
- Da questo punto in poi il comportamento di S' sarà identico a quello di S per cui andrà incontro allo stesso numero di cache miss.

Continua nella prossima slide

179

Farthest-In-Future: ottimalità

Notiamo che siccome i due scheduling fino al tempo j' si sono comportati in modo diverso un'**unica** volta (al passo j+1), il contenuto della cache nei due scheduling differisce in un singolo elemento che è uguale ad e in S ed è uguale a f in S'.

S [... | e] S' [... | f]

Indichiamo con g l'elemento richiesto al tempo j'.

I casi che al tempo j' avrebbero permesso ad S' di fare la stessa scelta di S sono:

- $g \neq e, g \neq f$, g è presente nella cache di S: in questo caso g è presente anche nella cache di S' ed S' non fa niente come S.
- $g \neq e, g \neq f$, g non è presente nella cache di S ed S espelle un elemento diverso da e: in questo caso g non è neanche nella cache di S' ed S' può espellere lo stesso elemento espulso da S.

Nella prossima slide vediamo i casi in cui S' non può fare la stessa scelta di S.

180

Farthest-In-Future: ottimalità

Caso 3.1: $g \neq e$, $g \neq f$, g non è nella cache di S ed S espelle e. In questo caso g non è neanche nella cache di S'. Facciamo in modo che S' espella f. In questo modo dopo il tempo j' il contenuto della cache di S è uguale a quello della cache di S'. Il numero di cache miss di S è lo stesso di S'.

Caso 3.2: $g = f$ ed S espelle e. In questo caso S' non fa niente e da quel momento in poi le cache di S è uguale a quello di S'. Il numero di cache miss di S' è minore di quello di S.

Caso 3.3: $g = f$ ed S espelle $e' \neq e$. In questo caso e' è presente anche nella cache di S' . Facciamo in modo che, al tempo j', S' espella e' ed inserisca e. Da questo momento la cache di S e quella di S' hanno lo stesso contenuto e il numero di cache miss in cui incorreranno i due scheduling sarà lo stesso. Il teorema non è ancora dimostrato per questo caso in quanto S' non è ridotto. Abbiamo però dimostrato che possiamo rendere S' ridotto senza aumentare il numero di inserimenti. Lo scheduling ridotto farà le stesse scelte di S_{FF} per i primi j+1 elementi in quanto sara identico ad S' fino al tempo j'-1.

181

181

Farthest in Future: ottimalità

Resterebbe il caso g=e.

- Notiamo che al tempo j' non può accadere che g=e. Vediamo perché.
  - Al tempo j+1 S_{FF} ha espulso e al posto di f per cui, dopo il tempo j+1, e viene richiesto più tardi di f o non viene richiesto affatto.
  - → Se dopo il tempo j+1 vi è una richiesta di e allora questa richiesta deve essere preceduta da una richiesta di f.

  - Come abbiamo visto nella slide precedente (casi 3.2 e 3.3) la richiesta di f in un tempo successivo al tempo j+1 porterebbe S' a fare una scelta diversa da S ma ciò non è possibile perché stiamo assumendo che j' è il primo momento (successivo al tempo j+1) in cui accade che S' non può fare la stessa scelta di S.

182

182

Farthest-In-Future: ottimalità

2. Ogni schedule ridotto può essere trasformato nello schedule FF senza aumentare il numero di cache miss

Dim.

- Consideriamo un eviction schedule ridotto S.
- Applicando il teorema precedente con j=0, si ha che possiamo trasformare S in uno schedule ridotto S_1 che per la prima richiesta si comporta come S_{FF} e determina un numero di cache miss non maggiore del numero di cache miss di S.
- Applicando il teorema con j=1, si ha che possiamo trasformare S_1 in uno schedule ridotto S_2 che per la prime due richieste si comporta come S_{FF} e determina un numero di cache miss non maggiore del numero di cache miss di S_1 e quindi di S.
- Continuando in questo modo, applicando cioè il teorema precedente per j=0,1,…,m-1, arriviamo ad uno schedule S_m che effettua esattamente le stesse scelte di S_{FF} ($S_m = S_{FF}$) e determina un numero di cache miss non maggiore del numero di cache miss di S.

183

Farthest-In-Future: ottimalità

Teorema. Farthest-in-future produce un eviction schedule S_{FF} ottimo.

Dim.

- Sia S^* uno schedule ridotto ottimo. Per il punto 2 (slide precedente) si ha che S^* può essere trasformato nello schedule S_{FF} senza aumentare il numero di cache miss. Di conseguenza S_{FF} determina lo stesso numero di cache miss di S^* ed è quindi uno schedule ridotto ottimo.
- Osserviamo che S_{FF} è ottimo non solo se restringiamo la nostra attenzione agli schedule ridotti ma è ottimo se consideriamo tutti i tipi di schedule (ridotti e non ridotti) perchè per il punto 1 possiamo trasformare uno schedule ottimo in uno schedule ridotto che effettua lo stesso numero di inserimenti.
  - NB: in questo caso parliamo di numero di inserimenti

184

Il problema del caching nella realtà

- Il problema del caching è tra i problemi più importanti in informatica.
- Nella realtà le richeste non sono note in anticipo come nel modello offline.
- E' più realistico quindi considerare il modello online in cui le richieste arrivano man mano che si procede con l'esecuzione dell'algoritmo.
- L'algoritmo che si comporta meglio per il modello online è l'algoritmo basato sul principio *Least-Recently-Used* o su sue varianti.
- *Least-Recently-Used* (LRU). Espelli la pagina che è stata richiesta meno recentemente
  - Non è altro che il principio Farthest in Future con la direzione del tempo invertita: più lontano nel passato invece che nel futuro
  - E' efficace perché in genere un programma continua ad accedere alle cose a cui ha appena fatto accesso (locality of reference). E' facile trovare controesempi a questo ma si tratta di casi rari.

Esercizio 13 Cap. 4

- Una piccola ditta che si occupa di fotocopiare documenti ogni giorno riceve le richieste di n clienti. La ditta dispone di un'unica fotocopiatrice e fotocopiare i documenti dell'i-esimo cliente richiede tempo t_i. A ciascun cliente i è associato un peso w_i che indica l'importanza del cliente i.
- Indichiamo con C_i il tempo in cui viene terminata la copia dei documenti del cliente i. Se i documenti del cliente i vengono fotocopiati per primi allora $C_i=t_i$. Se i documenti di i vengono fotocopiati dopo quelli di j allora $C_i=C_j+t_i$
- Vogliamo eseguire le fotocopie in un ordine che minimizzi $\sum_{i=1}^{n} w_i C_i$

Soluzione.

- Strategia greedy: Ordiniamo le richieste in modo non crescente rispetto ai valori w_i/t_i
- Dimostriamo che la soluzione greedy è ottima utilizzando la tecnica dello scambio.
- Sia G l'ordinamento greedy e O un ordinamento ottimo. Supponiamo $G \neq O$
- Esistono due richieste j e k tali che j precede k in G e k precede j in O. Inoltre devono esistere due richieste siffatte disposte una dopo l'altra in O.
  - Consideriamo le richieste k e j piu` vicine in O per cui risulta k precede j in O e j precede k in G. Se esistesse una richiesta i che in O viene eseguita tra k e j questa dovrebbe essere eseguita dopo k anche in G altrimenti k e i sarebbero due richieste eseguite in ordine diverso in O e sarebbero tra di loro piu` vicine di k e j .

continua

Soluzione esercizio 13 Cap. 4

Indichiamo con C'_i i tempi in cui termina la copia del cliente i nello scheduling O.
Siano k e j due richieste adiacenti in G eseguite in ordine inverso in O.
Se in O scambiamo k con j otteniamo che la somma $\sum_{i=1}^{n} w_i C'_i$ cambia come segue:

I valori C_i' delle richieste diverse dalla k e la j non cambiano
Sia C' il momento in cui viene soddisfatta la richiesta del cliente che precede k in O
Dopo aver scambiato k con j la somma $\sum_{i=1}^{n} w_i C'_i$ è modificata di una quantità pari a

$- w_k (C'+t_k)+ w_k (C'+t_k+t_j)- w_j (C'+t_k+t_j)+w_j (C'+t_j)= w_k t_j- w_j t_k = t_j t_k (w_k / t_k - w_j /t_j)$

Siccome $w_j/t_j \geq w_k/t_k$ allora $w_j t_k > w_k t_j$ e di conseguenza la somma $\sum_{i=1}^{n} w_i C'_i$ risulta minore o uguale del valore che aveva prima dello scambio.
Possiamo quindi scambiare in O tutte coppie adiacenti che risultano invertite rispetto ad G fino a trasformare O in G. Nel fare questi scambi il valore di $\sum_{i=1}^{n} w_i C'_i$ non aumenta per cui anche G è ottimo.

187