Algoritmica per il web

Francesco Tomaselli

6 ottobre 2021

Indice

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.