Appunti di Algoritmica 2

Nicola Corti - 454413
Corso di Laurea Magistrale in Informatica - Università di pisa

28 Novembre 2012

Indice

I Randomization, hashing and data streaming **5**

1 Data Streaming Model **7**

1.1 Nozioni di probabilità 7

1.1.1 K-Wise Indipendence 8

1.1.2 Markov's Inequality 9

1.1.3 Chernoff's Bounds 10

2 Count-min Sketch **11**

2.1 Algoritmo 11

2.2 Dimostrazione 12

2.3 Space Lower Bound 15

2.3.1 Communication Complexity 15

2.3.2 L'*INDEX problem* 15

2.3.3 Dimostrazione del lower bound 16

3 Esercizi sul Count-min sketch **19**

3.1 Rimozione dell'invariante 19

3.2 Inner Product 20

3.3 Range Query 21

4 Cuckoo Hashing **23**

4.1 Descrizione 23

4.2 Analisi 25

4.3 Cicli 26

4.4 Re-hashing 26

4.5 Cancellazione 27

5 Filtri di Bloom **29**

5.1 Definizione matematica 29

5.2 Analisi 30

5.2.1 Scelta dei parametri 31

5.2.2 Rimozione di un elemento 31

5.3 Operazioni Booleane 32

5.4 Compressione 32

6 Algoritmi Randomizzati **33**

6.1 Il quicksort 33

6.2 Le skiplist 35

6.2.1 Ricerca nelle *skiplist* 35

6.2.2 Cancellazione nelle *skiplist* 36

6.3 I Random Binary Search Trees 36

6.3.1 Inserimento in un RBST 36

7 Compressione Dati **39**

7.1 Entropia 39

7.2 Complessità di Kolmogorov 40

7.3 Elias Coding 40

7.3.1 Il $\delta - code$ 40

7.3.2 Il $\gamma - code$ 40

7.4 Codifica di Hufmann 41

7.4.1 Descrizione della codifica 41

7.5 Codifica LZ77/LZ78 42

7.5.1 LZ77 42

7.5.2 LZ78 42

II Hard Problems **43**

8 R-Approssimazione **45**

8.0.3 Problema del Commesso Viaggiatore 46

Parte I

Randomization, hashing and data streaming

Capitolo 1

Data Streaming Model

Nel modello del data streaming dobbiamo analizzare grosse quantità di dati e non ci possiamo permettere di memorizzarli, ma dobbiamo memorizzarli in spazio polilogaritmico ed effettuare una singola scansione dello stream.

Questo modello ha subito una notevole evoluzione negli ultimi anni, specialmente perchè presenta molte applicazioni concrete:

1. Analisi del traffico in un router
2. Database e log delle query effettuate
3. Gestione delle telefonate

In questo contesto, semplici problemi possono diventare molto difficili, inoltre solo alcuni problemi possono essere risolti facilmente in modo deterministico, ecco alcuni esempi:

Esempio 1 Trovare il massimo di uno stream di dati, sono necessari solamente $O(\log m)$ bit per memorizzare il massimo attuale.

Esempio 2 Trovare l'elemento mancante in uno stream che rappresenta una permutazione da 1 a n, dove un elemento è mancante: utilizzando la formula

$$\sum_{e \in [1 \cdots n]} e = \frac{n(n+1)}{2}$$

ed effettuare delle sottrazioni per trovare il numero mancante, ma occuperemo $2 \log m$ bits.

Possiamo invece utilizzare l'operazione di XOR ($\oplus$) ed utilizzare $\log m$ bit.

1.1 Nozioni di probabilità

Prima di procedere nell'analisi di metodologie per affrontare il modello del data streaming sono necessarie alcune nozioni di probabilità:

1. K-Wise Indipendence
2. Markov's Inequality
3. Chernoff's Bounds

1.1.1 K-Wise Indipendence

Partiamo dal concetto del K-Wise indipendence per delle variabili aleatorie e definiamo dapprima il K-Wise limited indipendence:

K-Wise Limited Indipendence

Siano $X_1, X_2, \cdots, X_n$ variabili aleatorie e definiamo con $S_i = \text{support}(X_i)$ come il loro insieme di supporto, ovvere come l'insieme di valori che possono assumere le varie X_i.

Tali variabili aleatorie si dicono K-Wise indipendent se $\forall$ scelta di $i_1 < i_2 < \cdots < i_k \in [n]$ e $a_{i_j} \in S_{i_j}$ succede che:

$$\Pr[X_{i_1} = a_{i_1} \wedge X_{i_2} = a_{i_2} \wedge \cdots \wedge X_{i_k} = a_{i_k}] = \Pr[X_{i_1} = a_{i_1}] \times \Pr[X_{i_2} = a_{i_2}] \times \cdots \times \Pr[X_{i_k} = a_{i_k}]$$

Ne segue facilmente che

$$E[X_{i_1}, X_{i_2}, \cdots, X_{i_k}] = E[X_{i_1}] \times E[X_{i_2}] \times \cdots \times E[X_{i_k}]$$

Possiamo adesso estendere il concetto ad una famiglia di funzioni hash:

Hash K-wise Indipendence

Definiamo come $\{h\}_{h \in H}$ una famiglia di funzioni hash $[n] \to [b]$ che supponiamo essere uniformamente distribuita, ovvero che

$$\Pr[h \in H] = \frac{1}{|H|}$$

Diremo che la famiglia H di funzioni hash è K-wise indipendente se succede che per ogni scelta di $x_1 < x_2 < \cdots < x_k \in [n], b_1 < b_2 < \cdots < b_k \in [b]$ succede che

$$\Pr_{h \in H}[h(x_1) = b_1 \wedge \cdots \wedge h(x_k) = b_k] = \Pr_{h \in H}[h(x_1) = b_1] \times \cdots \times \Pr_{h \in H}[h(x_k) = b_k]$$

Esempio Consideriamo una famiglia di funzioni hash della seguente forma:

$$h(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{k-1} x^{k-1}$$

$$a_i \in F \text{finite field}$$

Notiamo che una funzione risulta facilmente identificabile memorizzando i coefficienti $(a_0, a_1, a_2, \cdots, a_n)$ per cui possiamo cacolare facilmente la cardinalità della famiglia di funzioni hash come $|H| = |F|^K$

Notiamo allora che ci sono $|F|^{k-1}$ funzioni hash che risolvono l'equazione $h(x_i) = b_i$ del tipo

$$a_0 = b_i - a_1 x_i - a_2 x_i^2 - \ldots - a_k x_i^k$$

Succede dunque che:

$$\underbrace{\Pr_{h\in H}[h(x_1) = b_1 \wedge \cdots \wedge h(x_k) = b_k]}_{\frac{1}{|F|^K}} = \underbrace{\Pr_{h\in H}[h(x_1) = b_1]}_{\frac{|F|^{K-1}}{|F|^K} = \frac{1}{|F|}} \times \cdots \times \Pr_{h\in H}[h(x_k) = b_k]$$

1.1.2 Markov's Inequality

Sia X una variabile aleatoria ≥ 0 e sia $a \in \mathbb{R}, a > 0$ allora vale:

$$\Pr[X \geq a] \leq \frac{\mathbb{E}[X]}{a}$$

Dimostrazione. In questa dimostrazione useremo le variabili indicatrici:

$$I = \begin{cases} 0 & \text{se } X < a \\ 1 & \text{se } X \geq a \end{cases}$$

Proviamo a calcolare il valore atteso della variabile indicatrice:

$$\begin{aligned} \mathbb{E}[I] &= 0 \cdot \Pr[I = 0] + 1 \cdot \Pr[I = 1] \\ &= 1 \cdot \Pr[I = 1] \\ &= \Pr[X \geq a] \end{aligned} \tag{1.1}$$

Adesso notiamo che $aI \leq X$:

- $X < a \Rightarrow I = 0$
- $X \geq a \Rightarrow aI = a, a \leq X$

Adesso utilizzando i due fatti appena esaminati:

$$\begin{aligned} \mathbb{E}[X] &\geq \mathbb{E}[aI] \\ &= a \cdot \mathbb{E}[I] \\ &= a \cdot \Pr[X \geq a] \end{aligned} \tag{1.2}$$

Dividiamo per a che ricordiamo essere reale positivo ed abbiamo dimostrato la disuguglianza di Markov.

□

1.1.3 Chernoff's Bounds

Siano $Y_1, Y_2, \ldots, Y_r$ variabili aleatorie indipendenti ed equamente distribuite tali che

$$\Pr[Y_j = 1] = p \quad \Pr[Y_j = 0 = 1 - p$$

Definiamo inoltre:

$$Y = \sum_{j=1}^{r} Y_j \quad \mu = \mathbb{E}[Y] = \sum_{j=1}^{r} [Y_j] = rp$$

Allora possiamo effettuare la seguente stima anche detta Chernoff's Bound

$$\forall \lambda > 0 \quad \Pr[Y \geq (1+\lambda)\mu] < \left(\frac{e^{\lambda}}{(1+\lambda)^{1+\lambda}} \right)^{\mu}$$

Capitolo 2

Count-min Sketch

Descriviamo adesso il modello del count-min sketch (presentato da G. Cormode & S. Muthukrishnan) che ci permette di approssimare le frequenze degli elementi in uno stream.

Supponiamo di n elementi distinti che possono occorrere nello stream, noi vorremmo calcolare $F[i]$, ovvero il numero di volte che un elemento i occorre nello stream.

Le operazioni di base a nostra disposizione sono $F[i]++$ e $F[i]--$ ed andremo a calcolare $\tilde{F}[i]$ ovvero una versione approssimata di $F[i]$. Ricordiamoci che abbiamo a disposizione soltanto spazio $O(\log n)$ e non possiamo permetterci di memorizzare $F[i]$ per esteso. Inoltre operiamo sotto il seguente invariante:

Invariante: $F[i] \geq 0$

Ci accontenteremo di calcolare $\tilde{F}[i]$ tale che

$$\forall i \; F[i] \leq \underbrace{\tilde{F}[i] \leq F[i] + \varepsilon ||F||}_{\text{con prob.} \leq 1-\delta}$$

Dove $\varepsilon \geq 1$ rappresenta il parametro di approssimazione, ovvero quanto vogliamo approssimare il risultato e $\delta \geq 0$ rappresenta il parametro di errore, indica di quanto stiamo sbagliando il risultato.

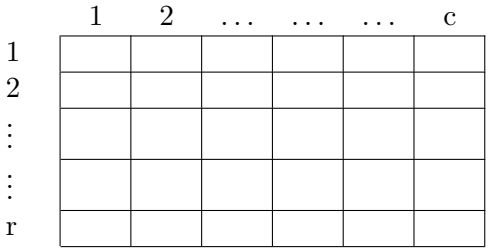
Infine indichiamo con $||F||$ la norma del vettore F e la calcoliamo come

$$||F|| = \sum_{x \in [n]} F[x]$$

2.1 Algoritmo

Il seguente è l'algoritmo per inizializzare e utilizzare il count-min sketch:

1. Definiamo $r = \log_2 \frac{1}{\delta}$ e $c = \frac{e}{\varepsilon}$, che saranno le dimensioni del nostro count-min sketch (e è la costante di eulero).
2. Definiamo T come un tabella di dimensioni $r \times c$ formata da contatori inizialmente settati a 0:



3. Si scelga una famiglia H di funzioni hash che siano 2-wise independent. Ad esempio potremmo scegliere come famiglia la seguente

$$h \in H \Leftrightarrow h(x) = (((ax + b) mod\ p) mod\ c) + 1$$

4. Si scelgano r funzioni hash $h_1, h_2, \ldots, h_r$ in modo uniforme ed indipendente dalla famiglia di funzioni hash H.

5. Preso un elemento i, associamo ad essi r contatori, in modo che per ogni riga vi sia un elemento associato ad i, li scegliamo nel modo seguente:

$$T(1, h_1(i)), T(2, h_2(i)), \ldots, T(r, h_r(i))$$

.

Ovviamente succede che per due elementi $i \neq i'$ ci siano delle collisioni.

6. Eseguiamo le operazioni del modo seguente:

   - $F[i]{+}{+} \Rightarrow$ incrementiamo di uno tutte le caselle associate all'elemento i, ovvero effettuaimo la seguente operazione $T(j, h_j(i)) + +\forall j = 1, 2, \ldots, n$.
   - $F[i]{-}{-} \Rightarrow$ decrementiamo di uno tutte le caselle associate all'elemento i, ovvero effettuaimo la seguente operazione $T(j, h_j(i)) + +\forall j = 1, 2, \ldots, n$.

7. Restituiamo l'approssimazione di $F[i]$ come il minimo valore assunto dalle caselle associate:

$$\tilde{F}[i] = \min_{1 \leq j \leq r} T(j, h_j(i))$$

2.2 Dimostrazione

Procediamo adesso a dimostrare le proprietà del count-min sketch

Fatto 1 Lo spazio occupato è $O(r \cdot c) = O(\varepsilon^{-1} \log \delta^{-1})$ parole di memoria, dove ogni parola di memoria può memorizzare $||F||$

Dimostrazione. Ogni contatore contiene un intero appartenente ad $[||F||]$ □

Fatto 2 $F[i] \leq \tilde{F}[i]$

Dimostrazione. Per dimostrare questo fatto dobbiamo prima introdurre una variabile indicatrice che ci indica se c'è stata una collisione per una specifica riga, dati 2 elementi:

$$I_{j,i,k} = \begin{cases} 1 & \text{se } k \neq i \wedge h_j(i) = h_j(k) \\ 0 & \text{altrimenti} \end{cases}$$

Definiamo supponiamo che abbiamo preso come approssimazione la $j-esima$ colonna:

$$T(j, h_j(i)) = \tilde{F}[i]$$

Per il modo in cui abbiamo costruito il nostro count-min sketch allora $\exists i_1, i_2, \ldots, i_f$ elementi dello stream tali che uno di loro è i e che hanno contribuito tutti ad incrementare il valore della casella considerata:

$$T(j, h_j(i)) = \sum_{l=1}^{f} F[i_l] = F[i] + X_{ji}$$

Dove X_{ji} rappresenta la "sporcizia", ovvero l'eccesso dovuto alla presenza di collisioni sui contatori del count-min sketch. Per come l'abbiamo costruito vale che $X_{ji} \geq 0$, per cui il fatto risulta banalmente dimostrato.

□

Osservazione 1 Osserviamo facilmente che possiamo espriemere la "sporcizia" nel modo seguente:

$$X_{ji} = \sum_{k=1}^{n} I_{j,i,k} F[k]$$

dato che X_{ij} vale 1 solamente quando c'è stata una collisione fra il nostro elemento i e un altro elemento dello stream k.

Osservazione 2 Possiamo dimostrare che:

$$\mathbb{E}[I_{j,i,k}] = \frac{\varepsilon}{e}$$

Dimostrazione.

$$\begin{aligned}
\mathbb{E}[I_{j,i,k}] &= 0 \cdot \Pr[I_{j,i,k} = 0] + 1 \cdot \Pr[I_{j,i,k} = 1] \\
&= \Pr[I_{j,i,k} = 1] \\
&= \Pr\left[\exists a \in [c] : k \neq i \wedge h_j(i) = a \wedge h_j(k) = a\right] \\
&= \Pr\left[\bigcup a \in [c] : k \neq i \wedge h_j(i) = a \wedge h_j(k) = a\right] \\
&= \sum_{a \in [c]} \Pr\left[k \neq i \wedge h_j(i) = a \wedge h_j(k) = a\right] \\
&= \sum_{a \in [c]} \underbrace{\Pr[h_j(i) = a]}_{1/c} \times \underbrace{\Pr\left[k \neq i \wedge h_j(k) = a\right]}_{\text{“circa”}\, 1/c} \\
&= \sum_{a \in [c]} \frac{1}{c^2} = \frac{1}{c} = \frac{\varepsilon}{e}
\end{aligned}$$

□

Notiamo come nel penultimo passaggio si sia usata la probabilità che h_j è 2-Wise indipendent. Inoltre notiamo come il secondo termine sia circa $1/c$ dato che con la prima disuguaglianza escludiamo uno dei valori del dominio.

Fatto 3 $\tilde{F}[i] \leq F[i] + \varepsilon||F||$ con probabilità $\geq 1 - \delta$.

Dimostrazione. Al solito poniamo $\tilde{F}[i] = F[i] + X_{ij}$ dove $X_{ij} \geq 0$ è la solita “sporcizia”.

Calcoliamo dapprima il valore atteso della sporcizia:

$$\mathbb{E}[X_{ij}] = \mathbb{E}\left[\sum_{k=1}^{n} I_{j,i,k} F[k]\right] = \sum_{k=1}^{n} \mathbb{E}\left[I_{j,i,k} F[k]\right] \leq \sum_{k=1}^{n} \left(F[k] \cdot \underbrace{\mathbb{E}\left[I_{j,i,k}\right]}_{\varepsilon/e}\right) =$$

$$= \sum_{k=1}^{n} (F[k] \cdot \frac{\varepsilon}{e}) = \frac{\varepsilon}{e}||F||$$

Che equivale a dire che $e\mathbb{E}[X_{ij}] = \varepsilon||F||$.

Con queste premesse calcoliamo:

$$\begin{aligned}
\Pr[\tilde{F}[i] > F[i] + \varepsilon||F||] &\leq \delta \\
\Pr[\forall j\ F[i] + X_{ij} > F[i] + \varepsilon||F||] &\leq \delta \\
\Pr[\forall j\ X_{ij} > \varepsilon||F||] &\leq \delta \\
\Pr[\forall j\ X_{ij} > e\mathbb{E}[X_{ji}] &\leq \delta \\
\Pr[X_{1i} > e\mathbb{E}[X_{1i}]] \times \ldots \times \Pr[X_{ri} > e\mathbb{E}[X_{ri}]] &\leq \delta
\end{aligned}$$

Notiamo che l'ultimo passaggio si basa sul fatto che le $h_1, \ldots h_r$ funzioni hash sono scelte in modo indipendente e uniforme. Adesso approssimiamo un singolo termine con la disuguaglianza di Markov:

$$\Pr[X_{ji} > e\mathbb{E}[X_{ji}]] \leq \frac{\mathbb{E}[X_{ji}]}{e\mathbb{E}[X_{ji}]} = \frac{1}{e} < \frac{1}{2}$$

Detto questo riscriviamo:

$$\Pr[X_{1i} > e\mathbb{E}[X_{1i}]] \times \ldots \times \Pr[X_{ri} > e\mathbb{E}[X_{ri}]] < \left(\frac{1}{2}\right)^r = \left(\frac{1}{2}\right)^{\log_2 \frac{1}{\delta}} =$$

$$= 2^{-1} \log_2 \delta^{-1} = \log_2 (\delta^{-1})^{2^{-1}} = \delta$$

□

2.3 Space Lower Bound

Se consideriamo il problema del count-min sketch, notiamo che possiamo calcolare un limite inferiore per lo spazio necessario: $\Omega(1/\varepsilon)$.

Per verificare questo limite inferiore dobbiamo introdurre alcuni concetti di *communication complexity*

1. Definiamo dapprima l'*INDEX problem*
2. Riduciamo l'*INDEX problem* al calcolo di $\tilde{F}[i]$

2.3.1 Communication Complexity

I problemi di communication complexity si presentano nel modo seguente: siano A (Alice) e B (Bob) due soggetti che devono comunicare fra di loro.

- Alice è in possesso di una sequenza di bit x
- Bob è in possesso di una sequenza di bit y
- L'obiettivo è che uno dei due soggetti riesca a calcolare una funzione $f(x, y)$ utilizzando il minor numero di bit per le comunicazioni possibile.

2.3.2 L'*INDEX problem*

L'*INDEX problem* rappresenta un particolare problema di communication complexity così definito:

- Alice è in possesso di una sequenza di bit $x \in \{0, 1\}^n$
- Bob è in possesso di una sequenza di bit $y \in \{0, 1\}^{\log_2 n}$ ovvero è in possesso di un bit indica una posizione nella sequenza di Alice.
- La funzione da calcolare è $f(x, y) = x_y$ ovvero conoscere il bit in posizione.
- La comunicazione deve essere *one round* da Alice a Bob, deve essere inviato cioè un solo messaggio.
- Bob vuole calcolare $f(x, y)$ con $\Pr > 1/2$.

Notiamo che nel caso in cui la probabilità sia $= 1/2$ allora il problema si risolve banalmente tirando una moneta e considerando il risultato.

Ci chiediamo allora quanti bit deve mandare Alice a Bob: deve mandare m bit dove

$$m \geq n(1 - H(p))$$

Dove $H(p)$ rappresenta l'entropia così definita:

$$H(p) = -p \log_2 p - (1-p) \log_2 (1-p)$$

Notiamo infine che

$$n \underbrace{(1 - H(p))}_{0<\ldots<1} = \Omega(n)\text{bits}$$

Alice deve mandare essenzialmente buona parte del messaggio.

2.3.3 Dimostrazione del lower bound

Per dimostrare il nostro lower bound relativo allo spazio istanziamo l'*INDEX problem* al caso del count-min sketch:

Dimostrazione. Istanziamo un *INDEX problem* e scegliamo ε tale che $n = \frac{1}{\varepsilon}$ ed inoltre $|x| = n$ ovvero n è la dimensione dello stream dell'*INDEX problem*.

Definiamo inoltre

$$F\left[1 \ldots \frac{1}{\varepsilon}\right] : F[i] = \begin{cases} 2 & \text{se } x[i] = 1 \\ 0 & \text{se } x[i] = 0 \end{cases}$$

Notiamo che stiamo amplificando i valori della $F[i]$ in modo da poter distinguere successivamente fra $\tilde{F}[i]$ e $F[i]$.

Restringiamo l'insieme delle possibili stringhe del tipo $F\left[1 \ldots \frac{1}{\varepsilon}\right]$ solamente a quelle tali che $\#2 = \#0 = n/2$ che risultano essere in numero pari a $\sum^{n} \binom{n}{n/2}$; questa restrizione ci servirà per calcolare in modo semplice la norma del vettore F, e vediamo che applicare questa restrizione è lecito dato che la funzione binomiale $\binom{n}{k}$ ha il suo punto di massimo in $k = n/2$.

Così facendo risulta

$$||F|| = 2 \times \frac{n}{2} = \frac{1}{\varepsilon}$$

.

Il count-min sketch ci dice che

$$\begin{aligned} F[i] &\leq \tilde{F[i]} \leq F[i] + \varepsilon ||F|| \\ F[i] &\leq \tilde{F[i]} \leq F[i] + 1 \end{aligned} \tag{2.1}$$

Dato che abbiamo amplificato $F[i]$ che può valere solamente 0 o 2, possiamo risalire al valore di $F[i]$ partendo da $\tilde{F}[i]$, quindi applicando il risultato calcolato per l'*INDEX*

problem possiamo determinare che:

$$\text{space}\left(\tilde{F}\left[1\ldots\frac{1}{\varepsilon}\right]\right) = \Omega(n) = \Omega(1/\varepsilon)$$

□

Capitolo 3

Esercizi sul Count-min sketch

In questo capitolo mostriamo alcuni esercizi e applicazioni del count-min sketch visto in precedenza.

3.1 Rimozione dell'invariante

È possibile rimuovere l'invariante sul count-min sketch che avevamo imposto in precendenza, ovvero:

Invariante: $F[i] \geq 0$

Così facendo dobbiamo però ridefinire $\tilde{F}[i]$ come il mediano fra i valori dei contatori ovvero

$$\tilde{F}[i] = \text{median}_{0 \leq j \leq r}\, T(j, h_j(i))$$

Rimuovendo l'invariante possiamo dimostrare, usando i Chernoff's Bound che vale la seguente relazione:

$$F[i] - 3\varepsilon||F|| \leq \tilde{F}[i] \leq F[i] + 3\varepsilon||F||$$

$$\text{con } p \geq 1 - \delta^{1/4}$$

che riscrivendo $\tilde{F}[i] = F[i] + X_{ij}$ equivale a dire che:

$$-3\varepsilon||F|| \leq X_{ij} \leq +3\varepsilon||F||$$

ovvero

$$|X_{ij}| \leq 3\varepsilon||F||$$

Dimostrazione. Applichiamo dapprima la Markov inequality:

$$\Pr[|X_{ij}| \geq 3\varepsilon||F||] < \frac{\overbrace{\mathbb{E}[|X_{ij}|]}^{\varepsilon/e||F||}}{3\varepsilon||F||} < \frac{1}{3e} < \frac{1}{8}$$

Definiamo adesso una variabile indicatrice che vale 1 se il contatore selezionato non rispetta il vincolo che stiamo cercando di dimostrare:

$$Y_j = \begin{cases} 1 & \text{se } |X_{ij}| > 3\varepsilon||F|| \text{ con } p < 1/8 \\ 0 & \text{altrimenti} \end{cases}$$

Consideriamo adesso il seguente fatto: ci verrà ritornato un mediano che rispetta il nostro vincolo $|X_{ij}| \leq 3\varepsilon||F||$ se, prese r righe ci sono meno di $r/2$ contatori i' per cui vale $|X_{i'j}| > 3\varepsilon||F||$.

Per calcolare dunque la probabilità di errore consideriamo il caso in cui il numero di righe errate è maggiore di $1/2$ che equivale a dire

$$Y \geq \frac{r}{2}$$

Applichiamo adesso i Chernoff's Bound con parametro $\mu = rp$ e $(1+\lambda)\mu = r/2$:

$$\Pr[Y \geq (1+\lambda)\mu] < \left(\frac{e^\lambda}{(1+\lambda)^{1+\lambda}}\right)^\mu = \frac{1}{e^\mu}\left(\frac{e}{1+\lambda}\right)^{(1+\lambda)\mu}$$

Adesso maggioriamo:

$$\frac{1}{e^\mu}\left(\frac{e}{1+\lambda}\right)^{(1+\lambda)\mu} = \frac{1}{e^{rp}}(2ep)^{r/2} \leq \frac{1}{2^{r/4}} = \delta^{1/4}$$

Invertiamo la disequazione:

$$2^{\frac{r}{4}} \leq \underbrace{e^{rp}}_{\geq 1}\frac{1}{(2ep)^{r/2}}$$

Quindi ci basta prendere $\frac{1}{2pe} > \sqrt{2} \Leftrightarrow p < \frac{1}{2\sqrt{2}e}$ □

3.2 Inner Product

Il count-min sketch può essere anche utilizzato per calcolare l'*inner product*, ovvero il prodotto scalare di due vettori. Questo utilizzo può trovare parecchie applicazioni in molti campi dell'informatica, in particolare nel campo dei database, può essere utilizzato per stimare la dimensione di un'operazione di JOIN del tipo $R_a \bowtie R_b$ al fine di effettuare scelte e ottimizzazioni sul possibile piano di esecuzione di una query.

Ricordiamo che dati due vettori $a[i]$ e $b[i]$ il loro prodotto vettoriale è definito come $a \odot b = \sum_{i=1}^{n} a[i]b[i]$. Creiamo dunque due count-min sketch, uno per il vettore a e uno per il vettore b, vediamo che ogni riga del count-min sketch rappresenta una versione "succinta" del vettore associato, possiamo quindi effettuare il prodotto vettoriale fra le varie righe e poi scegliere il minimo fra questi valori:

$$P_j = \sum_{i=1}^{c} T_a(j,i)T_b(j,i)$$

$$\widetilde{(a \odot b)} = \min_{0 \leq j \leq r} P_j$$

Possiamo dimostrare che vale

$$a \odot b \leq \underbrace{\widetilde{a \odot b} \leq a \odot b + \varepsilon ||a|| \cdot ||b||}_{\Pr > 1-\delta}$$

Dimostrazione. Riscriviamo l'*inner product* come

$$\widetilde{(a \odot b_j)} = \sum_{i=1}^{n} a_i b_i + \sum_{p \neq q, h_j(p) = h_j(q)} a_p b_p$$

Dove il secondo termine rappresenta la solita "sporcizia" vista nella dimostrazione del count-min sketch.

Calcoliamo adesso il seguente termine:

$$\mathbb{E}(\widetilde{a \odot b_j} - a \odot b) = \sum_{p \neq q} \Pr[h_j(p) = h_j(q)] a_p b_p \leq \sum_{p \neq q} \frac{\varepsilon a_p b_p}{e} \leq \frac{\varepsilon ||a|| \cdot ||b||}{e}$$

Notiamo che le ultime due approssimazioni sono effettuate utilizzando fatti già noti: il termine $\Pr[h_j(p) = h_j(q)]$ lo approssimiamo con la speranza della variabile indicatrice $I_{x,j,k}$ introdotta nella dimostrazione del count-min sketch; le norme vengono invece introdotte perchè si nota che stiamo sommano su tutti gli elementi dei vettori a meno di elementi di indici uguali.

Adesso possiamo utilizzare il termine appena calcolato nella Markov's inequality:

$$\Pr[\widetilde{a \odot b_j} - a \odot b > \varepsilon ||a|| \cdot ||b||] < \frac{\varepsilon ||a|| \cdot ||b||}{e \varepsilon ||a|| \cdot ||b||} = \frac{1}{e} < \delta$$

□

3.3 Range Query

Nel problema del *Range Query* si vuole fare la somma di una sottosequenza di uno stream F a partire dall'elemento i fino all'elemento j.

Potremmo utilizzare un procedimento simile a quello visto per l'*inner product*, definendo come vettore $a[i]$ il nostro stream in ingresso e come vettore $b[i]$ un vettore che contenga tutti 0 tranne nelle posizioni comprese nell'intervallo $[i, j]$ dove sono presenti degli 1, così si ottiene facilmente la somma della sottosequenza desiderata.

Si presenta però un problema, infatti la disequazione:

$$\widetilde{a \odot b} \leq a \odot b + \varepsilon ||a|| \cdot ||b||$$

può avere il termine a destra arbitrarimanete grande, dato che $||b|| = j - i + 1$.

Per risolverlo possiamo allora dividere l'intervallo utilizzando i *dyadic ranges*. Assumiamo che il nostro intervallo in questione sia lungo $n = 1024$ bit. Possiamo definire $\log_2 n = 10$ insiemi di *dyadic ranges*:

- 1024 intervalli di lunghezza 1,
- 512 intervalli di lunghezza 2,
- 256 intervalli di lunghezza 3,
- ...

Vediamo che un generico punto nell'intervallo appartiene a $\log_2 n$ *dyadic ranges*, uno per ogni y nell'intervallo $[0, \log_2 n]$. Un generico intervallo $[i, j]$ può essere ridotto in al più $2 \log_2 n$ *dyadic ranges*.

Operiamo dunque in questo modo: realizziamo $\log_2 n$ count-min sketch, uno per ogni insieme di *dyadic ranges*, che aggiorniamo ad ogni update di elementi.

Quando si vuole calcolare un range query, si riduce l'intervallo nei $2 \log_2 n$ *dyadic ranges*, si calcolano tante query, una per ogni *dyadic range* al relativo sketch associato (se un *dyadic range* è lungo 2^i allora prendiamo il i-esimo count-min sketch). Calcoliamo infine la somma di tutti i valori e la restituiamo

Definiamo con $a[l, r]$ il risultato effettivo del range query, e con $\tilde{a}[l, r]$ il risultato approssimato con il procedimento visto prima; possiamo dimostrare che con probabilità superiore a $1 - \delta$ vale:

$$\tilde{a}[l, r] \leq a[l, r] + 2\varepsilon \log n ||a||$$

Dimostrazione. Ricordiamo che definiamo $\tilde{a}[l, r] = a[l, r] + X_{l,r}$ dove $X_{l,r,j}$ rappresenta la "sporcizia", che deriva dai fattori additivi di ogni singolo count-min sketch che viene considerato (possono essere al massimo $2 \log n$) quando si esegue una query.

Il valore atteso della "sporcizia" risulta allora:

$$\mathbb{E}(l, r, j) = 2 \log n \frac{\varepsilon}{e} ||a||$$

Notiamo che il termine

$$||a||$$

deriva da un'approssimazione, dato che stiamo facendo la somma su un range di a, lo estendiamo a tutto a e ne calcoliamo dunque la norma.

Adesso possiamo utilizzare la Markov inequality per dimostrare la proprietà:

$$\Pr[\tilde{a}[l, r] - a[l, r] > 2\varepsilon \log n ||a||] \leq \frac{2 \log n \frac{\varepsilon}{e} ||a||}{2\varepsilon \log n ||a||} \leq \frac{1}{e} \leq \delta$$

□

Capitolo 4

Cuckoo Hashing

Il cuckoo hashing è uno schema particolare che ci permette di risolvere in modo efficente il problema del dizionario.

Il dizionario è una struttura dati che contiene un insieme S di n chiavi e che ci permette di effettuare le seguenti operazioni:

1. INS(X) per inserire una nuova chiave.
2. DEL(X) per rimuovere una chiave esistente.
3. SEARCH(X) per cercare se una chiave appartiene ad S

Queste operazioni devono essere effettuare nel modo più efficente possibile. Il cuckoo hashing presenta un costo pari ad $O(1)$ per la ricerca e per l'inserimento risultato quindi molto competitivo.

4.1 Descrizione

Il cuckoo hashing si basa su 2 funzioni hash h_1 e h_2 scelte da una famiglia di funzioni hash $H : S \to [r]$. Le due funzioni vengono scelte, senza perdità di generalità nel modo seguente:

1. h_1 e h_2 vengono scelte in modo indipendente.
2. $h_1(x)$ e $h_2(x)$ possono venire calcolate in tempo $O(1)$ $\forall x$
3. h_1 e h_2 sono equidistribuite, ovvero $\Pr(h(x) = i) = \frac{1}{r}$

L'algoritmo procede in modo simile al comportamento del cuculo (*Cuculus canorus*) che prova ad insediarsi in un luogo, e se lo trova occupato, caccia il vecchio inquilino del luogo.

Si prova ad inserire l'elemento x nella collezione: inizialmente si prova ad inserirlo nella posizione $h_1(x)$, se la si trova occupata, lo si inserisce in $h_2(x)$; se alla posizione $h_2(x)$ era presente un elemento y, allora si itera il problema su y.

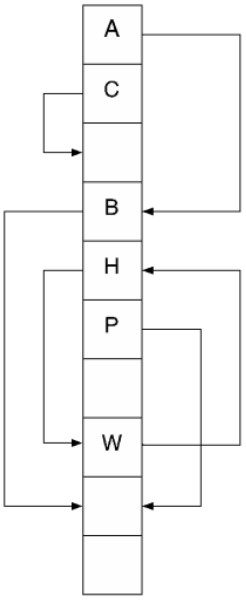


Figura 4.1: Rappresentazione grafica di un cuckoo hashing

Nell'immagine seguente è presente un schema di un cuckoo hashing:

La ricerca può essere realizzata con poche righe di codice:

```
int search(int x){
    if(T[h1(x)]==x) return h1(x);
    if(T[h2(x)]==x) return h2(x);
    return -1;
}
```

Per quanto riguarda invece la procedura di inserimento notiamo che le funzioni h_1 e h_2 vengono utilizzate in modo equivalente, non è rilevante quale utilizzare per prima nel confronto.

```
void insert(int x){
    if(search(x) != -1) return;
    int pos = h1(x);
    for(int i = 0; i < n; i++){
        if(T[pos] == NULL) {T[pos] = x; return}
        swap(x, T[pos]);
        if (pos == h1(x)) pos = h2(x); else pos = h1(x);
    }
    rehash;
    insert(x);
}
```

Notiamo che può succedere che non riusciamo ad inserire un elemento all'interno del dizionario perché si è venuto a creare un ciclo. Per renderci conto che ci troviamo in questa situazione effettuiamo il ciclo di inserimento per n volte, nel caso in cui non siamo riusciti a sistemare tutti gli elementi, effettuato un rehash(), ovvero ricreiamo tutta la struttura.

4.2 Analisi

Per effettuare l'analisi del cuckoo hashing dobbiamo prima partire dal seguente Lemma:

Lemma *Prese due posizioni i e j ed una costante $c > 1$, se vale che $r \geq 2cn$ (ovvero che il nostro cuckoo hashing di fatto è un grafo molto sparso) allora vale che*

$$\Pr(\exists \textit{un cammino di lunghezza } l) \leq \frac{1}{c^l r}$$

Ricordiamo che se immaginiamo il cuckoo hashing come un grafo, r rappresenta il numero dei nodi e n il numero degli archi.

Dimostrazione. La dimostrazione procede per induzione su l.

Per il caso $l = 1$, se considero due posizioni i e j la probabilità che esista un cammino fra queste due dipende strettamente dalla probabilità che due elementi abbiamo i e j come possibili elementi (equivale a scegliere 2 posizioni su r e si ripete l'operazione per due elementi).

Succede dunque che la probabilità per due elementi risulta essere $\frac{2}{r^2}$, se invece considero tutti i possibilit elementi la probabilità risulta

$$\sum_{x \in S} \frac{2}{r^2} \leq \frac{2n}{r^2} \leq \frac{c^{-1}}{r}$$

Ricordiamoci che avevamo imposto che:

$$r \geq 2cn \quad \text{ovvero} \quad \frac{r}{c} \geq 2n$$

Per il caso induttivo, assumiamo che esista:

1. Un cammino di lunghezza $l - 1$ tra i un elemento k
2. Un collegamento fra k e j.

Il primo punto può accadere, per ipotesi induttiva, con probabilità $\frac{1}{c^{(l-1)}r}$ mentre il secondo può venire con probabilità $\frac{1}{cr}$ per il caso base visto prima.

Se ne deduce che la probabilità che esista un cammino di lunghezza l fra due elementi risulta essere

$$\frac{1}{c^l r^2} \leq \frac{1}{c^l r}$$

□

Di fatto un *bucket* in un cuckoo hashing è un cammino fra due nodi. Abbiamo dimostrato che due elementi finiscono nello stesso *bucket* con probabilità $O(\frac{1}{r})$ e al contempo che decresce esponenzialmente al crescere di l.

Possiamo dare un limite superiore alle operazioni necessarie per un generico elemento x, considerando la dimensione del bucket di x. Vediamo che la probabilità di effettuare operazioni su x risulta essere $O(1/r)$, sommiamo per tutti gli elementi in S e vale che:

$$|S| \cdot O(1/r) = O(1) \quad \text{dato che} \quad r \geq n \geq |S|$$

4.3 Cicli

Analizziamo adesso qual è la probabilità che esista un ciclo fra due generici elementi.

Dato S consideriamo una sequenza di εn inserimenti, con $\varepsilon > 0$ e piccolo, vediamo che possiamo considerare la probabilità di avere un ciclo su k con la probabilità di avere un cammino da k a k (di lunghezza non fissata):

$$\Pr(\text{ciclo su k}) \leq \Pr(\exists\text{un cammino fra k e k}) = \sum_{l=0}^{n-1}(Lemma1) \leq \left(\sum_{l} \frac{1}{c^l r}\right) = O\left(\frac{1}{r}\right) \leq \frac{1}{(c-1)r}$$

Notare che prendiamo $r \geq 2c(1+\varepsilon)n$.

Per calcolare la probabilità effettiva che ci sia un ciclo, consideriamo tutti i vertici:

$$r \cdot \sum_{l} \frac{1}{c^l r} = r \cdot \frac{1}{(c-1)r} \leq \frac{1}{c-1} \leq \frac{1}{2}$$

Abbiamo utilizzato lo sviluppo della serie x^l per arrivare a questo risultato. Qui potremmo utilizzare i chernoff bound per limitare questo risultato a piacimento.

4.4 Re-hashing

La probabilità di fare un rehashing può essere maggiorato dalla probabilità di avere un ciclo nel grafo.

Supponiamo ad esempio di scegliere la costante $c = 3$:

1. Con probabilità $1/2$ faccio 1 rehashing,
2. Con probabilità $1/4$ faccio 2 rehashing,
3. Con probabilità $1/8$ faccio 3 rehashing,
4. etc...
5. Con probabilità $1/2^i$ faccio i rehashing,

Cerchiamo adesso di determinare il numero medio di rehashing che eseguo:

$$\sum_{i=1}^{\infty} \frac{1}{2^i} \leq 2$$

.

La maggiorazione risulta dallo schema seguente:

$$
\begin{array}{ccccc}
 & & & \ldots & \longleftarrow \leq \ldots \\
 & & & 1/16 & \longleftarrow \leq 1/8 \\
 & & 1/8 & 1/16 & \longleftarrow \leq 1/4 \\
 & 1/4 & 1/8 & 1/16 & \longleftarrow \leq 1/2 \\
1/2 & 1/4 & 1/8 & 1/16 & \longleftarrow \leq 1
\end{array}
$$

Sommando i numeri sulla sinistra possiamo notare che otteniamo una serie nota che converge a 2. Quindi in media ho 2 rehashing, nel caso ne avessi di più significa che ho scelto in modo sbagliato i parametri a_i e b_i delle mie funzioni hash.

Ogni rehashing costa $O(n)$ in tempo, ma risulta che il costo *ammortizzato* del rehashing è $O(1)$.

4.5 Cancellazione

Analizziamo adesso il problema della cancellazione da un cuckoo hashing. Notiamo che il *cluster* (ovvero l'insieme delle posizioni nelle quali si può trovare un elemento) è 2, quindi ritrovare la chiave per rimuoverla è relativamente semplice.

Il problema è che la struttura dati generata è randomizzata mentre l'operazione di cancellazione è deterministica.

Domanda Una volta rimosso un elemento dal cuckoo hashing, riesco a scegliere altri h'_1 e h'_2 che ricreino la mia configurazione attuale?

Risposta Sì, scelgo h'_1 in modo che mappi gli elementi esattamente come sono posizionati adesso nella struttura, e scelgo h'_2 in modo libero.

Capitolo 5

Filtri di Bloom

I filtri di Bloom sono una struttura dati presentata negli anni '70 che permette la memorizzazione di un insieme di dati S offrendo un notevole risparmi in termini di spazio, causando però dei falsi positivi, ovvero rispondendo al test di appartenenza di un elemento all'insieme con `true` anche se l'elemento non appartiene all'insieme.

Sono utilizzati in quegli ambiti dove i falsi positivi non destano problemi, perchè la probabilità che hanno di occorrere sia sufficientemente bassa, specialmente nel campo del networking, ad esempio nei router e nei proxy.

I filtri di bloom si basano sul seguente principio:

Bloom Filter Principle 1. *When ever a list or set is used, and space is at premium, consider using a bloom filter if the effect of false positive can be mitigated.*

5.1 Definizione matematica

Un filtro di bloom per memorizzare un insieme $S = \{x_1, x_2, \cdots, x_n\}$ è un vettore di m bit che inizialmente sono settati tutti a 0.

Si scelgono inoltre in modo indipendente $h_1, \cdots, h_k$ funzioni hash da una famiglia di funzioni hash $H : S \to [m]$.

L'inserimento di un nuovo elemento avviene nel modo seguente:

```
INS(x)
    for(i=0; i<k; i++) T[hi(x)]=1;
```

Si calcolano ovvero tutte le k funzioni hash per il nuovo elemento, e si settano a 1 i bit nelle posizioni risultati dalle funzioni hash.

La ricerca viene avviene nel modo seguente:

```
SEARCH(X)
    j = 1
    while(j <= k && T[hj(x)] == 1)
        j++;
```

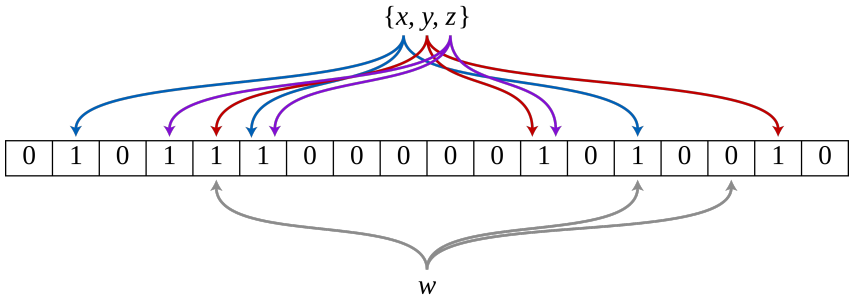


Figura 5.1: Filtro di bloom con $m = 18$ e $k = 3$

```
return j > k;
```

Si va dunque a verificare se tutti i bit nelle posizioni risultanti dal calcolo delle funzioni hash sull'elemento sono settati a 1.

Si vede facilmente che questa ricerca può generare dei falsi positivi, specialmente dopo sequenze di inserimenti abbastanza lunghe e per valori di m piccoli.

Nell'immagine seguente 'e presente un esempio di filtro di bloom: vediamo l'inserimento degli elementi x, y, z e la ricerca sull'elemento w.

5.2 Analisi

Vediamo facilmente che la probabilità per uno specifico bit di essere a 1 è $1/m$. Allora la probabilità per uno specifico bit di essere sempre a 0 dopo n inserimenti nel filtro di bloom può essere facilmente calcolato con il complementare come:

$$p' = \left(1 - \frac{1}{m}\right)^{kn}$$

Utilizzando uno sviluppo di Taylor ricordiamoci che possiamo esprimere e^x come:

$$e^x = 1 + x + \frac{1}{2}x^2 + \cdots$$

Quindi riscriviamo:

$$p' = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{-kn/m} = p$$

Intuitivamente notiamo che su aumentiamo la dimensione della tabella, aumenta il numero di zeri. Se invece aumentiamo il numero delle funzioni hash, il numero di zeri diminuisce.

Consideriamo adesso il fattore di carico ρ come il rapporto fra il numero di zeri e il totale degli elementi del filtro di bloom, risulta che il valore atteso per ρ è $E(\rho) = p'$. La probabilità di un falso positivo è dunque legata alla probabilità di ottenere un 1 per le k funzioni hash di un elemento ovvero:

$$(1 - \rho)^k \approx (1 - p')^k \approx (1 - p)^k$$

Lo sviluppo può essere visto anche come:

$$\begin{aligned}
\Pr(\text{false positive for y}) &= \Pr(\forall j : T[h_j(y)] = 1) \\
&= \prod_{j=1}^{k} \Pr(T[h_j(y)] = 1) \\
&= \prod_{j=1}^{k} 1 - \underbrace{\Pr(T[h_j(y)] = 0)}_{p'} \\
&= (1-p')^k \\
&= \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \\
&\approx (1-p)^k \\
&= \left(1 - e^{\frac{-kn}{m}}\right)^k
\end{aligned}$$

Intuitivamente notiamo che aumentando k, aumenta la probabilità di un falso positivo, mentre aumentando m, diminuisce la probabilità di un falso positivo.

5.2.1 Scelta dei parametri

Avendo calcolato la probabilità di un falso positivo, è possibile determinare una scelta ottima per il parametro k:

Consideriamo la derivata di $(1-p)^k$, che si azzera per

$$k = \ln 2 \cdot (m/n)$$

Così facendo la probabilità di un falso positivo diventa

$$(1/2)^k \approx (0.6185)^{m/n}$$

5.2.2 Rimozione di un elemento

La rimozione può essere complicata, soprattutto se l'unione dei bit relativi all'elemento e l'unione dei bit relativi a tutti gli altri elementi a intersezione non vuota.

Alternative possono essere:

1. Un algoritmo dove non si utilizzano bit, ma si utilizzano dei contatori, e si effettuano operazioni del tipo $T[h_j(x)] + +$. Questa soluzione presenta però un costo in spazio molto più elevato rispetto ai filtri di bloom classici. Si utilizzano $\log n$ bit (anche se con i chernoff bound si può dimostrare che si utilizza meno spazio).

2. Un algoritmo dove il vettore è separato in settori di dimensione m/k.

5.3 Operazioni Booleane

Supponiamo di avere due filtri di bloom S_1 e S_2, se vogliamo sapere se un elemento appartiene all'unione dei due insiemi, è sufficiente considerare il filtro di bloom $S_1 \vee S_2$, dove i bit sono messi in OR fra di loro.

Inoltre è possibile stimare la dimensione dell'intersezione di due insiemi nel modo seguente:

$$|S_1 \cup S_2| = \frac{S_1 \odot S_2}{k}$$

dove $S_1 \odot S_2$ rappresenta il prodotto scalare fra i due filtri di bloom coinvolti.

5.4 Compressione

Ai fini della compressione è conveniente realizzare un filtro di bloom più grosso, ma che contiene più 0, in modo che riuscirò a comprimerlo meglio, e trasmetterò meno bit.

Capitolo 6

Algoritmi Randomizzati

In questa sezione vedremo alcuni algoritmi randomizzati, che sfruttano il ruolo della casualità per cercare di rendere trascurabili le condizioni sfavorevoli dell'algortimo.

Considereremo un'analisi di tipo avversariale, dove consideriamo l'esistenza di un avversario che, con maliziosità, cerca di presentarci sempre le condizioni sfavorevoli. In questo contesto cercheremo di usare la casualità a nostro favore.

Consideriamo il caso del quicksort:

6.1 Il quicksort

L'algoritmo di ordinamento del quicksort è descritto nel codice seguente:

```
1 void QuickSort(A, sx, dx){
2     if(sx < dx){
3         pivot = random(sx..dx);
4         px = distribution(A, sx, pivot, dx);
5         QuickSort(A, sx, px-1);
6         QuickSort(A, px+1, dx);
7     }
8 }
```

Risulta evidente che se il pivot venisse scelto come uno dei due estremi dell'array ci troveremo nella situazione peggiore. La chiamata ricorsiva costerebbe infatti

$$T(n) \leq T(n-1) + cn$$

E quindi l'algoritmo avrebbe un costo quadratico

$$T(n) = O(n^2)$$

Proviamo ad effettuare un'analisi cercando di calcolare il tempo di esecuzione atteso che sappiamo essere $n \log n$:

$$\underbrace{E[\text{runningtimeQS}]}_{\tilde{T}} = O(n \log n)$$

Partiamo dall'ipotesi che la nostra funzione random() restituisca numeri distribuiti in modo uniforme fra sx e dx.

A seguito dell'invocazione della funzione distribution() abbiamo che il nostro px risulta distribuito in modo uniforme fra sx e dx (proprio come per la funzione random()). Vale dunque che una scelta casuale di pivot equivale ad una scelta casuale di px.

Adesso supponiamo di dividere il nostro array in 4 zone[1] contigue, dato che il nostro px è equidistribuito possono succedere 2 casi:

1. px si trova in una delle due zone interne, ciò può accadere con probabilità $\frac{1}{2}$. Questo è per noi un caso fortunato.
2. px si trova in una delle due zone esterne, ciò può accadere con probabilità $\frac{1}{2}$. Questo è per noi un caso sfortunato.

Nel primo caso, consideriamo il caso pessimo dove px si trova in uno dei due estremi delle zone, allora vale che:

$$\tilde{T}(n) = \tilde{T}\left(\frac{1}{4}n\right) + \tilde{T}\left(\frac{3}{4}n\right) + cn$$

Mentre nel secondo caso vale che:

$$\tilde{T}(n) \leq \tilde{T}(n-1) + cn = O(n^2)$$

Facciamo la media pesata dei due termini e risolviamo:

$$\tilde{T}(n) \leq \frac{1}{2}\left[\tilde{T}\left(\frac{1}{4}n\right) + \tilde{T}\left(\frac{3}{4}n\right) + \tilde{T}(n-1)\right] + cn$$

$$\tilde{T}(n) \leq \frac{1}{2}\left[\tilde{T}\left(\frac{1}{4}n\right) + \tilde{T}\left(\frac{3}{4}n\right) + \tilde{T}(n)\right] + cn$$

$$2\tilde{T}(n) \leq \left[\tilde{T}\left(\frac{1}{4}n\right) + \tilde{T}\left(\frac{3}{4}n\right) + \tilde{T}(n)\right] + 2cn$$

$$\tilde{T}(n) \leq \left[\tilde{T}\left(\frac{1}{4}n\right) + \tilde{T}\left(\frac{3}{4}n\right)\right] + 2cn$$

$$\tilde{T}(n) = O(n \log n)$$

Vediamo dunque che con una scelta casuale equamente distrubuita del pivot riusciamo ad evitare di finire sempre nel caso sfortunato, ed abbiamo un algoritmo che costa così $O(n \log n)$.

[1] L'analisi vale per qualunque numero $i \geq 3$

6.2 Le skiplist

Le *skiplist* (o liste a saltelli) sono delle liste speciali che permettono di fare operazioni di ricerca su liste in un tempo medio di $O(\log n)$.

In questa struttura dati, proprio come nel quicksort, il caso svolge un ruolo determinante per avere un tempo medio logaritmico.

Le *skiplist* si costruiscono in questo modo: partendo da una lista ordinata di n elementi, si creano delle repliche della lista in modo che ogni elemento in posizione i della lista sia presente nelle r_i liste a lui superiore, dove 2^{r_i} rappresenta la massima potenza del 2 che divide i. Il numero delle liste è dato dal massimo numero di $r_i + 1$ e vale che $h = O(\log n)$.

Il problema sussiste nel momento in cui si vuole aggiungere un nuovo elemento, è infatti semplice trovare il luogo adatto dove inserire il nuovo elemento, però per mantenere le proprietà della *skiplist* si dovrebbe ricostruire tutta la struttura.

Si preferisce piuttosto di decidere in modo casuale in quali liste inserire un nuovo elemento: lanciamo una moneta (quindi un evento casuale che ha probabilità $\frac{1}{2}$) e continuiamo a lanciarla fin quando otteniamo testa, non appena otteniamo una croce ci fermiamo. Supponendo di aver ottenuto croce dopo i lanci, allora replichiamo l'elemento nelle $i - 1$ liste superiori[2]. Il costo di questa operazione risulta essere $O(\log n)$.

Dato che i lanci della moneta sono indipendenti, notiamo che la probabilità che un elemento sia presente nella $i-$ esima lista risulta essere $\frac{1}{2^i}$.

6.2.1 Ricerca nelle *skiplist*

Vediamo che la ricerca di un elemento k è proporzionale a $O(T(l))$, dove $T(l)$ è il numero di elementi che vengono incontrati a partire dalla lista di altezza l fino al predecessore di k nella lista L_0.

Osserviamo che il percorso inverso di ricerca è a gradini, per cui per un generico elemento $e \in L_l$ possono succedere due avvenimenti:

1. Il percorso proviene dall'elemento al livello inferiore, dove il costo risulta essere $T(l-1)$. Ciò può avvenire con probabilità $\frac{1}{2^i}$, dato che è la stessa probabilità con cui abbiamo effettuato i lanci della nostra moneta per verificare se replicare o meno un elemento.

2. Il percorso proviene dall'elemento a destra, dove il costo medio risulta essere $T(l)$, ciò vuol dire che e non ha una copia al livello superiorie. Ciò avviene con probabilità uguale a $\frac{1}{2^i}$, ovvero un lancio dove è uscito croce.

Risulta allora che:

$$T(l) \leq \frac{1}{2}T(l) + \frac{1}{2}T(l-1) + c'$$

$$T(l) \leq T(l-1) + c'$$

[2]Questo procedimento può essere simulato con una funzione `random()` che restituisce 1 o 0 in modo equidistribuito.

Quindi:

$$T(h) = O(h) = O(\log n)$$

6.2.2 Cancellazione nelle *skiplist*

Per quanto riguarda la cancellazione è sufficiente rimuovere l'elemento dalla struttura, e non si perde la casualità della struttura.

Rimuovendo un elemento si vede infatti facilmente che è possibile ricreare la struttura senza quell'elemento in modo casuale.

6.3 I Random Binary Search Trees

I Random Binary Search Trees (RBST) sono un'altra struttura dati che sfrutta la casualità per generare degli alberi binari di ricerca. Essi mantengono le stesse proprietà dei Binary Search Tree.

L'altezza di un BRST è in media logaritmica, rendendo il costo delle operazioni di inserimento e ricerca in media logaritmici.

I RBST si basano su una proprietà fondamentale: data una sequenza di n chiavi K, ogni chiave può essere radice del RBST con probabilità $\frac{1}{n}$.

6.3.1 Inserimento in un RBST

Considerando la proprietà precedente, nel momento in cui voglio inserire un nuovo elemento, devo prima testare se questo nuovo elemento può essere la radice del mio RBST. Questo può avvenire con probabilià $\frac{1}{n+1}$, provo quindi a generare un numero casuale da 0 a n:

- Se esce 0, allora l'elemento sarà radice, e collego il vecchio albero come figlio sinistro/-destro a seconda del valore. Dopodiché eseguo una procedura split() per effettuare un taglio del vecchio RBST in modo da redistribuire i nodi[3].
- Se esce $k \neq 0$ allora itero su uno dei due sottoalberi (che sono anch'essi dei RBST), secondo il valore

,

Segue l'algoritmo di inserimento nel RBST:

```
1  rbst insert(int x, rbst T){
2      int n, r;
3
4      n = T->size;
5      r = random(0, n);
6
7      if (r == n)
```

[3]Vedi articolo di Martizen, Rouna, pag. 293

```
8           return insert_at_root(x, T);
9       if (x < T->key)
10          T->left = insert(x, T->left);
11      else
12          T->right = insert(x, T->right);
13      return T;
14  }
```

Capitolo 7

Compressione Dati

Gli algoritmi di compressione permettono di comprire un dato S in un dato Z con un numero inferiore di bit.

Esistono due famiglie di algoritmi di compressione:

algoritmi loseless ovvero che permettono di comprimere e di decomprimere senza perdita di informazioni

algoritmi lossy ovvero che comportano perdita di informazioni durante il processo di compressione.

Per studiare gli algoritmi loseless dobbiamo introdurre il concetto di entropia.

7.1 Entropia

L'entropia, secondo quanto definito da Shannon, rappresenta la quantità di incertezza presente in un segnale.

Supponiamo $S \in \Sigma^n$ dove S rappresenta una stringa dell'alfabeto Σ di lunghezza n. Contiamo le occorrenze di ogni simbolo distinto in questo modo:

$$n_c = \{i : 0 \leq i \leq n-1 | S[i] = c\}$$

e definiamo le probabilità di occorrenza di ogni singolo simbolo come $p_c = \frac{n_c}{n}$, definiamo allora l'entropia come segue:

$$H(s) = -\sum_{c \in \Sigma} p_c \log p_c = \sum_{c \in \Sigma} p_c \log \frac{1}{p_c}$$

Grazie alla definizione di entropia di Shannon è possibile enunciare il primo teorema di Shannon:

Primo Teorema di Shannon 1. *Per trasmettere un segnale S di lunghezza n senza perdita di informazione sono necessari almeno $nH(s)$ bit.*

Nell'ambito della complessità risulta interessante considerare anche la complessità di Kolmogorov.

7.2 Complessità di Kolmogorov

Sia F un formal computation model allora definiamo la Complessità di Kolmogorov ($K_F(S)$) come il più breve programma scritto in F che riesce a generare S.

Ovviamente $K_F(S)$ non è calcolabile, e risulta invariante rispetto F, a meno di una costante, vale infatti che:$K_{F'}(S) = K_F(S) + \Theta(1)$

Notiamo che l'entropia definita da Shannon è relativa ad una sorgente di stringhe, mentre invece la complessità di Kolmogorov è relativa ad una singola stringa.

7.3 Elias Coding

Vediamo adesso due codifiche proposte da Peter Elias nel '75, il $\gamma - code$ e il $\delta - code$.

I codici di Elias sono codici a codifica variabile, ovvero codici che assegnano quantità di bit differente ad ogni simbolo ed inoltre sono codici *prefix-free*, ovvero non può accadere che un simbolo venga codificato con k e un altro simbolo venga codificato con k' tale che k' è prefisso di k.

7.3.1 Il $\delta - code$

Per codificare un numero x con il suo $\delta - code$ si eseguono i seguenti passi:

1. Si calcola la più grande potenza del 2 contenuta dentro x ovvero $c(x) = \lceil \log_2(x) \rceil$,
2. Si codifica $c(x)$ in unario, ovvero lo si rappresenta come una sequenza di $c(x) - 1$ zeri seguiti da un uno ($0^{c(x)-1}1$),
3. Accodiamo la rappresentazione binaria di x senza il primo uno (che è stato inserito al punto precedente).

Abbiamo così ottenuto la codifica che risulta occupare $2\lfloor \log_2(x) \rfloor + 1$ bit.

Il $\delta - code$ viene utilizzato per codificare sequenze di numeri di cui non è noto a priori l'upper bound oppure sequenze dove i numeri più piccoli sono più frequenti (dato che la loro rappresentazione è più breve).

7.3.2 Il $\gamma - code$

Il $\gamma - code$ procede in modo simile al $\delta - code$ visto poco fa:

1. Si calcola la più grande potenza del 2 contenuta dentro x ovvero $c(x) = \lceil \log_2(x) \rceil$,
2. Si calcola $\delta(c(x) + 1)$
3. Accodiamo la rappresentazione binaria di x senza il primo uno (che è stato inserito al punto precedente).

Il $\gamma - code$ occupa così $\lfloor \log_2(x) \rfloor + 2\lfloor \log_2(\lfloor \log_2(x) \rfloor + 1) \rfloor + 1$

7.4 Codifica di Hufmann

La codifica di Huffman è stata realizzata nel 1952 da David Huffman, è una codifica a lunghezza variabile, *prefix-free* ed è una codifica che assegna codifiche di lunghezza minore ai simboli più frequenti.

Si può dimostrare inoltre che la codifica di Hufmann assenga le codifiche più brevi ai singoli più frequenti rispetto ad altre codifiche. Vale ovvero che il termine

$$\sum_{c \in \Sigma} l_c p_c$$

dove l_c rappresenta la lunghezza della codifica del simbolo c e p_c la sua frequenza nella stringa viene minimizzato, e vale $nH(S) + n$.

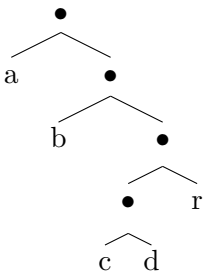
7.4.1 Descrizione della codifica

La codifica opera in questo modo:

1. Calcola le frequenze per ogni simbolo della stringa e li inserisce in una coda.
2. Fin quando la coda non è vuota:
   (a) Considera i due elementi a frequenza minore e li rimuove dalla lista,
   (b) Crea un albero usando come figli i due elementi e inserendo come padre un nuovo elemento.
   (c) Si inserisce nella lista il padre che ha come frequenza la somma delle frequenze dei figli.
3. Una volta generato l'albero si assegna una codifica binaria ad ogni numero (ad esempio 0 se si sceglie il ramo sinistro e 1 se si sceglie il ramo destro).

Come si vede l'albero viene generato partendo dal basso ed inserendo prima gli elementi meno frequenti che quindi avranno cammini più lunghi e di conseguenza codifiche più lunghe.

Esempio Vogliamo codificare la stringa *abracadabra*. L'albero risultate è:



Possiamo dunque codificare i vari caratteri nel modo seguente: $a = 0$, $b = 10$, $c = 1100$, $d = 1101$, $r = 111$.

7.5 Codifica LZ77/LZ78

Gli algoritmi LZ77 ed LZ78 sono due algoritmi di compressione loseless cercano di sfruttare la ricerca di sottostringhe ricorrenti all'interno dello stream al fine di ridurre la quantità di dati da memorizzare.

7.5.1 LZ77

La prima codifica, LZ77, analizza lo stream in modo sequenziale va a cercare per ogni simbolo se vi sono delle occorrenze passate. Per ogni simbolo codifica una tripla così fatta:

$$(i - i', |\alpha|, c)$$

dove

$$i - i'$$

rappresenta la distanza dal simbolo attuale e l'occorrenza passata[1], $|\alpha|$ rappresenta la dimensione della sottostringa, e c rappresenta il prossimo carattere incontrato.

Ad esempio la stringa *abracadabra* viene codificata così:

$$S = |a|b|r|ac|ad|abra\$$$

$$Z = (0, 0, a), (0, 0, b), (0, 0, r), (3, 1, c), (2, 1, a), (7, 4, \$)$$

Oppure la stringa *abababababab* viene codificata così:

$$S = a|b|abababab\$$$

$$Z = (0, 0, a), (0, 0, b), (2, 8, \$)$$

7.5.2 LZ78

La seconda codifica, LZ78, usa invece un dizionario per memorizzare le sottostringhe incontrate. Risulta però leggermente meno efficente della codifica LZ77.

[1] Si noti che viene memorizzata la posizione relativa e non assoluta, dato che questo algoritmo può essere utilizzato anche per file di grandi dimensioni

Parte II

Hard Problems

Capitolo 8

R-Approssimazione

In questo capitolo vedremo un modo per approssimare problemi complessi, ad esempio problemi appartenendi alla classe NP-Hard. Per i problemi in questa classe si conoscono degli algoritmi per risolverli in tempo EXP ma non si conoscono algortimi che li risolvono in tempo P.

Molti problemi appartengono alla classe NP-Hard, quali ad esempio problemi legati ai trasporti, o all'economia. Per questo motivo si cercano algoritmi che approssimino le soluzioni dei problemi in tempo inferiore ad EXP.

Definiamo la classe NP come

$$\text{NP} = \{\pi \in \{0,1\}^* \ : \exists \text{poly } p(), V :$$

$$\begin{aligned} &1) \quad \forall x \in \{0,1\}^*,\ x \in \pi \Leftrightarrow \quad \exists y \in \{0,1\}^*, |y| = p(|x|), V(x,y) = 1 \\ &2) \quad \forall x \in \{0,1\}^*,\ x \notin \pi \Leftrightarrow \quad \forall y \in \{0,1\}^*, V(x,y) = 0\} \end{aligned} \tag{8.1}$$

Fra i possibili problemi all'interno di una classe si possono definire due tipi di problemi:

- Problemi decisionali,
- Problemi di ottimizzazione.

Nei problemi decisionali si tratta di restituire ad una domanda di appartenenza o meno ad un insieme, quindi le possibili risposte possono essere true o false. Nei problemi di ottimizzazione si cerca una soluzione che minimizzi/massimizzi una funzione costo/beneficio.

In particolare definiamo la funzione costo/beneficio come:

$$C : \pi \to \mathbb{R}$$

E distinguiamo due tipi di problemi di ottimizzazione: i problemi di minimo, in cui cerchiamo di minimizzare il costo, cerchiamo dunque

$$\arg\min_{i \in \pi} C(i)$$

Mentre nei problemi di massimo cerchiamo di massimizzare il beneficio, ovvero:

$$\arg\max_{i \in \pi} C(i)$$

.

8.0.3 Problema del Commesso Viaggiatore

Il problema del commesso viaggiatore (*TSP - Travel Salesman Person*) è un problema di minimo in cui, dato un insieme di N città tra cui sono definite le distanze $D[]$, si cerca un Tour (una sequenza) di città che attraversi tutte le città tale che minimizzi:

$$C(i) = (\sum_{i} D(T[i], T[i+1])) + D(T[n], T[0])$$

Possiamo anche definire la versione decisionale, in modo da decidere se $\exists$ un Tour tale che $C(T) < k$, dove k è una costante data.