



UNIVERSITÀ DEGLI STUDI DI SALERNO

Appunti del corso di Compressione dati

A.A. 2021/2022

Indice

0	Informazioni sul corso	5
1	Introduzione	6
1.1	Compressione lossless e lossy	6
2	Basi teoriche	7
2.1	Alfabeto e sorgente	7
2.1.1	Sorgente costante K-aria	7
2.1.2	Sorgente casuale K-aria	7
2.2	Entropia	8
2.3	Sorgenti di ordine i-esimo	8
2.4	Considerazioni importanti	8
3	Tecniche di codifica	10
3.1	Codifica sorgente	10
3.2	Codifica univocamente decifrabile	11
3.3	Disuguaglianza di Kraft/MacMillian	11
3.4	Codici prefissi	11
3.5	Codifica di Huffman	12
3.6	Codifica aritmetica	15
3.6.1	Codifica	16
3.6.2	Decodifica	17
3.6.3	Problemi pratici	18
4	Metodi di sostituzione testuale on-line	21
4.1	Compressione basata su dizionari	21
4.1.1	Euristiche	22
4.1.2	Puntatori a lunghezza variabile	23
4.1.3	Metodi basati su static dictionaries	23
4.1.4	Metodi basati su sliding dictionaries	23
4.1.5	Metodi basati su dynamic dictionaries	24
4.1.6	LZ78	26
4.1.7	Implementazione LZ78	27
4.1.8	Dizionario LZ78	28
4.2	LZW (Lempel Ziv Welch)	28
4.3	The catch	30

5	Lossless image compression	31
5.1	Images	31
5.2	Lossless image compression	31
5.3	Modeling and Coding	32
5.3.1	Una strategia di modellazione	32
5.4	Lossless JPEG	34
5.5	FELICS	35
5.5.1	Esempio: codice binario corretto	36
5.6	Le regioni esterne (outer regions)	36
5.7	JPEG-LS	37
5.7.1	Un nuovo standard di compressione delle immagini senza perdita.	37
5.7.2	Predittore, errore di previsione e codifica	38
5.8	Lo standard JPEG	39
5.9	Baseline JPEG	39
5.9.1	Funzionamento codifica baseline	40
6	Compressione video	43
6.1	Standard MPEG	43
6.1.1	Origini dello standard	43
6.2	MPEG	43
6.2.1	Principi generali della codifica MPEG video	44
6.2.2	Esempio di codifica DPCM	44
6.2.3	Standard MPEG 2: Frame I	45
6.2.4	Standard MPEG: Frame P	46
6.2.5	Compensazione del moto	46
6.2.6	Standard MPEG: Frame B	46
6.2.7	I livelli della sintassi MPEG video	46

Capitolo 0

Informazioni sul corso

Il corso è diviso in due parti: nella prima parte si affronteranno le lezioni teoriche, nella seconda parte si organizzerà il progetto. La modalità d'esame prevede appunto delle domande sulla prima parte e l'esposizione del progetto. Il progetto riguarda l'approfondimento su un argomento di compressione dati che si trovano "al confine" tra lo stato dell'arte e la ricerca, nel senso che è possibile ampliare un progetto portato in passato aggiungendo nuovi algoritmi oppure partire ex novo da un argomento nuovo, andando a cercare materiale su articoli, pubblicazioni, ecc.

Un buon progetto è diviso in tre parti:

- 1a parte: esposizione di un argomento da approfondire;
- 2a parte: soluzioni esistenti in letteratura o sperimentali al problema esposto;
- 3a parte: testare e modificare le soluzioni esistenti o trovarne di nuove.

I gruppi devono essere di 4 o 5 persone.

Capitolo 1

Introduzione

Che cos'è la compressione dati? Non esiste una definizione precisa: intuitivamente è il processo che permette di codificare un insieme di dati D in un altro insieme di dati D^I che ha dimensioni inferiori. Bisogna però essere in grado di risalire a D partendo da D^I con un programma che mi permetta di ricostruire l'insieme di dati iniziali esattamente (compressione *lossless*), o di risalire ad una approssimazione accettabile dell'insieme di dati iniziale (compressione *lossy*).

Originariamente la compressione dati veniva utilizzata per minimizzare l'utilizzo dello spazio su disco, in quanto c'erano dischi piccoli che dovevano ospitare grandi basi di dati. La compressione dati però ha un compito fondamentale nella comunicazione: senza di essa oggi non avremmo la televisione in alta definizione, gli smartphone, lo streaming via internet, in quanto tutti i dati che viaggiano in rete sono dati che vengono compressi appunto per aumentare la banda di comunicazione o per inviare più trasmissioni contemporaneamente sullo stesso canale.

1.1 Compressione *lossless* e *lossy*

Esistono due tipi di compressione:

- **Compressione *lossless* (senza perdita)** - viene usata quando i dati devono essere compressi per qualche motivo, ma poi devono essere decompressi senza che venga persa alcuna informazione. Dopo aver decompresso il dato, questo è identico all'originale. È anche detta **compressione *bit preserving*** o compressione reversibile. Viene utilizzata su alcuni tipi di dati specifici, come ad esempio dati monodimensionali (programmi oggetto o sorgente, testo, ecc.). La compressione *lossless* viene utilizzata anche in casi in cui l'acquisizione dei dati è molto costosa e quindi non voglio perdere un singolo bit di informazione (si pensi ai dati raccolti nello spazio e trasmessi sulla Terra); oppure quando non sono ancora note le operazioni da effettuare sui dati raccolti, quindi li si comprime facendo in modo che questi possano essere recuperati interamente. La compressione *lossless* porta a rapporti di compressione inferiori rispetto alla *lossy*: file di testo tipicamente vengono compressi con rapporto¹ 8 a 1 o 10 a 1, mentre immagini con rapporto 2 a 1. La compressione *lossless* è molto importante anche nella trasmissione delle informazioni perché anche solo risparmiando il 10% della banda porta un risparmio economico.
- **Compressione *lossy* (con perdita)** - nella compressione *lossy* non c'è più un unico parametro che ci dice di quanto è stato compresso il file, ma qui si parla di due parametri bilanciati tra di loro: l'ammontare della compressione e la **fidelity**, ovvero la fedeltà del file decompresso rispetto all'originale. È possibile utilizzare varie combinazioni di questi parametri: si può ottenere una fedeltà massima aumentando il *bit rate* o lo si può diminuire per raggiungere una fedeltà fissata.

¹Un rapporto 2 a 1 indica che il file compresso pesa la metà dell'originale, un rapporto 4 a 1 indica che il compresso pesa un quarto dell'originale, e così via.

Capitolo 2

Basi teoriche

La compressione dati si è evoluta come sottoinsieme del campo di Teoria dell'Informazione diventando poi un campo a sé stante.

2.1 Alfabeto e sorgente

Un alfabeto è un insieme finito di caratteri contenente almeno un elemento. Gli elementi di un alfabeto sono chiamati *caratteri*. Una *stringa* su un alfabeto è una sequenza di caratteri dell'alfabeto.

Definizione: Sia $\Sigma = \{s_1 \dots s_k\}$, $k \geq 1$ un alfabeto. Una **sorgente** è un processo che sequenzialmente produce caratteri di Σ . Σ è chiamato **alfabeto sorgente**. Una sorgente è detta **del primo ordine** se le probabilità indipendenti di emissione $p_1 \dots p_k$, che danno come somma 1, sono associate con il corrispondente elemento di Σ . Di norma un carattere s_i avrà probabilità p_i di essere il prossimo carattere che viene trasmesso.

Nelle sorgenti del primo ordine l'emissione del carattere *i-esimo* non dipende dai caratteri che sono stati trasmessi precedentemente. Vediamo ora due esempi di sorgente che possono tornare utili.

2.1.1 Sorgente costante K-aria

È definita su un certo alfabeto $\Sigma = \{s_1 \dots s_k\}$, e nonostante sia $k \geq 1$ abbiamo che all'interno di questo alfabeto viene scelto sempre lo stesso carattere come prossimo carattere da trasmettere. Detto s_i questo carattere, avremo che la probabilità p_i sarà sempre 1, mentre per ogni s_j con $j \neq i$ avremo che $p_j = 0$.

Questa sorgente ha il minimo grado di "*sorpresa*", ovvero un osservatore esterno che conosce il tipo di sorgente non sarà sorpreso a veder trasmettere sempre lo stesso carattere.

2.1.2 Sorgente casuale K-aria

È definita anch'essa su un alfabeto $\Sigma = \{s_1 \dots s_k\}$, con $k \geq 1$. Il prossimo carattere che viene emesso viene scelto in modo uniforme e indipendente tra tutti i caratteri, quindi abbiamo che per ogni s_i , $p_i = \frac{1}{k}$. L'osservatore non potrà fare nessuna predizione sul prossimo carattere perché tutti i caratteri hanno la stessa probabilità di emissione. Questa sorgente ha il livello di "*sorpresa*" più alto.

2.2 Entropia

Possiamo identificare il concetto di sorpresa visto finora con il concetto di entropia, espresso formalmente di seguito.

Definizione: Sia $k \geq 1$ un intero e sia S una sorgente di primo ordine che genera i caratteri dall'alfabeto

$$\Sigma = \{s_1 \dots s_k\}$$

con probabilità indipendenti $p_1 \dots p_k$. L'entropia di S in base $r, r \geq 1$, è data da:

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

Quando la base non è specificata assumiamo che sia 2, in quanto ragioniamo in logica binaria, per cui abbrevieremo $H_2(S)$ come $H(S)$.

Questa informazione mi dice che per comprimere un oggetto ho bisogno di almeno $H(S)$ bit se non voglio incorrere in errori. Inoltre è sempre possibile costruire uno schema di codifica che codifica l'output di quella sorgente usando un numero di bit pari all'entropia. Questo è indicato nel **Teorema fondamentale della codifica sorgente**:

Teorema: Sia S una sorgente di primo ordine su un alfabeto Σ e sia Γ un alfabeto di $r > 1$ caratteri. Codificare i caratteri di S con i caratteri di Γ richiede una media di $H_r(S)$ caratteri di Γ per ogni carattere di Σ . Inoltre, per ogni numero reale $\epsilon > 0$ esiste uno schema di codifica che usa in media $H_r(S) + \epsilon$ caratteri di Γ per ogni carattere di Σ .

2.3 Sorgenti di ordine i-esimo

Raramente in scenari reali le sorgenti sono del primo ordine. Tipicamente la probabilità che la sorgente emetta un carattere dipende dai caratteri che ha emesso in passato. Ad esempio in un file di testo, consideriamo una sorgente che emette parole italiane. Se troviamo il carattere Q allora è molto probabile che il carattere successivo sia una U . Questa osservazione porta alla definizione teorica di **sorgente di ordine i-esimo**, con $i \geq 1$, dove i denota il numero di caratteri dal quale il carattere che viene emesso dipende. Ad esempio, in una sorgente di ordine 2 il prossimo carattere dipende dal precedente, in una sorgente di ordine 3 il prossimo carattere dipende dai due precedenti, e così via.

2.4 Considerazioni importanti

L'importanza dell'entropia nella compressione dati ci dà un *lower bound* per la compressione che possiamo ottenere. Per una certa sorgente quindi esisterà una certa quantità detta *entropia* sotto cui non possiamo scendere per la compressione. Tipicamente gli algoritmi di compressione lossless che riescono a comprimere al limite dell'entropia sono detti ottimali. Vedremo che le considerazioni che facciamo sulle compressioni vanno applicate nella pratica a sorgenti finite che emettono una quantità finita di bit, e non a sorgenti infinite.

Possiamo quindi elencare tre importanti considerazioni che vanno fatte:

- Dato che il limite della compressione lossless è dato dall'entropia, e nel caso di sorgenti casuali k-arie abbiamo a che fare con un'entropia massima, i dati casuali non possono essere compressi perché non contengono ridondanza. Se proviamo a comprimere tali dati, la dimensione del compresso sarà uguale o maggiore a quella dell'originale.

- I dati che sono compressi da un compressore ottimale (cioè che riesce ad arrivare al limite dettato dall'entropia) non possono essere ulteriormente compressi.
- Non si può garantire che un compressore ottenga una performance definita su tutte le istanze di un certo tipo di dati a causa del fatto che abbiamo a che fare con sequenze finite di dati e non infinite.

Capitolo 3

Tecniche di codifica

Teoricamente qualsiasi metodo di compressione dati può essere visto come esempio di codifica. In realtà ci renderemo conto che le tecniche standard di codifica occorrono ad un livello più basso rispetto ai metodi di compressione.

Un esempio di codifica² molto comune è il **codice a blocchi k-ario**, che mappa ciascun elemento di un insieme finito S di n elementi in una stringa di lunghezza $\lceil \log_k(n) \rceil$ su un alfabeto di dimensioni k . Supponiamo quindi di voler codificare l'output di una sorgente su un certo alfabeto Σ in un alfabeto binario nel modo più efficiente possibile. In questo caso utilizzando la codifica appena vista avrò bisogno di $\lceil \log_2(n) \rceil$ bit per codificare ciascun simbolo di Σ .

Un altro esempio è dato dalla codifica ASCII, che è un codice a blocchi binario che mappa ognuno dei 128 caratteri ASCII in una stringa unica binaria di 8 bit, dove il primo bit è sempre 0.

3.1 Codifica sorgente

Definizione: Data una sorgente S e un alfabeto Σ , una codifica da S a Σ è una funzione f che mappa ogni elemento di S a una stringa non vuota su Σ .

Quindi se ho una sorgente S che emette simboli su un alfabeto k -ario e voglio codificarlo su Σ che è un alfabeto binario, codificare S in binario significa prendere ciascun elemento che può essere emesso da S e codificarlo con una stringa di simboli binari. Il range di f è chiamato insieme delle *parole codice* di f . Le parole codice della funzione f nell'esempio precedente saranno tutte le stringhe binarie che codificano elementi di S .

Una funzione f è *iniettiva* se non mappa mai due elementi nella stessa stringa. Ogni codifica f può essere estesa a qualsiasi lista finita di elementi definendo:

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

dove la produttoria indica la concatenazione di stringhe.

Ad esempio, se voglio codificare $a, b, c \in S$, avrò che $f(a, b, c)$ sarà la concatenazione della codifica di a , della codifica di b e della codifica di c .

²Useremo i termini *codifica* e *codice* in maniera intercambiabile.

3.2 Codifica univocamente decifrabile

Dato che per ora stiamo parlando principalmente di compressione lossless, dovremo assumere che le funzioni di codifica siano iniettive laddove non specificato. Tuttavia, lavorare con funzioni iniettive non assicura la decodificabilità univoca di quello che viene codificato. esserci condizioni in cui date due liste L_1 e L_2 si verifica che $f(L_1) = (L_2)$. Questo ci porta a dare la seguente

Definizione: Sia f una codifica da un insieme S a un alfabeto Σ . Una stringa α su Σ è *univocamente decifrabile* rispetto a f se vi è al più una lista L di elementi di S tali che $f(L) = \alpha$. Inoltre diremo che f è univocamente decifrabile se tutte le stringhe su Σ sono unicamente decifrabili rispetto a f .

Esempio: Sia $S = \{a, b, c, d, e\}$, $\Sigma = \{0, 1\}$ e sia f definita come segue:

$$f(a) = 00$$

$$f(b) = 01$$

$$f(c) = 10$$

$$f(d) = 11$$

$$f(e) = 100$$

Chiaramente, $f(a, b, c, d, e) = 00011011100$. Inoltre, possiamo verificare che la stringa di bit 00011011100 è univocamente decifrabile rispetto a f . Tuttavia, non tutte le stringhe binarie sono univocamente decifrabili rispetto a f . Infatti, $f(cba) = f(ee) = 100100$.

3.3 Disuguaglianza di Kraft/MacMillian

Una domanda che ci viene naturale per ogni insieme finito S e per ogni alfabeto Σ , se le lunghezze delle parole codice sono specificate a priori, quali sono le condizioni necessarie e sufficienti su queste lunghezze per assicurarci che esista una codifica univocamente decifrabile da S a Σ .

Il problema è il seguente: ho una sorgente S con un certo alfabeto sorgente di k simboli, e un alfabeto di codifica Σ (per semplicità pensiamo all'alfabeto binario). Mi vengono date in anticipo delle lunghezze per quanto riguarda le parole codice, quindi nel mio caso avrò k lunghezze perché ho k simboli sorgente. Ad esempio voglio codificare il primo simbolo con 2 bit, il secondo simbolo con 1 bit, il terzo con 3 bit, e così via. Quali sono le condizioni necessarie e sufficiente su queste lunghezze che mi vengono date in anticipo per far sì che esista una codifica univocamente decifrabile da S a Σ ? Il teorema seguente, detto appunto *disuguaglianza di Kraft/MacMillian*, ci dà queste condizioni.

Teorema: Siano S e Σ rispettivamente un insieme finito di simboli e un alfabeto di simboli. Se le parole codice sono vincolate ad avere lunghezza $l_1, \dots, l_{|S|}$, allora una condizione necessaria e sufficiente affinché esista una codifica univocamente decifrabile da S a Σ è:

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

3.4 Codici prefissi

Definizione: Una codifica da un insieme S ad un alfabeto Σ è un *codice prefisso* se nessuna parola codice è prefisso di un'altra parola codice.

Nell'esempio precedente avevamo a che fare con un codice che non era prefisso perché $f(c) = 10$ è prefisso di $f(e) = 100$. Infatti quel codice non è né univocamente decifrabile né prefisso.

Un codice prefisso è sempre univocamente decifrabile, poiché quando vado a leggere una stringa da sinistra a destra la fine di una parola codice può essere determinata proprio nel momento in cui viene raggiunta, perché non ci sono altre parole codice che sono prefisso di quella. Per questo motivo, i codici prefissi sono spesso chiamati *codici istantanei*, perché riesco a fare il parsing da sinistra a destra in modo semplice e immediato, leggendo una parola codice alla volta, dato che nessuna è prefisso di un'altra.

Non è vero il contrario: un codice non prefisso potrebbe essere univocamente decifrabile, come possiamo vedere di seguito.

Esempio: Consideriamo il codice f dall'insieme $S = \{a, b, c, d\}$ all'alfabeto $\Sigma = \{0, 1\}$ definito come segue:

$$f(a) = 0$$

$$f(b) = 01$$

$$f(c) = 011$$

$$f(d) = 111$$

Chiaramente f non è un codice prefisso, perché $f(a)$ è prefisso di $f(b)$ e $f(c)$; $f(b)$ è prefisso di $f(c)$. Tuttavia, è univocamente decifrabile.

Teorema: Esiste un codice prefisso (e quindi un codice univocamente decifrabile) per qualsiasi sequenza di parole codice che abbiano delle lunghezze che soddisfano la disuguaglianza di Kraft/MacMillan.

3.5 Codifica di Huffman

Questo teorema ci porta a definire una serie di algoritmi di codifica tra cui la codifica di Huffman, che probabilmente è la tecnica di codifica più conosciuta. Nonostante potrebbe anche essere visto come algoritmo di compressione, in realtà vedremo che è solo una codifica.

Invece di utilizzare una codifica a blocchi k-aria, cioè codificare ogni carattere emesso dalla sorgente con una stringa di lunghezza fissa di simboli dell'alfabeto di codifica, usiamo la codifica di Huffman per codificare ciascun simbolo emesso dalla sorgente con una stringa che avrà un numero variabile di bit. I caratteri che verranno emessi con più probabilità verranno codificati con una stringa di lunghezza più corta mentre i simboli meno probabili saranno codificati con una stringa di lunghezza maggiore. La codifica di Huffman costruisce delle parole codice che sono prefisse. Inoltre, nonostante sia una codifica a lunghezza variabile, sarà univocamente decifrabile. Per decodificare un flusso di caratteri codificati tramite Huffman utilizzeremo un albero binario di decodifica.

Data una sorgente S , supponiamo di conoscere le sue probabilità del primo ordine. Con *probabilità del primo ordine* indichiamo la probabilità di ogni carattere di S di essere emesso come prossimo carattere. Supponiamo poi di conoscere ogni carattere x di S e per ogni carattere di conoscere la probabilità di emissione di quel carattere p_x . Anziché assegnare un codice di $\lceil \log_{|\Sigma|} |S| \rceil$ bit a ogni elemento di S (come sarebbe stato fatto con una codifica a blocchi), possiamo salvare spazio assegnando parole codice più corte a caratteri con più probabilità di essere emessi dalla sorgente e parole codice più lunghe a caratteri meno probabili.

Algoritmo

(1) Initialize *FOREST* to have a 1-node trie T_x for each element x of S .

Set $weight(T_x) = p_x$,

(2) **while** $|FOREST| > 1$ **do begin**

Let Y and Z be the two tries in *FOREST* of lowest weight (resolve ties arbitrarily).
Combine Y and Z by creating a new root r with weight $weight(Y) + weight(Z)$ that
is attached to one of Y and Z via a 0 and to the other via a 1 (the order doesn't
matter).

end

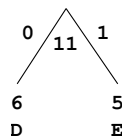
L'algoritmo di Huffman parte da una foresta di alberi. Inizializziamo questa foresta di alberi in maniera tale che abbia tanti alberi di un singolo nodo quanti sono i caratteri della sorgente S , e per ogni x il peso di quell'albero T_x sarà posto uguale alla probabilità p_x .

Poi c'è il ciclo che costruisce man mano l'albero di codifica: finché la foresta ha più di un albero, si prendono i due alberi che hanno peso più basso (e quindi hanno associato il carattere che ha probabilità di emissione più bassa), si combinano in un unico albero creando una nuova radice r con peso uguale alla somma dei pesi dei due alberi, e che ha attaccati come figli i due alberi scelti, etichettati rispettivamente con 0 e con 1. Questo ciclo continua finché non si ottiene un singolo albero binario, che sarà il nostro albero di Huffman.

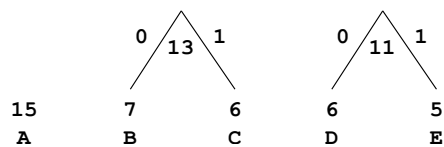
Esempio: Supponiamo di voler costruire un albero di Huffman dati i seguenti simboli e la loro frequenza:

15	7	6	6	5
A	B	C	D	E

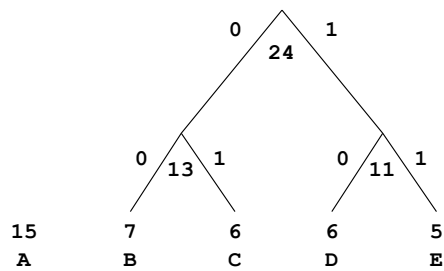
Il primo passo è quello di costruire una foresta contenente i cinque alberi. Dopodiché prendiamo i due alberi con peso minore, ovvero quelli dei simboli D e E , e li combiniamo in un albero la cui nuova radice avrà peso 11 ($6+5$), ed etichettiamo i rami con 0 e 1.



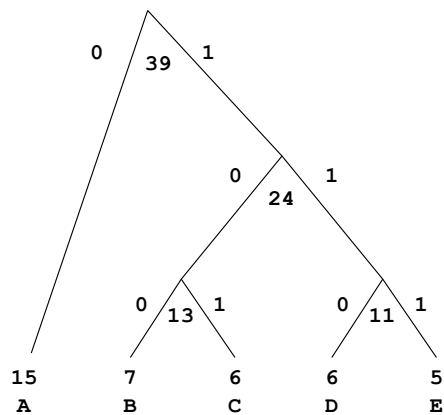
Facciamo la stessa cosa e prendiamo i due alberi con probabilità minore. In questo caso sono B e C , per cui costruiamo un nuovo albero con peso 13 ($7+6$), che si aggiunge alla foresta, che ora è:



Continuiamo nel ciclo e prendiamo i due alberi con peso minore, quindi 13 e 11, e costruiamo un altro albero che avrà peso 24. Il risultato è il seguente:



A questo punto restano due alberi, quelli con peso 15 e 24. Costruiamo allora l'albero finale:



Per determinare la codifica per ogni simbolo, basta percorrere l'albero dalla radice fino al simbolo desiderato. Per cui la codifica finale sarà:

A	0
B	100
C	101
D	110
E	111

Abbiamo inoltre costruito un codice prefisso e quindi univocamente decifrabile, perché nessuna parola codice sarà prefissa di un'altra: quando arrivo a una foglia mi fermo, non costruisco altre parole codice a partire da quella foglia.

Inoltre, possiamo notare che i simboli che hanno più probabilità sono codificati con stringhe di bit più corte, in particolare A, che aveva probabilità maggiore, è codificata con una stringa lunga 1 bit. Per questo motivo la codifica di Huffman applica una compressione migliore rispetto a un codice a blocchi k-ario standard.

La codifica di Huffman è molto utilizzata in applicazioni di compressione ben note, ad esempio in alcune versioni dell'algoritmo JPEG. Inoltre questa codifica è detta *ottimale* nel senso della Teoria dell'informazione, ovvero se ho un flusso infinito di simboli di un alfabeto sorgente e li codifico con Huffman, allora ottengo una codifica che tocca il limite ottimo dettato dall'entropia.

Uno dei limiti di questa codifica è avrò sempre bisogno di almeno 1 bit per codificare qualsiasi carattere, quindi ad esempio se vogliamo codificare con Huffman immagini binarie non abbiamo nessun vantaggio. Spesso infatti si usa per codificare caratteri a gruppi piuttosto che singoli, per migliorarne l'efficienza. Ad esempio, se la probabilità di emissione di un carattere è di $1/3$, il numero di bit ottimale per

codificare quel carattere tenendo conto dell'entropia è circa 1,6 bit, e con Huffman non possiamo codificare con un numero non intero di bit.

3.6 Codifica aritmetica

Sappiamo che la codifica di Huffman è ottimale dal punto di vista della Teoria dell'Informazione, perché se abbiamo una stringa di lunghezza infinita raggiungiamo il limite dato dall'entropia; nella pratica però le stringhe non sono infinite.

Supponiamo di conoscere le probabilità di ogni simbolo emesso dalla sorgente S , e siano queste probabilità $p_1 \dots p_{|S|}$. Consideriamo la linea dei numeri reali che comprende l'intervallo $[0, 1)$ e supponiamo di volerla dividere in maniera ricorsiva in varie parti, ciascuna proporzionale alle probabilità dei vari simboli emessi dalla sorgente. Consideriamo poi il seguente esempio: supponiamo che la sorgente $S = \{A, B, C\}$ emetta tre simboli, e che le loro probabilità di emissione siano:

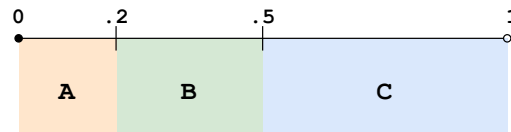
$$p_A = .2$$

$$p_B = .3$$

$$p_C = .5$$

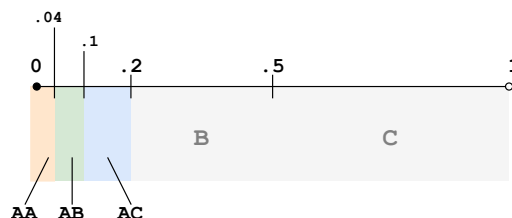
Divideremo allora la linea dei numeri reali tra 0 e 1 in tre parti. Gli intervalli saranno i seguenti:

- A corrisponde all'intervallo $[0, .2)$
- B corrisponde all'intervallo $[.2, .5)$
- C corrisponde all'intervallo $[.5, 1)$



Posso poi iterare questo ragionamento: supponiamo che la mia sorgente emetta prima A. Una volta emesso, l'algoritmo andrà a restringere l'intervallo da $[0, 1)$ a $[0, .2)$. Questo intervallo viene diviso in tre parti in base alle probabilità di emissione: avremo quindi che una seconda A corrisponde a $[0, .04)$, B corrisponde a $[.04, .1)$ e C corrisponde a $[.1, .2)$. Altri esempi:

- AC corrisponde all'intervallo $[.1, .2)$
- CA corrisponde all'intervallo $[.5, .6)$
- BC corrisponde all'intervallo $[.35, .5)$
- CAC corrisponde all'intervallo $[.55, .6)$



Quindi l'idea della codifica aritmetica è restringere sempre di più l'intervallo. Quando la stringa emessa dalla sorgente sarà finita, l'intervallo finale raggiunto indicherà la codifica della stringa che è stata emessa. Man mano che il codificatore riceve caratteri e restringe l'intervallo, manda informazioni al decodificatore che man mano riesce a ricostruire l'intervallo che è stato ristretto dal codificatore.

Codificatore e decodificatore riescono a lavorare in *lockstep*, per cui non c'è bisogno di assegnare un numero finito di bit ad ogni simbolo della sorgente. Visto che il codificatore invia un'informazione che rappresenta più di un simbolo, in realtà con 1 bit sarà possibile codificare più di un simbolo. La codifica aritmetica rimpiazza una stringa di simboli emessi dalla sorgente con un singolo numero in floating point. Ovviamente, più è complesso il messaggio, più avrò bisogno di numeri reali che abbiano una precisione maggiore e quindi più "grandi" da rappresentare.

L'output di un codificatore aritmetico sarà un numero reale che è strettamente minore di 1 e maggiore o uguale a 0. Questo numero potrà essere decodificato **univocamente** per costruire la sequenza di simboli da cui è stato ricavato, a patto che il decodificatore sappia la lunghezza del messaggio.

3.6.1 Codifica

Consideriamo una sorgente che emette la stringa **BILL GATES**, con distribuzione di probabilità:

Carattere	Probabilità
spazio	1/10
A	1/10
B	1/10
E	1/10
G	1/10
I	1/10
L	1/10
S	2/10
T	1/10

Una volta che conosciamo le probabilità sorgente, dobbiamo dividere la linea dei numeri reali in tante parti quante sono le lettere emesse dalla sorgente (quindi in 9 parti). Normalmente non importa in che ordine vado ad assegnare i simboli agli intervalli, l'importante è che siano proporzionati alla loro probabilità di emissione e che siano decodificati nello stesso ordine in cui sono stati codificati. Avremo quindi la seguente divisione:

Carattere	Probabilità	Range
spazio	1/10	$0.00 \leq r < 0.10$
A	1/10	$0.10 \leq r < 0.20$
B	1/10	$0.20 \leq r < 0.30$
E	1/10	$0.30 \leq r < 0.40$
G	1/10	$0.40 \leq r < 0.50$
I	1/10	$0.50 \leq r < 0.60$
L	1/10	$0.60 \leq r < 0.80$
S	2/10	$0.80 \leq r < 0.90$
T	1/10	$0.90 \leq r < 1.00$

La prima lettera che viene emessa dalla sorgente è la lettera B. Per andare a decodificarla, l'intervallo deve essere ristretto a $[0.20, 0.30)$. Quindi tutte le lettere successive saranno comprese in questo sotto-intervallo, così come il numero reale finale. Durante il resto del processo di codifica, ogni simbolo restringerà ulteriormente l'intervallo. Il prossimo carattere è la lettera I, il cui intervallo di riferimento

è $[0.50, 0.60)$. Per cui dividerò l'intervallo $[0.20, 0.30)$ in 9 sotto-intervalli proporzionali alle probabilità dei simboli, e sceglierò quello a cui fa riferimento il simbolo I, ovvero $[0.25, 0.26)$.

Una volta chiarita l'idea di fondo, scrivere l'algoritmo di codifica funzionante su stringhe di lunghezza arbitraria è relativamente semplice:

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

La tabella finale sarà la seguente:

Carattere	Low	High
	0.0	1.0
B	0.2	0.3
I	0.25	0.26
L	0.256	0.258
L	0.2572	0.2576
spazio	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

Il codificatore quindi potrà prendere un qualsiasi numero reale compreso tra 0.2572167752 e 0.2572167756, inviarlo al decodificatore, e il decodificatore conoscendo la lunghezza del messaggio riuscirà a decodificare il messaggio. Per semplicità viene inviato il valore corrispondente all'estremo inferiore dell'intervallo.

3.6.2 Decodifica

Il codificatore controlla all'interno di quale intervallo cade il valore ricevuto. Il valore si trova tra 0.2 e 0.3, per cui capirà che la prima lettera è B. A questo punto deve rimuovere il contributo di B dal valore da decodificare per invertire il processo. Viene quindi sottratto il valore di B, che sappiamo essere 0.2. Il valore ottenuto è 0.0572167752. Ora si divide questo numero per la dimensione dell'intervallo di B, che è $1/10$. Il numero che otteniamo è 0.572167752. Continuando il processo, vediamo che questo numero cade tra 0.5 e 0.6, che è l'intervallo della lettera I, per cui facciamo la stessa cosa: sottraiamo 0.5, ottenendo 0.072167752, e dividiamo per la lunghezza dell'intervallo di I (anche stavolta $1/10$), ottenendo 0.72167752.

Anche in questo caso indichiamo l'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;
}
```

Si può notare come nell'algoritmo di decodifica il ciclo **for** non ha una condizione. Per indicare la fine di un messaggio allora si può utilizzare un separatore, come nel nostro caso la virgola. Per completezza mostriamo anche la tabella per la decodifica:

Valore codificato	Output	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	spazio	0.0	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				

3.6.3 Problemi pratici

Codificare e decodificare stringhe utilizzando la codifica aritmetica non è complicato ma risulta poco pratico: le macchine hanno una precisione finita, tipicamente riescono a rappresentare circa 80 bit in virgola mobile prima di andare in overflow. Questo problema implica resettare gli intervalli ogni 10-15 simboli, il che non è accettabile. Inoltre c'è da considerare anche il fatto che processori che gestiscono i numeri a virgola mobile in modo diverso potrebbero avere problemi nella comunicazione e nella rappresentazione di questi numeri.

Un'altra problematica è data dal fatto che il decodificatore deve aspettare il numero finale ottenuto dal codificatore, quindi deve aspettare che la codifica finisca, quando sarebbe comodo che lavorasse in modo concorrente con il codificatore. Questo diventa possibile solo utilizzando uno schema di codifica incrementale, ovvero uno schema in cui il codificatore man mano che codifica invia delle informazioni al decodificatore che intanto può iniziare il suo processo di decodifica.

Per semplificare il processo di codifica e decodifica, si passa da valori in virgola mobile a valori interi, facendo in modo che da operazioni costose (moltiplicazioni e divisioni) su decimali si passi ad addizioni, sottrazioni e shift su interi. Questo tipo di cambiamento permette anche di effettuare una codifica incrementale. Abbiamo visto che l'algoritmo funziona tenendo traccia di un estremo superiore e inferiore, che vengono fissati rispettivamente a 1 e 0. La prima semplificazione che facciamo è quella di scambiare 1 con 0.999... e 0 con 0.000..., rappresentando poi solo le cifre decimali, quindi tralasciando lo zero iniziale. Se lavoriamo ad esempio con variabili intere a 16 bit, vuol dire che possiamo tenere traccia solo delle prime cifre significative del valore perché in teoria sappiamo che continuano all'infinito.

Consideriamo allora la frase **BILL GATES** da codificare, utilizzando dei registri a 5 cifre decimali (useremo i decimali piuttosto che i binari per semplicità, ma il concetto è lo stesso). Avremo la seguente situazione:

```
HIGH: 99999 (e a seguire tanti 999999...)
LOW:  00000 (e a seguire tanti 000000...)
```

Per come abbiamo definito la nostra rappresentazione, sappiamo che il valore **HIGH** contiene in realtà 0.999... e il valore **LOW** contiene 0.000... Continuiamo allora ad applicare l'algoritmo, e quindi facendo la differenza tra questi due valori per ottenere l'intervallo. Intuitivamente sappiamo che la differenza è 100000, non 99999, perché assumiamo che la variabile **HIGH** abbia un numero infinito di 9, e quindi vale praticamente 1. Calcoliamo allora il nuovo valore usando l'istruzione corrispondente nell'algoritmo:

```
high = low + high_range(symbol)
```

3. Tecniche di codifica

In questo caso **high_range** valeva .30, per cui il nuovo valore di **HIGH** è 30000. Prima di salvare il dato dobbiamo decrementarlo per come abbiamo definito la nostra rappresentazione, per cui sarà 29999. Anche per **LOW** vale la stessa cosa, e infatti varrà 20000.

```
high: 29999 (999...)
low:  20000 (000...)
```

A questo punto, la cifra più significativa delle due variabili è la stessa, ed è 2. Per cui qualsiasi numero vado a inviare al decompressore so che inizierà con 2 (0.2 in realtà). Dal punto di vista pratico si invia 2 al decompressore come output cumulativo e si fa uno shift a sinistra di 1 cifra di entrambe le variabili **HIGH** e **LOW**, per cui le nuove cifre più significative saranno rispettivamente 9 e 0. Continuando a seguire questo metodo per tutta la frase si ottiene il risultato mostrato nella tabella seguente.

	High	Low	Range	Output cumulativo
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

Questo schema funziona molto bene per codificare incrementalmente un messaggio, ma c'è un problema quando le cifre più significative di **HIGH** e **LOW** non corrispondono. Infatti se per un certo numero di volte non corrispondono, non riesco a completare la codifica perché non ho abbastanza precisione.

Se la parola codificata ha una stringa di 0 o 9, i valori **HIGH** e **LOW** convergono lentamente allo stesso valore e andremo quindi a ridurre sempre di più l'intervallo. Se arrivo a un punto in cui **HIGH** vale 700004 e **LOW** vale 699995, l'intervallo calcolato sarà lungo solo una cifra, il che significa che la parola in output non avrà abbastanza precisione per essere codificata correttamente. Peggio ancora se arrivo in un punto in cui **HIGH** è 70000 e **LOW** è 69999, sono sicuro che nei prossimi passaggi non riuscirò a rappresentare questi valori con 5 cifre perché l'intervallo è diventato troppo piccolo.

È possibile risolvere questo problema di underflow facendo un altro controllo sulle cifre che non corrispondono. In particolare se le cifre più significative di **HIGH** e **LOW** sono adiacenti (ovvero differiscono per 1) allora andiamo a vedere la seconda cifra: se sono 0 e 9 allora sappiamo che si verificherà un underflow. Cancelliamo quindi la seconda cifra sia da **HIGH** che da **LOW** e facciamo lo shift a sinistra del resto delle cifre. Quindi cancello 0 e 9 e faccio uno shift a sinistra per far sì che **HIGH** e **LOW** ricevono un 9 e uno 0. La cifra più significativa rimane al suo posto. Usiamo una variabile chiamata *underflow*

counter che ci ricorda che abbiamo “buttato via” uno 0 o un 9, così quando andremo a inviare al decompressore il numero, dovremo ricordarci di mettere in quella posizione uno 0 o un 9. In tabella è mostrato questo processo:

	Prima	Dopo
High:	40344	43449
Low:	39810	38100
Underflow:	0	1

Dopo ogni ricalcolo si controlla se le cifre più significative non combaciano per verificare se c'è stato un altro underflow. Visto che qui le cifre più significative non coincidono e quelle immediatamente successive sono 0 per HIGH e 9 per LOW, cancelliamo 0 e 9, incrementiamo l'underflow counter e facciamo shift a sinistra. È possibile provare che questa soluzione risolve il problema.

Così non è molto chiaro il motivo per cui questa codifica sia migliore rispetto a Huffman. Consideriamo allora questo esempio. Se dobbiamo codificare il flusso **AAAAAAA**, e la probabilità di A è 9/10, c'è un 90% di probabilità che il prossimo carattere emesso dalla sorgente sia la A. Impostiamo la nostra tabella di probabilità in modo che A occupi l'intervallo da .0 a .9, e il simbolo di fine messaggio occupi l'intervallo da .9 a 1. Il processo di codifica è mostrato di seguito:

Nuovo carattere	Low	High
	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

Ora dobbiamo scegliere un numero per codificare il messaggio. Il numero .45 farà decodificare questo messaggio in modo univoco in "AAAAAAA". Queste due cifre decimali richiedono poco meno di 7 bit per la codifica, il che significa che abbiamo codificato 8 simboli in meno di 8 bit, mentre un messaggio con Huffman ottimale avrebbe richiesto un minimo di 9 bit.

Abbiamo visto queste due codifiche perché quando vedremo algoritmi di compressione per esempio JPEG o MPEG, scopriremo che anche se sono lossy hanno alla fine della loro compressione un passo che è relativo a una compressione ulteriore andando a comprimere in maniera lossless certe cose, usando Huffman o codifica aritmetica. In particolare JPEG fu introdotto come algoritmo che serviva per comprimere le foto digitali. Si decise che alla fine di JPEG ci voleva anche una fase entropica e si indicò nello standard due algoritmi possibili da usare nei compressor JPEG per la codifica entropica: Huffman o codifica aritmetica. A seconda dell'algoritmo utilizzato si parlava di compressione e decompressione JPEG Huffman/arithmetic coding. Arithmetic coding portava a una compressione maggiore ma non fu molto utilizzato per un problema di proprietà intellettuale, perché fu brevettata da dipendenti di IBM, la quale chiese una *loyalty* per permettere l'utilizzo di arithmetic coding, quindi tutti usavano Huffman. Quando IBM smise di chiedere le *loyalty* sull'utilizzo tutti passarono a arithmetic coding perché è più efficiente per la codifica entropica.

Capitolo 4

Metodi di sostituzione testuale on-line

I metodi visti finora sono considerati strumenti di codifica piuttosto che di compressione, che usavano un metodo statistico per codificare i simboli emessi dalla sorgente. La sostituzione testuale si basa sulla costruzione di un modello che viene poi effettivamente utilizzato per comprimere il testo.

4.1 Compressione basata su dizionari

I metodi di compressione basati su dizionari non vanno a codificare i caratteri sorgente come stringhe di bit a lunghezza variabile, ma prendono invece in input stringhe di lunghezza variabile dalla sorgente e li codificano come **indici**, o *tokens*. Se gli indici hanno una dimensione più piccola delle parole che vanno a rimpiazzare, otteniamo ovviamente un risparmio nella codifica e quindi compressione.

Questo modello è semplice da comprendere perché è quello che utilizziamo nella vita di tutti i giorni con gli elenchi telefonici o con un codice postale, ad esempio la stringa 84100 sappiamo che codifica la stringa "Città di Salerno".

Vediamo ora l'algoritmo di codifica, che legge un flusso di caratteri su Σ e scrive un flusso di bit, e l'algoritmo di decodifica, che riceve un flusso di bit e produce un flusso di caratteri su Σ .

Algoritmo di codifica

```
(1) Initialize the local dictionary  $D$  with the set  $INIT$ .

(2) repeat forever

    (a) Get the current match:
         $t := MH(inputstream)$ 
        Advance the input stream forward by  $|t|$  characters.
        Transmit  $\lceil \log_2 |D| \rceil$  bits corresponding to  $t$ .

    (b) Update the local dictionary  $D$ :
         $X := UH(D)$ 
        while  $X \neq \{\}$  and ( $D$  is not full or  $DH(D) \neq \{\}$ ) do begin
            Delete an element  $x$  from  $X$ .
            if  $x$  is not in  $D$  then begin
                if  $D$  is full then Delete  $DH(D)$  from  $D$ .
                Add  $x$  to  $D$ .
            end
        end

end
```

L'idea fondamentale è quella di avere da una parte un dizionario di frasi e dall'altra un insieme di frasi emesse dalla sorgente. Vogliamo codificare l'output della sorgente attraverso puntatori nel dizionario. Se il dizionario è conosciuto dal compressore e dal decompressore, possiamo usarlo per codificare e decodificare il testo da comprimere.

In generale le due parti non hanno un dizionario fisso durante l'esecuzione di questi algoritmi, ma lo compongono e lo modificano man mano che arriva altro output dalla sorgente.

Vediamo ora cosa fa l'algoritmo di codifica: **(1)** inizia con una fase di inizializzazione del dizionario locale, che chiameremo *INIT*, con un insieme iniziale di stringhe. **(2)** Poi inizia il ciclo principale (finché non finisce il messaggio emesso dalla sorgente). **(a)** Prendiamo il match corrente e vediamo qual è il match migliore tra la stringa di caratteri emessa dalla sorgente e quello che si trova nel dizionario. Trovato il match corrente, chiamato t , ci spostiamo di $|t|$ caratteri in avanti sullo spazio tra la fine di quella parola e la successiva, e trasmettiamo $\lceil \log_2 |D| \rceil$, che corrispondono al match appena trovato. **(b)** Poi si aggiorna il dizionario locale: la funzione *UH* (*Update Heuristic*) aggiunge al dizionario una o più frasi che dipendono dal match corrente. Chiamiamo questo insieme di stringhe X . Per aggiungere al dizionario si controlla se c'è spazio³ o se bisogna cancellare qualcosa (usando la *Deletion Heuristic*). In ogni caso viene aggiunto l'elemento x al dizionario.

Algoritmo di decodifica

- (1) Initialize the local dictionary D by performing Step 1 of the encoding algorithm.
- (2) **repeat forever**
 - (a) Get the current match:
Receive $\lceil \log_2 |D| \rceil$ bits.
Obtain the current match t by a dictionary lookup.
Output the characters of t .
 - (b) Update the local dictionary D by performing Step 2b of the encoding algorithm.

Il decompressore riceve una serie di indici e deve andare a sostituire ognuno di questi indici con un elemento del dizionario. L'*UH* e la *DH* utilizzate dal decompressore sono le stesse del compressore. **(1)** Il decompressore inizializza il dizionario locale D andando ad applicare lo step 1 dell'algoritmo di codifica. **(2)** Inizia poi il ciclo: **(a)** viene decompresso il match corrente, prendendo l'indice rappresentato come $\lceil \log_2 |D| \rceil$ bit e a partire da questo indice va a vedere nel dizionario qual è la frase a cui corrisponde. Questo sarà il match corrente i cui caratteri vengono mandati in output. **(b)** Dopodiché viene aggiornato il dizionario locale utilizzando lo step 2b dell'algoritmo di codifica.

4.1.1 Euristiche

- **Initialization heuristic, INIT:** serie di stringhe per inizializzare il dizionario locale. Normalmente l'alfabeto Σ sarà un sottoinsieme di *INIT* e quindi si avrà $|INIT| \leq |D|$.
- **Match heuristic, MH:** funzione che rimuove dall'input stream una stringa (il match corrente) che è già nel dizionario.
- **Update heuristic, UH:** funzione che prende il dizionario locale D e restituisce un insieme di stringhe che dovrebbero essere aggiunte al dizionario se riusciamo a trovare abbastanza spazio per inserirle.

³È buona norma avere un dizionario non troppo grande, altrimenti costerebbe troppo inviare gli indici.

- **Deletion heuristic, DH:** funzione che prende D come argomento e restituisce un insieme che è vuoto oppure che contiene una o più stringhe di D che non fanno parte di $INIT$ e che possono essere cancellate da D .

Per cui, ricapitolando, D è inizializzato per contenere almeno i caratteri di Σ (dato che Σ è un sottoinsieme di $INIT$) e questi caratteri non possono mai essere cancellati. Da questo deriva che la funzione MH è sempre ben definita, per cui ogni stringa che va in input all'algoritmo di codifica può sempre essere codificata (nel peggiore dei casi con un carattere alla volta). La codifica è univoca, e i dizionari locali del compressore e del decompressore sono sempre identici (questo perché lo step 2b dell'algoritmo di codifica è uguale allo step 2b dell'algoritmo di decodifica).

4.1.2 Puntatori a lunghezza variabile

Un'altra osservazione importante è che questi algoritmi passeranno puntatori di lunghezza variabile se $\lceil \log_2 |INIT| \rceil < \lceil \log_2 \langle D \rangle \rceil$. Infatti è possibile che inizialmente $|D|$ sia più piccolo di $\langle D \rangle$ e quindi i puntatori più corti di $\lceil \log_2 \langle D \rangle \rceil$ bit possono essere trasmessi. La dimensione dei puntatori aumenta di un bit ogni volta che $|D|$ raggiunge una potenza di 2 fino a quando $|D|$ raggiunge $\langle D \rangle$. Un uso più significativo dei puntatori a lunghezza variabile è quello di scegliere dinamicamente la dimensione di $\langle D \rangle$ oppure far crescere o diminuire di dimensione D . In questo caso viene usata un'euristica.

4.1.3 Metodi basati su static dictionaries

Esistono diversi tipi di dizionari. Con un dizionario statico compressore e decompressore condividono un dizionario che già è pieno. Utilizzando questo metodo non viene fatto l'aggiornamento del dizionario perché non cambia, la compressione infatti si basa solo su quello che ho a disposizione, che posso identificare con lo stesso insieme $INIT$.

Questo metodo è utile quando la sorgente è conosciuta a priori: in questo caso ha il vantaggio di essere veloce e semplice rispetto agli altri metodi. Inoltre non dovendo aggiornare il dizionario, non si possono verificare errori di inconsistenza tra i due dizionari di compressore e decompressore anche in caso di linea di comunicazione rumorosa o errori di comunicazione; cosa che si può verificare ad esempio con altri metodi.

Se la distribuzione delle voci nel dizionario statico non è uniforme nel testo sorgente, può essere vantaggioso usare puntatori di lunghezza variabile assegnando codici più brevi alle voci che si presentano più frequentemente.

4.1.4 Metodi basati su sliding dictionaries

L'algoritmo di compressione LZ77 è basato su questo metodo. Con il metodo dello *sliding dictionary*, $INIT = \Sigma$ e il resto del dizionario locale è visto come una finestra scorrevole che passa sopra le stringhe da comprimere. Consideriamo due parametri: $MAX_DISPLACEMENT$ e MAX_LENGTH (determinati dalla dimensione del dizionario locale), tali che il dizionario locale consiste di tutte le sottostringhe di lunghezza minore o uguale a MAX_LENGTH dei precedenti caratteri $MAX_DISPLACEMENT$ dell'input stream. Consideriamo un puntatore come una coppia di interi (m, n) dove m è lo spostamento indietro dalla posizione corrente al target del puntatore e n è la lunghezza del target del puntatore. Lo step 2b degli algoritmi di codifica e decodifica equivale a far scorrere una finestra (sliding window) sull'input stream: possiamo pensare come ai caratteri che entrano da destra e escono a sinistra. In questo caso allora $UH(D)$ è un insieme costituito da tutte le sottostringhe della finestra di lunghezza y o inferiore che si sovrappongono a t , quando t è concatenato all'estremità destra della finestra, e $DH(D)$ è un insieme di sottostringhe di lunghezza $|y|$ che si sovrappongono ai caratteri più a sinistra della finestra. Vediamo un esempio:

← coded text...

sir_sid_eastman_easily_t	eases_sea_sick_seals
--------------------------	----------------------

 ... ← text to be read

La finestra che sto guardando al momento, ovvero `MAX_DISPLACEMENT`, parte dalla stringa `sir` fino a `easily t`, contando quindi 24 caratteri all'indietro dalla `t`. Il parametro `MAX_LENGTH` mi dice invece quanto può essere lungo al massimo il match.

Il principio dell'algoritmo di compressione a sliding window è quello di usare una parte dello stream di input visto in precedenza come dizionario. Il compressore mantiene una finestra sullo stream di input e sposta l'input in quella finestra da destra a sinistra mentre le stringhe di simboli vengono codificate. La finestra è divisa in due parti: la parte a sinistra è il buffer di ricerca, ovvero il dizionario corrente, e include i simboli che sono stati recentemente inseriti e codificati. La parte a destra invece è il buffer look-ahead, che contiene il testo ancora da codificare. Nelle implementazioni pratiche il buffer di ricerca è lungo alcune migliaia di byte, mentre il buffer look-ahead è lungo solo decine di byte.

Nell'esempio che abbiamo visto, la barra verticale tra la `t` e la `t` rappresenta la linea di demarcazione attuale tra i due buffer. Assumiamo che il testo `sir sid eastman easily t` sia già stato compresso, mentre il testo `eases sea sick seals` deve ancora essere compresso.

Il compressore scansiona il buffer di ricerca all'indietro (da destra a sinistra) cercando una corrispondenza per il primo simbolo `e` nel buffer look-ahead. Ne trova uno alla `e` della parola `easily`. Questa `e` è ad una distanza (offset) di 8 dalla fine del buffer di ricerca. Il compressore quindi fa corrispondere il maggior numero possibile di simboli che seguono le due `e`. Tre simboli corrispondono a `eas` in questo caso, quindi la lunghezza del match è 3. Il compressore poi continua la scansione all'indietro, cercando di trovare match più lunghi. Nel nostro esempio, c'è un altro match alla parola `eastman`, con offset 16, che ha la stessa lunghezza. Il codificatore seleziona il match più lungo e prepara il token (16, 3, `e`). Scegliere l'ultimo match anziché il primo semplifica il codificatore, perché in questo modo deve solo tenere traccia dell'ultimo match trovato.

Limiti nell'approccio con sliding window

LZ77 usa l'assunzione implicita che i modelli nei dati di input si verifichino vicini l'uno all'altro. I flussi di dati che non soddisfano questo presupposto non vengono compressi correttamente: un esempio comune è il testo dove una certa parola, ad esempio *economia*, ricorre spesso ma è distribuita uniformemente in tutto il testo. Quando questa parola viene spostata nel buffer look-ahead, la sua occorrenza precedente potrebbe essere già stata spostata fuori dal buffer di ricerca. Un algoritmo migliore salverebbe le stringhe che ricorrono comunemente nel dizionario e non lo farebbe solo scorrere tutto il tempo.

Un altro svantaggio di LZ77 è la dimensione limitata L del buffer look-ahead. La dimensione delle stringhe che possono matchare è limitata a $L - 1$, ma L deve essere tenuta piccola perché il processo di match delle stringhe implica il confronto di singoli simboli. Se la dimensione di L fosse raddoppiata, la compressione migliorerebbe, poiché sarebbero possibili corrispondenze più lunghe, ma allo stesso tempo sarebbe molto più lento quando cerca match lunghi.

4.1.5 Metodi basati su dynamic dictionaries

L'algoritmo di compressione LZ78 è basato su questo metodo. Con il metodo della sliding window il dizionario viene aggiornato in modo molto ristretto: viene aggiunto un nuovo match alla destra della finestra e gli viene fatto spazio rimuovendo i caratteri dalla sinistra della finestra. Intuitivamente, il metodo del dizionario scorrevole elimina i caratteri all'estremità sinistra della finestra scommettendo che le sottostringhe lì presenti siano quelle che hanno meno probabilità di ripetersi di nuovo e aggiunge il match corrente all'estremità destra della finestra scommettendo che le sottostringhe lì

presenti siano le più probabili a ripetersi. Questa intuizione quindi ci porta a considerare un'euristica di aggiornamento e cancellazione più generale.

Update Heuristic

I metodi basati su dizionari dinamici si basano sul fatto che la *Update Heuristic* concatena il match precedente con un insieme di stringhe basate sul match corrente; ovvero se indichiamo con pm il match precedente, con cm il match corrente, e con INC una funzione di incremento che mappa una stringa a un insieme di stringhe, allora avremo che per qualche scelta di INC , varrà che:

$$UH(D) = pm \text{ concatenato con tutte le stringhe di } INC(cm)$$

Le seguenti sono tre scelte efficaci per la funzione INC :

- **FC, First Character heuristic** - $INC(cm)$ è il primo carattere di cm .
- **ID, Identity heuristic** - $INC(cm)$ è cm .
- **AP, All-Prefixes heuristic** - $INC(cm)$ è l'insieme di tutti i prefissi non vuoti di cm (incluso cm stesso).

Esempio: Supponiamo che il match precedente sia "THE_" e il match corrente sia "CAT" (usiamo un trattino basso per indicare uno spazio). Allora $UD(D)$ contiene i seguenti valori considerando le tre scelte appena viste:

- **FC:** "THE_C"
- **ID:** "THE_CAT"
- **AP:** "THE_C", "THE_CA", "THE_CAT"

In generale, le euristiche FC e ID producono sempre una sola stringa, mentre AP produce un insieme di stringhe la cui cardinalità è la lunghezza del match corrente. Infatti dall'esempio possiamo vedere come AP contenga le stringhe prodotte da FC e ID .

Deletion Heuristic

Tipicamente vengono utilizzate quattro tipologie di deletion heuristic:

- **FREEZE:** $DH(D)$ è la stringa vuota, per cui quando il dizionario è pieno rimane la stessa (viene freezeata) da quel momento in poi.
- **LRU, Least Recently Used:** $DH(D)$ è la stringa di D di cui è stato trovato un match meno recentemente di tutte.
- **LFU, Least Frequently Used:** $DH(D)$ è la stringa di D di cui è stato trovato un match meno frequentemente di tutte.
- **SWAP:** quando il dizionario primario diventa pieno, si inizia a riempire un dizionario secondario, ma la compressione basata sul primo dizionario viene fatta continuare normalmente. Ogni volta il dizionario ausiliario diventa pieno, si invertono dizionario primario e secondario, dopodiché si svuota il dizionario secondario.

4.1.6 LZ78

Il metodo LZ78 [Ziv e Lempel 78] non usa una sliding window, ma un dizionario di stringhe precedentemente incontrate. Questo dizionario inizia vuoto (o quasi vuoto), e la sua dimensione è limitata solo dalla quantità di memoria disponibile. Il compressore emette token che hanno due campi: il primo è un puntatore al dizionario, mentre il secondo è la codifica di un simbolo. I token non contengono la lunghezza di una stringa perché già è implicita nel dizionario. Ogni token corrisponde ad una stringa di simboli in ingresso, e questa stringa viene aggiunta al dizionario dopo che il token viene scritto sullo stream che già è stato compresso.

Quando si usa l'algoritmo LZ78, sia il codificatore che il decodificatore iniziano con un dizionario quasi vuoto. Per definizione, il dizionario ha una singola stringa codificata: la stringa nulla. Ogni carattere che viene letto viene aggiunto alla stringa corrente, e finché la stringa corrente ha un match con qualche frase nel dizionario questo processo continua. A un certo punto la stringa non avrà più una frase corrispondente nel dizionario, quindi vengono emessi un token e un carattere: in questo modo la stringa corrente è definita come quell'ultimo match con un nuovo carattere aggiunto. La nuova frase infine viene aggiunta al dizionario.

Esempio: supponiamo di dover codificare il testo DAD DADA DADDY DADO con LZ78. Il compressore inizia con un dizionario vuoto. Legge il primo carattere D (che non è presente nel dizionario) e crea una coppia (0,D). Di seguito vediamo la tabella che indica i valori utilizzati dall'algoritmo.

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	0	DADO

Il risultato finale degli indici è il seguente:

```

0  ''
1  D
2  A
3  D
4  DA
5  DA
6  DAD
7  DY
8  ''
9  DADO

```

4.1.7 Implementazione LZ78

Come in LZ77, LZ78 può impostare arbitrariamente la dimensione del dizionario delle frasi, e come LZ77, in LZ78 dobbiamo preoccuparci degli effetti di questo in due modi:

1. Innanzitutto, dobbiamo considerare il numero di bit allocati nel token di output per il codice della frase.
2. In secondo luogo, e cosa più importante, dobbiamo considerare quanto tempo richiederà la CPU per gestire il dizionario.

In teoria, LZ78 dovrebbe comprimere sempre meglio all'aumentare delle dimensioni del dizionario. Ma questo è vero solo perché la lunghezza del testo di input che tende all'infinito. Nella pratica, i file più piccoli inizieranno rapidamente a soffrire man mano che la dimensione del codice aumenta.

La vera difficoltà con LZ78 risiede proprio nella gestione del dizionario. Se ad esempio usiamo un codice a 16 bit per l'indice delle frasi, possiamo ospitare 65.536 frasi, incluso il codice nullo. Le frasi possono variare enormemente in lunghezza, inclusa l'improbabile possibilità di 65.536 versioni diverse di una frase composta da sequenze di un singolo carattere ripetuto. Queste frasi sono convenzionalmente memorizzate in un albero a più vie. L'albero inizia da un nodo radice, 0, che rappresenta la stringa nulla. Ogni possibile carattere che può essere aggiunto alla stringa nulla è un nuovo ramo dell'albero, con ogni frase creata in questo modo che ottiene un nuovo numero di nodo.

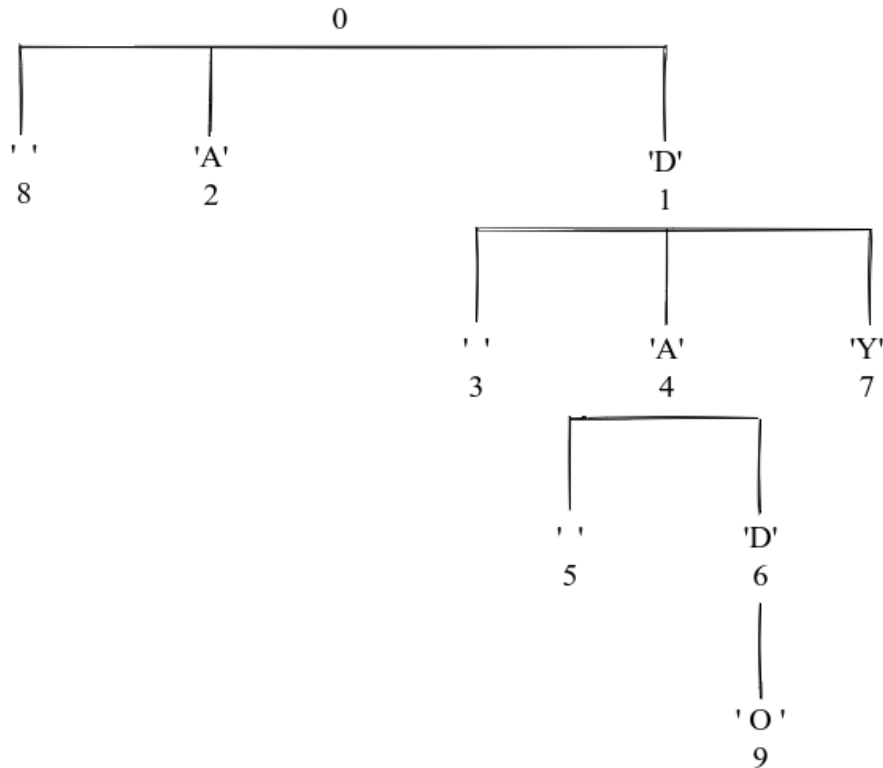


Figura 4.1: LZ78 Dictionary tree

4.1.8 Dizionario LZ78

La maggiore difficoltà nella gestione di un albero come questo è il numero potenzialmente elevato di rami uscente da ogni nodo. Quando si comprime un file binario con un alfabeto a otto bit, sono possibili 256 rami da ciascun nodo. Potremmo semplicemente allocare un array di indici o puntatori a ciascun nodo che sia abbastanza grande da ospitare tutti i 256 possibili discendenti. Ma poiché la maggior parte dei nodi non avrà così tanti discendenti, sarebbe incredibilmente dispendioso allocare così tanto spazio di archiviazione. Invece, i nodi discendenti sono generalmente gestiti come un elenco di indici non più lungo del numero di nodi discendenti che effettivamente esistono. Questa tecnica fa un uso migliore della memoria disponibile, ma è anche significativamente più lenta.

Con un albero come questo, confrontare una stringa esistente con il dizionario è semplice. È solo questione di camminare attraverso l'albero, attraversando un singolo nodo dell'albero per ogni carattere nella frase. Se la frase termina in un nodo particolare, abbiamo una corrispondenza. Se ci sono più frasi ma abbiamo raggiunto un nodo foglia, non c'è corrispondenza. Dopo che il simbolo è stato codificato, anche aggiungerlo al nodo foglia è semplice: basta aggiungere spazio all'elenco dei discendenti, quindi inserire un nuovo nodo discendente nell'ultimo nodo trovato.

Un effetto collaterale negativo di LZ78 non trovato in LZ77 è che anche il decoder deve mantenere questo albero. Con LZ77, un indice del dizionario era solo un puntatore o un indice a una posizione precedente nel flusso di dati. Ma con LZ78, l'indice è il numero di un nodo nell'albero del dizionario. Il decodificatore, quindi, deve mantenere l'albero esattamente allo stesso modo dell'encoder, altrimenti si verificherà un disastroso mismatch. Un altro problema finora ignorato è quello del riempimento del dizionario. Indipendentemente da quanto sia grande lo spazio del dizionario, prima o poi si riempirà. Se stiamo usando un codice a 16 bit, il dizionario si riempirà dopo aver definito 65.535 frasi. Ci sono diverse scelte alternative per quanto riguarda un dizionario completo.

Probabilmente la scelta predefinita più sicura è smettere di aggiungere nuove frasi al dizionario dopo che è pieno, ma abbandonare il dizionario potrebbe non essere la scelta migliore perché quando si va a comprimere grandi flussi di dati, potremmo vedere cambiamenti significativi nei caratteri dei dati in entrata. Ad esempio quando si comprime l'immagine binaria di un programma (come un file EXE), ci aspetteremmo di vedere un cambiamento importante nel modello statistico dei dati mentre ci spostiamo dalla sezione codice del file alla sezione dati. Se continuiamo a utilizzare il nostro dizionario di frasi esistente, potremmo essere bloccati con un dizionario obsoleto che non si comprime molto bene. Allo stesso tempo, dobbiamo stare attenti a non buttare via un dizionario che sta comprimendo bene. Il programma UNIX compress, che utilizza una variante LZ78, gestisce il problema del dizionario completo monitorando il rapporto di compressione del file. Se il rapporto di compressione inizia a deteriorarsi, il dizionario viene eliminato e il programma ricomincia da capo.

4.2 LZW (Lempel Ziv Welch)

Come con LZ77, LZ78 è stato pubblicato per la prima volta su una rivista di ricerca e discusso in modo molto tecnico e astratto. Fino al 1984 è stato una variante di LZ78 finché non ha fatto progressi nel mondo della programmazione. Questo è avvenuto quando Terry Welch ha pubblicato "A Technique for High Performance Data Compression" su IEEE Computer. Il lavoro sul programma UNIX compress è iniziato quasi immediatamente dopo la pubblicazione dell'articolo di Terry Welch e la tecnica descritta da Welch e l'implementazione in compress sono indicate come compressione LZW.

LZW ha migliorato la compressione LZ78 eliminando il requisito che ogni token emettesse un indice di un carattere. Infatti, sotto LZW, il compressore non emette mai singoli caratteri, ma solo frasi. Per fare ciò, il principale cambiamento in LZW consiste nel precaricare il dizionario delle frasi con frasi a

simbolo singolo pari al numero di simboli nell'alfabeto. Pertanto, non esiste un simbolo che non possa essere codificato immediatamente anche se non è già apparso nel flusso di input.

Esempio: Di seguito viene mostrata una stringa di esempio utilizzata per dimostrare l'algoritmo. La stringa di input è un insieme di parole inglesi da un dizionario ortografico, separate dal carattere " ". Al primo passaggio viene eseguito un controllo per vedere se la stringa " W" è presente nella tabella. Poiché non lo è, il codice per " " viene emesso in output e la stringa "W" viene aggiunta alla tabella. Poiché il dizionario ha i codici 0-255 già definiti come i 256 possibili valori di carattere, alla prima stringa viene assegnato il codice 256. Dopo che la terza lettera, 'E', è stata letta, il secondo codice per la stringa, "WE", viene aggiunto alla tabella e viene emesso il codice per la lettera 'W'.

Nella seconda parola vengono letti i caratteri " " e 'W', stringa corrispondente al numero 256. Viene quindi emesso il codice 256 e viene aggiunta una stringa di tre caratteri alla tabella delle stringhe. Il processo continua fino all'esaurimento della stringa e all'emissione di tutti i codici.

Stringa in input: " WED WE WEE WEB WET "

Caratteri in input	Codice in output	Nuovo valore del codice e stringa associata
" 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 4.3: Esempio LZW

Ora vediamo la decompressione: **Codici in input:** "WED<256>E<260><261><257>B<260>T"

Input/New code	Old code	String/Output	Carattere	New table entry
" "	" "	" "		
"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 4.4: Esempio decompressione LZW

4.3 The catch

Sfortunatamente, l'algoritmo di decompressione mostrato è un pò troppo semplice. Una singola eccezione nell'algoritmo di compressione LZW causa alcuni problemi nella decompressione. Ogni volta che il compressore aggiunge una nuova stringa alla tabella delle frasi, lo fa prima che l'intera frase sia stata effettivamente emessa nel file. Se per qualche motivo il compressore utilizzasse quella frase come codice successivo, il codice di espansione avrebbe un problema. Ci si aspetterebbe di decodificare una stringa che non era ancora nella sua tabella.

Sfortunatamente esiste un modo in cui ciò può accadere. Se nella tabella è già presente una frase composta da una coppia CHARACTER, STRING e il flusso di input vede una sequenza di CHARACTER, STRING, CHARACTER, STRING, CHARACTER, l'algoritmo di compressione emetterà un codice prima che il decompressore lo definisca.

Stringa in input: "IWOMBAT ... IWOMBATIWOMBATIXXX"

Character Input	New code value and associated string	Code Output
...I		
WOMBATA	300 = IWOMBAT	208 (IWOMBA)
.	.	.
.	.	.
...I	.	.
WOMBATI	400 = IWOMBATI	300 (IWOMBAT)
WOMBATIX	401 = IWOMBATIX	400 (IWOMBATI)

Tabella 4.5: Quando l'algoritmo di decompressione vede questo flusso di input, k prima decodifica il codice 300 ed emette la stringa IWOMBATI. Aggiungerà quindi la definizione per il codice 399 alla tabella, qualunque essa sia. Quindi indirizza il successivo codice di input 400 e scopre che non è nella tabella.

Fortunatamente, questa è l'unica volta in cui l'algoritmo di decompressione incontrerà un codice non definito. Poiché è l'unica volta, possiamo aggiungere un gestore di eccezioni all'algoritmo. L'algoritmo modificato cerca solo il caso speciale di un codice non definito e lo gestisce. Nell'esempio, la routine di decompressione vede un codice 400. Poiché 400 non è definito, il programma torna al codice/stringa precedente, che era "IWOMBAT", o al codice 300. Quindi aggiunge il primo carattere della stringa al fine della stringa, ottenendo "IWOMBATI", il valore corretto per il codice 400. L'elaborazione procede quindi normalmente.

Capitolo 5

Lossless image compression

La compressione dati senza perdita (o compressione dati lossless), in informatica e telecomunicazioni, è una classe di algoritmi di compressione dati che non porta alla perdita di alcuna parte dell'informazione originale durante la fase di compressione/decompressione dei dati stessi.

Un esempio di questo tipo di compressione è dato dai formati Zip, Gzip, Bzip2, Rar, 7z. I file per cui non è accettabile una perdita di informazione, come i testi o i programmi, utilizzano questo metodo. Per le immagini fotografiche generalmente non si usano algoritmi lossless in quanto sarebbero veramente poco efficienti, ma per le immagini che contengano ampie aree con colori puri spesso la compressione "senza perdita" non solo è applicabile, ma anche conveniente (GIF, PNG, MNG, TIFF con compressione LZW, ZIP o RLE).

5.1 Images

Le immagini sono considerate come un insieme di array bidimensionali di dati interi (i campioni), rappresentati con una data precisione (numero di bit per componente). Ogni array è definito come un componente e le immagini a colori hanno più componenti, che di solito risultano in una rappresentazione di uno spazio di colore (ad es. RGB, YUV, CMYK). Un'immagine a tono continuo, a sua volta, è un'immagine i cui componenti hanno più di un bit per campione.

5.2 Lossless image compression

La compressione delle immagini senza perdita di dati è richiesta (o desiderata) in applicazioni in cui le immagini sono soggette ad una ulteriore elaborazione (ad esempio, allo scopo di estrarre informazioni specifiche), editing intensivo o compressione/decompressione ripetuta. In genere è la scelta anche per immagini ottenute a caro prezzo, o in applicazioni dove la qualità desiderata dell'immagine renderizzata è ancora sconosciuta. Come, l'imaging medico, l'industria della pre stampa, i sistemi di archiviazione delle immagini, le preziose opere d'arte da preservare e le immagini telerilevate sono tutti candidati per la compressione senza perdita di dati. Ci si può aspettare di essere in grado di comprimere le immagini senza perdita di dati poiché i dati contengono ridondanze, nel senso che possono essere modellati in modo efficiente utilizzando distribuzioni non uniformi.

5.3 Modeling and Coding

Lo schema di compressione dei dati può essere suddiviso in due fasi:

- Modellazione: i dati sulla ridondanza vengono analizzati e presentati come un modello (trasformando i dati originali da una forma o rappresentazione ad un'altra);
- Codifica: viene codificata la differenza tra i dati effettivi e il modello (codeword).

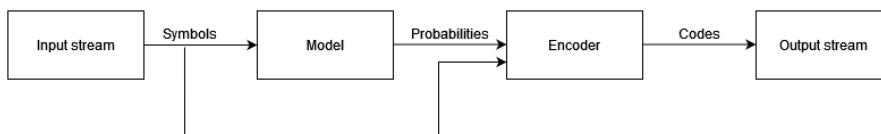


Figura 5.1: statistical model with Huffman encoder

Gli schemi di compressione delle immagini lossless più all'avanguardia possono essere suddivisi in due fasi distinte e indipendenti, modellazione e codifica, che sono spesso intercalate. Un'immagine viene osservata campione per campione in un ordine predefinito (normalmente raster-scan) e la fase di modellazione ha lo scopo di raccogliere informazioni sui dati, sotto forma di un modello probabilistico, che viene utilizzato per la codifica. La modellazione può essere formulata come un problema di inferenza induttiva, in cui ad ogni istante t , e dopo aver scansionato i dati passati $x^t = x_1 x_2 \dots x_t$, si vogliono fare inferenze sul valore campionario successivo x_{t+1} assegnando una probabilità condizionata distribuzione $P(\bullet|x^t)$ ad esso. Idealmente, la lunghezza del codice fornita da x_{t+1} è $-\log P(x_{t+1}/x^t)$ che fa la media dell'entropia del modello probabilistico.

Questa assegnazione del codice viene implementata nella fase di codifica. In uno schema sequenziale (o in linea), il modello per la codifica di x_{t+i} è adattivo e utilizza i valori dei campioni precedentemente codificati. Pertanto, può essere imitato dal decodificatore mentre decodifica la stringa passata, in sequenza.

In alternativa, in uno schema a due passaggi, un modello apprende dall'intera immagine in un primo passaggio e deve poi essere descritta al decodificatore come informazione di intestazione. La separazione concettuale tra le operazioni di modellazione e codifica è stata resa possibile dall'avvento dei codici aritmetici che possono realizzare qualsiasi modello probabilistico con una precisione prefissata.

Queste pietre miliari nello sviluppo della compressione dei dati senza perdita di dati hanno permesso ai ricercatori di considerare la codifica delle immagini semplicemente come un problema di assegnazione di probabilità, concentrandosi sulla progettazione di modelli fantasiosi per le immagini.

Tuttavia, quando si considera l'asse della complessità, la suddetta separazione tra modellazione e codifica diventa meno netta. Questo perché l'uso di un codificatore aritmetico generico, che abilita i modelli più generali, è escluso in molte applicazioni a bassa complessità, soprattutto per le implementazioni software.

5.3.1 Una strategia di modellazione

La maggior parte degli schemi più all'avanguardia (state of the art) si basa ancora su una strategia di modellazione in cui l'assegnazione di probabilità adattiva è suddivisa nelle seguenti componenti:

1. Una fase di previsione, in cui viene indovinato un valore \hat{x}_{t+1} per il campione successivo x_{t+1} sulla base di un sottoinsieme finito (un modello causale) dei dati passati disponibili.
2. La determinazione di un contesto in cui si verifica x_{t+1} . Il contesto è, di nuovo, una funzione di un modello causale (possibilmente diverso).

3. Un modello probabilistico per la previsione del residuo (o segnale di errore) $e_{t+1} \triangleq x_{t+1} - \hat{x}_{t+1}$, condizionato dal contesto di x_{t+1} .

La previsione rimanente è codificata in base alla distribuzione di probabilità progettata nel punto (3). La stessa previsione, contesto e distribuzione di probabilità sono disponibili per il decodificatore, che può quindi recuperare il valore esatto del campione codificato. La fase di previsione è, infatti, uno degli strumenti più vecchi e di maggior successo nella casella degli strumenti di compressione delle immagini. È particolarmente utile in situazioni in cui i campioni rappresentano una grandezza fisica che varia continuamente (ad esempio luminosità) e il valore del campione successivo può essere previsto con precisione utilizzando una semplice funzione (ad esempio una combinazione lineare) di campioni vicini osservati in precedenza. La consueta interpretazione dell'effetto benefico della previsione è che decora i campioni di dati, consentendo così l'uso di modelli semplici per l'errore di previsione.

Il progettista mira a una lunghezza del codice che si avvicini all'entropia empirica dei dati sotto il modello. Entropie inferiori possono essere ottenute attraverso contesti più ampi (vale a dire, un valore maggiore di), catturando dipendenze di ordine elevato, ma i risparmi di entropia potrebbero essere compensati da un costo del modello. Questo costo cattura le penalità della "context dilution" che si verificano quando le statistiche di conteggio devono essere distribuite su troppi contesti, influenzando così l'accuratezza delle stime corrispondenti. Al fine di bilanciare la suddetta "tensione" tra risparmio di entropia e costo del modello, la scelta del modello dovrebbe essere guidata dall'uso, ove possibile, delle conoscenze pregresse disponibili sui dati da modellare, evitando così inutili costi di "apprendimento" (cioè sovradattamento). Ciò spiega il relativo fallimento degli strumenti di compressione universali basati sugli algoritmi standard di sostituzione testuale o statistica quando applicati direttamente alle immagini naturali e la necessità di schemi progettati specificamente per i dati di immagine.

Ad esempio, un predittore lineare fisso riflette le conoscenze pregresse sull'uniformità dei dati. La conoscenza precedente può essere ulteriormente utilizzata adattando ai dati distribuzioni parametriche con pochi parametri per contesto. Questo approccio consente a un numero maggiore di contesti di acquisire dipendenze di ordine superiore senza penalità nel costo complessivo del modello. I moderni codificatori di immagini lossless di successo fanno ipotesi sofisticate sui dati da incontrare. Tuttavia, questi presupposti non sono troppo forti e, sebbene questi algoritmi siano principalmente destinati alle immagini fotografiche, si comportano molto bene su altri tipi di immagini come documenti composti, che possono includere anche porzioni di testo e grafica. Tuttavia, data la grande varietà di possibili sorgenti di immagini, non ci si può aspettare da un singolo algoritmo di gestire in modo ottimale tutti i tipi di immagini. Pertanto, esiste un compromesso tra ambito ed efficienza di compressione per tipi di immagine specifici, oltre ai compromessi tra efficienza di compressione, complessità computazionale e requisiti di memoria. Di conseguenza, il campo della compressione delle immagini senza perdita di dati è ancora aperto a ricerche fruttuose.

5.4 Lossless JPEG

Nel 1986, una collaborazione di tre importanti organizzazioni di standard internazionali (ISO, CCITT e IEC), ha portato alla creazione di un comitato noto come Joint Photographic Experts Group (JPEG), con una carta per sviluppare uno standard internazionale per la compressione e la decompressione di immagini a tono continuo, fermo immagine, monocromatiche e a colori. Lo standard JPEG risultante include quattro metodi di compressione di base: tre di essi (codifica sequenziale, codifica progressiva, codifica gerarchica) sono basati sulla trasformata del coseno discreta (DCT) e utilizzati per la compressione con perdita. Il quarto, JPEG senza perdita di dati, è parzialmente basato su un algoritmo di compressione senza perdita di dati chiamato Sunset.

					Selection value	Prediction
					0	no prediction
					1	A
					2	B
					3	C
					4	$A + B - C$
					5	$A + ((B - C)/2)$
					6	$B + ((A - C)/2)$
					7	$(A + B)/2$

	C	B				
	A	X				

(a)
(b)

Figura 5.2: Esempio di lossless jpeg

La figura precedente mostra un pixel X e tre pixel vicini A, B e C. Sono mostrati otto possibili modi (predizioni) per combinare i valori dei tre vicini. Nella modalità JPEG lossless, l'utente può selezionare una di queste previsioni e l'encoder la utilizzerà per combinare i tre pixel vicini e sottrarre la combinazione dal valore di X. Il risultato è normalmente un numero piccolo, che è quindi entropia-codificata tramite Huffman o codifica aritmetica. Anche se il gap di compressione tra la versione di Huffman e quella di codifica aritmetica è significativa, quest'ultima non ha ottenuto un uso diffuso, probabilmente a causa dei suoi requisiti di maggiore complessità e per problemi di proprietà intellettuale relativi all'uso del codificatore aritmetico (IBM). Il predittore 0 viene utilizzato solo nella modalità gerarchica di JPEG. I predittori 1, 2 e 3 sono chiamati unidimensionali. I predittori 4, 5, 6 e 7 sono bidimensionali. Va notato che la modalità senza perdita di JPEG non ha mai avuto molto successo. Produce fattori di compressione tipici di 2 o meno ed è quindi inferiore ad altri metodi di compressione delle immagini senza perdita di dati. Per questo motivo, molte implementazioni JPEG non implementano nemmeno questa modalità.

5.5 FELICS

L'algoritmo FELICS (Fast, Efficient Lossless Image Compression System) [Paul Howard e Jeff Vitter] può essere considerato un primo passo per colmare il divario di compressione tra schemi basati sulla semplicità e schemi basati sulla modellazione del contesto e sulla codifica aritmetica, poiché incorpora l'adattività in un framework a bassa complessità. La sua compressione sulle immagini fotografiche non è lontana da quella fornita da JPEG senza perdita di dati (versione con codifica aritmetica) e si dice che il codice venga eseguito fino a cinque volte più velocemente. Per motivi di complessità, FELICS evita l'uso di un codice aritmetico generico e utilizza almeno un bit per ogni campione codificato. Di conseguenza, non funziona bene su immagini altamente comprimibili.

FELICS

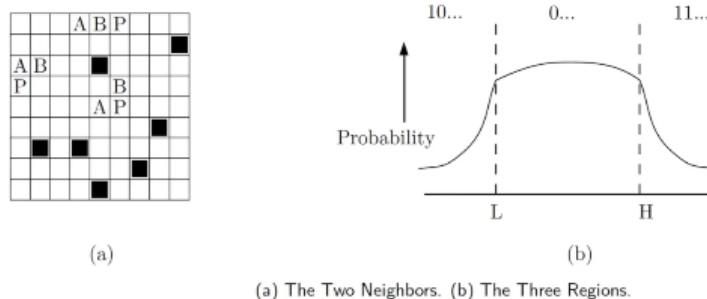


Figura 5.3: Esempio di FELICS

Considera i due vicini A e B di un pixel P. Usiamo A, B e P per denotare sia i tre pixel che le loro intensità (valori in scala di grigi). Indichiamo con L e H i vicini con l'intensità minore e maggiore, rispettivamente. Al pixel P dovrebbe essere assegnato un codice di lunghezza variabile a seconda di dove si trova l'intensità P rispetto a L e H. Abbiamo quindi tre casi:

1. L'intensità del pixel P è compresa tra L e H (si trova nella regione centrale della figura). È noto sperimentalmente che questo caso si verifica in circa la metà dei pixel e a P viene assegnato, in questo caso, un codice che inizia con 0. (Un caso speciale si verifica quando $L = H$. In tal caso, l'intervallo $[L, H]$ consiste di un solo valore e la possibilità che P abbia quel valore è piccola.) La probabilità che P si trovi in questa regione centrale è quasi, ma non completamente, piatta, quindi a P dovrebbe essere assegnato un codice binario che ha circa le stesse dimensioni in tutta la regione ma leggermente più corta al centro della regione.
2. L'intensità di P è inferiore a L (P è nella regione sinistra). Il codice assegnato a P in questo caso inizia con 10.
3. L'intensità di P è maggiore di H. A P viene assegnato un codice che inizia con 11.

Quando il pixel P si trova in una delle regioni esterne, la probabilità che la sua intensità differisca di molto da L o H è piccola, quindi in questi casi a P può essere assegnato un codice lungo.

Il codice assegnato a P dovrebbe quindi dipendere fortemente dal fatto che P si trovi nella regione centrale o in una delle regioni esterne. Ecco come viene assegnato il codice quando P si trova nella regione centrale:

Abbiamo bisogno di codici a lunghezza variabile $H-L+1$ che non differiranno molto in termini di dimensioni e, ovviamente, soddisferanno la proprietà del prefisso, poniamo $k = \lfloor \log_2(H - L + 1) \rfloor$ e calcoliamo gli interi a e b con $a = 2^{k+1} - (H - L + 1)$, $b = 2(H - L + 1 - 2^k)$.

5.5.1 Esempio: codice binario corretto

Se $H - L = 9$, allora $k = 3$, $a = 2^3 + 1 - (9 + 1) = 6$ e $b = 2(9 + 1 - 2^3) = 4$. Selezioniamo ora i codici $2^k - 1, 2^k - 2, \dots$ espressi come numeri a k bit e i codici $b, 1, 2, \dots$ espressi come numeri $(k + 1)$ a bit. Nell'esempio sopra, i codici a sono $8 - 1 = 111$, $8 - 2 = 110$, fino a $8 - 6 = 010$ e i codici b , 0000 , 0001 , 0010 e 0011 . I codici funzione a sono assegnati ai valori di P nel mezzo della regione centrale, e i b codici lunghi sono assegnati a valori di P più vicini a L o H . Si noti che b è pari, quindi i codici b possono sempre essere suddivisi in due insiemi uguali. **$L = 15$, $H = 24$**

Pixel P	Region code	Pixel code
L = 15	0	0000
16	0	0010
17	0	010
18	0	011
19	0	100
20	0	101
21	0	110
22	0	111
23	0	0001
H = 24	0	0011

Tabella 5.1: I codici per la regione centrale

5.6 Le regioni esterne (outer regions)

Quando P si trova in una delle regioni esterne, diciamo quella superiore, al valore $P-H$ dovrebbe essere assegnato un codice di lunghezza variabile la cui dimensione può crescere rapidamente man mano che $P - H$ diventa più grande. Un modo per farlo è selezionare un piccolo intero non negativo m (tipicamente $0, 1, 2$ o 3) e assegnare all'intero n un codice a due parti. La seconda parte sono gli m bit inferiori di n e la prima parte è il codice unario di $[n$ senza i suoi m bit inferiori]. Esempio: Se $m = 2$, allora $n = 1101$ è assegnato al codice 110101 , poiché 110 è il codice unario di 11 . Questo codice è un caso speciale del codice Golomb⁴, dove il parametro b è una potenza di 2 (2^m). La tabella seguente mostra alcuni esempi di questo codice per $m = 0, 1, 2, 3$ e $n = 1, 2, \dots, 9$. Il valore di m utilizzato in un particolare lavoro di compressione può essere selezionato, come parametro, dall'utente.

Pixel P	P-H	Region code	m =			
			0	1	2	3
H + 1 = 25	1	11	0	00	000	0000
26	2	11	10	01	001	0001
27	3	11	110	100	010	0010
28	4	11	1110	101	011	0011
29	5	11	11110	1100	1000	0100
30	6	11	111110	1101	1001	0101
31	7	11	1111110	11100	1010	0110
32	8	11	11111110	11101	1011	0111
33	9	11	111111110	111100	11000	10000
...

Tabella 5.2: I codici per le regioni esterne

⁴Il codice di Golomb prende in input un parametro b e codifica $x > 0$ come segue: codifica un valore $q+1$ in unario dove $q = (x+1/b)$ e un valore r in binario dove $r = x - q*b - 1$. Nel codice di Rice che si basa su Golomb si ha che $b = 2^m$.

L'idea chiave è assumere, per ogni campione, una distribuzione di probabilità come mostrato nella figura 5.3 dove L e H indicano, rispettivamente, il minimo e il massimo tra i valori del campione nelle posizioni A e B; e il decadimento su entrambi i lati è come esponenziale e simmetrico. Pertanto, FELICS si discosta dal tradizionale paradigma di codifica predittiva in quanto vengono utilizzati non uno, ma due "valori di previsione". Il tasso di decadimento dipende da un contesto, determinato dalla differenza $\Delta = H - L$. Si assume una probabilità prossima a 0,5 per la parte centrale della distribuzione, portando ad uno schema di codifica che utilizza un bit per indicare se il valore del campione corrente x si trova tra H e L. In questo caso, viene utilizzato un codice binario corretto, poiché si presume che la distribuzione sia quasi piatta.

Altrimenti, un bit aggiuntivo indica se è al di sopra o al di sotto dell'intervallo $[L, H]$ e un codice Golomb specifica il valore di " $x - H - 1$ " o " $L - x - 1$ ". I codici Golomb sono i codici ottimali per le distribuzioni geometriche di interi non negativi. Dato un parametro intero positivo m , il codice Golomb di ordine m -esimo codifica un intero in due parti: una rappresentazione unaria di $\lfloor y/m \rfloor$ e una rappresentazione binaria modificata di $y \bmod m$ (usando $\lceil \log m \rceil$ bit se $y < 2^{\lceil \log m \rceil} - m$ e $\lceil \log m \rceil$ bit altrimenti). FELICS utilizza il caso speciale dei codici Golomb con $m = 2^k$, che porta a procedure di codifica/decodifica molto semplici. Il parametro k viene scelto come quello che avrebbe funzionato meglio per le precedenti occorrenze del contesto.

5.7 JPEG-LS

5.7.1 Un nuovo standard di compressione delle immagini senza perdita.

La modalità lossless di JPEG è inefficiente e spesso non viene nemmeno implementata. Di conseguenza, l'ISO, in collaborazione con l'IEC, ha deciso di sviluppare un nuovo standard per la compressione lossless (o quasi lossless) delle immagini a tono continuo. Nel 1994, il comitato JPEG ha sollecitato proposte per un nuovo standard internazionale per la compressione delle immagini senza perdita di tono continuo. Hanno risposto alla sollecitazione soltanto otto organizzazioni industriali e accademiche che hanno presentato un totale di nove proposte, tra cui CALIC e LOCO-I. Due proposte si basavano sulla codifica trasformata reversibile (reversible transform coding), mentre le altre si basavano su una codifica predittiva, adattativa e basata sul contesto o su blocchi. Come risultato del processo di standardizzazione, LOCO-I è l'algoritmo alla base del nuovo standard, denominato JPEG-LS. È stato selezionato per la sua collocazione ottimale in una curva concettuale compressione/complessità, entro pochi punti percentuali dei migliori rapporti di compressione disponibili (dati da CALIC), ma a un livello di complessità molto inferiore.

JPEG-LS esamina molti dei vicini visti in precedenza del pixel corrente e li usa come contesto del pixel. Usa il contesto per prevedere il pixel e per selezionare una distribuzione di probabilità tra molte di queste distribuzioni e usa quella distribuzione per codificare l'errore di previsione con uno speciale codice Golomb. Esiste anche una modalità di esecuzione, in cui viene codificata la lunghezza di una sequenza di pixel identici.

Il contesto utilizzato per prevedere il pixel x corrente è mostrato nella figura successiva. Il codificatore esamina i pixel di contesto e decide se codificare il pixel corrente x in modalità di esecuzione o in modalità normale. Se il contesto suggerisce che i pixel y, z, \dots seguono il pixel corrente allora sono probabilmente identici, l'encoder allora seleziona la modalità di esecuzione (run). In caso contrario, seleziona la modalità normale. Nella modalità quasi senza perdite la decisione è leggermente diversa.

Se il contesto suggerisce che i pixel che seguono il pixel corrente sono probabilmente quasi identici (all'interno del parametro di tolleranza NEAR), l'encoder seleziona la modalità di esecuzione. In caso contrario, seleziona la modalità normale. Il resto del processo di codifica dipende dalla modalità selezionata.

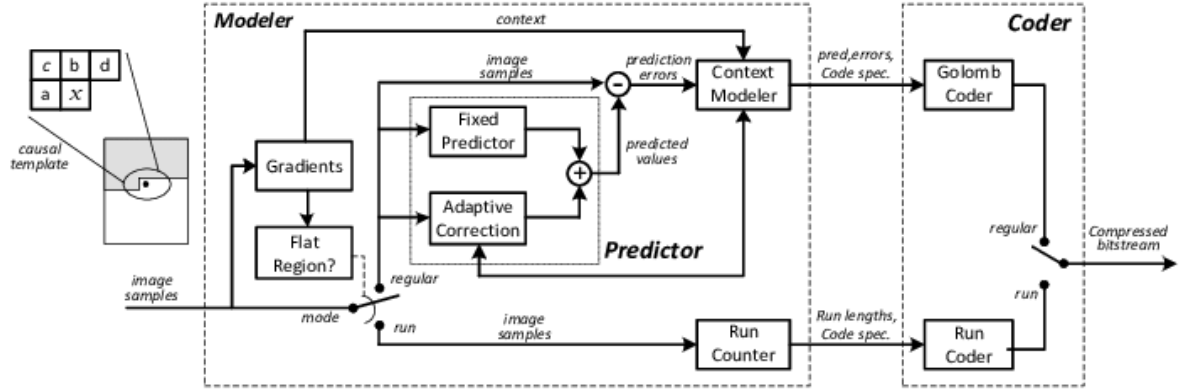


Figura 5.4: JPEG-LS encoder

5.7.2 Predittore, errore di previsione e codifica

La componente fissa del predittore switcha tra tre semplici predittori (I_w , I_n e $I_w + I_n - I_{nw}$), risultanti in una funzione non lineare dei campioni nel modello causale, data da:

$$\hat{x}_{MED} \triangleq \min(I_w, I_n, I_{nw}) + \max(I_w, I_n, I_{nw}) - I_{nw}.$$

Questa funzione è denominata rilevatore di bordi mediani (MED = median edge detector), poiché incorpora conoscenze pregresse attraverso una capacità rudimentale di rilevamento dei bordi. Nella modalità normale, il codificatore utilizza i valori dei pixel di contesto per prevedere il pixel x e sottrae la previsione da x per ottenere l'errore di previsione. Questo errore viene quindi corretto da un termine che dipende dal contesto (questa correzione viene eseguita per compensare errori sistematici nella previsione) e codificato con un codice Golomb. La codifica Golomb dipende da tutti e quattro i pixel del contesto e anche da errori di previsione che erano stati precedentemente codificati per lo stesso contesto.

5.8 Lo standard JPEG

Lo standard JPEG è uno standard per la compressione di immagini a tono continuo (fotografie). JPEG ha quattro modi di funzionamento (e molte opzioni):

- Baseline;
- Progressive Encoding;
- Hierarchical (Pyramidal) Encoding;
- Lossless Encoding.

Il modo base lossy di funzionamento è quello "baseline". La codifica di un'immagine nel modo baseline è costituita da sei passi.

JPEG, nella sua versione baseline, è uno degli standard di compressione più utilizzati in assoluto, ne sono state sviluppate molteplici implementazioni sulle più svariate architetture computazionali.

Lo scopo del Joint Photographic Experts Group (JPEG), che lavorava sotto gli auspici dell'ISO ed in stretta coordinazione con CCITT SGVIII, è stato quello di sviluppare uno standard general-purpose per la compressione di immagini che potesse essere applicato a qualsiasi problema di trasmissione o storage di immagini.

Per fare ciò JPEG si basa su anni di ricerca precedente nel campo della compressione di immagini ed introduce numerose innovazioni che hanno portato alla realizzazione di uno standard ad elevate performance e bassa complessità computazionale ancora oggi largamente in uso.

Gli scopi di JPEG sono i seguenti (Wallace 1991; Pennebaker and Mitchell 1993):

- Raggiungere una compressione ed una qualità dell'immagine decompressa prossimi allo stato dell'arte con immagini compresse fino a raggiungere qualità classificabile come "very good" o "excellent";
- Essere utilizzabile per la compressione di qualsiasi immagine a toni continui, sia a più toni di grigio che a colori, con qualsiasi spazio dei colori e di qualsiasi dimensione;
- Avere una complessità computazionale che ne permetta una implementazione software sulla maggior parte delle architetture computazionali moderne ed anche implementazioni hardware poco costose.

5.9 Baseline JPEG

La struttura del codificatore e del decodificatore JPEG sono illustrati nella prossima immagine, FDCT è l'acronimo di forward DCT ed IDCT di inverse DCT.

La codifica baseline di JPEG è chiamata sequential encoding. Per prima cosa l'immagine è partizionata in blocchi di 8 x 8 pixel che sono ordinati in ordine "rasterlike": da sinistra a destra, dall'alto in basso. La FDCT viene calcolata su ciascuno dei blocchi di pixels 8 x 8, ed i risultanti 64 coefficienti della DCT sono quantizzati scalarmente utilizzando tabelle uniformi di quantizzazione basate su esperimenti psicovisuali (Lohscheller 1984).

Queste tabelle scalari di quantizzazione sono fornite come parte dello standard ma il loro uso non è obbligatorio: l'utente potrà utilizzare anche tabelle specifiche da lui scelte.

Dopo che i coefficienti della DCT sono stati quantizzati, i coefficienti di ciascun blocco sono ordinati tramite una "zigzag scan".

Il bitstream risultante viene codificato con la tecnica della codifica runlength così da generare una sequenza di simboli intermedi e poi questi simboli sono infine codificati con Huffman.

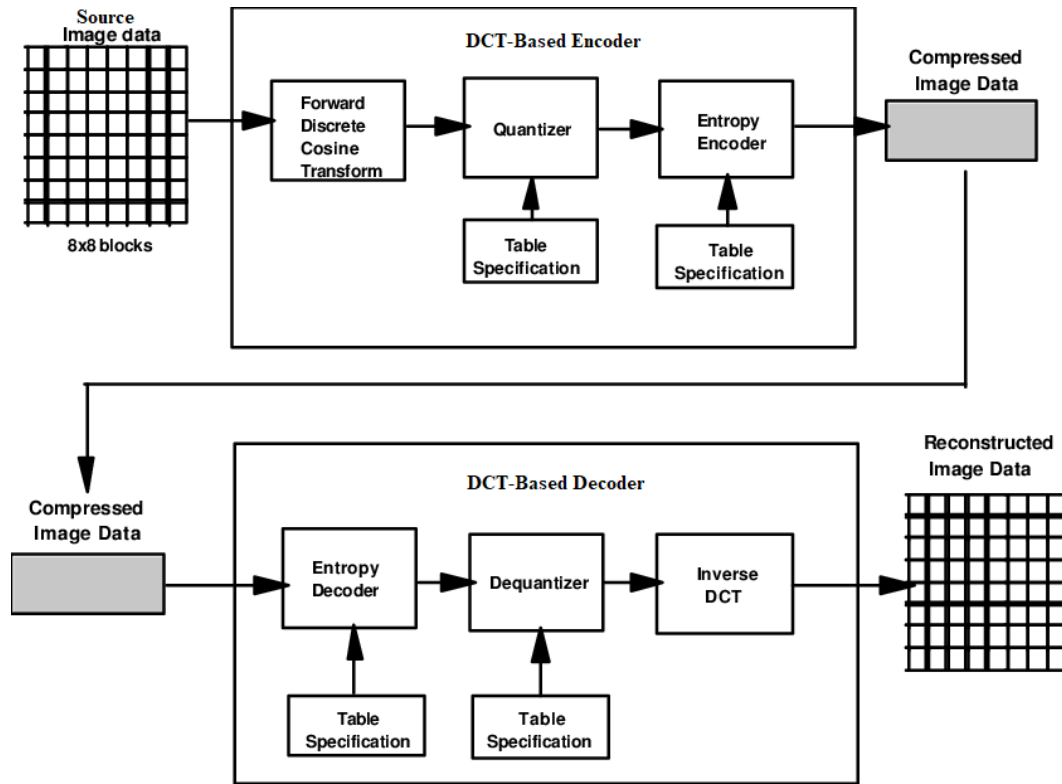


Figura 5.5: JPEG ENC_DEC

5.9.1 Funzionamento codifica baseline

Il passo 1 dell'algoritmo consiste nella preparazione dei blocchi. Supponiamo che l'input JPEG sia un'immagine RGB 640 X 480, con 24 bit/pixel. Dai valori RGB si calcolano la luminanza e due segnali di cromaticità. Chiamati rispettivamente, per NTSC: Y, I, Q e per PAL: Y, U, V (trasformazione dello spazio cromatico). Si costruiscono matrici separate per Y, I, Q. Ciascuna conterrà elementi aventi valori compresi tra 0 e 255.

Si effettua la media su quadrati di 4 pixel nelle matrici di I e Q, per ridurle a 320 X 240 (downsampling). Si sottrae 128 da ciascun elemento di tutte e tre le matrici, per porre lo zero al centro dell'intervallo. Infine ciascuna matrice è divisa in blocchi 8X8. La matrice Y ha 4800 blocchi, le altre due ne hanno 1200 ciascuna.

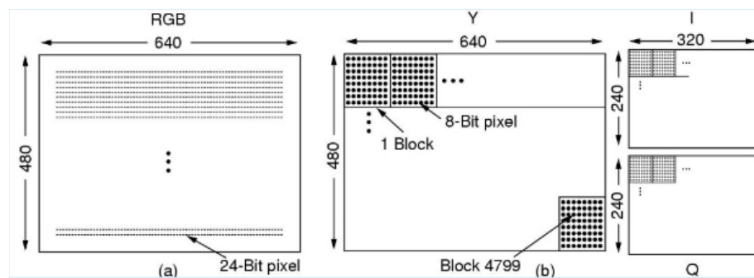


Figura 5.6:

Il passo 2 consiste nell'applicare la DCT (Trasformata Coseno Discreta) a ciascuno dei 7200 blocchi separatamente (figura 5.7).

- L'output di ciascuna DCT è una matrice 8X8 di coefficienti DCT;
- L'elemento DCT(0,0) è il valore medio del blocco, e gli altri elementi esprimono quanta potenza spettrale è presente in ciascuna frequenza spaziale;
- In teoria la DCT non genera perdita di informazione ma in pratica le computazioni necessarie introducono errori di arrotondamento.

Nel passo 3 si effettua la quantizzazione (Figura 5.8):

- i coefficienti DCT meno importanti sono eliminati;
- Questa trasformazione viene effettuata dividendo ciascuno dei coefficienti della matrice DCT 8X8 per un peso preso da una tabella detta tabella di quantizzazione ed arrotondandoli all'intero più vicino;
- Questo è il procedimento fondamentale che porta all'eliminazione dell'informazione meno significativa e permette di comprimere l'immagine.

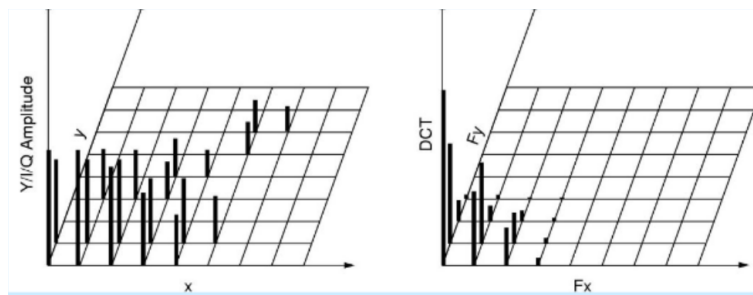


Figura 5.7: coefficienti DCT

Nel passo 4 si riduce il valore (0,0) di ciascun blocco rimpiazzandolo con la quantità di cui differisce

DCT Coefficients	Quantized coefficients	Quantization table
150 80 40 14 4 2 1 0	150 80 20 4 1 0 0 0	1 1 2 4 8 16 32 64
92 75 36 10 6 1 0 0	92 75 18 3 1 0 0 0	1 1 2 4 8 16 32 64
52 38 26 8 7 4 0 0	26 19 13 2 1 0 0 0	2 2 2 4 8 16 32 64
12 8 6 4 2 1 0 0	3 2 2 1 0 0 0 0	4 4 4 4 8 16 32 64
4 3 2 0 0 0 0 0	1 0 0 0 0 0 0 0	8 8 8 8 8 16 32 64
2 2 1 0 0 0 0 0	0 0 0 0 0 0 0 0	16 16 16 16 16 16 32 64
1 1 0 0 0 0 0 0	0 0 0 0 0 0 0 0	32 32 32 32 32 32 32 64
0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	64 64 64 64 64 64 64 64

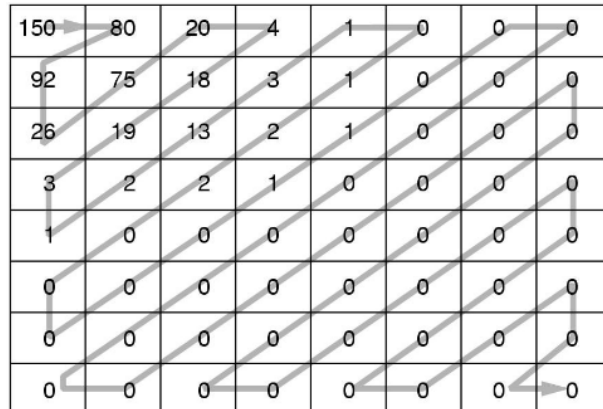
Figura 5.8: calcolo dei coefficienti DCT quantizzati

dall'elemento corrispondente del blocco precedente. I valori (0,0) sono detti componenti DC, gli altri valori AC.

Nel passo 5 si linearizzano i 64 elementi (zig-zag scan) e si applica alla lista un algoritmo di run length encoding.

Nel passo 6 si utilizza l'algoritmo di Huffman (o arithmetic coding) per codificare il risultato del passo 5.

I dati codificati sono ora pronti per la memorizzazione e/o la trasmissione (Figura 5.9). L'algoritmo di decodifica compie all'indietro gli stessi passi. JPEG è, a grandi linee, simmetrico cioè richiede lo stesso lavoro sia per codificare che per decodificare un'immagine.



150	80	20	4	1	0	0	0
92	75	18	3	1	0	0	0
26	19	13	2	1	0	0	0
3	2	2	1	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Figura 5.9: Ordine di trasmissione

Capitolo 6

Compressione video

6.1 Standard MPEG

Gli standard MPEG sono i principali algoritmi utilizzati per comprimere video digitali, e sono standard internazionali dal 1993. MPEG-1 è stato progettato per una qualità di uscita da video-registratore (352x240 per NTSC) a 1,2 Mbps. MPEG-2 è stato progettato per comprimere i segnali di qualità video televisiva da 4 a 6 Mbps. Entrambi si avvantaggiano dei due tipi di ridondanza: Ridondanza spaziale e ridondanza temporale.

6.1.1 Origini dello standard

Nel 1988 l'ISO-IEC (International Organization for Standardization and International Electrotechnical Commission) decise di sviluppare: uno standard per la compressione e la rappresentazione del video digitale e dell'audio ad esso associato che fosse adatto alla memorizzazione su dispositivi di memoria di massa (dischi ottici, DAT) e alla trasmissione su canale di telecomunicazione (ISDN, LAN, TV). La nascita di uno standard si era resa necessaria principalmente per due motivi: per assicurare l'interoperabilità tra diversi sistemi hardware e software minimizzando l'effetto degli interessi di parte; per garantire sia i fornitori che gli utenti di prodotti multimediali. Il Moving Picture Expert Group (MPEG) è il comitato internazionale nato in seno all'ISO per raggiungere tale obiettivo. Formalmente MPEG è il gruppo di lavoro 11 del subcomitato 29 del Joint Technical Group 1 dell'ISO-IEC (ISO-IEC/JTC1/SC29/WG11). Il lavoro di questo gruppo di esperti ha portato al rilascio di varie versioni dello standard MPEG.

6.2 MPEG

E' stato progettato per la codifica in forma digitale di immagini in movimento e per l'audio ad esse associato. E' nato principalmente per la diffusione di contenuti multimediali tramite CD-ROM a singola velocità: infatti esiste un insieme di parametri (constrained parameter set) pensato appositamente per questo tipo di applicazione che presenta un bitrate di circa 1,5 Mbps. Tra i principali limiti di questo standard: non prevede la modalità interlacciata. Non è previsto alcun supporto alla rivelazione di errori e alla perdita di informazioni che possono avvenire su canali geografici e collegamenti radio.

La parte video è stata completata nell'aprile 1993 ma è divenuta standard internazionale con la sigla IS-13818 nel 1995. MPEG-2, nato per trasmissioni video digitali e per la diffusione televisiva, supporta un ampio intervallo di risoluzioni e di bitrate. Consente la modalità interlacciata e il trasporto di più flussi tra loro indipendenti su di un mezzo non affidabile e soggetto a perdita di informazione. Siccome le applicazioni di questo standard sono molteplici, si è pensato di definire dei Profili ben precisi che individuano univocamente delle modalità operative. Queste modalità permettono che un flusso sia visto

da più utenti con risoluzioni e qualità diverse a seconda delle caratteristiche del canale trasmissivo e del ricevitore: in questo modo le nuove applicazioni rimangono compatibili con i vecchi apparati.

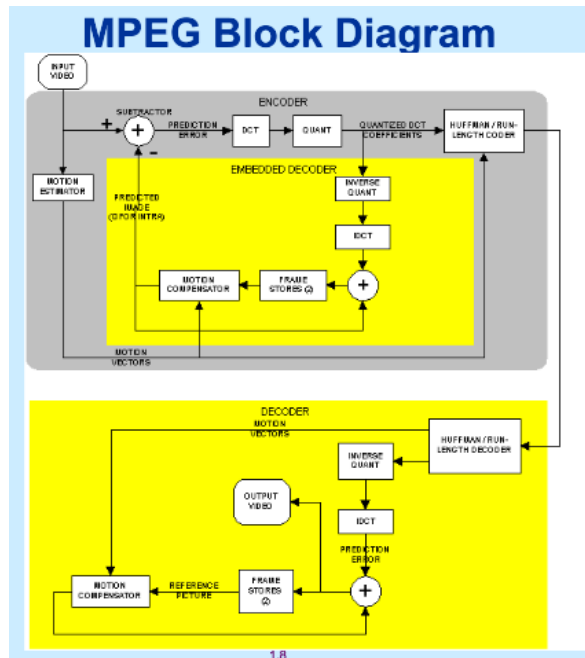
6.2.1 Principi generali della codifica MPEG video

Si presuppone che il valore di un particolare pixel di un'immagine può essere predetto tramite: i pixel adiacenti (tecniche di compressione intra-frame), il valore dei pixel di un frame vicino temporalmente (tecniche di compressione inter-frame). Intuitivamente appare chiaro che in alcune circostanze, per esempio durante un cambio scena in una sequenza, la correlazione temporale tra pixel corrispondenti di frame successivi è molto piccola o scompare del tutto. In questo e in simili casi le tecniche di codifica intra-frame sono le più appropriate per sfruttare la correlazione spaziale e ottenere una compressione efficiente.

MPEG utilizza la trasformata coseno discreta (Discrete Cosine Transform o DCT) su blocchi quadrati di 8 pixel di lato per sfruttare la correlazione spaziale tra pixel vicini della stessa immagine. Tuttavia se la correlazione tra i pixel di immagini vicine è alta, cioè nel caso che due frame consecutivi siano molto simili o al limite identici, è preferibile usare anche tecnica di codifica inter-frame di tipo DPCM (Differential PCM) che impiega una predizione temporale (predizione con compensazione del moto tra frame). Per ottenere un'elevata compressione MPEG video usa uno schema di codifica che è una combinazione adattativa di tecniche di predizione temporale con compensazione del moto seguite da una codifica con trasformata DCT delle rimanenti informazioni spaziali si ottiene una codifica ibrida del video DPCM/DCT.

6.2.2 Esempio di codifica DPCM

La figura seguente mostra un esempio di codifica DPCM. Per codificare il campione n -esimo del segnale di ingresso si cerca una sua predizione dagli $n-1$ campioni precedenti già codificati. Quindi si calcola la differenza tra la predizione e il campione reale e si codifica solo la differenza trovata. Il decoder, che usa lo stesso algoritmo, predice il campione n -esimo dai campioni precedenti già decodificati, quindi gli somma algebricamente la differenza decodificata, ottenendo così esattamente il segnale di partenza. Il motivo che spinge a usare questa tecnica sta nel fatto che viene ridotta la dinamica (cioè l'intervallo di variabilità) del valore da codificare, cosicché è necessario un minor numero di bit per rappresentarlo. Di fianco abbiamo MPEG Block Diagram.



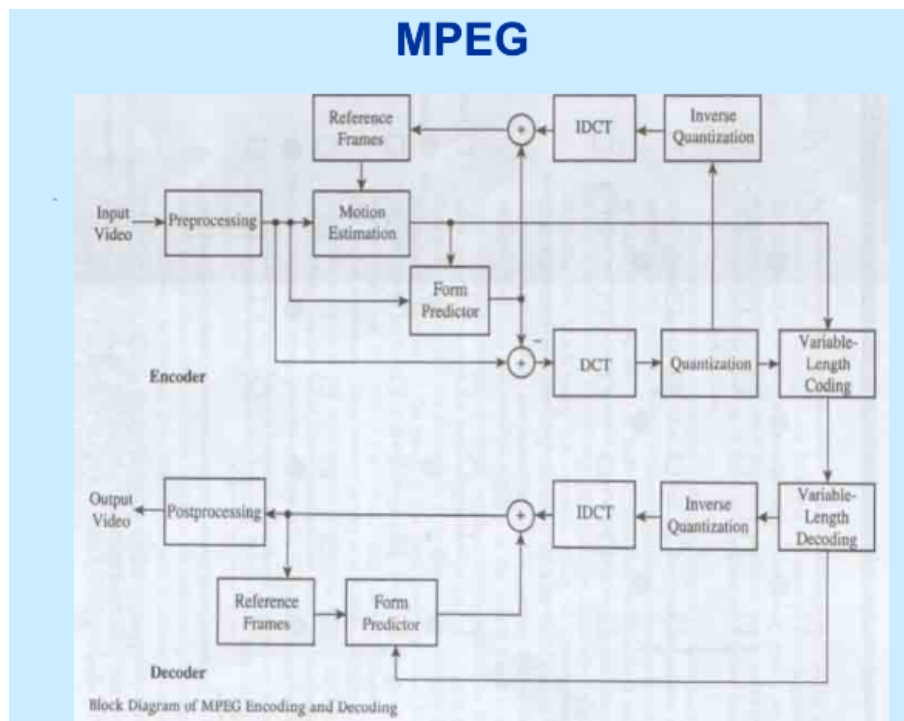


Figura 6.1: MPEG encoder/decoder

Il problema principale nella realizzazione di MPEG è il seguente: si vuole raggiungere un'elevata compressione, cosa non realizzabile se le immagini che compongono la sequenza video vengono codificate ognuna in modo indipendente dalle altre; inoltre si vuole garantire la possibilità di effettuare un accesso casuale, cosa che si realizza meglio se si codificano le immagini in modo indipendente. L'uscita di MPEG-2 si compone di tre diversi tipi di frame, che devono essere elaborati dal programma di visualizzazione: I (frame intracodificati): immagini fisse autocontenute codificate tramite JPEG P (frame predittivi): differenze blocco per blocco con l'ultimo frame. B (frame bidirezionali): differenze rispetto al frame precedente e a quello successivo.

6.2.3 Standard MPEG 2: Frame I

I frame I sono immagini fisse codificate con un algoritmo simile a JPEG. Utilizzano la luminanza a piena risoluzione e la crominanza a metà risoluzione, lungo ciascun asse. I frame I devono apparire periodicamente in uscita per tre ragioni: 1. In una trasmissione televisiva, con gli spettatori che si sintonizzano quando lo desiderano, se tutti i frame dipendessero dai precedenti fino al primo, chiunque abbia perso il primo frame non potrebbe mai decodificare i successivi, e ciò non permetterebbe agli spettatori di sintonizzarsi dopo che il film è iniziato. 2. Se un frame fosse ricevuto erroneamente non sarebbe possibile decodificare i successivi. 3. Mentre si effettua un avanti o indietro veloce, il decodificatore dovrebbe calcolare ciascun frame su cui è passato, se vuole conoscere quello su cui si è fermato, ma con i frame I è possibile saltare avanti o indietro fino a trovare un frame I, ed iniziare da quel punto la visualizzazione.

6.2.4 Standard MPEG: Frame P

I I frame P codificano le differenze tra frame consecutivi. Si basano sull'idea di macroblocchi. Un macroblocco è codificato cercando nel precedente frame il macroblocco stesso, o qualcosa che differisce poco da esso.



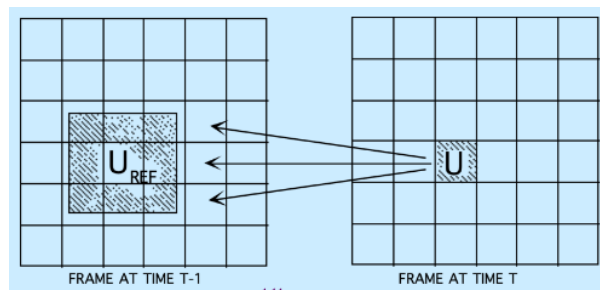
6.2.5 Compensazione del moto

Le tecniche più usate sono del tipo block matching, il vettore di moto è ottenuto minimizzando una opportuna funzione di costo. La predizione nella compensazione del moto significa assumere che, localmente, l'immagine attuale possa essere divisa in parti che possano essere modellate come una traslazione di parti di un'immagine precedente. L'interpolazione è una delle principali caratteristiche dell'algoritmo MPEG, i macroblocchi interpolati sono formati da una media tra la predizione di un frame passato e la predizione di un frame futuro. $U = MxNU_{ref} = (M + 2p)x(N + 2p)$

Distortion Function: $D(i, j) = \frac{1}{MN} \sum_{m=1}^M \sum_{n=1}^N g(U(m, n)_{ref} - U(m + i, n + j))$

DMD: DIrection of minimum Distortion

Trovare la DMD per un blocco => calcolo di $D(i, j)$ in $(2p + 1) \times (2p + 1)$ posizioni



6.2.6 Standard MPEG: Frame B

I frame B sono simili frame interpolati a partire da macroblocchi di riferimento che sono nel frame precedente (o in un altro frame precedente) e in quello successivo (o in un altro frame successivo) che possono essere I oppure P frame.

6.2.7 I livelli della sintassi MPEG video

La sintassi MPEG è una sintassi stratificata. Vengono infatti definiti diversi livelli incapsulati uno dentro l'altro. Ad ogni livello corrisponde un header seguito dai dati. In cima alla gerarchia si trova la sequenza (cui corrisponde un sequence header), a questo livello si definiscono la dimensione delle immagini, il frame rate, il bit rate. La sequenza è suddivisa in gruppi di immagini (group of pictures o GOP); un GOP è un insieme di immagini in ordine contiguo di codifica (che può essere diverso dall'ordine di visualizzazione). Ciascun GOP contiene tre tipi di immagini: I, irta P, predette B, interpolate bidirezionalmente. Ogni immagine è composta da tre componenti: una di luminanza e due di crominanza (con eventuale sottocampionamento).

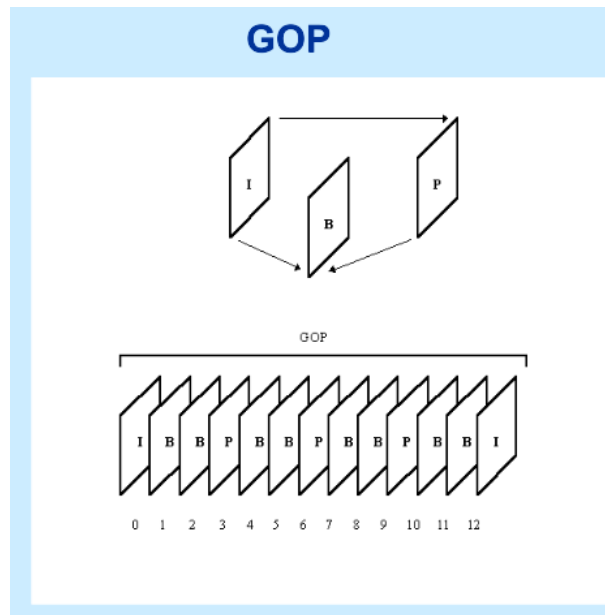


Figura 6.2: GOP

I frame sono a loro volta suddivisi in slice. Gli slice header contengono delle informazioni importanti per la sincronizzazione e per il controllo degli errori. La dimensione delle slice non è standardizzata e in MPEG-1 una slice poteva anche stare su più righe. MPEG-2 stabilisce invece che una slice deve iniziare e finire sulla stessa riga. Le slice contengono un numero intero di macroblocchi.

Un macroblocco contiene un'area quadrata di 16x16 pixel di luminanza e le corrispondenti aree di crominanza. I macroblocchi sono l'unità base che si usa per la compensazione del moto tra frame: e nel macroblock header che vengono codificati i vettori di moto. I macroblocchi sono composti da blocchi di 8x8 pixel. I blocchi sono l'unità elementare su cui si calcola la DCT e su cui viene applicata la quantizzazione. Poiché MPEG-1 supportava solo il formato 4:2:0 in ingresso, da ogni macroblocco si ottenevano sempre sei blocchi (4 di luminanza e 2 di crominanza). MPEG-2 supporta invece anche i formati 4:2:2 e 4:4:4; in quei casi ogni macroblocco a formato rispettivamente da 8 0 12 blocchi.