Algoritmica per il web

Francesco Tomaselli

27 ottobre 2021

Indice

1 Crawling **3**
- 1.1 Principi di base . . . 3
- 1.2 Strutture dati . . . 3
  - 1.2.1 Esempio strutture dati . . . 4
- 1.3 Crivelli . . . 5
  - 1.3.1 Filtri di Bloom . . . 6
  - 1.3.2 LMS tree . . . 7
  - 1.3.3 Memorizzazione offline . . . 8
  - 1.3.4 LRU cache . . . 9
- 1.4 Quasi duplicati . . . 9
- 1.5 Politeness . . . 10
- 1.6 Concorrenza . . . 10
  - 1.6.1 Componenti di un crawler . . . 11
  - 1.6.2 Strutture concorrenti . . . 11
- 1.7 Parallelismo . . . 12
  - 1.7.1 Permutazioni aleatorie . . . 13

2 Retrieval **15**
- 2.1 Codici istantanei . . . 15
  - 2.1.1 Disuguaglianza di Kraft-Mcmillan . . . 16
- 2.2 Esempi di codici . . . 17
  - 2.2.1 Codici binari . . . 17
  - 2.2.2 Elias . . . 17
  - 2.2.3 Golomb . . . 18
  - 2.2.4 Codici a blocchi a lunghezza variabile . . . 18
  - 2.2.5 Distribuzione intesa . . . 18
  - 2.2.6 PFOR-DELTA . . . 19
  - 2.2.7 Asymmetric Numeral System . . . 19

1 Crawling

1.1 Principi di base

Insiemi di nodi in una visita All'interno di un processo di crawling, si distinguono tre insiemi di nodi

- U: nodi sconosciuti
- F: frontiera, ovvero nodi che sono conosciuti ma non ancora visitati
- V: nodi visitati

Una visita del web opera scegliendo un nodo della frontiera, aggiungendolo ai visitati, e aggiungendo i siti raggiungibili da esso alla frontiera.

In una visita di un grafo classica, gli insiemi U e F coincidono nel secondo.

Osservazione 1. *Teoricamente una visita potrebbe esaurire la frontiera, praticamente non finisce mai, a parte in rari casi.*

Inizializzazione frontiera La frontiera va inizializzata, tipicamente si scelgono siti web popolari, per esempio giornali etc. In generale si considerano *semi* di inizializzazione, ovvero un insieme di siti web scelti da umani, che sono promettenti.

La scelta del seme governa l'esplorazione della frontiera, verranno scelti infatti prima siti più vicini ai siti di inizializzazione.

Un'altra strada per la frontiera è scegliere a priori un insieme di siti web, e considerarla direttamente come frontiera, tralasciando i siti raggiungibili da essi.

1.2 Strutture dati

Strutture in memoria Sono cruciali le strutture usate per memorizzare i nodi visitati e la frontiera.

Mantenere i visitati in una hashtable è ragionevole, non lo è per la frontiera, tipicamente di ordini di grandezza più grande dell'insieme V.

Limitare la grandezza della frontiera è cruciale, si potrebbere preferire i link iniziali in una pagina piuttosto che quelli a fondo. Oppure limitare il numero massimo di pagine estratte da un singolo dominio o considerare un limite alla profondità all'interno di un singolo sito web.

Scelta del prossimo nodo Ciò che determina il comportamento una visita di crawling è la scelta del prossimo nodo della frontiera. Una scelta ragionevole potrebbe essere una visita in ampiezza a partire dalla frontiera iniziale. L'assunzione è che pagine di qualità puntino ad altre pagine di qualità.

Operare in profondità non funziona, significherebbe continuare a scendere all'infinito, al contrario di una classica DFS che termina e si *torna indietro* con la ricorsione. Questo perchè i siti non visitati sono praticamente infiniti.

Criteri più sofisticati possono associare una sorta di priorità alle pagine. Tali criteri sono legati ai contenuti delle pagine, alla struttura dell'url, alcuni esempi sono:

- Url corti, con l'assunzione che i livelli siano separati da un backslash;
- Url fuori dal sito corrente;
- Parole chiave di interesse all'interno dell'url;
- Utilizzo il contenuto della pagina corrente per avere informazioni sulla pagina successiva.

In questo caso si utilizza una coda a priorità, con qualche valore di priorità associato ad ogni nodo.

Osservazione 2. *In questo paragrafo si sta assumendo un contesto single thread, ovviamente nella realtà si utilizzerà un approccio multithread, e molti fattori, quali latenza, tempi di risposta, race condition, influenzano sull'ordine di visita effettivo.*

1.2.1 Esempio strutture dati

Visti hashati Una prima idea per mantenere l'insieme V è non memorizzare url completi ma una certa firma digitale di essi.

Consideriamo ad esempio una funzione di hash:

$$f : URL \longrightarrow 2^{64} = \{0, \ldots, 2^{64-1}\}$$

È possibile che due url collidano, creando errori sui positivi, ovvero un url non davvero visitato viene visto come già esplorato.

Solitamente le collisioni non importano, ma, dati k elementi contenuti nella struttura, data una funzione di hash buona, tipicamente il numero di collisioni è dell'ordine di $\frac{k^2}{2n}$, dove n è la grandezza del codominio.

Nella pratica quindi, funzioni di hash come f, non creano un numero spiacevole di collisioni.

Osservazione 3. *Una tabella di firme è molto più efficiente di mantenere i dati effettivi, basti pensare a problemi di allocazione di memoria inefficiente,*

frammentazione etc. Pagare il prezzo dei conflitti, implica risparmiare molti problemi e spazio.
Mantenere una tabella di firme significa di fatto mantenere in memoria strutture più piccole dell'information theoretical lower bound, pagando il prezzo delle collisioni.

Database NoSQL Sono database che memorizzano entry chiave valore, un esempio è *RocksDB*. Se ordinati sono implementazioni efficienti di B-Tree, altrimenti di dizionari classici, in parte memorizzati su disco.

Si utilizzano anche LSM-Tree, alberi che dividono per livelli le chiavi e mantengono chiavi utilizzate di frequente all'inizio, spostando chiavi poco frequenti nei livelli più bassi.

1.3 Crivelli

Un crivello è una struttura dati che accetta le seguenti primitive:

- *add(u)*: aggiunge un url alla struttura;
- *get()*: preleva un elemento dalla struttura.

Un crivello ha un aspetto insiemistico, ovvero ricorda cosa è stato inserito o no, gestisce anche l'ordine in cui restituisce i dati, tipicamente si considera una visita in ampiezza.

Un' implementazione base è un' hashtable in memoria, per i già visti, ovvero $V \cup F$, e una coda in memoria per gestire l'ordine di visita, BFS nel caso di una coda classica.

Osservazione 4. *È necessario comunque del processing degli url estratti dal crivello, ad esempio, potremmo ordinarli per dominio, in modo da raggruppare richieste allo stesso sito.*

Implementazione È già stata accennata la possibilità di avere una hashtable più una coda. L'hashtable potrebbe, come introdotto a sottosezione precedente, solo le firme degli url, in modo da ridurre lo spazio in memoria necessario.

Graceful degradation È importante garantire il *graceful degradation*, ovvero, il sovraccarico della struttura ne peggiora le performance ma non fa crollare l'intero processo. Una struttura dati classica, come una hashtable, non garantisce questa proprietà, al termine della memoria essa fallisce.

1.3.1 Filtri di Bloom

Un filtro di Bloom è un dizionario approssimato, che garantisce una certa probabilità di errore positivo assumendo un certo upper bound al numero di chiavi inserite.

L'impronta in memoria di un filtro di questo tipo è costante, non varia quindi al variare del carico della struttura.

Funzionamento Sia b un vettore di m bit, sia d il parametro che regola la precisione del filtro, e $f_i : U \to m, 0 < i < d$ funzioni di hash che mappano un elemento dell'universo in un indice del vettore.

Le primitive disponibili sono, dato $x \in U$:

- *add(x)*: imposto ad uno le posizioni date dalle funzioni di hash, ovvero $f_0(x), \ldots, f_{d-1}(x)$;
- *contains(x)*: restituisco l'and logico delle posizioni $b[f_i(x)]$.

L'operazione contains restituisce zero se e solo se x non è stato inserito nel filtro. Un risultato positivo però può significare che x sia stato inserito oppure che inserimenti precedenti abbiano settato ad uno tutte le celle relative agli hash di x.

Probabilità di falsi positivi Se fissiamo d ad uno non si sta guadagnando rispetto ad una classica tabella di hash. Più d è grande, più sono improbabili i falsi positivi, ma più uni si vanno ad aggiungere, quindi i falsi positivi aumentano.

Bisogna quindi trovare un tradeoff tra m e d per minimizzare la probabilità di falsi positivi. Si considera ora un n, ovvero il numero di chiavi inserite.

Si ottiene che la probabilità di un falso positivo equivale a $\frac{1}{2}^d$ e la dimensione $m = 1.44dn$. Questo implica che fissata una precisione e il numero di chiavi massimo, si può ottenere lo spazio necessario, similmente, fissato m e n si può ottenere la precisione ottenuta.

Osservazione 5. *La struttura ha degrado grazioso, infatti, eccedendo n l'analisi di precisione non vale più e i falsi positivi aumentano. Si nota anche che sotto la soglia n la probabilità di falsi positivi diminuisce. La struttura non fallisce, peggiora in precisione fino a che è satura, restituendo sempre positivo.*

Osservazione 6. *Gli accessi in cache sono pessimi per quanto riguarda un filtro di bloom, dovendo controllare celle di memoria non correlate e sostanzialmente casuali, quindi con poca probabilità di essere in cache.*

Osservazione 7. *I risultati negativi in media accedono a solo due celle.*

Blocked Bloom filter Si può pensare di dividere un filtro di Bloom in due filtri più piccoli di dimensione $1.44d\frac{n}{2}$ ciascuno. Una funzione g sceglie quale filtro scegliere, poi, si inserisce l'elemento nel filtro selezionato.

La divisione permette di mantenere i filtri in cache e velocizzare il tutto. Un'analisi avanzata individua un degrado di precisione. Questo perché alcuni filtri mantengono poche chiavi, ed altri saranno più sovraccarichi.

Calcolo funzioni di hash Supponendo di avere due funzioni di hash $h(x)$ e $g(x)$, siano $a = h(x)$ e $b = g(x)$, considerando l'i-esima funzione di hash come $ai + b$, l'analisi sulla precisione di falsi positivi rimane valida.

Osservazione 8. *Il calcolo fatto in questo modo è estremamente più efficiente di calcolare d funzioni separate.*

1.3.2 LMS tree

Un *Log-Structure-Merge* tree memorizza file di log in cui si scrive solo in append.

A differenza di un classico *B-tree*, i registri non cambiano. Questi alberi funzionano molto bene quando si scrive molto e legge poco.

Livelli L'idea di base di un *LMS* tree è avere un certo numero di livelli nella struttura dati. Idealmente i vari livelli possono essere mantenuti in mezzi di memorizzazione differenti. Si assume ora che il primo livello sia in memoria, in una struttura classica come un B-tree e i successivi su disco come registri di coppie chiave-valore ordinate.

Ogni livello ha memorizzate un certo numero di coppie chiave-valore, ed ogni livello successivo occupa dieci volte più del precedente.

Ricerca Per trovare una chiave, si cerca nei livelli in maniera sequenziale, una volta individuata, al livello più alto possibile[1], si restituisce. Visto che le chiavi sono ordinate, si potrebbe effettuare una ricerca dicotomica[2].

Inserimento L'aggiunta all'albero inserisce la chiave al livello zero. Se la chiave non è presente il B-tree sale di dimensione, se eccede la dimensione massima: si considera un segmento contiguo di chiavi e si fondono con il livello uno[3]. Si procede fino a che non si trova un livello che riesce a mantenere le chiavi. Se si raggiunge l'ultimo livello, se ne crea uno nuovo.

[1] Potrebbe comparire anche in livelli successivi, visto che si ammettono duplicati

[2] Bisogna stare attenti al tipo di memoria, a volte accessi sequenziali sono molto più veloci di accessi aleatori, un esempio è nel nastro.

[3] La fusione è molto veloce essendo le due strutture ordinate

Il principio di base è che le fusioni dei livelli, che sono molto costose, avvengono sempre più raramente allo scendere nell'albero.

Rimozione La rimozione consiste nella ricerca della chiave e nella sostituzione del suo valore con una *tombstone*. Una ricerca successiva troverà questo valore e capirà che la chiave è stata rimossa.

Frammentazione livelli Ogni livello nella pratica è frammentato in tanti file piccoli. Questo offre molti vantaggi, ad esempio operazioni di merge parallele. Inoltre, favorisce l'evitare collisioni di concorrenza.

La ricerca poi è più veloce, poiché è possibile mantenere un indice sparso, con un sottoinsieme di chiavi, con puntatori ai frammenti relativi alla chiave. Ogni livello ha poi un filtro di Bloom. La ricerca quindi testa il filtro e se la risposta è negativa si procede al livello successivo, altrimenti, si controlla l'indice sparso in ricerca dicotomica, e si procede sequenzialmente dalla maggior chiave minore uguale di quella ricercata.

1.3.3 Memorizzazione offline

Si mantengono solo file su disco e non strutture in memoria. I file vengono ordinati e fusi spesso. In particolare, si hanno a disposizione:

- *VIS*: url da visitare
- F: frontiera
- V: url visti

Il crawler accumula gli url che trova nelle pagine nel file F, fino a che si terminano i siti da visitare, contenuti in *VIS*, oppure lo spazio su disco.

A questo punto la frontiera viene ordinata e de-duplicata. L'operazione è necessaria poiché il file potrebbe contenere duplicati, visto che non esiste nessuna struttura in memoria che impedisce l'aggiunta di ripetizioni.

A questo punto si fondono i file F e V:

- Se un url sta in entrambi non faccio nulla;
- Se trovo un url in V ma non in F non faccio nulla;
- Se trovo un url in F ma non in V, lo aggiungo a V e a *VIS*.

La fusione è lineare nella lunghezza dei file F e V, poiché gli url sono in ordine lessicografico. Procedo a questo punto con la visita, dagli url contenuti in *VIS*.

Osservazione 9. *Questa tecnica è utilizzata dal crawler Nutch, le operazioni su disco sono effettuate tipicamente da architetture distribuite quali map-reduce.*

Osservazione 10. *È possibile mantenere all'interno di V solo gli hash dei siti in modo da risparmiare memoria, a patto di ordinare gli url di F secondo il loro hash e di effettuare i confronti sugli url hashati.*

1.3.4 LRU cache

Può essere una buona idea anteporre al crivello una cache LRU, questo rimuove molti duplicati, infatti se un url è all'interno della cache è già stato inserito all'interno del crivello.

Ogni tanto un url verrà inserito più volte, perché finito fuori dalla cache.

1.4 Quasi duplicati

Per molteplici ragioni, è possibile incontrare spesso le stesse pagine, per esempio, la stessa pagina con http e https, pagine che differiscono per una sola data, etc.

È necessario un sistema che decida se due pagine sono quasi identiche.

Digest Si puo generare un digest dalla pagina web, buttando via alcuni elementi, per esempio i tag html, le maiuscole, etc. Si può anche campionare la pagina in qualche intervallo e memorizzare solo i campionamenti.

Una volta generati questi digest saranno inseriti in un filtro di Bloom.

SimHash L'idea di base è generare hash simili per testi simili. La distanza tra due testi si misura in distanza di Hamming. Si risolvono problemi come ad esempio il considerare due pagine diverse per un singolo errore ortografico.

A partire dal testo si ottengono delle feature da esso. Un esempio potrebbe essere estrarre le parole, oppure considerare tre-gram da esso, ovvero sotto-sequenze di tre caratteri o magari tre parole alla volta, etc. N-gram corti implicano il non considerare troppo importanti errori di ortografia.

Siano ora $s \in S$ le feature estratte e $h : \mathit{Stringhe} \to 2^b$ funzione di hash. Consideriamo ora il valore di hash per ogni feature, $h(s)$, l'hash finale del documento si ottiene contando la maggioranza per ogni bit degli hash delle feature, per esempio:

- $s_0 : 1, 0, 1, 1, 0$;
- $s_1 : 0, 1, 0, 0, 1$;
- $s_2 : 0, 1, 0, 1, 1$;

L'hash finale del documento sarà: $0, 1, 0, 1, 1$

Il calcolo effettuato in questo modo sistema i casi di poche differenze tra i testi, dipende ovviamente dalle feature che si considerano, ma è auspicabile che poche feature differenti tra due documenti non modifichino i voti di maggioranza, oppure, che influiscano su pochi bit.

1.5 Politeness

Dal punto di vista etico, una persona che mette una pagina sul web deve accettare che sarà visitata, da umani.

La visita e l'indicizzazione automatica tramite crawler non dovrebbe continuare per periodi troppo prolungati.

Politeness assoluta Intervallare di un certo numero di secondi le richieste consecutive ad un sito, oppure allo stesso ip. Esempio, faccio 10 richieste, pausa di qualche secondo, altre 10 richieste.

Politeness relativa Per un certo intervallo di tempo scarico tutto quello che posso. Poi mi adatto in base a come la pagina ha risposto, ad esempio, scarico per un secondo tutto quello che riesco, ma la pagina impiega 10 secondi a rispondere, aspetto quindi 30 secondi prima di effettuare una nuova richiesta.

Potrei usare HTTP 1.1, per mantenere aperta la connessione.

Osservazione 11. *È necessario quindi un'architettura multiflusso per raccogliere dati da molti siti contemporaneamente, altrimenti si aspetterebbe molto per ragioni di politeness e si diventerebbe troppo dipendenti dall'ordine della coda dei siti da visitare, infatti, dovrei aspettare se il prossimo sito fosse nello stesso dominio di quello attuale, no altrimenti.*

Robots.txt File che deve comparire nella radice di un sito web dove si possono escludere parti di sito da visitare. Se uno ignorasse i contenuti del file probabilmente sarebbe bannato.

1.6 Concorrenza

Ogni host estratto dal crivello ha una coda con gli url relativi ad esso. Bisogna quindi ora capire quale host considerare e visitarne gli url associati.

Visite ad host diversi possono essere fatte in concorrenza, per capire quale host è disponibile, secondo le regole di politeness, si utilizza una priority queue.

1.6.1 Componenti di un crawler

Coda di host Si considera una coda di host, ordinati per tempo da aspettare prima di poter scaricare dal sito.

Se il sito in testa ha tempo minore uguale a quello attuale esso si estrae, si scaricano tutti i siti che si riescono da quell'host, poi si reinserisce in coda con timestamp uguale al tempo attuale più intervallo di politeness.

La coda è gestita da un semaforo per permettere concorrenza.

Osservazione 12. *Considerando una dimensione massima per la coda potrei dover escludere dei siti se il timer crescesse troppo. Sta a chi progetta il crawler decidere se potare la coda nel caso in cui cresca troppo in memoria.*

Coda di ip Il discorso si complica nel caso si considerino gli ip. Si considera una coda di ip con un timestamp,[4], ogni ip ha una coda di host come quella presentata in precedenza. Il timestamp dell'ip equivale al massimo tra il timestamp suo e quello della testa della sua coda di host.

Una volta individuato l'ip da estrarre si estrae dalla coda e si procede come in precedenza.

Questo garantisce di rispettare la politeness anche verso ip con molti host associati.

Fetching e parsing threads Nella pratica ci sono fetching thread che scaricano secondo le policy descritte negli ultimi due paragrafi, che scrivono anche su disco.

I parsing thread poi, molto minori rispetto a quelli di fetching, si occupano di processare i dati scaricati. Quello che fanno è scrivere da qualche parte i dati, cercare gli url nuovi e passarli al crivello.

Osservazione 13. *Si potrebbero inserire filtri tra fetching e parsing threads, per evitare di processare alcune pagine troppo lunghe, troppo corte, di lingua differente etc.*

1.6.2 Strutture concorrenti

Non è auspicabile utilizzare strutture bloccanti basate su semafori poichè i thread paralleli sono troppi, quindi moltissimi sarebbero costretti ad aspettare lo sblocco della struttura condivisa.

[4] Si può assumere un timer di politeness di 10 secondi.

CAS L'istruzione compare and swap, cas(p,a,b), assegna il valore b al puntatore p se il suo contenuto è a. È atomica ed utilizzata per implementare concorrenza.

Linked list lock-free L'inserimento in una linked list lock-free cerca la posizione dove si vuole inserire, e si inserisce con compare and swap.

L'algoritmo ripete l'assegnamento quando due thread stanno modificando in maniera concorrente lo stesso nodo, infatti la cas di uno dei due fallisce.

Non si possono prevedere quante iterazioni del ciclo ci vorranno, ma ad ogni iterazione si ha del progresso verso la fine delle operazioni, visto che se il thread attuale fallisce è perché un'altro ha avuto accesso alla struttura.

Non si sta avendo nè deadlock ne starvation, infatti la struttura sta facendo progresso. È un concetto leggermente diverso dal busy waiting, poiché sebbene si stia facendo polling, la struttura sta evolvendo e non è ferma.

Osservazione 14. *Una coda lock-free potrebbe essere inserita tra fetching threads e la coda di host, che regola l'ordine di visita attuale. Questa coda potrebbe influenzare l'ordine di visita regolato dalla coda a priorità di host.*

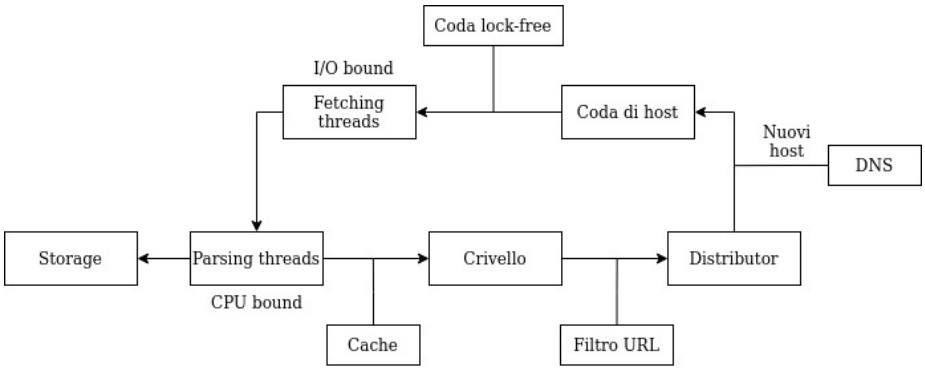

Figura 1: Componenti di un crawler

1.7 Parallelismo

Coordinazione agenti È auspicabile avere più agenti che fanno attività di crawling in parallelo, bisogna pensare a come distribuire il carico, un'esempio potrebbe essere quello di dividere per host.

Fare ciò non è molto efficiente, si può ricorrere a qualcosa di centralizzato. Ogni agente chiede al coordinatore se tocca a lui occuparsi di un determinato host. Ovviamente è un single point of failure, si può pensare quindi di avere una distribuzione del coordinatore[5]. Non si affronterà un algoritmo del genere, si vede ora un altro approccio.

[5]Un algoritmo possibile è PAXOS, una delle sue implementazioni è ZOOKEEPER

Gestione statica Tolta la condizione di rottura degli agenti, quindi assumendo staticità, assegnare un url ad un agente richiede calcolare una funzione di hash modulo n, oppure sfruttare un approccio round robin.

Il problema di questo approccio è che non funziona bene se gli agenti cambiano nel tempo.

Requisiti di gestione dinamica P agenti

A agenti vivi $\subset$ B $\delta_A : U \to A$

$\delta_A(u) \in A$

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

$A \subset B \implies \delta_B^{-1}(a) \subset \delta_A^{-1}(a) \forall a \in A$

1.7.1 Permutazioni aleatorie

Arrivato un url, creo un generatore di numeri casuali con seed pari a u. Permuto l'insieme P di agenti e scorro finché non trovo un agente attivo $a \in A$.

Se si aggiunge un nuovo agente, gli unici url che cambiano assegnamento sono quelli che hanno nella lista degli agenti generata per essi, hanno un nuovo agente che precede quello attivo selezionato precedentemente. Quindi, gli url che cambiano posto vanno solo ad agenti nuovi.

Knuth-FisherYates shuffle Parto da un array lungo n. Scorro l'array, all'i-esimo passo, seleziono una posizione $k \in [i, n-1]$ e scambio l'elemento k e i.

Sono possibili $n!$ scelte, l'algoritmo genera in modo equiprobabile una permutazione possibile.

Si nota che, una volta individuato un elemento di A posso fermarmi, tanto sarà lui a gestire l'url. In media quindi effettuo $\frac{|A|}{|P|}$ scambi.

Min-hash Sia u un url e A l'insieme degli agenti, h una funzione di hash. Un url viene assegnato all'agente $a = \text{argmin}_{a \in A}\ h(u, a)$.

Una collisione si risolve in maniera deterministica. Questo e l'approccio precedente sono molto simili.

Hashing coerente Un'interpretazione geometrica è la seguente: si considera una circonferenza, per ogni agente si inseriscono C punti casuali sulla circonfe-

renza. Ogni agente inizializza un seed con l'identificativo degli altri, generando casualmente i loro punti, quindi ogni agente ha la stessa circonferenza con C punti per ogni $a \in A$.

Per capire dove va un url u, si parte dal punto definito da $h(u)$, si procede in senso orario finché non si incontra un agente. Le repliche servono a rendere più bilanciati gli assegnamenti.

Osservazione 15. *È possibile in generale capire chi si occupava dell'url prima dell'arrivo di un nuovo agente, l'approccio cambia a seconda della tecnica di hashing utilizzata, nello shuffle basta procedere con l'estrazione, nel min-hashing si può mantenere una finestra di k minimi. Nell'ultimo caso si può procedere nello scan della circonferenza.*

2 Retrieval

L'attenzione passa ora alle pagine scaricate. Ogni documento è numerato e contiene una certa quantità di caratteri, l'obiettivo ora è indicizzarli.

Osservazione 16. *Tipicamente bisogna fare guessing per capire la codifica del testo, esistono tabelle dedicate a questo in ogni browser.*

Segmentazione dei documenti Partendo da un documento di ottengono sequenze di token, in qualche modo, per esempio spezzando il testo agli spazi, rimuovendo le stop-words etc. Tipicamente si applicano operazioni di normalizzazione, come troncamento, lemmatizzazione, etc.

Matrice termini-documenti Ordinando la totalità dei termini nei documenti si può ottenere una rappresentazione matriciale con documenti sulle colonne e termini per righe. Una cella indica la presenza o meno di una certa parola nel testo di un documento.

Quello che si fa poi è creare la trasposta ordinata, ovvero una matrice che per ogni token ha una lista di interi che rappresentano i documenti in cui esso è contenuto.

2.1 Codici istantanei

Un codice è un sottoinsieme di parole binarie: $C \subseteq 2^* = \{0,1\}^*$.

Definizione 1. *Si dice che x è un prefisso di y, $x \preceq y$ se $\exists z \in 2^*$ t.c. $xz = y$, ovvero se concatenato ad un altra parola binaria ottengo y.*

Definizione 2. *Due parole x e y sono confrontabili se $x \preceq y$ oppure $y \preceq x$.*

Definizione 3. *Un codice C si dice istantaneo se $\forall x, y \in C$, x e y sono inconfrontabili, ovvero privo di prefissi.*

Definizione 4. *Un codice C si dice completo se ogni parola binaria è confrontabile con una parola $x \in C$. Ovvero, aggiungendo una qualsiasi parola binaria, il codice non è più istantaneo.*

Motivazioni Memorizzare sequenze di interi ordinate in maniera crescente può essere fatto in maniera efficiente memorizzando gli scarti, differenze, tra due interi consecutivi.

$$1, 6, 7, 10, 12 \longrightarrow 1, 5, 1, 3, 2$$

Queste liste potrebbero essere le righe della matrice termini-documenti.

Un'altra proprietà interessante di un codice istantaneo è l'univocità di decodifica scorrendo una sequenza di parole concatenate. Infatti, visto che nessuna parola

è prefisso di un'altra, ho un solo modo di decodificare una sequenza di zeri e uni.

Ordinamento nei codici istantanei Due esempi di codici istantanei sono:

- $k = 0^k 1$
- $k = 1^k 0$

Il secondo codice mantiene l'ordine, il secondo no.

Importanza della completezza Se un codice è istantaneo e completo si può decodificare ogni stringa binaria casuale in modo univoco, altrimenti no.

2.1.1 Disuguaglianza di Kraft-Mcmillan

Definizione 5. *Sia C un codice istantaneo, vale:*

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

Se C istantaneo, C è anche completo sse $\sum_{w \in C} \frac{1}{2^{|w|}} = 1$. Nell'interpretazione geometrica[6] non sarebbe libero nessun intervallo.

Osservazione 17. *Il fatto che la sommatoria dia un valore piccolo non dice nulla su un linguaggio qualsiasi, non è vera quindi l'implicazione inversa.*

Teorema interi Siano $t_0 \leq t_1 \leq \cdots \leq t_n$ interi, se vale:

$$\sum_{i \in N} \frac{1}{2^{t_i}} \leq 1$$

Allora esiste un codice istantaneo con n, dove la parola i-esima ha lunghezza t_i.

Questo fatto rende possibile un'eventuale correzione, infatti, se avessi un codice con quelle lunghezze ma non fosse istantaneo, so che potrei sistemarlo.

Visione probabilistica La quantità $\frac{1}{2^{|w|}}$ si può vedere come una probabilità, visto che nel caso in cui un codice sia istantaneo tutte quelle quantità si sommano ad uno.

Potrei pensare di codificare probabilità maggiori a parole più corte. La disuguaglianza di prima, e l'algoritmo greedy associato[7] genera il codice ottimo per una distribuzione di probabilità del tipo $\frac{1}{2}, \frac{1}{4}, \ldots, \frac{1}{2^k}$.

[6] Vedi dimostrazione
[7] Non presentato qui, ma simile a Huffman

2.2 Esempi di codici

2.2.1 Codici binari

Notazione binaria ridotta Codice binario in cui si rimuove il primo 1. Per esempio:

$$1001010 \to 001010$$

Codice binario minimale Sia $s = \lceil \log k \rceil$, se $x < 2^s - k$, esso è codificato dall'x-esima parola binaria di lunghezza $s - 1$, altrimenti tramite la $(x - k + 2^s)$-esima parola binaria di lunghezza s.

2.2.2 Elias

Codice γ Ogni numero è scritto in notazione binaria ridotta, ovvero in binario senza il primo uno, preceduto dalla sua rappresentazione unaria.

Numero	Codifica γ	Unario
1	1	1
2	010	01
3	011	001
4	00100	0001
5	00101	00001
6	00110	000001
...	...	...

La lunghezza di x nella codifica γ è $2\lambda(x)+1$, dove λ è $\lfloor \log_2 x \rfloor$. In binario invece spenderei $\lambda(x)$, ma non è istantaneo. In unario invece spendo una quantità pari a x, che in casi in cui i numeri siano piccoli va bene, altrimenti no.

Codice δ Molto simile al precedente, invece che usare l'unario, scrivo davanti alla rappresentazione binaria ridotta la lunghezza del codice γ associato all'x-esimo numero.

Confrontando i codici visti fin'ora:

Codifica	Lunghezza
Unario	$x + 1 = O(x)$
Elias γ	$2\lambda(x + 1) + 1 = 2\log(x + 1) + 1$
Elias δ	x

Il codice δ sui dice asintoticamente ottimo, visto che utilizza un numero logaritmico di bit più qualcosa di più piccolo. Nel lungo termine risparmia rispetto agli altri due codici.

2.2.3 Golomb

Si fissa un parametro b che si dice modulo del codice. Preso un $x > 0$ si codifica in unario la quantità $\lfloor x/b \rfloor$, seguito dalla binaria minimale di $x \ mod \, b$.

Dato un codice, posso tornare al numero con $b\lfloor x/b \rfloor + x \ mod \, b$.

Fissato $b = 3$, il codice diventa

Numero	Codifica Golomb
0	10
1	110
2	111
3	010
4	0110
5	0111
...	...

Osservazione 18. *Golomb con $b = 1$ la prima parte del codice equivale all'unario. Inoltre, questo codice ha problemi simili all'unario, la prima parte infatti cresce asintoticamente come x, nella pratica è minore, ma comunque occupa molto spazio.*

2.2.4 Codici a blocchi a lunghezza variabile

Quello che si fa è memorizzare tanti blocchi per un singolo numero. Il primo bit di ogni byte contiene il fatto che il codice è terminato oppure no. Nel caso in cui non sia terminato bisogna leggere uno o più altri byte per completarlo.

Un miglioramento è mettere tutti i bit *di continuazione* nel primo byte, per capire fin da subito quanto proseguire.

Sio può pensare dal punto di vista logico ad una lista concatenata che contiene pezzi di un numero, la decodifica consiste nello scorrere la lista e concatenare i valori in ogni nodo.

2.2.5 Distribuzione intesa

Ogni codice crea implicitamente una distribuzione intesa, ovvero parole più corte hanno probabilità maggiore, quindi, si può studiare come varia questa probabilità in relazione alla lunghezza che assumono le parole nei vari codici.[8]

Conoscendo la distribuzione di probabilità dei dati si può utilizzare

[8] Vedi appunti, distribuzioni intese.

2.2.6 PFOR-DELTA

Compressione che funziona molto bene quando numeri piccoli compaiono molto più frequentemente di quelli grandi.

Si fissa una dimensione di blocco $B \approx 256$, la compressione avviene per blocchi di B elementi alla volta. Quello che si fa è scorrere gli elementi di un blocco e scegliere un numero di bit b che rappresenti almeno una percentuale $\alpha \approx 90\%$ della totalità dei numeri del blocco.

Compressione A questo punto scorro la lista, se un numero è rappresentabile in b bit lo scrivo, altrimenti scrivo un codice di escape, per esempio tutti uni o zeri.[9] Accodo alla fine i numeri non rappresentabili, tutti a lunghezza a, la minima per rappresentarli tutti. All'inizio della lista inserisco il numero b e il valore a.

Decompressione La decompressione della lista è lineare, bastano due puntatori, uno all'inizio e uno che parte prima dei numeri accodati. Si ottiene una buona compressione, b viene molto piccolo se tanti valori sono piccoli, inoltre nell'*alpha*-percento dei casi non devo accedere alla memoria accodata alla lista, che è la parte più costosa della decompressione.

2.2.7 Asymmetric Numeral System

Sono sistemi in cui cifre diverse sono scritte con numero diverso di bit, il che permette di scrivere numeri dove alcune cifre costano tanto, altre di meno.

Premessa Vorremmo che, dato un messaggio x, l'aggiunta di un simbolo che compare con probabilità p, crei un messaggio $x' \approx \frac{x}{p}$.
Quindi, $\log(x') \approx \log(x) + \log(\frac{1}{p})$. Ovvero, vorrei che, aggiungendo ad un messaggio un simbolo che compare frequentemente, tale messaggio si allunghi di molto poco.

La codifica binaria risponde alla richiesta precedente, dove i simboli sono equiprobabili. Dato un messaggio infatti, l'aggiunta di un simbolo corrisponde a raddoppiare il valore precedente e aggiungerne uno, ovvero aumentare di un fattore $x/\frac{1}{2}$.

Codificando ad esempio il numero 0100110 in decimale, quello che sto facendo sono operazioni del tipo $x' = 2x + p$, dove p è 0 oppure 1. Ciò è coerente con la premessa, sto aggiungendo simboli che sono equiprobabili.

[9] Necessito quindi di un valore libero per questo codice nei b bit.

Caso asimmetrico Il caso binario è semplice, codifica e decodifica si fanno mettendo e spostando caratteri, bisogna ora capire come fare una cosa simile quando la probabilità dei simboli cambia.

Siano $0, 1, \ldots, n-1$ simboli con una relativa probabilità associata p_i.

Rappresento in maniera discreta le probabilità, considerando 2^d blocchi, divisi in segmenti in base alle probabilità. Vorrei che per ogni segmento, che ha lunghezza f_i, valga $f_i/2^d \approx p_i$.

Calcolo l'inizio di ogni segmento in maniera cumulativa e lo chiamo c_i, semplicemente sommo gli $f_j, j \leq i$.

Ogni simbolo quindi ha associato un segmento di dimensione f_i, lungo in maniera proporzionale alla sua probabilità associata.

Esempio Considero tre simboli con probabilità:

$$\frac{9}{10}, \frac{1}{30}, \frac{2}{30}$$

Fisso d a 10, ottengo tre blocchi lunghi f_i rispettivamente:

$$\frac{922}{1024}, \frac{34}{1024}, \frac{68}{1024}$$

Calcolo ora i c_i, ottengo rispettivamente:

$$0, 922, 956$$

Osservazione 19. *Se estraggo a caso da 1024, usando i c_i per capire in che segmento sono finito, ottengo ogni valore quasi con le probabilità iniziali.*

Primitive Definiamo ora le primitive *push*, *sym* e *pop*:

$$push(x, s) = \big(\lfloor x/f_s \rfloor << d\big) + c_s + \big(x \ mod \ f_s\big)$$

Quindi, eseguo uno shift e metto nei bit più bassi quella quantità.

$$sym(x) = s \mid c_s \leq x \leq c_{s+1}$$

L'ultimo simbolo è quello associato agli ultimi d bit.

$$pop(x) = \langle \big(x >> d\big) \cdot f_s + \big(x \ mod \, 2^d\big) - c_s, s \rangle$$

La decodifica riporta la x a prima dell'inserimento.