DISCLAIMER

Il materiale contenuto nel drive è stato raccolto e richiesto tramite autorizzazione ai ragazzi frequentanti il corso di studi di Informatica dell'Università degli Studi di Salerno. Gli appunti e gli esercizi nascono da un uso e consumo degli autori che li hanno creati e risistemati per tanto non ci assumiamo la responsabilità di eventuali mancanze o difetti all'interno del materiale pubblicato.

Il materiale sarà modificato aggiungendo il logo dell'associazione, in tal caso questo possa recare problemi ad alcuni autori di materiale pubblicato, tale persona può contattarci in privato ed elimineremo o modificheremo il materiale in base alle sue preferenze.

Ringraziamo eventuali segnalazioni di errori così da poter modificare e fornire il miglior materiale possibile a supporto degli studenti.

CoScienze
Associazione

Università degli Studi di Salerno

DIPARTIMENTO DI INFORMATICA

CORSO DI LAUREA MAGISTRALE IN CYBERSECURITY

COMPRESSIONE DATI

Professore:

Bruno Carpentieri s

Studente:

Coscienze

matr

Anno Accademico 2024 - 2025

CoScienze
Associazione

Indice

Capitolo 1

Compressione Dati

La compressione dati è il processo per codificare un insieme di dati D in un insieme più piccolo D', la condizione da dover rispettare per tale codifica è che da D', tramite la decodifica, possiamo tornare a D oppure ad una *approssimazione accettabile* di D.
Il tipo di ricostruzione che riusciamo a raggiungere rappresenta la differenza fondamentale tra:

- **Compressione Lossless** (*Bit Preserving Compression*), in cui non si ha alcuna perdita di informazioni e dalla quale si può ricostruire esattamente il dato originale. Viene usata quando si lavora con dati che, una volta decompressi, non devono differire da quelli originali (es.: file di testo).
- **Compressione Lossy**, in cui si accetta di perdere qualche informazione a patto di poter migliorare la compressione senza recare alcun danno all'end-user a causa della perdita di informazioni. Viene usata quando si lavora con dati la cui perdita di informazioni, a causa del processo di compressione-decompressione, non è percepibile dall'end-user (es.: immagini e/o video).

Non tutti i dati possono essere compressi : **per essere compressi devono essere presenti ridondanze** altrimenti non è possibile alcun tipo di compressione.

1.1 Entropia

Un **alfabeto** è un insieme finito che contiene almeno un elemento (chiamato simbolo). Una stringa su un alfabeto è una sequenza di simboli appartenenti all'alfabeto stesso.
Sia $\Sigma = \{s_1 ... s_k\}$ con $k \geq 1$ un alfabeto, una **sorgente** è un processo che sequenzialmente produce caratteri dell'alfabeto sorgente Σ.

Una **sorgente** è **di primo ordine** se ad ogni elemento di Σ sono collegate probabilità indipendenti $p_1...p_k$ (dove p_i è la probabilità che il simbolo s_i venga emesso dalla sorgente) la cui somma è 1.
Una **sorgente costante K-aria**, con $k \geq 1$, è caratterizzata dalla presenza di un unico carattere dell'alfabeto che viene sempre scelto per la trasmissione, c'é quindi un i tale che $p_i = 1$, quindi, per $j \neq i$ risulterà che $p_j = 0$.
In una **sorgente casuale K-aria**, con $k \geq 1$, la probabilità che il prossimo carattere venga trasmesso è scelto in maniera uniforme ed indipendente, quindi $p_i = 1/k$ con $1 \leq i \leq k$.

L'**entropia** va a misurare il contenuto di informazione che si ottiene andando ad osservare la sorgente, ovvero il grado di sorpresa che si ha andando ad osservare i simboli in output da una sorgente. L'entropia risulterà:

- **massima per la *sorgente casuale***, visto che ogni simbolo viene dato in output con la stessa probabilità, quindi l'informazione è massima.
- **minima per la *sorgente costante***, visto che il simbolo in output è sempre lo stesso, quindi l'informazione è minima.

Esiste una **relazione tra entropia e rapporto di compressione** : più alta è l'entropia di una sorgente e minore sarà la mia possibilità di poter comprimere l'output di tale sorgente.
Supponiamo di avere un intero $k \geq 1$, sia S una sorgente del primo ordine che genera caratteri dall'alfabeto $\Sigma = \{s_1...s_k\}$ con probabilità indipendenti $\{p_1...p_k\}$. **L'entropia di S con *radice* r, con $r > 1$ è data da**:

$$H_r(S) = \sum_{i=1}^{k} p_i \cdot \log_r \left(\frac{1}{p_i}\right)$$

quando la radice non è specificata, si assume che $r = 2$ e si abbrevia $H_2(S)$ in $H(S)$.
Il **teorema fondamentale di codifica sorgente** afferma che:

- Sia S una *sorgente del primo ordine* su un alfabeto Σ e sia Γ un alfabeto composto da r caratteri, con $r \geq 1$. **Codificare i caratteri di S utilizzando i caratteri di Γ richiederà in media $H_r(S)$ caratteri di Γ per ogni carattere di Σ.**
- Inoltre, per ogni numero reale $\epsilon > 0$, **esiste uno schema di codifica che utilizza in media $H_r(S) + \epsilon$ caratteri di Γ per ogni carattere di Σ.**

Data una certa sorgente esiste una misura del suo contenuto di informazione che non possiamo battere, tale misura è l'entropia. Esistono algoritmi di codifica che sono ottimali nel senso della teoria dell'informazione, ovvero che sono garantiti arrivare al limite dell'entropia, ma per dimostrare tale affermazione bisogna utilizzare stringhe di lunghezza infinita.

L'algoritmo di Huffman, ad esempio, ha la limitazione che ogni carattere emesso dalla sorgente deve spendere almeno un bit.

Alcune informazioni sulla compressione sono:

- Dati casuali non possono essere compressi: entropia massima $\rightarrow$ non c'è ridondanza per la compressione.
- I dati compressi da una compressione ottimale non possono essere compressi ulteriormente.
- Non si può garantire che una compressione ottenga una performance specifica su tutti i tipi di dati, non si può quindi definire a priori il rapporto di compressione.

Capitolo 2

Tecniche di Codifica

Per compressione dati intendiamo il processo di codifica con il quale prendiamo **simboli dall'alfabeto sorgente e li trasformiamo in simboli di un alfabeto di codifica**. Se la compressione è stata efficace, la stringa codificata risultante sarà più piccola della stringa originale di simboli. Il processo decisionale con il quale decidiamo di mappare i caratteri tra i due alfabeti si basa su modelli, una collezione di dati e regole che vengono usati per definire per ogni simbolo in input quale codice dare in output.
Un primo tipo di codifica è quella a **blocchi k-ari**, il cui codice mappa stringhe di lunghezza n di un alfabeto sorgente in stringhe di un alfabeto di codifica di lunghezza $\lceil \log_k(n) \rceil$. Un esempio è la codifica **ASCII**, però utilizzare stringhe di lunghezza fissata potrebbe non essere un approccio ottimale.
Un'idea migliore potrebbe essere avere un codice che **associa a simboli più comuni delle stringhe di lunghezza minore** (in quanto presentano più ridondanza, mi aspetto lunghe stringhe di **a**) e a simboli meno comuni delle stringhe più lunghe.
Sia S un insieme sorgente e Σ un alfabeto codice. Un **codice da** S **a** Σ è una **funzione** f **che mappa** ogni elemento di S (alfabeto sorgente) in una stringa non vuota di Σ (alfabeto di codifica). Il dominio di f è chiamato l'insieme delle parole codici di f. La funzione f può essere estesa per applicarsi a qualsiasi lista finita di elementi definendo:

$$f(s_1 ... s_k) = \prod_{i=1}^{k} f(s_i)$$

dove per produttoria si intende la concatenazione, quindi avremo che $f(s_1 ... s_k) = f(s_1) \cdot f(s_2) \cdot ... \cdot f(s_k)$.
Un codice f **è iniettivo se non mappa mai due elementi alla stessa stringa**, però non garantisce che stringhe differenti non abbiano la stessa codifica.

2.1 Codici Univocamente Decifrabili

Sia f un codice da S su Σ, una **stringa** α su Σ è **univocamente decifrabile** rispetto ad f se esiste al più una lista L di elementi di S tale che $f(L) = \alpha$.

- Quindi **l'unico modo per avere** α **è tramite la codifica di** $f(L)$.

Inoltre, f **è univocamente decifrabile se tutte le stringhe su** Σ **sono univocamente decifrabili secondo** f.
Esempio:
$S = \{a, b, c, d, e\}$
$\Sigma = \{0, 1\}$
$f(a) = 00, f(b) = 01, f(c) = 10, f(d) = 11, f(e) = 100$
Dato l'alfabeto sorgente S, l'alfabeto di codifica Σ ed il codice f, possiamo dire che f non è univocamente decifrabile perché:

- $f(cba) = f(ee) = 100100$

2.2 Disuguaglianza Kraft-McMillan

Dato un insieme finito S ed un alfabeto di codifica Σ, se le lunghezze delle parole codice sono specificate a priori (con lunghezza $l_1, ..., l_{|S|}$ rispettivamente per i simboli $s_1, ..., s_{|S|}$), **un codice univocamente decifrabile da** S **a** Σ **esiste** se si soddisfa la **Disuguaglianza Kraft-McMillan**:

$$\sum_{i=1}^{|S|} \left(\frac{1}{|\Sigma|^{l_i}} \right) \leq 1$$

2.3 Codice Prefisso

Un codice f da un insieme S ad un alfabeto Σ è definito **codice prefisso** se nessuna parola codice è prefisso di un'altra parola codice. Un **codice prefisso** è necessariamente **univocamente decifrabile**, infatti leggendo sia da destra che da sinistra non c'è ambiguità tra la fine di un simbolo e l'inizio di un altro, garantendo inoltre una codifica immediata (per questo motivo vengono definiti **codici istantanei**), però non è detto che un codice univocamente decifrabile sia un codice prefisso.
Un codice prefisso **esiste per ciascuna sequenza di parole codici che abbiano una lunghezza tale che la disuguaglianza di Kraft-McMillan si può applicare**.
Di conseguenza, se la disuguaglianza di Kraft-McMillan è soddisfatta, non solo esiste un codice univocamente decifrabile ma è anche codice prefisso.

Capitolo 3

Codifica di Huffman

L'Huffman Coding permette di avere un codice prefisso e crea codici di lunghezza variabile, perché simboli più probabili saranno compressi in stringhe più corte mentre simboli meno probabili saranno compressi in stringhe più lunghe. Il codice di Huffman è preciso: la decodifica si effettua in maniera veloce ed efficiente tramite un albero binario di decodifica nonostante la lunghezza variabile.
Dato un insieme sorgente S, supponiamo di sapere per ogni elemento x in S a priori la probabilità p_x che x sarà il prossimo elemento emesso dalla sorgente. Invece di usare un codice di $\lceil \log_{|\Sigma|} |S| \rceil$, come avverrebbe con un codice a blocchi, **possiamo salvare spazio assegnando stringhe più corte a caratteri più comuni**.
D'ora in poi, per semplificare i calcoli, assumeremo che $\Sigma = \{0, 1\}$ *e quindi* $|\Sigma| = 2$.
L'algoritmo della codifica di Huffman è il seguente:

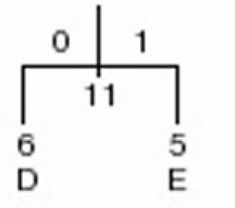
1. **Inizializza** $FOREST$ con alberi per ogni elemento x di S ed **assegna un peso** $T_x = p_x$.
2. **while** $|FOREST| > 1$ **do begin**:
   - Selezioniamo i due alberi in $FOREST$ con il peso minore, indichiamoli con Y e Z.
   - **Combina** Y **e** Z creando una radice r per i due alberi:
     - Con peso $weight(Y) + weight(Z)$.
     - Collegato ad Y e ad Z tramite path con peso 0 e 1 (l'ordine non conta).

Esempio: Consideriamo il seguente esempio:

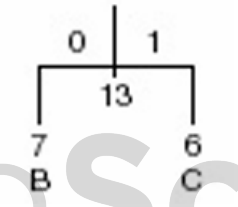
Symbol	Count
A	15
B	7
C	6
D	6
E	5

Ad ogni simbolo associamo il conteggio di quante volte tale simbolo è uscito dalla sorgente (simile alla probabilità). L'algoritmo procede come segue:

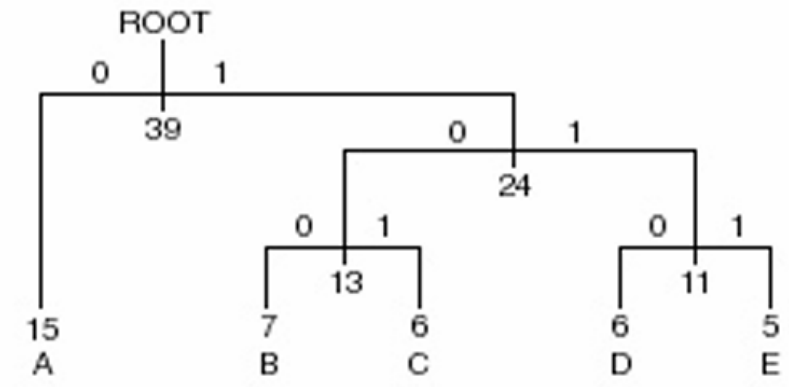
Passo 1: Prendiamo gli alberi con peso minore D ed E ed introduciamo un nodo padre con peso $weight(D) + weight(E) = 6 + 5 = 11$ e associamo in maniera arbitraria la codifica 0 ed 1 per l'arco di D e di E, creando quindi l'albero DE con peso 11 che prenderà il posto di D ed E all'interno di $FOREST$.



Passo 2:
Ripetiamo tale procedimento per tutti gli alberi fin quando non rimane un ultimo albero. Esempio di B e C:



L'albero risultante sarà:



L'Huffman coding risulta essere **ottimale nel senso della teoria dell'informazione** : applicando tale codifica su una stringa infinita di simboli emessi dalla sorgente, si andrà a raggiungere il limite dell'entropia.

Capitolo 4

Problemi della Codifica di Huffman e Arithmetic Coding

La codifica di Huffman, per quanto ottimale dal punto di vista della teoria dell'informazione, non va sempre bene. Tra le **problematiche** troviamo:

- Con le immagini a due toni di grigio $(0, 1)$ avremo che ogni pixel dovrà essere modificato con almeno un bit indipendentemente dalla probabilità dello 0 e dell' 1 nell'immagine binaria, quindi non si otterrà mai compressione.
- Per come è strutturata la codifica, avremo che **ogni simbolo verrà codificato in una stringa di bit di lunghezza intera**, quindi servirà **almeno un bit per codificare**. Questo approccio porta ad uno **spreco di spazio** che diventa significativo in base a casi particolari :
  - per un carattere con probabilità pari ad $1/3$ avremo che il numero ottimale di bit per la codifica sarà 1.6 bit, comportando quindi ad un coding ambiguo e non ottimale sulla quantità di bit da assegnare:
    - assegnando due bit ci sarà una perdita in termini di spazio,
    - assegnando un singolo bit la perdita ci sarà su altri simboli.
  - tale problema diventa più evidente quando determinati caratteri hanno probabilità molto alte. Se un carattere presenta il 90% di probabilità, la codifica ottimale arriva a 0.15 bit ed assegnando un bit per tale carattere andremmo ad impiegare uno spazio grande 6 volte di quello richiesto.

Con la **Codifica Aritmetica** non si usa un albero di codifica, bensì **un singolo numero reale compreso tra** 0 **e** 1. Tale algoritmo ha come idea di base la conoscenza delle probabilità $p_1, ..., p_{|S|}$ degli elementi della sorgente S. Ogni elemento di S risiederà **nell'intervallo di numeri reali** $[0, 1)$ in base alla propria probabilità.

4.1 Esempio

Consideriamo il messaggio $M = \alpha_1\alpha_2\alpha_3\alpha_3\alpha_4$ dato in output dalla sorgente con la seguente probabilità per ogni simbolo:

$$\alpha_1 = 0.2 \quad \alpha_2 = 0.2$$
$$\alpha_3 = 0.4 \quad \alpha_4 = 0.2$$

Dividiamo l'intervallo $[0, 1)$ nei seguenti sotto-intervalli di codifica:

$$\alpha_1 : [0.0, 0.2) \quad \alpha_2 : [0.2, 0.4)$$
$$\alpha_3 : [0.4, 0.8) \quad \alpha_4 : [0.8, 1.0)$$

Ogni simbolo comprenderà il proprio *low* e non il proprio *high*. Un messaggio decodificato compreso tra $[0.0, 0.2)$ rappresenterà α_1.

4.2 Codifica con Arithmetic Coding

L'output del **Coding Aritmetico** è un singolo numero reale compreso tra 0 e 1 (con 1 escluso) che può essere **decodificato in maniera univoca** per ricreare l'esatta stringa di simboli in input. Prima di procedere con la codifica, ad ogni simbolo dobbiamo assegnare la probabilità.

4.2.1 Esempio: Codifica del messaggio "BILL GATES"

1. In primis assegniamo per ogni carattere la probabilità di apparire nel messaggio:

Character	Probability
SPACE	1/10
A	1/10
B	1/10
E	1/10
G	1/10
I	1/10
L	2/10
S	1/10
T	1/10

2. Dividiamo il sotto-intervallo $[0, 1)$ tra tutti i simboli proporzionalmente alla probabilità di ogni simbolo:

Character	Probability	Range
SPACE	1/10	$0.00 \leq r < 0.1$
A	1/10	$0.10 \leq r < 0.2$
B	1/10	$0.20 \leq r < 0.3$
E	1/10	$0.30 \leq r < 0.4$
G	1/10	$0.40 \leq r < 0.5$
I	1/10	$0.50 \leq r < 0.6$
L	2/10	$0.60 \leq r < 0.8$
S	1/10	$0.80 \leq r < 0.9$
T	1/10	$0.90 \leq r < 1$

Ad ogni carattere assegniamo una porzione del range da 0 a 1; ogni carattere possiede il range prestabilito, tranne il valore dell'high.

3. La porzione più significativa del numero dato in output dalla codifica aritmetica è data dai primi simboli:

   - Visto che la prima lettera in "**Bill Gates**" è la **B**, allora il messaggio codificato, per essere decodificato in maniera corretta, deve iniziare con un numero compreso nell'intervallo $[0.20, 0.3)$.
   - Per codificare tutte i simboli rimanenti, andiamo a suddividere l'intervallo appena ottenuto dalla B in tanti intervalli, sempre in base alla probabilità di tutti i simboli.
   - Ad esempio, visto che la lettera I detiene il range $[0.50, 0.6)$, allora nel nuovo sotto-intervallo occuperà tutti i numeri compresi nell'intervallo $[0.25, 0.26)$ e così via.

4.2.2 Algoritmo di codifica

L'algoritmo per applicare tale logica è il seguente:

```
low = 0.0;
high = 1.0;
while ( ( c = getc( input ) ) != EOF ) {
    range = high - low;
    high = low + range * high_range( c );
    low = low + range * low_range( c );
}
output ( low );
```

Avremo che la codifica finale sarà:

New Character	Low value	High Value
	0.0	1.0
B	0.2	0.3
I	0.25	0.26
L	0.256	0.258
L	0.2572	0.2576
SPACE	0.25720	0.25724
G	0.257216	0.257220
A	0.2572164	0.2572168
T	0.25721676	0.2572168
E	0.257216772	0.257216776
S	0.2572167752	0.2572167756

L'output dell'algoritmo darà per il messaggio "**Bill Gates**" un numero compreso tra 0.2572167752 e 0.2572167756 che lo codificherà in maniera univoca.

4.3 Decodifica con Arithmetic Coding

Il processo di decodifica si basa sul trovare il primo simbolo del messaggio andando a controllare in quale range risiede il primo digit del numero in output:

- Il numero in output risiede nel range $[0.2572167752, 0.2572167756)$, che sicuramente è compreso tra $[0.2, 0.3)$, quindi il primo digit rappresenta una **B**.
- A questo punto andiamo a **togliere il *contributo della B***:
  1. Rimuovendo il *low value* della B dal numero, in questo caso 0.2, ottenendo 0.0572167752.
  2. Dividendo il numero per la larghezza del range di B, in questo caso $0.3 - 0.2 = 0.1$, ottenendo alla fine 0.572167752.

Andiamo poi a ripetere questo ragionamento per tutti gli altri valori del numero dato in output dal passaggio precedente.

4.3.1 Algoritmo di decodifica

```
number = input_code();
for ( ; ; ) {
    symbol = find_symbol_straddling_this_range( number );
    putc( symbol );
    range = high_range( symbol ) - low_range( symbol );
    number = number - low_range( symbol );
    number = number / range;
}
```

Tale codice non presenta una condizione di fine, ma si può rimediare o andando a introdurre controlli per un carattere EOF oppure sapendo a priori quante iterazioni devono essere eseguite per la codifica (bisogna però sapere a priori quanto il messaggio sia lungo).
Il processo di decodifica sarà come segue:

Encoded Number	Output Symbol	Low	High	Range
0.2572167752	B	0.2	0.3	0.1
0.572167752	I	0.5	0.6	0.1
0.72167752	L	0.6	0.8	0.2
0.6083876	L	0.6	0.8	0.2
0.041938	SPACE	0.0	.1	0.1
0.41938	G	0.4	0.5	0.1
0.1938	A	0.2	0.3	0.1
0.938	T	0.9	1.0	0.1
0.38	E	0.3	0.4	0.1
0.8	S	0.8	0.9	0.1
0.0				

Avremo che:

- Il processo di **codifica** è un **processo di restringimento del range** ogni volta che incontriamo un nuovo simbolo in maniera proporzionale alla probabilità predefinita associata al simbolo.
- Il processo di **decodifica** è la procedura inversa di **espansione del range** in base alla probabilità di ciascun simbolo.

Capitolo 5

Approccio Pratico Arithmetic Coding

Il **Coding Aritmetico** risulta essere **poco pratico utilizzando una matematica a virgola mobile**, che non è né richiesta né utile poiché le operazioni tra numeri reali vanno a complicare di molto i calcoli e le approssimazioni. La sua realizzazione migliore è tramite l'utilizzo di una **matematica intera a 16 bit oppure a 32 bit**, con la quale possiamo limitare la rappresentazione per i valori di *high* e *low* dei numeri, caricando soltanto la quantità di cifre che vogliamo per la rappresentazione (16 oppure 32), **lasciando gli altri digits impliciti**. Desideriamo inoltre che codificatore e decodificatore lavorino in **lock-step**, utilizzando uno schema di trasmissione incrementale che permetta al decodificatore di elaborare i simboli man mano che vengono prodotti dal codificatore, riducendo così il tempo complessivo, soprattutto nel caso di dati molto lunghi, anche di milioni di bit.

Poiché un carattere detiene il *low* ma non il *high* del range che occupa, ci sarà una nuova assegnazione di intervalli per adattare i calcoli alla matematica intera.

Esempio:
Se il carattere α occupa un range $[0.2, 0.3)$, avremo che:

1. I numeri $0.2000...$ e $0.3000...$ sono costituiti dalla parte intera 0, inutile per la decodifica, e dalla parte decimale, che viene presa in considerazione. Possiamo quindi scartare lo zero e pensare soltanto alla parte decimale $2000...$ e $3000....$

2. Il range va da $2000....$ fino a $2999...$, visto che il valore $3000...$ dell'high non è compreso:

   - *low* = $200...0(00000....)$
   - *high* = $299...9(99999....)$

In questo modo, l'intervallo $[0.2, 0.3)$ verrà visto come $[200...0, 299...9]$, considerando che dopo l'ultimo digit di *high* e *low* avremo come digit impliciti rispettivamente una stringa infinita di 9 e 0.

Considerando quindi il range iniziale di tutti i numeri nell'intervallo $[0.0, 1.0)$, il range iniziale per l'algoritmo di codifica sarà $[0000...0, 9999...9]$. Il nuovo algoritmo risulta essere:

```
low = 0000...0;
high = 9999...9;
while ( ( c = getc( input ) ) != EOF ) {
    range = high - low;
    high = low + range * high_range( c );
    low = low + range * low_range( c );
}
output ( low );
```

5.1 Codifica e Decodifica con Numeri Interi

Codificatore e decodificatore possono lavorare in **lock-step**:

- Ogni volta che **un numero viene codificato**, si effettua uno **shift a destra** di un digit ai valori di *high* e *low* per far entrare il valore codificato.
- Ogni volta che **un numero viene decodificato**, si effettua uno **shift a sinistra** di un digit ai valori dell'*high* (facendo entrare un 9) e del *low* (facendo entrare 0).

Esempio Codifica: Quando si codifica un nuovo simbolo, andiamo ad inserire il valore relativo al range del nuovo simbolo:

- *high* = 9999...9
- Codifica di B con range $[2000...0, 2999...9] \rightarrow new\_high = 2999...9$
- *low* = 0000...0
- Codifica di B con range $[2000...0, 2999...9] \rightarrow new\_low$

Esempio Decodifica:

- *high* = 2999...9 $\rightarrow new\_high = 9999...9$
- *low* = 2000...0 $\rightarrow new\_low = 0000...0$

Togliamo quindi il primo digit (2) per shiftare i valori di *high* e *low* a sinistra, facendo entrare i valori impliciti, tornando alla condizione di partenza del range $[0000...0, 9999...9]$.
Come effetto della decodifica, ***high* e *low* inizieranno ad avvicinarsi sempre di più**. Tuttavia, non sempre la cifra più significativa di *high* e *low* combacia subito, in tal caso continuiamo a codificare finché non coincidono, ripetendo i calcoli.

	High	Low	Range	Cumulative Output
Initial state	99999	00000	100000	
Encode B (0.2—0.3)	29999	20000		
Shift out 2	99999	00000	10000	.2
Encode I (0.5—0.6)	59999	50000		.2
Shift out 5	99999	00000	100000	.25
Encode L (0.6—0.8)	79999	60000	20000	.25
Encode L (0.6—0.8)	75999	72000		.25
Shift out 7	59999	20000	40000	.257
Encode SPACE (0.0—0.1)	23999	20000		.257
Shift out 2	39999	00000	40000	.2572
Encode G (0.4—0.5)	19999	16000		.2572
Shift out 1	99999	60000	40000	.25721
Encode A (0.1—0.2)	67999	64000		.25721
Shift out 6	79999	40000	40000	.257216
Encode T (0.9—1.0)	79999	76000		.257216
Shift out 7	99999	60000	40000	.2572167
Encode E (0.3—0.4)	75999	72000		.2572167
Shift out 7	59999	20000	40000	.25721677
Encode S (0.8—0.9)	55999	52000		.25721677
Shift out 5	59999	20000		.257216775
Shift out 2				.2572167752
Shift out 0				.25721677520

Nel caso della codifica della prima L, vediamo che *high* e *low* hanno valori 79999 e 60000, quindi le cifre più significative non combaciano. Procediamo quindi con la codifica della successiva L, e tramite i calcoli vedremo che le cifre più significative coincidono, consentendoci di effettuare lo *shift out* del 7, che rappresenta il simbolo L.
Tuttavia, **ci possono essere complicazioni**, soprattutto in termini di precisione dei registri, che potrebbero portare a problemi.

- Se *high* e *low* si avvicinano troppo, come nel caso $h = 70004$ e $l = 69995$, lo "spazio" per la codifica inizia a diventare molto stretto.
- In casi estremi, con valori come $h = 70000$ e $l = 69999$, **il sistema potrebbe bloccarsi**, poiché non ci sarà più spazio per nuove codifiche né per decodificare i simboli già presenti.

5.2 Problemi di Underflow

Si possono prevenire problemi di **underflow** con controlli aggiuntivi. Se i digits più importanti di *high* e *low* sono distanti di uno, bisogna fare un secondo controllo:

- Se il secondo digit di *high* è 0 e il secondo digit di *low* è 9, questo potrebbe portare a un **underflow**.
- In questo caso, si effettua uno *shift* dei digits problematici:
  - Cancelliamo i secondi digit di *high* e *low*, facendo entrare un 9 per *high* e uno 0 per *low*.

Esempio:

	Before	**After**
High:	40344	43449
Low:	39810	38100
Underflow:	0	1

Tabella 5.1: High, Low, and Underflow values before and after

Il controllo va effettuato dopo ogni calcolo se i digits più significativi di *high* e *low* non combaciano. Se si verifica una situazione di **underflow**, shiftiamo i digits e **incrementiamo il contatore di *underflow***. Quando i digits coincidono, diamo in output il valore e i digits di *underflow* scartati in precedenza.

5.3 Considerazioni Teoriche

La codifica aritmetica può essere più efficiente della codifica Huffman, specialmente quando le probabilità dei simboli sono molto sbilanciate. Ad esempio, consideriamo la codifica del messaggio ”AAAAAAA”:

- La probabilità di A è pari a 0.9, quindi assegniamo ad A il range $[0.0, 0.9)$.
- Al simbolo di fine messaggio assegniamo invece il range $[0.9, 1)$.

New Character	Low value	High value
	0.0	1.0
A	0.0	0.9
A	0.0	0.81
A	0.0	0.729
A	0.0	0.6561
A	0.0	0.59049
A	0.0	0.531441
A	0.0	0.4782969
END	0.43046721	0.4782969

Per codificare tale messaggio, possiamo scegliere un numero nell’intervallo tra il *Low Value* e l’*High Value*, ad esempio 0.45, che occupa meno spazio rispetto ai 7 bit necessari in Huffman. Così, possiamo codificare 8 simboli in meno di 8 bit.

5.4 Limitazioni dell’Approccio Statistico

L’arithmetic coding si basa su distribuzioni di probabilità associate ai simboli. Tuttavia, può essere difficile conoscere tali probabilità a priori, e anche se si riesce a stimarle, queste potrebbero cambiare nel tempo.

Capitolo 6

Metodi di Compressione via Dizionario (Textual Substitution)

Con tali metodi non si utilizza la probabilità come informazione di partenza, a causa dei limiti dell'approccio statistico. Con gli algoritmi di compressione via dizionario, non si considera esplicitamente la probabilità dei simboli in output, ma **si stima il comportamento della sorgente** basandosi su ciò che si è osservato in passato. Questa famiglia di algoritmi non codifica simboli in stringhe di lunghezza variabile, bensì in **token singoli** di taglia proporzionale alla grandezza del dizionario, di $\log_2\lceil D \rceil$. Il token forma **un indice all'interno di un dizionario** e, se la lunghezza del token è inferiore al messaggio da codificare, la compressione diventa efficace.

6.1 Algoritmo di Codifica On-line

Un algoritmo di codifica che legge uno stream di caratteri su Σ e scrive uno stream di bit è definito **on-line** se, man mano che il testo appare, questo viene codificato. Lavorando online non si ha la visione di tutto ciò che si deve comprimere, quindi non si può costruire a priori un dizionario da inviare al decodificatore, che dovrà costruirsi un dizionario *on-the-fly*.

L'algoritmo generico di codifica via **textual substitution** è il seguente:

```
(1) Inizializzazione del dizionario D con un insieme INIT
(2) Repeat Forever
    a) Ottenimento del match corrente:
       1. t := MH(inputstream)
       2. Avanza nell'inputstream di t caratteri
       3. Trasmetti i log2D bits che corrispondono a t
    b) Aggiornamento del dizionario locale D:
       1. X := UH(D)
       2. while X != {} and (D non è pieno or DH(D)!= {}) do begin:
           - Cancella un elemento x in X
           - if x not in D then begin:
               - if D è pieno then cancella DH(D) da D
               - Aggiungi x a D
           - end
       - end
```

L'algoritmo generico di decodifica è il seguente:

```
(1) Inizializzazione del dizionario D eseguendo lo step (1)
    dell'algoritmo di codifica
(2) Repeat Forever
    a) Prendi il match corrente:
       1. Ricevi i log2D bits
       2. Risali al match t dal dizionario di lookup
       3. Dai in output il carattere t
    b) Aggiornamento del dizionario D eseguendo lo step (2)
       dell'algoritmo di codifica
```


6.2 Le Euristiche del Dizionario

Le **euristiche** che dobbiamo specificare per definire con quali algoritmi vogliamo lavorare sono:

- **Initialization Heuristic** $INIT$: insieme di stringhe che devono essere utilizzate per **inizializzare il dizionario locale** D. D è inizializzato per contenere almeno i caratteri di Σ, che non possono essere cancellati. Avremo che Σ deve essere un sottoinsieme di $INIT$ e che $|INIT| \leq D$.
- **Matching Heuristic** MH: una funzione che **rimuove dall'input stream una stringa** t **già presente nel dizionario** D. Finché D contiene Σ, la MH è sempre ben definita poiché possiamo garantire un match con qualsiasi input string.

- **Update Heuristic** UH: una funzione che prende il dizionario locale D e **restituisce un insieme di stringhe che dovrebbero essere aggiunte al dizionario**, se presente dello spazio.
- **Deletion Heuristic** DH: una funzione che prende in input il dizionario locale D e **può restituire**:
  - **un insieme vuoto**, in caso non ci siano stringhe da cancellare;
  - **una singola stringa** str appartenente a D che non fa parte di $INIT$ e **che può essere eliminata da** D.

Per far sì che il procedimento di codifica-decodifica sia corretto, i **dizionari locali di codificatore e decodificatore devono rimanere identici**. Altrimenti, per un determinato token in output dal codificatore, il decodificatore darà in output una stringa errata.

6.3 Tipi di Algoritmi

Tra i tipi di algoritmi di **textual substitution** possiamo trovare:

- **Dizionario Statico**: una volta inizializzato, si saltano i passaggi di UH e DH.
- **Dizionario su finestra scorrevole (sliding window)**: costruito su una porzione di testo visto precedentemente (ad esempio l'algoritmo **LZ-77**).
- **Dizionario Dinamico (o dizionario on-the-fly)**: si basa sull'utilizzo di una struttura dati esterna sia dal compressore che dal decompressore che viene costruito da zero mentre i dati vengono processati(esempi: **LZ-78** e **LZW**).

6.4 Dizionario Statico

Nel dizionario statico l'*update* risulta essere nullo, $DH = \{\}$, e non si aggiunge né si cancella nulla. La compressione avviene soltanto in base a $INIT$, che può contenere delle sottostringhe comuni. Tale approccio è utile se si conosce a priori il comportamento della sorgente, offrendo:

- velocità e semplicità;
- resistenza agli errori su un canale rumoroso (i dizionari di codificatore e decodificatore non potranno mai differire);
- possibilità di utilizzare indici di lunghezza variabile in base alla probabilità di apparizione delle stringhe nel dizionario.

6.5 Dizionario a Finestra Scorrevole

L'approccio del **Dizionario Statico** viene raramente utilizzato, se non in casi molto specifici. Più comune e pratico è l'utilizzo del **Dizionario a Finestra Scorrevole** (*Sliding Window Dictionary*), particolarmente efficace nei metodi di compressione dei dati come LZ77.

All'inizio dell'algoritmo, il dizionario ($INIT$) è inizializzato con l'intero alfabeto Σ per garantire che ogni simbolo possa essere almeno rappresentato singolarmente. Questo dizionario viene poi aggiornato dinamicamente durante la compressione tramite due parametri fondamentali:

- $maxDisplacement$: definisce quanto può andare indietro la finestra per cercare delle corrispondenze;
- $maxLength$: indica la lunghezza massima delle sottostringhe che si possono confrontare.

Il dizionario D conterrà quindi **tutte le sottostringhe di lunghezza** $maxLength$ che si trovano nei **precedenti** $maxDisplacement$ caratteri dello stream di input. In questo modo, il dizionario locale è composto da:

- **Tutte le lettere dell'alfabeto**, per garantire che ogni simbolo possa essere compresso;
- **Le sottostringhe dei** $maxDisplacement$ **caratteri precedenti**, permettendo di trovare corrispondenze con sequenze già viste.

6.5.1 Puntatori (m, n)

Per rappresentare le corrispondenze nel testo, si usano i **puntatori**, coppie di interi (m, n), dove:

- m **(displacement back)** indica di quante posizioni bisogna tornare indietro rispetto alla posizione corrente per trovare il match;
- n rappresenta la **lunghezza della sequenza** che corrisponde.

In pratica, questa coppia indica *"torna indietro di m caratteri e copia i successivi n caratteri"*.

6.5.2 Aggiornamento del Dizionario

Quando viene trovato un match di lunghezza $|t|$, il dizionario si aggiorna **spostando la finestra scorrevole verso destra di $|t|$ posizioni**. Questo aggiornamento si può descrivere tramite:

- $UH(D)$: l'insieme delle sottostringhe nella finestra che si sovrappongono con la sequenza t;
- $DH(D)$: l'insieme delle sottostringhe che si sovrappongono con gli ultimi $|t|$ caratteri a sinistra della finestra.

In questo modo, il dizionario si mantiene aggiornato includendo le nuove sequenze e scartando le più vecchie.

6.5.3 Struttura della Sliding Window

La **Sliding Window** è divisa in due parti:

- **A sinistra**: i precedenti $maxDisplacement$ caratteri già compressi;
- **A destra**: il *look-ahead buffer*, che contiene i prossimi caratteri da comprimere (fino a un massimo di $maxLength$).

Quando si trova un match, la finestra si sposta di $|t|$ posizioni verso destra, facendo spazio a nuovi caratteri nel *look-ahead buffer*.

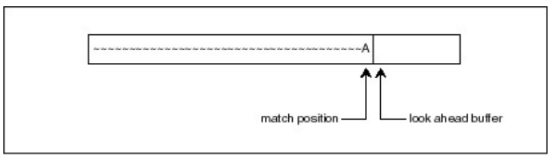


Figura 6.1: Caption

6.5.4 Problema della Sliding Window

L'idea alla base di questo metodo è che **le sottostringhe più vicine alla posizione corrente abbiano maggiori probabilità di produrre match lunghi**, rispetto a quelle più lontane (che vengono scartate man mano che la finestra si sposta).

Tuttavia, questa ipotesi non è sempre vera. In alcuni casi, le sequenze scartate potrebbero offrire match migliori, portando a una compressione inefficace. Questo limite ha spinto allo sviluppo di algoritmi di compressione più sofisticati, che gestiscono meglio i casi in cui il semplice scorrimento della finestra non è sufficiente per ottenere buoni risultati.

6.5.5 Problema della Sliding Window - 2

La speranza con questo metodo è che **le sottostringhe più vicine alla posizione corrente permettano di fare match più lunghi rispetto a quelle che stiamo scartando** con lo scorrimento della finestra. Tuttavia, questa ipotesi non è sempre rispettata e la compressione può risultare inefficace. Questi problemi hanno portato all'introduzione di metodi di codifica a **Dizionario Dinamico**.

6.6 Dizionario Dinamico

Nei metodi di codifica a **Dizionario Dinamico**, il dizionario non è fisso ma si aggiorna continuamente in base ai dati processati. Il dizionario è mantenuto come una struttura dati condivisa tra compressore e decompressore, garantendo coerenza nella codifica e decodifica.
L'aggiornamento del dizionario UH avviene concatenando il **match precedente** (pm) con alcune stringhe derivate dal **match corrente** (cm). Questo aggiornamento si basa su una funzione di incremento, INC, che mappa un insieme di stringhe a una nuova stringa o a un insieme di stringhe.
Per una data scelta di INC, l'aggiornamento del dizionario può essere descritto come:

- **UH(D) = { pm concatenato con tutte le stringhe di INC(cm) }.**

Le possibili scelte per la funzione INC includono:

- **First Character Heuristic (FC):** $INC(cm)$ rappresenta solo il primo carattere del cm.
- **Identity Heuristic (ID):** $INC(cm)$ è semplicemente cm stesso.
- **All-Prefixes Heuristic (AP):** $INC(cm)$ include tutti i prefissi non vuoti di cm, incluso cm stesso.

Esempio:
Supponiamo $pm = "THE\_"$ e $cm = "CAT"$ (dove il carattere "_" indica uno spazio). A seconda della scelta di INC, l'aggiornamento del dizionario UH sarà:

- **FC**: $UH = \{"THE\_C"\}$. In questo caso, viene aggiunto solo il primo carattere del cm.
- **ID**: $UH = \{"THE\_CAT"\}$. In questo caso, viene aggiunto il cm stesso.
- **AP**: $UH = \{"THE\_C", "THE\_CA", "THE\_CAT"\}$. In questo caso, vengono aggiunti tutti i prefissi di cm.

Il dizionario dinamico permette di adattarsi ai dati in tempo reale, migliorando l'efficienza della compressione rispetto al dizionario statico, soprattutto in contesti in cui la sorgente non è prevedibile.

6.7 Euristiche di Deletion

Tra le scelte di DH possiamo trovare:

- **Freeze Heuristic (**$FREEZE$**)** : $DH(D)$ **è la stringa vuota**. Una volta che il dizionario è pieno, non vengono aggiunte né rimosse stringhe.
- **Least Recently Used Heuristic (**LRU**)** : $DH(D)$ è **la stringa in** D **che ha avuto il match meno recentemente**. Esiste una variante di LRU, chiamata LFU (**Least Frequently Used Heuristic**), in cui $DH(D)$ è la stringa di D utilizzata meno frequentemente per il match.
- **Swap Heuristic (**$SWAP$**)** : vengono mantenuti **due dizionari distinti**, uno primario e uno ausiliario. Quando il **dizionario primario si riempie**, esso **viene congelato** ($FREEZE$), e viene costruito il dizionario ausiliario nello stesso modo del primario. La compressione continua secondo il dizionario primario. Quando anche il dizionario ausiliario si riempie, i ruoli dei due dizionari vengono invertiti: il nuovo dizionario ausiliario viene svuotato, e la codifica prosegue.

6.8 Algoritmo LZ-78

L'algoritmo **LZ-78** si differenzia da altri metodi di compressione perché **non utilizza una *sliding window***, ma impiega un **dizionario esplicito** che cresce dinamicamente durante la compressione. Questo dizionario viene inizializzato con $INIT = \epsilon$ (la stringa vuota) e memorizza progressivamente tutte le stringhe incontrate nel testo. La dimensione del dizionario è limitata solo dalla quantità di memoria disponibile.

6.8.1 Funzionamento del Codificatore

Il codificatore produce in **output una coppia** (x, y), dove:

- x è il **puntatore al dizionario**, che indica la posizione della stringa più lunga trovata nel dizionario;
- y è l'***innovation***, ovvero il simbolo che interrompe il match in corrispondenza dell'indice x.
  - Non è necessario specificare la lunghezza del match, poiché questa informazione può essere recuperata consultando il dizionario.

Questa coppia (x, y) viene poi aggiunta come nuovo elemento del dizionario, sia dal codificatore sia dal decodificatore. Il dizionario parte quindi da ϵ e cresce progressivamente includendo nuovi match:

- All'inizio, i caratteri singoli x dello stream di input vengono codificati come $(0, x)$.
  - 0 indica che non è stato trovato alcun match nel dizionario, quindi si parte dalla stringa vuota ϵ;
  - x rappresenta l'*innovation*, poiché $\epsilon * x$ diventa il nuovo elemento del dizionario.
- Con l'espansione del dizionario, supponiamo che la stringa "x" sia all'indice 1. Quando si incontra la stringa "xy":
  - Il match più lungo sarà in posizione 1;
  - L'*innovation* sarà "y";
  - La coppia prodotta sarà $(1, y)$.

6.8.2 Esempio di Codifica

Input text: "DAD DADA DADDY DADO..."

Output Phrase	Output Character	Encoded String
0	'D'	"D"
0	'A'	"A"
1	' '	"D "
1	'A'	"DA"
4	' '	"DA "
4	'D'	"DAD"
1	'Y'	"DY"
0	' '	" "
6	'O'	"DADO"

Il numero di bit b assegnato alla codifica del dizionario determina il numero massimo di frasi memorizzabili, pari a 2^b. Le frasi sono organizzate in una struttura ad albero, in cui:

- La **radice** 0 rappresenta la **stringa vuota** ϵ;
- Ogni **carattere** è collegato tramite un **ramo**;
- Si **scende** lungo i rami per trovare il **match più lungo possibile** presente nel dizionario.

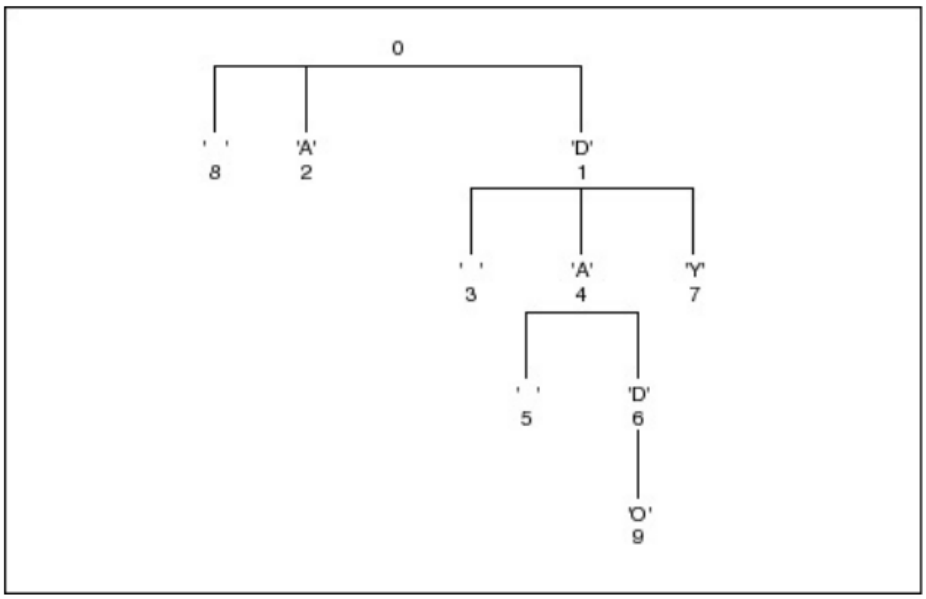


Figure 9.1 An LZ78 Dictionary Tree.

6.8.3 Gestione del Dizionario

Poiché ogni nodo può avere fino a 2^b rami, la gestione della memoria per il dizionario richiede delle strategie ottimizzate:

- Una soluzione semplice è usare un **array** in cui ogni elemento può avere fino a 2^b figli. Tuttavia, molti nodi potrebbero non avere così tanti discendenti, causando un notevole spreco di memoria.
- Un'alternativa più efficiente è una **lista linkata** per ogni nodo, contenente solo i figli effettivamente esistenti. Questo approccio risparmia memoria, ma rende la ricerca più lenta.

6.8.4 Sincronizzazione tra Codificatore e Decodificatore

Un aspetto critico dell'algoritmo è che **codificatore e decodificatore devono mantenere lo stesso dizionario**. Questo implica che **un singolo errore di trasmissione** in un canale rumoroso può causare la **propagazione dell'errore**, portando a divergenze crescenti tra i due dizionari.

6.8.5 Gestione del Dizionario Pieno

Quando il dizionario raggiunge la dimensione massima di 2^b frasi, è necessario adottare una strategia per gestirlo. Le soluzioni più comuni includono:

- **Bloccare l'inserimento di nuove frasi** e continuare a comprimere utilizzando solo le frasi già presenti. Questa strategia è semplice ma può risultare inefficace, specialmente con dati che cambiano nel tempo.

- Un approccio più sofisticato consiste nel **monitorare il rapporto di compressione**. Se questo rapporto peggiora, il dizionario viene **azzerato e ricostruito da zero**. Se invece il rapporto rimane stabile, il dizionario continua a essere utilizzato senza aggiungere nuove frasi.

Un esempio pratico di questa problematica si osserva nella compressione di un file binario (come un file `.exe`): le caratteristiche statistiche dei dati possono cambiare drasticamente tra la *call section* e la *data section*. In questo caso, un dizionario creato per la *call section* potrebbe risultare inefficace per comprimere la *data section*.

6.9 LZW

LZW è una variante dell'algoritmo $LZ-78$, implementata in modo che:

- I dizionari del codificatore e del decodificatore vengono **precaricati con** Σ, ovvero tutti i simboli dell'alfabeto sorgente. Di conseguenza, non esiste più il match della stringa nulla ϵ, poiché è garantita la presenza di un match di lunghezza almeno 1.
- **L'invio dell'*innovation* al decodificatore viene eliminato**. Il decodificatore potrà recuperarla soltanto come primo carattere del match nel passo successivo. **I dizionari non lavorano più in lock-step stretto, ma sono sfasati di un passo**: se il codificatore è all'istante t, il decodificatore si troverà all'istante $t-1$. Di conseguenza, ciò che è presente nel dizionario del compressore al tempo $t-1$ sarà presente nel dizionario del decompressore soltanto nel passo successivo, ovvero al tempo t.

Esempio

Supponiamo che il compressore al tempo t trovi la stringa data dal match più l'*innovation* "␣W". Sappiamo che:

- "␣" è presente nel dizionario, in quanto contiene tutti i caratteri dell'alfabeto sorgente, ma non la stringa "␣W".

Il compressore invia quindi al decompressore l'indice relativo a "␣" ed aggiunge al proprio dizionario la stringa "␣W". Tuttavia, il decompressore non può ancora aggiungere questa stringa, poiché non ha ancora ricevuto il carattere che interrompe il match. Supponiamo che, nel passo successivo, il compressore debba codificare la stringa "WE". Come per il ragionamento precedente, "W" è presente nel dizionario, ma non "WE". Pertanto, il compressore invia al decompressore l'indice relativo a "W", consentendo al decompressore di aggiungere al proprio dizionario la stringa "␣W". A questo punto, il compressore aggiunge alla propria struttura la codifica di "WE", che il decompressore potrà includere solo al passo successivo.

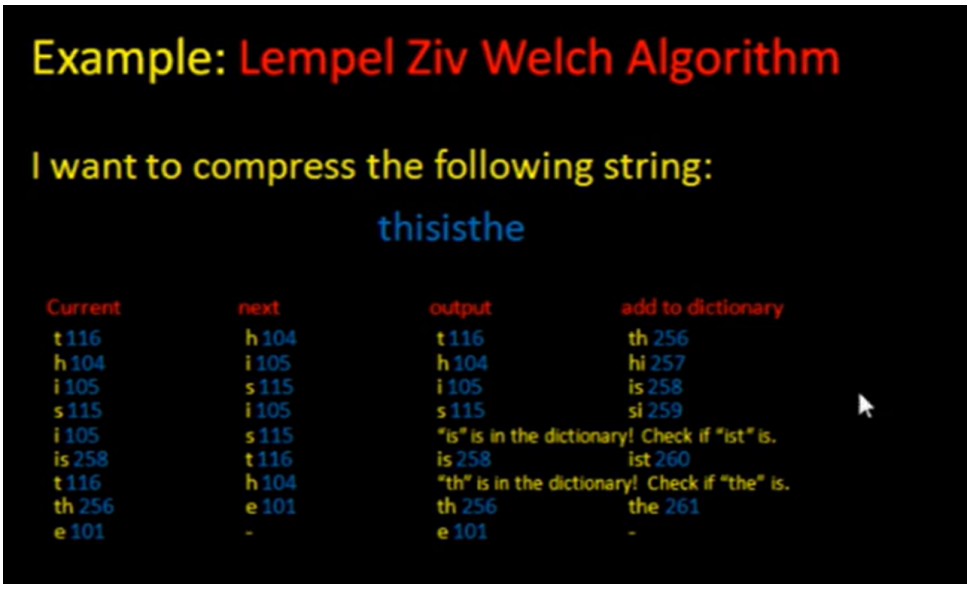
Poiché l'*innovation* non viene inviata al decompressore, **il compressore inizierà il prossimo match partendo dall'*innovation* stessa**, così da permettere che essa sia aggiunta al decompressore nel passo successivo, ricostruendo gradualmente lo stesso dizionario man mano che riceve i dati.

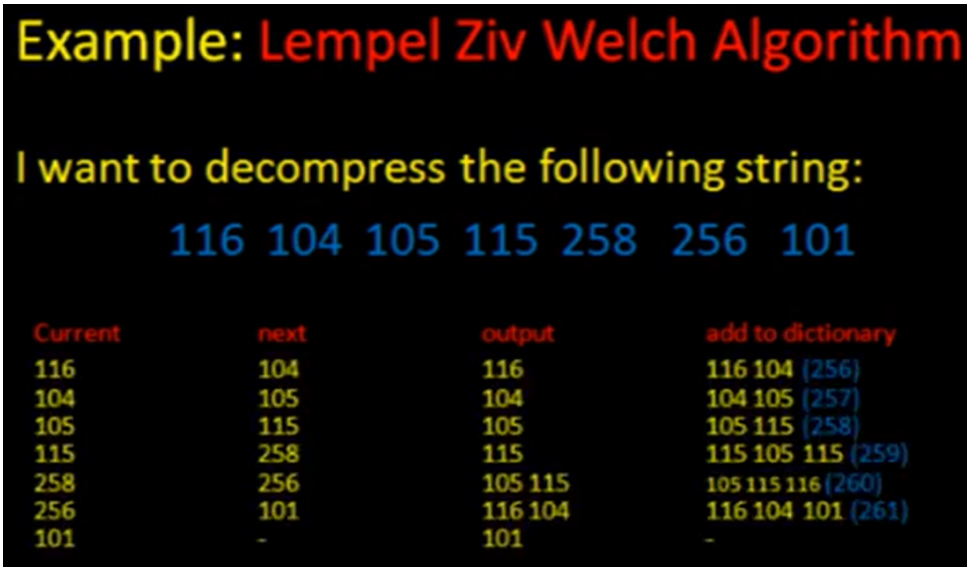
Questo processo si ripeterà per tutti i passaggi successivi, con il compressore che invia sempre indici relativi a match già incontrati, senza includere l'*innovation* che interrompe il match corrente.

Input String: " WED WE WEE WEB WET "

Characters Input	Code Output	New code value and associated string
" 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	

Un altro esempio di codifica - decodifica è il seguente:





6.10 Problema LZW

Lo "sfasamento" tra compressore e decompressore che caratterizza l'algoritmo LZW rappresenta anche il suo principale problema: può verificarsi il caso in cui il decompressore riceva un indice che non ha ancora salvato all'interno del proprio dizionario. Consideriamo il caso seguente:

- Al passo t, il compressore aggiunge una nuova stringa al proprio dizionario con un indice x, che il decompressore non potrà conoscere fino al passo $t + 1$.
- Al passo $t + 1$, il compressore invia proprio l'indice x.

Fortunatamente, questo problema può verificarsi solo quando, nell'input stream, si incontra la sequenza $ch\text{-}str\text{-}ch\text{-}str\text{-}ch$, in questo caso, **il compressore produce in output un valore prima che il decompressore sia in grado di definirlo**.
Poiché questo problema si presenta solo in un **caso specifico**, è possibile aggiungere **un'eccezione all'algoritmo** per gestirlo.

6.10.1 Esempio:

Input String: IWOMBAT.....IWOMBATIWOMBATIXXX

<Problem section>

Character Input	New code value and associated string	Code Output
...I		
WOMBATA	300 = IWOMBAT	288 (IWOMBA)
.	.	.
.	.	.
...I	.	.
WOMBATI	400 = IWOMBATI	300 (IWOMBAT)
WOMBATIX	401 = IWOMBATIX	400 (IWOMBATI)

In questo caso, quando inviamo la stringa $ch + str$, diamo in output il codice 300, che sarà noto sia al compressore sia al decompressore, anche se in momenti differenti. Supponiamo che nell'input stream si presenti la seguente sequenza:

- $IWOMBATI$: il compressore darà in output l'indice 300, che corrisponde a $IWOMBAT$, ed aggiungerà al proprio dizionario la stringa $IWOMBATI$ con codice $400 = (300 + I)$.
- $IWOMBATIX$: il compressore darà in output l'indice 400 (che rappresenta $IWOMBATI$) al decompressore ed aggiungerà al proprio dizionario la stringa $IWOMBATIX$ con codice $401 = (400 + X)$.

Il problema, però, è che il decodificatore non sa ancora a cosa corrisponda l'indice 400.
Poiché tale situazione può verificarsi solo in questo specifico caso, possiamo gestirla con una soluzione ad hoc:

- Sappiamo che al codice 300 corrisponde la stringa $ch - str$.
- Il decodificatore, vedendo che l'indice 400 non è ancora presente nel proprio dizionario, utilizza un controllo di eccezione per determinare che l'indice 400 corrisponderà esattamente a $ch - str - ch$, ottenuto apponendo ch al valore associato all'indice 300.

Capitolo 7

Compressione Lossless di Immagini

Le immagini possono essere viste come array bidimensionali di dati interi (detti campioni) che rappresentano una certa precisione. In base alla precisione, troviamo diverse tipologie di immagini, come quelle in bianco e nero, a livelli di grigio (con numero di bit variabile) e a colori.
La **compressione lossless** viene utilizzata per immagini destinate a ulteriori analisi o estrazioni di informazioni, mentre non è adatta per immagini fotografiche, dove l'end-user (l'occhio umano) non percepisce la differenza marginale nei pixel derivante da una compressione lossy.
Tra i parametri per la compressione delle immagini, troviamo:

- Nel caso di compressione lossless si considera soltanto il rapporto di compressione, dato che l'immagine decompressa sarà identica all'originale.
- Nel caso di compressione lossy, invece, sono rilevanti sia il rapporto di compressione sia la qualità dell'immagine decompressa.

Le immagini destinate alla compressione sono tipicamente **immagini a toni continui**, che hanno più di un bit per campione (altrimenti la compressione non sarebbe possibile).
Gli schemi che eseguono la compressione lossless svolgono due fasi, secondo un paradigma noto come **Modeling and Coding.**

7.1 Modeling and Coding

Nella fase di **Modeling**, i dati dell'immagine e la ridondanza vengono analizzati campione per campione secondo un ordine predefinito (ad esempio tramite ***raster scan***, ovvero riga per riga) e rappresentati in un **modello probabilistico** utilizzato successivamente per la fase di **Coding**. Il **Modeling** si può considerare un **problema di inferenza induttiva**, in cui a ogni istante t, dopo aver osservato e inviato i dati $x^t = x_1 x_2 ... x_t$ al compressore, **si cerca di prevedere il valore del prossimo campione da codificare** x_{t+1}, assegnando una distribuzione di probabilità condizionata su x_t. La quantità di dati che contribuisce alla previsione di x_{t+1} è $-\log P(x_{t+1}|x^t)$, che in

media è pari all'entropia di un modello probabilistico.

In questa fase vengono utilizzate anche **trasformate**, come la **Trasformata Discreta del Coseno (**DCT**)**, che permettono di rappresentare i dati in un dominio differente, dove risultano più facili da modellare senza introdurre errori.

Se il modello è adattivo (**modello on-line**), utilizza i **valori dei campioni precedenti** per prevedere il nuovo valore del pixel da codificare (ad esempio, se tutti i pixel precedenti sono neri, probabilmente anche il prossimo sarà nero). Il decodificatore utilizza lo stesso modello del codificatore, avendo accesso agli stessi pixel una volta ricevuti, e può così prevedere il prossimo pixel.

Con un modello non adattivo (**modello off-line**), invece, **si costruisce un modello a priori**, che viene inviato al decompressore. Questo modello è più preciso, ma implica un costo aggiuntivo per la sua trasmissione.
La fase di **Modeling** si può suddividere in tre sottofasi:

1. **Predizione:** il valore x_{t+1} da codificare viene predetto con $\hat{x}_{t+1}$ basato su un sottoinsieme finito (un ***casual template***) dei dati già codificati. Si utilizza un sottoinsieme finito per evitare un costo computazionale elevato dovuto a un'analisi troppo estesa.
2. **Definizione del contesto:** si stabilisce il contesto in cui x_{t+1} si trova, **funzione di un *casual template* differente**. Ad esempio, il contesto può essere definito considerando tutti i pixel intorno al pixel da codificare, adottando una strategia opportuna.
3. **Coding dell'errore di predizione** $\epsilon_{t+1} \triangleq x_{t+1} - \hat{x}_{t+1}$**:** questo errore viene codificato condizionatamente al contesto di x_{t+1}. Per modellare la distribuzione di probabilità dell'errore di predizione, si utilizzano solo le informazioni rilevanti per quel campione specifico.

Nella fase di **Coding**, il modello probabilistico viene usato per codificare ϵ_{t+1} tramite algoritmi come l'arithmetic coding o il coding di Huffman. Predizione, contesto e distribuzione di probabilità sono disponibili per il decodificatore, che riceve l'errore di predizione e, recuperando la distribuzione di probabilità relativa al contesto, può risalire al valore originale del campione.
La codifica dell'errore di predizione è vantaggiosa poiché una buona fase di **Modeling** può decorrelare l'output dal campione originale, trasformando una sorgente di primo ordine (quella dei campioni) in una sorgente di ordine superiore. Questo consente una codifica più efficiente e una maggiore compressione.
Nella progettazione di un algoritmo di compressione lossless di immagini, si cerca di avvicinarsi ai limiti teorici della compressione, il che dipende fortemente dalla scelta del contesto:

- Più piccolo è il contesto, maggiore è l'entropia.

- Più grande è il contesto, minore sarà l'entropia.

Sebbene l'uso di contesti molto ampi non presenti problemi teorici, nella pratica si incontra il problema del **context dilution**: un contesto è così raro da non fornire statistiche accurate per costruire una distribuzione di probabilità affidabile. Questo fenomeno sottolinea l'importanza di una buona conoscenza dei dati per costruire modelli adeguati, ottenendo così algoritmi specifici che offrono predizioni e compressioni migliori, sebbene a scapito della riutilizzabilità (trade-off tra ambito e efficienza).

7.2 Lossless JPEG

Lossless JPEG è un **Open Standard**, il che significa che non descrive specificamente un algoritmo ma solo come devono essere i dati in ingresso e in uscita dal sistema, rispettando determinate fasi implementabili in modi differenti.

Lo standard JPEG presenta quattro metodi di compressione di base:

- I primi tre metodi (**sequential encoding, progressive encoding, hierarchical encoding**) utilizzano la DCT e sono usati per la compressione **lossy**. La DCT consente di passare dal dominio spaziale a quello delle frequenze, rendendo più semplice individuare le frequenze a cui l'occhio umano è meno sensibile, per poi filtrare tali componenti.
- Il quarto metodo, **Lossless JPEG**, si basa in parte su un algoritmo di compressione **lossless** chiamato **Sunset**. Tuttavia, quest'ultimo è poco implementato perché offre bassi rapporti di compressione, tipicamente attorno a 2 o anche inferiori, risultando inferiore a molti altri algoritmi di compressione già disponibili all'epoca.

Lossless JPEG opera trovando una previsione del pixel X tramite un **casual template** dei pixel vicini P, Q, R già codificati. Sono presenti otto metodi di predizione per X e, in base al metodo scelto, si ottiene un valore predetto $\hat{X}$. L'errore di predizione ϵ viene quindi inviato al decodificatore. Essendo ϵ generalmente più piccolo di X, verrà codificato tramite Huffman o la codifica aritmetica, in base al tipo di compressione JPEG utilizzato.

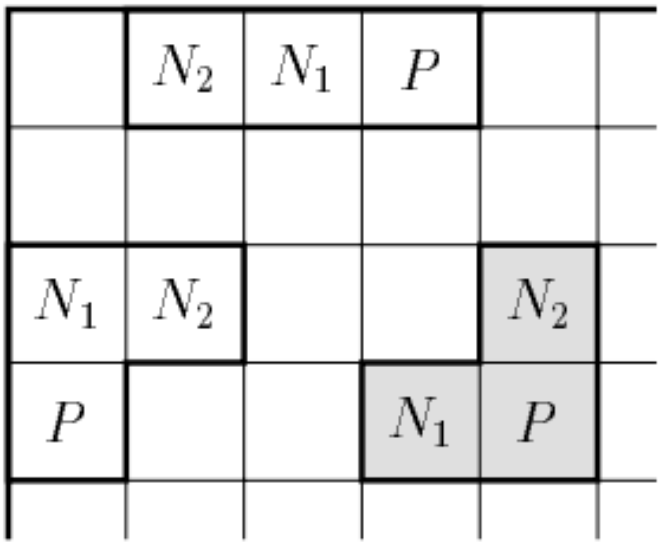
Prediction selection value	Predictions
0	NO prediction
1	P
2	Q
3	R
4	P+(Q-R)/2
5	R+(P-Q)/2
6	P+Q-R
7	(P+Q)/2

7.3 FELIX

Lossless JPEG è stato immediatamente eclissato da algoritmi che offrivano rapporti di compressione maggiori; uno di questi è **FELIX**, usato per la compressione lossless di immagini. Tale algoritmo non utilizza la codifica aritmetica, ma cerca di mantenere una complessità computazionale bassa, utilizzando **Adjusted Binar Codes** e la codifica di Golomb. Di conseguenza, FELIX risulta essere:

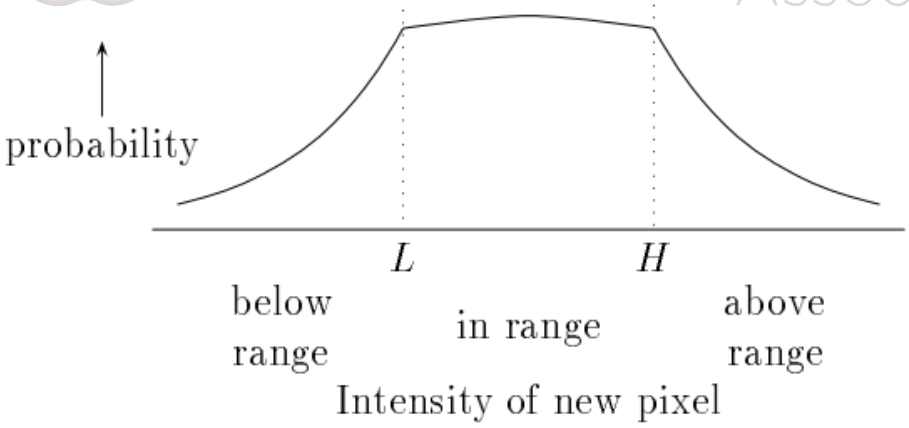
- computazionalmente superiore a Lossless JPEG
- in termini di performance uguale a Lossless JPEG, se non migliore

Il problema di **FELIX** è che utilizza almeno un bit per campione, rendendolo meno efficace su immagini altamente comprimibili.
L'idea alla base di FELIX è di avere un campione P e codificarlo utilizzando due vicini A e B di valori già noti al decompressore.



Considerando L e H rispettivamente come il più piccolo e il più grande tra N_1 e N_2, l'intensità di P può rientrare in tre intervalli:

- P risiede a sinistra di L, quindi $P < L$
- P risiede tra L e H, quindi $L \leq P \leq H$
- P risiede a destra di H, quindi $H < P$



La codifica di P sarà un codice a lunghezza variabile, in base all'intervallo in cui l'intensità di P si trova:

- Se P è compreso tra L e H, supponiamo che la distribuzione di probabilità sia quasi uniforme con un leggero picco al centro (quindi è più probabile che il valore sia la media tra L e H). Tale caso si verifica circa metà delle volte, e la codifica di P inizierà con un

bit 0. Per la codifica binaria di P, si utilizzeranno stringhe binarie di lunghezza simile, determinata dalla posizione di P nella regione tra L e H. Vengono quindi assegnate:

  - stringhe leggermente più corte nelle posizioni centrali, poiché più probabili,
  - stringhe leggermente più lunghe nelle posizioni marginali, poiché meno probabili.

- Se P risiede a sinistra di L, la codifica di P inizierà con 10.
- Se P risiede a destra di H, la codifica di P inizierà con 11.

Per gli ultimi due casi, è importante considerare che la probabilità che P sia molto distante dai valori di riferimento ($P \gg H$ e $P \ll L$) diminuisce man mano che ci si allontana da L e H. Di conseguenza, si assegneranno stringhe più corte per valori vicini a L e H e stringhe più lunghe per valori lontani.
Nella regione centrale ci sono $L - H + 1$ posizioni da codificare, il che richiede codici a lunghezza variabile non molto diversi in termini di dimensione, e che devono essere **codici prefisso**. Tuttavia, questi valori sono quasi tutti equiprobabili. Per codificare tali valori si può usare l'**Adjusted Binar Code**.

7.4 Adjusted Binar Code

Supponiamo che $H - L = 9$ e poniamo:

- $k = \lfloor \log_2(H - L) \rfloor = 3$
- $a = 2^{k+1} - ((H - L) + 1) = 6$
- $b = 2 \cdot ((H - L) + 1 - 2^k) = 4$

Avremo che:

- a codificherà i numeri $2^k - 1, 2^k - 2, \ldots$ come numeri a k bit. Di conseguenza:
  - $8 - 1 = 111$, $8 - 2 = 110$, fino a $8 - 6 = 010$
- b codificherà i numeri $0, 1, 2, \ldots, k$ come numeri a $(k + 1)$ bit. Di conseguenza:
  - $0000, 0001, 0010, 0011$

Le parole codice di a verranno utilizzate al centro della regione, mentre le parole codice di b verranno usate ai margini, cioè vicino a L e H. Si utilizza quindi un mix di stringhe di bit più lunghe e stringhe di bit più corte, dove:

- La scelta della quantità di bit da usare viene decisa tramite il calcolo di a e b. Si associano le stringhe più corte ai valori centrali (più probabili) e le stringhe più lunghe ai valori marginali (meno probabili).

7.4.1 Golomb code rivisitato

Quando P si trova nelle regioni esterne, ad esempio quella a destra, il valore $P - H$ dovrebbe avere una codifica a lunghezza variabile, la cui dimensione cresce velocemente man mano che l'intensità di P si allontana da quella di H. Un modo per ottenere ciò è selezionare un intero non negativo m (di solito $0, 1, 2$ oppure 3) e assegnare all'intero $n = P - H$ una codifica in due parti:

- La seconda parte è costituita dagli m bit meno significativi di n.
- La prima parte è la codifica unaria di n senza i suoi m bit meno significativi.

Esempio

Se $m = 2$, allora al valore $n = 1101$ viene assegnato il codice $110|01$, dove 110 è la codifica unaria di 11, e 01 sono gli m bit meno significativi di n. Questo codice è un caso speciale del **Golomb code**, dove il parametro b è una potenza di 2 (ovvero 2^m).

Sintesi

- Se ci troviamo nella regione tra L e H, codifichiamo P tramite la codifica di $H - L + 1$ e assegniamo un numero di bit minore se P si trova tra i valori centrali, mentre assegniamo un numero maggiore di bit per i valori più distanti.
- Se ci troviamo nelle regioni esterne, codifichiamo la differenza $n = P - H$ (oppure $L - P$) e assegniamo a P una codifica in due parti utilizzando un **codice di Golomb** modificato. Si sceglie un intero m e si suddivide la codifica in due parti, concatenando la codifica unaria dei $|n| - m$ bit più significativi e gli m bit meno significativi di n.

Golomb Rice	$m = 1$ $k = 0$	$m = 2$ $k = 1$	$m = 3$	$m = 4$ $k = 2$	$\cdots$	$m = 6$	$\cdots$	$m = 8$ $k = 3$
$n = 0$	0·	0·0	0·0	0·00		0·00		0·000
1	10·	0·1	0·10	0·01		0·01		0·001
2	110·	10·0	0·11	0·10		0·100		0·010
3	1110·	10·1	10·0	0·11		0·101		0·011
4	11110·	110·0	10·10	10·00		0·110		0·100
5	111110·	110·1	10·11	10·01		0·111		0·101
6	1111110·	1110·0	110·0	10·10		10·00		0·110
7	11111110·	1110·1	110·10	10·11		10·01		0·111
8	111111110·	11110·0	110·11	110·00		10·100		10·000
9	1111111110·	11110·1	1110·0	110·01		10·101		10·001
⋮	⋮	⋮	⋮	⋮		⋮		⋮

Tale metodo viene utilizzato poiché, man mano che il valore di P si allontana da H (oppure da L), la codifica di P aumenta rapidamente di lunghezza.

Capitolo 8

JPEG

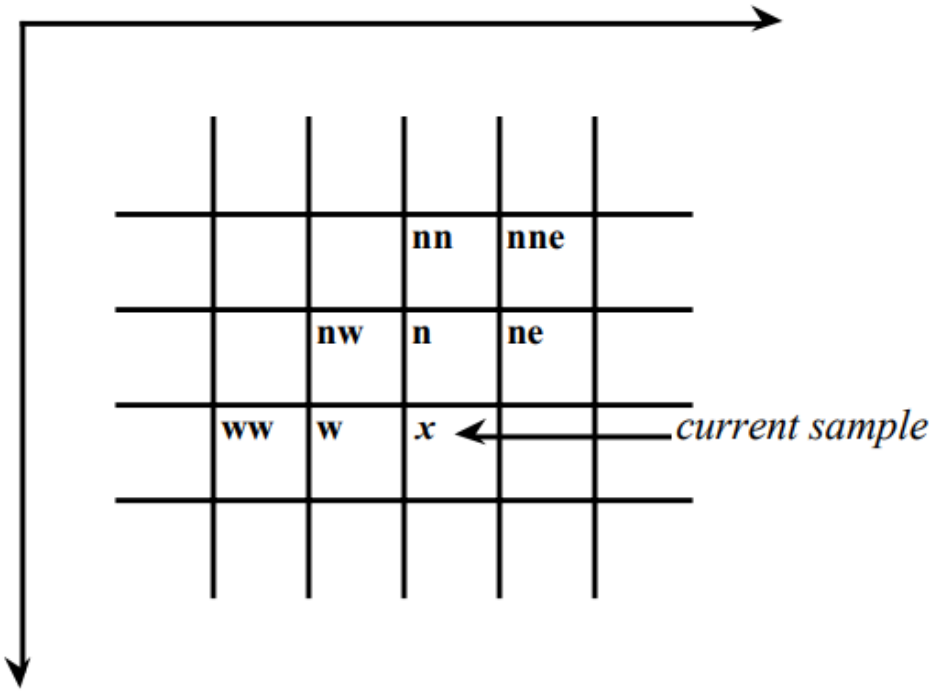
8.1 JPEG-LS

JPEG-LS è un nuovo standard nato perché, sebbene fosse già presente, l'algoritmo originario per la compressione di immagini lossless era già vecchio e superato da altri algoritmi contemporanei, motivo per cui non è stato quasi mai implementato. L'algoritmo ritenuto **ottimale in termini di rapporto tra "qualità di compressione e complessità computazionale"** è stata la proposta $LOCO - I$.
JPEG-LS esamina alcuni dei vicini già codificati del pixel che deve essere codificato e utilizza un contesto per la predizione del pixel, selezionando una distribuzione di probabilità per codificare l'errore di predizione ϵ, che verrà codificato tramite codifica di Golomb. Ci sono due modalità di funzionamento:

- **regular mode**
- **run mode**, specifico per **situazioni con un grande numero di pixel con lo stesso valore**. In questo caso, anziché indicare per tot volte un pixel con una determinata intensità, si indica che viene ripetuto tot volte con la stessa intensità.

Il contesto utilizzato per predire il pixel è dato dai pixel circostanti; considerando un pixel x, i pixel del suo intorno vengono indicati tramite le direzioni cardinali:

- nw
- n
- ne
- w

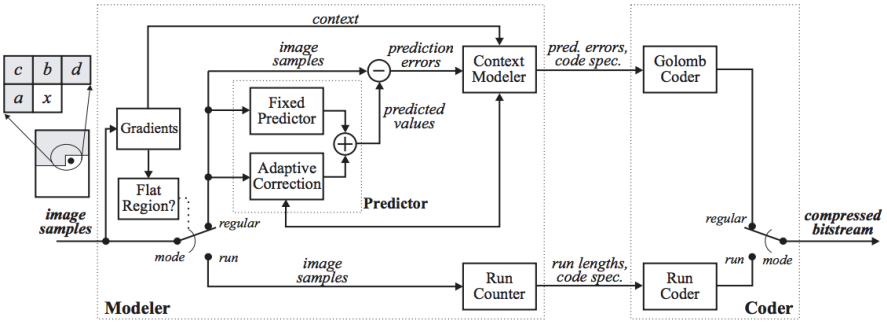


Il codificatore esamina i pixel del contesto e decide se codificare il pixel corrente in *regular mode* o in *run mode*:

- Se tutti i pixel precedenti a x hanno lo **stesso valore di** x, allora si rimane in ***run mode*** e si prosegue considerando un nuovo pixel.
- Se il contesto suggerisce che i **pixel precedenti e successivi non sono identici**, si entra in ***regular mode***.

Il predittore è computazionalmente veloce, basandosi su calcoli rapidi per ottenere la predizione $x_{MED} \triangleq \min(I_w, In, I_{nw}) + \max(I_w, In, I_{nw}) - I_{nw}$, nota come *median edge detector*. Tale predizione incorpora la conoscenza dei pixel precedenti (detti gradienti), che permette di comprendere il contorno di x e, in base ai gradienti, determinare se il pixel si trova in una regione piatta o meno.
Se il pixel appartiene a una **regione piatta**, tutti i pixel hanno la stessa intensità e si entra in ***run mode*****,** in cui:

- Si **contano le occorrenze** di pixel uguali tramite il ***run counter***.
- Si utilizzano **tecniche di *run-length encoding*** per codificare in modo efficiente le lunghe sequenze di pixel identici. Il decompressore, che utilizza le stesse informazioni del codificatore, riceverà un dato del tipo “sono presenti 500 pixel tutti dello stesso colore”, rimanendo quindi anch’esso in *run mode* e in grado di decodificare.



Se il pixel x **non fa parte di una regione piatta**, allora i pixel presentano valori diversi e si entra in ***regular mode***:

- **Si ottiene la predizione** x_{MED} tramite il ***Fixed Predictor*** e la stessa previsione viene effettuata dal decompressore.
- Tale previsione passa attraverso l'***Adaptive Correction***, in cui la predizione viene corretta tenendo conto dell'errore osservato ogni volta che si verifica un contesto simile.
- **Si calcola l'errore di predizione** (che non può essere calcolato dal decompressore poiché non conosce il valore effettivo del pixel) e lo si manda al ***Context Modeler*** per correggere adattivamente le future previsioni.
- **Si sceglie un contesto** e una distribuzione di probabilità per codificare l'errore di predizione tramite codifica di Golomb.

8.2 Lossy JPEG

JPEG è uno standard di compressione di immagini a tono continuo (general-purpose), utilizzato per problemi di trasmissione o memorizzazione di immagini fotografiche digitali. Tale standard offre quattro modalità operative:

- Il primo è *lossless encoding*, ovvero gli algoritmi di Lossless JPEG.
- Gli altri tre funzionano in modo simile:
  - *baseline sequential encoding*
  - *progressive encoding*
  - *hierarchical encoding*, con gli ultimi due che inviano l'immagine tramite approssimazioni successive.

8.2.1 Baseline JPEG

Baseline JPEG è una delle versioni più utilizzate poiché raggiunge alte prestazioni sia in termini di compressione sia di qualità dell'immagine decompressa, risultando visivamente simile all'originale per l'occhio umano. Negli algoritmi *lossy* devono essere considerati **due parametri fondamentali**, poiché grandi rapporti di compressione senza qualità accettabile dell'immagine rappresentano un approccio inefficace.

L'idea alla base di *Baseline JPEG* è il ***transform coding***, utilizzando la trasformata *Forward DCT* per passare dal dominio spaziale (dei pixel) al dominio delle frequenze. Una volta portati i pixel nel dominio delle frequenze, diventa più facile individuare le informazioni (frequenze) a cui l'occhio umano è meno sensibile, per poi eliminarle tramite *filtraggio*. La quantità di informazione filtrata determina la qualità dell'immagine di output ottenuta con la trasformata inversa:

- Maggiore è il filtraggio, maggiore sarà la compressione, ma la qualità dell'immagine sarà inferiore.
- Minore è il filtraggio, minore sarà la compressione, ma la qualità dell'immagine sarà superiore.

La trasformata *Forward DCT* non rende teoricamente l'algoritmo *lossy*, poiché non introduce errore; tuttavia, in pratica, la precisione limitata delle macchine potrebbe produrre immagini di output differenti.
Per applicare la trasformata, l'immagine viene suddivisa in **blocchi di** 8×8 **pixel**; se le dimensioni dell'immagine non sono multipli di 8, si effettua del *padding*, e i blocchi vengono **ordinati in modo *raster-like***. La *Forward DCT* viene applicata su ciascun blocco indipendentemente, ottenendo 64 coefficienti *DCT*, scorrelati con i pixel di partenza e rappresentanti frequenze, che verranno sottoposti a ***quantizzazione*** (filtraggio) tramite tabelle apposite, **eliminando le frequenze meno visibili all'occhio umano**. Le tabelle di filtraggio possono essere di default o definite dall'utente per applicazioni specifiche. Una volta filtrati, **i coefficienti di ciascun blocco vengono ordinati tramite *zig-zag scan*** e lo stream di bit risultante è **codificato tramite *run-length coding*** per ottenere simboli intermedi, che saranno ulteriormente compressi tramite *Arithmetic Coding* o *Huffman* e inviati al decompressore per l'operazione inversa.
Supponiamo di avere un'immagine in input 640×480. I passi di *Baseline JPEG* sono sei:

1. ***Preparazione dei blocchi***: si calcolano luminanza e crominanza YUV, costruendo matrici separate per ogni componente. Le matrici U e V vengono ridotte a 320×240 mediante *downsampling*, mentre la matrice Y rimane 640×480. Si sottrae poi 128 da ciascun elemento per centrare l'intervallo attorno a 0, e ogni matrice viene suddivisa in blocchi 8×8.

2. *Applicazione della Forward DCT*: applicata ai blocchi 8×8, genera una matrice 8×8 di *coefficienti DCT*, dove l'elemento $(0, 0)$ rappresenta il valore medio del blocco.

3. ***Quantizzazione***: i coefficienti *DCT* meno visibili vengono eliminati dividendo ciascuna matrice 8×8 per una *tabella di quantizzazione* e arrotondando i valori risultanti. Le tabelle possono azzerare i coefficienti *DCT* non necessari, rimuovendo informazioni non significative e comprimendo l'immagine.

4. Ogni valore $(0, 0)$ di ciascun blocco viene sostituito con la differenza rispetto all'elemento corrispondente nel blocco precedente.

5. ***Linearizzazione dei coefficienti DCT***: utilizzando uno *zig-zag scan*, si genera una sequenza lineare di numeri, producendo una lunga sequenza di zeri che sarà compressa con *run-length encoding* per indicare il numero di zeri presenti.

6. ***Codifica della lista***: la lista viene codificata tramite *Huffman* o *Arithmetic Coding* e inviata al decompressore.

L'algoritmo di decodifica esegue gli stessi passaggi in ordine inverso rispetto all'algoritmo di codifica, rendendo *JPEG* un algoritmo di compressione di immagini *simmetrico*, che richiede lo stesso lavoro per codificare e decodificare le immagini.

Capitolo 9

MPEG

MPEG è lo standard utilizzato per la compressione di dati **tridimensionali**, in particolare dei video. Un video è una serie di immagini digitali a una frequenza tale (minimo 24) da dare all'occhio umano l'impressione di movimento. È considerato un **dato tridimensionale** poiché presenta due tipi di ridondanze, sui quali si basano gli standard **MPEG-1** e **MPEG-2**.

Queste ridondanze sono:

- **Ridondanza spaziale**, la correlazione tra un pixel P e i pixel appartenenti al proprio intorno (come nel caso delle immagini in JPEG).
- **Ridondanza temporale**, la correlazione tra il pixel di un'immagine e lo stesso pixel dell'immagine successiva nella stessa posizione. Le immagini consecutive possono quindi essere correlate tra di loro, rendendo più facile comprimere un'immagine in base alle immagini precedenti.

Considerare la terza dimensione e il terzo tipo di ridondanza permette un maggiore rapporto di compressione.

9.1 MPEG-1

Lo standard **MPEG-1** è stato progettato per la codifica in forma digitale di immagini in movimento e audio, pensato principalmente per contenuti multimediali con un bitrate fino a $1,5$ Mbs. Tra i limiti di questo standard, troviamo:

- **Assenza della modalità interlacciata**, che non supporta la rilevazione di errori e la perdita di informazioni dovuti a problemi di trasmissione.

9.2 MPEG-2

Lo standard **MPEG-2** è nato per le trasmissioni video digitali e per la diffusione televisiva. Supporta una vasta gamma di risoluzioni e bitrate e include la **modalità interlacciata**, che consente il trasporto di più flussi indipendenti su un mezzo non affidabile e soggetto a perdita di informazioni, affinché possano essere decodificati singolarmente.

Data la varietà di applicazioni, sono stati definiti diversi profili operativi, che consentono la visualizzazione di un flusso a risoluzioni differenti in base alle caratteristiche del canale trasmissivo e del ricevitore, permettendo la decodifica a qualità ridotta.

Il valore di un particolare pixel può essere predetto tramite:

- pixel adiacenti, utilizzando tecniche di compressione **intra-frame** che sfruttano la ridondanza spaziale,
- il valore di un pixel di un frame vicino temporalmente, utilizzando tecniche di compressione **inter-frame** che sfruttano la ridondanza temporale. Durante gli stacchi di scena, però, la correlazione temporale tra i pixel è bassa o nulla, quindi in questi casi si preferiscono tecniche **intra-frame** per una compressione più efficiente.

MPEG-2 utilizza la DCT su blocchi di pixel 8×8 per sfruttare la correlazione spaziale tra i pixel vicini della stessa immagine. Se invece la correlazione tra pixel di immagini vicine è alta, e quindi si prediligono tecniche **inter-frame**, viene usata la **DPCM** (Differential PCM), che implementa una **predizione con compensazione del moto tra frame**. Per ottenere un'elevata compressione, **MPEG** video utilizza uno schema di codifica che combina tecniche di predizione temporale e compensazione del moto con la trasformata **DCT** delle informazioni spaziali rimanenti.

9.2.1 Codifica DPCM

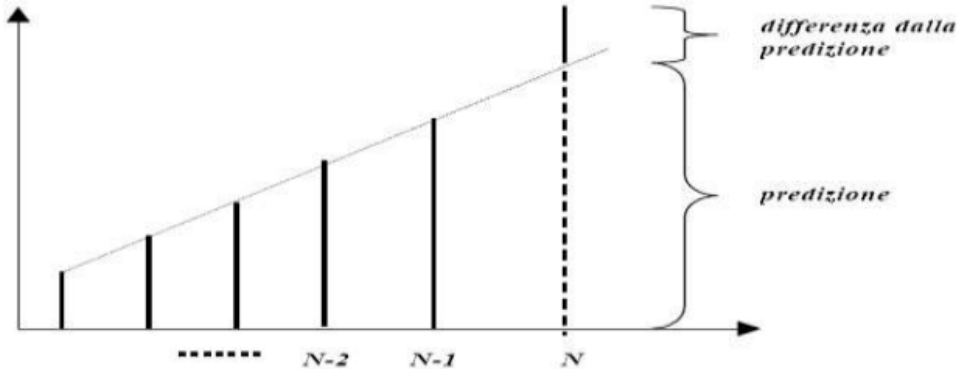
Per codificare il campione n-esimo del segnale di ingresso:

1. si cerca una **predizione dagli** $n-1$ **campioni precedenti** già codificati,
2. si calcola e si codifica soltanto la **differenza tra la predizione e il campione reale**.

Il decodificatore utilizza lo stesso algoritmo per ottenere la predizione del campione n-esimo e ricava il valore effettivo del pixel sommando alla predizione la differenza decodificata.

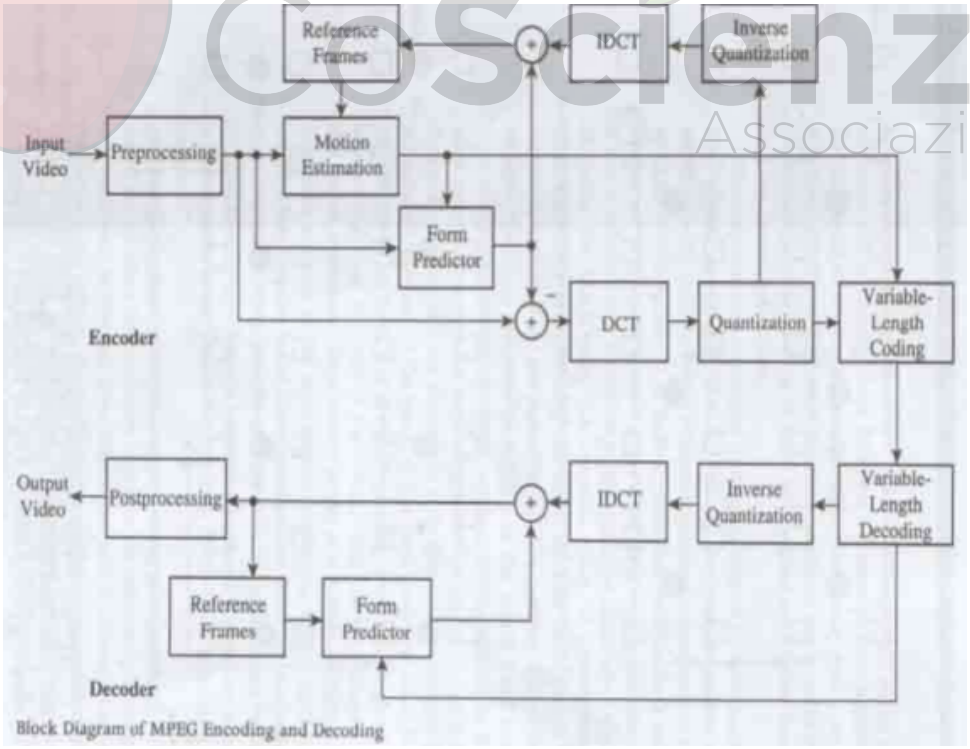
Questa tecnica mira a ridurre l'intervallo di variabilità del pixel da codificare per ottenere una rappresentazione più efficiente in termini di bit. Ad esempio, in un video composto da 100

immagini differenti, se per un determinato pixel si individuano 20 valori differenti, si otterrà una compressione migliore codificando solo questi 20 valori anziché tutti i 100.



9.2.2 MPEG Block Diagram

Il video in input viene inviato a una parte di **Motion Estimation**, che cerca di identificare le differenze tra i frame precedenti o successivi e il frame attuale, pixel per pixel, in base a dei **reference frames**. Una volta ottenuta la predizione, la differenza tra la predizione e il valore reale viene inviata al decompressore tramite un procedimento simile a **JPEG** (viene calcolata la DCT, quantizzata e codificata tramite **Huffman** prima di essere inviata al decompressore). Il decompressore esegue gli stessi passaggi a ritroso per ottenere il valore effettivo del pixel.



I principali problemi nella realizzazione di **MPEG** sono:

- L'elevato livello di compressione desiderato non è raggiungibile comprimendo tutte le immagini che compongono il video di input.

- Si vuole garantire un accesso casuale, impossibile se si codificano le immagini in modo dipendente; ad esempio, per accedere al frame n, sarebbe necessario codificare prima tutti i $n-1$ frame precedenti.

Per risolvere questi problemi, lo standard **MPEG** introduce tre tipi di frame, che devono essere decodificabili singolarmente:

- **I frame** (**frame intracodificati**): immagini fisse autocontenute codificate tramite **JPEG**, importanti per garantire l'accesso casuale ai video.
- **P frame** (**frame predittivi**): previsione del frame corrente sfruttando la ridondanza temporale tra frame correlati.
- **B frame** (**frame bidirezionali**): differenze rispetto al frame precedente e al successivo tramite interpolazione, permettendo di costruire il frame n conoscendo i frame $n-1$ e $n+1$.

FRAME I

I **frame I** sono immagini fisse codificate con un algoritmo simile a **JPEG**. Questi frame devono apparire periodicamente per quattro ragioni:

1. Per tenere traccia dei cambiamenti di scena.
2. Per consentire l'**accesso casuale**, poiché la perdita di un frame iniziale impedirebbe la decodifica dei frame successivi.
3. **Gli errori di comunicazione si propagherebbero nel tempo**; pertanto, un frame ricevuto con errori causerebbe errori in tutti i frame successivi.
4. Per effettuare il fast **forward o rewind**, si salta al **frame I** più vicino per poi continuare la decompressione e la visualizzazione.

FRAME P

I **frame P** forniscono gran parte della compressione e sfruttano la ridondanza temporale. Codificano le differenze tra frame consecutivi utilizzando **macroblocchi** che vengono cercati nel frame precedente, trovando il **macroblocco** corrispondente o una versione simile.

La tecnica di **block-matching** si basa su una **ricerca sintattica** (intensità dei pixel) e determina il vettore di moto minimizzando una funzione di costo. La compensazione del moto si basa sull'idea che l'immagine attuale possa essere divisa in parti modellabili come traslazioni di un'immagine precedente. Questo approccio ha dei limiti, poiché in casi particolari, come stacchi di scena o movimenti della telecamera, possono insorgere problemi. I **macroblocchi** interpolati sono generati come media tra la predizione di un frame passato e la predizione di un frame futuro.

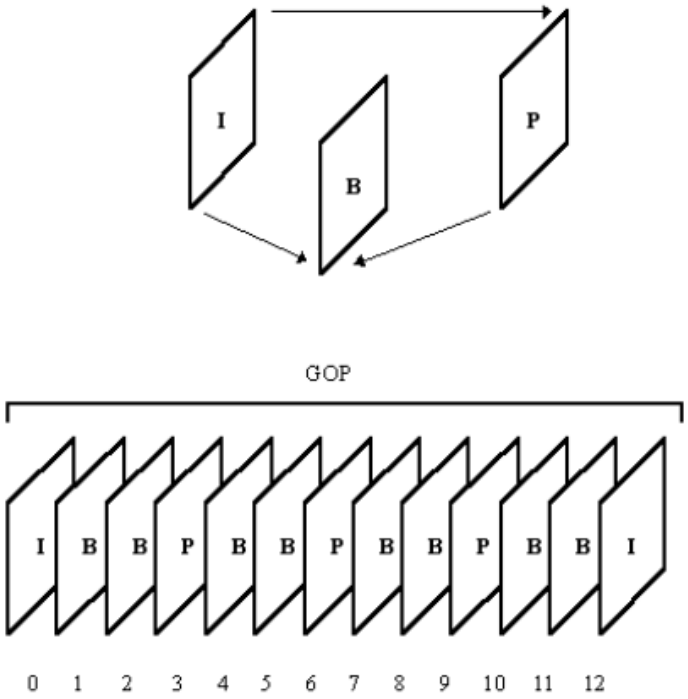
Con la compensazione del moto, ogni blocco del frame viene confrontato in un'area di ricerca, che può includere l'intera immagine o solo una parte, per trovare la migliore approssimazione. Questo processo porta alla **Direzione di Minima Distorsione (DMD)**.

FRAME B

I **frame B** sono frame interpolati a partire da **macroblocchi** di riferimento dei frame precedenti e successivi, che devono essere di tipo I o P. I frame non vengono inviati nell'ordine di visualizzazione, ma seguono la **Sintassi MPEG**.

La **Sintassi MPEG** è una struttura stratificata:

- **Livelli incapsulati**, ognuno con un header seguito dai dati.
- Il **sequence header** è al vertice della gerarchia e definisce dimensioni delle immagini, frame rate e bit rate.
- Ogni **sequence header** è diviso in gruppi di immagini (group of pictures, **GOP**), un insieme di immagini in **ordine contiguo di codifica**, che può differire dall'ordine di visualizzazione. Ciascun **GOP** contiene tre tipi di immagini: frame di tipo I, frame di tipo P e frame di tipo B, e si costruiscono i frame di tipo B interpolando i frame di tipo I e P più vicini.



I frame sono ulteriormente suddivisi in **slice**, e gli **slice header** contengono informazioni importanti per la sincronizzazione e il controllo degli errori. **MPEG-2** stabilisce che ogni slice deve iniziare e finire sulla stessa riga e deve contenere un numero intero di macroblocchi di dimensione 16×16 pixel di luminanza e le relative aree di crominanza.