Nozioni per il corso di Algoritmica per il Web

Versione 1.0

Modifiche di Alessandro Biagiotti

Materiale originale di Sebastiano Vigna

Indice

Le dispense di seguito presentate sono ad opera del professor Sebastiano Vigna (la revisione è ad opera di Alessandro Biagiotti), si tratta di un collage di tutta una serie di informazioni ricavate nell'arco del processo di preparazione all'esame, con il supporto di Alessandro Clerici. Ringrazio anche Davide Polidori, il quale mi ha prestato i suoi appunti, senza i quali alcuni passaggi sarebbero ancora oscuri.

L'intento delle dispense è quello di raccogliere le informazioni necessarie, fornendo una spiegazione per i passaggi meno chiari e aggiungendo parti inizialmente mancanti.

1 Notazione e definizioni di base

Il prodotto cartesiano degli insiemi X e Y è l'insieme $X \times Y = \{\langle x, y\rangle \mid x \in X \wedge y \in Y\}$ delle coppie ordinate degli elementi X e Y. La definizione si estende per ricorsione a n insiemi. Al prodotto cartesiano $X_1 \times X_2 \times \cdots \times X_n$ sono naturalmente associate le *proiezioni* $\pi_1, \pi_2, \ldots, \pi_n$ definite da

$$\pi_i(\langle x_1, x_2, \ldots, x_n\rangle) = x_i \tag{1}$$

poniamo

$$X^n = \overbrace{X \times X \times \cdots \times X}^{n \text{ volte}} \tag{2}$$

e $X^0 = \{*\}$ (qualunque insieme con un solo elemento). La *somma disgiunta* degli insiemi X e Y è, intuitivamente, un'unione di X e Y che però tiene separati gli elementi comuni, quindi evita i conflitti. Formalmente:

$$X + Y = X \times \{0\} \cup Y \times \{1\} \tag{3}$$

Di solito ometteremo, con un piccolo abuso di notazione, la seconda coordinata. Una *relazione* tra gli insiemi X_1, X_2, ..., X_n è un sottoinsieme R del prodotto cartesiano $X_0 \times X_1 \times \cdots \times X_n$. Se $n = 2$ si tende a scrivere $x\ R\ y$ per $\langle x, y\rangle \in \mathbb{R}$. Una relazione tra due insiemi è detta *binaria*. Se R è una relazione binaria tra X e Y, X è detto *dominio* di R, ed è denotato da $\mathrm{dom}(R)$, mentre Y è detto *codominio* di R, ed è denotato da $\mathrm{cod}(R)$. Il *rango* o *insieme di definizione* di R è l'insieme $\mathrm{ran}(R) = \{x \in X \mid \exists y \in Y, x\ R\ y\}$, e in generale può non coincidere con il dominio di R. L'*immagine* di R è l'insieme $\mathrm{imm}(R) = \{y \in Y \mid \exists x \in X, x\ R\ y\}$, e in generale può non coincidere con il codominio di R. Una relazione binaria R tra X e Y è *monodroma* se per ogni $x \in X$ esiste al più un $y \in Y$ tale che $x\ R\ y$. È *totale* se per ogni $x \in X$ esiste un $y \in Y$ tale che $x\ R$, cioè se $\mathrm{ran}(R) = \mathrm{dom}(R)$. È *iniettiva* se per ogni $y \in Y$ esiste al più un $x \in X$ tale che $x\ R\ y$. È *suriettiva* se per ogni $y \in Y$ esiste un $x \in X$ tale che $x\ R\ y$, cioè se $\mathrm{imm}(R) = \mathrm{cod}(R)$. È *biiettiva* se la relazione è sia iniettiva che suriettiva. Una *funzione* da X a Y è una relazione monodroma e totale tra X e Y (notate che l'ordine è rilevante[1]); in tal caso scriviamo $f : X \to Y$ per dire che f "va da X a Y". Se f è una funzione da X a Y è uso scrivere $f(x)$ per l'unico $y \in Y$ tale che $x\ f\ y$, diremo che f *mappa* x in $f(x)$ o, in simboli, $x \mapsto f(x)$. Le nozioni di dominio, codominio, iniettività, suriettività e biiettività vengono ereditate dalle relazioni. Se una funzione $f : X \to Y$ è biiettiva, è facile verificare che esiste una funzione inversa f^{-1}, che soddisfa le equazioni $f(f^{-1}(y)) = y$ e $f^{-1}(f(x)) = x$ per ogni $x \in X$ e $y \in Y$. Una *funzione parziale* (che tecnicamente non è una funzione perché non è definita sull'interezza del suo dominio) da X a Y è una relazione monodroma tra X e Y; una funzione parziale può non essere definita su elementi del suo dominio, fatto che denotiamo con la scrittura $f(x) = \perp$ ("$f(x)$ è indefinito" o "f è indefinita su x"), che significa che $x \notin \mathrm{ran}(f)$. Date funzioni parziali $f : X \to Y$ e $g : Y \to Z$, la *composizione* $g \circ f$ di f con g è la funzione definita da $(g \circ f)(x) = g(f(x))$. Si noti che, per convenzione, $f(\perp) = \perp$ per ogni funzione parziale f. Dati gli insiemi X e Y, denotiamo con $Y^X = \{f | f : X \to Y\}$ l'insieme delle funzioni da X a Y. Si noti che per insiemi finiti[2] $|Y^X| = |Y|^{|X|}$.

Denoteremo con n l'insieme $\{0, 1, \ldots, n-1\}$.

Dato un insieme X, il *monoide libero* su X, denotato da X^*, è l'insieme di tutte le sequenze finite, (inclusa quella vuota, normalmente denotata da ε) di elementi di X, dette *parole* su X, dotate dell'operazione di concatenazione, di cui la parola vuota è l'elemento neutro. Denoteremo con $|w|$ il numero di elementi di X della parola $w \in X$. Dato un sottoinsieme A di X, possiamo

[1] Secondo la mia interpretazione, una funzione è monodroma e totale perché una funzione è definita come una relazione in cui ogni elemento del dominio è mappato in uno e un solo elemento del codominio, dunque:

- monodroma garantisce che ogni elemento dell'insieme di definizione ha un'unica immagine.
- Totale garantisce che $\mathrm{dom}(R) = \mathrm{ran}(R)$.

[2] Si noti che l'uguaglianza è vera in generale, utilizzando i cardinali cantoriani

associargli la sua *funzione caratteristica* $\chi_A : X \to 2$ definita da:

$$\chi_A = \begin{cases} 0, & \text{se } x \notin A \\ 1, & \text{se } x \in A \end{cases} \tag{4}$$

Per contro, a ogni funzione $f : X \to 2$ possiamo associare il sottoinsieme di X dato dagli elementi mappati da f in 1, cioè l'insieme $\{x \in X \mid f(x) = 1\}$; tale corrispondenza è inversa alla precedente, ed è quindi naturalmente equivalente considerare sottoinsiemi di X o funzioni di X in 2. Date due funzioni f, $g : \mathbb{N} \to \mathbb{R}$, diremo che f è di *ordine non superiore* a g, e scriveremo che $f \in \mathcal{O}(g)$ ("f è $\mathcal{O}$-grande di g") se esiste una costante $a \in \mathbb{R}$ tale che $|f(n_0)| \leq |ag(n_0)|$ definitivamente. Diremo che f è di *ordine non inferiore* a g, e scriveremo che $f \in \Omega(g)$ se $g \in \mathcal{O}(f)$. Diremo che f è *dello stesso ordine di* g e scriveremo $f \in \Theta(g)$, se $f \in \mathcal{O}(g)$ e $g \in \Omega(f)$.

Un *grafo semplice* G è dato da un insieme finito di vertici V_G e da un insieme di lati $E_g \subseteq \{\{x, y\} \mid x, y \in V_G \wedge x \neq y\}$; ogni lato è cioè una coppia non ordinata di vertici distinti. Se $\{x, y\} \in E_G$, diremo che x e y sono vertici *adiacenti* in G. Un grafo può essere rappresentato graficamente disegnando i suoi vertici come punti sul piano, e rappresentato i lati come segmenti che congiungono vertici adiacenti. Per esempio, il grafo con insieme di vertici 4 e insieme di lati $\{\{0, 1\}, \{1, 2\}, \{2, 0\}, \{2, 3\}\}$ può essere rappresentato come segue: L'*ordine* di G è il numero naturale $|V_G|$. Una *cricca* o una *clique* di G è un insieme di vertici $C \subseteq V_G$ mutualmente adiacenti (nell'esempio in figura $\{0, 1, 2\}$ è una cricca). Dualmente, un *insieme indipendente* di G è un insieme di vertici $I \subseteq V_G$ mutualmente non adiacenti. Un *cammino* di lunghezza n in G è una sequenza di vertici $x_0, \ldots, x_n$ tale che x_i è adiacente a x_{i+1} con $(0 \leq i < n)$. Diremo che il cammino va da x_0 a x_n. Nell'esempio in figura, 0, 1, 2 è un cammino, 1, 3 non lo è.

Un grafo *orientato* G è dato da un insieme di nodi N_G e un insieme di archi A_G e da funzioni s_G, $t_G : A_G \to n_G$ (*source*, *target*) che specificano l'inizio e la fine di ogni arco. Due archi a e b tali che $s_G(a) = s_G(b)$ e $t_G(a) = t_G(b)$ sono detti *paralleli*. Un grafo senza archi paralleli è detto *separato*. Il *grado positivo* od *outdegree* $d^+(x)$ di un nodo x è il numero di archi uscenti da x, cioè $|s_G^{-1}(x)|$. Dualmente, il *grado negativo* o *indegree* $d^-(x)$ di un nodo x è il numero di archi entranti in x, cioè $|t_G^{-1}(x)|$. In un grafo orientato G un *cammino* di lunghezza n è una sequenza di vertici e archi $x_0, a_0, x_1, a_1, \ldots, a_{n-1}, x_n$ tale che $s_G(a_i) = x_i$ e $t_G(a_i) = x_{i+1}$ per $0 \leq i < n$. Diremo che il cammino va da x_0 a x_n. Definiamo la relazione di *raggiungibilità*: $x \rightsquigarrow y$ se esiste un cammino da x a y. La relazione di equivalenza $\sim$ è ora definita da $x \sim y \iff x \rightsquigarrow y \wedge y \rightsquigarrow x$. Le classi di equivalenza di $\sim$ sono dette *componenti fortemente connesse* di G, G è *fortemente connesso* quando è costituito da una sola componente.

La funzione $\lambda(x)$ denota il bit più significativo dell'espansione binaria di x: quindi $\lambda(1) = \lambda(1_2) = 0$, $\lambda(2) = \lambda(10_2) = 1$ etc…

Per convenzione $\lambda(0) = -1$. Si noti che per $x > 0$ si ha $\lambda(x) = \lfloor \log x \rfloor$.

2 Crawling

Il *crawling* è l'attività di scaricamento delle pagine web. Un *crawler* è un dispositivo software che visita, scarica e analizza i contenuti delle pagine web a partire da un insieme di URL dato, detto *seme*. Il crawler esegue una visita seguendo i collegamenti ipertestuali contenuti nelle pagine analizzate.

Le pagine web durante il processo di crawl si dividono in tre:

- L'insieme delle pagine *visitate*, V, che sono già state scaricate e analizzate;
- La *frontiera*, F, che è l'insieme delle pagine conosciute ma che non sono ancora state visitate;
- L'insieme U degli URL sconosciuti.

Le differenze tra l'attività di crawling e una banale visita all'interno di un grafo sono molto importanti, prima di tutto c'è il fatto che un crawl ha una dimensione ignota, non conosciamo $|V_G|$; secondariamente la frontiera è un enorme problema, in quanto la sua dimensione tende a crescere molto più velocemente dell'insieme dei visitati.

In generale l'operazione di crawling parte caricando il seme in frontiera e, finchè la frontiera non è vuota, viene estratto un URL dalla frontiera, secondo determinate politiche, l'URL viene visitato (e quindi scarica la pagina corrispondente), lo analizza derivandone nuovi URL tramite i collegamenti ipertestuali contenuti all'interno della pagina e sposta l'URL nell'insieme dei visitati. I nuovi URL vengono invece aggiunti alla frontiera se sono sconosciuti, e quindi non sono in $V \cup F$.

Diverse politiche di prioritizzazione della frontiera possono poi dare luogo ad approcci diversi al processo di crawling, posso per esempio estrarre prima degli URL a cui si arriva partendo da pagine che contengono determinate parole chiave.

2.1 Il crivello

Il crivello è la struttura dati di base di un crawler, questo accetta in ingresso URL potenzialmente da visitare e permette di prelevare URL pronti alla visita. Ogni URL viene estratto una e una sola volta in tutto il processo di crawling, indipendentemente da quante volte è stato inserito all'interno della struttura. In questo senso il crivello unisce le proprietà di un dizionario a quelle di una coda con priorità e rappresenta al tempo stesso la frontiera, l'insieme dei visitati e la coda di visita. Combinare questi aspetti in una sola struttura è un lavoro complesso ma permette risparmi notevoli dal punto di vista pratico[3].

Una prima osservazione è che spesso, per mantenere l'informazione di quali URL sono stati già visitati ($V \cup F$) è preferibile sostituire gli URL con delle *firme*, cioè con il risultato del calcolo di $h(u)$, dove h è una funzione di hash definita sulle stringhe e restituisce una stringa di bit di dimensione arbitraria, per esempio 64 bit. Questo ha due grandi benefici:

- Molti URL possono essere di grandi dimensioni, mantenerli salvati in memoria con un numero di bit scelto a priori (per esempio 64 bit), indipendentemente dalla loro reale dimensione, si rivela un risparmio di spazio affatto trascurabile.
- Uniformiamo le lunghezze degli URL a un valore standard.

Il drawback di una soluzione del genere è che accettiamo il fatto che vi siano delle collisioni, è dunque possibile che due URL diversi vengano mappati sullo stesso valore di hash. Questo fenomeno è inevitabile, però se abbiamo una funzione di hash che lavora su un numero di bit abbastanza grande, la probabilità d'incontrare una collisione sarà così bassa da essere trascurabile.

Github.com implementa una soluzione del genere, viene impiegato SHA-1 (funzione di hash a 160bit) per calcolare un hash dell'URL di ciascuna delle repository nei loro database, la probabilità di collisione è così bassa che è sostanzialmente impossibile. Adesso sembra che vogliano muoversi verso SHA-256.

[3] Si noti che è possibile riordinare ulteriormente gli URL *dopo* l'uscita dal crivello

Supponiamo ora di avere in memoria n firme, la probabilità che una nuova firma collida con una di quelle esistenti è n/u, dove u è la dimensione dell'universo delle possibili firme. Nel caso di un sistema a 64 bit $u = 2^{64}$, e quindi possiamo memorizzare 100 miliardi di URL con una probabilità di falsi positivi nell'ordine di $10^{11}/2^{64} < 2^{37}/2^{64} = 1/2^{27} < 1/10^8$, quindi avremo meno di un errore ogni 100 milioni di URL.

Anche un semplice dizionario di firme in memoria che rappresenta $V \cup F$, accoppiato a una coda o pila su disco che tiene traccia di F, è sufficiente per un'attività di crawling di piccole dimensioni. Per dimensioni più grandi è necessario ingegnarsi e impiegare delle strutture dati, in parte, o completamente su disco, che consentano di mantenere l'occupazione totale di memoria centrale costante.

Questo è un compromesso tipico delle attività di crawling - strutture che si espandono in memoria centrale proporzionalmente alla frontiera sono troppo fragili e quindi rischiano di mandare in crash il processo o di sovraccaricare il sistema di gestione della memoria virtuale.

Solitamente le strutture dati impiegate in ambiti di crawling importanti dovrebbero entrare in un processo di *degrado grazioso*, riducendo le performance, ma senza interrompere all'improvviso il funzionamento del programma.

2.2 I filtri di Bloom

La prima struttura che vedremo con queste proprietà è il *filtro di Bloom*. Un filtro di Bloom [Blo70] è una semplicissima struttura dati probabilistica a falsi positivi che rappresenta un dizionario, cioè un insieme di elementi da un universo dato. Permette di aggiungere elementi all'insieme e chiedere se un elemento è presente o meno nell'insieme.

Un filtro di Bloom di universo X è rappresentato da un vettore $\mathbf{b}$ di m bit e da d funzioni di hash $f_0, \ldots, f_d$ da X in m. Per aggiungere un elemento al filtro è necessario calcolare i valori per tutte le d funzioni di hash e mettere a 1 il bit $d_{f_i(x)}$ con $0 \leq i < d$. Per sapere se un elemento è presente all'interno del filtro è comunque necessario calcolare tutte le d funzioni di hash, qualora esista $i \in [0, d) \mid d_{f_i(x)} = 0$ allora il valore non è presente nella struttura, altrimenti la risposta è positiva.

Intuitivamente, ogni volta che un elemento viene aggiunto al filtro la conoscenza della presenza dell'elemento viene sparsa in d bit a caso, che vengono interrogati quando è necessario sapere se quell'elemento è stato memorizzato: è però possibile che i d bit siano stati messi a 1 a seguito di una serie d'inserimenti precedenti, quindi la risposta a un'interrogazione per un elemento che non è presente nell'insieme risulta essere ugualmente positiva. Questo implica che a causa di collisioni sulle varie funzioni di hash si possono avere *falsi positivi*, questo significa che il filtro risponde positivamente, sebbene l'elemento non sia presente in struttura.

I filtri di Bloom sono chiamati in questa maniera perché sono molto utili come filtri per strutture dati più lente che stanno su disco. Se si prevede che la maggior parte delle richieste avrà risposta negativa, un filtro di Bloom può ridurre significativamente gli accessi alla struttura soggiacente; oltre a questo il filtro tende a rispondere molto velocemente a richieste che hanno risposta negativa, basta infatti che una sola delle posizioni indicate dalle funzioni di hash abbia bit a 0 per rispondere falso, mentre è necessario controllare tutte le posizioni e accedere alla struttura dati soggiacente nel caso in cui la risposta sia positiva.

Di fatto, i filtri di Bloom sono risultati estremamente pratici per mantenere insiemi di grandi dimensioni in memoria, in particolare quando le dimensioni delle chiavi sono significative (e.g. degli URL).

Andiamo ora a vedere qual è la probabilità di un falso positivo. Con un'analisi ragionevolmente precisa (quella che presenta Bloom in [Blo70]) saremo in grado di fornire valori ottimi di m e d data la probabilità di falsi positivi desiderata e il massimo numero di elementi memorizzabili nel filtro. In questo modo saremo in grado di scegliere la struttura dati meno ingombrante per ottenere una probabilità di falsi positivi scelta a piacere.

2.2.1 Dimostrazione dell'efficacia dei filtri di Bloom

Per calcolare l'efficacia dei filtri di Bloom, come detto in precedenza, si considererà la probabilità di osservare un falso positivo. Una prima semplificazione lecita (seguendo l'analisi di [Blo70]) è quella di andare a calcolare la probabilità di un positivo qualunque dopo n inserimenti, che è ovviamente una maggiorazione del caso dei falsi positivi. Supponiamo di avere un vettore di m posizioni e d funzioni di hash uniformemente distribuite e indipendenti. Dopo l'n-esimo inserimento, la probabilità che un bit sia 0 è data da:

$$P[\mathbf{b}[i] = 0 \mid N = n] = 1 - P[\mathbf{b}[i] = 1 \mid N = n] = \left(1 - \frac{1}{m}\right)^{dn}$$

Si ottiene un positivo quando tutte le posizioni controllate sono a 1, ciò avviene con probabilità

$$\varphi = \left(1 - \left(1 - \frac{1}{m}\right)^{dn}\right)^{d}$$

siccome $(1 + \alpha/n)^n \to e^\alpha$ per $n \to \infty$

$$\varphi = \left(1 - \left(1 - \frac{1}{m}\right)^{-m\frac{-dn}{m}}\right)^{d} \sim \left(1 - e^{-\frac{dn}{m}}\right)^{d}$$

Sia ora $p = e^{-\frac{dn}{m}}$, allora

$$\ln(p) = -\frac{dn}{m}$$
$$m\ln(p) = dn$$
$$d = -\frac{m\ln(p)}{n}$$

Voglio ora minimizzare la probabilità di ottenere un positivo φ, quindi prendo $\left(1 - e^{-\frac{dn}{m}}\right)^{d}$ e sostituisco p, quindi $(1-p)^{m/n\cdot\ln(p)}$.
Riscrivo come esponenziale:

$$e^{\ln(1-p)^{m/n\cdot\ln(p)}} = e^{m/n\cdot\ln(1-p)\ln(p)}$$

Faccio la derivata rispetto a p che viene:

$$-\frac{m}{n}\cdot\left(\frac{1}{p}\ln(1-p) - \frac{1}{1-p}\cdot\ln(p)\right)\cdot e^{m/n\cdot\ln(1-p)\ln(p)}$$

Il punto stazionario è quello per cui la parte tra parentesi si annulla, l'esponenziale è sempre > 0 quindi:

$$\frac{1}{p}\ln(1-p) - \frac{1}{1-p}\cdot\ln(p) = 0$$
$$(1-p)\cdot\ln(1-p) = p\cdot\ln(p)$$

Se $1 - p = p$ allora $p = 1/2$, questa è l'unica soluzione, come prova del 9 si studi $g(p) = p\ln(p) - (1-p)\ln(1-p)$ e risulterà che agli estremi la funzione vale 0 dato che:

$$\lim_{p\to 0} p\ln(p) = \lim_{p\to 0}\frac{\ln(p)}{1/p} = \lim_{p\to 0}\frac{1/p}{-1/p^2} = \lim_{p\to 0} -p = 0$$

la derivata invece è $\dot{g(p)} = \ln(1-p) + \ln(p) + 2$.

Chiaramente va a meno infinito in 0 e 1, ma in $p = 1/2$ è positiva, ed è l'unico punto di massimo (dato che la derivata seconda ha un solo zero in $p = 1/2$). Concludiamo che $g(p)$ ha esattamente un massimo e un minimo in $[0, 1)$, e quindi esattamente uno zero in $(0, 1)$. Per concludere, se $p = 1/2$ allora $d = -\frac{m \ln(p)}{n} = \frac{m \ln 2}{n}$ per quanto riguarda la probabilità di avere uno in tutti i punti che andiamo a controllare all'interno del vettore:

$$\varphi = \left(1 - e^{-\frac{dn}{m}}\right)^d = \left(1 - e^{-\ln 2}\right)^d = \left(1 - \frac{1}{2}\right)^d = 2^{-d}$$

Alla fine, la probabilità di (falsi) positivi è minimizzata da $d \approx \frac{m \ln 2}{n}$, e in tal caso la probabilità di un (falso) positivo è 2^{-d}. Vale a dire che aumentando linearmente il numero delle funzioni di hash impiegate e il numero di bit della struttura a $m \approx dn/\ln 2 \approx 1.44dn$, si ha una riduzione esponenziale del numero di (falsi) positivi che possono essere incontrati. Passiamo ora a fare alcune osservazioni tecniche:

- È abbastanza intuitivo, ed è possibile dimostrare, che per avere falsi positivi con probabililtà 2^{-d} occorre utilizzare almeno d bit per elemento. Quindi un filtro di Bloom perde 44% in spazio rispetto al minimo possibile.
- In linea di principio il filtro di Bloom ha una modalità di accesso alla memoria pessima, a causa dei d accessi casuali al vettore, che possono causare d fallimenti in cache.
- Ciononostante, il dimensionamento ottimo di un filtro di Bloom è lineare nel numero di chiavi attese. Questo fa sì che se dividiamo le chiavi in k segmenti utilizzando una funzione di hash e costruiamo un filtro per segmento, l'occupazione in spazio non aumenta.

  Dimensionando k in modo che i segmenti abbiano la dimensione di una o due linee di cache si può abbattere il numero di cache failures (questa implementazione è detta *block*). In questo caso però l'approssimazione che abbiamo utilizzato perde di precisione; inoltre, la divisione delle chiavi in segmenti non è mai uniforme ma ha una distribuzione binomiale negativa. Questi fattori peggiorano la probabilità di errore [Sin10].
- Dall'analisi che abbiamo effettuato, 1/2 è anche la probabililtà di un bit a 0, quindi, come si diceva all'inizio del paragrafo, un filtro di Bloom è molto più efficace nel riportare il fatto che un elemento non è presente nel filtro piuttosto che a riportarne la presenza.
- In teoria per utilizzare un filtro di Bloom dobbiamo calcolare d funzioni di hash diverse, il che può essere molto costoso in termini di tempo. In realtà Kirsch e Mitzenmacher hanno dimostrato che estraendo due numeri interi a 64 bit a e b tramite una funzione di hash, i numeri $ai + b$, $0 \leq i < d$ sono d hash sufficienti a replicare l'analisi condotta utilizzando funzioni indipendenti e pienamente casuali.
- Se un filtro di Bloom viene utilizzato per rappresentare gli URL già visti, soddisfa pienamente le nostre richieste: utilizza una quantità di memoria centrale costante, è relativamente veloce e affidabile e degrada graziosamente, man mano che il vettore si riempie la probabilità di falsi positivi aumenterà fino a diventare 1.

2.3 Crivelli basati su database NoSQL

Un modo più sofisticato e con degrado più grazioso di implementare un crivello è quello di utilizzare un cosiddetto *database NoSQL*, che consiste semplicemente in una struttura parzialmente su disco che permette di memorizzare coppie chiave/valore utilizzando una quantità limitata di memoria centrale.

Uno degli esempi classici di database NoSQL è il BerkeleyDB, che permette di memorizzare coppie/chiave valore in maniera non ordinata od ordinata tramite una hash table e un B-tree parzialmente su disco. La memoria centrale è utilizzata come cache per accelerare le operazioni su disco.

Un approccio più sistematico, implementato inizialmente a Google sotto il nome di BigTable, è l'LSM tree [O'N96]. BigTable è stato successivamente reimplementato come progetto open-source, LevelDB, che è poi stato utilizzato come base per altri database NoSQL come RocksDB, l'implementazione di Facebook, che è utilizzata da commoncrawler, un crawler open-source.

Gli LSM tree sono basati su un concetto relativamente semplice, ma necessitano di un'implementazione accurata che sfrutti parallelismo e concorrenza per essere efficienti.

Un LSM-tree è diviso in vari livelli, ognuno dei quali contiene un sottoinsieme delle coppie chiave/valore che si intende rappresentare. Ogni livello ha una dimensione di base che cresce di un fattore dato rispetto al livello precedente, ma ha una certa elasticità nel dimensionamento (può essere, ad esempio, grande il doppio rispetto alla sua dimensione di base).

Il primo livello è sempre in memoria centrale e ha dimensione limitata a priori, solitamente è implementato tramite un normale dizionario ordinato (RB-tree o B-tree).

I livelli successivi sono memorizzati sotto forma di *log* e sono una successione immutabile di coppie chiave/valore ordinate. Il primo aspetto importante di un LSM-tree è che una chiave può comparire in più livelli, il valore associato è quello che compare nel livello più alto in cui è possibile trovare la chiave. Un'interrogazione in lettura consiste quindi in una ricerca della chiave a partire dal primo livello, non appena la chiave viene trovata si sa il suo valore.

La parte interessante è quella di scrittura: la coppia chiave/valore viene inizialmente inserita nel primo livello. Se a questo punto il primo livello eccede la sua dimensione massima, si esegue l'operazione di *scarico* in cui un gruppo di chiavi viene estratto dal primo livello e aggiunto al secondo, in modo da riportare il primo livello alla sua dimensione naturale.

A questo punto l'operazione di scarico continua ricorsivamente verso il basso fino a quando, se accade, anche l'ultimo livello eccede la propria dimensione massima, ed esegue un'operazione di scarico su un nuovo livello dell'albero.

Si noti che tutte le operazioni su disco avvengono sequenzialmente, e che tutti i dati memorizzati su disco sono *immutabili*. Queste due caratteristiche rendono le fusioni estremamente efficienti nelle architetture moderne, e semplificano notevolmente la gestione degli accessi paralleli.

Per cancellare un'associazione chiave valore viene inserita una coppia con la stessa chiave e un valore arbitrario noto come *lapide*; la tecnica è simile a quella utilizzata per le tabelle di hash. La lapide viene trattata come ogni altro valore, ma in fase di ricerca, se ne viene trovata una associata alla chiave il processo si ferma e il valore viene considerato assente dall'albero. Se una lapide arriva all'ultimo livello, viene scartata.

Ci sono a questo punto numerose e importanti questioni ingegneristiche e implementative da considerare:

- Il formato in cui vengono memorizzati i livelli può non essere uniforme, dipende dal tipo di supporti di memorizzazione utilizzati; salvare dati efficacemente ed efficientemente su nastri richiede tecniche diverse rispetto a un processo di salvataggio su disco magnetico.

- Ogni livello può essere frammentato in file più piccoli per permettere di selezionare più liberamente le chiavi da fondere, e per rendere più semplice operare le fusioni in parallelo. In questo caso ogni chiave compare in un solo frammento.

- Ogni frammento può essere arricchito con un dizionario approssimato a falsi positivi a bassa precisione (come un filtro di Bloom) che evita l'accesso al file nel caso in cui la chiave che stiamo cercando non sia all'interno del file. La bassa precisione fa sì che l'occupazione in memoria del filtro non sia particolarmente rilevante.

- Ogni frammento può contenere un indice sparso che memorizza le posizioni di un sottoinsieme di chiavi campionate a intervalli regolari; in questo modo, in fase di ricerca è possibile identificare rapidamente la zona del frammento che potenzialmente contiene la chiave, per poi procedere con una ricerca binaria o lineare. La scelta della frequenza di campionamento consente di bilanciare lo spazio occupato dalla struttura e l'efficacia del processo di ricerca.

- Anche in assenza di fusioni di livelli, in generale in un LSM-tree vengono lasciati in esecuzione dei thread che si occupano di fare il *compattamento* della struttura:

  - Controllano che il numero di copie per chiave non sia eccessivo
  - Rimuovono eventuali lapidi in eccesso
- Infine, tutte le operazioni di fusione non vengono effettuate veramente durante gli inserimenti, ma piuttosto vengono svolte con continuità da processi concorrenti.

Ci sono anche soluzioni ibride, che reinseriscono parzialmente gli alberi bilanciati negli LSM tree. Questo tipo di tecnologia è in continua evoluzione, anche perché diverse implementazioni o politiche di aggiornamento possono essere adatte a diversi carichi di lavoro.

Si noti che al crescere della frontiera l'LSM-tree continua a occupare la stessa quantità di memoria centrale e non ha decrementi di precisione, però il crivello rallenta e occupa più memoria di massa.

2.4 Un crivello offline

Un modo meno *responsive* ma molto più semplice di implementare il crivello che effettua una visita in ampiezza e richiede memoria centrale costante senza utilizzare strutture dati, consiste nel tenere traccia, in ogni istante, di tre file:

- Un file Z di URL già visitati ($V \cup F$), in ordine lessicografico.
- Un file F di URL ancora dia visitare, quindi la frontiera, in ordine di scoperta.
- Un file A, di lunghezza limitata a priori, che accumula temporaneamente gli URL incontrati durante la visita.

All'inizio dell'attività di crawl, Z e F sono inizializzati utilizzando il seme, e A è vuoto. Durante il crawl, gli URL da visitare vengono estratti da F (eventualmente alterandone l'ordine secondo qualche politica), e i nuovi URL che vengono incontrati vengono accumulati in A.

Quando F è vuoto o quando A raggiunge la dimensione massima si procede a eseguire l'operazione di *fusione*:

- A viene ordinato (lessicograficamente) e deduplicato, il risultato è A'.
- Z e A' vengono fusi per ottenere un file Z' che andrà a rimpiazzare Z.
- Durante la fusione, gli URL che sono in A' ma non Z vengono accodati a F.

La fusione di Z e A' può procedere in maniera sequenziale perché i due file sono ordinati. È evidente che ogni URL che viene incontrato dal crawler viene accodato a F esattamente una volta, e cioè durante la fusione che avviene dopo la prima volta che compare in A. Questo tipo di organizzazione non è particolarmente performante se viene effettuata sulla macchina che sta eseguendo l'attività di crawling, sebbene l'ordinamento di A si possa effettuare in memoria costante. Se però è possibile ordinare e fondere file utilizzando un framework di ordinamento distribuito, come MapReduce [Ghe08], o la sua implementazione open-source, Hadoop, le prestazioni possono essere molto migliorate, e la semplicità del codice può giocare a favore di questa scelta.

Va notato che l'ordinamento effettuato su A altera l'ordine di accodamento. Per recuperare l'effetto di una visita in ampiezza è necessario recuperare l'ordine originale degli URL. Questo risultato si può ottenere, per esempio, memorizzando in A, oltre agli URL, la posizione ordinale della loro prima occorrenza, e riordinando i nuovi URL scoperti in tale ordine prima di accodarli a F. È anche possibile tenere A quando si crea A', e durante la fusione mantenere invece di una lista di URL scoperti una lista di *posizioni* in A di URL scoperti. A quel punto è sufficiente ordinare la lista di posizioni e scandirla in parallelo con A per estrarre sequenzialmente e nell'ordine di accodamento in A gli URL scoperti.

Infine, da un punto di vista pratico è conveniente mantenere in Z non gli URL già visti, ma le loro firme. Per fare funzionare correttamente il passo di fusione è però a questo punto necessario ordinare e deduplicare A utilizzando come chiavi le firme degli URL.

Si noti che al crescere della frontiera il crivello offline diventa più lento e occupa più memoria di massa, ma l'utilizzo di memoria centrale resta costante e non si hanno decrementi di precisione.

2.5 Il crivello di Mercator

Mercator è un crawler [Naj99] il cui crivello è una versione parzialmente in memoria del crivello offline descritto in precedenza. Le firme degli elementi in A vengono mantenute in un vettore in memoria, evitando di eseguire ordinamenti su disco.

Il crivello è formato da un vettore **S**, di dimensione fissata n, in memoria centrale che contiene firme di URL, inizialmente vuoto, ed è riempito incrementalmente. Su disco, invece, teniamo un file Z che contiene tutte le firme degli URL sinora mai incontrati e un file ausiliario A, inizialmente vuoto.

Ogni volta che un URL u viene inserito nel crivello, aggiungiamo $h(u)$ a **S** e u al file A. Il punto chiave è che cosa succede quando **S** raggiunge la massima dimensione; operiamo allora uno *scarico* nel seguente modo:

1. Ordino **S** indirettamente, cioè creo il vettore **V** contenente gli indici i associati ai valori **S**[i], ordino stabilmente **V** utilizzando come chiave le firme in **S**. Al termine del processo, le firme **S**[**V**[i]] sono in ordine crescente al crescere di i[4].

2. Deduplico **S**, quindi elimino le occorrenze successive alla prima per ogni firma.

3. Z' = Z $\cup$ **S** marchiamo utili le firme in **S** che non compaiono in Z.

4. Scandiamo ogni entry di **S** e A in parallelo (sono entrambi ordinati) e diamo in output gli URL in A corrispondenti alle entry marcate come utili al passo precedente.

5. **S** e A vengono svuotati e Z viene sostituito da Z'.

Innanzitutto, si noti che Z, alla fine di uno scarico, contiene di nuovo le firme di tutti gli URL mai incontrati. Inoltre in output abbiamo dato tutti e soli gli URL la cui firma non era parte di Z, dunque si trattava di URL che non erano ancora stati visitati. Infine, gli URL in output sono stati ovviamente emessi nell'ordine di accodamento in A.

2.6 Gestione dei quasi duplicati

Durante l'attività di crawling è comune trovare pagine che sono quasi identiche (varianti dello stesso sito, calendari, immagini, etc…). In base al tipo di crawling (e quindi in base alle sue politiche), queste pagine andrebbero considerate duplicate e non ulteriormente elaborate.

Un modo semplice ma efficace di gestire il problema in memoria centrale è quello di analizzare una forma normalizzata del documento, rimuovendo marcatura, date e altri dati che sono standard ma che potrebbero risultare differenti sulla base di meccanismi automatici. Il documento dovrebbe poi essere memorizzato in un filtro di Bloom.

Metodi molto più sofisticati per la rilevazione dei duplicati possono essere utilizzati offline prima del processo di indicizzazione.

Un metodo efficace di gestione online del problema, che è stato utilizzato per qualche tempo dal crawler di Google [Sar07], è quello di porre in un dizionario (eventualmente approssimato) uno hash generato dall'algoritmo di SimHash di Charikar [Cha02]. L'algoritmo genera hash che sono simili (nel senso che hanno distanza di Hamming bassa) per pagine simili. In particolare, si può usare l'identità di SimHash come definizione di quasi-duplicato.

Per calcolare SimHash dobbiamo prima di tutto stabilire il numero b di bit dello hash, e fissare una buona funzione di hash h che mappa stringhe in hash di b bit. A un maggiore numero di bit corrisponderà una nozione più accurata di somiglianza. A questo punto il testo della pagina, in forma normalizzata, viene trasformato in un insieme di segnali S: un modo banale è utilizzare le parole del testo come segnali, ma è più accurato considerare i cosiddetti *shingles* (segmenti di testo di 3-5 caratteri).

[4] Si supponga per esempio che **S**={C, F, B, A, A, E, D}, allora il vettore associato a **S** sarà banalmente **V**={0, 1, 2, 3, 4, 5, 6}. Eseguendo l'ordinamento di **V** con le firme in **S** come chiave darà il seguente risultato (provare per credere) **V**={3, 4, 2, 0, 6, 5, 1}, che risulta effettivamente essere un ordinamento *stabile* delle posizioni associate alle firme in **S**.

A ogni segnale $s \in S$ associamo ora uno hash $h(s)$. Il SimHash del testo ha il bit i ($0 \leq i < b$) impostato a 1 se e solo se:

$$|\{s \in S \mid h(s)[i] = 1\}| > |\{s \in S \mid h(s)[i] = 0\}$$

Due documenti che hanno lo stesso SimHash sono molto simili, e la somiglianza diventa sempre meno significativa se si permettono distanze di Hamming superiori. Si noti che è banale *pesare* i segnali in modo che alcuni siano più importanti di altri.

Trovare elementi a breve distanza di Hamming è un problema interessante una cui soluzione pratica per distanze piccole è descritta è [Pag98].

2.7 Gestione della politeness

Un altro dei problemi pratici che rende l'attività di crawling diversa da una semplice visita è la gestione della *politeness*: non si dovrebbe eccedere nella quantità di tempo dedicato allo scaricamento da un singolo sito (pena, in genere, email furiose o taglio del traffico dal vostro IP).

Ci sono due modi fondamentali di operare questa limitazione:

- Limitare il tempo tra una richiesta e l'altra.
- Limitare il rapporto tra il tempo di scaricamento e quello di non scaricamento.

Nel primo caso, dato un intervallo di tempo t, diciamo, quattro secondi, siamo costretti ad aspettare t tra la fine di una richiesta e l'inizio della successiva per lo stesso sito. Nel secondo caso, data una frazione p e un tempo di scaricamento massimo s (diciamo, di un secondo) dobbiamo fare in modo che la proporzione tra il tempo di scaricamento e quello di non-scaricamento sia p. Si noti che questa condizione contempla anche una misurazione effettiva del tempo di scaricamento, dato che risorse particolarmente lente potrebbero richiedere un tempo maggiore di s.

La seconda soluzione è più interessante, perché permette di sfruttare una caratteristica della versione 1.1 del protocollo HTTP: è possibile cioè effettuare richieste multiple attraverso la stessa connessione TCP, evitando la (lenta) apertura e chiusura di una connessione per ogni risorsa scaricata. Le attività di scaricamento terminano non appena si supera la soglia s, con tempo di scaricamento effettivo s', e a questo punto si aspetta per tempo s'/p, in maniera da forzare la gentilezza. Per implementare questo tipo di politica, però è necessario alterare l'ordine di visita, dato che visitando gli URL nell'ordine in cui escono dal crivello si potrebbe incorrere in attese a vuoto consistenti.

2.8 La coda degli host

Un altro problema finora lasciato in parte è il ruolo della concorrenza. Certamente vorremo scaricare contemporaneamente da più siti: per farlo, possiamo istanziare molti flussi (sotto forma di migliaia di *visiting thread*) di esecuzione che si occupano di scaricare i dati, e saranno quindi sempre occupati in attività di I/O. Le pagine scaricate possono essere poi analizzate da un gruppo di flussi più ridotto (i *parsing thread*). Si noti che, al di là delle questioni di politeness, non possiamo permetterci che due flussi accedano allo stesso sito.

Questi problemi vengono risolti riorganizzando gli URL che escono dal crivello. Consideriamo una *coda con priorità* contenente i siti noti al crawler. A ogni sito assegnamo come priorità il primo istante di tempo in cui sarà possibile scaricare dal sito senza violare la politeness. Si tratta di una coda di min-priorità, dunque in cima alla coda c'è il minimo.

Per ogni sito manteniamo una coda (FIFO nel caso della visita in ampiezza). Quando degli URL vengono emessi dal crivello, vengono accodati alla coda del sito cui appartengono.

Ogni flusso del crawler procede iterativamente nel seguente modo:

1. Estrae il sito in cima alla coda (eventualmente aspettando il tempo necessario a far sì che questo sia scaricabile).
2. Procede a scaricare una o più risorse.

3. Riaccoda il sito aggiustando il timestamp di "readiness" secondo la politica di gestione della politeness.

Se c'è un URL disponibile allo scaricamento, il sito deve essere già stato reso pronto per lo scaricamento prima del tempo corrente. Quindi o è in cima alla coda, o in cima alla coda c'è un sito che era pronto per lo scaricamento ancora prima. Il punto è che la cima della coda è sempre scaricabile.

Questo meccanismo rende automatica l'esclusività del download tra flussi: gli elementi della coda agiscono come *token*, quando un elemento è in cima alla coda, il thread ha in possesso un token per scaricare da quel sito fino allo scadere del tempo.

Il costo della coda è logaritmico e l'aggiunta e la rimozione sono operazioni relativamente costose (al più polilogaritmiche) ma possono diventare problematiche in momenti di concorrenza intensa.

Infine, nel caso sia necessaria una politica di gentilezza da applicare agli indirizzi IP possiamo organizzare gli IP in una lista come quella descritta sopra. Rimanendo lungo il solco tracciato dalla metafora del token: quando un thread trova in cima alla lista un IP e un URL significa che il thread è in possesso del token per scaricare i dati associati da quell'URL e quello specifico indirizzo IP per una quantità limitata di tempo.

2.9 Tecniche di programmazione concorrente lock-free

Nella gestione delle strutture dati che abbiamo discusso è possibile che l'elevata concorrenza renda le strutture stesse poco efficienti. Se il numero di host è molto grande, per esempio, i flussi di scrittura e lettura potrebbero rimanere fermi per molto tempo ai punti di sincronizzazione o sui semafori che proteggono le zone di mutua esclusione.

è possibile alleviare il conflitto tra i flussi utilizzando delle tecniche di programmazione non bloccante, dette *lock-free*. Una struttura dati lock-free non utilizza semafori: utilizza invece istruzioni hardware che permettono implementazioni efficienti. Quella che useremo è CAS (*compare-and-swap*): dato un indirizzo p e due valori a e b, la CAS controlla che il valore memorizzato in p sia a e in questo caso lo sostituisce atomicamente con b, restituendo vero se la sostituzione è avvenuta.

Per dare un'idea delle tecniche lock-free, l'algoritmo 1 mostra come aggiungere concorrentemente un nodo a una lista con puntatori: l'implementazione è dovuta a Harris [Har01].

La correttezza dell'algoritmo è banale, ma si noti anche che se l'istruzione CAS fallisce, un altro flusso ha eseguito l'inserimento: quindi, per ogni ciclo eseguito da uno dei flussi concorrenti c'è stato *progresso* nell'aggiornamento della struttura. Detto altrimenti: non è possibile dire quante volte uno specifico flusso debba eseguire il ciclo prima di avere successo nell'inserimento, ma a ogni fallimento corrisponde un successo di un altro flusso. L'algoritmo per la cancellazione è più

Algorithm 1 Algoritmo lock-free per aggiungere un nodo (n) a una lista (dopo p)

```
do
    t ← p.next
    n.next ← t
while !CAS(&p.next, t, n)
```

complesso: l'idea di utilizzare la stessa tecnica non funziona perché è possibile cancellare un nodo mentre sta venendo inserito un successore, cancellando così l'effetto dell'inserimento.

Si noti che non è richiesta nessuna sincronizzazione in lettura: la lista non è mai in uno stato incoerente, per cui può essere scandita tramite lo stesso algoritmo utilizzato per una normale lista collegata. Inoltre, in pratica, per evitare che in condizioni di elevata concorrenza il numero di volte che l'operazione CAS fallisce divenga troppo importante, si può utilizzare una tecnica di *exponential backoff* che consiste nel mettere in attesa thread che falliscono per un tempo che ha una crescita esponenziale.

Molti linguaggi moderni offrono strutture lock-free built-in, come la `ConcurrentLinkedQueue` di Java, che implementa l'algoritmo di Michael e Scott [eMLS96].

3 Tecniche di distribuzione del carico

Per coordinare un insieme A di agenti indipendenti che effettuano attività di crawling è necessario assegnare in qualche modo ciascun URL a uno specifico agente: denoteremo con $\delta_A(-) : U \to A$ la funzione che, dati l'insieme dei agenti A e l'universo degli URL U, assegna a ciascun URL l'agente che ne è responsabile. Assumiamo qui che gli agenti in A siano identici, in particolare che abbiano a disposizione le stesse risorse.

Una richiesta banale ma necessaria è che la funzione sia *bilanciata*, ovvero che:

$$|\delta_A(-)^{-1}| \approx \frac{|U|}{|A|}$$

Quindi ogni agente deve gestire, all'incirca, lo stesso numero di URL.

Nel caso in cui l'insieme degli agenti sia statico, basta ad esempio numerare gli agenti a partire da zero, fissare una funzione di hash h applicabile agli URL, e dato un URL u assegnargli l'agente $h(u) \mod |A|$.

La questione è più interessante quando l'insieme degli agenti varia nel tempo. In questo frangente non vogliamo solo che la funzione di assegnamento sia bilanciata, deve anche essere *controvariante*.

Una funzione si dice controvariante quando, dato un insieme $B \supseteq A$ e $a \in A$

$$|\delta_B(a)^{-1}| \subseteq |\delta_A(a)^{-1}|$$

Questo significa che se B è un insieme di agenti più grande (al più uguale) di A, l'insieme di URL associato a ciascuno degli agenti in B sarà più piccolo (al più uguale) dell'insieme di URL associato al corrispettivo agente in A. In particolare, se si aggiunge un agente, nessuno degli agenti preesistenti si vede assegnare nuovi URL.

Questo problema è comune ad altri contesti, come il caching nelle *content delivery network* e i sistemi di calcolo distribuito *peer-to-peer*. È facile convincersi che una strategia come quella proposta per l'ambito statico risulterà disastrosa. Vedremo adesso tre modi diversi per gestire la questione.

3.1 Permutazioni aleatorie

Il primo approccio che esploreremo si basa sulle permutazioni aleatorie. Assumiamo vi sia un universo P di possibili agenti. A ogni istante dato l'insieme di agenti effettivamente utili è un insieme $A \subseteq P$. Per semplicità assumiamo che P sia formato dai primi $|P|$ numeri naturali (anche se in realtà è solo necessario che P sia totalmente ordinato).

Fissiamo una volta per tutte un generatore di numeri pseudocasuali. La strategia è ora la seguente: dato un URL u, inizializziamo il generatore utilizzando u (per esempio, passando u attraverso una funzione di hash prefissata) e utilizziamolo per generare una permutazione aleatoria di P scelta in maniera uniforme. A questo punto, l'agente scelto è il primo agente in A nell'ordine indotto dalla permutazione così calcolata.

Chiaramente, dato che le permutazioni vengono generate in maniera uniforme approssimativamente la stessa frazione di URL viene assegnata a ogni agente. Inoltre, se consideriamo un insieme $B \supseteq A$ le uniche variazioni di assegnazione consistono nello spostamento di URL verso elementi di $B \setminus A$, spostamento che avviene esattamente quando uno di tali elementi precede tutti quelli di A nella permutazione associata all'URL.

In sostanza, la permutazione fornisce un ordine di preferenza per assegnare u all'interno dell'insieme degli agenti possibili P. Il primo agente effettivamente disponibile (cioè, in A) sarà responsabile per il crawling di u. Dato che la permutazione è generata tramite un generatore comune a tutti gli agenti e utilizza lo stesso seme (u) tutti gli agenti calcolano lo stesso ordine di preferenza.

Si noti che è possibile generare una tale permutazione in tempo e spazio lineare in $|P|$. La tecnica, nota come *Knuth shuffle* o *Fisher-Yates shuffle* (lo pseudocodice viene mostrato nell'Algoritmo 2), consiste nell'inizializzare un vettore di $|P|$ elementi con i primi $|P|$ numeri naturali (o con gli elementi di P, nel caso generale).

A questo punto si eseguono $|P|$ iterazioni: all'iterazione di indice i (a partire da zero) si scambia l'elemento i-esimo con uno delle seguenti $|P| - i$ scelto a caso uniformemente (si noti che i stesso è una scelta possibile). Alla fine dell'algoritmo, il vettore contiene una permutazione aleatoria scelta in maniera uniforme. In realtà non è necessario generare l'intera permutazione: non appena

Algorithm 2 Fisher-Yates shuffle

Require: vettore di n posizioni `array`
  for $i \in \{0, \ldots, n-2\}$ **do**
    $j \leftarrow i + \texttt{UNIFORM}(n - i)$
    $\texttt{SWAP}(\texttt{array}[i], \texttt{array}[j])$
  end for

completiamo l'iterazione i, se troviamo in posizione i un elemento di A possiamo restituirlo, dato che nei passi successivi non verrà più spostato. Inoltre, tutte le modifiche ad A avvengono in tempo costante (dato che non c'è nulla da fare).

3.2 Min hashing

Il secondo approccio non richiede che ci sia un universo predeterminato o ordinato. Fissiamo una funzione di hash aleatoria $h(-,-)$ a due argomenti che prende un URL e un agente. L'agente $a \in A$ responsabile dell'URL u è quello che realizza il minimo tra tutti i valori $h(u, a)$.

Di nuovo, assumendo che h sia aleatoria, il bilanciamento è banale. Ma anche la proprietà di controvarianza è elementare: se $B \supseteq A$, gli unici URL che cambiano assegnamento sono quegli URL u per cui esiste un $b \in B \setminus A \mid h(b, u) < h(a, u)\ \forall\ a \in A$. Si noti che questo metodo richiede tempo di calcolo proporzionale a A, e spazio costante (basta tenere traccia del minimo elemento trovato). Anche in questo caso, le modifiche a A richiedono tempo costante, dato che non c'è nulla da fare.

3.3 Hashing Coerente

L'ultimo approccio è il più sofisticato. Consideriamo idealmente una circonferenza di lunghezza unitaria e una funzione di hash aleatoria h che mappa gli URL nell'intervallo $[0 \ldots 1)$ (cioè sulla circonferenza unitaria). Fissiamo inoltre un generatore di numeri pseudocasuali.

Per ogni agente $a \in A$, utilizziamo a per inizializzare il generatore e scegliere C posizioni pseudoaleatorie sul cerchio (ad esempio, in pratica, $C = 300$): queste saranno le *repliche* assegnate all'agente a. A questo punto per trovare l'agente responsabile di un URL u partiamo dalla posizione $h(u)$ e proseguiamo in senso orario finchè non troviamo una replica: l'agente associato è quello responsabile per u.

In pratica, il cerchio viene diviso in segmenti massimali che non contengono repliche. Associamo a ogni segmento la replica successiva in senso orario, e tutti gli URL mappati sul segmento avranno come agente responsabile quello associato alla replica. La funzione così definita è ovviamente controvariante. L'effetto di aggiungere un nuovo agente è semplicemente quello di spezzare tramite le nuove repliche i segmenti preesistenti: la prima parte verrà mappata sul nuovo agente, la seconda continuerà a essere mappata sull'agente precedente.

La parte delicata è, in questo caso, il bilanciamento: se C viene scelto sufficientemente grande (rispetto al numero di agenti possibili; si veda) i segmenti risultano così piccoli da poter dimostrare che la funzione è bilanciata con alta probabilità.

Si noti che la struttura può essere realizzata in maniera interamente discreta memorizzando interi, che rappresentano (in virgola fissa) le posizioni delle repliche, in un dizionario ordinato (come un b-tree), A questo punto, dato un URL u è sufficiente generare uno hash intero e trovare

il minimo maggiorante presente nel dizionario (eventualmente debordando alla fine dell'insieme e restituendo quindi il primo elemento).

Questo metodo richiede spazio lineare in $|A|$ e tempo logaritmico in $|A|$. Aggiungere o togliere un elemento ad A richiede, contrariamente ai casi precedenti, tempo logaritmico in $|A|$.

3.4 Note generali

In tutti i metodi che abbiamo delineato c'è una componente pseudoaleatoria. Questo fa sì che la distribuzione degli URL agli agenti non sia veramente bilanciata, ma segua piusttosto una distribuzione binomiale negativa. Considerato però il grande numero di elementi in gioco, il risultato approssima bene una distribuzione bilanciata.

Sia il min hashing che lo hashing coerente possono presentare *collisioni* - situazioni in cui non sappiamo come scegliere l'output perché due minimi, o due repliche, coincidono - in tal caso, è necessario disambiguare deterministicamente il risultato (imponendo, ad esempio, un ordine arbitrario sugli agenti).

Tutti i metodi permettono (eventualmente con uno sforzo computazionale aggiuntivo) di sapere quale sarebbe il prossimo agente responsabile per un URL, se quello corrente non fosse presente (l'agente designato è crashato). Nel caso delle permutazioni aleatorie è sufficiente progredire nello shuffle, cercando l'elemento successivo in A. Nel caso del min hashing è necessario tenere traccia di due valori: il minimo e il valore immediatamente successivo. Infine, nel caso dell'hashing coerente, basta continuare a procedere in senso orario fino a giungere alla replica di un nuovo agente.

4 Codici istantanei

Un *codice* è un insieme $C \subseteq 2^*$, cioè un insieme di parole binarie. Si noti che per ovvie ragioni di cardinalità C è al più numerabile.

Definiamo l'*ordinamento per prefissi* delle sequenze in 2^* come segue:

$$x \preceq y \iff \exists z \mid y = xz \tag{5}$$

Cioè $x \preceq y$ se e solo se x è un prefisso di y. Ricordiamo che in un *ordine parziale* due elementi sono inconfrontabili se nessuno dei due è minore dell'altro.

Un codice è detto *istantaneo* o *privo di prefissi* se ogni coppia di parole distinte del codice è inconfrontabile. L'effetto pratico di questa proprietà è che a fronte di una parola w formata da una concatenazione di parole del codice, non esiste una diversa concatenazione che dà w. In particolare, leggendo uno a uno i bit di w è possibile ottenere in maniera istantanea le parole del codice che lo compongono.

Ad esempio, il codice $\{0, 1\}$ è istantaneo, mentre $\{0, 1, 01\}$ non lo è. Se prendiamo ad esempio la stringa 001 e la confrontiamo con il primo codice sappiamo che è formata da 0, 0, 1, ma nel secondo caso possiamo scegliere tra 0, 0, 1 e 0, 01.

Un codice si dice *completo* o *non ridondante* se ogni parola $w \in 2^*$ è confrontabile con qualche parola del codice (esiste quindi una parola del codice di cui w è prefisso o una parola del codice che è un prefisso di w). Il primo dei due codici summenzionato è completo, mentre il secondo non lo è.

Quando un codice istantaneo è completo, non è possibile aggiungere parole al codice senza perdere la proprietà d'istantaneità; inoltre, qualunque parola *infinita* è decomponibile in maniera unica come una sequenza di parole del codice, qualunque parola *finita* è decomponibile in maniera unica come sequenza di parole del codice più un prefisso di qualche parola del codice.

Ora enunciamo la disequazione di *Kraft-McMillan*, che dimostreremo successivamente.

Teorema 1. *Sia $C \subseteq 2^*$ un codice, se C è istantaneo, allora*

$$\sum_{w \in C} 2^{-|w|} \leq 1$$

C è completo se e solo se l'uguaglianza vale. Inoltre, data una sequenza, eventualmente infinita, $t_0, \ldots, t_{n-1}, \ldots$ che soddisfa:

$$\sum_{i \in n} 2^{-t_i} \leq 1$$

esiste un codice istantaneo formato da parole $w_0, \ldots, w_{n-1}, \ldots$ tali che $|w_i| = t_i$

Prima di cominciare la dimostrazione facciamo un preambolo utile a eseguirla in modo veloce.

Un *diadico* è un razionale della forma $k2^{-h}$. A ogni parola $w \in 2^*$. Possiamo associare un sottointervallo semiamperto di $[0 \ldots 1)$ con estremi diadici come segue:

- Se w è la parola vuota, l'intervallo è $[0 \ldots 1)$
- Se w privata dell'ultimo bit ha $[x \ldots y)$ come intervallo associato, se l'ultimo bit è 0 allora l'intervallo di w è $[x \ldots (x + y)/2)$, altrimenti l'intervallo di w è $[(x + y)/2 \ldots y)$

Per poter visualizzare quanto segue è utile costruire qualche esempio semplice e inserire le parole del codice in un albero, in cui ogni nodo ha al più due figli etichettati 0 e 1. Si possono fare le seguenti **osservazioni**:

1. L'intervallo associato a una parola di lunghezza n ha dimensione 2^{-n}
2. se $v \preceq w$ l'intervallo associato a v contiene quello associato a w

3. Due parole sono inconfrontabili se solo i corrispondenti intervalli sono disgiunti; infatti, se v e w sono inconfrontabili e z è il loro massimo prefisso comune, assumendo senza perdita di generalità che $z_0 \preceq v$ e $z_1 \preceq w$ l'intervallo di v sarà contenuto in quello di z_0 e l'intervallo di w in quello di z_1: dato che gli intervalli di z_0 e z_1 sono disgiunti per definizione, lo sono anche quelli di v e w

4. Dato un qualunque intervallo diadico $[k2^{-h} \dots (k+1)2^{-h})$ esiste un'unica parola di lunghezza h a cui è associato l'intervallo, vale a dire, la parola formata dalla rappresentazione binaria di k allineata ad h bit; questo è certamente vero per $h = 0$, e data una parola w con $|w| = h + 1$ se l'intervallo associato a w privato dell'ultimo carattere è $[k2^{-h} \dots (k+1)2^{-h})$ l'intervallo associato a w è $[(2k)2^{-h-1} \dots (2k+1)2^{-h-1})$; se l'ultimo carattere di w è 0, $[(2k+1)2^{-h-1} \dots 2(k+1)2^{-h-1})$

Fatte le dovute premesse possiamo passare alla dimostrazione del Teorema 1.

Dimostrazione. Sia ora C un codice istantaneo. La sommatoria contenuta nell'enunciato del Teorema 1 è la somma delle lunghezze degli intervalli associati alle parole di C; questi intervalli sono disgiunti e la loro unione forma un sottoinsieme di $[0 \dots 1)$ che ha necessariamente lunghezza minore o uguale di 1.

($\Longrightarrow$) Se la sommatoria è strettamente minore di 1 deve esserci per forza un intervallo scoperto, diciamo $[x \dots y)$. Questo intervallo contiene necessariamente un sottointervallo della forma $[k2^{-h} \dots (k+1)2^{-h})$ per qualche h, k. Ma allora la parola associata a quest'ultimo potrebbe essere aggiunta al codice (essendo inconfrontabile con le altre [osservazione 3]). Che quindi risulta essere incompleto.

($\Longleftarrow$) D'altra parte, se il codice è incompleto l'intervallo corrispondente a una parola inconfrontabile con tutte quelle del codice è necessariamente scoperto, e rende la somma strettamente minore di 1.

Andiamo ora a dimostrare l'ultima parte dell'enunciato, assumendo, senza perdita di generalità, che la sequenza $t_0, \dots, t_{n-1}, \dots$ sia monotona non decrescente.

Genereremo le parole $w_0, \dots, w_{n-1}, \dots$ in maniera *miope*. Sia d l'estremo sinistro della parte d'intervallo unitario correntemente coperta dagli intervalli associati dalle parole già generate: inizialmente, $d = 0$. Manterremo vero l'invariante che prima dell'emissione della parola w_n si ha $d = k2^{-t_n}$ per qualche k, il che ci permetterà di scegliere come w_n l'unica parola di lunghezza t_n il cui intervallo ha estremo sinistro d. Dato che l'intervallo associato a ogni nuova parola è disgiunto dall'unione dei precedenti, le parole generate saranno tutte inconfrontabili.

L'invariante è ovviamente vero quando $n = 0$. Dopo aver generato w_n, d viene aggiornato sommandogli 2^{-t_n} e diventa quindi $(k+1)2^{-t_n}$. Ma dato che $(k+1)2^{-t_n} = ((k+1)2^{t_{n+1}-t_n})2^{-t_{n+1}}$ e $t_{n+1} \geq t_n$, l'invariante viene mantenuto. □

4.1 Codici istantanei per gli interi

Alcuni dei metodi più utilizzati per la compressione degli indici utilizzano codici istantanei per gli interi. Questa scelta può apparire a prima vista opinabile per il fatto che i valori che compaiono in un indice hanno delle limitazioni superiori naturali e sono facili da calcolare, e quindi potrebbe essere più efficiente calcolare un codice istantaneo per il solo sottoinsieme d'interi effettivamente utilizzato.

In realtà se si lavora con collezioni documentali di grandi dimensioni la semplicità teorica e implementativa dei codici per gli interi li rende molto interessanti.

Innanzitutto si noti che un codice istantaneo per gli interi è numerabile. L'associazione tra interi e parole del codice va specificata di volta in volta, anche se, in tutti i codici che vedremo, l'associazione è semplicemente data dall'ordinamento prima per lunghezza e poi lessicografico delle parole. Inoltre assumeremo che le parole rappresentino numeri naturali, e quindi la parola minima (cioè lessicograficamente minima tra quelle di lunghezza minima) rappresenti lo zero[5]. La

[5]Questa scelta non è uniforme in letteratura, e in effetti si possono trovare nello stesso libro due codici per gli interi che, a seconda della bisogna, vengono numerati a partire da zero o da uno

rappresentazione più elementare di un intero n è quella *binaria*, che però non è istantanea (le prime parole sono 0, 1, 10, 11, 100). È possibile rendere il codice istantaneo facendo un allineamento a k bit. La lunghezza di una parola di codice binario (non allineato) è $\lambda(n)+1$[6].

Chiameremo *rappresentazione binaria ridotta* di n la rappresentazione binaria di $n+1$ alla quale viene rimosso il bit più significativo; anch'essa non è istantanea. La lunghezza della parola di codice per n è $\lambda(n+1)$. Le prime parole sono ε, 0, 1, 00, 01, 10.

Un ruolo importante nella costruzione dei codici istantanei è svolto dai *codici binari minimali* - codici istantanei e completi per i primi k numeri naturali che utilizzano un numero variabile di bit. Esistono diverse possibilità per le scelte delle parole del codice[7], ma in quanto segue diremo che il codice binario minimale di n (nei primi n naturali) è definito come segue: sia $s = \lceil \log(k) \rceil$; se $n < 2^s - k$, n è codificato dall'n-esima parola binaria (in ordine lessicografico) di lunghezza $s-1$; altrimenti, n è codificato utilizzando la $(n-k+2^s)$-esima parola binaria di lunghezza s.

La lunghezza di una parola di binario minimale è $s + [n < 2^s - k]$

	k						
	1	2	3	4	5	6	7
0	ε	0	0	00	00	00	00
1		1	10	01	01	01	010
2			11	10	10	100	011
3				11	110	101	100
2					111	110	101
2						111	110
2							111

La base di tutti i codici istantanei per gli interi è il codice *unario*. Il codice unario rappresenta il naturale n tramite n uno seguiti da uno 0[8]. Le prime parole del codice sono 0, 10, 110, …. La lunghezza di una parola in unario è banalmente $n+1$. Il codice è sia istantaneo e completo.

Il codice γ [Eli75] codifica un intero n scrivendo il numero di bit della rappresentazione ridotta in unario, seguito dalla rappresentazione binaria ridotta di n. Le prime parole del codice sono 1, 010, 011, 00100, 00101, …. La lunghezza della parola di codice è quindi $2\lambda(n+1)+1$ e il codice è sia istantaneo che completo, questo si deve al fatto che l'unario è istantaneo e completo.

Analogamente, il codice δ [Eli75] codifica un intero n scrivendo il numero di bit della rappresentazione binaria ridotta di n in γ, seguito dalla rappresentazione binaria ridotta di n. Le prime parole del codice sono 1, 0100, 0101, 01100, 01101, …. La lunghezza della parola di codice per n è quindi $2\lambda(\lambda(n+1)+1)+1+\lambda(n+1)$ e il fatto che il codice sia istantaneo e completo deriva dal fatto che il γ lo sia.

Si potrebbe provare a continuare in questa direzione, ma come vedremo, senza vantaggi significativi.

Il *Codice di Golomb di modulo k* [Gol80] codifica un numero intero n scrivendo il quoziente della divisione di n per k in unario, seguito dal resto in binario minimale. Le prime parole del codice per $k=3$ sono 10, 110, 111, 010, …. La lunghezza della parola di codice per n è quindi $\lfloor n/k \rfloor + 1 + \lambda(x \mod (k)) + [x \mod (k) \geq 2^{\lceil \log(k) \rceil} - k]$ e il fatto che sia istantaneo e completo deriva dal fatto che lo sono sia il codice unario che il codice binario minimale.

Infine conviene ricordare i *codici a blocchi di lunghezza variabile*, come il codice variabile a nibble o a byte. L'idea è che ogni parola è formata da un numero variabile di blocchi di k bit (4 nel caso di nibble e 8 nel caso di byte). Il primo bit non è codificante ed è noto come *bit di continuazione*, se posto a 1 il blocco che stiamo considerando non è quello finale, se posto a 0 abbiamo raggiunto il blocco terminale.

[6] Ricordo che: $\lambda(n) = \lfloor \log(n) \rfloor$

[7] In realtà, un codice binario minimale è semplicemente un codice ottimo per la distribuzione uniforme, il che spiega perché sono possibili scelte diverse per le parole del codice

[8] Nelle note originali il professore definisce l'unario all'incontrario e mette in una nota quello che ho definito io, ma è evidente che questa definizione è più importante all'atto pratico per il semplice fatto che fa coincidere ordine lessicografico e l'ordine dei valori rappresentati

In fase di codifica un intero n viene scritto in notazione binaria, allineato a un multiplo di $k+1$ bit, diviso in blocchi di $k+1$ bit (k bit codificanti e un bit di continuazione che non codifica i valori), e rappresentato tramite una sequenza di suddetti blocchi, ciascuno preceduto dal bit di continuazione. La lunghezza della parola di codice è pari a $\lceil(\log(x)+1)/k\rceil\,(k+1)$. I codici a lunghezza variabile sono ovviamente istantanei ma non completi, questo perché sequenze di 0 che sono più lunghe di un blocco non sono confrontabili con nessuna delle parole del codice. Uno standard alternativo per questo tipo di codici è quello implementato da UTF-8, che anzichè perdere il primo bit di ogni blocco per i bit di continuazione, sfrutta il primo blocco dell'intera parola per codificare quanti saranno i blocchi costituenti la parola.

	γ	δ	Golomb ($b=3$)	nibble
0	1	1	10	1000
1	010	0100	110	1001
2	011	0101	111	1010
3	00100	01100	010	1011
4	00101	01101	0110	1100
5	00110	01110	0111	1101
6	00111	01111	0010	1110
7	0001000	00100000	00110	1111
8	0001001	00100001	00111	00011000
9	0001010	00100010	00010	00011001
10	0001011	00100011	000110	00011010
11	0001100	00100100	000111	00011011
12	0001101	00100101	000010	00011100
13	0001110	00100110	0000110	00011101
14	0001111	00100111	0000111	00011110
15	000010000	001010000	0000010	00011111

4.2 Caratteristiche matematiche dei codici

Esistono delle caratteristiche intrinseche dei codici istantanei per gli interi che permettono di classificarli e distinguerne il comportamento. In particolare, un codice è *universale* se per qualunque distribuzione p sugli interi monotona non crescente ($p(i) \leq p(i+1)$) il valore atteso della lunghezza di una parola rispetto a p è minore o uguale dell'entropia di p a meno di costanti additive e moltiplicative indipendenti da p. Ciò significa che se $l(n)$ è la lunghezza della parola di codice per n e $H(p)$ è l'entropia di una distribuzione p (nel senso di Shannon), esistono c, d costanti tali che:

$$\sum_{x\in\mathbb{N}} l(n)p(n) \leq cH(p) + d$$

Un codice è detto *asintoticamente ottimo* quando a destra il limite superiore è della forma $f(H(p))$ con $\lim_{n\to\infty} f(n) = 1$.

Il codice unario e i codici di Golomb non sono universali, mentre lo sono γ e δ inoltre, quest'ultimo, è anche asintoticamente ottimo.

4.3 Codifiche alternative

È possibile utilizzare tecniche standard quali i codici di Huffman o la compressione aritmetica per la codifica di ogni parte di un indice. Ci sono però delle considerazioni ovvie che mostrano come questi metodi, utilizzati direttamente, non siano di fatto implementatibili. La codifica di Huffman richiederebbe un numero di parole di codice esorbitante, e le parole dovrebbero essere calcolate separatamente per ogni termine. La codifica aritmetica richiede alcuni bit di scarico prima di poter essere interrotta, e quindi non si presta a essere inframezzata da altri dati (come i conteggi e le posizioni). Inoltre, per quanto efficientemente implementata, è estremamente lenta.

4.3.1 PFOR-DELTA

Un approccio che ha rivoluzionato il modo di memorizzare le liste di affissione, è quello proposto in [Bon06], comunemente chiamato PFOR-DELTA o FOR. L'idea è molto semplice: si sceglie una dimensione di blocco B (di solito 128 o 256), e per ogni blocco consecutivo di B scarti si trova l'intero b tale per cui la maggior parte dei valori[9] possono essere rappresentati con b bit.

A questo punto viene scritto un vettore di B interi a b bit, che descrive correttamente il 90% dei valori, quelli che non possono essere espressi in b bit sono dette *eccezioni* e vengono scritte in un vettore a parte impiegando, il minimo numero di bit, atti a descrivere il massimo valore nell'array, per ogni elemento. Gli scarti vengono memorizzati in ordine in B, quando incontriamo un'eccezione inseriamo una sequenza di escape (tipo un elemento dell'array costituito da soli 1) che consente di capire che lì è presente un eccezione, è dunque possibile in fase di decodifica rimpiazzare il valore di escape con l'eccezione corrispondente; siccome tutti e due i vettori sono ordinati l'operazione può essere eseguita sequenzialmente.

In fase di decodifica, si copiano i primi B valori a b bit in un vettore di interi. Quest'operazione è molto veloce, in particolare se vengono creati dei cicli srotolati diversi[10] per ogni possibile valore di b. A questo punto si passa attraverso la lista delle eccezioni, che vengono copiate dal secondo vettore nelle posizioni corrette.

Questo approccio fa sì che il processore esegua dei cicli estremamente predicibili, e riesca a decodificare un numero di scarti al secondo significativamente più alto delle tecniche basate su codici. Le performance di compressione sono di solito buone, anche se è difficile dare garanzie teoriche. Lo svantaggio (che può risultare significativo) è che è sempre necessario decodificare B elementi.

4.3.2 Elias Fano

Elias Fano è un meccanismo di codifica e compressione che può essere utilizzato per memorizzare in maniera efficiente sequenze di interi monotone non decrescenti.

La lista dei puntatori documentali associati a un determinato token in una lista di affissione possono essere memorizzati tramite una lista per scarti (la stessa cosa può essere fatta, volendo, su posizioni e conteggi). Il problema di una lista per scarti è che senza ulteriori magheggi (tabelle di salto) hanno performance di rango e selezione pessime e la loro implementazione efficiente ed efficace può non essere banale, oltre a richiedere un ulteriore sforzo non indifferente.

La codifica di Elias Fano consente di rappresentare sequenze monotone in maniera quasi succinta[11].

Supponiamo di avere la seguente successione monotona non decrescente, di lunghezza n, $x_0, \ldots, x_{n-1}$. Supponiamo che il limite superiore della successione sia un valore u.

Elias Fano consiste nella memorizzazione dei $\lfloor \log(u/n) \rfloor$ bit meno significativi di ogni valore in maniera esplicita (avremo un vettore L); i bit più significativi di ogni valore verranno memorizzati come una lista di scarti in unario dove 0^k1 rappresenta k (avremo quindi anche un vettore H). Questa struttura ci consente di memorizzare n valori utilizzando solamente $\log(u/n) + 2$ bit al più per elemento, affermazione che dimostreremo al termine di questa sottosezione.

Consideriamo il seguente esempio: Abbiamo la seguente successione di valori

$$5 \quad 8 \quad 8 \quad 15 \quad 32$$

[9] Il professore trattando l'argomento propone di coprire il 90% dei valori con b bit, assumeremo 90% come valore di riferimento da qui in avanti. In questo caso però la filosofia è semplicemente più valori si coprono con b, meglio è (attenzione a non cadere nell'approccio miope di coprire tutti i valori, soprattutto qualora si fosse costretti a impiegare un numero di bit considerevole.)

[10] Unrolling loops, significa semplicemente scrivere un'istruzione per ogni valore del loop, richiede di scrivere migliaia di righe di codice che però possono essere generate automaticamente.

[11] Una struttura dati è quasi-succinta quando lo spazio occupato dalla struttura si avvicina molto al lower bound teorico. Si tratta di un termine impiegato dal professor Vigna nelle dispense di cui non sono riuscito a trovare ulteriore evidenza in rete.

vogliamo memorizzarla con Elias Fano usando $u = 36$. Siccome $n = 5$ $l = 2$, dunque il vettore contenente i bit meno significativi avrà la seguente struttura

$$01 \quad 00 \quad 00 \quad 11 \quad 00$$

il vettore di valori in unario sarà il seguente

$$0\ 1\ 0\ 1\ 1\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 1$$

Prendiamo qualche valore per capire come funziona il processo di memorizzazione: per il valore 5 (101) prendo gli l bit meno significativi (01), li metto nel vettore L, prendo i bit rimanenti (1) e li memorizzo in unario (01) all'interno del vettore H.

Per memorizzare 8 (il valore successivo, 1000) prendo gli l bit meno significativi (00), li metto nel vettore L, prendo i bit rimanenti (10), calcolo lo scarto rispetto a $H[0]$ (1 - 2 = 1) e lo memorizzo in unario (01) all'interno del vettore H, e il ciclo continua fino a quando tutti i valori non sono stati memorizzati.

I vantaggi di Elias Fano sono i seguenti:

- Utilizzo ottimale dello spazio.
- Per utilizzare questa codifica non è necessario che i valori seguano una particolare distribuzione.
- La lettura sequenziale richiede pochissime operazioni.
- Restrizione del problema di rango e selezione a un array di circa 2n bit contente per metà 0 e per metà 1.

Visto che abbiamo nominato il problema di rango e selezione, come viene calcolato rango e selezione in Elias Fano?

Per avere il k-esimo elemento recuperiamo gli l bit meno significativi usando $L[k-1]$[12] mentre i bit più significativi corrispondono al numero di zeri prima del k-esimo 1.

Se per esempio volessi recuperare il terzo valore all'interno della sequenza d'esempio che si trova sopra:

I bit meno significativi sono quelli in posizione $L[2] = 00$, i bit più significativi sono il numero di 0 prima del k-esimo 1, quindi $2 = 10$. Il terzo valore è 1000 cioè 8.

Il processo di rango è un po'meno intuitivo ma comunque in tutto simile a quello di selezione. Volendo trovare il minimo valore $x \geq b$ facciamo quanto segue

- $b >> l$ per capire quanti bit a 0 devo saltare dentro H
- Calcolo la posizione di blocco in cui sono
- Gli upper bits sono la somma di tutti i valori in unario che incontro fino a quando non trovo l'$(b >> l)$-esimo bit a 0.
- Prendo il valore di blocco dentro l'array dei lower bits nella posizione che ho trovato prima.

Proviamo a chiarire il concetto con un esempio (rimanendo sempre sui valori definiti prima). Il minimo valore maggiore di 11011 (27) è

- So di dover saltare 110 bit a 0, quindi 6.
- La posizione del blocco in cui sono è 4 (partendo da 0).
- La parte alta del numero è data da 01 + 01 + 1 + 01 + 000001 = 000000001 (8), 1000 in binario.

[12] Uso $k - 1$ supponendo che la numerazione parta da 0 e che quindi si voglia ottenere il terzo valore secondo un conteggio ordinale

- I lower bits sono quelli in posizione 4 all'interno di L, quindi 00. Il risultato finale è 100000, cioè 32.

Prendiamo ora il caso in esame, vogliamo memorizzare l'indice inverso, il che significa memorizzare i puntatori documentali ed eventualmente posizioni e conteggi. Per quel che riguarda i puntatori documentali utilizziamo, ovviamente, una lista quasi-succinta implementando una tabella di salto che ci consenta di rendere le letture ancora più veloci facendo un piccolo sacrificio in termini di spazio. Per quel che riguarda le posizioni e i conteggi basta tenere a mente che possiamo tenere in memoria una lista quasi-succinta memorizzando $x_i - i$ per successioni strettamente monotone.

Al posto di memorizzare i conteggi, possiamo memorizzare la loro cumulata $(x_0, x_0 + x_1, x_0 + x_1 + x_2, \dots)$, che risulta non decrescente, mentre per le posizioni si può salvare la cumulata degli scarti. L'idea di base è che la cumulata dei conteggi può essere utilizzata come funzione per indicizzare le posizioni.

Elias Fano è una tecnica di codifica per indici veloce e compatta, che può essere battuta per compressione solo da codifiche più lente come il Golomb, per contro è scalabile e ha un accesso locale molto migliore di altri approcci, rendendolo una tecnica che si può quasi definire un *go-to*.

4.3.3 Analisi spaziale di Elias Fano

Prima di partire con l'analisi spaziale di Elias Fano definiamo una serie di strumenti che potrebbero tornare utili nell'arco della dimostrazione.

Sia $X = \{x_i\}$ una successione di valori monotona non decrescente, sia u l'upper bound di questa successione di valori, siano:

- L il vettore in cui verranno salvate le parti inferiori dei valori della successione
- H il vettore in cui verranno salvate le parti superiori dei valori della successione

siano inoltre n la dimensione della successione X, l il numero dei bit meno significativi ($l = \lfloor \log(u/n) \rfloor$). A questo punto reintroduciamo il teorema e poi passiamo a dare la sua dimostrazione.

Teorema 2. *Una successione monotona non decrescente può essere memorizzata tramite la codifica di Elias Fano impiegando, al più*

$$\lceil \log(u/n) \rceil + 2 \tag{6}$$

bit per elemento

Dimostrazione. Come si è già detto in precedenza, per ogni elemento x, memorizzo:

- I bit più significativi come 0^x1, impiegando $x + 1$ bit, chiameremo i bit più significativi u_i.
- I bit meno significativi in binario con $\log(u/n)$ elementi.

Quindi lo spazio occupato dal vettore U è

$$\sum_{i=0}^{n-1} u_i + 1$$

Ricordo che i valori più significativi u_i sono salvati come scarti in U, quindi posso scrivere la somma qui sopra come:

$$\sum_{i=0}^{n-1} \left\lfloor \frac{x_i}{2^l} \right\rfloor - \left\lfloor \frac{x_{i-1}}{2^l} \right\rfloor + 1$$

A questo punto se consideriamo $\left\lfloor \frac{x_i}{2^l} \right\rfloor - \left\lfloor \frac{x_{i-1}}{2^l} \right\rfloor$ e calcoliamo l'espressione per tutti i valori di i quello che scopriremo è che viene una somma finita di questo tipo:

$$\left\lfloor \frac{x_0}{2^l} \right\rfloor - \left\lfloor \frac{x_{-1}}{2^l} \right\rfloor + \left\lfloor \frac{x_1}{2^l} \right\rfloor - \left\lfloor \frac{x_0}{2^l} \right\rfloor + \cdots + \left\lfloor \frac{x_{n-1}}{2^l} \right\rfloor - \left\lfloor \frac{x_{n-2}}{2^l} \right\rfloor$$

È intuitivo notare quanto segue:

$$\cancel{\left\lfloor \frac{x_0}{2^l} \right\rfloor} - \left\lfloor \frac{x_{-1}}{2^l} \right\rfloor + \cancel{\left\lfloor \frac{x_1}{2^l} \right\rfloor} - \cancel{\left\lfloor \frac{x_0}{2^l} \right\rfloor} + \cdots + \left\lfloor \frac{x_{n-1}}{2^l} \right\rfloor - \cancel{\left\lfloor \frac{x_{n-2}}{2^l} \right\rfloor}$$

Quindi rimangono solamente i termini $-\left\lfloor \frac{x_{-1}}{2^l} \right\rfloor$ e $\left\lfloor \frac{x_{n-1}}{2^l} \right\rfloor$. Facendo la supposizione che $x_{-1} = 0$[13] possiamo dire quanto segue

$$\sum_{i=0}^{n-1} \left\lfloor \frac{x_i}{2^l} \right\rfloor - \left\lfloor \frac{x_{i-1}}{2^l} \right\rfloor + 1 = n + \left\lfloor \frac{x_{n-1}}{2^l} \right\rfloor \leq n + \frac{u}{2^l}$$

Siccome $l = \lfloor \log(u/n) \rfloor$ possono succedere due cose sulla base del valore di u/n Se $u/n \mod (2) = 0$ allora

$$n + \frac{u}{\frac{u}{n}} = 2n$$

Altrimenti

$$n + \frac{u}{2^{\lfloor \log(u/n) \rfloor}} \leq n + \frac{u}{2^{\log(u/n)-1}} = n + \frac{u}{\frac{u}{2n}} = 3n$$

Quindi la struttura dati occupa lo spazio necessario a memorizzare il vettore U, cioè ln e poi $2n$ o $3n$ bit sulla base di u/n. Per poter uniformare la formula ricordo che:

$$\lceil \log(u/n) \rceil = \begin{cases} l & \text{Se } u/n \text{ è potenza di 2} \\ l+1 & \end{cases}$$

Quindi facendo una banale sostituzione, si ricava la seguente formula unificata:

$$D_n = 2n + \left\lceil \log\left(\frac{u}{n}\right) \right\rceil n \tag{7}$$

□

4.3.4 Sistemi di numerazione asimmetrica (approfondimento)

Nel 2013, Duda [Dud13] ha introdotto un nuovo metodo di codifica di sequenze di simboli che ha rivoluzionato il campo della compressione dati. I *sistemi di numerazione asimmetrica* hanno velocità pari o superiore a un codice di Huffman, ma performance di compressione confrontabili con quelle della codifica aritmetica, quindi davvero ottime (molto vicine al limite teorico di Shannon). Sono alla base di tutti i moderni sistemi di compressione come `zstd` di Facebook e lo standard JPEG XL.

L'idea alla base dei sistemi di numerazione arimtetica è che per comprimere una sequenza di simboli vorremo (idealmente) utilizzare un intero molto grande. Se una sequenza è codificata da x, vorremmo che aggiungere un simbolo y che ha probabilità di comparire p_y portasse a una codifica $x' \approx x/p_s$, perché in questo modo $\log(x') = \log(x) + \log(1/p_y)$. Aggiungendo quindi y alla sequenza, spenderemo $\log(1/p_y)$ bit. Complessivamente il costo del messaggio sarà l'entropia di Shannon.

Possiamo vedere questo concetto nella codifica banale della sorgente più semplice, con 0 e 1 equiprobabili. Se x è la codifica dei simboli visti fino ad ora, $x' = 2x$ oppure $x' = 2x+1$ a seconda del simbolo che compare. Il risultato è un intero che in base 2 è descritto dalla sequenza di 0 e 1 da codificare.

Se vogliamo decodificare la sequenza rispetto a x', l'ultimo valore codificato è semplicemente $x' \mod 2$, la sequenza rimane $\lfloor x'/2 \rfloor$. Iterando la procedura otteniamo le cifre in ordine *inverso* rispetto alla codifica.

[13] Supposizione fatta durante le lezioni del professor Paolo Boldi, non ho guardato dimostrazioni alternative quindi non so se sia standard o una semplificazione introdotta per far tornare i conti.

Questa è una caratteristica comune a tutti i sistemi di numerazione asimmetrica: si codifica in una direzione, si decodifica nella direzione opposta, come in una pila. Possiamo decomprimere i valori nell'ordine corretto comprimendo la sequenza al contrario.

È chiaro che il ragionamento è analogo nel caso in cui si abbiano k valori equiprobabili: L'unica differenza è che i valori saranno codificati tramite stringhe di k simboli.

Generalizziamo ora l'idea all'ambito di un numero arbitrario di simboli con distribuzione arbitraria. Definiremo:

- L'operazione di *push* che riceve in ingresso l'intero rappresentante la sequenza, un nuovo simbolo e restituisce la rappresentazione della sequenza con il simbolo aggiunto.
- L'operazione di *pop* che riceve in ingresso l'intero rappresentante la sequenza e restituisce il valore rappresentante la coda privato dell'ultimo simbolo e l'ultimo simbolo.

Sia $\Sigma = [0 \dots k)$ l'insieme dei simboli, per ogni $s \in \Sigma$ sia p_s la sua probabilità. Fissata una precisione d, tutte le probabilità verranno espresse, da qui in avanti, come approssimazioni di multipli $1/2^d$.

Dunque per ogni s, abbiamo che $p_s = f_s/2^d$. Definiamo $f_n = 2^d$, più grande è d, maggiore sarà la precisione con cui rappresentiamo le probabilità iniziali (quindi migliore sarà la compressione).

Prendiamo ora i primi 2^d interi e dividiamoli in n segmenti contigui di lunghezza f_s. A simboli più frequenti corrisponderanno segmenti di lunghezza maggiore, definiamo la cumulata degli f_s come segue

$$\sum_{t<s} f_t \quad \text{per } s \in [0 \dots n]$$

Notiamo che ogni elemento x di 2^d sta in uno dei segmenti, e quindi a ogni simbolo è associato esattamente un simbolo s. Più precisamente, esiste un solo s tale per cui $c_s \leq x \leq c_{s+1}$. Poniamo $sym(x) = s$ e $sym(f_s) = s$, questo ci permette di fare in modo che, prendendo un elemento x di 2^d uniformemente a caso, $sym(x) \in \Sigma$ ha esattamente distribuzione p.

Procediamo ora a dare la definizione delle operazioni di cui si parlava prima:

$$\texttt{push(x, s)} = (\lfloor x/f_s \rfloor << d) + c_s + x \mod f_s$$
$$\texttt{pop(x)} = \langle (x >> d) \cdot f_s + x \mod 2^d - c_s, s \rangle \quad \text{dove } s = sym(x \mod 2^d)$$

È facile notare che le due operazioni sono una l'inversa dell'altra. Inoltre, `push(x, s)` $\approx (x/f_s) \cdot 2^d = x \cdot (f_s/2^d) = x/p_s$, come volevamo.

Mettiamo ora alla prova ciò di cui abbiamo parlato con un semplice esempio. Abbiamo tre simboli:

- 0, che compare con probabilità 9/10 ($\log(1/p_0) \approx 0.152$)
- 1, che compare con probabilità 1/30 ($\log(1/p_1) \approx 4.9$)
- 2, che compare con probabilità 2/30 ($\log(1/p_2) \approx 3.9$)

L'entropia di Shannon, che viene calcolata come $\sum_{x \in X} p(x) \log(p(x))$ è circa 0.558. Se utilizziamo $d = 10$ abbiamo le seguenti probabilità:

- 922/1024 per 0
- 34/1024 per 1
- 68/1024 per 2

Consideriamo la sequenza

$$0, 0, 0, 0, 0, 1, 0, 0, 0, 2, 0, 0$$

Partiamo ora da $x = 0$, ora

$$\begin{aligned}
\texttt{push(0, 0)} &= (\lfloor 0/922 \rfloor << 10) + 0 + 0 = 0 \qquad (1\text{bit}) \\
\texttt{push(0, 0)} &= (\lfloor 0/922 \rfloor << 10) + 0 + 0 = 0 \qquad (1\text{bit}/2) \\
\texttt{push(0, 0)} &= (\lfloor 0/922 \rfloor << 10) + 0 + 0 = 0 \qquad (1\text{bit}/3) \\
\texttt{push(0, 0)} &= (\lfloor 0/922 \rfloor << 10) + 0 + 0 = 0 \qquad (1\text{bit}/4) \\
\texttt{push(0, 0)} &= (\lfloor 0/922 \rfloor << 10) + 0 + 0 = 0 \qquad (1\text{bit}/5) \\
\texttt{push(0, 1)} &= (\lfloor 0/34 \rfloor << 10) + 922 + 0 = 922 \qquad (9\text{bit}/6) \\
\texttt{push(1024, 0)} &= (\lfloor 1024/922 \rfloor << 10) + 0 + 0 = 1024 \qquad (10\text{bit}/7) \\
\texttt{push(1024, 0)} &= (\lfloor 922/922 \rfloor << 10) + 0 + 102 = 1126 \qquad (11\text{bit}/8) \\
\texttt{push(1126, 0)} &= (\lfloor 1126/922 \rfloor << 10) + 0 + 204 = 1228 \qquad (11\text{bit}/9) \\
\texttt{push(1228, 2)} &= (\lfloor 1228/68 \rfloor << 10) + 956 + 4 = 19392 \qquad (15\text{bit}/10) \\
\texttt{push(19392, 0)} &= (\lfloor 19392/922 \rfloor << 10) + 0 + 30 = 21534 \qquad (15\text{bit}/11) \\
\texttt{push(21534, 0)} &= (\lfloor 21534/922 \rfloor << 10) + 0 + 328 = 23880 \qquad (15\text{bit}/12)
\end{aligned}$$

Nelle implementazioni pratiche viene eseguito un processo di *rinormalizzazione*: quando l'intero calcolato diviene troppo grande da maneggiare, la metà superiore dei suoi bit viene scritta in un buffer di output e si continua a gestire la metà inferiore. Laa stessa operazione, invertita, consente di rileggere. La rinormalizzazione comporta un'ulteriore perdita di compressione, ma permette di operare su interi di dimensione fissata.

4.4 Distribuzioni intese

Ogni codice istantaneo completo per gli interi definisce implicitamente una *distribuzione intesa* sugli interi, che assegna a w la probabilità $2^{-|w|}$. Il codice, in effetti, risulta *ottimo* per la distribuzione associata. La scelta di un codice istantaneo può quindi essere ricondotta a considerazioni sulla distribuzione statistica degli interi che si intendo rappresentare.

Al codice unario è associata una distribuzione geometrica.

$$\frac{1}{2^{x+1}}$$

mentre la distribuzione associata a γ è

$$\frac{1}{2^{2\lfloor \log(x+1) \rfloor + 1}} \approx \frac{1}{2(x+1)^2}$$

e quella associata a δ è

$$\frac{1}{2^{\lfloor \log(\lfloor \log(x+1) \rfloor) + 1 \rfloor + 1 + \lfloor \log(x+1) \rfloor}} \approx \frac{1}{2(x+1)(\log(x+1)+1)^2}$$

Il codice di Golomb ha, naturalmente, una distribuzione dipendente dal modulo, che però è geometrica:

$$\frac{1}{2^{\lfloor x/k \rfloor + \log(k) + [x \mod (k) \geq 2^{\lceil \log(k) \rceil} - k]}} \approx \frac{1}{k\left(\sqrt[k]{2}\right)^x}$$

Infine, sebbene i codici a lunghezza variabile non siano completi, a meno di un fattore di normalizzazione è comunque possibile osservarne la distribuzione intesa:

$$\frac{1}{2^{\lceil \log(x)/k \rceil (k+1)}} \approx \frac{1}{(x+1)^{1+1/k}}$$

4.5 Struttura di un indice

Dai documenti crawlati si costruisce un indice inverso, cioè una relazione che mappa ogni termine nei documenti in cui compare. Partendo dalla lista di termini, costituita da elementi della forma $\langle documento, termine, posizione \rangle$, posso fare ordinamento rispetto alla colonna centrale e a partire da questo costruire l'indice inverso, ovvero l'indice che contiene, per ogni termine, l'insieme dei documenti (e delle posizioni nel documento) in cui il termine compare, in ordine crescente per valore del puntatore documentale.

Una lista di affissione è la lista dei puntatori a documenti associati a un unico termine, spesso, all'informazione essenziale dei puntatori documentali, sono associate una serie d'informazioni accessorie quali la frequenza con cui il token compare nei documenti associati e le posizioni occupate.

La struttura di un elemento di una lista di affissione potrebbe essere il seguente

$$\langle p, c, \langle p_1, \ldots, p_n \rangle \rangle$$

dove p è il puntatore al documento, c è il conteggio, ovvero il numero di volte in cui t compare in p, la n-pla ordinata (in ordine crescente) $\langle p_1, \ldots, p_n \rangle$ indica le posizioni che il token occupa nel file.

4.6 Codici per gli indici

Nella scelta dei codici istantanei da utilizzare per la compressione di un indice è fondamentale la conoscenza della distribuzione degli interi da comprimere. In alcuni casi è possibile creare un modello statistico che spiega la distribuzione empirica riscontrata.

I termini sono tipicamente salvati per ID (e.g. rango lessicografico), con un meccanismo per ricavarlo, come MWHC (che sarà trattato in seguito).

Per quanto riguarda frequenze e conteggi, la presenza significativa degli *hapax legomena* rende il codice γ quello usato più di frequente.

La questione degli scarti tra puntatori documentali è più complessa. Il modello *Bernoulliano* di distribuzione prevede che un termine con frequenza f compaia in una collezione di N documenti con probabilità $p = f/N$ indipendentemente in ogni documento. L'assunzione di indipendenza ha l'effetto di semplificare enormemente la distribuzione degli scarti, che risulta una geometrica di ragione p: lo scarto $x > 0$ compare cioè con probabilità $p(1-p)^{x-1}$.

Abbiamo già notato come il codice di Golomb abbia distribuzione intesa geometrica: si tratta quindi di trovare il modulo adatto a p, un risultato importante di teoria dei codici dice che il codice ottimo per una geometrica di ragione p è un Golomb di modulo

$$k = \left\lceil -\frac{\log(2-p)}{\log(1-p)} \right\rceil \tag{8}$$

Una controindicazione può essere però la *correlazione* tra documenti adiacenti. Per esempio, se i documenti della collezione provengono da un crawl e sono nell'ordine di visita, è molto probabile che documenti vicini contengano termini simili. Questo fatto sposta significativamente da una geometrica la distribuzione degli scarti.

Più spesso gli scarti tra puntatori documentali all'interno delle liste di affissione vengono memorizzati usando Elias-Fano (di parametro dipendente dal termine) o la γ di Elias.

Per quanto riguarda le posizioni, non esistono modelli noti e affidabili degli scarti, si usa pertanto un codice dalle buone prestazioni come il δ. È anche possibile utilizzare Golomb, assumendo un modello Bernoulliano anche per le posizioni, ma per calcolare il modulo è necessario avere la lunghezza del documento, che quindi deve essere disponibile in memoria centrale. Alla fine, in ogni caso, considerazioni come la facilità d'implementazione, velocità di decodifica, e altri fattori, possono essere forze determinanti nella scelta delle tecniche di codifica.

4.7 Problemi implementativi

I codici istantanei comprimono in maniera ottima rispetto alle distribuzioni intese. Riducono quindi la dimensione dell'indice in maniera significativa, cosa particolarmente utile se l'indice verrà caricato in tutto o in parte in memoria centrale. In effetti, esperimenti condotti all'inizio dell'attività di ricerca sugli indici inversi hanno mostrato che un indice compresso è significativamente più veloce di un indice non compresso, dato che l'I/O è ridotto in maniera significativa e il prezzo da pagare nella decodifica è di pochi cicli macchina per intero.

L'implementazione della lettura dei codici istantanei va però effettuata con una certa cura se si vogliono ottenere prestazioni ragionevoli. L'idea più importante è quella di mantenere un *bit buffer* che contiene una finestra sul file delle liste di affissione. Le manipolazioni relative alla decodifica dei codici dovrebbero essere confinate al bit buffer nella stragrande maggioranza dei casi. In particolare, un numero significativo di prefissi (per esempio, 2^{16}) può essere decodificato tramite una tabella che contiene, per ogni prefisso, il numero di bit immediatamente decodificabili e l'intero corrispondente, o l'indicazione che è necessario procedere a una decodifica manuale.

4.8 Salti

Non è possibile ottenere in tempo costante un elemento arbitrario di una lista compressa per scarti: è necessario decodificare gli elementi precedenti. In realtà, più problematica è l'impossibilità di saltare rapidamente al primo elemento della lista maggiore o uguale a un limite inferiore b, operazione detta comunemente *salto*. I salti sono fondamentali, come vedremo, per la risoluzione veloce delle interrogazioni congiunte.

La tecnica di base per ovviare a questo inconveniente è quella di memorizzare, insieme alla lista di affissioni, una *tabella di salto* che memorizza, dato un *quanto* q, il valore degli elementi d'indice $iq, i > 0$, e la loro posizione (espressa in bit) nella lista di affissioni. Quando si vuole effettuare un salto, e più precisamente quando si vuole trovare minimo valore maggiore o uguale a b, si cerca nella tabella (per esempio tramite una ricerca dicotomica o una *ricerca esponenziale*) l'elemento di posto $iq \leq b$, e si comincia a decodificare la lista per cercare il minimo maggiorante di b. Al più q elementi dovranno essere decodificati, e q deve essere scelto sperimentalmente in modo da, al tempo stesso, non accrescere eccessivamente la dimensione dell'indice e fornire un significativo aumento di prestazioni.

Si noti che una volta che è possibile saltare all'interno della lista dei puntatori ai documenti, è necessario mettere in piedi strutture di accesso per conteggio e posizioni, se presenti e memorizzate separatamente. Se cioè è possibile accedere in modo diretto ai puntatori documentali d'indice iq, deve essere possibile accedere allo stesso modo alle parti rimanenti dell'indice.

5 Gestione della lista dei termini

La lista dei termini di un indice può essere molto ingombrante. Esistono diversi metodi per rappresentarla: in questo contesto, ne vedremo uno piuttosto sofisticato, che ha il grosso vantaggio di essere in grado, data una lista di stringhe, di restituire il numero ordinale di qualunque elemento della lista, ma di *non memorizzare la lista stessa*: in effetti, l'occupazione di memoria sarà di $1,23n$ interi di $\log(n)$ bit, dove n è il numero delle stringhe.

Una funzione di hash per un insieme di chiavi $X \subseteq U$, dove U è l'universo di tutte le chiavi possibili, è una funzione $f : X \to m$ detta:

- *perfetta* se la funzione è iniettiva[14].
- *minimale* se $|X| = m$.
- *preserva l'ordine* se esiste un ordine lineare su X e si ha $x \leq y \iff f(x) \leq f(y)$.

Il nostro scopo è quello di costruire, per un insieme di stringhe arbitrario, una funzione di hash minimale, perfetta, e che preservi l'ordine. (si noti che di per sè non stiamo chiedendo qual è il risultato di f su $U \setminus X$: questo problema verrà risolto a parte).

Esistono numerose tecniche per la costruzione di hash perfetti [eBSM97].

Quello che andiamo a descrivere è basato sulla teoria dei grafi casuali, e utilizza la randomizzazione per la costruzione della struttura dati. Quella che descriveremo è in effetti una tecnica completamente generale per memorizzare una funzione statica (immutabile) $f : X \to 2^b$ dall'insieme X all'insieme dei valori 2^b. Scegliendo $b = \lceil \log(n) \rceil$ e mappando ogni termine in un indice al suo rango lessicografico, otterremo il risultato richiesto.

L'idea della costruzione è la seguente: prendiamo due funzioni di hash aleatorie $h : U \to m$ e $g : U \to m$, dove $m \geq n$ e n è la cardinalità di X, e consideriamo un vettore $\mathbf{w}$ di interi a b bit che sia soluzione del sistema

$$w_{h(x)} \oplus w_{g(x)} = f(x) \qquad x \in X$$

Chiaramente, memorizzando h, g e il vettore $\mathbf{w}$ abbiamo memorizzato f: per calcolare $f(x)$ basta semplicemente risolvere il sistema, che è di n equazioni con w_i incognite, che, però, potrebbe non avere soluzione. Il sistema deve essere risolto una sola volta, in fase di inizializzazione della struttura, e poi potremo accedere a tutti i valori associati alla funzione $f(x)$ in tempo lineare nel numero di funzioni di hash (calcolo le funzioni di hash, accedo alle posizioni del vettore e calcolo lo XOR dei valori). Possiamo rappresentare i vincoli imposti dal sistema come segue: costruiamo un grafo G non orientato con m vertici e n lati in cui, per ogni $x \in X$, abbiamo che $h(x)$ è adiacente a $g(x)$ tramite un lato etichettato da x.

Andiamo ora a eseguire l'operazione di *esfoliazione* dell grafo così costruito: partendo da una qualunque foglia v (un vertice di grado 0 o 1), la rimuoviamo dal grafo. Se questa è adiacente a un altro vertice, rimuoviamo anche il lato l che li connette, a questo punto mettiamo la coppia $\langle v, l \rangle$ dentro uno stack.

Il processo continua fino a quando il grafo non è vuoto o non si incontra un ciclo. Se non incontriamo alcun ciclo, significa che il sistema può essere risolto, questo perché ogni equazione del sistema deve comparire una sola volta nello stack, altrimenti il processo si conclude quando rimuoviamo gli ultimi nodi, che avranno degree zero, e di conseguenza il loro valore w_i sarà pari a 0. Se risolviamo l'equazione associata a ogni elemento $\langle v, l \rangle$ sfruttando il fatto che $v = h(x) \vee v = g(x)$, e v non è stato certamente assegnato precedentemente, dato che ogni vertice in una coppia non compare mai in coppie successive. Essenzialmente stiamo *triangolando* la matrice associata al sistema (si vedano i lucidi animati online).

Ora, assumendo che h e g siano casuali e indipendenti, il grafo che andiamo a costruire è un grafo casuale di m vertici con n archi, e un risultato importante di teoria dei grafi casuali dice che per n sufficientemente grande, quando $m > 2.09n$ il grafo è quasi sempre privo di cicli (questo

[14] Ricordo: una funzione è iniettiva se per ogni elemento nel codominio esiste al più un elemento nel dominio, tale che xRy. In questo caso significa che la funzione di hash non presenta collisioni.

significa che il rapporto tra il numero di grafi privi cicli e quelli totali tende a 1). In sostanza, scegliendo bene h e g quasi tutti i grafi che otterremo permetteranno di risolvere il sistema.

Il caso ora descritto non è in realtà ottimo. La teoria degli ipergrafi casuali ci dice che utilizzando 3-ipergrafi (cioè un insieme di sottoinsiemi di ordine 3 dell'insieme dei vertici) è possibile dare una nozione di aciclicità che permette di risolvere i sistemi nel caso di *tre* funzioni di hash, ma in questo caso il limite che garantisce l'aciclicità è $m > 1.23n$ - un miglioramento netto rispetto al caso di ordine 2. Alla fine, per memorizzare la funzione statica $f : X \to 2^b$ dovremo utilizzare solo 1.23 bit per elemento.

5.1 Firme

La funzione così costruita, per quanto sia minimale, perfetta e preservi l'ordine non permette di riconoscere se un elemento fa parte di X o no. Per ovviare all'inconveniente utilizziamo un insieme di *firme* associato all'insieme X delle chiavi. Consideriamo cioè una funzione $s : U \to 2^r$ che associ a ogni chiave possibile una sequenza di r bit "casuale", nel senso che la probabilità che $s(x) = s(y)$ se x e y sono presi uniformemente a caso da U è 2^{-r} (per esempio una buona funzione di hash). Consideriamo l'enumerazione ordinata $x_0 \leq x_1 \leq \cdots \leq x_{n-1}$ degli elementi di X. Oltre a f, memorizziamo un vettore di dimensione n in cui i valori di X sono mappati nelle posizioni $f(x)$, che è il rango lessicografico di x, affinché, preso $x \in X$, calcolati $f(x)$ ed $s(x)$ avrò che

$$S_{f(x)} = s(x)$$

mentre se $x \notin X$ il valore di $f(x)$ sarà arbitrario, quindi $S_{f(x)} = -1$ e il valore di $s(x)$ sarà a sua volta arbitrario.

Per interrogare la struttura risultante su input $x \in U$, agiamo come segue:

1. Calcoliamo $f(x)$ (che è un numero in n).
2. Recuperiamo $S_{f(x)}$.
3. Se $S_{f(x)} = s(x)$, restituiamo $f(x)$; altrimenti restituiamo -1 (a indicare che x non fa parte di X).

Si noti che se $x \in X$, $f(x)$ è il suo rango in X, e quindi $S_{f(x)}$ conterrà, per definizione, $s(x)$. Se invece $x \notin X$, $S_{f(x)}$ sarà una firma arbitraria, e quindi restituiremo quasi sempre -1, tranne nel caso ci sia una collisione tra firme, il che avviene, come detto, con probabilità 2^{-r}. Tarando il numero r di bit delle firme, è quindi possibile bilanciare spazio occupato e precisione della struttura.

5.2 Ottimizzazioni

È possibile migliorare ulteriormente l'occupazione in spazio, quando b non è troppo piccolo, utilizzando un *vettore compatto* per memorizzare i valori della soluzione del sistema. In effetti, per come abbiamo descritto il processo di risoluzione, non è possibile che vengano poste a valori diversi da 0 più di n variabili. Se potessimo memorizzare, con una piccola perdita di spazio, *solo i valori diversi da zero* potremmo ridurre l'occupazione di memoria a quasi nb bit.

Per farlo consideriamo un vettore di bit $\mathbf{b}$ che contiene in posizione i un uno se la variabile corrispondente w_i è diversa da zero, e zero altrimenti. Definiamo l'operazione *rank*

$$rank(\mathbf{b}, i) = |\{j < i \mid b_j \neq 0\}|$$

che conta il numero di uno che compaiono a sinistra della posizione corrente. Chiaramente se mettiamo in un vettore $\mathbf{c}$ i valori delle variabili w_i diversi da zero avremo che:

$$w_i = \begin{cases} 0 & \text{se } b_i = 0 \\ \mathbf{c}[rank(\mathbf{b}, i)] & \text{se } b_i \neq 0 \end{cases}$$

Quanto spazio occorre per calcolare rapidamente il rango? Esistono soluzioni molto sofisticate che occupano spazio $o(n)$[15], ma qui proporremo un metodo molto semplice: scelto un quanto q, memorizziamo in una tabella R i valori $rank(\mathbf{b}, kq)$. Per calcolare il rango in posizione p, basta recuperare $R[\lfloor p/q \rfloor]$ e completare il calcolo contando gli uno dalla posizione $q \lfloor p/q \rfloor$ alla posizione p (utilizzando le istruzioni di conteggio efficiente per le CPU moderne). Per q fissato, il tempo del calcolo è costante. Lo spazio occupato dipende anch'esso da q, che permette quindi di bilanciare lo spazio occupato e la velocità.

Alla fine, la funzione occuperà $nb + 1.23n$ bit, più il necessario per la struttura di rango (supponendo $q = 512$ saranno necessari altri $0.125n$ bit). Si noti che è impossibile scrivere una funzione in meno di n^2 bit, dato che esistono $(2^b)^n$ funzioni da un insieme di n a un insieme di 2^b elementi.

5.3 Auto-firma

Un utilizzo interessante delle funzioni che abbiamo descritto è quello di memorizzare efficientemente un dizionario statico approssimato. Per farlo, consideriamo un insieme $X \subseteq U$ che vogliamo rappresentare, e una funzione di firma $s : U \to 2^r$. Possiamo utilizzare la tecnica generale di rappresentazione delle funzioni per scrivere in $nr + 1.23n$ bit la funzione $f : X \to 2^r$ data da $f(x) = s(x)$. La funzione f, cioè, mappa ogni chiave nella propria firma. Il dizionario viene a questo punto interrogato come segue:

1. Calcoliamo $f(x)$
2. Se $f(x) = s(x)$, restituiamo "appartiene a X"; altrimenti, restituiamo "non appartiene a X".

Si noti che, se $x \in X$, $f(x)$ è la sua firma $s(x)$, e quindi restituiremo "appartiene a X". Se invece $x \notin X$, $f(x)$ sarà un valore arbitrario, e quindi restituiremo "non appartiene a X", tranne nel caso in cui ci sia una collisione tra firme, il che avviene con probabilità 2^{-r}. Tarando il numero r di bit delle firme è quindi possibile bilanciare spazio occupato e precisione della struttura.

Rispetto a un filtro di Bloom, un dizionario di questo tipo è molto più compatto (un filtro di Bloom, per avere una precisione di 2^{-r} richiede 1.44 bit per elemento) e molto più veloce (le funzioni sono valutabili con al più tre fallimenti di cache, contro gli r del filtro di Bloom). Per contro, non è modificabile.

[15] Un esempio non banale di strutture efficienti per la memorizzazione del rango è la struttura di Jacobson.

6 Risoluzione delle interrogazioni

La risoluzione di un'interrogazione dipende fondamentalmente da due fattori:

- Il linguaggio di interrogazione e la corrispondente semantica.
- Il tipo di dati contenuti nelle affissioni dell'indice.

Per il momento, ci concentreremo sulla risoluzione di interrogazioni *booleane*, cioè in cui gli unici operatori sono la congiunzione, la disgiunzione e la negazione logica. Una formula del linguaggio di interrogazione è: un termine oppure la congiunzione, disgiunzione o negazione di formule.

Per quanto riguarda la semantica:

- La semantica di una formula è una lista di documenti.
- La semantica di un termine è data dalla lista di documenti in cui il termine compare.
- La semantica della congiunzione è data dall'intersezione delle liste.
- La semantica della disgiunzione è data dalla fusione delle liste.
- La semantica della negazione è data dalla complementazione rispetto all'intera collezione documentale.

Chiaramente, per risolvere un'interrogazione Booleana è sufficiente che le affissioni contengano i puntatori documentali. Inoltre, se l'indice contiene i puntatori in ordine crescente, è possibile ottenere intersezione e unione di liste in tempo lineare nel numero di puntatori.

È di grande interesse operare in maniera *pigra* (document-at-a-time) - leggere cioè dalle liste in input solo i puntatori necessari a emettere un certo prefisso dell'output. Questo permette di gestire in maniera efficiente la *early termination*, tecnica con cui il calcolo della semantica viene arrestato sulla base di stime della qualità dei documenti già rinvenuti.

Per fondere liste in tempo lineare (con perdita logaritmica nel numero di liste in ingresso) è sufficiente una *coda di priorità indiretta* che contiene puntatori alle liste, prioritizzate secondo il documento corrente. A ogni passo, il prossimo documento restituito sarà il documento corrente della lista in cima alla coda (cioè quella che correntemente è posizionata sul documento più piccolo). Una volta deciso il documento, avanziamo la lista in cima alla coda e aggiustiamo la coda stessa finchè la lista in cima non è posizionata su un documento diverso. Si noti che in questo modo siamo anche in grado di sapere *quali* liste contengono il documento corrente.

Per quanto riguarda l'intersezione, sebbene in linea di principio il limite inferiore lineare non sia superabile, esistono diverse euristiche che permettono di accelerare la computazione nel caso l'indice fornisca la possibilità di effettuare *salti*.

In linea di principio, potremmo sempre tenere conto tramite una coda di priorità indiretta del documento minimo corrente, come nel caso precedente, ma tenere al tempo stesso traccia del massimo puntatore in una variabile[16]. Quando coincidono, il puntatore sta nell'intersezione e può essere restituito (e immediatamente dopo, una qualunque lista viene fatta avanzare). Altrimenti la lista che realizza il minimo viene fatta saltare fino al massimo e la coda viene aggiornata. Dato che viene effettuato al più un aggiornamento per avanzamento, il tempo richiesto è lineare nella dimensione di input, con una perdita logaritmica nel numero di input.

Da un punto di vista pratico, però, è spesso più efficiente utilizzare una soluzione che ha in linea teorica un comportamento molto peggiore (la perdita nel numero di input è *lineare*). L'idea è di far avanzare in maniera miope le liste in ordine fino al massimo corrente m (all'inizio, il primo puntatore della prima lista) finchè non sono tutte allineate. Al primo disallineamento (che causa un incremento di m) si ricomincia ad allineare la prima lista. Quando si arriva ad allineare l'ultima lista, si ha un puntatore da restituire.

[16] Supponendo che la coda a priorità indiretta sia implementata tramite uno Heap (per esempio) recuperare il massimo puntatore documentale richiede tempo e spazio costante perché il valore più a destra nella struttura è sempre il massimo corrente.

L'aspetto interessante di questo approccio è che ordinando le liste in ordine di frequenza crescente (quando è nota - per esempio nel caso di termini) la maggior parte degli avanzamenti verrà giocato dalle prime liste, che essendo quelle di minima frequenza genereranno pochi allineamenti. Le liste più dense subiranno pochi avanzamenti molto consistenti, che potranno essere gestiti in maniera efficienti tramite un sistema di salti. È anche possibile implementare una versione adattiva che mano a mano che scandisce le liste in input (non necessariamente liste di termini) ne stima la frequenza e le riordina al volo.

Si noti che nel caso le liste siano interamente caricabili in memoria, il problema diventa completamente diverso, ed esistono soluzioni molto sofisticate per il problema dell'intersezione [eJIM00].

L'osservazione fondamentale è che intersecare due liste, se una è molto più corta dell'altra (per esempio esponenzialmente più corta), può essere conveniente cercare con una ricerca dicotomica gli elementi della lista più corta in quella più lunga.

Questa osservazione permette di risolvere rapidamente in memoria centrale, in maniera non pigra (term-at-a-time), intersezioni di liste. È sufficiente ordinare le liste in ordine di frequenza (questo è sempre possibile, essendo l'algoritmo non pigro) e procedere a ridurre, una lista alla volta, ma la speranza è che quando arriveremo alle liste più lunghe (le ultime) i candidati siano così pochi che sia possibile utilizzare tecniche di intersezione come quelle summenzionate.

6.1 Indici distribuiti

Le dimensioni degli indici del web sono tali da rendere poco pratica la memorizzazione dell'intero indice in una sola macchina. È quindi necessario *segmentare* l'indice complessivo in sottoindici, detti *segmenti*, le cui risposte verranno poi opportunamente combinate. È sottointeso che i segmenti saranno in genere memorizzati su un gruppo di macchine collegate in rete, e che un opportuno protocollo di comunicazione permetterà di accedere al contenuto dei segmenti in una macchina remota.

In linea di principio, un indice può essere partizionato tramite una funzione che assegna a ogni affissione un segmento destinazione. A ogni segmento saranno associati i termini per i quali compare almeno un'affissione.

In pratica però è più comune scegliere un singolo criterio di partizionamento - *lessicale* o *documentale* - su cui basare il processo. In un partizionamento documentale, la collezione documentale viene divisa in blocchi (non necessariamente contigui) e a ogni blocco viene assegnato un segmento. Tutte le affissioni relative a un blocco andranno a finire nel suo segmento, e al segmento saranno associati i termini che compaiono nel blocco. Di norma, lo stesso termine comparirà in più segmenti (si pensi a termini comuni come le congiunzioni), mentre gli *hapax legomena* compariranno esattamente in un solo segmento.

In alternativa, è possibile partizionare la collezione documentale lessicalmente: viene scelto un criterio per dividere l'insieme dei termini in più blocchi (non necessariamente lessicograficamente contigui), e a ogni blocco viene associato un segmento: la lista di affissioni relativa a un termine sarà interamente contenuta nel segmento relativo.

La ricostruzione delle liste di affissione a partire da una segmentazione lessicale è banale: basta individuare il segmento che contiene il termine e interrogarlo. L'individuazione del segmento può non essere un problema banale se, ad esempio, i blocchi di termini non sono lessicograficamente consecutivi. Nel caso peggiore, l'unica soluzione è l'interrogazione di tutti i segmenti.

La ricostruzione delle liste di affissione a partire da una segmentazione documentale è più delicata, dato che nella maggior parte dei casi sarà necessario leggere vari frammenti della lista complessiva e combinarli. Questo richiede *in primis* di interrogare tutti i segmenti per sapere quali possiedono una lista di affissione per il termine dato. Le liste vanno poi combinate - con una semplice concatenazione e rinumerazione se i blocchi di documenti sono consecutivi, o con una fusione di liste in caso contrario.

In entrambi i casi precedenti può essere utile filtrare le richieste ai segmenti tramite dei dizionari approssimati, che possono rappresentare in maniera molto compatta l'insieme presente in ogni

segmento. Dimensionando i dizionari a seconda della memoria disponibile è possibile ridurre il numero di richieste inutili fatte ai segmenti.

Va notata una caratteristica importante del partizionamento documentale: è possibile cioè risolvere *direttamente* un'interrogazione complessa a livello di segmento. Questo fatto può portare a un miglioramento netto delle prestazioni, perché ad esempio, in presenza di un'interrogazione congiunta di termini, alcuni segmenti, pur contenendo tutti i termini dell'interrogazione, potrebbero non restituire nessun documento. Nel caso di un'interrogazione disgiunta, non c'è in ogni caso riduzione delle prestazioni. Va però fatto notare che se il processo di risoluzione della query restituisce artefatti (quali valori di ranking, calcolo della prossimità, etc…) il protocollo di rete con cui ci si connette all'indice dovrà essere in grado di trasmetterli. Detto altrimenti, risolvendo e trasmettendo solo liste di affissioni, il protocollo dipenderà solo dalla struttura dell'indice, mentre la risposta a interrogazioni con punteggio lo renderà dipendente dal meccanismo di assegnamento del punteggio.

D'altra parte, il partizionamento lessicale offre un'interessante possibilità: quella cioè di tenere in memoria centrale la parte dell'indice corrispondente ai termini più interessanti in qualche senso, o che risultano più utilizzati da una rilevazione empirica.

7 Centralità

La centralità è un argomento fondamentale nel campo dell'analisi delle reti. Partendo da un grafo G, vogliamo associare a ogni nodo un *punteggio* che definisce l'importanza del nodo in un grafo. Immaginiamo che i nodi corrispondano a qualche tipo di entità, e che un arco da x in y rappresenti una *relazione* tra l'entità x e l'entità y. Questa relazione può misurare l'appoggio, ad esempio "x crede in y tanto così", o il punteggio di x contro y, come nel caso di "x batte y". Il primo caso è comune nel campo della psicometria e della sociometria, mentre il secondo è comune nel caso dei tornei.

Molti *indici di centralità* sono basati su semplici somme sulle righe o le colonne delle matrici di adiacenza e sono comuni da molto tempo nel campo della psicometria e della sociometria. Un esempio banale, consiste nell'utilizzare il *degree* (indegree od outdegree) come misura di centralità. Il problema principale con il degree è che è una misura interamente locale - non tiene conto della sua topologia.

Più in generale, se si considerano *grafi pesati*, in cui un peso è associato a ogni arco, è possibile considerare il grafo come la matrice di adiacenza M che gli è associata.

7.1 Misure di centralità geometriche

Chiamiamo *geometriche* quelle misure che vedono la centralità come dipendente dalla distanza di un nodo o un arco dagli altri elementi del grafo. Più precisamente, una centralità geometrica dipende unicamente da quanti nodi esistono a ogni distanza. Queste sono alcune delle misure più antiche definite in letteratura.

7.1.1 Indegree

L'indegree è il numero di archi che entrano in un determinato nodo, è denotato come $d_+^{(x)}$ e può essere considerata una misura di centralità geometrica. Si tratta semplicemente del numero di nodi a distanza 1[17]. L'indegree è probabilmente una delle più vecchie misure di centralità, in quanto è equivalente alla maggioranza dei votanti alle elezioni ($x \to y$ se x ha votato per y), il vincitore delle elezioni è quello che ha preso più voti, quindi quello che ha l'indegree più alto.

L'indegree ha una serie di problematiche evidenti (e.g. è facile da aggirare), ma è un buon punto di partenza, e in alcuni casi ha raggiunto risultati migliori di tecniche più sofisticate (si veda, ad esempio [Ups03a]).

7.1.2 Closeness

Bavelas ha introdotto il concetto di closeness alla fine degli anni quaranta in [Bav50]; la closeness di x è definita come segue

$$\frac{1}{\sum_y d(y,x)} \tag{9}$$

L'intuizione che sta dietro al concetto di closeness è che nodi più centrali dovranno avere distanze più piccole da tutti gli altri nodi, avendo pertanto un denominatore più piccolo e quindi una centralità più grande. È importante notare che, affinchè la notazione abbia senso, il grafo deve essere fortemente connesso. In mancanza di questa condizione, alcuni dei denominatori andranno a infinito, risultando in una definizione di closeness nulla per tutti quei nodi che non possono raggiungere tutti i nodi del grafo.

[17] La maggioranza delle misure di centralità proposte in letteratura sono state descritte solo per grafi non direzionati e connessi. Siccome lo studio del grafo del web e delle reti sociali elettronicamente mediate ha posto il problema di estendere i concetti di centralità a reti che sono direzionate, e possibilmente non fortemente connesse, nel resto di questo articolo considereremo le misure dipendenti dal numero di archi entranti di un nodo (e. g. percorsi entranti, autovettori dominanti sinistri, distanze da tutti i nodi verso un nodo specifico). Se necessario, queste misure possono essere chiamate "positive", in opposizione alle misure "negative" ottenute considerando percorsi uscenti, o (equivalentemente trasponendo il grafo)

Non era probabilmente nelle intenzioni di Bavelas di applicare la misura ai grafi direzionati, e ancora meno di applicarla a grafi con distanze infinite, ma spesso la closeness viene patchata rimuovendo i nodi che non possono essere raggiunti nel modo seguente

$$\frac{1}{\sum_{d(y,x)<\infty} d(y,x)}$$

e assumendo che i nodi con un insieme di nodi raggiungibili banale abbiano centralità 0 per definizione: questa è, in realtà, la definizione che useremo nel resto dell'articolo. Questi aggiustamenti, apparentemente innocui, inducono una forte polarizzazione nella rete verso quei nodi che hanno un insieme di nodi raggiungibili di piccole dimensioni. Nel 2016 un gruppo di scienziati a Facebook ha calcolato l'inverso della closeness per tutti gli utenti[18]

7.1.3 Centralità armonica

Come abbiamo notato il problema della closeness sta nel fatto che vi sono coppie di nodi non raggiungibili. Prendiamo quindi ispirazione da Marchiori e Latora [Lat00]: presentatogli il problema di offrire una nozione sensata di "lunghezza media del percorso più breve" per un grafo direzionato generico, hanno proposto di rimpiazzare la media delle distanze con la *media armonica di tutte le distanze*[19].

In generale, per ogni nozione teorica sui grafi basata sulla media aritmetica o la massimizzazione v'è una definizione alternativa basata sulla variante armonica dell'operatore. Se consideriamo la closeness il reciproco di una media denormalizzata delle distanze, è naturale considerare anche una media armonica denormalizzata delle distanze. Definiamo dunque la *centralità armonica* di x come segue

$$\sum_{y\neq x} \frac{1}{d(y,x)} = \sum_{d(y,x)<\infty, y\neq x} \frac{1}{d(y,x)} \tag{10}$$

la differenza tra quest'ultima equazione e (7) potrebbe apparire minima, però si tratta di un cambiamento radicale. La centralità armonica è fortemente legata alla closeness in reti semplici, ma la prima è in grado di tenere da conto automaticamente anche dei nodi y che non sono in grado di raggiungere x. Quindi può essere applicata a grafi che non sono fortemente connessi.

7.1.4 Betweenness

La *betweenness* è stata introdotta da Anthonisse [Ant71] per gli archi, ed è stata poi ripresa da Freeman per i nodi [Fre77]. L'idea è quella di misurare la probabilità che un cammino brave a caso passi attraverso un determinato nodo: sia σ_{xy} il numero di cammini brevi che vanno da x a y e $\sigma_{xy}(z)$ il numero di cammini brevi che vanno da x a y e passano per z, definiamo la *betweenness* di x come

$$\sum_{x,y\neq z, \sigma_{xy}\neq 0} \frac{\sigma_{xy}(z)}{\sigma_{xy}}$$

L'intuizione che sta dietro la betweenness, è che se una grande parte dei cammini minimi passa da z, allora z deve essere un punto di articolazione importante all'interno del grafo.

7.2 Misure di centralità spettrale

Verso la metà del XIX secolo i risultati di un torneo di scacchi furono rappresentati tramite una matrice M popolata di 0, 1 e 1/2, rappresentando sconfitta, vittoria e pareggio. La diagonale di

[18] `https://research.fb.com/blog/2016/02/three-and-a-half-degrees-of-separation/`

[19] Definiamo media armonica per un gruppo di valori $a_0, \ldots, a_{n-1}$ il seguente valore

$$\left(\frac{\sum_i^n a_i^{-1}}{n}\right)^{-1}$$

tali matrici doveva per forza essere zero, e la somma di entry simmetriche avrebbe fatto sempre 1. Le somme sulle righe, ovvero $M\mathbf{1}^T$, erano un modo facile di fornire un punteggio globale per un giocatore, dal quale il suo rank poteva essere calcolato, in modo tale da poter assegnare premi o dividere i soldi in maniera proporzionale.

Per incrementare la precisione del sistema, lo scacchista austriaco Oscar Gelbfuhs propose nel 1873 di iterare la procedura, ovvero di calcolare $M^2\mathbf{1}^T$, raffinando i punteggi precedenti. Essenzialmente, partendo da un punteggio iniziale di 1 dato a tutti i giocatori, ciascuno avrebbe ottenuto un nuovo punteggio, risultato della somma di:

- metà dei punteggi dei giocatori con cui avessero pareggiato.
- L'interezza dei punteggi di coloro con cui avessero vinto.

La procedura sarebbe stata ripetuta per ottenere i punteggi dei turni successivi.

Edmund Landau [Lan95] notò nel primo articolo che pubblicò [Sch69] che se uno considera il processo iterato proposto da Gelbfuhs, il risultato non è attendibile perché il valore oscilla e quindi la classifica dei giocatori cambia sulla base del numero di iterazioni k. Egli propose dunque di calcolare un vettore di punteggio $\mathbf{r}$ che soddisfacesse la seguente equazione

$$M\mathbf{r} = \lambda\mathbf{r} \tag{11}$$

dunque $\mathbf{r}$ sarebbe dovuto essere un autovettore destro[20], la cui direzione e verso sono fissati durante la procedura, mentre il modulo viene moltiplicato per una costante (e quindi è soggetto a dilatazioni). Questa è stata la prima apparizione del *ranking spettrale* - che usa gli autovettori di una qualche matrice, solitamente derivata da un grafo, per calcolare una misura di centralità.

7.2.1 L'autovettore dominante sinistro

Un *autovalore dominante* è un autovalore di modulo massimo. Un autovettore associato all'autovalore dominante è noto come *autovettore dominante*. Nella maggior parte dei casi pratici di ranking spettrale l'autovalore di modulo massimo è unico[21].

Il teorema di Perron Frobenius ci dà qualche garanzia a riguardo.

Teorema 3. *Se A è una matrice irriducibile[22], quindi il grafo associato ad A è fortemente connesso, esiste un unico autovalore di modulo massimo, detto autovalore di Perron Frobenius, o dominante, a cui è associato un autovettore positivo. Se la matrice è stocastica (ha quindi somma di ogni riga pari a 1 $\implies$ è associata a un grafo i cui pesi sono normalizzati) l'autovalore dominante ha modulo 1.*

La prima ovvia misura spettrale è l'autovettore dominante sinistro della matrice di adiacenza associata al grafo. L'idea realizza l'intuizione di Landau: assumiamo $\mathbf{e}_i$ sia una base di autovettori per la matrice M con autovalori λ_i tali che $|\lambda_0| > |\lambda_1| > \dots$. Se consideriamo un vettore $\mathbf{x}$ e lo vediamo come combinazione lineare dei vettori dell'autobase

$$\mathbf{x} = \sum_i \alpha_i \mathbf{e}_i$$

possiamo scrivere $x_k = M^k\mathbf{x}$ come segue

$$M^k\mathbf{x} = M^k \sum_i \alpha_i \mathbf{e}_i = \sum_i \alpha_i \lambda_i^k \mathbf{e}_i$$

posso portare la potenza k-esima della matrice dentro la sommatoria perché è indipendente dalla variabile di somma, inoltre:

$$M\mathbf{v} = \lambda\mathbf{v}$$

[20] Importante puntiglio di notazione, considereremo l'autovettore destro come il vettore colonna di N elementi.
[21] Questo significa che la soluzione del polinomi caratteristico ha molteplicità 1
[22] Una matrice $M \in \mathbb{R}^{k\times k}$ si dice irriducibile quando $\forall\, i,j \in k, \exists n \in \mathbb{N} \mid a_{i,j}^n > 0$

continuando

$$\sum_i \alpha_i \lambda_i^k \mathbf{e}_i = \lambda_0^k \left(\alpha_0 \mathbf{e}_0 + \sum_{i>0} \alpha_i \left(\frac{\lambda_i}{\lambda_0} \right)^k \mathbf{e}_i \right)$$

È chiaro che per $k \to \infty$ tutti i termini $\frac{\lambda_i}{\lambda_0}$ tenderanno a 0, fintanto che $\alpha_0 \neq 0$ il risultato dell'iterazione tenderà verso l'autovettore dominante $\mathbf{e}_0$:

$$\frac{M^k \mathbf{x}}{\|M^k \mathbf{x}\|} \to \mathbf{e}_0$$

Abbiamo dunque provato che il *metodo della potenza* per calcolare l'autovettore dominante funziona sotto le ipotesi fatte in precedenza.

Gli autovettori dominanti non si comportano come ci aspetteremmo quando il grafo in questione non è fortemente connesso. In base all'autovalore dominante della componente fortemente connessa, l'autovettore dominante potrebbe o potrebbe non essere diverso da zero per componenti terminali (un discorso più completo può essere trovato in [Ple94]). Un esempio recente di applicazione di centralità basata su autovettori dominanti può essere trovato in [Sl19].

7.2.2 Indice di Seeley

Un altro passo fondamentale verso il ranking spettrale fu fatto mezzo secolo dopo da John R. Seeley [See49] , il quale non era al corrente del lavoro fatto da Landau: notò che gli indici basati su somme di righe o colonne non avevano chissà quale significato a causa del fatto che non prendevano in considerazione l'importante fatto che bisogna piacere a chi piace a molti, e così via. In altre parole, un indice di importanza, di centralità o di autorità, dovrebbe essere definito in maniera *ricorsiva*, così che il mio punteggio sia uguale alla somma dei punteggi di tutti quelli che mi sostengono. In notazione matriciale

$$\mathbf{r} = \mathbf{r}M$$

Ovviamente, ciò non è sempre possibile. A ogni modo, Seeley, considera una matrice non negativa senza entry nulle e normalizza le righe così che abbiamo norma l_1 unitaria (e.g. dividi ogni entry per la somma sulla riga cui appartiene); le sue righe hanno sempre entry diverse da 0, quindi è sempre possibile, e l'equazione indicata sopra ha soluzione, questo perché $M\mathbf{1}^T = \mathbf{1}^T$. Dunque 1 è autovalore di M, e il suo autovettore(i) **sinistro** fornisce soluzioni per l'equazione. L'unicità è una questione più complicata che Seeley non discute e che può essere facilmente analizzata impiegando la teoria di Perron-Frobenius, che mostra anche che 1 è il raggio spettrale, quindi $\mathbf{r}$ è un autovettore dominante, e vi sono soluzioni positive[23].

L'indice di Seeley può essere espresso tramite la seguente equazione

$$s(x) = \sum_y \frac{s(y)}{d_+(y)} \tag{12}$$

È interessante notare che le motivazioni dei due lavori citati (Landau e Seeley) sono profondamente diverse: il primo era interessato al limite di un processo iterato per migliorare un punteggio, e definisce il limite ricorsivamente; Seeley vuole definire direttamente un punteggio ricorsivo.

La matrice risultante dal processo di l_1-normalizzazione è stocastica, quindi il punteggio può essere interpretato come uno stato stabile di una catena di Markov[24]. In particolare, se il grafo sottostante è simmetrico allora l'indice di Seeley collassa al degree (normalizzato) a causa del noto carattere della distribuzione stazionaria nel caso di una camminata casuale all'interno di un grafo simmetrico. Ripetendo il processo fino a stabilità otteniamo l'autovettore sinistro della matrice del grafo normalizzata. Anche l'indice di Seeley non gestisce molto bene la mancanza di connettività forte: gli unici nodi con un punteggio non zero sono quelli che appartengono a componenti terminali che non sono formate da un singolo nodo di outdegree 0.

[23] In realtà, Seeley espone l'intera questione in termini di equazioni lineari. Il calcolo matriciale viene impiegato solamente per risolvere un sistema lineare tramite la regola di Cramer

[24] Il punteggio può essere interpretato come il tempo che spenderemo in un vertice durante un processo di visita casuale ed equiprobaile in un grafo

7.2.3 Indice di Katz

Katz ha introdotto il suo celebre indice [Kat53] usando la somma su tutti i percorsi entranti in un nodo, ma pesando ciascun cammino di modo che la somma avesse un valore finito. L'indice di Katz può essere espresso come

$$\mathbf{k} = \mathbf{1} \sum_{i=0}^{\infty} \beta^i A^i \tag{13}$$

grazie alle interazioni tra le potenze della matrice di adiacenza e il numero di cammini che collegano due nodi. Perché la somma sia positiva è necessario che il fattore di attenuazione β sia di valore minore di $1/\lambda_0$, dove λ_0 è autovalore dominante di A.

Katz notò immediatamente che l'indice poteva essere espresso usando operatori classici dell'algebra lineare:

$$\mathbf{k} = \mathbf{1}(1 - \beta A)^{-1} \tag{14}$$

una semplice generalizzazione (consigliata da Hubbell in [Hub65]) rimpiazza il vettore $\mathbf{1}$ con un vettore di preferenze $\mathbf{v}$ affinchè i cammini vengano pesati in maniera differente sulla base del nodo di partenza.

7.3 PageRank

PageRank è una delle centralità spettarli oggi in uso più discusse, sembrerebbe essere usata all'interno dell'algoritmo di ranking di Google[25].

Di fatto quando lavoriamo con PageRank è come se avessimo a disposizione una moneta truccata ($p_T > p_C$) e continuassimo ripetutamente a fare dei lanci, se esce testa, si continua la visita corrente, altrimenti sir ricomincia da un'altra parte (eseguendo la cosiddetta operazione di teletrasporto). La cosa interessante è che se il processo di visita si blocca in una componente connessa basta attendere alcuni try e prima o poi il processo aleatorio è in grado di tirarsene fuori da solo. Questo concetto dovrebbe far drizzare le antenne a coloro che hanno seguito corsi riguardanti processi aleatori ripetuti (e.g. Catene di Markov) o su algoritmi euristici (e.g. algoritmo di Simulated Annealing che cerca di sfuggire da un minimo locale) Per definizione PageRank è l'unico vettore $\mathbf{p}$ (autovettore dominante) soddisfacente

$$\mathbf{p} = \alpha \mathbf{p} \overline{A} + (1 - \alpha)\mathbf{v} \tag{15}$$

Dove $\overline{A}$ è ancora la matrice di adiacenza del grafo l_1-normalizzata, $\alpha \in [0 \dots 1)$ è un *damping factor*, e rappresenta la probabilità di teletrasportarsi in una pagina qualsiasi come quello visto per l'indice di Katz, $\mathbf{v}$ è un vettore di preferenza, che deve essere una distribuzione, e rappresenta le pagine disponibili per eseguire il teletrasporto (può essere regolato in base alle esigenze). Questa definizione appare nel paper di Brin e Page su Google [Pag98]; gli autori affermano che il punteggio ricavato con PageRank è una distribuzione di probabilità sulle pagine del web, ciò significa che ha norma l_1 unitaria, ma non è necessariamente vero se A ha righe nulle. Successivi articoli scientifici hanno provato a patchare la matrice $\overline{A}$ per renderla stocastica, il che avrebbe garantito $\|\mathbf{p}\|_1 = 1$. Una soluzione comune è quella di andare a rimpiazzare ogni riga nulla con il vettore di preferenza $\mathbf{v}$ stesso, ma altre soluzioni sono state proposte (e.g. aggiungere un ciclo a tutti i nodi che di outdegree 0), portando a diversi punteggi. Questo problema non è certamente accademico, dato che nella maggior parte degli snapshot del web una parte significativa dei nodi avrà sempre outdegree zero (sono noti come *dangling nodes*).

È interessante notare, a ogni modo, che nel preprint scritto in collaborazione con Motwani e Winograd e comunemente citato come definizione di PageRank [MW98]. Brin e Page stessi proposero una ricorrenza lineare diversa ma essenzialmente equivalente a quella dell'indice di Hubbell [Hub65], e dichiararono che $\overline{A}$ poteva avere righe nulle, nel qual caso l'autovalore dominante di

[25] Il lettore dovrebbe essere a conoscenza, a ogni modo, del fatto che la letteratura riguardante l'efficacia di PageRank nel campo dell'information retrieval è alquanto povera, ed è composta essenzialmente di risultati negativi quali [Tay07] e [Ups03b]

$\overline{A}$ sarebbe potuto essere più piccolo di 1, dunque un processo di normalizzazione sarebbe stato necessario per ottenere norma l_1 pari a 1.

L'equazione (12) può essere risolta anche senza patch, infatti

$$\mathbf{p} = (1-\alpha)\mathbf{v}(1-\alpha\overline{A})^{-1} \tag{16}$$

dunque

$$\mathbf{p} = (1-\alpha)\mathbf{v}\sum_{i=0}^{\infty}\alpha^i\overline{A}^i$$

Il che mostra immediatamente che PageRank e l'indice di Katz differiscono semplicemente per il fattore $1-\alpha$ che, tra l'altro, è costante e per la l_1-normalizzazione applicata alla matrice A, similmente alla differenza tra l'autovettore dominante e l'indice di Seeley.

Se A non ha righe nulle, o $\overline{A}$ è stata patchata di modo da essere stocastica, PageRank può essere definito in modo equivalente alla distribuzione stazionaria (i.e. l'autovettore dominante) della catena di Markov con matrice di transizione

$$\alpha\overline{A} + (1-\alpha)\mathbf{1}^T\mathbf{v}$$

Nel caso delle catene di Markov possiamo pensare alla situazione nel modo seguente: tutti i cammini che vanno verso x ci danno l'importanza di x, più lungho è il cammino che porta da un nodo a x, più valore viene dissipato. che è analoga all'equazione (11). Del Corso, Gullì e Romani [GR06] hanno mostrato che i punteggi risultanti (che hanno sempre norma l_1 unitaria) differiscono dal vettore PageRank definito in (12) solo per un fattore di normalizzazione, a patto che le righe nulle della matrice di transizione della catena di Markov siano state rimpiazzate con $\mathbf{v}$. Se A non avesse righe nulle, i punteggi sarebbero, ovviamente, identici in quanto l'equazione (13) può essere facilmente derivata dalla matrice di transizione di cui sopra.

Entrambe le definizioni sono state utilizzate in letteratura: la ricorrenza lineare mostrata in (12) è particolarmente utile qualora vi sia bisogno di dipendenza lineare da $\mathbf{v}$ come in [SV05]. La definizione basata su catene di Markov non è meno comune, nonostante il problema di dover patchare le righe nulle.

8 Una nozione astratta di sistema per il reperimento di informazioni

Nella sua accezione più generale, un sistema di reperimento di informazioni (*information retrieval system*) è dato da una collezione documentale D (insieme di documenti) di dimensione N, da un insieme Q di interrogazioni, e da una funzione di ranking $r : Q \times D \to \mathbb{R}$ che assegna a ogni coppia data da un'interrogazione e un documento, un punteggio (un numero reale), L'idea è che a fronte di un'interrogazione, a ogni documento viene assegnato un punteggio: i documenti con punteggio nullo non sono considerati rilevanti, mentre quelli a punteggio non nullo sono tanto più rilevanti quanto più il punteggio è alto.

Fissata un'interrogazione q il sistema assegna un rango (cioè una graduatoria) tra i documenti rilevanti, ordinandoli per punteggio; il rango è essenziale per restituire i documenti in un ordine specifico, in particolare quando la collezione documentale è di grandi dimensioni.

I criteri per assegnare punteggi si dividono in *endogeni* ed *esogeni*. I due termini (non completamente formali) distinguono punteggi che utilizzano il contenuto del documento (cioè l'interno) da quelli che utilizzano la struttura esterna(per esempio, il grafo dei collegamenti ipertestuali tra i documenti). I criteri si dividono ulteriormente in *statici* (o indipendenti dall'interrogazione) e *dinamici* (o dipendenti dall'interrogazione). Nel primo caso, il punteggio assegnato a ciascun documento è fisso. Tutte le misure di centralità che abbiamo discusso possono essere utilizzate come punteggi esogeni, e quindi ci concentreremo su quelli endogeni.

La valutazione di un information retrieval system è basata sull'assunzione che a ogni interrogazione q sia assegnato un insieme di documenti *rilevanti* - quelle che un ipotetico utente considererebbe risposte valide all'interrogazione stessa. Il concetto può essere raffinato assumendo un ordine (totale o parziale) sui documenti rilevanti, che deve essere il più possibile coincidente con il punteggio assegnato dal sistema.

9 Punteggi endogeni

I punteggi endogeni utilizzano il contenuto di un documento. Possono essere statici o dinamici (quindi dipendere o meno dall'interrogazione). Ci occuperemo dei metodi più classici, che si basano su un'interrogazione espressa come *bag of words*. In questo modello, un'interrogazione è semplicemente un insieme di termini: non ci sono operatori booleani. Di solito viene utilizzata un'interpretazione implicita *disgiunta* (i documenti rilevanti sono quelli che contengono almeno uno dei termini), ma è anche possibile seguire un'interpretazione congiunta.

Il criterio più banale per assegnare un punteggio a un documento è quello di *conteggio*: assegnamo a un documento il punteggio dato dalla somma dei conteggi dei termini dell'interrogazione che compaiono nel documento stesso. In questo modo, i documenti in cui i termini compaiono più di frequente avranno un punteggio elevato (questo perché se contengono con alta frequenza un termine desiderato devono essere più rilevanti). Se $c_{t,d}$ è il conteggio per un certo termine in un certo documento e Q è l'insieme dei termini contenuti nella query, in formule

$$r(d) = \sum_{t \in Q} c_{t,d}$$

Il conteggio è un metodo molto primitivo: innanzitutto si presta molto facilmente a manipolazione. Inoltre non tiene conto del fatto che alcuni termini occorrono con grande frequenza non perché rilevanti, ma perché altamente frequenti all'interno di ogni documento. Per esempio, nell'interrogazione "Romeo e Giulietta" il metodo di conteggio valuterà in maniera estremamente positiva documenti contenenti un gran numero di "e", ignorando il fatto che potrebbero essere semplicemente documenti molto lunghi riguardanti tutt'altro.

Per ovviare a questi inconvenienti sono stati sviluppati degli *schemi di pesatura* che aggiustano il punteggio assegnato a ogni termine in ogni documento in modo da ovviare agli inconvenienti del conteggio.

Il primo e più classico metodo è TF/IDF (*term frequency / inverse term frequency*) [Jon72]. Esso normalizza il conteggio dividendolo per il massimo conteggio all'interno del documento o per la lunghezza del documento stesso, e inoltre lo attenua moltiplicandolo per l'inverso della frequenza del termine, o, più frequentemente, per il logaritmo del numero di documenti diviso per la frequenza (cioè per il logaritmo dell'inverso della frequenza in senso probabilistico). Se l è la lunghezza de documento e f è la frequenza di t nella collezione documentale, in formule

$$r(d) = \frac{c_{t,d}}{l} \log\left(\frac{N}{f}\right)$$

Come accennato, al posto di l è possibile utilizzare il massimo conteggio di un termine che compare in d. Sono possibili molte altre varianti (come applicare un logaritmo a $c_{t,d}$).

Tornando all'esempio precedente, il termine "e" comparirà in quasi tutti i documenti, e verrà quindi pressoché ignorato dallo schema TF/IDF, dato che $N/f \approx 1$. La normalizzazione sulla lunghezza del documento, per contro, cerca di tenere conto del fatto che in documenti lunghi è naturale che un termine compaia più volte. Si noti che, comunque, il termine di conteggio compare in maniera lineare, ed è quindi facilmente soggetto a manipolazione.

Lo studio degli schemi di pesatura è parte arte, parte scienza e parte magia. Numerosi schemi sono stati inventati in maniera puramente euristica, e si sono dimostrati efficaci alla prova dei fatti. Uno degli schemi più celebri è BM25 (stando agli autori, il 25-esimo tentativo), uno schema di pesatura basato sul *modello probabilistico*. BM25 pesa un termine come segue

$$\frac{(1+k_1)c_{t,d}}{k_1((1-b)+bl/L)+c_{t,d}} \log\left(\frac{N-f+0.5}{f+0.5}\right)$$

dove b e k_1 sono dei parametri liberi che devono essere tarati sulla collezione documentale, e l è la lunghezza media di un documento nella collezione. La parte dentro al logaritmo è una versione del punteggio IDF. Si noti che la formula non è più lineare in c. In effetti, la formula cresce

monotonicamente in c, ma ha limite asintotico $k_1 + 1$ (quindi un numero eccessivo di ripetizioni del termine non influisce più di tanto).

Si noti che se $k_1 = 0$ la forma si riduce alla parte IDF, mentre per k_1 molto grande la formula è approssimativamente lineare in c (nelle applicazioni reali k_1 è in genere tra 1 e 3).

Il termine b serve a controllare l'influenza della lunghezza del documento rispetto alla lunghezza media.

Riferimenti bibliografici

[Ant71] Jacob M. Anthonisse. The rush in a directed graph. *Technical Report BN 9/71*, 1971.

[Bav50] Alex Bavelas. Communication patterns in task-oriented groups. *J. Acoust. Soc. Am.*, 6(22):725–730, 1950.

[Blo70] Burton H. Bloom. Space-time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[Bon06] Marcin Zukowski Sándor Héman Niels Nes Peter A. Boncz. Super-scalar ram-cpu cache compression. *In Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA, a cura di Ling Liu, Andreas Reuter, Kyu-Young Whang, e Jianjun Zhang*, 2006.

[Cha02] Moses Charikar. Similarity estimation techniques from rounding algorithms. *In STOC*, pages 380–388, 2002.

[Dud13] Jarek Duda. Asymmetric numeral systems: entropy coding combining speed of huffman coding with compression rate of arithmetic coding. *CoRR*, 2013.

[eBSM97] Zbigniew J. Czech George Havas e Bohdan S. Majewski. Perfect hashing. *Theoretical Computer Science*, 182(1-2):1–143, 1997.

[eJIM00] Erik D. Demaine Alejandro López-Ortiz e J. Ian Munro. Adaptive set intersections, unions, and differences. *In Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages pag. 743–752, 2000.

[Eli75] Peter Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21:194–203, 1975.

[eMLS96] Maged M. Michael e Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. *n Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pag. 267-275, 1996.

[Fre77] Linton C. Freeman. A set of measures of centrality based on betweenness. *Sociometry*, 1(40):35–41, 1977.

[Ghe08] Jeffrey Dean Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[Gol80] Solomon W. Golomb. Sources which maximize the choice of a huffman coding tree. *Inform. and Control*, 45(3):263–272, 1980.

[GR06] Gianna Del Corso Antonio Gullì and Francesco Romani. Fast pagerank computation via a sparse linear system. *Internet Math.*, 3(2):251–273, 2006.

[Har01] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. *Proceedings of the 15th International Conference on Distributed Computing*, volume 2180:pag. 300–314, 2001.

[Hub65] Charles H. Hubbell. An input-output approach to clique identification. *Sociometry*, 4(28):377–399, 1965.

[Jon72] Karen Sparck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 1(28):11–21, 1972.

[Kat53] Leo Katz. A new status index derived from sociometric analysis. *Psychometrika*, 1(18):39–43, 1953.

[Lan95] Edmund Landau. Zur relativen wertbemessung der turnierresultate. *Deutsches Wochenschach*, (11):366–369, 1895.

[Lat00] Massimo Marchiori Vito Latora. Harmony in the small-world. *Physica A: Statistical Mechanics and its Applications*, 3-4(285):539–546, 2000.

[MW98] Lawrence Page Sergey Brin Rajeev Motwani and Terry Winograd. The pagerank citation ranking: Bringing order to the web. *Technical Report SIDL-WP-1999-0120*, 1998.

[Naj99] Allan Heydon Marc Najork. Mercator: A scalable, extensible web crawler. *World Wide Web*, pages 219–229, December 1999.

[O'N96] Patrick O'Neil Edward Cheng Dieter Gawlick Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[Pag98] Sergey Brin Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 1(30):107–117, 1998.

[Ple94] Abraham Berman Robert J. Plemmons. Nonnegative matrices in the mathematical sciences. *Classics in Applied Mathematics*, 1994.

[Sar07] Gurmeet Singh Manku Arvind Jain Anish Das Sarma. Detecting near-duplicates for web crawling. *In Proceedings of the 16th international conference on World Wide Web*, pages 141–150, 2007.

[Sch69] Isaac J. Schoenberg. Publications of edmund landau. *Number Theory and Analysis: A Collection of Papers in Honor of Edmund Landau*, pages pages 335–355, 1969.

[See49] John R. Seeley. The net of reciprocal influence: A problem in treating sociometric data. *Canadian Journal of Psychology*, 4(3):234–240, 1949.

[Sin10] Felix Putze Peter Sanders Johannes Singler. Cache-, hash-, and space-efficient bloom filters. *Journal of Experimental Algorithmics (JEA)*, 14, 2010.

[Sl19] J.M. Buldú J. Busquets I. Echegoyen F. Serul-lo. Defining a historic football team: Using network science to analyze guardiola's fc barcelona. *Scientific reports*, 1(9):1–14, 2019.

[SV05] Paolo Boldi Massimo Santini and Sebastiano Vigna. Pagerank as a function of the damping factor. *Proc. of the Fourteenth International World Wide Web Conference (WWW 2005)*, pages pages 557–566, 2005.

[Tay07] Marc Najork Hugo Zaragoza Michael J. Taylor. Hits on the web: how does it compare? *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 471-478, 2007.

[Ups03a] Nick Craswell David Hawking Trystan Upstill. Predicting fame and fortune: Pagerank or indegree? *In Proceedings of the Australasian Document Computing Symposium*, pages 31–40, 2003.

[Ups03b] Nick Craswell David Hawking Trystan Upstill. Predicting fame and fortune: Pagerank or indegree? *In Proceedings of the Australasian Document Computing Symposium*, pages 31–40, 2003.

GNU Free Documentation License

Version 1.3, 3 November 2008

<https://fsf.org/>

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "**Document**", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "**Invariant Sections**" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "**Cover Texts**" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "**Transparent**" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for

input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "**Opaque**".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "**Title Page**" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "**publisher**" means any person or entity that distributes copies of the Document to the public.

A section "**Entitled XYZ**" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "**Acknowledgements**", "**Dedications**", "**Endorsements**", or "**History**".) To "**Preserve the Title**" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `https://www.gnu.org/licenses/`.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with … Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.