Luca Baronti

Appunti del Corso di Algoritmica 2

Anno: 2009/2010

Indice

Disclaimer

Questo documento costituisce la trascrizione degli appunti del corso di Algoritmica 2 della Laurea Magistrale dell'Università di Pisa, redatti dallo studente Luca Baronti nell'anno accademico 2009/2010. Sebbene siano stati trascritti con l'intenzione di essere quanto più completi e precisi possibile è bene precisare che **non** sono stati revisionati da alcun docente, è quindi possibile che ci siano parti inesatte o incomplete. Il documento dovrebbe coprire tutto il programma del corso dell'anno accademico specificato, con alcune limitate eccezioni (come la parte relativa al list ranking).

Il presente documento è stato rilasciato sul forum non ufficiale degli studenti di informatica di Pisa www.informateci.org con l'intento di essere un aiuto per gli studenti che devono sostenere l'esame. Chiunque volesse integrarlo con parti agiuntive o revisioni è invitato a contattare l'autore (Silyus) sul suddetto forum.

Parte I

Problemi difficili e loro soluzione

In informatica esistono diversi metodi per classificare i problemi, il più usato è certamente la classificazione mediante la complessità asintotica della soluzione trovata. In base alle caratteristiche del problema è possibile dare stime di massimo o di minimo che circoscrivono la complessità computazionale, o l'occupazione di spazio, dell'algoritmo che lo risolve, mediante i seguenti criteri:

$O(f(n))$ Limite superiore che indica la complessità massima *dell'algoritmo*

$\Omega(f(n))$ Limite inferiore che fornisce una stima di complessità minima che *il problema* possiede

$\Theta(f(n))$ Indica che da un certo valore di input in poi, la complessità tenderà a stabilizzarsi su un certo andamento

Le classificazioni asintotiche forniscono un buon metodo per classificare soluzioni a problemi, e per poterle confrontare tra loro. Similmente è possibile classificare i problemi stessi in base alla complessità asintotica dei loro migliori algoritmi risolutori [1]. Dato un problema A, esso appartiene alla classe:

P se è decisionale [2] ed è risolubile in tempo *deterministicamente polinomiale*

NP se è decisionale ed è risolubile in tempo *non deterministicamente polinomiale*

NP-Hard se è solo se *Soddisfacibilità* $\propto A$, ovvero se A è difficile almeno quanto il problema della Soddisfacibilità (Vedi Cap. 2.1)

NP-Completo se è decisionale e se $A \subset NP$-Hard e $A \subset NP$, ovvero se è difficile come la Soddisfacibilità, ma è risolubile in tempo non deterministicamente polinomiale

Naturalmente è possibile notare che $P \subseteq NP$ [3]; inoltre la classe P, che teoricamente può includere problemi le cui soluzioni presentano anche gradi di complessità importanti ($O(n^{10})$, $O(n^{20})$, ecc..), risultano di fatto gli unici per i quali è possibile trovare una soluzione corretta e completa in tempi certamente ragionevoli.

Nella classe NPC esistono molti problemi pratici per i quali è stato necessario trovare soluzioni che approssimassero una soluzione, mediante metodi polinomiali, usando criteri di approssimazione (Vedi sezione 3) o mediante algoritmi randomizzati (Vedi sezione 4).

Dati due problemi arbitrari P_1, P_2, si dice che P_1 è polinormiarmente riconducibile a P_2

$$P_1 \propto_{poly} P_2$$

se esiste un algoritmo che risolve P_1, indipendentemente dalla sua complessità, per il quale è possibile, mediante adattamenti polinomiali, risolvere P_2.

[1] Di fatto l'appartenenza o meno di un problema ad una certa classe non è certa, ad esempio è sempre possibile trovare un algoritmo polinomiale che risolva un problema fino ad allora ritenuto NP, facendolo quindi declassare

[2] Ovvero che prevede una risposta dicotomica ad un dato problema

[3] La questione che P sia uguale ad NP è tutt'ora aperta.

Capitolo 1

Algoritmi Non Deterministici

Per chiarire gli aspetti legati ai problemi NP è utile introdurre dei costrutti sintattici, con una relativa semantica informale, che permettono di scrivere veri e propri programmi non deterministici. Il nostro linguaggio farà uso di tutti i costrutti più noti dei classici linguaggi imperativi, con l'aggiunta di soli tre comandi:

x=choice(S) considerando S come un insieme di cardinalità n, il comando divide il normale flusso del programma in n flussi distinti che continuano la computazione in maniera parallela ed indipendente tra di loro. Tutte le ramificazioni hanno inizialmente tutti i loro stati identici tranne per la variabile x che assume in ogni ramificazione un valore distinto dell'insieme S

failure comando che arresta l'esecuzione della ramificazione in cui è invocato (e solo quella), in genere perchè la strada seguita non porta ad una soluzione

success comando che termina tutte le computazioni, in quanto è stato trovato un risultato valido

Per valutare la complessità computazionale dell'algoritmo si valuta la complessità a partire dall'inizio, fino a quando non è stato trovato success. A questo fine, le ramificazioni che portano a failure è come se *non fossero mai esistite*.
A contrario, si ha failure solo quando *tutte* le computazioni danno failure; per questo, in queste situazioni, la complessità di un problema non avrà in genere la stessa complessità del suo complementare.

Vedremo adesso un esempio di algoritmo non deterministico.

1.1 Sorting non Deterministico

Abbiamo un vettore A di interi con $|A| = n$, si stampa un vettore B inizializzato a 0, che contiene gli elementi di A ordinati.

```
1  Sort(A, n){
2      for i=0 to n-1 do {
3          j = choice({1 ... n});
4          if(B[j] != 0) failure;
5          B[j] = A[i];
6      }
7      for i=0 to n-2 do
8          if(B[i] > B[i+1]) failure;
9
10     print(B);
```

```
11      success;
```

Alle righe 2-6 è possibile osservare come ad ogni iterazione del *for* il processo si ramifichi in n processi distinti, ognuno che lavora su un elemento specifico di B. Dopo il primo passo avremo quindi n array B differenti, dove:

$$\forall B_0[0] \dots B_{n-1}[n-1] = A[0]$$

ovvero n array di output diversi, contenti 0 in tutte le posizioni tranne che in un'unica posizione (sempre diversa gli uni dagli altri) in cui è presente il primo elemento di A. Al secondo passo abbiamo invece n^2 processi a cui bisogna togliere quelli che si sono ritrovati per la seconda volta con lo stesso valore della variabile j (a causa del controllo in posizione 4), restando quindi $n^2 - n$ processi con altrettanti array distinti, i quali contengono i primi due elementi di A in tutte le combinazioni possibili.

Questo processo risulta essere completo, in quanto non solo viene garantito che tutti gli array finali B_i siano distinti gli uni dagli altri, ma anche che essi contengono tutte le possibili configurazioni di A. Il metodo usato per trovare gli array è per *differenza* in quanto si generano tutte le possibili combinazioni dell'array A, elimiando in corsa gli array che non saranno sicuramente distinti alla fine del processo.

Ora che abbiamo una serie di array B_i che contengono tutte le possibili permutazioni dell'array A, non resta altro che eliminare quelli non ordinati mediante il controllo in 7-8 che lascia proseguire solo i processi che contengono l'array ordinato.

Visto che il ramo che ha success esegue due cicli *for* in sequenza, la complessità del Sorting non deterministico risulta essere $\Theta(n)$

Capitolo 2

I problemi NP-hard

La classe di problemi NP-hard ed NPC contiene una serie di problemi noti ai quali è possibile far ricondurre un'insieme molto ampio di problemi pratici.

2.1 Soddisfacibilità

E' il problema NPC per eccellenza, in quanto è possibile far ricondurre qualsiasi altro problema NPC a questo (Vedi teorema 2.1.1).

Data un'espressione booleana a due livelli di logica in CNF[1], dove sono presenti variabili logiche affermate o negate legate mediante congiunzioni di disgiunzioni, per esempio:

$$E = (x_1 \wedge \neg x_2) \bigvee (x_3 \wedge x_4) \bigvee x_5 \bigvee (x_6 \wedge x_7 \wedge x_8) \cdots \bigvee x_n$$

L'obiettivo è trovare se è possibile avere [2] una possibile configurazione di $x_1, x_2, \ldots, x_n$ di variabili booleane che soddisfano E, in termini formali occorre valutare se:

$$\exists \{x_1, \ldots, x_n\},\ x_i = \{0, 1\} \mid E = 1$$

2.1.1 Soddisfacibilità non deterministico

Per quanto possa risultare, a prima vista, complesso, il problema della Soddisfacibilità può essere risolto in $\Theta(n)$ mediante un semplice algoritmo non deterministico che prende in input un'espressione E di cardinalità n, e restituisce success se E è soddisfacibile, failure altrimenti.

```
1 Sat(E){
2     for i=0 to n-1 do
3         X[i] = choice({true, false});
4     if(E(X[0] : X[n-1])) success;
5     else failure;
6 }
```

Con l'espressione $E(X[0] : X[n-1])$ si vanno ad instanziare tutte le variabili libere di E con i termini *ground* generati dall'inizializzazione di X. E' chiaro osservare come l'algoritmo termini in success solo se viene trovata una qualunque configurazione valida di termini ground che soddisfa E, che è esattamente quanto vogliamo ottenere.

[1] Congiuntive Normal Form

[2] In generale non chiede di trovarla, ma solo se esiste, essendo un problema decisionale; di fatto è possibile dimostrare che l'algoritmo per valutare l'esistenza è polinomiarmente riconducibile a quello per trovare la soluzione, e viceversa

Teorema 2.1.1 (di Cook)**.**

$$\textit{Soddisfacibilità} \subset P \Leftrightarrow P = NP$$

Dimostrazione Si prende un algoritmo non deterministico $A(I)$, con I i dati di input, purchè polinomiale[3] in $p(n)$ e di l istruzioni in un *word model* a W bit. Si crea una formula che definisce il programma usando le variabili logiche.
Usiamo inoltre un costrutto $B_X(i,j,t)$, relativo al programma, per cui

X indica l'insieme di parole su cui si applica, siano essi dati o un algoritmo

i indica **la parola** $1 \leq i \leq p(n)$

j indica il $0 \leq j \leq w$ **bit** alla parola i

t indica **il tempo** di esecuzione $0 \leq t < p(n)$, misurato in base al numero di passi, per cui, relativamente ai due estremi, $t = 0$ rappresenta l'inizializzazione, e $t = p(n)$ la fine del programma (non significativa ai nostri scopi, in quanto è la semplice valutazione della success o della failure).

dunque $B_A(i,j,t)$ non fa altro che restituire il bit j-esimo della parola i valutata al tempo t nell'algoritmo A.
Definiamo inoltre $S(i,t)$ che restituisce un valore booleano, ed indica se l'istruzione i con $1 \leq i \leq l$ è eseguita o meno al tempo t.
Costruiamo un'espressione booleana che rappresentanta, in tutto e per tutto, il nostro algoritmo:

$$Q = C \wedge D \wedge E \wedge F \wedge G \wedge H \tag{2.1}$$

dove

C rappresenta lo **stato iniziale** della memoria, che contiene solo l'input, ovvero

$$C = \textstyle\bigwedge_{i,j} T(i,j,0) \quad \text{con } 1 \leq i \leq p(n) \text{ e } 0 \leq j \leq w$$

dove

$$T(i,j,0) = \begin{cases} B_A(i,j,0) & B_I(i,j,0) = 1 \\ \overline{B}_A(i,j,0) & \text{altrimenti} \end{cases}$$

Quindi se $C = 1$ significa che all'inizio c'è **solo** l'input in memoria

D controlla che **la prima istruzione ad essere eseguita è la numero 1**, ovvero che al tempo 1 si esegua solo l'istruzione 1 $S(1,1)$ e nessun'altra istruzione $\overline{S}(i,1)$ per ogni $i > 1$, ovvero

$$D = S(1,1) \bigwedge_{1<i<p(n)} \overline{S}(i,1)$$

E indica che dopo il passo i-esimo di A, **si esegue esattamente la prossima istruzione**; ovvero si valuta che ad ogni passo (2.2) viene eseguita almeno un'istruzione e non più di una (prima e seconda parte di 2.3), quindi

$$E = \bigwedge_t E_t \tag{2.2}$$

dove

$$E_t = \left(\bigvee_{1 \leq i \leq l} S(i,t) \right) \bigwedge \left(\bigwedge_{i,j} (\overline{S}(i,t) \vee \overline{S}(j,t)) \right) \tag{2.3}$$

F esprime la prossima istruzione, ovvero considerando solo l'istruzione corrente, si valutano tutti i costrutti sintattici e si valuta se viene effettivamente eseguita **l'istruzione logicamente successiva**

$$F = \bigwedge_{i,t} F_{i,t}$$

[3] è da considerarsi sia come il tempo del programma, che come il numero di parole generate

dove

$$F_{i,t} = \overline{S}(i,t) \vee L$$

in cui $\overline{S}(i,t)$ risulta vera in tutti i casi, e quindi non ci interessa il valore di L, tranne nel caso in cui siamo all'istruzione corrente (è un modo per selezionarla, in pratica), dove L è

$L = S(i,t+1)$ se $S(i,t)$ è failure, success, end.
$L = S(k,t+1)$se $S(i,t)$ è goto k
$L = ((B_A(j,1,t-1) \wedge S(k,t+1)) \vee (\overline{B}_A(j,1,t-1) \wedge S(i+1,t+1))$se $S(i,t)$ è if x[4] then goto k
$L = S(i+1,t+1)$ altrimenti

G esprime **il contenuto della memoria** dopo l'istruzione i[5]

H verifica che **l'istruzione al tempo $p(n)$ è success**[6]. ovvero se una delle k istruzioni success presenti nel programma (indicata genericamente con S_i) è eseguita al tempo $p(n)$

$$H = \bigvee_{1 \leq i \leq k} S(S_i, p(n)) \tag{2.4}$$

In questo modo abbiamo una formula di lunghezza

$$r(N) \leq O(p^4(n))$$

che rappresenta **qualsiasi** algoritmo NP, se esistesse un algoritmo deterministico $q(r(n))$ per valutare la formula Q in tempo polinomiale, sarebbe possibile risolvere qualsiasi problema NP in tempo polinomiale, e quindi troveremo che $P = NP$ □

In altre parole, visto che il problema della Soddisfacibilità è il più difficile della sua classe, dimostrare che è un problema P significa dimostrare che $P = NP$, ovvero la non esistenza di problemi non risolubili in maniera deterministicamente polinomiale.

2.2 Clique Massima

Dato un grafo G, si dice *Clique* di ordine n, un'insieme di n nodi distinti $N_1, \ldots, N_n \in G$ tale per cui ogni nodo è connesso con tutti gli altri dell'insieme. Il problema consiste, per un generico grafo G, di trovare la clique al suo interno di ordine massimo.
è facile dimostrare come:

$$\text{Max-Clique} \propto \text{K-Clique}$$
$$\text{e}$$
$$\text{K-Clique} \propto \text{Max-Clique}$$

2.2.1 K-Clique non deterministico

Esamineremo ora, a titolo d'esempio, un algoritmo non deterministico, che risolve il problema di verificare l'esistenza di una clique di ordine k su un grafo G con N nodi

```
1 DecisionalK-Clique(G, N, k){
2     for i=1 to k do
3         t = choice(1:N);
4         if(t.isIn(S)) failure;
5           else S=S U {t};
6
```

[4] rappresentato nella cella j

[5] troppo prolissa da descrivere in queste pagine, ma in pratica prende tutti i costrutti sintattici del linguaggio e calcola la modifica dello stato dopo la loro esecuzione

[6] per semplicità assumiamo che se il programma deve eseguire l'istruzione success, la esegue fino al tempo $p(n)$

```
7     for allpairs(i,j), i.isIn(S), j.isIn(S), i !=j do
8         if(!(i,j).isIn(G)) failure;
9     success;
10 }
```

in pratica quello che si fa è selezionare tutti i possibili insiemi di nodi presi k a k, poi si eliminano quelli non presenti all'interno del grafo in questione, se resta almeno un insieme di k nodi collegati fra loro, si restituisce success.

2.3 Knapsack

E' un problema molto noto che è possibile descrivere illustrando il problema pratico di dover inserire degli oggetti in uno zaino di capienza limitata.

Formalmente abbiamo un insieme di oggetti, descritti mediante n variabili booleane che rappresentano l'evento in cui l'oggetto i viene portato (1) o scartato (0):

$$O = \{o_1, \ldots, o_n\}$$

a cui associamo un insieme di valori interi dei pesi:

$$W = \{w_1, \ldots, w_n\}$$

e un insieme di valori interi che rappresentano il valore di ciascun oggetto:

$$V = \{v_1, \ldots, v_n\}$$

Il problema consiste in trovare la combinazione di valori delle variabili di O tale percui viene massimizzato il valore complessivo:

$$\max \left\{ \sum_i o_i v_i \right\}$$

avendo il vincolo di mantenere il peso complessivo sotto una soglia K:

$$\sum_i o_i w_i \ \leq K$$

2.3.1 D-Knapsack non deterministico

Esamineremo qui una versione non determinista che risolve il problema del Knapsack decisionale, ovvero quel problema che si occupa di verificare l'esistenza di una serie di oggetti che, insieme, raggiungono un valore complessivo maggiore di un certo minimo:

$$\sum_i o_i v_i \ \geq M$$

Anche qui, è facile dimostrare che il problema così esposto è polinomiarmente riconducibile al problema del Max Knapsack.

```
1  DecisionalKsProblem(O, W, V, k, m){
2      for i=0 to n-1 do
3          X[i] = choice({0, 1});
4      if(Sum(X[i] W[i]) > k or
5          Sum(X[i] V[i]) < m) failure;
6      else success;
7  }
```

Con Sum() funzione ausiliaria utilizzata per effettuare la sommatoria dei valori passati come parametri. Dato che per n iterazioni viene inserito un valore diverso, per un diverso flusso, all'interno del vettore X, avremo alla fine tanti vettori X, in rispettivi processi diversi, quante sono le possibili permutazioni di n elementi binari. A beneficio di chiarezza è possibile immaginare che il vettore X contenga, per ogni elemento, contemporaneamente il valore 0 e 1. Eliminare successivamente quei processi che non contengono il corretto sottoinsieme di oggetti risulta piuttosto semplice.

La complessità del DKP risulta dunque essere $\Theta(n)$

Capitolo 3

Algoritmi di approssimazione

Gli algoritmi di approssimazione vengono utilizzati soprattutto quando è necessario affrontare un problema NP-Hard cercando di ottenere una soluzione approssimata che dista (numericamente parlando) dalla soluzione reale per un certo errore massimo ϵ. Questo errore deve essere certificato, altrimenti non si tratta di una soluzione approssimata, ma solo di un'insieme di euristiche.

Marcando con un opportuno simbolo S^* l'ipotetica soluzione ottima, e con un altro $\widehat{S}$ la soluzione approssimata, è possibile certificare un algoritmo approssimato, definito k-**competitivo**, la cui soluzione dista da quella ottima al massimo per un fattore moltiplicativo k, tipici problemi sono di:

Massimizzazione $M^* \geq \widehat{M} \geq \dfrac{M^*}{k}$

Minimizzazione $m^* \leq \widehat{m} \leq km^*$

Ad esempio un algoritmo che risolva il problema della Clique Massima (vedi 2.2) in maniera 2-competitiva, significa che se esiste nell'instanza del problema una clique di massima ordine 6, l'algoritmo potrà al massimo commettere l'errore di restituire 3.

Algoritmi di questo tipo sono semplici da certificare, ma non sono certo il massimo dell'utilità

3.1 Approssimazione assoluta

Dato un problema P e date le sue istanze [1] I. Un algoritmo A si dice di approssimazione assoluta se e solo se:

$$\forall I \ |F^*(I) - \widehat{F}(I)| \ \leq k \qquad \text{per un } k \text{ fissato}$$

ovvero se la distanza numerica tra le due soluzioni non supera mai k

Teorema 3.1.1 (12.3)**.** *Il problema del Knapsack con approssimazione assoluta è NP-Hard*

Dimostrazione Per assurdo poniamo di avere un algoritmo $A \subset P$ che generi $\widehat{F}(I)$ entro k, e siano

$$I(P_i, W_i) \qquad 1 \leq i \leq n$$
$$I'((k+1)P_i, W_i) \qquad 1 \leq i \leq n$$

due istanze del problema con lo stesso M, chiaramente le due istanze hanno lo stesso insieme di soluzioni possibili; inoltre è possibile osservare che $F^*(I') = (k+1)F^*(I)$.
A questo punto se consideriamo $\widehat{F}(I') \neq F^*(I')$ significa che esiste almeno un elemento della soluzione ottima che non ho considerato in quella approssimata, ma in tal caso si avrebbe

[1] configurazioni di dati di input

$$\widehat{F}(I') \leq F^*(I') - k + 1$$

Teorema 3.1.2 (24)**.** MAX Clique $\propto$ MAX Clique con approssimazione assoluta

3.2 Epsilon-approssimazione

Definiamo A un algoritmo $f(n)$-approssimato $\Leftrightarrow \forall I$ di dimensione n vale:

$$\frac{|F^*(I) - \widehat{F}(I)|}{F^*(I)} \leq f(n)$$

se $f(n)$ è un ϵ scelto a priori, la $f(n)$-approssimazione risulta essere una ϵ-approssimazione.

E' anche possibile definire *schemi* di ϵ-approssimazioni se impostiamo ϵ come parametro di A

Teorema 3.2.1 (12,8)**.** *ciclo hemiltoniano $\propto_{poly}$ commesso viaggiatore ϵ-approssimato*

Dimostrazione Il ciclo hemiltoniano è un percorso su un grafo che tocca tutti i vertici una sola volta, la differenza col commesso viaggiatore è che quest'ultimo ha anche i pesi sugli archi.
E' dunque abbastanza naturale definire

commesso viaggiatore ϵ-approssimato $\propto_{poly}$ ciclo hemiltoniano

per ottenere l'inverso è invece possibile definire il problema del commesso viaggiatore come il problema di trovare tutti i cicli hemiltoniani sul grafo, e poi di prendere quello di costo minimo

3.3 Problema del Load Balancing

Abbiamo m processori ed n processi, rispettivamente con tempo di completamento pari a $t_1, \ldots, t_n$. Il problema è distribuire i processi sui processori in modo da minimizzare il tempo di completamento complessivo.

Allo scopo è possibile usare l'euristica *LPT* (lesser process time), ovvero si ordinano i processi per ordine di lunghezza decrescente e poi si assegnano ai vari processori mano a mano che diventano liberi

Teorema 3.3.1 (12,5)**.** *Nel problema del Load Balancing usando l'euristica LPT, per ogni istanza di I si ha che*

$$\frac{|F^*(I) - \widehat{F}(I)|}{F^*(I)} \leq \frac{1}{3} - \frac{1}{3m}$$

3.4 Problema del Bin Packing

Un problema simile al Load Balancing in cui abbiamo n oggetti di lughezza $l_1, \ldots, l_n$, una serie di *bin* di dimensione L. Il problema è far entrare tutti gli n oggetti nel numero minore di bin

3.4.1 Euristiche del BP

E' possibile sfruttare delle euristiche al fine di ottenere una buona approssimazione, di seguito alcune delle più usate:

First Fit Per ogni oggetto, si inserisce nel primo bin capace di contenerlo

Best Fit Per ogni oggetto, si inserisce nel bin capace di contenerlo di capacità residua minore

Decreasing First Fit First Fit con gli oggetti presi in ordine di lunghezza decrescente

Decreasing Best Fit First Fit con gli oggetti presi in ordine di lunghezza decrescente

Teorema 3.4.1 (12,7)**.** *First Fit e Best Fit usano un numero di bin n:*

$$n \leq \frac{17}{10}F^* + 2$$

Decreasing First Fit e Decreasing Best Fit usano un numero di bin n:

$$n \leq \frac{11}{3}F^* + 4$$

Capitolo 4

Algoritmi randomizzati

Utilizzati prevalentemente per l'ottenimento di risultati accettabili per problemi difficili (NP, NP-C, ecc..), gli algoritmi randomizzati fanno uso del caso per effettuare scelte in un qualche punto dell'algoritmo. Esistono due paradigmi di programmazione che rientrano in questa categoria, gli algoritmi:

Las Vegas: forniscono una soluzione **certamente corretta** in un tempo **probabilmente breve**

Monte Carlo: forniscono una soluzione **probabilmente corretta** in un tempo **certamente breve**

4.1 Algoritmo di Miller e Rabin

Il problema di stabilire se un dato numero N è primo è da sempre stato molto importante per diverse applicazioni, di cui la crittografia ne è solo l'esempio più noto.

Uno degli algoritmi Monte Carlo più semplici è senz'altro quello di Miller e Rabin che presuppone le seguenti assunzioni dettate da deduzioni logiche

1. N è dispari, altrimenti è senz'altro composto
2. $N - 1 = 2^{\omega}z$ con z dispari e ω è il più grande esponente che possiamo dare a 2

Considerando un intero arbitrario y tale per cui $2 \leq y \leq N-1$, se N è primo deve rispettare le seguenti proprietà

$P_1 \ : \ mcd(N, y) = 1$

$P_2 \ : \ (y^z mod N = 1)$ or $(\exists i, \ 0 \leq i \leq \omega - 1 \mid y^{2^i z} \mod N = -1)$

P_1 deriva dalla definizione di numero primo, mentre P_2 deriva da proprietà note dell'algebra modulare.

Lemma 4.1.1. *Se N è composto, allora il numero di interi y compresi tra 2 e $N-1$ che soddisfano P_1 e P_2 è inferiore ad* $\dfrac{N}{4}$

In pratica, se prendiamo un numero y (chiamato *certificato probabilistico*) **a caso** tra 2 e $N - 1$, e almeno uno dei due predicati sono falsi, N è certamente composto. D'altro canto se sono veri, possiamo solo dire che N è composto con probabilità minore di $\frac{1}{4}$, quindi l'algoritmo in questo caso si limita ad affermare che il numero è probabilmente primo. Per effettuare il controllo utilizziamo la seguente procedura

```
1 int Verifica(N, y):
2  if(P1(N, y)==false or P2(N, y)==false) return 1
3    else return 0
```

Se assume 1 allora N è certamente composto, se assume 0 allora N è *probabilmente* primo.

Valutazione costi

Per ottenere $N-1 = 2^{\omega}z$ è sufficiente dividere $N-1$ per $O(\log N)$ volte. Nella procedura, la verifica di P_1 è eseguita in tempo polinomiale con l'algoritmo di Euclide, ma il calcolo delle potenze $y^z, y^{2z}, \ldots, y^{2^{\omega-1}z} = y^{\frac{(N-1)}{2}}$ in P_2 è critico, in quanto esponenziale. Per rendere l'algoritmo polinomiale è necessario eseguire quest'ultima operazione mediante un preciso algoritmo polinomiale. L'algoritmo vero e proprio esegue k iterazioni, con k fissato a priori o fornito con l'input, prendendo k numeri a caso tra $2, \ldots, N-1$ in modo da ottenere k risultati probabilmente corretti. Siccome i risultati sono tutti indipendenti ed ognuno ha la stessa probabilità $\frac{1}{4}$ di affermare erroneamente che un numero è composto, la possibilità che sbagli in caso affermi che è primo risulta essere $\frac{1}{4^k}$.

```
1 int isPrimoMR(N):
2    for i=1 to k do
3      y=rand(2, N-1);
4      if Verifica(N,y)==1 return 0;
5    return 1
```

Restituisce 1 in caso il numero sia primo, 0 altrimenti.
La complessità è banale da calcolare, in quanto esegue k volte un ciclo di operazioni polinomiali $p(n)$, quindi la complessità finale è $O(kp(n))$

I numeri primi godono di una proprietà forte riguardo la loro densità sull'asse dei naturali, è infatti noto che la cardinalità di numeri primi minori di N tende ad $\frac{N}{\log_e N}$ per N molto grande. Questo significa che se N è grande, nel suo intorno di $\ln N$ cade mediamente un numero primo. Essendo $\ln N$ proporsionale alla dimensione n di bit di N, il numero di prove attese è polinomiale in n. E' possibile creare un algoritmo per la generazione dei numeri primi dove si prende un *seme* N di n bit, i cui estremi sono uguali ad 1 e i valori nel mezzo sono generati casualmente con la stringa S.

```
1 long Primo(n):
2  N=1S1 con S sequenza di n-2 bit casuali
3  while Test_MR(N)==0 do N=N+2;
4  return N;
```

Non tutti i problemi NP hanno la fortuna di avere (o poter ottenere) un certificato sul quale costruire un algoritmo P randomizzato. E' quindi possibile estendere le precedenti classi di complessità introducendo una nuova classe

RP classe di problemi decisionali verificabili in tempo polinomiale randomizzato

Dato infatti un problema decisionale e una sua arbitraria istanza I di lunghezza n, definiamo un certificato probabilistico y di I un'informazione di lunghezza polinomiale in n estratta perfettamente a caso da un insieme associato ad I, tale per cui esiste un algoritmo polinomiale in n ed applicabile ad ogni coppia $< I, y >$, che si occupa di verificare che I abbia la proprietà richiesta dal problema con probabilità maggiore di $\frac{1}{2}$ o, altrimenti, che attesta con **certezza** che non la possiede. In questo caso il problema si dice verificabile in tempo polinomiarmente randomizzato.

Parte II

Accesso ai dati e loro compressione

Capitolo 5

Compressione di testi e di interi

Tutti i file[1] non sono altro che stringhe di un alfabeto Σ memorizzate su un supporto, in genere il disco fisso. Da sempre è stato presente il problema di comprimere questi file in modo da contenerli nel minor spazio possibile pur mantendendo intatto il contenuto informativo. L'idea di base per comprimere un testo è di ricercare in questo, matematicamente, o comunque in modo automatico, dei *pattern* comuni, o comunque una ridondanza di informazioni che renda possibile la suddivisione tra informazioni essenziali e derivate.

Esempio Avendo un testo formato da n volte il carattere a

$$aaaaa \dots a$$

è possibile comprimerlo semplicemente memorizzando entrambi questi valori

$$< n, a >$$

occupando lo spazio di 2 caratteri invece che di n, mantenendo intatte tutte le informazioni sul testo, dato che è possibile ritornare al testo di origine molto facilmente □

Un testo si definisce *compromibile* quando non è casuale, viceversa si avrebbe per definizione un testo con nessun contentuto informativo derivabile, in quanto ogni carattere è generato in maniera indipendente dall'altro.

Uno dei modi più semplici per comprimere un testo è quello di associare ad ogni carattere una *codeword*, ovvero un codice binario univoco, in accordo con la sua frequenza nel testo (e quindi la probabilità di trovarlo); più sarà frequente un carattere, più piccola sarà la sua codeword.

Esempio Prendiamo ad esempio un testo con $\Sigma = \{a, b, c, d\}$ e con le relative probabilità e codeword

Carattere	Probabilità	Codeword
a	.5	1
b	.3	01
c	.1	101
d	.1	011

da qui possiamo comprimere il testo andando a sostituire i caratteri con le codeword corrispondenti.
Essendo i caratteri conservati in almeno 8 bit, convertirli in stringhe di bit di cui il carattere meno probabile conta solo 3 bit è un grosso guadagno. □

[1] in questo contesto usiamo il termine *file* a tutti i livelli di astrazione, considerando quindi documenti, immagini e, naturalmente, stringhe di bit

Le codeword generate nell'esempio 5 non sono però accettabili, in quanto nel testo codificato sarebbero presenti ambiguità (vedi ab con il carattere c). Per ovviare a questo si usa un così detto *prefix code*, ovvero un codice a lunghezza variabile dove nessuna codeword è prefisso di un'altra.

Definendo $L_C(s)$ come la lunghezza della codeword del carattere $s \in \Sigma$, codificato mediante il prefix code C, definiamo la **lunghezza media** di in prefix code C come

$$L_a(C) = \sum_{s \in \Sigma} p(s) L_C(s)$$

dove $p(s)$ indica la probabilità del carattere s nel testo.
C risulta essere ottimo se

$$\forall C' \; L_a(C) \leq L_a(C')$$

Teorema 5.0.2 (Kraft-McMillan)**.** *Per ogni codice univocabilmente codificabile con una certa lunghezza media, esiste un prefix code con la stessa lunghezza media*

Teorema 5.0.3 (Golden Rule)**.** *Se C è un prefix code ottimo allora per ogni $s, s' \in \Sigma$ abbiamo che*

$$p(s) < p(s') \Rightarrow L(s) \geq L(s')$$

Ovvero che se un prefix code è ottimo allora le codeword più lunghe sono associate ai caratteri più rari

A questo punto possiamo definire il concetto di *entropia* come indice di disordine nel testo. Maggiore è questo indice, e meno il testo risulterà comprimibile, il testo creato casualmente ha quindi, per definizione, la massima entropia possibile

$$H_o(T) = \sum_{s \in \Sigma} p(s) \log_2 \frac{1}{p(s)}$$

Da questa definizione possiamo ricavarci i seguenti bound

$$H_o(T) \leq L_a(C)$$
$$0 \leq H_o(T) \leq \log_2 |\Sigma|$$

con C identifichiamo il codice univocamente codificato.

Questo è derivato dal fatto che in presenza della massima ridondanza dei dati, e quindi della massima comprimibilità, abbiamo tutti i caratteri del testo uguali, quindi $H_o(T) = 1 \log_2 1 = 1 \cdot 0 = 0$; viceversa nel caso della minima comprimibilità, abbiamo tutti i caratteri che hanno la stessa frequenza, quindi $p(s) = \frac{1}{|\Sigma|}$ per qualsiasi s, da qui otteniamo

$$H_o(T) = \sum_{s \in \Sigma} p(s) \log_2 p(s)^{-1} = p(s) \sum_{s \in \Sigma} \log_2 p(s)^{-1} = |\Sigma|^{-1} \cdot |\Sigma| \log_2 |\Sigma| = \log_2 |\Sigma|$$

Ora vedremo alcuni algoritmi utili a comprimere un testo.

5.1 Huffman Code

E' storicamente uno dei metodi migliori per generare il prefix code ottimo, e possiamo riassumere le sue proprietà come segue:

- genera il prefix code ottimo
- è veloce in codifica e decodifica

- $L_a(C) = H_o(T)$ se le probabilità dei simboli sono potenze del due
- occupazione di $O(|\Sigma| \log |\Sigma|)$ in spazio
- si comporta male in caso di distribuzione probabilistica dei caratteri sbilanciata

Dato Σ e un insieme P delle probabilità di ogni $c \in \Sigma$, con $|\Sigma| = n$, possiamo definire un metodo che trovi il prefix code ottimo.

1. si crea un insieme T formato dagli insiemi composti inizialmente dagli n singoletti corrispondenti a tutti i $c \in \Sigma$ etichettati con la loro probabilità
2. in T trovo i due sottoinsiemi $m_1, m_2 \subset T$ con minore probabilità
3. rimpiazzo gli insiemi m_1, m_2 con l'insieme $\{m_1, m_2\}$ la cui probabilità è pari alla somma delle probabilità degli insiemi di partenza $p(m_1) + p(m_2)$
4. ripeto i passi dal punto 2 a qui per $n - 1$ volte
5. adesso T contiene un solo insieme che è la radice dell'*albero huffman*

Considerando le unioni degli insiemi come collegamenti da un'insieme di caratteri all'altro in tempi diversi, è possibile vedere il procedimento come la costruzione di un albero sulle cui foglie abbiamo i caratteri e i cui nodi sono le unioni.
Il prefix code è ricavabile assegnando agli archi verso i figli sinistri lo 0 e verso i figli destri il valore 1, concatenando i numeri binari dalla radice al carattere otteniamo la codeword associata a quel carattere.

Esempio Considerando i dati dell'esempio 5 abbiamo che gli insiemi costituenti i caratteri c e d sono i primi ad unirsi, ottenendo un insieme con associata probabilità .2, il quale si fonderà col singoletto b ottenendo un insieme con probabilità .5, finendo per fondersi col singoletto a. Costruendo le codeword abbiamo i seguenti valori:

Carattere	Probabilità	Codeword
a	.5	0
b	.3	10
c	.1	110
d	.1	111

Le codeword sono corrette in quanto nessuna è prefisso di un'altra

Naturalmente per la decodifica oltre al testo codificato occorre inviare le codeword associate ad ogni carattere.

5.1.1 Canonical Huffman

Huffman pur creando codeword corrette non rappresenta il massimo in quanto a compressione, molte informazioni possono essere ancora derivate, e non sono quindi essenziali. Per questo motivo esiste un procedimento che permette di strutturare la compressione in modo che sia molto veloce in decodifica. Avendo una serie di caratteri e la loro codeword ottenuta con Huffman, occorre seguire il seguente procedimento:

1. si ordinano i caratteri in modo crescente prima secondo la lunghezza della codeword e poi alfabeticamente
2. il primo elemento della lista assume come nuova codeword tanti zeri quanto era la lunghezza della codeword originaria

3. i successivi elementi che hanno la lunghezza della codeword uguale a quella dell'elemento precedente assumono, come codeword, il numero binario successivo a quello associato al carattere precedente

4. se la lunghezza della codeword aveva invece una lunghezza maggiore della precedente, oltre ad associare il numero binario successivo si aggiunge uno 0 in coda

L'algoritmo si spiega meglio con un esempio

Esempio Presentiamo ora un esempio di codeword generata da Huffman trasformata in forma canonica

Huffman	Canonical
a=11	b=0
b=0	a=10
c=101	c=110
d=100	d=111

Grazie a come è stato costruito, è possibile inviare solamente il numero di bit che una codeword possiede se conosciamo a priori la sequenza (e l'odine) di invio dei caratteri

$$<' a', 2 > <' b', 1 > <' c', 3 > <' d', 3 >$$

E' una sequenza in ordine alfabetico, essendo questa un'informazione implicita dall'ordine della sequenza, risulta derivata; quindi possiamo omettere i caratteri.

$$< 2 > < 1 > < 3 > < 3 >$$

Col procedimento inverso si ottengono le codeword associate ai caratteri, e da qui la decodifica del testo.

5.2 Move To Front

Algoritmo decisamente noto in letteratura, il Move to front lavora sulla sequenza di caratteri che comprendono l'alfabeto Σ di un testo T.
Si parte con una data sequenza di Σ nota a priori, e con il primo carattere di T, e si procede nel seguente modo

1. si prende il carattere in questione e per codificarlo ci segnamo la **posizione** dello stesso, all'interno della sequenza dei caratteri

2. spostiamo in testa alla sequenza il carattere in questione

3. selezioniamo il carattere seguente del testo e ripetiamo la procedura dalla posizione 1

Esempio Avendo una stringa da codificare $T = aabbc$ ed una sequenza iniziale $S = a, b, c, d$ abbiamo i seguenti passi di codifica

Codifica	S
0	a,b,c,d
00	a,b,c,d
001	b,a,c,d
0010	b,a,c,d
00102	c,b,a,d

Move To Front ha $L_a = 2H_o + O(1)$, considerando che Huffman paga H_o è solo 2 volte peggiore al caso pessimo.
Mentre al caso ottimo, per esempio con la stringa $T = 1^n, 2^n, \ldots, n^n$ abbiamo che Huffman lo codifica in $O(n^2 \log n)$, mentre questo algoritmo la codifica in $O(n \log n)$, questo è possibile solo perchè Move To Front non è un codice prefisso, si basa sulla posizione dei caratteri, non sulla loro probabilità

5.3 Run Length Encoding

Algoritmo di compressione decisamente semplice da implementare sia in codifica che in decodifica, inventato da *Golomb*[2], viene in genera usato per implementare i fax, visto la natura dei messaggi scambiati con questo strumento.

Avendo la stringa T da codificare, si generano una serie di coppie $(carattere, numero)$ che contano il numero di volte che il carattere appare consecutivamente

Esempio Avendo la stringa $T = abbbaacccca$ si codifica con

$$abbbaacccca \rightarrow (a,1)\ (b,3)\ (a,2)\ (c,4)\ (a,1)$$

Come è possibile intuire, funziona bene in testi in cui compaiono sequenze di caratteri identici

5.4 Lempel-Ziv

Utile per codificare le stringhe mediante una sequenza di triple del tipo:

$$< back, copy, c >$$

dove:

back sono il numero di caratteri per cui dobbiamo tornare indietro

copy sono il numero di caratteri da copiare

c il cui valore è semplicemente il prossimo carattere da inserire

In pratica data una stringa arbitraria T la codifica restituendo una serie di triple, il cui valore permette di recuperare la stringa iniziale in questo modo.
Avendo una generica tripla da analizzare, e un testo parziale T' che è stato già codificato, per codificare la parte successiva si

1. si sposta il cursore dalla fine della stringa T' indietro di $back$ posizioni
2. si copiano in fondo alla stringa, $copy$ elementi partendo dalla posizione del cursore
3. si inserisce in fondo a T' il carattere c

Esempio la stringa $T = aacaacabcaaa\$$ si codifica con la sequenza di triple:

$$< 0,0,a > < 1,1,c > < 3,4,b > \ldots$$

con la stringa parziale $T' = aacaacab$ in decodifica, la prossima tripla che ci troveremo ad analizzare è $< 6,3,a >$ che ci permette di ottenere $T' = aacaacab\mathbf{caaa}$.

Come è possibile osservare, nella terza tripla si copiano più elementi di quanti si potrebbe fare, in realtà questa scrittura è perfettamente coerente in quanto si arriva a copiare elementi già copiati durante il procedimento. □

Questo algoritmo è molto comodo in caso di alta ridondanza di pattern molto grandi

Esempio La stringa a^n si codifica con l'unica tripla $< a, n, \$ >$ □

Per la codifica viene tenuta traccia di un dizionario, la cui dimensione varia a seconda delle implementazioni, comunque ad ogni passo di codifica viene considerata la stringa più lunga del dizionario (politica *greedy*)

[2] Run-length encodings - S. W. Golomb (1966); IEEE Trans Info Theory 12(3):399

5.5 Codifica dei numeri

Esistono una serie di algoritmi utili per codificare numeri molto piccoli (vedi 5.5.1), formanti un intorno di un certo valore (vedi 5.5.2), o che cadono tutti entro un certo intervallo, presentando nel contempo diverse ripetizioni (vedi 5.5.3). Questi cercano di sopperire, in diversi modi, al fatto che tutti gli interi vengono codificati su parola, con molto spreco di spazio. Il risultato della loro codifica è una stringa di bit rappresentante i numeri codificati, che occupano molto meno spazio rispetto alla stringa orginaria, con la conseguenza di essere meno immediata in lettura.

Purtroppo tutti questi algoritmi, per come sono strutturati "rompono" tutte le pipeline dei processori attualmente in commercio, in quanto per ogni stringa codificata in questo modo occorrerebbe analizzarla bit a bit, ammesso di averne le primitive che lo consentano, fino a trovare i bit contenenti il numero effettivo, a quel punto caricare un blocco di bit, in numero, minori di una parola e poi passare ad analizzare bit a bit il numero successivo. Dato che i processori moderni sono progettati per operare su parola, il processo di decodifica in pratica risulta tutt'altro che semplice od efficiente.

5.5.1 Gamma Code

L'algoritmo γ-code è una codifica a lunghezza variabile e serve per comprimere i numeri. Questi sono codificati in binario, generalmente su parola di 32 bit, quindi se prendiamo ad esempio il numero 9 con la sua codifica in binario abbiamo

$$9 \rightarrow 000\ldots01001$$

ovvero 27 zeri prima di avere cifre significative, questo è un forte spreco.

Col γ-code il numero diventa una coppia $< cifre, numero >$, di cui si tiene traccia della parte significativa del numero in questione, e del numero di bit dello stesso, espresso in tanti "0" per quante sono le cifre meno uno, l'informazione sul numero delle cifre è essenziale per distinguere i vari numeri all'interno del testo (che diventa una stringa indistinta di bit). Quindi 9 viene codificato in

$$9 \rightarrow < 000, 1001 >$$

Con questo sistema, ogni numero x occupa $2\lfloor \log_2 x \rfloor + 1$ bit, considerando che l'ottimo è $\log_2 x$ bit, è un buon risultato

Esempio Abbiamo una stringa codificata col γ-code

$$T = 00010000\ldots$$

siccome ci sono 3 "0" prima del primo 1, il numero è composto dalle prime $3 + 1 = 4$ cifre dopo lo 0, ovvero

$$000\ \mathbf{1000}\ 0\ldots$$

che in decimale diventa il numero 8.

5.5.2 Rice Code

Usato per codificare numeri, questo algoritmo è un caso particolare del *Golomb-code*, ed utilizza un parametro k per il suo funzionamento.

Per codificare un numero arbitrario x si trovano due valori, un quoziente $q = \lfloor \frac{x-1}{k} \rfloor$ ed un resto $r = x - k \cdot q - 1$, con i quali si costruisce la stringa codificata piazzando all'inizio tanti "0" per quanto è il quoziente, un singolo "1" e la codifica in binario dei $\log k$ bit più significativi del resto.

Questo algoritmo risulta utile quando i numeri sono concentrati in un intorno di k, da scegliere opportunamente. In genere viene preso come riferimento il valore $k = 0.69 \cdot \mu_X$, considerando X come il campione di numeri da codificare, e μ_X come la media di X.

5.5.3 PForDelta Code

Altro codice utile per codificare numeri a basso contenuto di bit è il PForDelta code. Esso sfrutta l'idea che, facendo una statistica della distribuzione del numero di bit significativi su un campione di numeri omogeneo (provenienti dalla stessa fonte), la gaussiana risultante tende ad essere molto alta. Da qui l'idea di codificare il grosso dei numeri, in genere l'80-90esimo percentile, con un numero di bit pari a quelli più significativi del numero più alto di quell'insieme, considerando nel contempo la coda della gaussiana come *outlier*.

In pratica quello che viene fatto è ridurre tutti i numeri di una certa base k, in genere il numero minimale del campione, si fissa un numero di bit arbitrario b atto a contenere buona parte del campione, e si convertono tutti gli n numeri del campione come sequenza di n blocchi formati da b bit, che è la rappresentazione binaria su b bit. Qualora un numero abbia una rappresentazione binaria con cardinalità maggiore di b si segna opportunamente il suo blocco con un puntatore ad una lista di eccezioni, o con un codice di escape, dove vengono conservati i numeri (non convertiti) che non entrano in b bits.

La scelta di b risulta quindi critica, in quanto all'aumentare di b si aumenta lo spazio sprecato dai numeri, mentre la sua riduzione fa aumentare il numero di outlier; la cosa migliore è di scegliere b in modo da far ricadere il 90% del campione all'interno della codifica, lasciando il 10% dei numeri come eccezioni.

Esempio Abbiamo un campione formato da i seguenti numeri

$$3, 42, 2, 3, 3, 1, 1, \ldots, 3, 3, 23, 1, 2$$

considerando

$$b = 2$$
$$base = 0$$

abbiamo

Campione	3	42	2	3	3	1	1	...	3	3	23	1	2		
PForDelta	11	↪	10	11	11	01	01	...	11	11	↪	01	10	42	23

5.5.4 Interpolative Coding

5.6 Arithmetic Code

L'arithmetic code è il più grande compressore statistico, in genere usato per codificare immagini (jpg) o audio (mp3), e si comporta meglio di Huffman per testi con entropia bassa, quindi altamente comprimibili. In pratica quello che fa è ridurre il testo al solo valore della probabilità che quello specifico testo sia stato compresso.
Per spiegarne meglio il funzionamento affianchiamo un esempio alla spiegazione. Supponiamo di avere in un testo $T =$ "*bca*", i seguenti caratteri con la loro probabilità:

$$(a, .2)\ (b, .5)\ (c, .3)$$

Si collocano ora i caratteri su un asse di probabilità $[0, 1]$, mediante un certo ordine, in accordo alla loro probabilità

$$c_{.7}^{1}$$
$$b_{.2}$$
$$a_0$$

essendo il primo carattere del testo "b" ci segnamo l'intervallo $[.2, .7]$ in cui compare questo carattere, ed aggiorniamo l'asse con le nuove probabilità di ottenere uno qualsiasi degli altri caratteri

$$\begin{array}{cc} c^{1}_{.7} & c^{7}_{.55} \\ b_{.2} \Rightarrow & b_{.3} \\ a_0 & a_{.2} \end{array} \tag{5.1}$$

il successivo carattere risulta essere "c", quindi la nostra probabilità sarà compresa nell'intervallo $[.55, .7]$, ottenendo un nuovo asse incluso in quell'intervallo

$$\begin{array}{ccc} c^{1}_{.7} & c^{7}_{.55} & c^{.7} \\ b_{.2} \Rightarrow & b_{.3} \Rightarrow & b_{.58} \\ a_0 & a_{.2} & a_{.55} \end{array} \tag{5.2}$$

l'ultimo carattere è "a", quindi l'intervallo finale risulta essere $[.55, .58]$.
Dare i due estremi è ridondante ai fini della decodifica, in effetti sarebbe sufficiente dare un qualsiasi valore compreso in quell'intervallo, essenziale è che sia numericamente più corto possibile (ovvero con il numero di cifre decimali significative minimo).
In altre parole, quello che facciamo è, avendo l'intervallo $[l_n, l_n + S_n]$ prendiamo il valore in mezzo, ovvero $l_n + \frac{S_n}{2}$, e lo tronchiamo all'h-esimo bit, per un certo h.

Teorema 5.6.1. *Se $h = \lceil \log_2 \frac{2}{S_n} \rceil$, il valore ottenuto dal troncamento è ancora all'interno dell'intervallo, ovvero il valore dello scarto del troncamento è $\sum_{i=h+1}^{\infty} b_i \cdot 2^{-i} \leq \frac{S_n}{2}$, dove b_i è pari al valore della cifra i-esima in binario*

Dimostrazione Per garantire questo è necessario che

$$\sum_{i=h+1}^{\infty} b_i \cdot 2^{-i} \leq \sum_{i=h+1}^{\infty} 2^{-i} = 2^{-h} \sum_{i=1}^{\infty} 2^{-i}$$

ovvero è uguale alla serie geometrica di ragione 2^{-1} che converge ad 1, quindi

$$\sum_{i=h+1}^{\infty} b_i \cdot 2^{-i} \leq 2^{-h}$$

Considerando che il troncamento deve essere $\leq \frac{S_n}{2}$, abbiamo che

$$\frac{S_n}{2} = 2^{\log_2 \frac{S_n}{2}} = 2^{-\log_2 \frac{2}{S_n}}$$

quindi $h = \lceil \log_2 \frac{2}{S_n} \rceil$ □

Una volta ottenuto l'intervallo finale è possibile notare che tutte le seguenti affermazioni sono equivalenti:

- Il testo è comprimibile
- S_n è un numero grande
- L'intervallo è grande
- La probabilità è alta

Il limite inferiore per la compressione è $H_o(T)$ quindi è quasi ottimo, c'è però da considerare che qui facciamo i conti con un processore a precisione infinita, nella pratica le cose vanno in maniera peggiore.

5.7 Trasformata di Burrows-Wheeler

Si tratta di uno dei più complessi algoritmi di codifica, che allo stesso tempo permette di ottenere risultati molto apprezzabili. Dato il testo da comprimere T di lunghezza m, le operazioni *preliminari* da fare sono le seguenti:

1. si aggiunge un marcatore speciale # in fondo al testo
2. si crea una matrice m x m la cui prima riga contiene il testo da comprimere, e i caratteri occupano una cella ognuno, la riga successiva contiene il testo i cui caratteri sono *shiftati* di una posizione a destra in modo che il carattere uscente, l'ultimo, sia il primo, mentre il primo sarà il secondo e così via; riempiamo le altre colonne con i successivi shift fino ad avere un totale di m righe
3. si ordinano le righe lessicograficamente

Chiamiamo la prima e l'ultima colonna di questa matrice rispettivamente F ed L.
A questo punto possiamo osservare che entrambe le colonne contengono in realtà l'*intero* testo, il vantaggio è che L risulta essere localmente omogenea, e quindi facilmente comprimibile.
Possiamo notare che per un qualsiasi carattere in posizione i, $L[i]$ precede sempre $F[i]$ in T inoltre, dato L è possibile ricavare F.

A questo punto, il testo compresso non sarà altro che la stringa corrispondente alla colonna L a cui viene applicato nell'ordine Move to Front (Sez. 5.2), Run Length Encoding (Sez. 5.3) e l'Arithmetical Code (Sez. 5.6).

Esempio Diciamo di voler comprimere mediante la trasformata la stringa $T = BANANA$, si crea la matrice degli shift:

$$
\begin{array}{l}
BANANA\# \\
\#BANANA \\
A\#BANAN \\
NA\#BANA \\
ANA\#BAN \\
NANA\#BA \\
ANANA\#B
\end{array}
$$

si ordina lessicograficamente

F		**L**
A	NANA#	B
A	NA#BA	N
A	#BANA	N
B	ANANA	#
N	ANA#B	A
N	A#BAN	A
#	BANAN	A

a questo punto prendiamo $L = BNN\#AAA$ che risulta meglio comprimibile, possiamo quindi completare la compressione mediante l'applicazione di Move To Front, Run Lenght Encoding e Arithmetical code.

Decompressione

La decompressione risulta leggermente più complicata, data la stringa compressa T'

1. si decomprime con Arithmetical Code
2. si decomprime con Run Length Encoding

3. si decomprime con Move to Front
4. si ricava F ordinando L in ordine alfabetico

Una volta ricavata F abbiamo:

F	**L**
A	B
A	N
A	N
B	#
N	A
N	A
#	A

da qui si ricostruisce il testo all'incontrario, partendo quindi segnando il carattere # presente in L e osservando in che posizione è presente lo stesso carattere in F (l'ultima), da questa posizione si guarda quale carattere è corrisposto in L e si segna in cima al testo finora ottenuto (ottenendo quindi "$A\#$").
Si prosegue cercando in F il carattere in questione (A) prendendo, in caso di occorrenze multiple, l'indice del carattere in posizione uguale a quello presente in L, in questo caso abbiamo il carattere A che in L è l'ultimo di 3, quindi prendiamo l'ultimo carattere A presente in posizione 3 di F e andiamo a segnare sul nostro testo il carattere corrispondente in L, avendo ora la sequenza "$NA\#$". Procedendo in questo modo si ottiene la stringa originaria.

Capitolo 6

Hashing e strutture dati randomizzate

L'hashing è un sistema pratico per la creazione di dizionari, e risponde alla necessità di poter recuperare in tempi brevi una chiave specifica in un insieme di chiavi di grandezza n.
Il funzionamento di base usa un array di lunghezza m ($m << n$) di liste *di trabocco*, ed una funzione, detta appunto *funzione hash* che va da Chiavi $\rightarrow$ Indice, la quale viene usata sia per l'inserimento che per la ricerca.

L'inserimento di una chiave nella struttura dati avviene mediante la chiave stessa, a cui viene applicata la funzione hash, che restituirà l'indice nell'array in cui verrà inserita. Qualora ci fosse già un'elemento in quella posizione si mette la chiave da inserire in coda alla lista.

La ricerca avviene applicando la funzione hash alla chiave da ricercare, l'indice restituito identificherà la lista nell'array dove la chiave è contenuta. E' facile aspettarsi uno scenario al caso pessimo in cui tutti gli elementi stanno nella stessa posizione dell'array, occupando quindi un'unica lista di lunghezza n, con un tempo di ricerca di $\Theta(n)$. Se la funzione hash è uniforme[1] la lunghezza media delle liste di trabocco sarà di $\frac{n}{m}$ e avremo quindi un tempo di accesso al caso medio di $O(\frac{n}{m})$.

Il problema principale è, dunque, quello di trovare una funzione hash uniforme tra tutte le possibili m^n funzioni hash.

6.1 Hashing Universale

L'Hashing Universale è un metodo che, per ogni dataset da inserire U, seleziona casualmente una funzione hash presa da una precisa classe

$$H = \{h : U \rightarrow \{0, 1, \ldots, m-1\}\}$$

tale che

$$\forall x, y \in U \ |\{h \in H, h(x) = h(y)\}| = \frac{|H|}{m}$$

ovvero si tratta di quella classe il cui numero di funzioni che porta a collisioni è pari al rapporto tra la cardinalità della classe con la dimensione della tabella.

Definendo C_{xy} come la variabile che prende 1 se è presente una collisione tra x e y, ovvero $h(x) = h(y)$, troviamo che il numero di elementi C_x che collidono con x è pari a

[1] ovvero mappa le chiavi sull'array in maniera causale, e quindi uniformamente distribuita

$$C_x = \sum_{y \in U \setminus \{x\}} C_{xy}$$

da qui abbiamo che il valore atteso che avvenga una collisione con x è

$$E[C_x] = E[\sum_{y \in U \setminus \{x\}} C_{xy}]$$

e per la linearità della speranza abbiamo

$$E[C_x] = E[\sum_{y \in U \setminus \{x\}} C_{xy}] = \sum_{y \in U \setminus \{x\}} E[C_{xy}] \leq \sum \frac{1}{m}$$

La probabilità di collisione tra due elementi x ed y è $\frac{1}{m}$, quindi è possibile dimostrare che, al caso medio, se ci troviamo in una situazione in cui $n \leq m$ ovvero il numero di posizioni è maggiore degli elementi da inserire, la distribuzione mediante il meccanismo delle funzioni hash universali non genera collissioni.

Esempio Possiamo definire la famiglia di funzioni hash $h_{a,b}$ per qualsiasi $a \in Z_p^*$ e $b \in Z_p$ con $Z_p = \{0, 1, \ldots, p-1\}$ e $Z_p^* = Z_p \setminus \{0\}$ e p primo, in questo modo:

$$h_{a,b}(k) = ((ak + b) \mod p) \mod m$$

Chiamiamo questa famiglia

$$H_{p,m} = \{h_{a,b} : a \in Z_p^* \wedge b \in Z_p\} \tag{6.1}$$

6.2 Hashing Consistente

L'Hashing Consistente è un tipo particolare di struttura dati utilizzata per gestire strutture distribuite dove sia i documenti che i server, che per quanto ci riguarda sono semplicemente oggetti atti a contenere i documenti, sono distribuiti e aggiunti e rimossi in maniera dinamica. Il problema principale è di riuscire a distribuire il carico in maniera efficiente e gestire eliminazioni o inserimenti sia di documenti che di server nella maniera più locale possibile (che non coinvolga tutto il sistema).

L'idea di base è di distribuire sia i server che i documenti su una circonferenza provvista di un verso di rotazione, o una lista linkata circolare, se preferite. La distribuzione iniziale avviene più o meno casulmente mediante una funzione hash scelta casualmente da una famiglia di funzioni hash H uniformi; una volta distribuiti entrambi gli elementi, un generico documento verrà conservato nel server ad esso più vicino rispetto al senso di rotazione, la figura 6.1 mostra un esempio chiarificatore.

L'inserimento avviene semplicemente inserendo il nuovo server (o documento) all'interno della circonferenza, il quale si prenderà carico (o andrà a carico) dei relativi documenti (server).

Questa semplice intuizione permette di avere una serie di proprietà interessanti:

- segue una politica *fair*, ovvero i documenti vengono distribuiti a caso tra i server
- una volta che viene aggiunto un server, gli unici documenti che dovranno essere riassegnati sono quelli presenti sul server in questione, garantendo la consistenza del sistema

Hashing Congruente Lineare In questo esempio[2] abbiamo un insieme di oggetti I ed un insieme di bucket B entrambi uguali a $\{0, 1, \ldots, p-1\}$ per un numero primo p arbitrario. La famiglia $H_{p,m}$ definita nell'esempio 6.1, ovvero tutte quelle funzioni hash della forma $h(x) = (ax+b) \mod p$ per tutti gli $a, b \in \{0, 1, \ldots, p-1\}$.

In pratica, i parametri a, b vengono scelti a caso di volta in volta (che è equivalente a dire di scegliere le funzioni hash a caso) e l'elemento i verrà assegnato al bucket $(ai + b) \mod p$

[2] tradotto dalla tesi di master di Daniel M. Lewin

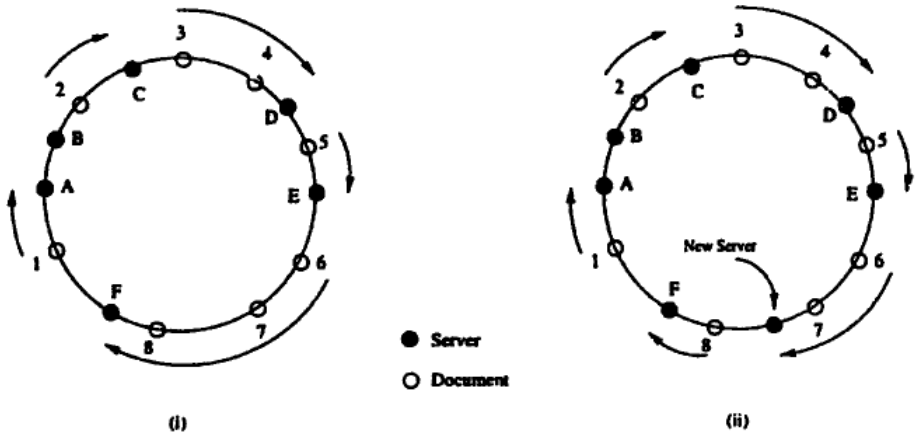


Figura 6.1: Un esempio di lista circolare in sui sia i server (in nero) che i documenti (in bianco) vengono inseriti con una funzione hash, l'appartenenza dei documenti segue il senso circolare e i documenti 6,7,8 vengono gestiti dal server F fino all'aggiunta del nuovo server che prenderà in gestione il 6,7

L'esempio 6.2 permette di definire una proprietà importante

Lemma 6.2.1. *Sia $H_{p,m}$ la famiglia di funzioni hash definita in 6.1 e siano x e y due oggetti distinti, la probabilità che $h(x) \neq h(y)$ è pari ad $\frac{1}{p}$, ovvero alla probabilità che vengano mappati in maniera uniforme ed indipendente*

La differenza sostanziale dell'hashing consistente con le funzioni hash semplici è che questi devono gestire una situazione in cui il codominio della funzione cambia (vengono inseriti o rimossi bucket). Adesso forniremo una spiegazione più formale del meccanismo usato.

Sia I l'insieme di oggetti e B l'insieme di bucket, definiamo una *view* come un sottoinsieme di B, che indica i bucket disponibili.
Una *ranged hash function* è una funzione di tipo $h : 2^B \mathrm{x} I \to B$ che specifica un assegnamento degli oggetti ai bucket *per ogni* view V.
Definiamo inoltre $h_v(i)$ come il bucket a cui l'oggetto i è stato assegnato nella view V, naturalmente vale che

$$h_v(I) \subseteq V \text{ per ogni vista } V$$

Possiamo adesso rendere più formali le caratteristiche elencate poco sopra

Bilanciamento gli oggetti vengono distribuiti a caso in ogni view, garantita se la probabilità che un oggetto i sia assegnato ad un bucket $b \in V$ è $O(\frac{1}{|V|})$

Monotonicità ogni volta che viene aggiunto un bucket gli unici oggetti da riassegnare sono quelli da assegnare a quel bucket, senza la possibilità che vengano assegnati a vecchi bucket, garantita se per ogni vista $V_1 \subseteq V_2 \subseteq B$ se $h_{V_2}(i) \in V_1$ allora $h_{V_1}(i) = h_{V_2}(i)$

Carico il carico di un bucket è pari al numero di oggetti assegnati, calcolata su un bucket b e su un insieme di view è pari a $\cup_{V_j} h^{-1}_{V_j}(b)$, dove $h^{-1}_{V_j}(b)$ è l'insieme di oggetti assegnati al bucket b nella vista V_j, idealmente deve essere un valore piccolo

Dispersione la dispersione di un oggetto è pari al numero di bucket dove esso è assegnato, ovvero calcolata su un insieme di view è pari a $|\{h_{V_1}(i), h_{V_2}(i), \ldots, h_{V_k}(i)\}|$, idealmente deve essere un valore piccolo

6.3 Hash Perfetto

Sebbene l'hashing sia utilizzato per le sue straordinarie performance di O(1) al caso medio, è possibile ottenere le stesse performance al caso pessimo, qualora l'insieme delle chiavi sia **statico**. Chiamiamo *perferct hashing* la tecnica che ci permette di ottenere questo risultato.

L'idea di base è semplice, usiamo due livelli di hashing con funzioni diverse prese dalla stessa classe di funzioni hash universali come nell'esempio 6.1; il primo livello funziona praticamente allo stesso modo dell'hashing normale, il secondo livello entra in gioco in caso di collissioni ad una generica posizione j. Invece di avere le liste di trabocco, creiamo una seconda tabella hash S_j relativa alla posizione j a cui è associata un'altra funzione, se abbiamo scelto con cura questa seconda funzione possiamo garantire l'assenza di collissioni.

Per garantire questo, occorre che la dimensione di S_j sia il quadrato del numero di chiavi che collidono in j; nonostante questo aumento è possibile, a patto di scegliere correttamente la prima funzione, mantenere l'occupazione in spazio in $O(n)$

Teorema 6.3.1. *Se noi memorizziamo n chiavi in una tabella hash di dimensioni $m = n^2$ usando una funzione h scelta a caso tra una classe di funzioni hash universali, la probabilità che avvenga una collisione è minore di* $\frac{1}{2}$

Dimostrazione Esistono $\binom{n}{2}$ coppie di chiavi che possono collidere, ognuna con probabilità $\frac{1}{m}$ dato che lavoriamo con funzioni hash universali. Indicando con C il numero di collissioni, con $m = n^2$ ci aspettiamo di avere

$$E[C] = \binom{n}{2}\frac{1}{n^2} = \frac{n^2 - n}{2} \cdot \frac{1}{n^2} < \frac{1}{2}$$

Applicando a questo risultato la disequazione di Markov con $t = 1$ otteniamo

$$P(C \geq t) \leq \frac{E[C]}{t} \Rightarrow P(C \geq 1) < \frac{1}{2}$$

□

Comunque nel teorema 6.3.1 abbiamo assunto $m = n^2$, possiamo rilassare questo vincolo usando $m = n$ come tabella primaria, e chiamando n_j e m_j il numero di collisioni con l'indice j e la tabella hash di secondo livello corrispondente, assumiamo $m_j = n_j^2$

6.4 Bloom Filter

Struttura dati nata all'inizio degli anni '70[3] per risolvere il problema di avere insiemi che rispondano in maniera efficiente (in tempo e spazio) a richieste di appartenenza di elementi. Il funzionamento è piuttosto semplice, per quanto riguarda la struttura abbiamo:

- l'insieme di elementi $S = \{x_1, \ldots, x_n\}$ di cardinalità n
- un array di m bit inizializzato a 0
- una famiglia di k funzioni hash $h_1, \ldots, h_k$ indipendenti ed uniformi, che generano indici in un intervallo $[1, m]$

Per ogni Bloom Filter vale la proprietà

$$\forall x \in S \; B[h_j(x)] = 1 \; j = 1, \ldots, k$$

[3] inventato da Burton Bloom nel 1970

ovvero, per ogni elemento presente nell'insieme, si va a settare ad 1 i bit dell'array costituente il Bloom Filter, aventi come indice il risultato dell'applicazione delle funzioni hash a quell'elemento.
Con questa costruzione è facile capire che se abbiamo un elemento x e vogliamo sapere se appartiene ad S, è sufficiente controllare se $h_j(x) = 1$ per ogni j, in tal caso x *probabilmente* appartiene ad S, mentre se troviamo anche una sola funzione hash che applicata ad x punta ad un bit 0, sappiamo che *sicuramente* x non appartiene ad S.
In effetti non abbiamo la certezza dell'appartenenza in quanto l'applicazioni delle funzioni hash possono andare a settare ad 1 un bit settato ad 1 precedentemente dall'applciazione delle funzioni ad un altro elemento. Il fatto che il Bloom Filter possa dare falsi positivi potrebbe renderlo poco affidabile (e difatti lo è), ma andando ad osservare meglio è possibile notare che la probabilità che un bit dell'array sia 0 è pari a 1 - la probabilità che il bit sia settato ad 1 (pari ad $\frac{1}{m}$) con una singola applicazione, elevato per il numero di funzioni hash applicate e il numero di elementi inseriti, riassumendo:

$$P(B_i = 0) = (1 - \frac{1}{m})^{kn} = e^{-\frac{kn}{m}}$$

quindi la probabilità di falsi positivi ϵ risulta essere

$$\epsilon = (1 - P(B_i = 0))^k = (1 - e^{-\frac{kn}{m}})^k$$

quindi, nel progettare la funzione hash, ci troviamo di fronte ad un *tradeoff* per k, all'aumentare delle funzioni hash infatti pur aumentando la densità di "1" nell'array, e quindi aumentando la possibilità di falsi positivi, si diminuisce la possibilità che per due elementi x e y si settino *gli stessi* bit dell'array.
Con un pò di calcoli possiamo osservare che ϵ si minimizza per

$$k = \frac{m}{n} \ln 2$$

Inoltre il Bloom Filter risulta molto efficiente[4] per piccole costanti c tale percui $m = cn$

Una nota interessante è che se uniamo due insiemi per cui sono stati separatamente calcolati i Bloom Filter aventi la stessa famiglia di funzioni hash, è possibile costruire il nuovo Bloom Filter semplicemente con l'OR bit a bit dei precedenti.

6.4.1 Spectral Bloom Filter

In questa variante abbiamo un multi-insieme che contiene gli elementi con una certa moltiplicità, invece dell'array di bit abbiamo un array di contatori, che rappresentano la moltiplicità degli oggetti, inizializzati a 0. Quando si inserisce un elemento x si incrementano di 1 i contatori C_i relativi alle posizioni fornite dalla famiglia di funzioni hash usate

$$C_{h_1(x)}, C_{h_2(x)}, \ldots, C_{h_k(x)} \quad \text{con} \quad k = \ln 2(\frac{m}{n})$$

L'eliminazione avviene similmente, riducendo di 1 i contatori relativi.
La ricerca, che deve restituire il numero di occorrenze di y nell'insieme, avviene nel modo classico, restituendo però tra tutti i valori possibili, il minore, ovvero

$$min\{C_{h_1(y)}, \ldots, C_{h_k(y)}\}$$

La ricerca ritornerà 0 qualora l'elemento non sia presente nell'insieme, e la probabilità di falso posivo, ivi inclusa la probabilità che ritorni un numero di occorrenze sovrastimato, è pari a quello del Bloom Filter

[4] Saar Cohen, Yossi Matias, *Spectral Bloom Filter*

Capitolo 7

Motori di ricerca

Lo scopo dei motori di ricerca è trovare i documenti rilevanti per la *query* definita dall'utente. I problemi sono del tipo

- Estrarre dati significativi è difficile in quanto il web cambia di continuo
- Rispondere correttamente a quello che gli utenti chiedono è difficile a causa di ambiguità del linguaggio e dal ristretto numero di risposte che un utente può considerare

Durante questi anni siamo passati attraverso diverse generazioni di motori di ricerca

1. l'importanza di una pagina era data dalla frequenza delle keyword rilevanti usate nella stessa
2. l'importanza di una pagina era data da come questa veniva riferita, dalle parole precedenti o successive che una pagina usava attorno al link alla pagina analizzata (*anchor text*)
3. nella terza generazione si cerca di analizzare cosa vuole l'utente, piuttosto che le parole usate nella query

7.1 Liste invertite

Un problema direttamente correlato alla questione, ma che ricorre in molte altre situzione è quello dell'indicizzazione. Abbiamo un insieme molto vasto di documenti e vogliamo effettuare efficientemente delle query su (insiemi di) parole contenute in essi, uno dei possibili approcci è quello delle *liste invertite*

Ogni documento D viene associato ad un identificatore univoco, per ogni parola contentuta in uno qualsiasi dei documenti viene fatta associare una lista degli identificatori in cui la lista compare.

In questo modo le ricerche richiedono una semplice scansione, e le ricerche multiple (su molti caratteri) richiedono solo un AND tra gli id trovati per una parola e quelli trovati per l'altra.

Per come è strutturata, le liste invertite sono composte da piccoli interi, quindi facilmente comprimibili (ad esempio con l'algoritmo visto in 5.5.1), per rendere ancora più efficiente questa compressione è possibile utilizzare la tecnica del *Gap-coding*. La tecnica consiste semplicemente di memorizzare in una lista invertita (ordinata) non l'id del documento, ma la differenza numerica tra l'id attuale e quello precedente nella lista.

Esempio Supponiamo di avere la parola "Brutus" che ricorre nei documenti 33, 47, 154, 159, 202, la lista con gap-coding corrispondente è la seguente:

Normale	Brutus:	33	47	154	159	202
Gap-coding	Brutus:	33	14	107	5	43

7.2 Pagerank

Tecnica di paging introdotta dai fondatori di google[1], dove si assegna ad ogni pagina una rilevanza che non tiene di conto del suo contenuto testuale nè dell'interrogazione posta dall'utente.

Nel Pagerank si ordinano le pagine in funzione del numero e della *qualità* dei link che puntano alla pagina stessa. Più precisamente la popolarità di una pagina p è espressa da un *rank* $R(p)$ calcolato come la probabilità che un utente raggiunga p camminando a caso sulla rete, ovvero prendendo con uguale probabilità uno dei link presenti nella pagina attualmente visitata. Definendo come $p_1, \ldots, p_k$ le pagine che hanno almeno un link verso p e $N(p_i)$ il numero di pagine puntate da ogni p_i, la formula di base per il calcolo di $R(p)$ è:

$$R(p) = \sum_{i=1,\ldots,k} \frac{R(p_i)}{N(p_i)}$$

in pratica è come se ogni pagina assegnasse una parte uguale del suo rank a tutte le pagine raggiunte dai link uscenti, le quali usano questa parte per formare il loro rank, definito come la somma dei valori degli archi entranti (del grafo Web).
Questa formula non tiene in considerazione di quelle pagine "alla frontiera" del Web, ovvero quelle pagine che non hanno archi entranti e/o archi uscenti, per questo si preferisce usare quest'altra formula leggermente più complessa:

$$R(p) = d \sum_{i=1,\ldots,k} \frac{R(p_i)}{N(p_i)} + \frac{1-d}{n} \tag{7.1}$$

con n il numero di pagine raccolte dal *crawler*[2] ed indicizzate, e d la probabilità di proseguire nella catena di link, in genere questo è un valore arbitrario $0 \leq d \leq 1$, al cui variare varia sensibilmente il modo in cui si valutano le pagine, per esempio con $d = 0$ tutte le pagine avrebbero la stessa rilevanza $\frac{1}{n}$, viceversa con $d = 1$ la rilevanza dipenderebbe interamente dalla struttura del grafo del Web; in genere si usa il compromesso di porre $d = 0.85$.

Per calcolare effettivamente l'equazione 7.1 si utilizza una serie di calcoli matriciali tanto semplici nella loro descrizione, quanto complessi nel loro svolgimento in quanto prendono in esame matrici di enormi dimensioni. Considerando W la matrice di adiacenza del grafo del Web, le sue successive potenze W^k indicano i percorsi di k passi sul grafo, possiamo a questo punto indicare con la matrice Z di dimensioni nxn che indica la probabilità che un utente i vada nella pagina j, con le sue successive potenze Z^k che stanno ad indicare che si svolgono percorsi di k passi

$$Z[i,j] = d \cdot W[j,i] + \frac{1-d}{n} \tag{7.2}$$

Inoltre possiamo rappresentare i rank delle n pagine come un vettore R di n elementi, a cui vengono fatti associare k vettori, che rappresentano i rank calcolati fino al k-esimo passo, nel seguente modo

$$\begin{aligned} R_1 &= Z \cdot R_0 \\ R_2 &= Z \cdot R_1 = Z^2 \cdot R_0 \\ &\ldots \\ R_k &= Z^k \cdot R_0 \end{aligned}$$

Essendo la formula 7.1 essenzialmente, una catena di Markov ergodica[3], si può assumere che il calcolo finale del pagerank non sarà dipendente dal valore iniziale R_0

[1] Larry Page e Sergey Brin
[2] strumento software che si occupa di esplorare il web per raccogliere informazioni sulle pagine presenti
[3] aperiodica ed irriducibile

7.3 HITS

La tecnica chiamata HITS (*Hyperlink Induced Topic Search*) è stata sviluppata dalla IBM[4], e per assegnare la rilevanza ad una pagina, fa uso di un sottografo del Web costruito in base alla richiesta dell'utente.

Per una data interrogazione q, si recuperano tutte le pagine che contengono tutti i termini di q e si mettono nell'insieme P, a queste si aggiungono le pagine che puntano a P o sono puntate da P, chiamiamo questo insieme *base*, il quale contiene tutte le pagine correlate direttamente o indirettamente a q. Si costruisce il sottografo che ha per nodi le pagine della base e come archi i link tra essi, si calcolano per questi nodi due attributi, definiti

autority $A(p)$ che indica l'autorevolezza della pagina p relativa all'interrogazione q

hubness $H(p)$ che indica la quantità di riferimenti a siti autorevoli in p, in relazione a q

Il significato rispecchia quindi il senso comune, in quanto un sito è una buona rassegna (*hubness*) se raccoglie molti link a siti autorevoli, viceversa un sito è autorevole se è riferito da buone rassegne. Definendo $z_1, \ldots, z_k$ le pagine che puntano a p e con $y_1, \ldots, y_h$ le pagine puntate da p possiamo derivare le seguenti formule

$$A(p) = \sum_{i=1,\ldots,k} H(z_i)$$
$$H(p) = \sum_{i=1,\ldots,h} A(y_i)$$

Per risolvere le equazioni, similmente al PageRank (vedi 7.2), si definisce la matrice di adiacenza B del sottografo indotto dalla base e si calcolano i vettori A ed H impiegando l'algebra delle matrici partendo da B ed impostando opportuni valori iniziali di A ed H.

Il vantaggio di un simile approccio è che le matrici in gioco sono nettamente più piccole del PageRank, lo svantaggio è che il tutto dipende dalla specifica query dell'utente, e non può essere calcolato "a priorì'. Inoltre, sebbene sia migliore del PageRank per calcolare la *rilevanza* delle pagine, questo lo rende più esposto a *spam*, ovvero a sistemi atti a falsare il calcolo per fini di lucro o di notorietà.

[4] introdotta da Jon Kleinberg

Parte III

Stringologia

Capitolo 8

Pattern matching esatto

La stringologia è un ramo dell'informatica che si occupa di manipolare stringhe di dati allo scopo di rispondere a richieste di *pattern matching* ovvero di ricercare le occorrenze di una o più stringhe date su un preciso testo di un alfabeto Σ. Definiamo

T Testo su cui effettuare le ricerche

m lunghezza del testo

P insieme di pattern $P = \{P_1, \ldots, P_z\}$ di lunghezza totale pari ad n

Inoltre, presa una generica stringa S, consideriamo $S[i, j]$ come sottostringa di S e per $i < j$ definiamo

$$S[i, |S|] \text{ suffisso di } S$$
$$S[1, j] \text{ prefisso di } S$$

prendiamo inoltre una stringa $\omega \in \Sigma^+$ sottostringa di S, definamo

bordo di ω, come la più lunga sottostringa propria, ovvero diversa da ω, che sia contemporaneamente prefisso e suffisso di ω

Esempio Data la stringa ACAGACA abbiamo che:

$$\text{bordo(ACAGACA)=ACA}$$

In seguito verranno mostrati gli approcci più classici al problema, per il dettaglio dei costi computazionali fare riferimento alla tabella A.2 nell'appendice.

8.1 Metodo Naive

Il metodo brute force è senz'altro l'approccio più semplice per trovare le occorrenze di un pattern lungo n in un testo lungo m, ed esegue i seguenti passi

1. allinea il pattern con l'inizio della stringa T
2. confronta gli elementi del pattern, da sinistra a destra, con quelli di T
3. se trova un carattere diverso in posizione i del pattern, interrompe la scansione, esegue uno shift destro del pattern di una posizione, e riprende a confrontare dal primo elemento del pattern dal punto 2
4. se si finiscono i caratteri del pattern senza trovare *mismatch*, si salva la posizione dell'occorrenza trovata, si shifta il pattern di una posizione, e si riprende dal punto 2
5. si termina quando è finita la stringa T da analizzare

E' facile osservare che il metodo naive ha complessità $O(n \cdot m)$, vedremo nelle prossime sezioni degli algoritmi migliori per risolvere il problema.

8.2 Knuth-Morriss-Pratt

Questo algoritmo presenta alcune migliorie rispetto all'algoritmo brute force, ed opera seguendo precisi criteri

- effettua un preprocessing sul pattern, in modo da memorizzarsi le lunghezze dei bordi di tutte le sottostringhe formate da prefissi propri di p, su un array B, in $O(m)$
- Esegue una scansione lineare da sinistra a destra come nell'algoritmo brute force
- In caso di mismatch si esegue lo shift allineando alla posizione di mismatch i, la fine del prefisso costituente il bordo più lungo della stringa precedente ad i nel pattern, nel caso in cui il carattere che ha causato il mismatch è uguale all'ultimo carattere del suddetto bordo, si prende invece un bordo più piccolo (se esiste).

Questo funziona in quanto una volta ottenuto un mismatch ad un carattere nel pattern i, tutti i caratteri che precedono i nel pattern sono uguali ai corrispettivi della parte del testo che stiamo analizzando, dunque nel valutare il salto noi dobbiamo essere sicuri che quello che precede i nel testo, non sia un prefisso di p. Ma essendo le sottostringhe del testo e del pattern fin qui analizzate, uguali, quello che ci stiamo chiedendo è se esiste un bordo nella sottostringa precedente ad i nel pattern, in tal caso il salto deve essere fatto alla posizione indicata da $B[i]$, altrimenti possiamo eseguire lo shift di i posizioni, allineando quindi l'inizio ($B[i] = 0$) del pattern con il punto di mismatch nel testo.

Esempio Abbiamo un testo ed un pattern di questo tipo:

T=ABCXABCYABCD
P=ABCD

Da qui abbiamo i seguenti confronti e shift

1	2	3	4	5	6	7	8	9	10	11	12
A	B	C	X	A	B	C	Y	A	B	C	D
Â	B̂	Ĉ	Ḋ								
			Ȧ	B	C	D					
				Â	B̂	Ĉ	Ḋ				
							Ȧ	B	C	D	
								Â	B̂	Ĉ	D̂

La lista degli shift risultanti (fornita come indice nel testo a cui si allinea il primo elemento del pattern) la seguente:
$Sh = \{1, 4, 5, 8, 9\}$ Mentre questa è la lista dei confronti $(I_{testo}, I_{pattern})$:
$C = (1,1), (2,2), (3,3), (4,4), (4,1), (5,1), (6,2), (7,3), (8,4), (8,1), (9,1), (10,2), (11,3), (12,4)$

Preprocessing

Il preprocessing viene eseguito usando un array B dove, per ogni predecessore di ogni carattere, viene memorizzato l'indice del salto *sicuro* più lungo da effettuare nel caso di un mismatch con il carattere stesso.
Ovvero, per ogni elemento j di B, si mette il valore del bordo più lungo della stringa $p(0, j-1)$ che precede il carattere j nel pattern.

Esempio Per un pattern $p = ababaa\$$ l'array B sarà composto da:

j	=	0	1	2	3	4	5	6
p[j]	=	a	b	a	b	a	a	$
B[j]	=	-1	0	0	1	2	3	1

8.3 Boyer-Moore

L'algoritmo di Boyer-Moore rappresenta una piccola evoluzione che, sebbene sia sempre di complessità lineare al caso pessimo, presenta in pratica un comportamento *sublineare* al caso medio.
Esso si basa sempre sul sistema matching/shifting, ma fa utilizzo combinato di 3 idee importanti:

R-L scanning una volta che il pattern è stato allineato, esegue una scansione dei caratteri da destra a sinistra, questo perchè, in caso di mismatch nelle prime posizioni, è possibile, sotto certe condizioni, eseguire uno shift di buona parte del pattern

Bad Character una volta trovato un mismatch con un carattere $x \in T$, si esegue uno shift tale da far corrispondere la posizione di mismatch con il carattere $x \in P$ più a destra nel pattern

Buon Suffisso estende la regola precedente, controllando se esiste una stringa in P che corrisponde al suffisso finora controllato (e riconosciuto) del pattern

Le ultime due regole richiedono un preprocessing del pattern.

8.4 Albero dei suffissi

Finora gli algoritmi che abbiamo visto utilizzano un approccio intuitivo al problema che al massimo sfrutta delle informazioni date dai pattern per velocizzare la ricerca; questa struttura dati, invece, esegue un preprocessing *sul testo* in $O(m)$ per poter rispondere a richieste di matching di un pattern P di lunghezza n in $O(n)$ ovvero in tempo lineare rispetto alla lunghezza *del pattern*. L'albero dei suffissi è una struttura dati che data una stringa T in ingresso di lunghezza m con un alfabeto Σ, la rielabora per mettere in luce alcune sue proprietà interessanti per la ricerca di sue sottostringhe.
Un albero dei suffissi è un albero radicato che gode delle seguenti proprietà:

1. ci sono esattamente m foglie
2. ogni nodo interno, esclusa la radice, ha almeno 2 figli
3. ogni arco è etichettato da una sottostringa $T' \subset T$
4. due archi uscenti dallo stesso nodo hanno caratteri iniziali delle due etichette diversi
5. la concatenazione delle etichette lungo il cammino radice-foglia è il suffisso $T(i, m)$ di lunghezza $m - i + 1$

Inoltre, in genere ogni arco che termina in una foglia ha come carattere il \$ che segna la fine della stringa.

In pratica si costruisce prendendo la stringa T e creando il primo nodo che parte dalla radice e comprende come etichetta tutto T; poi si prende in considerazione la sottostringa costituita da T meno il primo carattere e si valuta se ha prefissi in comune con le etichette dei nodi già inseriti (in questo caso solo T), in caso affermativo si divide in due il nodo in questione lasciando come ramo radice-nodo l'etichetta corrispondente al prefisso in questione, e come etichette dei due nodi le rimanenti parti delle rispettive stringhe.
La ricerca di un pattern P è effettuata semplicemente scorrendo il pattern e facendo corrispondere i caratteri alle etichette dell'albero dei suffissi, il tutto in $O(n)$.

Naturalmente quello che viene guadagnato in tempo in questo modo, viene pagato in spazio occupando $\Theta(m|\Sigma|)$

8.5 Array dei suffissi

L'array dei suffissi di una stringa S è l'array degli indici dei suffissi in S, ordinati lessicograficamente. Naturalmente è possibile utilizzare uno spazio molto minore per conservare questo array rispetto alla corrispettiva forma ad albero (vedi Sezione 8.4), del resto è abbastanza intuitivo come si possa ottenere l'array partendo dall'albero dei suffissi, in tempo lineare.
Il vantaggio di usare questa struttura è la possibilità di effettuare, similmente all'albero dei suffissi, ricerche dicotomiche con un impiego in tempo di $O(n + \log_2 m)$.

Esempio Considerando la stringa

a	b	r	a	c	a	d	a	b	r	a	$
1	2	3	4	5	6	7	8	9	10	11	12

i suffissi ordinati lessicograficamente, con i relativi indici della stringa di partenza, sono

Indice	Suffisso Ordinato
12	$
11	a$
8	abra$
1	abracadabra$
4	acadabra$
6	adabra$
9	bra$
2	bracadabra$
5	cadabra$
7	dabra$
10	ra$
3	racadabra$

L'array dei suffissi diventa quindi la colonna degli indici

Capitolo 9

Pattern matching su insiemi di pattern

Un'importante generalizzazione del problema del matching esatto è di trovare le occorrenze di un *insieme* di pattern $P = \{P_1, \ldots, P_z\}$, di lunghezza complessiva n, all'interno del testo T. Ovviamente è possibile risolvere il problema applicando z volte uno degli algoritmi visti in precedenza, con una complessità di $O(n + zm)$; del resto alcuni degli algoritmi fin qui presentati con pochi adattamenti ottengono buone prestazioni anche in presenza di diversi pattern, ma esistono modi migliori che, lavorando sui pattern, riescono a trovare le corrispondenze in minor tempo.

9.1 Aho-Corasick

Uno dei più importanti algoritmi che trattano questo problema è senz'altro quello di Aho-Corasick che fa uso di una struttura dati particolare, ovvero il *keyword tree*

Definizione Il keyword tree è un albero radicato K che soddisfa 3 condizioni:

1. ogni arco è etichettato con un carattere, e uno soltanto
2. ogni arco uscente da uno stesso nodo ha etichette diverse
3. ogni pattern $P_i \in P$ corrisponde ad un qualche percorso dalla radice a un nodo v, ottenuto concatenando i caratteri sugli archi del percorso
4. per ogni foglia $v \in K$ esiste un pattern che corrisponde alla concatenazione delle etichette dalla radice di K a v

E' facile osservare che ogni nodo nel keyword tree corrisponde al prefisso di uno dei pattern in P e viceversa. Avendo un alfabeto finito su cui lavorare è possibile costruire l'albero in $O(n)$.

Con questo approccio è possibile eseguire il matching esatto semplicemente scorrendo l'albero mano a mano che si leggono i caratteri da T, una volta raggiunta una foglia segnaliamo il pattern relativo come riconosciuto alla posizione corrente e proseguiamo dall'inizio. Il tutto viene fatto in $O(nm)$.

Problema del dizionario

Un problema noto, degno di menzione in questo contesto, è quello del dizionario.
Abbiamo un insieme di pattern P decisamente ampio, ed un'unica stringa T piuttosto corta. Il problema è di stabilire se la stringa T corrisponde **esattamente** ad uno dei pattern P.

Questo problema è abbastanza speculare a quello finora affrontato, la differenza sta nella ricerca *nei* pattern usando il testo T e non viceversa. Questo è un classico problema in cui l'approccio keyword tree risulta utile, ed è una delle applicazioni in cui viene maggiormente usato.

9.1.1 Failure Link

Per velocizzare la ricerca estendiamo la struttura del keyword tree con i cosidetti *failure link*. Essi partono dall'assunzione abbastanza debole che non esistano pattern che sono prefissi propri di altri pattern, e per definirli usiamo alcune definizioni ausiliarie:

L(v) rappresenta la concatenazione delle etichette dalla radice al nodo v

lp(v) rappresenta la lunghezza del più lungo suffisso proprio di $L(v)$ che è prefisso di qualche pattern in P

f(v) rappresenta l'unico nodo in K con il cui percorso radice-nodo è possibile comporre il suffisso di $L(v)$ lungo $lp(v)$

failure link arco diretto $(v, f(v))$

Ovvero, per ogni nodo $a \in K$ si guarda la stringa che compone e si effettua un collegamento con il nodo $b \in K$ di un pattern, la cui stringa che compone è il più lungo prefisso possibile della stringa radice-nodo di a.
Se non esistono prefissi di pattern corrispondenti ad un suffisso del nodo, il failure link è quello (nodo, radice).

La ricerca usando i failure link permette di non ripartire da 0 a considerare i pattern, in caso di mismatch, ma di procedere per controllare se il match può avvenire con altri pattern, senza però ricalcolarisi match già eseguiti, e contestualmente scartando i pattern per cui si avrebbe sicuramente mismatch. Questo porta la complessità della ricerca ad $O(n + z + m)$

9.2 Karp-Rabin

E' un semplice algoritmo randomizzato, utile in quanto tende ad avere un andamento lineare in tutti i casi pratici, anche se al caso pessimo va come $\Theta(nm)$. Sebbene possa essere utilizzato come exact matching di un solo pattern, risulta molto pratico per effettuare i matching multipli su un insieme di pattern.

L'idea di base è di trasformare il pattern in un valore numerico mediante una funzione hash (vedi capitolo 6), e confrontarlo con le sottostringhe di T anchesse convertite con la medesima funzione hash. Definendo s_i come la sottostringa in T di lunghezza n che parte dalla posizione i, i controlli vengono fatti semplicemente confrontando $h(p) == h(s_i)$, per ogni $0 \leq i < m$ e segnalando l'eventuale match. Siccome il valore di ogni blocco così valutato dipende da quello precedente, è possibile effettuare una scansione lineare del testo al caso medio.

Dato che lavoriamo con funzioni hash è possibile che ci siano dei falsi positivi, per questo è necessario confrontare le occorrenze trovate col pattern, in maniera classica, prima di restituirle.

Parte IV

Strutture di dati evolute

Capitolo 10

Algoritmi on-line

Sono algoritmi che rispondono ad una sequenza di richieste

$$\sigma = \sigma(1), \ldots, \sigma(m)$$

non nota a priori.
Definendo A un generico algoritmo on-line che risolve un problema, e OPT l'algoritmo ottimo che risolve lo stesso problema, ovvero che genera il minimo numero di fault per la stessa sequenza σ di richieste, definiamo

$$F_A(\sigma)$$

come il numero di fault generati dall'algoritmo A con la sequenza σ

Definizione Un algorimo di paging on-line A si dice k-competitivo se esiste una costante a tale percui

$$F_A(\sigma) \leq k \cdot F_{OPT}(\sigma) + a$$

Questa analisi, già accennata nel Cap.3, risulta essere in questo caso molto potente per misurare la performance di un certo algoritmo online, in quanto confrontato con un algoritmo che conosce la sequenza σ di richieste a priori.

10.1 Move To Front

Un algoritmo on-line tipico è il *Move To Front* (MTF) che lavora su una lista di n elementi, sulla quale vengono eseguite operazioni di ricerca. Nel caso semplice, ogni richiesta pagherà al caso pessimo $O(n)$.

La variante MTF si occupa, una volta ricevuta una richiesta, di spostare in cima alla lista l'elemento cercato.
Ricordiamo che per essere k-competitivo, l'algoritmo MTF confrontato con l'algoritmo ottimo deve avere un costo pari a

$$costo(MTF) \leq k \cdot \text{costo}(OPT) + a$$

Definiamo

C(j) costo richiesta $\sigma(j)$ con MTF, misurata come il numero di aggiornamenti del puntatore della lista

inversione di due elementi della lista (e_1, e_2) quando nelle liste composte dai due metodi, il MTF e OPT, i due elementi risultano con ordine relativo di occorrenza diverso nelle due liste, ovvero se in una abbiamo che e_1 precede e_2, e nella seconda avviene che e_2 precede e_1, abbiamo un'inversione

Definendo inoltre Φ_j come il numero di inversioni dopo j richieste, abbiamo che:

Teorema 10.1.1. *Se $m > n^2$ l'algoritmo MTF è 2-competitivo*

Dimostrazione Abbiamo

$\sum_{j=0}^{m} C(j) \leq 2 \sum_{j=0}^{m} C`(j) + O(n^2)$

e siccome

$0 \leq \Phi_j \leq \frac{n(n-1)}{2}$

usando una tecnica chiamata *telescoping*, abbiamo che:

$C(j) + \Phi_j - \Phi_{j-1} \leq 2 \cdot C`(j)$

$C(0) + \cancel{\Phi_0} - \Phi_{-1} + C(1) + \cancel{\Phi_1} - \cancel{\Phi_0} + C(2) + \Phi_2 - \cancel{\Phi_1} + \cdots + \cancel{\Phi_{m-1}} + C(m) + \Phi_m - \cancel{\Phi_{m-1}}$

$\sum_{j=0}^{m} C(j) + \Phi_m - \Phi_{-1} \leq 2 \sum_{j=0}^{m} C'(j)$

con un costo complessivo pari a

$$costo(MTF) \leq 2 \cdot costo(OPT) + O(n^2)$$

se il numero di richieste di σ è maggiore del valore asintotico di cui sopra, ovvero se

$$m > n^2$$

il valore asisntotico può essere trascurabile, quindi l'algoritmo risulta essere 2-competitivo. □

Capitolo 11

Problema del Paging

Abbiamo due livelli di memoria, il problema è **minimizzare il numero di page fault** su σ richieste. In questo problema risulta cruciale la scelta delle pagine da deallocare, abbiamo quindi che l'algoritmo ottimo OPT, con k pagine che possono contemporaneamente risiedere in memoria, minimizza il numero di fault complessivo. Un esempio dell'algoritmo ottimo è dato da

MIN algoritmo che in caso di fault, dealloca la pagina che sarà richiesta più in là nel tempo (più lontana nella sequenza σ)

Teorema 11.0.2. *Dato A un algoritmo on-line di paging deterministico che opera su k pagine in memoria*

$$\textit{Se } A \textit{ è } c\textit{-competitivo, allora } c \geq k$$

Tutti gli algoritmi che andremo ad analizzare saranno confrontati con OPT per avere un'analisi competitiva della loro qualità, la tabella riassuntiva A.1 è consultabile nell'appendice

11.1 Paging semplice

Cadono in questa sezione tutti gli algoritmi che utilizzano euristiche note ed usate in moltissimi altri ambiti, e che costituiscono il modo più semplice di approcciarsi al problema

11.1.1 Least Recently Used

Per ogni richiesta, l'algoritmo dealloca la pagina richiesta meno di recente.

Teorema 11.1.1. LRU *è k-competitivo, ovvero*

$$F_{LRU}(\sigma) \leq k \cdot F_{OPT}(\sigma)$$

11.1.2 First-in First-out

Nell'algoritmo $FIFO$, per ogni richiesta, l'algoritmo dealloca la pagina inserita meno di recente.

Teorema 11.1.2. FIFO *è k-competitivo, ovvero*

$$F_{FIFO}(\sigma) \leq k \cdot F_{OPT}(\sigma)$$

11.1.3 Least Frequently Used

Nell'algoritmo *LFU*, per ogni richiesta, l'algoritmo dealloca la pagina usata meno di frequente. A questo proposito è necessario mantenere, in qualche modo, traccia del numero di volte che la pagina è acceduta.

Teorema 11.1.3. *L'algoritmo LFU* ***non*** *è k-competitivo per nessuna costante k*

11.2 Paging Randomizzato

11.2.1 Random

L'algoritmo più semplice di tutti per cui, per ogni richiesta, dealloca una pagina a caso.

Teorema 11.2.1. *Random è al massimo k-competitivo, ovvero*

$$F_{RAND}(\sigma) \leq k \cdot F_{OPT}(\sigma)$$

11.2.2 Marking

L'algoritmo di marking è lievemente meno intuitivo dei precedenti. Trattasi un'evoluzione del *LRU* e lavora marcando le pagine che sono state richieste recentemente.
Per valutare quale pagina deallocare utilizza i seguenti criteri

1. Si comincia inizializzando tutte le pagine come non marcate
2. in caso di fault dealloca una pagina **non marcata** a caso
3. in caso di inserimento le pagine vengono inserite marcate
4. in caso di richiesta di pagina, essa viene marcata
5. se sono tutte marcate si smarcano e si comincia una nuova fase

Il processo è quindi diviso in fasi, ognuna termina col passo 5 e fa cominciare quella successiva, dove per ognuna è possibile associare un attributo ad ogni pagina, che stabilisce, per quella fase, il suo ruolo nell'algoritmo.
Ad ogni fase, ogni pagina può essere considerata:

marked ovvero marcata in questa fase

clean ovvero che non è stata marcata nè in questa fase, nè in quella precedente

steal ovvero che non è stata marcata in questa fase, ma è stata marcata in quella precedente

Teorema 11.2.2. *L'algoritmo di Marking è* $2H_k$*-competitivo, dove*

$$H_k = \sum_{i=1}^{k} \frac{1}{i} \simeq \ln k$$

ovvero è il k-esimo elemento della serie armonica

11.3 Paging con località di riferimenti

In questa sezione porremo l'attenzione su un fatto pratico, ovvero che generalmente σ non viene generata in maniera del tutto arbitraria.

Di fatto la sequenza di richieste generata da problemi concreti, segue un criterio di *località di riferimenti*, ovvero una pagina richiesta in un certo momento sarà probabilmente richiesta di nuovo in tempi brevi.

11.3.1 Dynamic Access Graph

Un buon algoritmo per sfruttare questa euristica è il *Dynamic Access Graph* (DAG), ovvero un grafo di accessi, dove i nodi rappresentano le pagine.
L'euristica si basa sul principio che se una pagina viene acceduta (un nodo viene selezionato), la prossima pagina (nodo) ad essere richiesta sarà una adiacente nel relativo grafo di accessi.
Nel descrivere nel dettaglio l'algoritmo definiamo le seguenti notazioni

V : insieme di nodi su cui opera il DAG

E : insieme di archi su cui opera il DAG

$w(e)$: peso dell'arco $e \in E$

k : numero di pagine che possono risiedere in memoria

P : puntatore alla pagina corrente

α, β, γ : pesi opportuni (generalmente usati $\alpha = 0.8, \beta = 1.5, \gamma = 10$)

Per l'inizializzazione, considerando g come la pagina che è richiesta per prima, abbiamo che:

$$V = \{g\}$$
$$E = \emptyset$$
$$P = v$$

con v il puntatore a g.
Considerando un istante arbitrario in cui viene effettuata una richiesta, chiamando x la pagina puntata da P ed y con $y \neq x$ la pagina richiesta attualmente, si eseguono i seguenti passi:

1. Aggiungere y a V se non già presente
2. Sia $\epsilon = (x, y)$
3. Se esiste già l'arco ϵ allora si aggiunge un prodotto α al suo peso $w(\epsilon) = \alpha \cdot w(\epsilon)$, altrimenti si aggiunge agli archi E e si setta il suo peso $w(\epsilon) = 1$
4. Se $w(\epsilon) > 1$ allora si setta $w(\epsilon) = 1$
5. Ogni γ richieste complessive, si aumenta il peso di ogni arco di un fattore β ovvero $\forall \epsilon \in E \; w(\epsilon) = \beta \cdot w(\epsilon)$
6. Se al momento della richiesta, non c'è spazio libero e y non è in memoria, si elimina la pagina di distanza massima da y calcolata con i relativi pesi .
7. Si aggiunge y in memoria.
8. Si setta $P = y$

E' facile notare che se, in un momento successivo, la richiesta per la pagina x è immediatamente seguita dalla richiesta della pagina y, l'arco (x, y) pesa al massimo 1. Il peso dell'arco si riduce all'aumentare delle richieste di x seguite da quelle di y.
Viceversa se le pagine x e y non sono richieste in successione per diverso tempo, il peso del loro arco aumenta.

E' possibile usare una variante che impedisce che il peso di un arco scenda sotto $\frac{1}{k}$.
Di questa variante esiste un teorema per la sua analisi competitiva:

Teorema 11.3.1. *L'algoritmo DAG modificato è $O(k \log k)$-competitivo*

Capitolo 12

Problema del Web-Caching

Con l'aumentare dell'interesse verso il Web la percentuale di traffico generato da richieste HTTP è aumentata considerevolmente, da qui la necessità di conservarsi in qualche zona intermedia tra il client ed il server, quelle pagine che vengono richiamate più spesso, realizzando così un vero e proprio Web-Caching.
Di fatto è possibile estendere la gerarchia di memoria comunemente nota[1] inserendo il web come memoria esterna, quindi il Web-caching altro non è che una cache che si frappone tra la memoria secondaria, rappresentata dal server, e quella primaria del client che manda le richieste.

12.1 Greedy Dual

Questo algoritmo implementa una tecnica efficiente per il problema del Web-caching, e si occupa del caso in cui le pagine in cache hanno la stessa dimensione ma costo di riperimento diverso.

- Ad ogni pagina p in cache viene assegnato un valore $H(p)$, rappresentante il costo di p
- Quando p viene caricata in cache, $H(p)$ assume come valore il costo del reperimento di p in memoria secondaria (il Web)
- Quando è necessario rimpiazzare una pagina per mancanza di spazio, si elimina la pagina di $H(p)$ minore (di valore min_H), ed ogni altra pagina riduce il suo H di min_H
- quando una pagina viene acceduta, si ripristina il suo $H(p)$ al suo costo originario

Avendo k pagine in cache, sia il fault che l'hit (reperimento di pagina già in memoria) si eseguono con $O(\log k)$ operazioni in più del necessario

Teorema 12.1.1. *Greedy Dual è s-competitivo, dove s è il rapporto tra la dimensione della cache e la dimensione della pagina più piccola*

12.1.1 Greedy Dual-Size

La variante GdS utilizza per la stima dei costi, il rapporto $\frac{costo}{size}$ per realizzare la funzione $H(p)$, dove per *costo* si intende il costo di reperimento in memoria secondaria della pagina, e la *size* si riferisce alla dimensione in bit della pagina stessa. E' provabile che GdS è ottimo

[1] Quella interna al computer, costituente la gerarchia Registri/Cache/Memoria Principale(RAM)/Memoria Secondaria (Disco Fisso)

Parte V

Memorie gerarchiche

Capitolo 13

Modelli di computazione

Uno dei problemi più attuali è relativo alla gestione dell'immensa quantità di dati presenti nel modo. Essi vengono generati da sensori, statistiche demografiche o di mercato, telecamere, e tutto ciò che può venire digitalizzato.
Per affrontare e risolvere parzialmente il problema, i computer *general-pourpose* hanno al loro interno una gerarchia di memoria, la quale permette di contenere una grande quantità di dati ai livelli superiori, e di trasferirli ai livelli inferiori, meno capienti ma maggiormente veloci.
Un buon algoritmo può sfruttare questo fatto per aumentare la sua efficienza, utilizzando al massimo i dati presenti a questi livelli, minimizzando così i trasferimenti dai livelli superiori. Questi prendono il nome di algoritmi **EMM** (*External Memory Model*).

Per valutare gli algoritmi EMM, si usano una serie di parametri:

N numero di oggetti del problema

M numero di oggetti memorizzabili (size) della memoria interna

B numero di oggetti contemporaneamente trasferibili (con un solo accesso) dalla memoria esterna a quella interna (block size)

D numero di dischi indipendenti usati

P numero di CPU usate

Q numero di richieste (queries)

Z dimensione della risposta, fornita in numero di oggetti

n numero di blocchi del problema, ovvero il numero di trasferimenti necessari dalla memoria secondaria a quella primaria per trasferire tutti gli N dati del problema

m numero di blocchi memorizzabili in memoria interna

z numero di blocchi scritti in memoria relativi alla risposta

Da queste definizioni è possibile evincere le seguenti formule e vincoli:

1. $M < N$
2. $1 \leq DB \leq \frac{M}{2}$
3. $n = \frac{N}{B}$
4. $m = \frac{M}{B}$

5. $z = \dfrac{Z}{B}$

Le operazioni da eseguire nel modello EMM possono essere divise all'interno delle seguenti classi di problemi:

Scanning viene effettuata su un file di N oggetti, del quale viene eseguita una lettura o una scrittura

Sorting viene effettuata su un file di N oggetti, i quali vengono ordinati

Searching esegue una ricerca tra N oggetti ordinati

Outputting fornisce Z oggetti di una risposta ad una richiesta

A queste operazioni sono associate delle complessità che definiscono un bound sul numero di operazioni di I/O richieste.

Operazione	**I/O Bound**
Scan(N)	$\Theta(\frac{N}{B}) = \Theta(n)$
Sort(N)	$\Theta(n \log_m n)$
Search(N)	$\Theta(\log_B N)$
Output(Z)	$\Theta(max\{1, z\})$

Lo Scan ha una complessità lineare in quanto richiede una operazione di lettura per ogni blocco trasferito. Il Sort esegue il sorting di n elementi (trasferimento in blocchi) con m oggetti memorizzabili, mentre il Search esegue una ricerca B-aria su N elementi. Per quanto riguarda l'Output la risposta può essere immediata o richiedere z trasferimenti.

Oltre a queste operazioni base esiste anche la **permutazione** di N oggetti in memoria esterna, il cui caso peggiore è uguale a quello medio e richiede

$$\Theta(min\{N, sort(N)\})$$

Visto che, essendo la permutazione un caso particolare di ordinamento, è necessario eseguire un *sort* o, in alternativa, prendere un elemento alla volta senza usare i blocchi nel caso in cui $B \log m = O(\log n)$

13.1 Sorting

13.1.1 Distribution Sort

Usa un insieme di $S - 1$ *bucket* che partizionano l'insieme di oggetti

$$e_1, e_2, \ldots, e_{S-1}$$

con la caratteristica che tutti gli oggetti contenuti in un bucket sono maggiori di quelli del bucket precedente. Per convenzione si definisce:

$$e_0 = -\infty$$
$$e_S = +\infty$$

Il funzionamento è il seguente:

1. Si prendono gli N elementi uno alla volta e si inseriscono nei rispettivi $S - 1$ bucket delimitati precedentemente da elementi presi a campione
2. Si itera l'operazione precedente sugli $S - 1$ bucket generando altrettanti bucket per ognuno di quelli correnti

3. l'algoritmo termina quando i bucket contengono un solo elemento

Con questo sistema si generano al massimo $\Theta(\log_S n)$ livelli di separazione dei bucket in bucket più piccoli.

Il problema principale di questo algoritmo è la scelta del numero di bucket da usare, e degli elementi cardine in base ai quali effettuare la separazione degli elementi di input (e successivamente dei sotto-bucket).
Come numero complessivo di bucket da usare si prende:

$$S = \Theta(min\{m, \frac{n}{m}\})$$

Per decidere gli elementi partizionanti si prendono $S \log S$ oggetti di input a caso, si ordinano, e si prende un elemento ogni $\log S$ così da avere complessivamente S elementi partizionanti. Questo si esegue in:

$$O(S \log S + Sort(S \log S)) = O(\sqrt{n} \log^2 n) = O(n) \text{ I/O}$$

quindi con un numero di accessi trascurabile.

13.1.2 Merge Sort

Un'altro algoritmo che permette di risolvere lo stesso problema è il Merge Sort.

1. Si creano diversi cluster, grandi $\frac{N}{M}$, di elementi ordinati
2. Si fondono insieme i cluster m a volta.

Abbiamo quindi $O(\log_{\frac{M}{B}} \frac{N}{M})$ fasi che usano $O(\frac{N}{B})$ trasferimenti ciascuno. E' simile al distribution sort ma c'è la complicazione di partizionare gli elementi.

Capitolo 14

Modello Cache Oblivious

In molti casi concreti la gerarchia di memoria con cui dobbiamo fare i conti, include uno o più livelli di *cache* che si frappongono tra il processore e la memoria principale.

Fino ad ora abbiamo considerato soltanto algoritmi che tengono conto della memoria principale, ma esistono diversi algoritmi progettati espressamente per sfruttare questo livello intermedio, ottenendo dei risultati spesso migliori, ma non sempre superiori ad algoritmi specificatamente progettati con una cache specifica in mente. D'altra parte, è provato che gli algoritmi *cache-oblivious* lavorano bene su tutti i livelli della gerarchia e sono più robusti al variare della dimensione del problema.
L'operazione primitiva più importante nel modello *cache-oblivious* è lo **scanning**, mediante il quale è possibile risolvere diversi problemi come il reperimento della mediana.

14.1 (a,b)-tree

Un albero prende il nome di (a,b)-tree se:

- $a \geq 2$, e $b \geq 2a - 1$
- Tutte le foglie hanno la stessa profondità
- Tutti i nodi interni hanno grado (arietà) al massimo di b
- Tutti i nodi interni hanno grado almeno a (tranne la radice)
- la radice ha grado almeno 2

Ogni nodo interno ha generalmente più di un elemento, non più di b e almeno a, i quali puntano a sottoalberi che hanno la caratteristica di contenere nodi con chiavi di valore sempre minore o uguale all'elemento radice (le foglie contengono dunque un solo elemento). Questo consente di effettuare ricerche in $O(\log_a N)$ I/O a patto che $b = O(B)$.

14.1.1 Aggiornare un (a,b)-tree

E' di cruciale importanza effettuare con la dovuta attenzione gli aggiornamenti alla struttura dati, preoccupandoci di bilanciare i nodi opportunamente mediante i vincoli forniti da a e b. Per le inserzioni ed eliminazioni occorre quindi effettuare le seguenti operazioni (mostrate anche in Fig. 14.1):

split avviene nel caso in cui dopo un'inserzione l'arietà di un elemento superi b. in questi caso si separa il nodo in due nodi distinti di arietà $\frac{b+1}{2}$,

share avviene nel caso in cui dopo un'eliminazione l'arietà di un elemento sia inferiore ad a. in questi caso si prende un'opportuno elemento da un nodo fratello di arietà maggiore di a e lo si inserisce nel nodo corrente.

fusion avviene nel caso in cui occorre effettuare uno *share* ma i fratelli hanno arietà a, in questo caso il nodo corrente viene fuso con uno dei suoi fratelli ottenendo un nodo di arietà al più $2a-1$ (da qui il vincolo degli (a,b)-tree).

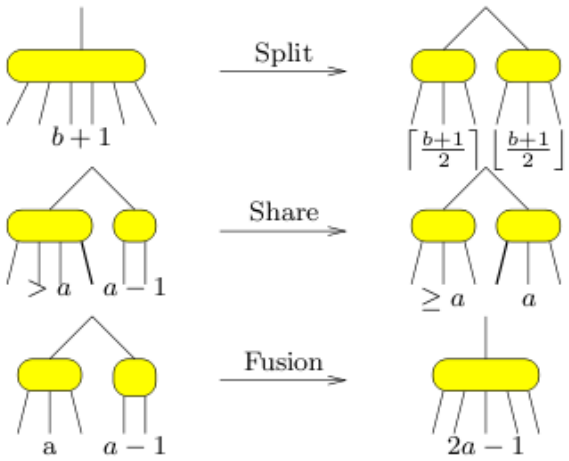


Figura 14.1: Le tre possibili fasi critiche in cui è necessario un aggiornamento della struttura dell'albero

14.2 B-tree

I B-tree sono una delle strutture dati principali nel modello I/O, specialmente riguardo i dizionari, e sono gli (a,b)-tree con b minimo, ovvero $(a, 2a-1)$-tree. Siccome ha un'altezza di $O(\log_B N)$ ed ogni nodo può essere acceduto in un numero di trasferimenti in memoria costante $O(1)$, può garantire ricerche in $O(\log_B N)$ trasferimenti in memoria. Permette inolte di effettuare ricerche in un *range* di valori K, invece che per uno solo di essi, in $O(\log_B N + \frac{K}{B})$ trasferimenti in memoria.

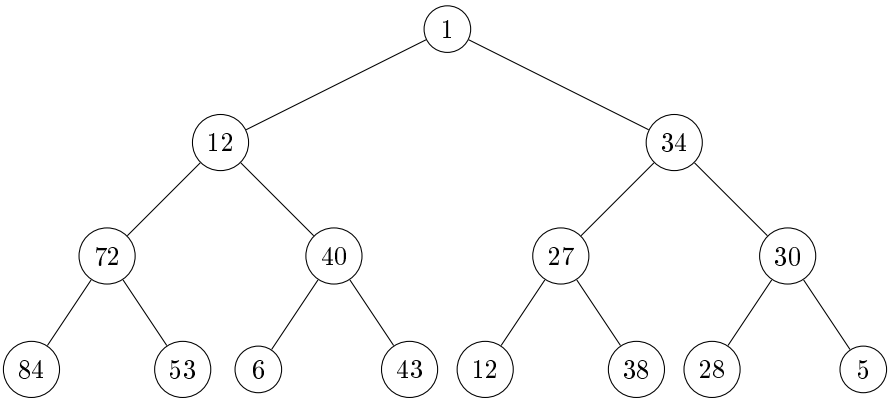
14.2.1 Van Emde Boas Layout

In questo contesto prenderemo in considerazione solo alberi binari completi, ovvero albero di altezza h con $2^h - 1$ nodi.
L'idea di base consiste in un albero binario completo T con N foglie trasformato dividendo T a metà e procedendo ricorsivamente per ambo le parti mediante il seguente criterio

1. se T ha un solo nodo, risiederà come singolo nodo in memoria
2. altrimenti, considerando $h = \log N$ come l'altezza di T, poniamo T_0 come il sottoalbero dei nodi nei primi $\frac{h}{2}$ livelli di T, e $T_1, \ldots, T_k$ come i $\Theta(\sqrt{N},)$ sottoalberi che hanno come radice i nodi al livello $\frac{h}{2}$ di T
3. si riapplica il procedimento dal punto 1

E' facile notare come tutti i sottoalberi hanno dimensione $\Theta(\sqrt{N})$. Chiamiamo Van Emde Boas Layout di T come il Van Emde Boas Layout di T_0 seguito da quello di $T_1, \ldots, T_k$

Example Considerando il seguente albero nel quale abbiamo inserito numeri come nodi per comodità, sebbene avremmo potuto usare valori arbitrari



Per inserirlo in maniera cache-oblivious occorre scomporlo nel seguente modo:

$$T = [T_0, T_1, T_2, T_3, T_4]$$

dove

$$T_0 = \{1, 12, 34\}$$
$$T_1 = \{72, 84, 53\}$$
$$T_2 = \{40, 6, 43\}$$
$$T_3 = \{27, 12, 38\}$$
$$T_4 = \{30, 28, 5\}$$

Dunque, iterando l'algoritmo per i sottoalberi, l'albero risultante viene salvato in memoria con i nodi in quest'ordine

$$T = [1, 12, 34, 72, 84, 53, 40, 6, 43, 27, 12, 38, 30, 28, 5]$$

Avendo diviso l'albero in sottoalberi è facile intuire come sia possibile massimizzare il numero di accessi ad un blocco già trasferito prima di richiedere il successivo, in caso di ricerca, la quale avviene con un numero asintotico di accessi in memoria pari a $O(\log_B N)$. Mentre nel caso di richieste su un range di valori il numero di accessi diventa $O(\log_B N + \frac{K}{B})$

Capitolo 15

List Ranking

Data una lista *linkata* L ad N-nodi memorizzata in memoria secondaria come una sequenza di nodi *non ordinata*, ognuno con un puntatore *next* al successivo, vogliamo determinare per ogni nodo v di L, il *rango* di v, ovvero il numero di collegamenti tra v e la fine della lista (assumendo che il rango dell'ultimo nodo sia 1).
Il list ranking è un algoritmo che usa l'ottimale $\Theta(sort(N))$ operazioni di I/O.

15.1 Funzionamento List Ranking

Inizialmente assegnamo $rango(v) = 1$ per ogni nodo v nella lista L, con $O(scan(N))$ I/O.

Appendice

Appendice A

Tabelle Comparative

Euristica	Competitività
LRU	k
FIFO	k
LFU	k
Random	k
Marking	$2 \ln k$
DaG	$k \log k$
Greedy Dual-Size	$\frac{DIM_{cache}}{DIM_{paginaMinore}}$

Tabella A.1: Tabella che riassume il grado di competitività per algoritmi di Paging, con k pagine che possono risiedere contemporaneamente in memoria

Algoritmo	Preprocessing	Ricerca	Occupazione in Spazio
Naive	-	$O(nm)$	-
Knuth-Morris-Pratt	$O(m)$	$O(n)$	$\Theta(n)$
Karp-Rabin[1]	-	$O(nm)$	$O(p)$
Boyer-Moore[2]	$O(m)$	$O(n)$	$\Theta(n)$
Suffix Tree	$O(m)$	$O(n)$	$O(m\|\Sigma\|)$
Suffix Array	$O(m)$	$O(n + \log_2 m)$	$\Theta(m)$
Naive(patterns)	-	$O(m + zn)$	-
Aho-Corasick	-	$O(m + z + n)$	$O(m)$

Tabella A.2: Tabella comparativa degli algoritmi di pattern matching, su un testo lungo m ed un insieme di z pattern di lunghezza complessiva n