Università degli Studi di Milano
Facoltà di Scienze Matematiche, Fisiche e Naturali

Corso di Laurea in Informatica

Studio di euristiche per il miglioramento di algoritmi di ranking per il World–Wide Web

Relatore: Dr. Sebastiano Vigna
Correlatori: Dr. Paolo Boldi
Dr. Massimo Santini

Tesi di Laurea di:
Marco Olivo
Matr. n. 592150

Anno Accademico 2002-2003

A Milano, la mia città

Desidero ringraziare anzitutto
Sebastiano Vigna, Paolo Boldi e Massimo Santini
non solo per avermi seguito in tutti questi mesi come
nessun altro avrebbe mai fatto, ma anche — anzi,
soprattutto — per avermi dato l'occasione di poter coronare un
mio grande sogno: quello di costruire un motore di ricerca.
Senza di loro questa tesi non sarebbe mai esistita...

Un grazie particolare a mio zio Gian Paolo,
per essermi stato sempre vicino e per avermi sempre trattato
da adulto, anche quando adulto non lo ero affatto.

Ed infine, ma non certo per ordine di importanza,
ringrazio i miei genitori per avermi dato tutto, sacrificando
spesso i loro sogni per accontentarmi. È grazie a voi se
sono arrivato alla Laurea, ed in tempi da record.

Grazie mamma e papà.

Indice

Capitolo 1

Introduzione

Uno dei problemi aperti dell'informatica e dell'*information retrieval* più affascinanti degli ultimi anni è quello di rispondere alle interrogazioni poste da esseri umani ai motori di ricerca.

I motori di ricerca si distinguono dalle basi di dati e dai sistemi di *information retrieval* tradizionali per diverse ragioni: il World–Wide Web (web) è immenso[1], raddoppia di dimensione ogni dodici mesi e si aggiorna con una frequenza impredicibile; inoltre i dati non sono strutturati e sono altamente eterogenei nel contenuto e nella qualità.

Il recupero di tali quantità di dati è reso difficile non soltanto dalla loro mole, ma anche da una serie di fattori legati all'affidabilità dei server su cui tali dati vengono ospitati e dalle reti attraversate per raggiungerli.

La studio dei motori di ricerca è relativamente recente, e le pubblicazioni in letteratura su questo argomento, per quanto esso sia molto dibattuto, tendono ad essere molto poche e soprattutto molto poco approfondite; tra

[1]Si calcola che, compressa, la porzione di web ad oggi raggiungibile occupi circa 50 terabyte, per un totale di oltre tre miliardi di pagine.

i motivi che spingono i ricercatori che se ne occupano a non divulgare tutti i dettagli del loro lavoro vi sono forti interessi economici legati all'uso commerciale dei motori di ricerca.

Lo scopo di questa tesi è stata la costruzione di un motore di ricerca scalabile, preciso e soprattutto flessibile che potesse competere con i motori di ricerca commerciali — se non per numero di pagine trattate, almeno per qualità dei risultati. Per raggiungere questi obbiettivi si è partiti da una implementazione accurata dei migliori algoritmi noti in letteratura, a cui è seguita una fase di affinamento basata da un lato su considerazioni di tipo ingegneristico e dall'altro su un attento confronto dei risultati ottenuti con quelli restituiti dai motori di ricerca commerciali in risposta a varie interrogazioni.

Lo schema seguito da un motore di ricerca nel momento in cui un utente inserisce un'interrogazione è, in prima approssimazione, il seguente. Anzitutto il motore opera una scrematura delle pagine che soddisfano l'interrogazione sottopostagli, in maniera tale da eliminare subito un numero significativo di pagine che probabilmente non interessano all'utente; ciononostante, a causa delle dimensioni del web, il numero di pagine rimanenti è in generale molto elevato. Pertanto, l'ordinamento relativo di tali pagine è forse la cosa più importante che un motore di ricerca si deve occupare di fornire, in quanto è proprio tale ordinamento a dare all'utente la sensazione di aver trovato quel che stava cercando.

Una tecnica possibile, ad esempio, è quella di preferire le pagine a cui si riferiscono molte altre pagine, oppure quelle in cui i termini dell'interrogazione compaiono in posizioni ravvicinate. Altre tecniche, più sofisticate, si basano sulla struttura di interconnessione tra le pagine del web.

Nei primi capitoli vengono anzitutto presentati ed analizzati alcuni algoritmi noti proposti da tempo in letteratura e che si reputa vengano utilizzati dai più famosi motori di ricerca commerciali per decidere l'ordine di presentazione dei risultati. Alcuni di questi algoritmi sono stati utilizzati nel motore di ricerca che costituisce lo scopo di questa tesi.

Nei capitoli successivi si analizzano alcune euristiche tese anzitutto a migliorare la qualità dei risultati e che cercano di fornire risposte più mirate e precise alle interrogazioni sottoposte al motore di ricerca, in maniera da emulare e, per quanto possibile, migliorare lo stato attuale dell'arte, rappresentato in questo caso dai motori di ricerca commerciali; in seconda battuta, vengono presentate altre euristiche volte a diminuire i tempi di risposta alle interrogazioni, in maniera da rendere realistico l'utilizzo del motore di ricerca sviluppato.

Infine, è stata studiata ed implementata una tecnica per aggregare i risultati dei vari algoritmi tra di loro in maniera efficiente ed in modo altamente parametrico, così da rendere possibile la sperimentazione dell'impatto dei vari algoritmi sulla qualità della risposta.

Capitolo 2

Algoritmi di ranking

Un *algoritmo di ranking* è un algoritmo che è preposto all'assegnazione di un punteggio a ciascuna delle pagine di una collezione di pagine web: tale punteggio dovrebbe riflettere la significatività di quella pagina in senso assoluto o relativo ad una *query*[1]. Talvolta, in questa tesi, saranno anche chiamati *ranker*, ossia "assegnatori di punteggi". Pur essendo possibile fornire numerose tipologie di punteggi, per quanto riguarda la ricerca su web ci si affida tipicamente ad algoritmi che si possono suddividere, *in primis*, in due famiglie:

- algoritmi su grafo;
- algoritmi sul contenuto.

La prima classe di algoritmi annovera al suo interno algoritmi quali PageRank e Hits, il primo molto usato nei motori di ricerca, come ad esempio Google [1]: essi sono noti da qualche anno in letteratura e verranno presentati

[1]Nel corso di questa tesi diremo intercambiabilmente *query* od interrogazione, intendendo lo stesso significato.

per illustrare l'attuale stato dell'arte. In questi algoritmi vi è la necessità di manipolare il grafo del web[2]: questo è possibile grazie al recupero delle pagine in rete e ad una successiva elaborazione. Questi algoritmi, proprio perché hanno come input primario il grafo, sono anche detti *algoritmi esogeni*, in quanto per stabilire il punteggio delle pagine essi si basano sulle proprietà strutturali del grafo, e non sul contenuto testuale delle pagine.

La seconda classe di algoritmi invece è stata solamente in parte trattata in letteratura; tra gli algoritmi noti si ricordano LSI e Proximity, molto utilizzati per la ricerca su web. Questa seconda classe di algoritmi per assegnare il punteggio tende a considerare il contenuto intrinseco delle pagine, e per questo motivo questi algoritmi sono spesso chiamati *algoritmi endogeni*.

Ognuno degli algoritmi che verranno illustrati si compone inoltre di due parti, una che viene precomputata, l'altra che viene calcolata al momento dell'interrogazione. Solitamente la parte che viene precomputata è anche la più costosa, mentre la parte da calcolare quando perviene l'interrogazione dovrebbe essere più rapida. In alcuni algoritmi, quali PageRank, la parte calcolata al momento dell'interrogazione è del tutto inesistente, se escludiamo l'operazione di restituire i risultati trovati.

[2]O una porzione significativa di esso. Per grafo del web, rimandando ad una definizione più formale nel paragrafo 3.1, qui intendiamo il grafo che si ottiene considerando le pagine come nodi e i link tra le pagine come archi del grafo.

Capitolo 3

Algoritmi su grafo

Come visto, una classe di algoritmi di *ranking* per funzionare necessita di disporre di un grafo del web, o di una sottoparte di esso. In questa sezione si presenteranno i più noti algoritmi di questo tipo.

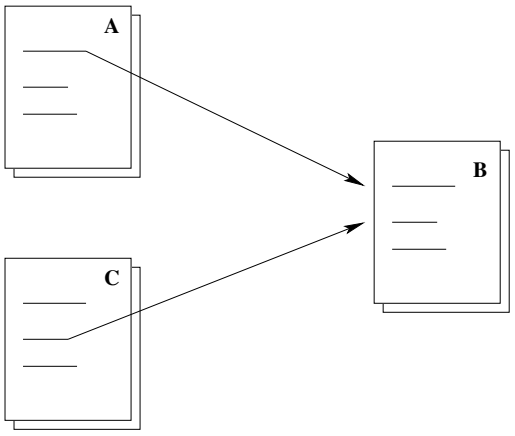

Figura 3.1: Le pagine `A` e `C` puntano a `B`.

3.1 Il grafo del web

Si consideri un qualunque insieme di pagine web: si può pensare alle pagine come a dei nodi di un grafo mentre gli archi sono rappresentati dai link con cui le pagine si puntano. Questa semplice visione del web permette di costruire un grafo a partire da un qualunque insieme di pagine recuperate tramite *crawl* e la costruzione di questo grafo permette di poter fare delle analisi riguardanti la struttura ipertestuale del web e di eseguire noti algoritmi quali PageRank e Hits, che verranno approfonditi nelle sezioni seguenti.

3.1.1 Generalità sui grafi: notazioni e definizioni

Definizione 1 *Un* grafo (orientato) $G = (V, E)$ *è dato da un insieme* V *di nodi e un insieme* $E \subseteq V \times V$ *di archi. La notazione* $x \to_G y$ *indica che c'è un arco in* G *da* x *a* y *(cioè che* $(x, y) \in E$*); se* G *è chiaro dal contesto, scriveremo semplicemente* $x \to y$*. In questo caso,* x *si chiama* sorgente *dell'arco, e* y destinazione *dell'arco. L'insieme degli archi che hanno* x *come sorgente (destinazione, rispettivamente) si indica con* $G(x, -)$ *(*$G(-, x)$*); tali archi vengono chiamati archi* uscenti *da (*entranti *in)* x*. I nodi destinazione di archi uscenti da* x *vengono chiamati* successori *di* x*; i nodi sorgente di archi entranti in* x *vengono chiamati* predecessori *di* x*. Quando non ci sia rischio di confusione, indicheremo con* $G(x, -)$ *e* $G(-, x)$ *anche l'insieme dei successori e predecessori di* x*. Un nodo senza successori è chiamato* pozzo*.*

Definizione 2 *Sia* G *un grafo, e* $x, y \in V$ *due nodi. Un* cammino (orientato) da x a y *è una sequenza* $x_0, x_1, \ldots, x_n$ *(*$n \geq 0$*) di nodi tali che* $x = x_0$*,* $y = x_n$ *e* $x_{i-1} \to x_i$ *per ogni* $i = 1, \ldots, n$*. Se esiste un cammino da* x *a* y *si dice che* y *è* raggiungibile *da* x*. L'insieme dei nodi raggiungibili da* x *si denota*

con $G^*(x, -)$; l'insieme dei nodi da cui x è raggiungibile sarà analogamente denotato con $G^*(-, x)$. Un cammino non vuoto (cioè con $n \neq 0$) da un nodo x a se stesso viene chiamato ciclo. *Se non ci sono cicli il grafo viene chiamato* aciclico.

Definizione 3 *Sia G un grafo, e $x, y \in V$ due nodi; diciamo che i due nodi sono* coraggiungibili *nel grafo sse $x \in G^*(y, -)$ e $y \in G^*(x, -)$, cioè sse x è raggiungibile da y e viceversa. È banale dimostrare che la relazione di coraggiungibilità è una relazione di equivalenza fra nodi. Le classi di equivalenza di questa relazione vengono chiamate* componenti (fortemente) connesse *del grafo G. Se c'è un'unica componente fortemente connessa diremo che G è fortemente connesso.*

Definizione 4 *Sia G un grafo. Definiamo un nuovo grafo G' come segue:*

- *i nodi di G' sono le componenti fortemente connesse di G;*
- *c'è un arco dal nodo c_1 al nodo c_2 sse esiste un nodo x_1 nella classe c_1 e un nodo x_2 nella classe c_2 tali che $x_1 \to_G x_2$.*

Il grafo G' viene chiamato grafo ridotto *di G.*

È facile verificare che il grafo G' è aciclico.

3.1.2 Grafi e processi stocastici

Definizione 5 *Una* matrice stocastica *è una matrice quadrata M con le seguenti proprietà:*

- *gli elementi della matrice sono reali nell'intervallo $[0, 1]$;*

- *la somma degli elementi di ciascuna riga è* 1.

Indichiamo con m_{ij} l'elemento di M che si trova sulla i-esima riga e sulla j-esima colonna. Si può pensare a una matrice stocastica come alla descrizione di un processo markoviano finito di memoria uno (cioè, una catena di Markov finita): m_{ij} rappresenta la probabilità che il sistema passi nello stato j quando si trova nello stato i [2].

Indichiamo con M^n l'n-esima potenza della matrice M (cioè, M moltiplicata con se stessa n volte); con $m_{ij}^{(n)}$ indicheremo l'elemento di indici (i, j) nella matrice M^n. Nella precedente interpretazione $m_{ij}^{(n)}$ è la probabilità che il sistema raggiunga lo stato j in n passi se parte dallo stato i.

Definizione 6 *Una matrice stocastica M è detta:*

- irriducibile[1] *sse per ogni i, j esiste un $n > 0$ tale che $m_{ij}^{(n)} > 0$;*
- aperiodica *sse per ogni i, il massimo comun divisore di $\{n \geq 1 \mid m_{ii}^{(n)} > 0\}$ è 1.*

Definizione 7 *Sia M una matrice stocastica di dimensione n, e $\vec{p}$ una distribuzione di probabilità su n stati (cioè, un vettore riga n-dimensionale con componenti in $[0, 1]$ aventi somma 1). Diremo che $\vec{p}$ è una* distribuzione stabile *per M sse $\vec{p} \cdot M = \vec{p}$. Si noti che se la catena è finita esiste sempre una distribuzione stabile: l'equiprobabile.*

Teorema 1 *Se M è una matrice stocastica irriducibile ed aperiodica, allora:*

1. *M ammette un'unica distribuzione stabile $\vec{p}$;*

[1] Si noti che, se si chiama $G = (V, E)$ il grafo che ha per nodi gli stati della catena dove $(i, j) \in E$ sse $m_{ij} > 0$, allora l'irriducibilità corrisponde a chiedere che G sia fortemente connesso.

2. *tale distribuzione soddisfa*

$$p_j = \lim_{n\to\infty} m_{ij}^{(n)};$$

3. *la convergenza al limite avviene con velocità esponenziale, cioè esistono* $c \geq 0$ *e* $\rho < 1$ *tali che*

$$|p_j - m_{ij}^{(n)}| \leq c \cdot \rho^n;$$

4. *per ogni distribuzione iniziale* $\vec{q}$*,*

$$p_j = \lim_{n\to\infty} m_{ij}^{(n)} \cdot q_j.$$

5. *Se indichiamo con* $N_{\vec{q}}^{(n)}(i)$ *il numero di volte che il processo (a partire dalla distribuzione iniziale* $\vec{q}$*) si trova nello stato* i *durante i primi* n *passi, allora*

$$p_i = \lim_{n\to\infty} \frac{N_{\vec{q}}^{(n)}(i)}{n}$$

per ogni $\vec{q}$*.*

Si noti che, in uniformità con l'argomento trattato, nel corso di questa tesi i nodi del grafo verranno chiamati *pagine* mentre gli archi entranti (*uscenti*, rispettivamente) verranno chiamati *link entranti* (uscenti).

3.1.3 Grafo: rappresentazione e compressione

Da un punto di vista della rappresentazione, il grafo del web ammette diverse rappresentazioni. Tuttavia, scelte come, ad esempio, la rappresentazione dell'intera matrice di adiacenza sono del tutto fuori luogo, dal momento che il

grafo del web è tendenzialmente sparso poiché le pagine sono poco collegate tra di loro direttamente (il grado uscente medio di una pagina web, secondo alcuni studi, è pari a sette[2]).

Si rendono pertanto necessarie altre tecniche basate direttamente sulle proprietà intrinseche del web: ad esempio, al posto della matrice (che occuperebbe $O(n^2)$ in spazio, dove n è il numero di nodi) si possono utilizzare liste di adiacenza, decisamente più compatte. È stata proprio questa la scelta che abbiamo deciso di seguire.

Va tuttavia tenuto conto che anche questa tecnica non scala ragionevolmente con l'aumentare delle dimensioni della collezione di pagine a disposizione: infatti con appena sedici milioni di pagine web recuperate dalla porzione di web italiano il grafo rappresentato in questa maniera arriva già ad essere quasi un gigabyte di dimensione, e il suo caricamento in memoria inizia a diventare particolarmente problematico, in virtù anche di alcuni limiti di cui soffre Java, il linguaggio scelto per l'implementazione[3].

Si è pertanto reso necessario comprimere il grafo del web in maniera tale che l'occupazione su disco ma, in particolare, quella in memoria una volta caricato, fosse il più possibile ridotta, sacrificando in parte anche la velocità di scansione di una particolare lista di adiacenza di un dato nodo.

La compressione avviene sostanzialmente su due diversi fronti: il primo è l'ordinamento alfabetico delle URL trovate, il secondo è rappresentato dalla compressione vera e propria.

Le URL recuperate, prima di creare il grafo, vengono ordinate alfabeticamente, al fine di sfruttare il *principio di località*. Si tenga anzitutto conto

[2]Dai nostri esperimenti, fatti peraltro su un insieme di pagine ridotto, tale valore risulta essere pari a circa il doppio.

[3]Limiti che verranno evidenziati in altri capitoli.

del fatto che, per le tecniche che verranno illustrate in seguito, sequenze non decrescenti di numeri con differenze piccole si comprimono meglio. Tenendo conto di questa osservazione, si rifletta ora sul fatto che, in media, il maggior numero di link uscenti da una pagina punta verso pagine che appartengono allo stesso *host*, e che quindi — dal punto di vista dell'URL — sono alfabeticamente molto vicini, in quanto non solo lo *host* è in comune ma anche, con buona probabilità, una parte del suffisso, ossia del percorso all'interno dello *host* in cui si trovano le due pagine. Solamente una piccola percentuale di link uscenti punta verso pagine di altri *host*. Gli indirizzi delle pagine puntate saranno quindi presumibilmente alfabeticamente lontani dall'indirizzo della pagina che punta. Ordinando le pagine alfabeticamente, si ottiene proprio l'effetto di mettere vicine le pagine che con più probabilità hanno link tra di loro e si massimizza anche la percentuale di compressione ottenibile tramite le tecniche utilizzate successivamente. Per maggiori dettagli riguardanti l'ordinamento alfabetico si veda [3].

Le tecniche di compressione vere e proprie, come visto, non possono prescindere dal fatto che le URL che compongono il grafo sono state precedentemente ordinate alfabeticamente. Le tecniche utilizzabili per sfruttare la compressibilità indotta dall'ordinamento sono molte, e sono state previste opportune opzioni all'interno degli strumenti sviluppati per scegliere quale compressione adottare, sia per quanto riguarda l'indicazione del grado uscente, sia per quanto riguarda la vera e propria lista di adiacenza. Tra le tecniche previste vi sono la compressione δ, la compressione γ, la compressione skewed-Golomb e la tecnica interpolativa.

Viene fornito un breve riassunto riguardante i risultati ottenuti tramite le tecniche di compressione da noi implementate: il primo grafo si riferisce ad una *crawl* di 18 520 486 pagine aventi come estensione `.uk`, per un totale

cod. grado	cod. link uscenti	dimensione	bit/grado	bit/link
Gamma	Gamma	343 760 629	6.29	9.00
Gamma	Delta	295 912 449	6.29	7.69
Gamma	Interpolativa	313 770 704	6.29	8.18
Gamma	Skewed-Golomb	345 213 821	6.29	9.04
Delta	Gamma	344 154 250	6.46	9.00
Delta	Delta	296 306 071	6.46	7.69
Delta	Interpolativa	314 164 325	6.46	8.18
Delta	Skewed-Golomb	345 607 443	6.46	9.04

Figura 3.2: Compressione del grafo di `.uk`: la tabella riporta, per ogni codifica utilizzata, la dimensione del grafo (in byte) e il numero medio di bit utilizzati per grado uscente e per link uscente.

di 298 420 812 link; il secondo invece si riferisce ad una *crawl* di 16 221 344 pagine aventi come estensione `.it`, per un totale di 234 573 373 link.

Entrambi i grafi non compressi occuperebbero all'incirca tra i 700 megabyte ed un gigabyte ciascuno[4].

Tra le altre opzioni e gli altri strumenti sviluppati si ricordano la possibilità di rimuovere i *link navigazionali*, ossia i link *intra-host* che alcuni algoritmi non considerano (come ad esempio Hits), e la possibilità di trasporre il grafo al fine di avere, invece del grado uscente e della lista delle pagine puntate, il grado entrante e le pagine che puntano alla pagina considerata.

[4]Questo conto viene fatto sulla base del fatto che la struttura non compressa dovrebbe immagazzinare in 16 bit il numero di link uscenti dalla pagina, seguito da una lista di interi a 32 bit per indicare le pagine puntate.

cod. grado	cod. link uscenti	dimensione	bit/grado	bit/link
Gamma	Gamma	290 480 748	5.81	9.69
Gamma	Delta	243 615 121	5.81	8.06
Gamma	Interpolativa	267 960 478	5.81	8.74
Gamma	Skewed-Golomb	285 608 041	5.81	9.52
Delta	Gamma	290 834 130	5.91	9.69
Delta	Delta	243 968 504	5.91	8.06
Delta	Interpolativa	258 231 830	5.91	8.74
Delta	Skewed-Golomb	285 961 424	5.91	9.52

Figura 3.3: Compressione del grafo di `.it`: la tabella riporta, per ogni codifica utilizzata, la dimensione del grafo (in byte) e il numero medio di bit utilizzati per grado uscente e per link uscente.

3.2 L'algoritmo di PageRank

Uno degli algoritmi più largamente utilizzati e conosciuti nell'ambito del problema del ranking di documenti — e più in generale di pagine web - è l'algoritmo di PageRank. Tale algoritmo [4] è divenuto largamente conosciuto non solo grazie alla sua efficacia ma anche grazie alla sua implementazione nel famoso motore di ricerca Google [1].

L'idea di fondo di PageRank è che una pagina è rilevante se è puntata da altre pagine rilevanti. Quel che l'algoritmo di PageRank aggiunge a questa nozione estremamente intuitiva è l'idea di pesare quanto un collegamento, ossia un *link*, possa contribuire all'importanza della pagina puntata.

L'algoritmo di PageRank si basa dunque sul fatto che una pagina non è importante perché pretende di esserlo, ma è importante perché le altre pagine dicono che lo è. L'algoritmo di PageRank è pertanto un algoritmo esogeno.

Questa idea non è del tutto nuova: in letteratura erano già noti da tempo algoritmi simili per valutare la rilevanza di pubblicazioni scientifiche a partire dalle citazioni bibliografiche (si veda ad esempio [5]). Con PageRank,

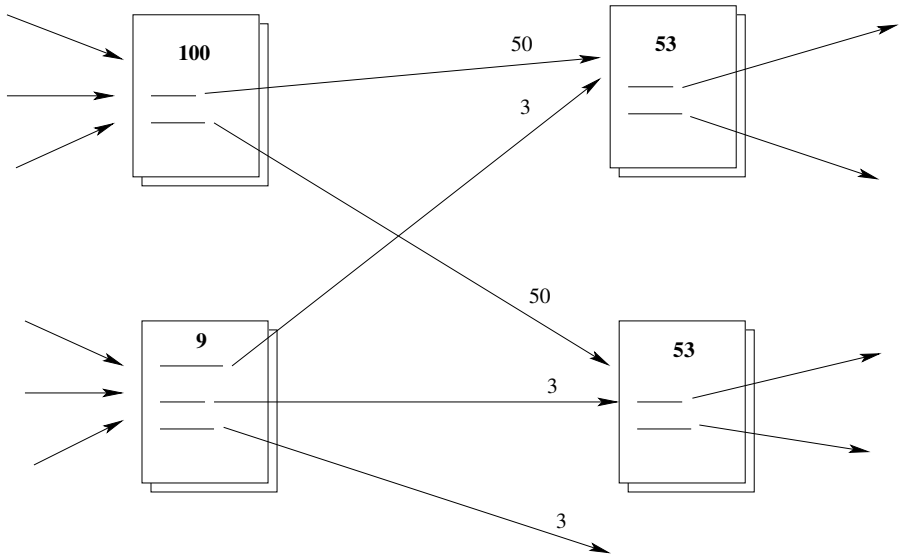

Figura 3.4: Calcolo semplificato di PageRank (prima figura).

analogamente a quanto illustrato in [6], si ha il concetto di peso di una citazione: se una pagina è importante, allora un suo link contribuirà in maniera significativa all'importanza della pagina puntata. E una pagina è importante proprio perché è puntata da altre. Un semplice conteggio dei link entranti, ossia del numero di link che puntano verso quella data pagina, non è sufficiente in quanto può essere facilmente manipolato ed inoltre non tiene conto della natura del web. Si noti che il processo descritto funziona anche per quelle pagine che hanno pochi link provenienti da pagine importanti. Infatti, mentre il semplice algoritmo di conteggio delle citazioni attribuirebbe poca importanza ad una pagina poco puntata, PageRank attribuisce una certa rilevanza ad una pagina che ricada nel caso descritto. Un tipico esempio è quello di una pagina che è puntata solamente da una pagina di Yahoo [7]. In tal caso infatti a questa pagina verrà attribuito un rank maggiore rispetto a quello che le verrebbe attribuito se il suo unico link entrante provenisse, ad esempio, da una pagina poco nota o sconosciuta del tutto (si veda [8]).

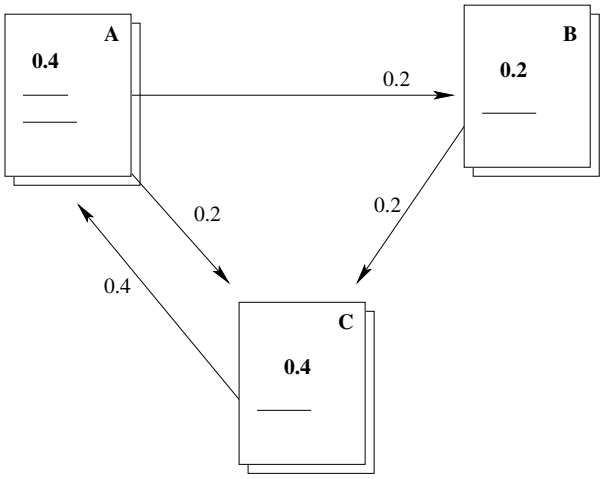

Figura 3.5: Calcolo semplificato di PageRank (seconda figura).

L'algoritmo stabilisce un ordine statico delle pagine del web: infatti, una volta che l'algoritmo di PageRank sia giunto a convergenza, ad ogni pagina viene assegnato un punteggio che determina, in maniera assoluta, quanto quella pagina sia importante.

Si noti che l'algoritmo di PageRank è garantito convergere sotto le ipotesi che la matrice di adiacenza del grafo sia stocastica, irriducibile ed aperiodica. L'ipotesi di aperiodicità nella pratica è garantita, in quanto la probabilità che tutte le pagine del web formino un ciclo è pressoché nulla. Per quanto riguarda invece l'ipotesi di irriducibilità, si rimanda al paragrafo 3.2.3.

Per quanto questo algoritmo sia efficace, esso soffre di una serie di problemi che si evidenziano nel paragrafo seguente.

3.2.1 Problemi di PageRank

PageRank purtroppo soffre di qualche limite che fa sì che il suo utilizzo debba sempre essere accompagnato da un qualche altro algoritmo da com-

putare al momento dell'interrogazione. Infatti, come si è visto, PageRank è un algoritmo totalmente precomputato: una volta che si è recuperata una porzione significativa di pagine web, è possibile eseguire tale algoritmo sull'archivio e disporre di un ordine di importanza delle pagine. Tuttavia proprio questo implica che qualunque sia l'interrogazione che un utente può immettere nel motore di ricerca, i risultati saranno sempre gli stessi: tutte le pagine del web, ordinate in base a PageRank. In altre parole, dire che `http://www.corriere.it/` è rilevante in assoluto non ha alcun senso se l'utente sta cercando informazioni su un argomento su cui il noto quotidiano non è assolutamente rilevante — ad esempio le parole crociate, argomento per il quale "Il corriere della sera" non è di certo rinomato (pur magari contenendo accenni all'argomento) ma per il quale il sito de "La settimana egnigmistica" sarebbe decisamente più appropriato, pur non essendo altrettanto famoso. Per ovviare a questo inconveniente occorre affiancare qualche altro algoritmo o qualche altra euristica a PageRank: un esempio può essere rappresentato da un algoritmo di matching testuale che potrebbe selezionare le pagine in base alla presenza o meno delle parole dell'interrogazione all'interno delle pagine. Le pagine soddisfacenti l'interrogazione potrebbero poi essere ordinate per valore di PageRank decrescente.

Un altro problema di PageRank, presente comunque in maniera decisamente minore, è quello dello *spamming*. Ricordiamo che per *spam*, nel contesto dei motori di ricerca, si intende il tentativo (riuscito o meno, voluto o meno) di posizionare una data pagina in posizione più prominente nell'indice. Tipicamente tale tecnica consiste nello sfruttare debolezze intrinseche dell'algoritmo che viene applicato, e per combattere questo fenomeno si usano spesso sofisticate tecniche *anti-spam* che per ovvie ragioni non vengono divulgate, oppure semplici tecniche di compilazione manuale di *blacklist*, ossia di

"libri neri" nei quali i siti che "fanno *spam*" vengono inseriti. Per manipolare PageRank sarebbe sufficiente inserire dei link da molte pagine — o da poche pagine rilevanti — verso la pagina che si intende manipolare, ed automaticamente l'algoritmo di PageRank sarebbe indotto a credere che tale pagina sia realmente importante. Tuttavia ciò è molto difficile che accada per varie ragioni:

- fare inserire alcuni link da pagine rilevanti o molti link da pagine poco rilevanti non è un'operazione economicamente fattibile nella stragrande maggioranza dei casi;
- esistono euristiche per abbattere il fenomeno dello *spamming* sia direttamente a tempo di *crawl* — ossia quando le pagine vengono recuperate — sia a tempo di indicizzazione, ossia quando si calcolano PageRank e gli altri indici sull'archivio ottenuto dalla *crawl*;
- l'algoritmo di PageRank non è l'unico algoritmo ad essere solitamente applicato: la manipolazione dovrebbe coinvolgere pertanto anche gli altri algoritmi in uso dal motore, compreso le euristiche *anti-spam*: questo non rende la manipolazione impossibile, ma la rende sicuramente molto più difficile.

3.2.2 PageRank su un insieme ridotto di pagine

La determinazione delle dimensioni del grafo del web è un problema aperto e di difficile soluzione. Al momento in cui questa tesi viene scritta, Google afferma di disporre nel suo archivio di oltre 3 miliardi di pagine web. Un tale numero di pagine, in costante crescita mese dopo mese e anno dopo anno, rappresenta un notevole ostacolo per chi desideri cimentarsi con il proposito

di recuperare tutte le pagine raggiungibili del web. Per la determinazione del PageRank di una pagina è però necessario conoscere i link entranti di quella pagina; pertanto, per avere una stima ragionevole sull'importanza di una pagina è anche necessario aver recuperato una porzione significativa del grafo del web. In caso contrario, l'algoritmo di PageRank pur convergendo potrebbe portare a risultati che non rispecchiano l'idea di importanza delle pagine che il tipico navigatore ha. In generale, comunque, anche basandosi su un archivio di poche decine di milioni di pagine, l'algoritmo di PageRank riesce ugualmente a determinare in maniera ragionevolmente accurata i punteggi associati a ciascuna pagina.

3.2.3 Propagazione del ranking e convergenza

In base alla discussione appena fatta, si può dare la seguente descrizione di PageRank: una pagina v ha un punteggio elevato se la somma dei punteggi delle pagine che la puntano (ossia l'insieme $G(-, v)$) è elevato. Questa definizione copre sia il caso in cui una pagina abbia pochi link entranti ma di punteggio elevato, sia quando una pagina ne abbia molti ma di punteggio basso.

Sia u una pagina web. Sia $N_u = |G(u, -)|$ il numero di link uscenti da u. Iniziamo con il definire una condizione che la funzione di punteggio R dovrebbe verificare:

$$R(u) = \sum_{v \to u} \frac{R(v)}{N_v} \tag{3.1}$$

Questa semplice espressione formalizza l'intuizione precedentemente esposta. Si noti che il punteggio di una pagina viene diviso tra i suoi successo-

ri equamente, in modo tale da contribuire al punteggio di tutte le pagine puntate.

L'equazione data definisce in realtà la distribuzione stabile di una catena di Markov avente le pagine come stati, e la probabilità di transizione tra u e v definita come:

$$m_{uv} = \begin{cases} \frac{1}{N_u} & \text{se } u \to v \\ 0 & \text{altrimenti,} \end{cases} \tag{3.2}$$

assumendo che non ci siano pagine senza link uscenti. Quindi, R può essere facilmente calcolato iterativamente a partire da un vettore qualunque di PageRank (solitamente, un vettore costante); sotto ipotesi di irriducibilità ed aperiodicità della matrice stocastica associata al processo, si può essere sicuri che la computazione prima o poi termini (anche se nella pratica, invece di attendere la convergenza, si danno solitamente delle *condizioni di arresto* basate, ad esempio, sulla norma del residuo tra due iterazioni successive oppure sul numero di iterazioni dell'algoritmo).

L'aperiodicità viene assunta come dato di fatto a causa delle grandi dimensioni del web e della sua struttura irregolare. La stessa assunzione non è però ragionevole per l'irriducibilità. Ci sono infatti situazioni nelle quali la convergenza può essere disturbata dalla mancata connessione forte del grafo in esame.

In particolare, è necessario prendere in considerazione i *rank-sink*, ovvero i pozzi del grafo ridotto, così chiamati perché assorbono importanza senza mai restituirla al sistema (la riducibilità di una catena corrisponde infatti alla non banalità del grafo ridotto che le soggiace). L'idea è quella di aggiungere un lato fittizio uscente da ogni pagina verso ogni altra pagina del web non

puntata da essa. Nella pratica, ciò si ottiene grazie all'introduzione di una *sorgente di rank*.

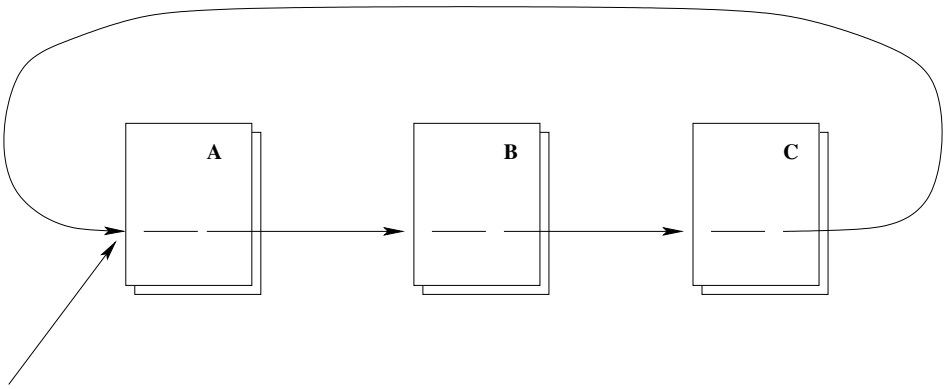

Figura 3.6: Un insieme di pagine che rappresenta un *rank-sink*.

Sia $E(u)$ un vettore sulle pagine web che corrisponde ad una qualche sorgente di rank e sia α un numero utilizzato per la normalizzazione (così che la somma del punteggio di tutte le pagine sia costante); quest'ultimo prende il nome di *fattore di spargimento*. Allora, il PageRank di un insieme di pagine web è un assegnamento R' che soddisfa l'equazione:

$$R'(u) = \alpha \cdot \sum_{v \to u} \frac{R'(v)}{N_v} + (1 - \alpha) \cdot E(u) \tag{3.3}$$

così che $\|R'\| = 1$.

Questa formula garantisce che la matrice stocastica associata al processo sia irriducibile, sotto l'ipotesi che il vettore E sia a componenti positive (in caso contrario è da valutare se l'assegnamento scelto soddisfa l'irriducibilità della matrice associata). Tuttavia, il problema dei pozzi nel grafo G originario non è stato affrontato.

Si noti che un pozzo nel grafo del web può essere causato da varie ragioni:

- la pagina non ha effettivamente link uscenti;
- la pagina ha link uscenti ma le pagine puntate non sono state recuperate;
- la pagina ha uno o più link uscenti ma le pagine puntate hanno dato errore durante la fase di *crawling*.

Per risolvere il problema, occorre aggiungere un addendo nella formula appena vista. Infatti, se applichiamo ad una distribuzione S la formula appena vista, otteniamo un nuovo vettore le cui componenti hanno somma

$$\sum_u \left[\alpha \cdot \sum_{v \to u} \frac{S(v)}{N_v} + (1-\alpha) \cdot \frac{1}{N} \right] = \alpha \cdot \sum_u \sum_{v \to u} \frac{S(v)}{N_v} + (1-\alpha). \tag{3.4}$$

Dato che, per ogni pagina v con almeno un link uscente, il fattore $S(v)/N_v$ viene sommato per ciascun arco uscente (ce ne sono N_v in tutto), il tutto si può scrivere come

$$\alpha \cdot \sum_{G(v,-) \neq \varnothing} S(v) + (1-\alpha), \tag{3.5}$$

e, come si vede, la somma rimane pari a 1 se e solo se non ci sono pozzi, altrimenti la somma sarà minore di 1. Diventa quindi necessaria l'introduzione di un addendo che assicuri che la matrice del processo rimanga stocastica anche in presenza di pozzi nel grafo originario.

La formula definitiva, che tiene conto anche dei pozzi, è la seguente:

$$R'(u) = \alpha \cdot \sum_{v \to u} \frac{R'(v)}{N_v} + (1 - \alpha) \cdot E(u) + \alpha \cdot \sum_{G(v,-)=\varnothing} R'(v) \tag{3.6}$$

dove $\sum_{G(v,-)=\varnothing} R'(v)$ è il punteggio che sarebbe accumulato dai link uscenti dalla pagina v ma che viene invece perso a causa del fatto che le pagine puntate da tali link non appartengono al grafo del web che si ha a disposizione, mentre R' è tale che $\|R'\| = 1$.

La convergenza di PageRank, quindi, deriva sostanzialmente dalla convergenza del processo stocastico sottostante e la velocità della convergenza dipende dalla velocità esponenziale della convergenza del processo stocastico.

3.2.4 Il modello del navigatore casuale

La definizione di PageRank appena data può essere interpretata come una passeggiata casuale su grafo. Intuitivamente, si può pensare ad un navigatore che casualmente clicca su uno dei link presenti nella pagina web che sta visitando. Il tempo che questo navigatore virtuale si trova a passare su una certa pagina corrisponde in realtà alla probabilità stazionaria della pagina web in questione. Tuttavia, quando un navigatore reale si trova a finire in un *rank-sink*, probabilmente esso cambierà completamente pagina saltando nel grafo. Per modellare tale evento, si introduce il fattore $E(u)$ appena visto: il navigatore virtuale, periodicamente, con una certa probabilità salta in un altro punto del grafo del web. Come verrà illustrato in seguito, la scelta del vettore $E(u)$ induce valori differenti di PageRank (per tale motivo questo fattore viene solitamente chiamato *vettore di personalizzazione*).

Il vettore dei punteggi è in effetti la probabilità stazionaria del processo

che, detto in altro modo, è per ogni stato il tempo medio che il navigatore casuale passa in quello stato.

3.2.5 Personalizzazione di PageRank

Come precedentemente illustrato, il vettore E viene introdotto per contrastare gli effetti della perdita di punteggi dovuti ai pozzi: tale vettore è un vettore su tutte le pagine web che funge così da sorgente di rank. Oltre a questo utilizzo, il vettore E è però anche un parametro che si può utilizzare per ottenere dei vettori di PageRank personalizzati. Per avere una spiegazione di questo fatto bisogna ricorrere al modello del navigatore casuale prima introdotto. Come abbiamo visto, tale vettore è utile per permettere al navigatore di uscire dai pozzi o anche solo per cambiare completamente pagina quando il navigatore virtuale lo ritiene. Ovviamente, se tale vettore venisse preso uniforme (cosa che solitamente accade) allora la probabilità di passare ad una pagina piuttosto che ad un'altra sarebbe la stessa: in questo senso si può dire che un vettore uniforme rappresenta il vettore di PageRank più democratico in assoluto, in quanto tutte le pagine hanno la stessa probabilità di essere scelte soltanto per il fatto di esistere.

Un'altra scelta del vettore E di personalizzazione è quella di far sì che il navigatore possa muoversi verso una sola pagina (diciamo v), ponendo:

$$E(u) = \begin{cases} 1 & \text{se } u = v \\ 0 & \text{altrimenti.} \end{cases} \tag{3.7}$$

Con questa scelta, eseguendo l'algoritmo di PageRank si ottiene che l'unica pagina inserita nel vettore E di personalizzazione avrà un più alto valore di PageRank rispetto a quello che avrebbe con il vettore uniforme. Ciò che

è interessante è che l'inserimento in E di una pagina di un certo argomento tende a pesare maggiormente le pagine che appartengono a quell'argomento; questo fatto, apparentemente strano, è facilmente spiegabile se si considera che una pagina che tratta un certo argomento tenderà ad avere link verso pagine che trattano di quell'argomento, e così via. Scegliendo pertanto una di esse da inserire in E, si avrà un peso maggiore delle pagine ad essa correlate.

Come verrà illustrato in seguito, sono state usate anche altre tecniche per ottenere dei vettori pesati di PageRank [9]; tuttavia, anche seguendo la strada indicata da Brin e Page è possibile ottenere risultati interessanti, come si può vedere in [10].

3.2.6 Calcolo veloce di PageRank

L'algoritmo di PageRank non è di per sé un algoritmo lento, ma quando viene eseguito su un insieme di nodi molto grande come, ad esempio, può essere una parte significativa del web, può diventare piuttosto costoso in termini di tempo (la sua velocità di convergenza è esponenziale nel numero di iterazioni). In tali circostanze, specialmente se calcolato su una macchina singola, anche un algoritmo efficiente e molto ben scalabile diventa infatti presto quasi ingestibile. Per tale motivo Haveliwala [11] ha proposto un metodo per il calcolo efficiente del vettore di PageRank.

Implementazione naïf

Analizziamo in prima istanza l'algoritmo classico con il quale si può calcolare il vettore di PageRank su un insieme di pagine. Il grafo del web può venire immagazzinato ad esempio nel seguente formato: un intero da 32 bit per indicare il numero del nodo, un intero da 16 bit (o anche 32, se lo spazio non è

un limite) per indicare il grado uscente del nodo, ed una serie di interi a 32 bit per indicare i successori del nodo, ossia le pagine a cui punta il documento in questione, specificato dal primo indice visto. Si veda ad esempio la figura 3.7.

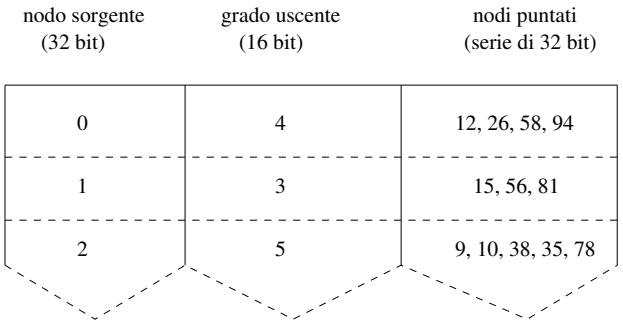

Figura 3.7: Il formato classico del grafo.

Una classica implementazione di PageRank prevede la creazione di due vettori di numeri in virgola mobile, chiamati *sorgente* (*Source*) e *destinazione* (*Dest*) concettualmente legati dal fatto che il vettore di destinazione è ottenuto dalla matrice di adiacenza del grafo moltiplicata, righe per colonne, con il vettore sorgente (i due vettori hanno come dimensione il numero di nodi presenti nel grafo).

Questi due vettori sono in generale piuttosto voluminosi e tendono ad occupare anche parecchie centinaia di megabyte di spazio in memoria anche per grafi piccoli.

Una singola iterazione dell'algoritmo di PageRank prende in input i valori contenuti nel vettore sorgente e calcola il vettore destinazione come visto. Nelle iterazioni successive il vettore che prima era di destinazione diventerà quello sorgente e verrà computato un nuovo vettore di destinazione, e così via.

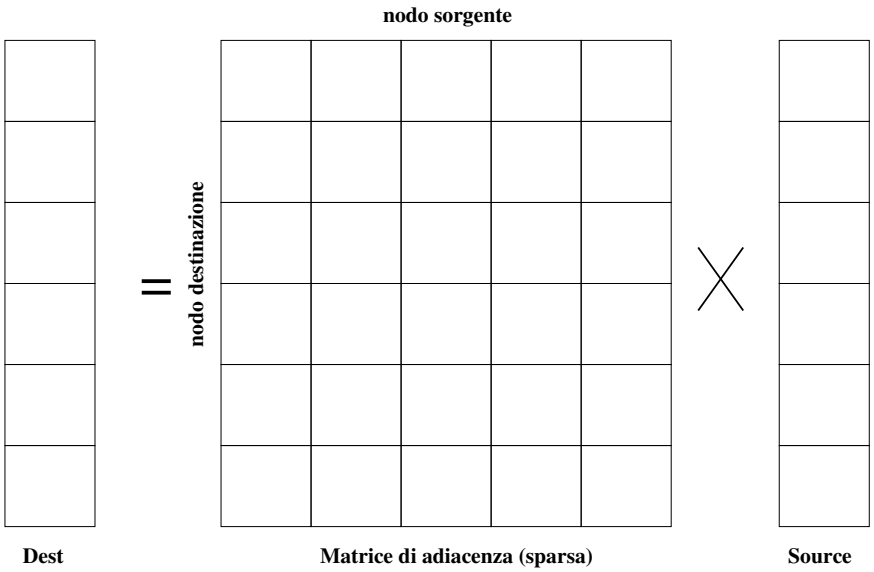

Figura 3.8: Una tipica iterazione di PageRank naïf.

Un abbozzo dell'algoritmo di PageRank risulta pertanto essere quello riportato in figura 3.9 (con notazioni ovvie).

Quel che viene fatto nella pratica è utilizzare i valori contenuti in *Source* per calcolare i valori che verranno salvati in *Dest*. L'iterazione è garantita convergere, e nella pratica ci si ferma solitamente quando la norma del residuo tra due iterazioni successive scende al di sotto di una certa soglia, oppure quando si è raggiunto un prefissato numero di iterazioni.

Assumendo che la memoria dell'elaboratore sia sufficiente per mantenere sia *Source* che *Dest*, il costo delle operazioni di I/O dell'algoritmo illustrato è dato da:

$$C = |Links|. \tag{3.8}$$

Se la memoria fosse invece sufficiente per contenere solamente il vettore

for $s = 1, 2, \cdots, n$ $Source[s] = \frac{1}{N}$
while $(residual > \tau)$
  for $d = 1, 2, \cdots, n$ $Dest[d] = 0$
  while not $Links.eof()$
    $Links.read(source, n, dest_1, \cdots, dest_n)$
    for $j = 1, 2, \cdots, n$ $Dest[dest_j] = Dest[dest_j] + \frac{Source[source]}{n}$
  for $d = 1, 2, \cdots, n$ $Dest[d] = c \cdot Dest[d] + \frac{1-c}{N}$
  $residual = \|Source - Dest\|$
  $Source = Dest$
return $Source$

Figura 3.9: L'algoritmo naïf per il calcolo di PageRank.

Dest allora il costo sarebbe pari a:

$$C = |Source| + |Dest| + |Links|. \tag{3.9}$$

Source dovrebbe infatti essere letto sequenzialmente dal disco durante il passo che permette la propagazione del punteggio, mentre *Dest* dovrà venire scritto sul disco perché servirà nell'iterazione successiva.

Le pecche di questa implementazione si vedono proprio nel secondo caso, che si presenta quando, ad esempio, si dispone di elaboratori con poca memoria oppure — caso decisamente più interessante — quando si è recuperato un ampio sottografo del web. In tale ipotesi infatti l'accesso al vettore *Source* non è computazionalmente costoso, in quanto avviene in modo lineare e può essere facilmente *bufferizzato*. Il problema sorge invece nell'accesso al vettore *Dest*, in quanto esso viene acceduto casualmente.

Implementazione efficiente

Un modo per calcolare PageRank in maniera efficiente è quello fornito dalla cosiddetta *strategia a blocchi*.

L'idea è quella di partizionare l'insieme *Dest* in β blocchi, per l'appunto, ciascuno di D pagine, come illustrato in figura

Una tipica iterazione sarà quindi come in figura:

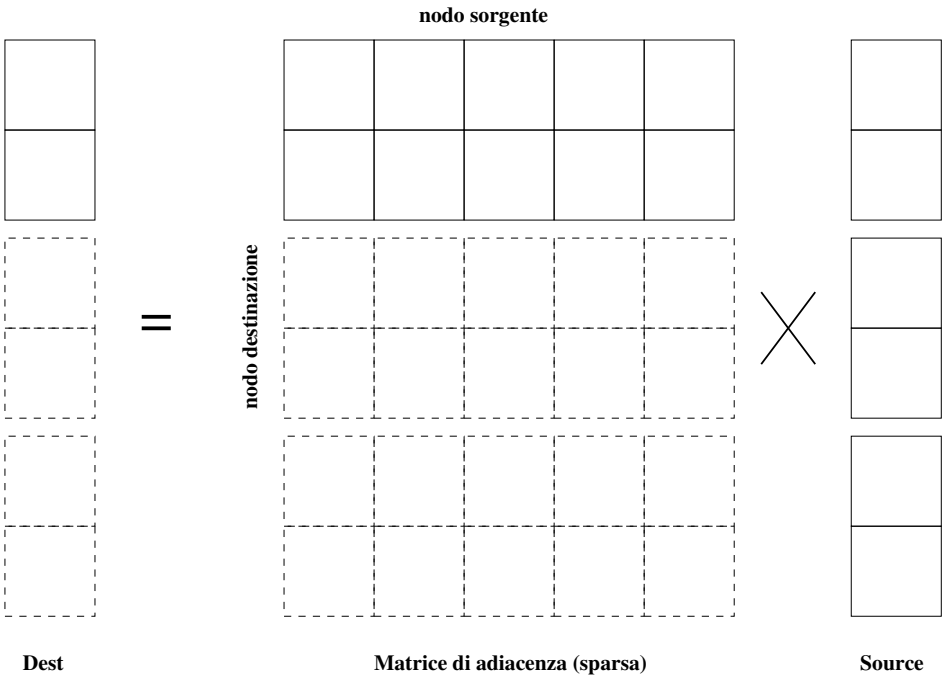

Figura 3.10: Una tipica iterazione di PageRank efficiente.

Se indichiamo con P il numero massimo di pagine che possono trovare posto nella memoria disponibile per il nostro processo, allora desideriamo che $D \leq P - 2$, dal momento che dobbiamo lasciare dello spazio per poter leggere *Source* e *Links*. Il file dei link, *Links*, dovrà di conseguenza essere modificato per riflettere le nuove scelte. Partizioneremo pertanto *Links* in β

file $Links_0$, $Links_1$, ... $Links_{\beta-1}$ in maniera tale che il campo *destinazione* in $Links_i$ contenga solamente quei nodi *dest* tali che

$$\beta \cdot i \leq dest \leq \beta \cdot (i+1). \tag{3.10}$$

In altre parole i link uscenti di un nodo sono raccolti insieme tra di loro in funzione del *range* nel quale il numero di nodo puntato ricade. Deve ovviamente risultare che $Links = \bigcup_i Links_i$, altrimenti si perderebbero alcuni link con il procedimento.

Risulta però ovviamente anche che $\sum_i |Links_i| > |Links|$ a causa dell'ulteriore *overhead* causato dalla moltiplicazione dell'informazione riguardante il numero di nodo sorgente ed il numero dei link uscenti all'interno di ciascuna delle partizioni.

Si può quindi definire l'algoritmo a blocchi per il calcolo del vettore di PageRank nella maniera riportata in figura 3.12

Dal momento che $Links_i$ è ordinato in base al campo *source*, ogni passata su $Links_i$ richiede soltanto una scansione sequenziale di $Source$. In questo modo si riesce ad usare esattamente la memoria a disposizione senza dover ricorrere allo *swap*, cosa che invece accadeva nel caso visto in precedenza quando la memoria non era sufficiente a contenere i vettori.

Il costo dello *swap* è solitamente più elevato del costo addizionale introdotto da questa variante e pertanto il vantaggio in tempo dato dall'adozione di questa soluzione può, secondo Haveliwala, essere comunque significativo e giustificare la maggiore complessità in spazio e tempo.

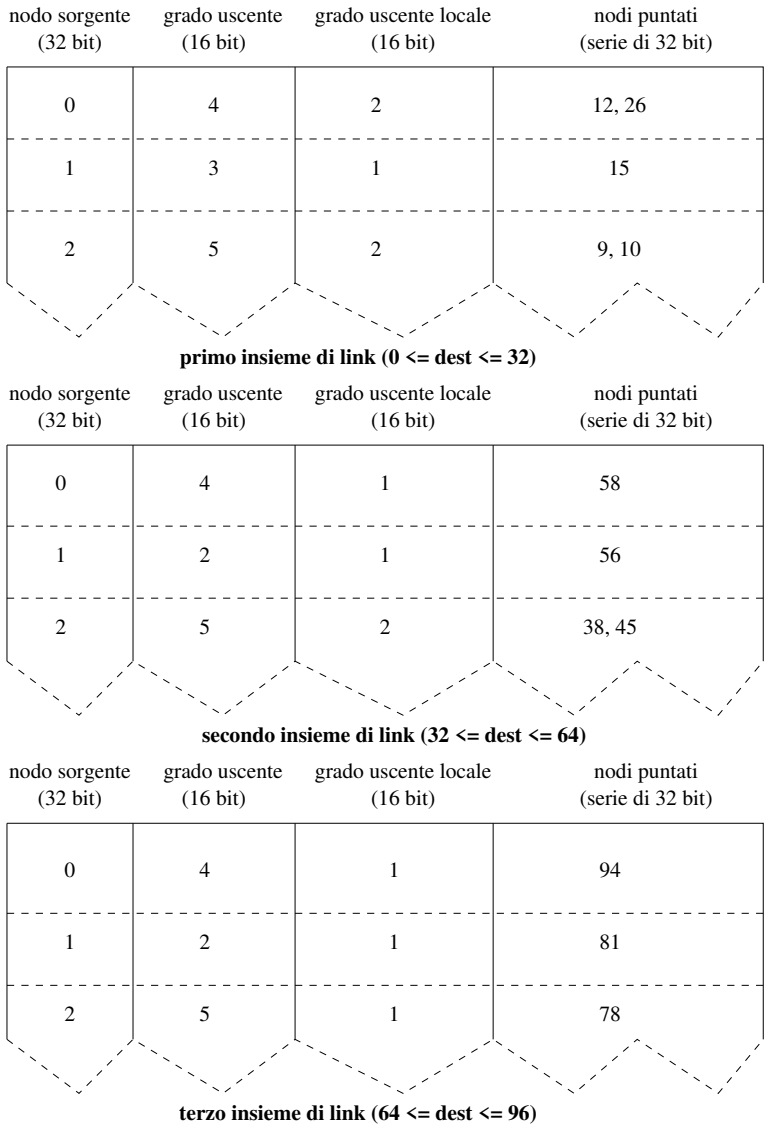

Figura 3.11: Il formato alternativo del grafo.

for $s = 1, 2, \cdots, n$ $Source[s] = \frac{1}{N}$
while $(residual > \tau)$
  for $i = 0, 1, \cdots, \beta - 1$
    for $d = 1, 2, \cdots, n$ $Dest_i[d] = 0$
    while not $Links_i.eof()$
      $Links_i.read(source, n, k, dest_1, \cdots, dest_k)$
      for $j = 1, 2, \cdots, k$ $Dest_i[dest_j] = Dest_i[dest_j] + \frac{Source[source]}{n}$
    for $d = 1, 2, \cdots, n$ $Dest_i[d] = c \cdot Dest_i[d] + \frac{1-c}{N}$
    scrivi $Dest_i$ su disco
  $residual = \|Source - Dest\|$
  $Source = Dest$
return $Source$ }

Figura 3.12: L'algoritmo efficiente per il calcolo di PageRank.

3.2.7 Esperimenti sulla convergenza

Nel paragrafo 3.2.3 si è vista una formula per l'algoritmo di PageRank che ne garantisce la convergenza su un insieme di pagine che a priori non soddisfa le ipotesi di irriducibilità e aperiodicità che la matrice stocastica associata al processo deve invece soddisfare. In questo paragrafo si esaminano invece alcuni risultati sperimentali trovati riguardo alla velocità di convergenza.

Da misure sperimentali eseguite da S. Brin e L. Page [4] sulla loro collezione di dati, PageRank su un database di 322 milioni di link converge con una norma del residuo tra due iterazioni successive piuttosto bassa in appena 52 iterazioni; la convergenza su metà di questi link avviene invece in circa 45 iterazioni. Appare quindi chiaro che PageRank scala molto bene con le dimensioni del database di link. Questi risultati suggeriscono l'utilità di PageRank come primo algoritmo da computare velocemente a priori per dare un qualche ordine statico alle pagine in modo da poter discriminare già in

prima battuta le pagine importanti dalle altre.

Altri studi sono venuti da Haveliwala [11] il quale si è focalizzato non soltanto sul valutare la norma del residuo tra due iterazioni successive, ma anche su altri criteri di convergenza per l'algoritmo di PageRank.

Le misure che Haveliwala propone sono due: convergenza sull'ordine globale delle pagine e convergenza sull'ordine introdotto da una *query*.

Convergenza sull'ordine globale

L'ordine globale sulle pagine introdotto da PageRank fornisce una chiara e dettagliata misura sul livello di convergenza dell'algoritmo di PageRank.

In particolare ciò che ci interessa è analizzare la posizione relativa tra pagine importanti, mentre non ci interessa in maniera significativa determinare se due pagine poco importanti compaiono nell'ordine corretto o meno. Haveliwala fa vedere come, su un insieme di circa un milione di pagine, già tra 25 e 50 iterazioni l'ordinamento relativo delle pagine sia già abbastanza ben definito, mentre mostra ampiamente quanto sia inutile passare da 50 a 100 iterazioni, in quanto il numero di pagine che cambiano di posizione è molto basso.

Convergenza su una query

Dal momento che generalmente PageRank viene utilizzato per rispondere in maniera appropriata ad interrogazioni fatte da esseri umani, una misura particolarmente significativa risulta essere quella data dall'ordinamento delle pagine fra quelle che soddisfano una certa interrogazione.

Haveliwala ha trovato che non solo un maggior numero di iterazioni porta una differenza soltanto nell'ordine in cui compaiono le pagine poco importanti ma anche che la differenza compare soltanto tra pagine non correlate, ossia tra pagine che difficilmente appariranno insieme nei risultati in risposta ad una qualche interrogazione. Pertanto il fatto che due pagine in realtà dovrebbero avere punteggi invertiti non inficia le ricerche che vengono effettuate all'interno di un motore, ricerche che peraltro, come verrà successivamente discusso (si veda il capitolo 5), sono pesate mediante moltissimi altri fattori.

3.2.8 Distribuzione di PageRank

Un recente studio [12] si è posto il problema di individuare se, dato un qualunque sottografo sufficientemente corposo del web, si possa dedurre qualche informazione riguardante la distribuzione dei valori di PageRank che vengono assegnati alle varie pagine.

La questione non è affatto di puro interesse matematico: infatti tramite una modellazione realistica della distribuzione di PageRank si potrebbe in linea teorica distribuire meglio gli indici inversi ottimizzando lo spazio da essi richiesto. Inoltre, PageRank è in qualche maniera correlato al numero di link uscenti delle pagine? In tal caso tutto quello che c'è dietro al funzionamento apparentemente sorprendente di PageRank potrebbe essere spiegato in termini molto più semplici di conteggio dei riferimenti (il cosiddetto *reference counting*).

Quel che risulta da questi studi è anzitutto il fatto che i punteggi assegnati dall'algoritmo di PageRank risultano distribuiti secondo una legge di tipo *power-law* con un esponente pari a 2.1, ossia: la frazione di pagine del grafo che hanno PageRank minore o uguale a r risulta essere proporzionale a r^{α}

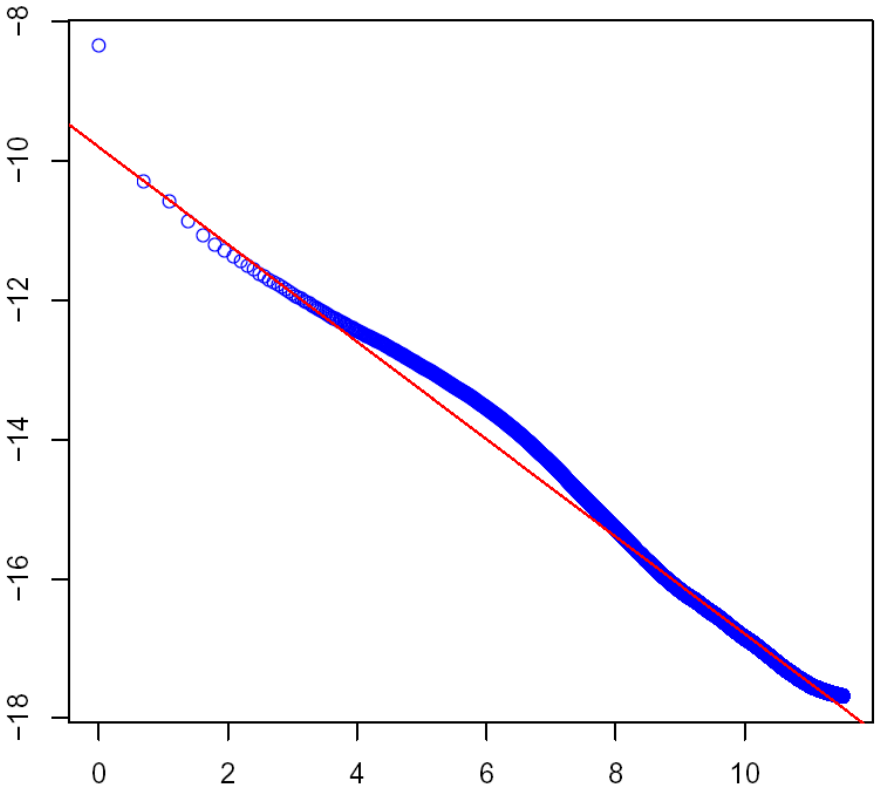

Figura 3.13: La distribuzione di PageRank su `.it`.

per $\alpha \approx -2$. Questo esponente risulta in particolare essere pari a quello che si osserva per la distribuzione dei link entranti.

Tuttavia tale fatto è una pura coincidenza, in quanto le pagine che hanno un PageRank elevato (o basso) non risultano solitamente essere le stesse che hanno anche un numero di link entranti elevato; ossia, pur coincidendo le due distribuzioni, non coincidono affatto[5] gli ordini relativi indotti sulle pagine del grafo del web.

Sempre lo studio [12] si è posto l'obbiettivo di capire a quale modello di grafo corrisponda una distribuzione simile. Alcuni studiosi hanno proposto un modello di grafo nel quale i nodi e gli archi sono aggiunti uno alla volta; tuttavia questo modello non porterebbe ad una distribuzione che segue la legge evidenziata. Il modo più semplice per risolvere tale problema è allora quello di introdurre nel modello la richiesta addizionale che ogni lato "sceglie" il nodo a cui puntare a caso, ma con probabilità non uniforme per la scelta dei vari nodi. In particolare, un lato punta ad un nodo q in proporzione al valore corrente del grado entrante di q. Questo dà un modello di grafo in cui il numero di pagine con k link entranti è proporzionale a $k^{-\alpha}$, con $\alpha \approx 2$.

Per spiegare l'esponente leggermente differente rispetto a quello calcolato empiricamente, alcuni studi [13] hanno tentato di dare una regola più complicata per scegliere a quale nodo un lato debba puntare. Per una frazione del tempo (un parametro che è stato indicato con $\beta \in [0, 1]$), il lato punta ad un nodo scelto uniformemente a caso; per il resto del tempo, ossia per una frazione $1 - \beta$, il lato "sceglie" un nodo intermedio v a caso e "decide" di puntare alla destinazione di uno dei link uscenti da v scelto a caso.

La spiegazione empirica è molto convincente: alle volte un creatore di una

[5] Anzi, sono molto diversi.

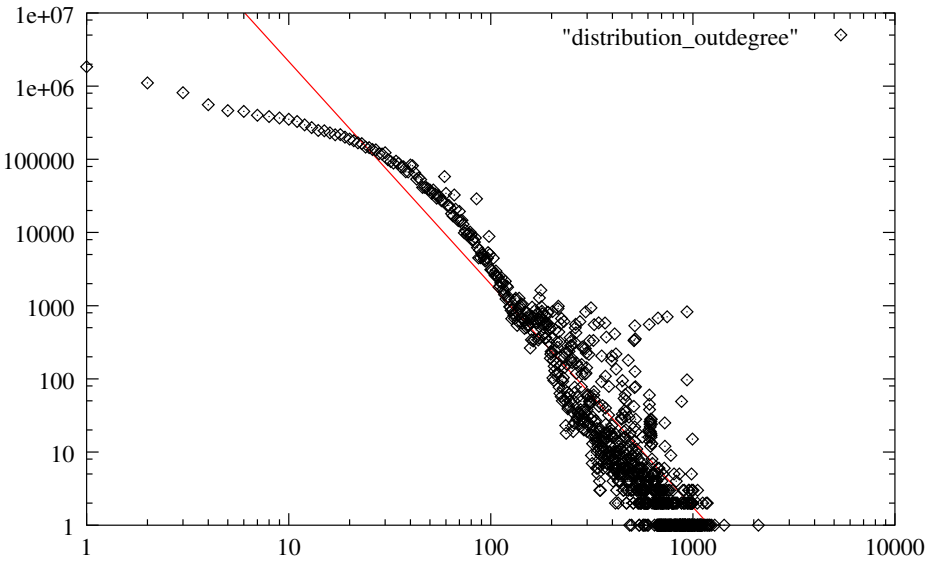

Figura 3.14: La distribuzione dei gradi uscenti su `.it`.

pagina web si riferisce ad un argomento a caso e pertanto fa puntare un collegamento ad una destinazione casuale. Per il resto del tempo invece il creatore della pagina copia uno dei collegamenti di un'altra pagina v, decidendo che il link è interessante.

Questa semplice spiegazione riesce a convincere anche da un punto di vista matematico: infatti in grafi modellati sulla base di questo modello si osservano le previste distribuzioni *power-law* ed inoltre si osservano anche alcune proprietà sulle cricche evidenziate in altri studi [14].

Come si può vedere in figura 3.13 e in figura 3.14, dai nostri dati sperimentali non è stato possibile verificare le congetture degli articoli citati, anche se si evidenziano comunque degli andamenti molto simili a quelli previsti teoricamente.

3.3 Topic-sensitive PageRank

Come evidenziato, PageRank fornisce un ordinamento statico alle pagine: è ovvio che un tale ordinamento non sempre va bene. Si possono tentare molte vie per risolvere questo problema, e alcune di queste sono state studiate e verranno presentate nei capitoli successivi. Una di queste tecniche è stata presentata da Haveliwala [9] ed è essenzialmente una variante del sopracitato algoritmo di PageRank.

Questo algoritmo, chiamato Topic-sensitive PageRank, è un algoritmo esogeno e si compone di una parte da calcolarsi — anche piuttosto velocemente — a priori, e di una parte che viene eseguita al momento dell'interrogazione e che tiene conto dell'interrogazione stessa e di un vettore di preferenze. Tuttavia, al contrario di Hits, un altro algoritmo che verrà presentato in seguito, Topic-sensitive PageRank tenta di fornire una risposta precisa alle *query* dell'utente, più che tentare di compilare una lista di risorse più o meno pertinenti all'argomento della *query*. Cioè, ricercando qualcosa di molto specifico si riuscirà con un po' di fortuna ad accedere alla pagina esatta che tratta di quell'argomento mentre con Hits si troveranno probabilmente risorse generiche sull'argomento ricercato.

Un pregio dell'algoritmo di Topic-sensitive PageRank è quello di essere poco suscettibile al fenomeno di *spamming*, ossia al tentativo di far assegnare un punteggio più alto del dovuto ad un certo insieme di pagine. Infatti, rendendo PageRank sensibile al contesto, si eliminano i problemi derivanti da pagine che puntano ad altre ad esse non attinenti.

Tuttavia, anche l'algoritmo di Topic-sensitive PageRank soffre di alcuni problemi che verranno in seguito evidenziati (si veda il paragrafo 3.3.4).

3.3.1 Struttura generale dell'algoritmo

Topic-sensitive PageRank richiede il calcolo a priori di alcuni vettori di PageRank: in tale maniera ad ogni pagina vengono assegnati punteggi multipli, che verranno poi utilizzati in maniera differente a seconda dell'argomento a cui si suppone che l'utente si stia riferendo nel momento in cui inserisce la *query*. Al momento della *query*, infatti, tali punteggi verranno combinati tra di loro per formare così un punteggio composito per le pagine che soddisfano l'interrogazione. Questo punteggio composito può poi essere inserito in complicate funzioni di punteggio per essere ulteriormente raffinato (anche se, nel suo articolo, Haveliwala fornisce una semplice funzione di punteggio).

Parte precomputata

L'idea di Haveliwala è stata quella di sfruttare al meglio una categorizzazione manuale compiuta da migliaia di volontari sparsi per il mondo che hanno dato vita ad una delle più grandi raccolte di link catalogati per argomento: l'Open Directory Project, abbreviato ODP [15].

Anzitutto si generano i vettori di PageRank basandosi su un insieme di argomenti basilari. Tali argomenti sono, secondo Haveliwala, le 16 categorie principali della ODP, ma il concetto potrebbe essere esteso ad un numero maggiore di categorie ed, eventualmente, di sottocategorie. In particolare, il vettore E di personalizzazione di PageRank viene qui calcolato basandosi sulle URL presenti nelle categorie della ODP. Sia T_j l'insieme delle URL nella categoria c_j della ODP. Quando verrà calcolato il vettore di PageRank per tale categoria, al posto di utilizzare il vettore uniforme $E = \left[\frac{1}{N}\right]_{N\times 1}$ (dove si è indicato con N il numero di pagine nell'archivio) si utilizza il vettore non uniforme $E = v_j$ dove:

$$v_{ji} = \begin{cases} \frac{1}{|T_j|} & \text{se } i \in T_j \\ 0 & \text{altrimenti.} \end{cases} \tag{3.11}$$

Il vettore di PageRank per la categoria c_j verrà d'ora innanzi indicato con $PR(\alpha, v_j)$ dove α è il fattore di spargimento già presentato nel capitolo dedicato a PageRank.

Oltre a tale vettore vengono anche calcolati i 16 vettori di termini D_j che consistono dei termini presenti nei documenti elencati sotto ciascuna delle 16 categorie della ODP. D_{jt} fornisce pertanto il numero totale di occorrenze del termine t nei documenti elencati al di sotto della categoria c_j della ODP.

Come fa giustamente notare Haveliwala, i modi e le fonti per calcolare tali vettori sono molteplici: tuttavia i dati della ODP sono disponibili liberamente e sono molto difficilmente suscettibili allo *spamming*, essendo la ODP un progetto non commerciale e quindi, in teoria, privo di manipolazioni di parte.

Parte al momento dell'interrogazione

Una volta compiute le operazioni descritte sopra, il resto dell'algoritmo funziona al momento dell'interrogazione, ossia al momento in cui un utente inserisce la *query* nel motore di ricerca. Data una interrogazione q, sia q' il contesto di q. In altri termini, se la *query* è stata fatta selezionando una parola di una pagina web u, sia q' fatta dalle parole di u. Per le altre *query* (che sono certamente ben più diffuse) sia $q' = q$. Si calcolano le probabilità per ciascuna delle 16 categorie della ODP, condizionate su q'. Sia q'_i il termine i-esimo in q'. Data l'interrogazione q si può pertanto calcolare per ogni c_j la seguente probabilità:

$$P(c_j|q') = \frac{P(c_j) \cdot P(q'|c_j)}{P(q')} = \frac{P(c_j) \cdot \prod_i P(q'_i|c_j)}{P(q')} \tag{3.12}$$

Sebbene nell'articolo non siano completamente chiarite le assunzioni sul modello stocastico del testo delle pagine che giustifichino tali passaggi, $P(q'_i|c_j)$ può essere facilmente calcolato dal vettore dei termini D_j in quanto tale probabilità altro non è che il numero di occorrenze totali presenti in D_j diviso per il numero di occorrenze del termine q'_i in D_j. La quantità $P(c_j)$ invece non è così immediata da calcolare. Nell'articolo citato, Haveliwala la considera uniforme ma suggerisce l'idea che tale probabilità potrebbe essere personalizzata a seconda dei gusti dell'utente, al fine da rendere maggiormente "pesanti" alcune categorie rispetto ad altre (in maniera tale che ad un ipotetico agricoltore a cui interessino le "API" non compaiano le "API" di un qualunque sistema operativo). Inoltre la personalizzazione di $P(c_j)$ risulta essere un buon espediente per variare la distribuzione di probabilità dei risultati senza però variare il vettore di personalizzazione E.

Il passo successivo alla stima della probabilità sopra citata risulta essere quello di utilizzare un indice testuale (ad esempio un indice inverso) per estrarre tutti i documenti che contengono i termini dell'interrogazione originale q. Infine, si calcola un punteggio composito di ciascuno di questi documenti come segue. Sia $rank_{jd}$ il punteggio del documento d dato dal vettore $PR(\alpha, v_j)$ (cioè il vettore di PageRank per la categoria c_j). Per il documento d viene calcolato un punteggio s_{qd} come segue:

$$s_{qd} = \sum_j P(c_j|q') \cdot rank_{jd} \tag{3.13}$$

I risultati vengono poi ordinati in base a questo punteggio finale. Ovvia-

mente, in sede di implementazione, sarebbe possibile complicare a piacere la formula appena citata per introdurre eventuali altri fattori di peso, come ad esempio la Proximity di un documento oppure il punteggio di altri algoritmi.

È da notare che il calcolo appena illustrato ha un'interpretazione probabilistica in termini del modello del navigatore casuale cui si è già fatto cenno in PageRank. Con probabilità $1 - \alpha$ infatti il navigatore che si trova sulla pagina u segue uno tra i link presenti in u (dove la scelta tra uno di questi è uniformemente casuale). Con probabilità $\alpha \cdot P(c_j|q')$ invece il navigatore salta ad una delle pagine in T_j (dove la scelta tra una di queste pagine è uniformemente casuale). La probabilità — vista sulla visita a lungo termine — che il navigatore si trovi a navigare sulla pagina v è esattamente data dal punteggio composito s_{qd} definito sopra. Pertanto gli argomenti della ODP influiscono sul punteggio finale in proporzione alla loro affinità con la *query* (od il suo contesto).

3.3.2 Effetti del peso con la ODP

Haveliwala ha calcolato quanto sia grande l'effetto di pesare i vettori di PageRank con le pagine della ODP. Anzitutto una prima considerazione che è stata fatta è stata quella di vedere cosa accade per diversi valori del parametro α. Si consideri anzitutto il caso estremo in cui $\alpha = 1$: in tale ipotesi le URL nell'insieme T_j avranno punteggio $1/|T_j|$ oppure 0. Viceversa, se α è prossimo a 0, il contenuto di T_j tende a diventare irrilevante per il punteggio finale.

La scelta euristica operata da Haveliwala è stata di scegliere α pari a 0.25. Tuttavia, in fase di sperimentazione, Haveliwala afferma che è emerso che una scelta di α tra 0.05 e 0.25 non varia significativamente i punteggi

indotti.

3.3.3 Punteggio context-sensitive

Come si è discusso prima, nell'articolo di Haveliwala che presenta Topic-sensitive PageRank si fa in più occasioni riferimento all'idea di estrapolare in una qualche maniera il contesto dall'interrogazione che l'utente inserisce. Ad esempio, viene proposta l'idea di prendere come contesto l'insieme delle parole vicine ad una parola che viene selezionata su una pagina per essere poi ricercata tramite il motore di ricerca. L'idea è molto interessante, ma occorrerebbero certamente *plug-in* software e si entrerebbe in un mondo che con i motori di ricerca e gli algoritmi di *ranking* ha poco a che fare. Ciononostante, l'idea è molto interessante da un punto di vista socio-linguistico: alcune parole, in certe lingue, hanno doppi significati (*polisemia*) e vengono spesso utilizzate in ambiti totalmente diversi per denotare concetti completamente diversi. Un esempio è sicuramente il termine "Apple" in lingua inglese, che indica la mela (il frutto) ma anche una nota industria nel campo dell'informatica, e il sopracitato termine "api", che può assumere nella nostra lingua almeno 3 significati: il nome della catena di distributori di benzina, il plurale del nome dell'insetto "ape", ed infine l'acronimo per "Application Programming Interface", oramai entrato in uso comune tra gli addetti ai lavori. Considerare le parole che stanno accanto alla parola "Api" può portare in tal senso a disambiguare il senso della frase.

Al momento dell'interrogazione, in tal caso, si calcolerebbe $P(c_j|q')$ come descritto precedentemente, usando per q' anche i termini adiacenti al termine ricercato. Nel nostro esempio, nel primo caso (distributore di benzina) $argmax_{c_j} P(c_j|q')$ potrebbe essere la categoria della ODP (italiana) AFFARI,

mentre nel caso ad esempio del terzo significato ("Application Programming Interface") la categoria giusta sarebbe COMPUTER.

È proprio in casi simili che la scelta del vettore di PageRank da applicare contribuisce a disambiguare in maniera definitiva il termine. Tuttavia, a nostro modo di vedere, questo approccio è di difficile applicabilità nell'ambito dello sviluppo di un motore di ricerca, e richiede certamente un notevole *feedback* da parte degli utenti per risultare veramente utile. Si tenga inoltre conto che le ricerche normalmente fatte dall'utente medio tipico sono di una o due parole direttamente inserite nel motore di ricerca [16], e non evidenziate in una pagina come suggerisce Haveliwala: pertanto la determinazione del contesto potrebbe essere impossibile in casi simili. In altri casi nei quali l'utente si trova di fronte a risultati fuorvianti, egli è già di per sé tipicamente portato a raffinare la ricerca con termini più specifici, e in questo senso un notevole aiuto viene portato da algoritmi quali Proximity (vedi paragrafo 4.2). Infatti con tali algoritmi si riescono già a scremare grosse quantità di risultati in maniera da rispondere al meglio ad interrogazioni più sofisticate (cioè a quelle interrogazioni che portano con sé il contesto in cui vengono effettuate).

3.3.4 Difetti di Topic-sensitive PageRank

Come evidenziato da Haveliwala stesso e come si può anche intuire dal suo lavoro, vi sono alcuni problemi che riguardano l'algoritmo di Topic-sensitive PageRank.

Anzitutto viene fatta una suddivisione molto blanda degli argomenti per i quali calcolare i vettori di PageRank. Questo è dovuto essenzialmente al fatto che, oltre allo spazio su disco che richiede ciascun vettore di PageRank

(spazio comunque relativamente trascurabile), il tempo richiesto per calcolare un numero elevato di vettori di PageRank può essere alto. Un altro problema che si può evidenziare è che inserire una ricerca nel suo contesto può essere una operazione non semplice. Haveliwala propone in maniera non meglio precisata la determinazione del contesto in base alle pagine da cui l'utente esegue la sua ricerca: come farlo, quali eventuali *plug-in* software adottare e a quali standard aderire rimangono tuttavia lasciati al lettore. C'è da notare infine che nessun motore di rilievo (commerciale o meno) dichiara di utilizzare tale algoritmo.

3.4 Hits

Mentre gli algoritmi illustrati finora sono concepiti per rispondere in maniera precisa alle richieste dell'utente, con l'algoritmo di Hits [17] l'attenzione si sposta di più verso l'individuare di pagine autorevoli riguardo all'argomento trattato. In particolare l'algoritmo di Hits è un algoritmo applicabile soltanto a seguito di una prima classificazione delle pagine: infatti esso richiede, proprio come *input*, un insieme di pagine determinate in una qualche maniera — ad esempio selezionate tramite gli algoritmi visti — e che soddisfino l'interrogazione dell'utente. Anche Hits è, al pari degli algoritmi visti finora, un algoritmo esogeno.

Uno degli aspetti più sorprendenti di questo algoritmo, peraltro molto poco utilizzabile su vasta scala (come sarebbe, ad esempio, in un vero e proprio motore di ricerca), è senza dubbio il fatto che, come nel caso di PageRank, per determinare le pagine autorevoli si faccia ricorso soltanto ad una analisi della struttura del grafo del web, e non invece a tecniche di matching testuale.

L'algoritmo di Hits è in particolare noto per permettere l'individuazione di *community* all'interno del web in quanto permette di estrarre moltissima informazione soltanto dall'analisi dei link tra le pagine, e quindi da una semplice analisi del grafo. Sebbene la tecnica proposta non sia solamente applicabile al web, certamente è proprio in questo ambito che essa risulta più utile: infatti il web è un insieme di pagine tra loro più o meno fortemente collegate che continua a crescere in dimensioni ad un ritmo esponenziale (si vedano [18] e [19]).

Un modo per cercare pagine autorevoli risulta essere quello di consultare i repertori esistenti (es. Yahoo [7], DMOZ [15]) compilati in maniera manuale

o semiautomatica. L'algoritmo Hits si propone invece di fare tutto questo in maniera totalmente automatica analizzando la struttura del grafo del web.

3.4.1 Query e pagine autorevoli

Anzitutto si deve tenere presente che le *query* comunemente ricevute dai motori di ricerca possono essere di varia natura: non solo le si può distinguere banalmente tra di loro per l'argomento trattato ma anche per il tipo di risultati che l'utente si aspetta di ricevere. Kleinberg fa la seguente classificazione:

- **interrogazioni specifiche**: ad esempio, "Mozilla supporta xforms?"
- **interrogazioni vaste**: ad esempio, "Linux tutorial"
- **interrogazioni volte a trovare pagine simili**: ad esempio, "Trova pagine simili a `http://java.sun.com/`"

Concentrandoci per il momento sui primi due tipi di *query*, va notato che mentre nel primo caso è molto difficile trovare la pagina "giusta" dal momento che ve ne sono molto poche che soddisfano l'interrogazione ed è difficile recuperarle, nel secondo caso ve ne sono invece molte, talmente tante che individuare esattamente quella — o quelle — volute dall'utente risulta un'impresa molto difficile.

La nozione di autorità, relativamente ad una interrogazione vasta, gioca un ruolo fondamentale. Anzitutto, come si può stabilire se una pagina è autorevole? Si può pensare, ad esempio, che la pagina corrispondente all'indirizzo `http://www.harvard.edu/` sia la pagina che per definizione un utente desidera vedere elencata al primo posto nei risultati della ricerca corrispondente alla *query* `Harvard`. Tuttavia esistono innumerevoli pagine web che

contengono il termine `Harvard`: come individuare tra tutte queste proprio la pagina giusta, cioè la *home-page* della Harvard University? Inoltre spesso e volentieri una pagina non avrà scritte dentro di essa (tutte) le parole con la quale viene comunemente ricercata. Ne sono un classico esempio i motori di ricerca, le cui *home-page* certo non contengono al loro interno le parole "motore di ricerca".

3.4.2 Analisi della struttura dei link

Analizzare la struttura a link del grafo del web fornisce la risposta a molte delle domande appena poste. I link contengono al loro interno una parte considerevole di giudizio degli esseri umani, ed è proprio questo giudizio ad essere quello che ci fa trovare le pagine autorevoli. In particolare, il creatore di una pagina web che inserisce un link verso un'altra pagina web non fa altro che dare un proprio giudizio alla pagina puntata: "ti punto perché ti stimo". È questa senza dubbio la forma più democratica di votazione e questo concetto è stato già largamente illustrato nel capitolo dedicato all'algoritmo di PageRank. Inoltre link di questo tipo ci danno la possibilità di trovare potenziali pagine autorevoli che altrimenti ci sarebbero sfuggite per il fatto che al loro interno non contengono (tutte) le parole inserite nell'interrogazione dell'utente.

Questa idea, che sembra ottima a prima vista, soffre tuttavia di alcuni problemi che vanno subito chiariti e risolti: anzitutto moltissimi link esistono per ragioni che sono esattamente l'opposto del "voto democratico" cui si accennava. Ne sono un esempio i link *intra-host* (talvolta chiamati anche *link navigazionali*) che servono, ad esempio, per ritornare all'*home-page* oppure i link presenti grazie alle sponsorizzazioni commerciali.

Inoltre, *rilevanza* e *popolarità* sono due concetti ben distinti, ma entrambi servono per trovare le pagine autorevoli. Infatti sarebbe molto sbagliato considerare valida questa euristica per trovare pagine autorevoli: di tutte le pagine che soddisfano la *query*, restituisci quelle più puntate. In questo caso infatti verrebbero sempre restituiti link come `http://www.google.com/` o `http://www.microsoft.com/`, mentre questo non è certo quel che vogliamo.

Il modello presentato da Kleinberg tiene conto di tutti questi fattori ed evidenzia i modi per ottenere pagine veramente autorevoli, senza troppi falsi positivi. Per fare ciò l'algoritmo proposto si basa sulla individuazione dei cosiddetti *hub*, ossia delle pagine che più largamente puntano a risorse autorevoli. Si viene così a creare un circolo virtuoso tra *hub* ed *autorità*: una autorità è una pagina puntata da molti hub, un hub è una pagina che punta a molte autorità.

3.4.3 Costruzione di un sottografo del web

Supponiamo di ricevere una *query* q vasta (nel senso visto prima): vogliamo determinare le pagina autorevoli basandoci su una analisi della struttura dei link, ma prima dobbiamo determinare un sottografo opportuno sul quale operare. Per determinare tale sottografo si potrebbero, ad esempio, considerare tutte le pagine che soddisfano la *query*. Tuttavia un tale insieme di pagine può essere di svariati milioni di pagine web, e pertanto potrebbe facilmente diventare ingestibile per quanto riguarda i costi di calcolo. Inoltre un tale approccio taglierebbe fuori le pagine magari più autorevoli ma che non soddisfano l'interrogazione (si pensi all'esempio di prima sui motori di ricerca).

Idealmente, vorremmo disporre di una collezione S_q di pagine con queste proprietà:

- S_q è relativamente piccolo;
- S_q è ricco di pagine rilevanti;
- S_q contiene tutte — o quasi — le pagine più autorevoli.

Si noti che tali richieste sono rivolte essenzialmente a minimizzare il costo computazionale di applicare un algoritmo decisamente non banale ma anzi piuttosto costoso e che va purtroppo eseguito al momento dell'interrogazione.

Come è possibile pertanto determinare tale insieme di pagine? Stabilito un parametro t (che Kleinberg pone pari a 200), si estraggono le t pagine con punteggio più elevato secondo un qualche algoritmo di ricerca testuale (come ad esempio Proximity) o secondo un qualche motore di ricerca (Kleinberg ha usato i risultati di Altavista); tali pagine vengono chiamate *root-set* R_q. Questo insieme soddisfa in generale i primi due punti della lista (è piccolo e ricco di pagine rilevanti) ma è ben lontano dal contenere tutte o quasi le pagine più autorevoli. È altresì da notare che spesso vi sono pochissimi link tra le pagine di questo insieme. L'idea di Kleinberg è di utilizzare tale insieme R_q per costruire un insieme di pagine S_q che soddisfi anche l'ultimo punto desiderato. Si consideri infatti un'autorità per la *query* q: se non è presente all'interno dei t risultati trovati, è molto probabile che sia puntata da almeno uno dei t documenti. Pertanto possiamo inserire delle autorità all'interno di S_q semplicemente seguendo i link che escono da R_q e comprendendo le pagine che puntano all'interno del root-set. Nella pratica seguiremo tutti i link uscenti dal root-set R_q per ogni pagina del root-set, e al più d link entranti nel root-set: questo requisito diventa essenziale per evitare che vengano inserite

dentro S_q troppe pagine, con la conseguenza che i tempi di calcolo aumentino notevolmente. S_q viene detto *base-set* per q.

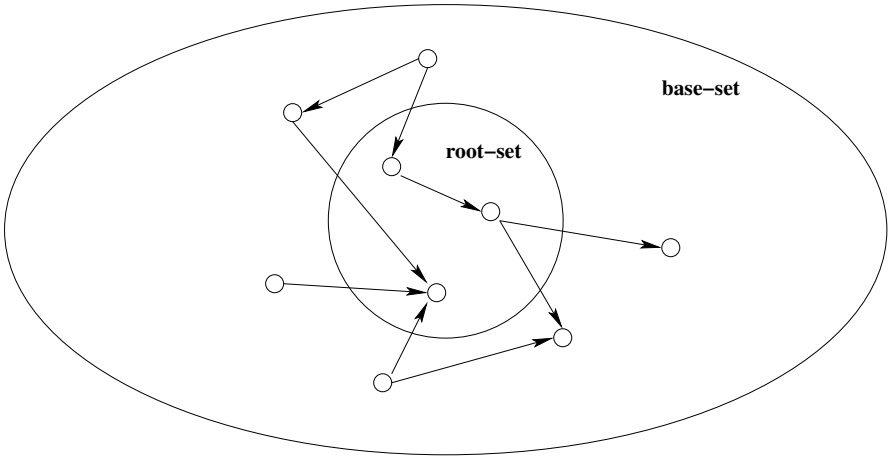

Figura 3.15: Espansione di un *root-set* in un *base-set*.

Prima di procedere ad analizzare l'algoritmo di Hits in sé, ha senso spiegare una piccola euristica che si può applicare al fine di eliminare i link utili solamente per la navigazione e che inficierebbero l'algoritmo appena presentato. Kleinberg propone di eliminare dal grafo tutti i link *intra-host*, ossia tutti i link interni ad uno stesso dominio. Questo è utile perché i link *intra-host* sono solitamente link puramente navigazionali e che non portano all'individuazione di autorità ma solo di pagine inutili per gli scopi l'algoritmo. Si possono naturalmente pensare altre euristiche per eliminare link che rischiano di portare *spam* all'interno dei risultati, come ad esempio non permettere ad una pagina di portare più di un numero fissato m di pagine dello stesso dominio. Secondo i test di Kleinberg tuttavia anche senza questa ulteriore restrizione l'algoritmo funziona molto bene.

In pseudo-codice, un algoritmo per determinare il base-set risulta essere quello riportato in figura 3.16:

$S_q = R_q$
for p **in** R_q
    aggiungi $G(p, -)$ a S_q
    if $|G(-, p)| \leq d$ $S_q = S_q \cup G(-, p)$
    else $S_q = S_q \cup \{d$ pagine di $G(-, p)\}$
return S_q

Figura 3.16: L'algoritmo per determinare il *base-set*.

3.4.4 Determinazione di hub ed autorità

Nella sezione precedente si è illustrato un metodo per calcolare un sottografo del grafo del web "a tema", ossia incentrato sull'argomento che si sta ricercando. Il problema è ora di estrarre le pagine rilevanti e le autorità da questo insieme basandosi solamente sull'analisi ipertestuale del sottografo preventivamente ottenuto.

Un'idea possibile è quella di ordinare questo insieme di pagine in base al grado entrante delle stesse. Purtroppo tale approccio si rileva fallimentare perché spesso e volentieri le pagine del sottografo contengono un insieme di link a pagine pubblicitarie che verrebbero pertanto indicate come autorità quando non è detto che lo siano. Pertanto, pagine con un alto grado entrante non sono necessariamente le pagine più autorevoli.

Invece, bisogna notare che un'autorità sarà puntata da molte pagine del base-set: pertanto vi saranno alcune pagine del base-set che avranno un insieme non disgiunto di link uscenti. Le pagine del base-set che puntano a pagine autorevoli saranno dette hub, mentre le pagine autorevoli sono tali perché puntate da molti hub. Questa relazione che si rafforza mutuamente è la chiave di tutto l'algoritmo di Hits.

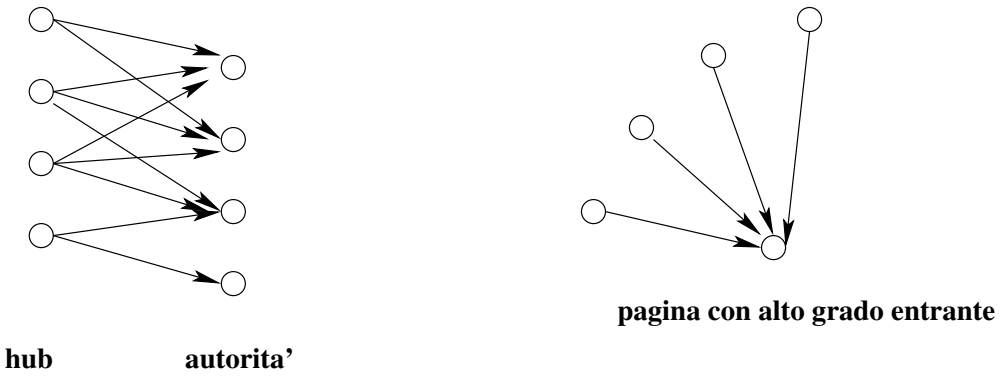

Figura 3.17: Un insieme molto collegato di hub ed autorità.

3.4.5 L'algoritmo Hits

La procedura per determinare gli hub e le autorità può essere implementata in modo iterativo ed essere eseguita per un prestabilito numero di iterazioni, oppure fintantoché la norma del residuo tra due iterazioni successive non scende al di sotto di una certa soglia, similmente a quanto discusso in precedenza per l'algoritmo di PageRank.

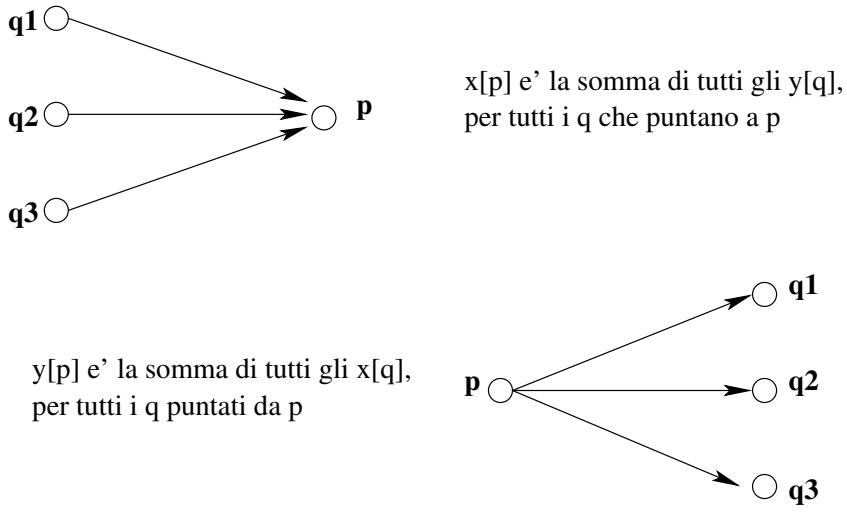

Figura 3.18: Le operazioni basilari di Hits.

$z = (1, 1, 1, \cdots, 1) \in \mathbf{R}^n$
$x_0 = z$
$y_0 = z$
for $i = 1, 2, \cdots, k$
    $x'(p) := \sum_{q \to p} y(q)$
    $y'(p) := \sum_{p \to q} x(q)$
    $x_i =$ normalizza x'_i
    $y_i =$ normalizza y'_i
return (x_k, y_k)

Figura 3.19: L'algoritmo Hits.

Usando le notazioni viste nella sezione precedente, un semplice algoritmo in pseudo-codice per Hits su un grafo G è quello riportato in figura 3.19, dove G è un grafo composto da n pagine e k un numero naturale.

3.4.6 Query volte a trovare pagine simili

L'algoritmo descritto precedentemente può anche essere applicato ad un altro tipo di problema, ossia al problema di trovare pagine simili ad una pagina data. Supponiamo di aver trovato una pagina p di interesse — ad esempio, una pagina autorevole su un argomento di nostro interesse — e ci chiediamo quali altri altre pagine gli utenti del web considerano essere simili a p quando creano i link.

Se p è una pagina molto referenziata, abbiamo un problema di abbondanza: la struttura di link vicina a p rappresenterà una enorme quantità di opinioni indipendenti circa la relazione di p con le altre pagine. Usando la nozione vista di hub ed autorità, possiamo chiederci: nelle regioni della strut-

tura di link vicino a p, quali sono le autorità? Tali autorità potrebbero infatti darci una idea molto chiara dell'argomento trattato dalle pagine simili a p.

Pertanto, anziché inziare la ricerca con una interrogazione q, inizieremo la ricerca con una pagina p e cercheremo un insieme di t pagine che puntano a p. A questo punto creeremo il nostro root-set R_p consistente nelle prime t pagine trovate e, successivamente, inseriremo altri elementi in R_p affinché diventi un insieme S_p come fatto in precedenza. Nel sottografo risultante cercheremo poi hub ed autorità.

3.4.7 Hits: sperimentazione

L'algoritmo di Hits è stato implementato e fa parte del progetto associato a questa tesi. Essendo tuttavia un algoritmo da calcolare completamente a tempo di interrogazione, i tempi di esecuzione sono parecchio elevati. Vi sono stati vari problemi durante l'implementazione di Hits: uno di questi è certamente il fatto che il grafo deve essere caricato interamente in memoria, e che pertanto occorrono delle tecniche di compressione efficienti ma al contempo rapide nell'accesso; mantenere le liste di adiacenza non compresse è stato invece causa di notevoli problemi di *garbage collection* (si veda il paragrafo 6.2).

Un altro problema riscontrato sottoponendo manualmente delle interrogazioni ad Hits è stato il fatto che Hits appare molto suscettibile al fenomeno dello *spamming*. Infatti, ricercando parole molto diffuse i risultati appaiono visibilmente falsati dalla presenza di pagine contenenti letteralmente dozzine di parole chiave e collegamenti verso altre pagine. L'algoritmo di Hits, a fronte dei problemi or ora evidenziati, non è pertanto sembrato particolarmente utile per l'utilizzo nel motore di ricerca, ed è pertanto stato inserito soltanto

come algoritmo ausiliario che può essere richiamato a scelta dell'utente in seconda battuta, dopo aver ricevuto i risultati calcolati dagli altri algoritmi.

Capitolo 4

Algoritmi basati sul contenuto

In questa sezione verranno presentati alcuni algoritmi che, a differenza di quelli illustrati nei precedenti capitoli, si basano sul contenuto delle pagine web per assegnare i punteggi e forniscono pertanto una misura endogena di rilevanza.

Bisogna ricordare che gli algoritmi che vengono qui presentati sono utilissimi per fornire un input ridotto agli altri algoritmi di ranking illustrati precedentemente. Infatti, come si è visto, PageRank di per sé fornirebbe sempre la stessa lista di risultati, sempre nello stesso ordine, essendo l'ordinamento di PageRank un ordinamento statico dei risultati. Combinando tuttavia PageRank con almeno uno degli algoritmi qui presentati si può ottenere un insieme di risultati che variano in funzione della interrogazione dell'utente, e che pertanto sono certamente più interessanti sotto ogni punto di vista.

I punteggi da assegnare a questi algoritmi, come combinarli tra loro e altre considerazioni sono invece materia di un capitolo successivo riguardante uno studio portato avanti durante questo lavoro di tesi (vedi capitolo 5).

Per quanto riguarda le parti da precomputare, gli algoritmi che verranno

illustrati richiedono generalmente la generazione a priori degli indici inversi, necessari per determinare quali documenti soddisfano l'interrogazione. Fa eccezione il solo algoritmo di Latent Semantic Indexing, che richiede invece il calcolo di una decomposizione a valori singolari di una matrice.

4.1 Latent Semantic Indexing

4.1.1 Introduzione

Il primo algoritmo endogeno che viene presentato è il cosiddetto algoritmo di Latent-Semantic Indexing (comunemente riferito come LSI). Tale algoritmo è un algoritmo dell'IR classico ed è stato storicamente adottato per un lungo periodo da famosi motori di ricerca come Altavista e si ritiene che venga tutt'oggi adottato in alcuni motori di ricerca di news e di immagini.

L'algoritmo di LSI [20] è in realtà contrapposto agli altri algoritmi di *matching* testuali, in quanto non serve ad estrarre le pagine che contengono certi termini e ad eliminare le altre, ma serve per estrapolare relazioni ben più sottili del semplice "è presente"/"non è presente". Infatti tale algoritmo non solo registra quali documenti contengono quali parole, ma considera anche l'intera collezione di documenti nel suo complesso per determinare quali documenti contengono un certo insieme di parole. LSI cataloga i documenti che hanno molte parole in comune come *semanticamente correlati*, e quelli che invece ne hanno poche in comune come *semanticamente distanti*. Questo metodo, dall'idea molto semplice, nella pratica si avvicina moltissimo a quel che farebbe un essere umano chiamato a catalogare quello stesso insieme di documenti, come dimostrano vari studi [21]. Sebbene LSI non utilizzi alcuna informazione semantica, i *pattern* che riconosce lo rendono tanto apparentemente quanto sorprendentemente intelligente.

Quando si effettua una ricerca all'interno di una base di dati indicizzata tramite l'algoritmo di LSI, anzitutto vengono confrontati i valori di somiglianza che sono stati calcolati per ogni parola, e vengono restituiti i documenti che LSI ritiene che meglio soddisfino l'interrogazione. Dal momento che due

documenti possono essere semanticamente correlati anche se non condividono parole particolari, LSI non richiede un *match* esatto per restituire risultati utili, cosa che invece richiedono gli altri algoritmi di matching testuali che si illustreranno.

La complessità maggiore dell'algoritmo di Latent Semantic Indexing risulta risiedere nella decomposizione della matrice, essendo gli algoritmi noti cubici in tempo. Quando eseguito su una collezione di dati ampia come quella del web, i tempi aumentano notevolmente al punto che l'algoritmo di LSI non risulta praticamente applicabile.

Un altro difetto dell'algoritmo di LSI è quello di richiedere tempi molto elevati per essere eseguito. L'algoritmo di LSI infatti si compone essenzialmente di due parti, una precomputata, molto dispendiosa, ed una calcolata invece al momento dell'interrogazione. La parte precomputata risulta essere una decomposizione in valori singolari di una matrice enorme, e non è pertanto realistico pensare ad aggiornamenti frequenti dell'insieme delle parole, nonostante in letteratura siano note tecniche per parallelizzare il calcolo.

4.1.2 Funzionamento

Per funzionare LSI richiede anzitutto la compilazione di una lista di termini contenuti all'interno della collezione di documenti da indicizzare. Tale lista dovrà essere opportunamente epurata dalle cosiddette *stop-word*, ossia dalle parole molto comuni che compaiono in pressoché tutti i documenti del nostro insieme ed anche più volte all'interno dello stesso documento. Esempi di tali parole sono gli aggettivi comuni, i verbi comuni, gli articoli, le preposizioni. Inoltre, dalla lista vengono eliminati gli *hapax legomena*, cioè i termini estremamente rari che compaiono in un solo documento, frutto spesso di errori di

digitazione.

Una volta generata tale lista si genera la cosiddetta *matrice dei termini e dei documenti*. Si può pensare a questa matrice come una griglia nella quale i documenti vengono messi sull'asse orizzontale, mentre i termini — ossia le parole contenute nei documenti — vengono posti sull'asse verticale. Per ogni parola della nostra lista, si inserisce un numero che indica la frequenza relativa del termine in corrispondenza dei documenti nei quali tale parola compare, cioè il rapporto fra le sue occorrenze nel documento e quelle nell'intera collezione.

Una volta compilata, tale griglia riporta in maniera concisa pressoché tutto quel che sappiamo sulla nostra collezione di documenti: potremmo, volendolo fare, elencare le parole di cui è composto un documento andando a vedere, per quel documento, le parole a cui corrisponde un valore non nullo, e potremmo anche, per ogni parola, dire in quali documenti essa compare con un procedimento analogo.

Quel che accade è che l'algoritmo LSI, durante la parte precomputata, tramite la decomposizione in valori singolari della matrice originale calcola un'approssimazione di rango basso della matrice dei termini e dei documenti che consente un calcolo approssimato ma più efficiente delle misure di affinità. Si osserva empiricamente che la riduzione di rango oltre ad aumentare la velocità migliora anche la precisione semantica riducendo il rumore.

Al momento dell'interrogazione invece l'interrogazione dell'utente viene rappresentata come un vettore k-dimensionale e confrontata con i documenti. Per determinare la somiglianza tra l'interrogazione e i documenti possono venire utilizzate varie misurazioni, ma quella in assoluto più nota e largamente più usata è una misura basata sul coseno e definita come segue.

Sia t il numero di termini trovati nella prima fase dell'algoritmo. Ad ogni pagina p è associato un vettore di t elementi d_p dove $(d_p)_j$ è il numero di occorrenze del termine j in p

Ad ogni interrogazione q si associa un analogo vettore d_q che ha un 1 in corrispondenza dei termini che compaiono nell'interrogazione, e uno 0 altrimenti. Si tratta di determinare l'affinità tra p e q:

$$\cos(\vec{d_p}\vec{d_q}) = \frac{\vec{d_p} \times \vec{d_q}}{\|d_p\| \cdot \|d_q\|} \tag{4.1}$$

Per capire il significato di questa formula, si pensi allo spazio multidimensionale rappresentato dai possibili argomenti a cui una pagina può appartenere: in tal caso, per ciascuna dimensione dello spazio vi sarà un vettore che rappresenta "quanto" la pagina tratta di quell'argomento. Considerando l'angolo compreso tra i vettori degli argomenti dell'interrogazione e quello della pagina tramite il coseno si riesce a stimare la vicinanza dei due vettori.

Viste le prestazioni di LSI e le critiche che gli sono state mosse in letteratura si è scelto di non implementare questo algoritmo.

4.2 Proximity

L'algoritmo di Proximity è un altro noto algoritmo endogeno che nell'opinione comune si ritiene sia utilizzato da quasi tutti i motori di ricerca commerciali. Questo algoritmo si basa su una idea molto semplice: più le parole dell'interrogazione sono vicine tra loro in una data pagina, più il punteggio di Proximity sarà alto per quella pagina. Escludendo le pagine con Proximity nulla, si ottiene una lista dei documenti che soddisfano l'interrogazione dell'utente; tale lista potrà poi essere passata ad algoritmi quali PageRank per affinarne il punteggio in una qualche maniera.

Si possono dare varie definizioni che permettono di ricavare il punteggio di Proximity: questo fatto — sicuramente non positivo — è dovuto sia alla possibile scelta dell'insieme di operatori inseribili in una data interrogazione sia alla scelta, completamente libera per chi fornisce l'implementazione, di considerare gli intervalli minimali o massimali.

Ad esempio, un punteggio possibile della Proximity è dato dalla formula:

$$s = \frac{N_q}{L_{min}} \tag{4.2}$$

dove N_q rappresenta il numero di parole presenti nella interrogazione q, mentre L_{min} rappresenta la lunghezza dell'intervallo minimo. L'intervallo minimo risulta essere definito come l'intervallo di lunghezza minima che si può trovare in un documento soddisfacente l'interrogazione. Tale grandezza può essere pari al numero di parole della *query* se l'*operatore booleano* utilizzato nell'interrogazione è il solo `AND`, minore se si utilizzano anche altri operatori booleani (ad esempio se si usa l'`OR`).

Nel caso del motore Ubi si è scelto di fornire una implementazione basata

sui seguenti operatori logici, come presentato in [22]:

- `AND` — operatore di default, vengono ricercate corrispondenze con tutti i due blocchi forniti come argomento;
- `OR` — richiede la presenza di almeno uno dei due blocchi forniti come argomento;
- `BLOCK` — richiede che le parole inserite come argomento del `BLOCK` tra loro siano consecutive, nell'ordine in cui vengono fornite;
- `CLUSTER` — si comporta come l'operatore `BLOCK`, ma non richiede che l'ordine delle parole sia quello fornito dall'utente;
- `NEAR` — permette di selezionare gli intervalli tali che le parole fornite come argomento siano al più ad una certa distanza l'una dall'altra (la distanza è fornita dall'utente);
- `LOWPASS` — permette di scegliere soltanto gli intervalli di al più una certa lunghezza all'interno di un documento che soddisfa l'interrogazione.

Si noti la mancanza dell'operatore `NOT`, non implementato a causa della difficoltà di interpretazione della sua semantica (vedi [22]). In particolare, tutti questi operatori sono quelli presenti nel software utilizzato per determinare i documenti che soddisfano le interrogazioni degli utenti. Questo software in particolare ha la caratteristica, dato un documento che soddisfa l'interrogazione, di restituire l'intervallo minimo all'interno di tale documento che soddisfa tale interrogazione. Alternativamente, possono venire fornite informazioni quali l'intervallo minimo e massimo, la lunghezza media degli intervalli, il numero di intervalli contenuti all'interno di un documento, ecc.

La formula fornita precedentemente per il calcolo del punteggio di Proximity nel nostro caso è stata leggermente variata e raffinata. Infatti, volendo avere valori di Proximity normalizzati tra 0.0 e 1.0 non si può considerare il numero di parole dell'interrogazione: cosa succederebbe con una interrogazione fatta di parole tutte in `OR`? In tal caso il numero delle parole sarebbe sicuramente molto maggiore della lunghezza dell'intervallo minimo, che sarà pari ad 1 essendo la lunghezza minima una qualunque delle parole. Si è perciò scelto di considerare, in luogo del numero delle parole dell'interrogazione, la minima lunghezza possibile di un intervallo soddisfacente la *query*. Ad esempio, si consideri l'interrogazione: `(microsoft corporation) OR (gates)` In tal caso il numero di parole è 3. Se trovassimo una pagina con la parola `gates`, usando la prima formula otterremmo un punteggio pari a 3; usando invece l'idea illustrata prima la minima lunghezza possibile sarebbe pari a 1 (ossia il numero di parole contenute nel secondo *token* dell'interrogazione, cioè la parola `gates`). In tal caso il punteggio sarà quindi pari a $1/1 = 1.0$.

L'algoritmo di Proximity è di indubbia utilità nel fornire un punteggio significativo alle pagine, e senza alcun dubbio il punteggio fornito è uno dei più chiari e meglio funzionanti tra quelli esistenti. Tuttavia anche questo algoritmo ha una pecca: ci stiamo riferendo al punteggio che viene fornito nel caso di un'interrogazione mono-parola o di un'interrogazione con parole singole tutte in `OR`.

In tali situazioni infatti il punteggio fornito da Proximity per i documenti che soddisfano l'interrogazione sarà sempre 1.0 (mentre sarà sempre 0.0 nel caso dei documenti che non la soddisfano), come si può facilmente verificare dalla formula appena introdotta. Questo significa che, per ogni documento che soddisfa l'interrogazione, il suo punteggio sarà sempre lo stesso. Questa

è ovviamente una limitazione piuttosto pesante, essendo le interrogazioni mono-parola molto diffuse — si vedano, ad esempio, i rapporti periodici dei motori di ricerca [23].

Come comportarsi in tali casi? Vi sono due semplici soluzioni: aggiungere un'altra euristica, oppure non farlo. La scelta di abbinare un altro algoritmo od un'altra euristica a Proximity non è stata tuttavia fatta in funzione della seguente considerazione: le interrogazioni mono-parola sono, per loro stessa natura, tese ad individuare pagine molto generali, molto note e facilmente raggiungibili. Stiamo parlando di pagine che corrispondono alle *home-page* di aziende famosissime, o di enti governativi, o di siti rinomati. I termini che si possono cercare sono quindi tutti marchi famosi, oppure parole comuni della lingua in cui viene fatta l'interrogazione.

In tali casi vi sono ben altri algoritmi che sopperiscono alle mancanze di Proximity, e non ha senso introdurre una ulteriore euristica per questi casi speciali (seppur diffusi). Il rischio è non soltanto quello di appesantire un software che deve lavorare a ritmi serrati e a velocità impressionanti, ma anche quello — ben peggiore — di introdurre *spam* nei risultati.

Va infine notato che recentemente alcuni motori di ricerca, in particolare Google, sembrano aver introdotto una variante all'algoritmo di Proximity basata sul fatto che gli intervalli di *match* soddisfino l'interrogazione dell'utente con le parole in ordine o meno rispetto all'interrogazione originaria. Ossia, se inseriamo la *query* `bill gates` sarà preferita una pagina contenente le due parole in questo ordine piuttosto che quella con le parole in ordine inverso. Non è parso tuttavia strettamente necessario implementare una tale euristica: infatti accade spesso che un utente non inserisca i termini nell'ordine nel quale si aspetta che essi compaiano, ma capita talvolta che li inserisca nell'ordine in cui gli vengono in mente pensando all'argomento per cui sta facendo

la sua interrogazione. Va anche segnalato che non risulta affatto chiaro come si possa calcolare il numero di inversioni su interrogazioni complicate, ossia su interrogazioni che contengano alcuni degli operatori *booleani* visti.

Si potrebbero, ad esempio, adottare sofisticate tecniche di calcolo della massima sottosequenza crescente, ma questo esula dagli obbiettivi della tesi.

4.3 Frequency-count

L'algoritmo di Frequency-count è un algoritmo endogeno molto semplice ma che nel contempo risulta decisamente inaffidabile nei risultati restituiti quando applicato alle pagine web. L'idea su cui l'algoritmo di Frequency-count si basa, come dice il nome stesso, è il conteggio della frequenza con cui una parola compare in un dato documento. Anch'esso, come gli altri algoritmi, fornisce soltanto i documenti che soddisfano l'interrogazione dell'utente. Tuttavia, in maniera ben peggiore degli altri casi, tale algoritmo fornisce punteggi davvero improbabili che non rispecchiano minimamente — salvo rarissime circostanze — la sensazione di "aver centrato l'obbiettivo" che il tipico utente di un motore di ricerca ha quando vede comparire i risultati della sua interrogazione.

In realtà questo algoritmo non è inadatto in ogni situazione: vi sono infatti casi particolari in cui la sua adozione può realmente portare benefici (si veda, ad esempio, l'algoritmo di AnchorRank che si presenterà successivamente), ma in generale non è un algoritmo ottimale per dare un primo punteggio alle pagine web.

Un interessante quesito sorge nel momento in cui ci si chiede come sia definito il punteggio di tale algoritmo. Si è detto che esso conta il numero di volte nella quale l'interrogazione compare nel documento considerato, e pertanto ne conta il numero di intervalli. Ma come assegnare il punteggio? Una buona scelta risulta essere quella di dividere tale numero di intervalli per il numero di parole presenti all'interno del documento considerato. In tal caso si otterrà un punteggio normalizzato tra 0.0 (ossia nessun *match*) e 1.0 (ossia tutto il documento è un unico grande *match*).

Si è detto che Frequency-count non va bene per assegnare un punteg-

gio significativo ad una pagina, almeno in prima battuta. Questo è dovuto al fatto che spesso e volentieri le pagine realmente importanti contengono poche volte le parole delle tipiche interrogazioni che dovrebbero invece individuarle: si pensi a pagine fatte di sole immagini, oppure a pagine quali `http://www.fiat.com/` nelle quali la parola "automobili" non compare neppure una volta.

Conteggiare il numero di occorrenze di una parola all'interno di un documento è inoltre rischioso perché si rischia di incappare molto facilmente nel fenomeno dello *spamming*. Sarebbe infatti sufficiente inserire una parola con la quale si desidera essere trovati per un numero molto elevato di volte all'interno di una pagina web per salire rapidamente in cima all'indice del motore di ricerca. Pur non essendovi fonti nella letteratura di settore al riguardo che provino quanto si sta affermando, la conoscenza comune riporta un esempio storico di questa spiacevole situazione in una delle primissime versioni del motore di ricerca Altavista [24], il quale si pensa che adottasse algoritmi basati su Frequency-count (prima di passare ad LSI). Ancora oggi, a distanza di anni, si trovano in rete testimonianze di quanto fosse facile "salire in cima" agli indici riempiendo le proprie pagine di parole che non riguardavano il contenuto della pagina. Di tale sfruttamento si sono ben presto resi conto anche gli ingegneri stessi di Altavista, che hanno provveduto poi a migliorare i loro algoritmi di ranking rendendo così la "scalata" molto più difficile.

Si è anche detto che l'algoritmo di Frequency-count non è del tutto inutilizzabile, almeno in alcune circostanze. Infatti tale algoritmo ha senso quando si considerano pagine che non sono legate tra loro, e che quindi non sono facilmente influenzabili. Tale situazione è quella tipica delle ancore, ossia del testo utilizzato per i collegamenti ipertestuali. Per maggiori informazioni si rimanda al paragrafo 5.1.4.

4.4 Prominence

Un algoritmo endogeno a cui si può pensare ma che in letteratura non è stato mai presentato è l'algoritmo di Prominence, ossia un algoritmo che dà più importanza al fatto che le parole ricercate dall'utente siano relativamente "in testa" ad un documento piuttosto che "in coda".

Si pensi, ad esempio, a quel che accade nei documenti estratti seguendo i link dei quotidiani che pubblicano i loro articoli in Internet: in tal caso il fatto che le parole ricercate siano relativamente in testa al documento piuttosto che in coda fa una differenza sostanziale, in quanto nel primo caso le parole faranno probabilmente parte del titolo dell'articolo oppure del suo occhiello, mentre nel secondo caso potrebbero essere contenute nel testo del documento come semplice riferimento, e non essere pertanto centrali all'interno dell'articolo in questione.

A questo punto vale la pena di chiedersi se un tale algoritmo scali sul web, ossia se tale algoritmo può essere applicato anche ad insiemi di documenti differenti ed eterogenei come quelli presenti sulle miriadi di *host* sparsi per il pianeta. Una formula che si potrebbe proporre è la seguente:

$$s = \frac{E_{dxmin}}{L_{doc}} \tag{4.3}$$

dove E_{dxmin} indica l'estremo destro dell'intervallo minimo di *match* mentre L_{doc} indica la lunghezza del documento in numero di parole.

Analizziamo tale formula. Anzitutto vorremmo in generale che il punteggio assegnato ad una data pagina non dipendesse dalle dimensioni del documento, altrimenti sarebbero sempre privilegiati i documenti più brevi in quanto, per forza di cose, gli intervalli di *match* sarebbero sempre nelle prime

"righe" dei documenti. Per evitare questo si è introdotta la normalizzazione per la lunghezza del documento (tale lunghezza è espressa in parole, uniformemente con la misura dell'estremo destro dell'intervallo, anch'essa espressa in numero di parole).

Il numeratore dell'espressione appena vista è invece dato dalla posizione (espressa sempre in numero di parole) dell'ultima parola tra quelle contenute nell'interrogazione e presente nel documento in questione. Questa misura ci dà una chiara indicazione della posizione fino alla quale l'intervallo si estende, e la formula vista risulta pertanto essere una ragionevole misura per stimare la posizione relativa in cui compare l'intervallo di *match* all'interno del documento considerato.

Si noti che nella stima fornita non è presente alcuna indicazione riguardo alla lunghezza dell'intervallo; questo, che apparentemente può sembrare un grosso difetto, viene in realtà già calcolato nella normalizzazione del punteggio rispetto alla lunghezza del documento. Si supponga infatti di trovare un documento in cui le parole dell'interrogazione sono abbastanza distanti tra di loro, ma si trovano ragionevolmente in testa al documento. In una tale situazione l'unico modo per avere un punteggio elevato è che il documento sia molto lungo, come discende direttamente dalla formula. In generale quindi vengono prediletti intervalli brevi rispetto ad intervalli lunghi, in quanto gli intervalli lunghi tendono ad essere centrati verso la fine o la metà del documento che si sta considerando.

Capitolo 5

L'aggregatore di ranker

Per rispondere in maniera efficiente alle interrogazioni degli utenti è necessario utilizzare più algoritmi di *ranking*, uno in contemporanea all'altro. Purtroppo[1] non esiste al giorno d'oggi un algoritmo ideale che restituisca i risultati che si desidererebbero; esistono solamente buoni algoritmi che, combinati tra loro, restituiscono i risultati sperati.

Per tali ragioni si è creato una sorta di *aggregatore di ranker*, ossia uno strumento software in grado di interrogare simultaneamente i vari algoritmi ed eseguire le varie euristiche per poi restituire i risultati così ottenuti ed opportunamente pesati.

Per poter aggregare i risultati di vari *ranker* vi sono varie tecniche. Noi abbiamo deciso di adottare una aggregazione lineare. Il vantaggio di questa soluzione è che questo approccio permette di essere facilmente cambiato se si desiderano cambiare i pesi dei vari algoritmi per fare esperimenti.

Gli algoritmi che sono stati utilizzati per dare i punteggi alle pagine sono cinque:

[1] O per fortuna, se si pensa al fatto che fare *spamming* con più algoritmi è più difficile.

- PageRank, per scremare i punteggi delle pagine;
- Proximity, usata per dare un punteggio al testo delle pagine;
- TitleRank, per dare un punteggio al titolo delle pagine;
- URLRank, utilizzato per considerare nel punteggio totale anche l'URL del sito web;
- AnchorRank, per considerare l'esogenicità derivante dalle ancore con cui i documenti sono puntati.

Si noti che ciascuno di questi algoritmi opera sull'intera collezione delle pagine web recuperate; tuttavia, poiché si desidera restituire all'utente soltanto una frazione di esse, è compito dell'aggregatore scartare le pagine con punteggio nullo in qualcuno dei vari ranker a disposizione.

Inizialmente l'insieme di risultati da restituire era fornito dai soli documenti che soddisfacevano l'interrogazione nel testo con un punteggio non nullo, basandosi per la scelta sui punteggi restituiti dall'algoritmo di Proximity. Successivamente, nel momento in cui si sono implementate le ancore, si è deciso di fondere questa lista di documenti assieme a quella restituita dall'algoritmo di AnchorRank (sempre soltanto per i documenti con punteggio non nullo in quest'ultimo ranker). Questa scelta è stata fatta perché spesso una pagina non contiene tutti i termini con i quali in realtà è conosciuta, un po' come accade per l'algoritmo di Hits (paragrafo 3.4. Ad esempio, il sito dell'agenzia ANSA (`http://www.ansa.it/`) non soddisfa l'interrogazione `agenzia ansa`, mentre le ancore con le quali viene puntata sì (per maggiori dettagli si veda il paragrafo 5.1.4).

I motori di ricerca commerciali, come ad esempio Google, si ritiene comunemente che utilizzino un algoritmo simile al seguente per trovare i documenti da restituire (ad esempio, si vedano le discussioni su [25]):

- trova i documenti che soddisfano la *query* il testo (Proximity);
- trova i documenti che soddisfano la *query* nei titoli (TitleRank);
- trova i documenti che soddisfano la *query* nelle URL (URLRank);
- trova i documenti che soddisfano la *query* nelle ancore (AnchorRank);
- unisci gli insiemi così trovati;
- calcola PageRank su questo insieme di documenti;
- combina linearmente i punteggi trovati dai vari algoritmi per ogni pagina;
- restituisci l'insieme di pagine con il punteggio associato a ciascuna di esse,

mentre l'algoritmo da noi utilizzato risulta essere il seguente:

- trova i documenti che soddisfano la *query* nel testo (Proximity);
- trova i documenti che soddisfano la *query* nelle ancore (AnchorRank);
- unisci gli insiemi così trovati;
- calcola PageRank, TitleRank ed URLRank su questo insieme di documenti;
- combina linearmente i punteggi trovati dai vari algoritmi per ogni pagina;

- restituisci l'insieme di pagine con il punteggio associato a ciascuna di esse.

Il motivo per cui si è scelto un algoritmo più semplice è stato quello di voler evitare di introdurre troppe pagine che non necessariamente riguardano l'interrogazione in esame e che pertanto rischiano di introdurre rumore nei risultati.

5.1 Algoritmi utilizzati

Nelle prossime sezioni verranno presentati tutti gli algoritmi di *ranking* che sono stati implementati nel lavoro di tesi e per ciascuno di essi verrà data un'accurata descrizione del suo funzionamento.

Per la presentazione di PageRank e Proximity, algoritmi noti in letteratura ed inseriti all'interno del motore di ricerca Ubi, si rimanda alle rispettive sezioni.

5.1.1 PageRank e Proximity non bastano

Una prima aggregazione elementare consiste nell'associare i risultati forniti dall'algoritmo di PageRank con quelli restituiti dall'algoritmo di Proximity. Tuttavia, una aggregazione basata solamente su questi due algoritmi risulta piuttosto inefficace, essendo questi due *ranker* insufficienti da soli per mettere in piedi un motore di ricerca sufficientemente preciso e raffinato. In particolare, il difetto più macroscopico risulta essere quello che spesso e volentieri tra i primi dieci risultati restituiti non compare la pagina cercata laddove la medesima interrogazione fatta ai motori commerciali individua immediatamente in prima posizione il documento ricercato.

La strada che si è seguita è stata utilizzare una combinazione lineare dei risultati di questi due algoritmi, pesando maggiormente PageRank poiché i suoi punteggi sono normalizzati a 1 sull'intera collezione delle pagine, mentre i punteggi di Proximity non lo sono. Pur potendo cambiare i pesi assegnati a questi due algoritmi, la situazione ugualmente non migliora anche scegliendo altre combinazioni di pesi.

Si riporta qui di seguito, ad esempio, quel che accade inserendo l'interrogazione `microsoft` al motore con in uso solamente gli algoritmi di PageRank e Proximity (vengono mostrati solamente i primi dieci risultati). I pesi scelti erano, in questo caso, di 8110.67 per PageRank (numero ottenuto grazie alla proporzione 5.3) e di 0.5 per Proximity (scelto per via di alcuni esperimenti condotti — si veda il paragrafo 5.2).

---

1. `http://www.adobe.it/products/acrobat/readstep.html`
   Acrobat Family
2. `http://www.canalerisparmio.it/`
   risparmio le offerte di lavoro e gli annunci
3. `http://www.html.it/`
   HTML.it
4. `http://www.repubblica.it/`
   Repubblica.it
5. `http://www.multiplayer.it/`
   Welcome to Multiplayer.it
6. `http://www.msn.it/global/incompbrowser.asp`
   Nota per gli utenti
7. `http://arianna.libero.it/`
   ARIANNA — Il Motore di Ricerca Italiano
8. `http://www.marcomedia.it/`
   Marco Medi@ — Computer Palmari - Pocket PC — GPS WAP GSM GPRS UMTS
9. `http://www.xtend.it/`
   Xtend new media Italia — soluzioni Internet, eBusiness e communication, [...]
10. `http://www.asm-settimo.it/informatica/`
    ASM Informatica Settimo Torinese

La stessa interrogazione, sottoposta a Google, individua invece subito il sito desiderato.

Come si vede invece, non soltanto nei primi dieci risultati non compare alcuna pagina afferente a Microsoft Corporation[2], ma non si trova neppure traccia di pagine di terzi riguardanti l'azienda o i suoi prodotti. Questa interrogazione risulta emblematica nel dimostrare che, da soli, PageRank e

[2]Se escludiamo l'unica pagina del sito `http://www.msn.it/`, che peraltro contiene solamente una nota informativa per gli utenti.

Proximity non bastano. L'unico fatto positivo di questo esperimento è che le pagine trovate ruotano perlomeno attorno all'argomento ricercato, l'informatica; si sono quindi sviluppate ulteriori euristiche, che verranno illustrate nei prossimi capitoli, e che sono state introdotte proprio per affinare i risultati.

5.1.2 TitleRank - punteggio sui titoli

Come si è visto PageRank e Proximity da soli non possono assolutamente bastare per restituire all'utente risultati mirati e competitivi. Essendo i titoli una risorsa preziosissima per le pagine, dovendo tipicamente riassumere in poche parole il contenuto dell'intero documento, si è pensato di adottare una qualche tipologia di punteggio che fosse basata sui titoli.

In generale, ciò che saremmo interessati a fare è vedere se il titolo della pagina soddisfa — interamente od in parte — l'interrogazione dell'utente, e in caso affermativo darne una qualche forma di punteggio.

All'inizio si era pensato di dare un punteggio anche ai titoli che soddisfacevano solamente in parte l'interrogazione dell'utente; un esempio potrebbe essere un titolo del tipo: `microsoft corporation` in relazione alla *query* `microsoft corp`. In tal caso infatti l'interrogazione non sarebbe contenuta nel titolo esaminato, tuttavia una parte di essa (la parola `microsoft`) lo sarebbe.

Questa scelta non è tuttavia stata seguita da noi dal momento che il software utilizzato per le interrogazioni e per determinare i documenti che soddisfano una interrogazione dalla lettura degli indici inversi non restituisce i *match* parziali, ma solamente quelli completi. Non volendo complicare un codice che viene usato anche in altri ambiti e che deve essere efficiente si è

perciò scelto di considerare solamente i titoli che soddisfano completamente l'interrogazione.

Il punteggio che viene assegnato ad un titolo che risulta soddisfare l'interrogazione è calcolato in maniera simile al punteggio di Proximity, essendo dato da

$$s = \frac{1}{(L_{min} - N + 1)} \tag{5.1}$$

dove L_{min} è la lunghezza dell'intervallo minimo di *match* mentre N è il numero di parole dell'interrogazione[3].

Si noti che qui, contrariamente a quel che accade con l'algoritmo di Proximity, i valori che vengono assegnati sono, nella stragrande maggioranza dei casi, valori vicini a 1.0 oppure 0.0, essendo i titoli molto corti; infatti sotto tale ipotesi le parole inserite nella *query* dell'utente verranno a trovarsi molto vicine nel titolo, quasi sempre a distanza 1 o comunque a breve distanza, ed il punteggio conseguente sarà normalmente più elevato di quello della Proximity.

Inserendo anche questo algoritmo di *ranking* all'interno del sistema di aggregazione dei risultati discusso in precedenza (si veda il capitolo 5) la qualità delle ricerche migliora sensibilmente fino ad arrivare già, su ricerche particolari, a potersi confrontare con i motori di ricerca commerciali.

Vengono qui riportati i primi dieci risultati ottenuti attivando l'algoritmo di TitleRank con peso 1.0 sulla *query* `governo` (i risultati vengono combinati con PageRank e Proximity, con i pesi discussi nel paragrafo 5.2):

---

[3]O meglio, come visto con la Proximity, la lunghezza minima possibile di un intervallo di *match*.

1. `http://www.governo.it/`
   Governo Italiano — Home page
2. `http://www.governo.it/servizi/ricerca.asp`
   Governo Italiano — Ricerca nel sito
3. `http://www.palazzochigi.it/`
   Governo Italiano — Home page
4. `http://www.palazzochigi.it/servizi/ricerca.asp`
   Governo Italiano — Ricerca nel sito
5. `http://www.maggioli.it/`
   MAGGIOLI - Il Partner del Governo Locale
6. `http://www.maggioli.it/index.htm`
   MAGGIOLI - Il Partner del Governo Locale
7. `http://www.progettoecotur.it/`
   Il Governo del Territorio tra beni culturali, [...]
8. `http://www.parusia.it`
   Il Governo del Territorio tra beni culturali, [...]
9. `http://www.prefettura.messina.it/default.htm`
   Ufficio Territoriale del Governo di Messina — Homepage
10. `http://www.prefettura.messina.it/`
    Ufficio Territoriale del Governo di Messina — Homepage

Come si può osservare, le pagine trovate grazie al peso ulteriore attribuito contengono tutte quante nel titolo la parola ricercata, `governo`. I documenti restituiti sono inoltre incentrati sull'argomento ricercato, e non sono più estremamente fuorvianti come accadeva con il solo utilizzo di PageRank e Proximity (si veda il paragrafo 5.1.1)

5.1.3 URLRank - punteggio sulle URL

Nonostante l'introduzione di un algoritmo quale TitleRank all'interno del sistema di aggregazione (vedi paragrafo 5) un problema persistente è quello di non trovare il sito web che si desidera trovare pur fornendone il suo dominio. Ad esempio, può capitare che ricercando `html` su un insieme di pagine estratte dalla gerarchia `.it` non si trovi in prima posizione il noto `http://www.html.it/`.

Tale fenomeno è spiacevole e frustrante per l'utente medio, che non si vede comparire in prima posizione il sito che cercava. Questo è tuttavia proprio il comportamento che vogliamo simulare, ossia quello di trovare un sito dando il suo nome a dominio.

Per sopperire a questa mancanza, al giorno d'oggi oramai presente solo in motori di ricerca dai risultati molto scadenti, si è scelto di creare un nuovo algoritmo che desse un punteggio anche alle URL delle pagine. Questo *ranker* è nella forma del tutto simile al già presentato TitleRank, dove, come si è detto, al posto dei titoli vengono usate le URL delle pagine.

La parte interessante dell'intero algoritmo è tuttavia a monte, ossia risiede nel recupero delle parole contenute nelle URL delle pagine. Infatti alcuni domini, come ad esempio `http://www.comunemilano.it/`, contengono due parole: nel nostro caso le parole `comune` e `milano`. Quel che vorremmo è che la *query* `comune milano` risultasse positiva a questo algoritmo, e che pertanto trovasse il sito in questione. Per fare tutto ciò sarà ovviamente necessario estrarre le parole di cui sono composte le URL, per poter poi creare gli indici inversi associando ad ogni documento le parole di cui è composta la sua URL (ammesso che per un dato documento vi siano parole contenute nell'URL). L'idea che si è sfruttata è stata quella di costruire un *ternary search tree* [26] con tutte le parole che si sono trovate analizzando il contenuto delle pagine web[4] ed andando a fare una ricerca all'interno dell'albero così costruito.

In tal modo le parole vengono emesse solamente nel momento in cui la ricerca arriva ad una foglia dell'albero di ricerca, e si è sicuri di non emettere nulla nel caso in cui una URL non contenga parole esistenti.

[4]Eliminando preventivamente i probabili errori di battitura, ossia considerando solamente parole con frequenza superiore ad una certa soglia, posta a 100 nel nostro caso.

Un punto importante risulta essere il seguente: cosa va considerato come URL? Ossia, va scomposto soltanto il nome a dominio oppure anche il percorso che contraddistingue un documento? Nel nostro caso abbiamo scelto il secondo approccio, essendo il primo alle volte ingannevole e più suscettibile al fenomeno dello *spam* più volte illustrato: si pensi, ad esempio, ad un utente malizioso che creasse percorsi lunghissimi contenenti moltissime parole comuni. Nell'implementazione fornita, si è inoltre scelto di eliminare dalle URL prefissi e suffissi comuni, come `http://`, `.com`, `.it`, `www.`, ecc.

Si noti infine che tenendo questi prefissi e suffissi e non eliminando il percorso dalle URL si potrebbe fare ricerca per indirizzo delle pagine: questo vorrebbe dire che se un utente inserisse l'URL di una pagina web, tale pagina potrebbe venire trovata (cosa che, al momento, non avviene).

Viene qui riportato un esempio di come URLRank riesca ad inviduare con ragionevole precisione pagine degli *host* che contengono, nel nome del dominio, le parole ricercare. Nel caso in esame si è ricercato `html` e i risultati con URLRank attivato (con peso[5] 1.0) sono stati i seguenti:

---

1. `http://www.html.it/`
   HTML.it
2. `http://www.html.it/guida/index.html`
   HTML.it — Guida all'HTML
3. `http://www.virgilio.it/`
   VIRGILIO
4. `http://hosting.html.it/`
   HTML HOSTING — le migliori offerte di hosting e housing in Italia
5. `http://download.html.it/`
   HTML DOWNLOAD — Software per webmaster
6. `http://adsl.html.it`
   ADSL.html.it — le offerte italiane di ADSL e HDSL
7. `http://webnews.html.it/`
   WEBnews.html.it — L'informazione dall'internet News e articoli [...]
8. `http://webnews.html.it/focus/focus.htm`
   Tutti gli articoli | Focus | WEBNEWS.HTML.it

[5]Si veda il paragrafo 5.2.

9. `http://webtool.html.it/forum/index.html`
   WEBtool.it — Forum di discussione
10. `http://webtool.html.it/`
    WEBtool.it — Servizi gratuiti per webmaster

Come si può vedere sono state trovate molte pagine tratte da un noto sito italiano sul linguaggio di *markup* HTML; le pagine restituite hanno la particolarità di contenere[6], nel nome dello *host* la parola ricercata tramite il motore. In prima posizione, in particolare, appare l'*home-page* vera e propria del sito in questione. Questo fatto ben rappresenta l'idea alla base di URL-Rank: spesso ciò che un utente vuole ricercare è il sito la cui URL contiene le parole che ha inserito[7].

5.1.4 AnchorRank - punteggio sulle ancore

Una euristica che in letteratura [28] si ritiene sia piuttosto buona è quella basata sul punteggio da associare alle ancore, ossia al testo dei collegamenti ipertestuali presenti nei documenti.

Il motivo per cui una tale euristica può funzionare è lo stesso per cui funziona l'algoritmo di Hits (vedi paragrafo 3.4): le autorità, ossia le pagine importanti, sono puntate da molte altre pagine. Inoltre le parole con le quali vengono puntate sono anche le parole con le quali tali pagine sono conosciute, come ad esempio:

```
agenzia ansa -- http://www.ansa.it/
george bush -- http://www.whitehouse.gov/president/
```

[6] Tutte tranne una, l'*home-page* di Virgilio, in quella posizione a causa del suo alto valore di PageRank.

[7] Si veda, ad esempio, l'analisi di [27] nella sezione intitolata "What Users Expected from Google".

Tale euristica è utilizzata nel noto motore di ricerca Google per stessa ammissione dei loro creatori (si vedano [29] e [30]) e mira ad individuare le pagine in maniera esogena, ossia non sulla base di ciò che le pagine dicono di trattare[8] ma sulla base di ciò che gli altri affermano che loro trattino.

I risultati che si ottengono applicando questo algoritmo sono sorprendenti, e utilizzare AnchorRank riesce a migliorare di molto la sensazione di qualità che si ha andando a vedere i link restituiti. Inoltre questo algoritmo, almeno nella nostra implementazione, rivaluta in un certo senso il cosiddetto Frequency-count (vedi paragrafo 4.3), un *ranker* basato sul semplice conteggio del numero di intervalli che soddisfano una data interrogazione. Come si era notato, l'algoritmo di conteggio della frequenza era molto suscettibile a *spam* quando veniva applicato alle pagine web; in questo caso tuttavia il fatto che le ancore siano tra loro poco legate rende il conteggio della frequenza un approccio realistico, in quanto i creatori di tali ancore generalmente non si conoscono e quindi i testi sono difficilmente influenzabili salvo casi molto particolari. Va comunque tenuto conto che il punteggio dato dalle ancore viene in realtà influenzato solamente in parte (e per di più in maniera logaritmica) da questo punteggio dato da Frequency-count. La parte più significativa è ancora data da un punteggio basato su un conteggio di Proximity.

In dettaglio, l'algoritmo di AnchorRank fornisce un punteggio così calcolato:

$$s = \frac{N}{L_{min}} \cdot (1 + \ln(C)) \tag{5.2}$$

dove C è il numero di occorrenze della interrogazione all'interno delle

[8] Il che può essere fuorviante, nel caso di pagine che fanno *spamming*, oppure non essere sufficiente, come in pagine fatte interamente di immagini, che non sono ad oggi ancora pienamente indicizzabili.

ancore che puntano al documento, N il numero di parole contenute nella *query*[9] e L_{min} la lunghezza minima dell'intervallo che soddisfa l'interrogazione. Il fatto che al logaritmo venga sommato 1 non è un caso: il motivo è che il comportamento desiderato è che nel caso di un documento puntato con una sola ancora si desidera che il punteggio del secondo fattore dia come risultato 1, ossia che il suo valore non influenzi il primo fattore. Nel caso invece che un documento contenga più ancore, l'addendo aggiunto (cioè 1) verrà di molto mitigato dalla presenza del logaritmo.

Ad esempio, si supponga che la pagina `http://www.microsoft.com/` sia puntata da altre 1000 pagine tramite 1000 ancore. Tra queste 1000 ancore, ve ne siano 900 che, tra le altre parole, contengono le parole "microsoft corporation". Si analizzi ora l'interrogazione `microsoft corporation`. Sotto tali ipotesi il punteggio di AnchorRank sarà quindi dato da: $(2/2) \cdot (1 + \ln(900)) = 7.8023948$.

Questo è naturalmente soltanto uno dei possibili punteggi assegnabili. Gli unici problemi riscontrati sono su alcune pagine che sono state create *ad hoc* per influenzare proprio algoritmi basati su questa tecnica. L'unico modo per eliminare tali influenze sembra essere quello di ricorrere a sofisticate quanto pericolose euristiche di *anti-spam* che non sono state implementate prevalentemente perché non si è voluto correre il rischio di proporre euristiche basate solamente sull'intuito di chi le scrive e non necessariamente funzionanti in ogni caso. Quel che però è certo è che i motori di ricerca commerciali adottano tecniche del genere, anche per loro stessa ammissione, ma spesso tali tecniche vengono criticate proprio per la loro inefficacia e per il fatto di penalizzare siti che non dovevano esserlo in quanto non portatori di *spam* al motore. A proposito di tali critiche si possono leggere gli archivi di [25].

[9] O meglio, la lunghezza minima possibile di un intervallo soddisfacente l'interrogazione.

Altre tecniche possibili per l'assegnazione del punteggio e probabilmente più resistenti allo *spam* sarebbero quelle di considerare soltanto il numero di ancore verso una data pagina che contengono le parole dell'interrogazione, oppure considerare soltanto il numero di pagine che contengono le ancore verso una data pagina con testo che soddisfa l'interrogazione. In questi due modi, ad esempio, il fattore logaritmico sarebbe meno facilmente manipolabile.

Estrazione di ancore

L'estrazione delle ancore dai documenti raccolti è un procedimento non solo lento ma anche non banale. Infatti, a parte le difficoltà tecniche di fornire una implementazione adeguata, non è chiaro cosa vada considerato testo dell'ancora e cosa no. Ad esempio, si potrebbe considerare come testo dell'ancora tutto il testo del collegamento ipertestuale più un altro numero di parole prima e dopo il link, oppure da un segno di interpunzione precedente al link fino ad un segno di interpunzione successivo, oppure ancora considerare come ancora non soltanto il testo del link ma anche il testo attorno che non sia separato da *tag* HTML. Di tutte le euristiche provate quella di gran lunga più efficace è stata anche la più semplice possibile: il testo di un ancora è solamente il testo contenuto all'interno dell'elemento `A` dell'HTML. Questa euristica sembra anche essere quella adottata da Google, almeno da quanto risulta dal *reverse-engineering* fatto sulle *query*.

L'estrazione delle ancore tra l'altro può anche essere condizionata dal fatto di considerare le ancore derivanti da pagine affiliate o meno, od eventualmente dal come pesarle. Nel nostro caso, per ridurre il più possibile la possibilità di *spam* diretto od indiretto abbiamo scelto di considerare solamente le ancore *extra-host*, ossia i testi dei link esterni allo *host* su cui

le pagine risiedono. Il tal modo, ad esempio, un link contenente la parola `microsoft` proveniente da `http://www.microsoft.com/italy/` e diretto verso `http://www.microsoft.com/` non verrebbe affatto considerato, mentre verrebbe considerato tale link nel caso lo stesso link provenisse da `http://www.zdnet.com/`. Si possono ovviamente proporre anche altre euristiche, come viene accennato in [29].

I rischi dell'utilizzo delle altre tecniche appena esposte e di altre che si potrebbero pensare è quello di introdurre molto rumore nei risultati portando, ad esempio, il sito `http://www.virgilio.it/`, molto puntato sulla rete italiana in una vasta varietà di contesti e di parole utilizzate, in cima a pressoché qualunque ricerca (cosa realmente successa in fase di sperimentazione).

In un capitolo precedente si è accennato al fatto che le ancore possono portare nuovi risultati rispetto a quelli trovati dall'algoritmo di Proximity: questo può accadere nel caso in cui una pagina non contenga le parole con cui è conosciuta, come ad esempio i motori di ricerca che, al loro interno, non contengono le parole "motori di ricerca". Nel nostro caso abbiamo deciso l'introduzione di questa variante, visti i benefici che si hanno. Sempre basandosi sul *reverse-engineering*, sembra che anche i motori di ricerca noti adottino questa politica.

A questo punto si potrebbe delineare una ulteriore euristica, che però non è stata implementata: eliminare dal testo delle pagine le parole inserite all'interno di un'ancora, visto che esse si riferiscono generalmente alla pagina puntata e non alla pagina contenitore. Questo porterebbe probabilmente non soltanto ad avere meno parole all'interno delle pagine web e conseguentemente ad indicizzazione e ricerche più rapide, ma anche a una migliore precisione nei risultati visto che verrebbero eliminati i *falsi positivi* causati dalla com-

parsa delle parole dell'interrogazione all'interno dei link. Se si scegliesse di implementare tale euristica sarebbe tuttavia da valutare con attenzione cosa farne esattamente dei testi delle ancore che puntano a pagine non recuperate, vuoi perché il *crawl* è stato interrotto, vuoi perché il link non è più valido. Forse la miglior soluzione sarebbe di eliminare le parole contenute nelle ancore delle pagine puntatori solamente se la pagina puntata è stata recuperata.

Risultati di AnchorRank

Si riporta qui di seguito un esempio di ciò che accade chiedendo al motore di ricerca `repubblica`, con e senza ancore.

Questo è il risultato utilizzando le ancore:

---

1. `http://www.repubblica.it/`
   Repubblica.it
2. `http://www.repubblica.it/auto/index.htm`
   RepubblicAuto.com
3. `http://repubblica.extra.kataweb.it/repubblica/frameset[...]`
   la Repubblica Extra — Il giornale in edicola
4. `http://www.quirinale.it/`
   Presidenza della Repubblica
5. `http://www.quirinale.it/presidente/ciampi.htm`
   Carlo Azeglio Ciampi - biografia
6. `http://www.senato.it/senato.htm`
   Senato della Repubblica - Italia
7. `http://www.senato.it/`
   Home page Parlamento Italiano
8. `http://www.finanza.repubblica.it/`
   Borsa
9. `http://www.repubblica.napoli.it/`
   Repubblica edizione di Napoli
10. `http://testo.camera.it/_presidenti/`
    L'elezione del Presidente della Repubblica

Questo invece è il risultato se non si utilizzano le ancore:

---

1. `http://www.repubblica.it/`
   Repubblica.it
2. `http://www.repubblica.it/speciale/formulauno/index.html`
   Repubblica.it/Formula Uno
3. `http://www.repubblicarts.repubblica.it/reparts/ita/index.jsp`
   la Repubblica of the ARTS
4. `http://www.repubblicarts.repubblica.it/reparts/ita/cal[...]`
   la Repubblica of the ARTS
5. `http://poll.repubblica.it/cgi-bin/sendpage/tellafriend.pl`
   Repubblica.it
6. `http://www.bologna.repubblica.it/`
   Repubblica edizione di Bologna
7. `http://www.bologna.repubblica.it/archivio/index.htm`
   Repubblica edizione di Bologna
8. `http://www.napoli.repubblica.it/`
   Repubblica edizione di Napoli
9. `http://www.milano.repubblica.it/`
   Repubblica edizione di Milano
10. `http://www.milano.repubblica.it/speciali/colaprico/`
    Repubblica edizione di Milano

Come si può notare, i primi dieci risultati restituiti dal motore sono decisamente più significativi e di sicuro maggior interesse nel caso si utilizzino le ancore.

5.2 Pesi dei ranker nella combinazione lineare

Si è detto in precedenza (vedi introduzione del capitolo 5) che gli algoritmi proposti sono stati aggregati tra loro utilizzando una combinazione lineare. I pesi di tale combinazione lineare sono stati determinati empiricamente sia sulla base degli effetti pratici ottenuti sui risultati a causa di loro variazioni, sia sulla base di alcune considerazioni che verranno qui di seguito esposte. L'ideale sarebbe stato poter disporre di una utenza eterogenea su cui fare dei veri e propri *test*, ma non avendo *beta-tester* a cui far provare il motore i pesi sono stati determinati principalmente sulla base del nostro *feedback*.

Per scegliere i pesi a tavolino ci si è resi conto che i valori di PageRank delle pagine sono molto bassi dal momento che il vettore di PageRank viene normalizzato ad 1 dopo ogni iterazione; gli altri algoritmi invece forniscono punteggi molto più elevati, e pertanto il coefficiente di PageRank nella combinazione lineare dovrà essere molto più alto degli altri, per poter ristabilire l'equilibrio. In seguito ad alcuni esperimenti si è notata che vale la seguente proporzione per determinare il peso x da attribuire a PageRank:

$$1.000.000 : 500 = N : x \tag{5.3}$$

dove N è il numero di pagine indicizzate per il quale si desidera calcolare il peso dell'algoritmo di PageRank.

Per quanto riguarda gli altri algoritmi di *ranking*, si è scelto di dare dei pesi che fossero il più possibile equamente distribuiti tra di loro.

Facendo alcune prove si è trovato, ad esempio, che il valore da attribuire a Proximity può variare tra 0.5 e 1.0 senza sbalzi significativi nei risultati, come ad esempio TitleRank può venire pesato con un coefficiente pari a 1.0

o anche a 2.0 mentre URLRank dovrebbe avere un peso sempre pari ad 1.0: un peso maggiore spingerebbe artificiosamente in alto i documenti che sono ospitati su *host* contenenti nomi comuni; d'altra parte un valore inferiore renderebbe vano l'uso dell'euristica basata sulle parole contenute nelle URL che sta alla base di URLRank.

Discorso a parte meritano invece le ancore, che come si è visto non forniscono un punteggio limitabile superiormente. Proprio per la mancanza di un maggiorante per il punteggio, non si è potuto trovare, come nel caso di PageRank, un fattore che andasse bene in ogni circostanza per normalizzare in qualche modo il punteggio. Se tuttavia si tiene conto del fatto che AnchorRank offre un punteggio (molto) elevato solo a nomi di dominio conosciuti o almeno abbastanza famosi e a pagine importanti, il fatto di alzare — anche di molto — il punteggio complessivo di quei documenti può non essere così spiacevole. Pertanto si è scelto di assegnare un coefficiente pari a 1.0 anche ad AnchorRank.

5.3 Valutazione lazy dei match

Nell'introduzione di questo capitolo si è spiegato che per determinare le pagine che soddisfano una interrogazione bisogna almeno considerare le pagine che l'algoritmo di Proximity restituisce. Tuttavia per certe interrogazioni risulta problematica questa valutazione, in quanto può portare rapidamente ad alzare i tempi di ricerca fino a svariate decine di secondi; ciò accade solitamente con interrogazioni contenenti *stop-word*.

Esempi di interrogazioni particolarmente pesanti per il sistema su un insieme di pagine italiane sono:

`comune di milano` (circa 158 000 pagine)
`la gazzetta dello sport` (circa 111 000 pagine)
`il corriere della sera` (circa 120 000 pagine)

Inoltre vi sono delle interrogazioni che, pur non contenendo al loro interno delle vere e proprie *stop-word*, sono molto pesanti per venir valutate completamente. Ne è un esempio lampante la *query* `milano` che su appena 16 milioni di pagine italiane compare in oltre un milione e duecentomila documenti.

Per affrontare situazioni simili si è pertanto pensato di sviluppare una euristica che permettesse di tagliare questa valutazione dopo poche migliaia (o decine di migliaia) di *match*. Euristiche simili vengono utilizzate anche dai motori di ricerca commerciali.

L'idea dalla quale si è partiti è che non ha senso calcolare tutti i *match* quando invece molte ricerche [16] affermano che gli utenti di un motore di ricerca non vanno a vedere più in là delle prime poche decine di risultati.

Questa considerazione non è affatto errata: nelle pagine successive alla prima decina è solitamente molto difficile trovare ciò che ci interessa.

Da un punto di vista tecnico, si è deciso anzitutto di fissare la nostra attenzione sui primi quattrocento risultati, una cifra ridotta rispetto a quella restituita dai motori di ricerca commerciali ma comunque piuttosto abbondante per l'uso medio che se ne fa. Quel che si voleva fare era predire con una precisione elevata (diciamo al 95%) i primi 400 risultati di una interrogazione senza dover però valutare fino in fondo i risultati. Come si è ampiamente illustrato, nel nostro motore di ricerca sono stati implementati cinque differenti algoritmi di ricerca. I più importanti sono tuttavia PageRank e Proximity perché sono quelli che riescono meglio di ogni altro a restituire i documenti più importanti ai fini di una ricerca; gli altri algoritmi si limitano a perturbare l'ordinamento restituito.

Pertanto si è scelto di valutare i *match* dei documenti in ordine di PageRank degli stessi; per fare ciò è stato necessario introdurre il concetto di permutazione degli indici inversi, un concetto che permette di leggere da disco i numeri di pagina che soddisfano l'interrogazione in maniera che essi siano ordinati in ordine di PageRank decrescente — prima le pagine più importanti, poi le meno importanti.

Una volta fatto questo si è provveduto a creare un strumento automatico che, data una interrogazione, calcolasse il numero di risultati che sarebbe stato necessario "chiedere" agli indici inversi (permutati in ordine di PageRank) affinché si inviduassero i primi 400 risultati con la precisione richiesta. Come insieme di interrogazioni non è stato purtroppo possibile avere a disposizione un insieme eterogeneo di *query* da sottoporre allo strumento automatico; pertanto si è stilata una lista di circa 200 interrogazioni il più possibile

diversificate da sottoporre al motore per stimare la funzione voluta[10].

I risultati trovati ci permettono di concludere che basta valutare poche pagine per stimare ragionevolmente le prime 400 con una precisione del 95%, una precisione molto spesso più che sufficiente. Questo risultato lo si può spiegare pensando al modo in cui vengono create le pagine e alle parole con le quali vengono sottoposte le interrogazioni. Infatti, se consideriamo interrogazioni con parole molto poco diffuse o che mirano ad individuare un argomento molto ben preciso potremmo magari dover analizzare tutte le occorrenze, ma tali occorrenze sarebbero poche per via dell'argomento ristretto e poco trattato[11]. Si è detto che potremmo dover analizzare tutte le occorrenze: questo perché probabilmente non vi saranno punti di riferimento sull'argomento che permettano di scremare pagine con un punteggio di PageRank elevato.

Le interrogazioni vaste invece, cioè quelle che tendono a portare con sé milioni di *match*, sono invece estremamente generali e trattate da moltissime pagine web. In tal caso, pur essendo moltissime le pagine che trattano di quell'argomento, ciò che l'utente tende a voler individuare sono le pagine principali sull'argomento, quelle di riferimento, quelle "ufficiali" o "ufficialmente non ufficiali". Stiamo parlando di interrogazioni contenenti, ad esempio, i nomi di *star* di *Hollywood*, di aziende famosissime, di capi di stato o di nomi di nazioni. In tale ipotesi è proprio PageRank a fare la differenza, e grazie a questo punteggio si può fermare la valutazione ancora prima di quanto accada nel caso illustrato in precedenza: solitamente su interrogazioni molto vaste si è visto che basta analizzare pochissime migliaia di occorrenze per individuare quelle giuste, cioè quelle che l'utente si aspetta di trovare.

[10]Le *query* erano tutte di almeno due parole perché nel caso di interrogazioni monoparola sarebbe bastato analizzare i primi 400 risultati, essendo solamente il PageRank a "contare".

[11]Si pensi, ad esempio, a ciò che accade cercando il nome di una persona che conosciamo ma che non è famosa nel vero senso della parola.

Viene mostrato un grafico delle occorrenze da richiedere in funzione delle occorrenze totali; tale grafico è stato ottenuto tramite la valutazione appena illustrata, e mostra sull'asse delle `Y` il numero di risultati che andrebbero richiesti per una interrogazione che è soddisfatta dal numero di documenti riportato sull'asse delle `X`. Sono state tracciate nello stesso grafico anche altre due curve: la prima rappresenta l'*inviluppo convesso* mentre la seconda rappresenta quella che empiricamente sembra essere la miglior approssimazione di tale inviluppo.

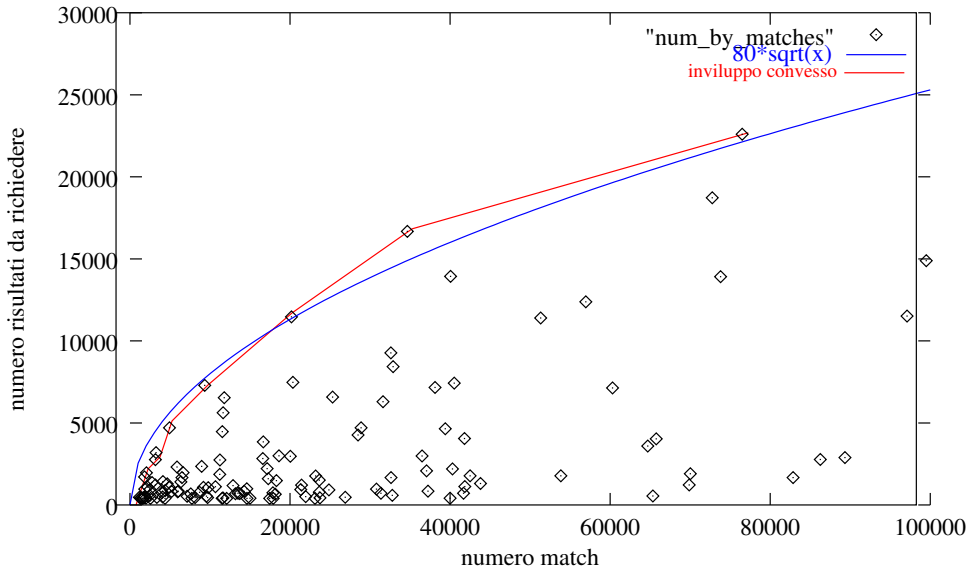

Figura 5.1: La funzione di aggregazione $y = 80 \cdot \sqrt{x}$.

La funzione scelta appartiene alla classe di funzioni della forma:

$$y = A \cdot \sqrt{x} \tag{5.4}$$

L'analisi empirica dei dati relativi allo *store* del web italiano utilizzato ha portato a scegliere $A = 80$.

La valutazione di questa funzione in realtà potrebbe essere estremamente più precisa se si analizzassero un insieme di interrogazioni più vasto, come ad esempio quello che si può ottenere da qualche mese di analisi dei *log* delle interrogazioni di cui un motore di ricerca pubblico può disporre. In particolare, la curva trovata potrebbe anche appartenere in realtà ad altre famiglie di funzioni continue note (come ad esempio la famiglia logaritmica) oppure potrebbe essere molto più complessa. Tuttavia va altresì notato che, sebbene la funzione sia in un certo qual modo molto approssimata essa, empiricamente, sembra ben comportarsi all'interno dell'intera infrastruttura del motore di ricerca Ubi.

Rimane a questo punto solamente da chiarire come si possa determinare il numero approssimativo di riscontri di una data interrogazione, visto che la valutazione viene fermata tipicamente dopo qualche migliaio di iterazioni. Questo dato inoltre è di vitale importanza per valutare la funzione (5.4) in quanto esso rappresenta il valore x.

Per stimare tale quantità si assume che le pagine contengano le parole in maniera più o meno distribuita. Ossia si assume che l'ordinamento di PageRank permuti in maniera sufficientemente casuale le pagine in maniera tale che pagine consecutive secondo tale ordine trattino argomenti differenti. A questo punto, in fase di valutazione si esaminano tutti i *match* fino a che l'indice di un documento tra essi non supera una soglia C prestabilita (nel nostro caso $C = 1000$). Superata tale soglia, si determina il numero x di *occorrenze previste* in modo lineare, secondo l'equazione:

$$x = \frac{m}{C} \cdot N \tag{5.5}$$

dove m è il numero di *match* trovati tra i documenti con indice minore di

c e N è il numero di documenti indicizzati.

5.4 Un raffinamento: il clustering

Nel corso dello sviluppo delle euristiche illustrate si è notato che troppo spesso tra i risultati restituiti vi erano delle lunghe serie di indirizzi appartenenti tutti al medesimo *host*.

Ad esempio, interrogando il motore di ricerca per recuperare le pagine che corrispondono all'interrogazione `parlamento` e senza utilizzare l'euristica del *clustering*, ai primi dieci posti si trovano queste pagine:

---

1. `http://www.parlamento.it/`
   Home page Parlamento Italiano
2. `http://www.parlamento.it/home.htm`
   Home page Parlamento Italiano
3. `http://www.parlamento.it/parlam/leggi/home.htm`
   Indici delle leggi
4. `http://www.parlamento.it/senato.htm`
   Senato della Repubblica — Italia
5. `http://www.parlamento.it/parlam/bicam/home.htm`
   Organismi bicamerali — XIV legislatura
6. `http://www.parlamento.it/parlam/leggi/deleghe/dlattcee.htm`
   Elenco attuativi di direttive comunitarie
7. `http://www.parlamento.it/parlam/leggi/`
   Indici delle leggi
8. `http://www.parlamento.it/parlam/leggi/97059l.htm`
   Legge n. 59 del 1997
9. `http://www.parlamento.it/parlam/leggi/deleghe/99079dl.htm`
   Dlgs 79/99
10. `http://www.parlamento.it/parlam/leggi/98269l.htm`
    Legge n. 269 del 1998

Come si può vedere, tutte le pagine restituite risultano appartenere al medesimo *host*, ossia `parlamento.it`. Tuttavia questo non è il risultato che desidereremmo, in quanto spesso si è più interessati ad ottenere una lista più eterogenea di risultati, comprendendo tra le prime dieci posizioni anche siti non afferenti direttamente al Parlamento Italiano, ma ad esempio siti di terze parti che trattino di leggi e di argomenti simili.

I risultati alla medesima interrogazione ma elaborati utilizzando l'euristica qui proposta risultano essere invece i seguenti:

---

1. `http://www.parlamento.it/`
   Home page Parlamento Italiano
2. `http://www.parlamento.it/home.htm`
   Home page Parlamento Italiano
3. `http://www.europadonna-parlamento.it/`
   Benvenuti su Europa Donna Parlamento
4. `http://www.europarl.it/`
   PARLAMENTO EUROPEO: Ufficio per l'Italia
5. `http://www.gruppi.margheritaonline.it/`
   Margherita in Parlamento
6. `http://www.senato.it/`
   Home page Parlamento Italiano
7. `http://www.senato.it/home.htm`
   Home page Parlamento Italiano
8. `http://www.cnnitalia.it/2002/MONDO/mediooriente/[...]`
   CNNItalia.it — Parlamento iracheno disapprova la risoluzione dell'Onu [...]
9. `http://www.radio.rai.it/grparlamento/`
   GR Parlamento
10. `http://www.governo.it/rapp_parlamento/index.html`
    Governo italiano — Rapporti con il Parlamento

Dal momento che è poco utile ricevere una lista monotematica di siti web si è pensato di sviluppare alcune euristiche atte a ridurre il numero di indirizzi appartenenti al medesimo *host* che compaiono consecutivamente tra i risultati.

Sono state due le euristiche sviluppate nel corso della tesi, la prima mirata a diminuire il numero di risultati dello stesso *host* e la seconda mirata a privilegiare pagine con una URL più breve. Qui di seguito vengono illustrate le due euristiche:

- la prima euristica prevede la scansione della lista degli URL da restituire alla ricerca di URL appartenenti allo stesso *host*; per ottenere questo effetto si mantiene la posizione dell'ultima URL vista per ogni

host incontrato, e nel momento in cui si incontra un'altra URL di un host visto recentemente[12] la si accoda alla prima, indipendentemente dal suo punteggio. In questo modo la lista dei risultati non è più necessariamente ordinata per punteggio composito decrescente, ma i risultati sono molto migliori;

- la seconda euristica che è stata proposta è stata abbandonata al momento dell'introduzione di AnchorRank perché tendeva a peggiorare la qualità dei risultati; ad ogni modo, se utilizzata senza AnchorRank, essa è una buona idea e si basa sul fatto che generalmente URL più brevi sono anche più rilevanti. Pertanto veniva fatto un ordinamento della lista trovata tramite l'euristica al punto precedente ordinando in base alla lunghezza in numero di *slash* contenuti nelle URL e, se uguali da questo punto di vista, in base alla lunghezza in numero di caratteri.

In figura 5.2 viene riportato il codice Java utilizzato per fare *clustering* dei risultati.

[12] Quanto recentemente è descritto da una soglia parametrizzabile.

```
private IntArrayList clusterResults( final int toBeClustered[] ) {
    long startTime, endTime;
    startTime = System.currentTimeMillis();

    ObjectArrayList objectList = new ObjectArrayList();// temporary
    IntArrayList list = new IntArrayList();// where we store results
    int i;

    /* the maximum distance used for clustering: if two URLs from the same host
     * appear in this distance, then they are clustered together
     */
    final int DELTA = 100;

    Int2IntMap host2pos = new Int2IntOpenHashMap();
    host2pos.defaultReturnValue( -1 );
    Int2IntMap lastFound = new Int2IntOpenHashMap();
    lastFound.defaultReturnValue( -1 );

    for ( i = 0; i < toBeClustered.length; i++ ) {
        int host = graph2urlinfo[ toBeClustered[ i ] ];

        if ( host2pos.get( host ) == -1 ) {// host not yet seen
            IntArrayList tmp = new IntArrayList(); tmp.add( toBeClustered[ i ] );
            objectList.add( tmp );

            host2pos.put( host, objectList.size() - 1 );
        }
            else if ( i - lastFound.get( host ) <= DELTA ) {// host already seen recently
                if ( ( (IntArrayList)objectList.get( host2pos.get( host ) ) ).size() == 1 ) {
                    ( (IntArrayList)objectList.get( host2pos.get( host ) ) ).add( toBeClustered[ i ] );
                }
            }
        else {// host already seen, but many results ago (i.e., more than DELTA)
            IntArrayList tmp = new IntArrayList(); tmp.add( toBeClustered[ i ] );
            objectList.add( tmp );

            host2pos.put( host, objectList.size() - 1 );
        }
        lastFound.put( host, i );
    }

    // now we sort each objectList so that "shortest" URLs from each host are shown first
    // ALERT: the heuristic below only works WITHOUT AnchorRank (you will receive bad results otherwise!)
    for ( i = 0; i < objectList.size(); i++ ) {
        IntArrayList tmp = ( IntArrayList ) objectList.get( i );

        jal.INT.Sorting.sort( tmp.elements(), 0, tmp.size(), new jal.INT.BinaryPredicate() {
               public boolean apply( int x, int y ) {
                   if ( numSlashes[ x ] != numSlashes[ y ] ) return numSlashes[ x ] < numSlashes[ y ];
                   else return urlLength[ x ] < urlLength[ y ];

               }
           );
    }
    for ( i = 0; i < objectList.size(); i++ ) list.addAllOf( ( IntArrayList ) objectList.get( i ) );
    return list;
}
```

Figura 5.2: Codice per il *clustering*.

Capitolo 6

Architettura ed implementazione del sistema

In questa sezione si presenterà l'intera architettura che sta dietro al motore di ricerca che si è sviluppato, il cui nome è Ubi, nome derivato dal latino *Ubicumque*, ossia "per ogni dove".

Si tenga presente che non tutto quel che verrà presentato in questo capitolo è stato sviluppato nel corso della tesi: tutta l'infrastruttura di *crawling*, ad esempio, è stata sviluppata in precedenza da [31] mentre gli indici inversi si appoggiano sul progetto MG4J [32], che fornisce una implementazione Java di gran parte delle tecniche di indicizzazione descritte in [33].

6.1 Linguaggio utilizzato e considerazioni varie

L'intero progetto è stato sviluppato nel linguaggio ad oggetti Java e le motivazioni dietro a tale scelta sono essenzialmente dettate dalle richieste insite nello sviluppo del progetto: comodità di avere un linguaggio ad alto livello, orientato agli oggetti, con codice prodotto che può essere eseguito su architetture differenti senza alcun tipo di *porting* e su cui si possa facilmente fare *debugging* in tempi ragionevoli grazie ad un substrato di macchina virtuale che permetta di rilevare agevolmente gli errori, disponendo di uno stack di esecuzione facilmente accessibile e descritto in una forma intelligibile. Inoltre un'ulteriore richiesta era quella del *rapid prototyping*, ossia della possibilità di scrivere rapidamente prototipi di codice basandosi sull'abbondanza delle librerie disponibili: strutture dati, *multi-threading*, RMI e rete.

Per alcune fasi dell'elaborazione dei dati sono già presenti componenti che possono essere eseguite in parallelo su più macchine. Per rendere semplice l'interazione e la comunicazione tra le varie componenti e per non dover gestire esplicitamente *multi-threading* su parti di codice si è scelto di sviluppare tali componenti tramite l'utilizzo di RMI, ossia della *Remote-Method Invocation*, una funzionalità del linguaggio Java che permette di richiamare metodi remoti e di eseguirli in maniera trasparente, come se fossero noti alla macchina virtuale su cui gira il chiamante.

Il progetto può al momento contare su varie macchine: otto *personal computer* in *rack* dislocati all'Istituto di Informatica e Telematica del Consiglio Nazionale delle Ricerche (presso l'Area di Ricerca di Pisa) [34] utilizzati per la fase di *crawling*, tre *personal computer end-user* e un server SUN di grosse dimensioni dislocati al Dipartimento di Scienze dell'Informazione dell'Università degli Studi di Milano [35]. Quest'ultimo server, essendo un quadri-

processore dotato di quasi un terabyte di dischi e di 16 gigabyte di memoria RAM è la macchina su cui vengono elaborati e mantenuti tutti gli indici di cui si discuterà più avanti.

Per avere degli standard con cui confrontarsi, durante tutto il corso dello sviluppo della parte di analisi si è guardato al motore di ricerca in assoluto più famoso e più preciso, Google. Sebbene la tecnologia dietro a questo motore — e a tutti gli altri — sia mantenuta segreta per ovvi motivi commerciali, si è tentato più volte di fare del *reverse-engineering* tramite una attenta analisi delle risposte alle interrogazioni.

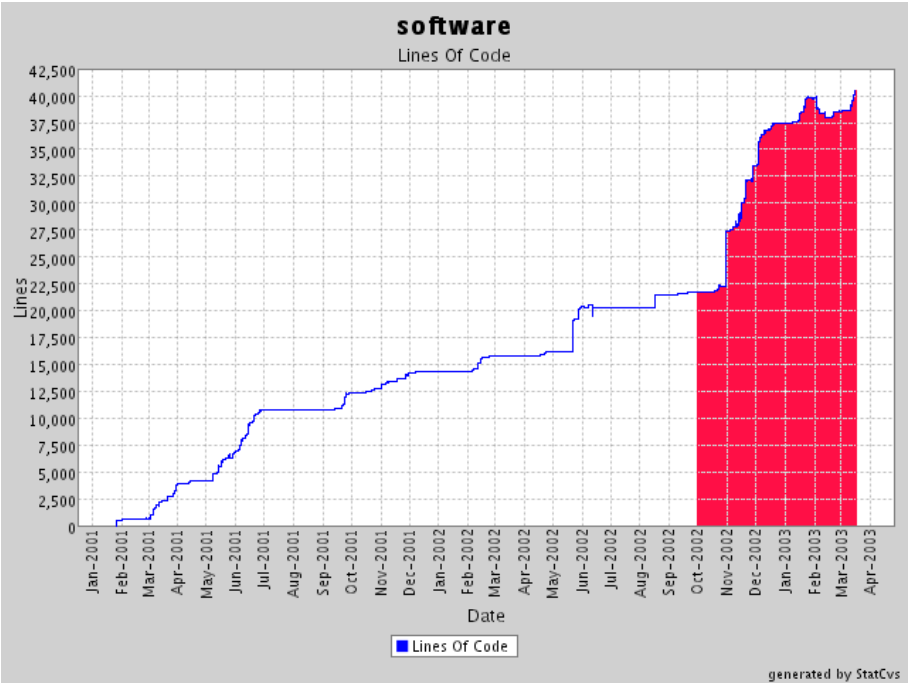

Figura 6.1: L'evoluzione del codice prodotto, con evidenziato il periodo di tesi.

Le classi di cui si compone il progetto nel momento della stesura di questa tesi sono quasi 180, per un totale di oltre 40000 linee complessive di codice. In figura 6.1 viene riportato l'andamento nel tempo delle linee di codice

dell'intero progetto, e viene evidenziato il periodo di questa tesi. Le metriche calcolate dal programma "statcvs" riportano un totale di oltre 25000 linee di codice scritte o modificate per questa tesi.

In figura 6.2 viene visualizzato il flusso dei dati soggiacente alla indicizzazione di uno *store*. I rettangoli rappresentano gli strumenti utilizzati per l'indicizzazione delle pagine e del testo in esse contenuto, mentre gli ovali rappresentano i file prodotti (e successivamente utilizzati) da questi strumenti[1].

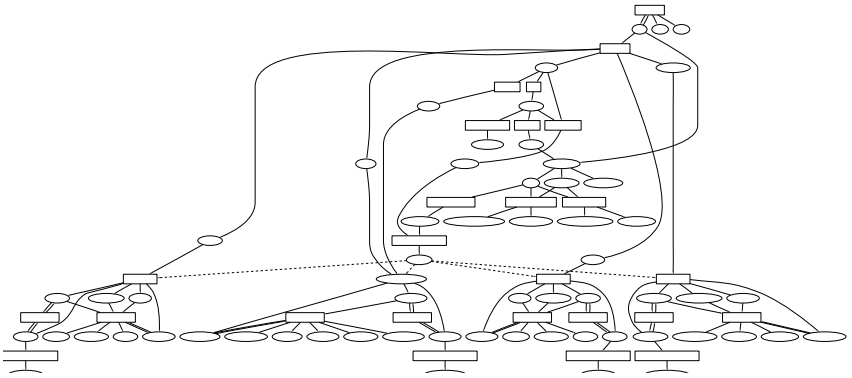

Figura 6.2: Flusso dei dati durante un'indicizzazione.

[1]Per ragioni tipografiche non vengono riportati i nomi degli strumenti utilizzati e dei file prodotti nelle apposite caselle.

6.2 Tecniche e dettagli implementativi

Si vuole qui fare un breve accenno ad alcune delle tecniche utilizzate per scrivere il codice del progetto e, in particolare, per poter gestire quantità di dati voluminose come quelle che si sono trattate. Il problema forse più difficile di fronte al quale ci si è scontrati è che purtroppo nel linguaggio Java non esiste una chiamata per deallocare la memoria esplicitamente in stile `free()`, come invece accade nel linguaggio C/C++; tutto viene demandato alla *garbage collector*, ossia ad un *thread* lanciato periodicamente ed in maniera automatica dalla Java Virtual Machine (JVM) che si occupa di reclamare la memoria non più in uso dal processo su cui viene invocato e libera lo spazio utilizzato dagli oggetti non più in uso. Sviluppando in Java ci si è accorti poi di quanto fossero pesanti tali chiamate nel momento in cui si aveva qualche milione di oggetti caricati in memoria (ad esempio quando si costruiva o si gestiva il grafo): le chiamate al *garbage collector* erano, in questi casi, estremamente dispendiose in tempo, fino a superare i dieci minuti[2]. Pur accadendo in fasi di precomputazione, tali tempi sono ben presto diventati inaccettabili e si sono dovute trovare soluzioni più complicate per la simulazione degli oggetti, come, ad esempio, l'allocazione di un unico vettore atto a contenere il contenuto dei milioni di oggetti che venivano utilizzati precedentemente.

Altri problemi che si sono riscontrati sono quelli dovuti al fatto che gli oggetti e le strutture dati fornite dal linguaggio Java sono per la maggior parte sincronizzate per il *multi-threading* oppure non propriamente pensate per gestire tipi primitivi[3]. Per tali ragioni ci si è dovuti alle volte creare delle strutture dati ausiliarie, spesso mere copie non sincronizzate o leggermente riadattate di quelle fornite con il linguaggio Java, oppure quando si è dovuto

[2]Ripetuti qualche volta nel corso di varie ore di calcolo.
[3]Anzi, normalmente sono pensate per gestire oggetti.

ricorrere a librerie esterne, come le COLT del CERN di Ginevra [36].

In particolare, il massiccio impiego di strutture dati per il mantenimento di insiemi e funzioni ha richiesto l'utilizzo di strutture specifiche per il trattamento dei tipi primitivi, come le liste di COLT e gli insiemi e funzioni di fastutil [37] (essendo i dati primitivi non soggetti a *garbage collection*). Ad esempio, nella lettura delle liste di adiacenza dei grafi si è scelto di restituire un iteratore di tipo `IntIterator`, dopo aver opportunamente scompresso tale lista.

Per ottenere le prestazioni migliori possibili si è spesso ricorsi ad opzioni poco documentate della JVM, come, ad esempio, quelle per cambiare il tipo di *garbage collector* (`-Xincgc`, `-Xconcgc`); una opzione che è stata molto utilizzata per andare oltre i 32 bit di indirizzamento è stata l'opzione `-d64`. Altre opzioni (`-XX:+UseMPSS`, `-XX:+AggressiveHeap`) sono state accompagnate da un *tuning* della macchina su cui veniva eseguito il codice, dovendo esse sfruttare al meglio il sistema operativo sottostante (Solaris, della stessa SUN).

Nel corso della tesi siamo anche incappati in qualehe baco della JVM, tutti riportati a SUN. In particolare è interessante notare che il primo di essi (bug-id: 4 797 189) era una *memory-leak* di cui ci si è accorti probabilmente per primi solo perché il nostro codice eseguiva milioni di iterazioni, e la perdita di anche soltanto poche decine byte di memoria ad ogni iterazione può farsi sentire a lungo andare.

6.3 Fasi dell'indicizzazione

Vengono qui di seguito presentate tutte le fasi necessarie al recupero dei dati, alla loro elaborazione fino ad arrivare all'ultima fase, l'utilizzo di tali dati da parte dell'utente finale.

6.3.1 Prima fase: il crawling

La prima fase dell'intero processo che porta ad avere un motore di ricerca funzionante è il *crawling*, ossia il recupero delle pagine web dalla rete.

Come già detto, i dati utilizzati in questa tesi sono stati ottenuti tramite UbiCrawler [31], un sistema ad agenti che si coordinano in maniera distribuita ed autonoma.

Ogni agente, che potenzialmente può girare su un *host* differente, inizia a recuperare le pagine web partendo da un *seme*, ossia da un insieme di URL inserite manualmente. L'agente visita tramite una ricerca mista in ampiezza ed in profondità del grafo del web queste pagine e le recupera, salvandole in formato compresso in un file su disco.

In tale file sono contenuti vari *record*, uno per pagina visitata. Ciascun record contiene l'indirizzo della pagina in questione, gli *header* restituiti dal web server quando si è recuperata la pagina, la data di visita, la pagina compressa più altre informazioni facilmente recuperabili anche a posteriori ma che è utile, per vari scopi, avere già pronte — questo ovviamente a scapito dello spazio su disco.

In particolare ogni agente mentre recupera le pagine fa anche il *parsing* delle stesse per cercare nuove URL da recuperare e visitare; tali URL, se nuove al sistema (ossia se nessun altro agente le ha già incontrate) vengono inserite

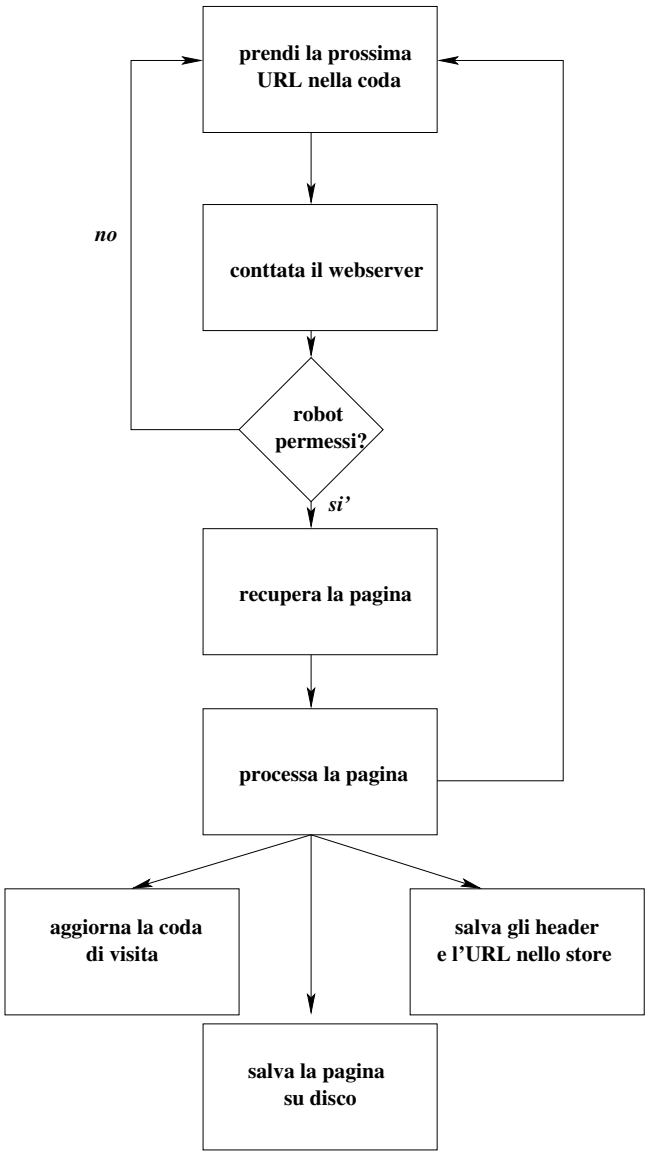

Figura 6.3: Un diagramma di flusso semplificato che illustra la fase di *crawling*.

in una lista che sarà la coda di visita. Il procedimento continua fintantoché non termina o, come avviene nel nostro caso, non viene interrotto perché si è raggiunto, ad esempio, il massimo spazio su disco.

Si noti che in questa fase si potrebbero anche già fare altre analisi che invece vengono fatte a posteriori: ad esempio, ogni agente, mentre analizza la pagina recuperata per cercare nuove URL, potrebbe estrarre il testo in esse contenuto, oppure il testo contenuto nei collegamenti (ossia il testo contenuto nell'elemento `A` dell'HTML) — tutte operazioni che vengono al momento svolte a posteriori. Il motivo per cui ciò non viene ancora fatto è che non solo è logicamente più corretto suddividere le varie fasi — anche per banali ragioni di *debugging* — ma anche che lo spazio disco sulle macchine utilizzate per il recupero delle pagine è abbastanza limitato.

Tipicamente, le collezioni di pagine così ottenute (chiamate anche *store*) vengono unite insieme con il comando UNIX `cat` dopo essere state spostate sul server SUN sopra menzionato. Il file risultante tipicamente supera di gran lunga il centinaio di gigabyte se il numero di pagine recuperate è di circa quindici milioni.

Un aspetto interessante che vale la pena di osservare è che la gestione di enormi quantità di dati evidenzia quanto l'architettura hardware e software dei comuni *personal computer* sia poco affidabile; nel corso degli esperimenti, e nella manipolazione dei dati, abbiamo spesso verificato problemi di varia natura, sia nell'hardware che nel software. Infatti non una sola volta gli *store* sono stati recuperati senza errori e senza problemi inspiegabili che inseriscono un bit errato una volta ogni qualche centinaio di miliardi, causando problemi per l'elaborazione successiva. Probabilmente per gli usi che si fanno tutti i giorni i personal computer non hanno bisogno di controlli di parità sulle memorie, *array* ridondanti di dischi, ecc. ma per gestire quantità di dati così

elevate tali controlli risultano vitali.

6.3.2 Seconda fase: l'estrapolazione dei dati

Una volta che si hanno questi enormi *store* a disposizione, la prima cosa da fare è estrarre da essi il maggior numero di dati possibili per poterli poi indicizzare. Molti degli strumenti che si andranno ad elencare qui di seguito sono stati sviluppati nel corso della tesi.

Anzitutto si estraggono gli indirizzi delle pagine web recuperate e li si ordina alfabeticamente. Il motivo di tale scelta è stato spiegato nel paragrafo 3.1.3.

Una volta estratte tali URL si calcola il *codice di ridondanza ciclica* (CRC) a 64 bit di ciascuna di esse; il CRC viene usato per poter occupare meno spazio in memoria nel momento in cui esse andranno caricate. L'idea del CRC è molto utilizzata in letteratura, ma uno dei problemi in cui si è incappati è il cosiddetto *paradosso del compleanno*. Infatti nel nostro caso l'insieme su cui vengono calcolati i CRC spesso è un insieme molto grande di elementi, e la probabilità che due di essi abbiano il medesimo CRC diventa molto alta piuttosto in fretta all'aumentare delle dimensioni dell'insieme base. Questo problema si è in particolare verificato tentando di mantenere in memoria tramite i CRC le parole contenute nelle pagine: questo approccio, pur funzionando con piccole quantità di dati, è diventato immediatamente ingestibile non appena si sono recuperati qualche milione di pagine. Al momento l'approccio a CRC viene ancora utilizzato per le URL poiché, essendo esse in numero molto minore, il paradosso del compleanno ha una esigua probabilità di verificarsi.

Si noti che a questo punto, avendo ordinato le URL, non c'è più una cor-

rispondenza tra l'ordine di visita — che è anche l'ordine in cui tali indirizzi sono salvati nello *store* — ed il nuovo ordine, che chiameremo l'ordinamento del grafo. Dal momento che è importante poter tornare indietro nell'associazione si rende pertanto necessaria una mappa da interi ad interi per poter ricostruire l'ordinamento originale.

Vi sono numerosi altri dati da estrarre dallo *store*: non si è infatti ancora fatta menzione di tutto il testo delle pagine — testo che ovviamente vorremmo in qualche modo tenere. A causa degli algoritmi implementati nel motore di ricerca, servono quattro tipi di dati testuali:

- il testo delle pagine, punteggiatura esclusa, con ogni parola nell'ordine in cui essa compare;
- il testo dei titoli delle pagine, ossia le parole — sempre punteggiatura esclusa — racchiuse all'interno dell'elemento `TITLE` dell'HTML;
- le parole contenute nelle URL incontrate: ad esempio, nel caso dell'URL `http://www.comunemilano.it/` vorremmo individuare al suo interno le parole `comune` e `milano`;
- le parole con cui una pagina viene puntata dalle altre, ossia i testi delle ancore con cui una pagina è puntata.

Si consideri una sequenza massimale di lettere e numeri che si trova in una pagina web. Sia ora una *parola* una sottosequenza delle precedenti e che inoltre rispetti regole più stringenti sulla sua lunghezza massima, sul numero di cifre contenute, ecc.[4]. Sono proprio le parole ad essere utilizzate in tutte le fasi dell'elaborazione, in quanto le sequenza massimali di lettere e numeri[5]

[4]Non viene qui fornita la regola esatta con la quale vengono ricavati i termini.
[5]Ossia quelle che comunemente sono riferite come parole.

nelle pagine web sono troppe. Recuperando infatti pochi milioni di pagine e considerando una parola come una sequenza di lettere e numeri separata da simboli di punteggiatura (spazi inclusi), le sequenze massimali che vengono estratte dalle pagine web tendono a diventare un numero ben presto ingestibile. In questo spiacevole fatto entrano in gioco non solo le molteplici lingue in cui tali pagine sono scritte, ma anche gli errori di punteggiatura, i numeri di telefono, i numeri di codice dei prodotti, ecc. Non potendo in alcun modo gestire per 10 milioni di pagine recuperate 20 milioni di sequenze massimali, si sono studiate alcune euristiche per ridurne il numero, come ad esempio quella di spezzare le parole più lunghe di una certa lunghezza o che contengono troppi numeri. Si noti che una tale euristica non pregiudica nella maniera più assoluta la possibilità di trovare, ad esempio, una sigla farmaceutica se essa è stata spezzata: infatti, se pervenisse al sistema una interrogazione contenente, ad esempio, la sigla farmaceutica in questione, il sistema provvederebbe a spezzare in vari termini la parola, e a ricercare ciascuno di essi consecutivamente. Il grosso vantaggio di questo modo di procedere, essendo la lunghezza massima di una sequenza massimale in qualche modo ridotta da un maggiorante ben stabilito, è che il numero di parole estratte è forzatamente limitato.

6.3.3 Terza fase: costruire il grafo

Una volta effettuata l'estrazione dei dati è tempo di costruire il grafo sul sottoinsieme di pagine che sono state recuperate. Il grafo, come visto nei capitoli precedenti, è fondamentale per poter eseguire gli algoritmi presentati nel capitolo 3. Il grafo che verrà prodotto[6] avrà come nodi le pagine web

[6]In realtà in forma compressa, come visto nel paragrafo 3.1.3.

recuperate, mentre come archi orientati i link che collegano una pagina alle altre. A seconda del numero di pagine recuperate, tale grafo potrà essere sparso oppure denso. Studi [38] dimostrano che una visita in ampiezza porta in fretta a recuperare comunque pagine di qualità, e quindi potenzialmente molto puntate anche da una ristretta porzione del web; basandosi su tale risultato anche *crawl* interrotte prematuramente porteranno alla produzione di grafi significativi. Durante la costruzione del grafo si eliminano i link che puntano a pagine esterne alla nostra collezione o non esistenti. Vengono invece mantenute le pagine con grado uscente nullo, delegando pertanto ad algoritmi quali PageRank il compito di gestire opportunamente queste situazioni, peraltro piuttosto comuni (per esempio tutti i documenti di puro testo rientrano in questa categoria). La costruzione del grafo produce anche una mappa ulteriore, che va dai numeri di nodo (ossia il numero progressivo che viene assegnato durante l'ordinamento delle URL dello *store*) alle posizioni delle pagine corrispondenti nello *store* su disco.

Creato il grafo, il passo successivo è di eseguire gli algoritmi che richiedono il grafo per funzionare.

6.3.4 Quarta fase: gli indici inversi

Dai dati testuali estrapolati nella seconda fase è necessario costruire gli indici inversi, ossia degli indici che, per ogni parola, forniscano i documenti (e le posizioni) nelle quali essa compare. Tali indici vengono utilizzati per la ricerca testuale quando viene fatta un'interrogazione al motore di ricerca, e sono uno strumento molto diffuso usato per rispondere in maniera efficiente e rapida a qualunque interrogazione. Per la loro creazione e il loro utilizzo come già detto ci si è avvalsi del progetto MG4J [32]. La creazione di tali indici non

è al momento distribuita, e pertanto questa fase risulta essere relativamente laboriosa e lenta; non essendo tuttavia il tempo un requisito al momento stringente, la parallelizzazione del calcolo non ci è sembrata essenziale ed è stata rimandata a futuri sviluppi dell'intera architettura.

Si noti che gli indici inversi vengono creati permutando gli indici dei documenti in ordine del punteggio di PageRank (ordinamento decrescente). Il motivo di questa scelta verrà spiegato nella sezione dedicata all'aggregazione (vedi capitolo 5) ma si fa notare qui che tale permutazione richiede un'ulteriore mappa che indica in quale ordine rispetto a PageRank si trova ciascuna pagina dello *store*.

6.3.5 Quinta fase: gli agenti e la servlet

Una volta che tutti gli indici sono stati creati e che tutti gli algoritmi precedentemente eseguiti hanno prodotto i loro output è possibile eseguire gli agenti preposti a rispondere alle interrogazioni ed è possibile anche usufruire dell'interfaccia web creata come *front-end*.

Gli agenti comunicano con l'interfaccia *front-end*, anch'essa scritta in Java, tramite RMI e pertanto tutta l'infrastruttura è *multi-threaded* e risulta quindi possibile già fin d'ora rispondere a più richieste contemporaneamente.

Gli agenti sono al momento due, e possono essere eseguiti su macchine differenti, a patto che ovviamente su entrambe vi siano i file necessari:

- l'agente che risponde alle richieste indicando i punteggi delle pagine e i numeri di pagina che soddisfano la ricerca;
- l'agente che, dato un numero di pagina, estrae la pagina dallo *store*, ne produce il sommario in relazione all'interrogazione (evidenziando nella

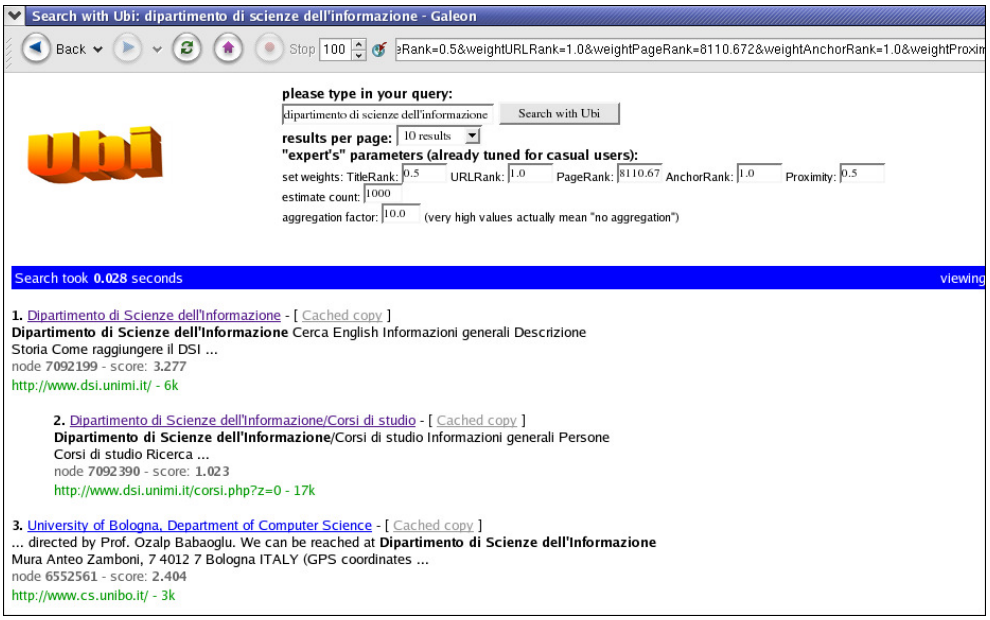

Figura 6.4: L'output della *servlet* all'interno di un *browser*.

pagina l'intervallo che soddisfa l'interrogazione) e permette di accedere alla copia della pagina che è stata recuperata a tempo di *crawl* dalla rete.

6.4 Il processo di vita di una query

In questo paragrafo verrà esaminato l'intero processo che parte dall'inserimento di una interrogazione nel motore di ricerca fino alla relativa risposta che viene fornita all'utente.

Quando un utente apre la pagina principale di Ubi in realtà fa eseguire dal *webserver* una *servlet* Java che si occupa di produrre il codice HTML che comprende il modulo per inserire le interrogazioni. Una volta introdotta l'interrogazione e sottoposta al *webserver*, la *servlet* si occupa semplicemente di spedirla al primo degli agenti descritti nella sezione 6.3.5, ossia all'agente in grado di restituire le pagine che soddisfano l'interrogazione in un ordine che rispecchi il più possibile le aspettative dell'utente. Come detto precedentemente, la servlet comunica con l'agente tramite una chiamata RMI, che secondo alcune misurazioni compiute durante la tesi non inficia particolarmente le prestazioni ma porta notevoli benefici in termini di semplicità di progetto e di facilità di *debugging*. Eventualmente, in fase di produzione, tale protocollo potrebbe essere sostituito a favore di protocollo di comunicazione più leggero.

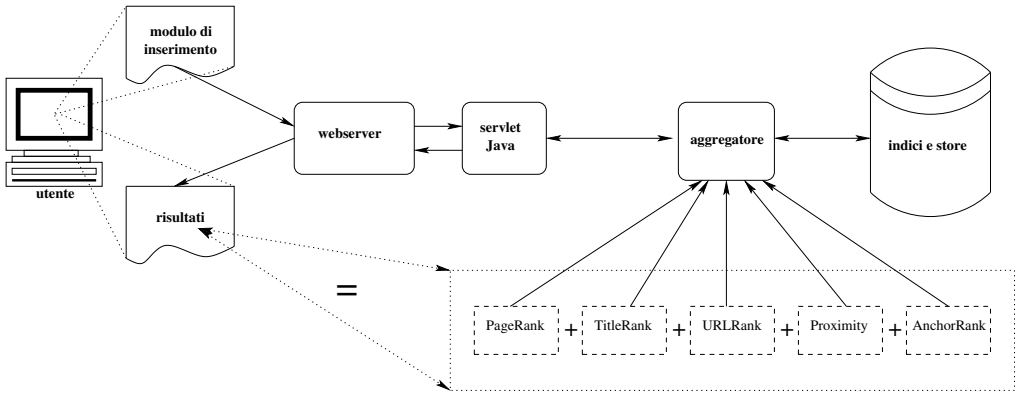

Figura 6.5: Il processo di vita di una *query*.

Una volta giunta all'agente, l'interrogazione viene inviata in serie a vari *ranker*, ossia a vari algoritmi che sono in grado di assegnare un punteggio ad ogni pagina dell'intera collezione. In particolare il primo *ranker* ad essere chiamato in gioco è l'algoritmo di Proximity, che, come visto in un capitolo precedente (vedi paragrafo 4.2), seleziona le pagine che soddisfano l'interrogazione e assegna loro un punteggio compreso tra 0.0 e 1.0. Successivamente, un algoritmo basato sulle ancore e che è stato discusso nel paragrafo 5.1.4, permette di trovare ulteriori pagine che soddisfano l'interrogazione, e l'insieme di pagine da valutare e da restituire viene così ampliato. Assieme alle nuove pagine compare anche una indicazione sul nuovo punteggio di questo algoritmo.

I successivi algoritmi con i quali vengono raffinati i risultati sono però diversi dai due appena presentati in quanto non portano nuove pagine all'interno dell'insieme di risultati, ma bensì si limitano a dare un punteggio alle pagine già trovate. Si noti che la prima scrematura operata da questi due *ranker* è di fondamentale importanza perché permette di limitare la valutazione a poche migliaia o decine di migliaia di pagine web contro vari milioni o miliardi disponibili, mentre un algoritmo quale PageRank da solo non potrebbe far altro che restituire tutte le pagine recuperate in fase di *crawl*, ciascuna con il suo punteggio.

Gli algoritmi chiamati successivamente in causa sull'insieme di pagine così determinato sono:

- PageRank, vedi capitolo 3.2;
- TitleRank, vedi paragrafo 5.1.2;
- URLRank, vedi paragrafo 5.1.3.

Una volta ottenuti tutti questi risultati, essi vengono tra loro combinati tramite una combinazione lineare con pesi forniti a priori od eventualmente scelti dall'utente che ha inserito la *query*. In questo modo, una volta ordinati i risultati in base a questo punteggio composito così calcolato, si ha già una lista di risultati abbastanza corretti da poter restituire all'utente.

Tuttavia una cosa che si è notata è che è necessario *clusterizzare* questo insieme di risultati per *host* di appartenenza per prevenire la spiacevole situazione nella quale una serie molto lunga di risultati siano tutte pagine dello stesso sito web. Per evitare ciò si sono ideate alcune euristiche che verranno presentate successivamente (vedi paragrafo 5.4) e che permettono di restituire pochi risultati consecutivi per ogni sito web, offrendo così una lista che spazi il più possibile sulle pagine realmente rilevanti. Per elaborare questa euristica ci si è ispirati a quello che accade con i motori di ricerca commerciali — Google *in primis* — che mostrano soltanto poche pagine per *host* consecutivamente.

A questo punto i risultati così elaborati vengono restituiti, sempre via RMI, alla *servlet* che estrae la parte che interessa all'utente (ossia la parte che corrisponde alle pagina di risultati che sta visualizzando) e si occupa di invocare il secondo agente, quello che permette di risolvere i numeri di documento nelle pagine vere e proprie e nei relativi sommari.

Il secondo agente viene invocato pertanto sempre via RMI su ogni pagina da risolvere, passandogli un numero di documento ed un intervallo da evidenziare nel sommario. Tale agente si occupa di posizionarsi nello *store* in corrispondenza della pagina che interessa e di estrarne il contenuto, creando poi il sommario basato sull'intervallo di parole richiesto. Viene quindi restituito un oggetto che rappresenta un singolo risultato dell'interrogazione fatta dall'utente e che la *servlet* si occuperà di visualizzare all'interno del *browser*

dell'utente.

Esiste poi la possibilità di recuperare le pagine che sono state salvate nello *store* per poter vedere i contenuti che sono stati restituiti al *crawler*. Questa possibilità è data da un ulteriore agente che, se invocato con un numero di pagina, si occupa di recuperare e restituire al chiamante il testo HTML della pagina così come essa appare nello *store*.

Capitolo 7

Conclusioni e sviluppi futuri

Il campo dell'*information retrieval* su web è oggi in pieno fermento. Quel che si è cercato di mostrare con la stesura di questa tesi è che non esiste un solo modo per ricercare le informazioni che interessano all'interno di quella enorme banca dati che è il web, ma che al contrario esistono una miriade di euristiche più o meno complicate che si possono adottare per migliorare la qualità delle ricerche effettuate dal proprio motore.

Al momento della stesura di questa tesi, la componente del grafo del web raggiungibile e conosciuta dai *robot* dei vari motori di ricerca commerciali supera i tre miliardi di pagine, ma come detto tale quantità è destinata ad aumentare sempre più. Essendo il materiale indicizzabile in crescita esponenziale, l'unico modo per poter sperare di riuscire a mantenere il passo di questa crescita è quello di continuare a studiare euristiche che limitino il fenomeno dello *spam*, che trovino tra le prime posizioni i documenti di interesse e che riescano ad indicizzare quantità di dati sempre più voluminose.

Il software che è stato sviluppato nel corso della tesi ha permesso di indicizzare notevoli quantità di dati e di recuperare all'interno di questo

consistente numero di pagine e a fronte di una interrogazione dell'utente, le pagine più significative. Su questa strada si può ancora procedere rendendo distribuite tutte le procedure in maniera che possano più agilmente scalare quando il numero di pagine raggiunga dimensioni realistiche per un motore commerciale. Altre tecnologie da esplorare sono quelle rivolte all'eliminazione dello *spam*, alla ricerca e alla catalogazione di immagini, video, suoni e notizie di giornale.

Elenco delle figure

Indice analitico

Bibliografia

[1] Google, inc.
`http://www.google.com/`.

[2] D. L. Isaacson and R. W. Madsen. *Markov Chains Theory and Applications*. John Wiley & Sons, New York, 1976.

[3] Sriram Raghavan and Hector Garcia-Molina. Representing web graphs. Technical report, 2002.

[4] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford Digital Library Technologies Project, 1998.

[5] E. Garfield. Citation analysis as a tool in journal evaluation. pages 178:471–479, 1972.

[6] G. Pinski and F. Narin. Citation influence for journal aggregates of scientific publications: Theory with application to literature of physics. pages 12:297–312, 1976.

[7] Yahoo! Inc.
`http://www.yahoo.com/`.

[8] L. Egghe and R. Rousseau. Introduction to infometrics: Quantitative methods in library, documentation and information science. 1990.

[9] T. Haveliwala. Topic-sensitive pagerank, 2002.

[10] Special Google Searches.
http://www.google.com/options/specialsearches.html.

[11] Taher Haveliwala. Efficient computation of pageRank. Technical Report 1999-31, 1999.
http://dbpubs.stanford.edu:8090/pub/1999-31.

[12] Gopal Pandurangan, Prabhakara Raghavan, and Eli Upfal. Using PageRank to Characterize Web Structure. In *8th Annual International Computing and Combinatorics Conference (COCOON)*, 2002.

[13] Kumar, Raghavan, Rajagopalan, Sivakumar, Tomkins, and Upfal. Stochastic models for the web graph. In *FOCS: IEEE Symposium on Foundations of Computer Science (FOCS)*, 2000.

[14] Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, and Andrew Tomkins. Trawling the Web for emerging cyber-communities. *Computer Networks (Amsterdam, Netherlands: 1999)*, 31(11–16):1481–1493, 1999.

[15] Open Directory Project — ODP.
http://www.dmoz.org/.

[16] Craig Silverstein, Monika Henzinger, Hannes Marais, and Michael Moricz. Analysis of a very large altavista query log. Technical Report 1998-014, Digital SRC, 1998.
http://gatekeeper.dec.com/pub/DEC/SRC/technical-notes/abstracts/src-tn-1

[17] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5):604–632, 1999.

[18] Krishna Bharat and Andrei Broder. Estimating the relative size and overlap of public web search engines. April 1998. `http://research.compaq.com/SRC/whatsnew/sem.html`.

[19] NetCraft Web Server Survey. `http://www.netcraft.com/survey/`.

[20] Scott C. Deerwester, Susan T. Dumais, Thomas K. Landauer, George W. Furnas, and Richard A. Harshman. Indexing by latent semantic analysis. *Journal of the American Society of Information Science*, 41(6):391–407, 1990.

[21] T. Landauer and S. Dumais. A solution to plato's problem: The latent semantic analysis theory of acquisition, induction and representation of knowledge. pages 104(2), 211–240.

[22] P. Boldi. An interval semantics for web-search query operators with efficient lazy evaluation. 2002.

[23] Google Zeitgeist. `http://www.google.com/press/zeitgeist.html`.

[24] Altavista Company. `http://www.altavista.com/`.

[25] Search Engine Forums. `http://searchengineforums.com/`.

[26] Jon Bentley and Bob Sedgewick. Ternary Search Trees, April 1998.

[27] Richard W. Wiggins. The effects of September 11 on the leading search engine. (volume 7, number 10, October 2001), 18 September 2001.
http://www.firstmonday.dk/issues/issue6_10/wiggins/.

[28] Krishna Bharat and George A. Mihaila. When experts agree: using non-affiliated experts to rank popular topics. In *World Wide Web*, pages 597–602, 2001.
http://citeseer.nj.nec.com/bharat01when.html.

[29] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.

[30] Sergey Brin, Rajeev Motwani, Lawrence Page, and Terry Winograd. What can you do with a web in your pocket? *Data Engineering Bulletin*, 21(2):37–47, 1998.

[31] Paolo Boldi, Bruno Codenotti, Massimo Santini, , and Sebastiano Vigna. Ubicrawler: Scalability and fault-tolerance issues. In *WWW Posters*, 2002.

[32] Sebastiano Vigna and Paolo Boldi. Managing Gigabytes for Java — MG4J.
http://mg4j.dsi.unimi.it/.

[33] Ian H. Witten, Alistair Moffat, , and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishing, San Francisco, 1999.
http://www.cs.mu.oz.au/mg/.

[34] UbiCrawler project.
http://ubi.imc.pi.cnr.it/projects/ubicrawler/.

[35] Universitï¿½ degli Studi di Milano — Dipartimento di Scienze dell'Informazione.
http://www.dsi.unimi.it/.

[36] Colt — open source libraries for high performance scientific and technical computing in Java.
http://hoschek.home.cern.ch/hoschek/colt/.

[37] Sebastiano Vigna. fastutil.
http://fastutil.dsi.unimi.it/.

[38] Marc Najork and Janet L. Wiener. Breadth-First Crawling Yields High-Quality Pages. In *Proceedings of the 10th International World Wide Web Conference*, pages 114–118, Hong Kong, May 2001. Elsevier Science.