DEPARTMENT OF INFORMATION ENGINEERING
UNIVERSITY OF PADOVA

Università degli Studi di Padova

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE
Corso di Laurea Triennale in Ingegneria dell'Informazione

Teoria e pratica della compressione dati senza perdita. L'esempio dell'algoritmo DEFLATE

Candidato:
Marco Boscolo Anzoletti
Matricola 591575

Relatore:
Dott. Lorenzo Finesso

Anno Accademico 2011–2012

a Michele

Indice

Capitolo 1

Introduzione

Il problema della compressione nasce dal bisogno di ridurre lo spazio di memorizzazione delle informazioni, sia per quanto riguarda l'uso del calcolatore, dove è sempre più critica la gestione delle informazioni a causa della smisurata quantità di cui oggi disponiamo, sia per quel che riguarda le trasmissioni digitali, dove il problema ha ripercussioni sul tempo di invio/ricezione e sulla qualità dei segnali.

In particolare, la compressione dati senza perdita (*loseless*) è la compressione tale da ottenere la stessa identica informazione dopo la compressione/decompressione dei dati. Infatti le informazioni quali il codice di un programma, testo, documenti, database e file di sistema devono rimanere integri in ogni parte se li si vuole utilizzare. Questa modalità di compressione comporta dei vincoli alla compressione stessa, come dimostrato dai risultati teorici che sono stati raggiunti nell'ultimo secolo [NAT].

Se da un lato la teoria ci indica la strada, da un punto di vista pratico la scrittura e l'implementazione di algoritmi capaci di comprimere dati in modo efficiente è una questione tutt'ora aperta e molte sono state le idee sviluppate per ottenere un buon livello di compressione. Per esempio, in questo lavoro mostreremo algoritmi che operano in maniera concettualmente diversa:

1. l'algoritmo di Huffman, che si basa sulla frequenza dei simboli da comprimere

2. l'algoritmo LZ77, che opera senza conoscere la frequenza

3. l'algoritmo DEFLATE, un metodo che viene utilizzato anche oggi all'interno dei più comuni programmi di compressione dati (Gzip, WinZip, PKZIP...), molto popolare grazie alle sue ottime prestazioni e alla licenza gratuita. Esso si serve dei due algoritmi sopracitati per raggiungere una codifica ottimale.

La tesi è una modesta (ma non banale) trattazione della codifica delle informazioni e della loro compressione: nel primo capitolo espongo alcuni limiti teorici alla codifica delle informazioni, mentre nel secondo capitolo descrivo brevemente l'algoritmo DEFLATE, il suo funzionamento e la strategia vincente che esso adotta per eseguire la compressione.

Buona lettura.

Capitolo 2

Limiti teorici alla compressione dati

2.1 Definizioni di base

Definizione 2.1 (Sorgente). Il luogo in cui viene generata l'osservazione o il messaggio che modelliamo come un processo aleatorio $\underbrace{X_1, X_2, \ldots, X_n}_{\text{sequenza di v. a.}}$.

Qui faremo riferimento al *flusso di dati in ingresso* come sorgente dei messaggi che vorremmo comprimere. Le ipotesi che facciamo riguardo alle variabili aleatorie sono:

- consideriamo il caso ad alfabeto finito di M simboli

$$X_i \quad \in \quad \mathcal{X} = \{x_1, x_2, \ldots x_M\}$$

- le statistiche del processo sono supposte note e assumiamo che le v.a. X siano indipendenti e identicamente distribuite (i.i.d.). Perciò la densità discreta

$$p_i := P[X = x_i] \qquad i = 1, 2 \ldots, M$$

caratterizza completamente il processo (scriveremo in seguito $p(x_i)$).

Definizione 2.2 (Codice). Per una variabile singola $X : \ (\Omega, \mathscr{F}, P) \to \mathcal{X}$ un codice è una mappa

$$\begin{aligned} \mathcal{C} : \qquad & \mathcal{X} \to \{0,1\}^* \\ & x \longmapsto \mathcal{C}(x) \end{aligned}$$

dove x è un generico simbolo dell'alfabeto $\mathcal{X}$ e $\mathcal{C}(x)$ la sua codifica (*codeword*).

La lunghezza in bits della codifica e la lunghezza media del codice sono:

$$l(x) := |\mathcal{C}(x)| \qquad \text{e} \qquad \bar{l} := E[l(X_i)] = \sum_{x \in \mathcal{X}} l(x)p(x)$$

Se la mappa $\mathcal{C}$ è iniettiva, allora è garantita la decodificabilità senza errore di simboli isolati. Nel caso pratico, però, ci interessa codificare una sequenza di simboli

$$X_1, X_2, \ldots, X_n$$

di lunghezza n arbitraria. Una soluzione potrebbe essere quella di dividere i simboli con uno spazio o una virgola, ma questa idea risulta inefficiente.

Definizione 2.3 (Estensione di un codice)**.** Dato un codice $\mathcal{C}$ definiamo il codice esteso $\mathcal{C}^*$

$$\begin{aligned} \mathcal{C}^* \quad : \quad & \mathcal{X}^n \to \{0,1\}^* \\ & x_1, x_2, \ldots, x_n \longmapsto \mathcal{C}(x_1 x_2 \ldots x_n) := \underbrace{\mathcal{C}(x_1)\mathcal{C}(x_2)\ldots\mathcal{C}(x_n)}_{\text{concatenazione}} \end{aligned}$$

Se per ogni n l'estensione $\mathcal{C}^*$ è *iniettiva* allora il codice è univocamente identificabile.

Definizione 2.4 (Codice a prefisso)**.** Un codice si dice *a prefisso* (prefix-code) se nessuna parola di codice è prefisso di un'altra.

Un codice a prefisso può essere decodificato senza avere informazioni sulla codeword successiva perchè la fine di ogni parola è immediatamente riconoscibile.

Esempio 2.1.

$$\mathcal{X} = \{a, b, c\} \qquad \text{e} \qquad \mathcal{C}' = \{0, 10, 110\}$$

x	$\mathcal{C}'(x)$
a	0
b	10
c	110

$\mathcal{C}'$ è un codice a prefisso perchè nessuna parola di codice può essere il prefisso di un'altra. Data una sequenza binaria posso istantaneamente decodificare i dati compressi nel loro alfabeto di origine, come nell'esempio sottostante:

$$\ldots 001011000100 \ldots \quad \to \quad aabcaaba$$

2.2 Disuguaglianza di Kraft

Il nostro obiettivo è quello di costruire un codice decodificabile che abbia lunghezza minima. L'insieme di tutte le lunghezze dei codici decodificabili è limitata dalla seguente:

Dati $\mathcal{X}$, $\mathcal{C}$ e $l(x)$ precedentemente definiti

Teorema 2.1 (Disuguaglianza di Kraft). *Dati $l_1, l_2 \dots l_M$ con $M \geq 1$, esse sono possibili lunghezze di un codice a prefisso $\mathcal{C}$ per $\mathcal{X}$, $l_1 = l(x_1), l_2 = l(x_2), \dots, l_m = l(x_M)$ se se solo se:*

$$\sum_{i=1}^{M} 2^{-l_i} \leq 1 \tag{2.1}$$

Dimostrazione. 1. $\mathcal{C}$ a prefisso $\Rightarrow$ disuguaglianza soddisfatta.

Siano $l_1 \leq l_2 \leq \dots \leq l_{\max}$ le lunghezze dei codici a prefisso. Costruiamo l'albero del codice fino alla profondità $l_{\max}$:

$\sharp$ rami uscenti da l_1 è $2^{l_{\max}-l_1}$

$\sharp$ rami uscenti da l_2 è $2^{l_{\max}-l_2}$

.

.

$\sharp$ rami uscenti da $l_{\max}$ è $2^{l_{\max}-l_{\max}}$

da cui otteniamo che

$$\sum \sharp \text{ rami uscenti} \leq \sharp \text{ totale di rami}$$

cioè

$$\sum_{i=1}^{l_{\max}} 2^{l_{\max}-l_i} \leq 2^{l_{\max}} \quad \Rightarrow \quad \text{disuguaglianza soddisfatta}$$

2. disuguaglianza soddisfatta $\Rightarrow$ $\mathcal{C}$ è a prefisso.

Se per $l_1, l_2 \dots l_M$ si ha che $\sum_{i=1}^{M} 2^{-l_i} \leq 1$ allora esiste un codice a prefisso con quelle lunghezze. Basta costruire un albero di codifica con quelle lunghezze.

□

L'idea è che le lunghezza l_i non possono essere prese tutte corte come si vorrebbe fare, altrimenti la condizione 2.1 sarà violata e nessun codice a prefisso può essere costruito.

2.3 Teorema di Shannon sulla codifica di sorgente

D'ora in poi vale che $\log = \log_2$

Definizione 2.5 (Entropia)**.** Data una misura di probabilità P su $\mathcal{X}$, si dice *entropia*:

$$H_p := -E[\log p(X)] = -\sum_{x\in\mathcal{X}} p(x)\log p(x) \tag{2.2}$$

Nota.

$$H_p = \sum_{x\in\mathcal{X}} p(x)\log\frac{1}{p(x)} \qquad \text{è una funzione concava in } \{p_1, p_2 \ldots, p_n\}$$

Lemma 2.3.1.

$$0 \le H_p \le \log|\mathcal{X}| \tag{2.3}$$

Dimostrazione.

$$0 \le \sum_i p_i \log\frac{1}{p_i} \le \log\Big(\sum_i p_i\frac{1}{p_i}\Big) = \log|\mathcal{X}| \tag{2.4}$$

□

Definizione 2.6 (Divergenza tra misure di probabilità)**.** Siano $\{p_i\}$ e $\{q_i\}$ due misure di probabilità tali che

$$p_i, q_i \le 0 \qquad \sum_i p_i = 1, \quad \sum_i q_i \le 1.$$

Si dice *Divergenza* di p, q:

$$D(p||q) = \sum_i^n p_i \log\frac{p_i}{q_i} \tag{2.5}$$

Lemma 2.3.2.

$$D(p||q) \ge 0 \quad , \qquad D(p||q) = 0 \quad per \quad p = q$$

Dimostrazione. per la concavità del logaritmo abbiamo che

$$-D(p||q) = \sum_i p_i \log\frac{q_i}{p_i} \le \log\Big(\sum_i p_i\frac{q_i}{p_i}\Big) = \log\Big(\sum_i q_i\Big) \le 0$$

□

Teorema 2.2. **(i)** *Sia $P = \{p_1, p_2, \ldots, p_n\}$ una sorgente data. La lunghezza media di ogni codice a prefisso (istantaneo) per P soddisfa:*

$$\bar{l} \geq H_P \tag{2.6}$$

(ii) *Esiste un codice a prefisso per cui*

$$\bar{l} \leq H_P + 1 \tag{2.7}$$

Dimostrazione. (i)

$$\begin{aligned}
\bar{l} - H_P &= \sum_i p_i l_i - \sum_i p_i \log \frac{1}{p_i} \\
&= \sum_i p_i l_i + \sum_i p_i \log p_i \\
&= \sum_i p_i \log 2^{l_i} + \sum_i p_i \log p_i \\
&= \sum_i p_i \log \frac{p_i}{2^{-l_i}}
\end{aligned}$$

- per la disuguaglianza di Kraft $\quad \sum_i 2^{-l_i} \leq 1$
- per il lemma sulla divergenza $\quad \sum_i p_i \log \frac{p_i}{q_i} \geq 0$

$$\Rightarrow \quad \bar{l} - H_P \geq 0 \quad \Rightarrow \quad \bar{l} \geq H_P$$

□

Dimostrazione. (ii) prendiamo come lunghezza del codice a prefisso:

$$l_i = \left\lceil \log \frac{1}{p_i} \right\rceil \qquad \text{allora} \qquad \sum_i 2^{-l_i} = \sum_i 2^{-\left\lceil \log \frac{1}{p_i} \right\rceil} \leq \sum_i 2^{-\log \frac{1}{p_i}} = 1$$

quindi, per la disuguaglianza di Kraft è un codice a prefisso valido.

$$\begin{aligned}
\log \frac{1}{p_i} \leq l_i &\leq \log \frac{1}{p_i} + 1 \\
p_i \log \frac{1}{p_i} \leq p_i l_i &\leq p_i \log \frac{1}{p_i} + p_i \\
\sum_i p_i \log \frac{1}{p_i} \leq \sum_i p_i l_i &\leq \sum_i p_i \log \frac{1}{p_i} + \sum_i p_i \\
H_P \leq \bar{l}_i &\leq H_P + 1
\end{aligned}$$

□

2.3.1 Codifica di Huffman

Dal punto di vista applicativo, mi interessa la lunghezza media del codice:

$$\bar{l} := E[l(X_i)] = \sum_{x \in \mathcal{X}} l(x)p(x)$$

in particolare, voglio avere un codice decodificabile che abbia lunghezza media minima e che rispetti la disuguaglianza di Kraft. Il problema è così descritto:

$$\begin{cases} \min\limits_{l_1, l_2 \ldots, l_M} \sum\limits_{i=1}^{M} l_i p_i \\ \sum\limits_{i=1}^{M} 2^{-l_i} \leq 1 \end{cases}$$

Un esempio di codice a prefisso che risolve questo problema data una distribuzione di probabilità, si può ricavare attraverso un algoritmo scoperto da Huffman. L'idea vincente di questo algoritmo consiste nel dare codici più corti ai simboli dell'alfabeto che compaiono con più frequenza. Chiariamo il suo funzionamento con un esempio

Esempio 2.2. Consideriamo una variabile aleatoria X che prende valori nell'insieme $\mathcal{X} = \{1, 2, 3, 4, 5\}$ con probabilità $0.25, 0.25, 0.2, 0.15, 0.15$. Vogliamo ottenere che la codifica binaria assegni i codici più lunghi agli ultimi due simboli, che sono quelli con minore probabilità, mentre vogliamo assegnare i codici più corti ai primi. Per ottenere ciò mettiamo in colonna i simboli dell'alfabeto, ordinati in ordine decrescente per probabilità. Sommiamo le probabilità degli ultimi due simboli e otteniamo un nuovo simbolo di probabilità 0.30 e rifacciamo la colonna ordinata, che avrà un simbolo in meno. Quando arriviamo ad avere solo due simboli diamo al simbolo più probabile la codifica 0 e all'altro 1. Ripetiamo il procedimento dividendo i simboli precedentemente sommati finchè non ottengo le codifiche di tutti i simboli. La tabella seguente elenca quanto detto:

Lunghezza del codice	Codice	X	Probability			
2	01	1	0.25	0.30	0.45	0.55
2	10	2	0.25	0.25	0.30	0.45
2	11	3	0.20	0.25	0.25	
3	000	4	0.15	0.20		
3	001	5	0.15			

Otteniamo una mappa che va dai simboli dell'alfabeto alla loro codifica

$\mathcal{X}$	1	2	3	4	5
$\mathcal{C}(x)$	01	10	11	000	001

Questo codice ha lunghezza media di 2.4 bits.

Si può provare con una lunga dimostrazione (reperibile in [CT91, paragrafo 5.8]) che la codifica di Huffman è ottimale, cioè se $\mathcal{C}^*$ è la codifica di Huffman e $\mathcal{C}'$ è un'altra codifica, allora $L(\mathcal{C}^*) \leq L(\mathcal{C}')$

2.4 Codifica a blocchi

Il teorema di Shannon determina un limite alla lunghezza minima del codice a prefisso. Si può migliorare l'upper bound costruendo codici per 'blocchi' di simboli di dimensione k

$$(X_1, X_2, \ldots, X_k) \to \mathcal{C}(X_1, X_2, \ldots, X_k) \quad \text{di lunghezza} \quad l(X_1, X_2, \ldots, X_k)$$

Allora è, indicando con $\bar{l}_k$ la lunghezza media

$$\bar{l}_k = \sum_{X_1^k \in \mathcal{X}^k} l(X_1^k) p(X_1^k)$$

(Qui non è $\bar{l}_k = k\bar{l}$. Il codice è costruito specificamente per i blocchi di lunghezza k e non come giustapposizione dei codici di lunghezza 1). Applicando la prima parte del teorema di Shannon abbiamo:

$$H_{p(k)} \leq \bar{l}_k$$

e, costruendo il codice di Shannon direttamente per il blocco prendiamo

$$l_k^s(X_1^k) = \left\lceil \log \frac{1}{p(X_1^k)} \right\rceil \quad \text{risulta quindi}$$

$$\log \frac{1}{p(X_1^k)} \leq l_k^s(X_1^k) \leq \log \frac{1}{p(X_1^k)} + 1$$

Calcolando la media otteniamo quindi:

$$kH_p \leq \bar{l}_k^s \leq kH_p + 1$$

Normalizzando rispetto a k otteniamo, per la lunghezza media in bits/simbolo

$$H_p \leq \frac{\bar{l}_k^s}{k} \leq H_p + \frac{1}{k}$$

Asintoticamente, in k, il codice di Shannon raggiunge lunghezza media, in bits/simbolo, pari all'entropia.

2.5 Codifica Universale

In tutti i risultati ricavati in precedenza abbiamo sempre ipotizzato i processi i.i.d. e si assumeva nota la statistica del processo, cioè la probabilità con cui la sorgente produceva il messaggio. Tuttavia, il modello i.i.d. non è sempre il migliore. In molti casi vi sono correlazioni tra i simboli, come nella lingua italiana (difficilmente dopo aver trovato una `q` in un testo seguirà una `r`) o in generale nei linguaggi umani, oppure in una fotografia, in cui pixel affiancati hanno buona probabilità di essere simili. Per esempio, un modello di correlazione a corto raggio è dato dalle 'catene di Markov', in cui la probabilità di transizione a uno stato del sistema dipende unicamente dallo stato precedente:

$$P(X_{n+1} = x_{n+1}|X_n, X_{n-1}, \ldots, X_0) = P(X_{n+1} = x_{n+1}|X_n)$$

Un modello utile per descrivere processi con correlazioni anche a distanza è fornita dalla classe dei processi stazionari, per cui vale:

$$P(X_1^k = x_1^k) = P(X_k^{t+k} = x_1^k)$$

La definizione di entropia per questo gruppo di processi è

$$H = \lim_{n\to\infty} E\left[\log \frac{1}{P(X_1^n)}\right] = \lim_{n\to\infty} \frac{1}{n} \sum_{x_1^n \in X_1^n} P(X_1^n = x_1^n) \log \frac{1}{P(X_1^n = x_1^n)}$$

per raggiungere la codifica ottimale dobbiamo raggiungere questo limite di entropia. Tuttavia spesso non è nota la statistica del processo. Perciò si tratta di trovare un metodo di codifica che sia ottimo anche senza l'informazione $P(X_1^n = x_1^n)$, utilizzando solo i messaggi che ci arrivano dalla sorgente. Una codifica con questa proprietà prende si dice *codifica universale*. Un esempio di algoritmo che opera in questo modo è il Lempel-Ziv (LZ). La compressione che si ottiene da esso è indipendente dalla probabilità con cui i simboli arrivano in ingresso, ha risultati eccellenti e raggiunge l'entropia nella classe dei processi che sono stazionari.

2.5.1 Algoritmo LZ

Questo algoritmo, dovuto ai due scienziati Abraham Lempel e Jacob Ziv, si basa su un'idea molto semplice e facilmente implementabile, cosa che, grazie anche alla sua efficienza e velocità, lo ha reso molto popolare come algoritmo standard per la compressione dati nei computer.

Consideriamo una sorgente binaria da cui arriva una sequenza di bit 1011010100010... e dividiamo questa sequenza in stringhe tali che non ci siano ripetizioni, come 1,0,11,01,010,00,10,.... Dopo ogni virgola prendiamo come stringa la sequenza di bit più corta che non è ancora stata vista. Essendo la più corta, essa è formata da un prefisso che è già apparso come stringa e da un ulteriore bit. Allora codifichiamo questa stringa con l'indice del suo prefisso e il suo ultimo bit.
Sia $c(n)$ il numero di divisioni dell'input in stringhe. Allora abbiamo bisogno di $\log c(n)$ bits per descrivere l'indice della locazione del prefisso e di un bit per descrivere l'ultimo bit. La codifica della sequenza d'esempio è (000,1)(000,0)(001,1)(010,1)(100,0)(010,0)(001,0), dove il primo termine è l'indice del prefisso e il secondo è l'ultimo bit. La decodifica è lineare e si può ricavare la sequenza iniziale senza errore.

In questo esempio banale può sembrare che non si sia ottenuto un grande risultato (il numero di bit sembra addirittura incrementato rispetto alla sequenza di base) ma per sequenze molto più lunghe questo tipo di descrizione diventa molto efficiente.

Si può dimostrare [CT91, paragrafo 13.8] che questo algoritmo raggiunge asintoticamente l'ottimalità

Capitolo 3

L'algoritmo DEFLATE

Il DEFLATE è un algoritmo di compressione dati senza perdita che utilizza una combinazione di due metodi di compressione: l'algoritmo LZ77 e la codifica di Huffman ([R1951] e [SAL]). Esso è ampiamente utilizzato nei software di compressione, merito della sua efficienza e dell'assenza di brevetto. Prima di avventurarci nella spiegazione del DEFLATE è fondamentale capire il funzionamento delle implementazioni di LZ77 e Huffman che si utilizzano in questo algoritmo.

3.0.2 LZ77

La compressione LZ77 si basa su un approccio adattivo, cioè si ricava un modello della statistica dei dati in ingresso direttamente dal processo di codifica. Esso opera sulle sequenze di dati di massimo 32kB (32*1024 bit) che contengono al loro interno stringhe ripetute. Quando nella sequenza comincia una stringa che è già comparsa prima, questa viene sostituita con un puntatore formato da due numeri: la distanza, che rappresenta i passi che devo fare all'indietro per arrivare al punto della sequenza in cui la stringa già vista comincia, e la lunghezza, che descrive il numero di caratteri che devono essere ripetuti a partire da quel punto. Un esempio renderà tutto più chiaro [FELD]:

Esempio 3.1. Supponiamo di avere un flusso di dati di questo tipo:

```
Blah blah blah blah blah!
```

l'algoritmo comincia a ricevere i dati e legge i seguenti caratteri: 'B', 'l', 'a', 'h', ' ' e 'b'. I cinque caratteri che seguono sono:

```
 vvvvv
Blah blah blah blah blah!
     ^^^^^
```

C'è una perfetta corrispondenza tra questi caratteri e quelli che abbiamo già ricevuto. Perciò possiamo mandare in output il puntatore con le due informazioni di distanza e lunghezza.

I dati che abbiamo fin qui sono:

```
Blah blah b
```

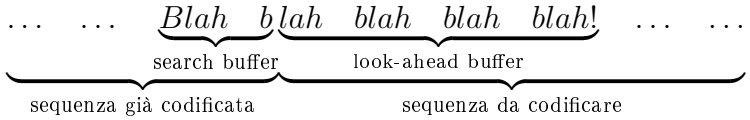
in output otteniamo:

```
Blah b[D=5, L=5]
```

La compressione può essere ulteriormente incrementata perchè nei successivi confronti si trovano gli stessi caratteri ripetuti. In questo particolare esempio risulta che i 18 caratteri che cominciano dal secondo carattere sono identici ai 18 caratteri successivi che cominciano al settimo. Sfruttando al massimo la compressione risulta:

```
Blah b[D=5, L=18]!
```

L'implementazione di questo algoritmo richiede due spazi di memoria al fine di gestire il *search buffer* e il *look-ahead buffer*. Il primo memorizza i dati già codificati, mentre il secondo contiene la parte di stringa successiva che deve essere codificata [GMD06]. Mostriamo queste due sequenze nella figura sottostante, supponendo di trovare la stringa dell'esempio precedente in un certo istante del flusso di ingresso:



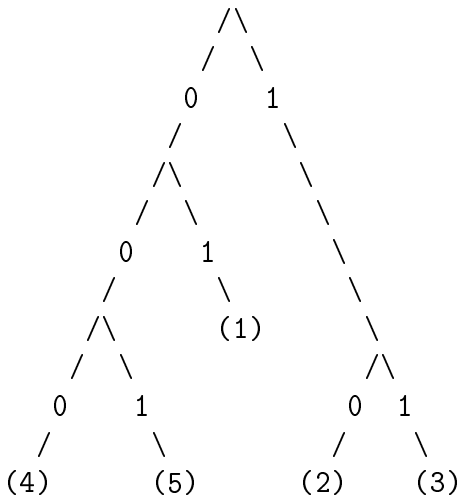
il codificatore cercherà il prefisso più lungo del search buffer contenuto nel look-ahead buffer.

3.0.3 Codifica di Huffman per DEFLATE

La codifica di Huffman è già stata descritta in precedenza. Riportiamo i risultati ottenuti dall'esempio 2.2 per introdurre il concetto di 'albero di Huffman'.

$\mathcal{X}$	1	2	3	4	5
$\mathcal{C}(x)$	01	10	11	000	001

Le codifiche dei simboli possono essere descritte mediante un albero binario i cui rami rappresentano 0 e 1 a seconda se sono il ramo destro o sinistro. Il percorso che porta dalla radice alle foglie letto nella sua sequenza di 0 e 1 è la codifica del simbolo presente nella foglia. Per la tabella precedente abbiamo che il suo albero di Huffman è:



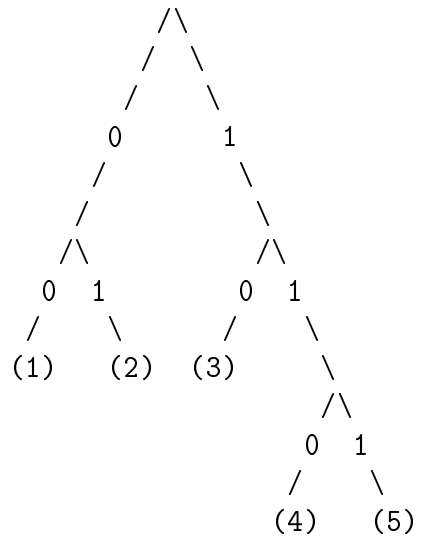
Simbolo	Codifica
1	01
2	10
3	11
4	000
5	010

Questa codifica, descritta precedentemente, si differenzia da quella base per due regole aggiuntive:

1. gli elementi che hanno la codifica più corta sono disposti alla sinistra di quelli con codifica più lunga.

2. tra gli elementi con la codifica della stessa lunghezza, quelli che compaiono per primi nell'insieme sono messi a sinistra.

Con l'aggiunta di queste due proprietà è possibile costruire un solo tipo di albero di Huffman (quindi una sola codifica) per ogni insieme di elementi e le loro rispettive frequenze. Nel nostro caso potremmo ricodificare l'esempio sopra per seguire questa regola come segue, partendo dal presupposto che l'ordine dell'alfabeto è 1 2 3 4 5: 01, 10 e 11 precedono 000 e 010, e i due gruppi di codici di lunghezza due e tre sono lessicograficamente ordinati:



Simbolo	Codifica
1	01
2	10
3	11
4	000
5	010

Queste specifiche consentono al parser, durante la decompressione, di codificare sempre senza ambiguità i simboli. Ogni volta che si trova davanti a un nuovo codice, esso deve 'camminare' attraverso l'albero seguendo il percorso di 0 e 1 fino a raggiungere il simbolo corretto.

3.1 Descrizione

DEFLATE comprime i dati del flusso di ingresso in blocchi di byte di lunghezza arbitraria (tranne nel caso di blocchi non compressi, la cui lunghezza massima è di 64kB -65535 bytes-). Ogni blocco è compresso separatamente mediante una combinazione di LZ77 e Huffman. Gli alberi di Huffman usati per un blocco sono indipendenti da quelli utilizzati per blocchi precedenti o successivi, mentre LZ77 può utilizzare un riferimento a una stringa duplicata vista in precedenza sulla sequenza di input, fino a un massimo di 32kB. La decisione su come dividere i blocchi e su quale strategia di compressione utilizzare non è presa da DEFLATE, bensì il programmatore che implementerà l'algoritmo dovrà occuparsi di scrivere ulteriori algoritmi che facciano in modo intelligente questa scelta.

Un singolo blocco è diviso in due parti: un codice di Huffman che descrive la compressione e i dati compressi. Quest'ultimi consistono di una serie di elementi di due tipi: semplici stringhe di byte di cui non si è trovato un duplicato dentro i 32kB precedenti, chiamati *literals*, e puntatori alle stringhe duplicate formati da coppie <distanza, lunghezza>. La rappresentazione usata da DEFLATE limita la 'distanza' a 32 kB e la 'lunghezza' a 258 bytes,

ma non è un vincolo alla grandezza del blocco (eccetto i dati non compressi, come detto in precedenza).

Ogni tipo di dato (literals, distanze, lunghezze) è rappresentato nei dati compressi con i codici di Huffman: un albero per i literals e le lunghezze e un albero separato per le distanze. Questi alberi sono memorizzati nel blocco appena prima dei dati compressi.

3.1.1 Convenzioni

Usiamo questo simbolo:

```
+---+
|   | <-- le barre verticali non sono per forza necessarie
+---+
```

per indicare un byte; mentre

```
+==========+
|          |
+==========+
```

rappresenta un numero variabile di bytes. Inoltre utilizzeremo spesso le parole 'distanza' e 'lunghezza': esse vanno intese come misure costruite dall'algoritmo LZ77. Con il termine 'blocco' indicheremo sempre le parti in cui l'algoritmo divide i dati. Il numero binario all'interno di un byte ha la cifra più significativa a sinistra.

3.1.2 Dettagli del blocco

Ogni blocco è preceduto da 3 bit che danno le seguenti informazioni:

- 1 bit: marcatore della sequenza
  - 1: questo blocco è l'ultimo della sequenza
  - 0: ci sono altri blocchi oltre a questo
- 2 bit definiscono il tipo di codifica per il blocco
  - 00: il blocco non viene codificato
  - 01: comprimi il blocco con la codifica di Huffman utilizzando un albero precostruito
  - 10: comprimi il blocco con la codifica di Huffman utilizzando un albero creato apposta per questo blocco (*Huffman dinamico*)

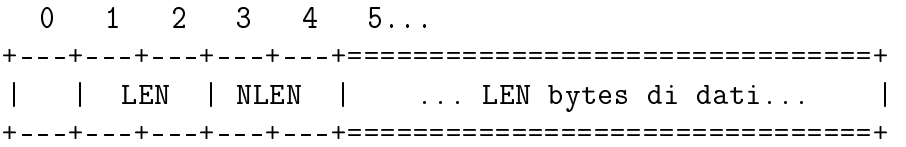
- 11: riservato (comando non utilizzato)

Molti blocchi saranno codificati con il metodo 01, che produce una codifica di Huffman ottimizzata individualmente per ogni blocco. Le istruzioni per generare l'albero di Huffman sono memorizzate subito dopo i 3 bit iniziali.

3.1.3 Codifica

(00) - nessuna compressione La forma del blocco è:

1. Tre bit iniziali 000 o 100.
2. Si salta il resto del byte corrente. I quattro bytes successivi contengono il LEN e il suo complemento NLEN, dove il LEN è il numero di bytes nel blocco. Questo è il motivo per cui la grandezza è limitata a 65535 bytes.
3. LEN bytes



Questo modo di comprimere ha senso per quei file, o parti di file, che sono incomprimibili o già compressi, o per i casi in cui il software è chiamato a segmentare i dati. Un caso tipico è un utente che vuole spostare un file di 8 Gb da un computer ad un altro ma ha solo un DVD riscrivibile. L'utente vorrebbe dividere il file in due parti da 4 Gb ciascuna senza comprimere. Un semplice software di compressione basato su DEFLATE può svolgere questo compito usando il modo (00).

(01) - compressione con albero di Huffman fisso La forma del blocco è:

1. Tre bit iniziali 001 o 101.
2. Seguono subito i codici a prefisso dei dati compressi nella forma di literals o di coppie distanza/lunghezza
3. Il codice 256 (o meglio, il suo codice a prefisso) che designa la fine del blocco

Edoc	Bits extra	Lunghezza	Codice	Bits extra	Lunghezza	Codice	Bits extra	Lunghezza
257	0	3	267	1	15, 16	277	4	67 – 82
258	0	4	268	1	17, 18	278	4	83 – 98
259	0	5	269	2	19 – 22	279	4	99 – 114
260	0	6	270	2	23 – 26	280	4	115 – 130
261	0	7	271	2	27 – 30	281	5	131 – 162
262	0	8	272	2	31 – 34	282	5	163 – 194
263	0	9	273	3	35 – 42	283	5	195 – 226
264	0	10	274	3	43 – 50	284	5	227 – 257
265	1	11, 12	275	3	51 – 58	285	0	258
266	1	13, 14	276	3	59 – 66			

Tabella 3.1: Edoc per stringhe/lunghezze del modo (01)

Edoc	Bits extra	Distanza	Codice	Bits extra	Distanza	Codice	Bits extra	Distanza
0	0	1	10	4	33 – 48	20	9	1025 – 1536
1	0	2	11	4	49 – 64	21	9	1537 – 2048
2	0	3	12	5	65 – 96	22	10	2049 – 3072
3	0	4	13	5	97 – 128	23	10	3073 – 4096
4	1	5, 6	14	6	129 – 192	24	11	4097 – 6144
5	1	7, 8	15	6	193 – 256	25	11	6145 – 8192
6	2	9 – 12	16	7	257 – 384	26	12	8193 – 12288
7	2	13 – 16	17	7	385 – 512	27	12	12289 – 16384
8	3	17 – 24	18	8	513 – 768	28	13	16385 – 24576
9	3	25 – 32	19	8	769 – 1024	29	13	24577 – 32768

Tabella 3.2: Edoc delle distanze nel modo (01)

edoc	Bits	Codice a prefisso
0 – 143	8	00110000 – 10111111
144 – 255	9	110010000 – 111111111
256 – 279	7	0000000 – 0010111
280 – 287	8	11000000 – 11000111

Tabella 3.3: Codici di Huffman per gli Edoc del modo (01)

La modalità (01) utilizza due tabelle di codici: uno per le lunghezze e un'altra per le distanze. I codici delle tabelle 3.1.3.1 e 3.1.3.2 non sono in realtà quelli che verranno scritti nei dati compressi ma sono un traguardo intermedio, perciò, al fine di togliere eventuali ambiguità, usiamo il termine *edoc* per riferirci ad essi. Ogni edoc è poi convertito in un codice a prefisso (tabella 3.1.3.3) che sarà il risultato della compressione. Gli edoc da 0 a 255 sono riservati ai literals e sono 256 come le possibili configurazioni di un byte, infatti nel caso peggiore avrò in ingresso 256 byte diversi (tutte le 2^8 possibili rappresentazioni di un byte) prima di un puntatore. Un esempio di caso limite è questo: in ingresso abbiamo in sequenza tutti gli interi da 0 a 255

```
     0        1       .   .   .    .      254       255
+--------+--------+================+--------+--------+
|00000000|00000001|    ...    ...   |11111110|11111111|
+--------+--------+================+--------+--------+
```

qui non è possibile operare sostituzioni con puntatori, e il codificatore assegna tutti gli edoc da 0 a 255 ai byte.

Gli edoc 257-285 sono usati per le lunghezze e quest'ultimi non sono sufficienti a rappresentare tutte le 256 possibili lunghezze, per cui ad alcuni vengono aggiunti dei bit extra. Per le distanze si riutilizzano gli edoc da 0 a 29 con l'aggiunta di bit extra. Dopo una lunghezza c'è sempre una distanza per cui il parser interpreterà sempre il codice successivo correttamente senza ambiguità. Infine l'edoc 256 è il codice 'fine-del-blocco'. Il risultato finale della compressione è il codice a prefisso degli edoc (tabella 3.1.3.3). Notiamo che gli edoc di 286 e 287 non sono creati, perciò i loro codici a prefisso non sono usati. Facciamo un esempio di codifica:

Esempio 3.2. Abbiamo in ingresso un blocco di dati e ad un certo istante l'algoritmo individua una ripetizione di 20 literals byte a distanza di 2051 byte. Il primo passo è creare il puntatore con gli edoc: <(269)01,(22)0000000010>. Poi si usa la codifica di Huffman trasformando tutto in binario:

$$< \underbrace{0001101}_{\text{Huffman code}} 01 \underbrace{01000110}_{\text{Huffman code}} 0000000010 >$$

(10) - compressione con albero di Huffman *dinamico* Questa modalità di compressione è senza dubbio la più complessa. Le due tabelle di edoc che abbiamo visto in precedenza, qui devono essere riscritte, e per farlo ci si basa sulle informazioni raccolte dai simboli che abbiamo sul nostro blocco di dati da comprimere, così come il codice di Huffman. Successivamente tutte le tabelle devono essere memorizzate all'interno del blocco e

'spedite' con esso per garantire la successiva decodifica (addirittura esse sono ulteriormente codificate con Huffman per diminuire ancora lo spazio di memorizzazione!). L'idea è che un buon codificatore DEFLATE possa raccogliere informazioni sui dati in ingresso e sui blocchi compressi, per poi utilizzare queste per costruire il codice a prefisso migliore per i blocchi successivi. Ma un' implementazione poco efficiente di questa modalità del DEFLATE potrebbe portare ad avere codici a prefisso simili a quelli della modalità (01) o, ancor peggio, ad ottenere blocchi di dati non compressi. Come detto in precedenza, sta al programmatore che implementerà l'algoritmo la capacità di prendere la decisione sull'uso o meno di questa tecnica di compressione. Per una descrizione approfondita di questa modalità si rimanda alla [R1951].

3.1.4 Decodifica

L'unica differenza tra i due casi di compressione è il modo in cui sono definiti i codici di Huffman per le stringhe/lunghezze e le distanze.

In ogni caso, l'algoritmo di decodifica non è complesso e possiamo darne una semplice rappresentazione in pseudocodice:

```
do
 { leggi l'inizio del blocco dal flusso di ingresso;
   if(modalità senza compressione)
    { ignora gli altri bit di questo byte;
      leggi il LEN e il NLEN;
      copia LEN bytes di dati in uscita;
    }
   else if(modalità con Huffman dinamico)
    { leggi la rappresentazione degli alberi di codice;
    }
   loop
    { decodifica la stringa/lunghezza dal flusso di ingresso;
      if(il suo valore < 256) /* un literal */
       { copia il dato in uscita;
       }
      if(valore = 256)  /* simbolo di fine-del-blocco */
       { break;
       }
      else /* valori da 257 a 285, cioè una lunghezza */
       { decodifica distanza e successiva lunghezza;
         torna al byte segnato dal puntatore distanza nel flusso
         in uscita e copia i byte da questa posizione al flusso di
         uscita fin quanto segnalato dal puntatore della lunghezza;
       }
    }
while(non è l'ultimo blocco)
```

Notiamo che le stringhe duplicate possono riferirsi a stringhe di blocchi precedenti, cioè un puntatore di distanza può attraversare uno o più blocchi singoli (ma non può mai puntare ad una posizione che vada oltre l'inizio del flusso di dati in uscita). Inoltre, è possibile che un puntatore produca una stringa ripetendo più volte la stringa a cui si riferisce; per esempio, se gli ultimi due byte decodificati sono X e Y, un puntatore <lunghezza = 5, distanza = 2> aggiunge X,Y,X,Y,X all'uscita.

Esempio 3.3. Usando il risultato dell'esempio 3.2, quando il parser eseguirà la decodifica non avrà mai il dubbio su cosa stà leggendo in quell'istante perché i literals e le lunghezze provengono da un codice di Huffman (perciò a prefisso) e sa che dopo una lunghezza troverà sempre una distanza. Ad esempio:

$$\ldots 00111001111000110101000110101010001100000000010000011\ldots$$

il parser riconosce i codici:

$$\ldots 001\big|\underbrace{110011110\big|00110101}_{\text{literals}}\big|\underbrace{0001101\big|0101000110000000010}_{\text{puntatore <lung./dist.>}}\big|0000011\ldots$$

li traduce ottenendo:

$$\begin{array}{l|c|c|c|l}
\ldots\ldots & 110011110 & 00110101 & <(269)01,(22)0000000010> & 0000011\ldots \\
\ldots\ldots & 158 & 5 & <20,2051> & \ldots\ldots\ldots
\end{array}$$

3.2 Conclusione

DEFLATE è un algoritmo di compressione dati senza perdita che negli anni si è rivelato un componente importante di molti programmi di compressione. Esso richiede memoria per il look-ahead e il search buffer e per le due tabelle di codici a prefisso. Tuttavia, la quantità di memoria di cui necessità il codificatore/decodificatore è indipendente dalla grandezza dei dati o dai blocchi. Inoltre la sua implementazione, certamente non banale, è più semplice di implementazioni più recenti presenti nei formati JPEG 2000 o MPEG. Gli algoritmi di compressione nati per appositi formati, come audio e video, sono in genere più performanti del DEFLATE, ma esso ha un fattore di compressione tra 2.5 e 3 per il testo, leggermente inferiore per i file eseguibili e ancor meglio per le immagini. Infine, è importante sottolineare che, nel caso peggiore, l'algoritmo espande i dati di soli 5 byte ogni 32 kB di blocco.

Bibliografia

[CT91] T.M. Cover, J.A. Thomas (1991), *Elements of Information Theory* 2th Ed., Wiley, New York, pagine 118–119.

[FELD] A. Feldspar (1997), An Explanation of the Deflate Algorithm, Sito web `http://zlib.net/feldspar.html`.

[FIN] L. Finesso (2006), Noiseless Source Coding, Introduzione al problema della compressione dati, Corso: *Probabilità e statistica*, LL Informatica 05/06, UniPd.

[GMD06] F. Lana, D. Masato, L. Polin (2006), Tecniche di compressione con dizionario, Corso: *Gestione ed elaborazione grandi moli di dati*, LS Ing. Informatica 05/06, UniPd.

[NAT] S. Natarajan (2001), Entropy, Coding and Data Compression, *Resonance*, Vol. 6, No. 8, pagine 35–45.

[R1951] P. Deutsch (1996), Request For Comments 1951, Sito web `http://www.ietf.org/rfc/rfc1951.txt`.

[SAL] D. Salomon (2007), *Data Compression. The Complete Reference* 4th Ed., Springer, pagine 230–241.

[WELC] T.A. Welch (1984), A Technique for High-Performance Data Compression, *Computer*, Vol. 17, No. 6, pagine 8–19.