

Algoritmi greedy

II parte

Progettazione di Algoritmi a.a. 2016-17

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, \dots, 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.

a	a	b
b	a	b
c	c	b
b	c	b
c	c	b
a	a	b
a	a	b
b	a	b

richieste

cache

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

Assume the requests d_1, d_2, \dots, 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 a priority queue

```
for j = 1 to n {
```

```
  if(list  $L[d_j]$  is empty and  $d_j$  is in the cache)
```

```
    insert ( $j, d_j$ ) in  $Q$     //j is the key
```

```
  append j to list  $L[d_j]$ 
```

```
}
```

```
for j = 1 to n {
```

```
  if ( $d_j$  is in the cache){
```

```
    remove first element from  $L[d_j]$ 
```

```
    if( $L[d_j]$  is empty)
```

```
      replace key of  $d_j$  with  $\infty$  in  $Q$ 
```

```
    else
```

```
      { $p \leftarrow$  first element of  $L[d_j]$ 
```

```
      replace key of  $d_j$  with  $p$  in  $Q$  }
```

```
}
```

```
else{    //dj needs to be brought into the cache
```

```
( $h, d_h$ )  $\leftarrow$  ExtractMax( $Q$ )
```

```
evict  $d_h$  from the cache and bring  $d_j$  to the cache
```

```
remove first element from  $L[d_j]$ 
```

```
if( $L[d_j]$  is NOT empty) {
```

```
   $p \leftarrow$  first element of  $L[d_j]$ 
```

```
  insert ( $p, d_j$ ) in  $Q$  }
```

```
}
```

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

$O(n \log k)$
 $k =$ dimensione
cache

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 `changeKey` che consente di modificare la chiave di un'entrata arbitraria dell'heap e l'operazione di `remove` che consente di cancellare un'entrata arbitraria. 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

Farthest-In-Future: ottimalità

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

- Ogni schedule ridotto 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 elementi inseriti 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 tempo 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 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.

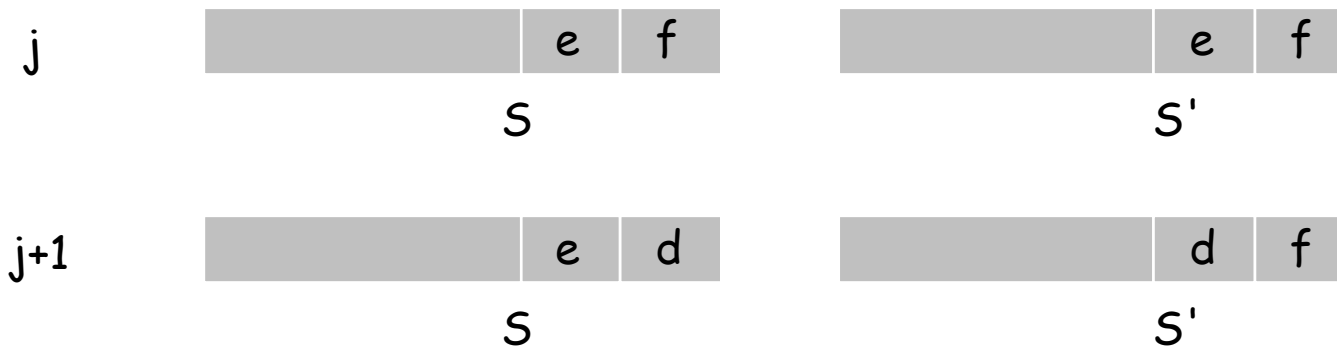
Produciamo S' nel seguente modo.

- 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

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 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 S' riesce ad effettuare successivamente delle scelte che non determinano un numero di cache miss maggiore di quello di S .

Continua nella prossima slide

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

Farthest-In-Future: ottimalità

Notiamo che siccome i due scheduling fino al tempo j' si sono comportati in modo diverso un'unica volta, 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' .



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

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

- $g \neq e, f$; g è presente nella cache di S : in questo caso g è presente anche nella cache di S' non fa niente come S .
- $g \neq e, f$; g non è presente nella cache di S ; 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 .

Farthest-In-Future: ottimalità

▪ **NB:** nel caso 3.3 il numero di inserimenti di S e quello di S' fino al passo j' (incluso) sono uguali. Il numero di inserimenti di S e quello di S' (prima della trasformazione in uno scheduling ridotto) continuano ad essere uguali fino alla fine perché dal passo j' in poi S ed S' hanno in cache gli stessi elementi e possono fare le stesse scelte. Indichiamo con m il numero totale di inserimenti. S è ridotto per cui si ha che il numero di cache miss di S è uguale ad m . Nel trasformare S' in uno scheduling ridotto il numero di inserimenti totale

1. non aumenta per cui risulta minore o uguale di m .
2. diventa uguale al numero totale di cache miss.

Ne consegue che il numero totale di cache miss di S' è minore o uguale di m ed è quindi minore o uguale del numero di cache miss di S .

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 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 .

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 le 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 $j=1,2,\dots,n$ 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

Il problema del caching nella realtà

- Il problema del caching è tra i problemi più importanti in informatica.
- Nella realtà le richieste 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