Bloom filter
Progetto d’esame

Matteo Ferrara

Dipartimento di Informatica - Scienza e Ingegneria

matteo.ferrara@unibo.it

Strutture dati probabilistiche (1)

Definizione

Le strutture dati probabilistiche sono progettate per memorizzare ed elaborare grandi insiemi di dati **preservando l'occupazione di memoria** e **il costo computazionale delle operazioni connesse**.

Tuttavia, le procedure collegate a queste strutture dati **non sono deterministiche** e possono portare a **errori**.

Strutture dati probabilistiche (1)

- Le strutture dati probabilistiche sono **soggette a errori**, che possono tuttavia essere controllati mediante alcuni parametri di costruzione.
- In caso tali errori non possano essere tollerati, è necessario avvalersi di **strutture dati tradizionali** (deterministiche).
- Le strutture dati probabilistiche **non memorizzano l'elemento vero e proprio**. Memorizzano invece un set minimo di informazioni utili ai fini di poter servire interrogazioni specifiche (es. conteggiare gli elementi di un set, determinare l'appartenenza di un elemento ad un set).
- Sono particolarmente adatte per trattare problemi dove la **velocità d'esecuzione** e il **contenimento della memoria impiegata** sono più importanti dell'accuratezza.

Bloom filter (1)

Definizione informale

Definizione informale

Dato un insieme $H=\{h_1, h_2, \ldots, h_k\}$ di k funzioni hash aventi per codominio l'insieme $\{1, \ldots, m\}$ e un insieme di elementi $S = \{a_1, \ldots, a_n\}$, il Bloom filter costruito sull'insieme S è rappresentato da un **array di *m* bit** $b[]$ tale che:

$$b[i] = \begin{cases} 1 & \text{se } \exists\, h \in H, a \in S \mid h(a) = i \\ 0 & \text{altrimenti} \end{cases}$$

È necessario che le k funzioni hash generino output diversi dato lo stesso input. Per fare ciò si può procedere:

- selezionando k funzioni hash diverse;
- selezionando la stessa funzione hash inizializzata con k diversi *seed*.

Un Bloom filter serve per rispondere efficientemente alla domanda:

Membership query

Dato un generico elemento *a* e un insieme *S*, *a* appartiene ad *S*?

Bloom filter (2)

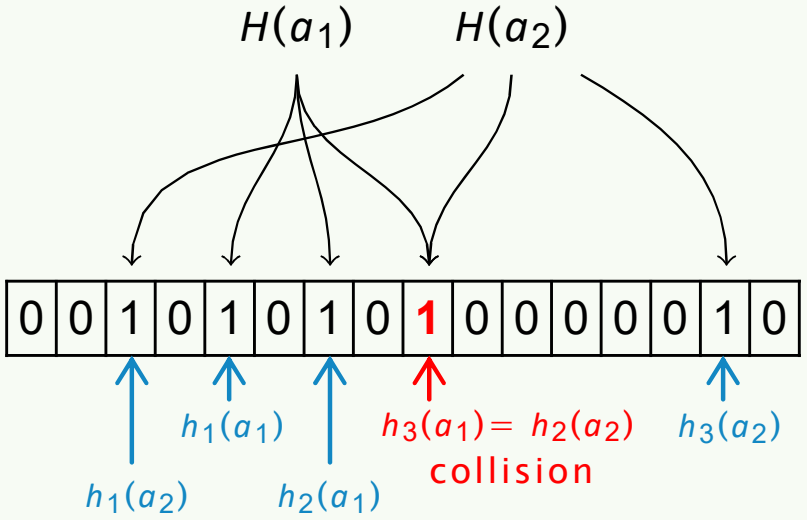
Esempio

PARAMETERS: $k = 3 \quad m = 16$

HASH: $H = \{h_1, h_2, h_3\}$

SETS: $S = \{a_1, a_2\}$

INSERTION



Inserimento

BF-Insertion $O(1)$

INPUT: $\delta \in E$, b, $H = \{h_1, h_2, \ldots, h_k\}$ OUTPUT: b

1. $i \leftarrow 1$;
2. **while** $i \leq k$ **do**
3. $\quad b[h_i(\delta)] \leftarrow 1$;
4. $\quad i \leftarrow i + 1$;

Assumendo che ogni *funzione hash* sia calcolabile in $O(1)$, è evidente che la procedura ha complessità $O(k)$.

Dato che il numero k di funzioni hash utilizzate è tipicamente compreso nell'intervallo $5<k<15$, possiamo asserire che BF-Insertion ha complessità $O(1)$, ovvero può essere calcolata in tempo costante.

Costruzione del filtro

BF-Construction $O(n)$

INPUT: $S \subset E, b, H = \{h_1, h_2, \ldots, h_k\}$ OUTPUT: b

1. **for each** $\delta \in S$ **do**
2. **BF-Insertion**(δ, b, H)

Dato che la procedura BF-Insertion è eseguita in tempo costante, la complessità di questa procedura di costruzione è dominata dal numero di elementi da inserire nel filtro, cioè dalla cardinalità dell'insieme S.

Se $|S| = n$, possiamo asserire che BF-Construction ha complessità $O(n)$.

Nota bene: **i Bloom filter non supportano operazioni di cancellazione**!

Ricerca

BF-Search $O(1)$

INPUT: $\delta \in E$, b, $H = \{h_1, h_2, \ldots, h_k\}$ OUTPUT: $q \in \{\text{TRUE}, \text{FALSE}\}$

```
1. i ← 1;
2. while i ≤ k do
3.     if b[h_i(δ)] = 0 then
4.         return FALSE;
5.     i ← i + 1;
6. return TRUE;
```

Analogamente alla procedura BF-Insertion, la procedura BF-Search ha complessità $O(1)$.

Tassonomia degli output (1)

Consideriamo un Bloom filter *b* costruito su un insieme di dati *S*:

True positive

Dato un elemento $\delta \in S$, BF-Search(δ) = TRUE

True negative

Dato un elemento $\delta \notin S$, BF-Search(δ) = FALSE

False positive

Dato un elemento $\delta \notin S$, BF-Search(δ) = TRUE

I **false negative** non sono possibili.

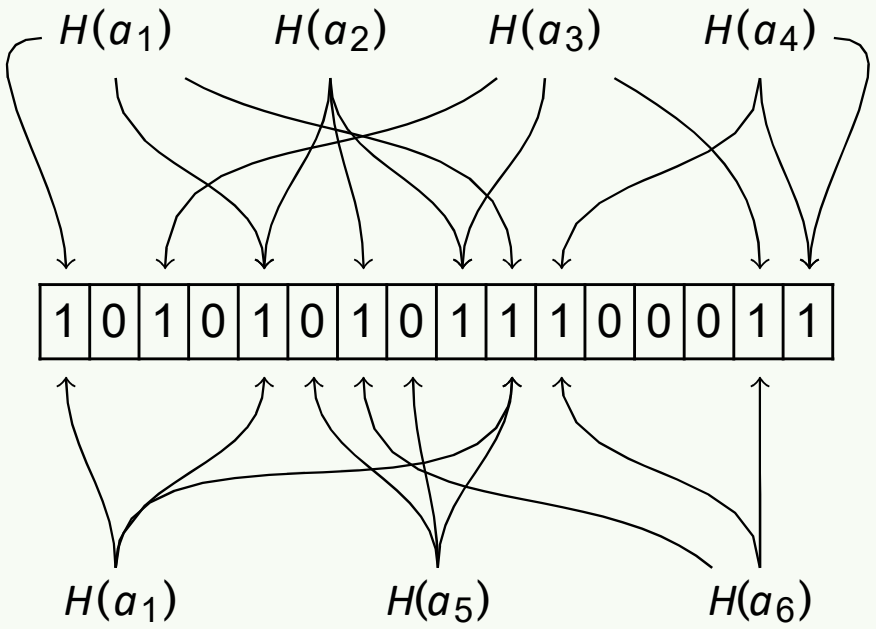
Tassonomia degli output (2)

Esempio

PARAMETERS: $k = 3$ $m = 16$

HASH: $H = \{h_1, h_2, h_3\}$

SETS: $S = \{a_1, a_2, a_3, a_4\}$ $a_5, a_6 \notin S$



True positive True negative False positive

Proprietà in sintesi

- Rappresentano un **singolo insieme** composto da un numero di elementi arbitrario (n);
- usano k funzioni hash che restituiscono valori in $\{1, \ldots, m\}$;
- risolvono efficientemente il problema di **membership query**;
- sono implementati con un **array di m bit**;
- sono soggetti a **falsi positivi**;
- non sono soggetti a **falsi negativi**;
- La **cancellazione** non è possibile.

Il tasso di falsi positivi

Consideriamo un Bloom filter *b* costruito su un insieme di dati *S*. Supponiamo che l'insieme di costruzione *S* contenga n elementi. Dopo aver costruito il filtro, consideriamo un insieme S', disgiunto da *S* ($S \cap S' = \emptyset$), contenente e elementi.

Controlliamo, per ciascun elemento dell'insieme S', l'appartenenza dell'elemento al filtro.

Sia *fp* il numero di falsi positivi riscontrati, ovvero il numero di volte che, durante questo processo, la procedura **BF-Search** risponde erroneamente TRUE.

Tasso di falsi positivi

Il **tasso di falsi positivi**, spesso denominato FPR (da *False Positives Rate*), è dato da:

$$\frac{fp}{e}$$

Progetto d'esame

Implementazione di un Bloom filter

Il progetto consiste nell'**implementazione di un Bloom filter**.
Il filtro dovrà essere riempito con gli elementi contenuti nel dataset fornito e dovrà essere testato con lo stesso dataset (self-check) e con un secondo dataset di verifica.
I risultati ottenuti dovranno essere salvati su un file di testo.
La correttezza dei risultati è verificabile mediante un file fornito nel materiale di supporto.

Il progetto è costituito da 5 file:

- **hash.h**
- **bloomfilter.h**
- **main.c**
- **hash.c**
- **bloomfilter.c**

I dataset forniti

Il materiale di supporto al progetto include due dataset:

- **dataset1.txt**: contiene 65280 elementi che devono essere inseriti nel filtro.
- **non-elements.txt**: contiene 400000 elementi, distinti da quelli di *dataset1*, che devono essere usati per testare il rate di falsi positivi a seguito della costruzione del filtro.

Nota: in fase di valutazione, il software sarà testato anche con un secondo dataset, non fornito tra il materiale di supporto.

Il formato del file di output

Il programma deve memorizzare i risultati prodotti nel file *output.txt*.

Se l'implementazione è corretta il file ottenuto dovrà **coincidere** con il file *OUTPUT_CORRETTO1.txt* (fornito tra il materiale di supporto), che è formattato come segue:

OUTPUT_CORRETTO1.txt

```
Elementi del dataset di costruzione: 65280
Elementi del dataset di verifica: 400000  True
positives (self-check): 65280
True negatives: 399802
False positives: 198
FPR: 0.000495
```

La comparazione fra il file ottenuto e quello fornito viene effettuata dalla funzione *filediff*, contenuta in *main.c*.

Se i due file coincidono, il programma stampa a video la dicitura:

"*Verifica dei risultati eseguita con successo.*"

Altrimenti stamperà:

"*Verifica dei risultati fallita.*"

Consegna del progetto

Si può procedere alla consegna del progetto solo quando l'applicativo stampa a video la dicitura "*Verifica dei risultati eseguita con successo.*".

Se l'applicativo produce al contrario la dicitura "*Verifica dei risultati fallita.*", è necessario correggere il file *bloomfilter.c* fino ad ottenere risultati identici a quelli forniti nel file *OUTPUT_CORRETTO1.txt*.

Per la consegna:

Procedura di consegna

- creare una cartella denominata con numero di matricola (es. 123456)
- includere nella cartella il **solo** file *bloomfilter.c*
- comprimere la cartella in formato *.zip* (es. 123456.zip)
- accedere al sito del corso su IoL *https://iol.unibo.it/course/view.php?id=48569*
- caricare il file .zip nella sezione *Consegna progetto*

La valutazione (Non superato/Superato) sarà pubblicata sul medesimo sito, in tempo utile per il successivo esame scritto.

I file forniti

Oltre a *datset1.txt*, *non-elements.txt* e *OUTPUT_CORRETTO1.txt*, il materiale di supporto include i seguenti file:

- **hash.h** - contiene la dichiarazione della funzione per il calcolo dell'hash digest;
- **bloomfilter.h** - contiene le dichiarazioni della struttura per memorizzare il bloom filter (BloomFilter) e di tutte le funzioni che devono essere implementate in *bloomfilter.c*;
- **main.c** - apre, legge e scrive tutti i file necessari. Gli elementi dei dataset letti vengono memorizzati in appositi array di appoggio. Richiama le funzioni che effettuano le operazioni sul bloom filter e producono i risultati richiesti. Contiene anche la funzione di verifica del risultato;
- **hash.c** - implementa la funzione dichiarata in hash.h che utilizza Murmur hash (esercitazione 4);
- **bloomfilter.c** - implementa le funzioni dichiarate in bloomfilter.h.

N.B.: i file forniti non devono essere modificati in alcun modo. Deve essere modificato solo file bloomfilter.c.

Le funzioni da implementare

Nel file *bloomfilter.c*, vanno implementate le seguenti funzioni (dichiarate in *bloomfilter.h*):

- **createBloomFilter** – inizializzazione della struttura BloomFilter e allocazione della memoria necessaria a contenere il filtro;
- **freeBloomFilter** - libera la memoria allocata per il filtro;
- **bfInsertion** - inserisce un elemento nel filtro (pseudocodice BF-Insertion);
- **bfConstruction** - inserisce nel filtro tutti gli elementi contenuti nel dataset di costruzione (pseudocodice BF-Construction);
- **bfSearch** - verifica se un elemento appartiene al filtro (BF-Search). Ritorna TRUE se l'elemento appartiene al filtro, FALSE altrimenti;
- **countDatasetMembership** - controlla l'appartenenza al filtro di tutti gli elementi del dataset di input (utilizzando *bfSearch*) restituendo il numero di elementi trovati.

N.B.: è necessario corredare il codice sorgente con commenti che ne descrivano nel dettaglio il comportamento.

Note all'implementazione

Nelle funzioni *bfInsertion* e *bfSearch* è necessario calcolare *k* funzioni hash (nel nostro caso, *k* = 10).

Per calcolare *k* funzioni hash diverse, utilizzeremo la funzione *hashFunction* con *k seed* diversi. I *seed* delle *k* funzioni hash dovranno essere memorizzati all'interno del campo *hashSeeds* della struct *BloomFilter*.

Per replicare i risultati di *OUTPUT_CORRETTO1.txt*, utilizzare i seguenti *seed*: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 (nello stesso ordine).

Il calcolo della funzione hash va eseguito richiamando la funzione:

```
unsigned int hashFunction(const char* key, int keyLen, int seed, int m)
```

Nota: in questa implementazione, il Bloom filter è memorizzato come un array di *bool* (campo *filter* della struct *BloomFilter*). Questo metodo impiega comunque un Byte per ogni bool memorizzato. Quindi, di fatto, il filtro è implementato come **array di Byte** e non come **array di bit**, con conseguente spreco di memoria (8 volte più grande del necessario). Un'implementazione del filtro come array di bit sarebbe d'altro canto più complessa (utilizzo di operazioni di mascheratura e shift).