Algoritmi greedy
II parte

Progettazione di Algoritmi a.a. 2018-19
Matricole congrue a 1
Docente: Annalisa De Bonis

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 $d_1, d_2, \ldots, d_n$ 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 ad un certo passo i 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 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	b	c
a	a	b	c

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. [Bellady, 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

Algoritmo di Belady basato sulla strategia Farthest in Future (FF)

```
Assume the requests d_1,d_2,…,d_n are arranged in ascending order
of arrival time
For each element d, let L[d] the list of j s.t.d_j=d
Let Q be a priority queue
for j = 1 to n {
  if(list L[d_j] is empty and d_j is in the cache)
       Insert(Q,d_j,j) //first time dj is requested
  append j to list L[dj] }
for j = 1 to n {
  remove first element from L[d_j]
  if (d_j is NOT in the cache){
  //d_j needs to be brought into the cache
    d_h ← ExtractMax(Q)
    evict d_h from the cache and bring d_j to the cache
   }
  else remove(Q,d_j) //later on it will be inserted
                    //with a new key
  if(L[d_j] is empty) // no further request of d_j
     Insert(Q,d_j,n+1)
  else {
     p ← first element of L[d_j]
     Insert(Q,d_j,p) }
  }
```

O(n+k log k)
k = dimensione cache

O(n log k)
k = dimensione cache

Algoritmo di Belady descrizione delle strutture dati utilizzate

- Per ciascun elemento richiesto d, la lista L[d] delle posizioni in cui d appare nella sequenza input. L[d] e` ordinata in modo crescente.
- Coda a priorita` Q contenente entrate $(k_d,d)$ dove d e` un elemento presente in cache e la chiave k_d e` la posizione della prossima richiesta di d
- **Inizializzazione:** per ogni elemento d della sequenza inseriamo in L[d] le posizioni in cui incontriamo d; inoltre se d e` nella cache inseriamo in Q la coppia (f,d) dove f e` la posizione della sequenza in cui appare d per la prima volta.
- **Aggiornamenti** per ogni iterazione **j** del secondo for:
  - cancelliamo il primo elemento di $L[d_j]$.
  - se d_j e` gia` in cache sostituiamo la chiave di d_j in Q con il primo elemento (dopo la cancellazione) di $L[d_j]$. Se $L[d_j]$ e` vuota sostituiamo la chiave di $d_j$ con n+1 (Nel codice la chiave e` aggiornata cancellando l'entrata e reinserendo l'entrata con la nuova chiave)
  - se d_j non e` in cache estraiamo l'entrata con chiave max da Q. Inseriamo in Q l'entrata $(p, d_j)$ dove p e` il primo elemento (dopo la cancellazione) di $L[d_j]$. Se $L[d_j]$ e` vuota p= n+1

Analisi dell'algoritmi di Belady

Tempo O(n log k) se

- Ad ogni elemento è associato un flag che è true se e solo l'elemento è in cache
- Usiamo un heap come coda a priorità
  - assumiamo che l'heap supporti l'operazione remove che consente di cancellare un'entrata arbitraria (occorrono alcuni accorgimenti affinche' venga eseguita in tempo O(log k))
- Consideriamo costante il tempo per espellere e inserire ciascun elemento in cache

49

Farthest-In-Future: ottimalità

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

- **Fatto.** Ogni schedule può essere trasformato nello schedule FF senza aumentare il numero di cache miss

Possiamo quindi trasformare uno scheduling ridotto **ottimo** nello scheduling FF senza aumentare il numero di cache miss. Ciò implica che FF va incontro allo stesso numero di cache miss dell'algoritmo ottimo ed è quindi anch'esso ottimo

Farthest-In-Future: ottimalità

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

Dim.

- Se ad un certo tempo t, S porta un certo elemento d in cache e d è stato richiesto al tempo t allora S' fa la stessa cosa.
- Se ad un certo tenpo t, S porta un certo elemento d in cache senza che d sia stata richiesto, S' fa finta di fare lo stesso ma di fatto non inserisce niente in cache ed eventualmente inserisce d successivamente quando d è richiesto.
- Il numero totale di inserimenti effettuati da S' è lo stesso di S se tutte le volte che S inserisce un elemento d non richiesto accade che d venga richiesto in seguito. Se invece qualcuno degli elementi inseriti da S non è richiesto nè in quel momento né successivamente allora S' effettua un numero minore di inserimenti.

Farthest-In-Future: ottimalità

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

Dim.

Produciamo S' nel seguente modo.

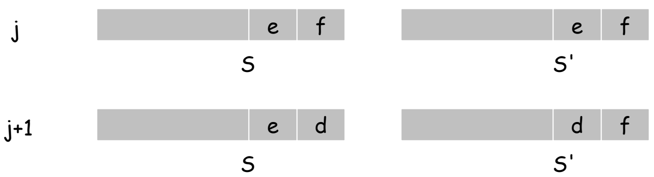
- Consideriamo la (j+1)-esima richiesta e sia $d = d_{j+1}$ l'elemento richiesto,
- Siccome S e S_{FF} hanno fatto le stesse scelte fino alla richiesta j-esima, quando arriva la richiesta (j+1)-esima il contenuto della cache per i due scheduling è lo stesso.
  - Caso 1: d è gia nella cache. In questo 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

52

Farthest-In-Future: ottimalità

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



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

Continua nella prossima slide

53

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 possiamo fare in modo che il comportamento di S' sia identico a quello di S per cui S' andrà incontro allo stesso numero di cache miss di S.

Continua nella prossima slide

Farthest-In-Future: ottimalità

Notiamo che siccome i due schedule fino al tempo j' si sono comportati in modo diverso un'**unica** volta, il contenuto della cache nei due schedule 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'.
- Indichiamo con C_S e $C_{S'}$ rispettivamente gli insiemi degli elementi presenti nella cash di S e e di quelli presenti nella cash di S' al tempo j'

Al tempo j' , S ed S' non possono fare la stessa scelta solo se g non appartiene ad almeno uno degli insiemi C_S e $C_{S'}$ (se g appartenesse ad entrambi gli insiemi allora entrambi gli schedule non farebbero niente al tempo j')
Dobbiamo quindi esaminare i tre sottocasi:

a. $g \notin C_S$ e $g \notin C_{S'}$
b. $g \notin C_S$ e $g \in C_{S'}$
c. $g \in C_S$ e $g \notin C_{S'}$

Farthest in Future: ottimalita`

a. $g \notin C_S$ e $g \notin C_{S'}$:

S ed S' devono entrambi espellere un elemento dalle proprie cache al tempo j'. Sia h l'elemento espulso da S.
Siccome S ed S' non riescono a fare la stessa scelta al tempo j' allora $h \notin C_{S'}$. Deduciamo allora che h=e in quanto e e` l'unico elemento di $C_S$ che non e` presente anche in $C_{S'}$.

In questo caso facciamo in modo che S' espella f. Da j' in poi le due cash conterranno gli stessi elementi → numero di cache miss di S' uguale a quello di S.

56

Farthest in Future: ottimalita`

b. $g \notin C_S$ e $g \in C_{S'}$:
Questo caso e` possibile solo se g=f in quanto f e` l'unico elemento di $C_{S'}$ che non e` presente anche in C_S.

Se S espelle e allora S' non fa niente. Da questo momento in poi le due cache conterranno gli stessi elementi ed S' potra fare le stesse scelte di S → numero di cache miss di S' minore di quello di S

Se S espelle un elemento h ≠e allora $h \in C_{S'}$ e facciamo in modo che S' espella anch'esso h . Da questo momento in poi le due cache conterranno gli stessi elementi ed S' potra` fare le stesse scelte di S → numero di cache miss di S' uguale a quello di S.

Lo schedule S' ottenuto in questo modo non e` ridotto a causa della scelta fatta al tempo j'. Per l'affermazione dimostrata in precedenza, possiamo trasformare questo schedule in uno schedule ridotto senza far aumentare il numero di inserimenti effettuati dallo schedule. **N.B.:** le modifiche necessarie per rendere S' ridotto non riguardano le scelte fatte prima del tempo j' visto che fino a quell'istante S' e` ridotto. Quindi la scelta di S' al tempo j+1 non viene modificata e rimane uguale a quella fatta da S.

57

Farthest in Future: ottimalita`

c. $g \in C_S$ e $g \notin C_{S'}$:

In questo caso g=e in quanto e e` l'unico elemento di $C_S$ che non e` presente anche in $C_{S'}$.

Dimostriamo che questo caso non si puo` verificare

- Al tempo j+1 S_{FF} ha espulso e al posto di f e cio` e` possibile solo se, dopo il tempo j+1, e viene richiesto più tardi di f o non viene richiesto affatto.
- Quindi se al tempo j' viene richiesto e allora deve esistere un tempo t compreso tra il tempo j+1 e il tempo j' (esclusi) in cui arriva una richiesta di f. Una tale richiesta avrebbe pero` impedito ad S ed S' di fare la stessa scelta al tempo t contraddicendo il fatto che j' e` il tempo (successivo al tempo j+1) in cui per la prima volta S ed S' non riescono a fare la stessa scelta.

58

Farthest-In-Future: ottimalità

- Teorema. Farthest-in-future produce un eviction schedule S_{FF} ottimo.
- Dim.
- Consideriamo un eviction schedule ridotto ottimo 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 va incontro allo stesso numero di cache miss di S^*.
- Applicando il teorema precedente 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 va incontro allo stesso numero di cache miss di S_1 e quindi di S^*.
- Continuiamo in questo modo applicando induttivamente il teorema precedente per j=0,1,2,…,n-1 fino a che non arriviamo ad uno schedule S_n che effettua esattamente le stesse scelte di S_{FF} (S_n= S_{FF}) e va incontro allo stesso numero di cache miss di S^*. Si ha quindi che S_{FF} e S^* vanno incontro allo stesso numero di cache miss e di conseguenza S^* è ottimo

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'algortimo 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