Alma Mater Studiorum · Università di Bologna

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
Corso di Laurea Magistrale in Matematica

Tecniche di compressione senza perdita per dati unidimensionali e bidimensionali

Tesi di Laurea Magistrale in Fisica Matematica

Relatore:
Chiar.mo Professore
Marco Lenci

Presentata da:
Chiara Marconi

Sessione terza
Anno Accademico
2011/2012

Indice

Introduzione

Nel corso degli ultimi decenni è avvenuta una rapida rivoluzione delle modalità e delle forme di comunicazione. La compressione dati ha avuto un ruolo determinante in questo processo che tuttora è in corso. L'enorme aumento della quantità di dati che i dispositivi di memoria possono contenere ha portato alla necessità di gestire quantità di dati sempre maggiori ed ha reso la compressione dati sempre più fondamentale. Lo scopo di questa tesi è quello di fornire i concetti base sulle tecniche di compressione dati, con particolare attenzione agli aspetti matematici di queste ultime.

Nel capitolo 1 si introduce la teoria dell'informazione, che costituisce la base teorica per tutti i metodi di compressione e che fornisce alcuni limiti al tasso di compressione realizzabile su una stringa di dati fissata.

Nel capitolo 2 vengono illustrati i principali metodi di compressione unidimensionale, implementati nei programmi più diffusi. Gli algoritmi di compressione unidimensionale vengono raggruppati in due principali categorie. La prima categoria comprende i metodi di compressione statistica, di cui fanno parte metodi di codifica tra i migliori conosciuti, come la codifica di Huffman, la cui idea è quella di codificare simboli comuni con pochi bit, mentre quelli più rari con parole di codice più lunghe. La seconda categoria comprende i metodi di sostituzione di testo o con dizionario, che sono basati sull'idea di rimpiazzare, in una stringa di dati, occorrenze di sequenze ripetute con puntatori a copie precedenti. Tra questi, viene presentato l'algoritmo di Lempel e Ziv LZ77, per il quale è data una dimostrazione rigorosa dell'ottimalità asintotica.

Nel capitolo 3 vengono descritti i metodi di compressione dati a due dimensioni. Anche questi ultimi possono essere divisi in due categorie: tecniche di compressione senza perdita di informazioni e con perdita di informazioni. Verranno illustrati i più importanti algoritmi di compressione dati bidimensionali senza perdita, implementati nella maggior parte dei formati comunemente utilizzati per la gestione di immagini digitali.

Nel capitolo 4 vengono presentati tre nuovi tentativi di implementazione di algoritmi di compressione dati bidimensionali per immagini a due livelli

di colore, bianco e nero. Infine i tre algoritmi vengono testati su diverse tipologie di immagini e confrontati fra loro.

Capitolo 1

Teoria dell'informazione

1.1 Teoria della probabilità

Definizione 1. *Dato un insieme Ω, si definisce σ-algebra su Ω una famiglia $\mathcal{F}$ di sottoinsiemi di Ω tale che:*

i. L'insieme Ω appartiene ad $\mathcal{F}$.

ii. Se un insieme A appartiene ad $\mathcal{F}$, allora anche il suo complementare appartiene ad $\mathcal{F}$.

iii. Se gli elementi A_i di una famiglia numerabile di insiemi $\{A_i\}_{i\in\mathbb{N}}$ appartengono ad $\mathcal{F}$, allora la loro unione:

$$A = \bigcup_{i=1}^{\infty} A_i$$

appartiene a $\mathcal{F}$.

Definizione 2. *Si dice spazio misurabile una coppia $(\Omega, \mathcal{F})$ dove Ω è un insieme ed $\mathcal{F}$ è una σ-algebra di sottoinsiemi di Ω. Gli elementi di $\mathcal{F}$ sono detti insiemi misurabili in Ω.*

Definizione 3. *Sia $(\Omega, \mathcal{F})$ uno spazio misurabile e $(\Omega', \mathcal{G})$ uno spazio topologico. Un'applicazione $f : \Omega \to \Omega'$ si dice misurabile se la controimmagine di ogni elemento di $\mathcal{G}$ appartiene ad $\mathcal{F}$. Formalmente*

$$f^{-1}(A) \in \mathcal{F} \qquad \forall A \in \mathcal{G}.$$

Definizione 4. *Si definisce misura una funzione* $\mu : \mathcal{F} \to [0, \infty]$ *definita sulla* σ*-algebra* $\mathcal{F}$ *di sottoinsiemi di un insieme* Ω*, con la proprietà della* σ*-additività, cioè* μ *deve essere tale che se* $A_1, A_2, \ldots \in \mathcal{F}$ *è una successione di insiemi due a due disgiunti di* $\mathcal{F}$ *vale*

$$\mu\left(\bigcup_k A_k\right) = \sum_k \mu(A_k).$$

Definizione 5. *Dato uno spazio misurabile* $(\Omega, \mathcal{F})$*, una misura di probabilità su* Ω *è una misura* $\mathbb{P}$ *sulla* σ*-algebra* $\mathcal{F}$ *tale che*

$$\mathbb{P}(\Omega) = 1.$$

Uno spazio di probabilità $(\Omega, \mathcal{F}, \mathbb{P})$ *è uno spazio misurabile* $(\Omega, \mathcal{F})$ *dotato di misura* $\mathbb{P}$*.*

Definizione 6. *Fissato uno spazio di probabilità* $(\Omega, \mathcal{F}, \mathbb{P})$ *si definisce variabile aleatoria una funzione misurabile* $X : \Omega \to \mathbb{R}$*.*

Definizione 7. *Sia* $(\Omega, \mathcal{F}, \mathbb{P})$ *uno spazio di probabilità, e sia* X *variabile aleatoria. Il valore atteso di* X *è dato dall'integrale di* X *rispetto alla misura di probabilità* $\mathbb{P}$*:*

$$\mathbb{E}[X] := \int_\Omega X(\omega) d\mathbb{P}(\omega).$$

Definizione 8. *Sia* $(\Omega, \mathcal{F}, \mathbb{P})$ *uno spazio di probabilità, si definisce processo stocastico una famiglia di variabili aleatorie indicizzate da un parametro* $t \in T$ *e lo si denota con* $\{X_t\}_{t \in T}$*. Il processo stocastico è detto a parametro discreto se* T *è discreto, mentre è detto a parametro continuo qualora* T *sia continuo.*

In particolare, per il nostro obiettivo, sarà sufficiente considerare solamente processi stocastici a parametro discreto.

Definizione 9. *Date due variabili aleatorie discrete* X *e* Y *si definisce la loro distribuzione congiunta come la distribuzione di probabilità associata al vettore* (X, Y)*:*

$$p(x, y) = p(y \mid x)p(x) = p(x \mid y)p(y).$$

Date due variabili aleatorie discrete X *e* Y*, la distribuzione condizionata di* Y *dato* X *è la probabilità di* Y *quando è noto il valore assunto da* X*. La distribuzione condizionata di* Y *nota* $X = x$*, è data da:*

$$p(y \mid x) = \frac{p(x, y)}{p(x)}. \tag{1.1}$$

Definizione 10. *Due variabili aleatorie X, Y si dicono stocasticamente indipendenti se, e solo se, si ha:*

$$p(y \mid x) = p(y) \qquad e \qquad p(x \mid y) = p(x),$$

e quindi se vale

$$p(x, y) = p(x)p(y).$$

Definizione 11. *Un processo stocastico $\{X_t\}_{t\in T}$ è detto stazionario se la funzione di distribuzione delle variabili aleatorie è invariante rispetto al tempo t, cioè se $p(X_{t_1}, X_{t_2}, \ldots, X_{t_k})$ non dipende da $(t_1, t_2, \ldots, t_k)$ per ogni $k \geq 1$. Formalmente:*

$$p(X_{t_1}, X_{t_2}, \ldots, X_{t_k}) = p(X_{t_1+j}, X_{t_2+j}, \ldots, X_{t_k+j}) \qquad \forall (t_1, t_2, \ldots, t_k) \text{ e } \forall j$$

Ne consegue, per $k = 1$ che $p(X_t) = p(X_{t+1})$, e quindi tutte le variabili aleatorie del processo sono identicamente distribuite, quindi avranno uguale valore di aspettazione.

Definizione 12. *Un processo stocastico stazionario $\{X_t\}_{t\in T}$, con valore d'aspettazione $\mathbb{E}[\cdot]$, è detto ergodico se per ogni $k \in \mathbb{N}$, per ogni funzione misurabile $f : \Omega^{k+1} \to \mathbb{R}$ per quasi ogni realizzazione $(X_t)_{t\in\mathbb{N}}$ vale*

$$\frac{1}{n}\sum_{j=1}^{n} f(X_j, X_{j+1}, \ldots, X_{j+k}) \xrightarrow{n\to\infty} \mathbb{E}[f(X_1, X_2, \ldots, X_{k+1})] \quad \textit{in probabilità.}$$

Nella teoria della probabilità, una sequenza di variabili casuali si dice indipendente e identicamente distribuita (i.i.d.) se le variabili hanno tutte la stessa distribuzione di probabilità e sono tutte stocasticamente indipendenti.

1.2 Disuguaglianza di Jensen

Definizione 13. *Una funzione $f : \mathbb{R}^n \to \mathbb{R}^n$ si dice convessa se:*
$f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda)f(y) \qquad \forall x, y \in \mathbb{R}^n \quad \forall \lambda \in [0, 1]$
Se l'uguaglianza vale solo nel caso in cui $x = y$ oppure se $\lambda = 0$ o $\lambda = 1$, allora si parla di funzione strettamente convessa.

La funzione f si dice concava (strettamente concava) se $-f$ è convessa (strettamente convessa).

Lemma 1. *(Disuguaglianza di Jensen): Sia f una funzione convessa a valori reali, sia X una variabile aleatoria. La disuguaglianza di Jensen afferma che*

$$f(\mathbb{E}[X]) \leq \mathbb{E}[f(X)].$$

Dimostrazione. Dimostriamo il lemma per distribuzioni discrete, cioè per Ω insieme discreto.

Per $\Omega = \{x_1, x_2\}$, la disuguaglianza diventa

$$f(p_1 x_1 + p_2 x_2) \leq p_1 f(x_1) + p_2 f(x_2),$$

che segue direttamente dalla definizione di funzione convessa.

Supponiamo che il teorema sia vero per $\Omega = \{x_1, x_2, \ldots, x_{n-1}\}$, mostriamo che vale per $\Omega' = \{x_1, x_2, \ldots, x_{n-1}, x_n\}$. Indichiamo con $p'_i = p_i/(1-p_n)$ per $i = 1, 2, \ldots, n-1$. Abbiamo

$$\begin{aligned}\sum_{i=1}^{n} p_i f(x_i) &= p_n f(x_n) + (1-p_n) \sum_{i=1}^{n-1} p'_i f(x_i) \geq \\ &\geq p_n f(x_n) + (1-p_n) f\left(\sum_{i=1}^{n-1} p'_i x_i\right) \geq \\ &\geq f\left(p_n x_n + (1-p_n) \sum_{i=1}^{n-1} p'_i x_i\right) = f\left(\sum_{i=1}^{n} p_i x_i\right), \quad (1.2)\end{aligned}$$

dove la disuguaglianza (1.2) segue dalla definizione di convessità.

Tale dimostrazione può essere estesa alle distribuzioni continue per la continuità degli argomenti. □

1.3 Entropia

Definizione 14. *Data una variabile aleatoria X discreta in un alfabeto $\mathcal{A}$ con densità di probabilità $p(x)$, sia*

$$i(x) = -\log p(x) = \log \frac{1}{p(x)}$$

l'informazione emessa alla sorgente. L'entropia $H(X)$ di X è data da:

$$H(X) = \sum_{x \in \mathcal{A}} p(x) \log \frac{1}{p(x)} = -\sum_{x \in \mathcal{A}} p(x) \log p(x) = \mathbb{E}\left[i(X)\right]$$

L'entropia $H(X)$ appena definita è la media statistica dell'informazione associata alla variabile aleatoria X. Essa rappresenta l'informazione che mediamente si acquisisce per effetto della conoscenza di X e dipende esclusivamente dalla distribuzione di probabilità $p(x)$.

Si utilizza il logaritmo in base 2 in quanto l'entropia è espressa in bits. Osserviamo anche che la base adottata per il logaritmo non è concettualmente importante, cambiarla equivale infatti ad introdurre una costante di proporzionalità, come appare ovvio se si ricorda la regola per il cambiamento di base dei logaritmi

$$\log_\beta a = \log_\beta \alpha \log_\alpha a = \frac{\log_\alpha a}{\log_\alpha \beta}.$$

Poiché $\lim_{x\to 0} xlogx = 0$, si pone per continuità $0 \log 0 = 0$.

Lemma 2. $H(X) \geq 0$.

Dimostrazione. Per ogni $x \in \mathcal{A}$ vale $0 \leq p(x) \leq 1$, da ciò si ottiene che vale $-\log p(x) \geq 0 \ \forall x \in \mathcal{A}$. □

Come si può facilmente vedere, eventi poco probabili danno maggiore informazione. L'entropia non dipende dall'alfabeto $\mathcal{A}$ ma solamente dalla sua distribuzione di probabilità. Il massimo dell'entropia si ha quando i simboli dell'alfabeto sono equiprobabili.

Teorema 1. *Sia X una variabile aleatoria con distribuzione di probabilità $p(x)$ a valori in un alfabeto $\mathcal{A}$ di cardinalità A, vale*

$$H(X) \leq \log A,$$

l'uguaglianza si ha se e solo se la distribuzione di probabilità di X è uniforme.

Dimostrazione. Definiamo $h : \Sigma_A \longrightarrow \mathbb{R}$

$$h(p_1, p_2, \ldots, p_A) = -\sum_{i=1}^{A} p_i \log p_i$$

con $\Sigma_A := \{(p_1, p_2, \ldots, p_A) \in (\mathbb{R}_0^+)^n \mid \sum_{i=1}^{A} p_i = 1\}$. Vale $h(p(x)) = H(X)$. Massimizziamo h su Σ_A con il vincolo $p_1, p_2, \ldots, p_A$ distribuzione di probabilità

$$\psi(p_1, p_2, \ldots, p_A) = \sum_{i=1}^{A} p_i - 1 = 0.$$

Utilizzando i moltiplicatori di Lagrange

$$\frac{\partial h}{\partial p_i}(p_1, p_2, \ldots, p_A) = \lambda \frac{\partial \psi}{\partial p_i}(p_1, p_2, \ldots, p_A)$$

si ottiene

$$-\log p_i - p_i \frac{1}{p_i} \log_2 e = \lambda$$
$$\Rightarrow \log p_i = -\lambda - \log_2 e$$
$$\Rightarrow p_i = 2^{-\lambda - \log_2 e}.$$

Si ottiene che $p_i = 2^{-\lambda-\log_2 e}$ per ogni $i \in \{1, 2, \ldots, A\}$, per il vincolo che $(p_1, p_2, \ldots, p_n)$ sia una distribuzione di probabilità si ha che $p(\frac{1}{A}, \frac{1}{A}, \ldots, \frac{1}{A})$ è un punto critico di h. Si può vedere che $(\frac{1}{A}, \frac{1}{A}, \ldots, \frac{1}{A})$ è un punto di massimo ed è unico, inoltre vale

$$H(X) = h(p(x)) \leq h\left(\frac{1}{A}, \frac{1}{A}, \ldots, \frac{1}{A}\right) = \log A.$$

□

Definizione 15. *Data una coppia di variabili aleatorie discrete (X, Y), con $\mathcal{A}$, $\mathcal{B}$ alfabeti su cui sono rispettivamente definite X e Y con distribuzione congiunta $p(x, y)$, si definisce entropia congiunta il valore*

$$H(X, Y) = -\sum_{X \in \mathcal{A}} \sum_{Y \in \mathcal{B}} p(x, y) \log p(x, y);$$

si definisce inoltre entropia condizionata il valore

$$H(X|Y) = \sum_{y \in \mathcal{B}} p(y) H(X|Y = y) = -\sum_{y \in \mathcal{B}} p(y) \sum_{x \in \mathcal{A}} p(x \mid y) \log p(x \mid y) =$$
$$= -\sum_{x \in \mathcal{A}} \sum_{y \in \mathcal{B}} p(x, y) \log p(x \mid y). \quad (1.3)$$

L'entropia condizionata esprime la quantità di informazione necessaria per comunicare la variabile aleatoria X nota Y. La conoscenza di Y riduce la quantità di informazione da fornire, vale quindi

Teorema 2. *$H(X|Y) \leq H(X)$, l'uguaglianza vale se e solo se X ed Y sono stocasticamente indipendenti.*

Prima di procedere con la dimostrazione del Teorema (2) introduciamo il concetto di informazione reciproca.

Definizione 16. *Date due variabili aleatorie X ed Y, con rispettive probabilità di ditribuzione $p(x)$, $p(y)$ e probabilità di distribuzione congiunta $p(x, y)$, si definisce informazione reciproca (mutual information) di X ed Y*

$$I(X; Y) = \sum_{x \in \mathcal{A}} \sum_{y \in \mathcal{B}} p(x, y) \log \frac{p(x, y)}{p(x)p(y)}. \quad (1.4)$$

Nella definizione, si assume la convenzione che $0 \log \frac{0}{0} = 0$.

Lemma 3. $I(X;Y) \geq 0$.

Dimostrazione. Sia $\mathcal{C} = \{(x,y) \mid x \in \mathcal{A}, y \in \mathcal{B}, p(x,y) > 0\}$ il supporto di $p(x,y)$.

$$I(X;Y) = \sum_{(x,y)\in\mathcal{C}} p(x,y) \log \frac{p(x,y)}{p(x)p(y)} \tag{1.5}$$

$$= - \sum_{(x,y)\in\mathcal{C}} p(x,y) \log \frac{p(x)p(y)}{p(x,y)} \tag{1.6}$$

$$\geq - \log \left(\sum_{(x,y)\in\mathcal{C}} p(x,y) \frac{p(x)p(y)}{p(x,y)} \right) \tag{1.7}$$

$$= - \log \sum_{x\in\mathcal{C}} p(x)p(y) \geq 0 \tag{1.8}$$

dove la (1.7) segue dalla disuguaglianza di Jensen in quanto la funzione logaritmo è una funzione concava. Vale l'uguaglianza se e solo se $p(x,y) = p(x)p(y)$, cioè se e solo se X e Y sono stocasticamente indipendenti. □

Dimostrazione. Teorema (2)

$$H(X) - H(X \mid Y) = - \sum_{x\in\mathcal{A}} p(x) \log p(x) + \sum_{x\in\mathcal{A}} \sum_{y\in\mathcal{B}} p(x,y) \log p(x \mid y) \tag{1.9}$$

$$= - \sum_{x\in\mathcal{A}} \sum_{y\in\mathcal{B}} p(x,y) \log p(x) - \left(- \sum_{x\in\mathcal{A}} \sum_{y\in\mathcal{B}} p(x,y) \log p(x \mid y) \right) \tag{1.10}$$

$$= \sum_{x\in\mathcal{A}} \sum_{y\in\mathcal{B}} p(x,y) \log \frac{p(x \mid y)}{p(x)} \tag{1.11}$$

$$= \sum_{x\in\mathcal{A}} \sum_{y\in\mathcal{B}} p(x,y) \log \frac{p(x,y)}{p(x)p(y)} \tag{1.12}$$

$$= I(X;Y) \geq 0, \tag{1.13}$$

dove la (1.12) segue dalla proprietà delle distribuzioni condizionate (1.1), mentre la (1.13) segue dalla definizione di informazione reciproca, che, per il lemma precedente, è sempre non negativa, ed è nulla se e solo se X ed Y sono stocasticamente indipendenti. □

L'informazione reciproca I(X,Y) rappresenta quindi la riduzione dell'incertezza su X dovuta alla conoscenza di Y.

Teorema 3. *(Chain Rule)* $H(X,Y) = H(X) + H(Y \mid X)$

Dimostrazione.

$$\begin{aligned}
H(X,Y) &= -\sum_{x\in\mathcal{A}}\sum_{y\in\mathcal{B}} p(x,y)\log p(x,y) \\
&= -\sum_{x\in\mathcal{A}}\sum_{y\in\mathcal{B}} p(x,y)\log p(x)p(y|x) \\
&= -\sum_{x\in\mathcal{A}}\sum_{y\in\mathcal{B}} p(x,y)\log p(x) - \sum_{x\in\mathcal{A}}\sum_{y\in\mathcal{B}} p(x,y)\log p(y \mid x) \\
&= -\sum_{x\in\mathcal{A}} p(x)\log p(x) - \sum_{x\in\mathcal{A}}\sum_{y\in\mathcal{B}} p(x,y)\log p(y \mid x) \\
&= H(X) + H(Y \mid X).
\end{aligned}$$

□

Corollario 1. $H(X,Y \mid Z) = H(X \mid Z) + H(Y \mid X,Z)$.

Dimostrazione. La dimostrazione è analoga a quella del precedente teorema. □

Teorema 4. *Siano $X_1, X_2, \ldots, X_n$ variabili aleatorie con distribuzione congiunta $p(x_1, x_2, \ldots, x_n)$, si ha:*

$$H(X_1, X_2, \ldots, X_n) = \sum_{i=1}^{n} H(X_i \mid X_{i-1}, X_{i-2}, \ldots, X_1) \tag{1.14}$$

Dimostrazione. Si dimostra per induzione, utilizzando il corollario precedente:

$$\begin{aligned}
H(X_1, X_2) &= H(X_1) + H(X_2 \mid X_1) \\
H(X_1, X_2, X_3) &= H(X_1) + H(X_2, X_3 \mid X_1) \\
&= H(X_1) + H(X_2 \mid X_1) + H(X_3 \mid X_2, X_1) \\
&\ldots \\
H(X_1, X_2, \ldots, X_n) &= H(X_1) + H(X_2 \mid X_1) + \ldots + H(X_n \mid X_{n-1}, X_{n-2}, \ldots, X_1) \\
&= \sum_{i=1}^{n} H(X_i \mid X_{i-1}, X_{i-2}, \ldots, X_1)
\end{aligned}$$

□

Proposizione 1. *Siano $X_1, X_2, \ldots, X_n$ variabili aleatorie con distribuzione congiunta $p(x_1, x_2, \ldots, x_n)$, si ha*

$$H(X_1, X_2, \ldots, X_n) \leq \sum_{i=1}^{n} H(X_i),$$

e l'uguaglianza vale se e solo se le variabili aleatorie sono stocasticamente indipendenti.

Dimostrazione. Per la Chain Rule al Teorema (1.14):

$$H(X_1, X_2, \ldots, X_n) = \sum_{i=1}^{n} H(X_i \mid X_{i-1}, X_{i-2}, \ldots, X_1) \leq \sum_{i=1}^{n} H(X_i)$$

dove la diseguaglianza segue direttamente dal Teorema (2). Si ha l'uguaglianza se e solo se ogni X_i è indipendente dalle precedenti $X_{i-1}, X_{i-2}, \ldots, X_1$, cioè se e solo se le X_i sono stocasticamente indipendenti. □

1.4 Proprietà di Equipartizione Asintotica (AEP)

La proprietà di equipartizione asintotica (Asymptotic Equipartition Property, AEP) in teoria dell'informazione rappresenta un analogo della legge debole dei grandi numeri nella teoria della probabilità.

Definizione 17. *Definiamo $\mathcal{A}^n$ l'insieme delle sequenze x di lunghezza n nell'alfabeto finito $\mathcal{A}$, $x = (x_1, x_2, \ldots, x_n) \in \mathcal{A}^n$, con $x_i \in \mathcal{A}$ per ogni $1 \leq i \leq n$.*

Definizione 18. *Dato $\epsilon > 0$, si definisce insieme tipico $\mathcal{T}_n(\epsilon)$ rispetto alla densità $p(x)$ l'insieme di tutti gli esiti $(x_1, x_2, \ldots, x_n) \in \mathcal{A}^n$ tali che:*

$$2^{-n(H(X)+\epsilon)} \leq p(x_1, x_2, \ldots, x_n) \leq 2^{-n(H(X)-\epsilon)}$$

Definizione 19. *Data una successione di variabili aleatorie $X_1, X_2, \ldots$, diciamo che la successione $X_1, X_2, \ldots$ converge ad una variabile aleatoria X:*

1. *In probabilità se, per ogni $\epsilon > 0$, $\mathbb{P}(|X_n - X| > \epsilon) \xrightarrow{n\to\infty} 0$*
2. *In media se $\mathbb{E}[X_n - X] \xrightarrow{n\to\infty} 0$*
3. *Con probabilità 1 (o quasi certamente) se $\mathbb{P}(\lim_{n\to\infty} X_n = X) = 1$.*

Il teorema AEP è una conseguenza diretta della legge debole dei grandi numeri, che stabilisce che per una successione di variabili aleatorie $X_1, X_2, \ldots$ i.i.d., la media campionaria $\frac{1}{n}\sum_{i=1}^{n} X_i$ converge in probabilità al valore di aspettazione $\mathbb{E}[X]$.

Teorema 5. *(Proprietà di equipartizione asintotica). Si consideri una sequenza di variabili aleatorie $X_1, X_2, \ldots$, i.i.d. con densità di probabilità $p(x)$. Allora:*

$$-\frac{1}{n}\log p(x_1, x_2, \ldots, x_n) \xrightarrow{n\to\infty} H(X) \tag{1.15}$$

in probabilità.

Dimostrazione. Per provarlo basta notare che funzioni di variabili aleatorie indipendenti sono esse stesse variabili aleatorie. Poiché le X_i sono i.i.d, anche le $p(x_i)$ lo saranno. Per la legge debole dei grandi numeri si ha dunque:

$$\begin{aligned} -\frac{1}{n}\log p(x_1, x_2, \ldots, x_n) &= -\frac{1}{n}\sum_i \log p(x_i) \\ &\xrightarrow{n\to\infty} -\mathbb{E}[\log p(x)] = H(X) \end{aligned} \tag{1.16}$$

in probabilità. □

Teorema 6. *Siano $X_1, X_2, \ldots$ variabili aleatorie i.i.d. con densità $p(x)$ e, per $\epsilon > 0$, sia $\mathcal{T}_n(\epsilon)$ l'insieme tipico di $\mathcal{A}^n$, valgono:*

i. per ogni $(x_1, x_2, \ldots, x_n) \in \mathcal{T}_n(\epsilon)$:

$$H(X) - \epsilon \leq -\frac{1}{n}\log p(x_1, x_2, \ldots, x_n) \leq H(X) + \epsilon;$$

ii. per n abbastanza grande $p(\mathcal{T}_n(\epsilon)) \geq 1 - \epsilon$;

iii. $|\mathcal{T}_n(\epsilon)| \leq 2^{n(H(X)+\epsilon)}$;

iv. per n abbastanza grande $|\mathcal{T}_n(\epsilon)| \geq 2^{n(H(X)-\epsilon)}$.

Dimostrazione. i. Segue direttamente dalla definizione di insieme tipico. Passando ai logaritmi:

$$-n(H(X) + \epsilon) \leq \log p(x_1, x_2, \ldots, x_n) \leq -n(H(X) - \epsilon).$$

Moltiplicando per $-\frac{1}{n}$ si ottiene la proprietà *(i)*.

ii. Per dimostrare la seconda proprietà utilizziamo (1.15). Poiché la probabilità di un evento $(x_1, x_2, \ldots, x_n) \in \mathcal{T}_\epsilon^{(n)}$ tende a 1 per $n \to \infty$, per ogni $\delta > 0$ esiste un n_0 tale che per ogni $n \geq n_0$ si ha

$$\lim_{n\to\infty} \mathbb{P}\left(\left|-\frac{1}{n}\log p(x_1, x_2, \ldots, x_n) - H(X)\right| \leq \epsilon\right) > 1 - \delta.$$

Ponendo $\delta = \epsilon$ otteniamo la seconda proprietà.

iii. Per dimostrare la terza proprietà scriviamo

$$1 = \sum_{x\in\mathcal{A}^n} p(x) \geq \sum_{x\in\mathcal{T}_\epsilon^{(n)}} p(x) \geq$$
$$\geq \sum_{x\in\mathcal{T}_\epsilon^{(n)}} 2^{-n(H(X)+\epsilon)} = 2^{-n(H(X)+\epsilon)}|\mathcal{T}_\epsilon^{(n)}|, \quad (1.17)$$

dove la seconda disuguaglianza segue dalla definizione di insieme tipico.

iv. Per n abbastanza grande, $\mathbb{P}(\mathcal{T}_\epsilon^{(n)}) > 1 - \epsilon$, allora

$$1 - \epsilon < \mathbb{P}(\mathcal{T}_\epsilon^{(n)}) \geq \sum_{x\in\mathcal{T}_\epsilon^{(n)}} 2^{-n(H(X)-\epsilon)} =$$
$$= 2^{-n(H(X)-\epsilon)}|\mathcal{T}_\epsilon^{(n)}|, \quad (1.18)$$

dove la seconda disuguaglianza segue dalla definizione di insieme tipico. Si ha quindi

$$|\mathcal{T}_\epsilon^{(n)}| \geq (1 - \epsilon)2^{n(H(X)-\epsilon)},$$

che completa la dimostrazione.

□

Dalla seconda proprietà del teorema precedente si deduce che gran parte della probabilità delle sequenze di $\mathcal{A}^n$ si concentra nell'insieme tipico $\mathcal{T}_\epsilon^{(n)}$ per valori abbastanza grandi di n. Ciò significa che una sequenza di $\mathcal{A}^n$ avrà un'elevata probabilità di appartenere all'insieme tipico nonostante la cardinalità di quest'ultimo sia praticamente trascurabile rispetto al numero totale di sequenze in $\mathcal{A}^n$. Confrontando il numero di elementi di entrambi gli insiemi si osserva che

$$\frac{|\mathcal{T}_\epsilon^{(n)}|}{|\mathcal{A}^n|} \simeq \frac{2^{n(H(X))}}{2^{n\log(|\mathcal{A}|)}} = 2^{n(H(X)-\log(|\mathcal{A}|))}$$

che tende a 0 per $n \to \infty$.

Sia $\mathcal{A}^n$ l'insieme delle sequenze di lunghezza n di variabili aleatorie i.i.d. con distribuzione $p(x)$, sia $\epsilon > 0$. Mostriamo che esiste un codice che mappa le sequenze $(x_1, x_2, \ldots, x_n)$ in stringhe di bit, tale che la mappa sia invertibile e che valga

$$\mathbb{E}\left[\frac{1}{n} l(x_1, x_2, \ldots, x_n)\right] \geq H(X) + \epsilon$$

per n abbastanza grande Possiamo quindi rappresentare le sequenze di $\mathcal{A}^n$ utilizzando $nH(X)$ bits in media.

Dividiamo $\mathcal{A}^n$ in $\mathcal{T}_\epsilon^{(n)}$ e sia $\mathcal{T}_\epsilon^{(n)^C} = \mathcal{A}^n \setminus \mathcal{T}_\epsilon^{(n)}$, insieme tipico ed il suo complementare. Ordiniamo le sequenze appartenenti agli insiemi secondo un certo ordine, ad esempio ordine lessicografico. A questo punto sarà possibile descrivere una particolare sequenza indicando l'insieme di appartenenza e dando il suo indice per l'ordinamento scelto. Con questo metodo sono sufficienti $\lceil n(H(X) + \epsilon) \rceil$ bits per descrivere una sequenza in $\mathcal{T}_\epsilon^{(n)}$. A questi occorre aggiungere un bit iniziale per stabilire se si tratta di una sequenza in $\mathcal{T}_\epsilon^{(n)}$ o in $\mathcal{T}_\epsilon^{(n)^C}$. Per descrivere invece gli elementi di $\mathcal{T}_\epsilon^{(n)^C}$ sono sufficienti al più $n \log |\mathcal{A}| + 1$ bit, essendo questo il numero di bit sufficiente a descrivere l'intero insieme $\mathcal{A}$. Anche qui è necessario aggiungere un bit per indicare l'insieme di appartenenza. Sia quindi $l(\mathbf{x})$ la lunghezza della parola di codice usata per codificare la sequenza $\mathbf{x} = (x_1, x_2, \ldots, x_n)$. Abbiamo che il valore atteso di tale lunghezza sarà

$$\begin{aligned}
\mathbb{E}[l(\mathbf{x})] = \sum_{\mathbf{x} \in \mathcal{A}^n} p(\mathbf{x}) l(\mathbf{x}) &= \sum_{\mathbf{x} \in \mathcal{T}_\epsilon^{(n)}} p(\mathbf{x}) l(\mathbf{x}) + \sum_{\mathbf{x} \in \mathcal{T}_\epsilon^{(n)^C}} p(\mathbf{x}) l(\mathbf{x}) \\
&\leq \sum_{\mathbf{x} \in \mathcal{T}_\epsilon^{(n)}} p(\mathbf{x})(n(H + \epsilon) + 2) + \sum_{\mathbf{x} \in \mathcal{T}_\epsilon^{(n)^C}} p(\mathbf{x})(n \log |\mathcal{A}| + 2) \\
&= \mathbb{P}(\mathcal{T}_\epsilon^{(n)})(n(H + \epsilon) + 2) + \mathbb{P}(\mathcal{T}_\epsilon^{(n)^C})(n \log |\mathcal{A}| + 2) \\
&\leq n(H + \epsilon) + \epsilon n(\log |\mathcal{A}|) + 2 = n(H + \epsilon'),
\end{aligned}$$

dove $\epsilon' = \epsilon + \epsilon \log \mathcal{A} + \frac{2}{n}$ tende a 0 per $\epsilon \to 0$ e per $n \to \infty$.

1.5 Classificazione codici

Una sorgente d'informazione discreta, è un sistema che emette in modo più o meno casuale sequenze di elementi appartenenti ad un insieme assegnato, l'alfabeto della sorgente, dotato di una misura di probabilità. Si può immaginare che la sorgente emetta una serie di variabili aleatorie di tipo discreto.

Definizione 20. *Una sorgente d'informazione discreta $\mathcal{S}$ è un processo stocastico a parametro discreto $\{X_k\}_{k\in\mathbb{N}}$ a valori nell'alfabeto della sorgente $\mathcal{A}$.*

Se la probabilità che in un dato istante la sorgente emetta una qualunque lettera non dipende dai simboli emessi negli istanti precedenti si dice che la sorgente è priva di memoria. Una sorgente di questo tipo potrebbe ad esempio essere quella che emette la sequenza dei risultati ottenuti lanciando ripetutamente una moneta, l'alfabeto della sorgente sarebbe in questo caso costituito da due simboli. Viceversa, una sorgente discreta dotata di memoria potrebbe trasmettere un testo in una qualsiasi lingua, in questo caso ci sarebbero forti dipendenze fra i simboli emessi. Ad esempio, prendendo un testo in lingua italiana, se l'ultimo simbolo emesso dalla sorgente è q, la probabilità che il successivo sia u è molto alta.

Questa definizione di sorgente come processo stocastico permette l'estensione del concetto di entropia alla sorgente stessa in modo intuitivamente analogo all'entropia della singola variabile aleatoria.

Definizione 21. *Data una sorgente $\mathcal{S}$ si definisce entropia o, più precisamente, tasso di entropia della sorgente la quantità*

$$H(\mathcal{S}) = \lim_{n\to\infty} \frac{1}{n} H(X_1, X_2, \ldots, X_n), \tag{1.19}$$

se tale limite esiste.

Si nota che l'entropia di una sorgente $\mathcal{S}$ dipende esclusivamente dalla distribuzione di probabilità dei simboli dell'alfabeto e non dalla loro natura.

Definiamo una diversa nozione di entropia di un processo stocastico

$$\tilde{H}(\mathcal{S}) = \lim_{n\to\infty} H(X_n \mid X_{n-1}, X_{n-2}, \ldots, X_1) \tag{1.20}$$

se tale limite esiste.

Mentre la (1.19) indica l'entropia per simbolo su n variabili aleatorie, la (1.20) è l'entropia condizionata dell'ultima variabile aleatorie note tutte le precedenti. Proviamo ora un risultato molto importante per processi stocastici stazionari.

Teorema 7. *Per un processo stocastico stazionario il limite (1.19) ed il limite (1.20) esistono e sono uguali:*

$$H(\mathcal{S}) = \tilde{H}(\mathcal{S}).$$

Mostriamo per prima cosa che esiste $\lim_{n\to\infty} H(X_n \mid X_{n-1}, X_{n-2}, \ldots, X_1)$.

Teorema 8. *Per un processo stocastico stazionario,* $H(X_n \mid X_{n-1}, X_{n-2}, \ldots, X_1)$ *è non crescente in funzione di* n *e tende a* $\tilde{H}(\mathcal{S})$.

Dimostrazione.

$$\begin{aligned} H(X_{n+1} \mid X_n, X_{n-1}, \ldots, X_1) \leq & H(X_{n+1} \mid X_n, X_{n-1}, \ldots, X_2) \\ = & H(X_n \mid X_{n-1}, X_{n-2}, \ldots, X_1), \end{aligned}$$

dove la disuguaglianza segue dal fatto che il condizionamento riduce l'entropia e l'uguaglianza dalla stazionarietà del processo stocastico. Poiché $H(X_n \mid X_{n-1}, X_{n-2}, \ldots, X_1)$ è una successione decrescente di numeri positivi esiste un limite che coincide con $\tilde{H}(\mathcal{S})$. □

Per poter dimostrare il Teorema (7) utilizzeremo il risultato seguente dall'analisi.

Teorema 9. *(delle Medie di Cesàro) Siano* $(a_n)_{n\in\mathbb{N}}$ *e* $(b_n)_{n\in\mathbb{N}}$ *due successioni di numeri reali tali che* $a_n \xrightarrow{n\to\infty} a$ *e* $b_n = \frac{1}{n}\sum_{i=1}^{n} a_i$, *allora* $b_n \xrightarrow{n\to\infty} a$.

Dimostrazione. Sia $\epsilon > 0$. Poiché $a_n \xrightarrow{n\to\infty} a$ esiste un numero $N(\epsilon)$ tale che $|a_n - a| \leq \epsilon$ per ogni $n \geq N(\epsilon)$. Quindi,

$$\begin{aligned} |b_n - a| = & \left| \frac{1}{n} \sum_{i=1}^{n} (a_i - a) \right| \\ \leq & \frac{1}{n} \sum_{i=1}^{n} |(a_i - a)| \\ \leq & \frac{1}{n} \sum_{i=1}^{N(\epsilon)} |(a_i - a)| + \frac{n - N(\epsilon)}{n} \epsilon \\ \leq & \frac{1}{n} \sum_{i=1}^{N(\epsilon)} |(a_i - a)| + \epsilon \end{aligned}$$

per ogni $n \geq N(\epsilon)$. Poiché il primo termine tende a 0 per $n \to \infty$, possiamo porre $|b_n - a| \leq 2\epsilon$ prendendo n sufficientemente grande. Quindi, $b_n \to a$ per $n \to \infty$. □

Dimostrazione. (Teorema (7)) Dalla Chain Rule,

$$\frac{H(X_1, X_2, \ldots, X_n)}{n} = \frac{1}{n} \sum_{i=1}^{n} H(X_i \mid X_{i-1}, X_{i-2}, \ldots, X_1),$$

che corrisponde alla media dell'entropia condizionata. Poiché abbiamo dimostrato che la successione delle entropie condizionate di un processo stocastico stazionario ha limite $\tilde{H}(\mathcal{S})$, per il teorema di Cesàro anche la successione delle medie delle entropie condizionate ha un limite, e tale limite è lo stesso $\tilde{H}(\mathcal{S})$. Vale quindi

$$H(\mathcal{S}) = \lim_{n\to\infty} \frac{H(X_1, X_2, \ldots, X_n)}{n} = \lim_{n\to\infty} H(X_n \mid X_{n-1}, X_{n-2}, \ldots, X_1) = \tilde{H}(\mathcal{S})$$

□

Definizione 22. *Un codice C per una variabile aleatoria X definita su uno spazio di probabilità $(\Omega, \mathcal{F}, p)$ a valori in un alfabeto $\mathcal{A}$ è una mappa*

$$\begin{aligned} C: \ & \mathcal{A} \longrightarrow \mathcal{D}^* \\ & x \mapsto C(x) \end{aligned}$$

dove $\mathcal{D}^* = \bigcup_n \mathcal{D}^n$ è l'insieme costituito dalle parole di tutte le lunghezze di elementi di $\mathcal{D}$, x è un generico simbolo dell'alfabeto $\mathcal{A}$ e $C(x)$ la sua codifica detta parola di codice.

	C_1	C_2	C_3
A	0	1101	0
B	0	1100000	10
C	1	00100	110
D	1	11111	111

Tabella 1.1: Esempi di codici: utilizziamo un alfabeto binario $\mathcal{D} = \{0, 1\}$, mentre $X = \{A, B, C, D\}$. Il codice C_1 è molto efficiente poiché utilizza un solo simbolo per ogni lettera, tuttavia è molto ambiguo, infatti alla ricezione di uno 0 non si è in grado di determinare se sia stata inviata una A o una B. Il codice C_2 è poco efficiente poiché le sequenze sono molto lunghe, ma non è ambiguo. Il codice C_3 rispetta invece entrambi i criteri, non utilizza molti simboli e non presenta ambiguità.

Ci chiediamo quando un codice è migliore rispetto ad un altro. Possiamo esprimere la bontà di un codice in base a due parametri: la sua lunghezza media (che deve essere minore possibile) e l'efficacia con cui il destinatario può ricostruire il messaggio ricevuto. Per chiarire meglio queste idee, nella Tabella 1.1 sono riportati alcuni esempi di codici.

Indichiamo con $l(x)$ la lunghezza in bits della codifica di un simbolo $x \in \mathcal{A}$ e con L la lunghezza media del codice:

$$l(x) := |C(x)|, \qquad L(C) := \mathbb{E}[l(X)] = \sum_{x\in\mathcal{A}} l(x)p(x).$$

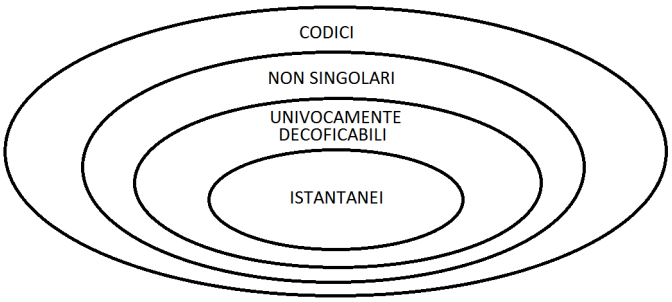

Figura 1.1: Classificazione codici

Definizione 23. *Un codice C è detto non singolare se è iniettivo, cioè se per ogni coppia di simboli distinti $x_1 \neq x_2$ dell'alfabeto $\mathcal{A}$ vale*

$$C(x_1) \neq C(x_2).$$

Nella pratica ci interessa codificare una sequenza di simboli $X_1, X_2, \ldots, X_n$ di lunghezza arbitraria n, sarà quindi necessario che sia ben definita la fine di ogni parola di codice e l'inizio della successiva. Una soluzione potrebbe essere quella di dividere le singole parole con uno spazio o una virgola, ma tale metodo risulta essere inefficiente.

Definizione 24. *Dato un codice C, il codice esteso C^* è una mappa*

$$\begin{aligned} C^* \quad & \mathcal{A}^* \longrightarrow \mathcal{D}^* \\ & (x_1, x_2, \ldots, x_n) \mapsto C(x_1, x_2, \ldots, x_n) := C(x_1)C(x_2)\cdots C(x_n), \end{aligned}$$

dove $C(x_1)C(x_2)\cdots C(x_n)$ è la concatenazione delle parole di codice $C(x_1)$, $C(x_2)$, ..., $C(x_n)$.

Definizione 25. *Se l'estensione C^* è non singolare il codice C si dice univocamente decodificabile.*

Definizione 26. *Un codice C si dice codice istantaneo o prefix-code (senza prefisso) se nessuna parola di codice è prefisso di un'altra parola.*

Un codice istantaneo può essere decoficato senza avere alcuna informazione sulle parole di codice successive in quanto la fine di ogni parola è immediatamente riconoscibile. Ricapitolando, si ha una classificazione dei codici che può essere descritta schematicamente come in Figura 1.1.

1.6 Disuguaglianza di Kraft

Sia $\mathcal{A}$ un alfabeto finito di cardinalità m. Un codice univocamente decodificabile può essere costruito assegnando ad ognuno dei simboli di $\mathcal{A}$ parole di codice tutte della medesima lunghezza. In questo caso la lunghezza minima di tali parole non potrà essere inferiore a $\lceil \log m \rceil$, dove $\lceil \cdot \rceil$ indica il primo intero non minore. Il nostro scopo è quello di costruire codici istantanei minimizzando la lunghezza media L. Chiaramente non possiamo assegnare parole di codice corte ad ognuno dei simboli facendo sì che il codice rimanga senza prefisso. La disuguaglianza di Kraft, in quanto condizione necessaria all'univoca decodificabilità, costituisce un vincolo sull'insieme delle lunghezze possibili per le parole di un codice.

Teorema 10. *(Disuguaglianza di Kraft) Per ogni codice istantaneo C, mappa a valori in un alfabeto $\mathcal{D}$ di cardinalità D, le lunghezze delle parole di codice $l_1, l_2, \ldots, l_m$ soddisfano la disugualianza*

$$\sum_{i=1}^{m} D^{-l_i} \leq 1.$$

Viceversa, date $l_1, l_2, \ldots, l_m$ che soddisfano tale disuguaglianza, esiste un codice istantaneo che ha i valori l_i come lunghezze delle parole di codice.

Dimostrazione. Consideriamo l'albero D-ario nel quale ciascun nodo ha D rami. Supponiamo che gli D possibili elementi di $\mathcal{D}$ siano rappresentati da tali rami. Poiché il codice C è istantaneo nessuna parola di codice è prefisso di un'altra. Nella rappresentazione nell'albero D-ario ogni percorso che dalla radice porta ad una foglia costituisce una delle m parole di codice, ogni assegnazione di parola di codice ad un percorso elimina tutti i rami discendenti, si riporta un esempio con $D = 2$ nella Figura 1.2. Sia l_{max} il massimo valore delle lunghezze delle parole di codice. Consideriamo tutti i nodi dell'albero al livello l_{max}: alcuni di questi sono parole di codice, altri sono rami discendenti da parole di codice, altri ancora nessuna delle due. Il percorso dell'albero che rappresenta una parola di codice di lunghezza l_i ha $D^{l_{max}-l_i}$ rami discendenti al livello l_{max}. L'insieme dei discendenti di ciascuna parola di codice è disgiunto dagli insiemi dei discendenti di tutte le altre, inoltre il numero totale dei nodi di questi insiemi è minore o uguale a $D^{l_{max}}$. Sommando su tutte le parole si ottiene

$$\sum_{i=1}^{m} D^{l_{max}-l_i} \leq D^{l_{max}},$$

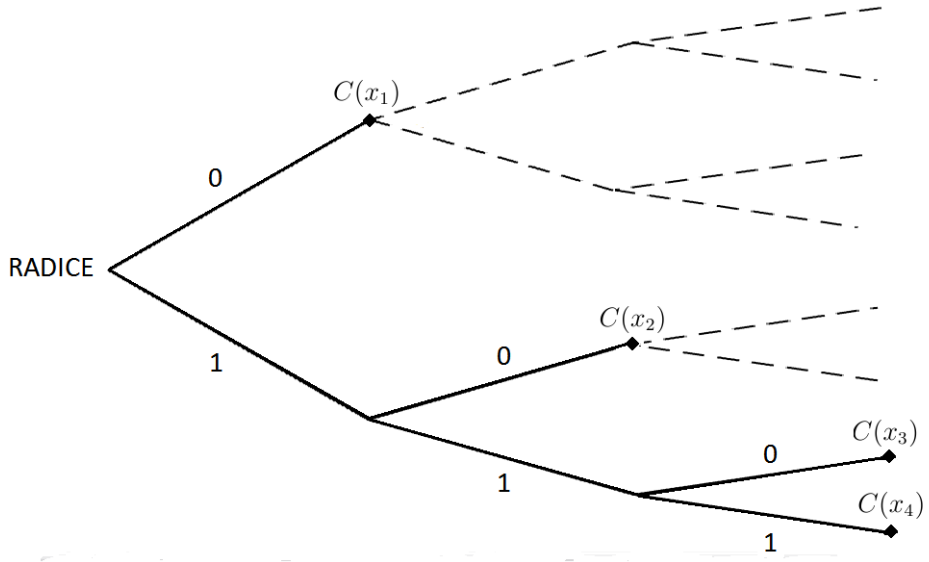

Figura 1.2: Esempio di codice istantaneo rappresentato in un albero binario ($D = 2$): l'assegnazione dei simboli x_1 e x_2 rispettivamente alle parole di codice $C(x_1) = 0$ e $C(x_2) = 10$ comporta l'eliminazione dei rami discendenti da entrambe le parole rappresentati tratteggiati in figura. Le parole di codice $C(x_3)$ e $C(x_4)$ hanno lunghezza massima $l_{max} = 3$.

e, dividendo entrambi i membri per $D^{l_{max}}$, si ottiene la Disuguaglianza di Kraft.

Viceversa, date $l_1, l_2, \ldots, l_m$ che soddisfano la Disuguaglianza di Kraft, è possibile costruire un albero simile a quello appena descritto. Si assegna la prima parola di codice al primo nodo (in ordine lessicografico) di lunghezza l_1, rimuovendo tutti i suoi rami discendenti dall'albero. Si assegna la seconda parola di codice al primo nodo libero tra i rimanenti di lunghezza l_2 e si eliminano tutti i relativi discendenti. Se $k < m$ non è possibile che non sia disponibile un ramo al quale assegnare la k-esima parola di codice. Ciò significherebbe che le eliminazioni eseguite ai passi precedenti hanno esaurito tutti i rami dell'albero, si avrebbe quindi la situazione in cui $\sum_{i=1}^{k} D^{-l_i} = 1$ che contraddice l'ipotesi iniziale che i parametri $l_1, l_2, \ldots, l_m$ soddisfino la Disuguaglianza di Kraft. È quindi possibile procedere con tali assegnazioni fino all'm-esimo elemento x_m e si ottiene un codice senza prefisso con le specificate lunghezze $l_1, l_2, \ldots, l_m$ per le parole di codice. □

Ogni codice che ha la proprietà di essere istantaneo deve soddisfare anche la disuguaglianza di Kraft. Notiamo che la seconda parte del teorema non afferma che se un codice verifica la Disuguaglianza di Kraft tale codice è is-

tantaneo, ma afferma che se tale disuguaglianza è verificata per dei parametri $l_1, l_2, \ldots, l_m$ e D allora è possibile costruire un codice istantaneo D-ario con parole di codice di lunghezza $l_1, l_2, \ldots, l_m$ mediante l'algoritmo presentato nella dimostrazione del teorema.

1.7 Codici ottimali

Siamo ora interessati al problema della ricerca di un codice istantaneo con lunghezza media L minima. Dai risultati visti notiamo che risolvere questo problema è equivalente a trovare l'insieme delle lunghezze $l_1, l_2, \ldots, l_m$ che soddifano la Disuguaglianza di Kraft e il cui valore atteso della lunghezza $L = \sum p_i l_i$ sia il minore tra quelli di tutti i codici istantanei. Minimizziamo

$$L = \sum_{i=1}^{m} p_i l_i$$

su tutti gli interi $l_1, l_2, \ldots, l_m$ con il vincolo della Disuguaglianza di Kraft

$$\sum_{i=1}^{m} D^{-l_i} \leq 1.$$

Indichiamo con $l^* = (l_1^*, l_2^*, \ldots, l_m^*) \in \mathbb{R}_+^m$ il punto di minimo che si ottiene eliminando il vincolo di l_i a valori interi e considerando il vincolo dell'uguaglianza $\sum D^{-l_i} = 1$. Tale problema si risolve con l'utilizzo dei moltiplicatori di Lagrange su $\mathbb{R}^m$ per estremizzare la funzione

$$f(l) = \sum_{i=1}^{m} p_i l_i + \lambda \left(\sum_{i=1}^{m} D^{-l_i} \right).$$

Differenziando rispetto alle l_i, otteniamo

$$\frac{\partial f}{\partial l_i}(l) = p_i - \lambda D^{-l_i} \log_e D,$$

inoltre, facendo tendere a zero il valore della derivata,

$$D^{-l_i} = \frac{p_i}{\lambda \log_e D}.$$

Per trovare λ, sostituiamo questo risultato nel vincolo. Otteniamo $\lambda = 1/\log_e D$, e quindi $p_i = D^{-l_i}$ da cui $l_i^* = -\log_D p_i$. Poiché inizialmente era

stato rimosso il vincolo di l_i a valori interi, se gli l_i^* ottenuti sono frazionari si considera il più piccolo intero maggiore di l_i^*

$$l_i^{**} = \left\lceil \log_D \frac{1}{p_i} \right\rceil \tag{1.21}$$

che prende il nome di **codice di Shannon**. Nel caso superottimale in cui i valori l_i^* sono tutti interi, la lunghezza attesa minima del codice C è data da

$$L^* = L(C) = \sum_{i=1}^{m} p_i l_i^* = -\sum_{i=1}^{m} p_i \log p_i = H_D(X),$$

quindi la lunghezza media di questo codice corrisponde all'entropia D-aria della variabile $H_D(X)$. Abbiamo quindi dimostrato il teorema seguente.

Teorema 11. *Per ogni codice istantaneo D-ario per una variabile aleatoria X vale*

$$L(C) \geq H_D(X)$$

dove $L(C)$ indica il valore atteso della lunghezza del codice C e $H_D(X)$ indica l'entropia D-aria della variabile aleatoria X. L'uguaglianza vale se e solo se $p_i = D^{-l_i}$, $\forall i \in \{1, 2, \ldots, m\}$, dove l_i è la lunghezza della parola di codice associata all'i-esimo elemento $x_i \in \mathcal{A}$, con $\mathcal{A}$ alfabeto di cardinalità m su cui è definita X.

Osservazione 1. *È possibile raggiungere la superottimalità del codice, e quindi l'uguaglianza nel teorema precedente, se e solo se le probabilità p_i di X sono D-adiche, cioè se esistono $k_1, k_2, \ldots, k_n \in \mathbb{N}$ tali che $p_i = D^{-k_i}$ per ogni $i = 1, 2, \ldots, m$.*

Proposizione 2. *Data una variabile aleatoria X, esiste un codice D-ario istantaneo C per X tale che*

$$H_D(X) \leq L(C) < H_D(X) + 1.$$

Dimostrazione. Sia $l_i = l_i^{**} = \lceil \log_D \frac{1}{p_i} \rceil$ il codice di Shannon (1.21). Un codice di questo tipo soddisfa la Disuguaglianza di Kraft, vale infatti

$$\sum_{i=1}^{m} D^{-l_i^{**}} \leq \sum_{i=1}^{m} D^{-\log_D \frac{1}{p_i}} = \sum_{i=1}^{m} p_i = 1$$

in cui la disuguaglianza si ottiene dal fatto che $l_i^{**} \geq \log_D \frac{1}{p_i}$. Infine notiamo che poiché $-\log p_i \leq l_i^{**} < -\log p_i + 1$, vale

$$\mathbb{E}[-\log p_i] \leq \mathbb{E}[l_i^{**}] < \mathbb{E}[-\log p_i + 1]$$

che dimostra la proposizione. □

Questo teorema afferma che la lunghezza attesa di un codice può aumentare fino ad un bit quando $\log \frac{1}{p_i}$ non è intero per ogni $i \in \{1, 2, \ldots, m\}$. Possiamo ridurre il costo di tale bit per simbolo inviando sequenze finite di più simboli.

Consideriamo il caso in cui si debba inviare una sequenza di n simboli di X. Assumiamo che tali simboli siano i.i.d. e con distribuzione di probabilità $p(x)$. Possiamo considerare una sequenza di questo tipo come una parola di $\mathcal{A}^n$, insieme delle sequenze di lunghezza n di simboli dell'alfabeto $\mathcal{A}$.

Sia ora L_n la lunghezza media della parola di codice per simbolo in input, cioè, se $l(x_1, x_2, \ldots, x_n)$ è la lunghezza del codice binario associato alla sequenza $x_1, x_2, \ldots, x_n$, allora

$$L_n = \frac{1}{n} \sum p(x_1)p(x_2) \cdots p(x_n) l(x_1, x_2, \ldots, x_n) = \frac{1}{n} \mathbb{E}[l(X_1, X_2, \ldots, X_n)].$$

Applicando la proposizione precedente si ottiene

$$H(X_1, X_2, \ldots, X_n) \leq \mathbb{E}[l(X_1, X_2, \ldots, X_n)] < H(X_1, X_2, \ldots, X_n) + 1. \quad (1.22)$$

Poiché le variabili aleatorie $X_1, X_2, \ldots, X_n$ sono i.i.d., $H(X_1, X_2, \ldots, X_n) = \sum H(X_i) = nH(\mathcal{S})$. Dividendo (1.22) per n, otteniamo

$$H(\mathcal{S}) \leq L_n < H(\mathcal{S}) + \frac{1}{n}.$$

Abbiamo quindi mostrato che attraverso l'utilizzo di blocchi di variabili aleatorie di grandi lunghezze possiamo ottenere la lunghezza media del codice per simbolo arbitrariamente vicina al valore dell'entropia.

Lo stesso ragionamento vale per sequenze di variabili aleatorie di un processo stocastico che non sia necessariamente i.i.d., infatti in questo caso vale ancora (1.22), e dividendo per n, otteniamo

$$\frac{H(X_1, X_2, \ldots, X_n)}{n} \leq L_n < \frac{H(X_1, X_2, \ldots, X_n)}{n} + \frac{1}{n}.$$

Notiamo che se il processo stocastico è stazionario

$$H(X_1, X_2, \ldots, X_n)/n \xrightarrow{n \to \infty} H(\mathcal{S}),$$

con $H(X_1, X_2, \ldots, X_n)$ entropia congiunta delle variabili aleatorie $X_1, X_2, \ldots, X_n$ e con $H(\mathcal{S})$ entropia del processo stocastico $X_1, X_2, \ldots, X_n$.

Abbiamo quindi dimostrato che vale il seguente teorema.

Teorema 12. *Il minimo L_n^* del valore atteso della lunghezza del codice per simbolo soddisfa*

$$\frac{H(X_1, X_2, \ldots, X_n)}{n} \leq L_n^* < \frac{H(X_1, X_2, \ldots, X_n)}{n} + \frac{1}{n}.$$

Inoltre, se $X_1, X_2, \ldots, X_n$ è un processo stocastico stazionario

$$L_n^* \xrightarrow{n\to\infty} H(\mathcal{S}),$$

dove $H(\mathcal{S})$ indica l'entropia del processo stocastico.

Teorema 13. *(McMillan) Le lunghezze delle parole di codice $l_1, l_2, \ldots, l_m$ di un codice univocamente decodificabile C soddisfano la Disuguaglianza di Kraft*

$$\sum_{i=1}^{m} D^{-l_i} \leq 1.$$

Viceversa, dato un insieme di lunghezze di codice che soddisfano tale disuguaglianza, è possibile costruire un codice univocamente decodificabile con queste lunghezze.

Dimostrazione. Consideriamo l'estensione k-esima del codice C,

$$C^k(x_1, x_2, \ldots, x_k) = C(x_1)C(x_2)\cdots C(x_k).$$

Dalla definizione di codice univocamente decodificabile C^k è non singolare. Poiché il numero delle strighe di lunghezza n con simboli in un alfabeto D-ario esistenti è D^n, la decodificabilità univoca implica che il numero delle sequenze di codici di lunghezza n nella k-esima estensione del codice non può essere maggiore di D^n. Utilizziamo quindi questa affermazione per provare la disuguaglianza. La lunghezza di una sequenza di codice esteso è

$$l(x_1, x_2, \ldots, x_k) = \sum_{i=1}^{k} l(x_i).$$

Proviamo che vale la disuguaglianza di Kraft $\sum_{x\in\mathcal{A}} D^{-l(x)} \leq 1$. L'idea è quella si considerare la potenza k-esima di questa quantità, quindi

$$\begin{aligned}\left(\sum_{x\in\mathcal{A}} D^{-l(x)}\right)^k &= \sum_{x_1\in\mathcal{A}}\sum_{x_2\in\mathcal{A}}\cdots\sum_{x_k\in\mathcal{A}} D^{-l(x_1)}D^{-l(x_2)}\ldots D^{-l(x_k)} \\ &= \sum_{x_1,x_2,\ldots,x_k} D^{-l(x_1)}D^{-l(x_2)}\ldots D^{-l(x_k)} \\ &= \sum_{x^k\in\mathcal{A}^k} D^{-l(x^k)},\end{aligned}$$

raccogliendo i termini rispetto alle lunghezze delle parole otteniamo

$$\sum_{x^k \in \mathcal{A}^k} D^{-l(x_k)} = \sum_{m=1}^{kl_{max}} a(m) D^{-m}$$

dove l_{max} è la massima lunghezza delle parole di codice ed $a(m)$ è il numero delle sequenze x^k a cui sono associate parole di codice di lunghezza m.

Poiché il codice è univocamente decodificabile esiste al massimo una sequenza x^k per ciascuna parola di codice di lunghezza m, ed esistono al massimo D^m parole di codice di lunghezza m.

Quindi $a(m) \leq D^m$ ed abbiamo

$$\begin{aligned}\left(\sum_{x^k \in \mathcal{A}^k} D^{-l(x^k)}\right)^k &= \sum_{m=1}^{kl_{max}} a(m) D^{-m} \\ &\leq \sum_{m=1}^{kl_{max}} D^m D^{-m} \\ &= kl_{max},\end{aligned}$$

e quindi dimostriamo la disuguaglianza di Kraft

$$\sum_i D^{-l_i} \leq (kl_{max})^{\frac{1}{k}} .$$

Viceversa, date $l_1, l_2, \ldots, l_m$ che soddisfano la disuguaglianza di Kraft, possiamo costruire un codice istantaneo come mostrato nella dimostrazione del teorema (10). Poiché ogni codice istantaneo è univocamente decodificabile, il codice costruito è univocamente decodificabile. □

Capitolo 2

Compressori Unidimensionali

2.1 Cenni storici

La compressione dati assunse un ruolo significativo a partire dagli anni '70, in contemporanea con l'aumento della popolarità di informatica e telematica e con la pubblicazione degli algoritmi di Abrahm Lempel e Jacob Ziv, ma la sua storia ebbe inizio molto prima. I primi esempi di compressione dati si hanno in due codici molto famosi, il codice Morse ed il codice Braille. Il codice Morse, riportato in Figura 2.1, fu inventato nel 1838. In esso le lettere più comuni nella lingua inglese come *"E"* e *"T"* corrispondono a codici di minore lunghezza. Il codice Braille, fu introdotto nel 1829. La sua scrittura richiede molto più spazio della scrittura in nero, questo fatto ha comportato l'introduzione, in alcune lingue, di un codice modificato, detto "Braille contratto", in cui un singolo segno rappresenta singole parole o particolari gruppi di lettere (ad esempio, in inglese, si ricorre ad un carattere contratto per la rappresentazione del suffisso "–ing").

In seguito, con l'invenzione dei primi calcolatori, nel 1949, Claude Shannon e Robert Fano inventarono la codifica di Shannon-Fano. Il loro algoritmo assegna parole di codice più corte ai simboli più frequenti.

Tre anni più tardi David Huffman elaborò una tecnica di compressione molto simile a quella di Shannon-Fano ma più efficiente. La differenza fondamentale tra queste tecniche sta nella costruzione dell'albero di probabilità, Huffman riuscì a creare un risultato ottimale. Nel corso degli anni '70, con la diffusione di Internet, iniziarono ad essere sviluppati molteplici software di compressione basati sulla codifica di Huffman.

Nelle situazioni reali non si hanno conoscenze a priori sulle caratteristiche della sorgente, in pratica non si conoscono le probabilità dei singoli elementi dell'alfabeto. Nasce la necessità di ricercare codici universali,

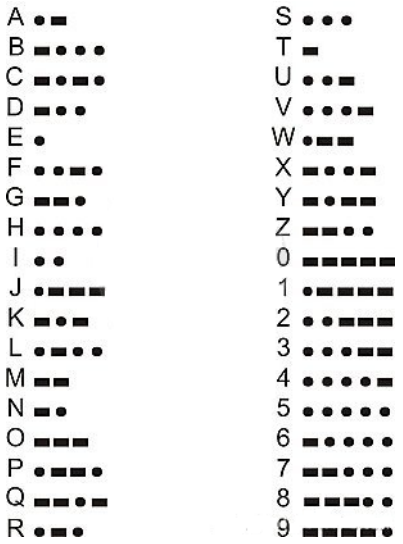

Figura 2.1: Codice Morse: è una codifica su un alfabeto di tre simboli: punto, linea e spazio. Lo spazio è di un'unità di tempo tra due simboli, tre unità di tempo tra due lettere e sei unità di tempo tra due parole.

cioè codici che non richiedono la conoscenza delle probabilità e che danno prestazioni prossime al tasso d'entropia della sorgente, senza conoscerlo, purché si codifichino N-ple sufficientemente lunghe.

Lempel e Ziv furono i primi a prendere in considerazione l'idea dell'utilizzo degli schemi a dizionario adattivo con i loro scritti del 1977 e del 1978. I due articoli descrivono due distinte versioni dell'algoritmo. Ci si riferisce ad esse con LZ77 o Lempel-Ziv con sliding window e con LZ78 o Lempel-Ziv con struttura ad albero. Entrambi gli algoritmi sono universali, crebbero rapidamente in popolarità e ne vennero sviluppate molte varianti. La maggior parte di queste scomparvero velocemente dopo la loro invenzione, mentre una ristretta minoranza è tuttora diffusa, come DEFLATE, LZMA, e LZX.

2.2 Tecniche di compressione

Le varie tecniche di compressione organizzano in modo più efficiente i dati. Sarà quindi necessario un decompressore per ricostruire i dati originali grazie all'algoritmo contrario a quello usato per la compressione. Come controparte la compressione dati necessita però di potenza di calcolo per le operazioni di compressione e decompressione, spesso anche elevata se tali operazioni devono essere eseguite in tempo reale.

Definizione 27. *Il parametro di qualità che valuta l'efficienza della compressione è il* ***rapporto*** *o* ***tasso di compressione****, cioè il quoziente del numero di bit nei dati compressi per il numero di bit nei dati originali, abitualmente espresso in percentuale.*

Esistono diversi algoritmi di compressione, alcuni sono generali e offrono prestazioni simili su molte varietà di dati, altri sono più mirati e in genere permettono compressioni superiori e presuppongono che l'utente conosca in anticipo il tipo di dati da comprimere.

Spesso le prestazioni della compressione dati dipendono da ciò che sappiamo delle caratteristiche del file sorgente, quindi dato un file, le sue caratteristiche possono essere usate per migliorare la compressione della sua stringa in output. Quando queste caratteristiche sono determinate prima della compressione, questa è detta conoscenza a priori delle caratteristiche dei dati da comprimere, ed è utile per ottenere algoritmi di compressione più efficienti.

Ad ogni modo, nella maggior parte delle applicazioni reali non possiamo avere una conoscenza a priori delle caratteristiche del file.

I diversi algoritmi di compressione possono essere raggruppati in due grandi categorie:

- **Compressione statistica**: si basa sullo studio dell'input da comprimere. Si fanno studi statistici sul formato dell'input per ottenere poi una buona compressione. Per esempio, nella compressione di un file di testo, si studia la frequenza relativa di ciascun carattere per associare poi al carattere presente più volte nel testo il codice più corto, viceversa a caratteri presenti con frequenza bassa si associa la parola codice più lunga.

- **Compressione mediante sostituzione di testo o con dizionario**: questa è basata sull'idea di rimpiazzare, in un file, occorrenze di stringhe ripetute con puntatori a precedenti copie. La compressione è dovuta al fatto che la lunghezza di un puntatore è in genere più piccola della lunghezza della stringa che è rimpiazzata. Da ciò risulta, che maggiori sono le ripetizioni di occorrenze di stringhe nel file da comprimere, maggiore è il grado di compressione raggiunta. Gli algoritmi di compressione basati sulla sostituzione, inoltre, vengono frequentemente usati poiché non richiedono una conoscenza a priori delle proprietà del file. Essi possono imparare in modo adattivo dalle caratteristiche del file durante la fase di codifica.

2.3 Codice di Shannon-Fano

Claude E. Shannon, ingegnere e matematico presso i laboratori Bell e professore al MIT, nel suo articolo *"A Mathematical Theory of Communication"* pubblicato nel 1948, presentò quello che oggi è conosciuto come codice di Shannon-Fano in collaborazione con Robert M. Fano. In seguito Peter Elias, un altro membro del MIT, ne presentò un'implementazione ricorsiva.

Come vedremo, nella codifica di Huffman la lunghezza di ogni parola di codice l_i dipende dalla probabilità di ogni simbolo dell'alfabeto. Il metodo di Shannon-Fano è meno efficiente del metodo di Huffman, ma ha il vantaggio che da ognuna delle probabilità p_i si ottiene direttamente la lunghezza l_i.

Siano date le probabilità p_i per ognuno degli A elementi dell'alfabeto $\mathcal{A}$. Dati $x_1, x_2, \ldots, x_A \in \mathcal{A}$ e le loro rispettive probabilità $p_1, p_2, \ldots, p_A$, abbiamo mostrato che il codice di Shannon (1.21) $l_i^{**} = \left\lceil \log \frac{1}{p_i} \right\rceil$ soddisfa la disuguaglianza di Kraft ed è un codice univocamente decifrabile.

Descriveremo una semplice procedura per l'assegnazione dei codici tramite l'utilizzo della funzione di distribuzione cumulativa.

Definizione 28. *Dati $p(x_i) = p_i > 0$ per ogni $i \in \{1, 2, \ldots, A\}$, definiamo la funzione di distribuzione cumulativa $F(x)$ è definita come*

$$F(x) = \sum_{x' \leq x} p(x'). \tag{2.1}$$

Come mostrato in Figura 2.2, $F(x)$ è una funzione a gradini e continua a destra.

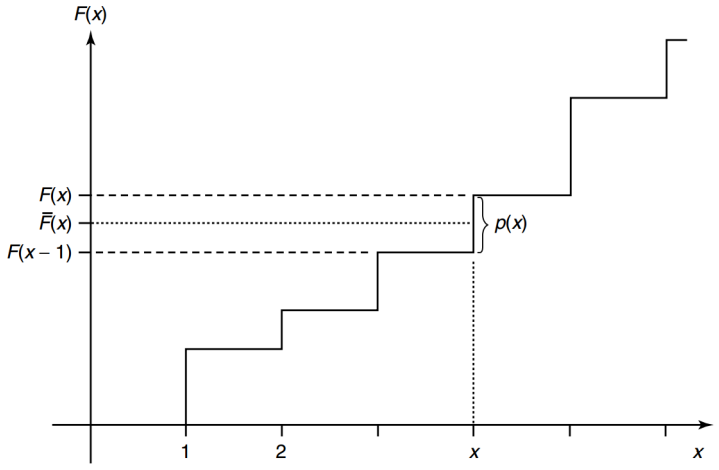

Figura 2.2: Funzione di distribuzione cumulativa

Definizione 29. *Si definisce funzione cumulativa modificata*

$$\bar{F}(x) = \sum_{x'<x} p(x') + \frac{1}{2}p(x) = \frac{F(x-1) + F(x)}{2},$$

$\bar{F}(x)$ denota la somma delle probabilità di tutti i simboli inferiori ad x maggiorata della metà della probabilità di x.

Poiché la variabile aleatoria è discreta, la funzione cumulativa è costituita da gradini ciascuno di altezza $p(x)$. Il valore della funzione cumulativa modificata $\bar{F}(x)$ è il punto medio del gradino corrispondente ad x. Poiché le probabilità sono tutte positive vale

$$\bar{F}(a) \neq \bar{F}(b) \quad \text{se} \quad a \neq b,$$

quindi se conosciamo $\bar{F}(x)$ possiamo determinare x semplicemente osservando il grafico della funzione cumulativa.

Il valore binario di $\bar{F}(x)$ può essere usato come parola di codice per x, ma, in generale, $\bar{F}(x)$ è un numero reale esprimibile solo attraverso un numero infinito di bit. Non risulta quindi efficiente utilizzare il suo esatto valore come codice per x ed occorre quindi approssimarlo in qualche maniera.

Definiamo con $\lfloor \bar{F}(x) \rfloor_{l(x)}$ il troncamento di $\bar{F}(x)$ a $l(x)$ cifre binarie. Usiamo i primi $l(x)$ bit di $\bar{F}(x)$ come parola di codice per x. L'errore di arrotondamento è dato da

$$\bar{F}(x) - \lfloor \bar{F}(x) \rfloor_{l(x)} < \frac{1}{2^{l(x)}}.$$

Se $l(x) = \lceil \log \frac{1}{p(x)} \rceil + 1$, allora

$$\frac{1}{2^{l(x)}} \leq \frac{p(x)}{2} = \bar{F}(x) - F(x-1),$$

e quindi $\lfloor \bar{F}(x) \rfloor_{l(x)}$ si trova all'interno del gradino corrispondente all'elemento x.

Mostriamo che questo codice è senza prefisso, cioè che nessuna parola di codice è prefisso di un'altra. Immaginiamo che ciascuna parola di codice $\alpha_1, \alpha_2, \ldots \alpha_l$ non rappresenti un punto ma un intervallo

$$\left[0.\alpha_1, \alpha_2, \ldots \alpha_l, 0.\alpha_1, \alpha_2, \ldots \alpha_l + \frac{1}{2^l} \right).$$

Il codice è senza prefisso se e solo se gli intervalli corrispondenti a ciascuna parola sono due a due disgiunti. Ognuno di questi intervalli ha lunghezza

$2^{-l(x)}$, che è minore della metà dell'altezza dello scalino corrispondente ad x in Figura 2.2.

Sono quindi sufficienti $l(x)$ bit per descrivere univocamente x in quanto l'errore generato dall'approssimazione rimane all'interno del segmento compreso tra $F(X)$ ed $F(X) - 1$ in Figura 2.2 e quindi non si rischia di compromettere la codifica del dato precedente.

Il codice di Shannon-Fano non è ottimale ma è comunque efficiente, infatti, utilizzando $l(x) = \left\lceil \log \frac{1}{p(x)} \right\rceil + 1$ bits per rappresentare x, il valore atteso della lunghezza del codice è pari a

$$L = \sum_{x \in \mathcal{A}} p(x) l(x) = \sum_{x \in \mathcal{A}} p(x) \left(\left\lceil \log \frac{1}{p(x)} \right\rceil + 1 \right) < H(X) + 2.$$

Esempio 2.1. consideriamo un esempio in cui tutte le probabilità sono biadiche. Si costruisce il codice attraverso una tabella:

x	$p(x)$	$F(x)$	$\bar{F}(x)$	$\bar{F}(x)$ (binario)	$l(x) = \lceil \log \frac{1}{p(x)} \rceil + 1$	Codice
1	0.25	0.25	0.125	0.001	3	001
2	0.5	0.75	0.5	0.10	2	10
3	0.125	0.875	0.8125	0.1101	4	1101
4	0.125	1.0	0.9375	0.1111	4	1111

In questo caso, il valore atteso della lunghezza del codice è pari a 2.75 bits mentre l'entropia è 1.75 bits. Osservando le parole di codice è ovvio che è inefficiente, si può infatti omettere l'ultimo bit delle ultime due parole di codice.

Esempio 2.2. Consideriamo ora un altro esempio per la costruzione del codice di Shannon-Fano. In questo caso, poiché la distribuzione non è biadica, la rappresentazione binaria della funzione cumulativa può avere un numero infinito di bits. Denotiamo $0.01010101\ldots$ con $0.\overline{01}$. Si costruisce il codice attraverso la seguente tabella:

x	$p(x)$	$F(x)$	$\bar{F}(x)$	$\bar{F}(x)$ (binario)	$l(x) = \lceil \log \frac{1}{p(x)} \rceil + 1$	Codice
1	0.25	0.25	0.125	0.001	3	001
2	0.25	0.5	0.375	0.011	3	011
3	0.2	0.7	0.6	$0.1\overline{0011}$	4	1001
4	0.15	0.85	0.775	$0.110\overline{0011}$	4	1100
5	0.15	1.0	0.925	$0.111\overline{0110}$	4	1110

Tale codice è di un bit più lungo in media del codice di Huffman per la stessa sorgente (Esempio 2.4).

La codifica di Shannon-Fano può anche essere applicata a sequenze di variabili aleatorie. Una diretta applicazione del metodo ai blocchi di lunghezza n richiederebbe il calcolo delle probabilità e la costruzione della funzione cumulativa per tutte le sequenze di lunghezza n, un procedimento la cui complessità crescerebbe esponenzialmente con la lunghezza del blocco. Tuttavia esiste un semplice stratagemma per calcolare probabilità e funzione cumulativa sequenzialmente, man mano che si riceve ognuno dei simboli del blocco, di conseguenza la complessità dei calcoli cresce solo linearmente con la lunghezza del blocco.

La precisione necessaria nei calcoli aumenta man mano che cresce la lunghezza del blocco, ciò non risulta essere pratico quando si trattano blocchi molto lunghi. Il problema della precisione finita fu risolto in manera indipendente da J. J. Rissanen e da R. Pasco nel 1976. In seguito, nel 1979, Rissanen e Langdon presentarono nel loro articolo *"Arithmetic coding"* la codifica aritmetica, un'estensione del metodo di Shannon-Fano.

2.4 Codifica Aritmetica

La codifica aritmetica è una forma di codifica entropica senza perdita sviluppata da J. J. Rissanen e da R. Pasco a partire dalla codifica di Shannon-Fano, e presentata nel loro articolo *"Arithmetic coding"* del 1979. Questo algoritmo codifica l'intero messaggio con un solo numero in virgola mobile che rappresenta in maniera univoca un sottointervallo aperto di $[0, 1)$.

L'algoritmo di compressione parte definendo un modello dei dati, cioè dei simboli $x_1, x_2, \ldots, x_m$ dell'alfabeto $\mathcal{A}$, e una loro distribuzione probabilistica che influenza in misura preponderante la bontà della compressione.

Per determinare gli intervalli si suddivide innanzitutto l'intervallo $[0, 1)$ in sottointervalli di ampiezza proporzionale alla frequenza di ogni simbolo dei dati da codificare della forma $[F(x_{j-1}), F(x_j))$ per $j = 1, 2, \ldots, m$, dove $F(x)$ indica la funzione di distribuzione cumulativa (2.1). L'ordine di tali intervalli non è importante purché questo sia lo stesso sia nel processo di codifica sia nel processo di decodifica.

L'algoritmo prende in ingresso un simbolo alla volta e, tramite elaborazioni numeriche, riduce l'intervallo che rappresenta il messaggio, e quindi aumenta il numero di bit necessari a rappresentarlo.

Supponiamo che il primo simbolo in input sia x_k. L'intervallo corrispondente ad x_k è $[F(x_{k-1}), F(x_k))$, l'algoritmo procede con la suddivisione in sottointervalli

$$\left[F(x_{k-1}) + \frac{F(x_{j-1})}{F(x_k) + F(x_{k-1})}, F(x_{k-1}) + \frac{F(x_j)}{F(x_k) + F(x_{k-1})}\right)$$

ciascuno corrispondente ad uno degli elementi x_j, $j = 1, 2, \ldots, m$. Per ognuno degli elementi della sequenza da codificare, l'algoritmo effettua una suddivisione dell'intervallo corrispondente all'elemento in input in sottointervalli di questo tipo. Alla fine del processo di codifica verrà elaborato un numero decimale, che il decodificatore sarà in grado di decifrare conoscendo solo le informazioni di base del processo (alfabeto e distribuzione di probabilità).

Esempio 2.3. Per fornire un esempio di questo tipo di algoritmo calcoliamo la codifica aritmetica della stringa ABC sull'alfabeto $\Sigma = \{A, B, C\}$, con $p(A) = 0.7$, $p(B) = 0.1$ e $P(C) = 0.2$. Come prima operazione si definiscono i valori della funzione cumulativa: $F(A) = 0.7$, $F(B) = 0.8$ e $F(C) = 1$.

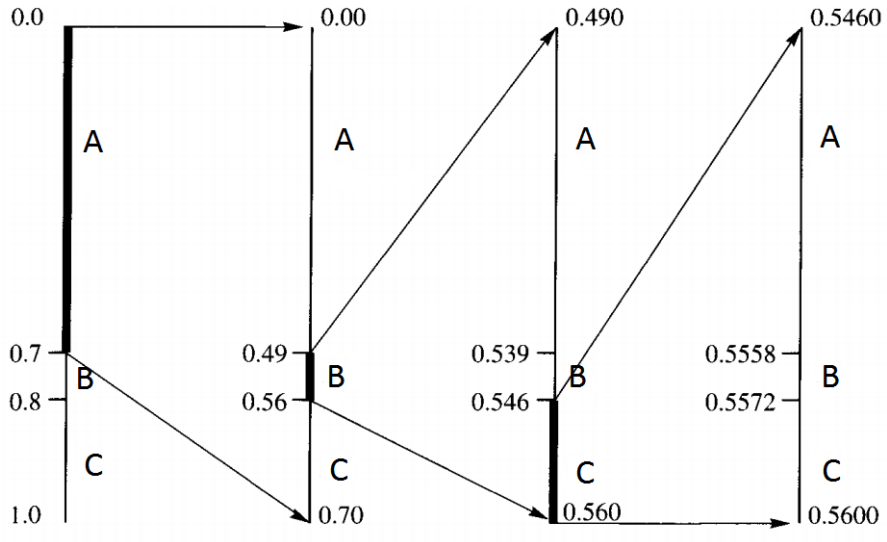

Figura 2.3: Suddivisione degli intervalli nella codifica aritmetica.

Poiché il primo simbolo in input è A, il primo intervallo da considerare è $[0, 0.7)$. Una volta determinato l'intervallo contenente la parola di codice, il resto dell'intevallo unitario non viene più considerato e l'intervallo ristretto $[0, 0.7)$ viene di nuovo suddiviso in sottointervalli come mostrato graficamente in Figura 2.3.

Notiamo che l'intervallo ottenuto per la codifica aritmetica di una particolare sequenza di simboli è disgiunto dagli intervalli che si potrebbero ottenere per la codifica aritmetica di qualsiasi altra stringa. In pratica ogni valore contenuto in questo intervallo descrive univocamente la stringa. Spesso si sceglie di utilizzare il limite inferiore dell'intervallo, oppure il punto medio. L'importante è trasmettere un valore che identifichi completamente l'intervallo,

occorre cioè inviare una sequenza di cifre binarie del punto scelto sufficientemente lunga in modo che l'intervallo diadico definito sia completamente contenunto nell'intervallo che identifica il messaggio.

Differentemente dalla fase di codifica in cui ogni volta si restringeva l'intervallo in cui cadeva il numero di output, nella fase di decodifica si allarga l'intervallo man mano che vengono decodificati i simboli. Il problema di come far terminare l'algoritmo quando non ci sono più simboli da codificare si può risolvere utilizzando l'informazione sulla lunghezza del messaggio originale.

Il decodificatore opera a partire dal valore codificato x che rappresenta un punto dell'intervallo $[0, 1)$ conoscendo la lunghezza della sequenza da codificare N. Si inizializza l'intervallo corrente a $[0, 1)$ e per ciascuno degli N caratteri da decodificare si suddivide l'intervallo corrente in sottointervalli, uno per ciascun simbolo dell'alfabeto. L'ampiezza di ciascun sottointervallo è definita proporzionalmente alla probabilità (stimata) che il carattere corrente sia uguale al simbolo corrispondente all'intervallo. Si seleziona il carattere corrispondente al sottointervallo in cui cade il valore x e si prende tale sottointervallo come intervallo corrente.

La codifica aritmetica presenta due principali difetti: lo scalamento dell'intervallo richiede l'impiego di aritmetica ad alta precisione, ed inoltre la rappresentazione compressa del messaggio originario non può essere prodotta in uscita fintanto che non si termina l'intera codifica.

Una delle caratteristiche più interessanti di questa codifica è la possibilità di ottenere meno di un bit di informazione codificata per ciascun simbolo in ingresso.

Uno dei vantaggi della codifica aritmetica è la convenienza nell'effettuare l'adattamento, ossia nel modificare la frequenza dei caratteri (e quindi la probabilità) nel corso dell'elaborazione dei dati. Nella decodifica si riottengono i dati originali in quanto la tabella delle frequenze viene modificata nello stesso modo e nella stessa fase della codifica. L'adattamento migliora significativamente il rapporto di compressione, aumentandone di 2 o 3 volte il tasso di compressione.

2.5 Codifica di Huffman

Presentata nell'articolo *A Method for the Construction of Minimum - Redundancy Codes* pubblicato nel 1952 da David A. Huffman, è una delle prime tecniche di compressione dati ed è tra le più conosciute.

Nel 1951 a David Huffman, ai tempi dottorando in Teoria dell'Informazione al Massachusetts Institute of Technology (MIT), fu assegnata da Robert M. Fano una tesi sul problema di trovare il codice binario più ef-

ficiente per rappresentare una sorgente probabilistica. Ad Huffman venne l'idea di usare un albero binario ordinato in base alle frequenze dei singoli simboli emessi dalla sorgente e così inventò un nuovo algoritmo di codifica che creava un codice a lunghezza variabile.

L'algoritmo di Huffman consiste nella costruzione ricorsiva di un albero binario. Ricordando che l'obiettivo è ottenere la minore lunghezza media possibile, si intuisce che a simboli meno probabili devono corrispondere parole di codice aventi lunghezza maggiore, ovvero foglie dell'albero aventi ordine maggiore.

È stato dimostrato che la codifica di Huffman è ottimale: nel caso in cui le frequenze effettive dei simboli corrispondano a quelle usate per creare il codice nessun'altra mappatura di simboli in stringhe binarie può produrre un risultato più breve.

Huffman si accorse del fatto che, in un codice istantaneo, i due simboli di minore probabilità devono avere parole di codice della stessa lunghezza, altrimenti è possibile eliminare un bit dalla parola di codice più lunga ed ottenere un codice ancora istantaneo, ma con lunghezza media minore. In generale, si costruisce un codice nel quale le due parole più lunghe differiscono solo nell'ultimo bit.

Sia $\mathcal{S}_1$ una sorgente a valori in un alfabeto $\mathcal{A} = \{x_1, x_2, \ldots, x_A\}$, siano $p_1, p_2, \ldots, p_A$ le rispettive probabilità dei simboli di $\mathcal{A}$. L'algoritmo di Huffman riordina i simboli in ordine decrescente e considera i due simboli meno probabili $x_{i_{A-1}}$ e x_{i_A}, con $i_{A-1}, i_A \in \{1, 2, \ldots, m\}$, come un unico simbolo di probabilità $p_{i_{A-1}} + p_{i_A}$. Si ottiene in questo modo una sorgente $\mathcal{S}_2$ a valori in un alfabeto $\mathcal{A}_2$ di $A-1$ simboli. Si ripete tale procedimento finché non si ottiene ad una sorgente con un alfabeto di soli 2 simboli. Dopo la costruzione della sequenza di sorgenti $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_{A-1}$, si assegna alla sorgente di due soli simboli il codice composto dalle parole 0 e 1. Si ripercorre all'indietro la sequenza di sorgenti $\mathcal{S}_i$. Ad ogni passo, per i simboli che corrispondono a simboli singoli in entrambe le sorgenti $\mathcal{S}_{i+1}$ ed $\mathcal{S}_i$, la parola di codice resta invariata, mentre ai due simboli di $\mathcal{S}_i$ che in $\mathcal{S}_{i+1}$ sono raggruppati in un'unica parola di codice si fa corrispondere la parola stessa con l'aggiunta di uno 0 finale al primo dei due simboli, e di un 1 finale all'altro.

In pratica l'algoritmo di Huffman lavora costruendo un albero di decodifica dal basso verso l'alto. Per ogni simbolo crea un nodo, detto foglia, contenente il simbolo e la sua probabilità. In seguito i due nodi con probabilità minori vengono riuniti diventando rami fratelli al di sotto di uno stesso nodo superiore, detto padre, il quale ha probabilità pari alla somma di quelle dei suoi figli.

Esempio 2.4. Consideriamo una sorgente $\mathcal{S}$ a valori in un alfabeto $\mathcal{A}$, con $\mathcal{A} = \{1, 2, 3, 4, 5\}$, con rispettive probabilità dei simboli $p_1 = 0.25$, $p_2 = 0.25$, $p_3 = 0.2$, $p_4 = 0.15$, $p_5 = 0.15$. Si effettua il procedimento descritto combinando i due simboli meno probabili in un unico simbolo fino a rimanere con due soli simboli, come mostrato in Figura 2.4.

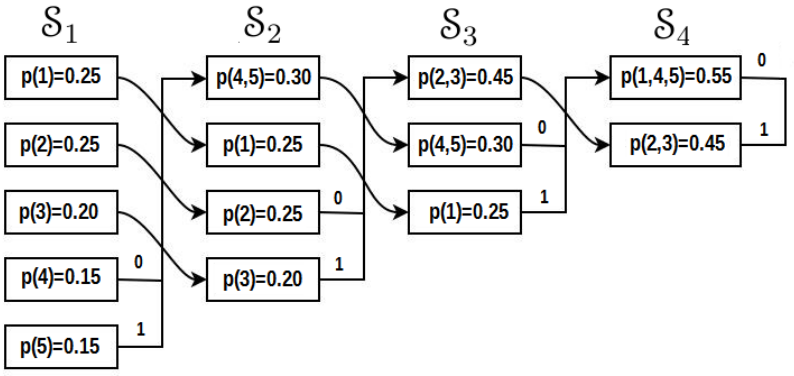

Figura 2.4: Esempio di codifica di Huffman

Ripercorrendo la successione delle sorgenti ottenuta all'indietro, si ottiene il codice mostrato nella tabella seguente. Il valore atteso della sua lunghezza è pari a 2.3 bits, un bit in meno rispetto alla codifica di Shannon-Fano per la stessa sorgente (Esempio 2.2).

x	$p(x)$	$l(x)$	Codice
1	0.25	2	01
2	0.25	2	10
3	0.2	2	11
4	0.15	3	000
5	0.15	3	001

2.5.1 Ottimalità del codice di Huffman

Proviamo per induzione che il codice di Huffman è ottimale. È importante ricordare che ci sono molti codici ottimali, infatti invertendo tutti i bit o scambiando due parole di codice della stessa lunghezza si ottiene ancora un codice ottimale. Per provare l'ottimalità, proviamo innanzitutto alcune proprietà di un particolare codice ottimale. Assumiamo, senza perdere generalità, che i simboli siano ordinati in modo che le probabilità

siano $p_1 \geq p_2 \geq \ldots \geq p_A$. Ricordiamo che un codice è ottimale se è minima la somma $\sum_{i=1}^{m} p_i l_i$, ove l_i è la lunghezza della parola di codice associata al simbolo x_i di probabilità p_i.

Lemma 4. *Per ogni distribuzione, esiste un codice istantaneo ottimale che soddisfa le seguenti proprietà:*

i) le lunghezze delle parole di codice sono ordinate inversamente rispetto alle probabilità dei simboli a cui esse corrispondono, cioè se $p_j > p_k$, allora $l_j \leq l_k$.

ii) le due parole di codice più lunghe hanno pari lunghezza.

iii) due delle parole di codice più lunghe differiscono solo per l'ultimo bit e corrispondono ai due simboli meno probabili.

Dimostrazione. Sia C_A un codice ottimale.

i) Sia C'_A il codice C_A in cui le parole di codice di due simboli x_j ed x_k sono state scambiate tra loro. Vale

$$L(C'_A) - L(C_A) = \sum_{i=1}^{A} p_i l'_i - \sum_{i=1}^{A} p_i l_i$$
$$= p_j l_k + p_k l_j - p_j l_j - p_k l_k = (p_j - p_k)(l_j - l_k).$$

Poiché $p_j - p_k > 0$, e C_A è un codice ottimale, vale $L(C'_A) - L(C_A) \geq 0$ e, di conseguenza, $l_k \geq l_j$. Quindi un codice C_A ottimale deve necessariamente soddisfare la proprietà (i).

ii) Poiché C_A è un codice senza prefisso, se le due parole più lunghe del codice non hanno la stessa lunghezza, eliminando l'ultimo bit della parola più lunga si ottiene un altro codice senza prefisso. Nel codice che si ottiene, il valore atteso della lunghezza risulta minore rispetto a quella del codice di partenza C_A. Quindi, affinché C_A sia ottimale, le due parole di codice più lunghe devono avere stessa lunghezza e, per la proprietà (i), devono corrispondere ai simboli con probabilità più basse.

iii) Non tutti i codici ottimali soddisfano questa proprietà ma, attraverso opportune trasformazioni di un codice C_A ottimale, è possibile ottenere un codice che la soddisfi. Notiamo, innanzitutto, che ognuna delle parole del codice con lunghezza massima possiede una parola sorella, cioè una parola del codice della stessa lunghezza che differisce dalla prima solo per l'ultimo simbolo. Due parole di questo tipo corrispondono a foglie

di rami fratelli dell'albero di Huffman. Se ciò non si verificasse, C_A non sarebbe ottimale, in quanto, eliminando l'ultimo bit dalla parola di lunghezza massima senza sorella, si ottiene un altro codice senza prefisso di lunghezza media minore. Quindi è possibile scambiare tra loro le parole di codice di maggiore lunghezza e far corrispondere ai due simboli di minore probabilità due parole fra loro sorelle. In questo modo il valore atteso della lunghezza del codice ottenuto $\sum_{i=1}^{A} p_i l_i$ rimane invariato, inoltre la (iii) è soddisfatta.

Riassumendo, abbiamo mostrato che se $p_1 \geq p_2 \geq \ldots \geq p_A$, esiste un codice con lunghezze delle parole $l_i \leq l_2 \leq \ldots \leq l_{A-1} = l_m$, e parole di codice $C(x_{A-1})$ e $C(x_A)$ che tra loro differiscono solamente nell'ultimo bit. □

Definizione 30. *I codici che soddisfano le tre proprietà del lemma precedente si dicono codici canonici.*

Per ogni distribuzione di probabilità su un alfabeto $\mathcal{A}$ di cardinalità A, $\mathbf{p} = (p_1, p_2, \ldots, p_A)$ con $p_1 \geq p_2 \geq \ldots \geq p_A$, definiamo la riduzione di Huffman $\mathbf{p}' = (p_1, p_2, \ldots, p_{A-1} + p_A)$ su un alfabeto di cardinalità $A - 1$.

Consideriamo un codice di Huffman $C_H(\mathbf{p})$, dimostriamo per induzione che questo è ottimale per $\mathbf{p}$. In seguito mostriamo che se $C(\mathbf{p})$ è un altro codice con le stesse probabilità dei simboli, vale $L(C_H(\mathbf{p})) \leq L(C(\mathbf{p}))$.

Per costruzione, il codice di Huffman $C_H(\mathbf{p})$ è canonico. È possibile costruire il codice della riduzione di Huffman per $\mathbf{p}'$ accorpando le parole di codice dei due simboli meno probabili con probabilità p_{A-1} e p_A in un unico simbolo con probabilità $p_{A-1} + p_A$. Il codice ottenuto in questo modo è un codice di Huffman sull'alfabeto di $A - 1$ simboli, e lo denominiamo $C_H(\mathbf{p}')$. Un codice di Huffman su un alfabeto $\mathcal{A}'$ di cardinalità 2 è costituito da due parole di codice, 0 e 1, ed è ovviamente ottimale. Per induzione sulla cardinalità dell'alfabeto $\mathcal{A}'$, si dimostra che ogni codice di Huffman su un alfabeto di $A - 1$ simboli è ottimale.

Sia $C_c(\mathbf{p})$ un qualsiasi codice canonico su $\mathbf{p}$, allora vale che

$$L(C(\mathbf{p})) \geq L(C_c(\mathbf{p})). \tag{2.2}$$

Poiché $C_c(\mathbf{p})$ è canonico, se ne può fare la riduzione di Huffman, che chiamiamo $C'(\mathbf{p}')$. Calcolando il valore atteso della lunghezza del codice $C'_c(\mathbf{p}')$ si ha

$$\begin{aligned} L(C'_c(\mathbf{p}')) =& \sum_{i=1}^{A-2} p_i l_i + p_{A-1}(l_{A-1} - 1) + p_A(l_A - 1) \\ =& L(C_c(\mathbf{p})) - p_{A-1} - p_A, \end{aligned} \tag{2.3}$$

in quanto, nella costruzione della riduzione di Huffman, gli elementi di probabilità minore, x_{A-1} ed x_A, sono accorpati in un unico elemento di probabilità $p_{A-1} + p_A$, al quale corrisponde una parola di codice di lunghezza minore di un bit rispetto alle lunghezze $l_{A-1} = l_A$. Inoltre, siccome $C_H(\mathbf{p}')$ è ottimale per p', vale

$$L(C'_c(\mathbf{p}')) \geq L(C_H(\mathbf{p}')). \tag{2.4}$$

Osserviamo infine che, per costruzione, vale

$$L(C_H(\mathbf{p})) = L(C_H(\mathbf{p}')) + p_{A-1} + p_A. \tag{2.5}$$

Dalle (2.2), (2.3), (2.4) e (2.5), si ottiene

$$L(C(\mathbf{p})) \geq L(C_H(\mathbf{p})), \tag{2.6}$$

e quindi si ha che il codice di Huffman $C_H(\mathbf{p})$ ha valore atteso della lunghezza minore o uguale al valore atteso della lunghezza di un qualsiasi codice istantaneo $C(\mathbf{p})$. Vale quindi il teorema seguente.

Teorema 14. *La codifica di Huffman è ottimale. Se C_H è un codice di Huffman e C' è un altro codice univocamente decodificabile, si ha*

$$L(C_H) \leq L(C').$$

2.6 Algoritmo di Lempel e Ziv LZ77

L'algoritmo di compressione LZ77 fu presentato per la prima volta da A.Lempel e J.Ziv nel loro articolo *"A Universal Algorithm for Sequential Data Compression"* [5]. L'idea chiave di questo algoritmo è quella di effettuare una suddivisione della stringa in input in sequenze e di sostituire le sequenze con puntatori al punto della stringa in cui le stesse sono comparse precedentemente.

L'algoritmo descritto codifica una stringa esaminando una finestra scorrevole, sliding window, di una sequenza di simboli appena codificati, cercando in essa il prefisso di maggiore lunghezza della sequenza da codificare che è presente anche nella sliding window. La sequenza viene quindi rappresentata con un puntatore che si riferisce al punto di inizio della sequenza nella window.

Nel loro articolo del 1977 Lempel e Ziv non provarono che il loro algoritmo è asintoticamente ottimale, cioè che il tasso di compressione converge all'entropia per sorgenti ergodiche. Questo risultato fu successivamente dimostrato da Wyner e Ziv nel loro articolo del 1994 *"The Sliding Window Lempel-Ziv Algorithm is Asymptotically Optimal"* [6].

La dimostrazione si basa su un semplice lemma dovuto a Kac che afferma che, per una sequenza stazionaria ed ergodica di variabili aleatorie in un alfabeto numerabile, il tempo medio che occorre attendere prima che compaia un particolare simbolo è l'inverso della probabilità che tale simbolo compaia. Quindi, prendendo una sliding window di dimensioni adeguate, le stringhe con probabilità più alta compaiono più frequentemente al suo interno e l'algoritmo riesce a codificarle efficientemente. Al contrario, le stringhe che non si trovano all'interno della finestra hanno una probabilità molto bassa tale che, asintoticamente, non influenzano il tasso di compressione.

2.6.1 Descrizione dell'Algoritmo LZ77

Definizione 31. *Un processo stocastico $\{X\}_{k=-\infty}^{\infty}$ a valori in un alfabeto finito $\mathcal{A}$ si dice ergodico se, per ogni stringa $y_1 y_2 \dots y_k$, per $\mathbb{P}$-q.o. realizzazione $(x_n)_{-\infty}^{\infty}$ del processo stocastico, vale*

$$\frac{|\{j \in \{-n, \dots, n\} : x_j x_{j+1} \dots x_{j+k-1} = y_1 y_2 \dots y_k\}|}{2n+1} \xrightarrow{n\to\infty} \mathbb{P}(y_1 y_2 \dots y_k).$$

Assumiamo che la sorgente d'informazione discreta sia un processo stocastico stazionario ergodico $\{X\}_{k=-\infty}^{\infty}$ a valori in un alfabeto finito $\mathcal{A}$ e che entrambi il codificatore ed il decodificatore abbiano accesso a tutti gli elementi passati $\dots, X_{-2}, X_{-1}$. Per $-\infty \le i \le j \le \infty$ rappresentiamo con X_i^j la sottostringa $(X_i, X_{i+1}, \dots, X_j)$. Sia

$$H_n = \frac{1}{n} \sum_{x \in \mathcal{A}^n} H(X_1^n) = -\frac{1}{n} \sum_{x \in \mathcal{A}^n} \mathbb{P}\{X_1^n = x\} \log(\mathbb{P}\{X_1^n = x\})$$

l'entropia normalizzata di $\{X_k\}_{k=-\infty}^{\infty}$ di ordine n. Definiamo con

$$H = \lim_{n\to\infty} H_n$$

l'entropia della sorgente. Sappiamo che tale limite esiste sempre per un processo stocastico stazionario e che i dati della sorgente possono essere codificati senza perdita utilizzando $H + \epsilon$ bits per simbolo, con $\epsilon > 0$ arbitrario.

Indichiamo con W intero positivo la dimensione della finestra. I primi W simboli della sorgente saranno codificati senza effettuare la compressione, la finestra sarà inizialmente costituita dai simboli X_1^W.

Il numero di bits richiesti per codificare la finestra X_1^W è $\lceil W \log A \rceil$, con $A = |\mathcal{A}| < \infty$. Questi bits verranno ammortizzati facendo tendere la lunghezza della stringa da codificare all'infinito.

Definizione 32. *Si definisce parsing di una stringa* $(X_1, X_2, \ldots, X_M)$ *di simboli di* $\mathcal{A}$ *una suddivisione della sequenza in blocchi lunghezza variabile* $Y_1, Y_2, \ldots, Y_C$ *che chiameremo frasi.*

Per iniziare la codifica occorre trovare la prima frase del parsing $Y_1 = X_{W+1}^{W+L_1}$ dove L_1 ha la proprietà di essere l'intero più grande tale che esiste m, $m \in [0, W-1]$

$$X_{W+1}^{W+L_1} = X_{W-m}^{W-m+L_1-1}. \tag{2.7}$$

L_1 rappresenta quindi la lunghezza massima della stringa che inizia dal simbolo nella posizione $W+1$ e che ha una copia di se stessa che inizia all'interno della finestra X_1^W. Sia ora m_1 il più piccolo m che soddisfa (2.7). Se la ricerca di una corrispondenza ha esito negativo, cioè se $X_{W+1} \neq X_m$ per ogni $m \in [1, W]$, poniamo $L_1 = 1$ e quindi $Y_1 = X_{W+1}$. Prendiamo il primo

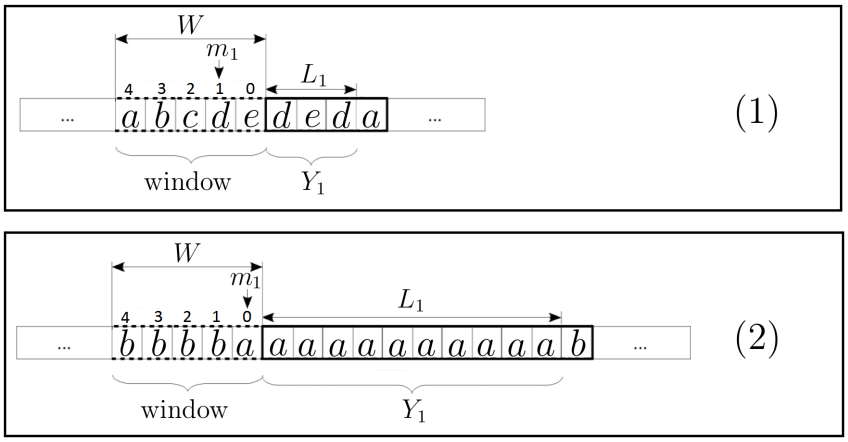

Figura 2.5: Esempi di codifica LZ77.

esempio in Figura 2.5, con $W = 5$ e $(X_1, X_2, \ldots) = (a, b, c, d, e \mid d, e, d, a \ldots)$, si ha $L_1 = 3$ poiché la stringa (d,e,d) inizia nel quarto simbolo della finestra, cioè $X_{W+1}^{W+3} = X_{W-1}^{W-1-2}$, quindi $m_1 = 1$. Notiamo che L_1 non è superiormente limitato, in particolare L_1 può superare W, nel secondo esempio in Figura 2.5 con $W = 5$ e $(X_1, X_2, \ldots) = (b, b, b, b, a \mid a, a, a, a, a, a, a, a, a, a, b, \ldots)$, si ha $L_1 = 10$.

Mostriamo ora uno schema per codificare in maniera univocamente decodificabile un intero L in una stringa di codice binario.

Sia $\hat{e}(k) = u(|b(k)|) * b(k)$ la rappresentazione di k tramite concatenazione della rappresentazione unaria di $|b(k)|$, $u(|b(k)|) = 0^{|b(k)|}1$, con $b(k)$ rappresentazione binaria di k. Poiché la lunghezza della rappresentazione binaria di k è data da $|b(k)| = \lceil \log(k+1) \rceil$, la lunghezza della rappresentazione $\hat{e}(k)$ sarà pari a $2|b(k)| = 2\lceil \log(k+1) \rceil$.

Sia ora

$$e(L) = \hat{e}(|b(L)|) * b(L),$$

con il prefisso $\hat{e}(|b(L)|)$ indichiamo la lunghezza della rappresentazione binaria di L mentre i successivi $|b(L)|$ bits indicano la rappresentazione binaria di L. Avremo

$$|e(L)| = 2|b(|b(L)|)| + |b(L)|.$$

Notiamo che per L grande

$$|e(L)| \sim \log L + 2 \log\log L.$$

La codifica LZ77 è costituita dalla stringa binaria $e(L_1)$ seguita dalla stringa binaria s_1 con

$$s_1 = \begin{cases} \text{codifica binaria di } m_1 & \text{se } \log W < \lceil L_1 \log A \rceil \\ \text{codifica binaria (non compressa) di } Y_1 & \text{se } \log W \geq \lceil L_1 \log A \rceil. \end{cases}$$

Assumiamo che la lunghezza della finestra W sia maggiore della dimensione dell'alfabeto A. Il numero totale di bits necessari a codificare la prima frase $Y_1 = X_{W+1}^{W+L_1}$ sarà pari a $|e(L_1)| + |s_1|$. Poiché $m_1 \in [0, W-1]$, la sua codifica binaria richiede $\log W$ bits. Per effettuare la codifica binaria di Y_1 senza compressione sono necessari $\lceil L_1 \log A \rceil$ bits. Quindi per codificare Y_1 sono necessari

$$|e(L_1)| + \min(\log W, \lceil L_1 \log A \rceil) \leq \min(\log W + \gamma_1 \log(L_1 + 1), \gamma_2 L_1), \quad (2.8)$$

bits, per γ_1,γ_2 abbastanza grandi.

Osserviamo ora che la conoscenza della finestra scorrevole X_1^W, di $e(L_1)$ e di s_1 permette al decodificatore di ricostruire Y_1. Il primo passo è decodificare $e(L_1)$ ottenendo L_1. A questo punto il decodificatore calcola $\lceil L_1 \log A \rceil$. Se tale quantità è minore di $\log W$ i successivi $\lceil L_1 \log A \rceil$ corrispondono alla codifica binaria di Y_1 non compresso. In caso contrario i successivi $\log W$ bits definiscono un puntatore alla posizione $W - m_1$ in cui inizia una copia di Y_1. Questa posizione si trova all'interno della finestra, cioè $W - m_1 \in [1, W]$. Il decodificatore copia X_{W-m_1} nella posizione $W + 1$, poi X_{W-m_1+1} nella posizione $W + 2$ e così via. Quando $X_{W-m_1+L_1-1}$ viene copiato nella posizione $W + L_1$ il processo è completato e Y_1 è completamente ricostruita. Per esempio, consideriamo nuovamente il primo caso della Figura 2.5, il decodificatore conosce la finestra di lunghezza $W = 5$, $L_1 = 3$ e $m_1 = 1$. Come mostrato nella figura 2.6, attraverso la conoscenza di $X_1^5 = (a, b, c, d, e)$ il decodificatore copia d ed e nelle posizioni 6 e 7 rispettivamente, poi copia d dalla posizione 6 alla posizione 8.

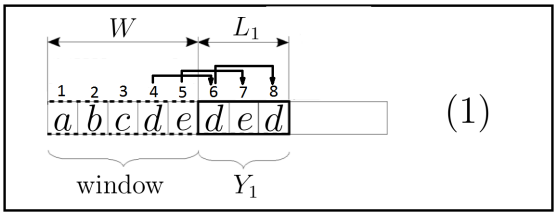

Figura 2.6: Esempio di decodifica LZ77.

Dopo aver codificato la prima frase Y_1, i primi L_1 simboli della finestra vengono rimossi e gli L_1 simboli di Y_1 vengono aggiunti in fondo alla finestra. Avremo quindi una nuova finestra $X_{L_1+1}^{L_1+W}$ di lunghezza W che verrà utilizzata per codificare la seconda frase. Il processo viene ripetuto fino al termine degli N simboli da codificare.

Indichiamo le frasi ottenute con $Y_1, Y_2, \ldots, Y_C$, mentre le rispettive lunghezze con $L_1, L_2, \ldots, L_C$.

Da (2.8) il numero totale dei bits necessari per codificare la sequenza X_1^N è dato da:

$$l(X_1^N) = \sum_{i=1}^{C} \{|e(L_i)| + \min(\log W, \lceil L_i \log A \rceil)\} + \lceil W \log A \rceil,$$

dove l'ultimo termine rappresenta il numero di bits necessari a codificare la finestra iniziale.

Notiamo inoltre che C e $\{L_i\}_{i=1}^{C}$ sono variabili aleatorie. Il tasso di compressione medio è dato da

$$\bar{\rho}(N) = \mathbb{E}\left[\frac{l(X_1^N)}{N}\right] \leq \frac{\lceil W \log A \rceil}{N} + \frac{1}{N}\sum_{i=1}^{C} \{|e(L_i)| + \min(\log W, \lceil L_i \log A \rceil)\}. \tag{2.9}$$

Utilizzando la disuguaglianza (2.8) otteniamo:

$$\bar{\rho}(N) \leq \frac{\lceil W \log A \rceil}{N} + \frac{1}{N}\mathbb{E}\left[\sum_{i=1}^{C} \min(\log W + \gamma_1 \log(L_i + 1), \gamma_2 L_i)\right]. \tag{2.10}$$

2.6.2 Matematica Preliminare

Prima procedere con la dimostrazione dell'ottimalità asintotica di LZ77 è necessario riportare alcuni risultati utili.

Lemma 5. *(di Kac) Sia $\{X_k\}_{k=-\infty}^{\infty}$ un processo stocastico stazionario ergodico in un alfabeto numerabile $\mathcal{A}$, $X_n \in \mathcal{A}$ per ogni n. Sia $y \in \mathcal{A}$ tale*

che $p(\{X_0 = y\}) > 0$. *Indichiamo con* $Q_y(i)$ *la probabilità condizionata alla conoscenza di* $X_0 = y$ *che la prima occorrenza di* y *scorrendo la sequenza all'indietro a partire da* X_0 *sia al posto* $-i$, *cioè:*

$$Q_y(i) = p(\{X_{-i} = y, X_j \neq y \text{ per } -i < j \leq -1 | X_0 = y\}), \qquad i = 1, 2, \ldots$$

Allora vale:

$$\sum_{i=0}^{\infty} iQ_y(i) = \frac{1}{p(\{X_0 = y\})}.$$

In pratica occorre spostarsi all'indietro dalla posizione di X_0 di $1/p(\{X_0 = y\})$ simboli in media per trovare il primo simbolo y nel passato.

Dimostrazione. Sia $X_0 = x$. Definiamo gli eventi per $j = 1, 2, \ldots$ e per $k = 0, 1, 2, \ldots$:

$$A_{j,k} = \{X_{-j} = x, X_l \neq x, -j < l < k, X_k = x\}.$$

L'evento $A_{j,k}$ corrisponde all'evento in cui l'ultima comparsa di x prima dello zero sia al tempo $-j$, mentre la prima comparsa di x successiva allo zero sia al tempo k. Tali eventi sono disgiunti e, poiché l'ergodicità del processo stocastico dà la certezza statistica di trovare l'elemento x sia nel passato sia nel futuro, $p(\bigcup_{j,k} A_{j,k}) = 1$. Quindi

$$1 = p(\bigcup_{j,k} A_{j,k}) = \sum_{j=1}^{\infty}\sum_{k=0}^{\infty} p(A_{j,k}) = \tag{2.11}$$

$$= \sum_{j=1}^{\infty}\sum_{k=0}^{\infty} p(X_k = x)p(X_{-j} = x, X_l \neq x, -j < l < k | X_k = x) = \tag{2.12}$$

$$= \sum_{j=1}^{\infty}\sum_{k=0}^{\infty} p(X_k = x)Q_x(j+k) = \tag{2.13}$$

$$= \sum_{j=1}^{\infty}\sum_{k=0}^{\infty} p(X_0 = x)Q_x(j+k) = \tag{2.14}$$

$$= p(X_0 = x)\sum_{j=1}^{\infty}\sum_{k=0}^{\infty} Q_x(j+k) = \tag{2.15}$$

$$= p(X_0 = x)\sum_{i=1}^{\infty} iQ_x(i), \tag{2.16}$$

dove (2.11) segue dal fatto che gli $A_{j,k}$ sono eventi disgiunti, (2.13) segue dalla definizione di $Q_x(i)$ e dalla stazionarietà del processo stocastico, (2.14)

segue dalla stazionarietà del processo stocastico e (2.16) segue dal fatto che esistono j coppie (j,k) tali che la loro somma $j+k=i$. □

Corollario 2. *Sia $\{X_k\}_{k=-\infty}^{\infty}$ un processo stocastico stazionario ergodico, denotiamo con X_i^j la sequenza $(X_i, X_{i+1}, \ldots, X_j)$ per $-\infty \leq i < j \leq \infty$. Sia $B_n(X_0^{n-1})$ il tempo in cui i simboli $X_0, X_1, \ldots, X_{n-1}$ sono comparsi nel passato, cioè*

$$B_n(X_0^{n-1}) = \max\{j < 0 \mid X_j^{j+n-1} = X_0^{n-1}\}. \tag{2.17}$$

Allora vale:

$$E\left[B_n(X_0^{n-1}) \mid X_0^{n-1} = x_0^{n-1}\right] = \frac{1}{p(x_0^{n-1})}.$$

Dimostrazione. Definiamo un nuovo processo stocastico $\{U_i\}_{i=-\infty}^{\infty}$ con $U_i = (X_i, X_{i+1}, \ldots, X_{i+n})$. Tale processo è stazionario ed ergodico, è quindi possibile applicare il precedente lemma di Kac. Si ottiene che il tempo di ricorrenza medio per $\{U_i\}_{i=-\infty}^{\infty}$ condizionato alla conoscenza si $U_0 = \mathbf{u}$ è $1/p(\mathbf{u})$. Tornando al processo $\{X_i\}_{i=-\infty}^{\infty}$ si prova il corollario. □

La disuguaglianza di Markov stabilisce che se X è una variabile aleatoria non negativa il cui valore d'aspettazione esiste, e $t > 0$, allora vale

$$P(\{X \geq t\}) \leq \frac{E[X]}{t}.$$

Applicando la disuguaglianza di Markov al corollario precedente si deduce:

$$\begin{aligned} p(\{B_n(X_0^{n-1}) > n | X_0^{n-1} = Y\}) \leq \\ \leq \frac{E\left[B_n(X_0^{n-1}) | X_0^{n-1} = Y)\right]}{n} = \frac{1}{np(\{X_0^{n-1} = Y\})}. \end{aligned} \tag{2.18}$$

Corollario 3. *Sia $\{X_k\}_{k=-\infty}^{\infty}$ un processo stocastico stazionario ergodico e sia $B_n(X_0^{n-1})$ il tempo in cui i simboli $X_0, X_1, \ldots, X_{n-1}$ sono comparsi nel passato. Per $n = 1, 2, \ldots$ ed $\epsilon > 0$ arbitrario:*

$$p(B_n(X_0^{n-1}) > 2^{n(H+\epsilon)}) \xrightarrow{n\to\infty} 0$$

dove H è l'entropia di $\{X_k\}_{k=-\infty}^{\infty}$.

Dimostrazione. Fissiamo $\epsilon > 0$, $\delta < \epsilon$, e sia $\mathcal{T}_\delta^{(n)}$ l'insieme tipico.

$$\begin{aligned} p(B_n(X_0^{n-1}) > 2^{n(H+\epsilon)}) = \\ = \sum_{\mathbf{y}\in\mathcal{T}_\delta^{(n)}} p(\{X_0^{n-1} = \mathbf{y}\}) p(B_n(X_0^{n-1} > 2^{n(H+\epsilon)} | X_0^{n-1} = \mathbf{y}\}) + \\ + \sum_{\mathbf{y}\notin\mathcal{T}_\delta^{(n)}} p(\{X_0^{n-1} = \mathbf{y}\}) p(B_n(X_0^{n-1} > 2^{n(H+\epsilon)} | X_0^{n-1} = \mathbf{y}\}) \leq \\ \leq \sum_{\mathbf{y}\in\mathcal{T}_\delta^{(n)}} p(\{X_0^{n-1} = \mathbf{y}\}) \frac{1}{2^{n(H+\epsilon)} p(X_0^{n-1} = \mathbf{y}\})} + p(\{X_0^{n-1} \notin \mathcal{T}_\delta^{(n)}\}) \quad (2.19) \end{aligned}$$

Tale disuguaglianza segue dalla (2.18) mettendo $2^{n(H+\epsilon)}$ al posto di n. Quindi, poiché $\epsilon - \delta > 0$

$$\begin{aligned} p(\{B_n(X_0^{n-1}) > 2^{n(H+\epsilon)}\}) \leq \\ \leq 2^{-n(H+\epsilon)} |\{X_0^{n-1} \in T_n(\delta)\}| + p(\{X_0^{n-1} \notin T_n(\delta)\}) \leq \\ \leq 2^{-n(\epsilon-\delta)} + \delta \quad (2.20) \end{aligned}$$

dove l'ultima disuguaglianza segue dalle proprietà dell'insieme tipico. Siccome $\epsilon - \delta > 0$ e δ è arbitrario, otteniamo l'asserto. □

Corollario 4. *Sia $k(n)$, per $n = 1, 2, \ldots$ tale che*

$$K := \lim_{n\to\infty} \inf \frac{1}{n} \log k(n) > H. \quad (2.21)$$

Allora

$$\lim_{n\to\infty} p(\{B_n(X_0^{n-1}) > k(n)\}) = 0.$$

Dimostrazione. Sia $\epsilon = K - H > 0$. Per $n > n_0$ abbastanza grande, $(1/n)\log k(n) \geq K - (\epsilon/2) = H + (\epsilon/2)$. Quindi $k(n) \geq 2^{n(H+(\epsilon/2))}$. Allora, per $n \geq n_0$

$$p(\{B_n(X_0^{n-1}) > k(n)\}) \leq p(\{B_n(X_0^{n-1}) > 2^{n(H+(\epsilon/2))}\}) \longrightarrow 0$$

per il teorema precedente. □

2.6.3 Dimostrazione dell'Ottimalità Asintotica di LZ77

Torniamo ora alla dimostrazione dell'ottimalità asintotica dell'algoritmo di Lempel e Ziv, dobbiamo mostrare che vale

$$\lim_{W\to\infty} \lim_{N\to\infty} \bar{\rho}(N) = H. \quad (2.22)$$

Poiché l'algoritmo definito è senza prefisso, vale

$$\mathbb{E}\left[l(X_1^N)\right] \geq H(X_1, X_2, \ldots, X_N) \geq NH.$$

Si ha quindi che

$$\bar{\rho}(N) = \mathbb{E}\left[\frac{l(X_1^N)}{N}\right] \geq H.$$

Mostriamo ora che vale anche

$$\lim_{W\to\infty} \lim_{N\to\infty} \bar{\rho}(N) \leq H. \tag{2.23}$$

Supponiamo di aver implementato l'algoritmo e di aver già codificato $X_1, X_2, \ldots, X_N$. Suddividiamo l'intervallo $[W+1, N]$ in N'/l_0 sottointervalli di lunghezza l_0, con

$$N' = N - W, \qquad l_0 = \frac{\log W}{H+\epsilon} \tag{2.24}$$

con $\epsilon > 0$ arbitrario. Assumiamo per semplicità che ϵ sia tale che l_0 sia un intero che divide N', in questo modo riusciamo suddividere la sequenza X_{W+1}^N in intervalli che comprendono tutti lo stesso numero di elementi semplificandone enormemente la notazione. È possibile applicare il ragionamento che segue ad una suddivisione della sequenza in sottointervalli di diverse lunghezza con un forte appesantimento della notazione. Denotiamo gli intervalli di lunghezza N'/l_0 ottenuti con

$$I_j = [W + (j-1)l_0 + 1; W + jl_0] \quad j = 1, 2, \ldots, \frac{N'}{l_0}.$$

Definiamo un tale intervallo I_j *bad set* se non esiste una copia di $(X_k)_{k\in I_j}$ che inizia nella stringa di $W - l_0$ simboli che precedono I_j, cioè

$$X_{W+(j-1)l_0+1}^{W+jl_0} \neq X_{W+(j-1)l_0+1-m}^{W+jl_0-m}, \qquad \forall 1 \leq m \leq W - l_0,$$

ricordando la definizione (2.17) del tempo in cui una sequenza di simboli è comparsa nel passato, si ha quindi che

$$\begin{aligned}\mathbb{P}(\{I_j \text{ bad set}\}) &= \\ &= \mathbb{P}(\left\{B_{l_0}(X_{W+(j-1)l_0+1}, X_{W+(j-1)l_0+1}, \ldots, X_{W+jl_0}) > W - l_0\right\}).\end{aligned} \tag{2.25}$$

Definiamo ora $k(l_0) := W - l_0 = 2^{l_0(H+\epsilon)} - l_0$ e notiamo che soddisfa la (2.21) con $K = H + \epsilon$, quindi vale

$$\mathbb{P}(\{I_j \text{ bad set}\}) \longrightarrow 0, \tag{2.26}$$

per W (e l_0) che tende ad infinito.

Sia ora $\{Y_i\}_{i=1,2,\dots,C}$ la successione delle frasi generata dall'algoritmo nella codifica di X_1^N. Per $i = 1, 2, \dots, C$ sia Y_i' la frase Y_i aumentata del primo simbolo che segue Y_i in X_1^N:

$$Y_i = X_k^{k+L_i-1} \Rightarrow Y_i' = X_k^{k+L_i}.$$

Diciamo che Y_i è una frase *interna* se Y_i' inizia e finisce all'interno dello stesso sottointervallo I_j.

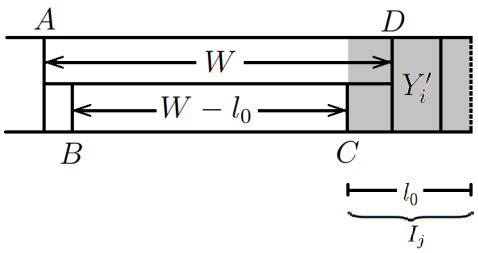

Figura 2.7: Esempio di finestra di lunghezza W e di un sottointervallo I_j contenente una frase interna Y_i.

Lo schema in Figura 2.7 mostra una finestra di lunghezza W (contenente i simboli compresi nel segmento AD) ed un sottointervallo I_j di lunghezza l_0 che ha intersezione non nulla con la finestra e che contiene una frase interna Y_i. Il segmento BC è costituito dai $W - l_0$ simboli che precedono I_j. Affinché I_j non sia un bad set è quindi necessario che esista una copia di $(X_k)_{k\in I_j}$ che inizi nel segmento BC.

Poiché per definizione Y_i è la sequenza più lunga per la quale esiste una copia che inizia nella finestra AD, non può esistere una copia di $Y_i' \in I_j$ che inizia nella finestra AD. Poiché $Y_i' \subseteq I_j$, non esiste una copia di $(X_k)_{k\in I_j}$ che inizia in AD e quindi poiché $BC \subseteq AD$ ne deduciamo che I_j è un bad set.

Abbiamo mostrato che se un sottointervallo I_j contiene una frase interna Y_i, I_j è un bad set. Inoltre se Y_i è interna la sua lunghezza non può essere maggiore della lunghezza del bad set che la contiene, quindi

$$L_i < l_0 = \frac{\log W}{H + \epsilon}.$$

Sia S_I l'insieme delle frasi di $\{Y_1, Y_2, \dots, Y_n\}$ che sono interne,

$$S_I := \{i \in 1, 2, \dots, C | Y_i \text{ è interna}\}.$$

Sia Ψ la frazione di bad set sul totale dei sottointervalli, si ha

$$\sum_{i\in S_I} |Y_i| \leq l_0 \frac{N'}{l_0} \Psi = N'\Psi, \tag{2.27}$$

dove $\frac{N'}{l_0}\Psi$ indica il numero totale dei bad set. In pratica questo sta ad indicare che la somma di tutte le lunghezze delle frasi interne è minore o uguale alla somma di tutte le lunghezze dei bad set in quanto, come abbiamo visto, i bad set contengono tutte le frasi interne.

Osserviamo inoltre che se Y_i non è una frase interna, si trova in parte in un sottointervallo I_j ed in parte nel successivo I_{j+1}. Allora Y_i deve occupare l'ultima posizione di uno ed uno solo degli intervalli I_j, quindi il numero delle frasi di $\{Y_1, Y_2, \ldots, Y_C\}$ che non sono interne ad uno degli intervalli I_j non può essere maggiore del numero totale $\frac{N'}{l_0}$ degli intervalli I_j. Indicando con S_I^c il complementare di S_I si ha

$$c' := |S_I^c| \leq \frac{N'}{l_0}. \tag{2.28}$$

Possiamo ora dimostrare la disuguaglianza (2.23). Questa dimostrazione si basa sulla distinzione tra frasi interne e non interne. Infatti le frasi interne, trovandosi all'interno dei bad set, hanno come questi ultimi probabilità che tende a 0 quando la dimensione della finestra tende ad ∞, mentre, come appena detto, il numero totale delle frasi che non sono interne non può essere maggiore del numero totale dei sottointervalli. Sarà quindi utile riscrivere la (2.10) nel seguente modo

$$\bar{\rho}(N) \leq \frac{\lceil W \log A \rceil}{N} + \frac{1}{N}\mathbb{E}\left[\sum_{i=1}^{C} \min(\log W + \gamma_1 \log(L_i + 1), \gamma_2 L_i)\right] \tag{2.29}$$

$$\leq \frac{\lceil W \log A \rceil}{N} + \frac{1}{N}\mathbb{E}\left[\sum_{i \in S_I} \gamma_2 L_i + \sum_{i \in S_I^c} [\log W + \gamma_1 \log(L_i + 1)]\right] \tag{2.30}$$

Il primo valore atteso della (2.30) lo otteniamo dalla (2.27)

$$\frac{1}{N}\mathbb{E}\left[\sum_{i \in S_I} \gamma_2 L_i\right] \leq \frac{\gamma_2 N'}{N}\mathbb{E}\left[\Psi\right] \leq \gamma_2 \mathbb{P}(\{I_j \text{ bad set}\}). \tag{2.31}$$

Il secondo termine della (2.29) di cui calcoleremo il valore atteso è dato

da

$$\frac{1}{N}\sum_{i\in S_I^c}\log W+\gamma_1\frac{1}{N}\sum_{i\in S_I^c}\log(L_i+1)= \tag{2.32}$$

$$=|S_I^c|\frac{\log W}{N}+\gamma_1\frac{c'}{N}\sum_{i\in S_I^c}\frac{1}{c'}\log(L_i+1)\leq \tag{2.33}$$

$$\leq|S_I^c|\frac{\log W}{N}+\gamma_1\frac{c'}{N}\log\left(\frac{1}{c'}\sum L_i+1\right)\leq \tag{2.34}$$

$$\leq\frac{N'}{N}\frac{\log W}{l_0}+\gamma_1\frac{c'}{N}\log\left(\frac{N}{c'}+1\right)\leq \tag{2.35}$$

$$\leq\frac{\log W}{l_0}+\frac{\gamma_1}{l_0}\log(l_0+1)= \tag{2.36}$$

$$=(H+\epsilon)+\frac{\gamma_1}{l_0}\log(l_0+1). \tag{2.37}$$

La (2.34) segue dalla disuguaglianza di Jensen e dalla concavità del logaritmo, la (2.35) dal fatto che $\sum_{i\in S_I^c}L_i\leq N$, la (2.36) dal fatto che $1/x\log(x+1)$, $x\geq 0$ decresce con x e dalla (2.28) che implica

$$\frac{c'}{N}\leq\frac{N'}{Nl_0}\leq\frac{1}{l_0},$$

mentre la (2.37) segue dalla (2.24).

Sostituendo la (2.37) e la (2.31) nella (2.29) si ha

$$\bar{\rho}(N)\leq\frac{\lceil W\log A\rceil}{N}+\gamma_2\mathbb{P}(\{I_j\text{ bad set}\})+H+\epsilon+\frac{\gamma_1}{l_0}\log(l_0+1) \tag{2.38}$$

e

$$\lim_{N\to\infty}\bar{\rho}(N)\leq\gamma_2\mathbb{P}(\{I_j\text{bad set}\})+H+\epsilon+\frac{\gamma_1}{l_0}\log(l_0+1). \tag{2.39}$$

Infine, per $W\to\infty$ e per la (2.26) si ottiene

$$\lim_{W\to\infty}\lim_{N\to\infty}\bar{\rho}(N)\leq H+\epsilon$$

che, per $\epsilon\to 0$ è la disuguaglianza (2.23).

2.7 Algoritmo di Lempel e Ziv LZ78

Si può notare che l'algoritmo LZ77 assume implicitamente che le sequenze ripetute siano sufficientemente vicine, in particolare all'interno della sliding

window. Con questa assunzione, nel caso in cui ci siano sequenze frequenti ma a distanza maggiore dell'ampiezza della finestra W, esse vengono codificate senza sfruttare le loro occorrenze ripetute. Si potrebbe pensare di risolvere il problema aumentando le dimensioni della sliding window, ma anche questa scelta porta alcuni problemi. L'aumento dell'ampiezza della finestra porta infatti all'aumento i bits necessari per la descrizione della posizione della stringa richiamata, ed al conseguente incremento del tasso di compressione.

Di questo problema si resero conto gli stessi Lempel e Ziv, che nel 1978 pubblicarono un articolo, *"Compression of Individual Sequences via Variable-Rate Coding"* [7], dove si illustra un nuovo algoritmo, poi denominato LZ78, in cui la precedente assunzione non compare. L'idea alla base dell'algoritmo di Ziv e Lempel del 1978 è proprio quella di individuare la presenza di prefissi anche a distanza maggiore della massima consentita dalla sliding window e per far questo viene costruito dinamicamente un dizionario contenente le sottosequenze incontrate. Bisogna fare particolare attenzione al fatto che entrambi il codificatore ed il decodificatore possano essere in grado di costruire dizionari fra loro identici.

Nel loro articolo, Lempel e Ziv descrivono un algoritmo che suddivide la stringa in sequenze, ognuna delle quali è la sequenza più corta che non è comparsa precedentemente. Questo algoritmo può essere interpretato come la costruzione di un dizionario ad albero, la cui dimensione cresce man mano che viene esaminato il testo, e nel quale i rami corrispondono alle sequenze che non sono comparse in precedenza. L'algoritmo è particolarmente semplice da implementare ed è noto per essere uno dei primi algoritmi standard per la compressione dei file su computer, è veloce ed efficiente.

L'algoritmo LZ78 effettua un parsing della stringa in sequenze finite, la differenza rispetto ad LZ77 sta nel fatto che ognuna delle sequenza non è mai comparsa in precedenza.

Definizione 33. *Sia $Y_1, Y_2, \ldots, Y_C$ un parsing di una stringa $(X_1, X_2, \ldots, X_M)$ di simboli di $\mathcal{A}$. Un parsing si dice distinto se non esistono due frasi fra loro identiche, cioè se $Y_i \neq Y_j$ per ogni $i \neq j$, con $i, j \in \{1, 2, \ldots, C\}$.*

Per esempio il parsing 0 | 111 | 1 è un parsing distinto della stringa 01111, ma 0 | 11 | 11 è un parsing ma non è distinto.

All'inizio della codifica il dizionario è vuoto. Per definizione, il dizionario vuoto contiene un'unica stringa codificata: la stringa vuota. Supponiamo di aver già codificato parte della stringa, e che quindi il dizionario contenga delle parole, ciascuna identificata tramite un indice. L'algoritmo, ad ogni passo, cerca, nella stringa da codificare, la sequenza più lunga che corrisponda ad una parola già presente nel dizionario. Inizia prendendo in input il primo simbolo x_k della stringa non ancora codificata e confrontandolo con le parole

nel dizionario. Se la ricerca di una corrispondenza va a buon fine, l'algoritmo prende in input il secondo simbolo ed effettua la ricerca nel dizionario della parola $x_k x_{k+1}$. Ripete questo procedimento considerando parole sempre più lunghe finché non trova una parola che non è contenuta nel dizionario. A questo punto, l'algoritmo codifica la coppia (i, c) costituita dall'indice nel dizionario relativo all'ultima corrispondenza trovata e dal primo carattere c che segue la parola nella stringa da codificare. Infine, aggiunge nel primo indice libero del dizionario la parola per la quale non ha trovato corrispondenze e ripete da capo il procedimento per il passo successivo. Si noti che il caso in cui il dizionario non contiene la stringa con il solo carattere $x_k = c$ viene considerata la corrispondenza con la stringa nulla. Ciò accade sempre nel primo passo di codifica, in questo caso LZ78 codifica la coppia $(0, c)$ ed aggiunge "c" al dizionario.

Esempio 2.5. Consideriamo la stringa

$$ABBABBABBBAABABAA\ldots,$$

essa viene suddivisa nel modo seguente:

$$A \mid B \mid BA \mid BB \mid AB \mid BBA \mid ABA \mid BAA \mid \ldots.$$

Poiché ogni frase è la sottosequenza di minore lunghezza mai comparsa nella stringa precedentemente, la sottosequenza costituita da tutti i simboli di ogni frase tranne l'ultimo deve essere comparso in precedenza. Possiamo allora codificare ogni sequenza indicando l'indirizzo del prefisso e il valore dell'ultimo simbolo. La stringa sopra si rappresenta nel modo seguente:

$$(0, A), (0, B), (2, A), (2, B), (1, B), (4, A), (5, A), (3, A)\ldots.$$

Analogamente all'algoritmo LZ77, l'algoritmo LZ78 è un codice universale asintoticamente ottimale. Si rimanda per la dimostrazione alla referenza [1].

Tuttavia, la necessità di inviare un carattere "c" non compresso alla fine di ogni sequenza, dovuta alla scelta di un dizionario iniziale vuoto, comporta una notevole perdita di efficienza. È possibile aggirare questo problema inizializzando il dizionario con tutti i simboli dell'alfabeto della sorgente e considerando il carattere "c" come parte della sequenza successiva. Questa variazione, dovuta a Welch, sta alla base di molte implementazioni pratiche di LZ78.

2.8 Algoritmo di Lempel, Ziv e Welch LZW

L'algoritmo LZW è il risultato delle modifiche apportate agli algoritmi di Lempel e Ziv già visti nel 1984 da Terry A. Welch, ed è stato da lui presentato

nell'articolo [8]. Welch propose una tecnica per rimuovere la necessità di inviare il secondo elemento nella coppia (i, c) della codifica LZ78. In pratica il codificatore invia soltanto l'indice i relativo alla posizione della sottosequenza ripetuta all'interno del dizionario. Affinché ciò sia possibile, il dizionario deve essere inizializzato con tutti gli elementi dell'alfabeto della sorgente.

Esiste un insieme di regole precise per la codifica del dizionario, che permetterà in seguito al sistema di decompressione di generare un dizionario esattamente uguale a quello di partenza, in modo tale da poter ricostruire l'esatto contenuto del file originale.

Il funzionamento dell'algoritmo risulta più chiaro attraverso degli esempi, ne mostriamo uno pubblicato nella referenza [9].

Esempio 2.6. Assumiamo che, come accade in buona parte dei casi reali, l'alfabeto $\mathcal{A}$ della sorgente sia dato dai 256 caratteri del codice ASCII. Utilizziamo la stringa campione "*/WED/WE/WEE/WEB/WET*" per capire il funzionamento dell'algoritmo di compressione. Tale stringa è una piccola lista di parole in inglese separate dal carattere "/".

L'algoritmo LZW, come LZ78, cerca nella stringa da codificare la parola più lunga che ha una corrispondenza nel dizionario ed aggiunge al primo indice libero del dizionario la parola per cui non è stata trovata alcuna corrispondenza. Tuttavia, a differenza di LZ78, codifica solamente l'indice del dizionario relativo alla corrispondenza più lunga trovata, mentre l'ultimo carattere non viene codificato e continua a far parte della stringa da codificare come primo carattere della parola al passo successivo.

Al primo passo il codificatore considera la stringa di due caratteri "*/W*" ed effettua il confronto con le stringhe già presenti nel dizionario. Poiché questa non compare tra le parole del dizionario, l'algoritmo dà in output il simbolo "/", ed aggiunge la parola '*/W*' associandola al codice 256. Infatti, il dizionario è inizializzato con i 256 caratteri del codice ASCII, numerati da 0 a 255. In seguito il codificatore considera il terzo carattere della stringa, "*E*", dà in output "*W*" ed aggiunge la seconda stringa di codice, "*WE*", al dizionario associandola al codice 257. L'algoritmo continua in questo modo finché non legge la sequenza "*/W*', che corrisponde alla parola 256 del dizionario. A questo punto dà in output codice 256 ed aggiunge al dizionario una stringa di tre caratteri. Il processo continua in questo modo fino all'ultimo carattere, dopo il quale si codifica un simbolo EOF "End Of File" che indica la fine della stringa. Il procedimento è mostrato nella Tabella 2.1.

Il dizionario cresce rapidamente. In questo esempio si utilizza volutamente una stringa altamente ridondante, ottendo cinque sostituzioni di codice in output. Negli esempi su testi reali, la compressione non ha luogo se non

Stringa: /WED/WE/WEE/WEB/WET

Sequenza Input	Codice Output	Nuovo codice Dizionario	Nuova stringa
/W	/	256	/W
E	W	257	WE
D	E	258	ED
/	D	259	D/
WE	256	260	/WE
/	E	261	E/
WEE	260	262	/WEE
/W	261	263	E/W
EB	257	264	WEB
/	B	265	B/
WET	260	266	/WET
EOF	T		

Tabella 2.1: Esempio di compressione LZW.

dopo aver costruito un dizionario piuttosto grande, di solitamente dopo aver letto almeno un centinaio di caratteri.

Analizziamo ora l'algoritmo di decompressione. Occorre disporre del codice ottenuto comprimendo la stringa con l'algoritmo precedente, ed utilizzarlo per ricostruire la stringa originale. Il decompressore, utilizzando come input il codice compresso, è in grado di ricostruire al suo interno un dizionario che corrisponde esattamente al dizionario creato dal compressore.

L'algoritmo di decodifica, data la stringa codificata, prende in input gli indici codificati e ricostruisce la stringa con le parole del dizionario ad essi associate. Ogni volta che ottiene una nuova parola, alla stessa maniera dell'algoritmo di codifica, la aggiunge al dizionario associandola ad un nuovo codice. Dalla Tabella 2.2 notiamo che il decodificatore in questo modo riesce a creare un dizionario che è esattamente lo stesso costruito durante la codifica.

Grazie alla sua velocità rispetto a LZ78, questo metodo è largamente utilizzato, ad esempio nell'utility *UNIX compress/uncompress*, nei formati grafici *TIFF* e *GIF* e nella compressione standard *V.42 bis* per la trasmissione dati tramite modem analogici.

Stringa: / W E D 256 E 260 261 257 B 260 T

Input	Dizionario Compressore	Output	Carattere	Nuovo Dizionario
/	/	/		
W	/	W	W	256 = /W
E	W	E	E	257 = WE
D	E	D	D	258 = ED
256	D	/W	/	259 = D/
E	256	E	E	260 = /WE
260	E	/WE	/	261 = E/
261	260	E/	E	262 = /WEE
257	261	WE	W	263 = E/W
B	257	B	B	264 = WEB
260	B	/WE	/	265 = B/
T	260	T	T	266 = /WET

Tabella 2.2: Esempio di decompressione LZW.

2.9 DEFLATE

DEFLATE, letteralmente "sgonfiare", è un algoritmo di compressione dati lossless che utilizza una variante del metodo di compressione LZ77 combinata con la codifica di Huffman. Fu originariamente progettato da Philip Katz come parte del software PKZIP. DEFLATE è un metodo di compressione di dominio pubblico ed è quindi implementato in un gran numero di software come nella compressione dei files tramite gzip, il formato immagine PNG e il formato file ZIP per il quale venne appositamente studiato. Al fine di comprenderne il funzionamento, è necessario ricordare le metodologie di compressione dei due algoritmi che lo costituiscono.

Come abbiamo già visto, il codice di Huffman è un codice senza prefisso. Ogni parola di codice rappresenta un elemento in uno specifico alfabeto. Ad ognuno degli elementi dell'alfabeto è assegnata una probabilità, un numero che rappresenta la frequenza relativa dell'elemento all'interno della stringa da comprimere. Tali probabilità possono essere ipotizzate senza conoscere il contenuto dei dati da comprimere o calcolate esattamente attraverso un'analisi dei dati, o addirittura attraverso una combinazione di entrambe queste modalità.

A partire da queste probabilità DEFLATE utilizza l'algoritmo di Huffman per costruire un albero di decodifica, ed assegna ad ognuno degli elementi una sequenza binaria che rappresenta univocamente gli elementi.

Ad esempio, nella compressione di un generico testo, con alta probabilità

molti degli elementi del codice ASCII saranno esclusi dall'albero di Huffman, mentre i caratteri frequentemente utilizzati (come "A", "E", "T"...) saranno associati ai codici di minore lunghezza.

Nella classica codifica di Huffman, un dato insieme di elementi con associate probabilità può generare molteplici alberi. Nella variazione utilizzata dal metodo standard DEFLATE, esistono due regole aggiuntive che inducono l'unicità dell'albero:

- gli elementi con i codici più corti devono essere posizionati alla sinistra (ramo 0) degli elementi con codici più lunghi;
- nel caso in cui due elementi abbiano la medesima lunghezza di codice, l'elemento che nell'alfabeto compare per primo deve essere posizionato alla sinistra (ramo 0) dell'elemento che compare per secondo.

Con queste due restrizioni, dato un insieme di elementi e le rispettive lunghezze dei codici ad essi associati, si ottiene un'unica possibilità di costruzione dell'albero di Huffman. Al fine di permettere la ricostruzione dell'albero di Huffman sarà quindi necessario trasmettere al decodificatore solamente le lunghezze dei codici identificativi di ogni elemento.

DEFLATE effettua la codifica di Huffman sulla stringa già codificata con l'algoritmo LZ77. Quest'ultimo ricerca sequenze di dati ripetute e, quando una sequenza di caratteri è identica a una sequenza che si trova all'interno della sliding window, la sostituisce da una coppia di interi, *(distanza, lunghezza)*, che rappresentano il numero dei caratteri da scorrere all'indietro per raggiungere il primo carattere della sequenza da copiare e la lunghezza della sequenza da copiare. DEFLATE utilizza una sliding window di 32Kbytes, ciò significa che la ricerca di ogni sequenza di caratteri viene effettuata negli ultimi $32 * 1024 = 32768$ caratteri comparsi nella stringa da codificare.

Il compressore DEFLATE è caratterizzato da una grande flessibilità nella modalità di compressione dati. Deflate ha infatti a disposizione tre diversi metodi per comprimere una sequenza:

1. *Non effettuare la compressione:* questa può essere una scelta intelligente quando, ad esempio, i dati sono già stati compressi. I dati memorizzati non compressi comporteranno una relativa espansione del codice. Tale espansione sarà comunque inferiore rispetto all'espansione che si avrebbe memorizzando dati già compressi in precedenza attraverso uno dei metodi che seguono.

2. *Compressione LZ77 + Codifica di Huffman statica:* Gli alberi utilizzati per questa tipologia di compressione sono definiti all'interno dello

stesso algoritmo DEFLATE, non è quindi necessario inviare l'albero di Huffman con la stringa compressa.

3. *Compressione LZ77 + Codifica di Huffman adattiva:* codifica utilizzando alberi creati dal compressore a partire dalla stringa originale. Gli alberi devono essere memorizzati insieme alla stringa compressa.

Una stringa compressa con DEFLATE è costituita da una successione di blocchi. Ogni blocco è preceduto da un codice di tre bits detto "header" che ne descrive le caratteristiche di compressione:

- bit 1: indicatore dell'ultimo blocco della stringa:
  - 1: il blocco è l'ultimo blocco che compone la stringa;
  - 0: il blocco è seguito da ulteriori blocchi;
- bit 2-3: indicatore del metodo di codifica utilizzato per comprimere il blocco:
  - 00: blocco non compresso;
  - 01: Il blocco usa la codifica di Huffman statica con un albero prestabilito;
  - 10: Il blocco usa la codifica di Huffman dinamica con albero adattato;
  - 11: riservato, comando non utilizzato.

La maggior parte dei blocchi saranno compressi con il metodo 10, codifica di Huffman dinamica, che produce un albero di Huffman adattato (ottimizzato) per ognuno dei singoli blocchi. Le istruzioni necessarie alla ricostruzione dell'albero di Huffman seguono immediatamente l'header. Gli alberi di Huffman usati per un blocco sono indipendenti da quelli utilizzati per blocchi precedenti o successivi, mentre in LZ77 le sequenze ripetute possono riferirsi a sequenze di blocchi precedenti, cioè un puntatore di distanza può attraversare uno o più blocchi singoli, senza puntare ad una posizione che vada oltre l'inizio del flusso di dati in input o oltre la dimensione della finestra. Infine, se un blocco presenta un alto tasso di entropia, l'algoritmo può decidere di non comprimerlo affatto.

2.10 Trasformata di Burrows e Wheeler BWT

La trasformata di Burrows-Wheeler (che abbrevieremo con BWT) è un algoritmo utilizzato in innumerevoli applicazioni per la compressione dati.

La trasformata, che costituisce gran parte dell'algoritmo, fu sviluppata da Wheeler nel 1983 e viene attualmente utilizzata come primo stadio di trasformazione della stringa da comprimere in molti programmi di compressione commerciali, primo tra tutti *bzip2*. Ad ogni modo, l'algoritmo di compressione BWT completo venne presentato da Burrows e Wheeler nel 1994 [11].

A differenza degli algoritmi visti finora, con la trasformata BWT è necessario avere a disposizione l'intera sequenza codificata prima di iniziare la fase di decodifica. Il metodo di compressione presentato da Burrows e Wheeler in [11] è una combinazione dei due seguenti algoritmi:

- trasformata di Burrows-Wheeler: se applicata ad una stringa di caratteri non effettua una vera e propria compressione, ma permuta in maniera reversibile l'ordine dei caratteri. Se la stringa originale contiene molte sottostringhe che si ripetono spesso, la stringa trasformata ha numerose parti in cui un carattere viene ripetuto diverse volte di fila.

- metodo Move-To-Front: è un semplice metodo di compressione utile a comprimere stringhe caratterizzate da caratteri ripetuti.

2.10.1 Trasformata di Burrows-Wheeler

Data una sequenza $x_1x_2\ldots x_N$ di lunghezza N, ne creiamo $N-1$ permutazioni, ciascuna delle quali è uno shift ciclico della sequenza originale che chiameremo rotazione. In pratica, otteniamo la i-esima rotazione concatenando la sottostringa degli ultimi i caratteri con la sottostringa dei primi $N-i$, ottenendo le seguenti permutazioni della stringa di partenza

$$x_{N-i+1}x_N\ldots x_Nx_1x_2\ldots x_{N-i} \qquad \forall i = 1, 2, \ldots, N-1.$$

Riordiniamo le N sequenze in ordine lessicografico. Il codificatore invia la sequenza di lunghezza N che si compone prendendo l'ultima lettera di ogni rotazione. La ricostruzione della stringa può essere assicurata in due diversi modi. Nel primo modo insieme alla sequenza permutata come appena descritto, il codificatore invia la posizione occupata dalla sequenza originale nella lista ordinata di rotazioni. In alternativa il codificatore aggiunge un carattere speciale alla fine della stringa prima di effettuare la trasformazione di Burrows-Wheeler.

La sequenza così trasformata è strutturata in modo che applicando ad essa l'algoritmo MTF si ottiene una compressione particolarmente efficiente. Prima di proseguire con la desrizione del metodo MTF, presentiamo un esempio di trasformata di Burrows-Wheeler per renderne più facile la comprensione.

Esempio 2.7. Utilizziamo la stringa campione "BANANA" per capire il funzionamento della trasformata di Burrows-Wheeler. Aggiungiamo ad essa un carattere finale assumendo che quest'ultimo non compaia all'interno del testo, al fine di distinguere l'ultimo carattere quando si ricostruisce la stringa. Otteniamo una nuova stringa $BANANA@$, che costituirà la stringa di input. Costruiamo ora una tabella con tutte le possibili rotazioni della stringa, ed infine riordiniamo in ordine alfabetico le sue righe, come mostrato in Tabella 2.3.

B	A	N	A	N	A	@		A	N	A	N	A	@	**B**
A	N	A	N	A	@	B		A	N	A	@	B	A	**N**
N	A	N	A	@	B	A		A	@	B	A	N	A	**N**
A	N	A	@	B	A	N	$\Longrightarrow$	B	A	N	A	N	A	**@**
N	A	@	B	A	N	A		N	A	N	A	@	B	**A**
A	@	B	A	N	A	N		N	A	@	B	A	N	**A**
@	B	A	N	A	N	A		@	B	A	N	A	N	**A**

Tabella 2.3: Esempio di trasformata di Burrows-Wheeler: Passo (1): elenco di tutte le rotazioni possibili sulla stringa "BANANA@"; Passo (2): riordinamento delle righe in ordine alfabetico.

La stringa in output sarà l'ultima colonna della tabella riordinata alfabeticamente cioè "BNN@AAA", stringa che contiene l'ultimo carattere di ognuna delle rotazioni ordinate alfabeticamente.

Descriviamo ora l'algoritmo inverso. Prendiamo una tabella vuota con righe e colonne pari al numero dei caratteri della stringa permutata. Conoscendo soltanto l'informazione della stringa permutata possiamo ricostruire facilmente la stringa originale. L'ultima colonna ci dice infatti quali sono i caratteri del file originale. Basterà soltanto riordinarli alfabeticamente per ottenere quella che nella precedente Tabella 2.3 riordinata costituiva la prima colonna.

A questo punto affiancando l'ultima colonna alla prima otteniamo le coppie di caratteri successivi del file originale. Ordiniamo ora alfabeticamente le coppie così ottenute, in questo modo otteniamo la prima e la seconda colonna della Tabella 2.3. Affianchiamo a queste di nuovo l'ultima colonna ottenendo le triple di caratteri successivi del file originale. Continuando in questo modo possiamo ricostruire l'intera stringa. A questo punto, la colonna contenente il carattere che indica la fine del testo come carattere iniziale rappresenta la stringa ordinata.

B
N
N
@
A
A
A

⇒

B	A
N	A
N	A
@	B
A	N
A	N
A	@

⇒

B	A	N
N	A	N
N	A	@
@	B	A
A	N	A
A	N	A
A	@	B

⇒

B	A	N	A
N	A	N	A
N	A	@	B
@	B	A	N
A	N	A	N
A	N	A	@
A	@	B	A

⇒

B	A	N	A	N
N	A	N	A	@
N	A	@	B	A
@	B	A	N	A
A	N	A	N	A
A	N	A	@	B
A	@	B	A	N

⇒

B	A	N	A	N	A
N	A	N	A	@	B
N	A	@	B	A	N
@	B	A	N	A	N
A	N	A	N	A	@
A	N	A	@	B	A
A	@	B	A	N	A

⇒

B	A	N	A	N	A	@
N	A	N	A	@	B	A
N	A	@	B	A	N	A
@	**B**	**A**	**N**	**A**	**N**	**A**
A	N	A	N	A	@	B
A	N	A	@	B	A	N
A	@	B	A	N	A	N

Tabella 2.4: Ricostruzione della stringa originale dalla stringa "BNN@AAA" trasformata attraverso il metodo BWT. Ad ogni passo si crea una nuova tabella riordinando le righe della tabella corrente ed aggiungendo ad esse la stringa "BNN@AAA" come prima colonna.

2.10.2 Move-To-Front

Move-To-Front o MTF è un algoritmo che non comprime i dati ma che aiuta a ridurre la ridondanza, specie dopo l'applicazione della trasformazione BWT, dove un simbolo che è comparso da poco compare di nuovo con alta probabilità. Il metodo MTF funziona nel seguente modo:

- si inizializza un vettore di lunghezza pari alla cardinalità dell'alfabeto contenente i simboli utilizzati. Questo vettore rappresenta in ogni momento una permutazione di tutti gli elementi dell'alfabeto.

- si leggono in serie i caratteri dell'input. Per ognuno viene dato in output la sua corrente posizione all'interno del vettore costruiti, in seguito si modifica il vettore: il simbolo corrente viene rimosso dalla posizione corrente e reinserito in testa alla struttura.

- si codifica con il metodo di Huffman, con la codifica aritmetica oppure con il metodo RLE la stringa ottenuta in output.

Esempio 2.8. Codifichiamo la stringa ottenuta tramite la trasformata di Burrows-Wheeler "BNN@AAA", a valori in un alfabeto $\mathcal{A}$ costituito dai 26

Iterazione	Lista Alfabeto	Codice Sequenza
BNN@AAA	ABCDEFGHIJKLMNOPQRSTUWXYZ@	1
B**N**N@AAA	BACDEFGHIJKLMNOPQRSTUWXYZ@	1, 13
BN**N**@AAA	NBACDEFGHIJKLMOPQRSTUWXYZ@	1, 13, 0
BNN**@**AAA	NBACDEFGHIJKLMOPQRSTUWXYZ@	1, 13, 0, 26
BNN@**A**AA	@NBACDEFGHIJKLMOPQRSTUWXYZ	1, 13, 0, 26, 4
BNN@A**A**A	A@NBCDEFGHIJKLMOPQRSTUWXYZ	1, 13, 0, 26, 4, 0,
BNN@AA**A**	A@NBCDEFGHIJKLMOPQRSTUWXYZ	1, 13, 0, 26, 4, 0, 0

Tabella 2.5: Esempio di procedimento del metodo move-to-front.

caratteri dell'alfabeto standard più il carattere di fine sequenza "@" $\mathcal{A} = \{A, B, \ldots, Z, @\}$ di cardinalità 27.

Il vettore iniziale sarà il seguente:

$$A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, W, X, Y, Z, @.$$

La prima lettera della sequenza da codificare è B, che compare all'indice 1 del vettore numerato da 0 a 26. Scriviamo 1 nella stringa di output. Il carattere B si sposta all'inizio della lista, e si produce il nuovo vettore

$$B, A, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, W, X, Y, Z, @.$$

Il carattere successivo è N, che ora compare all'indice 13. Aggiungiamo 13 alla stringa di output e spostiamo N in cima alla lista ottenendo

$$N, B, A, C, D, E, F, G, H, I, J, K, L, M, O, P, Q, R, S, T, U, W, X, Y, Z, @.$$

Continuando in questo modo, otteniamo che il codice della sequenza sarà $1, 13, 0, 26, 4, 0, 0$, e a quest'ultimo si va infine ad applicare il metodo di Huffman, la codifica aritmetica oppure il metodo RLE. Il procedimento completo è mostrato nella Tabella 2.5.

Il codificatore MTF rappresenta con numeri poco elevati caratteri che si sono presentati recentemente; considerato quello che ci aspettiamo come output dalla trasformazione di Burrows-Wheeler, MTF avrà ottime probabilità di produrre una grande quantità di valori vicini allo zero, dando ottimo materiale al compressore che segue. È facile notare che il metodo MTF è reversibile. Basta mantenere la stessa lista e decodificare sostituendo ogni indice nel flusso codificato con la lettera che corrisponde all'indice nella lista.

L'algoritmo BWT ottiene una compressione paragonabile a quella che si ottiene con i migliori modelli statistici, mentre la sua velocità computazionale è paragonabile a quella degli algoritmi che si basano sulle tecniche di Lempel e

Ziv. Al fine di ottenere una buona compressione è necessario che le dimensioni del file in input siano abbastanza grandi, almeno qualche migliaio di bytes [11].

In pratica l'algoritmo BWT ordina alfabeticamente tutte le possibili rotazioni di una stringa di input S e genera una nuova stringa che consiste nell'ultimo carattere di ciascuna rotazione. Per capire il motivo per cui la stringa così ottenutà è di più facile compressione, consideriamo l'effetto dato da una singola lettera appartenente ad una parola particolarmente comune in un blocco di testo in inglese. Utilizziamo l'esempio della lettera t nella parola *the*, e assumiamo che una stringa di input contenga più volte la parola *the*. Quando la lista delle rotazioni della stringa in input viene ordinata, tutte le rotazioni che iniziano con *he* si troveranno in righe vicine, e una buona parte di queste righe finirà con la lettera t. Una regione della nuova stringa conterrà quindi una forte concentrazione di caratteri uguali a t, mescolati con altri caratteri che usualmente precedono la stringa *he* in inglese come s, T, e S. Lo stesso discorso può essere applicato agli altri caratteri in tutte le parole, si avrà quindi che ogni regione localizzata della stringa permutata è tale da contenere un alto numero di pochi caratteri distinti. L'effetto generale è che la probabilità che un dato carattere compaia in un dato punto della stringa perturbata è molto alta se tale carattere compare in un punto vicino al punto dato, e bassa se questo non accade. Tale proprietà è esattamente la proprietà necessaria per ottenere una buona compressione con il metodo MTF, move-to-front.

2.11 Codifica *ppm* - Prediction by Partial Matching

La codifica predittiva si basa sull'idea che, avendo abbastanza informazioni a priori sulla sorgente, si può costruire un modello di predizione lineare in cui il campione corrente possa essere predetto accuratamente attraverso un'analisi sui campioni precedenti. Essa viene ampiamente usata per la compressione di file audio, di immagini singole (ad esempio trasmissione di fax) oppure per ridurre la ridondanza temporale del segnale video, dove è presente una forte correlazione tra i campioni di immagini successive.

L'algoritmo *ppm* è stato presentato da J.G. Cleary e I.H. Witten [10], nel 1984. Non è diventato popolare come i precedenti algoritmi di Lempel e Ziv principalmente a causa della sua esecuzione molto più lenta. In seguito, con lo sviluppo di varianti più efficienti, la popolarità degli algoritmi basati sul metodo *ppm* è aumentata e sta tuttora aumentando.

L'idea degli algoritmi *ppm* è molto semplice. Utilizzare un elevato numero di elementi della sequenza passata per stimare più accuratamente il successivo simbolo in input richiederebbe ogni volta la stima e la memorizzazione di un numero di probabilità condizionate estremamente grande, che potrebbe addirittura essere infattibile. Invece di stimare queste probabilità ad ogni simbolo, possiamo stimare le probabilità man mano che procediamo con la codifica. In questo modo è necessario memorizzare solamente gli ultimi elementi della sequenza incontrati. Chiameremo *context* la sottosequenza costituita da questi elementi, cioè dagli elementi che precedono nella sequenza il simbolo x_i da codificare, utilizzati dall'algoritmo *ppm* per effettuare la predizione di x_i. L'algoritmo *ppm* usa modelli a context finito. Piuttosto che limitarsi a context di lunghezza fissa, esso ne modifica le dimensioni a seconda delle proprietà osservate nel testo codificato precedentemente, da qui il termine "partial matching". La scelta di un context finito riduce la quantità di dati da memorizzare, ma, soprattutto all'inizio del processo di codifica, capiterà di dover codificare lettere che non compaiono all'interno del context. Per questo motivo, è necessario che l'alfabeto della sorgente contenga un simbolo speciale *escape* che indichi quando il carattere che si sta codificando non compare all'interno del suo context.

L'algoritmo controlla se nella stringa precedentemente codificata ci sono corrispondenze del context seguito dal simbolo, inizialmente si considera un context di dimensione maggiore possibile, prestabilita. Se il simbolo da codificare non compare nel passato successivamente allo stesso context, l'algoritmo codifica il simbolo escape ed rieffettua la ricerca diminuendo la dimensione del context. Questo processo continua finché non si trova la corrispondenza di un context seguito dal simbolo da codificare, o finché, nel caso peggiore, non si arriva alla conclusione che il simbolo non è comparso in nessun context. In questo caso, utilizziamo la probabilità $1/M$ per codificare il simbolo, dove M indica la dimensione del dizionario.

Per esempio, se dovessimo codificare la *a* di *probabilmente*, dovremo controllare se la stringa *proba* è apparsa nel testo in precedenza, ossia se *a* nel passato è comparsa con context *prob*. Se non è apparsa, codifichiamo un simbolo escape e controlliamo se *a* è comparsa nel passato con il context *rob*. Se la stringa *roba* non è mai comparsa, codifichiamo un altro simbolo escape e riproviamo la ricerca con il context *ob*. Continuando questo procedimento, tentiamo la ricerca con l'ultimo context *b* e, se quest'ultima fallisce, controlliamo se la lettera *a*, con un context di ordine zero, è comparsa in precedenza. Se *a* compare per la prima volta nella stringa, effettuiamo la codifica utilizzando un modello in cui tutte le lettere dell'alfabeto hanno la stessa possibilità di comparire. Ci si riferisce a questo modello equiprobabile come context di ordine -1.

Il modello utilizzato è adattivo, e ad ogni nuovo simbolo si ricalcolano le relative probabilità. Per consentire la codifica, deve essere attribuita una determinata probabilità anche al simbolo escape. E' difficile risolvere questo problema, in quanto si tratta di stimare la probabilità che si verifichi un evento mai osservato prima. Questo problema, detto anche della zero-frequenza (*zero frequency problem*), viene affrontato solitamente effettuando una stima empirica dei risultati sperimentali. Cleary e Witten nel loro primo documento [10] sul metodo *ppm* propongono due metodi.

Per quanto riguarda la lunghezza del context considerato, potrebbe sembrare che context di dimensioni maggiori determinino una compressione migliore, ma ciò non è necessariamente vero. Infatti, aumentando la dimensione massima del context si aumentano le probabilità di codificare caratteri con numero di occorrenze $c(\varphi) > 0$, ma allo stesso tempo si aumenta la probabilità di dover codificare lunghe sequenze di simboli escape. In pratica, il tasso di compressione di questo metodo, diminuisce al crescere della lunghezza del context, raggiunge un minimo e poi inizia ad aumentare. Il valore della lunghezza del context a cui tale minimo corrisponde dipende dalle caratteristiche della sorgente.

Alcune varianti del metodo *ppm* sono implementate in noti formati di compressione dati come *7-Zip* e *RAR*.

2.12 RLE, Run-Length Encoding

L'algoritmo Run-Length Encoding è un algoritmo di compressione dati estremamente semplice utilizzato per molti formati di immagini come *TIFF*, *BMP*, *PCX*, basato sulla ripetizione di elementi consecutivi. RLE comprime ogni tipologia di dati a prescindere dal contenuto, ma tale contenuto influisce molto sul tasso di compressione del metodo. Nonostante la maggior parte degli algoritmi RLE non possano avvicinarsi al tasso di compressione di metodi più avanzati, RLE è un algoritmo estremamente facile da implementare ed è molto veloce.

In questo tipo di compressione, ogni serie ripetuta di caratteri o, in inglese, *run*, viene codificata utilizzando una coppia di dati: il primo, utilizzato come contatore, serve per memorizzare la lunghezza della stringa, il secondo, invece, contiene l'elemento che si ripete nella stringa.

Esempio 2.9. La stringa "AAAAAAABBBBBBBBBBB" viene codificata tramite $(7, A), (11, B)$. Invece per la stringa "ABBCCBDBEF", nella quale la ripetizione dei caratteri è ridotta, codifica RLE è data da

$$(1, A), (2, B), (2, C), (1, B), (1, D), (1, B)(1, E), (1, F),$$

che risulta essere molto costosa.

Questo algoritmo funziona bene in presenza di immagini con pochi colori molto uniformi, ovvero in serie di dati che abbiano molte ripetizioni al loro interno. Essendo estremamente semplice, non comporta difficoltà implementative né eccessiva complessità di esecuzione, di contro non è adatto ad applicazioni generiche, in quanto il tipo di ridondanza non è comune.

È possibile definire per la compressione RLE delle regole particolari che permettono di comprimere quando necessario e di lasciare la stringa originale quando la compressione produce uno spreco.

Esistono numerose varianti di RLE le cui principali differenze consistono nella lunghezza minima da attribuire ad un run. Nell'esempio descritto abbiamo usato una lunghezza del run pari a uno, cioè prendiamo in considerazione un carattere alla volta e contiamo le sue ripetizioni. Allo stesso modo è possibile considerare blocchi di simboli e cercare ripetizioni dell'intero blocco all'interno della stringa.

Capitolo 3

Compressori bidimensionali

Attualmente, gran parte delle informazioni vengono scambiate sotto forma di immagini digitali. Infatti, a partire dall'introduzione della trasmissione via fax, anche documenti prevalentemente testuali sono stati digitalizzati, cioè rappresentati numericamente in una matrice di pixel. Con l'informatizzazione di svariati settori, dalla fotografia fino ai sistemi medici diagnostici, si è enormemente accentuata la necessità di gestire immagini digitali di considerevoli dimensioni e, di conseguenza, la ricerca di tecniche di compressione per tali tipologie di dati è diventata fondamentale.

L'aspetto di un'immagine digitalizzata dipende dalla sua risoluzione, cioè dal numero di pixel per unità, e dal numero di bits utilizzati per descrivere un pixel. Migliore è la sua qualità, maggiore sarà il numero di bits necessari per rappresentarla. Per ridurre tale quantità, le tecniche di compressione cercano di comprendere la struttura dell'immagine per eliminare le ridondanza ed ottenerne una efficiente rappresentazione.

Gli algoritmi di compressione si dividono in due categorie:

- tecniche di compressione lossless: permettono di comprimere un'immagine in maniera reversibile, senza alcuna perdita di informazioni. L'immagine originale può essere completamente recuperata.

- tecniche di compressione lossy: ottengono una compressione dell'immagine molto più compatta, ma generalmente portano ad una perdita di informazioni non reversibile.

Gli algoritmi presentati in seguito appartengono alla categoria lossless, un settore estremamente attivo. Sono infatti numerose le pubblicazioni di nuove proposte di schemi di questo tipo in molte riviste scientifiche come Journal of Electronic Imaging, Optical Engineering, IEEE Transactions on Image Processing, IEEE Transactions on Communications, Communications

of the ACM, IEEE Transactions on Computers, Image Communication, ed altre ancora.

3.1 Codifica standard CCITT fax a due livelli di colore

Il CCITT "Comité Consultatif International Téléphonique et Télégraphique", dal 1992 noto come ITU-T, è il settore della Unione Internazionale delle Telecomunicazioni che si occupa di regolare le telecomunicazioni telefoniche e telegrafiche. Il suo compito è quello di fornire delle specifiche standard riconosciute a livello internazionale. Alla fine degli anni '70 il CCITT riconobbe il potenziale delle tecnologie via fax che stavano emergendo e cominciò a sviluppare una serie di protocolli di comunicazione per la trasmissione di immagini in bianco e nero. Nel 1980 finalizzarono il loro lavoro con il Group 3 standard che rappresentò un progresso a livello mondiale. Una pagina che con i precedenti protocolli necessitava di almeno cinque minuti per essere trasmessa via fax, con il Group 3 standard veniva trasmessa in un solo minuto. Nel 1984 il CCITT pubblicò il Group 4 standard, che era essenzialmente simile al Group 3, ma era stato pensato per l'utilizzo con tecnologie digitali e, di conseguenza, non prevedeva la gestione di errori di trasmissione o di perdite di dati. L'assenza di tecniche di rilevamento e correzione degli errori rese Group 4 in grado di raggiungere migliori tassi di compressione rispetto a Group 3, e velocizzò ulteriormente la trasmissione delle informazioni.

L'algoritmo standard del CCITT codifica inizialmente le caratteristiche principali dell'immagine, come altezza e larghezza della matrice di pixel che la rappresenta. Nel Group 4 vengono specificati anche le proprietà della scala di colori utilizzata, mentre ciò non è necessario per il Group 3 che tratta soltanto immagini a due livelli di colore. Il Group 3 ha a disposizione due diverse modalità di codifica: uno schema monodimensionale, che tratta ogni linea in maniera indipendente dalle altre, ed uno schema bidimensionale, dove la codifica di una linea scansionata dipende fortemente dalla linea immediatamente precedente. In pratica, nello schema a due dimensioni, ogni linea viene rappresentata identificando le posizioni relative dei cosiddetti "changing elements" rispetto alla precedente, cioè dei pixel di passaggio dal colore bianco al colore nero o viceversa. Per far sì che questo procedimento inizi, la prima linea deve essere codificata con lo schema ad una dimensione.

Ogni linea scansionata viene rappresentata come una sequenza di run di pixel di diversa lunghezza alternativamente bianchi e neri. Le parole di codice corrispondenti alle lunghezze dei singoli run sono diverse a seconda del

Codici Terminating		
Lunghezza run	**Runs bianchi**	**Runs neri**
0	00110101	0000110111
1	000111	010
2	0111	11
3	1000	10
4	1011	011
5	1100	0011
⋮	⋮	⋮
62	00110011	000001100110
63	00110100	000001100111

Codici Makeup		
Lunghezza run	**Runs bianchi**	**Runs neri**
64	11011	000001111
128	10010	000011001000
⋮	⋮	⋮
1728	010011011	0000001100101
EOL	000000000001	000000000001

Tabella 3.1: Tabella delle parole di codice di Group 3 standard. I codici sono diversi a seconda del colore del run in quanto le lunghezze hanno frequenze diverse a seconda del colore. I codici di lunghezze maggiori di 63 si ottengono concatenando codici makeup e codici terminating.

colore del run a cui si riferiscono. Si utilizza un codice binario di Huffman non adattivo per codificare le lunghezze dei run di pixel consecutivi ed il relativo colore. La lunghezza di tali parole di codice fu ottimizzata dal CCITT, che costruì una tabella di codici di Huffman basandosi sulla statistica delle lunghezze di runs bianchi e neri ottenute esaminando un campione vario di documenti, per lo più scritti a mano o stampati a macchina. Questa è la ragione per cui le immagini con composizione radicalmente differente non vengono compresse in maniera efficace. Nei documenti scritti a mano o stampati a macchina le sequenze brevi compaiono più frequentemente di quelle lunghe, per questo motivo le parole di codice corrispondenti ai run con lunghezza compresa tra i 2 e i 4 pixel sono le più corte. La dimensione massima di un run è limitata dal numero massimo di pixel di una linea scansionata con Group 3.

Per mantenere la tabella dei codici di dimensioni ragionevoli, le lunghezze si esprimono tramite combinazioni di due diverse tipologie di parole di codice: parole di codice dette *makeup* e parole di codice dette *terminating*. Le pa-

role di codice del tipo *terminating* rappresentano le lunghezze da 0 a 63, mentre le lunghezze comprese fra 64 e 2623 (lunghezza massima stabilita per una linea scansionata) sono codificate tramite concatenazioni di parole di codice *makeup* con una parola di codice *terminating* finale. Entrambe le tipologie differiscono a seconda che si riferiscano a runs di pixel bianchi o neri, la Tabella 3.1 può aiutare a comprendere meglio la struttura del codice descritto. L'introduzione di queste due diverse tipologie di parole di codice è avvenuta a causa della necessità di codificare immagini di risoluzione sempre maggiore.

Nel codice Group 3 monodimensionale sono inoltre definite molte parole di codice "speciali" che permettono di approssimare i dati perduti nel caso di errori di trasmissione della stringa codificata. Ogni linea codificata termina con un codice EOL (End Of Line), una parola di codice lunga 12 bit. Tale codice è utilizzato per individuare l'inizio e la fine di ogni linea scansionata durante la trasmissione dell'immagine. Se il segnale della trasmissione viene temporaneamente alterato, il decodificatore Group 3 non considera i dati che riceve fino alla ricezione del codice EOL successivo, in seguito ricomincia la normale decodifica assumendo che i dati che seguono il codice EOL appartengono alla linea successiva. Il decodificatore può inoltre sostituire la linea corrotta con un predefinito insieme di dati, come ad esempio una linea bianca. Le trasmissioni di messaggi codificati con Group 3 terminano con un codice RTC (Return To Control) che indica la fine della trasmissione ed RTC è costituito da sei codici EOL consecutivi. Infine il FILL, che non è esattamente una parola di codice ma è una sequenza di zeri, può essere utilizzato per gonfiare la parola di codice RTC affinché la stringa codificata possa essere un multiplo di 8 e possa quindi essere trasmessa in bytes, segue la linea scansionata e precede il codice EOL.

Il codice Group 3 bidimensionale viene spesso denominato READ modificato, o più semplicemente MR, perché deriva da un codice più complicato chiamato READ. Nella modalità bidimensionale, il primo pixel di ogni run prende il nome di changing element. Ogni changing element segna la transizione di colore in una linea scansionata (il punto dove finisce un run di colore e ne inizia un altro). La posizione di ciascun changing element in una linea è descritta in relazione al numero di pixel di differenza dai changing elements della linea precedente.

Nei tre schemi in Figura 3.1 sono mostrate due linee: la linea superiore prende il nome di reference line, è la linea di riferimento ed è già stata codificata, la linea inferiore prende invece il nome di coding line ed è la linea corrente da codificare. L'algoritmo codifica la coding line prendendo in considerazione un gruppo di cinque changing elements: a_0, a_1, a_2 nella coding line e b_1 e b_2 nella reference line. Attraverso le relative posizioni decide fra

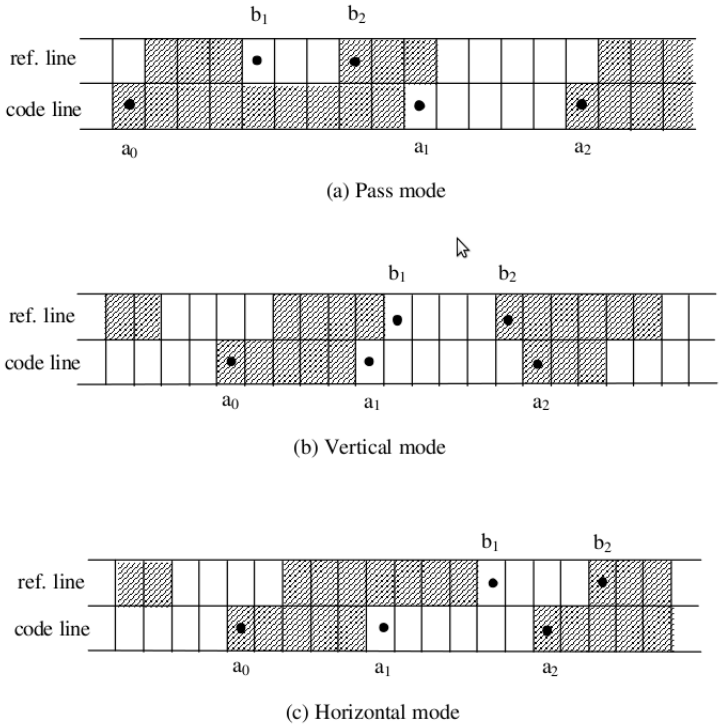

Figura 3.1: Group 3 bidimensionale standard: utilizzo di tre diverse modalità di codifica a seconda della disposizione dei changing elements.

tre metodi di codifica. I cinque changing elements condiderati sono:

- a_0: è il changing pixel di riferimento, indica la posizione di inizio del run corrente da codificare. All'inizio di una linea di codice, a_0 è un changing pixel bianco immaginario che precede il changing element del run da codificare;

- a_1: è il primo changing element che segue a_0 nella coding line. Poiché la lettura dei pixel avviene da sinistra a destra e dall'alto al basso, a_1 si trova a destra di a_0, ed a_1 è di colore diverso da a_0;

- a_2: è il primo changing element che segue a_1 nella coding line. Si trova alla destra di a_1 ed è dello stesso colore di a_0;

- b_1: è il changing element nella reference line che si trova più vicino alla posizione di a_0 da destra, e che ha lo stesso colore di a_1;

- b_2: è il changing element successivo a b_1 nella reference line.

Group 3 standard bidimensionale sceglie il metodo di compressione di ogni run a seconda delle posizioni relative dei cinque changing elements descritti:

- **Pass Mode**: viene scelto se il changing element b_2 si trova alla sinistra di a_1. Significa che il run della reference line che inizia con b_1 non è adiacente al run della coding line che inizia con a_1. Ricordiamo che b_1 è stato scelto in modo che abbia lo stesso colore di a_1. In questo caso l'algoritmo codifica una parola speciale "0001", in questo modo il decodificatore riconosce che il run che inizia con a_0 nella coding line non termina nel pixel che nella coding line è sottostante a b_2. A questo punto il pixel sottostante b_2 viene identificato come changing element a_0 del nuovo gruppo di changing elements per la codifica dei pixel successivi.

- **Vertical Coding Mode**: se la distanza $|a_1 - b_1|$ è minore o uguale a 3 pixel, il metodo di codifica è quello verticale. La posizione di a_1 viene espressa in riferimento alla posizione di b_1. Sono assegnate sette diverse parole di codice per sette casi differenti: caso in cui la distanza tra a_1 e b_1 sia uguale a 0, ± 1, ± 2, ± 3, dove $+$ significa che a_1 si trova alla destra di b_1 mentre $-$ significa che a_1 si trova alla sinistra di b_1.

- **Horizontal Coding Mode**: se la distanza relativa tra a_1 e b_1 è maggiore di tre pixel il metodo di codifica è quello orizzontale, ossia viene applicato il metodo RLE Run-Length Encoding monodimensionale. In particolare il codificatore inserisce una parola di codice costituita da tre parti: un codice "001" che indica il metodo di codifica, una parola di codice per descrivere il run da a_0 ad a_1 ed infine una parola di codice per descrivere il run da a_1 ad a_2, entrambe secondo la codifica RLE. A questo punto a_2 diventa il pixel di riferimento a_0 per il nuovo insieme di changing elements per la codifica della sequenza successiva.

Poiché una linea scansionata e codificata con Group 3 bidimensionale dipende della precedente linea, se un errore di trasmissione rovina un segmento di dati relativi ad una linea scansionata codificata, tutte le linee scansionate successivamente vengono decodificate impropriamente. Al fine di minimizzare tali errori di propagazione Group 3 bidimensionale utilizza una variabile chiamata "fattore K" che specifica il numero massimo di righe consecutive da codificare tramite codifica 2D. In questo modo $K - 1$ linee vengono codificate in due dimesioni la K-esima linea con il metodo Group 3 monodimensionale e, nel caso di un errore di trasmissione nei dati, vanno perse al massimo solamente K linee.

Il codice Group 4 bidimensionale fu sviluppato a partire dall'algoritmo Group 3 bidimensionale ottenendo un migliore schema di compressione bidimensionale. Group 4 fu progettato per codificare dati supportati in disk

Metodo	Condizioni	Codice Output	Nuova Posizione a_0
Pass	$b_2 - a_1 < 0$	0001	Sotto b_2 nella Coding Line
Vertical	$a_1 - b_1 = 0$	1	a_1
	$a_1 - b_1 = 1$	011	
	$a_1 - b_1 = 2$	000011	
	$a_1 - b_1 = 3$	0000011	
	$a_1 - b_1 = -1$	010	
	$a_1 - b_1 = -2$	000010	
	$a_1 - b_1 = -3$	0000010	
Horizontal	$\lvert a_1 - b_1 \rvert > 3$	001∗RLE$(l(a_0, \ldots, a_2) - 1)$	a_2

Tabella 3.2: Schema di codifica Group 3 bidimensionale.

drives e networks, quindi la struttura per la ricerca e la correzione di errori non è più necessaria. Group 4 standard effettua la codifica utilizzando il solo algoritmo bidimensionale. La prima linea di riferimento nel codice è una linea immaginaria di pixel bianchi. Nel codice Group 4 la parola di codice RTC è sostituita da un codice EOFB (End Of Facsimile Block) che consiste in due consecutivi EOL della stessa tipologia di quelli Group 3, mentre l'utilizzo del codice EOL per indicare la fine di ogni linea non è più necessario. In questo modo l'algoritmo viene enormemente velocizzato ottenendo migliori tassi di compressione che, a seconda delle proprietà dell'immagine, possono essere anche dimezzati rispetto a Group 3 standard.

3.2 Il vecchio formato JPEG Standard

La sigla JPEG identifica una commissione di esperti denominata "Joint Photographic Expert Group" formata nel 1986 con lo scopo di stabilire uno standard di compressione per le immagini a tono continuo – cioè di tipo fotografico – sia a colori sia in bianco e nero. Il lavoro di questa commissione ha portato alla definizione di una complessa serie di algoritmi. Il più famoso tra questi è il JPEG standard con perdita di dati, ma, parallelamente a questo, la commissione ne sviluppò un altro senza perdita. Ad oggi, quest'ultimo è obsoleto ed è stato sostituito dal più efficiente JPEG-LS standard. In ogni caso, la descrizione del vecchio JPEG standard è utile per comprendere gli algoritmi predittivi per la codifica delle immagini.

I colori che i pixel di un'immagine digitale possono assumere sono disposti in una tavolozza, cioè in una lista, ordinati secondo le tonalità a partire dal nero fino al bianco. Ciascuno di questi è associato ad un indice, cioè ad

un ordinale che identifica il colore all'interno della tavolozza. In seguito, rappresenteremo con $I(i,j)$ il valore numerico relativo all'indice del colore del pixel di posto (i,j) nell'immagine originale, e con $\hat{I}(i,j)$ la predizione del valore numerico dell'indice del colore del pixel di posto (i,j).

Il vecchio schema di compressione JPEG standard prevede otto differenti schemi predittivi tra i quali l'utente stesso può scegliere. Il primo di questi non effettua nessuna predizione e lo chiameremo metodo 0, mentre gli altri sette sono mostrati in seguito. Tre di questi sono predittori ad una dimensione, gli altri quattro sono predittori bidimensionali. I sette metodi sono i seguenti:

1. $\hat{I}(i,j) = I(i-1,j)$
2. $\hat{I}(i,j) = I(i,j-1)$
3. $\hat{I}(i,j) = I(i-1,j-1)$
4. $\hat{I}(i,j) = I(i,j-1) + I(i-1,j) - I(i-1,j-1)$
5. $\hat{I}(i,j) = I(i,j-1) + (I(i-1,j) - I(i-1,j-1))/2$
6. $\hat{I}(i,j) = I(i-1,j) + (I(i,j-1) - I(i-1,j-1))/2$
7. $\hat{I}(i,j) = (I(i,j-1) - I(i-1,j-1))/2$

Immagini diverse fra loro hanno diverse strutture che possono essere predette in maniera più precisa con uno qualsiasi di questi otto metodi. L'algoritmo offre la possibilità di provare ognuno dei metodi e di scegliere quello che offre il migliore tasso di compressione.

3.3 CALIC

CALIC (A Context Based Adaptive Lossless Image Codec) [12] è un algoritmo di compressione per immagini a tono continuo senza perdita di dati, ed è caratterizzato da un buon tasso di compressione, bassa complessità computazionale e tempi di computazione e memoria necessaria piuttosto bassi.

CALIC codifica e decodifica le immagini seguendo l'ordine *raster scan*, ossia pixel per pixel da sinistra a destra, riga per riga dall'alto verso il basso.

L'algoritmo presentato nell'articolo [12] è stato adattato ad immagini digitali in cui i colori dei pixel sono descritti con 8 bit, per un totale di 256 colori. Tuttavia con un opportuno ridimensionamento di alcune costanti fissate dagli autori, risulta particolarmente conveniente anche per tavolozze

di maggiori dimensioni. I colori sono ordinati in una tavolozza secondo le tonalità a partire dal nero fino al bianco. Ad ognuno dei pixel dell'immagine è associato un valore, da 0 a 255, che indica la posizione che il colore del pixel occupa all'interno della tavolozza.

Lo schema opera in due modalità: modalità continua o binaria. La modalità binaria viene scelta qualora nell'intorno del pixel corrente l'immagine sia costituita da pixel con due soli valori di intensità, è stata quindi progettata per la tradizionale classe di immagini binarie costituite da pixel bianchi o neri. Quando CALIC entra in modalità binaria utilizza un codificatore aritmetico ternario per codificare tre simboli, includendo un simbolo escape per indicarne l'uscita.

CALIC seleziona automaticamente una delle due modalità durante il processo di codifica analizzando l'intorno del pixel corrente. Generalmente, in un'immagine, un dato pixel ha valore vicino ad uno dei pixel ad esso adiacenti. Quale, tra questi, abbia valore più vicino al pixel da codificare dipende dalla struttura locale dell'immagine. A seconda della presenza o meno di discontinuità locali verticali o orizzontali, il pixel superiore, oppure il sinistro, oppure una combinazione dei pixel dell'intorno possono essere le migliori approssimazioni del pixel corrente. Infine, nelle regioni dell'immagine dove il colore dei pixel varia velocemente, la predizione potrebbe risultare poco accurata.

L'algoritmo, con lo scopo di tenere conto di tutti questi fattori, deve determinare le caratteristiche locali dell'immagine utilizzando soltanto i pixel nell'intorno del pixel corrente che sono stati già codificati. Per comprendere in che modo CALIC riconosce la presenza o meno di discontinuità faremo riferimento alla Figura 3.2. Nella figura è rappresentato il context del pixel corrente X, cioè l'insieme dei pixel che si trovano nell'intorno di X ed i cui valori vengono utilizzati per effettuarne la predizione. Il pixel superiore ad X è denominato north pixel e rappresentato con N, il pixel a sinistra è denominato west pixel e rappresentato con W e così via. Notiamo che i pixel che fanno parte del context sono stati tutti codificati al momento della codifica di X, e sono quindi noti sia al codificatore che al decodificatore. Possiamo avere un'idea delle caratteristiche del context calcolando

$$\begin{aligned} d_h =& |W - WW| + |N - NW| + |NE - N|, \\ d_v =& |W - NW| + |N - NN| + |NE - NNE|. \end{aligned} \tag{3.1}$$

I valori d_h e d_v rappresentano i gradienti orizzontale e verticale e vengono utilizzati per ottenere una predizione iniziale per X. Se il valore di d_h è molto alto rispetto al valore di d_v, significa che nel context di X avviene una variazione orizzontale piuttosto notevole, ed N risulta essere la migliore stima per X. Se, al contrario, d_v è molto più grande di d_h, significa che

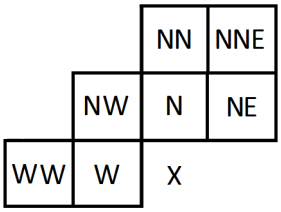

Figura 3.2: Context del pixel corrente X.

la discontinuità nel context è verticale e la stima migliore per X è W. Se invece la differenza fra d_h e d_v è più moderata, il predittore migliore si ottiene tramite una media pesata dei pixel del context.

L'esatto algoritmo utilizzato da CALIC per effettuare la predizione iniziale è il seguente:

```
if d_h − d_v > 80
    X̂ = N
if else d_v − d_h > 80
    X̂ = W;
else
{
    X̂ = (N + W)/2 + (NE − NW)/4;
     if d_h − d_v > 32
        X̂ = (X̂ + N)/2;
     else if d_v − d_h > 32
        Î = (X̂ + W)/2;
     else if d_h − d_v > 8
        X̂ = (3X̂ + N)/4;
     else if d_v − d_h > 8
        Î = (3X̂ + W)/4;
}
```

Per migliorare questa predizione servono altre informazioni riguardo alla relazione del pixel x con i pixel del suo intorno. Si calcola il vettore

$$\mathbf{I} = [N, W, NW, NE, NN, WW, 2N - NN, 2W - WW], \tag{3.2}$$

ed ogni sua componente viene confrontata con la predizione iniziale $\hat{X}$. Se il valore di una componente è minore del valore di predizione, sostituiamo il valore della componente con 1, in caso contrario la sostituiamo con uno 0. Si ottiene un vettore binario di otto elementi che potrebbe assumere 256 diverse possibili configurazioni ma, a causa delle dipendenze fra le varie componenti, le configurazioni possibili che il vettore (3.2) può assumere sono 144. CALIC

inoltre calcola una quantità che cattura le informazioni sulle discontinuità orizzontali e verticali del context e sugli errori di predizione precedenti con

$$\delta = d_h + d_v + 2|N - \hat{N}|, \tag{3.3}$$

dove N è il north pixel, il pixel che si trova sopra X, mentre $\hat{N}$ è la predizione iniziale calcolata per il pixel N. Il range dei valori di δ viene suddiviso in quattro intervalli, ciascuno rappresentato tramite due bit, indicati con δ_1. Questi quattro intervalli, insieme alle 144 diverse possibili configurazioni del vettore (3.2), permettono di raggruppare tutti i possibili intorni di X in $144 \times 4 = 576$ diverse possibilità per la coppia $(\mathbf{I}, \delta_1)$.

Il processo appena descritto elimina parecchie informazioni relative alla sequenza originale, mentre altre informazioni sono contenute nella sequenza degli errori di predizione. Siamo in grado di sfruttare queste ultime codificando il residuo in relazione agli errori commessi nella predizione dei pixel già codificati. CALIC prende il valore di $(\mathbf{I}, \delta_1)$ come descrizione numerica del context di X. Per ridurre ulteriormente la codifica, invece di utilizzare $(\mathbf{I}, \delta_1)$ stesso, CALIC effettua una quantizzazione, cioè assegna dei valori interi Q da 0 a 576 ai possibili valori che $(\mathbf{I}, \delta_1)$ può assumere, suddivide il rande di Q in otto intervalli prestabiliti, quindi $0 = q_0 < q_1 < q_2 < \ldots < q_8$, e se $q_{i-1} \leq Q < q_i$ considera Q appartenente all'i-esimo intervallo di quantizzazione. Man mano che procede con la codifica, CALIC tiene in memoria tutti gli errori di predizione passati. Supponiamo che per X valga $q_{k-1} \leq Q < q_k$, cioè che il context di X appartenga al k-esimo intervallo di quantizzazione. L'algoritmo calcola $\bar{\epsilon}_k$, media di tutti gli errori di predizione dei pixel con context appartenente al k-esimo intervallo di quantizzazione, e lo somma ad $\hat{X}$ per perfezionarne la predizione, ottenendo la predizione finale $\tilde{X}$.

Infine, dopo aver ottenuto la predizione finale, si codifica l'errore di predizione $X - \tilde{X}$. Se il pixel da codificare ha un valore compreso tra 0 ed $M-1$, l'errore di predizione avrà valore compreso tra $-(M-1)$ ed $M-1$. Anche se la maggior parte degli errori di predizione dovrebbe avere un valore vicino allo 0, utilizzando la codifica aritmetica dovremmo assegnare delle probabilità ad ogni valore possibile. Ciò significherebbe una riduzione delle dimensioni degli intervalli assegnati ai valori che realmente si devono codificare, provocando un notevole aumento dei bits necessari per la loro rappresentazione. Descriviamo con un esempio in che modo CALIC fa fronte a questo problema.

Consideriamo la sequenza $(0, 7, 4, 3, 5, 2, 1, 7)$. Poiché tutti i valori sono compresi tra 0 e 7, saranno sufficienti 3 bits per rappresentarli. Supponiamo di effettuare la predizione di ogni elemento utilizzando l'elemento che lo precede. La sequenza degli errori di predizione $r_n = x_n - x_{n-1}$ sarà data da (0,7,-3,1,-1,2, -3,-1,6). Conoscendo tali valori possiamo facilmente ricostruire

la sequenza originale calcolando $x_n = x_{n-1} + r_n$. Notiamo che gli errori di predizione appartengono all'intervallo $[-7, 7]$. In pratica, l'alfabeto richiesto per descrivere tali valori ha cardinalità quasi doppia rispetto all'alfabeto originale, ma, se osserviamo più attentamente, possiamo notare che il valore di r_n, per come è stato calcolato, deve appartenere all'intervallo $[-x_{n-1}, 7 - x_n - 1]$. In altre parole, dato un particolare valore di x_{n-1}, il numero di differenti valori che r_n può assumere è lo stesso dei differenti valori che può assumere x_n.

Tornando al caso generale, se il pixel X ha valore compreso tra 0 ed $M-1$, noto il valore di predizione $\tilde{X}$, l'errore di predizione $X - \tilde{X}$ può assumere solo i valori nell'intervallo $[-\tilde{X}, M-1-\tilde{X}]$, intervallo che possiamo mappare tramite un'applicazione iniettiva nell'intervallo $[0, M-1]$:

$$\varphi(\tilde{X} - X) = \begin{cases} 2(\tilde{X} - X) & \text{se } \tilde{X} - X \geq 0 \\ 2|(\tilde{X} - X)| - 1 & \text{se } \tilde{X} - X < 0. \end{cases} \tag{3.4}$$

Infine, CALIC codifica $\varphi(\tilde{X} - X)$. Qualsiasi codificatore entropico, di Huffman o aritmetico, statico o adattivo, binario o m-ario può essere facilmente interfacciato con il sistema CALIC.

L'insieme di tutte queste operazioni sembra essere complesso, ma in realtà il costo computazionale complessivo dell'algoritmo non è alto. Tuttavia è possibile ottenere un buon tasso di compressione anche semplificando alcuni dei più complessi passaggi di CALIC. Un esempio di semplificazione dello schema CALIC è l'algoritmo JPEG-LS.

3.4 JPEG-LS Standard

Il JPEG-LS standard assomiglia più al CALIC che al vecchio formato JPEG standard. Nel confronto con altre nuove proposte di metodi di compressione, CALIC raggiunse la prima posizione in sei delle sette categorie di immagini testate. Motivato da questi ottimi risultati e da altri aspetti di CALIC, un team della Hewlett-Packard decise di ristudiarlo e propose un codice predittivo molto più semplice [15] [16], denominato LOCO-I (LOw Complexity LOssless COmpression for Image), con perfomance di compressione vicine a quelle di CALIC [17] ed attualmente implementato nel formato JPEG-LS.

In esso la predizione iniziale si ottiene attraverso l'algoritmo seguente:

`if` $NW \geq \max(W, N)$
    $\hat{X} = \max(W, N)$;
`else`

{
    if $NW \leq \min(W, N)$
        $\hat{X} = \min(W, N)$;
else
        $\hat{X} = W + N - NW$;
}

Si effettua la predizione iniziale utilizzando solamente i pixel N, W, NW mostrati in Figura 3.2. La predizione iniziale, come per CALIC, viene in seguito migliorata utilizzando gli errori di predizione effettuati nel pixel già codificati.

Anche in JPEG-LS i context riflettono le variazioni locali dei valori dei pixel, tuttavia le loro descrizioni numeriche sono calcolate in un modo differente. Per iniziare si calcolano le differenze D_1, D_2 e D_3 come segue

$$\begin{aligned} D_1 &= NE - N, \\ D_2 &= N - NW, \\ D_3 &= NW - W. \end{aligned}$$

Questi tre valori vengono utilizzati per la quantizzazione del context in un vettore di tre componenti $\mathbf{Q} = (Q_1, Q_2, Q_3)$ attraverso la mappa seguente

$$\begin{cases} Q_i = -4 & \text{se} \quad D_i \leq -T_3 \\ Q_i = -3 & \text{se} \quad -T_3 < D_i \leq -T_2 \\ Q_i = -2 & \text{se} \quad -T_2 < D_i \leq -T_1 \\ Q_i = -1 & \text{se} \quad -T_1 < D_i < 0 \\ Q_i = 0 & \text{se} \quad D_i = 0 \\ Q_i = 1 & \text{se} \quad 0 < D_i \leq T_1 \\ Q_i = 2 & \text{se} \quad T_1 < D_i \leq T_2 \\ Q_i = 3 & \text{se} \quad T_2 < D_i \leq T_3 \\ Q_i = 4 & \text{se} \quad D_i \geq T_3, \end{cases} \tag{3.5}$$

dove T_1, T_2 e T_3 sono coefficienti prestabiliti positivi. Poiché Ciascuni dei Q_i può assumere nove possibili valori, si ottengono $9^3 = 729$ quantizzazioni di context possibili. È possibile effettuare una ulteriore riduzione del numero dei context rappresentando ogni quantizzazione $\mathbf{Q}$ con $-\mathbf{Q}$ ogni qualvolta che la prima componente del vettore non nulla è negativa. Quando ciò accade, un variabile SIGN viene impostata con il valore -1, altrimenti con $+1$. Ciò riduce il numero dei context a 365, e $\mathbf{Q}$ viene mappato in un numero compreso tra 0 e 364 (nello standard non si specifica l'applicazione utilizzata in questo passaggio).

La variabile SIGN viene utilizzata per raffinare la predizione. LOCO-I calcola $\bar{\epsilon}$, media degli errori di predizione dei pixel con context appartenenti

allo stesso intervallo di quantizzazione del pixel corrente (già codificati), la moltiplica per la variabile SIGN e poi la somma alla predizione iniziale.

L'errore di predizione finale ottenuto r_n viene mappato in un intervallo di valori della stessa dimensione dei valori che può assumere il pixel da codificare x_n. La mappatura utilizzata da JPEG-LS è la seguente

$$\begin{cases} \varphi(r_n) = r_n + M & \text{se} \quad r_n < -\frac{M}{2} \\ \varphi(r_n) = r_n - M & \text{se} \quad r_n > \frac{M}{2} \end{cases} \tag{3.6}$$

È stato dimostrato in [14] che la statistica globale dei residui di un predittore fissato per immagini a tono continuo è ben approssimata dalla Two Sided Geometric Distribution (TSGD) centrata nell'origine. Ciò significa che la probabilità che l'errore di predizione assuma un valore intero ϵ è proporzionale a $\theta^{|\epsilon|}$ dove $0 < \theta < 1$ controlla il tasso di decadimento esponenziale ed è la probabilità di ottenere ϵ unitario in valore assoluto ($\theta = p(1) = p(-1)$).

Il raffinamento della predizione di X effettuato attraverso la variabile SIGN serve a fare in modo che la TSGD si mantenga centrata nell'origine. Attraverso l'applicazione (3.6) l'algoritmo trasforma la TSGD in una distribuzione geometrica semplice, eliminando il problema della codifica di interi negativi.

In JPEG-LS, gli errori di predizione $\varphi(r_n)$ sono codificati utilizzando codici basati sulla codifica di Golomb [18], che sono metodi ottimali per la codifica di interi non negativi con distribuzione di probabilità geometrica [19].

3.4.1 Codifica di Golomb-Rice per interi positivi

I codici di Golomb-Rice [18] appartengono ad una famiglia di codici costruiti per codificare numeri interi con l'assunzione che più un numero è grande, minore è la probabilità che compaia. Il più semplice dei codici appartenenti a questa famiglia è il codice unario, che, dato un intero n, si ottiene nel modo seguente

$$\underbrace{11\ldots1}_{n \text{ volte}}0. \tag{3.7}$$

Questo codice ha la stessa lunghezza di un codice binario di Huffman per codificare valori $k \in \{1, 2, 3, \ldots\}$, con $p(k) = 2^{-k}$. Poiché il codice di Huffman è ottimale, anche il codice unario è ottimale per lo stesso insieme di probabilità. Anche se il codice unario è ottimale solo in codizioni molto ristrette, si nota che è estremamente semplice da implementare.

I codici di Golomb sono una famiglia di codici G_m parametrizzati da un parametro libero $m \in \mathbb{Z}^+$, utilizzato per dividere l'intero n in input in due parti: $q = \lfloor \frac{n}{m} \rfloor$, con $\lfloor x \rfloor$ che indica la parte intera di x, e $r = n - mq$, resto.

Il quoziente q può assumere ogni valore $1, 2, \ldots$ ed è rappresentato tramite codifica unaria. Il resto r può assumere valori $1, 2, \ldots, m-1$. Se m è una potenza di 2, utilizziamo $\log m$ bits per rappresentare r. Se al contrario m non è una potenza di 2 utilizziamo $\lceil \log m \rceil$ bits, dove $\lceil x \rceil$ è il più grande intero minore di x. Possiamo ridurre il numero di bits richiesti utilizzando $\lfloor \log m \rfloor$ bits per la rappresentazione binaria di r dei primi $2^{\lceil \log_2 m \rceil} - m$ valori, e $\lceil \log m \rceil$ bit per la rappresentazione binaria degli $r + 2^{\lceil \log_2 m \rceil} - m$ restanti valori.

3.5 FELICS

FELICS (Fast, Efficient, Lossless Image-Compression System) [20] è un algoritmo più semplice rispetto a CALIC e LOCO-I, di maggiore velocità ma con efficienza leggermente minore. In particolare, a prescindere dal livello di ridondanza dell'immagine, non riesce ad utilizzare meno di un bit per pixel.

I pixel vengono letti in ordine raster, ossia pixel per pixel da sinistra a destra, riga per riga dall'alto verso il basso, e ciascun nuovo pixel X viene codificato utilizzando il valore dei suoi due pixel più vicini che sono stati già codificati. Eccetto per i pixel che si trovano nella prima riga o nella prima colonna, facendo riferimento alla Figura 3.2 i pixel che si utilizzano sono il pixel direttamente sopra N ed il precedente a sinistra W. Rinominiamo il pixel con valore minore L e quello con valore maggiore H, e definiamo Δ come differenza $H - L$. Tratteremo Δ come context di predizione di X, utilizzato per la codifica del pixel.

L'idea di FELICS è quella di utilizzare un bit per indicare se il pixel X ha valore compreso tra il valore di L ed il valore di H, se il valore è esterno un ulteriore bit è necessario per indicare se X si trova al di sopra di H o al di sotto di L, e pochi altri bit sono necessari per specificare l'esatto valore. Questo metodo raggiunge una buona compressione per due motivi: i due pixel che si trovano direttamente vicini ad X costituiscono un buon context per calcolare la predizione, inoltre il modello probabilistico assunto dall'algoritmo si avvicina a molte delle distribuzioni di probabilità delle immagini reali.

Quando esaminiamo la distribuzione dei valori dei pixel di un'immagine per ogni context Δ, troviamo che sono generalmente distribuiti come in Figura 3.3.

In genere X si trova all'interno del range $[L, H]$ circa nella metà dei casi, richiedendo un bit di codifica, mentre se non è compreso tra L ed H si utilizza un pixel aggiuntivo per per indicare se X si trova al di sotto o al disopra del range. All'interno dell'intervallo $[L, H]$ i valori di X si distribuiscono pressoché unifomemente, con una probabilità leggermente più alta al centro

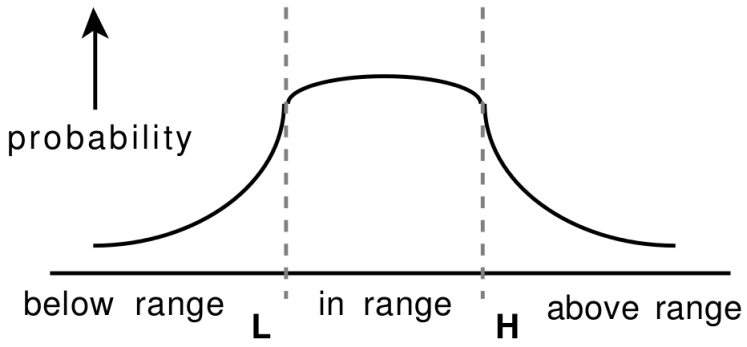

Figura 3.3: Schematizzazione della distribuzione di probabilità del pixel corrente con context $\Delta = H - L$.

del range. FELICS considera $X - L \in [0, \Delta]$ e, con una codifica binaria adattata si ottiene una compressione ottimale. Se $\Delta + 1$ è una potenza di due, utilizza un codice binario di lunghezza $\log(\Delta + 1)$, altrimenti è possibile sistemare il codice in modo da assegnare $\lfloor \log(\Delta + 1) \rfloor$ bits ad alcuni valori e $\lceil \log(\Delta + 1) \rceil$ bits agli altri. Poiché i valori centrali sono leggermente più probabili, si assegnano a questi ultimi le parole di codice più corte. Ad esempio se $\Delta = 4$, si costruisce un codice per i valori $0, 1, 2, 3, 4$. Le parole di codice sono 00, 01, 10, 110 e 111, si assegnano dalla più corta alla più lunga, in ordine lessicografico, a partire dai valori centrali, nel modo seguente: Come

$X - L$	0	1	2	3	4
codice	111	10	00	01	110

mostrato in Figura 3.3, la probabilità dei valori al di fuori del range decresce rapidamente man mano che ci si allontana da L o da H, quindi quando X si trova al di fuori del range è ragionevole utilizzare la codifica di Golomb-Rice.

Capitolo 4

Nuovi tentativi per la compressione di immagini

In questo capitolo vengono presentate ed analizzate tre nuove idee di metodi per la compressione di immagini a due livelli di colore, cioè in bianco e nero. I tre algoritmi proposti, implementati in linguaggio C, si basano sulla compressione tramite il metodo RLE ed agiscono su file in formato bitmap BMP. Un file bitmap è formato da due parti distinte. Nella prima parte, denominata HEADER, sono indicate informazioni come la grandezza in byte del file, il numero di colori utilizzati, la larghezza e l'altezza dell'immagine in pixel. I byte che seguono l'HEADER costituiscono il corpo della bitmap in cui l'immagine viene rappresentata attraverso una matrice (o mappa) di bit, dove ad ogni pixel si fa corrispondere un indice che ne indica il colore.

I programmi implementati richiedono in input il nome di un file in formato bitmap, lo aprono ed acquisiscono la matrice binaria, che denominiamo $\mathcal{I}$, relativa alla mappa di bit. Le dimensioni di $\mathcal{I}$ sono pari a $H \times L$, dove H indica l'altezza dell'immagine rappresentata nel file, mentre L indica la larghezza. L'immagine viene poi codificata attraverso uno dei metodi che verranno descritti e la stringa di codice ottenuta viene rappresentata in una nuova matrice $\tilde{\mathcal{I}}$ di dimensioni $\tilde{H} \times L$, con $\tilde{H} = \lceil l(C) \rceil$, dove $l(C)$ indica la stringa del codice dell'immagine. Agli $\tilde{H} \times L - l(C)$ elementi dell'ultima riga di $\tilde{\mathcal{I}}$ si assegna un valore corrispondente ad un indice relativo ad un terzo colore, ad esempio un rosso, ed infine $\tilde{\mathcal{I}}$ viene salvata in un file bitmap come mostrato nell'Esempio 4.1. Il file finale contiene la rappresentazione della stringa codice ottenuta per l'immagine. Gli eventuali bit finali di colore rosso non fanno parte del codice, ma sono necessari solo per la sua rappresentazione in formato BMP.

I tre programmi permettono inoltre di decodificare il file BMP finale relativo all'immagine compressa e di ricostruire l'immagine originale. Prima

di specificare il nome del file l'utente deve quindi scegliere tra modalità di codifica e modalità di decodifica. Nella modalità di decodifica i programmi acquisiscono la matrice $\tilde{\mathcal{I}}$, ricostruiscono la stringa di codice di lunghezza $l(C)$ escludendo gli ultimi elementi di $\tilde{\mathcal{I}}$ con valori relativi all'indice del colore rosso, e da questa ricostruiscono la matrice dell'immagine originale $\mathcal{I}$ salvandola infine in un file bitmap.

Esempio 4.1. Consideriamo l'immagine mostrata in Figura 4.1: le sue dimensioni sono di 70 pixel di altezza e 60 pixel di larghezza, per un totale di 4.200 pixel.

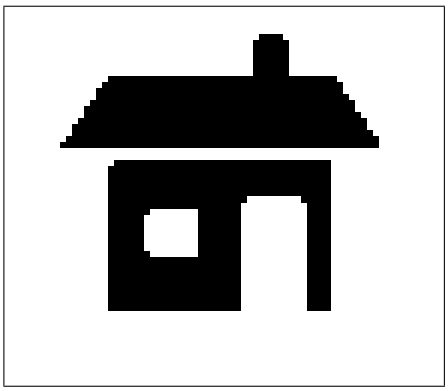

Figura 4.1: Immagine originale.

Figura 4.2: Compressione con il primo programma: RLE1.

Figura 4.3: Compressione con il secondo programma: RLEmod.

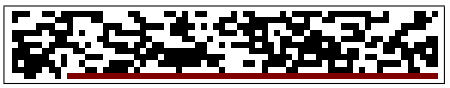

Figura 4.4: Compressione con il tezo programma: RLE4mod.

Figura 4.5: Le tre stringhe di codice binario ottenute con i tre programmi sono salvate come immagine in formato bitmap. I pixel neri corrispondono agli 0 del codice, i pixel bianchi agli 1, i pixel rossi non fanno parte della stringa di codice, sono necessari soltanto per riuscire a salvare il codice in questo formato.

Il primo metodo mostrato in Figura 4.2, RLE1, ottiene una stringa di codice lunga 1.316 bit ed un tasso di compressione pari a $31,33\%$. Il secondo, RLEmod, in Figura 4.3, codifica l'immagine con una stringa di 1.159 bit, ed ottiene un tasso di compressione pari a $27,60\%$. Infine, il terzo metodo

RLE4mod, in Figura 4.4, rappresenta l'immagine con una stringa di codice lunga 709 bit, ottenendo un tasso di compressione del $26,90\%$.

4.1 RLE1

Il primo metodo presentato, che chiameremo RLE1, è un semplice metodo di codifica RLE. L'algoritmo codifica gli elementi della matrice $\mathcal{I}$ in ordine orizzontale (cioè riga per riga), verticale (cioè colonna per colonna) ed infine in ordine obliquo a zig zag, cioè l'ordine rappresentato in Figura 4.6. A seconda delle caratteristiche proprie dell'immagine, uno qualsiasi di questi ordini può risultare essere il migliore.

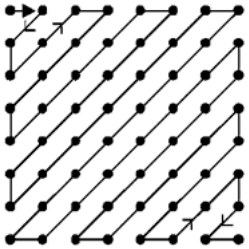

Figura 4.6: Pixel in ordine obliquo a zig zag.

RLE1 codifica i valori delle lunghezze dei run di pixel dello stesso colore consecutivi utilizzando 8 bit, cioè con valori compresi tra 1 e 256. In questo modo, ad esempio, un run di pixel bianchi di lunghezza $l > 256$ viene spezzato in $n = \left\lfloor \frac{l}{256} \right\rfloor$ run di pixel bianchi di lunghezza 256 (dove $\lfloor \cdot \rfloor$ indica il più grande intero minore), seguiti da un run finale di pixel bianchi di lunghezza pari al resto della divisione di l per 256. Si utilizzano 9 bit per ciascun run: il primo bit per indicare il colore del run ed i successivi 8 per rappresentarne la lunghezza.

L'algoritmo codifica con il metodo RLE1 le lunghezze dei run nei tre ordini di percorrenza della matrice $\mathcal{I}$, e confronta le lunghezze delle tre stringhe di codice ottenute. Infine costruisce la matrice $\tilde{\mathcal{I}}$ scegliendo la stringa di codice di lunghezza minore, con l'aggiunta di due bit iniziali che specificano quale dei tre ordini è risultato essere il migliore, al fine di permettere in seguito una corretta ricostruzione del file originale.

Esempio 4.2. Riprendiamo il caso dell'esempio 4.1. Comprimendo l'immagine mostrata in Figura 4.1 RLE1 ottiene una stringa di codice di 1.316 bit seguendo l'ordine orizzontale, di 1.820 bit con l'ordine verticale e di 2.738 bit con l'ordine obliquo a zig zag. RLE1 sceglie di codificare l'immagine

con la stringa ottenuta seguendo l'ordine orizzontale, e raggiunge un tasso di compressione del $31,33\%$.

4.2 RLEmod

Il secondo programma presentato, che chiameremo RLEmod, agisce in maniera analoga al precedente con una modifica nel metodo di codifica delle lunghezze. L'algoritmo, per ognuno degli ordini di percorrenza della matrice (orizzontale, verticale e obliquo a zig zag), effettua una scansione iniziale della matrice $\mathcal{I}$ e calcola le lunghezze massime dei run di pixel consecutivi per entrambi i colori, l_w bianco, l_b nero. Mentre RLE1 utilizza per ogni immagine un numero fisso di 8 bit per codificare le lunghezze dei run, RLEmod utilizza $\lceil \log_2 l_w \rceil$ bit per codificare le lunghezze dei run bianchi e $\lceil \log_2 l_b \rceil$ bit per le lunghezze dei run neri (dove $\lceil \cdot \rceil$ indica il più piccolo intero maggiore). In questo modo non è necessario spezzare in più parti un run di elevata lunghezza né utilizzare un bit prima di ogni lunghezza per indicarne il colore come avveniva per RLE1.

L'algoritmo, dopo aver effettuato la codifica in ognuno dei tre ordini, confronta le lunghezze delle tre stringhe di codice ottenute. Infine, analogamente ad RLE1, costruisce la matrice $\tilde{\mathcal{I}}$ scegliendo la stringa di codice di lunghezza minore ottenuta, con l'aggiunta di tredici bit iniziali contenenti informazioni utili alla corretta decodifica del file: i primi due bit specificano quale dei tre ordini è risultato essere il migliore, il terzo indica il colore del primo run codificato, mentre gli altri dieci bit contengono la rappresentazione binaria di l_w e l_b.

Esempio 4.3. Consideriamo di nuovo il caso dell'esempio 4.1 e comprimiamo l'immagine mostrata in Figura 4.1. Nella scansione orizzontale, RLEmod calcola $l_w = 787$ e $l_b = 53$, decide quindi di codificare le lunghezze dei run di pixel bianchi con $\lceil \log_2 l_w \rceil = 10$ bit, e con $\lceil \log_2 l_w \rceil = 6$ bit le lunghezze dei run di pixel neri, ottenendo per l'immagine una stringa di codice di 1.159 bit. RLEmod ripete lo stesso procedimento seguendo i restanti ordini, ottenendo una stringa di codice di 1.508 bit con l'ordine verticale, ed una stringa di codice di 1.159 bit con l'ordine obliquo. RLEmod sceglie di codificare l'immagine con la stringa più corta ottenuta. In caso di parità di lunghezza tra ordini orizzontale ed obliquo, RLEmod sceglie di codificare l'immagine con la stringa relativa all'ordine orizzontale, raggiungendo un tasso di compressione del $27,60\%$.

4.3 RLE4mod

L'ultimo programma presentato, che chiameremo RLE4mod, prevede la codifica dell'immagine negli ordini di percorrenza orizzontale o verticale considerando contemporaneamente 4 righe o (4 colonne).

Supponiamo di essere in modalità orizzontale, e sia $\mathcal{I} = (i_{j,k})$, con $1 \leq j \leq H$ e $1 \leq k \leq L$ la matrice binaria relativa alla mappa di bit. RLE4mod per riuscire a codificare 4 righe alla volta deve eventualmente aggiungere da una a tre righe bianche in fondo all'immagine per rendere il numero totale delle righe dell'immagine divisibile per 4. L'algoritmo, per calcolare le lunghezze dei run, scorre tutta la matrice $\mathcal{I}$ confrontando ogni vettore binario $(i_{j,k}, i_{j+1,k}, i_{j+2,k}, i_{j+3,k})$ con il successivo $(i_{j,k+1}, i_{j+1,k+1}, i_{j+2,k+1}, i_{j+3,k+1})$ come mostrato in figura 4.7.

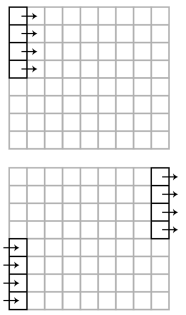

Figura 4.7: Modalità orizzontale.

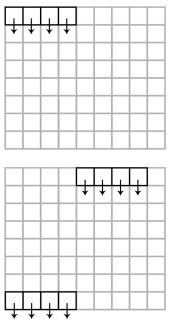

Figura 4.8: Modalità verticale.

Per ognuno dei due ordini di percorrenza, RLE4mod effettua una scansione iniziale della matrice $\mathcal{I}$ e, analogamente a RLEmod, calcola le lunghezze massime dei run consecutivi, ma lo fa solamente per i due elementi più probabili tra i 2^4 elementi possibili: l'elemento costituito da quattro pixel bianchi e l'elemento costituito da quattro pixel neri. Sia l_w la lunghezza massima dei run di elementi costituiti da quattro pixel bianchi, mentre sia l_b la lunghezza massima dei run di elementi costituiti da quattro pixel neri, si utilizzano $\lceil \log_2 l_w \rceil$ bit per codificare le lunghezze dei run di elementi del primo tipo e $\lceil \log_2 l_b \rceil$ bit per le lunghezze dei run di elementi del secondo tipo. Le lunghezze dei run costituiti dai restanti $2^4 - 2$ elementi vengono codificate con 3 bit per indicare valori compresi tra 1 ed 8 in binario. Come per il metodo RLE, per questo tipo di elementi elementi, è necessario suddividere i run di lunghezza $l > 8$ in più run consecutivi. Ogni lunghezza codificata viene preceduta da 4 bit che specificano l'elemento ripetuto nel run. Per effettuare la codifica in modalità verticale RLE4mod applica lo stesso procedimento sulla matrice $\mathcal{I}^t$, trasposta della matrice della mappa di bit $\mathcal{I}$.

Dopo aver effettuato la codifica in ordine orizzontale e verticale, RLE4mod confronta le lunghezze delle due stringhe di codice ottenute e costruisce la matrice $\tilde{\mathcal{I}}$ con la stringa di codice di lunghezza minore preceduta da 23 bit iniziali contenenti informazioni utili alla decodifica del codice: il primo bit specifica quale dei due ordini di percorrenza è risultato essere il migliore, i successivi dieci contengono la rappresentazione binaria di l_w ed l_b, e gli altri 12 la rappresentazione binaria dell'altezza H dell'immagine.

Esempio 4.4. Torniamo ancora al caso dell'esempio 4.1 e comprimiamo l'immagine mostrata in Figura 4.1 con RLE4mod. Nella scansione orizzontale, l'algoritmo calcola $l_w = 157$ e $l_b = 46$, decide quindi di utilizzare $\lceil \log_2 l_w \rceil = 8$ bit per rappresentare le lunghezze dei run orizzontali di elementi costituiti da quattro pixel bianchi, e $\lceil \log_2 l_b \rceil = 6$ bit per rappresentare le lunghezze dei run orizzontali di elementi costituiti da quattro pixel neri, ottenendo per l'immagine una stringa di codice lunga 709 bit. RLE4mod ripete lo stesso procedimento seguendo l'ordine verticale. Ottiene $l_w = 158$ e $l_b = 25$, e quindi rappresenta con $\lceil \log_2 l_w \rceil = 8$ bit le lunghezze dei run verticali di elementi costituiti da quattro pixel bianchi, e con $\lceil \log_2 l_b \rceil = 5$ le lunghezze dei run verticali di elementi costituiti da quattro pixel neri, riuscendo a rappresentare l'immagine con una stringa di codice lunga 761 bit. Infine l'algoritmo sceglie la stringa più corta ottenuta per la codifica finale dell'immagine, cioè la stringa relativa all'ordine orizzontale, raggiungendo un tasso di compressione del $16,88\%$.

4.4 Confronto fra gli algoritmi

Per stimare la bontà dei diversi metodi costruiti, è necessario stimare il contenuto informativo delle immagini, e per fare ciò utilizziamo il concetto di entropia. Il tasso di entropia di una sorgente è una quantità che ne misura il contenuto informativo, ma dipende dalla natura statistica della sorgente stessa, spesso sconosciuta nella realtà. Ricordiamo che, data una sorgente $\mathcal{S} = \{X_k\}_{k \in \mathbb{N}}$, il tasso di entropia di $\mathcal{S}$ è dato da

$$H(\mathcal{S}) = \lim_{n \to \infty} \frac{1}{n} H(X_1, X_2, \ldots, X_n), \tag{4.1}$$

se tale limite esiste. Non essendo in presenza di un processo stocastico, ma di una famiglia di simboli (bits o bytes) ben determinata e per di più non data in maniera naturale come sequenza, è difficile anche definire cosa significhi il tasso di entropia per un'immagine in bianco e nero. Possiamo tuttavia definire delle versioni "sperimentali" di tale quantità, con le quali possiamo

fornire una stima del limite inferiore del tasso di compressione per simbolo che un qualsiasi algoritmo può raggiungere nella codifica della stringa S.

Sia $\mathcal{A}$ l'alfabeto della sorgente $\mathcal{S}$ formato da A simboli, $\mathcal{A} = \{x_1, x_2, \ldots, x_A\}$, sia S una stringa di elementi emessi dalla sorgente $\mathcal{S}$ di lunghezza N, sia n_i il numero di occorrenze di x_i in S. Data la stringa S possiamo stimare il tasso di entropia $H(X)$ tramite il tasso di entropia empirica di ordine zero della stringa S, dato da

$$H_0(S) = -\sum_{x_i \in \mathcal{A}} \frac{n_i}{N} \log \frac{n_i}{N},$$

dove si assume $0 \log 0 = 0$. Notiamo che questa tipologia di entropia si ottiene considerando solamente la frequenza dei singoli elementi e non le varie interdipendenze tra essi, che possono anche essere molto marcate. Per sfruttare l'informazione relativa alle relazioni tra i simboli occorre calcolare i valori del tasso di entropia empirica per sequenze di maggiore lunghezza. Sia $\mathbf{w} = w_0, w_1 \ldots, w_k \in \mathcal{A}^k$ una generica sequenza di $k+1$ simboli nell'alfabeto $\mathcal{A}$. Data una stringa S di lunghezza N, possiamo stimare il valore di $H(X_0, X_1, \ldots, X_k)$ nella stringa S tramite il tasso di entropia empirica di ordine k della stringa S, dato da

$$H_k(S) = -\frac{1}{k+1} \sum_{\mathbf{w} \in \mathcal{A}^k} \frac{n_{\mathbf{w}}}{N-k} \log \frac{n_{\mathbf{w}}}{N-k},$$

dove $n_{\mathbf{w}}$ è il numero di occorrenze della sequenza $\mathbf{w}$ nella stringa S.

Al fine di effettuare il calcolo delle quantità appena definite sulla matrice $\mathcal{I}$ relativa alla mappa dei bit di un'immagine data, si è creato un quarto programma in linguaggio C, denominato Entropia. Data $\mathcal{I}$, il programma calcola il tasso di entropia empirica di ordine k per $k = 0, 1, \ldots, 10$, per blocchi di pixel di lunghezza $k+1$ orizzontali, verticali e obliqui disposti come mostrato in Figura 4.12, per quadrati di 2×2 pixel (tasso di entropia empirica di ordine 3) e di 3×3 pixel (tasso di entropia empirica di ordine 8). Infine effettua una stima del tasso di entropia della sorgente dell'immagine attraverso una media tra i valori del tasso di entropia empirica per gli ordini più grandi $k = 8, 9, 10$, più precisamente calcola

$$\tilde{H} = \frac{1}{3} \sum_{k=8}^{10} \frac{H_k^{\text{orizz}} + H_k^{\text{vert}} + H_k^{\text{obl}}}{3},$$

dove con H_k^{orizz}, H_k^{vert} e H_k^{obl} indichiamo i valori del tasso di entropia empirica di ordine k calcolati per blocchi di $k+1$ pixel orizzontali, verticali e obliqui.

Per avere un'approssimazione attendibile del tasso di entropia della sorgente di un'immagine è necessario effettuare questi calcoli su un elevato nu-

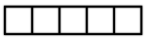

Figura 4.9

Figura 4.10

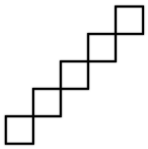

Figura 4.11

Figura 4.12: Disposizione dei blocchi di 5 pixel orizzontali 4.9, verticali 4.10 ed obliqui 4.11. Tali blocchi vengono utilizzati dal programma Entropia per calcolare i rispettivi tassi di entropia empirica H_k^{orizz}, H_k^{vert} e H_k^{obl} con k=4.

mero di pixel, per questo motivo sono state scelte immagini di dimensioni molto più elevate rispetto a quella mostrata nell'Esempio 4.1.

Esempio 4.5. Nell'immagine in Figura 4.14 si mostra un grafico di come varia il tasso di entropia empirica rispetto all'ordine k per blocchi di pixel orizzontali, verticali e obliqui, per l'immagine mostrata in Figura 4.13, che raffigura una mappa dell'Italia. I valori del tasso di entropia empirica

Figura 4.13: italy_map.bmp

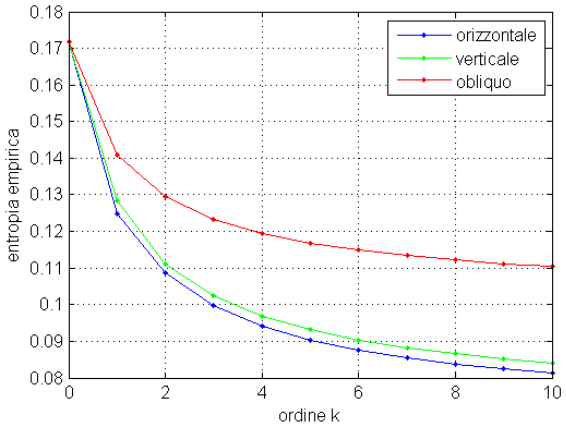

Figura 4.14: Grafico che mostra come varia il tasso di entropia empirica con l'ordine k considerando i tre diversi tipi di blocchi.

ottenuti per blocchi di pixel orizzontali risultano essere minori rispetto ai valori calcolati per blocchi orizzontali ed obliqui. Le interdipendenze tra i pixel sono più forti se questi vengono considerati nel verso orizzontale, se ne ha un

riscontro anche nel fatto che tutti e tre i metodi di codifica testati scelgono di codificare l'immagine italy_map.bmp nella modalità orizzontale.


Archeologia Industriale
oltre l'elaborazione teorica

Figura 4.15: art.bmp

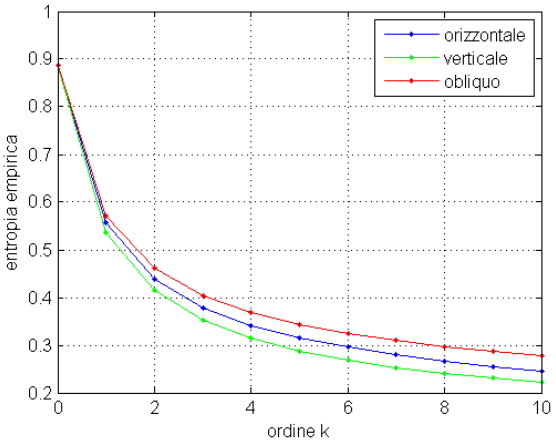

Figura 4.16: Grafico che mostra come varia il tasso di entropia empirica con l'ordine k considerando i tre diversi tipi di blocchi.

Nel caso dell'immagine mostrata in Figura 4.15, che contiene un articolo di giornale, i valori del tasso di entropia empirica sono mostrati nel grafico in Figura 4.16. Si nota che i valori ottenuti per blocchi verticali sono minori rispetto a quelli ottenuti per blocchi orizzontali ed obliqui. Anche in questo caso tutti e tre i metodi di codifica testati scelgono di codificare l'immagine art.bmp seguendo l'ordine rispetto al quale le interdipendenze tra i pixel sono più forti, cioè l'ordine verticale.

Nella Tabella 4.1 sono riportati i risultati ottenuti testando i tre programmi di codifica ed il programma Entropia su diverse tipologie di immagini campione, cioè le immagini (4.13) e (4.15) mostrate nell'Esempio 4.5, (4.17) che raffigura lo spartito di una canzone, (4.18) che contiene un disegno, (4.19) che contiene il Canto V della Divina Commedia, (4.20) che contiene un modello di lettera formale, (4.21) che contiene un altro disegno e (4.22) che contiene un cruciverba. Osservando i valori dei tassi di compressione ottenuti con i tre metodi sulle diverse tipologie di immagini si può notare che RLE4mod risulta essere il migliore compressore per tutte le immagini. Questo risultato è dovuto al fatto che, nelle immagini reali a due livelli di colore, è molto probabile trovare regioni ad alta concentrazione di pixel dello stesso colore, si pensi, ad esempio, al margine di un foglio di testo. Per questa tipologia di immagini la scelta di codificare quattro righe (o colonne)

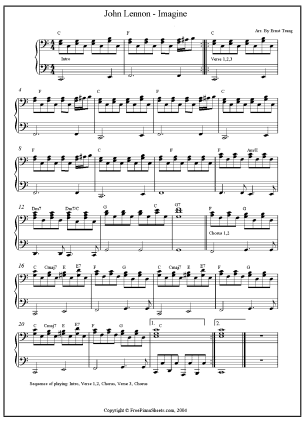

Figura 4.17: spartito.bmp

Figura 4.18: notes.bmp

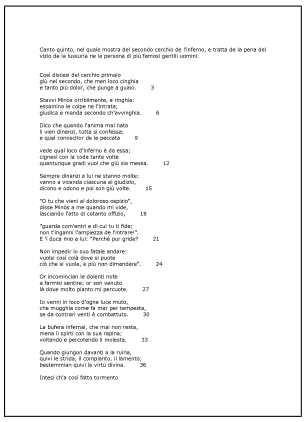

Figura 4.19: cantoV.bmp

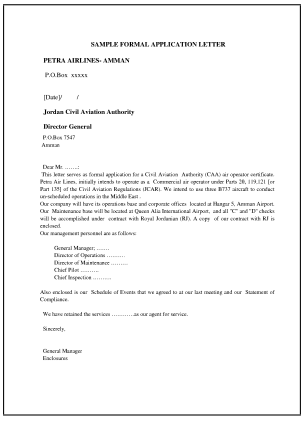

Figura 4.20: form.bmp

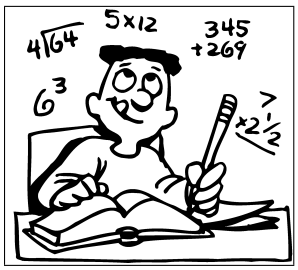

Figura 4.21: math.bmp

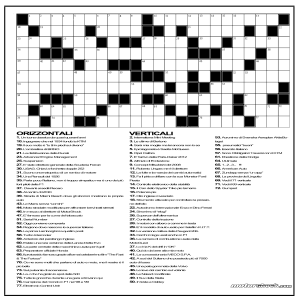

Figura 4.22: Cruciverba.bmp

contemporanemente è molto sensata. Si può inoltre osservare che RLE1 è migliore di RLEmod tranne in alcuni casi. Supponiamo di comprimere l'immagine art.bmp in Figura 4.15 con RLEmod. Consideriamo solamente la codifica in ordine di percorrenza verticale della matrice dei pixel, che risulta essere la migliore rispetto agli altri ordini. L'algoritmo calcola le lunghezze massime dei run bianchi $l_w = 233.134$ e neri $l_b = 1.788$, e rappresenta con $\lceil \log l_w \rceil = 18$ bit le lunghezze dei run bianchi e con $\lceil \log l_b \rceil = 11$ bit le lunghezze dei run neri. Questa scelta è conveniente nelle aree ad alta concentrazione di pixel dello stesso colore, come ad esempio nei margini della pagina oppure nella rappresentazione dell'immagine affiancata all'articolo, ma è fortemente controproducente nell'area in cui è rappresentato il testo, dove è presente un elevato numero di run di breve lunghezza. Rappresentare le numerose lunghezze brevi con lo stesso numero di bit necessari a descri-

Immagine	Dimensioni	$H_{3\times3}$	$\tilde{H}$	RLE1	RLEmod	RLE4mod
spartito.bmp	1.839768	0.16368	0.26482	21.72%	31.40%	13.96%
italy_map.bmp	2.089.600	0.06894	0.093387	13.95%	15.72%	7.82%
notes.bmp	2.113.382	0.13038	0.14608	6.72%	6.12%	4.16%
cantoV.bmp	2.174.960	0.09117	0.12103	18.04%	20.14%	10.66%
form.bmp	2.174.960	0.07640	0.10051	15.73%	17.13%	8.92%
math.bmp	3.610.000	0.12547	0.14718	8.40%	8.17%	5.26%
cruciverba.bmp	9.000.000	0.15605	0.20898	20.54%	31.52%	12.80%
art.bmp	9.358.263	0.22238	0.25745	27.41%	41.67%	18.42%

Tabella 4.1: Per ogni immagine si indicano le dimensioni in numero di pixel, il tasso di entropia empirica di ordine 8 calcolato per blocchi di 3×3 pixel $H_{3\times3}$, la stima del tasso di entropia della sorgente $\tilde{H}$, entrambe calcolate con il programma Entropia, ed i tassi di compressione ottenuti con i metodi RLE1, RLEmod e RLE4mod.

vere la lunghezza maggiore è estremamente costoso. Per questa immagine, RLE1, che rappresenta le lunghezze dei run con parole di codice di lunghezza fissa di 9 bit, ottiene un tasso di compressione del 27.41%, migliore di quello ottenuto da RLEmod che è pari al 41.67%. Supponiamo ora di comprimere la stessa immagine con RLE4mod. Anche in questo caso consideriamo solamente i risultati ottenuti con la codifica in ordine verticale. L'algoritmo calcola le lunghezze massime dei run di elementi costituiti da quattro pixel bianchi, $l_w = 1.717$, e dei run di elementi costituiti da quattro pixel neri, $l_b = 57.599$, e codifica con $\lceil \log l_w \rceil = 11$ bit le prime, e con $\lceil \log l_b \rceil = 16$ bit le seconde. In RLE4mod, la scelta di codificare quattro colonne contemporaneamente riduce la lunghezza del codice necessario a descrivere le regioni ad alta concentrazione di pixel dello stesso colore ad un quarto del codice di cui necessita RLEmod. Inoltre, le regioni dell'immagine 4.15 in cui è rappresentato il testo dell'articolo, sono composte da un'alta concentrazione di run costituiti dai $2^4 - 2$ elementi che non sono formati da quattro pixel dello stesso colore, e che sono quindi codificati con una parola di codice di soli 7 bit. Queste due caratteristiche rendono RLE4mod il compressore migliore, fra i tre presentati, in ognuna delle tipologie di immagini testate. Infine, notiamo che RLEmod risulta essere migliore di RLE1 solamente nelle immagini contenenti disegni, math.bmp e notes.bmp, dove le lunghezze dei run sono più omogenee. Più precisamente, sia l la lunghezza massima dei run di un colore fissato sulla matrice della mappa dei pixel di $\mathcal{I}$ percorsa in un dato ordine. La codifica RLEmod diventa svantaggiosa quando, per ognuno degli ordini di percorrenza della matrice $\mathcal{I}$, si verifica che $l \gg \bar{l}$, dove $\bar{l}$ è la media delle lunghezze dei run del colore fissato sulla matrice $\mathcal{I}$.

Bibliografia

[1] T. Cover and J. Thomas, *Elements of Information Theory, Second Edition*, John Wiley & Sons, 2006.

[2] I. H. Witten, A. Moffat, T. C. Bell, *Managing Gigabytes: Compressing and Indexing Documents and Images, Second Edition*, Morgan Kaufmann Publishers, 1994.

[3] K. Sayood, *Introduction to Data Compression, Third Edition*, Morgan Kaufmann Publishers, 2006.

[4] , R. W. Hamming, *Coding and Information Theory, Second Edition*, Prentice-Hall, 1986.

[5] J. Ziv, A. Lempel, *A Universal Algorithm for Sequential Data Compression*, IEEE Transactions on Information Theory, IT-23(3): 337-343, 1977.

[6] A. D. Wyner and J. Ziv,*The Sliding Window Lempel-Ziv Algorithm is Asymptotically Optimal*, Proceeding of the IEEE, 82(6): 872-877, 1994.

[7] A. D. Wyner and J. Ziv,*Compression of Individual Sequences by Variable Rate Coding*, IEEE Transactions on Information Theory, IT-24: 530-536, 1976.

[8] T. A. Welch, *A Technique for High-Performance Data Compression.*, IEEE Computer, pages 8-19, 1984.

[9] M. A. Nelson, *LZW Data Compression* Dr Dobb's Journal, 14(10): 29-36, 1989.

[10] J. G. Cleary, and I. H. Witten, *Data Compression Using Adaptive Coding and Partial String Matching*, IEEE Transactions on Communications, IT-32(4): 396–402, 1984.

[11] M. Burrows and D. Wheeler, *A Block Sorting Lossless Data Compression Algorithm*, Technical Report 124, Digital Equipment Corporation, 1994.

[12] X. Wu and N. D. Memon, *CALIC - A Context Based Adaptive Lossless Image Codec*, Proceedings of the IEEE, Vol 4: 1890–1893, 1996.

[13] S. Na and K. Sayood, *Recursive Indexing Preserves the Entropy of a Memoryless Geometric Source*, IEEE Transactions on Information Theory, IT-38(5): 1602-1609, 1996.

[14] J. O'Neal, *Predictive Quantizing Systems (differential pulse code modulation) for the Transmission of Television Signals*, Bell System Technical Journal, 45: 689—722, 1966.

[15] M. J. Weinberger, G. Seroussi and G. Sapiro, *LOCO-I: A Low Complexity Lossless Image Compression Algorithm*, IEEE Data Compression Conference, 140-149, 1996.

[16] M. J. Weinberger, G. Seroussi and G. Sapiro, *The LOCO-I Lossless Image Compression Algorithm: Principles and Standardization into JPEG-LS*, IEEE Transaction on Image Processing, 9(8): 1309-1324, 2000.

[17] N. Menom and X. Wu, *Recent Developments in Context-Based Predictive Techniques for Lossless Image Compression*, The Computer Journal, 40: 127-136, 1997.

[18] S. W. Golomb, *Run length encodings*, IEEE Transactions on Information Theory, IT-12: 399-401, 1966.

[19] R. Gallager and D. V. Voorhis, *Optimal Source Codes for Geometrically Distributed Integer Alphabets*, IEEE Transactions on Information Theory, IT-21: 228–230, 1975.

[20] P. G. Howard and J. S. Vitter, *Fast and Efficient Lossless Image Compression*, IEEE Data Compression Conference, 351-360, 1993.

Ringraziamenti

Un ringraziamento particolare al Prof. Marco Lenci innanzitutto per avermi guidato con entusiasmo, pazienza, capacità e professionalità durante tutto il lavoro di tesi. Vorrei inoltre ringraziarlo perché, senza rendersene conto, attraverso questa tesi è riuscito a farmi ritrovare quegli obiettivi che avevo un po' perso di vista in questi ultimi anni in cui la vita è stata piuttosto dura con me.

Un ringraziamento va anche alla mia famiglia, a mia madre, che credeva in me e lo ripeteva ogni giorno, a mio padre, che crede in me ma è troppo burbero per dirlo, e a mia sorella, per i suoi modi singolari di supportarmi.

Ringrazio Antonio per avermi accompagnato per tutto questo tempo, per non avermi mai lasciato la mano, per essere stato ogni giorno il mio punto fermo, per aver imparato a gestire la mia intrattabilità pre-esami pur di starmi vicino, per la sua allegria e tutto il bene che mi ha regalato.

Un immenso grazie anche a nonna Rita, perché ogni volta che piombo in casa sua per lei è una festa, ed è capace di organizzarmi un pranzo della domenica anche il lunedì alle due del pomeriggio.

Un grazie infinito a Jessica per esserci sempre, perché anche se mi faccio vedere meno di una volta al mese è come se fossi sempre stata lì. Grazie perché quando siamo insieme rivivo la spensieratezza dei nostri quindici anni.

Ringrazio anche tutta la squadra dei miei amici di Roncitelli perché sono riusciti a farmi ridere a crepapelle nei momenti in cui anche sorridere sembrava impossibile. Un grazie in più a Edoardo, che con le sue magie informatiche ripara i miei disastri con linux da 200 km di distanza, e a Claudia, la mia psicologa laureata in economia che non smette di ascoltare le mie paranoie anche se ogni volta rischia la sanità mentale.

Devo ringraziare anche i miei compagni di viaggio, gli amici dell'università. Grazie a quelli di ieri, che mi hanno aiutato tanto quando la matematica per me era già splendida ma ancora totalmente misteriosa, grazie Marco per i pomeriggi passati a fare esercizi. Grazie anche a tutti quelli di oggi, quelli che mi hanno accompagnato fin qui, Chiara, Sara, Virginia e tutti gli altri. Grazie alle People of Spritz per gli aperitivi e per le risate e i deliri dopo

una giornata passata sui libri. Anche qui un grazie in più va a Katia, perché negli ultimi mesi le ho dato almeno un buon motivo al giorno per picchiarmi e non l'ha mai fatto, e per aver organizzato tutto quello a cui non riuscivo a pensare. Potresti essere un'ottima segretaria!

Ringrazio tutti i miei coinquilini...che sono tanti! Grazie a Paola e Francesca per essere state la mia casa in questa città che abbiamo conosciuto insieme, per tutti i bei momenti che ci hanno legato così tanto, perché dopo tutto questo tempo passeggiando per Bologna mi ritrovo ancora a ridere ripensando a noi. Ringrazio Alessandra che ha vissuto con me nel periodo peggiore, ma non so come è riuscita a volermi bene lo stesso. Ringrazio le mie piccole Mimì e Cocò, le inseparabili Giorgia e Adriana, per avermi sopportato in questi giorni, e per aver riempito le mie giornate di Puglia, musica napoletana e risate.

Devo ringraziare tante altre persone, tutte quelle persone che mi sono state vicine nei momenti in cui rischiavo di cadere a pezzi. Ringrazio tutta la mia enorme famiglia, a Fano e a Senigallia, soprattutto mia cugina Silvia che mi ha sempre aiutato ad affrontare le cose, anche qui a Bologna.

Grazie di cuore a tutti.