Il metodo di compressione Lempel - Ziv - Welch

Renzo Sprugnoli

15 luglio 2005

Nella compressione dei testi, è di fondamentale importanza non perdere alcuna informazione, sia nella fase di codifica, sia in quella di decodifica. Come abbiamo visto, tanto il Run Length Encoding quanto l'algoritmo di Huffman godono di questa proprietà. Tale è anche il metodo di Lempel, Ziv e Welch che, tuttavia, si basa su principi completamente differenti. Il confronto più naturale è col metodo di Huffman: mentre questo usa codici a lunghezza variabile per far sì cha a caratteri più frequenti corrispondano codici più corti, il metodo LZW usa codici a lunghezza fissa per codificare sequenze di caratteri di lunghezza variabile. In questo metodo è interessante il fatto che le sequenze codificate non siano casuali, ma corrispondano a sequenze che compaiono con una certa frequenza; e se una sequenza lunga tende a ripetersi, il metodo se ne accorge e in modo dinamico l'aggiunge a un dizionario di conversione. Tale dizionario, a differenza di quello creato dal metodo di Huffman, non deve essere memorizzato o spedito con la codifica del testo, ma è ricostruibile dalla codifica stessa: è proprio tale ricostruzione che permette la decodifica del testo man mano che si procede alla sua scansione.

Sia $A = \{a_1, a_2, \ldots, a_k\}$ l'alfabeto su cui il testo T è scritto; nella pratica, A è l'insieme delle 256 configurazioni dei byte, così che ogni carattere ha la sua codifica (ad esempio, quella tradizionale ASCII) o, addirittura, si può pensare di convertire un testo, originariamente binario, visto come sequenza di byte. Il metodo fa uso di un dizionario $\mathcal{D}$ che inizialmente contiene la semplice lista dei caratteri di A: l'indice di ciascun elemento in $\mathcal{D}$, visto come vettore, è la codifica dell'elemento stesso. Via via che si procede nella scansione di T, sequenze di caratteri vengono inserite in $\mathcal{D}$ come nuovi elementi e, quando tali sequenze verranno di nuovo incontrate in T, verranno codificate col loro indice in $\mathcal{D}$. Vedremo più avanti le dimensioni di $\mathcal{D}$ e in che modo i numeri che gli fanno da indice vengono memorizzati nel testo compattato C. Per il momento, vedremo C come un vettore di interi, anche se questo può apparire a prima vista addirittura controproducente.

Utilizziamo come esempio l'alfabeto $A = \{a, b, c\}$ e consideriamo la parola:

$$T = bcababbcabcbcbbccaabbababb$$

generata casualmente su tale alfabeto. Il dizionario $\mathcal{D}$ contiene inizialmente tre soli elementi che rappresenteremo come coppie indice / sequenza:

1	a
2	b
3	c

ma che dobbiamo immaginare come vettore. Il metodo fa uso di una coda K nella quale si accumulano i caratteri scanditi in T. Inizialmente, inseriamo in K il primo carattere b di T. Poiché la codifica di b in $\mathcal{D}$ è 2, questo numero è l'elemento iniziale di C. Comincia ora un ciclo che andrà avanti fino a che i caratteri di T non saranno terminati.

1. si mette in una variabile Z l'indice in $\mathcal{D}$ della sequenza contenuta in K;
2. si prende il carattere successivo in T e si accoda in K. Nell'esempio, K verrà a contenere la sequenza bc. Se i caratteri di T sono finiti, si salta al punto 6.;
3. si cerca, nel dizionario $\mathcal{D}$, la sequenza contenuta in K;

$2 \to Z$	$K := \text{“}bc''$	$(4, \text{“}bc'') \to \mathcal{D}$
		$C := [2]$
		$K := \text{“}c''$
$3 \to Z$	$K := \text{“}ca''$	$(5, \text{“}ca'') \to \mathcal{D}$
		$C := [2, 3]$
		$K := \text{“}a''$
$1 \to Z$	$K := \text{“}ab''$	$(6, \text{“}ab'') \to \mathcal{D}$
		$C := [2, 3, 1]$
		$K := \text{“}b''$
$2 \to Z$	$K := \text{“}ba''$	$(7, \text{“}ba'') \to \mathcal{D}$
		$C := [2, 3, 1, 2]$
		$K := \text{“}a''$
$1 \to Z$	$K := \text{“}ab''$	$(\text{“}ab'' \in \mathcal{D})$
$6 \to Z$	$K := \text{“}abb''$	$(8, \text{“}abb'') \to \mathcal{D}$
		$C := [2, 3, 1, 2, 6]$
		$K := \text{“}b''$
$2 \to Z$	$K := \text{“}bc''$	$(\text{“}bc'' \in \mathcal{D})$
$4 \to Z$	$K := \text{“}bca''$	$(9, \text{“}bca'') \to \mathcal{D}$
		$C := [2, 3, 1, 2, 6, 4]$
		$K := \text{“}a''$
$1 \to Z$	$K := \text{“}ab''$	$(\text{“}ab'' \in \mathcal{D})$
$6 \to Z$	$K := \text{“}abc''$	$(10, \text{“}abc'') \to \mathcal{D}$
		$C := [2, 3, 1, 2, 6, 4, 6]$
		$K := \text{“}c''$
...	...	...

Tabella 1: Esecuzione della compressione

4. se (come nel caso attuale) K non si trova in $\mathcal{D}$, si aggiunge a $\mathcal{D}$ come nuovo elemento. Nell'esempio, $\mathcal{D}$ viene a contenere la sequenza bc come quarto elemento. Eseguita l'aggiunzione, si inserisce Z in C e si eliminano dalla testa di K tutti i caratteri, eccetto l'ultimo introdotto; nel nostro caso, K si riduce a c. Si ritorna quindi al punto iniziale 1.;

5. altrimenti, se cioè K si trova in $\mathcal{D}$, si torna al punto 1.;

6. quando si è finito, si inserisce Z come ultimo elemento di C e si esce.

Proseguendo nell'esempio, si ha $K = \text{“}c''$ e quindi si mette 3 in Z, si aggiunge il carattere a a K e poiché ca non è presente in $\mathcal{D}$, a questo si accoda ca cme quinto elemento. Nella

1	a	7	ba	13	bb
2	b	8	abb	14	bcc
3	c	9	bca	15	caa
4	bc	10	abc	16	$abba$
5	ca	11	cb	17	aba
6	ab	12	bcb		

Tabella 2: Il dizionario completo

Tabella **??** si è cercato di schematizzare il comportamento dell'algoritmo. Il lettore segua il processo riportato nella tabella e continui per conto proprio, fino a trovare la codifica:

$$2, 3, 1, 2, 6, 4, 6, 3, 4, 2, 4, 5, 8, 6, 8.$$

mentre il dizionario $\mathcal{D}$ completo è riportato nella Tabella **??**.

Come abbiamo accennato, pur illustrando abbastanza bene il metodo, questo esempio non è significativo. Se infatti facciamo un po' di conti, poiché ogni carattere di A può essere codificato con 2 bit, l'occupazione del testo T è di $26 \times 2 = 52$ bit in totale. La codifica usa 8 codici (i numeri da 1 ad 8) e quindi ogni codice deve essere rappresentato da 3 bit per un totale di $15 \times 3 = 45$ bit. La compressione risulta di $45/52 \approx 86.5\%$, con un risparmio di oltre il 13%. Con testi così brevi e con alfabeti così minuscoli, questo è un caso già molto favorevole e basta pensare a come il guadagno si sarebbe trasformato in perdita se solo avessimo dovuto utilizzare 4 bit, perché i numeri da decodificare sono più di 8. I veri vantaggi si hanno con alfabeti più ampi (tutti i 256 byte vanno benissimo) e con testi molto lunghi. Ad esempio, trattando i file di un calcolatore e identificando A con il contenuto dei singoli byte, si usa spesso una codifica a 12 bit, cioè ogni numero di C è rappresentato da una sequenza di 12 bit (un byte e mezzo). questo significa che il dizionario $\mathcal{D}$ è limitato alla dimensione $2^{12} = 4096$, e questo permette l'uso di $4096 - 256 = 3840$ codici per le sequenze lunghe almeno 16 bit. Il risparmio medio si aggira sul 40% per file di testo.

$T :=$ "b''				$L1 :=$ "b''
$T :=$ "bc''	$L2 :=$ "c''	$K :=$ "bc''	$(4,$ "$bc'') \to \mathcal{D}$	$L1 :=$ "c''
$T :=$ "bca''	$L2 :=$ "a''	$K :=$ "ca''	$(5,$ "$ca'') \to \mathcal{D}$	$L1 :=$ "a''
$T :=$ "$bcab''$	$L2 :=$ "b''	$K :=$ "ab''	$(6,$ "$ab'') \to \mathcal{D}$	$L1 :=$ "b''
$T :=$ "$bcabab''$	$L2 :=$ "ab''	$K :=$ "ba''	$(7,$ "$ba'') \to \mathcal{D}$	$L1 :=$ "ab''
$T :=$ "$bcababbc''$	$L2 :=$ "bc''	$K :=$ "abb''	$(8,$ "$abb'') \to \mathcal{D}$	$L1 :=$ "bc''
$T :=$ "$bcababbcab''$	$L2 :=$ "ab''	$K :=$ "bca''	$(9,$ "$bca'') \to \mathcal{D}$	$L1 :=$ "ab''
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$

Tabella 3: Il processo di decompressione

Vediamo ora come avviene la decodifica, cioè come si decomprime il testo che era stato compattato col metodo LZW. L'algoritmo riflette all'inverso le operazioni fatte in fase di compressione. Si parte con un dizionario $\mathcal{D}$ che contiene tutte e sole le lettere dell'alfabeto A; l'alfabeto è l'unico dato che deve essere conosciuto sia da chi comprime sia da chi decomprime. Il primo numero (codice) di C viene tradotto tramite $\mathcal{D}$ e la lettera corrispondente viene messa tanto in T (il testo che si vuole ricostruire) quanto in $L1$, una variabile di comodo utilizzata per ricostruire il dizionario $\mathcal{D}$. A questo punto, l'algoritmo procede nel modo seguente:

1. si prende da C il numero (codice) successivo, lo si traduce in X tramite il dizionario $\mathcal{D}$ e la sequenza X ottenuta si accoda a T e si mette in una variabile di comodo $L2$. Naturalmente, se non ci sono più elementi in C, l'algoritmo termina;

2. si concatena $L1$ col primo carattere di $L2$ e il risultato si pone in K;

3. si aggiunge K al dizionario $\mathcal{D}$;

4. si sposta $L2$ in $L1$ e si torna al passo 1..

Nel nostro esempio, il secondo numero, 3, corrisponde al carattere c per cui T diviene "bc" e $L2$ è "c"; di conseguenza è $K =$ "bc'' e tale sequenza è aggiunta a $\mathcal{D}$ come quarto elemento. Nella Tabella **??** si sono schematizzati i primi passaggi dell'algoritmo, dopo la fase iniziale riportata nella prima linea. Il lettore è invitato a proseguire fino alla fine, ricostruendo il testo decompresso T e il dizionario $\mathcal{D}$ che viene a coincidere con quello della Tabella **??**.

1	a	4	ab	7	ca
2	b	5	bc	8	aba
3	c	6	cc	9	$abab$

Tabella 4: Il dizionario del caso speciale

Questo algoritmo presenta un unico problema. Si consideri il testo "$abccababab c''$ sul solito alfabeto $A = \{a, b, c\}$. E' facile vedere che il risultato della compressione è $C = [1, 2, 3, 3, 4, 8, 5]$, corrispondente al dizionario della Tabella **??**. Quando si cerca di decomprimere C, tutto procede bene fino a che non abbiamo ricostruito $T =$ "$abccabab''$, abbiamo aggiunto $(7,$ "ca'') al dizionario $\mathcal{D}$ e abbiamo $L1 =$ "ab''. A questo punto troviamo un 8 in C, ma il dizionario contiene solo 7 elementi.

Questa situazione si verifica soltanto quando, in fase di compressione, si è aggiunta al dizionario una sequenza che inizia e termina con lo stesso carattere e che è stata generata sfruttando giusto il primo carattere della sequenza codificata proprio con quel codice. Qui la sequenza è "$ababa''$ (dopo una precedente codifica di "ab''), ma potrebbe essere "$aaaa''$ o anche "$abcabca''$, dopo una precedente codifica di "ab'' e "abc'': si provi a comprimere e decomprimere "$ababcabcabca''$.

Questa osservazione, fortunatamente, ci permette di risolvere il problema: quando si trova

in C un numero (codice) maggiore della dimensione del dizionario $\mathcal{D}$ generato fino a quel momento, si deve aggiungere a T e porre in $L2$ il medesimo valore che è in $L1$, concatenato col suo stesso primo carattere. Nel passo 1. del precedente algoritmo, la frase "lo si traduce in X tramite il dizionario", va così intesa:

> se il numero (codice) n è minore o uguale alla dimensione del dizionario $\mathcal{D}$, si assegna ad X la sequenza individuata come posizione da n, altrimenti si pone X uguale ad $L1$ concatenato col primo carattere dello stesso $L1$.

Con questo, l'algoritmo di decompressione è completo.