

Algoritmi per Big Data

Andrea Bonifati - A.A. 2023/2024

Introduzione al Data Mining

Data Mining

Con il termine *data mining* si fa spesso riferimento al processo di creazione di un modello a partire da determinati dati. Più generalmente, l'obiettivo del data mining è quello di definire un algoritmo. In generale, la definizione di un modello per Data Sets complessi e/o di grandi dimensioni può essere schematizzato come segue:

- Fase 1: Trasformare il Data Set in input originale in un modello/struttura formale **I** in maniera tale che il problema reale originale possa essere ridotto ad un task computazionale ben definito **T** su input **I**.
- Fase 2: Trovare il miglior algoritmo che risolva **T** su input **I**.

Esempio: Si consideri il problema di individuare le mail di phishing. Una soluzione algoritmica può essere schematizzata come segue:

1. Si classificano le mail ricevute in due sottoinsiemi: Phishing/Non-phishing.
2. Si estraggono le parole (o frasi) che appaiono insolitamente spesso nelle mail di phishing (ad esempio "send money").
3. Si assegnano pesi positivi alle parole di phishing individuate al passo precedente e pesi negativi alle restanti.
4. Si definisce il seguente algoritmo: per ogni e-mail in ingresso, calcola la somma dei pesi delle parole. Se tale somma è maggiore di una certa soglia, imposta la mail come Phishing. Altrimenti, impostala come Non-phishing.

Si fa presente ora come, in questo esempio, i task fondamentali più complessi sono i seguenti:

2. Definire un modello statistico adatto **M** (ad esempio una distribuzione di probabilità) sull'input di dati reali in maniera tale che l'informazione nascosta possa emergere come un evento probabile in **M**, e successivamente estrarre queste informazioni in maniera efficiente.
3. Assegnare i giusti pesi alle parole in maniera tale che la somma totale di una mail entrante sia proporzionale alla probabilità che questa sia di phishing.

Statistical modeling: definizione informale

Consiste nella costruzione di una distribuzione di probabilità sottostante al modello, dalla quale vengono campionati i dati visibili "grezzi".

Esempio: Il data set grezzo è un insieme \mathbf{D} di numeri. Un modello statistico per \mathbf{D} è dato dall'assunzione che i numeri vengano campionati secondo una *distribuzione Gaussiana* su \mathbf{D} . In questo caso, la media e la deviazione standard caratterizzano completamente la distribuzione in questione, la quale diventerebbe il modello per l'*Input Data*.

Adversarial data models: definizione informale

Altri scenari tipici nel data mining adottano la *worst-case hypothesis*, ossia l'ipotesi del caso peggiore:

Il Data Set in input, ossia da dove deve venir estratta l'informazione cercata, è gestito da una fonte avversaria la quale genera i dati con l'obiettivo di minimizzare l'efficienza della soluzione algoritmica utilizzata o, ancor peggio, la sua validità.

Esempio: Calcolare (e aggiornare) il numero di 1 nell'ultima finestra di lunghezza N su un flusso infinito di bits gestito da un avversario, il quale sceglie il prossimo bit in funzioni delle precedenti scelte effettuate dall'algoritmo.

Data Mining & Machine Learning

Il Machine Learning risulta spesso un ottimo approccio al data mining. Consiste nell'usare parte del Data Set in input come Training Set, per istruire i sistemi ML. Risulta essere un buon metodo nelle situazioni in cui non è possibile definire una funzione obiettivo chiara sul Data Set, ossia quando non si sa cosa l'input data dice in merito al problema da risolvere.

Limiti statistici sul Data Mining

L'obiettivo tipico dei problemi di Data Mining è quello di individuare eventi inusuali all'interno di un Data Set di grandi dimensioni. Spesso, gli elementi che costituiscono un Data Set possono essere ottenuti combinando diverse fonti di informazione. L'integrazione dell'informazione, ossia l'idea di relazionare e combinare diverse fonti di dati per ottenere risultati non disponibili a partire da una singola fonte, risulta essere spesso un passo principale per la risoluzione di un problema. Tale approccio può però portare a un problema tecnico: se il Data Set contiene un gran numero di elementi ottenuti da diverse fonti, allora, in maniera casuale, potrebbero verificarsi eventi inusuali, i quali possono non avere un particolare significato, essendo semplicemente artefatti statistici senza significato

Principio di Bonferroni

Si supponga di avere un data set DS in input, e di cercare uno specifico evento E al suo interno. Ci si può aspettare che tale evento accada, anche se DS è completamente casuale, e che il numero di occorrenze di tale evento aumenti al crescere della dimensione di DS . Queste occorrenze prendono il nome di *falsi* ("bogus"): il loro verificarsi non ha nessuna causa, sennonché dati random avranno sempre un certo numero di caratteristiche inusuali che sembrano rilevati, ma che in realtà non lo sono. La correzione di Bonferroni è un teorema che fornisce un metodo per l'individuazione di queste risposte "*falso-positive*" a seguito della ricerca dell'occorrenza di un dato evento all'interno di un data set. Il principio di Bonferroni è una versione informale del rispettivo teorema, che permette di evitare di trattare le occorrenze casuali di un evento come se queste fossero reali:

- Si calcola il numero atteso di occorrenze di un dato evento di interesse, assumendo che il data set sia random.
- Se questo numero risulta essere significativamente più grande del numero di istanze reali che si spera di individuare, allora ci si aspetta che la maggior parte di occorrenze individuate sia un falso.

Si vede ora un esempio di bogus: Gangs in Social Network. In questo problema si ha una rete sociale, modellata mediante grafo, dove ogni agente che vi appartiene è un potenziale criminale. Gli agenti visitano periodicamente dei luoghi pubblici. Assumendo che i criminali si incontrino periodicamente in gruppo, si vogliono individuare queste gangs.

Gangs in social network

Sia $G = (V, E)$ un grafo. Un insieme V di agenti, potenziali criminali ($|V| = n$ con n grande), visita un insieme H di luoghi pubblici in una città ($|H| = h$ con h grande).

L'arco (u, v) appartiene ad E se u e v visitano lo stesso luogo (appartenente ad H) lo stesso giorno per una sequenza di $T \gg 0$ giorni, dove $T =$ numero di giorni. Si assume che il processo di visita degli agenti nei vari luoghi sia sufficientemente casuale e uniforme. Inoltre, sia per ipotesi $h \gg T$. Si vogliono individuare le possibili *gangs*, che, per modellazione del problema, sono rappresentate all'interno di G mediante *cliques*.

Per fare ciò, ci si chiede quale sia la dimensione massimale delle cliques in G .

Si ha che $\forall u, v \in V$

$$\Pr[(u, v) \in E] \simeq \frac{T}{h} \doteq p$$

Sia $S \subseteq V$ con $|S| = s$. Si ha che

$$\Pr[S \text{ è una clique}] \simeq p^{(s^2/2)}$$

Quindi si ha che

$$\mathbf{E}[\# \text{ cliques di size } s] = C(n, s) \cdot p^{(s^2/2)} \simeq \left(\frac{n}{e \cdot s}\right)^s \cdot p^{(s^2/2)} \quad (1)$$

Osservando un grande sistema spazio-tempo, ad esempio per $n \simeq 10^7$, $h \simeq 10^4$, $T \simeq 10^3$ si ha che $p \simeq \frac{1}{10}$. Dalla (1) ci si aspetta un gran numero di cliques aventi size $s \simeq 10$. Assumendo che non ci si aspetta minimamente un numero così grande di gangs criminali, la presenza di molte cliques aventi size 10 non permette quindi l'individuazione dei criminali, e quindi l'occorrenza di questo evento non risultano utili per l'informazione cercata.

Algoritmi probabilistici

Richiami di probabilità e statistica

Axiomi di probabilità

Uno **spazio di campionamento** Ω è un insieme i quali elementi descrivono gli esiti di un esperimento di interesse.

I sottoinsiemi di uno spazio di campionamento prendono il nome di **eventi**. Si dice che un evento A accade se l'esito di un esperimento è un elemento di A .

Gli eventi possono essere combinati secondo le operazioni insiemistiche.

Sia Ω uno spazio di campionamento e siano $E_1 \subseteq \Omega$, $E_2 \subseteq \Omega$ due eventi.

L'evento $E_1 \cap E_2$ prende il nome di *intersezione* dei due eventi, ed accade se si verificano entrambi gli eventi E_1 ed E_2 .

L'insieme $E_1 \cup E_2$ prende il nome di *unione* dei due eventi, ed accade se si verifica almeno uno dei due eventi E_1 ed E_2 .

L'insieme $E^c = \{\omega \in \Omega : \omega \notin E\}$ è il *complemento* dell'evento E , ed accade se e solo se l'evento E non si verifica. Si ha che $\Omega^c = \{\emptyset\}$.

I due eventi E_1 ed E_2 si dicono *disgiunti* o *mutualmente esclusivi* se non hanno esiti in comune, ossia $E_1 \cap E_2 = \emptyset$.

Si dice che l'evento E_1 *implica* l'evento E_2 se l'esito di E_1 è contenuto in E_2 , ossia $E_1 \subset E_2$.

Essendo insiemi, gli eventi sono soggetti alle proprietà insiemistiche.

Leggi di DeMorgan

Per ogni coppia di eventi E_1 ed E_2 si ha che

$$(E_1 \cup E_2)^c = E_1^c \cap E_2^c \quad \text{e} \quad (E_1 \cap E_2)^c = E_1^c \cup E_2^c$$

Si vuole esprimere quanto sia probabile il verificarsi di un evento. Per fare ciò, si definisce una **funzione di probabilità**. In questo corso si farà sempre riferimento a spazi di probabilità *discreti*, ossia dove lo spazio di campionamento Ω risulta essere finito o numerabile.

Definizione (Funzione di probabilità discreta):

Una *funzione di probabilità (discreta)* è una funzione $\mathbf{Pr} : \Omega \rightarrow [0, 1]$ che soddisfa le seguenti condizioni:

1. $\mathbf{Pr}(\Omega) = 1$,
2. Per ogni sequenza finita o numerabile di eventi mutualmente disgiunti due a due E_1, E_2, E_3, \dots si ha che

$$\mathbf{Pr}\left(\bigcup_{i \geq 1} E_i\right) = \sum_{i \geq 1} \mathbf{Pr}(E_i)$$

Osservazione

Siano E_1 ed E_2 due eventi disgiunti, allora

$$E_1 \cap E_2 = \emptyset \Rightarrow \mathbf{Pr}(E_1 \cap E_2) = 0$$

Siano E_1 ed E_2 due eventi *non* disgiunti.

Si ha che

$$E_1 = (E_1 \cap E_2) \cup (E_1 \cap E_2^c)$$

A partire da ciò, si ottiene quindi la seguente uguaglianza:

$$\mathbf{Pr}(E_1) = \mathbf{Pr}(E_1 \cap E_2) + \mathbf{Pr}(E_1 \cap E_2^c) \quad (2)$$

Applicando lo stesso ragionamento per l'evento $E_1 \cup E_2$ si ha che

$$(E_1 \cup E_2) = ((E_1 \cup E_2) \cap E_2) \cup ((E_1 \cup E_2) \cap E_2^c)$$

Osservando che

$$(E_1 \cup E_2) \cap E_2 = E_2$$

$$(E_1 \cup E_2) \cap E_2^c = E_1 \cap E_2^c$$

Vale quindi

$$\mathbf{Pr}(E_1 \cup E_2) = \mathbf{Pr}(E_2) + \mathbf{Pr}(E_1 \cap E_2^c)$$

E, ponendo $\mathbf{Pr}(E_1 \cap E_2^c) = \mathbf{Pr}(E_1) - \mathbf{Pr}(E_1 \cap E_2)$ dalla (2) si ottiene il seguente lemma

Lemma

Siano E_1 ed E_2 due eventi, allora

$$\mathbf{Pr}(E_1 \cup E_2) = \mathbf{Pr}(E_1) + \mathbf{Pr}(E_2) - \mathbf{Pr}(E_1 \cap E_2)$$

📘 Osservazione

Siano E_1 ed E_2 due eventi disgiunti, allora

$$\mathbf{Pr}(E_1 \cup E_2) = \mathbf{Pr}(E_1) + \mathbf{Pr}(E_2)$$

essendo $\mathbf{Pr}(E_1 \cap E_2) = 0$.

Il lemma appena enunciato può essere generalizzato a più eventi:

📘 Lemma (Principio di inclusione ed esclusione):

Sia E_1, \dots, E_n una sequenza di n eventi. allora

$$\begin{aligned} \mathbf{Pr}\left(\bigcup_{i \geq 1}^n E_i\right) &= \sum_{i=1}^n \mathbf{Pr}(E_i) - \sum_{i < j} \mathbf{Pr}(E_i \cap E_j) + \sum_{i < j < k} \mathbf{Pr}(E_i \cap E_j \cap E_k) - \dots \\ &\dots + (-1)^{l+1} \sum_{i_1 < i_2 < \dots < i_l} \mathbf{Pr}\left(\bigcap_{r=1}^l E_{i_r}\right) + \dots . \end{aligned}$$

Inoltre, essendo che, dato un evento $E \subseteq \Omega$ vale $E \cup E^c = \Omega$, si ha che

$$\mathbf{Pr}(E) = 1 - \mathbf{Pr}(E^c)$$

Una conseguenza della definizione di funzione di probabilità discreta prende il nome di ***union bound***:

📘 Lemma (Union bound):

Per ogni sequenza finita o numerabile di eventi E_1, E_2, E_3, \dots

$$\mathbf{Pr}\left(\bigcup_{i \geq 1} E_i\right) \leq \sum_{i \geq 1} \mathbf{Pr}(E_i)$$

Si osserva esplicitamente come questo lemma differisca dalla seconda condizione della definizione di funzione di probabilità discreta, la quale richiede che gli eventi siano disgiunti due a due.

A partire dai concetti appena enunciati, si può definire lo ***spazio di probabilità***:

■ Definizione (Spazio di probabilità):

Uno spazio di probabilità ha tre componenti:

1. Uno spazio di campionamento Ω , ossia l'insieme di tutti i possibili esiti del processo aleatorio modellato dallo spazio di probabilità.
2. Una famiglia di insiemi \mathcal{F} che rappresenta tutti i possibili eventi, dove per ogni insieme $E \in \mathcal{F}$, detto *evento* si ha $E \subseteq \Omega$.
3. Una funzione di probabilità $\Pr : \mathcal{F} \rightarrow \mathbb{R}$.

Un elemento di Ω è chiamato *evento semplice*. In uno spazio di probabilità discreto, la funzione di probabilità è definita unicamente dalle probabilità di eventi semplici, quindi $\Pr : \Omega \rightarrow [0, 1]$.

Probabilità condizionata ed indipendenza

L'occorrenza di un evento può andare a modificare la probabilità che si verifichi un evento successivo correlato ad esso. Tale probabilità prende il nome di **probabilità condizionata**.

■ Definizione (Probabilità condizionata):

Siano E_1 ed E_2 due eventi.

La probabilità condizionata di E_1 dato E_2 è

$$\Pr(E_1|E_2) = \frac{\Pr(E_1 \cap E_2)}{\Pr(E_2)}$$

assumendo che $\Pr(E_2) > 0$

Da questa definizione, moltiplicando entrambi i membri dell'equazione per $\Pr(E_2)$ si ottiene la **regola della moltiplicazione**.

■ Regola della moltiplicazione

Per ogni coppia di eventi E_1 ed E_2 si ha

$$\Pr(E_1 \cap E_2) = \Pr(E_1|E_2)\Pr(E_2)$$

Da ciò segue nell'immediato che

$$\Pr(E_1|E_2) = \frac{\Pr(E_1 \cap E_2)}{\Pr(E_2)}$$

Si fa presente come il calcolo di $\Pr(E_1 \cap E_2)$ sia eseguibile sia condizionando la probabilità di E_1 su E_2 , avendo quindi

$$\Pr(E_1 \cap E_2) = \Pr(E_1|E_2)\Pr(E_2)$$

oppure condizionando la probabilità E_2 rispetto a E_1

$$\Pr(E_1 \cap E_2) = \Pr(E_2|E_1)\Pr(E_1)$$

Nella maggior parte dei casi, si ha che una delle due tra $\Pr(E_1|E_2)$ e $\Pr(E_2|E_1)$ risulta facile da calcolare, al contrario dell'altra.

E' possibile calcolare la probabilità di un evento mediante le probabilità condizionate di una serie di eventi disgiunti, i quali presi complessivamente vanno a formare lo spazio di campionamento.

■ Teorema (Teorema della probabilità totale)

Siano E_1, E_2, \dots, E_m eventi mutualmente disgiunti tali per cui $E_1 \cup E_2 \cup \dots \cup E_m = \Omega$.

La probabilità di un evento arbitrario $E \subseteq \Omega$ può essere espressa come

$$\begin{aligned}\Pr(E) &= \sum_{i=1}^m \Pr(E \cap E_i) \\ &= \Pr(E|E_1)\Pr(E_1) + \dots + \Pr(E|E_m)\Pr(E_m) \\ &= \sum_{i=1}^m \Pr(E|E_i)\Pr(E_i).\end{aligned}$$

■ Dimostrazione (Teorema della probabilità totale)

Essendo gli eventi E_i per $i = 1, \dots, m$ disgiunti e tali per cui $\bigcup_{i=1}^m E_i = \Omega$, si ha che

$$\Pr(E) = \sum_{i=1}^m \Pr(E \cap E_i)$$

Dalla definizione di probabilità condizionata risulta quindi

$$\sum_{i=1}^m \Pr(E \cap E_i) = \sum_{i=1}^m \Pr(E|E_i)\Pr(E_i)$$

■ Osservazione

Essendo E_1, E_2, \dots, E_m eventi mutualmente disgiunti

($\forall i, j = 1, \dots, m$ tali per cui $i \neq j$ vale $E_i \cap E_j = \emptyset$) e $E_1 \cup E_2 \cup \dots \cup E_m = \Omega$, si ha che tale sequenza di eventi definisce una partizione dello spazio di campionamento.

Si vede ora un teorema che mette in relazione le probabilità condizionate, nota come **teorema di Bayes**.

■ Teorema di Bayes:

Siano E_1, E_2, \dots, E_m eventi disgiunti tali per cui $E_1 \cap E_2 \cap \dots \cap E_m = \Omega$. La probabilità condizionata di E_i dato un evento arbitrario E può essere espressa come:

$$\Pr(E_i|E) = \frac{\Pr(E|E_i)\Pr(E_i)}{\Pr(E|E_1)\Pr(E_1) + \Pr(E|E_2)\Pr(E_2) + \dots + \Pr(E|E_m)\Pr(E_m)}$$

Tale teorema segue dalla relazione:

$$\Pr(E_i|E) = \frac{\Pr(E|E_i)\Pr(E_i)}{\Pr(E)}$$

Combinata con il Teorema della probabilità totale su $\Pr(E)$.

Eventi di uno stesso spazio di campionamento possono non essere correlati tra loro. In questo caso si parla di *indipendenza* tra eventi.

■ Definizione (Eventi indipendenti):

Un evento E_1 si dice indipendente da E_2 se

$$\Pr(E_1|E_2) = \Pr(E_1).$$

A partire dalla definizione di eventi indipendenti si possono effettuare diverse osservazioni.

Essendo

$$\Pr(E_1^c|E_2) = 1 - \Pr(E_1|E_2)$$

$$\Pr(E_1^c) = 1 - \Pr(E_1)$$

si ha che

$$E_1 \text{ è indipendente da } E_2 \iff E_1^c \text{ è indipendente da } E_2$$

Applicando la regola della moltiplicazione, se E_1 è indipendente da E_2 allora

$$\Pr(E_1 \cap E_2) = \Pr(E_1|E_2)\Pr(E_2) = \Pr(E_1)\Pr(E_2)$$

che mostra la seguente relazione

$$E_1 \text{ è indipendente da } E_2 \iff \Pr(E_1 \cap E_2) = \Pr(E_1)\Pr(E_2)$$

A partire da quest'ultima osservazione, si fornisce una definizione alternativa di indipendenza, estendendola ad un numero di eventi maggiore di 2.

■ Definizione (Eventi indipendenti):

Due eventi E_1 ed E_2 sono indipendenti se e solo se

$$\mathbf{Pr}(E_1 \cap E_2) = \mathbf{Pr}(E_1) \cdot \mathbf{Pr}(E_2)$$

Più generalmente, gli eventi E_1, E_2, \dots, E_k sono mutualmente indipendenti se e solo se, per un qualsiasi sottoinsieme $\mathbf{I} \subseteq [1, k]$,

$$\mathbf{Pr}\left(\bigcap_{i \in \mathbf{I}} E_i\right) = \prod_{i \in \mathbf{I}} \mathbf{Pr}(E_i)$$

In conclusione, dalla definizione di probabilità condizionata, se E_1 è indipendente da E_2 , allora

$$\mathbf{Pr}(E_2|E_1) = \frac{\mathbf{Pr}(E_1 \cap E_2)}{\mathbf{Pr}(E_1)} = \frac{\mathbf{Pr}(E_1)\mathbf{Pr}(E_2)}{\mathbf{Pr}(E_1)} = \mathbf{Pr}(E_2)$$

ossia

$$E_1 \text{ è indipendente da } E_2 \iff E_2 \text{ è indipendente da } E_1$$

Alla luce di quanto appena enunciato, si ha il seguente lemma

■ Lemma (Indipendenza):

Per mostrare che due eventi E_1 ed E_2 sono indipendenti, basta dimostrare una delle seguenti uguaglianze:

$$\mathbf{Pr}(E_1|E_2) = \mathbf{Pr}(E_1)$$

$$\mathbf{Pr}(E_2|E_1) = \mathbf{Pr}(E_2)$$

$$\mathbf{Pr}(E_1 \cap E_2) = \mathbf{Pr}(E_1)\mathbf{Pr}(E_2)$$

dove E_1 può essere sostituito con E_1^c ed E_2 con E_2^c (o entrambi). Se una di queste uguaglianze risulta essere vera, allora sono vere tutte le restanti. Se due eventi non sono indipendenti, allora sono chiamati *dipendenti*.

La definizione di eventi indipendenti può essere estesa a più eventi.

■ Teorema (Indipendenza di due o più eventi):

Gli eventi E_1, E_2, \dots, E_m sono detti indipendenti se

$$\mathbf{Pr}(E_1 \cap E_2 \cap \dots \cap E_m) = \mathbf{Pr}(E_1)\mathbf{Pr}(E_2) \cdots \mathbf{Pr}(E_m)$$

Ciò risulta vero anche sostituendo un numero arbitrario di eventi E_1, \dots, E_m con il rispettivo complemento all'intero della formula.

Variabili aleatorie discrete

Quando si studia un evento probabilistico, si è spesso interessati a dei valori associati agli eventi, piuttosto che agli eventi stessi. Una qualsiasi funzione dallo spazio di campionamento ai numeri reali prende il nome di **variabile aleatoria**.

Definizione (Variabile aleatoria):

Una variabile aleatoria X su uno spazio di campionamento Ω è una funzione su Ω a valori reali (e quindi misurabile), ossia $X : \Omega \rightarrow \mathbb{R}$. Una variabile aleatoria discreta è una variabile aleatoria che assume solo un numero finito o numerabile di valori.

Per una variabile aleatoria discreta X e un valore reale a , l'evento " $X = a$ " include tutti gli eventi di Ω in cui X assume il valore a . " $X = a$ " rappresenta quindi l'insieme

$$\{ s \in \Omega \mid X(s) = a \}$$

Si definisce la probabilità di tale evento come:

$$\Pr(X = a) = \sum_{s \in \Omega : X(s) = a} \Pr(s)$$

E' possibile estendere la definizione di indipendenza tra eventi alle variabili aleatorie.

Definizione (Variabili aleatorie indipendenti)

Due v.a. X ed Y sono indipendenti se e solo se

$$\Pr((X = x) \cap (Y = y)) = \Pr(X = x) \cdot \Pr(Y = y)$$

Analogamente, le variabili aleatorie X_1, X_2, \dots, X_k sono mutualmente indipendenti se e solo se, $\forall \mathbf{I} \subseteq [1, k]$ e $\forall x_i$ con $i \in \mathbf{I}$

$$\Pr\left(\bigcap_{i \in \mathbf{I}} (X_i = x_i)\right) = \prod_{i \in \mathbf{I}} \Pr(X_i = x_i)$$

Definizione (k -wise indipendenza)

Sia $W = \{X_1, \dots, X_n\}$ un insieme di n variabili aleatorie. Si dice che tale insieme è k -wise indipendente, per $k \leq n$, se e solo se

$$\Pr\left(\bigwedge_{j=1}^k X_{i_j} = x_{i_j}\right) = \prod_{j=1}^k \Pr(X_{i_j} = x_{i_j})$$

per ogni sottoinsieme $\{X_{i_1}, \dots, X_{i_k}\} \subseteq W$ di k variabili aleatorie. Per $k = n$, si dice che tale insieme W contiene variabili aleatorie completamente indipendenti.

Una caratteristica di base di una variabile aleatoria è il suo **valore atteso** (o media). La media di una variabile aleatoria è la media pesata dei valori che assume, dove ogni valore è pesato dalla probabilità che la variabile aleatoria assuma tale valore.

Definizione (Valore medio)

Il valore atteso di una variabile aleatoria discreta X , denotato $\mathbf{E}[X]$, è dato da

$$\mathbf{E}[X] = \sum_i i \Pr(X = i)$$

dove la sommatoria è definita su tutti i valori nel range di X , ossia i valori che questa v.a. può assumere.

Una proprietà fondamentale del valore atteso, che può semplificare in maniera significativa il suo calcolo, è la **linearità del valore atteso**. Da questa proprietà, si deriva che il valore atteso della somma di v.a. è uguale alla somma dei loro valori attesi

Teorema (Linearità del valore atteso)

Per ogni sequenza finita di v.a. discrete X_1, X_2, \dots, X_n con valore atteso finito, si ha che

$$\mathbf{E}\left[\sum_{i=1}^n X_i\right] = \sum_{i=1}^n \mathbf{E}[X_i]$$

Si mostrano ora delle proprietà del valore atteso

Teorema

Siano X ed Y due variabili aleatorie indipendenti. Allora

$$\mathbf{E}[X \cdot Y] = \mathbf{E}[X] \cdot \mathbf{E}[Y]$$

Nota

Se X ed Y non sono indipendenti, allora

$$\mathbf{E}[X \cdot Y] \neq \mathbf{E}[X] \cdot \mathbf{E}[Y]$$

Sia X una variabile aleatoria e \mathcal{E} un evento. Il valore atteso condizionale di X condizionato su \mathcal{E} è definito come

$$\mathbf{E}[X|\mathcal{E}] = \sum_x x \cdot \mathbf{Pr}[X = x|\mathcal{E}]$$

dove x assume valori nel range di X .

Il teorema della probabilità totale implica il seguente lemma

📘 Lemma (Legge delle aspettative iterate)

Se gli eventi B_i , per $i = 1, \dots, k$ sono una partizione dello spazio di campionamento, allora per ogni variabile aleatoria X :

$$\mathbf{E}[X] = \sum_{i=1}^k \mathbf{Pr}[B_i] \mathbf{E}[X|B_i]$$

La varianza misura di quanto il valore di una variabile aleatoria si discosta globalmente dal suo valore atteso.

📘 Definizione (Varianza e deviazione standard)

La varianza di una v.a. X è definita come

$$Var[X] = \mathbf{E}[(X - \mathbf{E}[X])^2] = \mathbf{E}[X^2] - (\mathbf{E}[X])^2$$

Alternativamente, può essere definita come segue:

Per una variabile aleatoria discreta, la varianza è calcolata sommando, il prodotto del quadrato della differenza tra il valore assunto dalla variabile aleatoria e il suo valore atteso con la probabilità che tale variabile assuma quei valori, per tutti i valori che può assumere la variabile aleatoria. In formule, sia $X = 1, \dots, n$ e $\mu = \mathbf{E}[X]$, allora

$$Var[X] = \sum_{k=1}^n (k - \mu)^2 \mathbf{Pr}[X = k]$$

Inoltre, se $X = 1, 2, \dots$, si ha che

$$Var[X] = \sum_{k=1}^{\infty} (k - \mu)^2 \mathbf{Pr}[X = k]$$

La deviazione standard di una v.a. X è

$$\sigma[X] = \sqrt{Var[X]}$$

💡 Corollario

Siano X e Y variabili aleatorie indipendenti. Allora

$$Var[X + Y] = Var[X] + Var[Y]$$

✍ Nota

Se X ed Y non sono indipendenti, allora

$$Var[X + Y] \neq Var[X] + Var[Y]$$

📘 Teorema

Siano X_1, X_2, \dots, X_n variabili aleatorie mutualmente indipendenti. Allora

$$Var\left[\sum_{i=1}^n X_i\right] = \sum_{i=1}^n Var[X_i]$$

Distribuzioni di probabilità

Distribuzione binomiale

Si supponga di eseguire un esperimento avente successo con probabilità p e fallimento con probabilità $1 - p$. Sia Y una variabile aleatoria tale che

$$Y = \begin{cases} 1 & \text{se l'esperimento ha successo} \\ 0 & \text{altrimenti} \end{cases}$$

La variabile Y prende il nome di variabile aleatoria *binomiale binaria*, o variabile aleatoria *Bernoulliana*. Si osserva che, per una variabile aleatoria binaria si ha che

$$\mathbf{E}[Y] = p \cdot 1 + (1 - p) \cdot 0 = p = \mathbf{Pr}(Y = 1)$$

Si consideri ora una sequenza di n esperimenti **indipendenti**, ciascuno con probabilità di successo p (\Rightarrow probabilità di fallimento $(1 - p)$). Tale sequenza di esperimenti prende il nome di *processo di Bernoulli*, mentre ogni esperimento prende il nome di *prova di Bernoulli*. Sia X la variabile aleatoria che conta il numero di successi negli n esperimenti, allora X ha una **distribuzione binomiale**.

■ Definizione (Distribuzione binomiale):

Una variabile aleatoria binomiale X con parametri n e p , denotata da $X \sim \text{Bin}(n, p)$, è definita dalla seguente distribuzione di probabilità su $j = 0, 1, \dots, n$:

$$\mathbf{Pr}(X = j) = \binom{n}{j} p^j (1 - p)^{n-j}$$

Ossia, X è uguale a j quando vi sono esattamente j successi e $n - j$ fallimenti in n esperimenti indipendenti, ciascuno dei quali ha successo con probabilità p .

■ Osservazione

$$\sum_{j=0}^n \mathbf{Pr}(X = j) = 1$$

In accordo con la definizione di funzione di probabilità.

E' possibile calcolare, partendo dalla definizione, il valore atteso di una variabile aleatoria binaria X con distribuzione binomiale:

$$\begin{aligned} \mathbf{E}[X] &= \sum_{j=0}^n j \binom{n}{j} p^j (1 - p)^{n-j} \\ &= \sum_{j=0}^n j \frac{n!}{j! (n - j)!} p^j (1 - p)^{n-j} \\ &= \sum_{j=1}^n \frac{n!}{(j - 1)! (n - j)!} p^j (1 - p)^{n-j} \\ &= np \sum_{j=1}^n \frac{(n - 1)!}{(j - 1)! ((n - 1) - (j - 1))!} p^{j-1} (1 - p)^{((n-1)-(j-1))} \\ &\stackrel{(1)}{=} np \sum_{k=0}^{n-1} \frac{(n - 1)!}{k! ((n - 1) - k)!} p^k (1 - p)^{((n-1)-k)} \\ &= np \sum_{k=0}^{n-1} \binom{n - 1}{k} p^k (1 - p)^{(n-1)-k} \\ &\stackrel{(2)}{=} np \end{aligned}$$

Dove:

- in (1) si è posto $k = j - 1$
- in (2) si è fatto riferimento al teorema binomiale:

$$(x + y)^n = \sum_{k=0}^n \binom{n}{k} x^k y^{n-k}$$

con $x = p$ e $y = 1 - p$.

Si fa presente ora come, sfruttando la linearità del valore atteso, quest'ultimo può essere calcolato in maniera molto più semplice.

Se $X \sim \text{Bin}(n, p)$, allora $X = \# \text{ successi in } n \text{ prove}$, dove ogni prova ha probabilità p di avere successo. Si definisce, per $i = 1, \dots, n$

$$X = \begin{cases} 1 & \text{se l'esperimento ha successo} \\ 0 & \text{altrimenti} \end{cases}$$

Allora $\mathbf{E}[X_i] = p \cdot 1 + 0 \cdot (1 - p) = p = \mathbf{Pr}(X_i = 1)$.

Inoltre, si ha che $X = \sum_{i=1}^n X_i$, allora, per la linearità del valore atteso

$$\mathbf{E}[X] = \mathbf{E}\left[\sum_{i=1}^n X_i\right] = \sum_{i=1}^n \mathbf{E}[X_i] = np$$

La varianza di una variabile aleatoria binomiale X con parametri n e p può essere determinata calcolando $\mathbf{E}[X^2]$ come segue

$$\begin{aligned} \mathbf{E}[X^2] &= \sum_{j=0}^n \binom{n}{j} p^j (1-p)^{n-j} j^2 \\ &= \sum_{j=0}^n \frac{n!}{(n-j)! j!} p^j (1-p)^{n-j} ((j^2 - j) + j) \\ &= \sum_{j=0}^n \frac{n! (j^2 - j)}{(n-j)! j!} p^j (1-p)^{n-j} + \sum_{j=0}^n \frac{n! j}{(n-j)! j!} p^j (1-p)^{n-j} \\ &= n(n-1)p^2 \sum_{j=2}^n \frac{(n-2)!}{(n-j)!(j-2)!} p^{j-2} (1-p)^{n-j} \\ &\quad + np \sum_{j=1}^n \frac{(n-1)!}{(n-j)!(j-1)!} p^{j-1} (1-p)^{n-j} \\ &= n(n-1)p^2 + np. \end{aligned}$$

Dove le sommatorie sono state semplificate facendo riferimento al teorema binomiale:
 $(x + y)^n = \sum_{k=0}^n \binom{n}{k} x^{n-k} y^k = \sum_{k=0}^n \binom{n}{k} x^k y^{n-k}$. Si conclude che

$$\begin{aligned} \text{Var}[X] &= \mathbf{E}[X^2] - (\mathbf{E}[X])^2 \\ &= n(n-1)p^2 + np - n^2 p^2 \\ &= np - np^2 \\ &= np(1-p) \end{aligned}$$

La varianza può essere alternativamente derivata facendo riferimento all'indipendenza. Una variabile aleatoria binomiale X può essere rappresentata come la somma di n prove di

Bernoulli indipendenti, ciascuna avente successo con probabilità p . Tale prova di Bernoulli Y ha varianza

$$\mathbf{E} [(Y - \mathbf{E}[Y])^2] = p(1-p)^2 + (1-p)(-p)^2 = p - p^2 = p(1-p).$$

E per la linearità della varianza, si ha che la varianza di X è quindi $np(1-p)$.

Distribuzione geometrica

Si supponga di lanciare una moneta sino a quando esce testa. La distribuzione del numero di lanci è un esempio di distribuzione geometrica. In generale, si ha una distribuzione geometrica quando si eseguono una sequenza di eventi indipendenti sino al primo successo, dove ogni prova ha successo con probabilità p .

Definizione (Distribuzione geometrica)

Una variabile aleatoria geometrica X con probabilità p è data dalla seguente distribuzione di probabilità su $n = 1, 2, \dots$

$$\mathbf{Pr}(X = n) = (1-p)^{n-1}p$$

Per la v.a. geometrica X , per essere uguale ad n , devono esserci $n - 1$ fallimenti seguiti da un successo. Ovviamente

Osservazione

$$\sum_{n \geq 1} \mathbf{Pr}(X = n) = 1$$

In accordo con la definizione di funzione di probabilità.

Le variabili geometriche sono "prive di memoria", nel senso che la probabilità di ottenere un successo dopo n prove è indipendente dal numero di fallimenti ottenuti in precedenza. Informalmente, si possono ignorare gli insuccessi passati, perché non cambiano la distribuzione del numero di prove future sino al primo successo. Formalmente

Lemma

Per una variabile aleatoria geometrica X con parametro p e per $n > 0$

$$\mathbf{Pr}(X = n + k \mid X > k) = \mathbf{Pr}(X = n)$$

Si vede ora come calcolare il valore atteso di una variabile aleatoria avente distribuzione geometrica. Quando una variabile aleatoria assume valori in \mathbb{N} , si ha una formula alternativa per calcolare il suo valore atteso

Lemma

Sia X una variabile aleatoria discreta che assume solo valori interi non negativi. Allora

$$\mathbf{E}[X] = \sum_{i=1}^{\infty} \mathbf{Pr}(X \geq i)$$

Si ha che per una variabile aleatoria geometrica X con parametro p

$$\mathbf{Pr}(X \geq i) = \sum_{n=i}^{\infty} (1-p)^{n-1} p = (1-p)^{i-1}$$

Allora

$$\begin{aligned} \mathbf{E}[X] &= \sum_{i=1}^{\infty} \mathbf{Pr}(X \geq i) = \sum_{i=1}^{\infty} (1-p)^i - 1 \\ &= \frac{1}{1 - (1-p)} = \frac{1}{p} \end{aligned}$$

Discostamento dal valore atteso

Si presentano delle tecniche per stimare il discostamento di una variabile aleatoria dal suo valore atteso, ossia la probabilità che una variabile aleatoria assuma valori lontani dal suo valore atteso. Questi bounds forniscono una stima della probabilità di insuccesso di un algoritmo, e permettono di stabilire bounds con alta probabilità sul running time.

Diseguaglianza di Markov

Theorema

Sia X una variabile aleatoria che assume solo valori non negativi. Allora, $\forall a > 0$ vale

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

Dimostrazione

Per $a > 0$, sia

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

Si osserva che, essendo $X \geq 0$, vale

$$I \leq \frac{X}{a}$$

Essendo I una variabile aleatoria binaria

$$\mathbf{E}[I] = \mathbf{Pr}[I = 1] = \mathbf{Pr}[X \geq a]$$

Facendo riferimento alla diseguaglianza precedente, si ha

$$\mathbf{Pr}[X \geq a] = \mathbf{E}[I] \leq \mathbf{E}\left[\frac{X}{a}\right] = \frac{\mathbf{E}[X]}{a}$$

Diseguaglianza di Chebyshev

Teorema

Per ogni $a > 0$

$$\mathbf{Pr}[|X - \mathbf{E}[X]| \geq a] \leq \frac{\text{Var}[X]}{a^2}$$

Analisi di algoritmi probabilistici

Verifica di identità polinomiali

Si hanno due polinomi di grado d , siano questi $F(x)$ e $G(x)$.

Il primo polinomio $F(x)$ è rappresentato nella forma di prodotto di monomi

$$F(x) = \prod_{i=1}^d (x - a_i)$$

mentre $G(x)$ è rappresentato mediante la sua forma canonica

$$G(x) = \sum_{i=0}^d c_i x^i$$

Dati questi due polinomi, si vuole verificare l'identità

$$F(x) \equiv G(x)$$

Per risolvere tale problema, è possibile convertire $F(x)$ nella sua forma canonica, moltiplicando consecutivamente l' i -esimo monomio con il prodotto dei primi $i - 1$ monomi precedenti. Così facendo, risulta sufficiente verificare che i coefficienti di ciascuna variabile

siano uguali per i due polinomi, ovvero, siano a_i e c_i i coefficienti del letterale x^i di grado i per $F(x)$ e $G(x)$ rispettivamente, allora

$$F(x) \equiv G(x) \iff (\forall i = 0, \dots, d [a_i = c_i])$$

Tale operazione richiede $\Theta(d^2)$ moltiplicazioni di coefficienti, ciascuna eseguibile in tempo costante.

Si vuole progettare un algoritmo probabilistico che risolva il problema in maniera efficiente.

Sia d il grado massimo, ossia il più grande esponente di x , in $F(x)$ e $G(x)$. L'algoritmo in questione sceglie un intero r uniformly at random nel range $\{1, \dots, 100d\}$. Calcola poi i valori $F(r)$ e $G(r)$, e li confronta. Se $F(r) = G(r)$, allora l'algoritmo dichiara che i due polinomi sono equivalenti, altrimenti, se $F(r) \neq G(r)$, dichiara che non lo sono.

Si assume che la generazione di un intero r casuale in $\{1, \dots, 100d\}$ richieda tempo costante.

Il calcolo di $F(r)$ e di $G(r)$ può essere effettuato in tempo $\mathcal{O}(d)$, che risulta essere più veloce del calcolo della forma canonica di $F(x)$. Tale miglioramento in termini di efficienza, comporta però la possibilità che l'algoritmo restituisca una risposta incorretta.

Si studiano quindi i possibili scenari in cui può trovarsi l'algoritmo:

- Se $F(x) \equiv G(x)$ allora l'algoritmo restituisce *sempre* la risposta corretta, essendo che per ogni valore di r , $F(r) = G(r)$.
- Se $F(x) \not\equiv G(x)$ e $F(r) \neq G(r)$, allora l'algoritmo restituisce la risposta corretta, essendo che ha trovato un r tale per cui $F(x)$ e $G(x)$ non assumono lo stesso valore. In particolar modo, essendo che $F(x) \equiv G(x)$ se e solo se $F(r) = G(r)$ per ogni r , ogni qual volta l'algoritmo trova un r tale per cui $F(r) \neq G(r)$, allora sicuramente $F(x) \not\equiv G(x)$, ossia, i due polinomi non sono equivalenti.
- Se $F(x) \not\equiv G(x)$ e $F(r) = G(r)$ allora l'algoritmo restituisce la risposta sbagliata. E' possibile quindi che venga selezionato un r tale per cui l'algoritmo decida che i due polinomi siano uguali, quando in realtà non lo sono. Affinché ciò avvenga, r deve essere una radice dell'equazione $F(x) - G(x) = 0$. Il grado del polinomio $F(x) - G(x) = 0$ è al più d , e per il teorema fondamentale dell'algebra, un polinomio di grado d non ha più di d radici. Quindi, se $F(x) \not\equiv G(x)$, ci sono al più d valori nel range $\{1, \dots, 100d\}$ tali per cui $F(r) = G(r)$. Essendovi $100d$ valori in tale range, la probabilità che l'algoritmo scelga un valore di questo tipo, e che quindi restituisca una risposta sbagliata è al più $1/100$.

Quindi, schematicamente, analizzando i casi in input:

- Se $F(x) \equiv G(x)$ l'algoritmo risponde correttamente.
 - Se $F(x) \not\equiv G(x)$ l'algoritmo può rispondere in maniera incorretta.
- Mentre valutando gli esiti in output si ha:
- Se l'algoritmo calcola $F(r) \neq G(r)$ allora sicuramente $F(x) \not\equiv G(x)$, e quindi risponde correttamente.
 - Se l'algoritmo calcola $F(r) = G(r)$ non necessariamente $F(x) \equiv G(x)$, e quindi può aver restituito una risposta sbagliata.

Si formalizza a probabilità di errore. Sia \mathcal{E} l'evento

$$\mathcal{E} = \text{L'algoritmo sbaglia}$$

Gli elementi dell'insieme corrispondente all'evento \mathcal{E} sono le radici del polinomio $F(x) - G(x)$ contenute in $\{1, \dots, 100d\}$. Non essendo queste più di d , si ha che \mathcal{E} include al più d eventi semplici. Quindi

$$\Pr[\mathcal{E}] \leq \frac{d}{100d} = \frac{1}{100}$$

Un approccio per aumentare la probabilità che l'algoritmo risponda correttamente è quello di eseguirlo più volte, utilizzando valori casuali diversi per testare l'identità. Questo perché, essendo che l'algoritmo può effettuare un errore da un solo lato, ossia può aver sbagliato solamente se restituisce che i due polinomi sono equivalenti, se almeno uno degli r scelti è tale per cui $F(r) \neq G(r)$, allora sicuramente i due polinomi sono diversi. Quindi, fissato un numero di round per i quali iterare l'algoritmo, se per ciascuno di essi risulta $F(r) = G(r)$ con r selezionato ad ogni round *u.a.r.* da $\{1, \dots, 100d\}$, si può aver maggior confidenza che l'identità risulti essere verificata.

Il campionamento di r dall'intervallo può avvenire in due modalità: *con reinserimento* o *senza reinserimento*. Nel primo caso, non si tiene traccia dei valori di r selezionati in precedenza. Di conseguenza, lo stesso valore per r può essere assunto più volte. Nel secondo caso, una volta che un valore per r è stato scelto, tale valore viene escluso dalla selezione casuale nei round successivi, dove questa avviene *u.a.r.* tra i valori dell'intervallo non ancora selezionati.

Sia quindi k il numero rounds, ossia il numero di volte per cui viene iterato l'algoritmo, e siano i due polinomi *non* equivalenti.

Si consideri il caso con reinserimento. Essendo che l'insieme da cui viene selezionato il valore di r rimane invariato ad ogni round, si ha che la selezione di ciascun valore è indipendente dalle altre. Sia \mathcal{E}_i l'evento

$$\mathcal{E}_i = \text{Al round } i \text{ viene scelto } r_i \text{ tale che } F(r_i) - G(r_i) = 0$$

La probabilità che l'algoritmo restituisca la risposta sbagliata è data da

$$\Pr[\mathcal{E}_1 \cap \mathcal{E}_2 \cap \dots \cap \mathcal{E}_k]$$

Essendo che $\Pr[\mathcal{E}_i] \leq 1/100$ e gli eventi $\mathcal{E}_1, \mathcal{E}_2, \dots, \mathcal{E}_k$ sono indipendenti, la probabilità che l'algoritmo restituisca la risposta sbagliata dopo k iterazioni è

$$\Pr[\mathcal{E}_1 \cap \mathcal{E}_2 \cap \dots \cap \mathcal{E}_k] = \prod_{i=1}^k \Pr[\mathcal{E}_i] \leq \prod_{i=1}^k \frac{d}{100d} = \left(\frac{1}{100}\right)^k$$

Si consideri il caso senza reinserimento. In questo scenario, la probabilità di scegliere un dato numero nell'intervallo è condizionata dagli eventi delle precedenti iterazioni.

Sia di nuovo \mathcal{E}_i l'evento

$$\mathcal{E}_i = \text{Al round } i \text{ viene scelto } r_i \text{ tale che } F(r_i) - G(r_i) = 0$$

La probabilità che l'algoritmo restituisca la risposta sbagliata è sempre data da

$$\Pr[\mathcal{E}_1 \cap \mathcal{E}_2 \cap \dots \cap \mathcal{E}_k]$$

In questo caso, gli eventi non risultano essere indipendenti tra loro. Applicando la definizione di probabilità condizionata, si ottiene

$$\Pr[\mathcal{E}_1 \cap \mathcal{E}_2 \cap \dots \cap \mathcal{E}_k] = \Pr[\mathcal{E}_k | \mathcal{E}_1 \cap \mathcal{E}_2 \cap \dots \cap \mathcal{E}_{k-1}] \cdot \Pr[\mathcal{E}_1 \cap \mathcal{E}_2 \cap \dots \cap \mathcal{E}_{k-1}]$$

Ripetendo lo stesso procedimento si ha che

$$\Pr[\mathcal{E}_1 \cap \mathcal{E}_2 \cap \dots \cap \mathcal{E}_k] = \Pr[\mathcal{E}_1] \cdot \Pr[\mathcal{E}_2 | \mathcal{E}_1] \cdot \Pr[\mathcal{E}_3 | \mathcal{E}_1 \cap \mathcal{E}_2] \cdots \Pr[\mathcal{E}_k | \mathcal{E}_1 \cap \mathcal{E}_2 \cap \dots \cap \mathcal{E}_{k-1}]$$

Si vuole dare un bound a

$$\Pr[\mathcal{E}_j | \mathcal{E}_1 \cap \mathcal{E}_2 \cap \dots \cap \mathcal{E}_{j-1}]$$

Si supponga che nelle prime $j - 1$ iterazioni si abbia ottenuto sempre $F(r) - G(r) = 0$, ossia che siano state selezionate, essendo $F(x) \not\equiv G(x)$, sempre radici della differenza tra i due polinomi.

Essendovi al più d valori r tali per cui $F(r) - G(r) = 0$, se le iterazioni che vanno da 1 a $j - 1 < d$ ne hanno trovati $j - 1$, allora, essendo che questi non vengono reinseriti nell'intervallo per la selezione, si ha che alla j -esima iterazione si hanno al più $d - (j - 1)$ radici rimanenti tra tutte le $100d - (j - 1)$ possibili scelte.

Allora

$$\Pr[\mathcal{E}_j | \mathcal{E}_1 \cap \mathcal{E}_2 \cap \dots \cap \mathcal{E}_{j-1}] \leq \frac{d - (j - 1)}{100d - (j - 1)}$$

e quindi è possibile dare il seguente bound sulla probabilità che l'algoritmo dia una risposta sbagliata dopo $k \leq d$ iterazioni

$$\Pr[\mathcal{E}_1 \cap \mathcal{E}_2 \cap \dots \cap \mathcal{E}_k] \leq \prod_{j=1}^k \frac{d - (j - 1)}{100d - (j - 1)} \leq \left(\frac{1}{100} \right)^k$$

ed essendo che per $j > 1$

$$\frac{d - (j - 1)}{100d - (j - 1)} < \frac{d}{100d}$$

si ha che la selezione senza reinserimento garantisce una probabilità minore che l'algoritmo sbagli.

Si osserva esplicitamente che, se si prendono $d + 1$ campioni senza reinserimento, e i due polinomi non sono equivalenti, si ottiene sicuramente un r tale per cui $F(r) - G(r) \neq 0$.

Quindi in $d + 1$ iterazioni si può garantire in output la risposta corretta. Ma il calcolo del polinomio a $d + 1$ punti richiede tempo $\Theta(d^2)$ utilizzando l'approccio standard. Tale opzione non risulta essere quindi più efficiente dell'approccio deterministico.

Verifica del prodotto di matrici binarie

Si vede un esempio di algoritmo probabilistico che, a discapito della sicurezza sulla produzione di una risposta corretta in output, impiega minor tempo per l'elaborazione di tale risposta.

- **Input:** $A, B, C \in \{0, 1\}^{n \times n}$
- **Output:** True se $A \cdot B = C \pmod{2}$, False altrimenti

Si ricorda il metodo deterministico (banale) per verificare il prodotto di due matrici:

- Si calcola $A \cdot B = C'$
- Si verifica $C' = C$

Il calcolo del prodotto di due matrici impiega tempo $\Theta(n^3)$. Questo perché, eseguendo il prodotto righe per colonne, per il calcolo di un'entrata di C' sono necessarie n moltiplicazioni. Quindi, per calcolare le n^2 entrate di C' sono necessarie $\Theta(n^3)$ operazioni. Esistono anche algoritmi che impiegano tempo $\Theta(n^{2.37})$.

Si vede ora un algoritmo probabilistico che impiega tempo $\mathcal{O}(n^2 \log n)$.

Osservazione

Se $AB = C$, allora, dato un qualunque vettore $r = [r_1 \ r_2 \ \dots \ r_n] \in \{0, 1\}^n$, vale che $ABr = Cr$

Osservazione

Il prodotto tra una matrice ed un vettore impiega tempo $\Theta(n^2)$

Se si prende un vettore $r = [r_1 \ r_2 \ \dots \ r_n]^\top \in \{0, 1\}^n$, si può calcolare ABr nella seguente maniera:

- Si calcola prima Br

$$B \cdot r = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix} \begin{bmatrix} r_{11} \\ r_{21} \\ \vdots \\ r_{n1} \end{bmatrix} = \begin{bmatrix} r_{B1} \\ r_{B2} \\ \vdots \\ r_{Bn} \end{bmatrix} = r_B$$

Calcolare r_B richiede n moltiplicazioni per n entrate \implies costo $\Theta(n^2)$.

- Si calcola poi $A(Br) = Ar_B$

$$A \cdot r_B = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} r_{B1} \\ r_{B2} \\ \vdots \\ r_{Bn} \end{bmatrix} = \begin{bmatrix} r_{AB1} \\ r_{AB2} \\ \vdots \\ r_{ABn} \end{bmatrix} = r_{AB}$$

Con costo $\Theta(n^2)$ per l'argomentazione precedente.

Quindi, l'algoritmo sceglie un vettore $r \in \{0, 1\}^n$ u.a.r e calcola $A(B \cdot r)$ e $C \cdot r$.

Essendo che $\forall r \in \{0, 1\}^n$, se $AB = C$ allora $ABr = Cr$, se $A(Br) \neq Cr$ allora sicuramente $AB \neq C$, portando l'algoritmo a restituire **False**. Restituisce **True** altrimenti.

MatrixMult(A, B, C):

1. Scegli $r \in \{0, 1\}^n$ u.a.r
 2. $r_B = B \cdot r$
 3. $r_{AB} = A \cdot r_B$
 4. $r_C = C \cdot r$
 5. **if** $r_{AB} \neq r_C$:
 6. **return False**
 6. **return True**
-

■ Osservazione

La scelta di $r \in \{0, 1\}^n$ u.a.r ha costo $\Theta(n)$. In $\{0, 1\}^n$ si hanno 2^n vettori, ciascuno avente la stessa probabilità di $1/2^n$ di essere scelto. Per generare r basta chiamare la funzione

$$\text{urand}() = \begin{cases} 1 & \text{w.p. } 1/2 \\ 0 & \text{w.p. } 1/2 \end{cases}$$

per ciascuna delle n entrate di r .

Si è osservato precedentemente che se $AB = C$, allora $r_{AB} = r_C$. Di conseguenza, se l'algoritmo restituisce **False**, allora sicuramente $AB \neq C$.

Inoltre,

■ Osservazione

Se $AB = C$ allora l'algoritmo restituisce correttamente **True**.

Il fatto che se l'algoritmo restituisce **False** implica che $AB \neq C$, non implica che se $AB \neq C$ allora l'algoritmo restituisce correttamente **False**.

Osservazione

Se $AB \neq C$, allora non è detto che l'algoritmo restituisca correttamente **False**.

Se $AB \neq C$ potrebbe verificarsi che

$$r_{AB} = r_C \quad \text{oppure} \quad r_{AB} \neq r_C$$

a seconda della scelta di r . Ad esempio, se $AB \neq C$ e $r = \mathbf{0}$, allora vale sempre $ABr = Cr$, portando l'algoritmo a restituire incorrettamente **True**.

Ricapitolando:

- Se $AB = C$ allora $\forall r \in \{0, 1\}^n$ vale
 $(ABr = ACr) \implies$ l'algoritmo restituisce **True** correttamente.
- Se $r_{AB} \neq r_C$ allora necessariamente
 $(AB \neq C) \implies$ l'algoritmo restituisce **False** correttamente.
- Se $AB \neq C$ non è detto che $r_{AB} \neq r_C \implies$ l'algoritmo può sbagliare restituendo **True**.
- Se $r_{AB} = r_C$ non è detto che $AB = C \implies$ l'algoritmo può sbagliare restituendo **True**.

Alla luce di queste considerazioni ci si chiede quindi, nel caso in cui $AB \neq C$, con quale probabilità l'algoritmo sbaglia?

Osservazione

Non ha senso chiedersi "Se l'algoritmo restituisce **True**, qual è la probabilità che $AB = C$? ", perché questo evento non ha probabilità: o è vero o è falso.

Si vuole quindi dare una delimitazione alla probabilità che si verifichi questo evento, ossia dare un upper bound alla probabilità di errore.

Osservazione

Se $AB \neq C$, sia $D = (AB - C)$. Sicuramente $D \neq \mathbf{0}$, perché almeno un elemento deve essere uguale ad 1. Si può definire quindi D come

$$D = (d_{i,j})_{i,j:1,\dots,n}, \exists i, j : d_{i,j} = 1$$

Si sfrutta l'entrata $d_{i,j} = 1$ per studiare la probabilità che l'algoritmo sbagli.

L'algoritmo sbaglia nel caso in cui $AB \neq C$ e r è stato scelto tale che $r_{AB} = r_C$. Essendo $D = AB - C$ si ha che $D \cdot r = ABr - Cr = r_{AB} - r_C = 0$.

Sia $d_{i,j}$ un elemento non nullo in D . Si considera l' i -esimo elemento del vettore Dr .

Tale elemento è ottenuto moltiplicando r con l' i -esima riga di D . Quindi

$$r_D[i] = \sum_{k=1}^n d_{i,k} \cdot r[k] \pmod{2}$$

Si tira fuori dalla sommatoria l'elemento uguale ad 1.

$$d_{i,j} \cdot r[j] + \sum_{k=1, k \neq j}^n d_{i,k} \cdot r[k] = r_D[i]$$

Allora, ricordando che $r_D[i] = 0$ e $d_{i,j} = 1$, si ha

$$r[j] = - \sum_{k=1, k \neq j}^n d_{i,k} \cdot r[k]$$

Se la sommatoria è uguale a 0, allora $r[j]$ deve essere uguale a 0 (lo è con probabilità 1/2).

Se la sommatoria è uguale ad 1, allora $r[j]$ deve essere uguale ad 1 (anch'esso con probabilità 1/2).

Quindi

$$r[j] = - \sum_{k=1, k \neq j}^n d_{i,k} \cdot r[k] = 0 \quad \text{w.p. } \frac{1}{2}$$

Dunque, la probabilità che $ABr = Cr$ quando $AB \neq C$ è al più 1/2.

Questa idea viene chiamata *principle of deferred decisions*: quando vi sono diverse variabili aleatorie, come le entrate $r[i]$ in r , risulta spesso utile considerare alcune di esse come valori fissati, mentre le restanti rimangono aleatorie. Formalmente, questo corrisponde ad effettuare un condizionamento sui valori rivelati, ossia, quando è noto il valore assunto da alcune variabili aleatorie, bisogna calcolare le probabilità condizionate su tali valori per il resto dell'analisi.

Con questa argomentazione abbiamo dimostrato il seguente teorema

Teorema

Se $AB \neq C$, allora

$$\Pr[\text{MatrixMult} = \text{True}] \leq \frac{1}{2}$$

Si osserva esplicitamente che, più entry in D sono uguali ad 1, più è bassa la probabilità di errore.

Dunque l'algoritmo:

- Se restituisce `False`, allora sicuramente $AB \neq C$.
- Se restituisce `True`, allora $AB = C$ con probabilità $1/2$.

Si vuole sviluppare ora un algoritmo che sbagli con probabilità minore.

In particolar modo, vogliamo un algoritmo che risponda correttamente con **alta probabilità**.

Definizione (Alta probabilità)

Sia $\{E_n\}$ una successione di eventi che dipende da un parametro n .

Si dice che E_n avviene con alta probabilità se $\exists n_0 \in \mathbb{N}, \exists c > 0$ tale per cui, $\forall n \geq n_0$:

$$\Pr[E_n] \geq 1 - \frac{1}{n^c}$$

Informalmente, se la probabilità che l'evento non succede decresce come l'inverso di un polinomio in n .

Per progettare un algoritmo con probabilità inferiore di sbagliare, è sufficiente ripetere l'algoritmo più volte.

MatrixMult2(A, B, C):

1. **for** t volte:
 2. Scegli $r \in \{0, 1\}^n$ u.a.r
 3. $r_B = B \cdot r$
 4. $r_{AB} = A \cdot r_B$
 5. $r_C = C \cdot r$
 6. **if** $r_{AB} \neq r_C$:
 7. **return** `False`
2. **return** `True`

Osservazione

La prima volta che si ottiene r tale per cui $r_{AB} \neq r_C$ allora si può restituire sicuramente **False**, perché se $AB = C$, lo spazio $\{0, 1\}^n$ sarebbe composto da **tutti** i vettori per cui l'algoritmo restituisce **True**.

■ Osservazione

Se $AB \neq C$

$$\Pr[\text{MatrixMult2} = \text{True}] \leq \frac{1}{2^t}$$

■ Osservazione

Il running time dell'algoritmo in funzione del parametro t è $\Theta(tn^2)$.

In base alla scelta di t si può quindi condizionare il running time dell'algoritmo, oltre alla probabilità di errore.

Si da ora una definizione di alta probabilità alternativa, legata maggiormente al concetto di algoritmo

■ Definizione (Alta probabilità)

Diciamo che un algoritmo probabilistico è corretto con alta probabilità se

$$\Pr[\text{Algoritmo sbaglia}] \leq \frac{1}{n^c}$$

dove n è la size dell'input e $c > 0$ costante.

Affinché **MatrixMult2** sia corretto è sufficiente scegliere $t = \Theta(\log n)$. Da ciò derivano i due seguenti teoremi

■ Teorema

Se $AB \neq C$ e $t = \Theta(\log n)$

$$\Pr[\text{MatrixMult2} = \text{True}] \leq \frac{1}{2^{\Theta(\log n)}}$$

■ Teorema

Se $t = \Theta(\log n)$, `MatrixMult2` ha running time $\Theta(n^2 \log n)$ ed è corretto con alta probabilità.

Random Quick Sort

Si ricorda il funzionamento del QuickSort deterministico.

In input si ha una lista di n elementi x_1, \dots, x_n presi da un universo totalmente ordinato.

L'algoritmo inizia scegliendo un *pivot* dalla lista, sia quest'ultimo x . Confronta poi ogni altro elemento della lista in input con x , dividendola in due sottoliste, una contenente gli elementi minori o uguali di x , e l'altra contenente gli elementi strettamente maggiori di x . Si osserva che se i confronti vengono eseguiti in ordine naturale, ossia da sinistra verso destra della lista, allora l'ordine degli elementi in ciascuna sottolista è lo stesso della lista iniziale.

L'algoritmo poi ordina ricorsivamente le sottoliste.

Nello scenario peggiore, il QuickSort impiega $\Omega(n^2)$ operazioni di confronto, e il running time del QuickSort è basato proprio sul numero di confronti che compie per ordinare una lista.

Ad esempio, si supponga che

$$x_1 = n, x_2 = n - 1, \dots, x_{n-1} = 2, x_n = 1$$

e che la regola per la scelta del pivot sia quella di selezionare il primo elemento della lista. La prima chiamata a QuickSort sceglie quindi come pivot $x_1 = n$, esegue $n - 1$ confronti e divide la lista in due sottoliste, una di size 0 e l'altra di size $n - 1$, dove gli elementi in quest'ultima hanno ordine dal più grande al più piccolo ($n - 1, n - 2, \dots, 2, 1$). Sceglie poi come pivot della sottolista $x_2 = n - 1$, ed operando come visto in precedenza, esegue $n - 2$ confronti.

Proseguendo in questa maniera, il numero di confronti impiegato dal QuickSort risulta essere

$$(n - 1) + (n - 2) + \dots + 2 + 1 = \sum_{i=0}^{n-1} i = \frac{n(n - 1)}{2}$$

Conseguendo nel running time di $\Omega(n^2)$ precedentemente espresso.

Essendo che il running time dell'algoritmo dipende dal numero di confronti effettuati, e che il numero di confronti effettuati dipende dalla scelta del pivot, una sua scelta ragionevole può conseguire in un numero minore di confronti.

Ad esempio, se il pivot dividesse le due sottoliste in maniera tale che la loro size risulti essere al più $\lceil n/2 \rceil$, il numero di confronti risulterebbe essere descrivibile dalla seguente relazione di ricorrenza

$$C(n) \leq 2 \cdot C(\lceil n/2 \rceil) + \Theta(n)$$

che se svolta porta a $C(n) = \mathcal{O}(n \log n)$.

Intuitivamente, ad ogni iterazione dell'algoritmo, esiste una lista "buona" di pivot, ovvero quella contenente gli elementi che, se selezionati come pivot, dividono la lista di elementi da ordinare in due sottoliste, le quali dimensioni si differenziano al più per un fattore costante. Si vuole far sì che l'algoritmo selezioni pivot buoni sufficientemente spesso, per garantire che questo termini velocemente.

■ Osservazione

Se l'algoritmo seleziona i pivot uniformly at random, è estremamente difficile che vengano scelti ripetutamente pivot non buoni.

Si fornisce ora una variante probabilistica del QuickSort. Sia I un vettore di numeri

RandomQS(I):

1. **If** $|I| \leq 1$:
 1. **return** I
 2. Scegli $a \in I$ u.a.r. e rimuovilo da I
 3. $L = []$, $R = []$
 4. **for each** $u \in I$:
 1. **if** $u \leq a$:
 1. Aggiungi u ad L
 2. **else**:
 1. Aggiungi u ad R
 5. **return** **RandomQS**(L), a , **RandomQS**(R)
-

Prima di effettuare l'analisi formale dell'algoritmo, si compiono le seguenti osservazioni

■ Osservazione

Due numeri vengono confrontati se e solo se uno di essi è scelto come pivot

■ Osservazione

Due numeri molto lontani vengono confrontati con probabilità bassa, mentre due numeri molto vicini vengono confrontati con probabilità alta.

Si analizza quest'ultima osservazione. Sia X la variabile aleatoria che conta il numero di confronti eseguiti dall'algoritmo

$$X = \# \text{ numero di confronti eseguiti da RandomQS}$$

Siano y_1, y_2, \dots, y_n gli n elementi di I ordinati, ossia

$$y_1 \leq y_2 \leq \dots \leq y_n$$

Per ogni $i, j = 1, \dots, n$ si definisce

$$X_{i,j} = \begin{cases} 1 & \text{se } y_i \text{ e } y_j \text{ vengono confrontati dall'algoritmo} \\ 0 & \text{altrimenti} \end{cases}$$

Si vuole valutare quando y_i ed y_j , con $i < j$ vengono confrontati. Si osserva inizialmente che y_i ed y_{i+1} vengono confrontati con probabilità 1, ossia $\Pr [X_{i,i+1} = 1] = 1$, questo perché finiranno necessariamente nella stessa lista quando avviene la separazione in due sottoliste, a meno che uno di essi non venga scelto come pivot, e anche in quel caso verranno comunque confrontati.

Siano y_i, y_j con $i < j$

![center][Qs1.png]

y_i ed y_j vengono confrontati quando, la prima volta che il pivot viene selezionato da quella lista, o è y_i o è y_j . Se per la prima volta l'elemento scelto come pivot è un elemento intermedio y_k con $i < k < j$, allora y_i ed y_j non verranno mai confrontati, perché verranno messi in liste distinte. Siccome il pivot è scelto *u.a.r.*, ciascun elemento ha la stessa probabilità di essere un pivot. Siano quindi

$$\mathcal{E}_1 = y_i \text{ viene scelto come pivot}$$

$$\mathcal{E}_2 = y_j \text{ viene scelto come pivot}$$

due eventi, dove si osserva che $\mathcal{E}_1 \cap \mathcal{E}_2 = \emptyset$, allora la probabilità che y_i o y_j sia un pivot è

$$\Pr [\mathcal{E}_1 \cup \mathcal{E}_2] = \Pr [\mathcal{E}_1] + \Pr [\mathcal{E}_2] = \frac{1}{j-i+1} + \frac{1}{j-i+1} = \frac{2}{j-i+1}$$

dove $j - i + 1$ è il numero di elementi nell'intervallo preso in analisi.

Osservazione

Se $j = i + 1$, ossia y_i ed y_j sono elementi consecutivi, allora

$$\frac{2}{j-i+1} = \frac{2}{2} = 1$$

■ Osservazione

Essendo $X_{i,j}$ una v.a. che assume solo valori 0 ed 1,

$$\mathbf{E}[X_{i,j}] = \mathbf{Pr}[X_{i,j} = 1]$$

■ Osservazione

Siccome y_i ed y_j vengono confrontati se e solo se uno dei due è un pivot, la probabilità che $X_{i,j} = 1$ è proprio pari alla probabilità che uno dei due sia un pivot. Quindi

$$\mathbf{E}[X_{i,j}] = \mathbf{Pr}[X_{i,j} = 1] = \frac{2}{j - i + 1}$$

Date queste osservazioni, si fa ora un'analisi formale.

■ Teorema

Si supponga che, ogni volta che un pivot viene scelto per RandomQS, tale selezione avvenga indipendentemente e u.a.r. tra tutti i possibili pivot. Allora, per ogni input, il valore atteso di confronti fatto da RandomQS è

$$2n \cdot \ln(n) + \mathcal{O}(n)$$

■ Dimostrazione

Siano y_1, y_2, \dots, y_n gli stessi valori di input x_1, x_2, \dots, x_n ordinati in maniera non decrescente ($y_1 \leq y_2 \leq \dots \leq y_n$).

Per $i < j$, sia

$$X_{i,j} = \begin{cases} 1 & \text{se } y_i \text{ e } y_j \text{ vengono confrontati dall'algoritmo} \\ 0 & \text{altrimenti} \end{cases}$$

Allora, il numero totale di confronti X soddisfa

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{i,j}$$

Per la linearità del valore atteso si ha

$$\mathbf{E}[X] = \mathbf{E}\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{i,j}\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \mathbf{E}[X_{i,j}]$$

Siccome $X_{i,j}$ assume solo valore 0 ed 1, $\mathbf{E}[X_{i,j}] = \Pr[X_{i,j} = 1]$. Per le osservazioni fatte in precedenza, si ha che $\Pr[X_{i,j} = 1] = 2/(j-i+1)$.

Usando la sostituzione $k = j - i + 1$ si ha

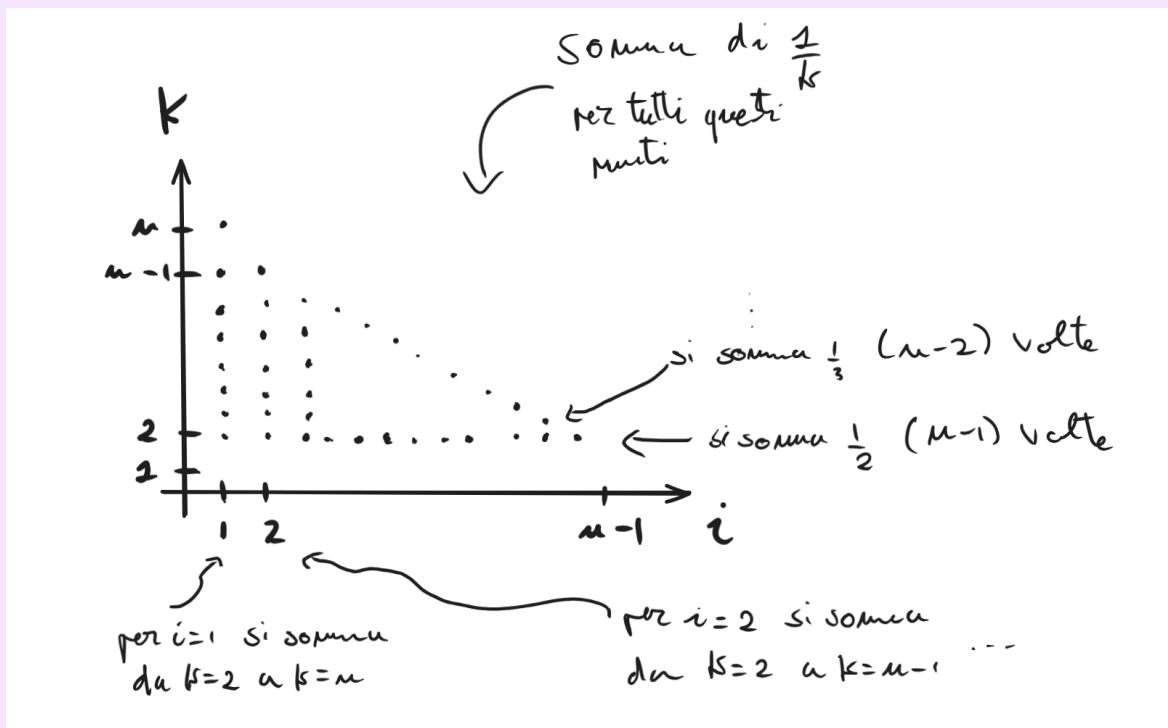
$$\begin{aligned}\mathbf{E}[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{j-i+1} \stackrel{(1)}{=} \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{2}{k} \\ &\stackrel{(2)}{=} \sum_{k=2}^n \sum_{i=1}^{n+1-k} \frac{2}{k} = \sum_{k=2}^n (n+1-k) \frac{2}{k} \\ &= \left((n+1) \sum_{k=2}^n \frac{2}{k} \right) - 2(n-1) \\ &= (2n+2) \sum_{k=1}^n \frac{1}{k} - 4n\end{aligned}$$

Dove:

- (1):

$$\sum_{j=i+1}^n \frac{1}{j-i+1} = \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n-i+1}$$

- (2): Invece di far variare k da i , si somma facendo variare i da k .



Ricordando che $H_n = \sum_{k=1}^n \frac{1}{k}$ soddisfa $H_n = \ln(n) + \Theta(1)$ si ha che

$$\mathbf{E}[X] = 2n \cdot \ln(n) + \Theta(n) = \Theta(n \log n)$$

Calcolo del mediano

Si studia ora un algoritmo probabilistico per il calcolo del mediano di una variabile aleatoria. Sia X una variabile aleatoria. Il *mediano* di X è un valore m tale per cui

$$\Pr[X \leq m] \geq \frac{1}{2} \text{ e } \Pr[X \geq m] \geq \frac{1}{2}$$

Ad esempio, per una variabile aleatoria discreta distribuita uniformemente su un numero dispari di valori distinti e ordinati $x_1, x_2, \dots, x_{2k+1}$, il mediano è il valore centrale x_{k+1} . Per una variabile aleatoria discreta distribuita uniformemente su un numero pari di valori distinti e ordinati x_1, x_2, \dots, x_{2k} , un qualsiasi valore nel range (x_k, x_{k+1}) è un mediano.

Il problema del calcolo del mediano può essere formalizzato come segue

Input: Un insieme S di $n = 2k + 1$ elementi presi da un universo totalmente ordinato.

Output: Il $k + 1$ -esimo elemento più grande in S .

Il mediano può venir calcolato facilmente ordinando in tempo $\mathcal{O}(n \log n)$, e selezionando l'elemento intermedio. Si mostra una soluzione probabilistica più efficiente.

L'idea dell'algoritmo è quella di trovare due elementi che risultino essere vicini nella sequenza ordinata di elementi contenuti in S , e tali per cui il mediano si trovi tra essi. Formalmente, si cercano due elementi $d, u \in S$ tali per cui

1. $d \leq m \leq u$, ossia il mediano m si trova tra d e u ,
2. per $C = \{s \in S : d \leq s \leq u\}$ si ha che $|C| = o(n/\log n)$, ossia, il numero di elementi tra d ed u è piccolo.

Una volta che questi due elementi vengono identificati, il mediano può esser trovato facilmente in tempo lineare. L'algoritmo conta, in tempo lineare, il numero l_d di elementi in S più piccoli di d ed ordina, in tempo sublineare o $o(n)$, l'insieme C . Essendo $|C| = o(n/\log n)$, l'insieme C può venir ordinato in tempo $o(n)$ utilizzando un qualsiasi algoritmo di ordinamento standard che impiega tempo $\mathcal{O}(k \log k)$ per ordinare k elementi. Il $(\lfloor n/2 \rfloor - l_d + 1)$ -esimo elemento nell'insieme C ordinato è il mediano m , essendovi esattamente $\lfloor n/2 \rfloor$ elementi in S più piccoli di m : $\lfloor n/2 - l_d \rfloor$ nell'insieme C ed l_d nell'insieme $S \setminus C$.

Per trovare gli elementi d ed u , si campiona con *reinserimento* un multi-insieme R di $\lceil n^{3/4} \rceil$ elementi di S . Essendo che il campionamento avviene con reinserimento, ogni elemento inserito in R viene scelto i.u.a.r. dall'insieme S . Quindi, un elemento di S può apparire più volte in R .

Essendo R un campione casuale di S , ci si aspetta che m , l'elemento mediano di S , sia vicino all'elemento mediano di R . Si scelgono quindi d ed u come elementi di R che

circondano il mediano di R .

In particolar modo, per garantire che l'insieme C contenga con alta probabilità m , si fissano d ed u in maniera tale che siano, rispettivamente, il $\lfloor n^{3/4}/2 - \sqrt{n} \rfloor$ -esimo e il $\lceil n^{3/4}/2 + \sqrt{n} \rceil$ -esimo elemento della sequenza degli elementi ordinati di R .

Si fornisce il seguente algoritmo

MedianAlgorithm

1. Scegli un multi-set R di $\lceil n^{3/4} \rceil$ elementi di S , scelti i.u.a.r. e con reinserimento.
 2. Ordina l'insieme R .
 3. Sia d il $\lfloor n^{3/4}/2 - \sqrt{n} \rfloor$ -esimo elemento più piccolo nell'insieme ordinato R .
 4. Sia u il $\lceil n^{3/4}/2 + \sqrt{n} \rceil$ -esimo elemento più piccolo nell'insieme ordinato R .
 5. Confrontando ogni elemento in S con d ed u , calcola l'insieme $C = \{s \in S : d \leq m \leq u\}$ e i numeri $l_d = |\{x \in S : x < d\}|$ e $l_u = |\{x \in S : x > u\}|$
 6. Se $l_d > n/2$ o $l_u > n/2$ allora **FAIL**
 7. Se $|C| \leq 4n^{3/4}$ ordina l'insieme C , altrimenti **FAIL**
 8. Restituisci il $(\lfloor n/2 \rfloor - l_d + 1)$ -esimo elemento nell'insieme ordinato C .
-

L'analisi dell'algoritmo mostra che la scelta della dimensione di R e le scelte per d ed u sono adatte per garantire che:

- L'insieme C risulta essere abbastanza grande da includere m con alta probabilità
- L'insieme C è sufficientemente piccolo per poter venir ordinato in tempo sublineare con alta probabilità

Si mostra ora che, indipendentemente dalle scelte casuali effettuate durante la procedura, l'algoritmo termina sempre in tempo lineare, e che questo o restituisce in output il risultato corretto o fallisce.

Teorema

L'algoritmo termina in tempo lineare, e se non restituisce in output **FAIL**, allora restituisce correttamente l'elemento mediano dell'insieme S in input.

Dimostrazione

La correttezza dell'algoritmo segue dal fatto che questo può restituire una risposta sbagliata solo se il mediano m non si trovi nell'insieme C . Ma nel caso in cui questo sia

vero, allora $l_d > n/2$ oppure $l_u > n/2$, e quindi il passo 6 dell'algoritmo garantisce che questo restituisca **FAIL**.

Similmente, sino a quando C rimane sufficientemente piccolo, il lavoro totale effettuato è lineare in S . Il passo 7 garantisce che l'algoritmo non impieghi più di tempo lineare, infatti, se l'ordinamento di C richiede troppo tempo vista la sua dimensione, l'algoritmo restituisce **FAIL** senza ordinare.

Si vuole ora delimitare la probabilità che l'algoritmo restituisca **FAIL**. Si identificano quindi tre eventi tali per cui, se nessuno di questi capita, allora l'algoritmo non fallisce.

Siano

$$\mathcal{E}_1 : Y_1 = |\{r \in R | r \leq m\}| < \frac{1}{2}n^{3/4} - \sqrt{n}$$

$$\mathcal{E}_2 : Y_2 = |\{r \in R | r \geq m\}| < \frac{1}{2}n^{3/4} - \sqrt{n}$$

$$\mathcal{E}_3 : |C| > 4n^{3/4}$$

Bookmark Lemma

L'algoritmo fallisce se e solo se almeno uno degli eventi \mathcal{E}_1 , \mathcal{E}_2 o \mathcal{E}_3 si verifica.

Bookmark Dimostrazione

Il fallimento nel passo 7 dell'algoritmo è equivalente all'evento \mathcal{E}_3 .

Il fallimento nel passo 6 avviene se e solo se $l_d > n/2$ o $l_u > n/2$. Ma per $l_d > n/2$ il $(n^{3/4}/2 - \sqrt{n})$ -esimo elemento più piccolo di R deve essere più grande di m , è quindi equivalente all'evento \mathcal{E}_1 . Similmente, $l_u > n/2$ è equivalente all'evento \mathcal{E}_2 .

Bookmark Lemma

$$\Pr[\mathcal{E}_1] \leq \frac{1}{4}n^{-1/4}$$

Bookmark Dimostrazione

Sia X_i una variabile aleatoria definita come segue

$$X_i = \begin{cases} 1 & \text{se l'} i\text{-esimo campione è minore o uguale al mediano} \\ 0 & \text{altrimenti} \end{cases}$$

Le variabili X_i per ogni i sono indipendenti, essendo che il campionamento viene fatto con reinserimento. Essendovi $(n - 1)/2 + 1$ elementi in S che sono minori o uguali del mediano, la probabilità che un elemento S scelto casualmente sia minore o uguale del mediano può essere scritta come

$$\Pr [X_i = 1] = \frac{(n - 1)/2 + 1}{n} = \frac{1}{2} + \frac{1}{2n}.$$

L'evento \mathcal{E}_1 è equivalente a

$$Y_1 = \sum_{i=1}^{n^{3/4}} X_i < \frac{1}{2} n^{3/4} - \sqrt{n}$$

Essendo Y_1 la somma di prove che costituiscono un processo di Bernoulli, è una variabile aleatoria binomiale con parametri $n^{3/4}$ e $1/2 + 1/2n$. Quindi si ha che, facendo riferimento al calcolo della varianza di una v.a. binomiale:

$$\begin{aligned} \text{Var}[Y_1] &= n^{3/4} \left(\frac{1}{2} + \frac{1}{2n} \right) \left(\frac{1}{2} - \frac{1}{2n} \right) \\ &= \frac{1}{4} n^{3/4} - \frac{1}{4n^{5/4}} \\ &< \frac{1}{4} n^{3/4} \end{aligned}$$

Applicando la diseguaglianza di Chebyshev si ottiene che

$$\begin{aligned} \Pr [\mathcal{E}_1] &= \Pr \left[Y_1 < \frac{1}{2} n^{3/4} - \sqrt{n} \right] \\ &\leq \Pr [|Y_1 - \mathbf{E}[Y_1]| > \sqrt{n}] \\ &\leq \frac{\text{Var}[Y_1]}{n} \\ &< \frac{\frac{1}{4} n^{3/4}}{n} = \frac{1}{4} n^{-1/4} \end{aligned}$$

Similmente si può ottenere lo stesso bound sulla probabilità dell'evento \mathcal{E}_2 .

Si da ora una delimitazione alla probabilità dell'evento \mathcal{E}_3 .

Lemma

$$\Pr [\mathcal{E}_3] \leq \frac{1}{2} n^{-1/4}$$

Dimostrazione

Se si verifica \mathcal{E}_3 , ossia $|C| > 4n^{3/4}$, allora si verifica almeno uno tra questi due eventi:

$\mathcal{E}_{3,1}$: almeno $2n^{3/4}$ elementi di C sono più grandi del mediano;

$\mathcal{E}_{3,2}$: almeno $2n^{3/4}$ elementi di C sono più piccoli del mediano.

Si da una delimitazione alla probabilità che il primo evento si verifichi. Tale bound, per simmetria, vale anche per il secondo. Se vi sono almeno $2n^{3/4}$ elementi di C maggiori del mediano, allora la posizione di u negli elementi di S ordinati è almeno $\frac{1}{2}n + 2n^{3/4}$ e quindi R ha almeno $\frac{1}{2}n^{3/4} - \sqrt{n}$ campioni tra gli $\frac{1}{2}n - 2n^{3/4}$ elementi più grandi in S .

Sia X il numero di campioni tra gli $\frac{1}{2}n - 2n^{3/4}$ elementi più grandi in S . Sia

$X = \sum_{i=1}^{n^{3/4}} X_i$, dove

$$X_i = \begin{cases} 1 & \text{se l'} i\text{-esimo campione è tra gli } \frac{1}{2}n - 2n^{3/4} \text{ elementi più grandi in } S, \\ 0 & \text{altrimenti} \end{cases}$$

X è una variabile aleatoria binomiale, e quindi si ha che

$$\mathbf{E}[X] = \frac{1}{2}n^{3/4} - 2\sqrt{n}$$

e

$$\text{Var}[X] = n^{3/4} \left(\frac{1}{2} - 2n^{-1/4} \right) \left(\frac{1}{2} + 2n^{-1/4} \right) = \frac{1}{4}n^{3/4} - 4n^{1/4} < \frac{1}{4}n^{3/4}$$

Applicando la diseguaglianza di Chebyshev si ottiene che

$$\begin{aligned} \mathbf{Pr}[\mathcal{E}_{3,1}] &= \mathbf{Pr}\left[X \geq \frac{1}{2}n^{3/4} - \sqrt{n}\right] \leq \mathbf{Pr}[|X - \mathbf{E}[X]| \geq \sqrt{n}] \\ &\leq \frac{\text{Var}[X]}{n} < \frac{\frac{1}{4}n^{3/4}}{n} = \frac{1}{4}n^{-1/4}. \end{aligned}$$

Similmente

$$\mathbf{Pr}[\mathcal{E}_{3,2}] \leq \frac{1}{4}n^{-1/4}$$

e quindi

$$\mathbf{Pr}[\mathcal{E}_3] \leq \mathbf{Pr}[\mathcal{E}_{3,1}] + \mathbf{Pr}[\mathcal{E}_{3,2}] \leq \frac{1}{2}n^{-1/4}$$

Combinando le delimitazioni ottenute per i tre eventi, tali per cui il verificarsi di almeno uno di questi comporta il fallimento dell'algoritmo, si ha che la probabilità che l'algoritmo fallisca è pari a

$$\mathbf{Pr}[\mathcal{E}_1] + \mathbf{Pr}[\mathcal{E}_2] + \mathbf{Pr}[\mathcal{E}_3] \leq n^{-1/4}$$

Si può ripetere l'algoritmo mostrato sino a quando questo non trova il mediano con successo. Così facendo, si può ottenere un algoritmo iterativo che non fallisce mai, ma con

un tempo di esecuzione casuale. I campioni presi nelle esecuzioni successive dell'algoritmo sono indipendenti, quindi il successo di ogni esecuzione è indipendente dalle altre. Da quest'ultima osservazione, si fa presente che il numero di esecuzioni necessarie affinché si ottenga il mediano con successo è una variabile aleatoria geometrica.

Contention Resolution

Si hanno n processi P_1, P_2, \dots, P_n , i quali competono per l'accesso ad un database condiviso. L'intervallo di tempo durante il quale avviene questa dinamica viene suddiviso in *rounds* discreti. Al database può accedere un singolo processo in ciascun round.

Se due o più processi tentano di accedere al database simultaneamente, ossia all'inizio dello stesso round, allora questi vengono rigettati e, per la durata di quel round, nessun processo accede al database.

Nonostante ogni processo voglia accedere alla risorsa condivisa il più frequentemente possibile, è inutile che ciascuno di essi provi a farlo per ogni round perché, se così fosse, nessuno di essi accederebbe al database, rendendo la risorsa inutilizzata.

Si vogliono quindi suddividere i rounds facendo sì che questi vengano assegnati equamente a ciascun processo, in maniera tale che ciascuno di essi possa accedere al database su base regolare.

Si fa presente inoltre come i processi non possano comunicare tra loro. Di conseguenza, un processo che tenta di accedere alla risorsa in un certo round, non può sapere se gli altri stiano compiendo la stessa operazione.

Si progetta ora un protocollo probabilistico per la risoluzione di questo problema:

In ciascun round, ogni processo decide se provare ad accedere al database con probabilità $p > 0$, indipendentemente dalla decisione effettuata dagli altri processi.

Se esattamente un processo decide di provare l'accesso, allora questo avrà successo. Se ci provano invece due o più processi, allora questi verranno tagliati fuori, come da definizione del problema. Se in un round, nessuno dei processi prova l'accesso, allora il round risulta "sprecato".

Questa strategia si trova al principio del *symmetry-breaking paradigm*: se tutti i processi operassero in maniera prefissata, provando ripetutamente ad accedere al database durante lo stesso istante, allora non vi sarebbe progresso.

Si vuole ora scegliere p affinché la probabilità che un processo possa accedere con successo al database sia massimizzata.

Sia P_i uno dei processi in input, e sia t un round arbitrario.

Si definisce l'evento

$$A[i, t] = \text{Il processo } P_i \text{ prova ad accedere al database al round } t$$

Si ha che ogni processo prova ad accedere al database ad ogni round con probabilità p , quindi $\forall i = 1, \dots, n$ e $\forall t$ si ha

$$\Pr[A[i, t]] = p$$

Il complementare dell'evento $A[i, t]$ è definito come segue

$A[i, t]^c =$ Il processo P_i non prova ad accedere al database al round t

e, ovviamente

$$\Pr[A[i, t]^c] = 1 - \Pr[A[i, t]] = 1 - p$$

A partire da questi eventi, si definisce l'evento tale per cui un processo riesce ad accedere con successo al database in un dato round:

$S[i, t] =$ Il processo P_i accede con successo al database al round t

■ Osservazione

Il processo P_i accede con successo al database al round t se e solo se tale processo prova ad accedere al round t e **tutti** gli altri processi non provano ad accedere al database al round t .

Quindi, si ha che

$$S[i, t] = A[i, t] \cap \left(\bigcap_{j \neq i} A[j, t]^c \right)$$

Dalla definizione del protocollo, si ha che tutti gli eventi sono indipendenti tra loro, ossia $\forall i, j = 1, \dots, n, j \neq i$

$$\Pr[A[i, t] \cap A[j, t]] = \Pr[A[i, t]] \cdot \Pr[A[j, t]]$$

Da questa osservazione si ottiene la probabilità che si verifichi l'evento $S[i, t]$:

$$\Pr[S[i, t]] = \Pr[A[i, t]] \cdot \prod_{j \neq i} \Pr[A[j, t]^c] = p(1 - p)^{n-1}$$

Si ha quindi la probabilità che P_i acceda con successo al database al round t .

Rimane ora da scegliere p per massimizzare tale probabilità.

■ Osservazione

Per $p = 0$ e $p = 1$, si ha che $\forall i = 1, \dots, n$ e $\forall t$

$$\Pr[S[i, t]] = 0$$

Infatti,

- per $p = 0$, per ogni round nessun processo prova ad accedere al database
- per $p = 1$, ad ogni round ogni processo prova ad accedere al database, rimanendo esclusi per definizione del problema

La funzione $f(p) = p(1 - p)^{n-1}$ assume valori positivi per $0 < p < 1$. Si vuole vedere per quale valore di p tale funzione risulti essere massimizzata. Per fare ciò, si calcola la sua derivata prima

$$f'(p) = (1 - p)^{n-1} - (n - 1)p(1 - p)^{n-2}$$

La quale assume valore 0 per $p = 1/n$, dove $f(p)$ risulta essere massimizzata. Quindi si ha che per tale valore di p , la probabilità che si verifichi $S[i, t]$ risulta essere massima.

Per $p = 1/n$ si ha quindi

$$\Pr[S[i, t]] = \frac{1}{n} \left(1 - \frac{1}{n}\right)^{n-1}$$

Si fanno le seguenti osservazioni sull'andamento asintotico di quest'ultima espressione:

Osservazione

1. La funzione $\left(1 - \frac{1}{n}\right)^n$ ha convergenza monotona da $\frac{1}{4}$ salendo fino a $\frac{1}{e}$ per $n \geq 2$
2. La funzione $\left(1 - \frac{1}{n}\right)^{n-1}$ ha convergenza monotona da $\frac{1}{2}$ scendendo fino a $\frac{1}{e}$ per $n \geq 2$

Si ha quindi che per $n \geq 2$,

$$\frac{1}{e} \leq \left(1 - \frac{1}{n}\right)^{n-1} \leq \frac{1}{2}$$

ciò implica che

$$\frac{1}{(en)} \leq \Pr[S[i, t]] \leq \frac{1}{(2n)}$$

ossia

$$\Pr[S[i, t]] = \Theta\left(\frac{1}{n}\right)$$

Assodato che $p = 1/n$ massimizzi la probabilità che un processo acceda con successo al database in un dato round, si vuole studiare quanto tempo sia necessario affinché un processo P_i acceda con successo alla risorsa almeno una volta.

Si definisce l'evento

$F[i, t] =$ Il processo P_i non accede al database con successo in ciascun round da 1 a t

Si osserva che

$$F[i, t] = \left[\bigcap_{r=1}^t S[i, r]^c \right]$$

Ed essendo tali eventi indipendenti, si ha

$$\Pr[F[i, t]] = \Pr\left[\bigcap_{r=1}^t S[i, r]^c\right] = \prod_{r=1}^t \Pr[S[i, r]^c] = \left[1 - \frac{1}{n}\left(1 - \frac{1}{n}\right)^{n-1}\right]^t$$

Per quanto osservato in precedenza,

$$\frac{1}{n}\left(1 - \frac{1}{n}\right)^{n-1} \geq \frac{1}{en} \implies \Pr[F[i, t]] = \prod_{r=1}^t \Pr[S[i, r]^c] \leq \left(1 - \frac{1}{en}\right)^t$$

Ponendo $t = en$, si può fare riferimento sempre all'osservazione fatta in precedenza.

Essendo che en non è un intero, si prende $t = \lceil en \rceil$

$$\Pr[F[i, t]] \leq \left(1 - \frac{1}{en}\right)^{\lceil en \rceil} \leq \left(1 - \frac{1}{en}\right)^{en} \leq \frac{1}{e}$$

Tale disegualanza mostra come la probabilità che il processo P_i non abbia successo in ciascuno dei round che vanno da 1 ad $\lceil en \rceil$ è delimitata superiormente dalla costante e^{-1} , indipendente da n .

Si vuole ora ridurre la probabilità che ciò accada.

Ponendo $t = \lceil en \rceil \cdot (c \ln n)$ si ha che

$$\Pr[F[i, t]] \leq \left(1 - \frac{1}{en}\right)^t \leq \left(\left(1 - \frac{1}{en}\right)^{\lceil en \rceil}\right)^{c \ln n} \leq e^{-c \ln n} = n^{-c}$$

Quindi, asintoticamente, si ha che la probabilità che P_i non abbia ancora avuto successo dopo $\Theta(n)$ rounds è delimitata da una costante, mentre per $\Theta(n \ln n)$ rounds, questa probabilità assume lo stesso comportamento dell'inverso di un polinomio in n .

Ci si chiede ora quanti rounds siano necessari tali per cui vi sia un'alta probabilità affinché tutti i processi accedano con successo al database almeno una volta.

Per studiare tale quantità, si definisce l'evento

$$F_t = \text{Il protocollo fallisce dopo } t \text{ round}$$

Dove il protocollo *fallisce* dopo t round se esiste almeno un processo che non è riuscito ad accedere con successo al database.

Si vuole trovare un valore ragionevole di t affinché $\Pr[F_t]$ sia sufficientemente piccolo.

Si osserva esplicitamente che si verifica F_t se e solo se accade uno degli eventi $F[i, t]$.

Allora

$$F_t = \bigcup_{i=1}^n F[i, t]$$

la quale risulta essere un'unione di eventi non indipendenti tra loro, questo perché se un processo fallisce l'accesso alla risorsa condivisa, allora ve ne deve necessariamente essere un'altro che ha compiuto la stessa operazione. Si può fare quindi riferimento all'*union bound*:

$$\Pr[F_t] \leq \sum_{i=1}^n \Pr[F[i, t]]$$

L'espressione a destra della diseguaglianza è una somma di n termini, ciascuno dei quali assume lo stesso valore. Per rendere la probabilità di F_t piccola, bisogna rendere ciascuno dei termini a destra significativamente più piccoli di $1/n$. Per $t = \Theta(n)$ ciascuno di tali termini risulta essere delimitato solamente da una costante. Per $t = \lceil en \rceil \cdot (c \ln n)$, si ha che $\Pr[F[i, t]] \leq n^{-c}$ per ogni i . In particolar modo, per $t = 2\lceil en \rceil \ln n$ si ha

$$\Pr[F_t] \leq \sum_{i=1}^n \Pr[F[i, t]] \leq n \cdot n^{-2} = n^{-1}$$

e ciò mostra il seguente teorema

Teorema

Con probabilità almeno $1 - n^{-1}$, tutti i processi accedono al database con successo almeno una volta entro $t = 2\lceil en \rceil \ln n$ rounds.

Hash function & Similar items

Hashing: un'implementazione randomizzata dei dizionari

Il problema del dizionario

Dato un universo U di possibili elementi, si vuole definire una struttura dati in grado di tener traccia di un suo sottoinsieme $S \subseteq U$ arbitrario di n elementi, dove n è una frazione di $|U|$, con l'obiettivo di inserire, eliminare e cercare elementi in S in maniera efficiente.

Tale struttura dati prende il nome di **dizionario**, che supporta le seguenti operazioni, alle quali gli elementi di S sono quindi soggetti:

- **MakeDictionary()**: Inizializza un dizionario in grado di mantenere un sottoinsieme S di U . Inizialmente, $S = \emptyset$.
- **Insert(u)**: Inserisci un elemento $u \in U$ in S .
- **Delete(u)**: Se $u \in S$, rimuovi u da S .

- $\text{Lookup}(u)$: Determina se $u \in S$ o meno.

Si vuole studiare il caso in cui la dimensione di U sia estremamente grande rispetto ad n , per il quale la definizione di un array di size $|U|$ non risulti essere ammissibile. Si vogliono trovare soluzioni proporzionali ad $|S| = n$.

Per fare ciò, si farà riferimento all'**hashing**, una tecnica utilizzabile per mantenere un insieme di elementi che subisce modifiche dinamicamente.

Si definisce un array H di dimensione $m \approx n$ per memorizzare tali informazioni, e si utilizza una funzione $h : U \longrightarrow \{0, 1, \dots, m - 1\}$ che mappa ciascun elemento di U in una delle posizioni dell'array. Tale funzione h prende il nome di **funzione hash**, mentre l'array H prende il nome di **tabella hash**.

Per aggiungere un elemento u all'insieme S , si inserisce u nella posizione $h(u)$ di H . Questo approccio funziona molto bene nel caso in cui, $\forall u, v \in S$ tali che $u \neq v$ vale $h(u) \neq h(v)$. Per tale scenario, l'operazione di ricerca di un elemento u impiega tempo costante, supponendo che il calcolo di $h(u)$ avvenga anch'esso in tempo costante: basta semplicemente controllare l'array in posizione $H[h(u)]$, la quale o è vuota o contiene u .

In generale, possono esservi però elementi $u, v \in S$ con $u \neq v$ tali per cui $h(u) = h(v)$.

Quando ciò accade, si dice che tali elementi *collidono*, essendo mappati nella stessa entrata in H .

Per gestire le collisioni, si assume che ogni posizione $H[i]$ della tabella hash memorizzi una lista concatenata di tutti gli elementi $u \in S$ tali per cui $h(u) = i$.

Le operazioni descritte in precedenza per il dizionario su un elemento $u \in U$ prevedono quindi il calcolo di $h(u)$ precedentemente alla loro esecuzione. In particolar modo

- **Insert(u)**: Calcola $h(u)$ e inserisci u nella lista in $H[h(u)]$.
- **Lookup(u)**: Calcola $h(u)$ e scandisci la lista in posizione $H[h(u)]$ per vedere se u è presente in essa.
- **Delete(u)**: Calcola $h(u)$ e scandisci la lista in posizione $H[h(u)]$ per vedere se u è presente in essa, ed eventualmente rimuovilo.

Il tempo per eseguire le operazioni appena descritte è proporzionale al tempo necessario per il calcolo di $h(u)$ sommato alla lunghezza della lista concatenata in posizione $H[h(u)]$, dove quest'ultima quantità è pari al numero di elementi in S che collidono con u .

Osservazione

L'efficienza del dizionario è basata sulla scelta della funzione hash h .

Si vuole definire quindi una funzione hash h che distribuisca bene gli elementi, ossia che renda sufficientemente rara la presenza di collisioni, per la quale nessuna entrata della tabella hash H possa contenere troppi elementi. Per fare ciò risulta necessario un approccio

probabilistico alla costruzione di tale funzione.

Ma prima, si mostra perché l'approccio deterministico non risulti essere buono nel caso in cui $|U| >> m$.

Proprietà

Sia $|U| > m^2$. Allora, per ogni funzione hash deterministica h , esiste un insieme S di dimensione n tale per cui ogni suo elemento viene mappato nella stessa entrata di H .

Dimostrazione

Sia $h : U \rightarrow \{0, 1, \dots, m - 1\}$ una funzione hash fissata che mappa ciascun elemento di U in H . Si vuole scegliere un insieme S di n elementi tale per cui ogni elemento venga mappato nella stessa entrata dell'array.

Sia $S = \{u \in U : h(u) = i\}$. Allora $H[i]$ conterrà tutti gli elementi di S .

Quindi, nel caso peggiore, tutti gli elementi di U inseriti in H apparterrebbero a $S = \{u \in U : h(u) = i\}$, e l'esecuzione delle varie operazioni richiederebbe tempo $\Theta(n)$.

Hashing deterministico

Si vede ora brevemente un esempio di hashing deterministico.

Sia U l'insieme universo tale per cui $|U| = N >> m \approx n := |S|$.

Si supponga di rappresentare gli elementi u di U come interi in $\{0, 1, \dots, N - 1\}$.

Sia ora p un numero primo tale per cui $m \leq p \leq 2m$.

Si definisce ora h tale per cui, $\forall u \in U$

$$h(u) = u \pmod{p}$$

Allora la proprietà vista in precedenza rimane vera, ma gli elementi di U risultano essere ben distribuiti.

Tale approccio risulta buono per applicazioni "statiche".

Hashing probabilistico

Si utilizza la probabilità per la costruzione della funzione hash h .

In generale, si vuole che tale funzione possegga le seguenti caratteristiche

1. $h(u)$ deve essere il "più casuale" possibile. Idealmente, h dovrebbe mappare gli elementi di U in maniera completamente uniforme. Si vedrà però che tale proprietà presenterà un costo estremamente eccessivo.

2. $h(u)$ deve essere veloce da calcolare algoritmamente. Idealmente, si vorrebbe calcolare $h(u)$ in tempo proporzionale al tempo necessario alla lettura di u , ossia $\mathcal{O}(1)$ se u è un intero, $\mathcal{O}(l)$ se u è una stringa di lunghezza l .
3. h occupa dello spazio in memoria, essendo essa implementata mediante una struttura dati. Questo spazio deve essere il più piccolo possibile, idealmente $\mathcal{O}(1)$.

Si formalizza il concetto di *famiglia di funzioni hash*.

Una famiglia di funzioni hash è un insieme $\mathcal{H} \subseteq \{0, 1, \dots, m - 1\}^N$ tale per cui ogni $h \in \mathcal{H}$ assegna un valore da $\{0, 1, \dots, m - 1\}$ a ciascuno degli N elementi di U come segue:

Sia $h = (y_1, y_2, \dots, y_N)$, allora per ogni $u_i \in U = \{u_1, u_2, \dots, u_N\}$ si definisce $h(u_i) = y_i$, dove ovviamente $\forall i = 1, \dots, N, y_i \in \{0, 1, \dots, m - 1\}$.

Si considera inizialmente l'approccio probabilistico banale:

Per ogni $u \in U$, si sceglie un valore $h(u)$ uniformly at random nell'insieme $\{0, 1, \dots, m - 1\}$, indipendentemente dalle scelte fatte per gli altri elementi dell'insieme universo, ossia, $\forall i \in \{0, 1, \dots, m - 1\}$,

$$\Pr[h(u) = i] = \frac{1}{m}$$

In questo caso, la probabilità che due valori $h(u)$ e $h(v)$ scelti casualmente siano uguali (e che quindi collidano) è relativamente bassa.

Proprietà

Con questo schema di hashing probabilistico uniforme, la probabilità che due valori $h(u)$ e $h(v)$ collidano, ossia $h(u) = h(v)$ è esattamente $1/m$

Dimostrazione

Per ogni coppia $(h(u), h(v))$ vi sono m^2 possibili scelte, ciascuna delle quali equiprobabile, ed esattamente m di queste risultano in una collisione.

Nonostante questo approccio definisca una buona distribuzione degli elementi indipendentemente da S , la sua efficienza non risulta essere buona. Infatti, per effettuare l'operazione di ricerca $\text{Lookup}(u)$ risulta necessario mantenere l'insieme di tutte le coppie $\{(u, h(u)) : u \in S\}$ per tener traccia dell'indice al quale è stato associato l'elemento u . In particolar modo, si dovrebbero memorizzare n indirizzi indipendenti, e la ricerca su tale insieme risulterebbe ancora pari a $\Theta(n)$, non portando quindi al miglioramento cercato.

La selezione di un valore $h(u)$ u.a.r. nell'insieme $\{0, 1, \dots, m - 1\}$ per ogni $u \in U$ equivale ad estrarre una funzione hash h u.a.r. dall'insieme di tutte le possibili funzioni hash che mappano gli elementi di U in $\{0, 1, \dots, m - 1\}$. Sia ora $\mathcal{H} \subseteq \{0, 1, \dots, m - 1\}^N$ una generica

famiglia di funzioni hash. L'analisi del caso atteso dell'algoritmo che estrae uniformemente una funzione hash da questo insieme deve prendere in considerazione la struttura di \mathcal{H} e il fatto che h è stata scelta uniformemente da esso. Nel worst-case, sono necessari quindi almeno $\log_2 |\mathcal{H}|$ bits per rappresentare, e memorizzare, h . Se, per contraddizione, si usassero $t < \log_2 |\mathcal{H}|$ bits per ogni $h \in \mathcal{H}$, allora si potrebbero distinguere solo $2^t < |\mathcal{H}|$ funzioni hash, i quali non sono sufficienti visto che, per via del fatto che la scelta della funzione hash avviene u.a.r., una qualsiasi h può essere scelta dall'insieme.

Idealmente, si vorrebbe che la funzione hash sia completamente uniforme:

Definizioni (Funzione hash uniforme)

Sia $h \in \mathcal{H} \subseteq \{0, 1, \dots, m - 1\}^N$ scelta uniformemente. Si dice che \mathcal{H} è uniforme se, per ogni $y_1, \dots, y_N \in \{0, 1, \dots, m - 1\}$ si ha che

$$\Pr[h = (y_1, \dots, y_N)] = \frac{1}{m^N}$$

Si osserva esplicitamente che una famiglia di funzioni hash \mathcal{H} è definita uniforme se e solo se $\mathcal{H} = \{0, 1, \dots, m - 1\}^N$, dove $\forall h \in \mathcal{H}, h : U \longrightarrow \{0, 1, \dots, m - 1\}$ con $|U| = N$.

I tre requisiti precedentemente enunciati sono in conflitto. Infatti, non è possibile ottenere tutti e tre simultaneamente. Ad esempio, si supponga che si voglia ottenere una funzione hash uniforme. Allora, per ogni scelta di $y_1, \dots, y_N \in \{0, 1, \dots, m - 1\}$,

$\Pr[h = (y_1, \dots, y_N)] = \frac{1}{m^N}$. Questo però è possibile solo se $\mathcal{H} = \{0, 1, \dots, m - 1\}^N$. In ogni altro caso, esisterebbe almeno una scelta di $y_1, \dots, y_N \in \{0, 1, \dots, m - 1\}$ tale per cui

$\Pr[h = (y_1, \dots, y_N)] < \frac{1}{m^N}$. Allora, una funzione hash uniforme richiederebbe $\log_2 |\mathcal{H}| = \log_2 m^N = N \log_2 m$ bits di memoria, una quantità tipicamente eccessiva.

Una funzione hash di questo tipo può essere implementata come segue:

Si riempie un vettore $V[1, N]$ con interi da 0 a $m - 1$ e si definisce $h(x) = V[x]$. Tale funzione hash soddisfa i requisiti 1 e 2 ma non il 3.

Si vuole mostrare come un uso più controllato della randomizzazione possa portare ad un'implementazione efficiente del dizionario.

Universal hashing

L'idea è quella di scegliere una funzione hash casualmente, ma non dall'insieme di tutte le possibili funzioni hash che assumono valori in $\{0, 1, \dots, m - 1\}$, bensì da una specifica famiglia di funzioni \mathcal{H} .

Ciascuna di queste funzioni $h \in \mathcal{H}$ mappa l'universo U in $\{0, 1, \dots, m - 1\}$ e deve essere definita in maniera tale che rispetti le due seguenti proprietà:

Proprietà

Per ogni coppia di elementi $u, v \in U$, la probabilità che una funzione $h \in \mathcal{H}$ scelta casualmente soddisfi $h(u) = h(v)$ è al più $1/m$.

Proprietà

Ogni $h \in \mathcal{H}$ può essere rappresentata in maniera compatta e, per un dato $h \in \mathcal{H}$ e $u \in U$, il valore $h(u)$ può essere calcolato in maniera efficiente.

Quest'ultima proprietà è stata definita informalmente, e verrà resa più precisa successivamente.

Una famiglia di funzioni \mathcal{H} che soddisfa la prima proprietà prende il nome di **famiglia di funzioni universale**.

Definizione (Famiglia di funzioni hash universale)

Una famiglia \mathcal{H} di funzioni è *universale* se $\forall u, v \in U$ tali che $u \neq v$:

$$\Pr_{h \in \mathcal{H}}[h(u) = h(v)] \leq \frac{1}{m}$$

Osservazione

La classe di tutte le possibili funzioni da U in $\{0, 1, \dots, m - 1\}$ è universale.

Questo perché, pescando casualmente una funzione hash da tale classe, la probabilità che $\forall u, v \in U$ $h(u) = h(v)$ è esattamente $1/m$ per l'argomentazione vista in precedenza.

Si vuole quindi individuare una famiglia di funzioni che rispetti le due proprietà precedentemente enunciate. Per fare ciò, si rende precisa la proprietà di base che caratterizza una classe di funzioni hash universale, e che motiva la scelta di funzioni appartenenti a queste classi. Si vuole mostrare come, sia h una funzione selezionata casualmente da una famiglia di funzioni hash universale \mathcal{H} , allora in ogni insieme $S \subseteq U$ di n elementi e per ogni $u \in S$, il numero atteso di elementi in S che collidono con u è costante per $m = \Theta(n)$.

Teorema

Sia \mathcal{H} una famiglia di funzioni hash universale. Sia $S \subseteq U$ un insieme di n elementi, e sia $u \in S$. Sia h una funzione scelta uniformemente a caso da \mathcal{H} e sia X la variabile aleatoria che conta il numero di elementi di S mappati in $h(u)$, ossia

$$X = |\{s \in S : h(s) = h(u)\}|$$

Allora

$$E[X] \leq 1 + \frac{n}{m}$$

Dimostrazione

Sia u un elemento di S fissato.

Per ogni $s \in S$ si definisce la variabile aleatoria X_s come segue

$$X_s = \begin{cases} 1 & \text{se } h(s) = h(u) \\ 0 & \text{altrimenti} \end{cases}$$

Si ha che

$$E[X_s] = \Pr[X_s = 1] \leq \frac{1}{m}$$

essendo la classe di funzioni hash \mathcal{H} universale.

Allora, essendo

$$X = \sum_{s \in S} X_s$$

Per la linearità del valore atteso si ha

$$\begin{aligned} E[X] &= E\left[\sum_{s \in S} X_s\right] = \sum_{s \in S} E[X_s] = \sum_{s \in S} \Pr[h(s) = h(u)] \\ &= 1 + \sum_{s \in S \setminus \{u\}} \Pr[h(s) = h(u)] \leq 1 + \frac{n}{m} \end{aligned}$$

Per $m = \Theta(n)$ ci si aspetta quindi che il numero di collisioni per un dato elemento u sia costante.

Si definisce ora una particolare classe di funzioni hash universale per la risoluzione del problema del dizionario.

Si sceglie un numero primo p tale per cui $n \leq p \leq 2n$ (il quale esiste sempre). La dimensione della tabella hash H utilizzata per memorizzare gli elementi di S è pari a tale numero p .

Si identifica ciascun elemento $x \in U$ con un intero base p di r cifre:

$$x = (x_1, x_2, \dots, x_r) : 0 \leq x_i < p \text{ per } i = 1, \dots, r$$

Se $|U| = N$, allora r deve essere tale per cui $p^r \geq N$, perché a ciascun elemento di U deve essere rappresentato con r cifre in base p in maniera univoca, quindi

$$r \geq \frac{\log(N)}{\log(p)}$$

Sia \mathcal{A} l'insieme di tutti i vettori della forma $a = (a_1, \dots, a_r)$, dove $a_i \in \{0, 1, \dots, p-1\}$ per $i = 1, \dots, r$.

Per ogni $a \in \mathcal{A}$ si definisce la funzione lineare

$$h_a(x) = \left[\sum_{i=1}^r a_i x_i \right] \pmod{p}$$

Così facendo, si definisce quindi una famiglia di funzioni hash

$$\mathcal{H} = \{h_a : a \in \mathcal{A}\}$$

Si osserva esplicitamente che il numero di funzioni così costruite è pari a p^r , questo perché ad ogni funzione in \mathcal{H} è associato un vettore a univoco di r cifre, ciascuna che assume valori da 0 a $p-1$, di conseguenza esistono p^r vettori distinti.

Quindi, essendo $p = \Theta(n)$ (si ricorda che $n \leq p \leq 2n$)

$$|\mathcal{H}| = p^r = \Theta(n^r)$$

Per eseguire `MakeDictionary()` si sceglie un vettore casuale $a = (a_1, a_2, \dots, a_r)$ da \mathcal{A} .

Questo è possibile scegliendo ciascun a_i uniformly at random da $\{0, 1, \dots, p-1\}$, formando così h_a .

Tale funzione rispetta la seconda proprietà precedentemente enunciata: ha una rappresentazione compatta, essendo che, scegliendo e memorizzando un a casuale, è possibile calcolare $h_a(u)$ per ogni $u \in U$. Per la scelta e la memorizzazione di a , e quindi di h_a , sono necessari $r = \Theta(\log(N)/\log(p))$ cifre, ciascuna avente lunghezza di $\log(p)$ bits.

La si usa successivamente per implementare le operazioni di `Insert(u)`, `Delete(u)` e `Lookup(u)` come enunciato in precedenza. Si ricorda che queste operazioni vengono precedute dal calcolo della funzione hash, il quale è stato mostrato essere efficiente. Per mostrare quindi che tale implementazione del dizionario risulti essere efficiente, rimane da dimostrare che gli elementi di U vengano distribuiti "bene" all'interno dell'hash table, e per fare ciò, bisogna mostrare che \mathcal{H} sia una famiglia di funzioni hash universale.

Osservazione

Una collisione $h_a(x) = h_a(y)$ definisce un'equazione lineare modulo p .

Per analizzare questa equazione, si usa la seguente "legge di cancellazione":

Lemma

Per ogni primo p ed ogni intero $z \neq 0 \pmod{p}$, e per ogni intero α, β

$$\alpha z = \beta z \pmod{p} \implies \alpha = \beta \pmod{p}$$

■ Dimostrazione

Sia $\alpha z = \beta z \pmod{p}$. Allora

$$z(\alpha - \beta) = 0 \pmod{p}$$

Ossia, $z(\alpha - \beta)$ è divisibile per p . Ma per ipotesi $z \neq 0 \pmod{p}$, quindi z non è divisibile per p . Essendo p un primo, quindi, deve essere necessariamente che $\alpha - \beta$ risulti essere divisibile per p , ossia

$$\alpha - \beta = 0 \pmod{p} \implies \alpha = \beta \pmod{p}$$

Tale lemma risulta necessario per la dimostrazione del seguente teorema

■ Teorema

La classe di funzioni hash $\mathcal{H} = \{h_a : a \in \mathcal{A}\}$ è universale.

■ Dimostrazione

Siano $x = (x_1, x_2, \dots, x_r)$ e $y = (y_1, y_2, \dots, y_r)$ due elementi distinti in U .

Si vuole mostrare che la probabilità che $\Pr[h_a(x) = h_a(y)]$, per un $a \in \mathcal{A}$ scelto casualmente, è al più $1/p$.

Essendo $x \neq y$ allora deve esistere un indice j tale per cui $x_j \neq y_j$.

Si sceglie ora un vettore casuale $a \in \mathcal{A}$ nella seguente maniera:

si scelgono prima tutte le coordinate a_i dove $i \neq j$. Successivamente, si sceglie la coordinata a_j casualmente.

Si può quindi assumere che a_i sia fissato per tutte le coordinate $i \neq j$.

Si mostra quindi come, indipendentemente dalla scelta di tutte le altre coordinate a_i , la probabilità che $h_a(x) = h_a(y)$, considerando la scelta finale di a_j , sia esattamente $1/p$.

Da ciò seguirà che la probabilità di $h_a(x) = h_a(y)$ per la scelta casuale di tutto il vettore a dovrà necessariamente essere al più $1/p$.

Ciò è chiaro intuitivamente: se la probabilità è $1/p$ indipendentemente dalla scelta di tutti gli altri a_i , allora sarà $1/p$ in generale.

Si da una dimostrazione diretta usando la probabilità condizionata:

Sia \mathcal{E} l'evento

$$h_a(x) = h_a(y)$$

e sia \mathcal{F}_b l'evento

Tutte le coordinate a_i (per $i \neq j$) ricevono una sequenza di valori b .

Si vuole mostrare che $\Pr[\mathcal{E} | \mathcal{F}_b] = 1/p$ per ogni b . Da ciò seguirà che

$$\Pr[\mathcal{E}] = \sum_b \Pr[\mathcal{E} | \mathcal{F}_b] \cdot \Pr[\mathcal{F}_b] = (1/p) \sum_b \Pr[\mathcal{F}_b] = 1/p$$

Si assuma quindi che tutti i valori per le coordinate a_i (eccetto a_j) siano state scelte arbitrariamente, e si consideri la probabilità di selezionare un a_j tale per cui $h_a(x) = h_a(y)$. Riordinando i termini, si osserva che

$$h_a(x) = h_a(y) \iff a_j(y_j - x_j) = \sum_{i \neq j} a_i(x_i - y_i) \pmod{p}$$

Essendo le scelte per tutte le coordinate a_i ($i \neq j$) fissate, si può vedere la parte destra dell'equazione come una quantità fissata α . Sia ora $z = y_j - x_j$.

Bisogna mostrare che esiste esattamente un solo valore $0 \leq a_j < p$ che soddisfa $a_j z = \alpha \pmod{p}$. Infatti, così facendo, essendo a_j scelto uniformly at random da \mathbb{Z}_p , allora la probabilità di scegliere tale valore per a_j è esattamente $1/p$.

Si supponga quindi che esistano due valori che soddisfino l'equazione vista in precedenza, siano essi a_j ed a'_j , ossia

$$a_j z = \alpha \pmod{p}$$

$$a'_j z = \alpha \pmod{p}$$

Dunque si avrebbe che $a_j z = a'_j z \pmod{p}$, e per il lemma enunciato in precedenza, si avrebbe $a_j = a'_j \pmod{p}$. Ma essendo per ipotesi $a_j, a'_j < p$, allora a_j ed a'_j sono esattamente lo stesso numero. Da ciò segue che esiste un singolo a_j in \mathbb{Z}_p che soddisfa $a_j z = \alpha \pmod{p}$.

Questo vuol dire che la probabilità di scegliere a_j tale per cui $h_a(x) = h_a(y)$ è $1/p$, indipendentemente dalla scelta delle altre coordinate a_i in a . Quindi la probabilità che x e y collidano è $1/p$, e ciò dimostra che \mathcal{H} è una classe di funzioni hash universale.

Osservazione

Si vuole mostrare esplicitamente che

$$h_a(x) = h_a(y) \iff a_j(y_j - x_j) = \sum_{i \neq j} a_i(x_i - y_i) \pmod{p}$$

Si ricorda per definizione che

$$h_a(x) = \sum_{i=1}^r a_i x_i \quad \text{e} \quad h_a(y) = \sum_{i=1}^r a_i y_i$$

Quindi

$$\begin{aligned} h_a(x) = h_a(y) &\iff h_a(x) - h_a(y) = 0 \iff \sum_{i=1}^r a_i x_i - \sum_{i=1}^r a_i y_i = 0 \pmod{p} \\ &\iff \sum_{i=1}^r a_i (x_i - y_i) = 0 \pmod{p} \iff \sum_{i=1, i \neq j}^r a_i (x_i - y_i) + a_j (x_j - y_j) = 0 \pmod{p} \\ &\iff \sum_{i=1, i \neq j}^r a_i (x_i - y_i) = -a_j (x_j - y_j) \pmod{p} \iff \sum_{i=1, i \neq j}^r a_i (x_i - y_i) = a_j (y_j - x_j) \pmod{p} \end{aligned}$$

Si conclude osservando che, mediante l'utilizzo di una funzione appartenente ad \mathcal{H} per l'implementazione del dizionario, si usa uno spazio pari a $\Theta(n)$.

L'esecuzione delle operazioni relative a tale struttura dati hanno costo $\mathcal{O}(1)$, questo perché la memorizzazione di $h_a \in \mathcal{H}$ richiede la memorizzazione di una singola chiave a , che richiede spazio costante, e il calcolo di tale funzione per ogni elemento in U , che avviene per l'esecuzione di ogni operazione definita per il dizionario, richiede tempo costante.

■ Osservazione

Nel modello word RAM manipolare un numero costante di parole macchina richiede tempo costante, ed ogni oggetto è descrivibile da una parola macchina. Memorizzare $h_a(x)$ richiede memorizzare un singolo valore a , che occupa 1 parola macchina. Calcolare $h_a(x)$ richiede tempo costante.

Per la prima funzione hash universale, sono richiesti $r \log p$ bits ed essendo $r \geq \log(N)/\log p$, si ha $\log(N) = \mathcal{O}(1)$ essendo N costante.

K-wise independent hashing

Siano gli N elementi dell'insieme universo U gli interi da 0 ad $N - 1$, ossia $U = \{0, 1, \dots, N - 1\}$. In questa sezione vengono trattate le funzioni hash su interi definite come $h : \{0, 1, \dots, N - 1\} \rightarrow \{0, 1, \dots, m - 1\}$. Il k -wise independent hashing è una versione più debole dell'hashing uniforme.

■ Definizione (Famiglia di funzioni hash k -wise independent)

Si dice che una famiglia di funzioni hash \mathcal{H} è k -wise independent se e solo se, per una scelta uniforme di $h \in \mathcal{H}$, si ha che

$$\Pr \left[\bigwedge_{i=1}^k h(x_i) = y_i \right] = m^{-k}$$

per ogni scelta di $x_1, \dots, x_k \in \{0, 1, \dots, N-1\}$ distinti e $y_1, \dots, y_k \in \{0, 1, \dots, m-1\}$ non necessariamente distinti.

Quindi, una famiglia \mathcal{H} uniforme è il caso particolare in cui $k = N$. Ciò implica che:

- Per ogni scelta di x_1, \dots, x_t tutti distinti con $t \geq k$, le variabili aleatorie $h(x_1), \dots, h(x_t)$ sono k -wise indipendenti.
- h mappa le k -tuple uniformemente: la k -tupla $(h(x_1), \dots, h(x_k))$ è una variabile aleatoria uniforme su $\{0, 1, \dots, m-1\}^k$ quando x_1, \dots, x_k sono distinti.

Osservazione

La 2-wise indipendenza (o pairwise indipendenza) di una famiglia di funzioni hash implica la sua universalità.

Si consideri il sottoinsieme $\{h(x_2) = y\}_{y \in [m]}$ dello spazio di campionamento

$$\Omega = \bigcup_{i=0}^{N-1} \left(\{h(x_i) = y\}_{y \in [m]} \right).$$

Per il teorema della probabilità totale e per la 2-wise indipendenza si ha che

$$\begin{aligned} \Pr[h(x_1) = h(x_2)] &= \sum_{y=0}^{m-1} \Pr[h(x_1) = h(x_2) \wedge h(x_2) = y] \\ &\sum_{y=0}^{m-1} \Pr[h(x_1) = y \wedge h(x_2) = y] = \sum_{y=0}^{m-1} m^{-2} = \frac{1}{m} \end{aligned}$$

Si mostra ora la costruzione una famiglia di funzioni pairwise indipendente.

Sia $p \geq N$ un numero primo. Si definisce la funzione hash $h_{a,b} : U \longrightarrow \{0, 1, \dots, p-1\}$ come

$$h_{a,b}(x) = (a \cdot x + b) \pmod{p}$$

Si definisce la famiglia di funzioni hash $\hat{\mathcal{H}}$ come segue

$$\hat{\mathcal{H}} = \{h_{a,b} : a, b \in \{0, 1, \dots, p-1\}\}$$

In altre parole, una funzione $h_{a,b} \in \hat{\mathcal{H}}$ uniforme è un polinomio di grado 1 in \mathbb{Z}_p scelto uniformemente al random. Tale funzione è specificata completamente da a, b, p e quindi può essere memorizzata in $\mathcal{O}(\log p)$ bits. Inoltre, può essere calcolata in tempo $\mathcal{O}(1)$. Si mostra ora che tale famiglia è pairwise indipendente:

■ Lemma

$\hat{\mathcal{H}}$ è una famiglia di funzioni hash pairwise indipendente.

■ Dimostrazione

Siano $x_1, x_2 \in \{0, 1, \dots, N - 1\}$ due elementi tali per cui $x_1 \neq x_2$ e siano $y_1, y_2 \in \{0, 1, \dots, p - 1\}$.

Si osserva esplicitamente che essendo $x_1 \neq x_2$ e $p \geq N$, allora $x_1 \not\equiv_p x_2$.

Allora, si ha che

$$\Pr[h(x_1) = y_1 \wedge h(x_2) = y_2] = \Pr[ax_1 + b \equiv_p y_1 \wedge ax_2 + b \equiv_p y_2]$$

$$= \Pr[b \equiv_p y_1 - x_1 \cdot \frac{y_2 - y_1}{x_2 - x_1} \wedge a \equiv_p \frac{y_2 - y_1}{x_2 - x_1}] \quad (1)$$

$$= \Pr[b \equiv_p y_1 - x_1 \cdot \frac{y_2 - y_1}{x_2 - x_1}] \cdot \Pr[a \equiv_p \frac{y_2 - y_1}{x_2 - x_1}] \quad (2)$$

$$= p^{-2}$$

Dove

(1) è dato risolvendo il seguente sistema nelle variabili a e b :

$$\begin{cases} ax_1 + b \equiv_p y_1 \\ ax_2 + b \equiv_p y_2 \end{cases}$$

Si osserva che $(x_2 - x_1)^{-1}$ esiste perché $x_2 \not\equiv_p x_1$ e \mathbb{Z}_p è un campo, quindi ogni elemento (eccetto 0) ha un inverso moltiplicativo. [Nota: vedere questa questione]
(2) a e b sono variabili aleatorie indipendenti.

■ Osservazione

I termini noti del sistema nella dimostrazione sono x_1, x_2, y_1, y_2 , mentre a, b sono variabili aleatorie, e sono l'incognita del sistema.

La funzione $h_{a,b}$ appena definita non può essere utilizzata per l'implementazione di un dizionario. Questo perché $p \geq N$, di conseguenza risulterebbe necessario un array H di dimensione p , e ciò sarebbe di dimensioni spropositate. Si definisce quindi un'altra funzione hash partendo da $h_{a,b}$:

Sia $m << N$ la dimensione dell'hash table e sia $p > N$ un numero primo, dove si ricorda che $U = [N]$ e quindi $|U| = N$. Siano $a \in \{1, \dots, p - 1\}$ e $b \in \{0, 1, \dots, p - 1\}$ due valori scelti

uniformly at random dai rispettivi insiemi di appartenenza.

Si definisce la funzione hash:

$$\bar{h}(x) = ((a \cdot x + b) \bmod p) \bmod m$$

e sia

$$\bar{\mathcal{H}} = \{\bar{h} \mid 1 \leq a \leq p-1, 0 \leq b \leq p-1\}$$

Lemma

$\bar{\mathcal{H}}$ è una famiglia di funzioni hash universale.

Dimostrazione

Si conta il numero di funzioni in $\bar{\mathcal{H}}$ per le quali due elementi distinti $x_1, x_2 \in U$ collidono.

Si osserva inizialmente che, per ogni $x_1 \neq x_2$ vale

$$ax_1 + b \neq ax_2 + b \pmod{p}$$

Questo è vero perché $ax_1 + b = ax_2 + b \pmod{p}$ implica che $a(x_1 - x_2) = 0 \pmod{p}$, ma sia a che $(x_1 - x_2)$ sono diversi da 0 modulo p .

Infatti, per ogni coppia di valori (u, v) tali per cui $u \neq v$ e $0 \leq u, v \leq p-1$, esiste esattamente una sola coppia di valori (a, b) per i quali $ax_1 + b = u \pmod{p}$ e $ax_2 + b = v \pmod{p}$

Questa coppia di equazioni ha due incognite, e l'unica soluzione è data da

$$a = \frac{v-u}{x_2-x_1} \pmod{p}$$
$$b = u - ax_1 \pmod{p}$$

Essendovi quindi esattamente una funzione hash per ogni coppia (a, b) , segue che vi è esattamente una funzione hash in $\bar{\mathcal{H}}$ tale per cui

$$ax_1 + b = u \pmod{p} \quad \text{e} \quad ax_2 + b = v \pmod{p}$$

Quindi, per dare un bound alla probabilità che $\bar{h}(x_1) = \bar{h}(x_2)$ quando \bar{h} è scelta u.a.r. da $\bar{\mathcal{H}}$, è sufficiente contare il numero di coppie (u, v) , $0 \leq u, v \leq p-1$ tali per cui $u \neq v$ ma $u = v \pmod{m}$.

Per ogni scelta di u vi sono al più $\lceil p/m \rceil - 1$ possibili valori appropriati per v , dando quindi al più

$$p(\lceil p/m \rceil - 1) \leq \frac{p(p-1)}{m}$$

coppie. Ogni coppia corrisponde ad una tra le possibili $p(p-1)$ funzioni hash, quindi

$$\Pr \left[\bar{h}(x_1) = \bar{h}(x_2) \right] \leq \frac{p(p-1)/m}{p(p-1)} = \frac{1}{m}$$

dimostrando che $\bar{\mathcal{H}}$ è universale.

Collision-free hashing

In alcune applicazioni risulta necessaria una funzione hash *collision-free*

Definizione (Funzione hash collision-free)

Una funzione hash $h : \{0, 1, \dots, N-1\} \rightarrow \{0, 1, \dots, m-1\}$ si dice collision free su un insieme $A \subseteq \{0, 1, \dots, N-1\}$ se, per ogni $x_1, x_2 \in A, x_1 \neq x_2$, si ha $h(x_1) \neq h(x_2)$.

In generale, basta che una funzione soddisfi tale proprietà con alta probabilità. Si riporta nuovamente tale definizione

Definizione (Alta probabilità)

Si dice che un evento accade con alta probabilità rispetto ad una quantità n se la sua probabilità è almeno $1 - n^{-c}$ per una costante c arbitrariamente grande.

Equivalentemente, si dice che l'evento accade con probabilità che decresce come l'inverso di un polinomio.

Scegliere dinamicamente la dimensione dell'hash table

Si fa presente come, nel problema del dizionario, S è un insieme dinamico, ossia subisce modifiche nel tempo, e si vuole utilizzare spazio pari a $\mathcal{O}(|S|)$.

Si vuole mostrare una tecnica per scegliere la dimensione della tabella hash in maniera dinamica.

Siano

- n : numero di elementi correntemente nella tabella, ossia $n = |S|$.
- N : dimensione virtuale della tabella/dimensione dell'universo
- m : dimensione reale della tabella, dove m è un primo tale per cui $N \leq m \leq 2N$.

Double/Halving technique:

1. Inizializza $n = N = 1$;
2. Quando $n > N$: //dopo che sono stati aggiunti elementi in S
21. Poni $N = 2N$;

22. Scegli un nuovo primo m tale per cui $m \in \Theta(n)$;
23. Ri-esegui il calcolo della funzione hash per tutti gli elementi; //in tempo $\mathcal{O}(n)$ per via del fatto che il costo ammortizzato di ogni inserimento/cancellazione è $\mathcal{O}(1)$
3. Quando $n < N/4$:
 31. Poni $N = N/2$;
 32. Scegli un nuovo primo m tale per cui $m \in \Theta(n)$;
 33. Ri-esegui il calcolo della funzione hash per tutti gli elementi;

Perfect hashing (Double hashing)

Si considera il problema del dizionario statico:

Dato un insieme S di n elementi (o chiavi) da un universo U di dimensione N , si vuole costruire una struttura dati di dimensione $\mathcal{O}(n)$ che supporti le operazioni di ricerca (dato $x \in U$, sapere se $x \in S$) in tempo costante. Si vuole un tempo di costruzione atteso per tale struttura dati che sia polinomiale con alta probabilità.

L'idea è quella di costruire una tabella a due livelli:

- Passo 1: Pesca uniformemente a caso una funzione hash $h_1 : U \rightarrow [m]$ da una famiglia di funzioni hash universale per $m = \Theta(n)$ (ad esempio m numero primo vicino ad n). Esegui l'hashing su tutti gli elementi di S usando h_1 , ossia costruisci un'array H di liste concatenate e inserisci ogni $x \in S$ in posizione $H[h_1(x)]$.
- Passo 2: Per ogni $j \in [m]$, sia l_j il numero di elementi in posizione j in H .

$$l_j = |\{i \mid h(x_i) = j\}|$$

Pesca uniformemente a caso una funzione hash $h_{2,j} : U \rightarrow [m_j]$ da una famiglia di funzioni hash universale per $l_j^2 \leq m_j \leq \mathcal{O}(l_j^2)$ (ad esempio m_j numero primo vicino ad esso). Sostituisce la lista concatenata in posizione j con una tabella hash H_j di dimensione m_j , mappando gli elementi in posizione j di tale lista in H_j utilizzando $h_{2,j}$.

La complessità spaziale è

$$\mathcal{O}\left(n + \sum_{j=0}^{m-1} l_j^2\right)$$

Per ridurla a $\mathcal{O}(n)$ bisogna aggiungere due passi intermedi:

- Passo 1.5: Se $\sum_{j=0}^{m-1} l_j^2 > cn$ dove c è una costante scelta, riesegui il passo 1.
- Passo 2.5: Quando $h_{2,j}(u) = h_{2,j}(v)$ per ogni $u, v \in S$ tali per cui $u \neq v$ e $h_1(u) = h_1(v)$, ossia quando si verifica una collisione al secondo livello di hash, pesca una nuova funzione $h_{2,j}$ e rimappa tutti gli l_j elementi in H_j .

Questi due passi garantiscono che non si verifichino collisioni al secondo livello, e che la complessità spaziale sia di $\mathcal{O}(n)$. Ciò garantisce che il tempo di ricerca di un elemento sia

$\mathcal{O}(1)$.

Si osserva esplicitamente che risulta improbabile che si verifichino le condizioni espresse nel passo 2.5, essendovi $\Theta(l_j^2)$ celle per mappare l_j elementi.

Si studia ora il tempo di costruzione della struttura dati impiegato dall'algoritmo.

Ricordando che il calcolo di una delle funzioni hash universali viste in precedenza richiede tempo costante, si ha che i passi (1) e (2) richiedono tempo $\mathcal{O}(n)$.

Per il passo (2.5) si ha che

$$\begin{aligned} \mathbf{Pr}_{h_{2,j}} [h_{2,j}(u) = h_{2,j}(v), \text{ per } u \neq v] &\leq \sum_{u,v \in S, u \neq v} \mathbf{Pr} [h_{2,j}(u) = h_{2,j}(v)] \\ &\leq \binom{l_j}{2} \cdot \frac{1}{l_j^2} < \frac{1}{2} \end{aligned}$$

Dove si osserva esplicitamente che, per l'universalità di $h_{2,j}$

$$\mathbf{Pr} [h_{2,j}(u) = h_{2,j}(v)] \leq \frac{1}{l_j^2}$$

Quindi, ogni prova è come un lancio di moneta. Se l'esito è "testa", si passa allo step successivo. Si ha quindi che $\mathbf{E} [\text{numero di prove}] \leq 2$, essendo il numero di prove una variabile aleatoria avente distribuzione geometrica, e $(\text{numero di prove}) = \mathcal{O}(\log n)$ con alta probabilità. Ogni prova richiede tempo $\mathcal{O}(l_j)$, che, per un Chernoff bound (rivedere quale è fare la dim per esercizio), $l_j = \mathcal{O}(\log n)$ con alta probabilità, quindi ogni prova richiede tempo $\mathcal{O}(\log n)$.

Dovendo compiere tale passo per ogni j , si ha che la complessità temporale totale è $\mathcal{O}(\log n) \cdot \mathcal{O}(\log n) \cdot \mathcal{O}(n) = \mathcal{O}(n \log^2 n)$ con alta probabilità.

Per il passo (1.5), si vuole mostrare che $\mathbf{E} \left[\sum_{j=0}^{m-1} l_j^2 \right] = \Theta(n)$, applicando poi la disuguaglianza di Markov.

Si definisce quindi la variabile aleatoria

$$X_{u,v} = \begin{cases} 1 & \text{se } h_1(v) = h_1(u) \\ 0 & \text{altrimenti} \end{cases}$$

■ Osservazione

$$\sum_{j=0}^{m-1} l_j^2 = \sum_{u \in S} \sum_{v \in S} X_{u,v}$$

Si ha quindi che

$$\begin{aligned}\mathbf{E} \left[\sum_{j=0}^{m-1} l_j^2 \right] &= \mathbf{E} \left[\sum_{u \in S} \sum_{v \in S} X_{u,v} \right] = \sum_{u \in S} \sum_{v \in S} \mathbf{E} [X_{u,v}] \\ &= \sum_{u \in S} \sum_{v \in S} \mathbf{Pr} [h_1(u) = h_1(v)] \quad (1)\end{aligned}$$

$$\leq \sum_{u \in S} \left[1 + \frac{n}{m} \right] \quad (2)$$

$$= n + \frac{n^2}{m} \leq 2n \quad (3)$$

Dove:

(1) $\mathbf{E} [X_{u,v}] = \mathbf{Pr} [h_1(u) = h_1(v)]$.

(2) Si ha che $\sum_{v \in S} X_{u,v} = X_u$ dove $X_u = |\{v \in S \mid h_1(v) = h_1(u)\}|$ è la v.a. che conta gli elementi in S mappati in $h_1(u)$. Si ha quindi

$$\sum_{v \in S} \mathbf{E} [X_{u,v}] = \mathbf{E} [X_u] \leq 1 + \frac{n}{m}$$

essendo h_1 universale.

(3) Vero perché $m \geq n$

Per la diseguaglianza di Markov si ha

$$\mathbf{Pr} \left[\sum_{j=0}^{m-1} l_j^2 > cn \right] \leq \frac{\mathbf{E} \left[\sum_{j=0}^{m-1} l_j^2 \right]}{cn} \leq \frac{2n}{cn} \leq 1/2$$

Scegliendo un c adatto, ad esempio $c \geq 4$.

Si ha quindi che $\mathbf{E} [\text{numero di prove}] \leq 2$ e $(\text{numero di prove}) = \mathcal{O}(\log n)$ con alta probabilità. Da ciò, si ha che il passo (1) ed il passo (1.5) combinati richiedono tempo atteso $\mathcal{O}(n \log n)$ con alta probabilità.

Similar items

Molti problemi possono essere espressi come "trovare insiemi simili".

Ad esempio:

1. Pagine con una grande frazione di parole simili in esse
 - Per l'individuazione di duplicati, classificazione in base all'argomento
2. Clienti che comprano una grande frazione di prodotti simili
 - Prodotti con insiemi dei clienti simili
3. Immagini con caratteristiche simili

Scene completion Problem

Dato un insieme di immagini \mathbf{I} di dimensione N molto grande, ed un'immagine target i_t , si vuole trovare all'interno di tale insieme l'immagine $i \in \mathbf{I}$ più simile rispetto al target i_t .

Le immagini in input possono venir rappresentate mediante una successione di N punti in alta dimensione x_1, x_2, \dots, x_N , dove l'immagine x è rappresentata mediante un lungo vettore di colori di pixel, tale per cui a ciascun colore è associato univocamente un intero da 1 a c , ad esempio:

$$x = \begin{bmatrix} 1 & 4 & 1 \\ 0 & 2 & 1 \\ 0 & 1 & 0 \end{bmatrix} \longrightarrow [1 \ 4 \ 1 \ 0 \ 2 \ 1 \ 0 \ 1 \ 0] = \in \{0, 1, \dots, c\}^h$$

dove h è la dimensione dei dati.

Dati due data points x_1 ed x_2 si vuole quantificare quanto essi siano "distanti", ossia simili tra loro. Per fare ciò si introduce una funzione distanza $d(x_1, x_2)$.

Si vogliono quindi trovare tutte le coppie di punti (x_i, x_j) che si trovano entro una data soglia di distanza s , ossia tali per cui $d(x_i, x_j) \leq s$.

La soluzione banale, ossia quella di controllare tutte le possibili coppie, richiederebbe tempo $\mathcal{O}(N^2h)$, essendo N il numero di data points ed h la loro dimensione.

Tale approccio, nonostante rispetti le delimitazioni polinomiali della trattabilità, risulta nella pratica proibitivo nel caso in cui il data set in input risulti essere di elevate dimensioni. Basti pensare ad un data set di un milione di oggetti, dove l'approccio banale richiederebbe l'analisi di mezzo trilione di coppie.

Ciò può essere fatto in tempo $\mathcal{O}(Nh')$, con $h' \ll h$ con alta confidenza.

Trovare documenti simili

Si studia il problema di trovare documenti simili all'interno di un data set di grandi dimensioni.

Dato quindi un numero N di documenti, nell'ordine di milioni o anche miliardi, si vogliono individuare le coppie di documenti simili.

Per fare ciò, bisogna inizialmente chiarire il concetto di *similarità* tra documenti: l'aspetto di similarità tra documenti ricercato in questo problema è a livello di caratteri, e non di significato. La similarità testuale ha diversi utilizzi, molti dei quali permettono di individuare documenti duplicati o "quasi duplicati". Si osserva che, dati due documenti, controllare se questi siano esattamente duplicati risulta possibile controllando semplicemente i due documenti carattere per carattere, e se questi differiscono allora non sono uguali. Bisogna tener presente che, in molte applicazioni, i documenti non risultano essere identici, ma nonostante ciò, condividono una grande porzione del loro testo. Alcuni esempi più significativi sono

- Plagio: Un'applicazione per testare la similarità testuale è quella dell'individuazione di documenti plagiati. Da un documento originale possono essere estratte solo singole parti, le quali possono venir utilizzate in un altro documento frutto di plagio. Potrebbero venir modificate in questo alcune parole, o l'ordine delle frasi nelle quali queste appaiono nel testo originale. Nonostante ciò, il documento frutto di questa operazione risulta contenere molto dell'originale. Nessun processo semplice di confronto tra documento carattere per carattere risulta essere in grado di identificare un plagio sofisticato.
- Mirror pages: Le pagine web possono venir duplicati in un maggior numero di host, affinché il loro carico venga distribuito. Le pagine di questi siti *mirror* saranno simili, ma raramente identiche. Ad esempio, ognuna di esse può contenere informazioni specifiche associate con il particolare host che le fornisce in rete, e ciascuna di esse può contenere riferimenti ad altri siti mirror. E' importante individuare pagine simili di questo tipo, affinché i motori di ricerca producano miglior risultati evitando di mostrare due pagine che sono quasi identiche.

Sono diverse le difficoltà tecniche che si possono incontrare al fine di risolvere questo problema, tra cui:

- Come nel plagio, molte porzioni brevi di un documento possono apparire in ordine differente in un altro.
- Come detto in precedenza, un data set di grandi dimensioni non permette il confronto tra tutte le possibili coppie di documenti.
- I documenti possono essere di un numero e di una dimensione così elevata che non risulta possibile rappresentarli nella memoria principale.

Per individuare quindi le similarità tra documenti, bisogna definire il concetto di distanza tra essi. Per fare ciò si fa riferimento ad una particolare nozione di "similarità" insiemistica, che, dati due insiemi guarda la dimensione relativa della loro intersezione.

Questa particolare similarità prende il nome di *Jaccard similarity*.

Definizione (Jaccard Similarity)

Dati due insiemi S e T , la loro *Jaccard similarity* è data dal rapporto tra la dimensione della loro intersezione e la dimensione della loro unione:

$$\text{J.sim}(S, T) = \frac{|S \cap T|}{|S \cup T|}$$

A partire da questo concetto di similarità, si definisce una particolare distanza tra due documenti.

Definizione (Jaccard Distance)

Dati due insiemi S e T , la loro *Jaccard distance* è definita come

$$d(S, T) = 1 - \text{J.sim}(S, T)$$

Essendo tale misura applicabile ad insiemi di elementi, risulta necessario rappresentare i documenti, ossia stringhe di lunghezza finita, a livello insiemistico.

Sia U un'alfabeto. Un documento Doc è una stringa di caratteri, ossia $Doc \in U^*$. Il metodo più efficace per rappresentare documenti come insiemi, con l'obiettivo di identificare documenti lessicalmente simili, consiste nel costruire dal documento un insieme di stringhe di lunghezza limitata che appaiono all'interno di esso. Così facendo, i documenti che condividono delle parti nella forma di frasi, anche se queste appaiono in ordine differente, avranno molti elementi in comune. Tale approccio prende il nome di *shingling*.

Schematicamente, il processo di individuazione di documenti simili può essere descritto come segue:

1. **Input:** Si ha un universo di documenti $Doc \in U^*$ di grandi dimensioni, dove U è l'alfabeto.
2. **Shingling:** Processo al fine di convertire documenti in insiemi di grandi dimensioni.
3. **Min-Hashing:** Processo al fine di convertire insiemi di grandi dimensioni in *signatures* di dimensioni ridotte, permettendo di stimare le *J. similarity* dei relativi insiemi.
4. **Locality-Sensitive Hashing:** Processo al fine di restringere il numero di confronti da effettuare per le coppie di documenti. Permette di concentrarsi su coppie di *signatures* provenienti probabilmente da documenti simili, evitando il costo quadratico dei confronti nel numero dei documenti.
5. **Output:** Coppie candidate di documenti.

Si studiano quindi i vari passi al fine della risoluzione del problema.

Shingling

Si vogliono rappresentare i documenti mediante dati ad alta dimensione.

Un documento è una stringa di caratteri. Si definisce un k -shingle per un documento come una sottostringa di lunghezza k trovata al suo interno. Individuando tali k -shingle, si può associare a ciascun documento l'insieme di k -shingles che appaiono una o più volte dentro di esso.

Analogamente, un k -shingle può essere definito come segue:

Definizione (k -shingle)

Un k -shingle per un documento D è una sequenza di k tokens che appare all'interno di D .

I tokens possono essere caratteri, parole o altro, a seconda dell'applicazione.

Sia U l'universo di tutti i possibili tokens.

Sia S la funzione che prende in input un documento e restituisce l'insieme dei suoi k -shingles per un dato k .

Si fa un esempio: Sia D il documento costituito dalla stringa `abcdabd`, sia $k = 2$ e siano i tokens nel documento dei caratteri. Allora, per D , il suo insieme di 2-shingles è $S(D) = \{\text{ab}, \text{bc}, \text{cd}, \text{da}, \text{bd}\}$.

Si osserva che la sottostringa `ab` appare due volte nel documento, ma solo una volta come shingle. E' possibile definire una variante dello shingling che produce dei multiset, che prendono il nome di *bag*. In questo caso, ogni shingle appare nel risultato tante volte quante appare all'interno del documento. Per l'esempio precedente, si ha che

$$S(D) = \{\text{ab}, \text{bc}, \text{cd}, \text{da}, \text{ab}, \text{bd}\}.$$

Ogni documento $D \in U^*$, dove U è l'alfabeto di caratteri che costituiscono i documenti, viene rappresentato quindi come l'insieme dei k -shingles $C = S(D)$ presenti al suo interno. Equivalentemente, può essere visto come un lungo vettore binario $S(D) = C \in \{0, 1\}^{U^k}$ avente una componente per ogni elemento dell'insieme U^k di tutti i possibili k -shingle. Ogni singolo shingle definisce una dimensione (ossia una componente) del vettore C , (si ha quindi una dimensione per ciascun elemento (shingle) nell'universo U^k).

Se l' i -esimo k -shingle dell'insieme ordinato U^k appartiene al documento D , allora l' i -esima entrata di C ha valore 1, altrimenti ha valore 0. Si fa presente come questi vettori risultino essere molto sparsi.

Data quindi la rappresentazione di documenti mediante questi insiemi, si fa riferimento al concetto di *J. similarity* per individuarne le similarità testuali.

E' possibile, dato un data set di documenti, dare un'interpretazione probabilistica della *J. sim.* Siano D_1 e D_2 due documenti, si ha che

$$\text{J. sim}(D_1, D_2) = \frac{|C_1 \cap C_2|}{|C_1 \cup C_2|} = \mathbf{Pr} [|C_1 \cap C_2| \mid |C_1 \cup C_2|]$$

Dove $C_i = S(D_i)$ per $i = 1, 2$.

I documenti aventi parecchi k -shingles in comune hanno testo simile, anche se questi appaiono all'interno di essi in ordine differente.

Bisogna prestare particolare attenzione alla scelta di k . Nonostante questa possa essere una qualsiasi costante, con la scelta di un k troppo piccolo ci si aspetta di trovare la maggior parte delle sequenze di k caratteri in molteplici documenti. Ciò implica che, per la metrica di valutazione della similarità utilizzata, si possono avere documenti per i quali i relativi insiemi di shingles possiedono alta *Jaccard similarity* nonostante questi non abbiano nessuna frase in comune.

Ad esempio, per $k = 1$, la maggior parte di pagine presenti sul Web risulterebbero simili tra

loro, avendo queste la maggior parte dei caratteri che le costituiscono in comune. La scelta di k dovrebbe dipendere a seconda della tipica lunghezza dei documenti e di quanto grande è il tipico insieme di caratteri che le costituisce. In particolar modo,

- k deve essere scelto sufficientemente grande affinché la probabilità che un qualsiasi shingle appaia in un qualsiasi documento è bassa.
In generale, si ha che:
- $k = 5$ risulta sufficiente per documenti brevi, come emails.
- $k = 10$ va bene per documenti di grande dimensione.

Min-Hashing

Gli insiemi di shingles sono di grande dimensione. Se si hanno milioni di documenti, potrebbe non risultare possibile memorizzare tutti gli shingle-sets nella memoria principale. Si vogliono convertire quindi insiemi di grandi dimensioni in rappresentazioni molto più piccole, che prendono il nome di *firme* (signatures), preservandone la similarità.

Risulta fondamentale perciò che, dal confronto delle firme di due insiemi, si possa stimare la Jaccard similarity di tali insiemi facendo riferimento solo ad esse. Non è possibile che le firme diano l'esatta similarità degli insiemi che rappresentano, ma le stime che forniscono sono vicine ad essa, e maggiore è la dimensione delle firme, più accurata è la stima.

Per fare ciò, si usa la tecnica del **Min-Hashing**.

Si supponga di dover trovare documenti simili in un dataset di dimensione $N = 10^6$, e quindi di disporre di 10^6 insiemi di k -shingles. Il Min-Hashing non permette di evitare i $\Theta(N^2)$ confronti tra le possibili coppie, ma rende ciascun confronto (dove ciascun confronto consiste nel calcolo della J. similarity), molto più efficiente.

Si fa presente come, siano C_1 e C_2 le rappresentazioni vettoriali ad alta dimensione di due documenti D_1 e D_2 per l'insieme di tutti i possibili k -shingles definiti su un'alfabeto U .

Allora $C_1, C_2 \in \{0, 1\}^{U^k}$, e il confronto banale tra essi comporta un costo di $\Theta(|U|^k)$, che per un alfabeto di dimensione $|U| = 27$ e $k = 10$ risulta eccessivo.

Prima di spiegare come sia possibile costruire piccole firme da grandi dimensioni, si fa presente che risulta utile visualizzare una collezione di insiemi mediante la loro *matrice caratteristica*. Le colonne della matrice corrispondono agli insiemi, e le righe corrispondono agli elementi dell'insieme universo dal quale vengono presi gli elementi costituenti gli insiemi stessi.

Per essere più chiari, nel nostro scenario, sia U l'alfabeto dei caratteri costituenti i documenti da prendere in analisi e sia k il valore per il quale vengono definiti i k -shingles. Allora, la matrice caratteristica possederà una colonna per ogni insieme di k -shingle associato al relativo documento $S(D_1) = S_1, S(D_2) = S_2, \dots, S(D_N) = S_N$, dove si fa presente che $S_i \subseteq U^k$ per ogni $i \in \{1, 2, \dots, N\}$, ed una riga per ogni elemento dell'insieme universo U^k . Nell'entrata (r, C) di tale matrice è presente 1 se l'elemento dell'universo nella riga r appartiene all'insieme associato alla colonna C , 0 altrimenti. Ad esempio:

<i>Element</i>	<i>S</i> ₁	<i>S</i> ₂	<i>S</i> ₃	<i>S</i> ₄
<i>b</i>	0	0	1	0
<i>e</i>	0	0	1	0
<i>a</i>	1	0	0	1
<i>d</i>	1	0	1	1
<i>c</i>	0	1	0	1

■ Osservazione

Data la rappresentazione di un documento mediante un vettore ad alta dimensione, tale matrice caratteristica è formata dalle rappresentazioni vettoriali di tutti i documenti presi in analisi.

Mediante questa rappresentazione, l'intersezione tra due insiemi è ottenuta eseguendo l'AND bitwise tra i bit appartenenti alle rispettive colonne, e l'unione mediante l'operazione OR bitwise. La dimensione di un insieme corrisponde al numero di entrate della rispettiva colonna che posseggono valore 1. La similarità tra due colonne è data dalla J. similarity dei corrispondenti insiemi, calcolabile eseguendo le operazioni appena descritte.

Prendendo di nuovo come esempio la matrice in figura, si vuole calcolare la similarità di *S*₁ ed *S*₃. Sia $C(S_i) = C_i$ la sequenza di bit contenuta nella colonna associata ad *S*_{*i*}. Si ha che $C_1 = 00110$, $C_3 = 11010$. Allora

$$(C_1 \text{ AND } C_3) = 00010 = C_{AND} \quad \text{e} \quad (C_1 \text{ OR } C_3) = 11110 = C_{OR}$$

$$\text{J.sim}(C_1, C_3) = \frac{|C_{AND}|}{|C_{OR}|} = \frac{1}{5}$$

dove $|C_i| = \#$ di entrate in C_i aventi valore 1.

E' importante tenere a mente che la matrice caratteristica è una maniera utile per visualizzare i dati, ma non risulta un buon metodo per memorizzarli. Uno dei motivi principali è che queste matrici sono quasi sempre sparse (hanno molti più entrate a 0 che 1). Una rappresentazione di tale matrice per risparmiare spazio ottenibile memorizzando le posizioni nelle quali appaiono le entrate poste ad 1.

Si è quindi data una rappresentazione dei documenti mediante sottoinsiemi di shingles, e tali sottoinsiemi sono stati rappresentati mediante colonne binarie di una matrice.

Il prossimo obiettivo è quello di trovare colonne simili calcolando delle signatures per tali

colonne, individuando la similarità per quelle di dimensione ridotta.

In particolar modo, si vuole definire un buon algoritmo di firme tale per cui la J. similarity delle colonne risulta non essere troppo distante dalla "similarità" delle firme.

Si segue il seguente approccio:

1. Si definiscono le firme delle colonne.
2. Si esaminano coppie di firme per trovare colonne simili. E' essenziale che le similarità delle firme e delle relative colonne siano correlate
3. Opzionalmente, controllare che le colonne con firme simili siano realmente simili
Si fa presente che con questo approccio:

- Controllare tutte le coppie può richiedere troppo tempo. Si gestirà tale problematica con la tecnica del locality sensitive hashing
- Nel caso in cui non venisse eseguito il passo 3, questo metodo può produrre falsi positivi (colonne non simili risultano essere simili mediante il confronto delle relative firme). L'approccio adottato può produrre falsi negativi (colonne simili risultano non essere simili mediante il confronto delle relative firme), i quali non possono venir individuati, essendo che i confronti vengono effettuati solamente per documenti aventi firme simili. Se si volessero effettuare confronti anche per le coppie di documenti aventi firme non simili, si effettuerebbero allora i confronti tra tutte le possibili coppie, e tale operazione richiederebbe tempo $\Theta(N^2)$.

Sia ora $|U^k| = m$ la dimensione dell'universo di k -shingles definiti su alfabeto U .

L'idea chiave è quindi quella di effettuare l'hash mediante una funzione h di ciascuna colonna C per ottenere una firma $h(C)$ tale per cui:

1. $h(C)$ è sufficientemente piccola affinché questa sia memorizzabile nella RAM, in particolar modo $|h(C)| \ll |C| = m$
2. Siano C_1 e C_2 due colonne. Allora $J.\text{sim}(C_1, C_2)$ è vicina all'equivalenza delle firme $h(C_1)$ ed $h(C_2)$.

Si deve trovare quindi una funzione $h(\cdot)$ tale per cui:

- Se $J.\text{sim}(C_1, C_2)$ è alta, allora, con alta probabilità, $h(C_1) = h(C_2)$.
- Se $J.\text{sim}(C_1, C_2)$ è bassa, allora, con alta probabilità, $h(C_1) \neq h(C_2)$.

La scelta della funzione hash dipende dalla metrica di similarità adottata.

Per la Jaccard Similarity, la funzione hash adatta prende il nome di MinHash.

Le firme che si vogliono costruire per gli insiemi sono composte dal risultato di un gran numero di computazioni, ciascuna delle quali è un "minhash" della matrice caratteristica. Per eseguire il minhash di un insieme, rappresentato da una colonna della matrice caratteristica, bisogna scegliere una permutazione delle righe. Il valore di minhash di una colonna è il numero della prima riga, in ordine permutato, nella quale la colonna ha come entrata 1.

La Minhash hash function può essere definita come segue

Definizione (MinHash hash function)

Sia M la matrice caratteristica del data set preso in analisi. Sia π una permutazione scelta casualmente delle righe di M . Sia m il numero di righe di M .

Per ciascuna colonna C di M , si definisce la funzione hash MinHash $h_\pi(C)$ come

$$h_\pi(C) = \min\{i \in [m] : C_{\pi[i]} = 1\} = \min(\pi(C))$$

Ossia l'indice della prima riga, secondo l'ordinamento indotto dalla permutazione π , in cui C ha valore 1, dove C_π è la colonna C ordinata secondo π .

Esiste un'importante connessione tra il minhashing e la J. Similarity degli insiemi sottoposti a tale procedura:

- La probabilità che la funzione MinHash per una permutazione casuale delle righe produca lo stesso valore per due insiemi è uguale alla Jaccard similarity di tali insiemi.

Ciò può essere formalizzato mediante il seguente teorema:

Teorema (J.Similarity preserving)

Siano C_1 e C_2 due colonne. Sia π una permutazione scelta uniformly at random. Allora

$$\Pr_\pi[h_\pi(C_1) = h_\pi(C_2)] = \text{J.sim}(C_1, C_2)$$

Prima di dimostrare il teorema, si da un'argomentazione informale relativa ad esso.

Si considerino le colonne C_1 e C_2 per i rispettivi insiemi S_1 ed S_2 . Le righe di queste due colonne possono essere divise in tre classi:

1. Righe di tipo X , aventi 1 in entrambe le colonne.
2. Righe di tipo Y , aventi 1 in una delle due colonne e 0 nell'altra.
3. Righe di tipo Z , aventi 0 in entrambe le colonne.

Essendo la matrice sparsa, la maggior parte delle righe sono di tipo Z . Nonostante ciò, è il rapporto del numero di righe di tipo X e di tipo Y che determina sia $\text{J.sim}(C_1, C_2)$, sia la probabilità che $h_\pi(C_1) = h_\pi(C_2)$. Sia x il numero di righe di tipo X ed y il numero di righe di tipo Y . Allora si ha che

$$\text{J.sim}(C_1, C_2) = \frac{x}{x + y}$$

Questo è vero perché x è la dimensione di $S_1 \cap S_2$, ossia il numero di righe nelle due colonne tali per cui i valori contenuti in esse sono uguali, ed $x + y$ è la dimensione di

$S_1 \cup S_2$.

Le righe di tipo Z non vanno prese in considerazione per questo tipo di conteggio, perché i rispettivi shingles associati ad esse non sono contenuti in nessuno dei due insiemi presi in considerazione.

Si consideri ora la probabilità che $h_\pi(C_1) = h_\pi(C_2)$. Se si considerano le righe permutate casualmente, partendo dal primo elemento permutato e seguendo l'ordinamento indotto dalla permutazione, la probabilità di incontrare una riga di tipo X prima di incontrare una riga di tipo Y è proprio $x/(x+y)$, perché su un numero totale di $x+y$ righe prese in questione, quelle di tipo X sono proprio x . Ma se la prima riga incontrata, escluse quelle di tipo Z , è una riga di tipo X , allora sicuramente $h_\pi(C_1) = h_\pi(C_2)$. Se invece la prima riga incontrata, escluse quello di tipo Z , è di tipo Y , allora la colonna con il valore 1 assume tale riga come indice di minhash. Ma la colonna con lo 0 in quella riga sicuramente avrà, successivamente ad essa, una riga contenente 1 nella lista permutata. Quindi $h_\pi(C_1) \neq h_\pi(C_2)$ se si incontra una riga di tipo Y prima di una riga di tipo X . Si conclude quindi che la probabilità che $h_\pi(C_1) = h_\pi(C_2)$ è $x/(x+y)$, che risulta essere anche la Jaccard similarity delle due colonne (e quindi dei due rispettivi insiemi).

Per dimostrare formalmente tale teorema, si fa riferimento alla seguente variabile aleatoria

Definizione (MinHash Estimator)

Siano S_1 e S_2 due insiemi di shingles e siano C_1 e C_2 le rispettive rappresentazioni mediante vettore binario ad alta dimensione. Sia π una permutazione. Si definisce MinHash estimator la seguente variabile aleatoria

$$J_\pi(S_1, S_2) = \begin{cases} 1 & \text{se } h_\pi(C_1) = h_\pi(C_2) \\ 0 & \text{altrimenti} \end{cases}$$

E si dimostra il seguente lemma, la quale implica la dimostrazione del teorema.

Lemma

Se π è una permutazione uniforme, allora $\mathbf{E}[J_\pi(S_1, S_2)] = \text{J.sim}(S_1, S_2)$

Dimostrazione

Siano S_1 ed S_2 due insiemi di shingles, e siano C_1 e C_2 le rispettive rappresentazioni mediante vettore binario ad alta dimensione.

Sia $|S_1 \cup S_2| = n$. Per un $i \in S_1 \cup S_2$ si consideri l'evento

$$s(i) = (\forall j \in (S_1 \cup S_2) \setminus \{i\})(\pi(i) < \pi(j))$$

che afferma che i è l'elemento di $S_1 \cup S_2$ mappato nel più piccolo indice $\pi(i)$ tra tutti gli elementi di $S_1 \cup S_2$. Essendo π una permutazione, esattamente un elemento di $S_1 \cup S_2$ verrà mappato nel più piccolo indice (ossia, $s(i)$ è vero per esattamente un $i \in S_1 \cup S_2$), allora $\{s(i)\}_{i \in S_1 \cup S_2}$ è una partizione di cardinalità $n = |S_1 \cup S_2|$ dello spazio degli eventi. Inoltre, essendo che π è completamente uniforme implica che

$$\mathbf{Pr}[s(i)] = \mathbf{Pr}[s(j)]$$

per ogni $i, j \in S_1 \cup S_2$, ossia, ogni elemento di $S_1 \cup S_2$ ha la stessa probabilità di venir mappato nel più piccolo indice. Questo implica che $\mathbf{Pr}[s(i)] = 1/n$ per ogni $i \in S_1 \cup S_2$. Si osserva che, se $s(i)$ fosse vero ed $i \in S_1 \cap S_2$, allora $J_\pi(S_1, S_2) = 1$, perché i appartiene ad entrambi gli insiemi e π raggiunge il suo minimo \min per i , quindi $h_\pi(C_1) = h_\pi(C_2) = \min$.

In alternativa, se $s(i)$ fosse vero e $i \in (S_1 \cup S_2) \setminus (S_1 \cap S_2)$ allora $J_\pi(S_1, S_2) = 0$, perché i apparterrebbe solamente ad uno tra S_1 e S_2 , e non ad entrambi, e π raggiunge il suo minimo \min per i , quindi si ha avrebbe che $h_\pi(C_1) \neq h_\pi(C_2) = \min$ o, alternativamente, $\min = h_\pi(C_1) \neq h_\pi(C_2)$.

Usando questa osservazione ed applicando la legge delle aspettative iterate alla partizione $\{s(i)\}_{i \in S_1 \cup S_2}$ dello spazio degli eventi, si ottiene

$$\begin{aligned} \mathbf{E}[J_\pi(S_1, S_2)] &= \sum_{i \in S_1 \cup S_2} \mathbf{Pr}[s(i)] \cdot \mathbf{E}[J_\pi(S_1, S_2) | s(i)] \\ &= \sum_{i \in S_1 \cup S_2} \frac{1}{n} \cdot \mathbf{E}[J_\pi(S_1, S_2) | s(i)] \\ &= \sum_{i \in S_1 \cap S_2} \frac{1}{n} \cdot \mathbf{E}[J_\pi(S_1, S_2) | s(i)] + \sum_{i \in (S_1 \cup S_2) \setminus (S_1 \cap S_2)} \frac{1}{n} \cdot \mathbf{E}[J_\pi(S_1, S_2) | s(i)] \\ &= \sum_{i \in S_1 \cap S_2} \frac{1}{n} \cdot 1 + \sum_{i \in (S_1 \cup S_2) \setminus (S_1 \cap S_2)} \frac{1}{n} \cdot 0 \\ &= \frac{1}{n} \cdot \sum_{i \in S_1 \cap S_2} 1 \\ &= \frac{1}{n} |S_1 \cap S_2| \\ &= \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|} = \mathbf{J}.\mathbf{sim}(S_1, S_2) \end{aligned}$$

Il teorema dimostra quindi che $\mathbf{J}.\mathbf{sim}(S_1, S_2) = \mathbf{E}[J_\pi(S_1, S_2)]$. Essendo $J_\pi(S_1, S_2)$ una variabile aleatoria binomiale, si ha che

$$\begin{aligned}\mathbf{E}[J_\pi(S_1, S_2)] &= \mathbf{Pr}[h_\pi(C_1) = h_\pi(C_2)] \cdot 1 + \mathbf{Pr}[h_\pi(C_1) \neq h_\pi(C_2)] \cdot 0 \\ &= \mathbf{Pr}[h_\pi(C_1) = h_\pi(C_2)]\end{aligned}$$

e quindi

$$\mathbf{Pr}[h_\pi(C_1) = h_\pi(C_2)] = \text{J.sim}(S_1, S_2) = \text{J.sim}(C_1, C_2)$$

Si pensi ancora ad una collezione di insiemi rappresentata dalla loro matrice caratteristica M .

Nella pratica, si combinano parecchie (ad esempio 100) funzioni hash scelte uniformly at random (ossia, permutazioni scelte casualmente), e quindi mutualmente indipendenti tra loro, per creare una firma per una colonna C della matrice. Questo approccio permette di ottenere maggior confidenza.

Si supponga quindi di scegliere un numero $t >> 1$ di permutazioni delle righe di M . Siano le funzioni di minhash determinate da queste permutazioni $h_{\pi 1}, h_{\pi 2}, \dots, h_{\pi t}$. Dalla colonna C , rappresentante l'insieme S , si definisce la *minhash signature* di S come il vettore

$$SIG(C) = [h_{\pi 1}(C), h_{\pi 2}(C), \dots, h_{\pi t}(C)]$$

Si rappresenta questo vettore come una colonna. Quindi, è possibile formare, a partire dalla matrice M , una *matrice di firme*, nella quale la i -esima colonna di M è sostituita dalla minhash signature per l'insieme dell' i -esima colonna.

La matrice di firme ha lo stesso numero di colonne di M ma solamente t righe. Anche se M non è rappresentata esplicitamente, ma in una forma compressa adatta ad una matrice sparsa (ad esempio, tenendo traccia della posizione delle entrate poste ad 1), è normale che la matrice delle firme risulti essere molto più piccola di M .

L'aspetto fondamentale delle matrici di firme è dato dal fatto che risulta possibile utilizzare le loro colonne per stimare la Jaccard similarity degli insiemi ad esse corrispondenti. Dal teorema enunciato in precedenza, si ha che la probabilità che due colonne abbiano lo stesso valore in una data riga della matrice delle firme è uguale alla J. similarity degli insiemi corrispondenti a queste colonne. Inoltre, essendo le permutazioni sulle quali questi valori di minhashing si basano state scelte indipendentemente, si può pensare a ciascuna di queste righe della matrice di firme come un esperimento indipendente. Quindi, il valore atteso del numero di righe nelle quali due colonne assumono lo stesso valore è uguale alla Jaccard similarity dei corrispettivi insiemi. In particolar modo, si definisce il seguente concetto di similarità tra firme:

Definizione (Signatures similarity)

Siano C_1 e C_2 due colonne, si definisce la similarità di due vettori di firme $SIG(C_1)$ e $SIG(C_2)$ come

$$SignSim(C_1, C_2) = \frac{|\{j : h_{\pi j}(C_1) = h_{\pi j}(C_2)\}|}{t}$$

Ossia la frazione delle min-hash signatures nelle quali C_1 e C_2 hanno lo stesso elemento

Si fa presente che $\text{J.sim}(C_1, C_2) \neq \text{SignSim}(C_1, C_2)$, ma per la proprietà del Min Hash descritto nel teorema, si ha che $\text{J.sim}(C_1, C_2) \equiv \mathbf{E}_\pi[\text{SignSim}(C_1, C_2)]$, quindi, maggiore è t , ossia più minhashings si usano, maggiori saranno le righe nella matrice di firme, e minore sarà l'errore atteso nella stima della Jaccard similarity.

Alla luce di quanto appena detto, si da il seguente corollario

☠ Corollario

La $\text{SignSim}(C_1, C_2)$ di due colonne tende alla $\text{J.sim}(C_1, C_2)$ al crescere di t .

Si da ora uno pseudocodice che descrive la procedura di MinHashing:

RndAlgorithm Doc-Pair Check

Input : Colonne $C_1, C_2 \in \{0, 1\}^h$, parametro di confidenza t

for $j = 1$ to t :

Scegli u.a.r. una permutazione $\pi_j \in \Pi_m$

Calcola le funzioni hash $h_{\pi_j}(C_1)$ e $h_{\pi_j}(C_2)$

return $\text{SignSim}(C_1, C_2) = |\{j : h_{\pi_j}(C_1) = h_{\pi_j}(C_2)\}|/t$

📘 Osservazione

Essendo che le t funzioni hash sono mutualmente indipendenti, si hanno risultati di concentrazione su $|\text{SignSim}(C_1, C_2) - \text{J.sim}(C_1, C_2)|$.

Si analizza ora la complessità spaziale del MinHash.

Data la matrice di firme, $SIG(C)$ è un vettore colonna di tale matrice associato alla colonna C della matrice caratteristica.

Sia $SIG(i, C)$ l' i -esima entrata di $SIG(C)$, ossia l'indice della prima riga che ha valore 1 nella colonna C secondo la i -esima permutazione casuale. Per rappresentare quindi univocamente un singolo intero che assume valori tra 1 e $|C|$, tale $SIG(i, C)$, sono necessari $\Theta(\log |C|) = \Theta(\log m)$ bits (dove $m = |C| = |U^k|$ essendovi in C un'entrata per ogni possibile k -shingle).

Essendo che, per una colonna C , la matrice di firme contiene t elementi, ossia le t funzioni della forma $h_{\pi_i}(C)$ per $i = 1, \dots, t$, si ha che il costo in memoria dello sketch, ossia la signature, $SIG(C)$ è ordine di $\Theta(t \log m)$. Si è quindi raggiunto l'obiettivo di "comprimere" lunghi vettori di bit avente complessità spaziale $\Theta(m)$ in firme di dimensione $\Theta(t \log m)$.

Si apre una digressione sul termine *sketch*.

Sia x una variabile che rappresenta dei dati, come ad esempio un insieme, una stringa o un intero. Un *data sketch* è l'output di una funzione casuale f (in generale è data dalla combinazione di un certo numero di funzioni hash) che mappa x in una sequenza di bit $f(x)$ che rispetta le proprietà 1-3 elencate di seguito, ed eventualmente, a seconda dell'applicazione, anche la 4.

1. La dimensione in bit di $f(x)$ è molto più piccola della dimensione in bit di x (di solito sublineare o polilogaritmica).
2. $f(x)$ può essere usata per calcolare, in maniera efficiente, delle proprietà di x . Ad esempio, se x è un multi set, allora $f(x)$ può essere usata per calcolare un'approssimazione del numero di elementi distinti contenuti in x , o l'elemento che appare con più occorrenze in x .
3. $f(x)$ può essere aggiornato efficientemente se x viene aggiornato. E' fondamentale che risulti possibile aggiornare $f(x)$ senza conoscere x . Ad esempio:
 - Se si aggiunge un elemento y ad un insieme x , deve essere possibile calcolare $f(x \cup \{y\})$ conoscendo solamente $f(x)$ ed y (e non x).
 - In generale, dati due sketches $f(x_1)$ ed $f(x_2)$, deve risultar possibile calcolare lo sketch della composizione di x_1 ed x_2 mediante un qualche operatore. Ad esempio, se x_1 e x_2 sono insiemi, si può essere interessati ad ottenere lo sketch di $f(x_1 \cup x_2)$ senza conoscere x_1 e x_2 .
4. Se x ed y sono simili rispetto ad una misura di similarità (Jaccard distance, distanza euclidea ...), allora $f(x)$ e $f(y)$ devono essere simili con alta probabilità.

Osservazione

Essendo f una funzione casuale, $f(x)$ è una variabile aleatoria.

Dalla definizione di data sketch, si osserva esplicitamente che l'output delle operazioni eseguite dalla procedura di min hashing rientra in questa categoria.

Nella pratica, le permutazioni sono ottenute mediante utilizzo di funzioni hash.

Questo perché, per la descrizione della procedura di minhashing, bisogna generare diverse permutazioni π casuali ed indipendenti tra loro dell'insieme U^k . Ma eseguire anche una singola permutazione delle U^k righe della matrice caratteristica risulta troppo costoso in termini di tempo. Ordinare mediante una funzione hash h restituisce una permutazione π (quasi) casuale.

Di conseguenza, è possibile simulare l'effetto di una permutazione casuale utilizzando una funzione hash casuale che mappa ogni indice di riga in tanti buckets quanti sono le righe. Si fa presente che una funzione hash $f : [m] \rightarrow [m]$ può mappare interi distinti nello stesso bucket, e lasciarne altri vuoti. E' possibile non tener conto di ciò finché m è di grande dimensione e il numero di collisioni non è eccessivo.

La scelta della funzione casuale, per rendere bassa la probabilità di collisioni, può essere fatta facendo riferimento all'Universal hashing, ossia si scelgono t funzioni della forma

$$h_{a,b}(x) = ((a \cdot x + b) \mod p) \mod m$$

dove a, b sono interi casuali e p è un numero primo tale che $p > m$.

Si fornisce quindi il seguente pseudocodice

Alg SignMatrix

```
1. foreach colonna  $C_l$  per  $l = 1, \dots, N$  e funzione hash  $f_i$  per  $i = 1, \dots, t$  do
   | 2. Inizializza la matrice di firme  $SIG(i, C) = \infty$ 
   |
   3. foreach riga  $j = 1, \dots, m$  della matrice originale  $M$  do
      | 4. foreach colonna  $C_l$  con  $l = 1, \dots, N$  do
         | 5. if  $M(j, C) = 1$  then
            | 6. foreach funzione hash  $f_i$  per  $i = 1, \dots, t$  do
               | 7. Calcola  $f_i(j)$ 
               | 8. if  $f_i(j) < SIG(i, C)$  then aggiorna  $SIG(i, C) \leftarrow f_i(j)$ 
```

Si fa ora un'analisi dell'algoritmo

Il ciclo principale (righe 3 – 8) impiega $\Theta(mN)$ iterazioni.

Ciascuna di queste iterazioni richiede t calcoli di funzioni hash $f_i(j)$ se e solo se $M(j, C) = 1$, ma essendo M sparsa, ciò è improbabile che sia vero.

Il problema principale di questo approccio è dato dal fatto che l'hash può causare collisioni, non portando una permutazione, ma come detto in precedenza si può non tenerne conto.

Notazione

M : matrice identità

SIG : matrice delle firme costruita da M mediante minhashing

$SIG(C)$: Colonna della matrice delle firme associata al documento C .

$SIG(i, C)$: i -esima entrata della colonna nella matrice delle firme associata al documento C .

Locality-sensitive Hashing

Il Minhash permette quindi di comprimere documenti di grandi dimensioni in firme di piccole dimensioni, e di preservare la similarità attesa di ogni coppia di documenti. Nonostante ciò, nel caso in cui il data set preso in analisi abbia un grande numero di elementi al suo interno, può risultare impossibile trovare le coppie di documenti con la maggior similarità in maniera efficiente.

Si supponga di voler trovare le coppie di documenti simili in un data set avente dimensione N . L'approccio banale consiste nel calcolare le J. similarities per ogni coppia di documenti e,

nonostante il minhashing permetta di calcolare tale misura di similarità in maniera molto più efficiente, non evita i $\Theta(N^2)$ confronti tra coppie, che per data sets di grandi dimensioni risulta essere un numero di operazioni eccessivamente grande.

In particolar modo, se l'obiettivo è quello di calcolare le similarità di *ogni coppia* di documenti, non si può fare a meno di effettuare tale numero di confronti.

Nella maggior parte dei casi, invece, si vogliono trovare le coppie che risultano essere simili entro una certa soglia di similarità s . In tal caso, è possibile concentrarsi solamente su coppie di documenti i quali risultano essere probabilmente simili, senza dover effettuare il confronto per ogni possibile coppia. Per fare ciò si fa riferimento alla tecnica del *locality-sensitive hashing*, che permette di individuare coppie di firme che, probabilmente, derivano da documenti tra loro simili.

L'idea generale del LSH è quella di utilizzare una funzione $f(x, y)$ che afferma se x ed y , dove x ed y sono due documenti, è una coppia di candidati, ossia una coppia di documenti la quale similarità deve venir valutata in maniera più approfondita.

Si studia una forma specifica di LSH adatta al problema studiato. Si fa presente che nella sua trattazione, si assume che i documenti siano stati già rappresentati mediante firme.

Un approccio generale al LSH è quello di eseguire più volte l'hash degli oggetti in analisi (le firme di ciascun documento nel nostro caso), in maniera tale che quelli simili vengano mappati nello stesso bucket con maggior probabilità rispetto a quelli tra loro dissimili. Si considerano poi tutte le coppie di oggetti mappate nello stesso bucket da almeno una procedura di hashing come *coppie candidate*, le quali saranno le uniche ad essere soggette al calcolo della similarità degli insiemi che costituiscono ciascuna di esse.

In particolar modo, sia $0 < s < 1$ una soglia di similarità. Si definisce formalmente il concetto di coppia candidata.

Definizione

Siano x ed y due documenti appartenenti al data set preso in analisi, e siano $SIG(x)$ e $SIG(y)$ le colonne nella matrice di firme associate rispettivamente ai due documenti, ottenute mediante la procedura di minhashing.

I documenti x ed y sono coppie di candidati se le colonne $SIG(x)$ e $SIG(y)$ assumono lo stesso valore per almeno una frazione s delle loro righe, ossia

$$\frac{|\{i \in [t] : SIG(i, x) = SIG(i, y)\}|}{t} \geq s$$

dove $SIG(i, C)$ è l' i -esima entrata della colonna C nella matrice di firme.

Si spera che la maggior parte di coppie di documenti tra loro dissimili non vengano mai mappate, da nessuna delle funzioni hash usate dalla procedura, nello stesso bucket, e che quindi non vengano controllate. Le coppie dissimili che vengono mappate nello stesso

bucket prendono il nome di *falsi positivi*, e si vuole che queste siano una piccola frazione tra tutte le possibili coppie.

Si vuole inoltre che la maggior parte di coppie veramente simili vengano mappate nello stesso bucket per almeno una delle funzioni hash utilizzate. Le coppie per cui ciò non accade vengono chiamate *falsi negativi*, e anche per questi si vuole che siano una piccola frazione delle coppie veramente simili.

Il problema di questo approccio è dato dal fatto che, eseguire l'hash dell'intera firma di un documento, ossia dell'intera colonna della matrice delle firme associata al documento in questione, in un bucket può generare diversi falsi positivi e falsi negativi.

Per ovviare a questo problema, si divide la matrice di firme in b bands (un sottoinsieme di righe adiacenti) composte ciascuna da r righe.

Per ogni band si definisce una funzione hash, che prende il vettore di r interi di una colonna della matrice delle firme (ossia la porzione di una colonna all'interno di tale band), e lo mappa in una hash table avente c buckets, dove c è abbastanza grande rispetto ad r per evitare, con alta probabilità, la collisione di firme differenti. Si può usare la stessa funzione hash per tutte le bands, ma si usa un array di bucket separato per ciascuna di esse, in maniera tale che le colonne con vettori uguali in bands differenti non vengano mappate nello stesso bucket.

Se i sottovettori di due colonne contenuti in una data bands finiscono nello stesso bucket, allora i due documenti associati a tali colonne sono probabilmente J. simili, e rientrano quindi nella categoria di coppie candidate.

Le coppie di colonne candidate sono quelle che vengono mappate nello stesso bucket per almeno una band.

Si fa ora un'assunzione che semplifica l'analisi dell'algoritmo:

- Ci sono abbastanza buckets ($c >> r$) in maniera tale che sia improbabile che le colonne vengano mappate nello stesso bucket a meno che queste siano identiche in una particolare band.
Si assume quindi che due vettori vengano mappati nello stesso bucket se e solo se questi sono identici.

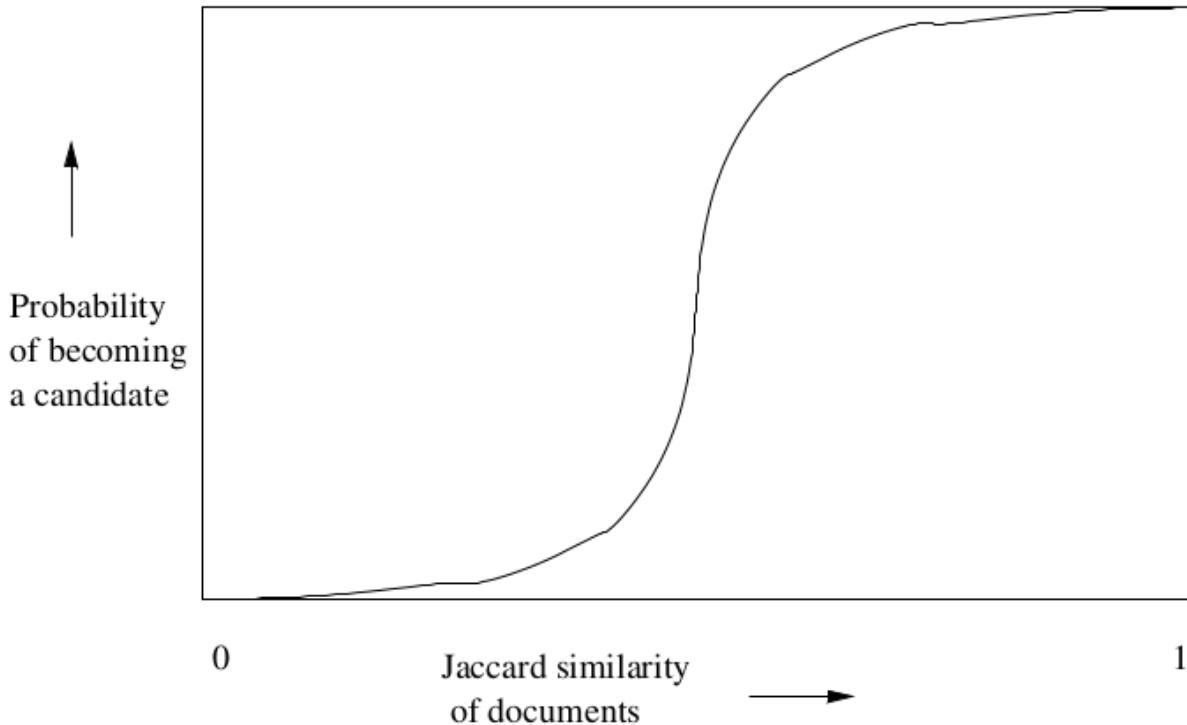
Si supponga quindi di dividere la matrice delle firme in b bands, ciascuna avente r righe, e si supponga che una particolare coppia di documenti abbia Jaccard similarity s . Si ricorda che la probabilità che le firme ottenute mediante minhashing per questi documenti assumono lo stesso valore in una data riga della matrice delle firme è proprio s .

Si può calcolare la probabilità che questi due documenti, o meglio, le loro firme, diventino una coppia di candidati come segue:

1. La probabilità che le firme assumano lo stesso valore in tutte le righe di una particolare band è s^r .

2. La probabilità che le firme siano diverse in almeno una riga di una particolare band è $1 - s^r$.
3. La probabilità che le firme siano diverse in almeno una riga di ogni band è $(1 - s^r)^b$.
4. La probabilità che le firme assumano lo stesso valore in tutte le righe di almeno una band, e quindi diventino una coppia candidata, è $1 - (1 - s^r)^b$.

Indipendentemente dalle costanti b e r scelte, questa funzione assume la forma di una *curva ad S*.



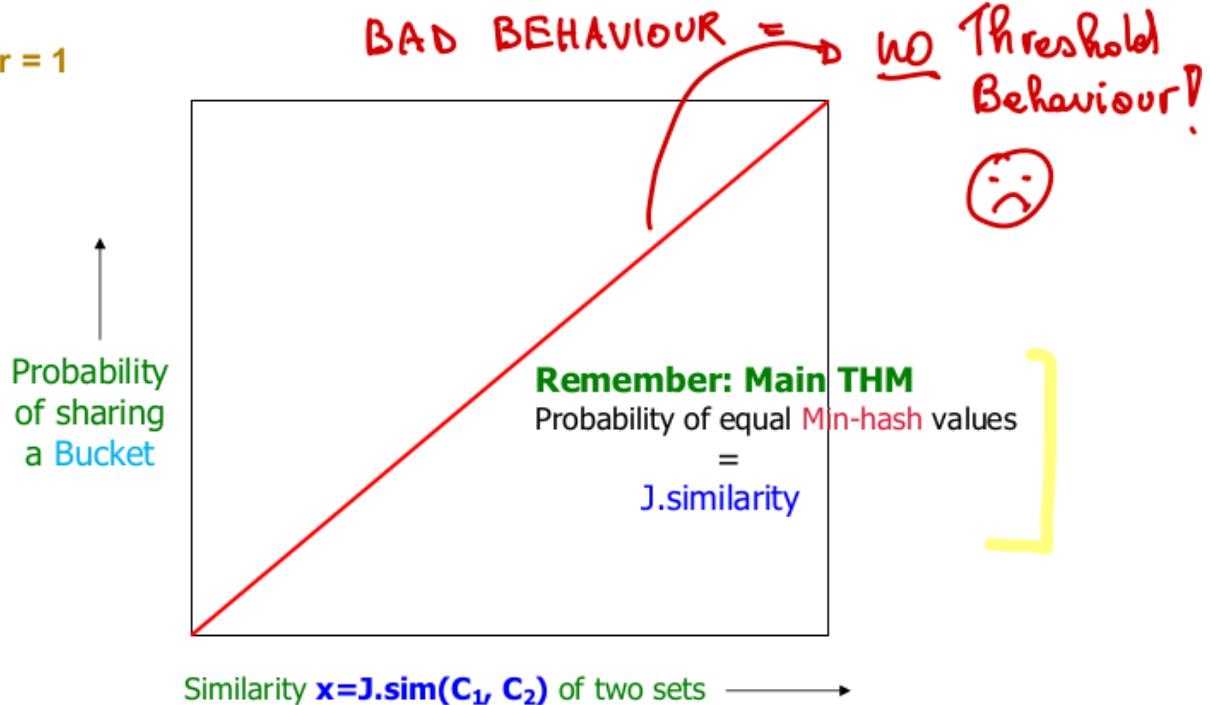
La soglia, ossia il valore della similarità s per il quale la probabilità che una coppia diventi un candidato è $1/2$, è in funzione di b ed r . La soglia è circa dove la funzione è "più ripida", e per b ed r di grandi dimensioni si ha che per le coppie aventi similarità sopra alla soglia è molto probabile che questi diventano candidati, mentre per quelli sotto alla soglia è improbabile che questi lo diventino. Un'approssimazione buona per la soglia è $(1/b)^{1/r}$.

Si ha quindi che, per una soglia fissata s , il quale è un parametro in input del problema, bisogna impostare i seguenti parametri dell'algoritmo:

- Il parametro t del numero di Minhashes da eseguire (che saranno quindi il numero di righe della matrice delle firme)
- I parametri b ed r , in maniera tale che $br = t$, e in maniera tale che si ottengano quasi tutte coppie con firme simili, ma che vengano anche eliminate le coppie per cui le firme non lo sono. La scelta più bilanciata per b ed r si ha quando $(1/b)^{1/r} \approx s$. Aumentare b diminuisce il numero di falsi negativi, mentre aumentare r diminuisce il numero di falsi positivi.

Si evidenziano i casi particolari per $r = 1$:

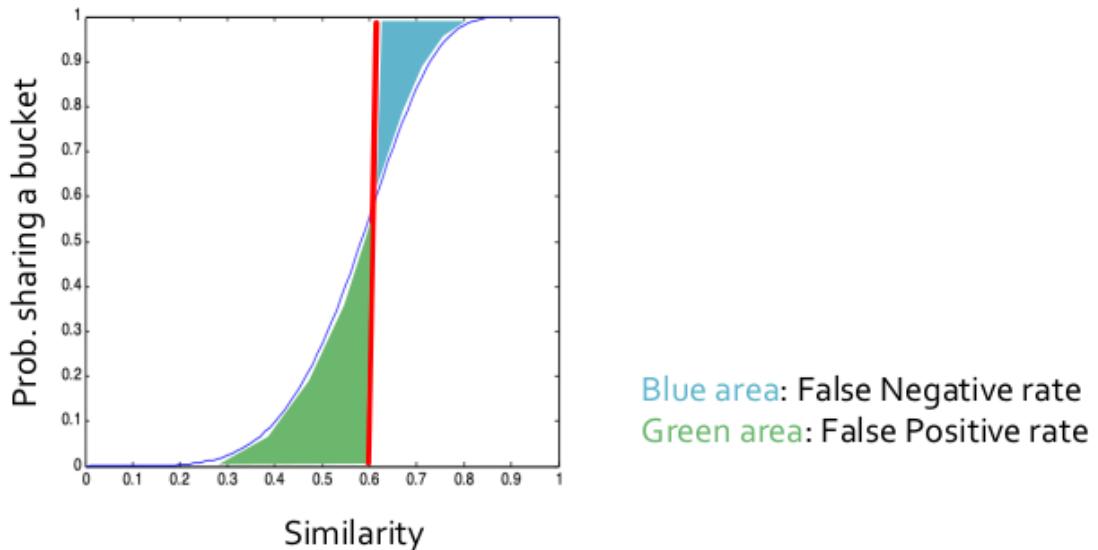
Case $r = 1$



e per la scelta ottimale dei parametri.

Picking r and b to get the best S-curve

- t= 50 hash-functions ($r=5, b=10$)



Si fa un riassunto generale della tecnica studiata:

- Obiettivo:** Dati in input le t colonne per ciascun documento (ossia la matrice di firme), e la soglia di similarità s (con $0 < s \leq 1$), si vogliono trovare tutte le coppie di documenti (x, y) tali per cui $\text{SignSim}(x, y) \geq s$.
- Soluzione mediante LSH:** Applicare la tecnica della divisione in bande dell'LSH alla matrice di firme, impostando i parametri b ed r affinché $b \cdot r = t$ e $(1/b)^{1/r} \approx s$.

- **Output:** tutte le coppie di documenti candidate aventi la stessa firma in *tutte* le r righe di *almeno* una band.

Combinazione delle tecniche

Si può ora dare un approccio per trovare l'insieme di coppie candidate di documenti simili, per poi individuare i documenti veramente simili tra esse. Si enfatizza che questo approccio può produrre falsi negativi, ossia coppie di documenti simili che non vengono identificate come tali per via del fatto che non rientrano nei possibili candidati, e falsi positivi, coppie di candidati che al termine della procedura vengono analizzati, ma per i quali si ha che non risultano sufficientemente simili.

1. Scegli un valore di k e costruisci da ciascun documento l'insieme di k -shingles.
2. Scegli una lunghezza t per le firme da ottenere mediante la procedura di minhash, e fornisci all'algoritmo di minhashing gli insiemi per calcolare le firme di minhash per tutti i documenti.
3. Scegli una soglia s che definisce quanto simili debbano essere i documenti affinché possano essere considerate "coppie simili". Scegli un numero di bands b ed un numero r di righe tali per cui $br = t$ e $s \approx (1/b)^{1/r}$. Se risulta più importante evitare i falsi negativi, si possono impostare i parametri b ed r per produrre una soglia minore di t . Se risulta di maggior importanza la velocità e si vogliono limitare i falsi positivi, scegli b ed r per produrre una soglia più grande.
4. Costruisci le coppie di candidati applicando la tecnica del locality sensitive hashing.
5. Esamina le firme di ogni coppia di candidati e determina se la frazione dei componenti per i quali i valori assunti sono uguali è almeno t .
6. Opzionalmente, se le firme sono sufficientemente simili, analizza i documenti in questione per verificare che questi siano veramente simili.

Data Stream

In molte situazioni di data mining, non si conosce preventivamente l'intero data set. Ossia, tutti i dati presi in esame non risultano essere disponibili ad ogni istante.

Si vuole studiare lo scenario in cui i dati arrivano in uno o più flussi, tali per cui questi risultino perduti nel caso in cui non vengano processati o memorizzati immediatamente nell'istante di arrivo. Si assume che questi flussi arrivino così rapidamente che non risulti possibile memorizzarli tutti in una memoria attiva, ossia in un database convenzionale.

I dati possono essere visti come infiniti e non stazionari, tali per cui quindi la loro distribuzione cambia nel tempo.

Gli algoritmi per processare flussi di dati richiedono che il flusso in input venga, in qualche modo, "schematizzato". Si studia quindi come poter raccogliere un campione di dati all'interno di un flusso affinché questo risulti essere utile, e come filtrare un flusso per eliminare gli elementi indesiderati. A partire da ciò, si mostra come stimare il numero di

elementi differenti all'interno di un flusso utilizzando molta meno memoria rispetto a quella richiesta nel caso in cui si volessero listare tutti gli elementi visti.

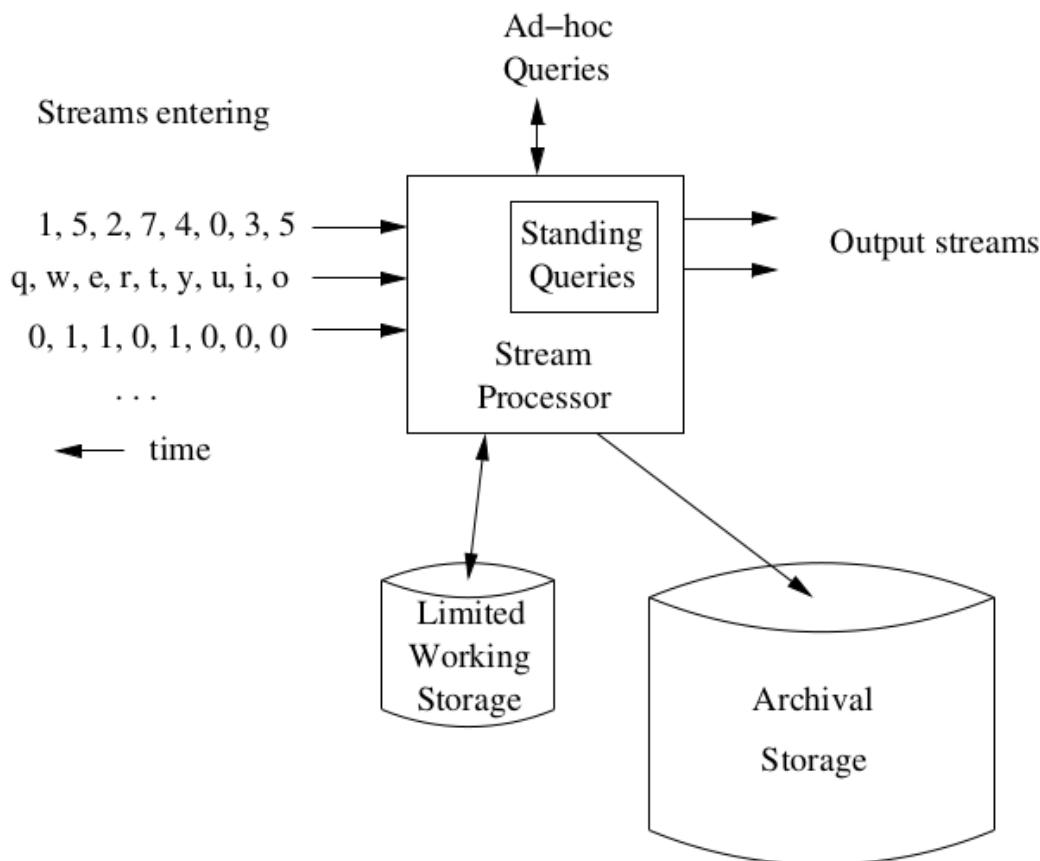
Un altro approccio per "riassumere" un flusso è quello di guardare solo ad una "finestra" di dati di lunghezza fissata, consistente negli ultimi n elementi di un flusso. Si interroga poi tale finestra come se fosse una relazione all'interno di un database. Se vi sono molti flussi e/o n risulta essere troppo grande, può non essere possibile memorizzare l'intera finestra per ogni flusso. Risulta quindi necessario schematizzare anche la finestra.

La gestione dei flussi risulta di fondamentale importanza quando l'*input rate*, ossia la velocità con cui l'input viene trasmesso, viene controllato esternamente.

Stream Model

Si studia il modello volto al processo di flussi di dati.

Facendo un'analogia con un sistema per la gestione di database, si può vedere un processore di flusso come una sorta di sistema di gestione di dati.



Un numero arbitrario di flussi può entrare nel sistema.

Gli elementi di un flusso possono essere visti come delle tuple.

Ogni flusso fornisce elementi al sistema seguendo la propria *schedule* personale, ossia, non devono necessariamente avere la stessa velocità di trasmissione o gli stessi tipi di dati degli altri flussi. Inoltre, l'intervallo di tempo che passa nella trasmissione di elementi contenuti all'interno di un flusso non deve necessariamente essere uniforme. La proprietà principale che distingue il processo di flussi di dati, rispetto al processo di dati contenuti all'interno di un

database, è data dal fatto che la velocità di arrivo degli elementi del flusso non è sotto il controllo del sistema che li elabora.

I flussi possono venir memorizzati all'interno di un archivio di grande capacità, ma si assume che l'accesso a quest'unità sia abbastanza lento da non poter permettere di rispondere alle queries, sui dati in essa memorizzati, efficientemente. Tale memoria può essere esaminata solo in circostanze particolari, utilizzando processi di recupero di informazioni i quali richiedono un tempo eccessivo. Il sistema dispone anche di una memoria di lavoro avente dimensione limitata, nella quale parti del flusso possono venir conservati e utilizzati per rispondere alle queries di interesse. Ovviamente, tale memoria dispone di una capacità sufficientemente limitata affinché non risulti essere possibile memorizzare tutti i dati provenienti da tutti i flussi presi in analisi.

Il sistema volto al processo di flussi non è in grado quindi di memorizzare l'intero stream in maniera tale che gli elementi che lo compongono risultino essere rapidamente accessibili. Possono quindi essere memorizzati e aggiornati solo brevi sketch di un flusso.

I tipici problemi riguardanti i flussi di dati sono:

- Campionamento di dati da un flusso, e costruzione di un campione casuale.
- Queries su una *sliding window*, come il problema del Pattern Matching, o l'identificazione del numero di oggetti di un certo tipo x negli ultimi k elementi del flusso.
- Filtrare un flusso di dati, ossia selezionare un elemento con una certa proprietà x dal flusso.
- Contare il numero di elementi distinti negli ultimi k elementi del flusso.
- Stimare la media o la deviazione standard degli ultimi k elementi.
- Individuare elementi frequenti.

Pattern Matching

Si studia ora il problema del Pattern Matching.

Sia x_i un elemento appartenente ad un certo alfabeto Σ . Il flusso è una stringa di lunghezza m su Σ , formalmente, sia $x = x_1x_2 \dots x_m$ tale flusso, allora $x \in \Sigma^m$.

Si ricorda che m è troppo grande affinché x possa venir interamente rappresentato in memoria.

Si supponga che venga fornito un pattern $y = y_1y_2 \dots y_n \in \Sigma^n$. La lunghezza del pattern n è tale che $n \ll m$, ma anch'essa può risultare abbastanza grande da non poter memorizzare totalmente il pattern all'interno di una memoria ad accesso veloce.

Si vuole sapere quante volte y appare in x come sua sottostringa, avendo quindi forma

$$y = x_i x_{i+1} \dots x_{i+n-1}.$$

Per risolvere il problema del pattern matching in maniera efficiente su un Data Stream, risulta necessario utilizzare un data sketch adeguato.

A tal fine, si definisce una misura di similarità: dati due elementi $\hat{x}, \hat{y} \in \Sigma^*$, si vuole sapere se

$\hat{x} = \hat{y}$. Tale misura prende il nome di *identità*. Si vuole quindi uno sketch $f(\cdot)$ tale per cui, si ha con alta probabilità che $f(\hat{x}) = f(\hat{y})$ se e solo se $\hat{x} = \hat{y}$. Ci si riferisce a tale proprietà, nella trattazione di questo problema, con il termine *string identity*.

Osservazione

Ricordando la definizione di data sketch, la proprietà 4 che esso deve avere per il problema PM è la proprio la *String Identity*.

Rabin Hash function

Si studia quindi la *Rabin hash function*, che risulta essere un data sketch valido per il problema pattern matching.

Senza perdita di generalità, sia x una stringa di lunghezza n sull'alfabeto $\Sigma = [0, \sigma - 1]$. Le stringhe possono essere quindi viste come interi di n cifre in base $\sigma > 0$. Si osserva esplicitamente che:

- Questo scenario può essere utilizzato per rappresentare sottoinsiemi di $[1, n]$ ponendo $\Sigma = \{0, 1\}$.
- Per ogni funzione f , se $\text{bitsize}(f(x)) < \text{bitsize}(x)$, dove $\text{bitsize}(\cdot)$ è una funzione che restituisce il numero di bit che un oggetto occupa in memoria, allora devono necessariamente verificarsi collisioni, ossia deve esistere necessariamente una coppia $x \neq y$ tale per cui $f(x) = f(y)$.

Una prima idea per risolvere il problema dell'identità (dati due elementi, verificare se questi sono uguali), può essere quella di utilizzare la funzione

$$\bar{h}(x) = ((a \cdot x + b) \bmod M) \bmod d$$

vista in precedenza, dove M è un numero primo e d è la dimensione dell'hash table, con $d \ll n$, considerando la stringa x come un numero con n cifre in base $|\Sigma|$.

Sfortunatamente, questo non risulta essere un buon approccio: essendo che, per definizione di tale funzione, risulti necessario che $M > x$ (dove si evidenzia che x deve essere preso come valore e non come la sua lunghezza) per ogni input x della funzione, bisognerebbe eseguire le operazioni di aritmetica modulare su interi con n cifre per aggiornare lo sketch, e ciò richiederebbe uno spazio pari a $\Omega(n)$. Ciò non rispetterebbe quindi il punto 3 nella definizione di data sketch.

Si definisce quindi il Rabin's hashing, uno schema di hashing di stringhe che risolve il problema appena descritto. Si fa presente che tale schema non risulti essere universale, ma nonostante ciò, garantisce una bassa probabilità di collisione.

Definizione (Funzione hash di Rabin):

Si fissi $q > \sigma$ numero primo, e si scelga uniformly at random $z \in \mathbb{Z}_q = \{0, 1, \dots, q - 1\}$.

Sia $x = \langle x[1], x[2], \dots, x[n] \rangle \in \Sigma^n$ una stringa di lunghezza n .

La funzione hash di Rabin $k_{q,z}(x)$ è definita come:

$$k_{q,z}(x) = \left(\sum_{i=0}^{n-1} x[n-i] \cdot z^i \right) \mod q$$

In forma espansa

$$k_{q,z}(x) = (x[n] + x[n-1] \cdot z + x[n-2] \cdot z^2 + \dots + x[1] \cdot z^{n-1}) \mod q$$

In altre parole, $k_{q,z}(x)$ è un polinomio modulo q valutato in z , e ha come coefficienti i caratteri di x .

Si osserva che il valore $k_{q,z}(x)$ è un numero modulo q , dunque lo spazio occupato da esso è logaritmico in q : $O(\log q)$.

Sia $|x|$ la lunghezza della stringa x . Si definisce lo sketch di Rabin $f(x)$ della stringa x come la coppia

$$f(x) = (k_{q,z}(x), z^{|x|} \mod q)$$

Si osserva che $f(x)$ necessita di $\Theta(\log q)$ bits per essere rappresentata, sempre perché $k_{q,z}(x)$ e $z^{|x|} \mod q$ sono numeri che vanno da 0 a $q - 1$.

Si vuole ora mostrare che questo sketch può essere calcolato ed aggiornato in maniera efficiente.

Si supponga di voler concatenare un carattere c alla stringa x , ottenendo quindi la stringa $x \cdot c$, dove tale notazione rappresenta x concatenato con c . Il valore di $x \cdot c$ può essere valutato come segue

Lemma

$$k_{q,z}(x \cdot c) = (k_{q,z}(x) \cdot z + c) \mod q$$

Avendo fatto riferimento al metodo di Horner per la valutazione di un polinomio:

$$k_{q,z}(x) = x[n] + z(x[n-1] + z(x[n-2] + z(x[n-3] + z(\dots + z(x[2] + z(x[1])))))) \mod q$$

La lunghezza di $x \cdot c$ è pari a $|x| + 1$ e $z^{|x|+1} \mod q = (z^{|x|} \mod q) \cdot z \mod q$.

Questo lemma fornisce anche un algoritmo efficiente per calcolare $k_{q,z}(x)$ in maniera efficiente: si inizia da $k_{q,z}(\varepsilon)$, dove ε è la stringa vuota, e si concatenano i caratteri di x uno alla volta.

Algoritmo A per $k_{q,z}(x)$ **input** : q primo, z u.a.r. in $[q]$, $x = \langle x[1], x[2], \dots, x[n] \rangle \in \Sigma^n$

1. $k_{q,z}(\varepsilon) = 0$
 2. **for** $j = 1$ to n **do**
 1. $(k_{q,z}(\langle x[1], x[2], \dots, x[j-1] \rangle) \cdot z + x[j]) \mod q$
-

Si osserva che $\text{space}(A(q, z, x)) = \mathcal{O}(\log q)$.

Usando un'idea simile, si possono concatenare gli sketches di due stringhe in tempo costante, come segue

Lemma

$$k_{q,z}(x \cdot y) = \left(k_{q,z}(x) \cdot z^{|y|} + k_{q,z}(y) \right) \mod q$$

Anche a partire da tale lemma, si definisce il seguente algoritmo

Algoritmo A_1 per $k_{q,z}(x \cdot y)$ **input**: q primo, z u.a.r. in $[q]$, $x = \langle x[1], x[2], \dots, x[n] \rangle, y = \langle y[1], y[2], \dots, y[m] \rangle \in \Sigma^n$

1. $k_{q,z}(x \cdot y) = (k_{q,z}(x) \cdot z^{|y|} + k_{q,z}(y)) \mod q$
-

Si osserva che $\text{space}(A_1(q, z, x, y)) = \mathcal{O}(\log q + \log |y| \cdot \log q)$.

Dove $z^{|y|}$ può venir calcolato usando un algoritmo per l'esponenziazione.

La lunghezza della stringa $x \cdot y$ è pari a $|x| + |y|$ e il valore $z^{|x|+|y|} \mod q$ può essere calcolato in maniera efficiente come

$$z^{|x|+|y|} \mod q = \left((z^{|x|} \mod q) \cdot (z^{|y|} \mod q) \right) \mod q$$

Si vuole mostrare ora che questo sia un buono sketch per il problema dell'identità, ossia che, se $x \neq y$, allora $k_{q,z}(x) \neq k_{q,z}(y)$ con alta probabilità. Ciò è implicato dal seguente lemma

Lemma

Sia $x \neq y$, con $|x| = |y| = n$. Allora

$$\Pr [k_{q,z}(x) = k_{q,z}(y)] \leq \frac{n}{q}$$

Dimostrazione

Si osserva che

$$\Pr [k_{q,z}(x) = k_{q,z}(y)] = \Pr [k_{q,z}(x) - k_{q,z}(y) \equiv_q 0]$$

Ora, la quantità $k_{q,z}(x) - k_{q,z}(y)$ è, a sua volta, un polinomio.

Sia $x - y$ la stringa definita come

$$x - y = \langle x(n) - y(n), x(n-1) - y(n-1), \dots, x(1) - y(1) \rangle$$

dove ogni operazione $x(i) - y(i)$ è effettuata in $\mod q$.

Allora, si ha che

$$k_{q,z}(x) - k_{q,z}(y) \mod q = k_{q,z}(x - y)$$

Da ciò segue che

$$\Pr [k_{q,z}(x) - k_{q,z}(y) \equiv_q 0] = \Pr [k_{q,z}(x - y) \equiv_q 0]$$

Essendo $x \neq y$, allora $k_{q,z}(x, y)$ è un polinomio di grado al più n in \mathbb{Z}_q , e non è il polinomio zero.

Si ricorda che ogni polinomio univariato diverso da zero di grado n su un campo ha al più n radici. Essendo q primo, \mathbb{Z}_q è un campo e quindi vi sono al più n valori di z tali per cui $k_{q,z}(x - y) \equiv_q 0$. Essendo che z viene scelto uniformemente da $[q]$, la probabilità di prendere una radice è al più n/q .

Corollario (Probabilità di errore)

Si scelga un primo $n^{c+1} \leq q \leq 2 \cdot n^{c+1}$ per una costante arbitrariamente grande c .

Allora, per ogni $\text{bitsize}(k_{q,z}(x)) \in \mathcal{O}(\log n)$ bits e, per ogni $x \neq y$

$$\Pr [k_{q,z}(x) = k_{q,z}(y)] \leq n^{-c}$$

Ossia, x ed y collidono con probabilità decrescente come l'inverso di un polinomio.

Si osserva che, essendo $\text{space}(k_{q,z}(x)) = \mathcal{O}(\log q) = \mathcal{O}(\log n)$, essendo $q = \Theta(n^c)$, lo sketch comprime una stringa di size n in una di size $\log n$.

Algoritmo di Karp-Rabin

Per risolvere il problema del pattern matching, si fa quindi riferimento alla funzione hash di Rabin.

Si osserva che tale tecnica porta ad una soluzione diretta che impiega spazio $\mathcal{O}(n)$.

Si supponga che sia stato già processato il flusso x_1, \dots, x_i con $i \geq n$, e che si siano già calcolati gli sketches $k_{q,z}(x_{i-n+1}x_{i-n+2} \dots x_i)$ e $k_{q,z}(y)$.

Confrontando questi due valori, in tempo costante, si può venire a sapere se il pattern compare o meno negli ultimi n caratteri del flusso di dati. Il passo cruciale è quello di aggiornare lo sketch dello stream non appena un nuovo elemento x_{i+1} arriva in input al processore del flusso.

Ciò può essere fatto come segue:

$$k_{q,z}(x_{i-n+2}x_{i-n+3} \dots x_{i+1}) = (k_{q,z}(x_{i-n+1}x_{i-n+2} \dots x_i) - x_{i-n+1} \cdot z^{n-1}) \cdot z + x_{i+1} \mod q$$

Il valore $z^{n-1} \mod q$ può essere pre calcolato, quindi l'operazione appena descritta richiede tempo costante, ossia, il numero di operazioni per ciascun oggetto entrante è $\Theta(1)$.

Si osserva che, dovendo accedere al carattere x_{i-n+1} per eseguire tale operazione, ad ogni istante l'algoritmo deve mantenere gli ultimi n caratteri visti nel flusso, utilizzando quindi spazio $\mathcal{O}(n)$.

Si fa presente che, se non vi sono collisioni tra il pattern e le $m - n + 1 \leq m$ sottostringhe del flusso di lunghezza n , allora l'algoritmo restituisce il risultato corretto, ossia il numero di occorrenze del pattern nello stream. La probabilità che il pattern collida con una qualsiasi di queste sottostringhe è al più n/q . Per l'union bound, la probabilità che il pattern collida con almeno una sottostringa è $mn/q \leq m^2/q$. Si vuole che ciò accada con bassa probabilità, ossia che decresce come l'inverso di un polinomio. Ciò può essere ottenuto scegliendo un primo q nel range $[m^{c+2}, 2 \cdot m^{c+2}]$, per ogni costante c . Tale numero primo, e quindi l'output della funzione hash di Rabin, può essere memorizzato in $\mathcal{O}(\log m)$ bits, ossia $\mathcal{O}(1)$ parole. Da ciò si ottiene quindi il seguente teorema

Teorema

L'algoritmo di Karp-Rabin risolve il problema del pattern matching nello streaming model utilizzando $\mathcal{O}(n)$ parole di memoria e con un delay di $\mathcal{O}(1)$. La soluzione corretta viene restituita con probabilità $1 - m^{-c}$, per ogni costante $c \geq 1$ scelta al momento dell'inizializzazione.

Campionamento da un Data Stream

Non potendo mantenere in memoria l'intero flusso, un approccio per lavorare con i dati ricevuti è quello di memorizzarne un campione.

Si studia quindi il problema di estrarre campioni utili all'interno di uno stream di dati.

In particolar modo, si prendono in considerazione i due seguenti problemi:

1. Campionare una proporzione fissata di elementi nel flusso.

2. Mantenere un campione casuale di grandezza fissata su uno stream potenzialmente infinito.

Campionare una proporzione fissata di elementi nel flusso

In questo problema, si vuole selezionare un sottoinsieme di elementi in un flusso, in maniera tale che si possano effettuare su di essi delle queries di interesse, e che le relative risposte risultino essere statisticamente rappresentative dell'intero flusso.

Esempio applicativo

Lo scenario applicativo per lo studio del problema è il seguente:

Un motore di ricerca riceve un flusso di queries, e vuole studiare il comportamento dell'utente tipico. Si assume che il flusso in input U consiste di tuple della forma $(userID, query, time)$ eseguite dagli utenti. Informalmente, si vuole rispondere a queries come "Quanto spesso un utente esegue la stessa query nello stesso giorno?" o "Quale frazione delle tipiche query di un utente sono state ripetute nell'ultimo mese?". Si assume inoltre che si desideri memorizzare solo 1/10 degli elementi dello stream.

Il problema può essere formalizzato come segue:

Input: Stream U di tuple $(userID, query, time)$.

Task algoritmico: Trovare un campione $S \subseteq U$ tale per cui, per il tipico utente u , e per ogni query q , approssimi bene, in valore atteso, la frazione delle q -occorrenze in U effettuate da u .

Definizione (q -occorrenza)

Dato un flusso U di tuple aventi struttura $(userID, query, time)$, si definisce q -occorrenza una tupla in U avente la componente $query = q$.

L'approccio banale per affrontare il problema è quello di generare un numero casuale, ad esempio un intero tra 0 e 9, come risposta ad ogni query di ricerca per ogni utente. Si memorizza la tupla se e solo se il numero casuale è 0. Così facendo, ogni utente ha in media, 1/10 delle sue query memorizzate nel campione.

N-Algo:

1. Inizializza $S = \emptyset$
2. **for each** tupla $= (userID, query, time) \in U$

1. Scegli indipendentemente uniformemente al caso $z \in [10]$
 2. **if** $z = 0$ **then** Memorizza $(userID, query, time)$ in S
 3. **else** scarta $(userID, query, time)$
3. **for each** query q :
1. restituisci i valori medi delle frazioni delle q -occorrenze calcolate solamente su S .
-

Si osserva esplicitamente che l'associazione tra tuple e numeri casuali può essere effettuato mediante funzioni hash, in particolare modo da una funzione $h : U \rightarrow [10]$, che mappa casualmente ogni elemento di U in uno tra 10 buckets. Gli elementi mappati nei buckets 1, 2, ..., 9 vengono scartati, mentre gli elementi mappati in 0 costituiranno il campione memorizzato per un utente.

Questo schema, però, fornisce una risposta sbagliata alla richiesta del numero medio di queries duplicate effettuate da un utente.

Si prende a titolo di esempio il seguente scenario:

Si considerino $s + d$ queries distinte q_1, q_2, \dots, q_s e q'_1, q'_2, \dots, q'_d .

Ogni utente u , nell'intervallo di tempo di un mese, effettua $s + 2d$ queries di ricerca nella seguente maniera:

- s queries vengono eseguite una singola volta.
- d queries vengono eseguite, ciascuna di esse, due volte.
- Nessuna query di ricerca viene effettuata più di due volte.

Nel flusso U sono presenti quindi $s + 2d$ q -occorrenze per ogni utente.

Se si dispone di un campione S pari ad 1/10 delle queries totali, ottenuto mediante l'algoritmo appena descritto, dove quindi $\mathbf{E}[|S|] = (1/10) \cdot |U|$, ci si aspetta di trovare in esso $s/10$ delle queries eseguite una singola volta da un dato utente. Delle d queries effettuate due volte, ci si aspetta che solo $d/100$ appariranno due volte nel campione. Tale rapporto è dato da d moltiplicato la probabilità che entrambe le occorrenze della query appaiano nel campione:

Sia X la variabile aleatoria che conta il numero di queries, ripetute due volte all'interno del flusso di dati in ingresso, che appaiono entrambe le volte nel campione.

Per ogni query q che appare due volte nello stream si definisce la variabile aleatoria X_q come segue:

$$X_q = \begin{cases} 1 & \text{se } q \text{ appare due volte nel campione } S \\ 0 & \text{altrimenti} \end{cases}$$

Si ha quindi che

$$X = \sum_q X_q$$

Ricordando che, la probabilità che una tupla (u, q, t) venga selezionata nel campione è proprio

$$\Pr[h((u, q, t)) = 0] = \frac{1}{10}$$

e che tale selezione risulta essere indipendente per ogni tupla, si ha che

$$\mathbf{E}[X_q] = 1 \cdot \frac{1}{10} \cdot \frac{1}{10} = \frac{1}{100}$$

Sfruttando la linearità del valore atteso si ottiene quindi

$$\mathbf{E}[X] = \mathbf{E}\left[\sum_q X_q\right] = \sum_q \mathbf{E}[X_q] = d \cdot \frac{1}{100} = \frac{d}{100}$$

Delle queries che appaiono due volte nell'intero stream, ci si aspetta che $19d/100$ appaiano almeno una volta. Infatti, siano (u, q, t_1) e (u, q, t_2) due query duplicate. Siano \mathcal{E}_1 ed \mathcal{E}_2 gli eventi

$$\mathcal{E}_1 = (u, q, t_1) \text{ appare nel campione } S$$

$$\mathcal{E}_2 = (u, q, t_2) \text{ appare nel campione } S$$

Tali eventi sono, ovviamente, indipendenti.

La probabilità che q appaia almeno una volta nel campione è:

$$\Pr[(\mathcal{E}_1 \cap \mathcal{E}_2) \cup (\mathcal{E}_1 \cap \bar{\mathcal{E}}_2) \cup (\bar{\mathcal{E}}_1 \cap \mathcal{E}_2)]$$

Essendo $\Pr[\mathcal{E}_i] = 1/10$ e, di conseguenza, $\Pr[\bar{\mathcal{E}}_i] = 1 - \Pr[\mathcal{E}_i] = 1 - 1/10 = 9/10$ per $i = 1, 2$, si ha che

$$\begin{aligned} \Pr[(\mathcal{E}_1 \cap \mathcal{E}_2) \cup (\mathcal{E}_1 \cap \bar{\mathcal{E}}_2) \cup (\bar{\mathcal{E}}_1 \cap \mathcal{E}_2)] &= \frac{1}{10} \cdot \frac{1}{10} + \frac{1}{10} \cdot \frac{9}{10} + \frac{9}{10} \cdot \frac{1}{10} \\ &= \frac{1}{100} + \frac{2 \cdot 9}{100} = \frac{19}{100} \end{aligned}$$

Sia Y la variabile aleatoria che conta il numero di queries, ripetute due volte all'interno del flusso di dati in ingresso, che appaiono *almeno* una volta nel campione.

Per ogni query q che appare due volte nello stream si definisce la variabile aleatoria Y_q come segue:

$$Y_q = \begin{cases} 1 & \text{se } q \text{ appare almeno una volta nel campione } S \\ 0 & \text{altrimenti} \end{cases}$$

Si ha che $\mathbf{E}[Y_q] = \Pr[Y_q = 1] = \Pr[(\mathcal{E}_1 \cap \mathcal{E}_2) \cup (\mathcal{E}_1 \cap \bar{\mathcal{E}}_2) \cup (\bar{\mathcal{E}}_1 \cap \mathcal{E}_2)]$, e che $Y = \sum_q Y_q$.

Quindi

$$\mathbf{E}[Y] = \sum_q \mathbf{E}[Y_q] = d \cdot \frac{19}{100}$$

Per un'argomentazione analoga, ci si aspetta che $18d/100$ appariranno esattamente una volta. Questo perché $18/100$ è la probabilità che una delle due occorrenze, di una query

effettuata due volte, appaia nell'1/10 del campione selezionato, mentre l'altra si trova nei 9/10 non selezionati:

$$\Pr \left[(\mathcal{E}_1 \cap \bar{\mathcal{E}}_2) \cup (\bar{\mathcal{E}}_1 \cap \mathcal{E}_2) \right] = \left(\left(\frac{1}{10} \cdot \frac{9}{10} \right) + \left(\frac{9}{10} \cdot \frac{1}{10} \right) \right) = \frac{18}{100}$$

Sia Z la variabile aleatoria che conta il numero di queries, ripetute due volte all'interno del flusso di dati in ingresso, che appaiono esattamente una volta nel campione, e sia Z_q la variabile aleatoria definita per ogni q che appare due volte nello stream come segue:

$$Z_q = \begin{cases} 1 & \text{se } q \text{ appare esattamente una volta nel campione } S \\ 0 & \text{altrimenti} \end{cases}$$

Si ha che $\mathbf{E}[Z_q] = \Pr \left[(\mathcal{E}_1 \cap \bar{\mathcal{E}}_2) \cup (\bar{\mathcal{E}}_1 \cap \mathcal{E}_2) \right]$ e che $Z = \sum_q Z_q$, e quindi

$$\mathbf{E}[Z] = \sum_q \mathbf{E}[Z_q] = d \cdot \frac{18}{100}$$

La risposta corretta alla query riguardante la frazione di ricerche ripetute è $d/(s + d)$. Nonostante ciò, la risposta ottenuta dal campione prodotto dall'algoritmo banale è $d/(10s + 19d)$. Per derivare tale formula, si osserva che, delle queries contenute nel campione, ci si aspetta che solo $d/100$ appariranno ripetute due volte in esso, mentre $s/10 + 18d/100$ appariranno una singola volta. Si ha quindi che il numero atteso di elementi *distinti* in S è pari a

$$\frac{s}{10} + \frac{d}{100} + \frac{18d}{100}$$

e

$$\mathbf{E}[|S|] = \frac{s}{10} + \frac{2d}{100} + \frac{18d}{100}$$

Quindi, la frazione di ricerche ripetute che compaiono due volte nel campione è data da dal numero di ricerche ripetute che ci si aspetta di avere nel campione, diviso il numero degli elementi distinti nel campione:

$$\frac{\frac{d}{100}}{\frac{s}{10} + \frac{d}{100} + \frac{18d}{100}} = \frac{d}{10s + 19d}$$

e per nessun valore positivo di s e d , si può avere che $d/(s + d) = d/(10s + 19d)$.

Campionamento degli utenti

Nello scenario appena descritto, il campione viene selezionato su *tutte* le queries effettuate da ogni singolo utente.

La query studiata per il problema perso in analisi, come molte altre queries riguardanti le statistiche del tipico utente, non può essere risposta prendendo un campione delle ricerche effettuate da ciascun singolo utente. Risulta necessario infatti prendere 1/10 degli utenti, e

prendere *tutte* le loro ricerche come campione. Se risultasse possibile memorizzare una lista di tutti gli utenti e la loro eventuale partecipazione alla definizione di tale campione, si potrebbe operare come segue:

- Ogni volta che una query di ricerca arriva nel flusso, si verifica se l'utente sia stato selezionato o meno per la costruzione del campione.
- Se è stato scelto, si aggiunge la query al campione, altrimenti no.
- Se l'utente che richiede la query non è stato ancora registrato, ossia, la query che arriva nel flusso è la prima per tale utente, si genera un intero casuale tra 0 e 9. Se il numero è 0, si memorizza tale utente tra quelli selezionati per la definizione del campione, altrimenti, si memorizza tra quelli non selezionati, dovendo necessariamente tener traccia anche di quest'ultimi al fine di scartare le eventuali queries future.

Questo metodo funziona sino a quando risulta possibile memorizzare l'intera lista di tutti gli utenti e la loro appartenenza alla definizione del campione nella memoria principale, non essendo possibile accedere al disco per ogni ricerca in ingresso.

Usando una funzione hash, si può evitare di mantenere la lista degli utenti. Si esegue l'hash per ogni *userID*, mappandoli in uno tra dieci buckets etichettati con gli interi che vanno da 0 a 9. Se l'utente è stato mappato nel bucket 0, si accettano le sue query per la definizione del campione, altrimenti no.

Si osserva esplicitamente che, in realtà, non si memorizzano gli utenti nei buckets.

Infatti, si usa una funzione hash come generatore di numeri casuali, con l'importante proprietà che, se applicata allo stesso utente diverse volte, si ottiene sempre lo stesso numero "casuale".

Così facendo, senza memorizzare l'appartenenza o meno al campione per ciascun utente, si può ricostruire tale informazione in ogni istante in cui arriva una query mediante la funzione hash.

Tale procedura può essere descritta mediante il seguente pseudocodice, dove l'insieme di tutti gli utenti è conosciuto in anticipo.:

User-Sample Algo:

1. Scegli 1/10 degli utenti e inserisci *tutte* le loro q -occorrenze nel campione S nella seguente maniera:
 1. Usa una funzione hash che mappa gli utenti in 10 buckets i.u.a.r.
 2. Selezione tutte le q -occorrenze degli utenti mappati nel bucket 0 e memorizzali in S .
 2. Esegui dei calcoli sulla media di S .
-

Si fa un'analisi dell'algoritmo descritto. Sia Q l'insieme delle query, e sia I l'insieme degli utenti. Si supponga che I sia conosciuto in anticipo.

Sia $x(v, q)$ il numero di q -occorrenze dell'utente $v \in I$ nello stream in input.

Sia $X_S(V, q)$ il numero di q -occorrenze dell'utente $V \in I$ selezionato casualmente per la definizione del campione S . Si osserva esplicitamente che V , e quindi $X_S(V, q)$, è una variabile aleatoria. Questo perché gli utenti da cui costruire il campione sono scelti u.a.r. indipendentemente.

Si ha quindi che la media del numero delle q -occorrenze per ogni utente $v \in I$, sia questa $\text{AVG}_{v \in I}(x(v, q))$, è uguale a $\mathbf{E}_{V \in S}[X_S(V, q)]$, per ogni q fissato:

$$\text{AVG}_{v \in I}(x(v, q)) = \mathbf{E}_{V \in S}[X_S(V, q)]$$

e quindi, si ha che:

$$\text{AVG}_{V \in S}(X_S(V, q)) \longrightarrow \text{AVG}_{v \in I}(x(v, q)) \text{ per } |\{\text{utenti in } S\}| \longrightarrow |I|$$

Ossia, l'output dell'algoritmo $\text{AVG}_{V \in S}(X_S(V, q))$ tende al valore target $\text{AVG}_{v \in I}(x(v, q))$ al tendere del numero di utenti scelti per il campione verso il numero complessivo di utenti, ossia, aumentando gli utenti selezionati, si ottiene un risultato sempre più vicino all'obiettivo.

Si evidenzia che la dimensione attesa di S è $1/10$ dello stream in input U di *tutti* gli oggetti. Infatti, sia C_j la variabile aleatoria definita come segue:

$$C_j = \begin{cases} 1 & \text{se } h(j) = 0 \\ 0 & \text{altrimenti} \end{cases}$$

dove $h : I \longrightarrow [10]$ è la funzione hash che mappa ogni utente in uno tra 10 buckets, e se $h(j) = 0$ allora l'utente j viene selezionato per la creazione di S .

Sia x_j il numero di elementi dell'utente j nell'intero stream U . Allora

$$\sum_{j \in I} x_j = |U|$$

e

$$\mathbf{E}[|S|] = \sum_{j \in I} x_j \cdot \mathbf{Pr}[C_j = 1] = \frac{1}{10} \cdot \sum_{j \in I} x_j = \frac{|U|}{10}$$

Più generalmente, si può ottenere un campione consistente in una frazione razionale a/b degli utenti, mappando gli *userID* in b buckets, che vanno da 0 a $b - 1$. Si aggiunge la query al campione se il valore hash dell'utente che la effettua è minore di a .

Il problema generale del campionamento

L'esempio applicativo studiato può essere generalizzato mediante il seguente problema:

Si ha un flusso di dati, consistente in tuple con n componenti (x_1, x_2, \dots, x_n) .

Un sottoinsieme delle componenti delle tuple $(x_{i,1}, x_{i,2}, \dots, x_{i,j})$ definisce le *chiavi*, sulle quali si basa la selezione del campione. La scelta delle chiavi dipende dall'applicazione, ad

esempio, per lo scenario applicativo visto, si hanno tuple a 3 componenti della forma $(userID, query, time)$ e la chiave risulta essere la componente $userID$.

Per ottenere un campione di size a/b , si mappa il valore della chiave, per ciascuna tupla, in b buckets, e si inserisce una tupla nel campione se e solo se la valutazione della funzione hash nel valore della chiave è al più a . Se la chiave è composta da più di una componente, la funzione hash deve combinare i valori di tali componenti per creare un singolo valore hash. Il risultato di tale operazione sarà un campione consistente da tutte le tuple con determinati valori chiave. I valori chiave selezionati saranno approssimativamente una frazione pari ad a/b di tutti i valori chiave che appaiono nello stream.

Ad esempio, si supponga di trovarsi nello scenario descritto in precedenza, dove i valori chiave sono dati dagli $userID$ di ciascun utente. Si supponga di avere un'hash table con b buckets. Si vogliono selezionare le tuple se l'hash del valore chiave è al più a . Per generare un campione del 30% degli utenti, si mappano le chiavi in $b = 10$ buckets, e si inseriscono le tuple nel campione se queste vengono mappate in uno dei primi 3 buckets.

Variare dinamicamente la dimensione del campione

La dimensione del campione può aumentare man mano che il flusso entrante in input nel sistema aumenta. Può quindi non risultare possibile memorizzare la frazione di flusso desiderata al fine di creare un campione da analizzare. Se si ha un limite sul numero di tuple provenienti dal flusso che si possono memorizzare, allora la frazione dei valori chiave che lo costituiscono deve variare, diminuendo con l'avanzare del tempo. Per far sì che, in ogni istante, il campione consista in *tutte* le tuple appartenenti ad un sottoinsieme dei valori chiave, si sceglie una funzione hash h che mappa i valori chiave in un grande numero B di buckets. Si mantiene una *variabile di soglia* $t \leq B - 1$, che, inizialmente, può essere anche il più grande numero di bucket $B - 1$. In ogni istante, il campione consiste in quelle tuple tali per cui la chiave K soddisfa $h(K) \leq t$. Le nuove tuple che entrano nel sistema dal flusso vengono aggiunte al campione se e solo se soddisfano tale condizione.

Se il numero di tuple memorizzate nel campione eccede lo spazio ad esse dedicato, si riduce t a t' , dove $t' << t$, e tutte le tuple mappate nei buckets aventi numero $b > t'$ memorizzate in precedenza nel campione vengono rimosse da esso. Ulteriore efficienza può essere ottenuta mantenendo un indice sui valori hash, in maniera tale che si possano individuare velocemente tutte le tuple le quali la chiave viene mappata in un particolare valore.

Tale procedura può essere descritta schematicamente come segue:

1. Mappa ogni chiave in un grande numero B di buckets.
2. Imposta un valore di soglia iniziale $t \leq B - 1$. Tutte le chiavi mappate nei buckets aventi etichetta $b \leq t$ verranno inclusi in S .
3. Se $|S|$ diventa troppo grande, riduci t a t' , con $t' << t$ e rimuovi tutte le tuple aventi chiave mappata nei buckets aventi etichetta $b > t'$.

4. Ripeti il passo 3 se necessario.

Mantenere un campione casuale di grandezza fissata su uno stream infinito.

Si studia il problema di mantenere un campione casuale S di dimensione fissata s nello scenario in cui la lunghezza del flusso che il sistema deve elaborare cresca nel tempo. In particolar modo, non si conosce la dimensione di tale flusso in anticipo, e questa può potenzialmente crescere all'infinito.

Si vuole costruire il campione S , in maniera tale che, supponendo che all'istante n il sistema abbia visto i primi n elementi del flusso, ogni elemento visto si trovi nel campione S corrente con probabilità s/n , dove con campione corrente si fa riferimento a quello elaborato sino all'ultimo istante in cui ci si trova.

Una prima idea banale può essere quella di memorizzare tutte le n tuple viste sino all'ultimo istante, e da esse estrarne s i.u.a.r. Ovviamente, questo approccio risulta fallimentare, necessitando di una quantità di memoria pari a $\Theta(n)$ per conservare gli n elementi.

La soluzione per il problema è data dal seguente algoritmo

Reservoir Sampling RS

Input: s

1. Memorizza i primi s elementi del flusso in S
2. Si supponga di aver visto i primi n elementi, e che all'istante $n + 1$, l' $(n + 1)$ -esimo elemento arrivi al sistema ($n + 1 > s$):
 1. Con probabilità $s/(n + 1)$, inserisci l' $(n + 1)$ -esimo elemento in S , altrimenti scartalo.
 2. Se è stato scelto l' $(n + 1)$ -esimo elemento per l'inserimento in S , scegli u.a.r. uno dei precedenti s elementi nel campione S ed eliminalo.

Si vuole mostrare che, dopo aver visto $(n + 1)$ elementi, S mantenga la proprietà che ogni elemento visto sino a quell'istante si trovi in esso con probabilità $s/(n + 1)$.

Lo si fa per induzione.

Il caso base è data dalla parte deterministica iniziale dell'algoritmo, ossia, per $n = s$ elementi visti, si ha che il campione S mantiene la proprietà desiderata. Infatti, ciascuno degli $n = s$ elementi si trova in S con probabilità $s/s = 1$.

Induttivamente, si supponga di aver visto n elementi, e che la probabilità che ciascuno di essi sia in S è s/n .

Quando l' $(n + 1)$ -esimo elemento arriva, lo si inserisce in S con probabilità $s/(n + 1)$. Si vuole mostrare che, la probabilità che ogni altro elemento *del flusso*, all'istante di arrivo dell'elemento $n + 1$, si trovi in S è proprio $s/(n + 1)$.

Per fare ciò definiscono i tre seguenti eventi:

$$\mathcal{E}_1 = \text{L'elemento } n + 1 \text{ viene scartato}$$

$$\mathcal{E}_{2,1} = \text{L'elemento } n + 1 \text{ non viene scartato}$$

$$\mathcal{E}_{2,2} = \text{L'elemento } x \text{ nel campione non viene eliminato}$$

Si studiano le probabilità di questi eventi.

L'elemento $n + 1$ non viene scartato, ossia viene inserito in S con probabilità

$$\Pr[\mathcal{E}_{2,1}] = \frac{s}{n + 1}$$

Si osserva che $\Pr[\mathcal{E}_1] = \Pr[\bar{\mathcal{E}}_{2,1}]$, quindi

$$\Pr[\mathcal{E}_1] = 1 - \frac{s}{n + 1}$$

Essendo che, nel caso in cui l' $(n + 1)$ -esimo venisse scelto, ogni altro elemento nel campione in quell'istante ha probabilità $1/s$ di venir scartato, si ha che

$$\Pr[\mathcal{E}_{2,2}] = 1 - \frac{1}{s} = \frac{s - 1}{s}$$

Per ogni vecchio elemento $x \in S$, la probabilità che l'algoritmo lo mantenga in S è data da

$$\Pr[\mathcal{E}_1] + \Pr[\mathcal{E}_{2,1}]\Pr[\mathcal{E}_{2,2}] = \left(1 - \frac{s}{n + 1}\right) + \left(\frac{s}{n + 1}\right)\left(\frac{s - 1}{s}\right) = \frac{n}{n + 1}$$

Infatti x rimane in S o se il nuovo elemento viene scartato, o se il nuovo elemento viene aggiunto e tale x non viene rimosso.

Quindi, all'istante $n + 1$, ciascuno dei vecchi elementi $x \in S$ rimane in esso con probabilità $n/(n + 1)$.

Per un generico elemento nel flusso processato in precedenza, si ha che la probabilità che questo si trovi in S all'istante $n + 1$ è data dalla probabilità che questo si trovi in S all'istante n , moltiplicata per la probabilità che tale elemento non venga rimosso per fare posto al nuovo, ossia

$$\frac{s}{n} \cdot \frac{n}{n + 1} = \frac{s}{n + 1}$$

E ciò dimostra, per induzione sulla lunghezza del flusso n , che tutti gli elementi hanno la stessa probabilità s/n di far parte del campione.

Queries su una finestra scorrevole

Si studia ora un problema di conteggio su streams. Si supponga di avere una finestra di lunghezza N su un flusso binario, dove una finestra di tale dimensione consiste negli N

elementi ricevuti più recentemente dal flusso.

In un dato istante, gli elementi nel flusso alla sinistra della finestra sono stati già visitati da essa, mentre gli elementi alla sua destra devono ancora essere visitati.

Si vuole sapere, ad ogni istante, quanti 1 sono presenti negli ultimi k bits per ogni $k \leq N$.

Anche in questo scenario, per via dei limiti sulla memoria, non è possibile memorizzare l'intera finestra, ossia, gli ultimi N bits che il sistema riceve in un certo istante.

Il problema può essere formalizzato come segue:

Input: Un flusso binario on-line infinito I , N lunghezza della finestra.

Obiettivo: Calcolare la funzione $\#1(I, N, k)$, che restituisce il numero di cifre 1 negli ultimi k bits del flusso, per ogni $k \leq N$.

Si mostra, inizialmente, che ottenere un risultato esatto per questo problema necessita la memorizzazione dell'intera finestra, e che quindi, per il vincolo appena enunciato, bisogna far riferimento ad un algoritmo che restituisca una soluzione approssimata.

Si vuole quindi contare *esattamente* il numero di 1 negli ultimi k bits dello stream, per ogni $k \leq N$. Per fare ciò, bisogna necessariamente memorizzare tutti gli N bits della finestra, essendo che una qualsiasi altra rappresentazione che utilizza meno di N bits non può funzionare. Per dimostrare ciò, si supponga di avere uno schema di rappresentazione, sia questo $R(N, k)$, che utilizza meno di N bits per rappresentare gli N bits nella finestra, ossia $|R(N, k)| \leq N - 1$. Essendovi in tutto 2^N possibili stringhe binarie di lunghezza N , ma meno di 2^N possibili rappresentazioni per esse, devono necessariamente esistere due stringhe di bits w ed x , tali per cui $w \neq x$, le quali hanno la stessa rappresentazione R . Essendo tali stringhe distinte, devono differire in almeno un bit. Si supponga quindi che i primi $k - 1$ bits di w ed x siano uguali, mentre differiscano solamente per il k -esimo. Se $k < N$, i restanti bits che costituiscono le due stringhe non sono di interesse alla risoluzione della query.

Ora, essendo le rappresentazioni per tali stringhe identiche, un qualsiasi algoritmo che calcola la funzione $\#1(I, N, k)$ facendo riferimento allo schema $R(N, k)$ non risulta essere in grado di distinguere le due stringhe x e w , producendo lo stesso output per entrambe e, quindi, fallendo, essendo che una di esse ha necessariamente un bit in più asserito ad 1 rispetto all'altra.

Tale dimostrazione quindi mostra che:

- Non si può fare riferimento ad un *data sketch* per ottenere una soluzione esatta al problema.
- Per contare *esattamente* il numero di 1 negli ultimi k bits dello stream, per una qualsiasi distribuzione del flusso in input arbitraria, lo spazio di memoria impiegato da un algoritmo che risolve tale problema è almeno N . Bisogna quindi usare almeno N bits per rispondere ad una qualsiasi query in merito agli ultimi k bits per ogni $k \leq N$.

Si vuole quindi trovare una soluzione approssimata al problema.

Algoritmo DGIM

Si mostra un algoritmo che utilizza $\mathcal{O}(\log^2 N)$ bits per rappresentare una finestra di N bits, che permette di stimare il numero di 1 nella finestra con un errore, nel caso peggiore, non più grande del 50%. Tale fattore di errore può essere ridotto ad ogni valore costante $c > 0$, che utilizza sempre $\mathcal{O}(\log^2 N)$ bits, ma con un fattore costante che cresce man mano che c diminuisce.

Per iniziare, si associa un *timestamp* ad ogni bit, ossia la posizione nella quale questo arriva nel flusso. Il primo bit ha timestamp 1, il secondo ha timestamp 2 e così via. Dovendo distinguere le posizioni degli elementi solamente all'interno della finestra di lunghezza N , si possono rappresentare i timestamps in modulo N , in maniera tale che questi possano venir rappresentati mediante $\log_2 N$ bits.

Così facendo, se si memorizzasse il numero *totale* dei bits visti nel flusso (ossia, sino al timestamp più recente) in modulo N , risulterebbe possibile determinare, da un timestamp in modulo N , la posizione del bit con tale timestamp all'interno della finestra.

Si divide la finestra in buckets. Un bucket, per quest'algoritmo, è un record $\langle A, B \rangle$ consistente in:

- A : Il time-stamp della fine di tale bucket (ossia il bit più a destra contenuto in esso), che richiede quindi $\mathcal{O}(\log N)$ bits.
- B : Il numero di 1 all'interno del bucket. Tale numero deve essere necessariamente una potenza di 2, e ci si riferisce a tale quantità come la *size* del bucket. Per rappresentare tale valore sono necessari $\mathcal{O}(\log \log N)$ bits. Si vede ora il motivo:

Per rappresentare un bucket, sono necessari $\log_2 N$ bits per rappresentare il timestamp (in modulo N) del suo estremo destro. Per rappresentare il numero di 1 al suo interno sono necessari $\log_2 \log_2 N$ bits. Questo è vero perché tale numero, sia questo i , è una potenza di 2, sia questa $2^j = i$. Si può quindi rappresentare i codificando j in binario.

$$2^j = i \iff \log_2 2^j = \log_2 i \iff j = \log_2 i \iff \log_2 j = \log_2 \log_2 i$$

Essendo j al più $\log_2 N$, per fare ciò sono necessari $\log_2 \log_2 N$ bits. Quindi, $\mathcal{O}(\log N + \log \log N) = \mathcal{O}(\log N)$ bits risultano sufficienti per rappresentare un bucket.

Ci sono sei regole che devono essere eseguite quando si rappresenta uno stream mediante buckets:

1. L'estremo destro di un bucket è sempre una posizione contenente 1.
2. Ogni posizione all'interno della finestra contenente un 1 deve trovarsi in un bucket.
3. I buckets non si sovrappongono nei timestamps, ossia, una posizione all'interno della finestra non può trovarsi in più di un bucket.
4. Possono esistere solo uno o due buckets con la stessa dimensione, ossia con lo stesso numero di 1 al loro interno.
5. Tutte le size di ogni bucket devono essere una potenza di due.
6. I buckets non possono decrescere nella dimensione muovendosi verso sinistra (indietro nel tempo), ossia sono ordinati secondo la loro dimensione, decrescente rispetto al tempo (più il bucket contiene elementi recenti, più è piccolo in dimensione).

In particolar modo, le proprietà (3), (4) e (6) sono mantenute dinamicamente.

Inoltre, si fa presente che un bucket viene eliminato quando il suo estremo destro si trova in una posizione maggiore di N unità di tempo nel passato.

Si vogliono studiare le seguenti questioni:

- Il motivo per cui il numero di buckets per rappresentare una finestra deve essere piccolo.
- Come mantenere le condizioni espresse man mano che nuovi bits entrano nel flusso.
- Come stimare il numero di 1 negli ultimi k bits per ogni k , con un errore non più grande del 50%.

Numero di buckets per la rappresentazione di una finestra

Si è osservato che ogni bucket può essere rappresentato mediante $\mathcal{O}(\log N)$ bits. Se la finestra ha lunghezza N , allora non possono sicuramente esserci più di N bits aventi valore 1. Si supponga ora che il più grande bucket abbia size 2^j . Allora j non può eccedere $\log_2 N$, perché se così fosse, ci sarebbero più 1 in questo bucket rispetto al numero degli 1 nell'intera finestra. Quindi, ci sono al più due buckets di tutte le dimensioni che vanno da $2^{\log_2 N}$ sino ad 1, e nessun bucket di size maggiore.

Si conclude da ciò che, per una finestra di dimensione N , esistono $\mathcal{O}(\log N)$ buckets.

Essendo che ogni bucket è rappresentato mediante $\mathcal{O}(\log N)$ bits, lo spazio totale necessario per memorizzare tutti i buckets rappresentanti una finestra di dimensione N è $\mathcal{O}(\log^2 N)$.

Come mantenere le condizioni espresse all'ingresso di nuovi bits nel flusso

Si studia ora come mantenere le condizioni espresse man mano che nuovi bits entrano nel flusso, ossia, come avviene l'aggiornamento dei buckets.

Si supponga di avere una finestra di lunghezza N , rappresentata da i buckets in maniera tale che vengano rispettate le condizioni definite in precedenza. Quando un nuovo bit entra nella finestra, può risultar necessario modificare i buckets, in maniera tale che questi continuino a rappresentare la finestra e a soddisfare le condizioni dell'algoritmo DGIM.

Prima di tutto, quando un nuovo bit entra nella finestra:

- Si controlla il bucket più a sinistra, ossia quello presente da più tempo. Se il suo timestamp, ossia il timestamp del bit più a destra contenuto in esso, è uguale al timestamp del bit corrente meno N , allora tale bucket non possiede più al suo interno nessuno degli 1 nella finestra. Quindi, lo si elimina dalla lista dei buckets.
- Bisogna ora considerare se il nuovo bit in ingresso è 0 o 1. Se è 0, allora non è necessaria nessuna modifica all'interno dei buckets. Se è 1, bisogna effettuare dei cambiamenti al loro interno:

- Si crea un nuovo bucket avente dimensione 1, contenente solo tale bit. Tale bucket avrà quindi come timestamp il tempo attuale.

Se, in precedenza, era presente un solo bucket di dimensione 1, allora non risulta necessario effettuare ulteriori cambiamenti. Altrimenti, se si hanno tre buckets di dimensione 1, non viene rispettato uno dei vincoli. Si risolve questo problema combinando i due buckets più a sinistra, ossia i più vecchi, aventi dimensione 1.

Per combinare due buckets adiacenti della stessa dimensione, li si sostituiscono con un unico bucket avente dimensione doppia. Il timestamp del nuovo bucket così creato è il timestamp del bucket più a destra tra i due coinvolti in tale operazione.

In generale, combinare due buckets di una certa dimensione 2^i può portare alla creazione di un terzo bucket avente dimensione 2^{i+1} , supponendo che gli altri due esistessero già in precedenza. Quindi, per ovviare questo problema, si continuano a combinare i buckets sino a quando non risulti essere più necessario. Essendovi al più $\log_2 N$ dimensioni differenti, ed essendo che la combinazione di due buckets adiacenti della stessa dimensione richiede tempo costante (AVL update?), si ha come risultato che ogni nuovo bit può venir processato in tempo $\mathcal{O}(\log N)$.

Come stimare il numero di 1 negli ultimi k bits per ogni k , con un errore non più grande del 50%.

Si supponga che si voglia sapere quanti 1 si trovino negli ultimi k bits della finestra, per $1 \leq k \leq N$. Per fare ciò, si prende il bucket b con il timestamp minore, ossia meno recente, che contiene almeno alcuni dei k bits più recenti. Si stima il numero degli 1 contenuti negli ultimi k bits come la somma delle sizes di tutti i buckets alla destra di b (ossia più recenti), più metà della dimensione di b stesso. Questo perché non si sa quanti 1 del bucket più vecchio b si trovino nel range della sotto-finestra di dimensione k .

Si supponga che la stima, riguardante la risposta alla query, coinvolga un bucket b di dimensione 2^j , che si trova parzialmente nel range della query k .

L'output dell'algoritmo Y è quindi il seguente

$$Y = \sum_{i=0}^{j-1} a_i 2^i + \frac{1}{2} \cdot 2^j = \sum_{i=0}^{j-1} a_i 2^i + 2^{j-1}$$

dove $a_i \in \{1, 2\}$ è il numero di buckets aventi size 2^i .

Si vuole studiare quanto la risposta corretta c sia lontana dalla stima effettuata.

Si possono verificare due possibili casi:

1. La stima restituita dall'algoritmo è minore della risposta corretta, ossia $Y < c$. Nel caso peggiore, tutti gli 1 di b si trovano nel range della query, quindi la stima non prende in considerazione metà della dimensione di b , ossia 2^{j-1} bits aventi valore 1. In questo caso, c è almeno 2^j , considerando tutti gli 1 presenti in b . In realtà, è almeno $2^{j+1} - 1$, essendovi almeno un bucket per ogni size $2^{j-1}, 2^{j-2}, \dots, 1$, ossia

$$c \geq \sum_{i=0}^{j-1} 2^i = 2^j - 1$$

comandovi i 2^j bits presenti nella finestra più vecchia, si ha che c è almeno $2^{j+1} - 1$:

$$c \geq 2^j - 1 + 2^j = 2^{j+1} - 1$$

Quindi la stima effettuata è almeno il 50% di c .

2. La stima restituita dall'algoritmo è maggiore della risposta corretta, ossia $Y > c$. Nel caso peggiore, solo il bit più a destra del bucket b si trova nel range della query. Nel caso in cui si avesse un singolo bucket per ciascuna delle size minori della size di b , si ha che

$$c = \sum_{i=0}^{j-1} 2^i + 1 = 2^j$$

e quindi

$$c \geq 2^j$$

e la stima fatta è

$$2^{j-1} + \sum_{i=0}^{j-1} 2^i = 2^j + 2^{j-1} - 1$$

che risulta essere non più grande del 50% di c .

Ridurre l'errore di approssimazione

Si tratta questo argomento informalmente.

L'idea chiave è quella di mantenere, invece di 1 o 2 buckets per ciascuna size, $b - 1$ o b buckets con $b > 0$, eccetto per il bucket avente size maggiore, per il quale si può avere un qualsiasi numero tra 1 ed r di questi. Così facendo, l'errore è al più $\mathcal{O}(1/r)$.

Scegliendo r in maniera appropriata, si può avere un tradeoff tra complessità spaziale ed errore di approssimazione.

Contare interi

Si vuole vedere come estendere l'algoritmo per sommare gli ultimi k elementi di uno stream, nel caso in cui tale flusso non fosse composto unicamente da cifre binarie ma da interi.

L'idea è quella che, partendo dall'ipotesi di avere all'interno del flusso interi positivi nel range che va da 1 a 2^m , per i quali sono quindi necessari m bit per rappresentare ciascuno di questi valori univocamente, si considerino gli m bits di ciascun intero come un flusso separato. Così facendo, si può usare l'algoritmo DGIM per stimare il numero di 1 che

costituiscono un intero. Si supponga che il conto dell' i -esimo bit sia c^i . Si ha allora che la somma degli interi è

$$\sum_{i=0}^{m-1} c_i 2^i$$

Filtrare elementi nei flussi

Un'altro processo comune che viene applicato su flussi è la selezione, o filtraggio, di elementi. Si vogliono selezionare quelle tuple all'interno di un flusso che rispettano un criterio. Le tuple accettate vengono passate ad un altro processo sotto forma di flusso, mentre le altre vengono scartate. Se il criterio di selezione è una proprietà della tupla che può essere calcolata, ad esempio, se la prima componente della tupla è minore di 10, allora la selezione è facile da effettuare.

Questo problema diventa più difficile quando il criterio di selezione comprende la verifica dell'appartenenza di una componente ad un dato insieme.

Il problema può essere formulato quindi come segue:

Modello di flusso: Ogni elemento del data stream X preso in analisi è una tupla $x = \langle key_1, key_2, \dots, key_k \rangle$.

Input: Una lista di "buoni" valori chiave S , ossia un sottoinsieme di valori che rispettano un determinato criterio per key_1 . S è quindi un sottoinsieme dell'insieme contenente tutti i valori assumibili dalle componenti.

Output: Determinare quali, e quindi filtrare, le tuple $x \in X$ tali per cui $key_1(x) \in S$.

Questo problema diventa particolarmente difficile da trattare quando l'insieme di valori considerati "buoni" per le chiavi è troppo grande per essere immagazzinato nella memoria principale.

Infatti, l'approccio banale di utilizzare una funzione hash h che mappa tutti gli elementi di S in un'hash table T , e verificare per ogni tupla x del flusso se $h(key_1(x))$ venga mappata correttamente in T , non risulta possibile vista la quantità di memoria necessaria per memorizzare tutti gli elementi in S in un'hash table.

Si studia inizialmente un esempio applicativo al fine di illustrare il problema.

Si vogliono filtrare le email, contenute all'interno di un flusso, provenienti da specifici indirizzi. In particolare modo, il flusso è costituito da coppie $\langle key_1, key_2 \rangle$, dove key_1 è l'indirizzo del mittente e key_2 il contenuto della mail.

Si supponga di avere un insieme S di un miliardo di indirizzi email ammissibili, ossia per i quali si è sicuri che, se una mail arriva da uno di essi, allora questa sicuramente non è spam.

Si vogliono quindi selezionare tali mail all'interno del flusso, scartando quelle che non rispettano il vincolo definito.

Un approccio per la risoluzione del problema è dato dall'algoritmo first cut, caratterizzato da una prima fase di pre-processing, e da una seconda fase effettuata online, ossia, durante l'arrivo del flusso.

Sia U l'insieme di *tutti* i valori assumibili da key_1 , ossia di tutti gli indirizzi email, e sia $S \subseteq U$ l'insieme dei valori per key_1 reputati validi. L'algoritmo opera come segue:

Fase 1: Pre-processing

- Si crea un array $B[0 \dots n - 1]$ di n bits, impostati inizialmente tutti a 0.
- Si sceglie una funzione hash $h : U \rightarrow [n]$.
- Per ogni componente $s \in S$ si calcola $h(s)$.
- Si imposta $B[h(s)] = 1$ per ogni componente. I restanti bit in B rimangono quindi impostati a 0.

Fase 2: Online

Sia $x = \langle x_1, \dots, x_k \rangle$ il nuovo elemento entrante nel flusso.

- Calcola $h(x_1)$
- Se $B[h(x_1)] = 1$ allora accetta l'elemento x dello stream, scartalo altrimenti.

Si fa presente che, se un elemento $x = \langle x_1, \dots, x_k \rangle$ in ingresso è tale per cui $B[h(x_1)] = 0$, allora si è sicuri che tale indirizzo non si trovi in S , essendo che h mappa ogni elemento di S all'interno dell'array B . Lo si può quindi scartare con sicurezza. In particolar modo, per ogni elemento nello stream tale per cui la chiave ha un valore contenuto in S , si ha che questa verrà accettata. L'algoritmo non restituisce quindi falsi negativi.

Il problema è che h potrebbe mappare un valore $v \in U \setminus S$, ossia non ammissibile secondo il vincolo definito, in un'entrata di B impostata ad 1, per via di una collisione $h(v) = h(s)$ con un elemento $s \in S$. Può quindi generare falsi positivi, ossia, mail che risultano essere spam verranno immesse nel flusso filtrato da prendere in analisi.

Sia $|S| = m$ il numero di indirizzi email ammessi, ad esempio 1 miliardo, e sia $|B| = n$ la dimensione dell'array, ad esempio 1 GigaByte, ossia 8 miliardi di bits.

Si hanno quindi, per l'applicazione studiata, i seguenti risultati:

Teorema (no false negatives)

Se l'indirizzo email è in S , ossia, per una mail $x = \langle x_1, x_2 \rangle$, $x_1 \in S$, allora sicuramente x_1 viene mappato da h in un bucket avente bit impostato ad 1, e quindi supera sicuramente la fase di filtraggio.

Teorema (falsi positivi)

Approssimativamente $m/n = 1/8$ dei bits sono impostati ad 1, quindi circa $1/8$ (ossia la probabilità che avvenga una collisione), degli indirizzi *non* in S superano la fase di filtraggio.

In realtà, meno di 1/8 indirizzi non in S superano la fase di filtraggio, essendo che più di un'indirizzo può essere mappato nello stesso bit.

Si fa quindi un analisi più accurata per il numero di falsi positivi, facendo riferimento al modello *balls into bins*. In questo problema si hanno m sfere che vengono lanciate in n cestini, dove il cesto per ogni sfera viene scelto indipendentemente e con distribuzione di probabilità uniforme.

Sia quindi $|S| = m$ e $|B| = n$. La probabilità per un elemento $u \in U$ di venir mappato in un bucket pieno, ossia tale per cui $B[h(u)] = 1$, è esattamente la frazione dei buckets già pieni in B . Per stimare tale valore si studia quindi un processo *balls into bins*.

Per fare ciò, ci si chiede, nel caso in cui si lanciassero m sfere in n cestini equiprobabili, quale sia probabilità che un cesto ottenga almeno una sfera.

Per il nostro problema, il modello può essere formulato come segue:

- Ai cestini corrispondono i buckets che vanno da 0 ad $n - 1$.
- Alle sfere corrispondono i valori hash casuali $h(key_1(x))$ degli m elementi in S . Si assume quindi che h mappi u.a.r. tali elementi.
- La probabilità che quindi una sfera entri in un cesto è $\Pr[ball - into - bin] = 1/n$ per la definizione di h .

Sia E l'evento "un cesto ottiene almeno una sfera".

Allora la probabilità che un cesto ottenga almeno una sfera è

$$\Pr[E] = 1 - \left(1 - \frac{1}{n}\right)^m \approx 1 - e^{-m/n}$$

usando l'approssimazione $(1 - x) \approx e^{-x}$.

Si osserva infine che per $m \ll n$, $\Pr[E] \approx m/n$.

Sia ora R la variabile aleatoria che conta il numero di cestini pieni. Per ogni cesto i , si definisce la variabile aleatoria R_i come segue

$$R_i = \begin{cases} 1 & \text{se } i \text{ contiene almeno una pallina} \\ 0 & \text{altrimenti} \end{cases}$$

Si ha quindi che

$$\mathbf{E}[R] = \sum_{i=1}^n \mathbf{E}[R_i] = n \cdot \Pr[E] = n \cdot (1 - e^{-m/n})$$

La probabilità che si verifichi un errore è uguale alla probabilità che una sfera finisca in un cesto pieno, quindi, sia ERR questo evento

$$\Pr[ERR] = \frac{\mathbf{E}[R]}{n} = \frac{n \cdot (1 - e^{-m/n})}{n} = 1 - e^{-m/n} = \Pr[E]$$

Quindi la frazione di 1 (ossia di cestini pieni) nell'array B è uguale alla probabilità di falsi positivi $1 - e^{-m/n}$.

Per risolvere il problema, si fa quindi riferimento ad una tecnica chiamata *Bloom filtering*

Bloom filter

Un bloom filter è una struttura dati che consiste in:

1. Un array B di n bits, impostati inizialmente tutti a 0.
2. Una collezione di funzioni hash h_1, h_2, \dots, h_k . Ciascuna funzione hash mappa i valori chiave in n buckets, corrispondenti agli n bit dell'array.
3. Un insieme S di m valori chiave.

Lo scopo di un Bloom filter è quello di filtrare e quindi accettare, tra tutti gli elementi dello stream, quelli aventi chiavi in S , scartando la maggior parte di quelli tali per cui la chiave non risulti essere in S .

Il suo utilizzo avviene come segue:

- Si inizializza l'array di bit impostando tutte le sue entrate a 0.
- Si prende ogni valore chiave $s \in S$, e se ne effettua l'hash utilizzando ciascuna delle k funzioni.
- Si pone $B[h_i(s)] = 1$ per ogni $i = 1, \dots, k$.

Per testare un elemento con chiave x al suo arrivo nel flusso, si verifica se $B[h_i(x)] = 1$ per ogni $i = 1, \dots, k$. Nel caso in cui questo risulti essere vero, ossia che x viene mappato in un bucket impostato ad 1 per ogni funzione hash h_i , allora si dichiara che x appartiene ad S , altrimenti lo si scarta.

Si fa ora l'analisi del bloom filtering.

Se un valore chiave è in S , allora l'elemento con tale valore nel flusso passerà sicuramente attraverso il Bloom filter. Se un valore chiave non si trova in S , l'elemento avente tale valore potrebbe comunque venir selezionato. Si vuole anche qui studiare la probabilità di falsi positivi, e lo si fa in funzione di n , m e k .

Anche per questa analisi, si fa riferimento al modello *balls into bins*, e si supponga di avere $k \cdot m$ sfere ed n cestini. Ogni sfera ha la stessa probabilità di finire in uno degli n cestini, ossia $1/n$. Analogamente a come fatto in precedenza, ci si chiede quale sia la frazione degli elementi in B impostati ad 1.

La probabilità che una sfera *non* venga lanciata in un cesto fissato è $(n - 1)/n$.

La probabilità che nessuna delle $k \cdot m$ sfere entri in un dato cesto è quindi

$$\left(\frac{n-1}{n}\right)^{k \cdot m} = \left(1 - \frac{1}{n}\right)^{n(\frac{k \cdot m}{n})}$$

Usando l'approssimazione $(1 - x) \approx e^{-x}$, si ha quindi che la probabilità che nessuna delle $k \cdot m$ sfere entri in un dato cesto è circa

$$e^{-k \cdot m / n}$$

Che è proprio la probabilità che un bit in B rimanga a 0.

Allora la frazione di entrate aventi valore 1 in B è:

$$\left(1 - e^{-k \cdot m/n}\right) \approx \frac{k \cdot m}{n}$$

Ma si hanno esattamente k funzioni hash mutualmente indipendenti, e si accetta l'elemento x se tutte le k funzioni hash mappano x in un bucket avente valore 1.

Allora, la probabilità di errore, ossia che un falso positivo venga accettato è proprio

$$\left(1 - e^{-k \cdot m/n}\right)^k$$

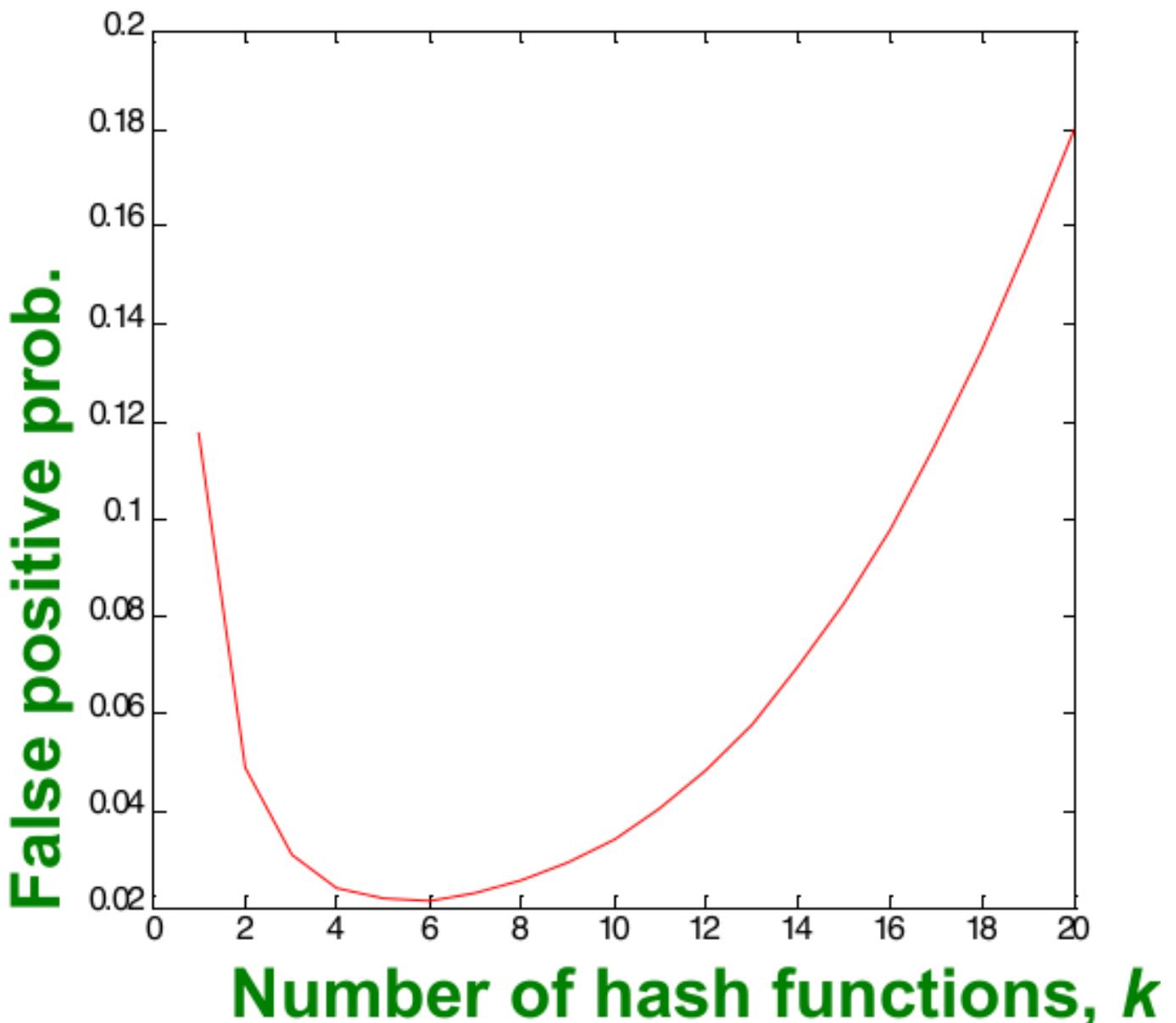
Si vuole studiare ora cosa succede all'aumentare del numero di funzioni hash per questa procedura, ossia al crescere di k .

Sia $m = 1$ miliardo e $n = 8$ miliardi.

Si ha che l'errore è:

- Per $k = 1$, $(1 - e^{-1/8}) = 0.1175$.
- Per $k = 2$, $(1 - e^{-1/4})^2 = 0.0493$.

E in generale, l'andamento della funzione che descrive la probabilità di falsi positivi all'aumentare delle funzioni hash può essere raffigurato come segue:



Il valore ottimale per k è $(n/m) \ln(2)$, nell'esempio visto $k = 8 \ln(2) = 5.54 \approx 6$
 E l'errore per $k = 6$ è $(1 - e^{-1/6})^2 = 0.0235$.

Ci si chiede ora se l'approccio migliore sia quello di mantenere un unico array B di grande dimensione, o se suddividerlo in k array B_1, \dots, B_k . Per la prima opzione si ha $(1 - e^{-km/n})^k$, mentre per la seconda $(1 - e^{-m/(n/k)})^k$. I due valori sono vicini, ma mantenere un unico bucket risulta più semplice.

Contare elementi distinti

Si studia ora il problema di contare elementi distinti all'interno di un flusso. Come nei problemi precedenti, di campionamento e di filtering, si fa riferimento a tecniche di hashing e algoritmi probabilistici per ottenere una soluzione approssimata in maniera tale da rispettare i vincoli sulla quantità di memoria principale, cercando quindi un buon trade-off tra spazio utilizzato e qualità della soluzione.

Si supponga che gli elementi del flusso vengano scelti da un insieme universale. Si vuole sapere quanti elementi differenti sono comparsi nello stream, contando dall'inizio del flusso. Formalmente, il problema può essere schematizzato come segue:

- Il flusso di dati consiste in una sequenza di elementi scelti da un'insieme universo U avente dimensione $|U| = N$.
- Si vuole mantenere un conto d del numero di elementi distinti visti all'interno del flusso, partendo dall'inizio del flusso sino all'ultimo istante, ossia sino all'ultimo elemento processato.

L'approccio ovvio alla risoluzione del problema è quella di mantenere in memoria principale un insieme $S \subseteq U$ contenente tutti gli elementi visti nel flusso fino all'ultimo istante della sua analisi.

Tale insieme può essere implementato in una struttura dati di ricerca efficiente, come un'hash table o un albero di ricerca, in maniera tale che risulti possibile aggiungere velocemente nuovi elementi, e controllare se l'elemento appena arrivato nello stream sia stato già visto o meno. Fino a quando il numero di elementi distinti non risulta essere troppo grande, ossia $|S| << |U|$, una struttura di questo tipo può richiedere una quantità di spazio sufficientemente adeguata da rispettare la capacità della memoria principale.

Come per gli altri problemi, se il numero di elementi distinti risulta essere troppo grande, o se vi è un numero troppo elevato di flussi da processare simultaneamente, allora non risulta possibile memorizzare i dati necessari nella memoria principale.

Tenendo a mente che l'obiettivo non è propriamente quello di memorizzare l'intero insieme S , ma di contare il numero d di elementi distinti visti nel flusso, e nonostante la memorizzazione di S possa garantire una risposta al problema, si fa riferimento ad un algoritmo probabilistico che garantisce una soluzione approssimata che, nonostante possa avere un piccolo errore, limita la probabilità che l'errore sia grande, e che permette di non memorizzare l'intero insieme S .

Algoritmo di Flajolet-Martin

Sia d il numero di elementi distinti. Si osserva inizialmente che, essendo $d \leq N$, è possibile memorizzare d utilizzando solamente $\mathcal{O}(\log N)$ bits.

E' possibile, mediante l'algoritmo di Flajolet-Martin, ridurre la complessità spaziale a $\mathcal{O}(\log \log N)$ bits. L'idea dell'algoritmo è quella di stimare $\log d$ invece che d . Si ci riferisce ad algoritmi che utilizzano questa strategia come *algoritmi di conteggio approssimato*.

L'algoritmo è preceduto da una fase di pre-processing, nella quale si sceglie una funzione hash h che mappa ciascuno degli N elementi di U in almeno $\log_2 N$ bits.

Senza perdita di generalità, si assume che $U = [N]$. Si definisce tale funzione h come segue

$$h : [N] \longrightarrow \{0, 1\}^s \text{ dove } s \geq \log_2 N \text{ bits}$$

Si assume che h sia veramente casuale, ossia, per ogni $i \in [N]$, si ha che $h(i)$ è scelto uniformly at random da $\{0, 1\}^s$.

Per ogni elemento $i \in U$, si definisce $r(i)$ come il numero di bit pari a 0, partendo da destra, che precedono il primo bit impostato ad 1 della stringa $h(i)$.

Ad esempio sia, per un dato elemento $a \in U$, $h(a) = 01100$, allora $r(a) = 2$.

Si osserva inoltre che, per la quantità $r(i)$, essendo $h(i)$ scelta u.a.r. da $\{0, 1\}^s$, si ha che

$$\Pr[r(i) \geq l] = \frac{2^{s-l}}{2^s} = \frac{1}{2^l}$$

Perché tra tutte le possibili stringhe binarie di lunghezza s , ve ne sono esattamente 2^{s-l} su un totale di 2^s tali per cui $r(i) \geq l$, ossia tali per cui i primi l bits di tale stringhe sono uguali a 0, tutte aventi la stessa possibilità di essere scelte. Analogamente, la questione può essere affrontata come segue: scegliere una stringa u.a.r. da $\{0, 1\}^s$ equivale a scegliere ciascuno degli s bits della stringa i.u.a.r. da $\{0, 1\}$. Quindi, la probabilità che una stringa binaria di lunghezza s pescata u.a.r. abbia almeno i primi l bits a 0 è uguale alla probabilità che i primi l bits selezionati u.a.r. siano uguali a 0, e siccome la selezione di ogni bit avviene indipendentemente ed uniformemente, la probabilità che ciò accada è pari ad $1/2^l$.

Si può ora descrivere l'algoritmo di conteggio probabilistico. Tale algoritmo mantiene un data sketch del flusso ad ogni passo, definito come segue

$$R = \max\{r(i), \text{ per ogni elemento } i \in U \text{ visti fino all'ultimo istante}\}$$

E viene descritto mediante il seguente pseudocodice:

Algoritmo di Flajolet-Martin FM

Input: Flusso di elementi $i \in U$

1. $R = 0$
 2. **for** i nel flusso **do**
 1. Calcola $r(i)$
 2. $R = \max(R, r(i))$
 3. **return** 2^R
-

Dove il valore in output 2^R è la stima del numero degli elementi distinti all'interno dello stream già visti dal sistema. L'algoritmo ha diverse proprietà interessanti:

- Le occorrenze ripetute dello stesso elemento non influiscono sul valore di R , essendo che lo stesso elemento i viene mappato sempre nello stesso valore $h(i)$, ed avrà quindi sempre lo stesso $r(i)$.
- Lo sketch R può essere facilmente combinato assieme ad altri: se si dispone di una collezione di sketches R_1, R_2, \dots, R_k provenienti da stream differenti, definiti tutti sullo

stesso alfabeto universale, e si desidera calcolare il numero d di elementi distinti del flusso definito dalla combinazione di questi k flussi, si può prendere, tra questi k sketches, quello avente valore massimo

$$\max\{R_1, R_2, \dots, R_k\}$$

Si vuole dimostrare ora la correttezza dell'algoritmo. Si ricorda che $\Pr[r(i) \geq l] = 1/2^l$.

Sia X_l la variabile aleatoria definita come segue:

$$X_l = \begin{cases} 1 & \text{se } \exists i \text{ nello stream tale per cui } r(i) \geq l \\ 0 & \text{altrimenti} \end{cases}$$

Siano d gli elementi distinti nel flusso. Allora si ha che

$$\begin{aligned} \Pr[X_l = 1] &= \Pr\left[\bigcup_i (r(i) \geq l)\right] \\ &= 1 - \Pr\left[\bigcap_i (r(i) < l)\right] \\ &= 1 - (1 - 2^{-l})^d \end{aligned}$$

Essendovi d elementi distinti nel flusso, ed essendo $(r(i) < l)$ un evento indipendente per ogni $i \in U$.

Allora si ha che

$$\Pr[X_l = 0] = (1 - 2^{-l})^d$$

Sia $m = 2^R$ l'output dell'algoritmo e siano d gli elementi distinti nel flusso. Si vuole calcolare la probabilità di errore dell'algoritmo.

Si calcola inizialmente la probabilità che $R > \lceil \log d \rceil + c$ per ogni $c > 0$

$$\begin{aligned} \Pr[m > 2^c \cdot d] &= \Pr[R > \lceil \log d \rceil + c] = \Pr[X_{\lceil \log d \rceil + c} = 1] \\ &= 1 - \left(1 - \frac{1}{2^{\lceil \log d \rceil + c}}\right)^d \leq \frac{d}{2^{\lceil \log d \rceil + c}} \leq 2^{-c} \end{aligned}$$

Dove, per effettuare la maggiorazione, si è fatto riferimento alla diseguaglianza $(1 - x)^d \geq 1 - xd$, vera per $|xd| < 1$.

Si calcola ora la probabilità che $R < \lceil \log d \rceil - c$ per ogni $c > 0$

$$\begin{aligned} \Pr[m < 2^c \cdot d] &= \Pr[R < \lceil \log d \rceil - c] = \Pr[X_{\lceil \log d \rceil - c} = 0] \\ &= \left(1 - \frac{1}{2^{\lceil \log d \rceil - c}}\right)^d \leq \exp\left(-\frac{d \cdot 2^c}{2^{\lceil \log d \rceil}}\right) \leq \exp(-2^{c-1}) \end{aligned}$$

Dove, per effettuare la maggiorazione, si è fatto riferimento alla disuguaglianza $1 - x \leq \exp(-x)$, vera per $|x| < 1$.

Queste disuguaglianze forniscono forti bounds esponenziali sulla probabilità che si ottenga una stima che si trova entro un errore additivo c dal risultato reale. In particolar modo, per $c = 2$ si ha che

$$\Pr[X_{\lceil \log d \rceil + c} = 1] = \Pr[R > \lceil \log d \rceil + 2] \leq 2^{-2} = \frac{1}{4}$$

$$\Pr[X_{\lceil \log d \rceil - c} = 0] = \Pr[R < \lceil \log d \rceil - 2] \leq \exp(-2) \leq \frac{1}{8}$$

E, per l'union bound, vale che

$$\Pr[|R - \lceil \log d \rceil| \leq 2] \geq 1 - \left(\frac{1}{4} + \frac{1}{8} \right) = \frac{5}{8} \approx \frac{2}{3}$$

(Rivedere quelle cazzo di disuguaglianze dagli appunti di pasquale cosa rappresentano)
In altre parole, si ha che

$$\frac{1}{4} \cdot 2^{\lceil \log d \rceil} \leq 2^R \leq 4 \cdot 2^{\lceil \log d \rceil}$$

Ossia una 8–approssimazione a d con probabilità circa $2/3$. Si può poi usare l'algoritmo median per aumentare la probabilità di successo.

Lo spazio richiesto per l'algoritmo di Flajolet-Martin è semplicemente lo spazio necessario per memorizzare R , ed essendo che $R \leq \lceil \log d \rceil + c$ con alta probabilità, la memoria per R sarà

$$\lceil \log R \rceil = \log \log d + \mathcal{O}(1) \leq \log \log n$$

e quindi **Space**[FM algorithm] = $\mathcal{O}(\log \log d)$.

Ci si chiede ora quale sia l'output atteso dell'algoritmo, ossia $\mathbf{E}[2^R]$. Flajolet e Martin hanno mostrato che $\mathbf{E}[2^R]$ tende rapidamente a d , scalato per una specifica costante:

$$\mathbf{E}[2^R] \approx 0.77551d$$

Questo rende l'algoritmo di conteggio probabilistico pratico: Si calcola semplicemente $\mathbf{E}[2^R]$, e lo si divide per 0.77551.

E' stata mostrata inoltre una rifinitura dell'algoritmo di Flajolet Martin che garantisce un algoritmo probabilistico $(1 - \varepsilon)$ –approssimante per d , che utilizza spazio par a $\mathcal{O}(\varepsilon^{-2} \cdot \log(1/\varepsilon)) + 2 \log \log n$.

Calcolo dei momenti

Si studia una generalizzazione del problema del conteggio di elementi distinti all'interno di uno stream. Questo problema riguarda la distribuzione delle *frequenze* di elementi distinti all'interno di un flusso.

Si supponga che un flusso I consista di elementi scelti da un'insieme universale $U = \{1, \dots, N\}$. Si assuma che l'insieme universale sia totalmente ordinato. Sia m_i la frequenza dell'elemento i , ossia il numero delle occorrenze di tale elemento nel flusso I , per ogni $i \in U$. Si definisce il k -momento del flusso come la somma su tutti gli i di $(m_i)^k$:

$$\sum_{i \in U} (m_i)^k$$

Si guardano ora dei casi particolari di k -momento:

- $k = 0$: Il 0-momento è dato dalla somma di 1 per ogni $m_i > 0$. Si fa presente per convenzione che, se $m_i = 0$ per qualche elemento nell'insieme universale, allora si considera $(m_i)^0 = 0^0 = 0$. Per le frequenze $m_i \geq 1$, si ha che la frequenza di ogni elemento contribuisce alla sommatoria per una quantità pari ad 1. Si osserva esplicitamente quindi che questo caso particolare del problema del calcolo dei momenti è proprio il problema del conteggio di elementi distinti, che si può affrontare con gli approcci spiegati in precedenza.
- $k = 1$: L' 1-momento è la somma di tutti gli m_i , ossia, si conta ogni elemento nel flusso ogni volta che questo appare. Risulta quindi essere proprio la lunghezza del flusso. Tale caso particolare è semplice da calcolare, basta contare la lunghezza del flusso visto sino all'ultimo elemento arrivato al sistema.
- $k = 2$: E' la somma dei quadrati degli m_i per ogni $i \in U$. E' spesso chiamato *surprise number*, essendo che misura quanto è irregolare la distribuzione degli elementi nel flusso. Maggiore è questo valore, più irregolare risulta essere la distribuzione degli elementi.

Come negli altri problemi visti, non ci sono difficoltà nel calcolo dei momenti di qualsiasi ordine nel caso in cui fosse possibile mantenere in memoria un contatore per ciascun elemento che appare nel flusso. Si prende in analisi il caso in cui non fosse possibile utilizzare tale quantità di memoria. Bisogna quindi stimare il k -momento mantenendo un limitato numero di valori in memoria, e calcolare una stima partendo da essi. Per il caso di elementi distinti visto in precedenza, ciascuno di questi valori era dato dal numero di zeri che precedono il primo 1 per la valutazione di una funzione hash in un elemento dell'universo.

Algoritmo di Alon-Matias-Szegedy

L'algoritmo AMS risolve il problema per tutti i momenti, dandone una stima *unbiased*. Il *bias* di un estimatore è dato dalla differenza tra il suo valore atteso, e il vero valore del parametro stimato. Un'estimatore si dice *unbiased* se il suo *bias* è uguale a 0 per tutti i valori del parametro preso in analisi.

Stima dei 2-Momenti

Si assuma che un flusso abbia una certa lunghezza n fissata. Si vedrà successivamente come affrontare il problema per flussi che crescono nel tempo.

Si supponga che non si abbia sufficiente spazio per contare tutti gli m_i per tutti gli elementi del flusso. Si può stimare il 2-momento del flusso usando una quantità di memoria limitata. Come per altri problemi studiati, maggiore è lo spazio usato, migliore sarà la stima di tale quantità.

Per questo algoritmo, si sceglie e si aggiorna un campione di variabili aleatorie indipendenti ed identicamente distribuite $\mathcal{X} = \{X_j : j = 1, \dots, k\}$ tali per cui, per ogni $X \in \mathcal{X}$ si mantiene in memoria:

1. Un particolare elemento dell'insieme universale, al quale ci si riferisce come $X.el$.
2. Un intero $X.val$, corrispondente al valore della variabile. Per determinare il valore di una variabile X , si sceglie una posizione nel flusso tra 1 ed n u.a.r., e si imposta $X.el$ come l'elemento presente in tale posizione, e si inizializza il valore $X.val$ ad 1. Man mano che il flusso viene letto, si aggiunge 1 ad $X.val$ ogni volta che si incontra un'altra occorrenza di $X.el$. In altre parole, $X.val$ corrisponde al conteggio delle future occorrenze di $i = X.el$.

Si osserva che il conteggio avviene in memoria principale, e quindi il numero k di variabili aleatorie deve essere limitato.

A partire da tale procedura, si può stimare il 2-momento da ogni variabile X come il valore

$$n(2X.val - 1)$$

Schematicamente, l'algoritmo si può rappresentare come segue, e viene applicato per un $X \in \mathcal{X}$:

- **Input:** Flusso $I = I[1 \dots n]$ di lunghezza n .
- Si sceglie un istante di tempo t (equivalentemente, una posizione) u.a.r. tale per cui $1 \leq t < n$.
- Si pone $X.el := i$, dove $i = I[t]$ è l' i -esimo elemento in I
- Si calcola il contatore $X.val = c$ come il numero di occorrenze di i nel sotto-stream $I[t, \dots, n]$.
- **Output:** La stima per il 2-momento $\sum_i m_i^2$ è

$$S = f(X) = n \cdot (2 \cdot c - 1)$$

Tale algoritmo viene calcolato per ogni $X \in \mathcal{X}$, e la stima finale viene data dalla media dei relativi output

$$S = \frac{1}{k} \sum_{j=1}^k f(X_j)$$

aumentando così la confidenza.

Per effettuare l'analisi dell'algoritmo, si mostra che il valore atteso di una qualsiasi variabile aleatoria costruita come descritto è il 2–momento del flusso dal quale viene costruita.

Sia $e(i)$ l'elemento dello stream che appare in posizione i all'interno di tale flusso, e sia $c(i)$ il numero di volte che l'elemento $e(i)$ appare nel flusso nelle posizioni che vanno da i ad n ($i, i+1, \dots, n$). Il valore atteso di $n \cdot (2X.val - 1)$ è la media su tutte le posizioni i tra 1 ed n di $n(2c(i) - 1)$, ossia

$$\mathbf{E}[n(2X.val - 1)] = \frac{1}{n} \sum_{i=1}^n n(2c(i) - 1) = \sum_{i=1}^n (2c(i) - 1)$$

Per dare un senso alla formula, bisogna cambiare l'ordine della sommatoria raggruppando tutte quelle posizioni aventi lo stesso elemento. Per ogni $a \in U$ fissato che appare m_a volte nel flusso, si ha che il termine per l'ultima posizione nella quale a appare deve essere $2 \times 1 - 1 = 1$. Il termine per la penultima posizione nella quale a appare è $2 \times 2 - 1 = 3$. Proseguendo in questa maniera, si ha che il termine per la prima posizione in cui a appare è $2m_a - 1$, perché in tale sotto-flusso, che va dalla prima posizione in cui a appare sino all'ultima, a appare esattamente m_a volte. Si ha quindi che la formula per il valore atteso $2X.val - 1$ può essere scritta come:

$$\mathbf{E}[n(2X.val - 1)] = \sum_a \sum_{i=1}^{m_a} (2i - 1)$$

ed essendo che $\sum_{i=1}^{m_a} (2i - 1) = (m_a)^2$, si ha che

$$\mathbf{E}[n(2X.val - 1)] = \sum_a (m_a)^2$$

che è proprio la definizione di 2–momento.

Momenti di alto ordine

La stima per il k –momento, per $k > 2$, si calcola eseguendo lo stesso algoritmo per il 2–momento, cambiando però la maniera in cui si deriva la stima a partire da una variabile. Per $k = 2$ si è utilizzata la formula $n(2v - 1)$ per rendere un valore v , dove tale valore è il conteggio del numero di occorrenze di un particolare elemento del flusso a , in una stima del 2–momento. Si è mostrato inoltre il motivo per cui questa formula funziona: la somma dei termini $2v - 1$, per $v = 1, \dots, m$, è uguale ad m^2 , dove m è il numero di volte in cui a appare nel flusso.

Si osserva che $2v - 1$ è la differenza tra v^2 e $(v - 1)^2$. Si supponga ora di voler calcolare il 3–momento. Allora, si può semplicemente sostituire $2v - 1$ con $v^3 - (v - 1)^3 = 3v^2 - 3v + 1$. Si ha che, $\sum_{v=1}^m 3v^2 - 3v + 1 = m^3$, si può quindi usare come stima del 3–momento la formula

$$n(3v^2 - 3v + 1)$$

dove $v = X.\text{val}$ è il valore associato ad una variabile X . In generale, si può stimare il k -momento per ogni $k \geq 2$ utilizzando un valore $v = X.\text{val}$ per la formula

$$n(v^k - (v - 1)^k)$$

Nella pratica, per aumentare la confidenza, si calcola $f(X) = n(v^k - (v - 1)^k)$ per tante variabili aleatorie mutualmente indipendenti X quante entrano in memoria. Si dividono poi le variabili aleatorie in sottogruppi, e si calcola la media di $f(X_j)$ su ogni X_j appartenente al sottogruppo, per ogni sottogruppo. Successivamente, si prende l'elemento mediano delle medie di ogni sottogruppo.

Flussi infiniti

Per il problema studiato, si è ottenuta una stima sotto l'ipotesi che la lunghezza del flusso n sia costante. Nella pratica, n cresce nel tempo, essendo la variabile che rappresenta il numero di input visti sino ad un certo istante.

Questo fatto in se stesso non causa problemi, essendo che vengono memorizzati solo i valori di alcune variabili, e, nell'istante in cui si vuole stimare il momento, si moltiplica una funzione definita su tali valori per n .

Se si conta il numero di elementi di un flusso visti, e si memorizza tale valore, che richiede solo $\log n$ bits, allora si può ottenere il valore n in ogni istante in cui risulti necessario.

Il problema risiede nella selezione delle posizioni per le variabili. Se tale selezione avviene una volta e si fissa, con la crescita del flusso, il conteggio effettuato sarà distorto (biased) a favore delle posizioni meno recenti, e la stima del momento risulterà essere troppo grande. D'altra parte, se si aspetta troppo tempo per scegliere le posizioni, allora nei primi istanti di analisi del flusso non si avranno molte variabili, e si otterrebbe quindi una stima non affidabile.

La tecnica che si utilizza per ovviare a questi problemi consiste nel mantenere tante variabili quanto sia possibile memorizzarne in ogni istante, ed eliminarne alcune man mano che il flusso cresce. Le variabili scartate vengono sostituite da delle nuove, in maniera tale che, in ogni istante, la probabilità di prendere una qualsiasi posizione per una variabile è uguale alla probabilità, per una variabile già presente in memoria, di venir scartata.

L'obiettivo è quindi che ogni istante iniziale t venga selezionato con probabilità k/n .

Tale approccio consiste proprio nel *fixed-size sampling*, ossia il campionamento di un sottoinsieme di dimensione fissata di elementi di un flusso che cresce all'infinito. Si utilizza quindi il Reservoir-Algorithm:

- Si scelgono i primi k elementi per k variabili.
- Quando l' n -esimo elemento arriva (con $n > k$), lo si inserisce nel campione con probabilità k/n .
- Se tale elemento è stato scelto, si elimina una delle variabili X memorizzate in precedenza, scegliendola u.a.r.

Link analysis

Si vogliono studiare problemi di Data Mining su reti sociali di enormi dimensioni.

Un esempio di queste reti è il Web.

Il Web può essere modellato come un grafo diretto, dove le pagine corrispondono ai nodi, e vi è un arco diretto dalla pagina p_1 alla pagina p_2 se vi sono uno o più link all'interno di p_1 che indirizzano a p_2 .

PageRank

Informalmente, il PageRank è un algoritmo utilizzato dai motori di ricerca, che permette di valutare l'importanza di pagine Web in maniera tale che tale parametro non venga manomesso facilmente.

Per entrare nei dettagli di tale funzionalità, ed individuare la motivazione del suo utilizzo, risulta necessario fornire un background su come operano i motori di ricerca.

In generale, il loro obiettivo è quello di, dato un insieme S di parole, trovare le pagine Web più rilevanti per S .

Prima di Google, i motori di ricerca lavoravano attraversando il Web, elencando i *termini* trovati in ogni pagina (dove con termine ci si riferisce a parole o stringhe di caratteri diversi dagli spazi bianchi), all'interno di un *indice invertito*, una struttura dati che permette di individuare facilmente, dato un termine, i puntatori a tutte le pagine web dove esso risulta essere presente.

Quando una *query di ricerca*, consistente in una lista di termini, veniva effettuata ad un motore di ricerca, le pagine contenenti quei termini venivano estratte dall'indice invertito, e classificate in base all'utilizzo di tali termini al loro interno. Ad esempio, la presenza di un termine nell'header di una pagina la rendeva più rilevante rispetto ad un'altra contenente lo stesso termine all'interno di una sezione di testo ordinario. Anche il numero di occorrenze di un dato termine influiva sulla classifica delle pagine estratte, infatti, maggiore era la presenza di un termine all'interno di una pagina, maggiore risultava essere la sua rilevanza.

Tale operazione è gestita da due moduli Software che costituiscono un motore di ricerca:

- Query Module: Converte una query scritta in linguaggio naturale, in una query avente un linguaggio comprensibile dal motore di ricerca, e consulta l'indice invertito, dal quale seleziona un insieme P di pagine che contengono i termini T della query.
Successivamente, il Query Module passa P al Ranking Module
- Ranking Module: Ha il compito di assegnare un punteggio ad ogni pagina $p \in P$, e di restituire P all'utente ordinato in base al punteggio. Inizialmente, il punteggio veniva determinato solamente in base al *Content Score*, ossia il punteggio del contenuto basato sui parametri descritti in precedenza. Si fa presente che tale componente è la più importante nel processo di ricerca di contenuti sul Web, perché evita all'utente di effettuare un'estensiva, e spesso impossibile, ricerca nelle migliaia di pagine rilevanti che restituisce il query module a seguito di una richiesta.

Questa modalità operativa dei motori di ricerca presentava delle debolezze, infatti, risultava possibile modificare impropriamente le pagine Web in maniera tale da ingannare il sistema di classificazione, portando i motori a fornire risultati non consoni alle richieste degli utenti. Ad esempio, si supponga di disporre di una pagina Web dedicata alla vendita di un prodotto. Per la modalità di funzionamento appena descritta, una maniera ingannevole per far sì che tale pagina venga visualizzata da un maggior numero di utenti consiste nell'inserire al suo interno un elevato numero di occorrenze di termini non inerenti al servizio fornito, cosicché questa appaia tra i primi risultati in risposta alle query di ricerca effettuate da un utente contenente tali termini.

Queste tecniche per manipolare i risultati forniti dai motori di ricerca prendono il nome di *term sparm*.

Google, per ovviare a tale problema, fornì due innovazioni:

1. Il PageRank, utilizzato per simulare il comportamento di navigatori del Web all'interno della rete. In particolar modo, simulava dove questi, partendo da una pagina casuale, tendevano a congregarsi nel caso cui avessero selezionato casualmente dei link uscenti dalla pagina in cui questi si trovavano in un dato istante, reiterando più volte tale processo. Le pagine che riportavano un grande numero di visitatori venivano considerate più importanti rispetto a pagine raramente visitate.
2. La valutazione del contenuto di una pagina non avveniva più semplicemente in base ai termini presenti al suo interno, ma anche mediante termini contenuti in altre pagine aventi un link diretto verso essa. In particolar modo, la valutazione prendeva in considerazione i termini vicini o contenuti all'interno di tali links. Questo perché, nonostante fosse facile per uno spammer aggiungere termini impropri in una pagina di sua proprietà, risultava difficile aggiungerli in pagine di proprietà altrui contenenti link diretti verso essa. Questo approccio di valutazione ha portato quindi alla definizione di un altro punteggio per il Ranking Module, ossia il popularity score (o autorità) di una pagina, il quale viene determinato da una link analysis della struttura del Web. Il Ranking Module quindi assegna un punteggio complessivo basato sul Content Score e sul Popularity Score. La rilevanza di una pagina è quindi calcolata tramite analisi dei link che la coinvolgono.

Queste due tecniche combinate fanno sì che il motore di ricerca non possa venir ingannato facilmente da comportamenti scorretti da parte di un proprietario di una pagina Web.

Intuitivamente, una semplice contromisura per contrastarle, può essere quella di creare diverse pagine Web, ciascuna delle quali contenenti link , aventi al loro interno i termini scelti per ingannare il sistema, diretti verso la pagina di interesse, ma tale approccio non risulterebbe funzionare essendo che tali pagine, a loro volta, non vengono referenziate da altre.

Le motivazioni per cui la simulazione di navigatori Web casuali permetta di approssimare l'importanza di una pagina sono le seguenti:

- Gli utenti del Web tendono ad inserire link diretti verso pagine che reputano di interesse, piuttosto che pagine inutili.

- Il comportamento di un navigatore casuale indica quali pagine saranno probabilmente visitate dagli utenti. E' più probabile che questi visitino pagine utili o di interesse piuttosto che pagine non rilevanti.

Formalmente, il PageRank è una funzione che assegna un numero reale a ciascuna pagina nel Web nota.

Maggiore è il PageRank di una pagina, maggiore sarà la sua importanza. Non esiste un algoritmo fissato per l'assegnamento del PageRank, e infatti, diverse varianti dell'idea di base possono portare a valutazioni diverse per la stessa pagina.

Si definisce quindi l'idea di base del PageRank, e si effettuano delle modifiche su di essa in maniera tale che risulti essere utilizzabile in determinati processi applicativi riguardanti la struttura del Web.

L'idea alla base dell'algoritmo per il calcolo del PageRank è quella che, un link da una pagina p_1 diretto verso una pagina p_2 è una *raccomandazione* per la pagina p_2 . In particolar modo, maggiore è il numero di raccomandazioni per una pagina, ossia maggiore è il numero di archi entranti nel nodo associato ad essa nella modellazione del Web mediante grafo, maggiore la pagina risulta essere importante. Un altro fattore che influenza il punteggio di popolarità per una pagina è dato dallo *stato* delle pagine che la puntano. In particolar modo:

- Maggiore è l'importanza di una pagina, maggiore sarà il suo peso sul fattore di popolarità delle pagine puntate da questa.
- Maggiore è il numero di pagine puntate da una data pagina, ossia maggiore è il numero di archi uscenti dal relativo nodo, minore sarà il peso di tali link sul fattore di popolarità delle pagine puntate.

Si danno due formulazioni equivalenti del PageRank:

- Modellazione algebrica
- Modellazione probabilistica

Modellazione algebrica del PageRank

Il PageRank può essere considerato come un metodo iterativo basato sull'analisi dei link entranti in una pagina, ossia, i soli link entranti in essa concorrono a determinarne il rank. L'idea è quella di considerare quindi i link come dei voti. Maggiore è il numero di link entranti in una pagina, maggiore è la sua importanza.

Informalmente, l'autorità di una pagina è un indice che esprime la sua rilevanza ai fini della ricerca.

Il popularity score di una pagina non dipende solamente dal numero di archi entranti in essa, ma anche dalla rilevanza che hanno tali pagine Web. Non tutti i link entranti in una pagina hanno quindi lo stesso peso sul come influenzano la sua popolarità, in particolar modo, i link provenienti da pagine importanti hanno un peso maggiore.

Bisogna ricordare anche che, maggiore è il numero di link uscenti da una pagina, minore

sarà la sua influenza sulle pagine puntate.

Si da quindi una formulazione ricorsiva del modello descritto.

- Ogni voto di un link ha un peso proporzionale all'autorità (importanza, rank) della sua pagina sorgente.
- Se la pagina j , avente importanza r_j , ha n links uscenti, ogni link ha un peso pari ad r_j/n . Tale peso corrisponde ai *voti* che andranno a influenzare la popolarità della pagina verso la quale l'arco è diretto.
- L'importanza della pagina j è data dalla somma dei voti presenti nei suoi archi entranti.

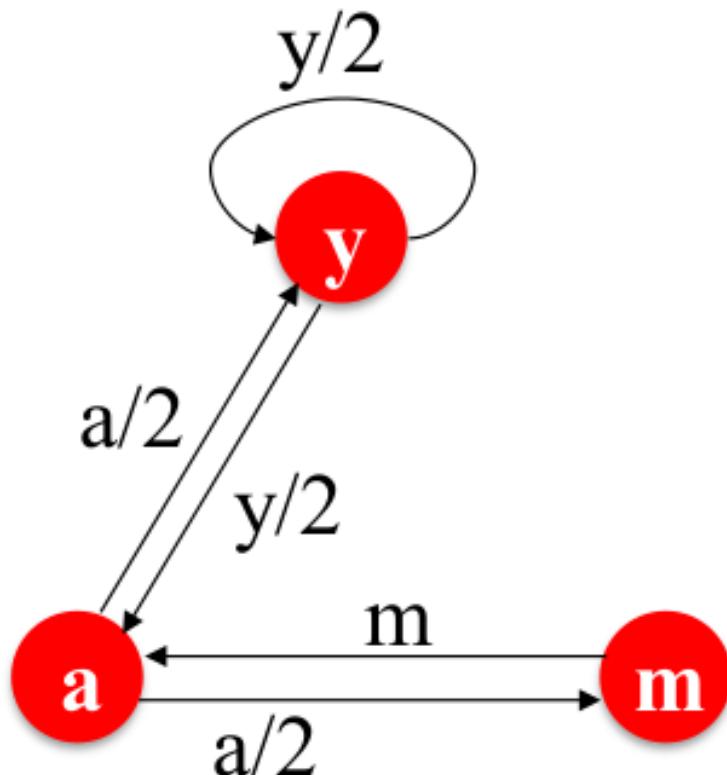
Quindi, il voto proveniente da una pagina importante ha un peso maggiore, ed, intuitivamente, una pagina è importante se è puntata da altre pagine importanti. Si definisce quindi il rank r_j di una pagina j come

$$r_j = \sum_{i \rightarrow j} \frac{r_i}{d_i}$$

dove d_i è il grado uscente del nodo i , ossia il numero di archi uscenti da esso.

Quindi l'importanza di una pagina è data dalla somma dei rapporti tra l'importanza delle pagine che la puntano e il loro grado uscente.

Si supponga ora di disporre di una rete costituita di pagine Web, per le quali si vuole calcolare l'importanza di ciascuna di esse. Allora, è possibile formulare tale scenario mediante un sistema lineare, dove le incognite sono date proprio dagli r_j per ogni pagina j . Tale modello prende il nome di *flow model*. Si fa un esempio



Si ha un sistema lineare costituito dalle equazioni che descrivono il flusso entrante in ogni nodo

$$\begin{cases} r_y = \frac{r_y}{2} + \frac{r_a}{2} \\ r_a = \frac{r_y}{2} + r_m \\ r_m = \frac{r_a}{2} \end{cases}$$

Si hanno 3 equazioni, 3 incognite e nessuna costante. Non si ha quindi un'unica soluzione.
Si pone quindi un ulteriore vincolo, che forza l'unicità della soluzione:

$$r_y + r_a + r_m = 1$$

Ossia, somma dell'autorità di tutte le pagine deve essere pari ad 1.

E si ha quindi

$$r_y = \frac{2}{5}, \quad r_a = \frac{2}{5}, \quad r_m = \frac{1}{5}$$

Il metodo dell'eliminazione Gaussiana per la risoluzione di sistemi lineari funziona per piccoli esempi, ma risulta necessario un metodo migliore per grafi di grande dimensione. Questo perché il metodo dell'eliminazione Gaussiana impiega tempo cubico nel numero di equazione.

Per fare ciò, si definisce una matrice di adiacenza M come segue:

Sia i una pagina avente d_i link uscenti.

Allora

$$M_{ji} = \begin{cases} \frac{1}{d_i} & \text{se } i \rightarrow j \\ 0 & \text{altrimenti} \end{cases}$$

Dove $i \rightarrow j$ indica che esiste un arco diretto da i a j . Tale matrice risulta essere stocastica rispetto alle colonne:

Definizione (Matrice stocastica per colonne)

Una matrice $M \in \mathbb{R}^{n \times n}$ si definisce stocastica per colonne se:

- $0 \leq M_{ji} \leq 1$ per ogni i, j .
- Per ogni i vale $\sum_{j=1}^n M_{ji} = 1$, ossia, per ciascuna colonna di M , la somma degli elementi è pari ad 1.

Si definisce un vettore r , avente un'entrata per ogni pagina. Per tale vettore, si ha che

- $r_j = \sum_{i \rightarrow j} \frac{r_i}{d_i}$ è il punteggio di importanza della pagina j .
- $\sum_j r_j = 1$, ossia r è un vettore di probabilità.

Si ricorda ora la definizione di autovettore.

Definizione (Autovettore)

Sia M una matrice quadrata, \mathbf{v} un vettore diverso dal vettore nullo, e λ un numero tale per cui

$$M\mathbf{v} = \lambda\mathbf{v}$$

Allora \mathbf{v} è detto un autovettore di M con autovalore λ .

Se \mathbf{v} è un autovettore di M con autovalore λ , allora lo sono anche tutti i multipli di \mathbf{v} diversi da 0.

Si danno ora i seguenti teoremi:

Teorema (Teorema di Perron semplificato)

Se A è una matrice reale $n \times n$ a elementi positivi, allora:

- A ha un autovalore $c \in \mathbb{R}^+$ tale che, $c > |c'|$ per ogni altro autovalore c' di A .
- L'autovettore x di A corrispondente a c è unico ed è a elementi reali e positivi, la cui somma è pari ad 1.

Essendo M una matrice stocastica, questa rispetta le ipotesi del teorema.

Inoltre:

Teorema

Se A è una matrice stocastica, allora A ha un autovalore λ tale che $|\lambda| = 1$ e λ è l'autovalore di modulo massimo di A

Tale autovalore prende il nome di autovalore dominante:

Definizione (Autovalore dominante)

Siano $\lambda_1, \lambda_2, \dots, \lambda_n$ gli autovalori di una matrice A $n \times n$. λ_1 è detto l'autovalore dominante di A se

$$|\lambda_1| > |\lambda_i|, \forall i = 2, \dots, n.$$

Gli autovettori associati a λ_1 sono chiamati autovettori dominanti di A .

Allora, il più grande autovalore di M è proprio $\lambda = 1$, e per il teorema di Perron, vi è un unico autovettore corrispondente a λ , la cui somma degli elementi è pari ad 1. Si vuole mostrare che r è proprio tale autovettore.

Si ha che

$$M \cdot r = \begin{bmatrix} M_{11} & M_{12} & \cdots & M_{1n} \\ M_{21} & M_{22} & \cdots & M_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ M_{n1} & M_{n2} & \cdots & M_{nn} \end{bmatrix} \begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_n \end{bmatrix} = \begin{bmatrix} \sum_i M_{1i}r_i \\ \sum_i M_{2i}r_i \\ \vdots \\ \sum_i M_{ni}r_i \end{bmatrix}$$

Sia

$$\sum_i M_{ji}r_i$$

il j -esimo elemento di $M \cdot r$. Essendo che

$$M_{ji} = \begin{cases} \frac{1}{d_i} & \text{se } i \rightarrow j \\ 0 & \text{altrimenti} \end{cases}$$

allora

$$\sum_i M_{ji}r_i = \sum_{i \rightarrow j} \frac{r_i}{d_i} = r_j$$

Le equazioni di flusso possono essere quindi scritte come

$$r = Mr$$

Si definisce quindi uno schema iterativo per individuare r , ossia l'autovettore dominante.

Si considera un Web graph con n nodi, dove i nodi sono le pagine e gli archi gli hyperlinks. Inoltre, si impone che tale grafo sia fortemente connesso. L'algoritmo opera come segue:

- Inizializza $\mathbf{r}^{(0)} = [1/n, \dots, 1/n]$.
- Itera: $\mathbf{r}^{(t+1)} = M \cdot \mathbf{r}^{(t)}$
- Termina quando $\|\mathbf{r}^{(t+1)} - \mathbf{r}^{(t)}\|_1 < \varepsilon$

Dove ε è una costante maggiore di 0 arbitraria molto piccola e $\|x\|_1 = \sum_{1 \leq i \leq n} |x_i|$ è la norma L_1 di x . Si osserva che $r_j^{(t+1)} = \sum_{i \rightarrow j} \frac{r_i^{(t)}}{d_i}$.

Bisogna ora mostrare il motivo per cui il metodo risulti essere funzionante, ossia, che la sequenza $M \cdot r^{(0)}, M^2 \cdot r^{(0)}, \dots, M^k \cdot r^{(0)}, \dots$ tende all'autovettore dominante di M , e che quindi l'algoritmo fornito dia una buona approssimazione.

Teorema

Se M è una matrice $n \times n$ diagonalizzabile con un autovalore dominante, allora esiste un vettore non negativo $r^{(0)}$ tale che la sequenza di vettori

$$M \cdot r^{(0)}, M^2 \cdot r^{(0)}, \dots, M^k \cdot r^{(0)}, \dots$$

tende all'autovettore dominante di M

Dimostrazione

Essendo M diagonalizzabile, si ha che questa ha n autovettori $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n$ linearmente indipendenti, con corrispondenti autovalori $\lambda_1, \lambda_2, \dots, \lambda_n$.

Si assume che questi autovalori siano ordinati in maniera tale che λ_1 sia l'autovalore dominante (con corrispondente autovettore \mathbf{x}_1). Essendo che gli n autovettori $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n$ sono linearmente indipendenti, allora questi devono formare una base per \mathbb{R}^n .

Per l'approssimazione iniziale $r^{(0)}$, si sceglie un qualsiasi vettore diverso da zero tale per cui la combinazione lineare

$$r^{(0)} = c_1 \mathbf{x}_1 + c_2 \mathbf{x}_2 + \dots + c_n \mathbf{x}_n$$

non abbia coefficiente c_1 uguale a zero. Questo perché, per $c_1 = 0$, il power method potrebbe non convergere, e risulterebbe necessario utilizzare un r_0 differente. Ora, moltiplicare per M entrambi i lati dell'equazione porta a

$$\begin{aligned} Mr^{(0)} &= M(c_1 \mathbf{x}_1 + c_2 \mathbf{x}_2 + \dots + c_n \mathbf{x}_n) \\ &= c_1(M\mathbf{x}_1) + c_2(M\mathbf{x}_2) + \dots + c_n(M\mathbf{x}_n) \\ &= c_1(\lambda_1 \mathbf{x}_1) + c_2(\lambda_2 \mathbf{x}_2) + \dots + c_n(\lambda_n \mathbf{x}_n) \end{aligned}$$

Moltiplicando per M entrambi i lati k volte si ottiene che

$$M^k r^{(0)} = c_1(\lambda_1^k \mathbf{x}_1) + c_2(\lambda_2^k \mathbf{x}_2) + \dots + c_n(\lambda_n^k \mathbf{x}_n)$$

che implica

$$M^k r^{(0)} = \lambda_1^k \left[(c_1 \mathbf{x}_1) + c_2 \left(\frac{\lambda_2^k}{\lambda_1^k} \right) \mathbf{x}_2 + \dots + c_n \left(\frac{\lambda_n^k}{\lambda_1^k} \right) \mathbf{x}_n \right]$$

Per ipotesi, essendo λ_1 il più grande autovalore in valore assoluto si ha che

$$\frac{\lambda_i}{\lambda_1} < 1 \quad \forall i = 2, \dots, n$$

in valore assoluto. Allora, per ogni i che va da 2 ad n si ha che

$$\left(\frac{\lambda_i}{\lambda_1} \right)^k$$

tende a 0 al tendere di k ad infinito. Questo implica che l'approssimazione

$$M^k r^0 \approx \lambda_1^k c_1 \mathbf{x}_1, \quad c_1 \neq 0$$

migliore all'aumentare di k .

Bisogna ora mostrare che M sia diagonalizzabile.

Per fare ciò, si osserva che se una matrice stocastica è irriducibile ed ammette un unico autovalore dominante uguale ad 1, allora la matrice è diagonalizzabile.

Una matrice si dice irriducibile se il suo grafo associato è fortemente connesso. Il grafo associato ad una matrice $A_{n \times n}$ è un grafo orientato con n nodi, dove esiste un arco $i \rightarrow j$ se e solo se $A_{ij} \neq 0$.

Ma allora il grafo associato ad M è proprio il grafo del Web con direzioni degli archi invertiti, infatti

$$M_{ji} = \begin{cases} \frac{1}{d_i} & \text{se } i \rightarrow j \\ 0 & \text{altrimenti} \end{cases}$$

di conseguenza il grafo associato ha un arco $j \rightarrow i$ se e solo se $i \rightarrow j$.

Essendo il web Graph fortemente connesso, lo è anche il grafo associato alla matrice. Per vedere meglio questa cosa, si ricorda che un grafo fortemente connesso ha un ciclo che attraversa tutti i suoi nodi. Se si invertono le direzioni degli archi, allora si ha tale ciclo attraversabile inversamente. Di conseguenza, M è diagonalizzabile.

Si ricorda inoltre che una matrice $n \times n$ è diagonalizzabile se e solo se possiede n autovettori linearmente indipendenti.

Modellazione probabilistica del PageRank

Si studia ora il PageRank facendo riferimento alla teoria delle catene di Markov.

Si supponga che un navigatore aleatorio si trovi, in un certo istante, su una pagina associata ad un nodo avente k archi uscenti, corrispondenti a dei link verso altre pagine. Allora, all'istante successivo, tale navigatore si troverà su ciascuna di queste pagine raggiungibili attraversando i link con probabilità $1/k$, ossia, segue un link uscente dal nodo in cui si trova scelto *u. a. r.*

Questo processo può quindi essere modellato mediante un particolare processo stocastico, che prende il nome di catena di Markov.

Definizione (Processo stocastico)

Un processo stocastico $\mathbf{X} = \{X(t) : t \in T\}$ è una collezione di variabili aleatorie.

L'indice t rappresenta un istante di tempo, e in tal caso il processo \mathbf{X} modella il valore di una variabile aleatoria X che cambia nel tempo.

Si definisce $X(t) = X_t$ come lo stato del processo all'istante t .

Se T è un insieme infinito numerabile, si dice che \mathbf{X} è un processo discreto.

Definizione (Catena di Markov)

Un processo stocastico X_0, X_1, X_2, \dots è una catena di Markov se

$$\Pr [X_t = a_t | X_{t-1} = a_{t-1}, X_{t-2} = a_{t-2}, \dots, X_0 = a_0] = \Pr [X_t = a_t | X_{t-1} = a_{t-1}]$$

Questa definizione esprime come lo stato di X_t dipenda dallo stato precedente X_{t-1} , ma come sia indipendente dal modo in cui il processo abbia raggiunto X_{t-1} . Questa proprietà prende il nome di *mancanza di memoria*. Bisogna far presente che questa proprietà non implica che X_t sia indipendente dalle variabili aleatorie X_0, X_1, \dots, X_{t-2} , ma implica solo che ogni dipendenza di X_t nel passato è catturata dal valore X_{t-1} . Senza perdita di generalità, si assume che lo spazi di stati discreti della catena di Markov sia $\{0, 1, 2, \dots, n\}$ (o $\{0, 1, 2, \dots\}$ se è infinito numerabile.) La probabilità di transizione

$$P_{ij} = \Pr [X_t = j | X_{t-1} = i]$$

è la probabilità che il processo si sposti da i a j in un passo. Per via della proprietà di mancanza di memoria, la catena di Markov è definibile da una matrice di transizione dove l'entrata nella riga i e colonna j è la probabilità P_{ij} . Da ciò segue che, per ogni i ,

$$\sum_{j \geq 0} P_{ij} = 1.$$

Si definisce una particolare catena di Markov:

Definizione (Processo di vita-e-morte)

Un processo (o catena) di vita-e-morte è una Catena di Markov avente insieme di stati $\mathcal{X} = \{0, 1, \dots, n\}$. In un passo, la catena può trovarsi allo stato successivo, allo stato precedente o rimanere nello stato in cui si trova. Le probabilità di transizione possono essere specificate dalle triple $\{(p_k, r_k, q_k)\}_{k=0}^n$ dove, per $k = 0, \dots, n$ e $p_k + r_k + q_k = 1$ si ha che

- p_k è la probabilità di passare dallo stato k allo stato $k+1$ quando $0 \leq k < n$.
- q_k è la probabilità di passare dallo stato k allo stato $k-1$ quando $0 < k \leq n$.
- r_k è la probabilità di rimanere a k per $0 \leq k \leq n$.
- $q_0 = p_n = 0$

In generale, è possibile definire la *matrice di transizione del Web* per descrivere cosa accade ad un navigatore aleatorio dopo un passo. Questa matrice M ha n righe ed n

colonne, se vi sono n pagine. L'elemento m_{ij} nella riga i e colonna j ha valore $1/k$ se la pagina j ha k archi uscenti, ed uno di questi è diretto verso la pagina i . Altrimenti, $m_{ij} = 0$.

Il comportamento di un navigatore aleatorio, in particolar modo, può essere descritto mediante una passeggiata aleatoria.

Definizione (Passeggiata aleatoria)

Sia $G = (V, E)$ un grafo non diretto connesso.

Una passeggiata aleatoria su G è una catena di Markov definita dalla sequenza di mosse di una particella che si muove tra i vertici di G . In questo processo, la posizione della particella in un dato istante di tempo definisce lo stato del sistema. Se la particella si trova sul vertice i ed i ha d_i archi incidenti, allora la probabilità che la particella attraversi l'arco (i, j) e si sposti su un vicino j è $1/d_i$

X_t è la pagina visitata da un navigatore aleatorio all'istante t , e rappresenta quindi lo stato del processo in tale istante.

Ad ogni istante t , l'utente può trovarsi in una tra le n pagine, corrispondenti agli n stati della catena di Markov.

Si assume che quando un utente si trovi sulla pagina i al tempo t , allora la probabilità di trovarsi su una pagina j all'istante $t + 1$ dipende solo dal fatto che l'utente si trovi sulla pagina i , e non dalle pagine visitate precedentemente.

La distribuzione di probabilità per la posizione di un navigatore aleatorio può essere descritta da un vettore colonna $\mathbf{p}(t) = \mathbf{p}_t$ il cui j -esimo componente è la probabilità che esso si trovi alla pagina j all'istante t . \mathbf{p}_t è una quindi distribuzione di probabilità sulle pagine, essendo $\sum_i (\mathbf{p}_t)_i = 1$.

Si supponga quindi di avere n pagine nel Web, e che un navigatore aleatorio si trovi inizialmente su una di esse con probabilità $1/n$. Allora, il vettore iniziale \mathbf{p}_0 avrà $1/n$ per ogni componente. Se M è la matrice di transizione del Web, allora dopo uno step, la distribuzione del navigatore sarà $\mathbf{p}_1 = M\mathbf{p}_0$, dopo due step sarà $\mathbf{p}_2 = M\mathbf{p}_1 = M(M\mathbf{p}_0) = M^2\mathbf{p}_0$ e, proseguendo in questa maniera, si ha che $\mathbf{p}_t = M\mathbf{p}_{t-1} = M^t\mathbf{p}_0$ è la distribuzione di probabilità per la posizione del navigatore dopo t passi.

Si osserva ora perché il prodotto tra un vettore di distribuzione \mathbf{p} ed una matrice di transizione M è uguale alla distribuzione $\mathbf{x} = M\mathbf{p}$ del passo successivo:

La probabilità x_i che un navigatore aleatorio si trovi al nodo i al passo successivo è

$$\sum_j m_{ij}\mathbf{p}_j$$

In questa sommatoria, m_{ij} è la probabilità che un navigatore al nodo j si sposti verso il nodo i al passo successivo, e \mathbf{p}_j è la probabilità che esso si sia trovato sul nodo j al passo precedente.

Si supponga ora che la passeggiata aleatoria raggiunga uno stato

$$\mathbf{p}(t+1) = M\mathbf{p}(t) = \mathbf{p}(t)$$

allora $p(t)$ è la distribuzione stazionaria del cammino aleatorio.

Sia \mathbf{P} una matrice $n \times n$ di transizione che descrive una catena di Markov.

Una distribuzione stazionaria π è un vettore tale per cui:

- $\sum_i \pi_i = 1, \pi_i \in [0, 1]$
- $\mathbf{P}\pi = \pi$

Dove π è un vettore colonna.

Questo significa che partendo da una situazione iniziale aleatoria con distribuzione di probabilità π , dopo un passo, e quindi dopo un numero arbitrario di passi, essendo

$$\mathbf{P}^n\pi = \mathbf{P}\mathbf{P}^{n-1}\pi = \mathbf{P}^{n-1}\pi = \dots = \pi$$

la distribuzione di probabilità è sempre π .

Una distribuzione stazionaria esiste sempre per una catena di Markov, ma non è garantito che sia unica.

Se esiste una sola distribuzione di probabilità stazionaria allora si ha che

$$\pi = \lim_{k \rightarrow \infty} \mathbf{P}^k x$$

dove x è una distribuzione generica sugli n stati (ossia è un vettore n -dimensionale le quali entrate sono minori o uguali ad 1, e la loro somma è proprio pari ad 1.)

Si osserva che tale condizione è proprio quella che rende valido il Power Method.

Lo stato $\mathbf{p}(t+1) = \mathbf{p}(t)$ si raggiunge se si verificano due condizioni:

1. Il grafo è fortemente connesso, ossia esiste un cammino per ogni coppia di nodi del grafo.
2. Non ci sono nodi pozzo, ossia nodi privi di archi uscenti.

La distribuzione stazionaria viene raggiunta quando il prodotto della distribuzione \mathbf{p} con la matrice M non cambia la distribuzione, ossia, \mathbf{p} è un autovettore di M .

Anche in questo caso, essendo M stocastica, \mathbf{p} è l'autovettore dominante, ossia il suo autovalore associato è il più grande tra tutti gli autovalori. Inoltre, sempre per via del fatto che tale matrice è stocastica, l'autovalore associato all'autovettore principale è uguale ad 1.

Il vettore r soddisfa $r = M \cdot r$, quindi r è una distribuzione stazionaria per la passeggiata aleatoria.

Si è detto che se esiste una sola distribuzione di probabilità stazionaria, si ha che

$$\pi = \lim_{k \rightarrow \infty} \mathbf{P}^k x$$

e che quindi l'applicazione del Power Method porti al vettore desiderato.

Si vedono quindi delle condizioni sufficienti per l'esistenza e l'unicità della distribuzione stazionaria per il processo del random surfer su un grafo.

Si ha che per grafi che soddisfano determinate condizioni, la distribuzione stazionaria è

unica, e questa verrà raggiunta mediante il metodo iterativo indipendentemente dalla distribuzione di probabilità iniziale all'istante $t = 0$.

Una tipologia di grafi che soddisfano queste condizioni sono le catene di Markov ergodiche. Informalmente, una catena di Markov è ergodica se esiste un cammino tra ogni coppia di stati (grafo formalmente connesso), e che gli stati non siano partizionati in insiemi tali per cui tutte le transizioni di stato avvengano ciclicamente da un insieme all'altro. Si descrive ora questo concetto formalmente. Per fare ciò, bisogna individuare delle proprietà di una catena di Markov ergodica.

Definizione (Catena di Markov irriducibile)

Sia S l'insieme degli stati di una catena di Markov.

Una catena di Markov si dice irriducibile se tutti gli stati sono raggiungibili dagli altri stati, ossia, per ogni $i, j \in S$, esiste un $t \in N$ tale per cui

$$p_{ij}^{(t)} > 0$$

dove $p_{ij}^{(t)}$ indica la probabilità che in t passi venga raggiunto lo stato j a partire da i .

Si introduce ora il concetto di periodo di una catena di Markov.

Informalmente, il "periodo" di una catena di Markov è una caratteristica che riflette la regolarità con cui un sistema può ritornare a uno stato particolare nel corso del tempo. Il periodo è definito come il massimo comun divisore di tutte le lunghezze dei cicli indipendenti presenti nella catena di Markov.

Un "ciclo indipendente" in una catena di Markov si riferisce a un insieme di stati attraverso i quali è possibile raggiungere uno stato qualsiasi all'interno del ciclo e ritornare al punto di partenza. Questo ciclo indipendente può essere attraversato più volte e in modi diversi, ma è indipendente da altri cicli presenti nella catena di Markov.

Una catena di Markov può essere classificata come "aperiodica" quando il suo periodo è 1. In altre parole, se il massimo comun divisore delle lunghezze dei cicli indipendenti è 1, allora la catena di Markov è aperiodica. In una catena di Markov aperiodica, non esiste un ciclo fisso regolare di ritorno ad uno stato, e quindi il processo non segue un modello di comportamento ciclico.

Per capire meglio:

- **Periodo:** Se la catena di Markov ha un periodo maggiore di 1, significa che esiste una certa regolarità nella frequenza con cui il sistema ritorna a uno stato specifico. Ad esempio, se il periodo è 2, la catena di Markov può tornare a uno stato solo in passi di lunghezza pari.
- **Aperiodica:** Se il periodo è 1, la catena di Markov è aperiodica e non segue un ciclo regolare. Questo implica che il sistema può ritornare a uno stato in passi di lunghezza

qualsiasi, senza seguire un modello ciclico prevedibile.

In sintesi, il periodo e l'aperiodicità in una catena di Markov descrivono la regolarità o l'assenza di essa nel ritorno a determinati stati nel corso del tempo.

📘 Definizione (Catena di Markov Aperiodica)

Il periodo di uno stato $j \in S$ è il più grande $\xi \in \mathbb{N}$ tale per cui

$$\{n \in \mathbb{N} \mid p_{jj}^{(n)} > 0\} \subseteq \{i \cdot \xi \mid i \in \mathbb{N}\}$$

Uno stato con periodo $\xi = 1$ è detto aperiodico, e la catena di Markov si dice aperiodica se lo sono tutti i suoi stati.

Si da ora la definizione di catena di Markov ergodica

📘 Definizione (Catena di Markov ergodica)

Se una catena di Markov è irriducibile e aperiodica, allora è detta ergodica.

Si da quindi il seguente teorema

📘 Teorema

Se una catena di Markov è ergodica, allora si ha che

$$\lim_{t \rightarrow \infty} \mathbf{P}^t x = \pi$$

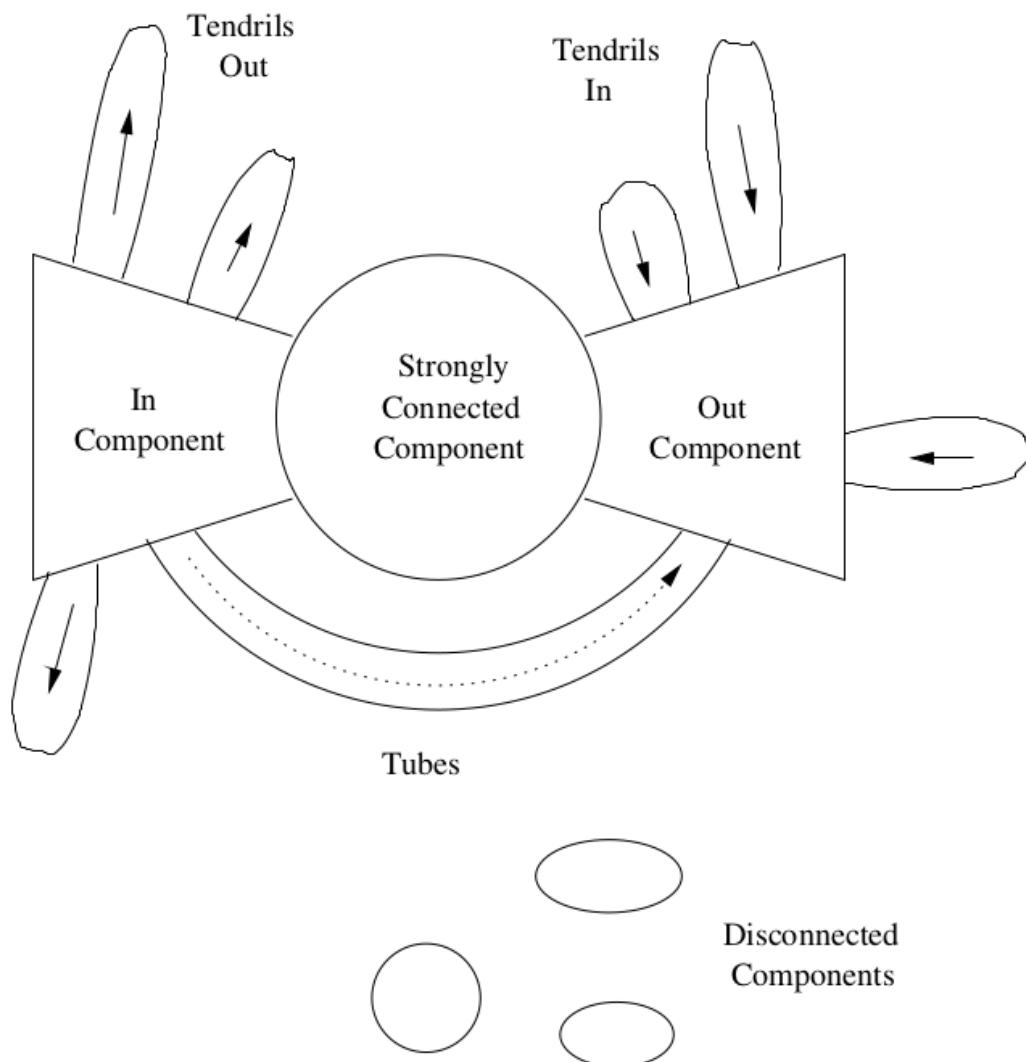
dove π è l'unica distribuzione stazionaria della catena e x è una distribuzione generica sugli n stati (ossia è un vettore n -dimensionale le quali entrate sono minori o uguali ad 1, e la loro somma è proprio pari ad 1.)

📘 Osservazione

- Il teorema è vero indipendentemente dalla distribuzione iniziale.
- La distribuzione stazionaria di una catena di Markov ergodica può essere quindi approssimata efficientemente, moltiplicando iterativamente un vettore con la matrice della catena.

Si ok ma tanto il grafo del web non è fortemente连通的

Le analisi per il calcolo del PageRank fatte sino ad ora si sono basate sull'ipotesi che il grafo del Web fosse fortemente connesso. Nella realtà dei fatti, il grafo del web presenta la struttura in figura.



Ed è descritta dalle seguenti componenti:

1. Una componente centrale, di grandi dimensioni, fortemente connessa.
2. Componente entrante, consistente in pagine in grado di raggiungere la componente fortemente connessa seguendo i links, ma non raggiungibili dalla componente fortemente connessa.
3. Componente uscente, consistente in pagine raggiungibili dalla componente fortemente connessa, ma non in grado di raggiungere la componente fortemente connessa.
4. I *tendrils* (in Italiano, viticci), distinguibili in due categorie. La prima comprende viticci consistenti da pagine raggiungibili dalla componente entrante, ma non in grado di raggiungere tale componente. La seconda comprende viticci in grado di raggiungere la componente uscente, ma non raggiungibili dalla componente uscente.
5. Tubi, i quali consistono in pagine raggiungibili dalla componente entrante e in grado di raggiungere la componente uscente, ma non in grado di raggiungere o di essere raggiunti dalla componente fortemente connessa.
6. Componenti isolati, irraggiungibili dalle porzioni di grande dimensione del grafo.

Queste strutture violano le assunzioni necessarie affinché la power iteration converga ad un limite. In particolar modo, si ha che tale grafo non risulti essere ergodico. Ad esempio, quando un navigatore aleatorio entra nella componente uscente, questo non può mai abbandonarla. Quindi, i navigatori che iniziano l'esplorazione della rete nella componente fortemente connessa o nella componente entrante tenderanno ad accumularsi o nella componente uscente, o in un viticcio della componente entrante. Quindi, nessuna pagina nella componente fortemente connessa o nella componente entrante avrà alcuna probabilità di essere raggiunta a partire da tali stati. Se si interpreta questa probabilità come misura dell'importanza di una pagina, si concluderebbe (in maniera incorretta) che nessuna delle pagine nella componente fortemente connessa e nella componente entrate siano di rilevanza.

Bisogna quindi modificare il PageRank affinché si prevengano queste anomalie. I problemi principali sono due:

- I vicoli ciechi, ossia pagine non aventi link uscenti. I navigatori che raggiungono tali pagine spariscono, e il risultato è che nel vettore di distribuzione stazionaria nessuna pagina che raggiunge un vicolo cieco può avere un PageRank. In particolar modo, la passeggiata aleatoria non può proseguire e si ha una "fuori uscita" di importanza.
- Trappole di ragno, gruppi di pagine aventi tutti archi uscenti che permettono di navigare tale porzione del grafo, ma che non escono mai da tale componente. Ciò comporta che le passeggiate aleatorie si "blocchino" all'interno di tali trappole, ossia, che i navigatori aleatori possono solamente viaggiare sugli stati della catena associati a tali pagine, le quasi assorbiranno poi tutta l'importanza

Entrambi questi problemi possono essere risolti da un metodo chiamato *tassazione*, o teletrasporto, per il quale si assume che un navigatore aleatorio abbia una probabilità finita di abbandonare il grafo del Web ad ogni passo del processo e, a risposta di ciò, che un nuovo agente inizi una passeggiata aleatoria partendo da un nodo casuale.

Evitare vicoli ciechi e trappole di ragno

Per evitare i problemi appena definiti, si può modificare il calcolo del PageRank dando ad ogni navigatore aleatorio una piccola probabilità di teletrasportarsi in una pagina casuale, piuttosto che seguire un link uscente dalla loro pagina uscente. Il passo iterativo, dove si calcola un nuovo vettore \mathbf{r}' che stima del PageRank calcolato a partire dal PageRank corrente \mathbf{r} e la matrice di transizione M è

$$\mathbf{r}' = \beta M \mathbf{r} + \left[\frac{1 - \beta}{n} \right]_n$$

dove β è una costante scelta, tipicamente contenuta tra 0.8 e 0.9 ed n è il numero di nodi nel grafo del Web. Il termine $\beta M \mathbf{r}$ rappresenta il caso in cui, con probabilità β , il navigatore aleatorio decide di seguire un arco uscente dalla pagina in cui si trova. Il termine $\left[\frac{1 - \beta}{n} \right]_n$ è un vettore con tutte le componenti aventi valore $\frac{1 - \beta}{n}$ e rappresenta la l'introduzione, con probabilità $1 - \beta$, di un nuovo navigatore aleatorio in una pagina casuale. Si nota che se il

grafo non ha vicoli ciechi, allora la probabilità di introdurre un nuovo navigatore aleatorio è esattamente uguale alla probabilità che il navigatore decida di non seguire un link uscente dalla pagina in cui si trova. In questo caso, si può visualizzare il navigatore come se, ad ogni passo, dovesse decidere di seguire un link uscente dalla pagina in cui si trova, oppure teletrasportarsi su una pagina casuale. Se invece vi sono vicoli ciechi, allora vi è una terza possibilità, per la quale il navigatore non transita dalla pagina in cui si trova. Essendo che il termine $\left[\frac{1-\beta}{n}\right]_n$ non dipende dalla somma delle componenti del vettore \mathbf{r} , vi sarà sempre una frazione di un viaggiatore che opera nella rete, ossia, quando vi sono vicoli ciechi, la somma delle componenti di \mathbf{r} può essere minore di 1, ma non raggiungerà mai 0.

Dunque la soluzione per uscire dai vicoli ciechi consiste nel teletrasportare il random surfer con probabilità 1 su una pagina casuale.

In conclusione, ad ogni passo il random surfer con probabilità β segue uno dei link uscenti dalla pagina in cui si trova, mentre con probabilità $1-\beta$ salta ad una pagina random. Dunque l'equazione del PageRank diventa

$$r_j = \sum_{i \rightarrow j} \beta \frac{r_i}{d_i} + (1 - \beta) \frac{1}{n}$$

che è l'equazione del PageRank di Google definita da Brin e Page nel 1998. Questa formulazione assume che non ci siano vicoli ciechi in M . Se in M sono presenti vicoli ciechi, allora si hanno due opzioni:

- Preprocessare la matrice M in modo da eliminare tutti i vicoli ciechi;
 - In modo esplicito, seguire un random teleport link con probabilità 1 dai vicoli ciechi.
- Implementando il teleporting, la matrice M diventa la matrice

$$A = \beta M + (1 - \beta) \left[\frac{1}{n} \right]_{n \times n}$$

dove $\left[\frac{1}{n} \right]_{n \times n}$ è una matrice $n \times n$ con tutte le entrate pari a $\frac{1}{n}$.

Si ottiene dunque l'equazione

$$\mathbf{r} = A\mathbf{r}$$

dove \mathbf{r} si può ottenere utilizzando il Power Method.

Per calcolare il PageRank \mathbf{r} , l'operazione cruciale è il prodotto matrice vettore: facile se si ha sufficiente memoria per memorizzare $A, \mathbf{r}_n, \mathbf{r}_o$ dove $\mathbf{r}_n = A\mathbf{r}_o$.

Rearrangando l'equazione $\mathbf{r} = A\mathbf{r}$, dove $A_{ji} = \beta M_{ji} + \frac{1-\beta}{n}$, si ottiene

$$\begin{aligned}
r_j &= \sum_{i=1}^n A_{ji} r_i \\
r_j &= \sum_{i=1}^n \left[\beta M_{ji} + \frac{1-\beta}{n} \right] \cdot r_i \\
&= \sum_{i=1}^n \beta M_{ji} r_i + \frac{1-\beta}{n} \underbrace{\sum_{i=1}^n r_i}_{\sum_{i=1}^n r_i = 1} \\
&= \sum_{i=1}^n \beta M_{ji} r_i + \frac{1-\beta}{n}
\end{aligned}$$

dunque si ottiene il vettore

$$\mathbf{r} = \beta M \mathbf{r} + \left[\frac{1-\beta}{n} \right]_n$$

dove $\left[\frac{1-\beta}{n} \right]_n$ è un vettore ad n entrate di valore $\frac{1-\beta}{n}$.

Dunque ad ogni iterazione del power method, si deve:

- Calcolare $\mathbf{r}_n = \beta M \cdot \mathbf{r}_o$;
- Sommare il valore costante $\frac{1-\beta}{n}$ ad ogni entrata del vettore \mathbf{r}_n ;

Si fanno ora considerazioni sullo spazio occupato e sui tempi di esecuzione dell'algoritmo.

La matrice M è sparsa, dunque è sufficiente memorizzare in memoria solo le entrate della matrice diverse da zero. Dato che il numero di entrate diverse da zero è proporzionale al numero di link, possibile memorizzare M con spazio proporzionale al numero di link. Generalmente i link sono miliardi, dunque si possono memorizzare solamente in memoria secondaria.

Dunque si analizza il costo di una iterazione del Power method.

Assumendo che il vettore \mathbf{r}_n sia mantenibile in memoria centrale, e che M ed \mathbf{r}_o siano memorizzati su disco, si ottiene che ad ogni iterazione si deve

- leggere \mathbf{r}_o e M dal disco;
- scrivere \mathbf{r}_n su disco;

dunque ogni iterazione del Power method costa

$$|\mathbf{r}_n| + |\mathbf{r}_o| + |M| = 2|\mathbf{r}| + |M|$$

Topic page rank

Tra i diversi miglioramenti che si possono fare al PageRank, si studia il topic-specific page rank, con lo scopo di valutare le pagine web non solo in base alla loro popolarità, ma anche in base a quanto sono vicine a un particolare argomento, ad esempio "sport" o "storia". Il meccanismo per applicare questa valutazione consiste nel modificare il comportamento dei random surfer, favorendo lo spostamento su pagine che trattano un particolare argomento.

Questo meccanismo permette al motore di ricerca di rispondere alle query in base agli interessi dell'utente. Ad esempio, spesso interessi diversi vengono espressi utilizzando lo stesso termine in una query. Per esempio, la query "jaguar" potrebbe fare riferimento all'animale, all'automobile o a una versione del sistema operativo MAC. Se il motore di ricerca può dedurre che l'utente è interessato alle automobili, ad esempio, può fare un lavoro migliore nel restituire pagine rilevanti all'utente.

Biased Random Walk

Per il problema dei punti ciechi e spider-trap, la soluzione è rendere il random walker in grado di teletrasportarsi su una qualsiasi pagina, in modo equiprobabile, con una bassa probabilità $(1 - \beta)$ ad ogni step della sua passeggiata.

Per implementare il topic-sensitive page rank, si utilizza la stessa tecnica del teletrasporto applicata però ad un insieme specifico di pagine rilevanti per un certo topic. Dunque il random walker con probabilità $(1 - \beta)$ si teletrasporta su una pagina appartenente ad uno specifico sottoinsieme di pagine S detto teleport set. Si osserva che S contiene solamente delle pagine che sono rilevanti rispetto ad uno specifico topic, dunque il vettore risultante r_S del PageRank con teleport set S sarà differente da un qualsiasi altro vettore risultante $r_{S'}$ per un differente teleport set S' .

In particolare, si modifica la parte del teleporting nella formulazione del page rank per renderlo topic specific nel modo seguente.

$$A_{ij} = \begin{cases} \beta M_{ij} + \frac{1-\beta}{|S|} & \text{se } i \in S \\ \beta M_{ij} + 0 & \text{altrimenti} \end{cases}$$

In questa formulazione, un random walker si teletrasporta su una delle pagine in S con probabilità uniforme: è possibile anche assegnare diversi pesi alle pagine in S , cioè rendere, nel momento del teletrasporto, più probabile il salto su specifiche pagine di S .

Per integrare il topic specific PageRank in un motore di ricerca, si deve:

1. Decidere k argomenti per i quali creare vettori PageRank specifici per tali topic.
2. Scegliere un teleport set S_k per ciascuno di questi argomenti e utilizzare tale insieme per calcolare il vettore topic specific PageRank r_k per ogni argomento.
3. Trovare un modo per determinare l'argomento o l'insieme di argomenti più rilevanti per una particolare query di ricerca.
4. Utilizzare i vettori PageRank per quell'argomento o quegli argomenti nell'ordinamento delle risposte alla query di ricerca.

Il terzo passo è probabilmente il più complicato, e sono stati proposti diversi metodi.

Alcune possibilità sono:

5. Consentire all'utente di selezionare un argomento da un menu;
6. Inferire l'argomento/i dalle parole che appaiono nelle pagine web recentemente cercate dall'utente, o dalle query recenti emesse dall'utente;

7. Inferire l'argomento/i dalle informazioni sull'utente, ad esempio i loro segnalibri o i loro interessi dichiarati su Facebook.

Combattere il Web Spam: TrustRank

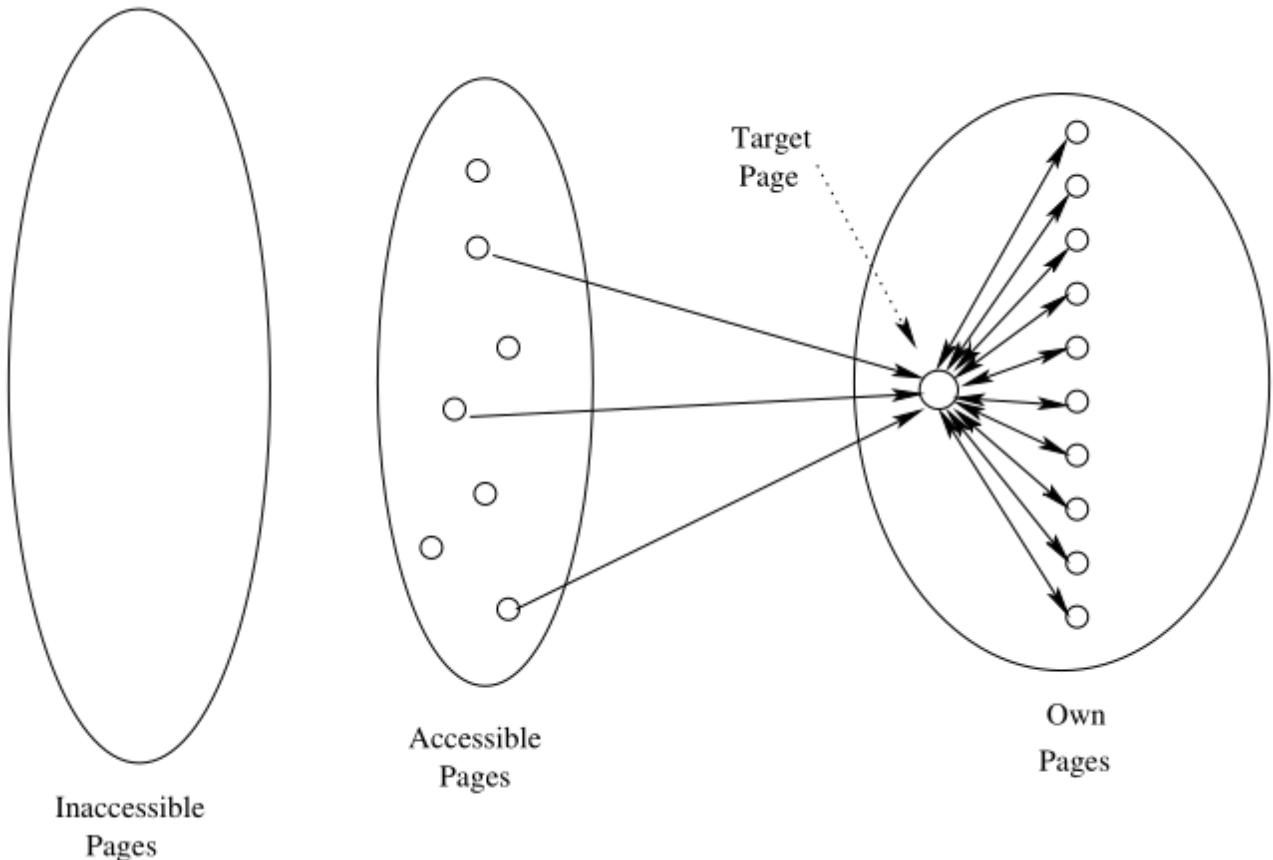
Con spamming, si intende un qualsiasi tentativo intenzionale mirato a migliorare il posizionamento di una pagina web nei risultati dei motori di ricerca, senza essere in linea con il vero valore della pagina stessa. Tutte le pagine web che sono il risultato dello spamming sono dette pagine spam. Approssimativamente, il 15% delle pagine web sono spam.

Le tecniche di spam prima dell'introduzione del PageRank consistevano nel cosiddetto term spam, cioè inserire nelle pagine termini che non sono in linea con il topic della pagina in modo da far apparire la pagina anche in query di ricerca che riguardano topic differenti da quelli trattati dalla pagina stessa. In questo modo, la pagina viene potenzialmente visitata da più utenti, anche quelli che non sono direttamente interessati. Questa tecnica era efficace rispetto ai primi motori di ricerca, che indicizzavano le pagine rispetto alle parole contenute all'interno di esse e rispondevano alle query di ricerca, cioè una lista di parole, con la lista delle pagine che contenevano tali parole.

Con l'introduzione del PageRank questa tecnica perde di efficacia. Infatti, il PageRank misura l'*importanza* di una pagina in base al numero e all'importanza delle pagine web che puntano a tale pagina. Intuitivamente, con il PageRank una pagina viene valutata in base a cosa dicono gli altri della pagina, e non in base al contenuto.

Quando è diventato evidente che il PageRank rendeva inefficace il term spam, gli spammer hanno cominciato a utilizzare metodi progettati per ingannare l'algoritmo del PageRank facendo sovrastimare il ranking di determinate pagine. Le tecniche per aumentare artificialmente il PageRank di una pagina sono collettivamente chiamate *link spam* e consistono nel creare strutture di link che aumentano il PageRank di una particolare pagina.

Un'insieme di pagine il cui scopo è aumentare il PageRank di una determinata pagina o pagine è detta spam farm. La figura sotto mostra una semplice struttura di una spam farm.



1. Pagine inaccessibili: le pagine che lo spammer non può influenzare. La maggior parte del Web si trova in questa parte.
2. Pagine accessibili: quelle pagine che, sebbene non siano controllate dallo spammer, possono essere influenzate da quest'ultimo.
3. Pagine proprie: l'insieme delle pagine che lo spammer detiene e controlla.

La spam farm consiste:

- Nelle pagine detenute dallo spammer, organizzate in modo speciale come si vede a destra;
- Un insieme di link dalle pagine accessibili alle pagine dello spammer. Senza questi link dall'esterno, la spam farm sarebbe inutile, poiché non verrebbe nemmeno indicizzata da un tipico motore di ricerca.

Si osserva che nonostante possa sembrare sorprendente che si possano influenzare delle pagine senza possederle, in realtà ad oggi molti siti permettono (e invitano) la pubblicazioni di propri commenti sul sito. Sfruttando questo fatto, per esempio, uno spammer può ottenere il maggior flusso di PageRank verso le proprie pagine pubblicando molti commenti del tipo "Sono d'accordo. Si prega di vedere il mio articolo su www.mySpamFarm.com."

Analisi di una Spam Farm

Nella spam farm è presente una pagina target t ; lo spammer ha l'obiettivo di massimizzare il valore del PageRank di t . Inoltre, ci sono un gran numero m di pagine di supporto, che accumulano una porzione del PageRank che viene distribuita equamente a tutte le pagine (la frazione $1 - \beta$ del PageRank che rappresenta i surfers che vanno su una pagina casuale). Le pagine di supporto impediscono anche che il PageRank di t venga perso, per quanto possibile, poiché una parte verrà tassata ad ogni round. Si noti che t ha un link verso ogni pagina di supporto, e ogni pagina di supporto ha un link solo verso t .

Supponiamo che il PageRank venga calcolato utilizzando un parametro di tassazione β , cioè la frazione del PageRank di una pagina che viene distribuita ai suoi successori al turno successivo, cioè le pagine a cui punta. Sia n il numero totale di pagine sul Web. Alcune di esse costituiscono una spam farm della forma suggerita nella figura, con una pagina target t e m pagine di supporto. Sia x la quantità di PageRank contribuita dalle pagine accessibili. Quindi, x è la somma, su tutte le pagine accessibili p che hanno un link verso t , del PageRank di p moltiplicato per β , diviso per il numero di pagine a cui punta p . Infine, sia y il PageRank sconosciuto di t .

Il valore del PageRank per ogni pagina di supporto nella spam farm è pari a

$$\frac{\beta y}{m} + \frac{1 - \beta}{n}$$

dove il primo termine rappresenta il contributo dato dalla pagina t . Il PageRank y di t viene tassato, quindi solo βy viene distribuito alle pagine puntate da t . Quel PageRank è diviso equamente tra le m pagine di supporto. Il secondo termine è la quota della pagina di supporto della frazione $1 - \beta$ del PageRank che viene divisa equamente tra tutte le pagine sul Web.

Si calcola ora il PageRank y della pagina target t . Questo valore arriva da tre sorgenti:

1. Il contributo x che arriva dalle pagine esterne;
2. β volte il PageRank di ogni pagina di supporto nella spam farm, ossia

$$\beta \left(\frac{\beta y}{m} + \frac{1 - \beta}{n} \right)$$

3. $\frac{1-\beta}{n}$ ovvero la quota della frazione $(1 - \beta)$ del PageRank che appartiene a t . Questa quantità è trascurabile e verrà eliminata per semplificare l'analisi.

Dunque da (1) e (2) (non si considera (3) essendo quantità trascurabile), si ottiene

$$y = x + \beta \left(\frac{\beta y}{m} + \frac{1 - \beta}{n} \right) = x + \beta^2 y + \beta(1 - \beta) \frac{m}{n}$$

Infine, risolvendo per y , si ottiene

$$y = \frac{x}{1 - \beta^2} + c \frac{m}{n}$$

$$\text{dove } c = \frac{\beta(1-\beta)}{(1-\beta^2)} = \frac{\beta}{(1+\beta)} .$$

Scegliendo $\beta = 0.85$, si ottiene $1/(1 - \beta^2) = 3.6$, e $c = \frac{\beta}{1-\beta^2} = 0.46$.

Ciò significa che la struttura ha amplificato la contribuzione esterna del PageRank del 360%, e ha ottenuto anche una quantità di PageRank che rappresenta il 46% della frazione del Web presente nella spam farm., ossia $\frac{m}{n}$.

Combattere il Link Spam

È diventato essenziale per i motori di ricerca individuare ed eliminare il link spam, proprio come era necessario eliminare il term spam.

Ci sono due approcci per il link spam. Uno è cercare strutture come la spam farm in figura, dove una pagina linka a un numero molto grande di pagine, ognuna delle quali linka nuovamente ad essa. I motori di ricerca dunque cercano tali strutture ed eliminano quelle pagine dal loro indice. Questo fa sì che gli spammer sviluppino diverse strutture che hanno essenzialmente lo stesso effetto di catturare il PageRank per una pagina o pagine target. Non c'è essenzialmente fine alle possibili variazioni della struttura della farm spam in figura, quindi questa guerra tra gli spammer e i motori di ricerca probabilmente continuerà per molto tempo.

Tuttavia, esiste un altro approccio per eliminare che si basa sulla modifica della definizione di PageRank per abbassare automaticamente il rank delle pagine spam. Considereremo due definizioni differenti:

- TrustRank: una variazione del topic-specific PageRank, dove il teleport set è costituito da pagine trusted; ad esempio, pagine con un dominio universitario o governativo. Così facendo, il teleport non è più uniforme su tutte le pagine, ma è biased su queste pagine fidate.
- Spam Mass: un calcolo che identifica le pagine che probabilmente sono spam e consente al motore di ricerca di eliminare tali pagine o di abbassarne fortemente il loro PageRank.

TrustRank

TrustRank è un topic-specific PageRank dove il topic è un insieme di pagine ritenute *affidabili* (non spam). L'idea è che mentre una pagina di spam potrebbe facilmente essere creata per collegarsi a una pagina affidabile, è *improbabile* che una pagina affidabile si collega a una pagina di spam.

Per definire l'insieme delle pagine affidabili, si può ad esempio creare automaticamente un sample di pagine web (cioè un insieme di pagine) e passare questo insieme ad un operatore umano in grado di identificare le pagine affidabili e scartare le pagine spam: è un lavoraccio, quindi il sample di partenza deve essere il più piccolo possibile.

Dato l'insieme delle pagine affidabili, si esegue un topic-specific PageRank impostando come teleport set questo insieme di pagine affidabili: questo permette di propagare la fiducia attraverso i link alle diverse pagine, dando ad ognuna di esse un *valore di fiducia* compreso tra 0 e 1.

Il modello (semplificato) è il seguente.

Ad ogni pagina affidabile si assegna un valore di fiducia pari ad 1.

Sia t_p la fiducia della pagina p e sia o_p l'insieme delle pagine puntate da p . Allora per ogni $q \in o_p$, la pagina p conferisce una *fiducia* a q pari a

$$\frac{\beta t_p}{|o_p|} \quad \text{per } 0 < \beta < 1$$

Infine la fiducia di una pagina p è additiva, dunque la fiducia di p è data dalla somma della fiducia assegnata a p da ogni pagine che puntano verso p .

Questo modello rispetta due caratteristiche importanti:

- Trust attenuation: la quantità di fiducia conferita da una pagina affidabile *decresce* al crescere della distanza della pagina nel grafo;
- Trust splitting: la fiducia è divisa tra gli out-links.

Si fanno due considerazioni importanti sull'insieme delle pagine candidate ad essere affidabili, cioè l'insieme di pagine che deve essere ispezionato da un umano:

- Questo insieme deve essere il più piccolo possibile;
- Si deve assicurare che ogni pagina affidabile riceva un rank di affidabilità adeguato, dunque si devono scegliere le pagine trusted iniziali in modo che tutte le pagine non spam siano raggiungibili con percorsi brevi.

Per scegliere buoni insiemi di possibili pagine affidabili, ci sono due approcci principali:

1. supponendo di voler scegliere k pagine, si prendono le prime k pagine date dal PageRank tradizionale: l'idea è che le pagine spam, per quanto possano salire nel ranking, non raggiungeranno posizioni molto alte nel ranking;
2. scegliere le pagine appartenenti a domini affidabili e controllati: ad esempio pagine con dominio .edu, .gov, .mil.

Spam Mass

L'idea dietro lo spam mass è quella di misurare per ogni pagina la frazione del suo PageRank che proviene dallo spam. Tale misura si ottiene calcolando sia il PageRank ordinario che il TrustRank basato su un insieme di pagine affidabili. Si supponga che la pagina p abbia un PageRank r e un TrustRank t . Allora la *spam mass* di p è $\frac{r-t}{r}$. Una *spam mass* negativa o poco positiva indica che p probabilmente non è una pagina di spam, mentre una *spam mass* vicina a 1 suggerisce che la pagina probabilmente è spam.

Dunque è possibile eliminare le pagine con una *spam mass* elevata dall'indice delle pagine Web utilizzato da un motore di ricerca, eliminando così una grande quantità di link spam senza dover identificare particolari strutture utilizzate dagli spammer.

Community Detection in Large Networks

Introduzione

Si vuole studiare il problema di identificare delle comunità all'interno di una rete sociale. Una comunità può essere vista come un sottoinsieme dei nodi della rete, aventi tra loro delle connessioni particolarmente forti. Alcune tecniche utilizzate per identificare delle comunità sono simili agli algoritmi di clustering. Bisogna far presente, però, che le comunità non partizionano quasi mai l'insieme dei nodi di una rete, ma spesso queste si sovrappongono, ossia, al contrario dei clusters, un nodo può appartenere a più comunità.

Si modella una rete sociale con un grafo.

In generale, le caratteristiche essenziali di una rete sociale sono le seguenti:

1. Vi è una collezione di entità che partecipano nella rete.
2. Vi è almeno una relazione tra le entità di una rete.
3. Vi è un'assunzione di località. Questa condizione è difficile da formalizzare, ma l'intuizione su cui si basa è data dal fatto che le relazioni tendono a raggrupparsi, ossia, se l'entità A all'interno della rete ha una relazione sia con B che con C , allora vi è un'alta probabilità che tra B e C sussista una relazione.

Le reti sociali vengono modellate naturalmente mediante grafi. Le entità della rete sono i nodi, e un arco connette due nodi se tra questi sussiste una relazione che caratterizza la rete. Se è presente un grado, o un peso, associato alla relazione, questo grado è rappresentato come etichetta degli archi.

Un esempio sono le reti telefoniche. I nodi rappresentano i numeri di telefono, i quali corrispondono a degli individui. Vi è un arco tra due nodi se una chiamata è stata eseguita tra i telefoni a loro associati in un lasso di tempo fissato. Gli archi possono, ad esempio, venir pesati dal numero di chiamate effettuate da questa coppia di telefoni durante il periodo di interesse. Le comunità in una rete telefonica formano gruppi di persone che comunicano frequentemente, come gruppi di amici o conoscenti.

Algoritmo di Kruskal per grafi pesati

Per grafi pesati metrici, si può fare riferimento al problema del Clustering.

Il problema riguarda la classificazione per similitudine di n oggetti p_1, \dots, p_n di un dato universo U , in un certo numero di gruppi tali che oggetti all'interno di uno stesso gruppo sono il più simili

possibile. Il livello di similitudine tra coppie di oggetti è dato dalla funzione distanza tra oggetti,

cioé un valore numerico che indica la vicinanza tra i due oggetti. Ad esempio nel caso di immagini

può essere la vicinanza dei colori nei pixel, oppure può essere la distanza euclidea nel caso di punti

del piano. Intuitivamente, due oggetti con una grande valore di distanza tra loro sono poco, o per nulla, simili. Data la funzione distanza, il problema fondamentale che si pone è quello di dividere gli oggetti in k partizioni, che prendono il nome di clusters, in modo che:

- gli oggetti appartenenti a stessi cluster siano vicini (simili) tra loro;
- gli oggetti in cluster diversi siano distanti (diversi) tra loro.

Si formalizza quindi il problema del clustering di spacing massimo:

Input: Insieme di n elementi $p_1, \dots, p_n \in U$, una matrice D di dimensione $n \cdot n$, dove $d[i][j] = d(p_i, p_j) \forall i, j$, e il numero k di gruppi che si vogliono ottenere. La funzione distanza $d(p_i, p_j)$,

indica la similitudine tra una qualunque coppia di elementi. Dunque se $d(p_i, p_j) \rightarrow 0$, allora p_i e p_j

sono simili, mentre se $d(p_i, p_j) \rightarrow \infty$, allora p_i e p_j sono diversi. La funzione distanza soddisfa le seguenti proprietà.

- $d(p_i, p_j) \geq 0 \forall i, j$ (non negativa);
- $d(p_i, p_j) = 0$ se e solo se $i = j$
- $d(p_i, p_j) = d(p_j, p_i)$ (simmetria)

Soluzione ammissibile: Una qualunque k partizione $C = \{C_1, C_2, \dots, C_k\}$ di U , ossia $\bigcup_{i=1}^k C_i = U$ e $C_i \cap C_j = \emptyset$ per ogni i, j .

Costo di una soluzione ammissibile: è definito dallo spacing, ovvero la distanza minima tra qualsiasi coppia di punti che si trovano in cluster diversi, formalmente

$$cost(C) = \min\{d(p_h, p_l) : p_h \in C_i \wedge p_l \in C_j, \forall i, j \in 1, \dots, k \wedge i \neq j\}$$

Goal: Massimizzare il costo della soluzione ammissibile, cioè trovare un k -clustering di spacing massimo.

Bisogna ora modellare il problema opportunamente.

Viene abbastanza naturale farlo mediante l'uso di grafi pesati, cioè si rappresenta l'istanza I del problema attraverso un grafo G completo e pesato, dove

- Gli oggetti di $\mathcal{U} = \{p_1, \dots, p_n\}$ sono i nodi del grafo, dunque $V = \mathcal{U}$;
- L'insieme degli archi è composto da tutte le coppie non ordinate di V , dunque il grafo è completo;
- Il costo di un generico arco $e = i, j$ è pari alla distanza $d(i, j)$, per ogni arco di E .

Allora una soluzione ammissibile al problema sarà una qualsiasi k -partizione dell'insieme dei nodi V di G .

Un algoritmo con approccio greedy che risolve il problema del clustering dato il grafo G come istanza è formato dai seguenti passi.

- Si parte da una n partizione di G dove ogni partizione è un cluster contenente un solo nodo;
- Avendo le distanze tra coppie di oggetti (archi di G) ordinati in ordine crescente, scelgo la più piccola distanza $d(p, p')$ tale che gli oggetti (nodi) p e p' non sono nello stesso cluster, e fondo i due clusters a cui appartengono p e p' in un unico cluster;
- Ripeto questa operazione per $n - k$ volte, cioè fino a che non raggiungo esattamente k clusters.

Notiamo che la procedura è esattamente l'algoritmo di Kruskal, cambia solamente che la procedura termina quando ci sono k componenti connesse.

L'algoritmo di Kruskal, dato un grafo pesato $G = (V, E)$, restituisce il suo albero dei cammini minimi T . Partendo da $T = \emptyset$, considera gli archi in ordine non decrescente rispetto al peso. Si aggiunge l'arco e in T se fare ciò non comporta la creazione di un ciclo in T .

Inoltre, l'algoritmo equivale a trovare l'MST T per G e da esso eliminare i $k - 1$ archi più costosi, formando così proprio k componenti connesse.

Questa seconda osservazione serve a dimostrare la correttezza dell'algoritmo.

Teorema

Sia $\mathcal{C}^* = \{C_1^*, \dots, C_k^*\}$ il clustering formato eliminando i $k - 1$ archi più costosi da un MST T di G .

Allora \mathcal{C}^* è un k -clustering di spacing massimo.

Dimostrazione

Lo spacing di C^* è esattamente la lunghezza d^* del $(k - 1)$ -esimo arco più costoso di T ; sarebbe la lunghezza dell'arco che l'algoritmo di Kruskal avrebbe aggiunto a T se non lo avessimo fermato.

Sia C un altro clustering $\{C_1, \dots, C_n\}$, dimostriamo che lo spacing d di C non può essere migliore dello spacing d^* di C^* .

Siccome i due clustering sono differenti, esisteranno due nodi p, p' tali che entrambi si trovano nello stesso cluster in C^* , chiamiamolo C_r^* , ma che si trovano in cluster differenti in C , siano tali clusters C_s e C_t .

Consideriamo allora questa coppia di nodi tale che $p \in C_s$ e $p' \in C_t$.

Siccome i due nodi fanno parte di C_r^* , allora tutti gli archi sul cammino $p \rightarrow p'$ hanno lunghezza minore o uguale a d^* .

Questo è vero perché l'algoritmo di Kruskal ha inserito questi archi prima dell'arco di lunghezza d^* .

Allora se e è l'arco del cammino $p \rightarrow p'$ che collega i due cluster C_s e C_t di C , allora $d \leq e \leq d^*$, e dunque lo spacing d di C è minore o uguale dello spacing d^* di C^*

Algoritmo di Girvan-Newman per grafi non pesati

Si studia ora un metodo per la community detection di grafi non pesati non metrici.

Si hanno diversi metodi euristici per partizionare un grafo in comunità, dove con comunità si intende, genericamente, un insieme coeso di nodi.

A grandi linee, è possibile classificare le tecniche per il partizionamento di grafi in metodi partitivi e metodi agglomerativi:

- In un metodo partitivo si inizia considerando l'intero grafo come un'unica grande comunità e poi, man mano, si rimuovono gli archi sino a quando il grafo risulta partizionato in componenti connesse. Il procedimento viene poi iterato su ciascuna componente, sino a quando si ottiene un insieme di comunità di dimensioni ritenute adeguate.
- In un metodo agglomerativo si inizia considerando ciascun nodo come una piccola comunità e poi, man mano, si aggiungono gli archi del grafo sino a quando si ottengono un numero di comunità ritenuto adeguato, o comunità di dimensioni ritenute adeguate.

Entrambi i metodi permettono di ottenere partizionamenti nidificati:

- In un metodo partitivo, ad ogni passo si ottengono delle comunità contenute in quelle ottenute al passo precedente.
- In un metodo agglomerativo, ad ogni passo si ottengono comunità che contengono quelle ottenute al passo precedente

Ottenendo quindi uno schema di partizionamento ad albero. I diversi metodi proposti si distinguono in base al criterio utilizzato per scegliere, ad ogni passo:

- Quale arco del grafo rimuovere (in un metodo partitivo)
- Quale arco del grafo aggiungere (in un metodo agglomerativo)

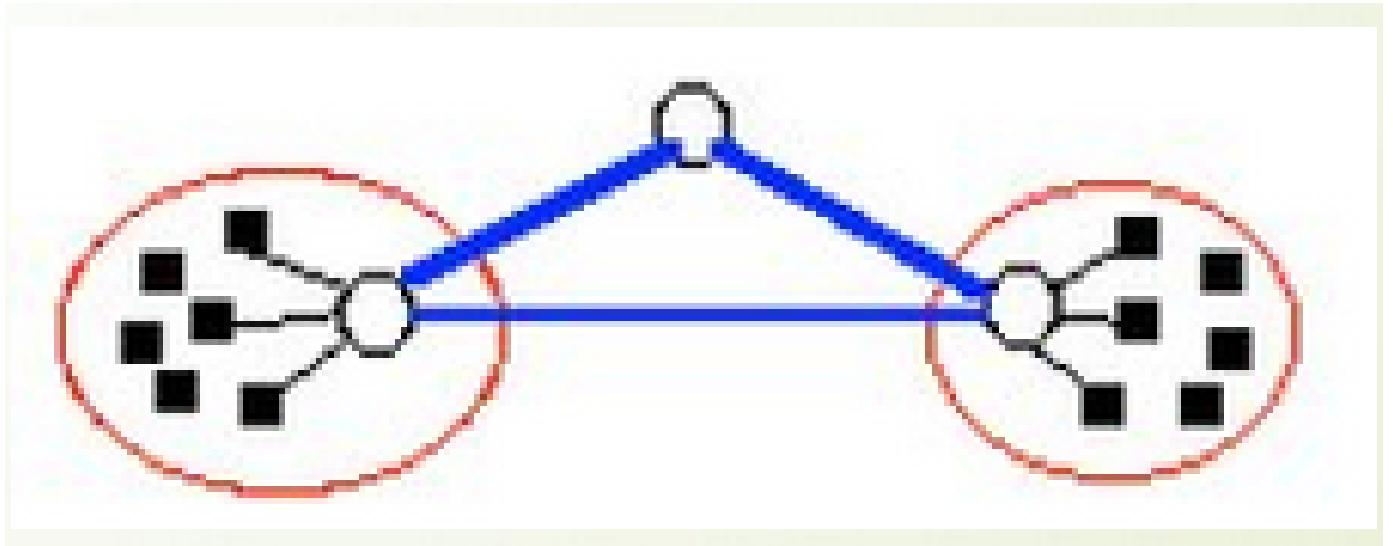
Si studia ora un criterio per rimuovere gli archi in un metodo partitivo per grafi non pesati. Il criterio è basato sul concetto di betweenness di un arco, a sua volta basato sui concetti di bridge e local bridge:

- Un bridge connette due regioni del grafo altrimenti non connesse. La rimozione quindi di un arco ponte disconnette la rete. In particolar modo, data una componente连通的, la rimozione di un bridge porta alla definizione di due componenti connesse distinte.
- Un local bridge connette due regioni che, senza di esso, sarebbero connesse in modo meno efficiente. Più formalmente, un arco è un local bridge se i suoi estremi non hanno vicini in comune: $(u, v) \in E$ è un local bridge se $N(u) \cap N(v) = \emptyset$, dove $N(u) = \{x \in V : (u, x) \in E\}$.

Quindi bridges e local bridges connettono regioni che, senza di loro, avrebbero difficoltà ad interagire.

A partire da queste considerazioni si intuisce che, rimuovendo bridges e local bridges, si può partizionare il grafo in componenti che possono essere considerate comunità.

Possono però verificarsi scenari dove non si hanno local bridge, ma due regioni dense, come in figura:



Si utilizza quindi una proprietà differente da quella del local bridge, basata sulla nozione di traffico.

E' possibile modellare il traffico all'interno di una rete come un flusso di un fluido: per ogni coppia di nodi s e t , si assuma che s voglia ad inviare a t una unità di fluido, che si suddivide equamente tra tutti i cammini minimi che collegano s a t . La betweenness di un arco è la quantità totale di fluido che lo attraversa, ottenuta sommando le frazioni di fluido per tutte le coppie $\langle s, t \rangle$. Formalmente, dato un grafo $G = (V, E)$ non orientato, per ogni coppia di nodi $s, t \in V$ e per ogni arco $(u, v) \in E$ si definisce

$$\sigma_{st}(u, v) = \text{numero di shortest paths fra } s \text{ e } t \text{ che attraversano } (u, v)$$

La betweenness relativa di (u, v) rispetto alla coppia $\langle s, t \rangle$, indicata con $b_{st}(u, v)$ è la frazione degli shortest paths fra s e t che attraversano (u, v) :

$$b_{st}(u, v) = \frac{\sigma_{st}(u, v)}{\sigma_{st}}$$

dove σ_{st} è il numero totale di shortest paths fra s e t .

Infine, la betweenness $b(u, v)$ di un arco $(u, v) \in E$ è la semi-somma delle betweenness relative ad ogni coppia di nodi

$$b(u, v) = \frac{1}{2} \sum_{s, t \in V} b_{st}(u, v)$$

Si studia ora il metodo di Girvan-Newman, un metodo partitivo basato sulla betweenness:

- Si inizia considerando l'intero grafo come un'unica grande comunità.

- Si calcola poi l'arco di betweenness massima e si rimuove: se il grafo residuo è non connesso allora è stata ottenuta una prima partizione in comunità.
- Il procedimento viene poi iterato calcolando gli archi di betweenness massima nel grafo rimanente e rimuovendoli.
- Il procedimento termina quando si è ottenuto un livello di granularità ritenuto adeguato.

Si vuole vedere ora come calcolare le betweenness degli archi. Si esclude immediatamente la possibilità di enumerare tutti gli shortest paths all'interno di un grafo, essendo che il loro numero è esponenziale nelle dimensioni del grafo.

Si fornisce quindi lo schema di un algoritmo per il calcolo delle betweenness degli archi:

- Per ogni $s \in V$ esegui i seguenti tre passi:
 1. Si calcola il sottografo $T(s)$ degli shortest paths uscenti da s mediante una Breadth First Search.
 2. Mediante una visita top-down di $T(s)$, per ogni $v \in V$, si calcola σ_{sv} .
 3. Mediante una visita bottom-up di $T(s)$, e usando quanto calcolato al punto (2), per ogni $(u, v) \in T(s)$ si calcola $b_s(u, v) = \sum_{t \in V - \{s\}} b_{st}(u, v)$
- Per ogni $(u, v) \in E$, infine, si calcola $b(u, v) = \frac{1}{2} \sum_{s \in V} b_s(u, v)$.

Si osserva esplicitamente che al punto (3) si calcola $b_s(u, v)$ per i soli archi (u, v) in $T(s)$. Infatti, gli archi che non sono in $T(s)$ non fanno parte di alcuno shortest path uscente da s .

Si vedono ora in dettaglio i tre passi.

Passo (1):

Si calcola $T(s)$ come insieme di archi e, contemporaneamente, una partizione in livelli di V .

Si pone inizialmente $L_0 \leftarrow \{s\}$ come livello 0, e $T(s) \leftarrow \emptyset$.

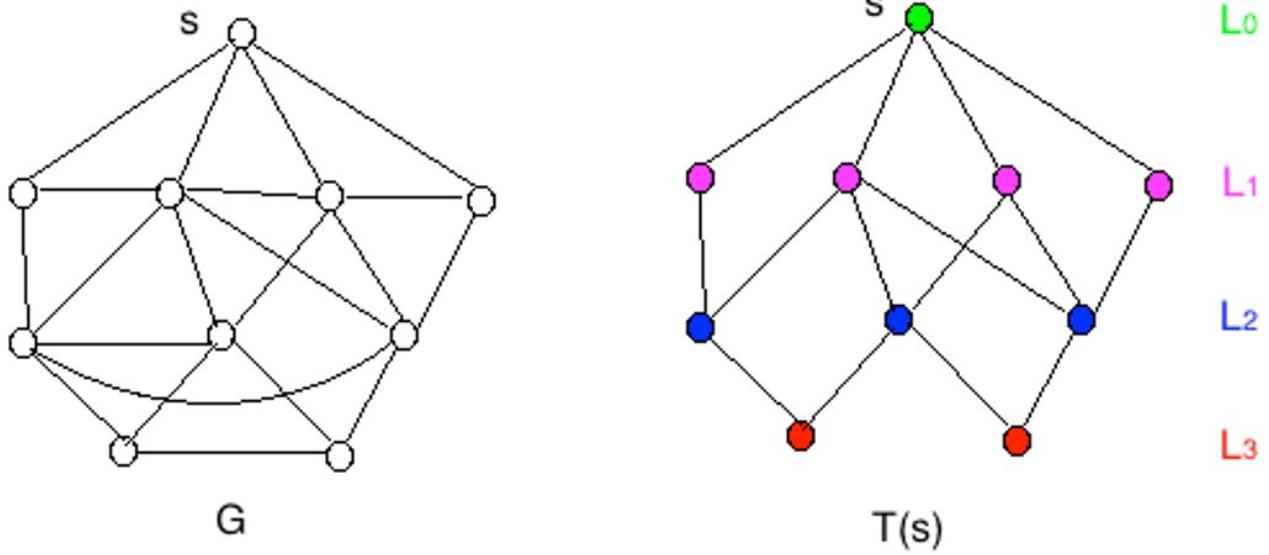
Per $h \geq 0$ e sino a quando $L_h \neq \emptyset$, calcola

$$L_{h+1} \leftarrow \{u \in V - \bigcup_{0 \leq i \leq h} L_i : \exists v \in L_h \text{ t.c. } (v, u) \in E\}$$

e

$$T(s) \leftarrow T(s) \cup \{(v, u) \in E : v \in L_h \wedge u \in L_{h+1}\}$$

Esempio:

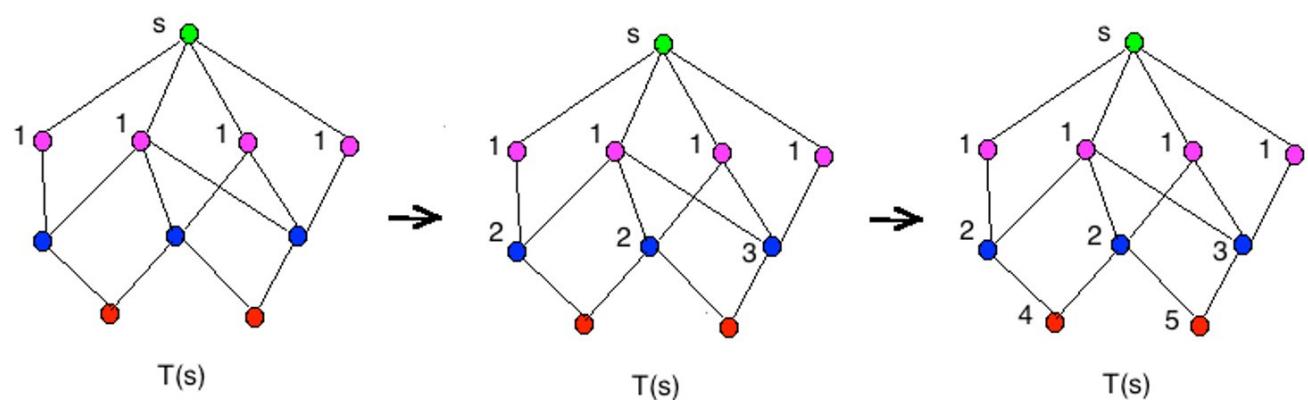


Passo (2):

Mediante una visita top-down di $T(s)$, per ogni $v \in V$, si calcola σ_{sv} .

Si osserva che il numero di shortest paths dalla radice s ad un nodo $v \in L_h$ è pari alla somma dei numeri dei percorsi da s a qualunque "padre" di v , ossia, una volta calcolato σ_{su} per ogni $u \in L_{h-1}$, e inizializzando $\sigma_{su} = 1$ per ogni $u \in L_1$, si può calcolare

$$\sigma_{sv} = \sum_{u \in L_{h-1}} \sigma_{su} \text{ per ogni } v \in L_h.$$



Passo (3) :

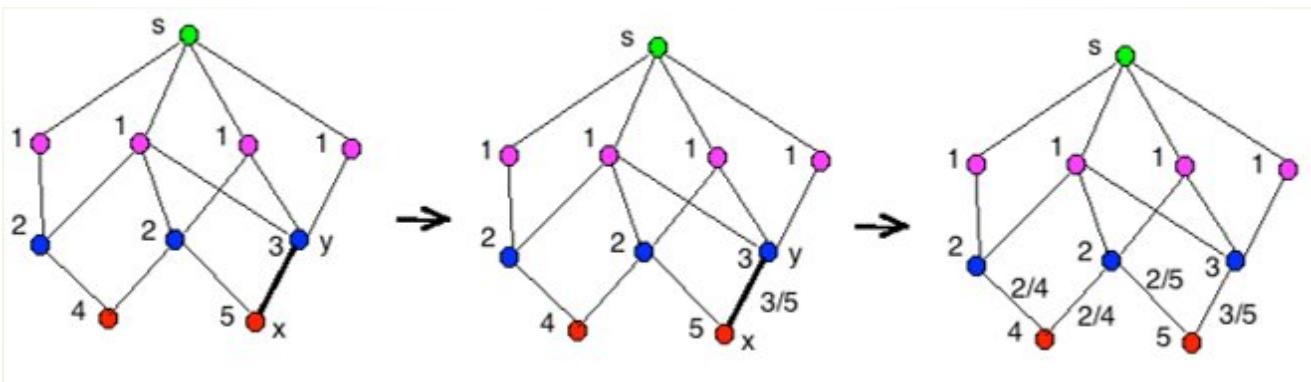
Si calcola ora, per ogni $(u, v) \in T(s)$, $b_s(u, v) = \sum_{t \in V - \{s\}} b_{st}(u, v)$.

Sia d il numero di livelli di $T(s)$.

Sia $(y, x) \in T(s)$ con $x \in L_d$. Gli unici shortest paths uscenti da s che passano attraverso (y, x) sono gli shortest paths da s ad y . Ciò implica che $b_{sz}(x, y) = 0$ per ogni $z \neq x$, e dunque, se $x \in L_d$, allora

$$b_s(y, x) = \sum_{z \in V} b_{sz}(y, x) = b_{sx}(y, x) = \frac{\sigma_{sx}(y, x)}{\sigma_{sx}} = \frac{\sigma_{sy}}{\sigma_{sx}}$$

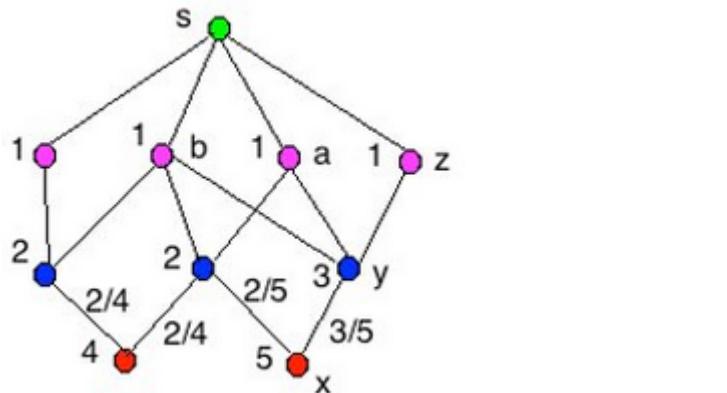
dove si ha che $\sigma_{sx}(y, x) = \sigma_{sy}$ perché il numero di shortest paths da s ad x che attraversano (x, y) è uguale al numero di shortest paths da s ad y .



Sia $(z, y) \in T(s)$ con $y \notin L_d$. Gli shortest paths che passano attraverso (z, y) sono alcuni shortest paths da s a y , più precisamente, per lo stesso ragionamento fatto nel caso precedente, quelli che passano attraverso z , e, per ogni discendente x di y , alcuni shortest path da s ad x , più precisamente quelli che passano attraverso z e attraverso y . Perciò, $b_s(z, y)$, ossia la frazione di tutti gli shortest paths uscenti da s che passa attraverso l'arco (z, y) , è la somma dei seguenti termini:

- $\frac{\sigma_{sz}}{\sigma_{sy}}$, ossia la frazione degli shortest paths da s ad y che passa attraverso (z, y) .
- Per ogni discendente x di y , una frazione $\frac{\sigma_{sz}}{\sigma_{sy}}$ della frazione di shortest paths da s ad x che passano attraverso (y, x) , ossia, $\frac{\sigma_{sz}}{\sigma_{sy}} \times \frac{\sigma_{sy}}{\sigma_{sx}}$.

Vediamo un esempio altrimenti non capiamo un cazzo

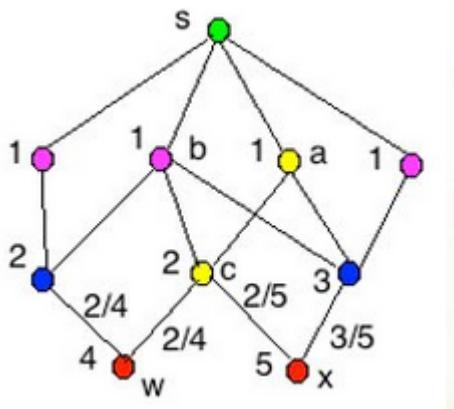


I $3/5$ degli shortest paths da s ad x passano per (y, x) .

Di questi $3/5$, $1/3$ passa per (z, y) , $1/3$ per (a, y) e $1/3$ per (b, y) .

Perciò, per (z, y) passano: $1/3$ degli shortest paths da s ad y e $1/3$ ($3/5$) degli shortest paths da s ad x .

Dunque, $b_s(z, y) = \frac{1}{3} + \frac{1}{3} \times \frac{3}{5} = \frac{8}{15}$, perché $b_{st}(z, y) = 0$ per ogni t che non è successore di y .



Si consideri ora l'arco (a, c) . Una frazione pari ad $1/2$ degli shortest path da s a c passano per l'arco (a, c) , ossia $\frac{\sigma_{sa}}{\sigma_{sc}} = \frac{1}{2}$. Si considerano ora i nodi w e x discendenti di c .

- $2/5$ degli shortest path da s a x passano per l'arco (c, x) : di questi, $1/2$ passano per l'arco (b, c) e $1/2$ passano per l'arco (a, c) ;
- $2/4$ degli shortest path da s a w passano per l'arco (c, w) : di questi, $1/2$ passano per l'arco (b, c) e $1/2$ passano per l'arco (a, c) ;

Dunque vale

$$b_s(a, c) = \frac{1}{2} + \frac{1}{2} \times \frac{2}{5} + \frac{1}{2} \times \frac{2}{4} = \frac{19}{20}$$

Più in dettaglio, la procedura può essere formalizzata come segue:

for $\{h \leftarrow d; h > 0; h \leftarrow h - 1\}$ **do**

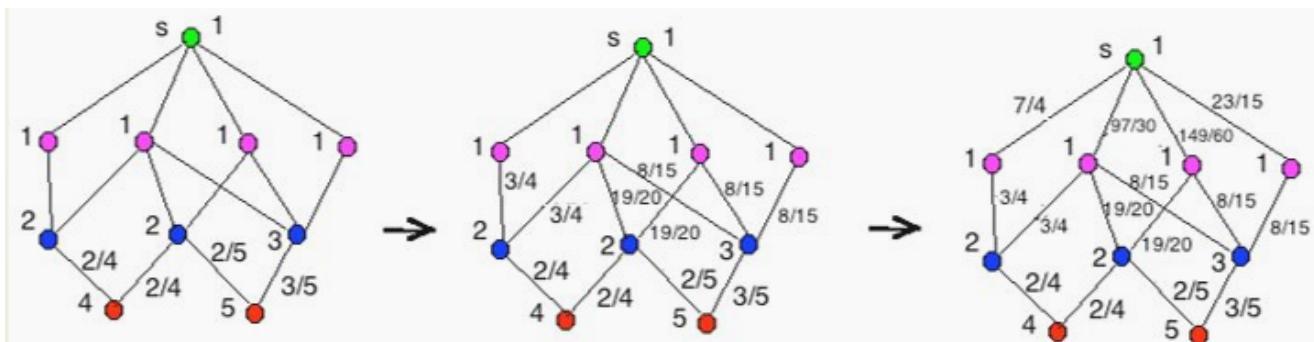
1. calcola $b_s(u, v)$ per ogni $(u, v) \in T(s)$ tale che $v \in L_h$:

$$b_s(u, v) = \frac{\sigma_{su}}{\sigma_{sv}} + \frac{\sigma_{su}}{\sigma_{sv}} \sum_{(v, x) \in T(s)} b_s(v, x)$$

Questa procedura è quindi una visita bottom-up, ossia dai livelli più in basso a salire.

Si fanno le due seguenti osservazioni:

1. $T(s)$ è un insieme di archi orientati: se $v \in L_h$ e $(v, x) \in T(s)$ allora $x \in L_{h+1}$
2. Si è assunto che $\sigma_{ss} = 1$.



Concludendo, si calcola la betweennes di tutti gli archi come già descritto in precedenza

$$b(u, v) = \frac{1}{2} \sum_{s \in V} b_s(u, v)$$

Notiamo che questa procedura permette di calcolare la betweennes degli archi in tempo *polinomiale* nella grandezza della rete (e non esponenziale).

Infatti la fase (1) è una semplice visita in ampiezza del grafo (e si può calcolare in tempo $O(|V| + |E|)$), la fase (2) è un'ulteriore visita in ampiezza di $T(s)$ (e si può calcolare ancora in tempo $O(|V| + |E|)$), e infine la fase (3) è nuovamente una visita in ampiezza, partendo però dal livello più basso (ancora una volta $O(|V| + |E|)$).

Dato che si itera il procedimento per ogni nodo del grafo, e dato che nel caso peggiore si trattano grafi molto densi con $\Theta(n^2)$ archi, la complessità temporale dell'esecuzione di questo algoritmo per il calcolo delle betweennes sarà $O(nm) \in O(n^3)$.

I valori di betweenness per gli archi di un grafo si comportano in qualche modo come una misura di distanza sui nodi del grafo. Non è esattamente una misura di distanza, perché non è definita per coppie di nodi non connessi da un arco e potrebbe non soddisfare la disuguaglianza triangolare, anche quando la misura di distanza è definita per una coppia di nodi. Tuttavia, si può raggruppare in comunità prendendo gli archi in ordine di betweenness crescente e aggiungerli al grafo uno alla volta. Ad ogni passo, i componenti connessi del grafo formano alcuni cluster. Più si permette un valore di betweenness elevato, più archi si ottengono e più grandi diventano i cluster. Più comunemente, questa idea è espressa come un processo di rimozione degli archi. Inizia con il grafo e tutti i suoi archi; poi rimuovi gli archi con il betweenness più alto, fino a quando il grafo non si è spezzato in un numero adeguato di componenti connesse.

Si è visto che il metodo descritto, applicato ad un grafo di n nodi ed m archi, impiega un tempo di esecuzione di $O(nm)$ per calcolare la betweenness di ogni arco. Se il grafo è molto grande, non ci si può permettere di eseguire tale processo nella maniera in cui è stato definito. Tuttavia, se si sceglie un sottoinsieme dei nodi a caso e si utilizzano questi come radici delle visite in ampiezza, si può ottenere un'approssimazione del betweenness di ogni arco che sarà utile nella maggior parte delle applicazioni.