Corso di Trasmissione ed Elaborazione Numerica dei Segnali

SUN - Seconda Università degli Studi di Napoli

Laurea Magistrale in
Ingegneria Informatica

aa. 2013-14

Prof.: F. A. N. Palmieri

Appunti sugli Algoritmi LZ

con particolare riferimento all'LZW

Autore: Giovanni Di Gennaro
email: giovannidigennaro@virgilio.it

CAPITOLO 1

L'algoritmo Lempel-Ziv-Welch

1.1. Introduzione

Nella compressione dei dati senza perdita d'informazione (*lossless*) grande importanza assume la famiglia degli algoritmi LZ. La maggior parte delle tecniche di compressione adattative oggi utilizzate, sono infatti riconducibili proprio ai concetti sviluppati dai ricercatori israeliani Abraham Lempel e Jacob Ziv, e pubblicati nel 1977 e nel 1978. Da tali pubblicazioni derivano rispettivamente gli algoritmi LZ77 (acronimo di Lempel-Ziv 1977), basato sulla compressione a *finestre scorrevoli* (sliding-window), ed LZ78, sviluppato proprio per cercare di migliorare l'efficienza dell'algoritmo precedente. Entrambi gli algoritmi sono oggi considerati obsoleti, ma le rispettive filosofie di base sono presenti in una miriade di versioni alternative, che definiscono gran parte degli standard più comuni di compressione. La forma più popolare assunta dall'algoritmo LZ78 è quella ottenuta nel 1984 da Tarry Welch, che cercò di migliorarne alcune caratteristiche (soprattutto sotto il profilo della velocità) dando origine all'algoritmo oggi noto come LZW (acronimo di Lempel-Ziv-Welch), che rappresenta ciò di cui ci occuperemo[1].

1.2. Descrizione dell'algoritmo

L'idea alla base dell'algoritmo è abbastanza semplice, ossia ottenere un codice "compresso" in uscita cercando di sfruttare la presenza di ripetizioni all'interno di una *stringa* in ingresso. L'algoritmo LZW esegue tale operazioni basandosi sulla creazione dinamica di un dizionario (*codebook*). All'inizio il codebook conterrà solamente la lista dei caratteri dell'alfabeto $\mathcal{A} = \{a_1, a_2, \dots, a_k\}$ con cui si desidera operare[2], aggiungendo a quest'ultimo nuovi elementi man mano che la stringa in ingresso viene esaminata. La cosa più interessante di tale tipo di algoritmo è dovuta al fatto che, a differenza di quanto accade col metodo Huffman, il codebook non richiede di essere inviato insieme alla codifica, ma è direttamente ricostruibile durante il processo di decodifica[3].

---

[1] Questo tipo di codifica è ancora oggi utilizzata in formati quali **TIFF** e **GIF**, oltre che dalla coppia di utility Unix **compress** ed **uncompress**. Nel secondo capitolo di queste note sono tuttavia proposti anche gli algoritmi LZ77 ed LZ78 nella loro forma originale, per chiunque fosse interessato ad un approfondimento.

[2] A differenza invece di quanto accade nell'algoritmo LZ78, che all'inizio contiene nel dizionario solo la stringa vuota.

[3] Ovviamente, poiché il codebook dovrà contenere in principio già tutta la lista dei caratteri dell'alfabeto $\mathcal{A}$, è banale osservare che tale lista debba essere già nota anche a destinazione (poiché se così non fosse si dovrebbe quantomeno trasmettere, con la codifica, anche l'alfabeto iniziale). Ecco il motivo per il quale l'alfabeto iniziale è di solito rappresentato da strutture già note, come ad esempio la codifica ASCII.

Il flusso dei simboli emessi dalla sorgente verrà suddiviso in frasi di lunghezza variabile, dove ogni frase è ottenuta estraendo, dalla stringa in input, la sottostringa di lunghezza massima che risulti già presente all'interno del codebook costruito fino a quel punto.

In confronto al metodo di Huffman, che usa codici a lunghezza variabile per far in modo che a caratteri più frequenti corrispondano codici più brevi, il metodo LZW capovolge quindi totalmente l'ottica, usando codici a lunghezza fissa per codificare sequenze di caratteri di lunghezza variabile. Un grande vantaggio di tale algoritmo deriva dal fatto di riuscire a comprimere in modo efficiente tipi di dati eterogenei (testo, immagini, database, ecc.), senza tuttavia richiedere informazioni *a priori* sui dati (poiché il processo acquisisce ciò di cui necessita direttamente durante la fase di codifica).

Nel seguito definiremo il metodo classico di codifica e decodifica dell'algoritmo LZW, fornendo anche una differente variante, definita di *codifica all'indietro*, che ne rappresenta una valida alternativa.

1.3. Processo di codifica

Il processo di codifica può essere inteso come composto di tre soli passi, da ripetere *usque ad finem*:

- si effettua una scansione del codebook cercando la più lunga sequenza di simboli in ingresso;
- si codifica in uscita la sottostringa, utilizzando il corrispettivo valore presente nel codebook;
- si aggiunge, all'interno del codebook stesso, un nuovo valore ottenuto concatenando la sottostringa appena utilizzata con il successivo simbolo emesso dalla sorgente.

Un esempio sicuramente chiarirà ogni dubbio. Utilizzando l'alfabeto $\mathcal{A} = \{a, b, c\}$, e considerando che la sequenza da comprimere sia $\{bcababbcbcbaaaabbc\}$, supporremo che il nostro processo di codifica assegni ad ogni sottostringa un valore numerico. Sotto quest'ottica il codebook sarà inizialmente formato dalle uguaglianze $\{a = 1, b = 2, c = 3\}$[4].

Il processo di codifica partirà quindi da una situazione raffigurabile attraverso lo schema seguente:

INPUT	OUTPUT	CODEBOOK
$bcababbcbcbaaaabbc$		$\{a = 1, b = 2, c = 3\}$

Nel seguito, per rendere la comprensione quanto più semplice possibile, rappresenteremo il processo di codifica *step by step*, ossia riportando i singoli passi che l'algoritmo compie all'interno di una tabella (da leggere dall'alto verso il basso).

[4] L'esempio potrebbe essere più attinente alla realtà codificando le varie lettere col loro corrispettivo codice ASCII, $\{\dots a = 97, b = 98, c = 99 \dots\}$, per poi proseguire inserendo i nuovi elementi a partire dalla posizione 128 (considerando una codifica ASCII a 7 bit). In generale è tuttavia bene affermare che la logica dell'algoritmo non impone obbligatoriamente l'utilizzo di un valore numerico, sebbene l'uso del valore dell'indice per la singola sottostringa codificata rappresenti sicuramente il modo più semplice d'utilizzo.

INPUT	OUTPUT	CODEBOOK
$\underbrace{\boldsymbol{b}c}\,ababbcbcbaaaabbc$	**2**	$\{a = 1, b = 2, c = 3, \boldsymbol{bc} = 4\}$
$b\,\underbrace{\boldsymbol{c}a}\,babbcbcbaaaabbc$	2, **3**	$\{a = 1, b = 2, c = 3, bc = 4, \boldsymbol{ca} = 5\}$
$bc\,\underbrace{\boldsymbol{a}b}\,abbcbcbaaaabbc$	2,3, **1**	$\begin{Bmatrix} a = 1, b = 2, c = 3, bc = 4, ca = 5, \\ \boldsymbol{ab} = 6 \end{Bmatrix}$
$bca\,\underbrace{\boldsymbol{b}a}\,bbcbcbaaaabbc$	2,3,1, **2**	$\begin{Bmatrix} a = 1, b = 2, c = 3, bc = 4, ca = 5, \\ ab = 6, \boldsymbol{ba} = 7 \end{Bmatrix}$
$bcab\,\underbrace{\boldsymbol{ab}b}\,cbcbaaaabbc$	2,3,1,2, **6**	$\begin{Bmatrix} a = 1, b = 2, c = 3, bc = 4, ca = 5, \\ ab = 6, ba = 7, \boldsymbol{abb} = 8 \end{Bmatrix}$
$bcabab\,\underbrace{\boldsymbol{bc}b}\,cbaaaabbc$	2,3,1,2,6, **4**	$\begin{Bmatrix} a = 1, b = 2, c = 3, bc = 4, ca = 5, \\ ab = 6, ba = 7, abb = 8, \boldsymbol{bcb} = 9 \end{Bmatrix}$
$bcababbc\,\underbrace{\boldsymbol{bcb}a}\,aaabbc$	2,3,1,2,6,4, **9**	$\begin{Bmatrix} a = 1, b = 2, c = 3, bc = 4, ca = 5, \\ ab = 6, ba = 7, abb = 8, bcb = 9, \\ \boldsymbol{bcba} = 10 \end{Bmatrix}$
$bcababbcbcb\,\underbrace{\boldsymbol{a}a}\,aabbc$	2,3,1,2,6,4,9, **1**	$\begin{Bmatrix} a = 1, b = 2, c = 3, bc = 4, ca = 5, \\ ab = 6, ba = 7, abb = 8, bcb = 9, \\ bcba = 10, \boldsymbol{aa} = 11 \end{Bmatrix}$
$bcababbcbcba\,\underbrace{\boldsymbol{aa}a}\,bbc$	2,3,1,2,6,4,9,1, **11**	$\begin{Bmatrix} a = 1, b = 2, c = 3, bc = 4, ca = 5, \\ ab = 6, ba = 7, abb = 8, bcb = 9, \\ bcba = 10, aa = 11, \boldsymbol{aaa} = 12 \end{Bmatrix}$
$bcababbcbcbaaa\,\underbrace{\boldsymbol{abb}c}$	2,3,1,2,6,4,9,1,11, **8**	$\begin{Bmatrix} a = 1, b = 2, c = 3, bc = 4, ca = 5, \\ ab = 6, ba = 7, abb = 8, bcb = 9, \\ bcba = 10, aa = 11, aaa = 12, \boldsymbol{abbc} = 13 \end{Bmatrix}$
$bcababbcbcbaaaabb\boldsymbol{c}$	2,3,1,2,6,4,9,1,11,8, **3**	$\begin{Bmatrix} a = 1, b = 2, c = 3, bc = 4, ca = 5, \\ ab = 6, ba = 7, abb = 8, bcb = 9, \\ bcba = 10, aa = 11, aaa = 12, abbc = 13 \end{Bmatrix}$

Come si può notare dall'esempio, ad ogni passo si produce in uscita una parte del codice, si scorre in avanti nell'analisi della stringa in ingresso e si aggiunge un nuovo valore al codebook. Logicamente, il codice in uscita avrà sempre un numero di simboli minore o al più uguale alla stringa in ingresso, ed è banale affermare che per ottenere una buona compressione è necessario che i dati in input contengano numerose ripetizioni. Nell'esempio precedente il rapporto di compressione è pari ad

$$\left(1 - \frac{11}{18}\right) * 100 \approx 39\%$$

tuttavia, come dimostrato dagli stessi Lempel e Ziv, tale valore tende al massimo possibile all'aumentare della lunghezza della stringa in ingresso.

1.4. Pseudocodice di codifica

Il processo di codifica dell'esempio precedente può essere generalizzato utilizzando un alfabeto $\mathcal{A} = \{a_1, a_2, \dots, a_k\}$, una sequenza d'ingresso $\mathcal{X} = \{x_1, x_2, \dots, x_n\}$ che risulti definita su $\mathcal{A}$, ed una sequenza d'uscita $\mathcal{Y} = \{y_1, y_2, \dots, y_m\}$, dove ogni y_i rappresenti l'indice i-esimo di un codebook $\mathcal{C}$ da cui prelevare le sottostringhe. Lo pseudocodice rappresentante l'algoritmo LZW sarà allora:

```
algoritmoLZWCodifica(sequenza X) → sequenza codificata

1.      C ← codebook vuoto
2.      Y ← sequenza vuota
3.      for(i ← 1 to k)
4.             C[i] ← ai   //inizializzo il codebook con l'alfabeto noto
5.      end
6.      j ← k + 1
7.      i ← 1
8.      while(i ≤ N)
9.             buffer ← X[i]
10.            while(i < N && buffer + X[i + 1] ∈ C)
11.                   buffer = buffer + X[i + 1]
12.                   i++
13.            end
14.            Y[j - k] ← indice all'interno del codebook della sottostringa
                         individuata dal buffer
15.            if(i < N)
16.                   C[j] ← buffer + X[i + 1]
17.            end
18.            j++
19.            i++
20.     end
21.     return Y
```

1.5. Processo di decodifica

Il processo di decodifica risulta un po' più complesso, poiché, come vedremo, necessita di una particolare chiarificazione in alcuni casi specifici. Per rendere il tutto più semplice utilizziamo la codifica in uscita ricavata nell'esempio visto e cerchiamo di ricostruire la stringa iniziale. Ovviamente, per quanto detto, il codebook sarà inizialmente formato dalle stesse uguaglianze utilizzate in precedenza, ossia $\{a = 1, b = 2, c = 3\}$, che dovranno essere note.

INPUT	OUTPUT	CODEBOOK
2, 3,1,2,6,4,9,1,11,8,3	$\boldsymbol{b}$	$\{a = 1, b = 2, c = 3\}$
2, **3**, 1,2,6,4,9,1,11,8,3	$\underbrace{b\boldsymbol{c}}$	$\{a = 1, b = 2, c = 3, \boldsymbol{bc} = 4\}$
2,3, **1**, 2,6,4,9,1,11,8,3	$b\,\underbrace{c\boldsymbol{a}}$	$\{a = 1, b = 2, c = 3, bc = 4, \boldsymbol{ca} = 5\}$

2,3,1, **2**, 6,4,9,1,11,8,3	$bc\ \underbrace{a\boldsymbol{b}}$	$\{a=1,b=2,c=3,bc=4,ca=5,\boldsymbol{ab}=6\}$
2,3,1,2, **6**, 4,9,1,11,8,3	$bca\ \underbrace{b\boldsymbol{a}}\ \boldsymbol{b}$	$\{a=1,b=2,c=3,bc=4,ca=5,ab=6,\boldsymbol{ba}=7\}$
2,3,1,2,6, **4**, 9,1,11,8,3	$bcab\ \underbrace{ab\boldsymbol{b}}\ \boldsymbol{c}$	$\{a=1,b=2,c=3,bc=4,ca=5,ab=6,ba=7,\boldsymbol{abb}=8\}$
2,3,1,2,6,4, **9**, 1,11,8,3	$bcabab\ \underbrace{bc\ ?}$	$\{a=1,b=2,c=3,bc=4,ca=5,ab=6,ba=7,abb=8\}$

Ci troviamo, a questo punto, dinnanzi ad uno di quei casi specifici di cui si è parlato in precedenza. Come possiamo osservare, infatti, il codebook non contiene ancora il simbolo 9, e pertanto sembrerebbe quasi che ci si trovi dinnanzi all'impossibilità di decodificare il tutto. Se riflettiamo un attimo sul processo di codifica precedente, ci rendiamo però facilmente conto del fatto che ciò accade solo in una particolare occasione, ossia quando un simbolo, codificato al passo p, viene utilizzato (nel processo di codifica) al passo immediatamente successivo (passo $p+1$). Questo però vuol dire che il simbolo successivo, di cui sembrava non conoscessimo nulla, sarà certamente composto (nella sua parte iniziale) dall'ultimo simbolo individuato; a cui dovrà poi essere aggiunto un solo singolo simbolo. In altre parole, nell'esempio proposto, avremo che $\boldsymbol{bc}?=9$, e pertanto potremmo inserire tale informazione nel nostro codebook, ottenendo

2,3,1,2,6,4, **9**, 1,11,8,3	$bcabab\ \underbrace{bc\ \boldsymbol{b}}\ \boldsymbol{c}?$	$\{a=1,b=2,c=3,bc=4,ca=5,ab=6,ba=7,abb=8,\boldsymbol{bc}?=9\}$

Il passo appena compiuto, come si vede, ci permette d'individuare il simbolo che ci mancava e pertanto di proseguire con la decodifica. Il fatto di codificare le sottostringhe all'interno del codebook aggiungendo sempre un <u>singolo</u> simbolo successivo al precedente, ci permette quindi di avere una decodifica coerente; che si dimostra cioè sempre effettuabile[5]. Nel seguito della decodifica si potrà osservare un altro caso simile a quello presentato, in cui, però, la sottostringa da cui ricavare la nuova associazione all'interno del codebook è composta da un solo simbolo (cioè a). Proseguendo abbiamo:

2,3,1,2,6,4, **9**, 1,11,8,3	$bcabab\ \underbrace{bc\boldsymbol{b}}\ \boldsymbol{cb}$	$\{a=1,b=2,c=3,bc=4,ca=5,ab=6,ba=7,abb=8,\boldsymbol{bcb}=9\}$
2,3,1,2,6,4,9, **1**, 11,8,3	$bcababbc\ \underbrace{bcb\boldsymbol{a}}$	$\{a=1,b=2,c=3,bc=4,ca=5,ab=6,ba=7,abb=8,bcb=9,\boldsymbol{bcba}=10\}$

[5] A rigore è necessario precisare che la codifica risulta coerente se ogni simbolo dell'alfabeto viene trattato effettivamente come un singolo simbolo. Se l'alfabeto è composto da simboli che rappresentano insiemi di alcuni sottosimboli allora sarà necessario porre estrema attenzione. Supponendo ad esempio di avere un alfabeto composto da simboli doppi, del tipo $\mathcal{A}=\{aa,bb,cc\}$, ed una stringa in ingresso del genere $\mathcal{X}=aaccbbaacc$, allora risulterà certamente conveniente trasformare il tutto ponendo $\mathcal{A}=\{\alpha,\beta,\gamma\}$ ed $\mathcal{X}=\alpha\gamma\beta\alpha\gamma$, per non cedere alla tentazione errata di aggiungere nel codebook qualcosa del tipo aac.

2,3,1,2,6,4,9,1, **11**, 8,3	$bcababbcbcb\,\underbrace{a}\,?$	$\begin{Bmatrix} a=1, b=2, c=3, bc=4, ca=5, \\ ab=6, ba=7, abb=8, bcb=9, \\ bcba=10 \end{Bmatrix}$
2,3,1,2,6,4,9,1, **11**, 8,3	$bcababbcbcb\,\underbrace{a\,\boldsymbol{a}}\,?$	$\begin{Bmatrix} a=1, b=2, c=3, bc=4, ca=5, \\ ab=6, ba=7, abb=8, bcb=9, \\ bcba=10, \boldsymbol{a}?=11 \end{Bmatrix}$
2,3,1,2,6,4,9,1, **11**, 8,3	$bcababbcbcb\,\underbrace{a\boldsymbol{a}}\,\boldsymbol{a}$	$\begin{Bmatrix} a=1, b=2, c=3, bc=4, ca=5, \\ ab=6, ba=7, abb=8, bcb=9, \\ bcba=10, \boldsymbol{aa}=11 \end{Bmatrix}$
2,3,1,2,6,4,9,1,11, **8**, 3	$bcababbcbcba\,\underbrace{aa\,\boldsymbol{a}}\,\boldsymbol{bb}$	$\begin{Bmatrix} a=1, b=2, c=3, bc=4, ca=5, \\ ab=6, ba=7, abb=8, bcb=9, \\ bcba=10, aa=11, \boldsymbol{aaa}=12 \end{Bmatrix}$
2,3,1,2,6,4,9,1,11,8, **3**	$bcababbcbcbaaaabbc$	$\begin{Bmatrix} a=1, b=2, c=3, bc=4, ca=5, \\ ab=6, ba=7, abb=8, bcb=9, \\ bcba=10, aa=11, aaa=12 \end{Bmatrix}$

1.6. Pseudocodice di decodifica

Utilizzando i medesimi valori generalizzati adoperati nello pseudocodice di codifica, precedentemente realizzato, è possibile ottenere una versione della decodifica come:

```
algoritmoLZWDecodifica(sequenza codificata Y) → sequenza
1.     C ← codebook vuoto
2.     X ← sequenza vuota
3.     for(i ← 1 to k)
4.         C[i] ← ai   //inizializzo il codebook con l'alfabeto noto
5.     end
6.     j ← k + 1
7.     i ← 1
8.     d ← 0
9.     buffer ← insieme vuoto
10.    while(i ≤ M)
11.        if(Y[i] ∉ C)
12.            C[j] ← buffer + primo simbolo in buffer
13.            j++
14.        end
15.        L ← lunghezza di C[Y[i]]
16.        for(m ← 1 to L)
17.            X[d + m] ← C[Y[i]][m]
18.        end
19.        if(buffer + X[d] ∉ C)
20.            C[j] ← buffer + X[d]
21.            j++
22.        end
23.        buffer ← C[Y[i]]
24.        d ← d + L
25.        i++
26.    end
27.    return X
```

1.7. La codifica all'indietro

Un diverso modo di procedere, che annulli l'eventualità di dover gestire, durante la decodifica, un valore non ancora presente all'interno del codebook, può essere quello di effettuare la codifica osservando il codice *all'indietro*. In altre parole, nel processo di codifica, invece d'inserire all'interno del codebook i nuovi valori considerando il simbolo immediatamente successivo a quello appena codificato, si osserveranno i valori già codificati, tornando indietro fino ad ottenere la prima sottostringa non ancora presente all'interno del codebook. Riprendendo l'esempio precedente si ha:

INPUT	OUTPUT	CODEBOOK
$\underbrace{\boldsymbol{b}}\,ababbcbcbaaaabbc$	$\boldsymbol{2}$	$\{a=1, b=2, c=3\}$
$\underbrace{b\boldsymbol{c}}\,ababbcbcbaaaabbc$	$2, \boldsymbol{3}$	$\{a=1, b=2, c=3, \boldsymbol{bc}=4\}$
$b\,\underbrace{c\boldsymbol{a}}\,babbcbcbaaaabbc$	$2,3,\boldsymbol{1}$	$\{a=1, b=2, c=3, bc=4, \boldsymbol{ca}=5\}$
$bc\,\underbrace{a\boldsymbol{b}}\,abbcbcbaaaabbc$	$2,3,1,\boldsymbol{2}$	$\{a=1, b=2, c=3, bc=4, ca=5, \boldsymbol{ab}=6\}$
$bca\,\underbrace{b\boldsymbol{ab}}\,bcbcbaaaabbc$	$2,3,1,2,\boldsymbol{6}$	$\{a=1, b=2, c=3, bc=4, ca=5, ab=6, \boldsymbol{bab}=7\}$
$bcaba\,\underbrace{b\boldsymbol{bc}}\,bcbaaaabbc$	$2,3,1,2,6,\boldsymbol{4}$	$\{a=1, b=2, c=3, bc=4, ca=5, ab=6, bab=7, \boldsymbol{bbc}=8\}$
$bcababb\,\underbrace{c\boldsymbol{bc}}\,baaaabbc$	$2,3,1,2,6,4,\boldsymbol{4}$	$\{a=1, b=2, c=3, bc=4, ca=5, ab=6, bab=7, bbc=8, \boldsymbol{cbc}=9\}$
$bcababbcb\,\underbrace{c\boldsymbol{b}}\,aaaabbc$	$2,3,1,2,6,4,4,\boldsymbol{2}$	$\{a=1, b=2, c=3, bc=4, ca=5, ab=6, bab=7, bbc=8, cbc=9, \boldsymbol{cb}=10\}$
$bcababbcbc\,\underbrace{b\boldsymbol{a}}\,aaabbc$	$2,3,1,2,6,4,4,2,\boldsymbol{1}$	$\{a=1, b=2, c=3, bc=4, ca=5, ab=6, bab=7, bbc=8, cbc=9, cb=10, \boldsymbol{ba}=11\}$
$bcababbcbcb\,\underbrace{a\boldsymbol{a}}\,aabbc$	$2,3,1,2,6,4,4,2,1,\boldsymbol{1}$	$\{a=1, b=2, c=3, bc=4, ca=5, ab=6, bab=7, bbc=8, cbc=9, cb=10, ba=11, \boldsymbol{aa}=12\}$
$bcababbcbcba\,\underbrace{a\boldsymbol{aa}}\,bbc$	$2,3,1,2,6,4,4,2,1,1,\boldsymbol{12}$	$\{a=1, b=2, c=3, bc=4, ca=5, ab=6, bab=7, bbc=8, cbc=9, cb=10, ba=11, aa=12, \boldsymbol{aaa}=13\}$
$bcababbcbcbaaaa\boldsymbol{bbc}$	$2,3,1,2,6,4,4,2,1,1,12,\boldsymbol{8}$	$\{a=1, b=2, c=3, bc=4, ca=5, ab=6, bab=7, bbc=8, cbc=9, cb=10, ba=11, aa=12, aaa=13\}$

Nel caso particolare, come si può notare, il rapporto di compressione si abbassa a circa il 33%, ma in generale tale modifica non comporta alcun cambiamento nelle prestazioni, che comunque all'aumentare della stringa in ingresso tenderanno sempre al massimo ottenibile.

1.8. Pseudocodice di codifica all'indietro

Il processo di codifica all'indietro, riprendendo le generalizzazioni già utilizzate in precedenza, può essere rappresentato attraverso lo pseudocodice seguente:

```
algoritmoLZWCodificaIndietro(sequenza X) → sequenza codificata

1.    C ← codebook vuoto
2.    Y ← sequenza vuota
3.    for(i ← 1 to k)
4.        C[i] ← ai   //inizializzo il codebook con l'alfabeto noto
5.    end
6.    j ← k + 1
7.    i ← 1
8.    while(i ≤ N)
9.        buffer ← X[i]
10.       while(i < N && buffer + X[i + 1] ∈ C)
11.           buffer = buffer + X[i + 1]
12.           i++
13.       end
14.       Y[j - k] ← indice all'interno del codebook della sottostringa
                     individuata dal buffer
15.       if(i < N)
16.           m ← i
17.           while(X[m - 1] + buffer ∈ C)
18.               m--
19.           end
20.           C[j] ← X[m - 1] + buffer
21.       end
22.       j++
23.       i++
24.   end
25.   return Y
```

Dal codice si può notare come esista una piccola precisazione, nel processo di codifica, che vale la pena enunciare. Potrebbe capitare, infatti, che nel guardare all'indietro s'incontri una sottostringa già presente all'interno del codebook, e quindi occorrerà decidere come comportarsi in tali situazioni. Supponiamo ad esempio che il nostro flusso in ingresso sia formato dalla stringa $aaabc$, otterremo:

INPUT	OUTPUT	CODEBOOK
$\underbrace{\boldsymbol{a}}aabc$	**1**	$\{a = 1, b = 2, c = 3\}$
$\underbrace{a\boldsymbol{a}}abc$	$1, \boldsymbol{1}$	$\{a = 1, b = 2, c = 3, \boldsymbol{aa} = 4\}$
$\underbrace{aa\boldsymbol{a}}bc$	$1,1, \boldsymbol{1}$	$\{a = 1, b = 2, c = 3, aa = 4, \boldsymbol{aaa} = 5\}$

Al punto in cui ci si è arrestati ci si accorge che la sottostringa aa da dover aggiungere al codebook si è già incontrata in precedenza. Le soluzioni in questo caso possono essere due: non considerare affatto tale sottostringa, proseguendo nella codifica, oppure continuare la ricerca all'indietro.

Come si può notare l'algoritmo sceglie sempre di continuare all'indietro fin quando non s'incontri una sottostringa non nota (in questo caso aaa), ma nulla vieta di utilizzare un diverso modo di procedere.

1.9. Processo di decodifica per una codifica effettuata all'indietro

Il processo di decodifica si semplifica decisamente nel caso la codifica sia effettuata all'indietro. In tal caso, infatti, non ci sarà la possibilità di visualizzare un codice che non sia ancora presente nel codebook, dato che ogni codifica è effettuata appunto "guardando" all'indietro, e non in avanti. Ciò, quindi, garantisce la coerenza della decodifica in ogni caso.

INPUT	OUTPUT	CODEBOOK
2, 3,1,2,6,4,4,2,1,1,12,8	$\boldsymbol{b}$	$\{a = 1, b = 2, c = 3\}$
2, **3**, 1,2,6,4,4,2,1,1,12,8	$\underbrace{b\boldsymbol{c}}$	$\{a = 1, b = 2, c = 3, \boldsymbol{bc} = 4\}$
2,3, **1**, 2,6,4,4,2,1,1,12,8	$b\,\underbrace{c\boldsymbol{a}}$	$\{a = 1, b = 2, c = 3, bc = 4, \boldsymbol{ca} = 5\}$
2,3,1, **2**, 6,4,4,2,1,1,12,8	$bc\,\underbrace{a\boldsymbol{b}}$	$\{a = 1, b = 2, c = 3, bc = 4, ca = 5, \boldsymbol{ab} = 6\}$
2,3,1,2, **6**, 4,4,2,1,1,12,8	$bca\,\underbrace{b\boldsymbol{ab}}$	$\{a = 1, b = 2, c = 3, bc = 4, ca = 5, ab = 6, \boldsymbol{bab} = 7\}$
2,3,1,2,6, **4**, 4,2,1,1,12,8	$bcaba\,\underbrace{b\boldsymbol{bc}}$	$\{a = 1, b = 2, c = 3, bc = 4, ca = 5, ab = 6, bab = 7, \boldsymbol{bbc} = 8\}$
2,3,1,2,6,4, **4**, 2,1,1,12,8	$bcababb\,\underbrace{c\boldsymbol{bc}}$	$\{a = 1, b = 2, c = 3, bc = 4, ca = 5, ab = 6, bab = 7, bbc = 8, \boldsymbol{cbc} = 8\}$
2,3,1,2,6,4,4, **2**, 1,1,12,8	$bcababbcb\,\underbrace{c\boldsymbol{b}}$	$\{a = 1, b = 2, c = 3, bc = 4, ca = 5, ab = 6, bab = 7, bbc = 8, cbc = 9, \boldsymbol{cb} = 10\}$
2,3,1,2,6,4,4,2, **1**, 1,12,8	$bcababbcbc\,\underbrace{b\boldsymbol{a}}$	$\{a = 1, b = 2, c = 3, bc = 4, ca = 5, ab = 6, bab = 7, bbc = 8, cbc = 9, cb = 10, \boldsymbol{ba} = 11\}$
2,3,1,2,6,4,4,2,1, **1**, 12,8	$bcababbcbcb\,\underbrace{a\boldsymbol{a}}$	$\{a = 1, b = 2, c = 3, bc = 4, ca = 5, ab = 6, bab = 7, bbc = 8, cbc = 9, cb = 10, ba = 11, \boldsymbol{aa} = 12\}$
2,3,1,2,6,4,4,2,1,1, **12**, 8	$bcababbcbcba\,\underbrace{a\boldsymbol{aa}}$	$\{a = 1, b = 2, c = 3, bc = 4, ca = 5, ab = 6, bab = 7, bbc = 8, cbc = 9, cb = 10, ba = 11, aa = 12, \boldsymbol{aaa} = 13\}$
2,3,1,2,6,4,4,2,1,1,12, **8**	$bcababbcbcbaaaa\boldsymbol{bbc}$	$\{a = 1, b = 2, c = 3, bc = 4, ca = 5, ab = 6, bab = 7, bbc = 8, cbc = 9, cb = 10, ba = 11, aa = 12, aaa = 13\}$

1.10. Pseudocodice di decodifica per la codifica effettuata all'indietro

Lo pseudocodice della decodifica, per una codifica effettuata all'indietro, risulta essere:

```
  algoritmoLZWDecodificaIndietro(sequenza codificata Y) → sequenza
1.       C ← codebook vuoto
2.       X ← sequenza vuota
3.       for(i ← 1 to k)
4.             C[i] ← ai   //inizializzo il codebook con l'alfabeto noto
5.       end
6.       j ← k + 1
7.       i ← 1
8.       d ← 0
9.       buffer ← insieme vuoto
10.      while(i ≤ M)
11.            L ← lunghezza di C[Y[i]]
12.            for(m ← 1 to L)
13.                  X[d + m] ← C[Y[i]][m]
14.            end
15.            m ← 1
16.            if not(d = 0)
17.                  buffer ← X[d] + C[Y[i]]
18.                  while(m < d && buffer ∈ C)
19.                        buffer ← X[d - m] + buffer
20.                        m++
21.                  end
22.                  if(buffer ∉ C)
23.                        C[j] ← buffer
24.                        j++
25.                  end
26.            end
27.            d ← d + L
28.            i++
29.      end
30.      return X
```

1.11. Esercizio

Sarebbe interessante eseguire un confronto tra l'algoritmo LZW originale e la versione modificata con codifica all'indietro, sebbene la possibilità delle due versione di poter codificare meglio o peggio una data stringa dipenda ovviamente dal tipo di stringa. In generale, come già ribadito, estendendo il flusso in ingresso si può dimostrare che entrambi gli algoritmi tendono ad ottenere il massimo livello di compressione possibile. Per fornire un caso di esempio in cui la codifica all'indietro si comporti meglio rispetto all'originale, si propone di eseguire un esercizio simile a quelli fin qui realizzati, dove la stringa in ingresso sia definita dalla sequenza:

$$bbcabcabcbbcbbbcbb$$

Si potrà facilmente calcolare che l'algoritmo originale produrrà in uscita la codifica {2,2,3,1,5,7,3,4,10,11,4}, che mostra un rapporto di compressione pari ancora al 39%, mentre la codifica all'indietro produce {2,2,3,1,5,7,4,9,10}, con rapporto di compressione del 50%.

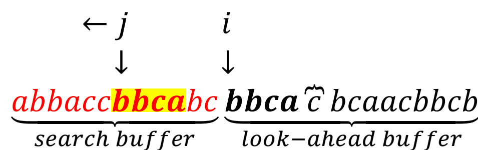
CAPITOLO 2

Gli algoritmi LZ77 ed LZ78

Si fornirà di seguito una semplice spiegazione degli algoritmi LZ77 ed LZ78, proposti da Lempel e Ziv rispettivamente all'interno delle pubblicazioni del 1977 e del 1978. Come detto in precedenza tali algoritmi oggi risultano obsoleti, tuttavia il principio introdotto dagli stessi è utilizzato in una miriade di versioni alternative, che definiscono non pochi standard di compressione (come **ZIP**, **GZIP**, ecc..).

2.1. L'algoritmo LZ77

L'idea alla base dell'algoritmo LZ77 è quella secondo la quale, data una sorgente $\mathcal{S}$ che genera una sequenza di simboli $\mathcal{X}$ appartenenti ad un alfabeto $\mathcal{A}$, la porzione già codificata della sequenza possa essere utilizzata come base attraverso cui codificare il resto. La codifica avviene quindi utilizzando una *finestra scorrevole* (sliding-window) composta da due parti: un *search buffer*, contenente la porzione già codificata, ed un *look-ahead buffer*, contenente il segmento ancora da codificare. La separazione della finestra è ovviamente ottenuta attraverso un puntatore i, che identifica l'inizio del look-ahead buffer. Un secondo puntatore j, partendo da i, esaminerà il search buffer cercando la più lunga sequenza (se esiste) che possa rappresentare un prefisso per il look-ahead buffer stesso.



Il codificatore produrrà in uscita sempre una *tripla* (o, l, s), dove:

- o rappresenta la distanza tra i due puntatori i e j (detta *offset*);
- l rappresenta la lunghezza del prefisso individuato;
- s rappresenta il simbolo che compare nel look-ahead buffer subito dopo il prefisso.

Ovviamente, se non esiste alcuna sottostringa nel search buffer che inizi con il primo simbolo del look-ahead buffer, il codificatore produrrà una tripla $(0,0,s)$, dove s rappresenterà in questo caso proprio il primo simbolo del look-ahead buffer stesso. La codifica ottenuta dall'algoritmo LZ77, essendo rappresentata dalla concatenazione delle triple appena viste, fornirà una vera compressione solo nel caso in cui i prefissi individuati si dimostrino abbastanza lunghi[6].

[6] I prefissi dovranno infatti far sì che le triple possano essere rappresentate con un numero di bit inferiore rispetto alla codifica banale dei singoli elementi della stringa in ingresso.

2.2. Processo di codifica dell'LZ77

Mostreremo di seguito un esempio del processo di codifica, scegliendo una particolare stringa d'ingresso che ci permetta di porre l'attenzione su un punto fondamentale. Supponiamo quindi che il nostro alfabeto sia formato da due soli simboli $\mathcal{A} = \{a, b\}$ e che la stringa rappresentante il flusso in ingresso sia la seguente:

$$babbababbaabbaabaabaaa$$

INPUT	OUTPUT
$\underbrace{babbababbaabbaabaabaaa}_{look-ahead\ buffer}$	$(0,0,b)$
$\underbrace{b}_{search\ buffer}$ $\underbrace{abbababbaabbaabaabaaa}_{look-ahead\ buffer}$	$(0,0,b)\ (0,0,a)$
$\underbrace{ba}_{search\ buffer}$ $\underbrace{bbababbaabbaabaabaaa}_{look-ahead\ buffer}$	$(0,0,b)\ (0,0,a)\ (2,1,b)$
$\underbrace{babb}_{search\ buffer}$ $\underbrace{ababbaabbaabaabaaa}_{look-ahead\ buffer}$	$(0,0,b)\ (0,0,a)\ (2,1,b)\ (3,2,a)$
$\underbrace{babbaba}_{search\ buffer}$ $\underbrace{bbaabbaabaabaaa}_{look-ahead\ buffer}$	$(0,0,b)\ (0,0,a)\ (2,1,b)\ (3,2,a)\ (5,3,a)$
$\underbrace{babbababbaa}_{search\ buffer}$ $\underbrace{bbaabaabaaa}_{look-ahead\ buffer}$	$(0,0,b)\ (0,0,a)\ (2,1,b)\ (3,2,a)\ (5,3,a)\ (4,4,b)$
$\underbrace{babbababbaabbaab}_{search\ buffer}$ $\underbrace{\widehat{aa}\ baaa}_{look-ahead\ buffer}$	$(0,0,b)\ (0,0,a)\ (2,1,b)\ (3,2,a)$ $(5,3,a)\ (4,4,b)\ (3,5,a)$

L'ultimo passaggio eseguito merita un'analisi più attenta. Sembrerebbe infatti che il simbolo *aab* presente nel search buffer non sia sufficiente a codificare i cinque simboli presenti nel look-ahead buffer (o, addirittura, che si utilizzi parte del look-ahead buffer per codificare il simbolo). Durante il processo di decodifica capiremo come ciò sia possibile, per il momento è importante sottolineare che non è assolutamente detto che l'offset debba essere necessariamente superiore alla lunghezza del prefisso.

2.3. Processo di decodifica dell'LZ77

Proviamo quindi a risalire dalla codifica alla stringa originale, ottenendo:

INPUT	OUTPUT
$(0,0,\boldsymbol{b})\ (0,0,a)\ (2,1,b)\ (3,2,a)\ (5,3,a)\ (4,4,b)\ (3,5,a)$	$\vert\boldsymbol{b}$
$(0,0,b)\ (0,0,\boldsymbol{a})\ (2,1,b)\ (3,2,a)\ (5,3,a)\ (4,4,b)\ (3,5,a)$	$b\vert\boldsymbol{a}$
$(0,0,b)\ (0,0,a)\ (2,1,\boldsymbol{b})\ (3,2,a)\ (5,3,a)\ (4,4,b)\ (3,5,a)$	$\overbrace{\boldsymbol{b}}\,a\,\vert\underbrace{b}\,\boldsymbol{b}$
$(0,0,b)\ (0,0,a)\ (2,1,b)\ (3,2,\boldsymbol{a})\ (5,3,a)\ (4,4,b)\ (3,5,a)$	$b\,\overbrace{\boldsymbol{ab}}\,b\,\vert\underbrace{ab}\,\boldsymbol{a}$
$(0,0,b)\ (0,0,a)\ (2,1,b)\ (3,2,a)\ (5,3,\boldsymbol{a})\ (4,4,b)\ (3,5,a)$	$ba\,\overbrace{\boldsymbol{bba}}\,ba\,\vert\underbrace{bba}\,\boldsymbol{a}$
$(0,0,b)\ (0,0,a)\ (2,1,b)\ (3,2,a)\ (5,3,a)\ (4,4,\boldsymbol{b})\ (3,5,a)$	$babbaba\,\overbrace{\boldsymbol{bbaa}}\,\vert\underbrace{bbaa}\,\boldsymbol{b}$

Per l'ultimo passo, ossia:

$(0,0,b)\ (0,0,a)\ (2,1,b)\ (3,2,a)\ (5,3,a)\ (4,4,b)\ (3,5,\boldsymbol{a})$	$babbababbaabb\,\overbrace{\boldsymbol{aab}}\,\vert\underbrace{\overbrace{aa}\,baa}\,\boldsymbol{a}$

bisognerà espletare un po' più in dettaglio cosa accade (per comprendere meglio non solo perché è stato codificato in tal modo ma anche come può la decodifica essere corretta). In realtà il concetto fondamentale da cogliere è che l'algoritmo non effettua la copia del prefisso in blocco, ma *simbolo per simbolo*. Il riconoscimento del prefisso nell'ultimo passo avviene quindi in tal modo:

$$\overbrace{\boldsymbol{aab}}??\rightarrow\overbrace{\underbrace{a}\,\boldsymbol{ab}\,\underbrace{a}}\,?\rightarrow\overbrace{a\,\underbrace{a}\,\boldsymbol{b}}\,\boldsymbol{a}\,\underbrace{a}\rightarrow\overbrace{\boldsymbol{aa}\,\underbrace{b}}\,\boldsymbol{aa}\,\underbrace{b}\rightarrow\overbrace{\boldsymbol{aab}\,\underbrace{a}}\,\boldsymbol{a}\,\boldsymbol{b}\,\underbrace{a}\rightarrow\overbrace{\boldsymbol{aab}\,a\,\underbrace{a}}\,\boldsymbol{ba}\,\underbrace{a}$$

Si può quindi affermare che esiste una regola di codifica che è la seguente:

Se il prefisso individuato termina al confine con il look-ahead buffer e, all'interno dello stesso, tale prefisso si ripete dall'inizio (anche se in parte), allora è possibile eseguire una codifica che tenga conto di tale ripetizione attraverso un offset inferiore alla lunghezza del prefisso stesso.

Per chiarire meglio, si può facilmente verificare che la codifica di

$\overbrace{\boldsymbol{abba}}\,\vert\underbrace{\overbrace{ab}\,baab}\,\boldsymbol{b}$ avviene attraverso la tripla $(4,6,\boldsymbol{b})$

così come quella di

$\overbrace{\boldsymbol{aaa}}\,\vert\underbrace{\overbrace{aaa}a\,aaa}\,\boldsymbol{a}$ avviene attraverso la tripla $(3,7,\boldsymbol{a})$

2.4. L'algoritmo LZ78

A differenza di quanto accade nell'LZ77, l'algoritmo LZ78 utilizza invece un vero e proprio *codebook* (dizionario). I limiti dell'LZ77 sono infatti determinati principalmente dall'impossibilità di mantenere puntatori troppo complessi, ovvero un search buffer troppo grande[7], e quindi dall'incapacità di sfruttare appieno la periodicità della sequenza. Utilizzando un codebook costruito in maniera dinamica, l'algoritmo LZ78 riduce sensibilmente gli effetti di tale problema[8].
Il codificatore produrrà in uscita sempre una $coppia$ (i, s) dove:

- i rappresenta l'indice, all'interno del codebook, della sottostringa da aggiungere;
- s rappresenta il simbolo da inserire subito dopo la sottostringa.

Il codebook sarà all'inizio costituito dalla sola stringa vuota "" (in posizione 0), a cui si aggiungeranno via via le varie sottostringhe incontrate lungo il processo di codifica. Questo modo di procedere permette di non dover inviare il dizionario insieme alla codifica, poiché anche a destinazione esso potrà essere ricreato in maniera dinamica. La comprensione dell'algoritmo LZW dovrebbe rendere ancor più chiari i vari passi eseguiti di seguito, per cui ci limiteremo alla semplice rappresentazione grafica dei processi di codifica e decodifica. Nell'esempio seguente supporremo che il nostro alfabeto sia $\mathcal{A} = \{a, b, c\}$ e che la stringa rappresentante il flusso in ingresso sia:

$$ccaccbcabcaba$$

2.5. Processo di codifica dell'LZ78

INPUT	OUTPUT	CODEBOOK
$\underbrace{\boldsymbol{c}}\,caccbcabcaba$	$(0, \boldsymbol{c})$	$\{"" = 0, \boldsymbol{c} = 1\}$
$c\,\underbrace{\boldsymbol{ca}}\,ccbcabcaba$	$(0, c)\ (1, \boldsymbol{a})$	$\{"" = 0, c = 1, \boldsymbol{ca} = 2\}$
$cca\,\underbrace{\boldsymbol{cc}}\,bcabcaba$	$(0, c)\ (1, a)\ (1, \boldsymbol{c})$	$\{"" = 0, c = 1, ca = 2, \boldsymbol{cc} = 3\}$
$ccacc\,\underbrace{\boldsymbol{b}}\,cabcaba$	$(0, c)\ (1, a)\ (1, c)\ (0, \boldsymbol{b})$	$\{"" = 0, c = 1, ca = 2, cc = 3, \boldsymbol{b} = 4\}$
$ccaccb\,\underbrace{\boldsymbol{cab}}\,caba$	$(0, c)\ (1, a)\ (1, c)\ (0, b)\ (2, \boldsymbol{b})$	$\{"" = 0, c = 1, ca = 2, cc = 3, b = 4, \boldsymbol{cab} = 5\}$
$ccaccbcab\,\underbrace{\boldsymbol{caba}}$	$(0, c)\ (1, a)\ (1, c)\ (0, b)\ (2, b)(5, \boldsymbol{a})$	$\{"" = 0, c = 1, ca = 2, cc = 3, b = 4, cab = 5, \boldsymbol{caba} = 6\}$

[7] Ci si deve infatti ricordare di far sì che la rappresentazione delle triple non necessiti di troppi bit.
[8] Tuttavia anche il codebook dell'algoritmo LZ78, così come quello dell'LZW, non può in ogni caso crescere troppo, altrimenti la rappresentazione degli indici a cui puntare potrebbe ridurre significativamente i benefici della codifica. Per risolvere tale problema si usano varie tecniche, che principalmente tendono o ad azzerare il codebook una volta raggiunto un certo livello (come nel GIF *classico*), oppure ad eliminare da esso le voci meno utilizzate.

È bene sottolineare che, a differenza di quanto accade con l'algoritmo LZW, l'algoritmo LZ78 non necessita affatto della conoscenza base dell'alfabeto, poiché i vari simboli sono inseriti direttamente all'interno delle coppie rappresentanti la codifica.

2.6. Processo di decodifica dell'LZ78

Utilizzando un processo diametralmente opposto a quello di codifica è quindi possibile decodificare l'insieme delle coppie, ottenendo:

INPUT	OUTPUT	CODEBOOK
$(0,\boldsymbol{c})\ (1,a)\ (1,c)\ (0,b)\ (2,b)(5,a)$	$\underbrace{\boldsymbol{c}}$	$\{"" = 0, \boldsymbol{c} = 1\}$
$(0,c)\ (1,\boldsymbol{a})\ (1,c)\ (0,b)\ (2,b)(5,a)$	$c\ \underbrace{\boldsymbol{ca}}$	$\{"" = 0, c = 1, \boldsymbol{ca} = 2\}$
$(0,c)\ (1,a)\ (1,\boldsymbol{c})\ (0,b)\ (2,b)(5,a)$	$cca\ \underbrace{\boldsymbol{cc}}$	$\{"" = 0, c = 1, ca = 2, \boldsymbol{cc} = 3\}$
$(0,c)\ (1,a)\ (1,c)\ (0,\boldsymbol{b})\ (2,b)(5,a)$	$ccacc\ \underbrace{\boldsymbol{b}}$	$\{"" = 0, c = 1, ca = 2, cc = 3, \boldsymbol{b} = 4\}$
$(0,c)\ (1,a)\ (1,c)\ (0,b)\ (2,\boldsymbol{b})(5,a)$	$ccaccb\ \underbrace{\boldsymbol{cab}}$	$\{"" = 0, c = 1, ca = 2, cc = 3, b = 4, \boldsymbol{cab} = 5\}$
$(0,c)\ (1,a)\ (1,c)\ (0,b)\ (2,b)(5,\boldsymbol{a})$	$ccaccbcab\ \underbrace{\boldsymbol{caba}}$	$\{"" = 0, c = 1, ca = 2, cc = 3, b = 4, cab = 5, \boldsymbol{caba} = 6\}$

Come si potrà facilmente intuire, l'intero processo di codifica e decodifica risulta sempre coerente.

BIBLIOGRAFIA

K. Sayood, *Introduction to Data Compression*, 3rd ed., Morgan Kaufmann, 2006

David Salomon, *Data Compression – The complete reference*, 4th ed., Springer, 2007

R. Sprugnoli, *Il metodo di compressone Lempel-Ziv-Welch*, 2005

Appunti dal corso di *Gestione ed elaborazione grandi moli di dati* del professor Andrea Caprina, per il corso di laurea specialistica in Ingegneria Informatica dell'università di Padova, 2006

Terry Welch, *A Technique for High-Performance Data Compression*, *IEEE Computer*, Giugno 1984, Vol. 17 n° 6, p. 8–19.