Università di Catania

Facoltà di Scienze MM.FF.NN.
Corso di Laurea in Informatica

Tesi di Laurea

Lo Standard JPEG2000: Applicazioni ed Estensioni

Gaetano Impoco

Relatore

Prof. Giovanni Gallo

Correlatore

Dott. Sebastiano Battiato

Ringraziamenti

Desidero ringraziare il Prof. Gallo e il Dott. Battiato per l'aiuto e i preziosi consigli che mi hanno permesso di migliorare e portare a termine questo lavoro. Ringrazio i dottorandi del Dipartimento di Matematica e Informatica dell'Università di Catania e i membri del gruppo DSC della STMicroelectronics di Catania per la disponibilità e l'amicizia dimostratemi.
Ringrazio tutti coloro che, durante gli anni di studio trascorsi all'università, hanno creduto in me e mi hanno sostenuto.
Un ringraziamento particolare ai miei genitori, i cui sforzi mi hanno permesso di raggiungere questa tappa importante della mia vita, e alla mia famiglia che ha saputo sopportarmi pazientemente.
Un grazie speciale a Gianmarco per l'amicizia fraterna e per essere stato sempre per me un punto di riferimento e confronto.

Sommario

Negli ultimi anni la comunicazione ha assunto un ruolo centrale sia nel mondo economico che nella vita quotidiana. La crescente diffusione di Internet ha aperto la strada alla multimedialità e all'interattività sottolineando l'importanza della comunicazione visiva. In questo contesto, le immagini digitali hanno assunto un ruolo principe ed hanno un numero sempre crescente di applicazioni. Si pensi, ad esempio, alla diffusione che le fotocamere digitali e gli apparati mobili quali i telefoni cellulari hanno avuto negli ultimi anni. Nasce, a questo punto, l'esigenza di un sistema flessibile, capace di gestire immagini provenienti da sorgenti diverse e in grado di adattarsi a condizioni di visualizzazione diverse.
In particolare i requisiti fondamentali richiesti ad uno standard per il trattamento di immagini digitali sono, soprattutto, qualità adeguata anche a bassi bit-rate, scalabilità e codifica progressiva dell'informazione, codifica differenziata di zone più importanti, robustezza alla propagazione degli errori.
Le immagini digitali, non opportunamente codificate, richiedono un cospicuo impiego di risorse, sia in termini di memoria in un calcolatore che di larghezza di banda in un canale di trasmissione. Per ridurre il numero di bit necessari a rappresentare un'immagine, sono state studiate varie tecniche di compressione specifiche, alcune delle quali con perdita di informazione. Tuttavia, la compressione dei segnali digitali, mentre ottimizza il segnale compresso, rende generalmente difficile l'elaborazione delle immagini codificate.
Gli standard di compressione esistenti sono inadeguati a fronteggiare le due esigenze contrapposte discusse sopra: flessibilità e risparmio di risorse. Per tale ragione l'organizzazione di standardizzazione ISO ha chiesto al comitato JPEG (Joint Photographic Expert Group) di studiare un nuovo sistema di compressione che fosse capace di garantire flessibilità unita a fattori di compressione adeguati. Il risultato di questo studio è lo standard JPEG2000.
Il nuovo standard non mira a superare quelli precedenti in termini di compressione ma fornisce, senza sacrificare l'efficienza della compressione, una serie di caratteristiche quali, ad esempio, scalabilità per risoluzione e qualità, accesso casuale a piccole porzioni dell'immagine, compressione *lossy* e *lossless*. Ovviamente, questo obiettivo può essere raggiunto solo a patto di accettare una notevole complessità computazionale.
Scopo di questo lavoro è illustrare le caratteristiche salienti dello standard JPEG2000, con approfondimenti nelle parti più interessanti e innovative, e presentare uno studio su una possibile ottimizzazione della qualità visiva delle immagini compresse mediante JPEG2000.
Nel primo capitolo sarà introdotto il concetto di compressione, con particolare attenzione ai segnali acustici e visivi, e verrà illustrato lo schema di compressione basato sulle trasformazioni lineari del segnale dal dominio spaziale (o temporale) a quello delle frequenze.

Il secondo capitolo descrive lo schema generale di codifica e decodifica e tratta le prime fasi della compressione: trasformazione dello spazio dei colori, trasformata wavelet e quantizzazione.
I tre capitoli successivi sono dedicati alla compressione entropica del segnale. In particolare, il terzo capitolo introduce l'idea dell'algoritmo e si sofferma sui dettagli della modellizzazione dei contesti, il quarto capitolo descrive la codifica aritmetica e il quinto tratta della distribuzione dei dati compressi nel flusso in uscita.
Nel sesto capitolo saranno discusse due estensioni al sistema di base: la codifica di regioni dell'immagine con una migliore qualità (regioni di interesse) e la codifica di immagini indicizzate.
Il settimo capitolo fornisce alcuni esempi che illustrano le caratteristiche di JPEG2000 e un confronto qualitativo con lo standard JPEG. Nell'ultima parte del capitolo è presentata un'analisi della complessità degli algoritmi di codifica e decodifica JPEG2000.
Infine, nell'ottavo capitolo è presentato un algoritmo innovativo, sviluppato nel corso della redazione di questa tesi, che cerca di ottimizzare la qualità visiva di un'immagine compressa mediante JPEG2000, a parità di dimensione dei dati codificati.

Indice

Capitolo 1

Compressione di Segnali Digitali

In questo capitolo introdurremmo il concetto di compressione e presenteremo il paradigma di compressione basata sulla trasformazione del segnale, mediante una trasformata lineare, da un dominio (tipicamente spaziale o temporale) in un altro che permetta di metterne meglio in evidenza alcune caratteristiche utili alla compressione.

1.1. Introduzione alla Compressione

Il principio su cui si basa la compressione dei dati è l'eliminazione della ridondanza contenuta nell'informazione. L'obiettivo è rappresentare un segnale utilizzando il minor numero possibile di simboli, in accordo a determinati vincoli di qualità sul segnale ricostruito.
Una buona compressione può essere ottenuta con tecniche *lossless* (senza perdita), in cui i dati decodificati restituiscono una copia esatta di quelli originali. La compressione lossless è necessaria in applicazioni in cui è importante che l'informazione sia decodificata in maniera fedele (ad es. compressione di archivi), tuttavia questo requisito limita la performance degli algoritmi di compressione che si basano su tali tecniche.
Rilassando questo vincolo di qualità e permettendo che i dati ricostruiti differiscano da quelli originali entro una certa soglia di errore, i dati possono essere compressi con efficienza crescente all'aumentare della tolleranza sull'errore nel segnale ricostruito. Poiché è necessario eliminare parte dei dati, il problema che si pone nella compressione *lossy* (con perdita) è quello di discriminare l'informazione essenziale da quella meno importante, al fine di poter scartare i dati meno utili alla ricostruzione fedele dei dati originali, per ottenere, a parità di dimensione dei dati compressi, la migliore qualità possibile.
Quando trattiamo segnali acustici o visivi, nella valutazione dell'errore è opportuno scegliere una misura che rifletta le caratteristiche del sistema percettivo dell'osservatore. Ad esempio, una caratteristica del sistema visivo umano è la diminuzione di sensibilità al contrasto al crescere della frequenza del segnale. Un codificatore di immagini potrebbe sfruttare questa peculiarità utilizzando più bit per codificare le basse frequenze rispetto a

quelli riservati alle alte frequenze, ottenendo in tal modo, a parità di *bit-rate*[1], una maggiore *qualità percepita*. In altre parole, la misura della distorsione deve essere consistente con ciò che l'osservatore è in grado di percepire. Tutto ciò che non può essere "catturato" dall'osservatore può essere rimosso senza rischi. Infatti, poiché l'osservatore non è in grado di percepire l'errore commesso eliminando tale informazione, dal suo punto di vista il segnale originale e quello "ripulito" dall'informazione superflua sono indistinguibili.
Ovviamente, le tecniche che permettono di mettere in evidenza l'informazione più significativa di un segnale dipendono dalle caratteristiche del segnale stesso, quindi è conveniente considerare classi di *sorgenti*[2] che abbiano proprietà statistiche simili. Per questo esistono molti sistemi di codifica specializzati nella compressione di classi specifiche di segnali, che ottengono performance migliori rispetto ai compressori generici, anche nel caso di compressione lossless. Questi sistemi traggono vantaggio dal comportamento tipico del segnale, utilizzando modelli capaci di prevedere la distribuzione dei coefficienti. Utilizzando questa informazione, assegnano codifiche brevi a distribuzioni con probabilità più alta e, viceversa, codifiche lunghe a sequenze meno probabili.

1.2. Trasformazione Lineare e Compressione

Uno degli approcci più comuni alla compressione basata su modelli statistici del segnale è quello descritto nel seguente diagramma:

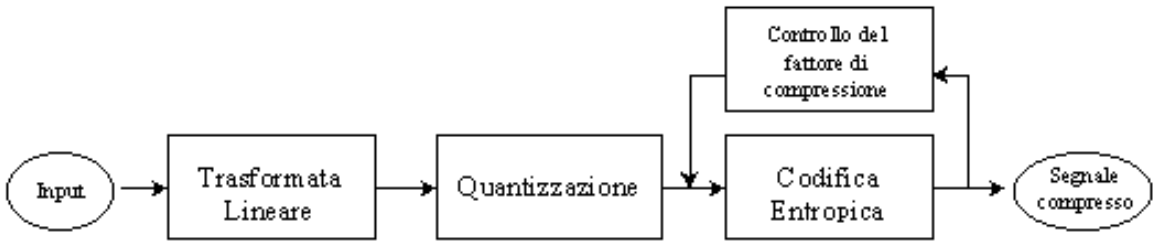

Figura 1: Sequenza di passi di un generico algoritmo di compressione basato sulla trasformata lineare del segnale.

I passi rilevanti, rappresentati da rettangoli nel diagramma precedente, hanno il seguente significato:

1. *Trasformata Lineare*: il segnale sorgente è decomposto nelle sue componenti di frequenza mediante una trasformazione lineare (tipicamente Discrete Cosine Transform o Discrete Wavelet Transform). La trasformazione lineare del segnale ha due scopi:
   - *decorrelazione del segnale*, in modo tale che il valore di ogni coefficiente trasformato sia statisticamente indipendente dal valore degli altri coefficienti.
   - *compressione dell'energia*, in maniera tale che la maggior parte dell'energia del segnale si concentri in una piccola porzione dei coefficienti trasformati. La Figura 2 mostra un esempio di trasformata lineare (DCT) in cui l'energia dell'immagine si

[1] Numero di simboli usati per rappresentare un campione del segnale. Nell'image processing, il bit-rate tipicamente è misurato in bit/pixel.
[2] Per "sorgente" si intende qualunque collezione di dati, siano essi provenienti, ad esempio, da un file nel disco di una workstation o da un canale di comunicazione.

concentra nei coefficienti a frequenza più bassa (nell'angolo in alto a sinistra). Un altro esempio di trasformata lineare (DWT) è quello di Figura 10. Anche in questo caso, mediante l'impiego di opportuni filtri, l'energia si concentra in una porzione ristretta dei coefficienti trasformati.

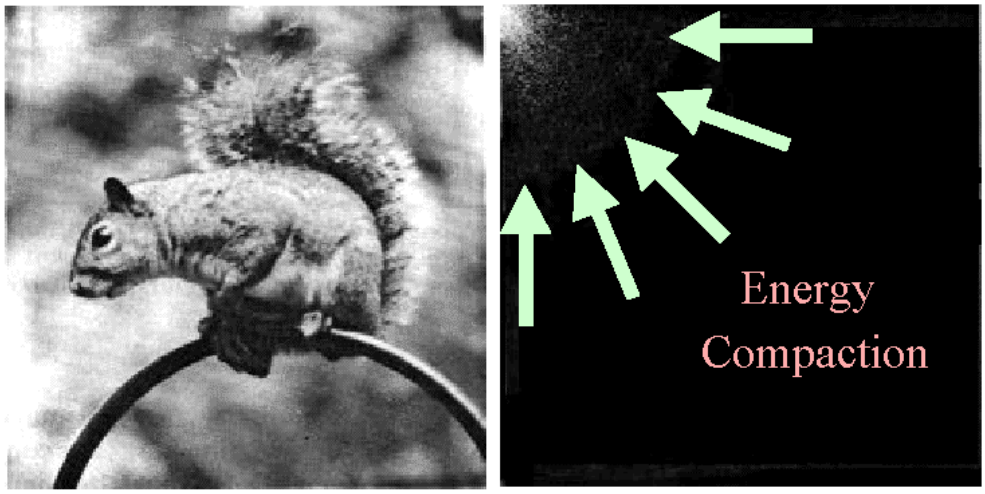

Figura 2: Immagine rappresentata : (a) nel dominio spaziale (b) nel dominio delle frequenze. L'energia è concentrata in una piccola frazione dei coefficienti trasformati.

2. *Quantizzazione*: dai coefficienti del segnale, rappresentati nel dominio delle frequenze, vengono "tagliati" i bit meno significativi, in maniera uniforme o adattativa rispetto alla frequenza di ogni singola componente del segnale. La quantizzazione è il passo fondamentale nella compressione lossy. É questa operazione, infatti, che consente di eliminare l'informazione meno importante, per ottenere una maggiore compressione. Per questo motivo, nella compressione lossless questo passo è omesso. La decorrelazione, dovuta alla trasformazione del segnale, permette di quantizzare i coefficienti trasformati in maniera indipendente gli uni dagli altri (quantizzazione scalare), il che rende più veloce il processo di codifica (e decodifica).
3. *Codifica Entropica*: i coefficienti quantizzati sono codificati in base a proprietà statistiche del segnale, secondo un modello noto a priori o in base a informazioni raccolte durante una fase di pre-processing (come per l'encoder di Huffman), in modo tale da ridurre la ridondanza nella rappresentazione del segnale. La codifica entropica trae vantaggio dalla compressione dell'energia, dovuta alla trasformazione lineare del segnale, perché la rappresentazione nel dominio trasformato gode della proprietà che l'energia del segnale si concentra in un piccolo numero dei coefficienti trasformati.
4. *Controllo del Fattore di Compressione*: è utilizzato solo quando la dimensione dei dati compressi è fissata a priori, indipendentemente del processo di codifica. Il controllo del fattore di compressione è usato, tipicamente, in applicazioni soggette a limitazioni per quanto riguarda la memorizzazione o la trasmissione del segnale compresso. Si pensi, ad esempio, ad una fotocamera digitale. Una funzione importante di una macchina fotografica è quella di fornire all'utente informazioni sullo stato della memoria. Tuttavia, ciò che l'utente vuole sapere è il numero di scatti rimanenti, non la quantità di bytes liberi. In un'applicazione di questo tipo, evidentemente, il controllo del bit-rate è fondamentale, perché è il meccanismo che ci permette di stabilire, data la quantità di memoria libera, quante foto possono ancora essere memorizzate.

Il seguente diagramma mostra la sequenza di passi dell'algoritmo di decompressione, duale del precedente.

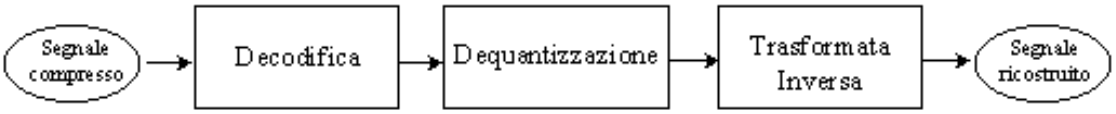

Figura 3: Sequenza di passi dell'algoritmo di decompressione duale al precedente.

Una nota importante riguarda il controllo del fattore di compressione. Mentre ogni passo di codifica trova nel decoder un passo duale (e questi passi sono eseguiti in ordine inverso rispetto a quello dell'encoder), il passo di controllo del bit-rate appare solo nell'encoder. Il decoder, infatti, per ricostruire il segnale, ha bisogno di conoscere solamente il fattore di quantizzazione usato per comprimere il segnale, non il metodo usato per trovarlo. In questo modo, l'encoder è libero di utilizzare uno schema di quantizzazione che si adatti ad una particolare applicazione.

Questo *framework* è utilizzato da molti standard di compressione di segnali audio o video, tra i quali MP3, JPEG e JPEG2000.

Capitolo 2

Overview su JPEG2000

Lo scopo di questo capitolo è quello di introdurre il sistema di codifica JPEG2000, così come è descritto nella PART I dello standard. Si discuterà, inoltre, della trasformazione delle componenti di colore dell'immagine, della trasformata wavelet e della quantizzazione.

2.1. Il Sistema di Codifica JPEG2000

Lo schema di compressione adottato da JPEG2000 è simile a quello descritto nella sezione precedente: all'immagine in input è applicata la trasformata wavelet; i coefficienti wavelet sono quantizzati e codificati mediante un codificatore entropico, **EBCOT** (*Embedded Block Coding with Optimized Truncation*), che è il cuore di JPEG2000. La codifica consiste di due fasi distinte o "*Tier*" ("*strati*", "*ordini*"). Nel tier 1 vengono raccolte informazioni statistiche sui simboli dello stream in input secondo un modello probabilistico predefinito, che sono utilizzate da un codificatore aritmetico (**MQ-coder**), responsabile della compressione (lossless) dell'immagine; nel tier 2 i blocchi di dati compressi sono raggruppati in *quality layers* progressivi, utilizzando informazioni aggiuntive fornite dal tier 1. Poiché è possibile troncare il bit-stream[3] in diversi punti, e quindi scartare qualcuno dei layers, il tier 2 rappresenta, implicitamente, una seconda fase di quantizzazione, che può essere guidata, come sarà mostrato nel seguito, sia dall'encoder che dal decoder. Lo schema descritto è riassunto in Figura 4.

[3] Sequenza di bit che costituisce la codifica dell'immagine originale.

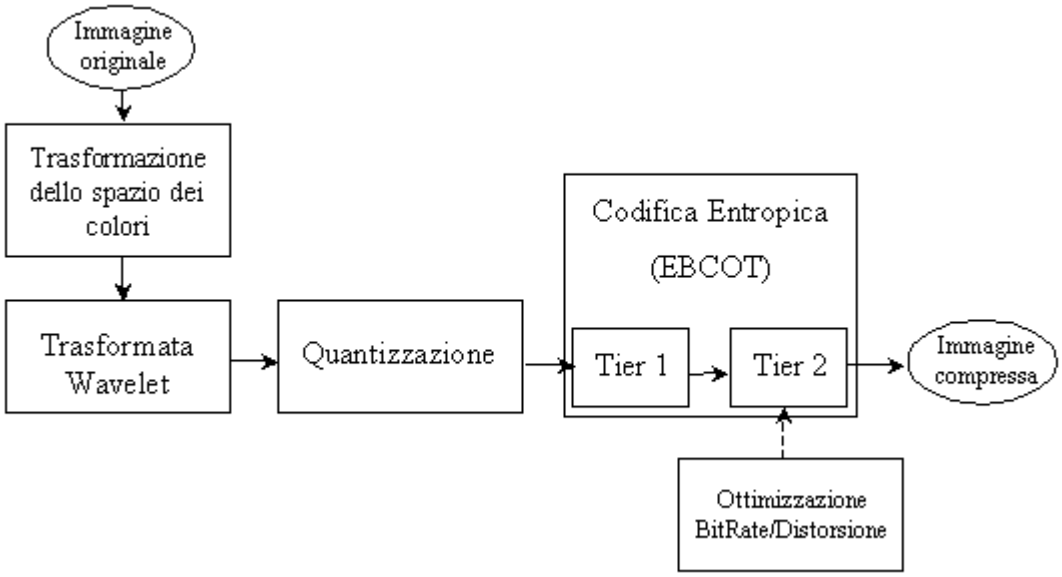

Figura 4: Sequenza di passi dell'encoder JPEG2000.

Prima di procedere con la descrizione dell'algoritmo, è opportuno mostrare come l'immagine è rappresentata nel dominio spaziale e come viene suddivisa durante le varie fasi mostrate nella figura precedente. La superficie su cui giace l'immagine è rappresentata mediante una griglia rettangolare detta *reference grid* (*griglia di riferimento*). L'angolo superiore sinistro dell'immagine ha coordinate non negative rispetto all'angolo superiore sinistro della griglia, mentre gli angoli inferiori destri della griglia e dell'immagine devono coincidere. La griglia è partizionata in una matrice di *tiles* rettangolari di dimensioni regolari che non si sovrappongono; le dimensioni delle tiles e l'offset della prima tile in alto a sinistra possono essere scelti arbitrariamente dall'encoder, con il vincolo che, pur potendo le tiles superare i limiti dell'immagine, ogni campione dell'immagine deve appartenere ad una e una sola tile, cioè le tiles non devono sovrapporsi. La figura mostra la reference grid e l'immagine suddivisa in tiles.

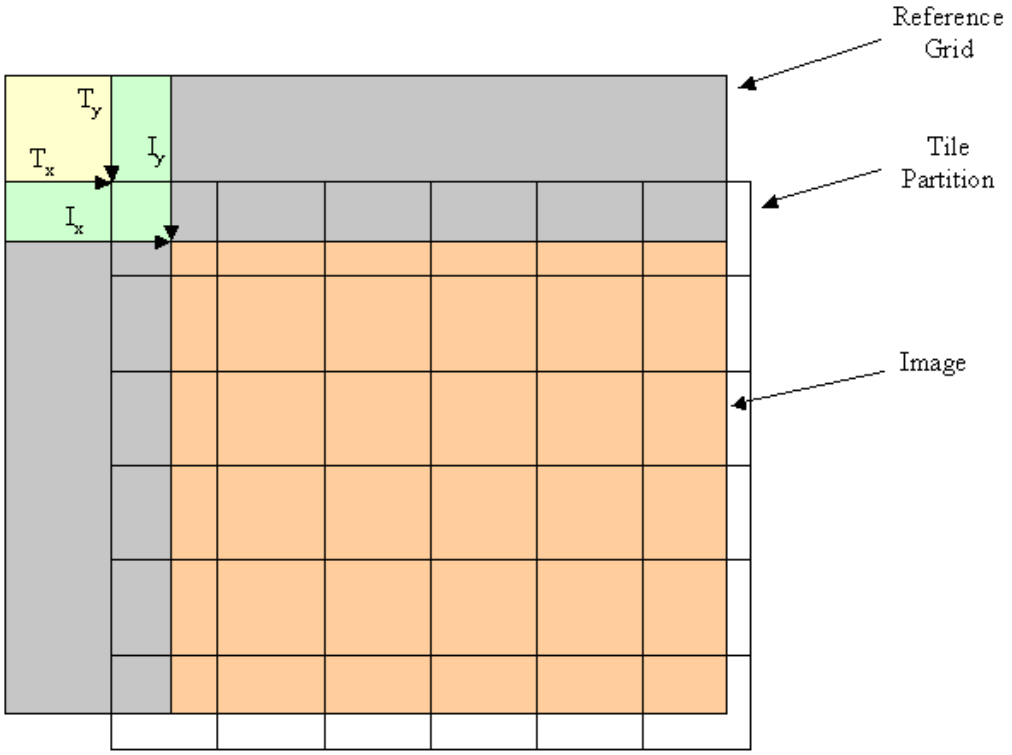

Figura 5: Reference Grid e suddivisione in tiles dell'immagine. T_x e T_y sono gli offset, rispettivamente orizzontale e verticale, dell'origine della prima tile in alto a sinistra rispetto all'origine della reference grid. I_x e I_y sono gli offset dell'immagine rispetto alla reference grid.

Ad ogni tile è applicata separatamente la trasformata wavelet. Dopo la trasformata, i coefficienti wavelet quantizzati sono raggruppati in più *code-blocks*, che sono codificati in maniera indipendente gli uni dagli altri dal codificatore aritmetico. Durante la codifica di un blocco si utilizzano informazioni sui coefficienti appartenenti ai blocchi "confinanti", purché facciano parte della stessa sottobanda.

Questa particolare struttura è giustificata dalle seguenti ragioni:

- permettere semplici manipolazioni geometriche senza dover ricodificare l'intera immagine (ad esempio, per "tagliare" la riga superiore basta eliminare i coefficienti delle tiles interessate e incrementare l'offset dell'ordinata dell'immagine rispetto alla reference grid);
- permettere modifiche localizzate dell'immagine ricodificando solo le tiles interessate da tali modifiche;
- codificare blocchi relativamente piccoli (con i conseguenti vantaggi in termini di memoria) evitando di introdurre artefatti ai bordi dei blocchi (artefatti dovuti alla codifica sono possibili solo ai bordi delle tiles, che sono di dimensioni relativamente grandi rispetto ai code-blocks).

Ogni code-block può, a sua volta, essere suddiviso in sub-blocks, per ottimizzare l'uso della primitiva di Run-Length, descritta più avanti, su grandi aree composte tutte da zeri.

Quanto appena detto introduce il processo di compressione JPEG2000 per immagini che hanno una sola componente di colore. Quando si codificano immagini a più colori, ogni

componente all'interno di una tile è codificata indipendentemente dalle altre ed è possibile assegnare fattori di scala diversi a ciascuna componente. Ad esempio, utilizzando il modello YC_bC_r, descritto nella prossima sezione, poiché le componenti di crominanza hanno un minore impatto sulla qualità percepita, tali componenti possono essere sottocampionate (come tipicamente opera lo standard JPEG). Si noti, comunque, che questo può essere fatto solo per una compressione di tipo lossy.

Nel seguito saranno analizzati in dettaglio i vari passi dell'algoritmo JPEG2000.

2.2. Trasformazione dello Spazio dei Colori

Nell'algoritmo JPEG2000, come è prassi per tutti gli algoritmi di compressione basati su una trasformata lineare, la trasformata wavelet è eseguita su ogni componente di colore in maniera indipendente rispetto alle altre. È dunque necessario che le componenti siano ragionevolmente decorrelate l'una rispetto all'altra. La rappresentazione RGB non gode di questa importante proprietà, dunque si preferisce utilizzare la rappresentazione dell'immagine in uno spazio di coordinate luminanza-crominanza, che assicura una sufficiente decorrelazione tra le tre componenti di colore dell'immagine. Bisogna, allora, operare una trasformazione dell'immagine dallo spazio RGB a quello basato su luminanza-crominanza.
Uno spazio luminanza-crominanza standard è quello YC_bC_r, che si ottiene a partire dallo spazio RGB mediante le seguenti trasformazioni:

$$\begin{pmatrix} Y \\ C_b \\ C_r \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.16975 & -0.33126 & 0.5 \\ 0.5 & -0.41869 & -0.08131 \end{pmatrix} \cdot \begin{pmatrix} R \\ G \\ B \end{pmatrix} \qquad \text{(E.1)}$$

dove le componenti R, G e B di un pixel dell'immagine assumono valori nell'intervallo [0,1].
La trasformazione inversa (da YC_bC_r a RGB) è la seguente:

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1.0 & 0 & 1.402 \\ 1.0 & -0.34413 & -0.71414 \\ 1.0 & 1.772 & 0 \end{pmatrix} \cdot \begin{pmatrix} Y \\ C_b \\ C_r \end{pmatrix} \qquad \text{(E.2)}$$

Come risulta evidente dalle precedenti formule, la trasformazione è non-reversibile a causa dell'approssimazione nei calcoli. Nella compressione lossless, allora, non possono essere utilizzate le formule precedenti; al loro posto si utilizzano le seguenti:

$$\begin{pmatrix} Y \\ C_b \\ C_r \end{pmatrix} = \begin{pmatrix} \left\lfloor \dfrac{R + 2G + B}{4} \right\rfloor \\ R - G \\ B - G \end{pmatrix} \qquad \text{(E.3)}$$

per la trasformazione diretta, e:

$$\begin{cases} G = Y - \left(\dfrac{C_b + C_r}{4} \right) \\ R = C_b + G \\ B = C_r + G \end{cases} \qquad \text{(E.4)}$$

per la trasformazione inversa.
È importante osservare che la trasformazione reversibile può essere usata solo se tutte le componenti hanno lo stesso fattore di sottocampionamento rispetto alla reference grid. Inoltre, i campioni di ciascuna componente devono avere lo stesso segno e lo stesso numero di bit nella magnitudo.

2.3. Trasformata Wavelet

A ciascuna tile è applicato l'intero algoritmo di codifica indipendentemente dalle altre: la discussione seguente, quindi, si concentrerà su una singola tile piuttosto che sull'intera immagine.
Prima di descrivere la trasformata wavelet in JPEG2000, introdurremo il concetto di sottocampionamento, partendo da segnali mono-dimensionali. Si consideri un segnale mono-dimensionale composto da N campioni. Supponiamo di voler sottocampionare tale segnale per ottenerne uno contenente solo $N/2$ campioni. Un metodo banale per ottenere il segnale sottocampionato è quello di scartare i coefficienti dispari o pari. I coefficienti scartati costituiscono l'informazione che, aggiunta al segnale sottocampionato, permette di ricostruire il segnale originale. La figura seguente mostra il processo di sottocampionamento e ricostruzione di un segnale con otto campioni.

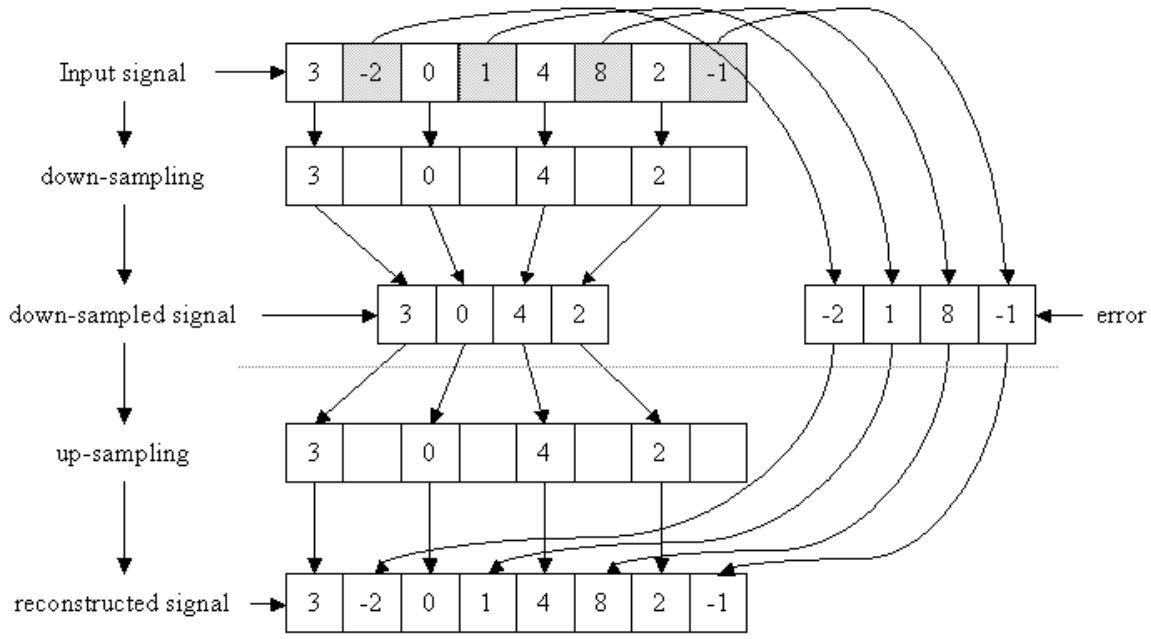

Figura 6: Esempio di decomposizione wavelet di un segnale mono-dimensionale con otto campioni.

Un sottocampionamento come quello descritto nella precedente figura non sarebbe di nessuna utilità pratica, se il segnale non fosse opportunamente filtrato, in maniera tale da concentrare l'informazione più importante nei coefficienti del segnale sottocampionato. Nella figura seguente, allo stesso segnale del precedente esempio è applicato un filtro che sostituisce il valore del coefficiente di indice $2n-1$ (si suppone che gli indici partano da 1) con la media aritmetica dei coefficienti di indice $2n-1$ e $2n$, mentre il coefficiente $2n$ è posto uguale alla differenza tra il valore precedente del coefficiente $2n-1$ e quello attuale. Il valore dei coefficienti di posto pari è inteso come l'errore commesso nel sottocampionamento.

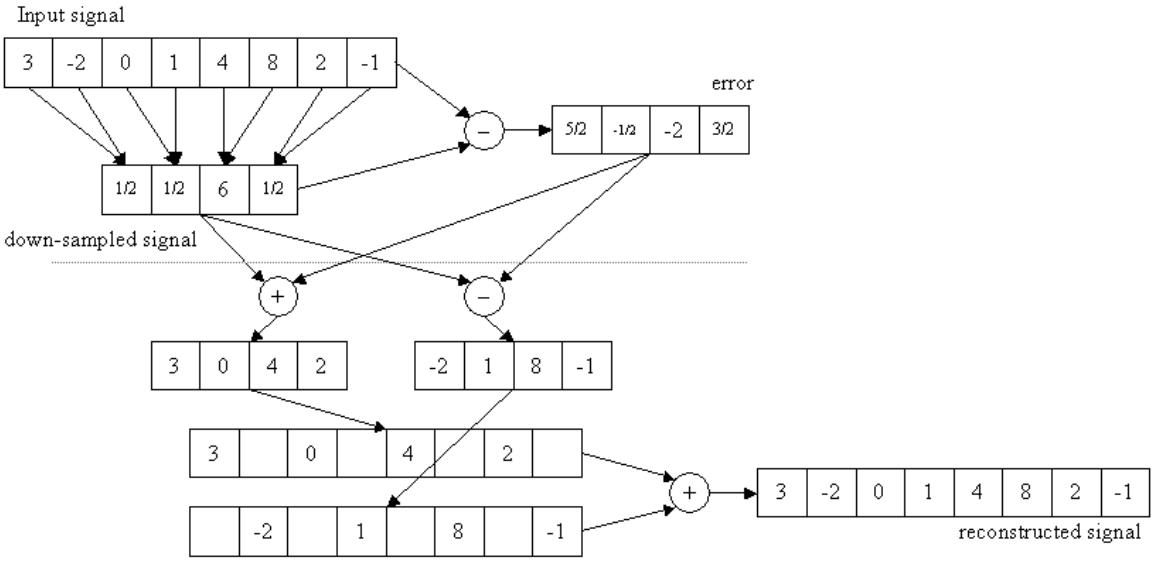

Figura 7: Esempio di filtraggio e decomposizione wavelet di un segnale mono-dimensionale con otto campioni.

Quello della figura precedente è un esempio di filtraggio. Nella pratica, quando si tratta con segnali acustici o visivi, vengono usati filtri passa-basso e passa-alto che suddividono il segnale in due bande di frequenza distinte.
Questo schema può essere facilmente esteso a segnali bidimensionali, sottocampionando alternativamente nella direzione orizzontale e in quella verticale, come mostrato nella figura seguente.

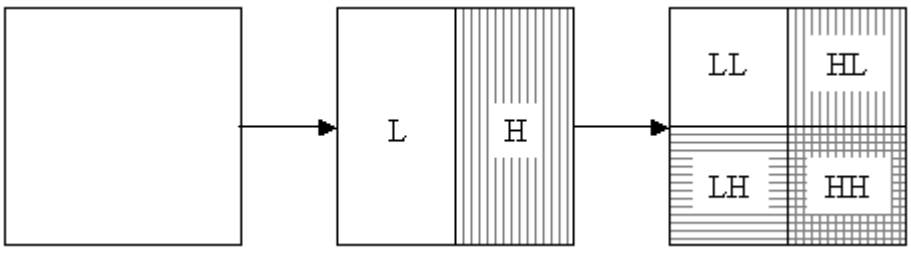

Figura 8: Esempio di decomposizione wavelet di un segnale bidimensionale.

Quello mostrato è un semplice esempio di decomposizione wavelet di un segnale bidimensionale. Esistono decomposizioni più sofisticate, ma lo schema è simile a quello descritto.

Vediamo, ora, come la trasformata wavelet sia implementata in JPEG2000. Ad ogni tile è applicata la trasformata wavelet, secondo uno tra tre diversi schemi di decomposizione: *Mallat*, *SPACL* e *Packet*. La seguente figura illustra la caratteristiche principali delle strutture di decomposizione supportate dallo standard, che sono essenzialmente un'estensione del ben noto schema di Mallat.

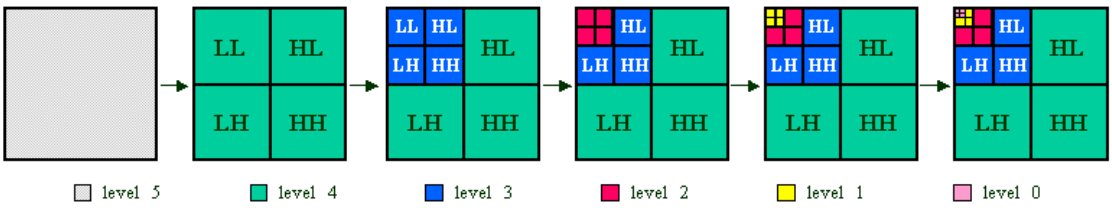

Figura 9: Immagine decomposta mediante lo schema di Mallat.

Denotiamo con L il numero totale dei livelli di decomposizione. Il numero dei livelli di risoluzione dell'immagine è allora L+1. Nel seguito per comodità i due termini saranno usati con significato equivalente. Identifichiamo con l = 0, 1, …, L i livelli di risoluzione, dove l = 0 corrisponde alla banda LL a più bassa frequenza e l = L corrisponde all'immagine originale. Le bande sono raggruppate rispetto al livello di risoluzione al quale contribuiscono. Il livello di risoluzione 0 comprende solo la banda LL. In generale, dato il livello di risoluzione l, per ricostruire l'immagine al livello di risoluzione l+1 sono necessari i coefficienti delle bande del livello (l+1)-esimo. Nella classica decomposizione "á la Mallat" ciascun livello da 1 a L aggiunge 3 sottobande, denotate rispettivamente HL (passa-alto orizzontale), LH (passa-alto verticale) e HH (passa-alto orizzontale e verticale). La seguente figura mostra i passi di una decomposizione wavelet a 3 livelli, secondo lo schema di Mallat.

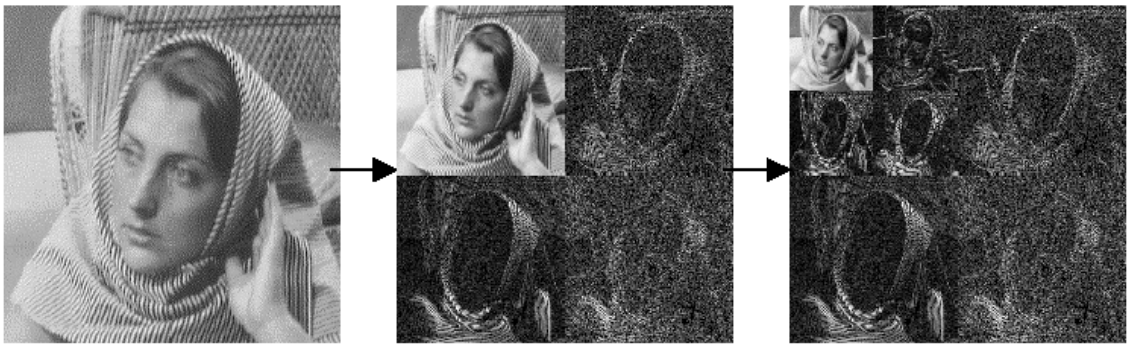

Figura 10: Immagine filtrata e decomposta mediante lo schema di Mallat.

Nello standard, la struttura descritta sopra è estesa con un parametro ϕ_l per ogni livello di risoluzione l=1,2,…,L, detto *high-pass descent depth*, che indica il numero di livelli di decomposizione delle bande contenenti alte frequenze (LH, HL e HH) al livello l. Ad esempio ϕ_l = 2 significa che le bande LH, HL, HH al livello l sono suddivise ciascuna in quattro sottobande; se ϕ_l = 3, ancora una volta le bande ad alta frequenza sono suddivise in quattro sottobande ciascuna e le dodici sottobande ottenute sono ulteriormente decomposte in quattro sottobande. I valori di ϕ_l ammessi nello standard sono 1, 2 e 3. In particolare, per la decomposizione Mallat ϕ_l = 1, l = 1,2,..L; per lo schema SPACL ϕ_l = 1, l = 1, 2, …, L-1

e $\phi_l = 2$, $l = L$; per Packet, $\phi_l = 1$, $l = 1,2,\ldots L\text{-}2$ e $\phi_l = 2$, $l = L\text{-}1$, $\phi_l = 3$, $l = L$. Le tre decomposizioni sono mostrate in figura.

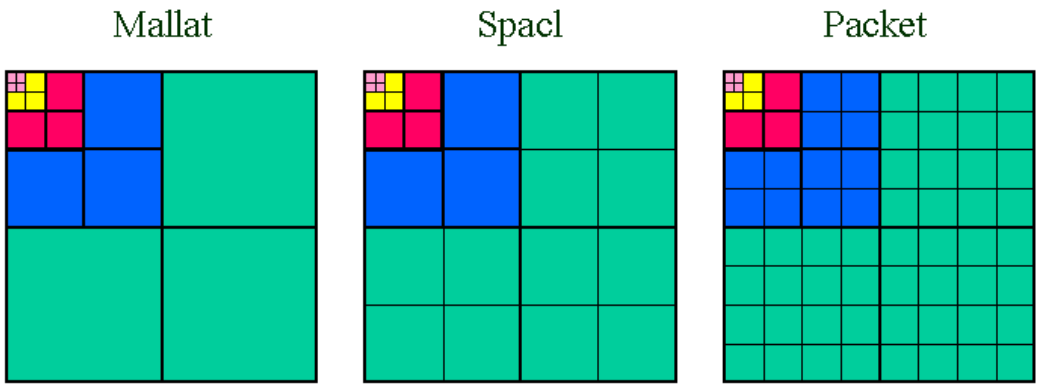

Figura 11: Confronto tra le decomposizioni wavelet supportate dallo standard

Lo standard JPEG2000 supporta sia la trasformata reversibile, a coefficienti interi, che quella non-reversibile, a coefficienti reali.
La trasformazione irreversibile utilizzata normalmente da JPEG 2000 impiega il filtro Daubechies 9/7. I coefficienti dei filtri di analisi, passa-alto e passa-basso, sono riportati in Tabella 1, quelli di sintesi in Tabella 2. La trasformazione reversibile, invece, viene realizzata mediante un filtro 5/3. La caratteristica di questo tipo di filtro è quella di permettere una ricostruzione completa e senza perdite (*lossless*) in fase di sintesi. I coefficienti dei filtri di analisi, passa-alto e passa-basso, sono riportati in Tabella 3, quelli dei filtri di sintesi in Tabella 4. La Part II dello standard permette di utilizzare altri tipi di filtri definiti dall'utente, oltre a quelli sopra indicati.

	9/7 - analysis	
k	low pass	high pass
0	0,6029490182	1,1150870524
±1	0,2668641184	-0,5912717631
±2	-0,0782232665	-0,0575435262
±3	-0,0168641184	0,0912717631
±4	0,0267487574	

Tabella 1: Filtro Daubechies 9/7 di analisi usato per la trasformata irreversibile. *k* è l'offset del coefficiente rispetto alla posizione centrale del filtro.

	9/7 - syntesis	
k	low pass	high pass
0	1,1150870524	0,6029490182
±1	0,5912717631	-0,2668641184
±2	-0,0575435262	-0,0782232665
±3	-0,0912717631	0,0168641184
±4		0,0267487574

Tabella 2: Filtro Daubechies 9/7 di sintesi usato per la trasformata irreversibile. *k* è l'offset del coefficiente rispetto alla posizione centrale del filtro.

5/3 - analysis		
k	low pass	high pass
0	3/4	1
±1	1/4	- 1/2
±2	- 1/8	

Tabella 3: Filtro 5/3 di analisi usato per la trasformata irreversibile. *k* è l'offset del coefficiente rispetto alla posizione centrale del filtro.

5/3 - syntesis		
k	low pass	high pass
0	1	3/4
±1	- 1/2	1/4
±2		- 1/8

Tabella 4: Filtro 5/3 di sintesi usato per la trasformata irreversibile. *k* è l'offset del coefficiente rispetto alla posizione centrale del filtro.

Lo standard permette di realizzare la trasformata wavelet in due modi diversi. L'implementazione più tradizionale è quella a banco di filtri; il secondo metodo si basa sul *lifting scheme*.
Il filtraggio basato sul lifting scheme consiste di una sequenza di operazioni elementari, dette *passi di lifting* (*lifting steps*), mediante le quali, alternativamente, il valore dei campioni di posto pari è aggiornato mediante una media pesata dei campioni di posto dispari (*prediction*), arrotondata a un valore intero, e il valore dei campioni di posto dispari è aggiornato in base ad una media pesata dei campioni di posto pari appena aggiornati (*update*), anche questa arrotondata.
Un semplice esempio di lifting è rappresentato nella seguente figura. Il blocco *SPLIT* separa i campioni di posto pari da quelli di posto dispari. I blocchi *P* e *U* sono responsabili di aggiornare il valore dei campioni di posto, rispettivamente, pari e dispari.

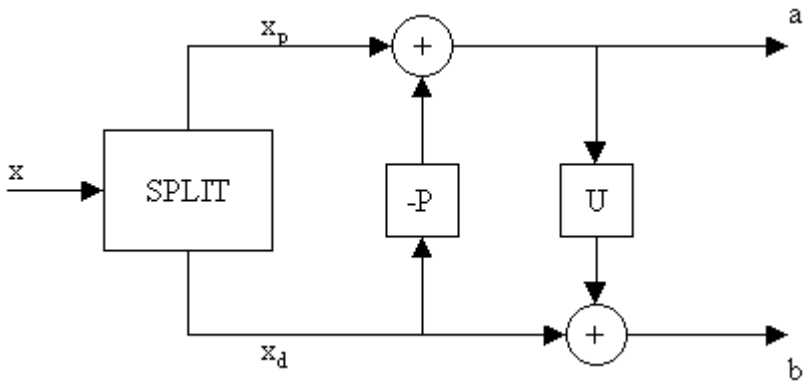

Figura 12: Esempio di *lifting scheme*. Analisi.

Per approfondimenti sul lifting scheme, si vedano [2] e [7].
Sia il metodo a banco di filtri che il lifting scheme possono essere usati per la trasformata irreversibile con il filtro 9/7. La trasformata reversibile, invece, può essere implementata solo mediante il lifting scheme, usando il filtro 5/3. Allora, per la compressione lossless,

che deve necessariamente fare uso di una trasformazione reversibile, l'unica implementazione possibile è quella che prevede l'uso del lifting scheme, con il filtro 5/3 (o un altro filtro a coefficienti interi). Nessuna restrizione, invece, si applica alla compressione lossy.
Le equazioni che implementano i filtri 5/3 e 9/7 mediante il lifting scheme sono riportate di seguito. Le prime due sono relative al calcolo coefficienti di posto dispari e di posto pari del filtro di analisi 5/3. Seguono, nell'ordine, le due formule del filtro 5/3 di sintesi, filtro 9/7 di analisi e filtro 9/7 di sintesi. Nelle formule, X e Y sono, rispettivamente, il segnale originale e i coefficienti wavelet, mentre X_{ext} e Y_{ext} sono i corrispondenti segnali estesi ai bordi (si veda più avanti).

$$Y(2n+1) = X_{ext}(2n+1) - \left\lfloor \frac{X_{ext}(2n) + X_{ext}(2n+2)}{2} \right\rfloor \tag{E.5}$$

$$Y(2n) = X_{ext}(2n) + \left\lfloor \frac{Y(2n-1) + Y(2n+1) + 2}{4} \right\rfloor \tag{E.6}$$

$$X(2n) = Y_{ext}(2n) - \left\lfloor \frac{Y_{ext}(2n-1) + Y_{ext}(2n+1) + 2}{4} \right\rfloor \tag{E.7}$$

$$X_{ext}(2n+1) = Y_{ext}(2n+1) + \left\lfloor \frac{X(2n) + X(2n+2)}{2} \right\rfloor \tag{E.8}$$

$$\begin{cases} Y(2n+1) = X_{ext}(2n+1) + \left(\alpha \times \left[X_{ext}(2n) + X_{ext}(2n+2)\right]\right) & [\text{step 1}] \\ Y(2n) = X_{ext}(2n) + \left(\beta \times \left[Y(2n-1) + Y(2n+1)\right]\right) & [\text{step 2}] \\ Y(2n+1) = Y(2n+1) + \left(\gamma \times \left[Y(2n) + Y(2n+2)\right]\right) & [\text{step 3}] \\ Y(2n) = Y(2n) + \left(\delta \times \left[Y(2n-1) + Y(2n+1)\right]\right) & [\text{step 4}] \\ Y(2n) = -K \times Y(2n+1) & [\text{step 5}] \\ Y(2n) = (1/K) \times Y(2n) & [\text{step 6}] \end{cases} \tag{E.9}$$

$$\begin{cases} X(2n) = K \times Y_{ext}(2n) & [\text{step 1}] \\ X(2n+1) = -(1/K) \times Y_{ext}(2n+1) & [\text{step 2}] \\ X(2n) = X(2n) - \left(\delta \times \left[X(2n-1) + X(2n+1)\right]\right) & [\text{step 3}] \\ X(2n+1) = X(2n+1) - \left(\gamma \times \left[X(2n) + X(2n+2)\right]\right) & [\text{step 4}] \\ X(2n) = X(2n) - \left(\beta \times \left[X(2n-1) + X(2n+1)\right]\right) & [\text{step 5}] \\ X(2n+1) = X(2n+1) - \left(\alpha \times \left[X(2n) + X(2n+2)\right]\right) & [\text{step 6}] \end{cases} \tag{E.10}$$

dove i valori dei parametri (α, β, γ, δ) sono definiti come segue:

$$\begin{cases} \alpha = -1.586134342 \\ \beta = -0.052980118 \\ \gamma = 0.882911075 \\ \delta = 0.443506852 \end{cases} \qquad \text{(E.11)}$$

e il fattore di scala K è uguale a $K = 1.1230174105$.
Quando si applica il filtraggio ai bordi di una tile, sorge il problema di quali valori mettere al posto dei campioni mancanti oltre il bordo. Usando filtri simmetrici, un modo tipico di procedere è quello che consiste nell'estendere l'immagine ai bordi in maniera simmetrica e periodica rispetto al bordo stesso. La figura seguente mostra un esempio di tale estensione. I coefficienti del segnale originale sono disegnati con una linea marcata, mentre i coefficienti aggiunti dall'estensione sono quelli tratteggiati.

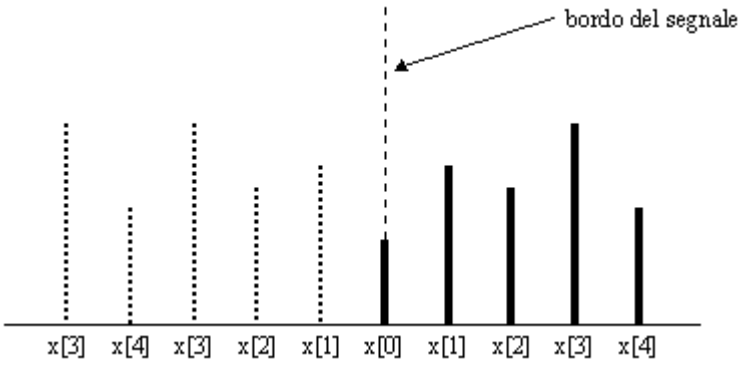

Figura 13: Esempio di estensione simmetrica del segnale al bordo.

La dimensione dell'estensione dipende, ovviamente, dalla dimensione del supporto del filtro usato. Ad esempio, se *i* è l'indice del coefficiente vicino al bordo, usando un filtro 9/7 devono essere aggiunti 3 o 4 coefficienti oltre il bordo, a seconda che *i* sia dispari o pari.
Pur essendo questa una soluzione accettabile quando il filtro è applicato ai bordi dell'immagine, questo metodo introduce artefatti quando applicato ai bordi delle tile, con conseguente effetto blocking (presenza di artefatti ai bordi dei blocchi), che è tanto più visibile quanto più alto è il fattore di compressione. Per ridurre l'effetto blocking, lo standard JPEG2000 prevede un'opzione che permette di utilizzare campioni appartenenti alle tiles vicine, ai fini dell'applicazione del filtro wavelet. Più precisamente, la tile da codificare, di dimensioni MxN, è estesa di una riga e una colonna; la trasformata wavelet è applicata a questo blocco di dimensioni (M+1)x(N+1), secondo uno dei metodi descritti sopra. In questo modo, si permette una parziale interdipendenza tra le tiles, dovuta alla sovrapposizione dei coefficienti ai bordi, ma in genere si riesce a ridurre l'effetto blocking.

2.4. Quantizzazione

La trasformata wavelet decompone la tile in livelli di risoluzione e, all'interno di ogni livello, in sottobande, ciascuna con un certo contenuto di frequenza. Il numero dei coefficienti wavelet è uguale a quello dei campioni che la tile conteneva prima

dell'applicazione della trasformata, quindi il passo della trasformata wavelet di per sé non comprime il segnale. La prima compressione avviene nel passo di quantizzazione, il cui compito è quello di fornire un'approssimazione dei coefficienti che sia conveniente in termini di numero di simboli utilizzati per rappresentare ciascuno di loro. L'approssimazione introdotta non permette la perfetta ricostruzione dei coefficienti e ciò fa della quantizzazione un processo inerentemente lossy. Lo schema di quantizzazione supportato dalla PART I dello standard JPEG 2000 è quello scalare. La PART II permette anche l'uso di uno schema più sofisticato, detto *Trellis Coded Quantization*. Qui sarà trattato solo il primo.

2.4.1. Quantizzazione Scalare

Questo schema di quantizzazione è detto scalare perché ciascun coefficiente quantizzato è ottenuto solo come funzione del solo coefficiente wavelet di partenza, diversamente da altri metodi che utilizzano informazioni su un certo numero di coefficienti vicini.
Prima di proseguire nell'argomento, introduciamo la notazione necessaria. Dato un code-block B_i, indichiamo con x_i[n] il coefficiente n-esimo di B_i, rispetto ad un ordine arbitrario, e sia q_i[n] il corrispondente coefficiente quantizzato. Allora q_i[n] e dato dall'equazione:

$$q_i[n] = sign(x_i[n]) \cdot \left\lfloor \frac{x_i[n]}{\Delta_b} \right\rfloor \qquad \text{(E.12)}$$

dove $sign(\cdot)$ è la funzione segno e Δ_b è il cosiddetto *passo di quantizzazione* per la sottobanda b, cui il blocco B_i appartiene. Come risulta chiaro dalla formula, i coefficienti quantizzati sono ottenuti troncando la rappresentazione decimale dei coefficienti originali divisi per il passo di quantizzazione. Ciò significa che i coefficienti che hanno magnitudo inferiore a Δ_b sono azzerati e nessuna informazione potrà essere ricostruita per loro durante la dequantizzazione. Per tale motivo questa quantizzazione è detta *dead-zone quantization* (quantizzazione a zona morta). Il parametro Δ_b suddivide l'asse delle frequenze in intervalli di dimensione Δ_b, come mostrato in Figura 14. Tutti i coefficienti wavelet che appartengono ad un dato intervallo hanno lo stesso valore quantizzato, quindi, al momento di dequantizzarli, non c'è modo di distinguerli. Dal punto di vista del dequantizzatore tali simboli risultano, dunque, approssimati ad un unico valore appartenente allo stesso intervallo. Ciò apparirà più chiaro fra breve, quando introdurremo la formula per ricavare i coefficienti dequantizzati a partire da quelli quantizzati.

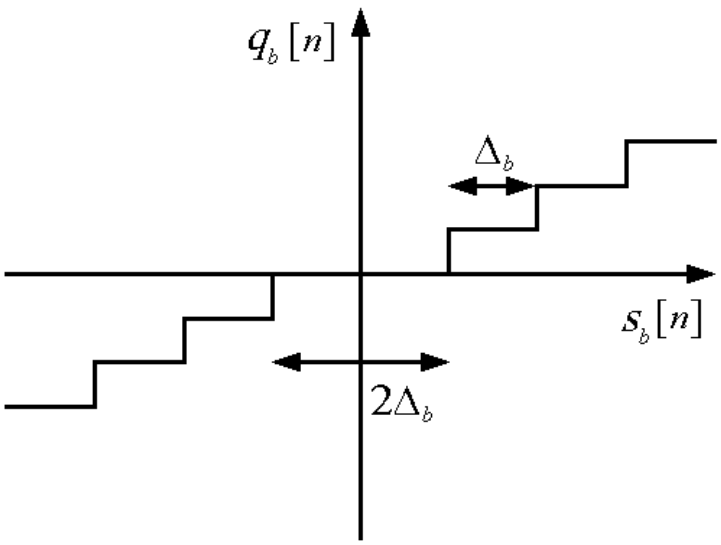

Figura 14: Asse delle frequenze diviso in intervalli di dimensione Δ_b

Come si può osservare dalla precedente figura, tutti gli intervalli hanno ampiezza Δ_b, tranne l'intervallo intorno allo zero (dead-zone), la cui ampiezza è $2\Delta_b$. Ciò è dovuto al fatto che a tale intervallo appartengono tutti i coefficienti la cui magnitudo è inferiore al passo di quantizzazione, positivi e negativi, quindi la dead-zone è in realtà l'unione di due intervalli i cui coefficienti hanno valori simmetrici rispetto allo zero.
Il passo di quantizzazione può essere diverso per ogni sottobanda. Questo permette di quantizzare in maniera più fine i coefficienti di una sottobanda rispetto a quelli di un'altra. Ciò fornisce un meccanismo grezzo di quantizzazione adattativa rispetto al contenuto delle bande. Tuttavia, proprio perché il passo di quantizzazione è uguale per tutti i coefficienti appartenenti alla stessa banda, la quantizzazione è ancora *uniforme*. La quantizzazione non-uniforme, unita a considerazioni di carattere psico-visivo, spesso raggiunge migliori risultati, in termini di qualità percepita. Diversi studi sono stati fatti su questo argomento. É importante osservare che, al fine di poter ricostruire in maniera affidabile i coefficienti wavelet, il modello usato, non solo il passo di quantizzazione, deve essere segnalato esplicitamente al decoder.

2.4.2. Dequantizzazione Scalare

Una delle caratteristiche di JPEG2000 è la scalabilità per qualità dell'immagine compressa. Ciò è ottenuto, come vedremo più avanti, mediante la codifica a bit-plane. Tanto l'encoder, quanto il decoder sono liberi di scartare alcuni tra i bit-planes meno significativi o parte di essi. Questa possibilità induce una forma di quantizzazione *embedded* (incapsulata), della quale è necessario tenere conto in fase di dequantizzazione. Per modellare correttamente questa situazione, l'equazione precedente deve essere generalizzata. Siano, allora, p_i[n] il numero di bit-planes scartati in corrispondenza al simbolo n-esimo del code-block B_i e M_b il massimo numero di bit-planes nella sottobanda b. L'equazione modificata è la seguente:

$$q_i^{\,p_i[n]}[n] = sign(x_i[n]) \cdot \left\lfloor \frac{x_i[n]}{\Delta_b^{p_i[n]}} \right\rfloor \qquad \text{(E.13)}$$

con $\Delta_b^{p_i[n]} = 2^{p_i[n]} \cdot \Delta_b$. Il fattore $2^{p_i[n]}$ rappresenta il fatto che sono stati "tagliati" i $p_i[n]$ bits meno significativi del coefficiente $x_i[n]$. Allora, l'equazione che permette di ricostruire i coefficienti wavelet a partire dai coefficienti quantizzati $\hat{x}_i[n]$ è la seguente:

$$\hat{x}_i[n] = \begin{cases} \left\lceil \left(q_i^{\,p_i[n]}[n] - r \cdot 2^{M_b - p_i[n]} \right) \cdot \Delta_b^{\,p_i[n]} \right\rceil & q_i^{\,p_i[n]} < 0 \\ 0 & q_i^{\,p_i[n]} = 0 \\ \left\lfloor \left(q_i^{\,p_i[n]}[n] + r \cdot 2^{M_b - p_i[n]} \right) \cdot \Delta_b^{\,p_i[n]} \right\rfloor & q_i^{\,p_i[n]} > 0 \end{cases} \qquad \text{(E.14)}$$

dove il parametro r è nel range $0 \le r < 1$ e dovrebbe essere scelto per produrre la migliore qualità dell'immagine ricostruita, secondo un criterio visivo o oggettivo. Un valore tipico è $r = 1/2$.

Capitolo 3

Codifica Entropica: EBCOT tier 1

In questo capitolo sarà presentato in dettaglio l'algoritmo di codifica entropica utilizzato in JPEG2000. Dopo una breve introduzione all'argomento, saranno presentati i principi della teoria bit-rate/distorsione, che giustificano l'algoritmo presentato. Infine, l'attenzione si concentrerà sulla modellizzazione dei contesti associati ai bit da codificare.

3.1. Introduzione

Introdotto da Taubman [1] nel 1998, EBCOT (*Embedded Block Coding with Optimized Truncation*) è stato adottato dallo standard JPEG2000 per la sua capacità di combinare un alto fattore di compressione a un buon grado di scalabilità[4] rispetto a risoluzione e qualità. Inoltre, grazie alla codifica a blocchi relativamente piccoli, l'accesso diretto ai dati compressi è piuttosto granulare.
I coefficienti quantizzati sono passati a EBCOT, che si occupa di "riscrivere" i dati in input, in maniera tale da eliminare le ridondanze dovute alla rappresentazione dei coefficienti wavelet, e riordinare l'output, trasmettendo prima l'informazione più importante. Nel far questo, l'encoder deve tenere conto di come l'immagine compressa dovrà essere ricostruita; deve, dunque, fare in modo che il decoder sia in grado di associare a ciascuna sequenza di bit un solo significato, cioè la codifica deve essere univoca. Ciò potrebbe essere fatto segnalando nel bitstream l'ordine di codifica e di trasmissione, tuttavia l'overhead risultante da questo metodo annullerebbe i vantaggi della codifica adattativa. É dunque necessario che questa informazione sia segnalata implicitamente nella sequenza di bit codificata. Prima di descrivere come questo tipo di codifica è ottenuto dal codificatore entropico, descriveremo brevemente la struttura di EBCOT.
Come mostrato in Figura 4, EBCOT è composto da due blocchi logici distinti, detti *tier*. Il tier 1 riceve in input un blocco rettangolare di coefficienti wavelet quantizzati e li codifica utilizzando un modello di probabilità adattativo, che segue uno schema prestabilito. Ogni blocco genera *codewords* (sequenza di dati compressi) indipendenti, che sono raggruppate

[4] Per "scalabilità" di un'immagine compressa si intende la possibilità di decodificare l'immagine in maniera parziale, ottenendo una ricostruzione in scala ridotta (scalabilità per risoluzione) o di minore qualità (scalabilità per qualità).

in livelli di qualità progressivi, detti *layers* o *quality layers*. Per costruire i layers, le *codewords* sono suddivise in pezzi, detti *chunks*, in maniera tale da non compromettere la corretta sintassi dei dati codificati. Il tier 2 raccoglie queste informazioni e dispone i chunks di tutti i blocchi nei layers. L'encoder è libero di disporre chunks di diversi blocchi nell'ordine desiderato, purché preservi l'ordine dei chunks all'interno di un blocco, in maniera tale da assegnare contributi incrementali da ciascun blocco ad ogni layer. Il bit-stream finale può essere troncato alla fine di questi chunks, sia in fase di codifica che di decodifica. La figura seguente mostra un esempio di suddivisione dei chunks in layers.

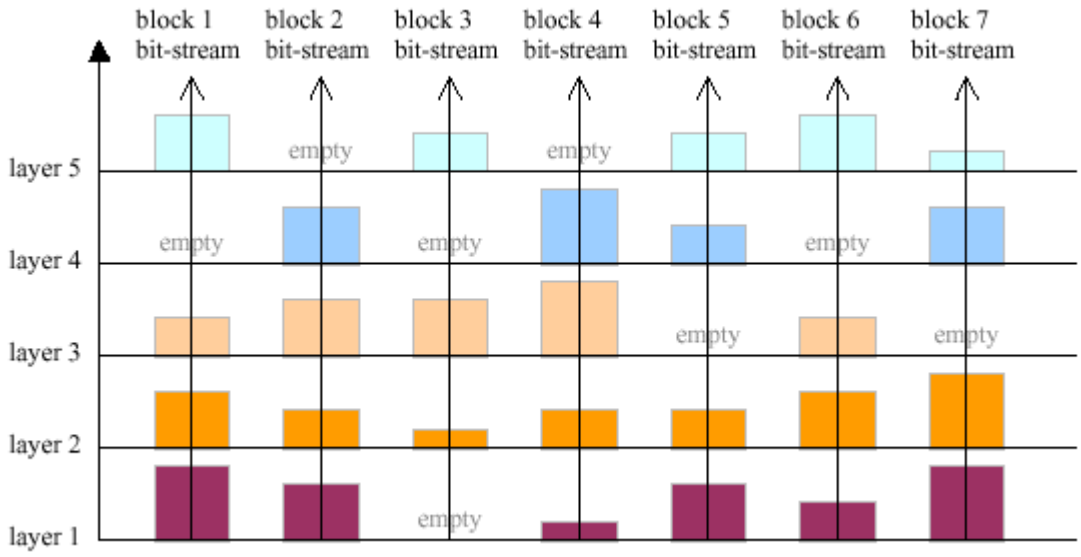

Figura 15: Illustrazione dei contributi dei blocchi ai layers. Si noti che non tutti i blocchi devono contribuire ad ogni livello e il numero di byte forniti a ciascun livello dai blocchi è, in generale, molto variabile. Tutte le operazioni di codifica procedono in maniera indipendente per ciascun blocco, quindi la codifica dei blocchi avviene in maniera verticale, nel verso indicato dalle frecce. L'organizzazione del bit-stream è, invece, orizzontale, nel senso che i chunks sono ordinati per layer nel bit-stream finale.

Non tutti i blocchi devono contribuire ad ogni livello e il contributo in byte dei blocchi in ciascun livello è, in genere, molto variabile. Tutte le operazioni di codifica procedono in maniera indipendente per ciascun blocco. Quando tutti i blocchi sono stati codificati, essi sono spezzati in chunks e suddivisi in maniera opportuna nei vari layers, in base al "valore" dell'informazione che contengono.
Infine, l'informazione codificata è raggruppata in una collezione di cosiddetti *pacchetti*, ciascuno contenente i dati codificati per un certa sottobanda (scalabilità per risoluzione) e per un certo layer (scalabilità per qualità). In particolare, un pacchetto contiene, per ciascun blocco nella sottobanda corrente, una porzione della codeword generata da tale blocco, corrispondente ad uno o più chunks, inclusi nel layer corrente.
Questa struttura è stata progettata con l'obiettivo di ottenere un bit-stream che fosse scalabile per qualità e risoluzione e, nello stesso tempo, possedesse un certo grado di scalabilità spaziale. La scalabilità per risoluzione è ottenuta mediante la decomposizione wavelet. La scalabilità spaziale è assicurata dal fatto che le operazioni di codifica sono eseguite, in maniera indipendente, su blocchi sufficientemente piccoli da garantire una certa granularità nell'indirizzamento dei singoli elementi. L'organizzazione a quality layers progressivi, infine, garantisce la scalabilità per qualità.

3.2. Ottimizzazione Bitrate-Distorsione

La chiave dell'efficienza di questo sistema di codifica è la scelta oculata dei punti di troncamento al fine di minimizzare il rapporto distorsione/bitrate, dato un certo target bitrate. Dato un code-block B_i appartenente ad una certa sottobanda b nella tile corrente, indichiamo con R_i^0, R_i^1, ..., R_i^n le lunghezze in bit a cui è possibile troncare la codeword relativa al blocco B_i, dove R_i^0 rappresenta la situazione iniziale, in cui nessun bit è trasmesso, e R_i^n è la codifica completa del blocco.

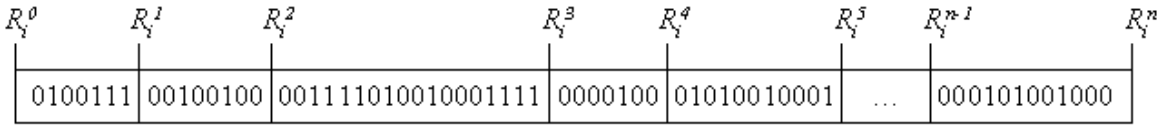

Figura 16: Esempio di codeword relativa ad un blocco. R_i^0, R_i^1, ..., R_i^n sono i punti di troncamento legali, cioè i punti in cui è possibile troncare il bit-stream, senza alterarne la sintassi.

A ciascun punto di troncamento R_i^j è associata una distorsione D_i^j, che rappresenta l'errore commesso nella ricostruzione del code-block se il bitstream è troncato alla lunghezza R_i^j. In particolare, se il bitstream non è troncato in nessun punto intermedio, cioè il numero di bit trasmessi è R_i^n, i coefficienti del code-block potranno essere ricostruiti in maniera fedele (a meno dell'errore commesso nella fase di quantizzazione), quindi la distorsione D_i^n è nulla. Denotiamo con

$$R = \sum_i R_i^{n_i} \qquad \text{(E.15)}$$

la dimensione totale dell'immagine compressa, escluso l'overhead dovuto alla segnalazione di informazioni a contorno, e con

$$D = \sum_i D_i^{n_i} \qquad \text{(E.16)}$$

la distorsione globale. Allora, fissato un bitrate massimo R_{max}, l'obiettivo è trovare un insieme di punti di troncamento n_i tali da minimizzare la distorsione D, soggetti al vincolo R_{max} sul bitrate, cioè:

$$R = R_{max} \qquad \text{(E.17)}$$

É importante osservare che l'equazione E.16 assume che la metrica usata per quantificare la distorsione sia additiva. Una metrica che soddisfa l'additività è il *Mean Square Error* (Errore Quadratico Medio), che tuttavia non è adeguata per rappresentare il modo in cui l'essere umano percepisce le immagini. In seno al progetto JPEG2000, sono state proposte diverse metriche che si basano sul sistema visivo umano.
Un metodo per trovare i punti di troncamento ottimali è il cosiddetto metodo dei *moltiplicatori di Lagrange* per i problemi di ottimizzazione vincolati, che consiste nel minimizzare la funzione

$$D + \lambda \cdot R \qquad \text{(E.18)}$$

dove il valore di λ deve essere scelto in maniera tale che il bitrate R, prodotto dai punti di troncamento che minimizzano la precedente funzione, soddisfi il vincolo $R = R_{max}$. Il metodo di Lagrange trova sempre una soluzione ottimale, ma opera con un insieme continuo di valori reali, mentre i valori $R_i^{n_i}$ sono discreti. Conseguentemente, tale metodo in generale non sarà in grado di trovare un valore λ per cui R è esattamente uguale a R_{max}, quindi dobbiamo rilassare il vincolo E.15, imponendo che $R \le R_{max}$. Nella pratica, tuttavia, la subottimalità del metodo è limitata se il numero dei punti di troncamento per ogni blocco è sufficientemente grande. Vedremo in seguito come questo obiettivo sia raggiunto da EBCOT. Ritornando al problema di ottimizzazione, dalla definizione di R e D e dall'ipotesi di additività della metrica usata per valutare la distorsione, è chiaro che la soluzione al problema globale è composizione delle soluzioni di problemi di ottimizzazione analoghi sui singoli blocchi. In breve, per ogni blocco B_i bisogna trovare il punto di troncamento n_i^{λ} che minimizza la funzione $(D_i^{n_i^{\lambda}} + \lambda R_i^{n_i^{\lambda}})$. Un semplice algoritmo per trovare n_i^{λ} è quello descritto da D.Taubman in [1] che riportiamo qui di seguito:

- Inizializza $n_i^{\lambda} = 0$ (nessuna informazione inclusa per il blocco corrente)
- **for** j = 1,2,3,…
  - Siano $\Delta D_i^{\,j} = D_i^{n_i^{\lambda}} - D_i^{\,j}$ *e* $\Delta R_i^{\,j} = R_i^{\,j} - R_i^{n_i^{\lambda}}$
  - **if** $\dfrac{\Delta D_i^{\,j}}{\Delta R_i^{\,j}} > \lambda$ **then** update $n_i^{\lambda} = j$

Algoritmo A.1: Algoritmo per il calcolo dei punti di troncamento ottimali. L'output è il punto di troncamento n_i^{λ} che minimizza la funzione $(D_i^{n_i^{\lambda}} + \lambda R_i^{n_i^{\lambda}})$.

Poiché questo algoritmo potrebbe dover essere eseguito per molti valori distinti di λ, ha senso identificare l'insieme N_i dei punti di troncamento candidati prima di eseguire l'algoritmo stesso. Sia $j_1, j_2, \ldots, j_m$ una enumerazione degli elementi di N_i tale che $R_i^{j_k} < R_i^{j_{k+1}}$ e sia $S_i^{j_k} = \dfrac{\Delta D_i^{j_k}}{\Delta R_i^{j_k}}$ il rapporto bitrate-distorsione per ciascun elemento j_k, dove $\Delta D_i^{j_k} = D_i^{j_{k-1}} - D_i^{j_k}$ *e* $\Delta R_i^{j_k} = R_i^{j_k} - R_i^{j_{k-1}}$. I rapporti $S_i^{j_k}$ devono essere strettamente decrescenti rispetto a k, infatti se per assurdo esistesse un indice k tale che $S_i^{j_{k+1}} \ge S_i^{j_k}$, il punto di troncamento j_k non potrebbe mai essere selezionato dall'algoritmo precedente, indipendentemente dal valore di λ, quindi N_i non sarebbe l'insieme dei punti di troncamento candidati. È facile verificare che, eseguito sull'insieme ristretto di punti di troncamento N_i, l'algoritmo A.1 implementa la funzione $n_i = \max\{j_k \in N_i \mid S_i^{j_k} > \lambda\}$.

Se rappresentiamo in un grafico bitrate-distorsione i punti $(R_i^{j_k}, D_i^{j_k})$, i punti di troncamento che appartengono all'insieme N_i sono quelli che giacciono sulla curva rappresentata in Figura 17.a, che è l'insieme dei punti che minimizzano la somma $(D_i^{n_i^{\lambda}} + \lambda R_i^{n_i^{\lambda}})$ per un certo λ fissato. Tale curva è l'*involucro convesso* dell'insieme dei punti di troncamento, Il rapporto $S_i^{j_k}$ può essere pensato come la pendenza della tangente alla curva nel punto $(R_i^{j_k}, D_i^{j_k})$ e per questo, d'ora in poi, ci riferiremo ad esso col nome

slope bitrate-distorsione. Da quanto detto risulta chiaro che l'insieme N_i può essere determinato mediante l'analisi dell'involucro convesso dell'insieme dei punti di troncamento.

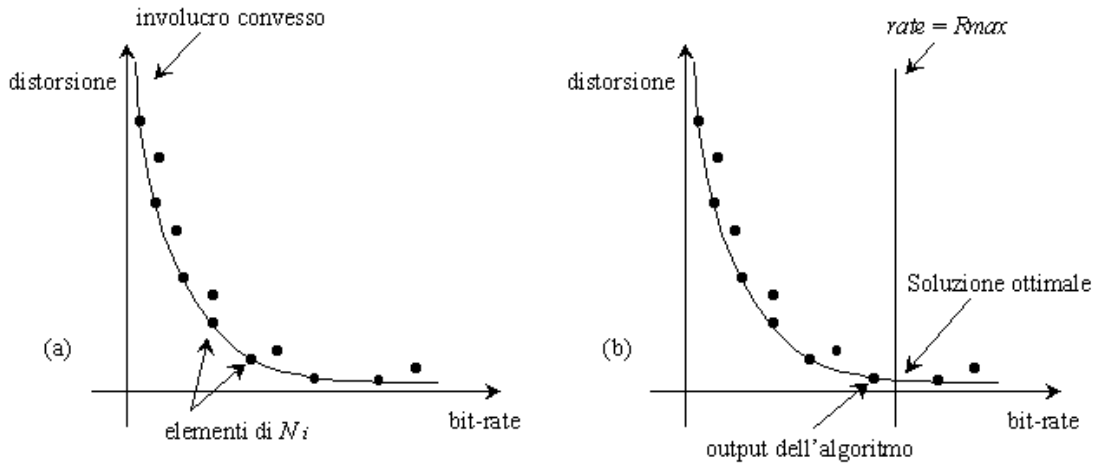

Figura 17: (a) problema discreto e involucro convesso; (b) rappresentazione del problema di ottimizzazione vincolato.

La Figura 17.b rappresenta il problema di ottimizzazione bitrate-distorsione. La retta in figura è la retta $rate = R_{max}$ e la curva ha lo stesso significato di quella rappresentata nella Figura 17.a. Allora, il minimo della somma $(D_i^{n_i^\lambda} + \lambda R_i^{n_i^\lambda})$ è il punto di intersezione tra curva e retta. Quando trattiamo con un insieme discreto di punti, come nel nostro caso, il punto di intersezione in generale non esiste, quindi non è possibile ottenere una soluzione ottimale. La soluzione sub-ottimale del problema è il punto che giace nell'involucro convesso e che ha ascissa massima, a sinistra della retta $rate = R_{max}$, cioè il punto la cui pendenza è massima tra quelli il cui bitrate non supera R_{max}. L'analisi condotta giustifica l'algoritmo A1.

Il parametro λ nell'algoritmo rappresenta un limite minimo sulla pendenza $S_i^{j_k}$ della tangente all'involucro convesso, in corrispondenza dei punti di troncamento j_k. Al diminuire di λ, nel calcolo del minimo della somma $(D_i^{n_i^\lambda} + \lambda R_i^{n_i^\lambda})$ i valori $R_i^{n_i^\lambda}$ tendono a crescere e, viceversa, il valore delle distorsioni $D_i^{n_i^\lambda}$ tende a diminuire. Dunque, λ è un peso che bilancia bit-rate e distorsione. Il perché di questa affermazione sarà più chiaro con un esempio. Si consideri un insieme di quattro punti di troncamento, n1, n2, n3, n4, a ciascuno dei quali associamo un bit-rate e una distorsione. La tabella a sinistra riporta i valori di bit-rate e distorsione per i quattro punti di troncamento, mentre quella a destra mostra i valori della somma $(D_i^{n_i^\lambda} + \lambda R_i^{n_i^\lambda})$ ottenuti con λ diversi. Al decrescere di λ, il peso del bit-rate nella sommatoria diminuisce, cioè siamo più disposti ad accettare un aumento di bit-rate pur di ottenere una diminuzione di distorsione.

truncation point	D	R	D + λ R			λ
			1	0.5	0.1	
n1	16	3	19	17.5	16.3	
n2	10	9	19	14.5	10.9	
n3	9	13	22	15.5	10.3	
n4	6	18	24	15	7.8	

Tabella 5: Esempio di calcolo del punto di troncamento ottimale, al variare di λ.

Nel caso $\lambda = 1$, la diminuzione di distorsione, ottenuta passando dal punto di troncamento n1 al punto n2, non giustifica l'aumento di bit necessario. Con $\lambda = 0.5$, siamo disposti a codificare il punto n2, ma non i successivi. Infine, per $\lambda = 0.1$ il punto di troncamento ottimale è n4. Si noti, comunque, che se il bit-rate massimo fosse, ad esempio, $R_{max} = 15$, in corrispondenza di $\lambda = 0.1$ dovremmo accontentarci del punto di troncamento n3.
I diagrammi seguenti descrivono l'algoritmo per il calcolo dei punti di troncamento ottimali (Algoritmo A.1) e quello per il calcolo dell'insieme N_i.

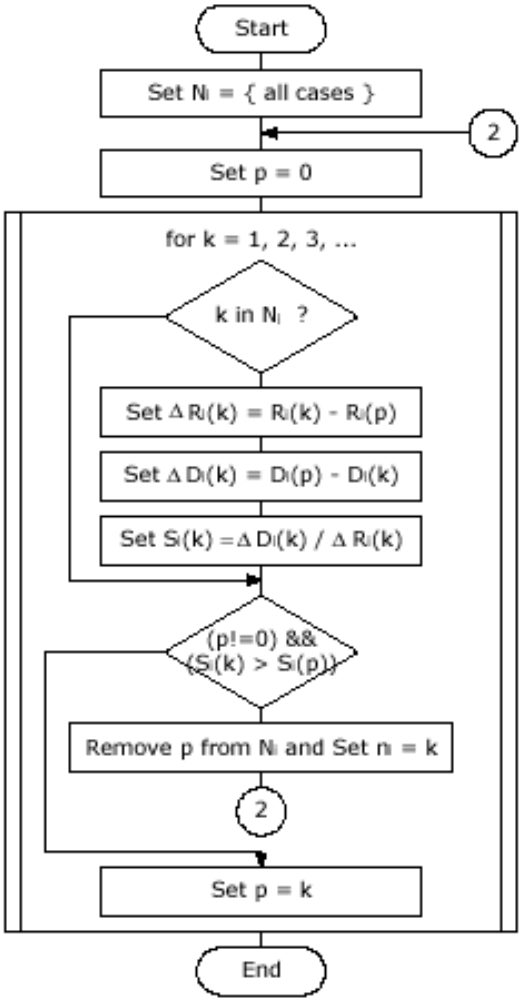

Figura 18: Algoritmo per il calcolo dei punti dell'insieme N_i.

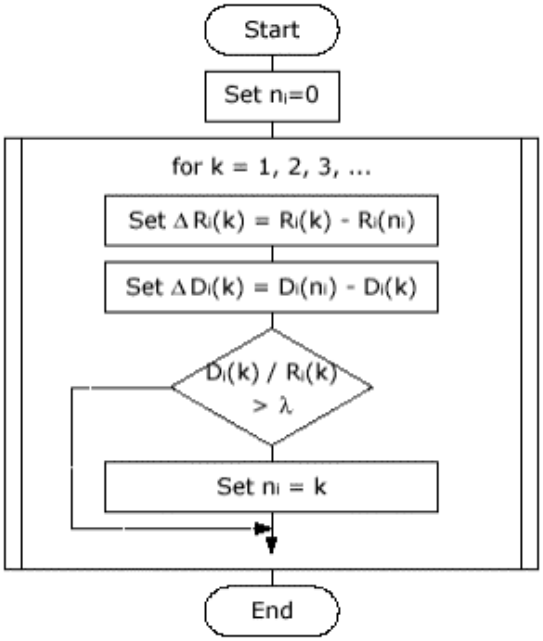

Figura 19: Algoritmo per il calcolo dei punti di troncamento ottimali (A.1).

Risulta chiaro, a questo punto, perché un insieme ampio di punti di troncamento sopperisce all'inefficienza di questo approccio nel caso discreto. Infatti, maggiore è il numero dei punti di troncamento, maggiore è il numero di quelli che giacciono nell'involucro convesso, quindi nell'insieme N_i. All'aumentare dei punti nell'involucro convesso, la soluzione trovata dall'algoritmo si avvicina, in generale, alla soluzione ottimale del problema.

3.3. Codifica dei Blocchi

Come messo in evidenza dalla precedente discussione, l'approccio al problema dell'ottimizzazione bit-rate/distorsione mediante lo studio dell'involucro convesso è efficiente solo per bit-streams dotati di un numero adeguato di punti di troncamento. Tali punti, inoltre, devono essere scelti in maniera tale da trasmettere l'informazione più importante prima possibile. Un metodo comune per risolvere questo problema è la codifica per *bit-plane*, che consiste nel codificare prima il bit più significativo di ciascun coefficiente del blocco, bit che costituiscono il bit-plane più significativo, poi i bit dei bit-planes successivi, in ordine di magnitudo.

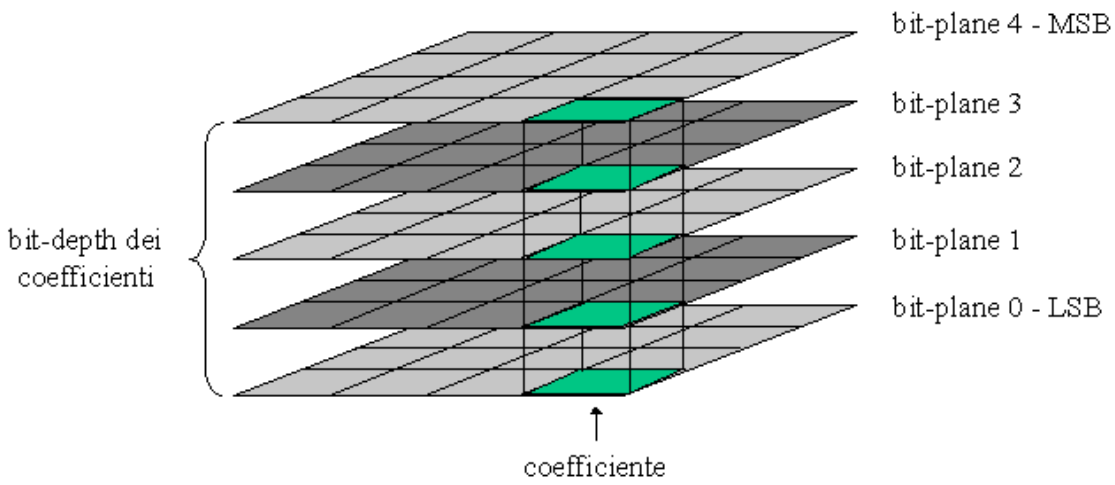

Figura 20: Un blocco diviso in 5 bit-planes. Sono evidenziati i bit relativi ad un coefficiente wavelet quantizzato.

Nella figura è mostrato un blocco costituito da coefficienti wavelet quantizzati rappresentati mediante word di 5 bits. La trasmissione avviene per bit-plane, in ordine decrescente dal più significativo al meno significativo. Evidentemente, essendo costituito dai MSBs di tutti i coefficienti del blocco, il primo bit-plane trasmesso è quello che contiene l'informazione più importante e la rilevanza dei bit-planes è strettamente decrescente rispetto all'ordine con cui essi sono inseriti nello stream, quindi la codifica per bit-plane raggiunge l'obiettivo della trasmissione progressiva ed inoltre ciascun bit-plane è un punto di troncamento naturale dello stream compresso. Tuttavia, 'tagliare' un certo numero n di bit-planes corrisponde a quantizzare i coefficienti con un passo di ampiezza $\Delta=2^n$, che è una codifica non sufficientemente granulare per i nostri scopi. In altre parole, i punti di troncamento forniti dai limiti dei bit-planes non sono in numero sufficiente per rendere efficiente l'analisi dell'involucro convesso discussa sopra, perciò è necessario trovare altri punti di troncamento all'interno di ciascun bit-plane. Per raggiungere questo scopo, JPEG2000 utilizza il cosiddetto *context modeling* (modellizzazione dei contesti), che descriveremo qui di seguito.
Per ciascun bit-plane, la codifica procede in tre passi distinti[5]. Denotiamo con $P_i^{p,1}$, $P_i^{p,2}$ e $P_i^{p,3}$ l'informazione codificata in ciascuno dei passi di codifica, dove p e i sono rispettivamente il bit-plane e l'indice del blocco cui i passi di codifica *1*, *2* e *3* appartengono. La figura seguente mostra l'organizzazione del bit-stream di un code-block codificato.

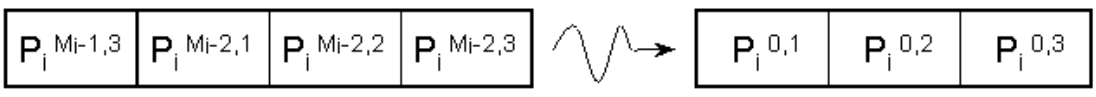

Figura 21: Composizione e organizzazione del bit-stream di un code-block codificato.

L'idea è quella di visitare ogni coefficiente in uno e uno solo dei tre passi associati a ciascun bit-plane p. In questo modo, si ottengono tre insiemi di bit disgiunti ed esaustivi

[5] Nel suo lavoro originale, Taubman aveva definito un *intorno* più esteso e quattro passi di codifica anziché tre.

per ciascun bit-plane, detti *Bit-Planes Frazionati*; i bit sono codificati seguendo l'ordine indotto dai passi di codifica in cui sono stati visitati. Poiché ciascun bit-plane è partizionato in tre bit-planes frazionati, ciascuno dei quali genera una codeword la cui decodifica è indipendente dai bit codificati di seguito, il bit-stream può essere troncato alla fine di ogni passo. In questo modo otteniamo un insieme di punti di troncamento più ampio e il bit-stream può essere spezzato in maniera più granulare. Perché questo meccanismo funzioni correttamente, è necessario scegliere con cura gli insiemi che costituiranno la partizione di ciascun bit-plane e, poiché le word codificate in ciascun passo di codifica sono trasmesse secondo un ordine fissato, anche l'ordine in cui detti passi vengono eseguiti è rilevante. È dunque necessario 'catalogare' ciascun bit secondo un certo schema che tenga in considerazione ciò che è già stato codificato per il coefficiente di cui il bit fa parte, ciò che sta intorno al bit corrente e il valore del bit stesso, cioè il suo *contesto*. Un'accurata modellizzazione dei contesti, inoltre, permette di mettere in evidenza alcune ridondanze tra bit vicini in uno stesso bit-plane e tra un bit-plane e quello successivo. Il *context modeling* ha, dunque, un duplice compito all'interno del sistema di codifica JPEG2000: da una parte, mettendo in evidenza determinate ridondanze spaziali tra i bit in un code-block, permette al codificatore entropico di raggiungere performance migliori, dall'altra, fornendo un numero adeguato di punti di troncamento aggiuntivi, permette all'algoritmo di ottimizzazione bit-rate/distorsione di limitare la sub-ottimalità dovuta all'approccio mediante lo studio dell'involucro convesso, nella fase seguente la compressione entropica (tier 2).
Sarà descritto ora il ruolo di ciascun passo di codifica e in seguito saranno illustrate le primitive di codifica, che si occupano della modellizzazione dei contesti in EBCOT. Introduciamo, innanzi tutto, la notazione necessaria ad affrontare l'argomento. Diciamo *intorno* (*neighborhood*) di un bit l'insieme dei bit 'vicini' appartenenti allo stesso bit-plane, nel blocco corrente.

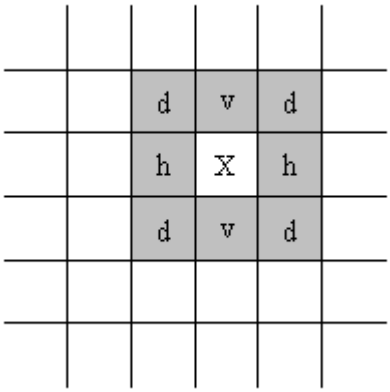

Figura 22: Un bit-plane di un blocco. L'intorno del bit contrassegnato con il simbolo X è ombreggiato.

Sia $q_i[m,n]$ un coefficiente quantizzato nell'*i*-esimo blocco, con *m* ed *n* indici rispettivamente di riga e colonna del coefficiente all'interno del blocco. Denotiamo con $\zeta_i[m,n]$ il bit di segno e con $v_i[m,n]$ la magnitudo del coefficiente $q_i[m,n]$. Siano, inoltre, M_i il numero massimo di bit nella magnitudo dei coefficienti quantizzati e $v_i^p[m,n]$ il *p*-esimo bit nella rappresentazione intera di $v_i[m,n]$, dove $0 \le p < M_i$ e $p = 0$ corrisponde al bit meno significativo. Durante la codifica del blocco sono utilizzate tre variabili di stato che concorrono alla formazione dei contesti:

- $\sigma_i[m,n]$ è una variabile binaria inizializzata a 0, che è posta a 1 quando si trova il primo bit non nullo durante la scansione di $v_i[m,n]$. Se $\sigma_i[m,n]=1$ il coefficiente $q_i[m,n]$ è detto *significativo*, altrimenti è *non-significativo*.
- $\eta_i[m,n]$ è una variabile binaria, usata per indicare se un coefficiente è stato visitato o no nel passo di codifica corrente. In particolare, $\eta_i[m,n]=1$ se il coefficiente $q_i[m,n]$ è stato visitato, zero altrimenti.
- $\delta_i[m,n]$ è anch'essa una variabile binaria, usata per indicare se l'operazione di *Magnitude Refinement* descritta di seguito è stata applicata o meno al coefficiente $q_i[m,n]$.

Adesso siamo in condizione di descrivere i tre passi di codifica usati nella scansione di ciascun bit-plane.

- **Significance Propagation pass, $P_i^{p,1}$ (passo di propagazione della significanza)**: Durante questo passo, i bit di ciascun bit-plane sono scanditi seguendo un pattern di scansione fissato, che sarà descritto in seguito. I bit codificati durante questo passo sono quelli appartenenti a coefficienti non-significativi che hanno un cosiddetto *intorno significativo* completamente contenuto nel blocco corrente. Un coefficiente ha un *intorno significativo* (*preferred neighborhood*) se almeno uno dei suoi otto immediati vicini è significativo. I bit nell'intorno del bit in esame che non sono contenuti nel blocco sono considerati non-significativi. Ad ogni coefficiente $q_i[m,n]$ ancora non-significativo (cioè tale che $\sigma_i[m,n]=0$) e che ha un intorno significativo è applicata la primitiva *Zero Coding*, che sarà descritta nel seguito, per determinare se diventa significativo o meno nel bit-plane corrente. Se diventa significativo, cioè $v_i^p[m,n]=1$, è invocata la primitiva *Sign Coding* per la codifica del segno del coefficiente. Indipendentemente dal fatto che il coefficiente diventi significativo o meno, la variabile $\eta_i[m,n]$ è posta a 1 per segnalare ai passi di codifica successivi che tutta l'informazione rilevante, relativa al coefficiente in esame, è stata codificata per questo bit-plane. Questo assicura che nessun bit che è codificato in questo passo possa essere codificato in nessun altro passo successivo.

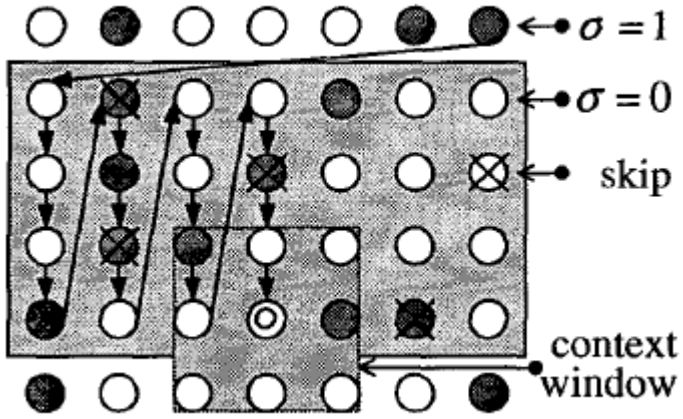

Figura 23: Scansione dei coefficienti durante la codifica nel Significance Propagation Pass. I coefficienti significativi sono ombreggiati, quelli non codificati in questo passo sono indicati mediante una croce.

Questo passo è detto di "propagazione della significanza" perché, quando un coefficiente diventa significativo in un certo bit-plane *p*, tutti i coefficienti successivi nell'ordine di scansione che hanno tale coefficiente nel loro intorno sono processati da questo passo, quindi i coefficienti che diventano significativi in *p* sono i punti di partenza di una sequenza nuove decisioni di propagazione della significanza, che si propaga lungo il bit-plane seguendo l'ordine di scansione.

- **Magnitude Refinement pass, $P_i^{p,2}$ (passo di raffinamento della magnitudo)**: Durante questo passo, seguendo lo stesso ordine di scansione del passo precedente, si codificano i bit che appartengono a coefficienti significativi ($\sigma_i[m,n]=1$) e per cui non è stata codificata alcuna informazione al passo precedente ($\eta_i[m,n]=0$). Questi bit sono processati mediante la primitiva di *Magnitude Refinement* che si occupa di codificare l'informazione utile a raffinare il valore del coefficiente in esame, aggiungendo bit meno significativi al coefficiente codificato. Non c'è alcun bisogno di modificare la variabile $\eta_i[m,n]$, poiché il passo di codifica successivo ignora i bit significativi e azzera il valore di $\eta_i[m,n]$ per ogni coefficiente del blocco.
- **Normalization (Cleanup) pass, $P_i^{p,3}$ (passo di normalizzazione)**: Questo passo, seguendo l'ordine di scansione usato dai precedenti passi, codifica tutti i bit rimanenti, cioè quelli non-significativi ($\sigma_i[m,n]=0$), per cui nessuna informazione è stata codificata durante il passo di propagazione della significanza ($\eta_i[m,n]=0$). Tali bit sono codificati utilizzando una combinazione delle primitive *Zero Coding* e *Run-Length Coding*; se il coefficiente diventa significativo, è invocata la primitiva *Sign Coding* per codificare il segno. Alla fine del passo, tutti i coefficienti sono marcati come non-visitati ($\eta_i[m,n]$ è posto a zero), in preparazione per il primo passo di codifica del bit-plane successivo. Questo passo codifica tutti i bit che non sono stati codificati nei passi precedenti; è detto di "normalizzazione" perché, dopo averlo eseguito, tutti i bit del bit-plane corrente sono codificati.

Le operazioni di codifica descritte partizionano ciascun bit-plane in tre aree, ciascuna contenente bit che hanno caratteristiche simili ai fini della codifica. Per decidere a quale di queste tre aree deve appartenere un certo bit, si esaminano le informazioni relative al bit stesso e ai sui otto immediati vicini, raccolte durante la codifica dei bit-planes precedenti. Una prima importante distinzione è data dalla significanza del coefficiente in esame. Se il coefficiente è significativo, il bit più significativo è il segno del coefficiente sono stati codificati: tutto ciò che dobbiamo fare è codificare il nuovo bit per raffinare il valore del coefficiente (*Magnitude Refinement*). Se, invece, il coefficiente non è ancora significativo, si esamina la significanza dei vicini. Se tutti i vicini sono non-significativi, è poco probabile che il coefficiente in esame diventi significativo in questo bit-plane (probabilmente fa parte di una zona omogenea). Questo bit è, dunque, codificato nell'ultimo passo (*Normalization*) perché è improbabile che aggiunga informazione significativa alla codifica. Infine, se il coefficiente in esame ha un vicino significativo, è probabile che il coefficiente corrente diventi significativo in questo bit-plane. Poiché la significanza di un coefficiente è una informazione che ha un notevole valore, questo bit verrà codificato nel primo passo (*Significance Propagation*). Il diagramma seguente chiarisce quanto detto.

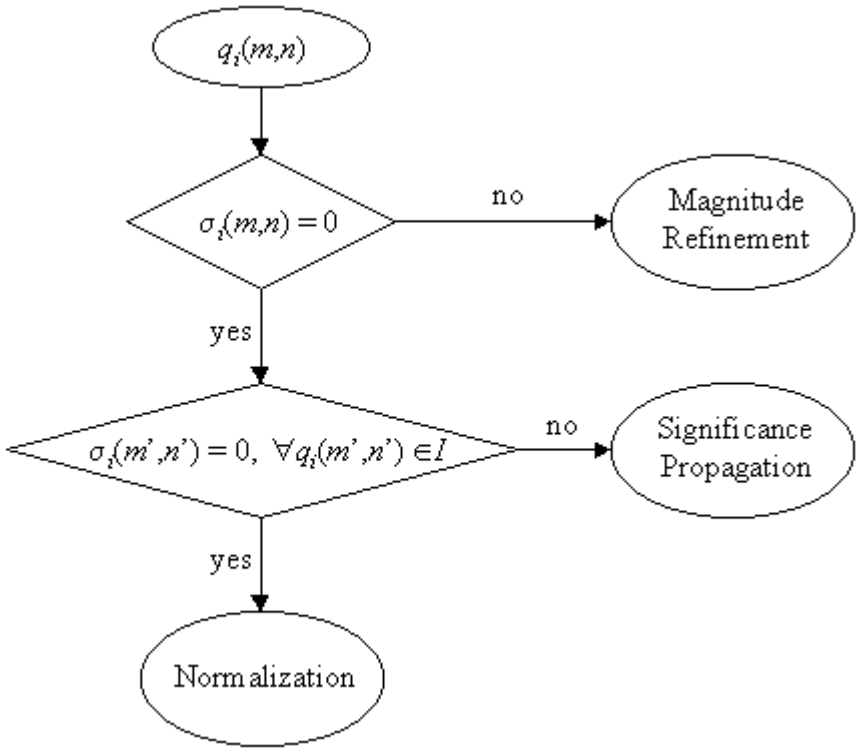

Figura 24: Criterio per stabilire l'appartenenza del coefficiente $q_i[m,n]$ ad uno dei tre passi di codifica. *I* indica l'insieme dei coefficienti nell'intorno di $q_i[m,n]$.

Le precedenti considerazioni suggeriscono anche l'ordine in cui le sequenze codificate per ciascun passo devono apparire nel bit-stream codificato, nell'ordine: *Significance Propagation*, *Magnitude Refinement*, *Normalization*. I risultati sperimentali dimostrano la correttezza di questo ragionamento.
Si noti che ciascuno dei tre passi di codifica descritti costituisce un'unità di codice indivisibile e identifica un punto di troncamento.
Abbiamo visto che la codifica di un bit-plane avviene mediante tre scansioni, una per ognuno dei passi di codifica. Un'osservazione importante riguarda il modo in cui i bit di un bit-plane, codificati in passi diversi, sono localizzati dal decoder. Nessuna informazione, infatti, è inserita nello stream compresso per indicare al decoder quali bit sono stati codificati in un certo passo e quali in un altro. In realtà, tale informazione non è necessaria perché le *etichette di contesto* (di cui si discuterà più avanti) sono assegnate ai coefficienti nello stesso modo sia dall'encoder che dal decoder. Inoltre, al momento di decodificare un coefficiente, le informazioni utilizzate per stabilire il contesto ad esso relativo saranno in possesso del decoder. Infatti, per assegnare un contesto ad un coefficiente $q_i[m,n]$, si utilizza la significanza di $q_i[m,n]$ stesso e degli otto vicini immediati, quindi, a patto di seguire lo stesso pattern nella scansione dei coefficienti del blocco e di utilizzare le stesse etichette di contesto, la significanza dei coefficienti sarà aggiornata nel medesimo modo sia in fase di codifica che di decodifica. La figura seguente è un esempio dei tre passi di codifica.

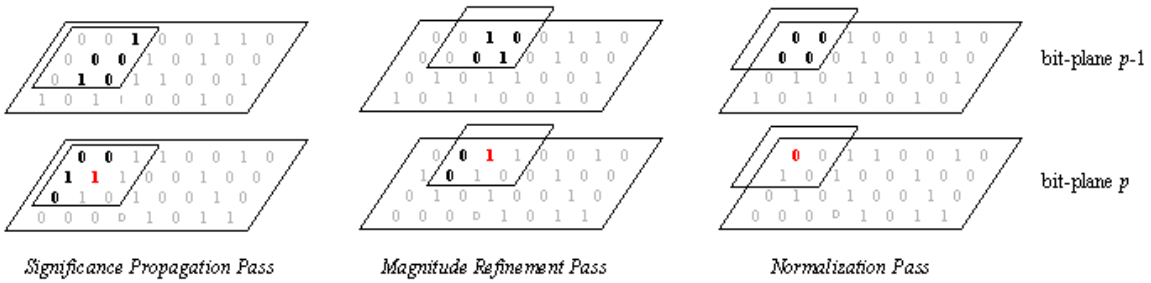

Figura 25: Esempio dei tre passi di codifica. Il bit da codificare è evidenziato in rosso. I bit in grassetto sono quelli coinvolti nel calcolo della significanza. Le didascalie sottostanti i tre esempi indicano il passo in cui il bit in rosso sarà codificato, supponendo che i coefficienti coinvolti siano tutti non-significativi prima del bit-plane p-1.

Osserviamo che, quando codifichiamo il primo bit-plane, tutti i coefficienti sono non-significativi, quindi l'unico passo possibile è quello di normalizzazione. Questo giustifica il fatto che sul primo bit-plane sia eseguito solo il passo di normalizzazione, come mostrato in Figura 21.
Prima di passare a illustrare le primitive di codifica cui si accennava sopra, descriviamo l'ordine seguito dai passi di codifica nella scansione dei coefficienti del blocco da codificare. Il code-block è processato a strisce (*stripes*) di quattro righe ciascuna che coprono l'intera larghezza del blocco. All'interno di una striscia, la scansione procede per colonne, dall'alto verso il basso. Quindi, prima sono scanditi i quattro coefficienti della prima colonna, poi quelli della colonna successiva, immediatamente a destra, e così via. Quando un'intera striscia è stata scandita, si procede con quella più in basso. La figura seguente chiarisce quanto detto.

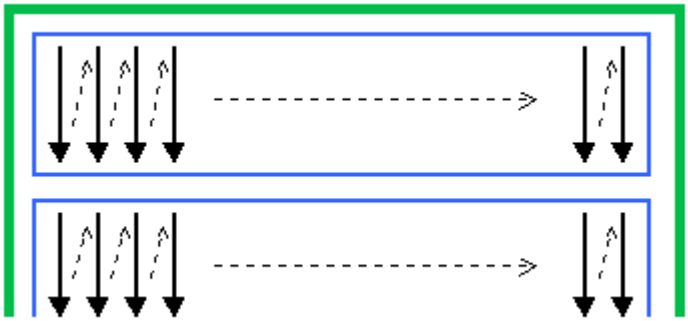

Figura 26: Schema di scansione fissato seguito da ogni passo di codifica.

Descriveremo adesso le primitive di codifica invocate durante l'esecuzione dei passi di codifica su ogni bit-plane. Tali operazioni possono essere pensate come delle funzioni di basso livello che implementano la modellizzazione dei contesti. Dato un coefficiente quantizzato $q_i[m,n]$, in corrispondenza di un bit-plane p, se il coefficiente non è ancora significativo, cioè $\sigma_i[m,n]=0$, è usata una combinazione delle primitive *Zero Coding* e *Run-Lenght* per codificare se il simbolo diventa significativo nel bit-plane corrente, cioè $v_i^{\,p}[m,n]=1$, o meno. In caso affermativo, è invocata anche la primitiva *Sign Coding* per codificare il segno $\chi_i[m,n]$. Se il coefficiente è già significativo, $\sigma_i[m,n]=1$, è usata la primitiva *Magnitude Refinement* per codificare il nuovo bit $v_i^{\,p}[m,n]$ del coefficiente $q_i[m,n]$. Il compito di queste primitive è quello di associare un contesto a ciascun bit del

bit-plane corrente, in base alle informazioni disponibili sullo stato dei bit vicini. Le primitive di codifica sono descritte qui di seguito.

- **Zero Coding (ZC) - codifica degli zeri**: questa primitiva è invocata per codificare un simbolo appartenente ad un coefficiente non ancora significativo. La scelta di uno tra nove contesti diversi è guidata dalla significanza degli otto immediati vicini, tenendo conto del tipo di sottobanda cui il blocco appartiene. L'intorno rilevante è illustrato in Figura 22. I vicini sono raggruppati in tre categorie: orizzontali, verticali e diagonali. Indichiamo con *h*, *v* e *d*, rispettivamente, il numero di vicini orizzontali, verticali e diagonali che sono già significativi. Evidentemente valgono le relazioni $0 \le h \le 2$, $0 \le v \le 2$ e $0 \le d \le 4$. I vicini che non appartengono al blocco corrente sono considerati non-significativi ai fini della codifica del simbolo corrente, per evitare dipendenza tra i blocchi. La tabella seguente identifica, per ciascun tipo di sottobanda, la relazione tra i contesti e le tre quantità descritte.

LH sub-band (also used for LL)				HL sub-band				HH sub-band		
h	v	d	context	h	v	d	context	d	h + v	context
2	x	x	8	x	2	x	8	>= 3	x	8
1	>= 1	x	7	>= 1	1	x	7	2	>= 1	7
1	0	>= 1	6	0	1	>= 1	6	2	0	6
1	0	0	5	0	1	0	5	1	>= 2	5
0	2	x	4	2	0	x	4	1	1	4
0	1	x	3	1	0	x	3	1	0	3
0	0	>= 2	2	0	0	>= 2	2	0	2	2
0	0	1	1	0	0	1	1	0	1	1
0	0	0	0	0	0	0	0	0	0	0

Tabella 6: Mappa di assegnamento dei contesti per la primitiva Zero Coding.

- **Run-Length Coding (RLC)**: la primitiva RLC è usata per ridurre il numero di simboli binari che devono essere codificati dal codificatore aritmetico. È invocata al posto di ZC se e solo se valgono le condizioni:
  - **a)** quattro coefficienti consecutivi devono essere non-significativi, cioè $\sigma_i[m,n] = 0$;
  - **b)** tutti e quattro i coefficienti devono avere intorno nullo, cioè le variabili di contesto *h*, *v* e *d* introdotte sopra devono valere zero per ciascuno dei quattro coefficienti;
  - **c)** il gruppo dei quattro coefficienti deve costituire una colonna di quattro coefficienti allineati rispetto all'ordine di scansione descritto sopra. In altre parole, gruppi legali di quattro coefficienti iniziano a partire dalle posizioni 1°, 5°, 9°, … della scan-line rilevante. Questa condizione è motivata da ragioni di efficienza nell'implementazione dello schema di raggruppamento dei simboli, sia in hardware che in software.

  Quando si codifica un gruppo di coefficienti che soddisfano le precedenti condizioni, un singolo bit è codificato per indicare se qualche coefficiente nel gruppo è diventato significativo nel bit-plane corrente o meno (rispettivamente 1 o 0). Il simbolo è codificato utilizzando un singolo contesto. Se qualcuno dei coefficienti diventa significativo ($v_i^p[m,n] = 1$), l'indice (0, 1, 2 o 3) del primo di essi è trasmesso come una quantità a due bit, con il bit più significativo seguito da quello meno significativo. Il contesto usato per codificare questi bit, detto *uniforme*, è associato ad un modello di probabilità che riflette una distribuzione uniforme e non adattativa.
- **Sign Coding (SC) - codifica del segno**: la primitiva SC è usata al più una volta per ciascun coefficiente di ogni blocco, immediatamente dopo che un simbolo non-significativo diventa significativo durante un'operazione ZC o RLC. Quando la procedura è invocata, $\sigma_i[m,n] = 0$ e $v_i^p[m,n] = 1$. La variabile di stato che rappresenta

la significanza del simbolo ($\sigma_i[m,n]$) è posta a 1 per indicare che il simbolo è diventato significativo e il segno $\chi_i[m,n]$ è codificato mediante l'uso di uno tra cinque contesti, che dipendono da segno e significanza degli immediati vicini verticali e orizzontali. Definiamo

$$h = \min\{1, \max\{-1, \sigma_i[m,n-1]\cdot(1-2\chi_i[m,n-1]) + \sigma_i[m,n+1]\cdot(1-2\chi_i[m,n+1])\}\} \quad \text{E.19}$$

e

$$v = \min\{1, \max\{-1, \sigma_i[m-1,n]\cdot(1-2\chi_i[m-1,n]) + \sigma_i[m+1,n]\cdot(1-2\chi_i[m+1,n])\}\} \quad \text{E.20}$$

Come risulta chiaro dalle formule precedenti, *h* si annulla se entrambi i vicini orizzontali sono non-significativi o sono entrambi significativi ma hanno segno opposto; altrimenti *h* vale 1 se uno o entrambi i vicini orizzontali sono significativi ed hanno segno positivo, -1 se hanno segno negativo. La stessa osservazione vale per la variabile *v*, utilizzando i vicini verticali. Come per ZC, i coefficienti che stanno al di fuori del blocco corrente sono considerati non-significativi.
Evidentemente, ci sono nove possibili permutazioni dei valori di *h* e *v*. Alcune di queste, comunque, sono ridondanti. In particolare, non c'è da aspettarsi una deviazione incondizionata dei bit di segno, quindi le probabilità condizionate $P(\chi \mid h,v)$ e $P(\overline{\chi} \mid h,v)$ dovrebbero essere identiche. Questa simmetria è utilizzata per ridurre il numero di contesti per la codifica del segno a 5, con l'aiuto di un *predittore di segno* $\hat{\chi}$. Il simbolo binario codificato sarà $\hat{\chi} \otimes \chi_i[m,n]$, dove $\otimes$ denota l'operatore di OR esclusivo. La tabella riportata di seguito mappa i valori delle variabili *h* e *v* sui contesti e il predittore di segno $\hat{\chi}$.

h	v	$\hat{\chi}$	context
1	1	0	4
1	0	0	3
1	-1	0	2
0	1	0	1
0	0	0	0
0	-1	1	1
-1	1	1	2
-1	0	1	3
-1	-1	1	4

Tabella 7: Mappa di assegnamento dei contesti per la primitiva Sign Coding.

- **Magnitude Refinement (MR) - raffinamento della magnitudo**: l'operazione di MR è chiamata quando il simbolo da codificare, $v_i^{\,p}[m,n]$, è già significativo, cioè $\sigma_i[m,n]=1$. Il contesto associato al bit da codificare è determinato utilizzando informazioni sul suo intorno e in base al fatto che questo sia o meno il primo bit-plane per cui l'operazione di magnitude refinement è applicata al coefficiente. Per mantenere quest'ultima informazione utilizziamo la variabile $\delta_i[m,n]$, che indica se la primitiva MR è applicata al coefficiente per la prima volta in questo bit-plane. La variabile è inizializzata a zero ed è posta ad uno alla fine dell'operazione MR. Definiamo *h* e *v* come per ZC; l'assegnamento dei contesti è indicato dalla tabella seguente.

δ	h + v	context
1	x	2
0	>= 1	1
0	0	0

Tabella 8: Mappa di assegnamento dei contesti per la primitiva Magnitude Refinement.

Osserviamo che le tabelle utilizzate nelle primitive ZC e SC possono essere implementate in hardware così come sono descritte, utilizzando un numero ridotto di porte, mentre per l'implementazione software è conveniente utilizzare una lookup table che, pur essendo ridondante, permette di ottenere una maggiore efficienza in termini di running-time.
I contesti determinati dalle precedenti primitive sono usati dal codificatore aritmetico per guidare la stima di probabilità associata a ciascun simbolo. Ad ogni contesto è associato un indice in una tabella che contiene informazioni utili a stabilire l'identità del simbolo più probabile e la probabilità che l'evento atteso si verifichi. In base a queste stime e all'identità del simbolo da codificare, il codificatore aritmetico è in grado di determinare gli estremi dell'intervallo associato alla sequenza di simboli in input. Il modo in cui un intervallo venga associato ad una stringa di bit sarà argomento della prossima sezione.
Concludiamo la sezione dedicata al context modeling con una breve discussione sull'inizializzazione del codificatore entropico. La tabella seguente mostra le etichette di contesto, associate a ciascuna primitiva, con le relative informazioni di inizializzazione.

context label	primitive	initial state	initial MPS
0	ZC	4	0
1 - 8	ZC	0	0
9 - 13	SC	0	0
14 - 16	MR	0	0
17	RLC	3	0
18	UNIFORM	46	0

Tabella 9: Inizializzazione dei contesti in EBCOT. Sono indicati l'etichetta di contesto, la primitiva cui il contesto fa riferimento, lo stato iniziale e il simbolo più probabile iniziale.

In aggiunta a 18 contesti dedicati alla codifica adattativa, EBCOT definisce un contesto *uniforme*, inizializzato allo stato speciale 46 (vedi Tabella 12), che riflette un comportamento non adattativo, ovvero una distribuzione di probabilità uniforme. Lo stato 46, infatti, non può essere raggiunto da nessun altro stato e non può raggiungere alcun altro stato. Molti dei contesti adattativi sono inizializzati allo stato zero, che assegna approssimativamente le stesse probabilità ai simboli e corrisponde all'inizio della curva di apprendimento. I contesti 0 e 17, invece, sono inizializzati, rispettivamente, allo stato 4 e allo stato 3, in cui la probabilità del simbolo meno probabile (in questo caso "1") è molto bassa. Questa scelta è giustificata dal fatto che tali contesti sono associati a coefficienti in zone con molti coefficienti non-significativi, quindi la probabilità che tali coefficienti diventino significativi è molto bassa.
Infine, tutti i coefficienti sono inizializzati con 0 come simbolo più significativo.

Capitolo 4

Codifica Aritmetica: MQ-Coder

In questo capitolo descriveremo brevemente i principi generali che stanno alla base della codifica aritmetica e le modifiche necessarie per adattare l'algoritmo all'aritmetica finita di un calcolatore. Infine discuteremo in dettaglio il codificatore aritmetico usato da JPEG2000, *MQ-Coder*. Ulteriori informazioni possono essere trovate in [6], per la descrizione generale della codifica aritmetica, e in [12], per QM-Coder, di cui MQ-Coder è una variante.

4.1. Codifica Aritmetica Ideale

La codifica aritmetica mappa una sequenza di simboli in un intervallo di numeri reali. L'intervallo iniziale, [0,1), corrisponde alla situazione in cui nulla è stato ancora codificato. Ad ogni passo, l'intervallo corrente è partizionato in sottointervalli, ciascuno associato ad un simbolo dell'alfabeto e di ampiezza proporzionale alla probabilità che tale simbolo appaia al passo successivo. Man mano che nuovi simboli sono codificati, l'intervallo corrente è sostituito dal sottointervallo associato al simbolo codificato. La sequenza di simboli codificata può essere ricostruita in maniera esatta, dato un qualsiasi punto nell'intervallo. Convenzionalmente il punto dell'intervallo scelto per rappresentare l'informazione codificata è l'estremo sinistro dell'intervallo, che è detto *code point* (*punto di codice*).

Denotiamo con $\Sigma = \{s_0, s_1, ..., s_k\}$ l'alfabeto cui i simboli della stringa da codificare appartengono e associamo una stima di probabilità a ciascun simbolo di Σ. Siano $p = \{p_0, p_1, ..., p_k\}$ e $P = \{P_0, P_1, ..., P_k\}$, dove p_i è la stima di probabilità associata a s_i e P_i è la somma delle probabilità associate ai simboli di indice inferiore a k. Si noti che i simboli $p_i \in p$ e $P_i \in P$ rappresentano, rispettivamente, l'ampiezza e l'estremo sinistro dell'intervallo di probabilità associato al simbolo $s_i \in \Sigma$. Durante l'esecuzione dell'algoritmo, useremo due variabili c ed a, rispettivamente base e ampiezza dell'intervallo che codifica la sequenza di simboli processati. L'inizializzazione $c = 0$, $a = 1$ riflette la definizione dell'intervallo iniziale, [0,1). Ad ogni passo dell'algoritmo, i

valori di c e a sono aggiornati, in accordo alla stima di probabilità associata al nuovo simbolo, utilizzando le formule:

$$a_{i+1} = p(s_i) \cdot a_i \qquad \text{(E.21)}$$

$$c_{i+1} = c_i + P(s_i) \cdot a_i \qquad \text{(E.22)}$$

dove l'indice i si riferisce al numero di simboli che sono stati processati, ovvero il numero di iterazioni eseguite dall'algoritmo. Al termine del procedimento, il codificatore aritmetico restituisce il valore di c (code point) che codifica univocamente la sequenza di simboli in input.
La figura seguente mostra un esempio di codifica aritmetica di una parola, mediante stime di probabilità fittizie sull'occorrenza di ciascuna lettera nelle parole del vocabolario inglese.

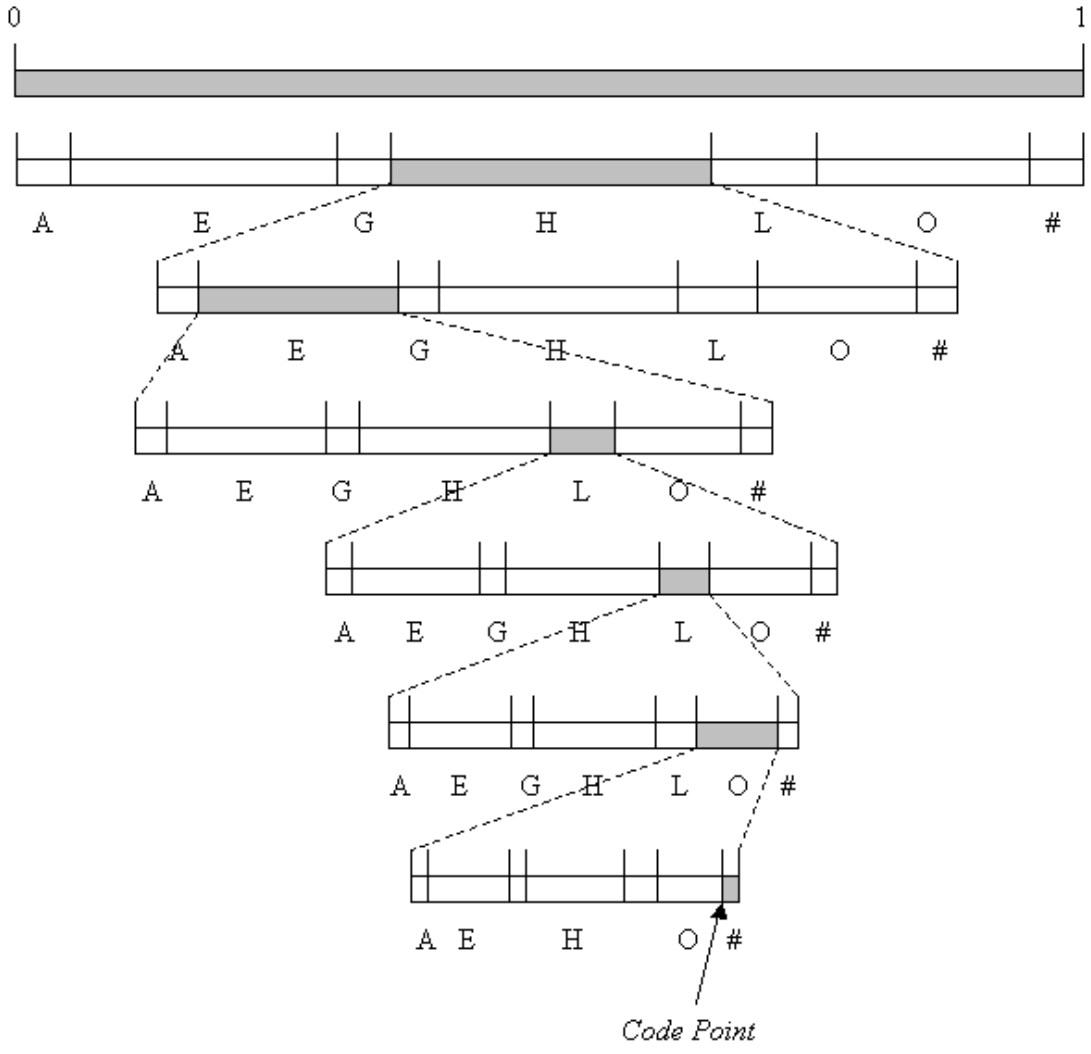

Figura 27: Codifica aritmetica della sequenza di simboli <'H','E','L','L','O','#'>.

Nell'esempio in figura, l'alfabeto utilizzato è $\Sigma = \{\mathrm{A}, \mathrm{E}, \mathrm{G}, \mathrm{H}, \mathrm{L}, \mathrm{O}, \#\}$, l'insieme delle probabilità associate è $p = \{0.05,\ 0.25,\ 0.05,\ 0.3,\ 0.1,\ 0.2,\ 0.05\}$ e l'insieme delle probabilità cumulative è $P = \{0,\ 0.05,\ 0.3,\ 0.35,\ 0.65,\ 0.75,\ 0.95\}$. I valori iniziali delle variabili, $c = 0$ e $a = 1$, rappresentano l'intervallo ombreggiato in figura. Tale intervallo è

partizionato in $Card(\Sigma) = 7$ sottointervalli, ciascuno di ampiezza proporzionale alla stima di probabilità del simbolo associato. Quando arriva il primo simbolo, 'H', gli estremi dell'intervallo vengono aggiornati in base alle formule $c = c + P('H') \cdot a$ e $a = p('H') \cdot a$. A questo punto, il nuovo intervallo è $[0.35, 0.65]$. Il procedimento si ripete con il simbolo 'E', utilizzando l'intervallo appena trovato. I risultati del processo di codifica sono mostrati nella Tabella 10.

s	*p(s)*	*P(s)*	*a*	*p(s) * a*	*c*	*P(s) * a*
H	0.3	0.35	1	0.3	0	0.35
E	0.25	0.05	0.3	0.075	0.35	0.015
L	0.1	0.65	0.075	0.0075	0.365	0.04875
L	0.1	0.65	0.0075	0.00075	0.41375	0.004875
O	0.2	0.75	0.00075	0.00015	0.418625	0.0005625
#	0.05	0.95	0.00015	0.0000075	0.4191875	0.0001425
			0.0000075		0.41933	

Tabella 10: Codifica aritmetica della sequenza di simboli <'H','E','L','L','O','#'>.

Alla fine del processo, il code-point, evidenziato in giallo nella tabella, è la codifica aritmetica della sequenza di simboli <'H','E','L','L','O','#'>. Si osservi che nella tabella precedente il valore della casella *a* ad un certo passo *i* è uguale al valore della casella $p(s) \cdot a$ al passo *i*-1 e, allo stesso modo, la casella *c* è ottenuta della somma dei valori di *c* e $P(s) \cdot a$ al passo precedente. Questo riflette le equazioni di ricorrenza definite sopra.

La decodifica del segnale procede seguendo gli stessi passi della codifica. Partendo dall'intervallo iniziale, [0,1), tra i suoi sotto-intervalli si sceglie quello che contiene il code-point *c* e si mette in output il simbolo corrispondente; infine l'intervallo iniziale è sostituito con il sotto-intervallo scelto. Il procedimento è iterato fino alla decodifica completa della sequenza di simboli originale. Il problema che si pone, a questo punto, è come fare a capire quando l'ultimo simbolo della sequenza è stato decodificato; infatti ad ogni passo, comunque preso un punto nell'intervallo corrente, non è possibile determinare se la fine della sequenza è stata raggiunta o meno, avendo come sola informazione la coordinata del punto. È allora necessario segnalare esplicitamente la fine della sequenza. Un metodo comune per segnalare al decoder la fine della sequenza è quello di aggiungere un simbolo speciale di "fine sequenza"; questo è il ruolo del simbolo '#' nel nostro esempio. Si noti che, per come sono definiti i processi di codifica e decodifica, il decoder è in grado di ricostruire perfettamente il segnale originale, quindi la codifica aritmetica è un modello di codifica entropica lossless.

4.2. Codifica Aritmetica a Precisione Finita

L'algoritmo appena descritto sembra promettere buoni risultati in termini di efficienza, tuttavia rimane una questione importante cui non si è ancora fatto riferimento. Ad ogni riga della precedente tabella, il valore di *a* diventa sempre più piccolo poiché ad ogni passo dell'algoritmo l'ampiezza dell'intervallo corrente è moltiplicata per una probabilità, che è un numero nell'intervallo [0,1). Quindi il numero di cifre necessarie per rappresentare *a* cresce ad ogni passo del processo di codifica. Lo stesso vale per *c*, che ad ogni passo è incrementato di $P(s) \cdot a$. Il fatto che il numero di cifre in *a* e *c* cresca in maniera illimitata non deve sorprendere. Come risulta chiaro dalla presentazione precedente, *c* contiene

informazione sufficiente a ricostruire la sequenza di simboli in input. Poiché la sequenza può essere di lunghezza arbitrariamente grande, è impossibile che un numero di cifre finito, fissato a priori, sia sufficiente a rappresentare *c* per ogni possibile input. È quindi necessario adattare l'algoritmo di codifica aritmetica alla *precisione finita* di un elaboratore reale. Inoltre l'algoritmo deve essere "*on-line*", nel senso che deve emettere i bytes codificati durante il processo di codifica stesso, per evitare l'onere di bufferizzare l'informazione codificata.
Osserviamo che, man mano che la codifica procede, il valore di a_i è strettamente decrescente, essendo prodotto di probabilità tutte minori di 1, quindi il numero degli zeri in testa non può mai diminuire. In altre parole, se scriviamo a_i nella forma $10^{-N_i} ma_i$, dove ma_i è la mantissa di a_i, mantenuta all'interno di un range fissato $\mathfrak{R}$, ed N_i è il numero di zeri in testa, allora la sequenza dei valori N_i è non decrescente. Nella seguente tabella è riportata la sequenza dei valori delle ampiezze degli intervalli ad ogni passo della codifica della sequenza del precedente esempio, con le relative sequenze dei valori delle mantisse, ma_i, e degli esponenti, N_i.

a	*mantissa*	*exponent*
1	1	0
0.3	3	1
0.075	7.5	2
0.0075	7.5	3
0.00075	7.5	4
0.00015	1.5	4

Tabella 11: Rappresentazione mantissa-esponente di una sequenza di ampiezze.

È importante osservare che, grazie alla presenza degli zeri in testa, nella somma $c_{i+1} = c_i + P(s_i) \cdot a_i$ i primi N_i bit di c_i non variano, eccetto che per la propagazione di un eventuale riporto. Queste osservazioni suggeriscono di cercare un algoritmo che operi solo sulla mantissa di a_i e sulla corrispondente porzione di c_i.
Naturalmente, il numero delle cifre necessarie a rappresentare la mantissa di a_i può ancora crescere in maniera illimitata, quindi quanto detto precedentemente risulterebbe inutile, senza un meccanismo che sia capace di limitare il numero di cifre nella rappresentazione di detta mantissa.
Questo può essere fatto osservando che, senza perdere la correttezza della codifica, è possibile mantenere solo un numero finito e fissato a priori di cifre per rappresentare la mantissa di a_i, approssimandone il valore dal basso, cioè troncando le ultime cifre di ma_i.
Torniamo nuovamente all'esempio precedente e supponiamo di volere mantenere una sola cifra decimale per rappresentare la mantissa di a_i. Consideriamo la sequenza delle ampiezze degli intervalli durante l'esecuzione dell'algoritmo. Ad ogni passo, prima calcoliamo il valore esatto di $p(s_i) \cdot a_i$, quindi lo approssimiamo, troncando tutte le cifre decimali della mantissa, tranne quella più significativa. Indichiamo con $\overline{\boldsymbol{p}(s_i) \cdot \boldsymbol{a}_i}$ il valore ottenuto; assegniamo tale valore ad a_{i+1}. Si noti che, poiché la dimensione delle mantisse di $P(s_i)$ e di a_i è limitata rispettivamente a 2 e 1 cifre, la dimensione di $P(s_i) \cdot a_i$ è anch'essa limitata, a tre cifre. Infine, si osservi che, ad ogni passo della codifica, solo queste tre cifre sono sommate al valore di c_i, quindi solo le corrispondenti cifre di c_i possono essere modificate (eccetto che per la propagazione di eventuali riporti).
La tabella seguente riassume i passi eseguiti dall'algoritmo a precisione finita per la codifica della sequenza di simboli <'H','E','L','L','O','#'> dell'esempio precedente.

s	p(s)	P(s)	a	p(s) * a	^(p(s) * a)	c	P(s) * a
H	0.3	0.35	1	0.3	0.3	0	0.35
E	0.25	0.05	0.3	0.075	0.07	0.35	0.015
L	0.1	0.65	0.07	0.007	0.007	0.365	0.0455
L	0.1	0.65	0.007	0.0007	0.0007	0.4105	0.00455
O	0.2	0.75	0.0007	0.00014	0.0001	0.41505	0.000525
#	0.05	0.95	0.0001	0.000005	0.000005	0.415575	0.000095
			0.000005			0.41567	

Tabella 12: Codifica aritmetica con precisione finita della sequenza di simboli <'H','E','L','L','O','#'>.

Come risulta chiaro dal confronto della Tabella 12 con la Tabella 10, durante il processo di codifica, le ampiezze degli intervalli sono più piccole approssimando il valore della mantissa; ciò in generale vale anche per i valori di c_i. Tuttavia, purché encoder e decoder utilizzino la stessa strategia di approssimazione, l'encoder sarà sempre in grado di ricostruire esattamente la sequenza di simboli codificata.
Le precedenti osservazioni suggeriscono che è possibile costruire un codificatore aritmetico che mantenga solamente la mantissa di a_i, con una certa precisione finita e fissata a priori, e la porzione di c_i sulla quale possono influire le cifre di detta mantissa. La parte rimanente di c_i, che si trova a sinistra della porzione sulla quale vengono eseguite le operazioni di codifica, è il *code stream*, che costituisce la rappresentazione compressa della sequenza di simboli in input ed è messa in output durante il corso dell'algoritmo. Poiché operiamo solo su due stringhe di cifre di lunghezza finita, mettendo in output le cifre di c_i che non sono interessate dalle operazioni eseguite dall'algoritmo, abbiamo ottenuto un algoritmo che è sia a precisione finita che on-line.
Daremo adesso i dettagli dell'algoritmo di codifica. Utilizziamo la seguente convenzione tipografica: i simboli scritti con un carattere normale indicano il valore esatto della quantità rappresentata, mentre i simboli scritti in grassetto indicano i bits correntemente memorizzati per rappresentare il valore di detta quantità. Così, per esempio, indicheremo con "a_i" il valore esatto di a_i e con "$\boldsymbol{a_i}$" il valore correntemente memorizzato per a_i. Al fine di rendere più chiara l'esposizione, introduciamo la seguente notazione:

#p	numero di bit per la rappresentazione di $p(s_i)$ e $P(s_i)$
$\boldsymbol{p(s_i)}$, $\boldsymbol{P(s_i)}$	rappresentazione a #p bit di $p(s_i)$ e $P(s_i)$
#a, #c	lunghezza dei registri $\boldsymbol{a_i}$ e $\boldsymbol{c_i}$, in bit
$\boldsymbol{a_i}$, $\boldsymbol{c_i}$	contenuto dei registri *intervallo* e *code-point*
R_i	numero di bit nel prodotto $\boldsymbol{P}(s_i) \cdot \boldsymbol{a}_i$
$\Re$	range legale di $\boldsymbol{a_i}$
X_i	numero di shift necessari per mantenere $\boldsymbol{a_i}$ nel range $\Re$
ζ_i	variabile temporanea
B_i	blocco di bit di codice, di dimensione variabile
T_k	blocco di bit di codice, di dimensione fissa (sequenza di output)

Infine, con la scrittura

$$a_i = 0.0000001\underline{1101}$$

intendiamo che il valore reale di a_i è 0.0000001101, mentre i bits $\boldsymbol{a_i}$ correntemente memorizzati sono quelli sottolineati, 1101. Seguono i passi eseguiti dall'algoritmo per codificare un simbolo.

1. calcolare il valore di $\boldsymbol{p}(s_i) \cdot \boldsymbol{a}_i$ in maniera esatta

2. determinare il numero X_i di shifts tali che $\boldsymbol{p}(s_i)\cdot\boldsymbol{a}_i\cdot 2^{X_i}\in\Re$; questa operazione è detta di *rescaling*
3. trovare l'approssimazione a #a bits dal basso di $\boldsymbol{p}(s_i)\cdot\boldsymbol{a}_i\cdot 2^{X_i}$; questo valore è assegnato ad a_{i+1}
4. calcolare il valore $\zeta_i = \boldsymbol{c}_i + \boldsymbol{P}(s_i)\cdot\boldsymbol{a}_i$ in maniera esatta
5. mettere in output il bit di riporto della precedente operazione e gli X_i bits più significativi di ζ_i; questo è il blocco B_i di dimensione variabile
6. shiftare a sinistra i bits rimanenti di ζ_i di X_i posti; il risultato è assegnato a $\boldsymbol{c}_{i+1}$

Il riquadro seguente mostra un esempio di codifica aritmetica a precisione finita. L'alfabeto e la stringa usati sono quelli in Figura 27.

1. calcolare il valore di $\boldsymbol{p}(s_i)\cdot\boldsymbol{a}_i$ in maniera esatta

$$\begin{aligned}\boldsymbol{a}_2 &= 0.01\underline{1001}\\ \boldsymbol{p}(\text{'E'}) &= 0.\underline{01}\\ \boldsymbol{p}(\text{'E'})\cdot\boldsymbol{a}_2 &= 0.0\underline{00100100}\end{aligned}$$

   Si noti che il prodotto ha precisione #p + #a = 8 bit.

2. determinare il numero X_2 di shifts tali che $\boldsymbol{p}('E')\cdot\boldsymbol{a}_2\cdot 2^{X_2}\in\Re=[0.1,1)$

$$\begin{aligned}X_2 &= 2\\ \boldsymbol{p}('E')\cdot\boldsymbol{a}_2\cdot 2^{X_2} &= 0.\underline{100100}\end{aligned}$$

3. trovare l'approssimazione a #a = 4 bits dal basso di $\boldsymbol{p}('E')\cdot\boldsymbol{a}_2\cdot 2^{X_2}$; questo valore è assegnato ad $\boldsymbol{a}_3$

$$\boldsymbol{a}_3 = 0.000\underline{1001}$$

4. calcolare il valore $\zeta_3 = \boldsymbol{c}_2 + \boldsymbol{P}('E')\cdot\boldsymbol{a}_2$ in maniera esatta.

$$\begin{aligned}\boldsymbol{P}(\text{'E'}) &= 0.00001\underline{1}\\ \boldsymbol{P}(\text{'E'})\cdot\boldsymbol{a}_2 &= 0.0\underline{00000110}\\ \boldsymbol{c}_2 &= 0.01\underline{1011}\\ \zeta_3 &= 0.01\underline{1011011}\end{aligned}$$

   Osserviamo che ζ deve avere precisione #c+1 = 9 bit, per conservare lo spazio per un eventuale riporto.

5. mettere in output il bit di riporto della precedente operazione e gli X_2 bits più significativi di ζ_3; questo è il blocco di dimensione variabile B_2. Il bit di riporto è sottolineato.

$$B_2 = \underline{0}10$$

6. shiftare a sinistra i bits rimanenti di ζ_3 di X_2 posti; il risultato è assegnato a $\boldsymbol{c}_3$

$$\boldsymbol{c}_3 = 0.01011011$$

Riquadro 1: Operazioni svolte dall'algoritmo a precisione finita per codificare il simbolo 'E', dopo che la sequenza <'H'> è già stata codificata.. Abbiamo scelto #p = 4, #a = 4, #c = 8, cioè i registri $\boldsymbol{a}_i$, $\boldsymbol{p}$, e $\boldsymbol{P}$ hanno precisione 4 e $\boldsymbol{c}_i$ ha precisione 8 bit. Il range legale della mantissa di a_i è [0.1,1).

L'essenza dell'idea, in questo algoritmo, è che i registri contengono, di volta in volta, solo la porzione attiva della computazione, cioè la mantissa approssimata di a_i e i bits di c_i che sono allineati con questa mantissa.
In simboli, la sequenza di operazioni descritta può essere espressa mediante le equazioni

$$\langle \boldsymbol{a}_{i+1}, X_i \rangle = f(\boldsymbol{p}(s_i), \boldsymbol{a}_i) \quad \text{(E.23)}$$

$$R_i = \boldsymbol{p}(s_i) \cdot \boldsymbol{a}_i \quad \text{(E.24)}$$

$$\langle \boldsymbol{c}_{i+1}, B_i \rangle = g(\boldsymbol{c}_i, R_i, X_i) \quad \text{(E.25)}$$

dove f e g sono le funzioni che rappresentano i passi dell'algoritmo descritto sopra.
La prima equazione rappresenta la computazione dell'ampiezza del nuovo intervallo, opportunamente approssimato e riscalato, e del numero di shifts necessari per eliminare dalla mantissa i bits non più necessari al calcolo; la seconda equazione aggiorna il numero di bit nella rappresentazione della nuova mantissa, la terza rappresenta l'aggiornamento del *code point* e la formazione del blocco di bits B_i da inserire nel flusso di output dell'algoritmo.
Consideriamo la sequenza di blocchi di lunghezza variabile, B_i, generata dall'algoritmo. Ciascun blocco contiene il bit di riporto e i bits corrispondenti agli shifts dei registri, in totale $1 + X_i$ bits. Poiché i blocchi B_i sono di lunghezza variabile, non è conveniente inserirli direttamente nello stream di output; per questo i blocchi B_i sono assemblati in una sequenza di blocchi di lunghezza fissa, T_k, e l'output dell'algoritmo è bufferizzato, finché non viene raggiunto un numero di bits sufficienti a completare almeno un blocco T_k. La figura seguente mostra come questo assemblamento è fatto, sovrapponendo il bit di riporto di un blocco all'ultimo bit del blocco precedente per formare una somma.

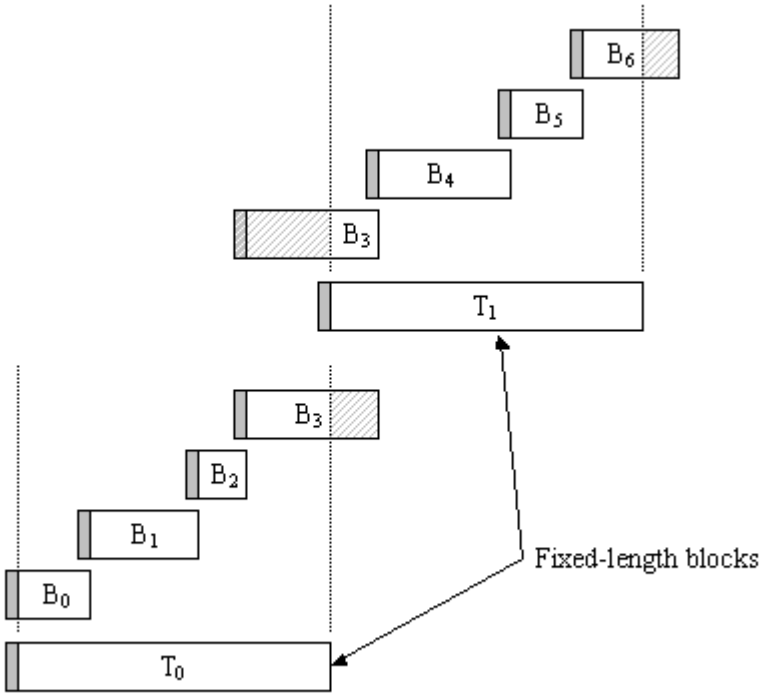

Figura 28: Assemblaggio di blocchi di lunghezza variabile in blocchi di lunghezza fissa.

Come risulta chiaro dalla precedente figura, i blocchi a lunghezza variabile B_i sono semplicemente concatenati ciascuno al precedente. Solo il bit di riporto è sommato al bit

meno significativo del blocco precedente. Man mano che i blocchi T_k sono assemblati, vengono messi in output (on-line). La sequenza di tali blocchi costituisce l'output dell'algoritmo.
I blocchi di lunghezza fissa sono inseriti nel flusso di output appena pronti. Tuttavia non abbiamo considerato il problema della propagazione dei riporti tra questi blocchi. Poiché un blocco eliminato dal buffer non può più essere modificato, un blocco T_k può essere messo in output solo se siamo sicuri che non potrà essere più interessato da riporti generati da blocchi successivi. Un blocco può generare un riporto solo se contiene bit tutti a "1", quindi il blocco T_k può essere eliminato in maniera sicura se il blocco T_{k+1} contiene qualche bit nullo o viene generato un riporto. Esiste anche la possibilità che si concentri nel buffer una sequenza di diversi blocchi che, tutti escluso eventualmente il primo, hanno tutti i bit settati a "1". Anche in questo caso, il primo blocco che contiene qualche bit nullo o che genera un riporto fa sì che tutti i blocchi precedenti possano essere scaricati dal buffer.
Il metodo appena descritto permette di gestire i riporti in maniera corretta, ma non pone alcun limite al massimo numero di blocchi contemporaneamente presenti nel buffer e, di conseguenza, alla dimensione del buffer stesso. Anche se nella pratica è molto raro, il buffer potrà contenere un numero considerevole di blocchi; nel caso peggiore dovrà essere capace di contenere l'intero code-point, che può raggiungere dimensioni proibitive. Inoltre, l'algoritmo per stabilire se ed eventualmente quali blocchi scaricare dal buffer aggiunge complessità all'encoder. Questi inconvenienti sono ostacoli considerevoli all'implementazione in hardware dell'algoritmo. Un metodo alternativo per risolvere il problema dei riporti e, nello stesso tempo, garantire che il buffer abbia dimensioni fissate a priori è quello di inserire nello stream di output il bit di riporto un blocco T_k composto tutto da bit "1". In questo modo, quando il decoder legge una sequenza di bit "1" di lunghezza pari a quella di un blocco T_k, sa che il bit seguente è il riporto del blocco successivo, che non è stato sommato al blocco corrente, e può decodificare in maniera corretta la sequenza di simboli codificati. Evidentemente, un algoritmo che utilizza questo approccio necessita di un buffer capace di contenere solamente un blocco alla volta, più il bit di riporto, quindi la dimensione di tale buffer è limitata e nota a priori. Tuttavia, il vantaggio in termini di risparmio di memoria è bilanciato da un aumento del numero di bit codificati che, comunque, risulta in genere accettabile.
Fin qui abbiamo descritto il funzionamento dell'encoder, adesso discuteremo brevemente del processo di decodifica. Il decoder determina, per la decodifica di ogni decisione binaria (MPS o LPS), a quale sotto-intervallo puntano i dati compressi. Questo è fatto ricorsivamente, usando la stessa suddivisione degli intervalli dell'encoder. Ad ogni decisione decodificata, il decoder sottrae dalla stringa codificata l'intervallo relativo; la sequenza codificata è dunque, dal punto di vista del decoder, un puntatore all'interno dell'intervallo corrente, relativo alla base dell'intervallo stesso. Al procedere della decodifica, il decoder sottrae dal code-stream tutti gli intervalli che erano stati aggiunti dall'encoder, 'consumando' tutti i bit della stringa codificata. Come detto sopra, purché utilizzi la stessa procedura di approssimazione usata nel processo di codifica, il decoder è in grado di ricostruire in maniera esatta la sequenza di decisioni codificate.
L'algoritmo di codifica aritmetica descritto fin qui lavora con simboli, le cui stime di probabilità sono note a priori e non variano durante l'esecuzione. Questo è, in linea generale, quello che accade in codificatori entropici come quello di Huffman, in cui una prima scansione raccoglie informazioni statistiche sull'occorrenza dei simboli nella sequenza di input e, successivamente, un'altra scansione codifica i simboli in base alle valutazioni di probabilità fissate durante la prima scansione. Il problema dei codificatori di questo tipo è la necessità di trasmettere, insieme ai dati compressi, il modello statistico usato nella compressione, che rappresenta un overhead non indifferente. Inoltre,

l'esperienza dimostra che spesso una codifica adattativa permette di ottenere performance migliori rispetto ad una statica, che non è in grado di adattarsi all'input. Infine, se vogliamo un algoritmo realmente on-line, non possiamo permetterci di eseguire più di una scansione di tutto l'input. L'idea allora è quella di associare, inizialmente, a tutti i simboli la stessa stima di probabilità, pari a $1/N$, se N è la cardinalità dell'alfabeto cui tali simboli appartengono; questo corrisponde ad una distribuzione uniforme che riflette il fatto che non c'è alcuna preferenza tra i simboli, dato che niente è stato ancora codificato. Man mano che nuovi simboli arrivano, le valutazioni di probabilità associate a ciascun elemento dell'alfabeto sono modificate in relazione all'identità del nuovo simbolo. Il processo che guida la variazione nelle stime di probabilità dei simboli può essere implementato usando un automa a stati finiti, dove ogni stato contiene una stima di probabilità per ogni simbolo dell'alfabeto e la funzione di transizione associa a coppie stato-simbolo un nuovo stato. È importante osservare, a questo punto, che tale macchina a stati finiti deve essere costruita ad-hoc per ogni classe di simboli che si intende codificare; ciò significa che le valutazioni fornite da una macchina possono essere buone, nel senso che producono una codifica più efficiente, per una classe di segnali e cattive per un'altra.
Nel caso di codifica adattativa, perché il processo di decodifica sia eseguito correttamente, basta che il decoder segua lo stesso processo di stima usato dal decoder. In tal modo, la suddivisione degli intervalli è uguale a quella del processo di codifica e, quindi, la stringa di simboli codificata potrà essere ricostruita fedelmente.

4.3. MQ-Coder

MQ-Coder è un codificatore aritmetico adattativo [12] che codifica decisioni binarie, utilizzando una macchina a stati finiti il cui stato corrente è determinato in base al contesto associato al simbolo da codificare. Il contesto e il bit da codificare sono inviati dalle primitive di codifica di EBCOT, discusse nel capitolo precedente. Ad ogni passo, gli intervalli non sono assegnati al simbolo da codificare (0 o 1), come abbiamo visto nella trattazione generale, ma al *simbolo più probabile* (*Most Probable Symbol* o **MPS**) e al *simbolo meno probabile* (*Least Probable Symbol* o **LPS**); l'identità del simbolo più probabile è valutata in base allo stato corrente, che dipende dal contesto associato al bit attuale. I simboli sono ordinati in modo tale che il sotto-intervallo associato al MPS è posto sopra quello associato al LPS, come mostrato nella seguente figura. *Qe* indica la stima della probabilità del LPS.

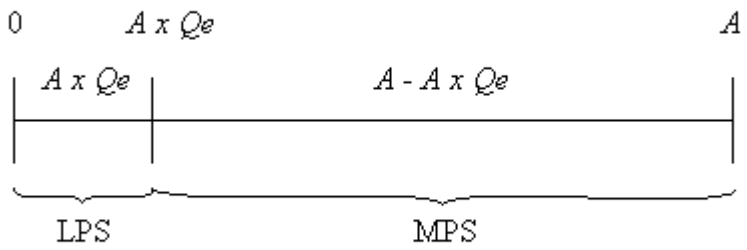

Figura 29: Ordinamento dei simboli e suddivisione ideale degli intervalli.

Le operazioni di codifica sono eseguite usando l'aritmetica a precisione finita e una rappresentazione intera dei valori reali, in cui i valori esadecimali 0x8000 e 0x10000 sono equivalenti, rispettivamente, ai decimali 0.75 e 1.5. L'ampiezza A dell'intervallo corrente è mantenuta nel range $\Re = [0.75, 1.5)$ shiftandola a sinistra quando il suo valore scende al di sotto di 0x8000 (**rinormalizzazione**); ogni qual volta il registro A è shiftato di X_i bits, anche il registro C è shiftato a sinistra di X_i bits bit per mantenere la sua identità di puntatore al sotto-intervallo codificato. Per evitare l'overflowing di C, periodicamente un byte è rimosso dai bit più significativi di C; questo equivale a porre la dimensione del blocco di lunghezza fissa, T_k, pari a 8. La propagazione del riporto è risolta mediante una procedura di riempimento che sarà descritta nel seguito.
La codifica dei simboli avviene come segue. Detta Qe la stima corrente[6] della probabilità del LPS, il calcolo esatto del successivo sotto-intervallo è descritto dalle formule:

Codifica del MPS:

$$C = C + A \times Qe \quad \text{(E.26)}$$
$$A = A \times (1\text{-}Qe) = A - A \times Qe \quad \text{(E.27)}$$

Codifica del LPS:

$$C: \textit{ nessuna modifica}$$
$$A = Qe \times A \quad \text{(E.28)}$$

Tuttavia, mantenere A nel range $\Re$ (cioè $0.75 \le A \le 1.5$) permette una semplice approssimazione aritmetica nella suddivisione dei sotto-intervalli. Precisamente, essendo A nell'ordine dell'unità, possiamo riscrivere le formule precedenti nel seguente modo:

Codifica del MPS:

$$C = C + Qe \quad \text{(E.29)}$$
$$A = A - Qe \quad \text{(E.30)}$$

Codifica del LPS:

$$C: \textit{ nessuna modifica}$$
$$A = Qe \quad \text{(E.31)}$$

Questa approssimazione non compromette la capacità, da parte del decoder, di ricostruire esattamente la sequenza originale a partire dai dati compressi e, inoltre, non spreca spazio di codifica perché la somma delle probabilità di LPS e MPS è ancora A. Il vantaggio delle formule approssimate, rispetto a quelle esatte, è che non contengono moltiplicazioni, che sono operazioni potenzialmente costose sia in implementazioni hardware che software. Un problema che si presenta con l'approssimazione della moltiplicazione è che, quando Qe è nell'ordine di 0.5 e A è piccolo, la dimensione del sottointervallo associato al MPS può essere inferiore a quella assegnata al LPS; per esempio, se A vale 0.75 e Qe è 0.5,

[6] La probabilità associata ai simboli è stimata in maniera adattativa rispetto alla sequenza di simboli codificati, quindi Qe varia in relazione alla sequenza codificata.

l'ampiezza del sotto-intervallo assegnato al MPS sarà nell'ordine di 0.25. Per evitare questa inversione nell'ampiezza degli intervalli, l'assegnamento di LPS e MPS è invertito quando il sotto-intervallo allocato al LPS diventa maggiore di quello del MPS. Questa operazione è nota con il nome di **scambio condizionale** (**conditional exchange**) e il termine condizionale si riferisce al fatto che il riassegnamento è fatto solo quando la probabilità del LPS occupa più della metà dell'ampiezza dell'intervallo corrente. La figura seguente mostra un esempio di due passi di codifica, uno senza e l'altro con scambio condizionale.

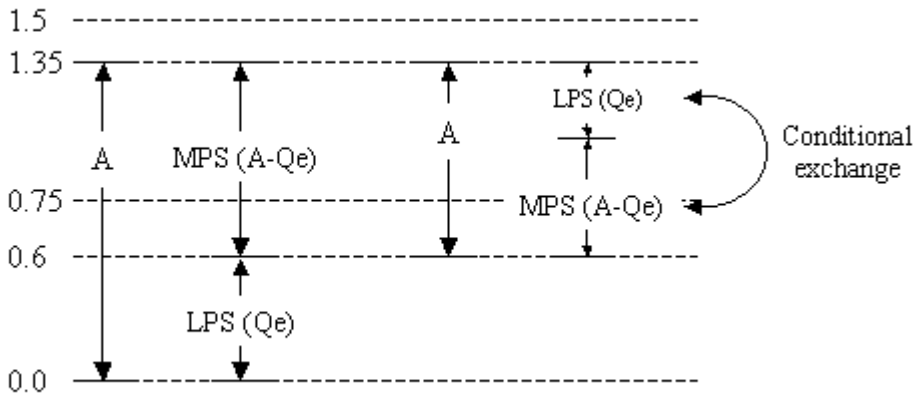

Figura 30: Esempio di suddivisione degli intervalli, rispettivamente senza e con scambio condizionale.

Si noti che, quando avviene uno scambio condizionale, vale la relazione $0.5 \geq Qe > (A\text{-}Qe)$. Chiaramente, entrambi gli intervalli sono inferiori a 0.75 e quindi è necessaria un'operazione di rinormalizzazione. Di conseguenza, il test per lo scambio condizionale può essere eseguito dopo che il test per la rinormalizzazione ha dato esito positivo. Gli algoritmi di codifica sono riportati di seguito.

Codifica del MPS:

```
A = A – Qe                      /* Calculate MPS subinterval       */
if  A < 0x8000                  /*  if renormalization is needed   */
    if  A < Qe                  /*  if interval size is inverted   */
        A = Qe                  /* Set interval to LPS subinterval */
    else                        /* otherwise                       */
        C = C + Qe              /* Point to MPS interval base      */
    end
    renormalize  A and C        /* renormalize                     */
else                            /* if no renormalization is needed */
    C = C + Qe                  /* Point to MPS interval base      */
end
```

Algoritmo A.2: Codifica del simbolo più probabile

Codifica del LPS:

```
A = A – Qe                      /* Calculate MPS subinterval      */
if  A < Qe                      /*  if interval size is inverted  */
      C = C + Qe                /* Point to MPS interval base     */
else                            /* otherwise                      */
      A = Qe                    /* Set interval to LPS subinterval */
end
renormalize  A and C            /* renormalization always needed */
```

Algoritmo A.3: Codifica del simbolo meno probabile.

Come anticipato nella discussione generale, un aspetto importante della codifica aritmetica è la probabilità associata ai simboli dell'alfabeto Σ. Nel caso del MQ-Coder, le probabilità sono associate a MPS e LPS anziché ai simboli dell'alfabeto (0 e 1) e le stime cambiano al procedere dell'operazione di codifica, quindi la stima delle probabilità deve essere fatta con cura al fine di ottenere un buon fattore di compressione. Il processo di stima della probabilità dei simboli è basato su una forma di conteggio approssimato in cui la rinormalizzazione del registro intervallo, *A,* è usata per stimare il numero di MPS e LPS. Quando avviene una rinormalizzazione, il conteggio dei MPS è azzerato e una nuova stima è ottenuta da una tabella che fornisce un valore di *Qe* più grande quando avviene la rinormalizzazione del simbolo LPS, più piccolo quando si ha una rinormalizzazione MPS. In questo modo, nonostante il sistema sia stocastico, la macchina a stati che guida il processo di stima della probabilità *Qe* naturalmente tende a muoversi verso la stima corretta. Se il valore di *Qe* è troppo grande, la rinormalizzazione MPS è più probabile di quella LPS e, di conseguenza, il valore di *Qe* tende a diminuire; viceversa, se *Qe* è troppo piccolo, la rinormalizzazione MPS è meno probabile di quella LPS e quindi il valore di *Qe* tende a crescere. Se l'identità del MPS è sbagliata (cioè il MPS è, in realtà, il meno probabile), *Qe* sarà incrementato fino a raggiungere approssimativamente il valore 0.5 e, a questo punto, i simboli meno probabile e più probabile sono scambiati. La Tabella 13 mostra l'automa a stati finiti che guida il processo di stima della probabilità *Qe*. Per ogni valore dell'indice *Index* nella tabella ci sono quattro colonne. *Qe_value* è il valore della probabilità del LPS associata allo stato corrente. *NMPS* è l'indice della nuova stima di probabilità dopo una rinormalizzazione dovuta alla codifica del MPS e *NLPS* è il nuovo indice, dopo una rinormalizzazione LPS. *Switch* indica se deve essere eseguito uno scambio condizionale quando si codifica il LPS; se *Switch* è diverso da zero ed è necessaria una rinormalizzazione LPS, il valore del MPS deve essere cambiato prima di passare allo stato indicato da *NLPS*.
Si noti che solamente i contesto e il bit da codificare sono passati al MQ-Coder dall'esterno. L'indice associato a ciascun contesto e tutte le informazioni ad esso relative sono mantenuti in variabili interne.

Index	Qe_Value			NMPS	NLPS	SWITCH
	(hexadecimal)	(binary)	(decimal)			
0	0x5601	0101 0110 0000 0001	0,503 937	1	1	1
1	0x3401	0011 0100 0000 0001	0,304 715	2	6	0
2	0x1801	0001 1000 0000 0001	0,140 650	3	9	0
3	0x0AC1	0000 1010 1100 0001	0,063 012	4	12	0
4	0x0521	0000 0101 0010 0001	0,030 053	5	29	0
5	0x0221	0000 0010 0010 0001	0,012 474	38	33	0
6	0x5601	0101 0110 0000 0001	0,503 937	7	6	1
7	0x5401	0101 0100 0000 0001	0,492 218	8	14	0
8	0x4801	0100 1000 0000 0001	0,421 904	9	14	0
9	0x3801	0011 1000 0000 0001	0,328 153	10	14	0
10	0x3001	0011 0000 0000 0001	0,281 277	11	17	0
11	0x2401	0010 0100 0000 0001	0,210 964	12	18	0
12	0x1C01	0001 1100 0000 0001	0,164 088	13	20	0
13	0x1601	0001 0110 0000 0001	0,128 931	29	21	0
14	0x5601	0101 0110 0000 0001	0,503 937	15	14	1
15	0x5401	0101 0100 0000 0001	0,492 218	16	14	0
16	0x5101	0101 0001 0000 0001	0,474 640	17	15	0
17	0x4801	0100 1000 0000 0001	0,421 904	18	16	0
18	0x3801	0011 1000 0000 0001	0,328 153	19	17	0
19	0x3401	0011 0100 0000 0001	0,304 715	20	18	0
20	0x3001	0011 0000 0000 0001	0,281 277	21	19	0
21	0x2801	0010 1000 0000 0001	0,234 401	22	19	0
22	0x2401	0010 0100 0000 0001	0,210 964	23	20	0
23	0x2201	0010 0010 0000 0001	0,199 245	24	21	0
24	0x1C01	0001 1100 0000 0001	0,164 088	25	22	0
25	0x1801	0001 1000 0000 0001	0,140 650	26	23	0
26	0x1601	0001 0110 0000 0001	0,128 931	27	24	0
27	0x1401	0001 0100 0000 0001	0,117 212	28	25	0
28	0x1201	0001 0010 0000 0001	0,105 493	29	26	0
29	0x1101	0001 0001 0000 0001	0,099 634	30	27	0
30	0x0AC1	0000 1010 1100 0001	0,063 012	31	28	0
31	0x09C1	0000 1001 1100 0001	0,057 153	32	29	0
32	0x08A1	0000 1000 1010 0001	0,050 561	33	30	0
33	0x0521	0000 0101 0010 0001	0,030 053	34	31	0
34	0x0441	0000 0100 0100 0001	0,024 926	35	32	0
35	0x02A1	0000 0010 1010 0001	0,015 404	36	33	0
36	0x0221	0000 0010 0010 0001	0,012 474	37	34	0
37	0x0141	0000 0001 0100 0001	0,007 347	38	35	0
38	0x0111	0000 0001 0001 0001	0,006 249	39	36	0
39	0x0085	0000 0000 1000 0101	0,003 044	40	37	0
40	0x0049	0000 0000 0100 1001	0,001 671	41	38	0
41	0x0025	0000 0000 0010 0101	0,000 847	42	39	0
42	0x0015	0000 0000 0001 0101	0,000 481	43	40	0
43	0x0009	0000 0000 0000 1001	0,000 206	44	41	0
44	0x0005	0000 0000 0000 0101	0,000 114	45	42	0
45	0x0001	0000 0000 0000 0001	0,000 023	45	43	0
46	0x5601	0101 0110 0000 0001	0,503 937	46	46	0

Tabella 13: Automa finito che guida il processo di stima della probabilità Qe.

Dato un contesto *CX*, se *Index()*, *Qe()* e *MPS()* sono, rispettivamente, indice, stima di probabilità e MPS correnti, i passi necessari per raggiungere una nuova stima di probabilità dopo una rinormalizzazione sono:

```
rinormalizzazione MPS:

    I = Index(CX)                   /* Current index for context CX      */
    I = NMPS(I)                     /* New index for context CX          */
    Index(CX) = I                   /* Save this index at context CX     */
    Qe(CX) = Qe_value(I)            /* New probability estimate for CX   */
```

Algoritmo A.4: Rinormalizzazione in occorrenza del simbolo più probabile

```
rinormalizzazione LPS:

    I = Index(CX)                   /* Current index for context CX      */
    if Switch(I) = 1                /* If a conditional exchange occurs  */
       MPS(CX) = 1 – MPS(CX)        /* Exchange MPS sense                */
    end                             /* (1→0  or  0→1)                    */
    I = NLPS(I)                     /* New index for context CX          */
    Index(CX) = I                   /* Save this index at context CX     */
    Qe(CX) = Qe_value(I)            /* New probability estimate for CX   */
```

Algoritmo A.5: Rinormalizzazione in occorrenza del simbolo meno probabile

Riportiamo di seguito i diagrammi di flusso che descrivono il funzionamento del codificatore aritmetico di JPEG2000. Ulteriori informazioni sull'argomento possono essere trovate in [2].
I simboli usati nei diagrammi hanno il seguente significato:

A	intervallo di probabilità (16 bit)
B	byte corrente di dati codificati (8 bit)
B1	byte di dati codificati successivo a quello corrente (B)
BP	puntatore al byte contenuto in B
BPST	valore iniziale di BP
C	valore del bit-stream nel registro di codice (32 bit)
Chigh	16 bit più significativi di C
Clow	16 bit meno significativi di C
CT	contatore di shift dovuti alla rinormalizzazione
D	decisione binaria da codificare/decodificare
TEMPC	registro temporaneo di MQ-Coder

Nella notazione precedente, il simbolo B non deve essere confuso con il simbolo B_i usato nella descrizione dell'algoritmo generale. Quest'ultimo, infatti, rappresenta una sequenza di bit variabile, mentre il primo è una quantità di lunghezza fissa, che viene rimossa dal registro C, non appena sono stati codificati otto bit, e messa in output. B, quindi, è il blocco di lunghezza fissa T_k, inserito in un buffer per tenere in conto un eventuale riporto.
CT, infine, è il simbolo usato nello standard JPEG2000 per indicare il numero di shift X_i.

4.3.1. Encoder

La struttura dei registri A e C dell'encoder è quella mostrata nella tabella che segue:

	MSB			LSB
C	0000 cbbb	bbbb bsss	xxxx xxxx	xxxx xxxx
A	0000 0000	0000 0000	aaaa aaaa	aaaa aaaa

Tabella 14 Struttura dei registri A e C dell'encoder.

I bit "a" sono la parte frazionaria del registro A, ovvero i bit della mantissa ma_i dell'ampiezza a_i dell'intervallo corrente, memorizzati nel registro A. I bit "x" sono la parte frazionaria del registro C; è questa la porzione di C direttamente interessata dalla somma con i bit del registro A. I bit "s" hanno il ruolo di spazi e forniscono vincoli utili sul riporto. I bit "b" sono le posizioni da cui sono rimossi i byte di dati codificati, non appena completi. Infine, il bit "c" è il riporto. Le descrizioni dettagliate della procedura di *bit-stuffing* (*riempimento*) e della gestione del riporto saranno date nel seguito, in questa sezione.
L'encoder è rappresentato nel seguente diagramma. Il blocco INITENC si occupa di inizializzare le variabili dell'encoder. Ad ogni passo l'ENCODER legge una coppia (CX,D) e la passa al blocco ENCODE che si occupa di codificare il nuovo simbolo. Quando tutti i simboli sono stati codificati (blocco Finished?), il blocco FLUSH mette in output i bit rimanenti.

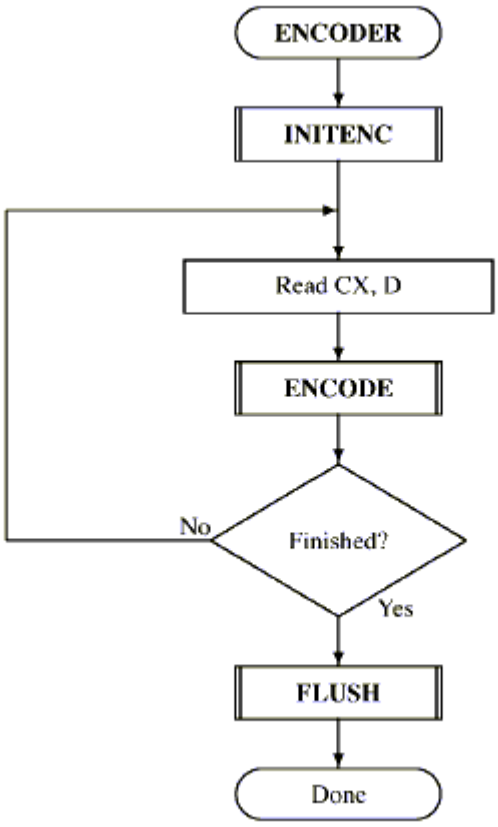

Figura 31: Diagramma di flusso dell'encoder.

La procedura INITENC è usata per inizializzare le variabili dell'encoder. Il diagramma seguente mostra i passi fondamentali di tale procedura.

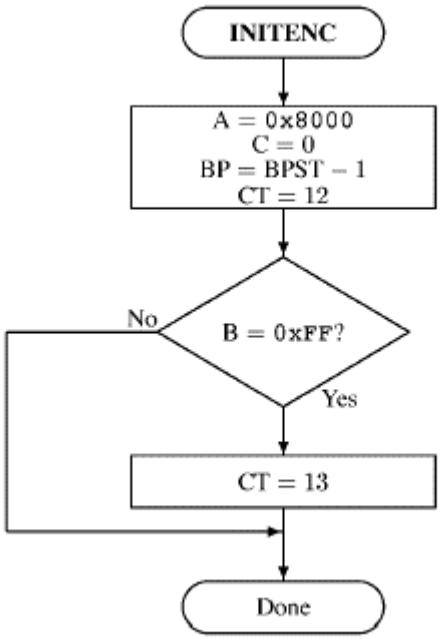

Figura 32: Inizializzazione dell'encoder.

I registri C ed A, rispettivamente base e ampiezza dell'intervallo, sono settati al loro valore iniziale. L'assegnamento CT=12 è dovuto al fatto che, all'inizio della computazione, i 12 bit significativi del registro C sono tutti inutilizzati. BP punta al byte precedente la posizione di BPSP, in cui si trova il primo byte. Quindi, se il byte precedente è 0xFF, è eseguita erroneamente la procedura di riempimento; ciò può essere compensato incrementando il valore di CT. I valori iniziali di *MPS* e *Index* sono indicati nella Tabella 9.
La procedura di codifica (ENCODE) esamina il bit D in input e chiama le procedure CODE0 o CODE1, a seconda che D sia "0" oppure "1". Queste ultime verificano se il simbolo D è o meno il MPS per il contesto CX e invocano la procedura di codifica appropriata.

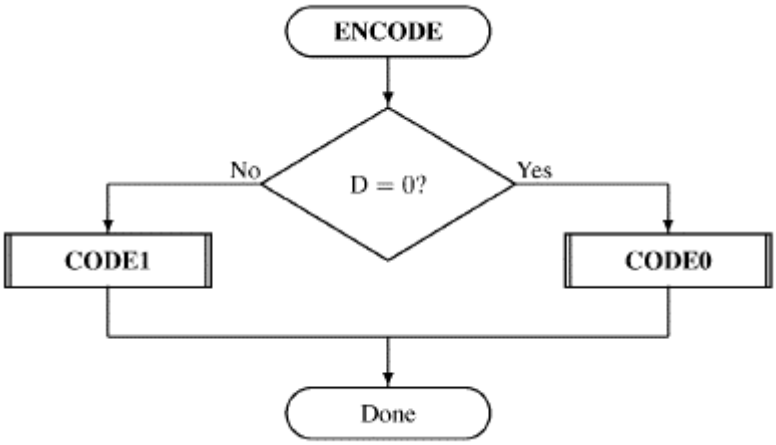

Figura 33: Procedura ENCODE.

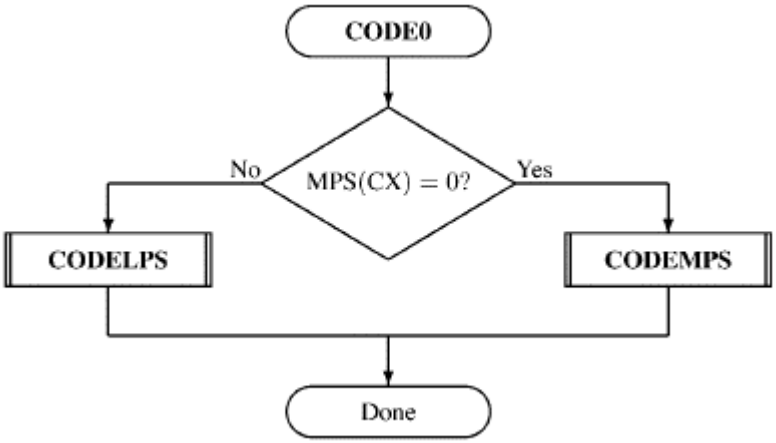

Figura 34: Procedura CODE0

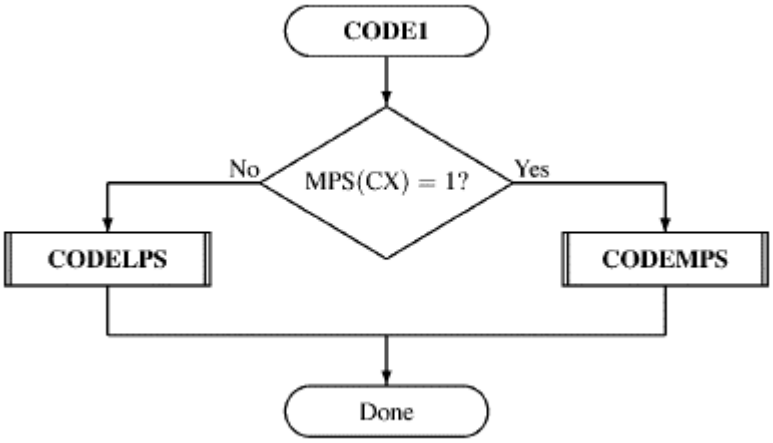

Figura 35: Procedura CODE1

La procedura CODEMPS, il cui diagramma di flusso è rappresentato nella seguente figura, è l'implementazione degli algoritmi A.2 e A.4. Dopo l'esecuzione di CODEMPS, solitamente l'ampiezza dell'intervallo A è ridotta a quella del sotto-intervallo associato al MPS e il registro C punta alla base di tale sotto-intervallo. Tuttavia, se la dimensione dei sotto-intervalli è invertita (il sotto-intervallo associato LPS è più ampio di quello associato al MPS), è codificato il sotto-intervallo associato al LPS. Come detto a proposito dell'algoritmo A.2, l'inversione della dimensione dei sotto-intervalli può avvenire solo in caso di rinormalizzazione. Questo chiarisce perché quest'evento è verificato solo dopo che il test sulla rinormalizzazione ha dato esito positivo. Infine, la stima di probabilità è aggiornata modificando l'indice I(CX), in accordo al corrispondente valore di NMPS nella Tabella 13.

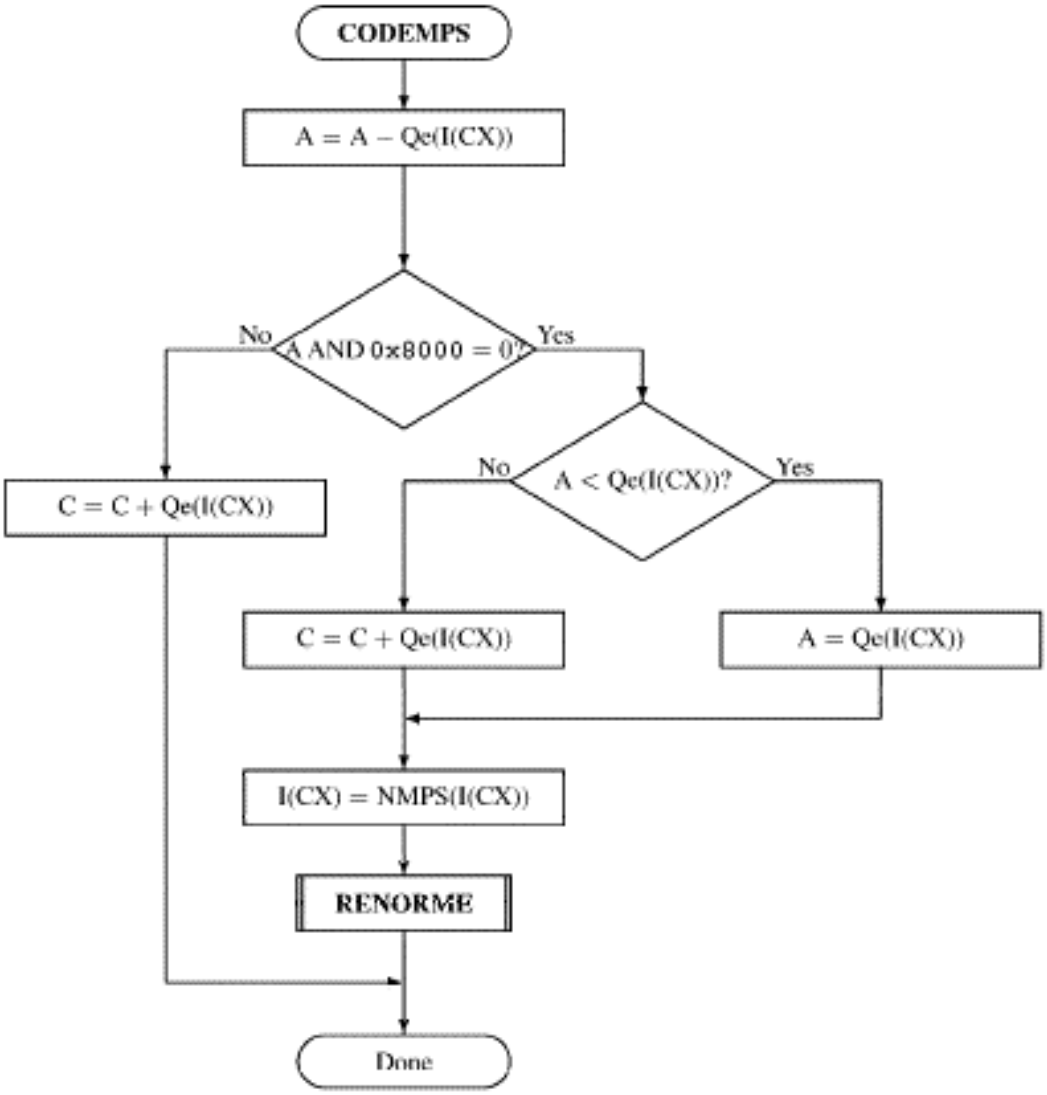

Figura 36: Procedura CODEMPS con lo scambio condizionale MPS/LPS

Il seguente diagramma rappresenta la procedura CODELPS, che si occupa della codifica del simbolo meno probabile. Se non c'è inversione nelle ampiezze degli intervalli, è codificato il sotto-intervallo inferiore (associato al LPS), altrimenti è codificato quello superiore (associato al MPS). La procedura è duale a quella precedente, ma ci sono due differenze. La prima è che la procedura di rinormalizzazione qui è eseguita in ogni caso. La seconda è che viene verificato il valore del campo *Switch* nella Tabella 13. Se *Switch*(I(CX)) = 1, l'identità del simbolo più probabile è scambiata.

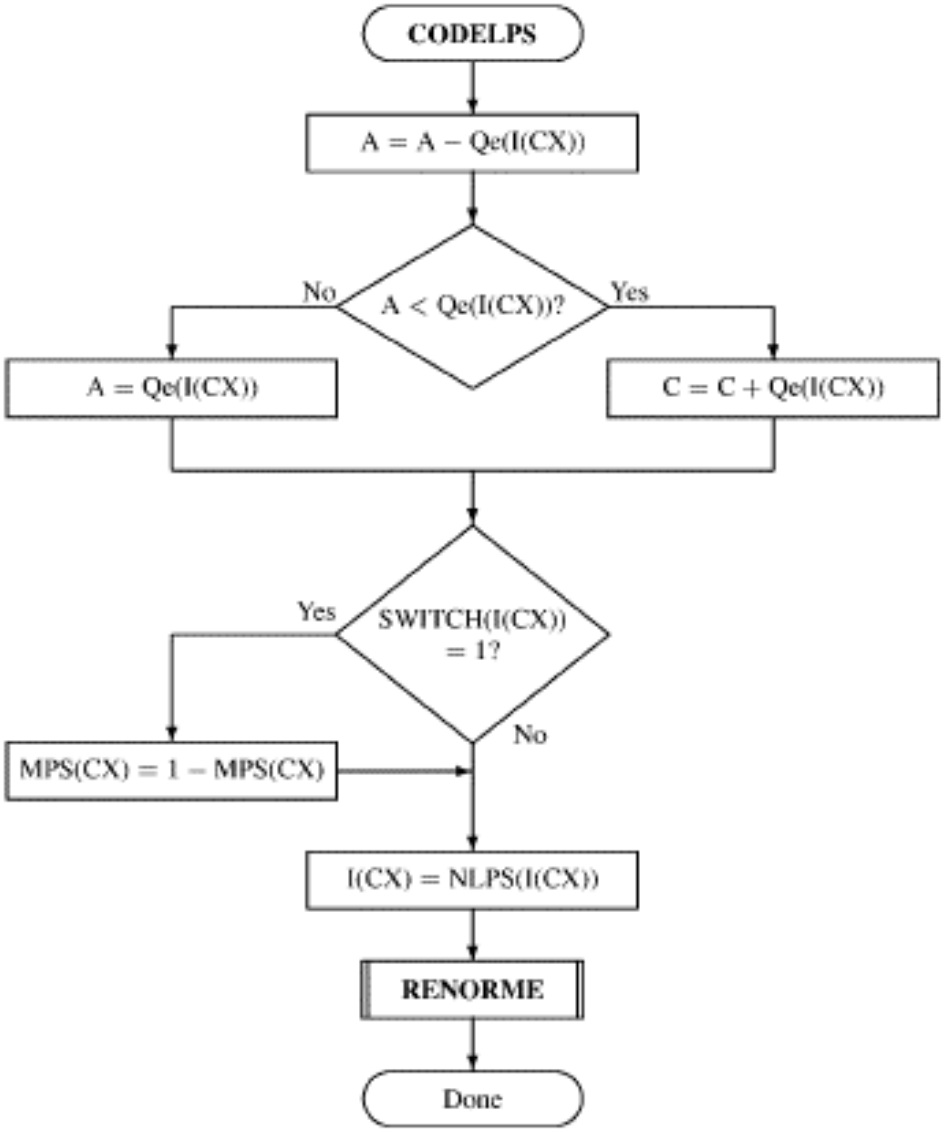

Figura 37: Procedura CODELPS con lo scambio condizionale MPS/LPS

Di seguito è riportato il diagramma che rappresenta la procedura di rinormalizzazione invocata dalle procedure di codifica descritte sopra.

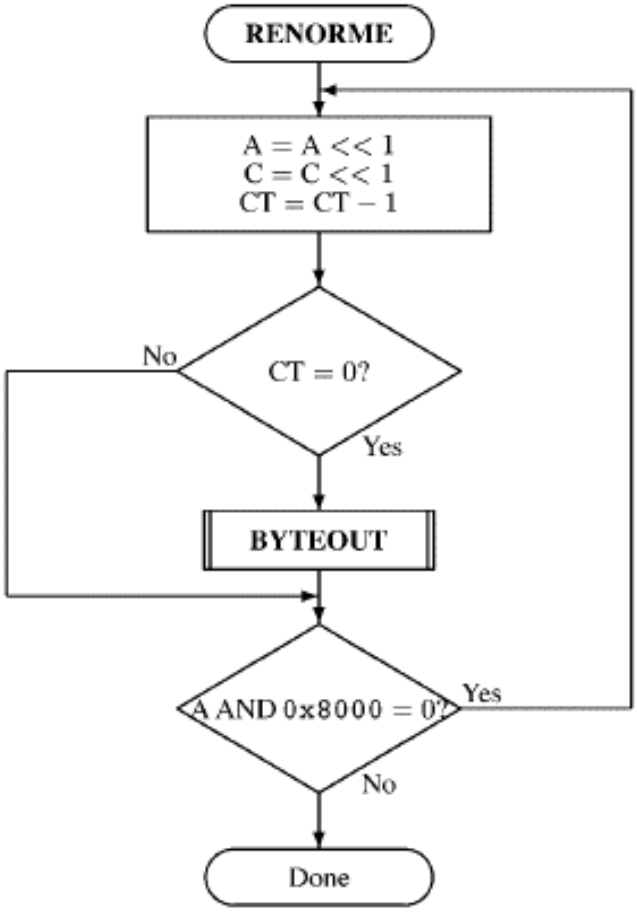

Figura 38: Procedura di rinormalizzazione dell'encoder.

Entrambi i registri A e C sono shiftati a sinistra di un bit alla volta, finché A non è nuovamente compreso nel range legale [0.75,1.5]. Il numero di shift è contato nel registro CT e, quando CT si annulla, un byte di dati compressi è rimosso da C dalla procedura BYTEOUT, riportata di seguito.

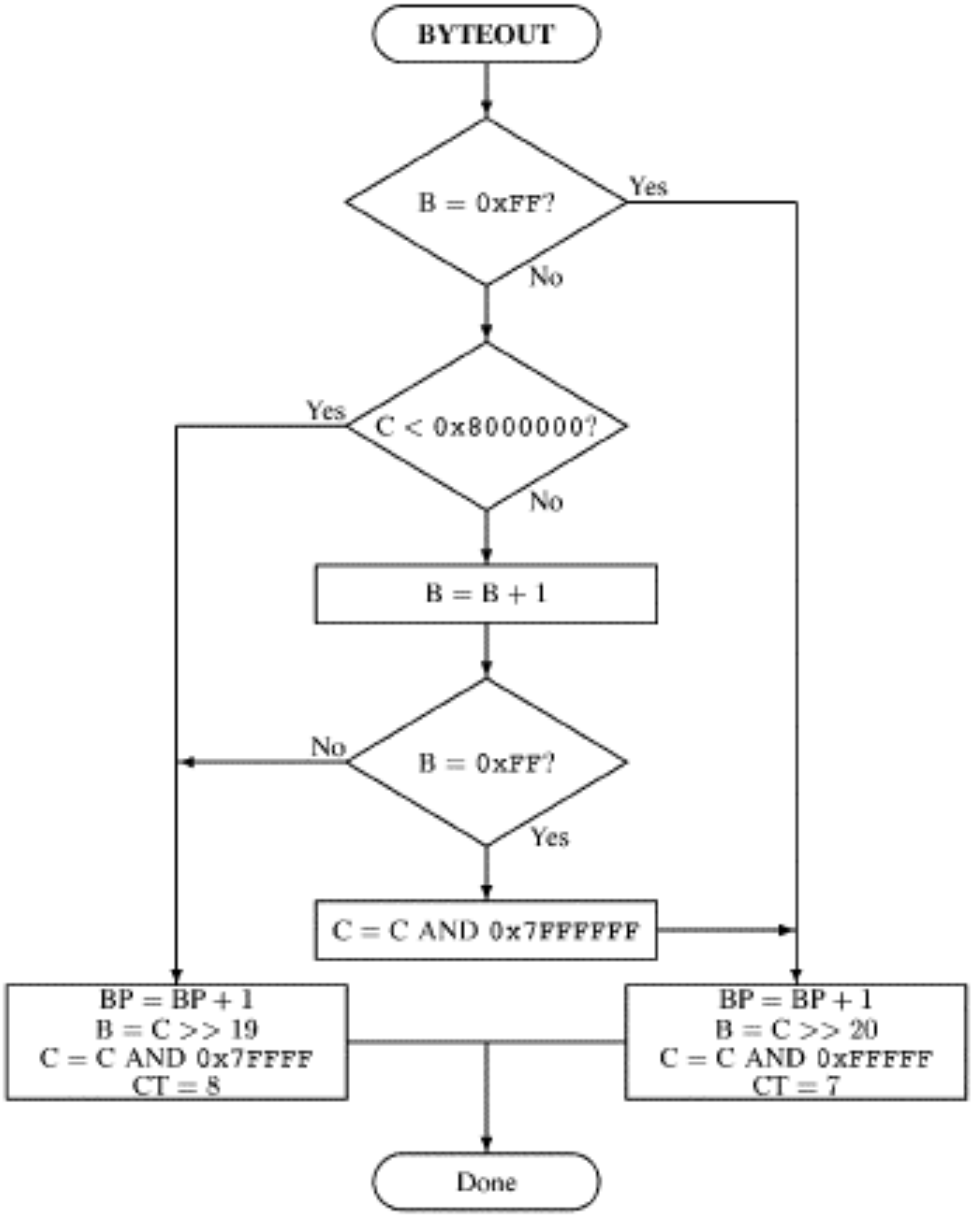

Figura 39: Procedura BYTEOUT (encoder).

La routine BYTEOUT si occupa di rimuovere un byte di dati compressi dal registro C e di metterlo in output. Contiene la procedura di riempimento, cui si accennava sopra, studiata per limitare la propagazione del riporto ai byte di dati compressi completati.
La procedura di bit-stuffing consiste nell'inserire il bit di riporto dopo un byte 0xFF, anziché sommarlo a tale byte. Non è difficile verificare che questa convenzione impedisce che un eventuale riporto si propaghi oltre al byte più recentemente scritto nello stream dei dati compressi. Ciò limita a un solo byte la dimensione del buffer necessario a contenere i blocchi in output che non sono stati ancora inseriti nello stream compresso. La procedura che si occupa del bit-stuffing è quella nel riquadro in basso a destra del diagramma.
Il primo test della routine sul contenuto di B è volto a stabilire se è necessario eseguire il bit-stuffing. Se B non è 0xFF, si verifica la presenza del riporto. Se il bit di riporto è settato, il riporto è sommato a B e un nuovo test verifica la necessità di eseguire il bit-

stuffing. Dopo aver determinato se è necessario un riempimento, siamo in grado di scegliere il percorso appropriato.
Quando non è necessario eseguire il bit-stuffing, i bit "b" di C sono copiati su B (shift a destra di C di 19 posti) e il contatore CT è posto uguale a 8, il numero di bit in output. Quando eseguiamo il bit-stuffing, su B è copiato il bit di riporto seguito dai 7 bit "b" più significativi di C (shift a destra di C di 20 posti). Il contatore CT stavolta è posto uguale a 7 perché il numero dei bit codificati non è 8, dato che il primo bit in output è il riporto. Si noti, inoltre, che in questo caso in C si conserva un bit in più (il bit "b" meno significativo) perché non è stato messo in output.
Per comprendere più chiaramente quanto detto, è utile riferirsi alla Tabella 14.

Infine, l'ultima procedura utilizzata dall'encoder, FLUSH, si occupa di terminare le operazioni di codifica e di generare il marcatore che indica la fine della codeword. La procedura fa in modo che il prefisso 0xFF del marcatore si sovrapponga agli ultimi bit dei dati compressi. Questo garantisce che ogni marcatore presente alla fine dei dati compressi sia riconosciuto e interpretato prima che la decodifica sia completa.

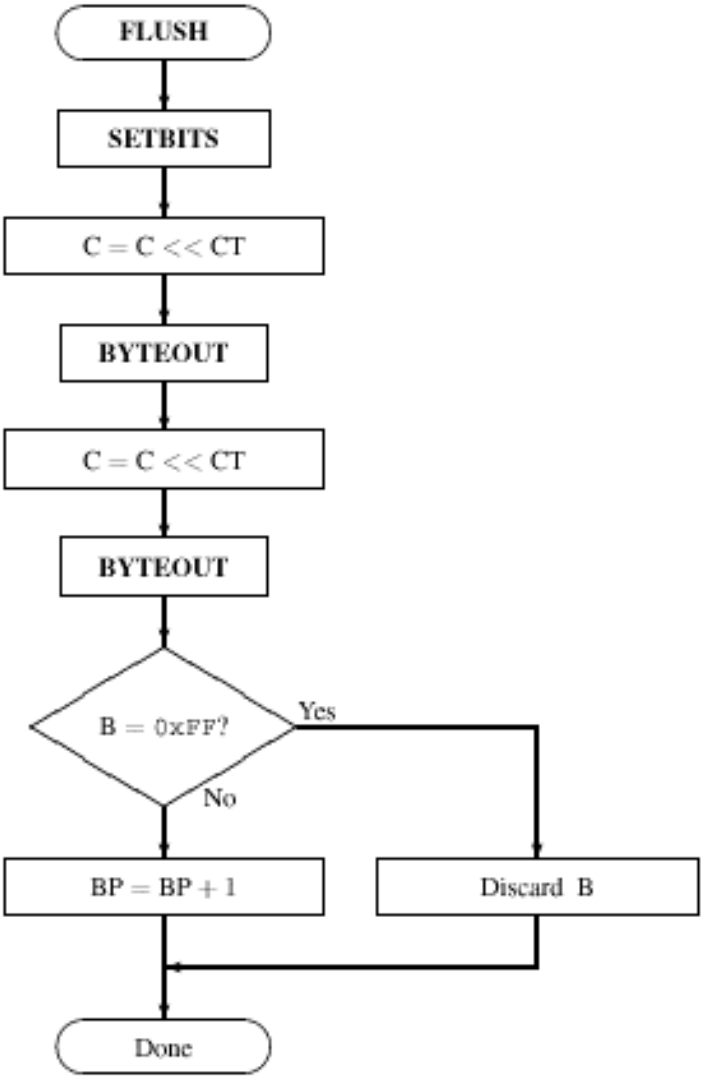

Figura 40: Procedura FLUSH

La prima parte della procedura è il blocco SETBITS, rappresentato nel diagramma seguente. Questa procedura ha il compito di settare a 1 il maggior numero di bit possibili nel registro C. Ciò è fatto nel seguente modo: i 16 bit di ordine più basso di C sono settati a 1, poi si impone che il nuovo valore di C non superi C+A, che è l'estremo destro dell'intervallo codificato. Questa operazione è lecita perché la sequenza di simboli codificati può essere rappresentata da un qualunque punto interno all'intervallo corrente. Se C supera l'estremo superiore dell'intervallo corrente, il bit più significativo è posto a zero per ridurre C a un valore interno all'intervallo. Se questo controllo non fosse eseguito, C potrebbe assumere un valore esterno all'intervallo di codifica e ciò violerebbe la correttezza della codifica aritmetica.

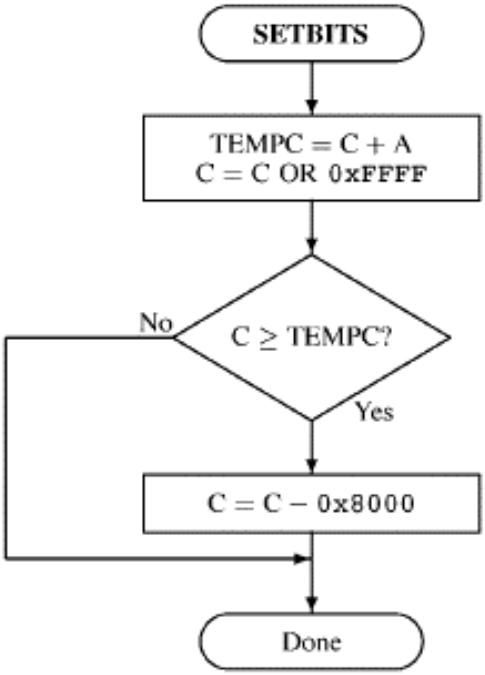

Figura 41: Procedura SETBITS.

Il byte nel registro C è completato shiftando C a sinistra di CT bits e due bytes sono rimossi da C. Questa operazione recupera gli ultimi bit rimasti e completa la codifica della codeword. Infine, se il byte nel buffer B è 0xFF è scartato, altrimenti è messo in output.
La procedura SETBITS acquista significato alla luce di quest'ultima operazione. Infatti, lo scopo di tutte queste operazioni è fare in modo che l'ultimo byte contenga tutti bit settati a 1, per poi poterlo eliminare sovrapponendo ad esso il primo byte del marcatore di fine sequenza. In tal modo la codeword risulta più corta di un byte: un risparmio non trascurabile quando il codificatore aritmetico è terminato ad ogni passo.
Un'ultima osservazione riguarda il quarto passo nella procedura FLUSH. Il registro C è shiftato di CT bit a sinistra prima di mettere in output l'ultimo byte. Questo è dovuto alla possibilità di un riporto nel byte appena estratto. All'uscita della procedura BYTEOUT, infatti, CT può assume valore 8 se non c'è stato riporto, 7 se è stata eseguita la procedura di bit-stuffing.

4.3.2. Decoder

Descriveremo adesso il funzionamento del decoder aritmetico. Il decoder riceve in input un contesto, determinato dall'unità che si occupa del context modeling, e i dati compressi. L'output è una decisione binaria D.
La tabella seguente mostra la struttura dei registri A e C del decoder.

	MSB	LSB
Chigh	xxxx xxxx	xxxx xxxx
Clow	bbbb bbbb	0000 0000
A	aaaa aaaa	aaaa aaaa

Tabella 15: Struttura dei registri A e C del decoder.

I bit "a", "b" e "x" hanno lo stesso significato che nell'encoder. I registri Chigh e Clow possono essere pensati come un unico registro a 32 bit poiché la rinormalizzazione di C shifta un bit di nuovi dati dal MSB di Clow al LSB di Chigh. Tuttavia, i confronti necessari per la decodifica utilizzano solo il registro Chigh. I nuovi dati compressi sono inseriti solo nei bit "b" del registro Clow, uno alla volta.
Il diagramma di flusso del decoder è mostrato nella figura seguente. Il blocco INITDEC inizializza le variabili interne del decoder. I contesti CX e i dati compressi sono letti e passati al blocco DECODE, fino a quando tutti i contesti sono stati letti. La routine DECODE decodifica una decisione binaria D e restituisce in output il valore del bit decodificato. Quando tutti i contesti sono stati letti (test: Finished?), i dati compressi sono stati tutti decompressi.

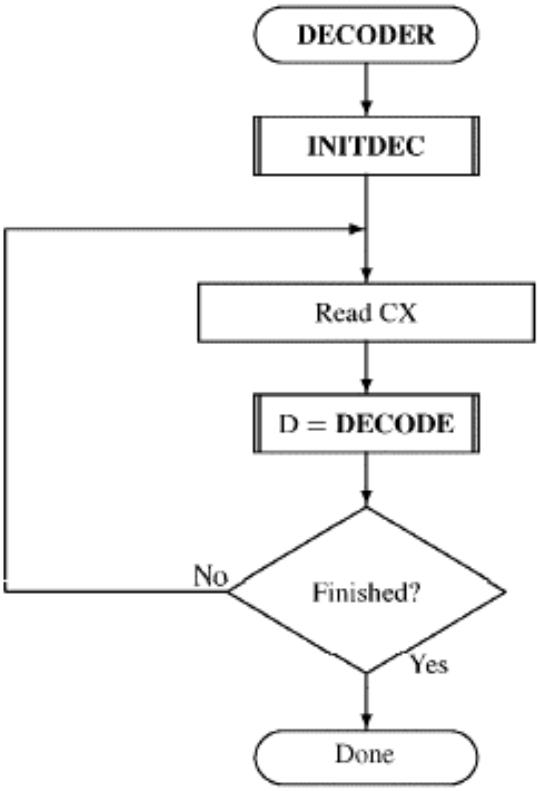

Figura 42: Diagramma di flusso per il decoder.

Il blocco INITDEC, utilizzato per l'inizializzazione delle variabili del decoder, è mostrato nel diagramma seguente.

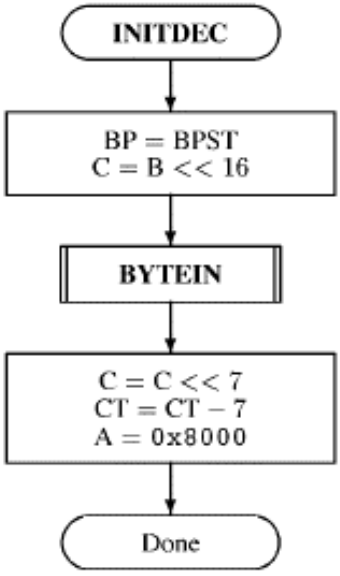

Figura 43: Inizializzazione del decoder.

Il puntatore BP ai dati compressi è inizializzato a BPST (che punta al primo byte dei dati compressi). Il primo byte dei dati compressi è shiftato nel byte di ordine più basso del registro Chigh e un nuovo byte è letto. Il registro C è shiftato a sinistra di 7 bit e CT è decrementato di 7, per allineare C al valore iniziale del registro intervallo, A. Il valore di A

è uguale a quello scelto per l'encoder, per assicurare la corretta ricostruzione della sequenza codificata.
Il blocco BYTEIN sarà descritto più avanti in questa sezione.

Il blocco DECODE si occupa della decodifica dei simboli, a partire dai dati compressi e dai contesti forniti dal context modeler. Il decoder decodifica una decisione binaria D alla volta. Dopo aver decodificato D, sottrae dai dati compressi la quantità che il decoder aveva aggiunto. La quantità rimanente nei dati compressi è l'offset, dalla base dell'intervallo corrente, del sotto-intervallo associato alle decisioni binarie non ancora decodificate.

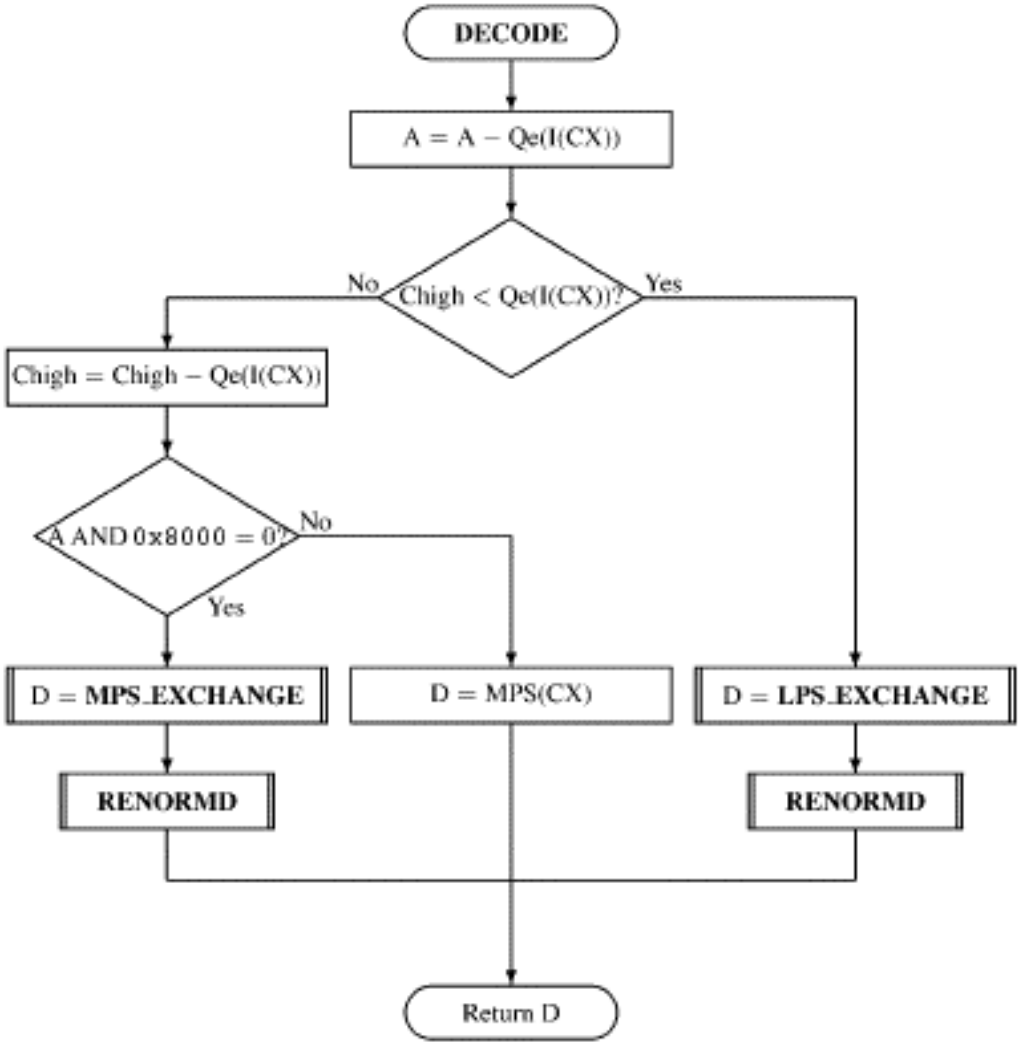

Figura 44: Decodifica di una decisione binaria D.

Inizialmente, l'ampiezza dell'intervallo è aggiornata sottraendo dal registro A la probabilità del LPS per il contesto CX. Il registro Chigh è confrontato con l'ampiezza del sotto-intervallo associato al LPS. Se non è necessario uno scambio condizionale, questo test determina se è stato codificato un MPS o un LPS. In particolare, se Chigh è minore di *Qe*(I(CX)), il sotto-intervallo codificato è quello inferiore, quindi il simbolo codificato è quello meno probabile, altrimenti il simbolo codificato è quello più probabile. In questo caso l'ampiezza del sotto-intervallo associato al LPS deve essere sottratta dal registro C, per far sì che questo mantenga il suo ruolo di puntatore alla base dell'intervallo corrente. Se l'ampiezza dell'intervallo corrente (aggiornata al primo passo) è nel range legale, il

simbolo in output è il MPS. Se invece A deve essere rinormalizzato, si deve verificare se è necessario eseguire uno scambio condizionale. Ciò è fatto mediante la procedura MPS_EXCHANGE. Si noti che, nel caso in cui il primo test dia esito positivo, la procedura LPS_EXCHANGE, che verifica se sia avvenuto uno scambio condizionale o meno, è chiamata subito poiché la rinormalizzazione è eseguita in ogni caso.

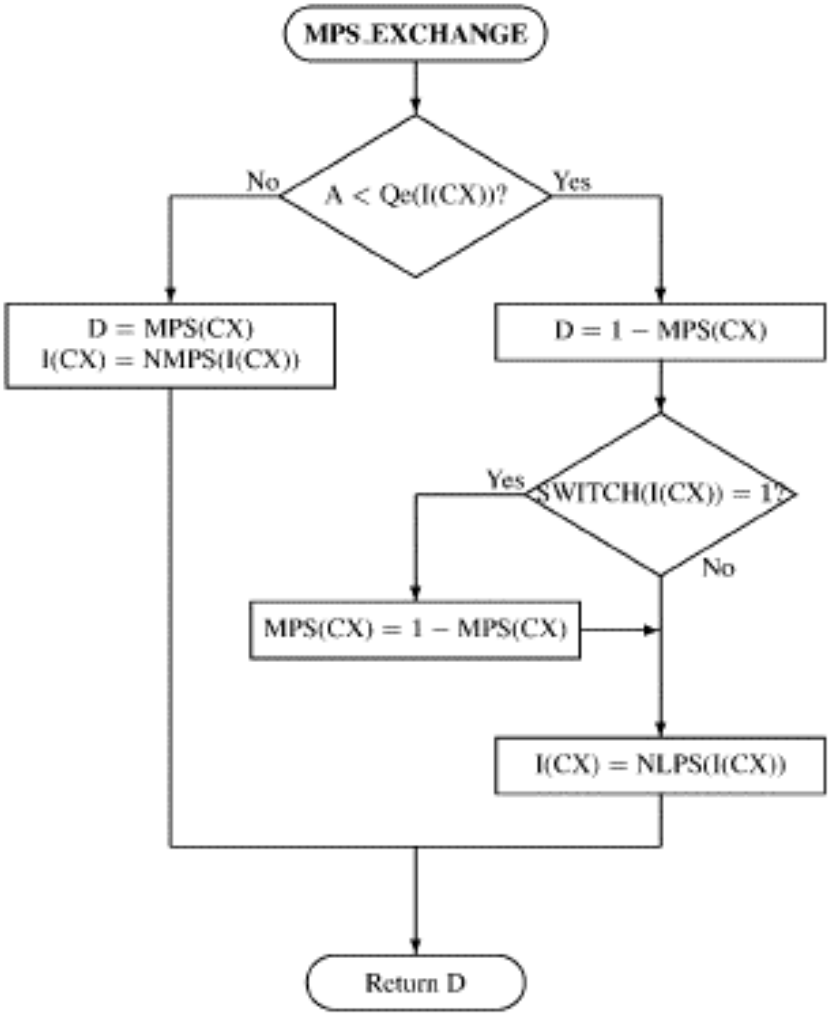

Figura 45: Procedura per determinare se è avvenuto uno scambio condizionale, sul ramo della computazione relativo al MPS.

Se si esegue la procedura MPS_EXCHANGE, per il primo test della procedura DECODE, è stato codificato l'intervallo superiore. Se l'ampiezza del sotto-intervallo corrente è maggiore della probabilità del LPS, poiché ci aspettiamo il MPS (per il primo test della procedura DECODE), non c'è stato scambio condizionale e il simbolo codificato è il MPS. Allora D è posto uguale al MPS e lo stato successivo è quello indicato da NMPS(I(CX)). Se l'ampiezza del sotto-intervallo corrente è minore o uguale a *Qe*(I(CX)), c'è stato scambio condizionale perché è stato codificato l'intervallo superiore, ma l'ampiezza del nuovo intervallo è minore o uguale a quella del sottointervallo associato al LPS. Allora la decisione binaria D è posta uguale al simbolo meno probabile e, se *Switch*(I(CX)) è settato, il senso del MPS è invertito. Lo stato seguente è quello indicato da NLPS(I(CX)).

La procedura LPS_EXCHANGE, duale a quella appena descritta, è descritta nel seguente diagramma.

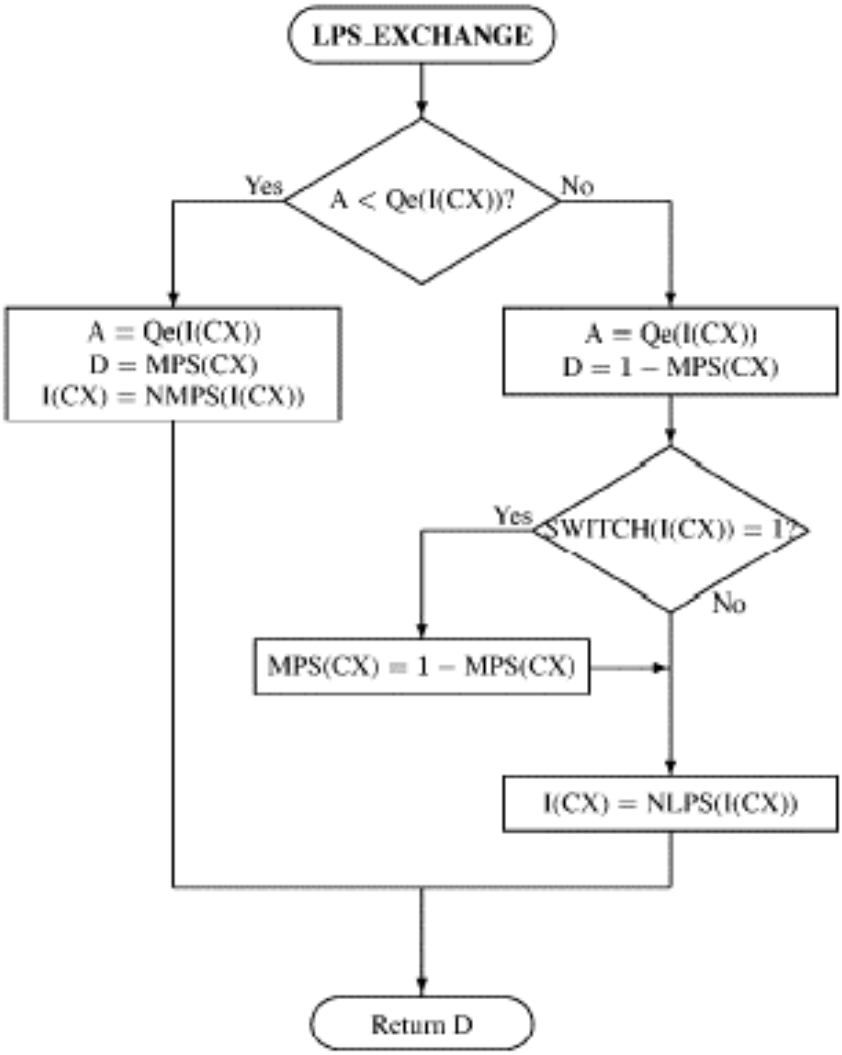

Figura 46: Procedura per determinare se è avvenuto uno scambio condizionale, sul ramo della computazione relativo al LPS.

La procedura che si occupa della rinormalizzazione, RENORMD, è mostrata nel diagramma seguente. Il contatore CT tiene traccia del numero di bit compressi nella sezione Chigh del registro C. Quando CT raggiunge lo zero, un nuovo byte è inserito in Clow mediante la procedura BYTEIN. Entrambi i registri A e C sono shiftati a sinistra, un bit alla volta, finché il valore di A è al di fuori del range legale.

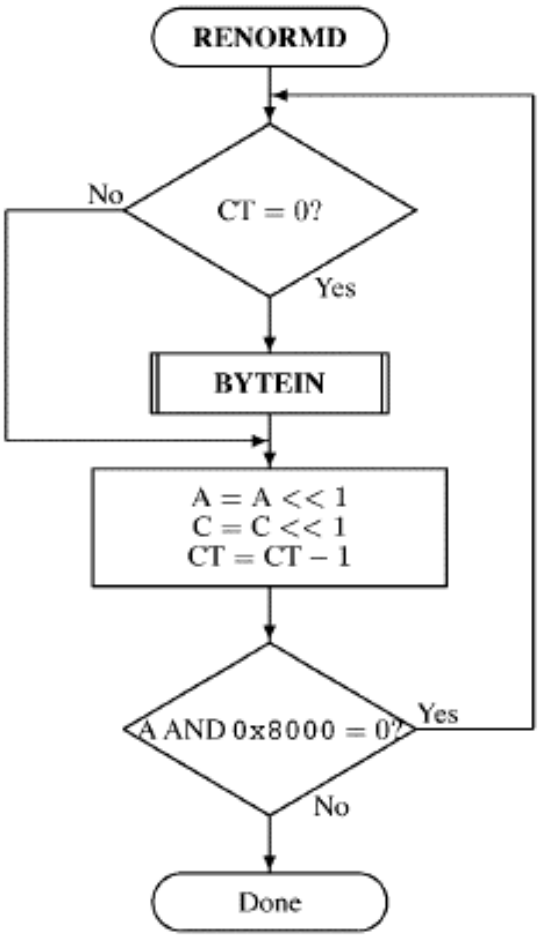

Figura 47: Procedura di rinormalizzazione del decoder

La procedura BYTEIN è invocata da RENORMD e da INITDEC. Questa procedura legge un byte di dati compressi, compensando eventuali bit di riempimento inseriti dal decoder dopo un byte 0xFF. Essa, inoltre, riconosce eventuali marcatori. Il registro C è la concatenazione dei bit dei registri Chigh e Clow.

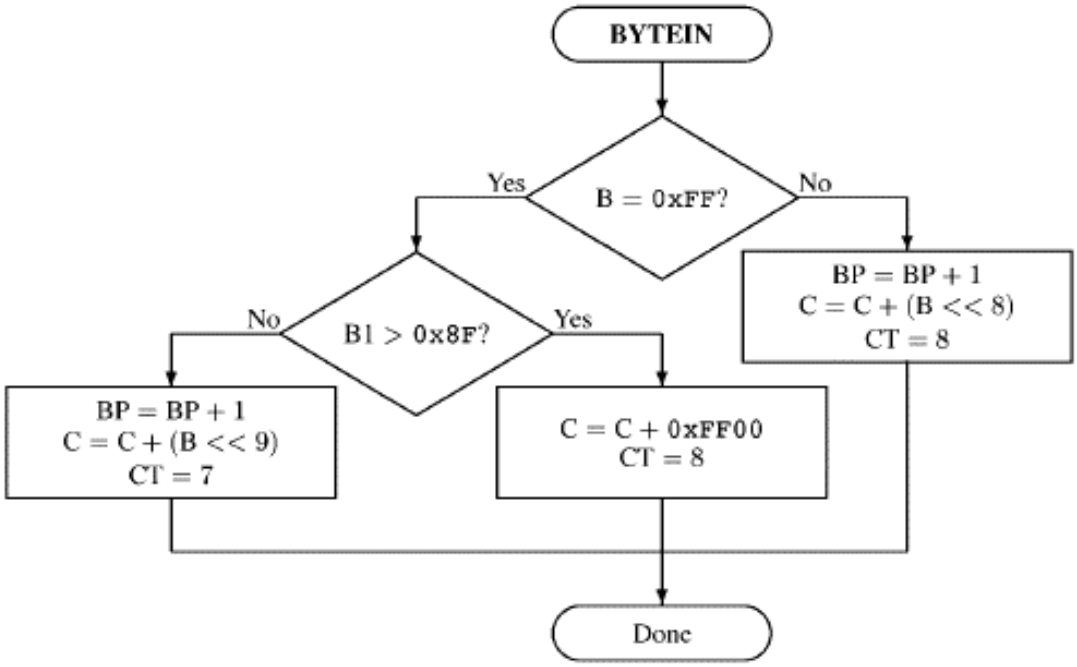

Figura 48: Procedura di input per il decoder.

B è il byte di dati compressi puntato da BP. Se B non è 0xFF, BP è incrementato per puntare al byte successivo e il valore di B è inserito negli otto bit di ordine più alto di Clow. Se B è 0xFF, per stabilire se è il primo byte di un marcatore, bisogna testare il valore del byte B1, che segue B nello stream di dati compressi (B1 è il byte puntato da BP+1). Se B1 è minore o uguale a 0x8F, B1 rappresenta una sequenza di 7 bit di dati codificati, preceduti da un bit di riporto. Allora BP è incrementato per puntare al successivo byte di dati, i 7 bit di dati di B sono copiati nel byte di ordine superiore di Clow e il bit di riporto è sommato a Chigh. CT è posto uguale a 7 perché sono stati inseriti solamente 7 nuovi bit nel registro C. Se B1 è maggiore di 0x8F, la coppia B, B1 è un marcatore. Allora bit 1 sono forniti al decoder, fino alla decodifica completa dei dati compressi. Ciò è fatto settando i bit nel byte di ordine più alto di Clow, mediante la somma di 0xFF00, e ponendo CT uguale a 8. Il puntatore BP deve rimane immutato per far sì che, alla chiamata successiva, BYTEIN ripeta lo stesso procedimento, fornendo otto bit 1 al decoder.

Questo conclude la discussione sulla codifica aritmetica in JPEG2000 e sul tier 1 di EBCOT. Prima di passare alla descrizione del tier 2, è opportuno dare uno sguardo d'insieme al sistema descritto finora.
EBCOT riceve in input un blocco di coefficienti wavelet quantizzati. Il blocco è codificato a bit-planes. Ciascun bit-plane è scandito tre volte, una per ogni passo di codifica, seguendo un ordine fissato. Durante ogni passata si codificano solo i coefficienti appartenenti al passo di codifica corrente. Per ciascun bit da codificare è fissato un contesto, determinato in base alle informazioni sulla significanza del bit stesso e dei suoi otto immediati vicini. Il bit da codificare e il contesto ad esso associato sono passati al codificatore aritmetico, che genera la sequenza di dati compressi. Durante la codifica dei blocchi, insieme ai dati compressi sono conservate informazioni relative ai punti di troncamento legali, che saranno utilizzate per l'analisi dell'involucro convesso descritta all'inizio di questa sezione, per trovare i punti di troncamento ottimali. I dati compressi e queste informazioni costituiscono l'output del tier 1.

Capitolo 5

Formazione del Bit-Stream Compresso: EBCOT Tier 2

In questo capitolo sarà discussa l'organizzazione del bit-stream compresso. Inoltre, saranno illustrate le tecniche che permettono di segnalare in maniera efficiente le informazioni sui quality layers (blocchi inclusi, quantità di informazione codificata per ciascun blocco incluso, ecc.).

5.1. Organizzazione del Bit-Stream Compresso

Uno degli obiettivi di JPEG2000 è quello di permettere all'utente di stabilire, in fase di codifica, la dimensione finale del file compresso. In altre parole, vogliamo che, fissato un target bit-rate *R*, l'algoritmo sia capace di generare un bit-stream compresso la cui codifica richieda *R* bit/pixel, assicurando la migliore qualità possibile. Questo compito è affidato al tier 2 di EBCOT.
A parità di dimensione, la qualità dell'immagine compressa, nel senso del rapporto segnale/rumore (SNR), è proporzionale al valore dell'informazione codificata. Il tier 1 è stato progettato con lo scopo di "riordinare" i bit di ciascun blocco in ordine decrescente di rilevanza. Quest'ordinamento ci permette di scartare l'informazione meno significativa, semplicemente troncando il bit-stream di ciascun blocco ad una certa lunghezza, il che equivale ad una quantizzazione "intelligente" dei coefficienti. Purtroppo, le inevitabili dipendenze tra i bit codificati, introdotte dalla codifica entropica, limitano il numero di lunghezze a cui la codeword generata da ogni blocco può essere troncata. I dati compressi si presentano, allora, in "pezzi", che nelle sezioni precedenti abbiamo chiamato *chunks*, ciascuno dei quali, concatenato ai precedenti, riduce la distorsione dovuta al troncamento del bit-stream. La Figura 16 mostra la codeword generata da un blocco, divisa in chunks.
L'idea è quella di estendere l'ordinamento tra i chunks di un blocco all'intera immagine, in maniera tale da ottenere un'ottimizzazione globale del SNR. Ciò è realizzato introducendo la struttura a quality layers progressivi. Il compito del tier 2 di EBCOT è, allora, quello di stabilire la distribuzione ottimale dei chunks nei diversi layers. Si noti che l'encoder è

libero di stabilire la distribuzione più appropriata, purché venga mantenuto l'ordinamento relativo tra i chunks di un blocco.
L'algoritmo di controllo del bit-rate agisce dopo la codifica entropica dei blocchi. Solo quando tutti i blocchi sono stati codificati, infatti, l'encoder conosce i punti di troncamento legali di ogni blocco ed è in grado, quindi, di stabilire la corretta distribuzione dei pezzi di codeword nei layers. Per questa caratteristica, questo algoritmo è detto *Post Compression Rate/Distortion optimization* (*PCRD optimization*).
Fissato un bit-rate massimo, l'algoritmo di controllo del bit-rate deve trovare i punti di troncamento ottimali, nel senso del SNR, di ogni codeword (ciò può essere fatto mediante l'analisi dell'involucro convesso discussa nella sezione 2.3.1) e stabilire, per ogni layer, quali chunks permettono, se aggiunti allo stream di output, di ottenere la maggiore diminuzione di distorsione nell'immagine ricostruita. I chunks trovati sono aggiunti al nuovo layer. Il layer così costruito è ottimizzato nel senso del SNR, perché aggiunge all'immagine decodificata informazioni che permettono di ottenere la maggiore diminuzione di distorsione possibile (quindi il miglior incremento in qualità), limitatamente all'incremento nel numero di bit nella codifica dell'immagine.
La figura seguente mostra l'organizzazione del bit-stream di JPEG2000. Il bit-stream è costituito da un'intestazione contenente informazioni globali, seguita da una o più sezioni corrispondenti alle tiles. Ciascuna sezione consiste di un'intestazione che contiene informazioni locali alla tile, seguita dalla rappresentazione a layer dei blocchi che costituiscono la tile, organizzata in una collezione di strutture, dette *pacchetti*.

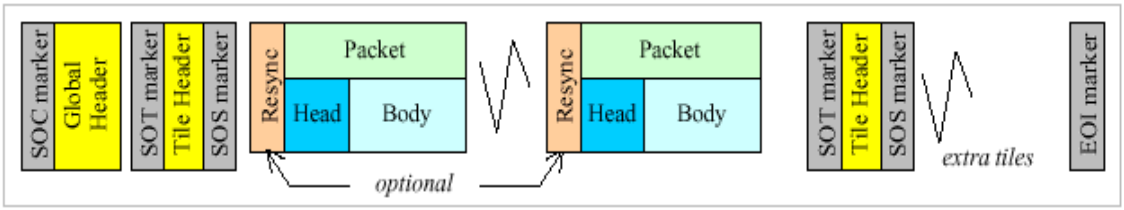

Figura 49: Organizzazione del bit-stream JPEG2000. I marcatori *Resync* sono opzionali.

SOC ed EOI sono, rispettivamente i marcatori di inizio e fine dello stream. SOT indica l'inizio della sezione relativa ad una tile. SOS è il marcatore di inizio del bit-stream compresso. I marcatori *Resync* sono utilizzati per il controllo di errori di trasmissione (*error resilience*) e sono opzionali. Per maggiori informazioni sui marcatori e sulle intestazioni, si faccia riferimento a [2].
Un pacchetto è costituito da un preambolo, seguito dal corpo. Il preambolo contiene informazioni relative ai blocchi che contribuiscono al layer rilevante, compresi i punti di troncamento associati a questi blocchi e il numero di bit di codice forniti da ogni blocco. Il corpo contiene i bit di codice generati dal tier 1. Sia la testa che il corpo del pacchetto occupano un numero intero di bytes, in modo tale da essere sempre allineati a byte. Testa e corpo di un pacchetto non sono delineati da marcatori. La dimensione di entrambi può essere dedotta dal contenuto del preambolo. La struttura del preambolo sarà descritta più avanti.
La disposizione dei pacchetti nello stream riveste un ruolo importante nel processo di costruzione dello stream stesso, perché determina l'ordine con cui l'immagine sarà trasmessa e decodificata. Sia Λ il numero di layers, etichettati con $\lambda = 1, 2, \ldots, \Lambda$. Ogni stream è inerentemente scalabile rispetto a risoluzione e componente di colore, perché è composto da pacchetti $K_\lambda^{l,c}$ separati, per ogni livello di risoluzione $l = 0, 1, \ldots, L$ e per ogni componente c. Inoltre, la struttura a layer assicura la scalabilità per qualità.

Consideriamo, inizialmente, immagini con una sola componente (monocromatiche). Se lo stream è codificato utilizzando un solo layer, i pacchetti $K_1^{l,1}$ appaiono nell'ordine mostrato nella seguente figura.

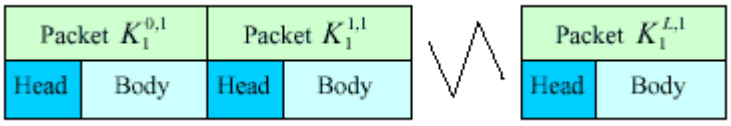

Figura 50: Organizzazione di un bit-stream con un solo layer.

In questo modo otteniamo un bit-stream progressivo per risoluzione, ma non scalabile per qualità. Per ottenere un bit-stream che sia scalabile anche per qualità, dobbiamo suddividere i blocchi in più layer. Sono possibili due disposizioni dei pacchetti: una progressiva per risoluzione, l'altra per qualità. Queste due organizzazioni sono mostrate nelle figure seguenti.

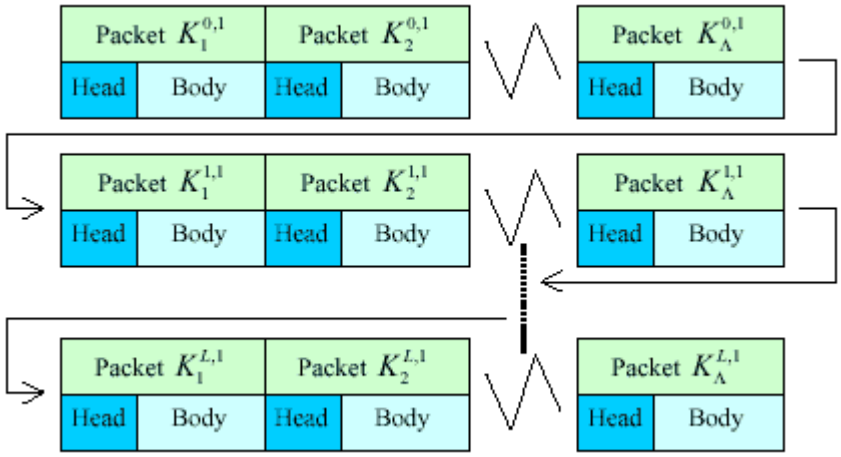

Figura 51: Organizzazione dello stream a più livelli, progressiva per risoluzione.

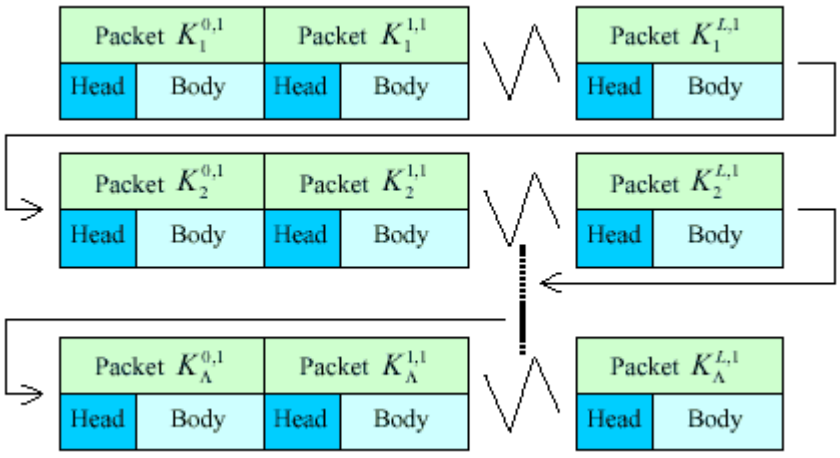

Figura 52: Organizzazione dello stream a più livelli, progressiva per qualità.

Nella prima disposizione, appaiono prima i pacchetti relativi ai layers 0, 1, …, Λ del livello di risoluzione 0 (il più basso), poi quelli dei livelli successivi, fino al livello *L* (corrispondente all'immagine originale). Evidentemente questa organizzazione è progressiva rispetto alla risoluzione, poiché decodificando i primi Λ pacchetti si ottiene il primo livello di risoluzione con la qualità massima, poi il secondo è ottenuto decodificando i successivi Λ, e così via. Nella seconda disposizione, invece, lo stream è organizzato a sequenze di *L* pacchetti tutti relativi allo stesso layer e le sequenze appaiono in ordine, dal layer 1 (il più basso) al layer Λ (massima qualità). Lo stream è, quindi, progressivo per qualità.
Per immagini a più componenti, nello stream appare un pacchetto separato per ogni componente, all'interno di un dato livello di risoluzione. I pacchetti associati a differenti componenti dell'immagine appaiono consecutivamente nello stream, in ordine crescente rispetto alla componente. Quindi, per una tipica immagine a tre componenti, i pacchetti $K_\lambda^{l,1}$, $K_\lambda^{l,2}$ e $K_\lambda^{l,3}$ appaiono insieme, per ogni livello di risoluzione *l* e per ogni layer λ. Poiché sono costruiti pacchetti separati per le diverse componenti di colore, lo stream è necessariamente scalabile rispetto alle componenti.
Per maggiori dettagli sull'ordine di progressione, si veda [2].
Prima di descrivere la sintassi dell'intestazione del pacchetto, esaminiamo la struttura *tag tree*, usata per codificare in maniera efficiente alcune delle informazioni segnalate in tale intestazione.

5.2. Tag Trees

Un tag tree è una particolare struttura ad albero che fornisce un modo efficiente di rappresentare un'informazione numerica associata alle foglie.
Sia $q_1[m,n]$ una matrice bidimensionale di interi non-negativi che vogliamo rappresentare mediante un tag tree. Associamo queste quantità alle foglie dell'albero. Indichiamo con $q_2[m,n]$ la matrice dei nodi del livello successivo. Ciascuno di questi nodi è associato a un blocco 2x2 di nodi foglia, tranne ai bordi della matrice originale. Allo stesso modo costruiamo le matrici $q_3[m,n]$, $q_4[m,n]$, …, fino a raggiungere il livello della radice, che consiste di un singolo nodo $q_K[0,0]$. La quantità associata ad ogni elemento di ciascuna matrice è il minimo dei sui discendenti:

$$q_k[m,n] = \min\{q_{k-1}[2m,2n], q_{k-1}[2m,2n+1], q_{k-1}[2m+1,2n], q_{k-1}[2m+1,2n+1]\} \qquad \text{(E.32)}$$

dove $k > 1$. I riferimenti a nodi che stanno al di fuori della matrice rilevante devono essere ignorati nel calcolo del minimo. La figura seguente mostra un esempio di un tag tree costruito a partire da una matrice 3x6 di interi.

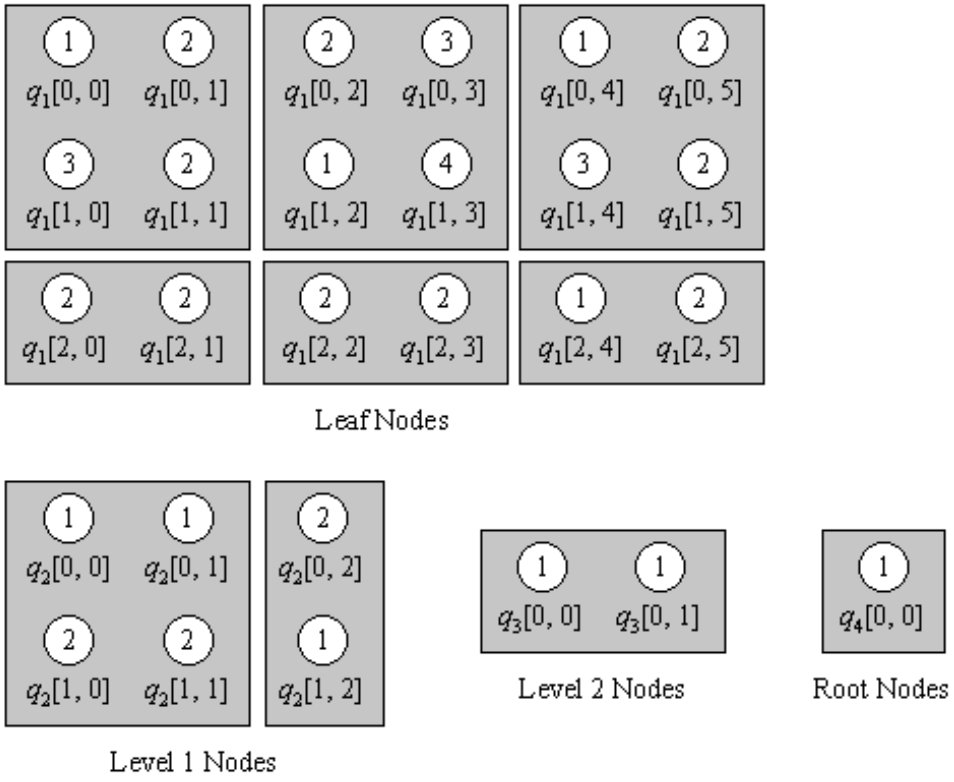

Figura 53: Esempio di un tag tree costruito a partire da una matrice 3×6 di interi.

Lo scopo dei tag trees è quello di codificare la minima quantità di informazione necessaria a stabilire se $q_1[\overline{m},\overline{n}] \geq t$ per una coppia di indici $(\overline{m},\overline{n})$ e una soglia *t*, dato quello che è stato già codificato per questo e per gli altri nodi. Più precisamente, esprimiamo l'algoritmo per il calcolo del tag tree come una procedura $\mathbf{T}(\overline{m},\overline{n},t)$, che codifica la quantità minima di bit necessaria ad indicare se $q_1[\overline{m},\overline{n}] \geq t'$ per ogni t' nell'intervallo $1 \leq t' \leq t$. Introduciamo la variabile di stato $t_k[m,n]$ per ogni nodo in ogni livello del tag tree, che rappresenta l'informazione che è stata codificata per il valore di $q_k[m,n]$. In particolare, essa specifica che è stata codificata informazione sufficiente a stabilire se $q_k[\overline{m},\overline{n}] \geq t'$ per t'=1, 2, …, $t_k[m,n]$. Tutte le variabili di stato sono inizializzate a 0, in accordo con il fatto che niente è stato ancora codificato, dato che tutti i valori $q_1[m,n]$ sono non-negativi. Durante il processo di codifica, i valori $t_k[m,n]$ sono incrementati monotonicamente; il valore di $q_k[m,n]$ risulta completamente identificato non appena $t_k[m,n] > q_k[m,n]$ Quando $t \leq t_1[m,n]$ abbiamo codificato informazione sufficiente a stabilire se $q_1[\overline{m},\overline{n}] \geq t'$ per ogni t', quindi la procedura termina.

L'algoritmo che descrive la procedura $\mathbf{T}(\overline{m},\overline{n},t)$ è il seguente:

(1) $k = K$ /* Start at the root node */

(2) $t_{\min} = 0$ /* used to propagate knowledge to descendants */

(3) $m_k = \left\lfloor \frac{\overline{m}}{2^{k-1}} \right\rfloor, \quad n_k = \left\lfloor \frac{\overline{n}}{2^{k-1}} \right\rfloor$ /* $[m_k, n_k]$ is the location of the leaf ancestor node in level k */

(4) if $t_k[m_k, n_k] < t_{\min}$ set $t_k[m_k, n_k] = t_{\min}$ /* update the state variable */

(5) if $t \le t_k[m_k, n_k]$

- if $k = 1$ we are done. /* we have reached the leaf and $t_k[m_k, n_k] = t_1[\overline{m}, \overline{n}]$. Sufficient information has been encoded to identify wheter or not $q_1[\overline{m}, \overline{n}] \ge t'$ for each $t' \le t_1[\overline{m}, \overline{n}]$ and $t \le t_1[\overline{m}, \overline{n}]$, so we are done */
- otherwise
  - $t_{\min} = \min\{t_k[m_k, n_k], q_k[m_k, n_k]\}$ /* update $t_{\min}$ to propagate knowledge to the node we will visit in the next level */
  - set $k = k - 1$ and go to step (3)

(6) otherwise (i.e. if $t > t_k[m_k, n_k]$) /* send enough information to increase $t_k[m_k, n_k]$ */

- if $q_k[m_k, n_k] > t_k[m_k, n_k]$ emit a "0" bit
- else if $q_k[m_k, n_k] = t_k[m_k, n_k]$ emit a "1" bit
- set $t_k[m_k, n_k] = t_k[m_k, n_k] + 1$ and go to step (5)

Algoritmo A.6: Algoritmo per il calcolo di $\mathbf{T}(\overline{m},\overline{n},t)$.

Nonostante l'algoritmo presentato possa sembrare un po' macchinoso, l'idea è piuttosto semplice. Si parte dalla radice. Se l'informazione minima per stabilire se $q_K[0,0] \ge t$ non è stata ancora codificata, la si codifica adesso. Poi ci si muove verso la foglia a cui siamo interessati, $q_1[\overline{m}, \overline{n}]$, aggiornando il nodo per riflettere ogni informazione che può essere dedotta da ciò che già è noto dal nodo genitore (cioè: il valore del nodo corrente non deve essere minore di $t_{\min}$). Questo processo è iterato fino a quando non si raggiunge la foglia. A questo punto avremo codificato informazione sufficiente per stabilire se $q_1[\overline{m}, \overline{n}] \ge t$, quindi la procedura termina. Il vantaggio di codificare le informazioni contenute nelle foglie mediante questa struttura sta nel fatto che essa è capace di mettere in evidenza le ridondanze tra nodi vicini e tra un nodo e i propri discendenti. Questa caratteristica è usata per ridurre notevolmente il numero di bit codificati per rappresentare l'informazione contenuta nelle foglie del tag tree. Un esempio di codifica mediante tag trees è presentato più avanti, a proposito della codifica delle informazioni di inclusione dei blocchi nei pacchetti.
Il diagramma seguente rappresenta l'algoritmo descritto.

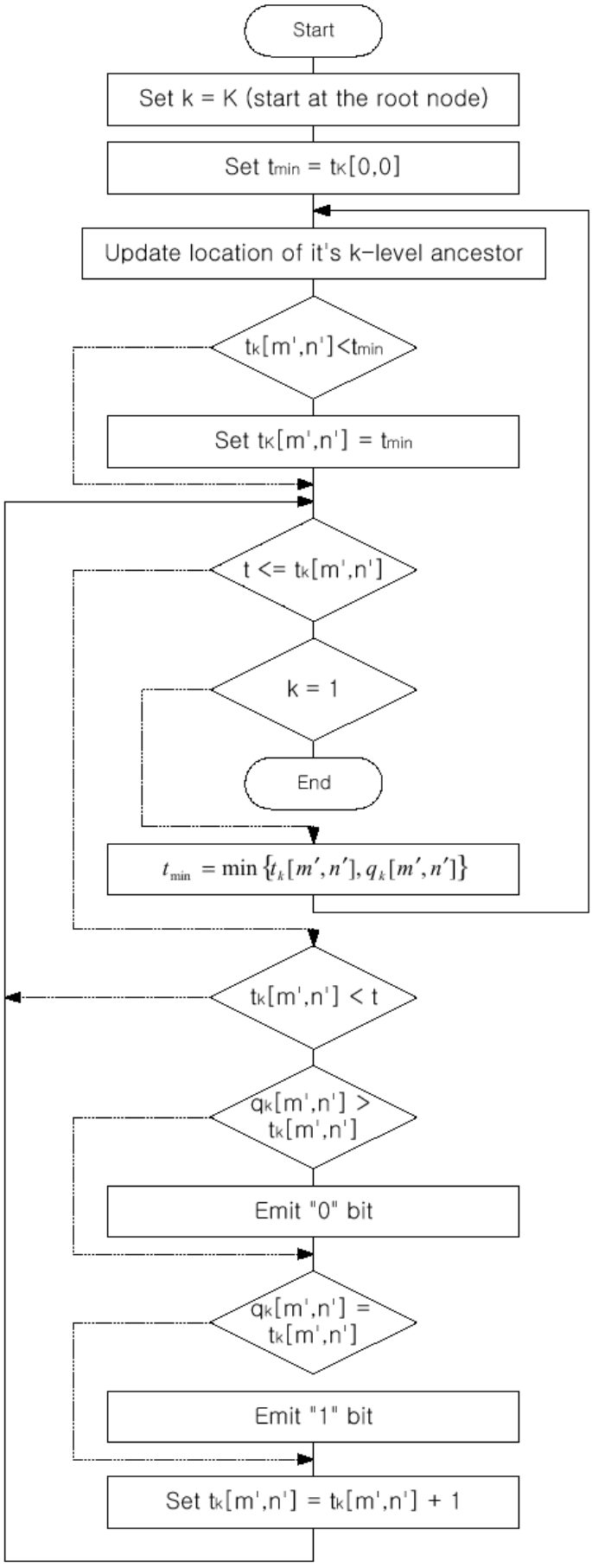

Figura 54: Procedura di codifica dei Tag Trees $\mathbf{T}(\overline{m},\overline{n},t)$.

Una caratteristica importante di questa struttura è la possibilità di modificare il valore delle quantità rappresentate dal tag tree, dopo aver codificato delle informazioni. In particolare, è possibile cambiare la quantità $q_1[m,n]$, associata ad una foglia, purché sia il nuovo che il vecchio valore associati al nodo siano maggiori o uguali alla più grande soglia *t*, codificata mediante la procedura $\mathbf{T}(\overline{m},\overline{n},t)$, per ogni nodo.

5.3. Intestazione dei Pacchetti

Vediamo, ora, in che modo la struttura descritta sia usata nella codifica dell'intestazione dei pacchetti. Un pacchetto $K_\lambda^{l,c}$ rappresenta la nuova informazione introdotta nel layer λ, per i blocchi nelle sotto-bande del livello di risoluzione *l* della componente *c*. L'intestazione dei pacchetti consiste di una sequenza di bit che identificano il contributo, al nuovo layer, di ciascun blocco nel livello di risoluzione corrente, relativamente alla componente *c*. Il primo bit è usato per indicare al decoder se il pacchetto è vuoto o contiene nuove informazioni. La parte rimanente dell'intestazione è divisa in diverse sezioni, ciascuna corrispondente ad una sottobanda nel livello di risoluzione cui il pacchetto si riferisce. Il modo in cui le bande sono ordinate nel pacchetto dipende dal tipo di decomposizione wavelet usata. Al livello $l = 0$, nel pacchetto c'è solo la sottobanda LL. Nei livelli successivi, la disposizione varia in relazione al valore del parametro ϕ_l. Per ϕ_l=1, il pacchetto contiene, nell'ordine, le bande HL, LH e HH. Per ϕ_l=2, le sotto-bande HL, LH e HH del livello di risoluzione più alto sono decomposte, ciascuna, in quattro sottobande. Nel pacchetto corrispondente ci sono, dunque, 12 bande. Associamo alle sotto-bande LL, HL, LH, HH un indice, rispettivamente 0, 1, 2, 3 e indichiamo le sottobande di livello più alto con b_1 e quelle di livello più basso con b_2. Allora l'ordine con cui le sotto-bande appaiono nel pacchetto è determinato da $b = 4b_1 + b_2$, cioè eseguiamo una visita pre-order dell'albero della decomposizione wavelet. Analogamente, per ϕ_l=3 le sotto-bande sono ordinate rispetto a $b = 16b_1 + 4b_2 + b_3$, dove b_1 identifica la decomposizione primaria, b_2 la decomposizione secondaria e b_3 quella finale. All'interno di ciascuna sotto-banda, le informazioni relative ai blocchi sono disposti in ordine raster.
Per ciascun blocco, l'intestazione contiene le seguenti informazioni:

- se il blocco è incluso o meno nel pacchetto
- massimo numero di bit nella mantissa dei coefficienti del blocco
- numero di passi di codifica inclusi
- lunghezza dei dati compressi (in bytes)

Si noti che né le sezioni, né i bit usati per rappresentare queste quattro informazioni sono allineati a byte; i bit sono semplicemente concatenati. Viceversa, l'intera intestazione e il corpo del pacchetto sono, ciascuno, costituiti da un numero intero di bytes.
I dati codificati che appaiono nel corpo del pacchetto seguono la stessa disposizione usata nell'intestazione.
Per rappresentare in maniera efficiente l'inclusione dei blocchi, si utilizza un tag tree separato per ogni sotto-banda. Le foglie dell'albero corrispondono ai blocchi e la quantità $q_1[m,n]$ è l'indice del layer in cui il blocco di indice (*m*,*n*) è incluso per la prima volta, meno 1, in simboli $q_1[m,n] = \lambda[m,n] - 1$. Gli indici dei layer partono da 1, quindi

l'assegnamento precedente non viola il vincolo che i valori dei nodi del tag tree devono essere non-negativi. Per ciascun blocco nel layer λ, l'informazione di inclusione è rappresentata in due modi diversi. Se il blocco è già stato incluso in un layer precedente, cioè $\lambda[m,n] < \lambda$, si codifica un solo bit, per indicare se il blocco contribuisce o meno al nuovo layer ("1" se il blocco è incluso, "0" altrimenti). Se, invece, il blocco non è stato incluso in nessuno dei layer precedenti, invochiamo la procedura di codifica dei tag tree $\mathbf{T}(m,n,\lambda)$. Questa procedura codifica un numero di bit sufficiente a stabilire se $q_1[m,n] \geq \lambda$, cioè se $\lambda[m,n] > \lambda$, che è l'informazione di inclusione richiesta.
La seguente figura mostra l'informazione di inclusione per un pacchetto contenente sei blocchi.

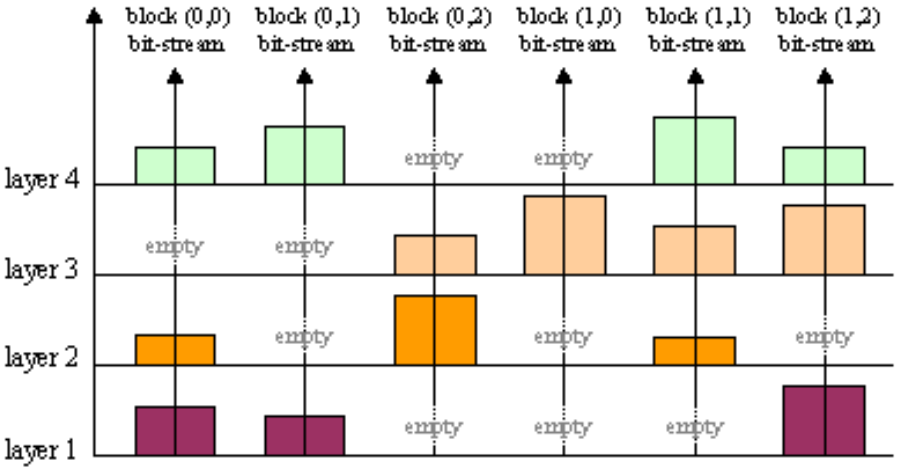

Figura 55: Esempio di informazione di inclusione per un pacchetto contenente sei blocchi.

Il tag tree generato per rappresentare l'informazione di inclusione in figura è mostrato di seguito.

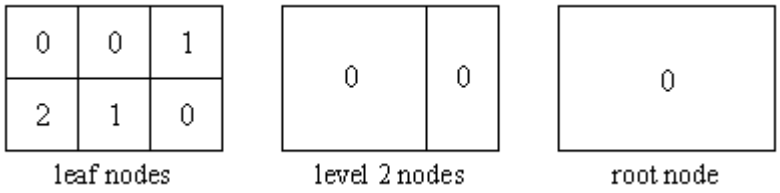

Figura 56: Tag tree generato per rappresentare l'informazione di inclusione nella precedente figura.

Durante la codifica dell'informazione di inclusione, i blocchi sono scanditi in ordine raster. La seguente figura mostra come tale informazione è codificata. I simboli in neretto sono quelli codificati mediante la procedura $\mathbf{T}(m,n,\lambda)$ (non inclusi in nessuno dei layer precedenti), per gli altri un solo bit è usato per indicare se il blocco contribuisce o meno al nuovo layer (blocchi già inclusi in qualcuno dei layer precedenti).

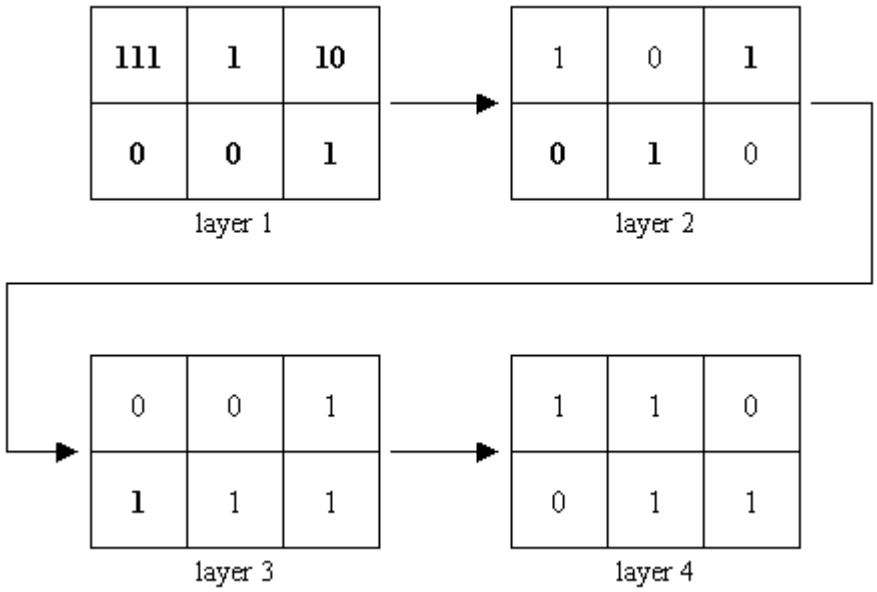

Figura 57: Codifica dell'informazione di inclusione rappresentata mediante il tag tree nella figura precedente.

Evidentemente, l'inclusione di tutti i blocchi del primo layer è codificata mediante la procedura di codifica dei tag tree perché non ci sono layer precedenti. Il primo blocco è codificato chiamando la procedura di codifica del tag tree con i parametri $m = 0$, $n = 0$ ($B(0,0)$ è il primo blocco) e $\lambda = 1$. La procedura esamina il nodo radice e vede che nessuna informazione è stata ancora codificata per stabilire se $q_3[0,0] \geq 1 = \lambda$ (passo 6 dell'algoritmo A.6); $q_3[0,0] < \lambda$, quindi è codificato un bit "1". Analogamente, nel passo successivo l'algoritmo verifica che nessuna informazione è stata codificata; $q_2[0,0] < \lambda$, quindi è codificato un altro bit "1". Lo stesso accade nel terzo passo. Allora il risultato è $\mathbf{T}(0,0,1) = \mathbf{111}$. Quando codifichiamo il secondo blocco, i nodi dei primi due livelli dell'albero sono uguali a quelli del blocco precedente. Poiché il layer non è cambiato, è ancora $\lambda = 1$; l'informazione necessaria a stabilire se $q_3[0,0] \geq \lambda$ e $q_2[0,0] \geq \lambda$ è stata codifica per il blocco precedente. Ci rimane da codificare l'informazione $q_1[0,1] \geq \lambda$. Il valore ritornato dalla procedura è allora $\mathbf{T}(0,1,1) = \mathbf{1}$. Per codificare il terzo blocco, la procedura di codifica scopre che il nodo radice è già stato codificato dalle chiamate precedenti, mentre i per i nodi $q_2[0,1]$ e $q_2[0,0]$ non è stata codificata alcuna informazione. L'output è allora $\mathbf{T}(0,2,1) = \mathbf{10}$. Il calcolo procede analogamente per tutti gli altri nodi del tag tree.

L'idea intuitiva che sta alla base della procedura di codifica dei tag trees è che l'informazione è codificata una sola volta per ogni nodo dell'albero. Le chiamate successive sfruttano la ridondanza tra elementi vicini per evitare di codificare i bit già codificati nelle chiamate precedenti. La prima chiamata deve codificare un bit per ogni livello dell'albero perché il decoder, prima della decodifica del primo valore, non conosce nulla del tag tree, se non il numero di livelli (il decoder conosce le dimensioni della matrice $q_1[m,n]$ e, quindi, può costruire un tag tree vuoto con il corretto numero di livelli). Nel nostro esempio, l'albero ha tre livelli e il valore di ciascun nodo nel primo percorso è minore del valore di *t*, quindi la codifica della prima foglia è "**111**". La seconda chiamata ha in comune con la prima il percorso che attraversa primi due livelli, quindi deve codificare solo il bit relativo alla foglia, "**1**". La terza chiamata, invece, condivide con le

prime due solo la radice, quindi la codifica richiede 2 bit, "**10**", uno per il nodo al secondo livello, l'altro per la foglia.
Si noti che non è necessario aspettare di codificare tutti i layer prima di generare il tag tree che descrive l'informazione di inclusione dei blocchi. In precedenza, infatti, si è accennato al fatto che è possibile aggiornare il valore delle foglie di un tag tree, purché sia il nuovo che il vecchio valore siano maggiori o uguali della più grande soglia t fornita alla procedura $\mathbf{T}(m,n,t)$. Questa proprietà ci consente di costruire l'albero man mano che i layer vengono codificati. Il valore dei nuovi elementi, infatti, è sempre uguale al valore del parametro t ($t = \lambda$) con cui sarà chiamata la procedura per il nuovo layer. Allora, basta associare infinito (o un intero sufficientemente grande) alle foglie del tag tree vuoto, per rendere vere le condizioni che ci assicurano la validità della proprietà.
Anche il massimo numero di bit nella magnitudo dei coefficienti del blocco è codificato mediante un tag tree, ma questo è usato in maniera diversa rispetto al caso precedente. Dato un blocco $B[m,n]$, sia $p^{\max}[m,n]$ il bit-plane più significativo nel quale qualche coefficiente del blocco è significativo. Il numero massimo di bit-planes disponibili, cioè il numero di bit nella magnitudo dei coefficienti, è dato da M_b, quindi deve essere $0 \le p^{\max}[m,n] < M_b$. Per rappresentare il valore di $p^{\max}[m,n]$ in maniera efficiente, si utilizza un tag tree, con il quale, però, viene rappresentato il numero dei bit-plane più significativi mancanti, cioè $q_1[m,n] = (M_b - 1) - p^{\max}[m,n]$. Per codificare l'informazione rilevante, la procedura $\mathbf{T}(m,n,t)$ è chiamata ripetutamente per t = 1, 2, …, finché $t > q_1[m,n]$. L'uso dei tag trees qui è diverso rispetto a quello fatto per la codifica dell'informazione di inclusione. In questo caso i valori da rappresentare non dipendono dall'esecuzione incrementale dell'algoritmo di ottimizzazione post-compressione, quindi non c'è la necessità di modificare i valori delle foglie durante la codifica del tag tree. Le figure seguenti mostrano un esempio di rappresentazione mediante tag tree dell'informazione $(M_b - 1) - p^{\max}[m,n]$.

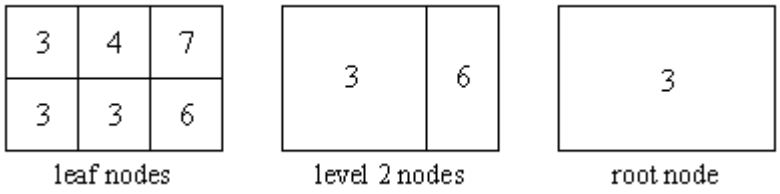

Figura 58: Tag tree generato per rappresentare il numero di bit-planes più significativi nulli.

La figura seguente mostra l'informazione codificata per rappresentare il numero di bit-planes più significativi nei quali nessun coefficiente del blocco è significativo. I bit codificati sono mostrati solo nel riquadro relativo al layer in cui il blocco corrispondente è incluso per la prima volta. La dicitura "coded" indica il fatto che l'informazione richiesta è stata già codificata in qualche layer precedente. Le caselle vuote indicano che il blocco non è stato ancora incluso in nessun layer.

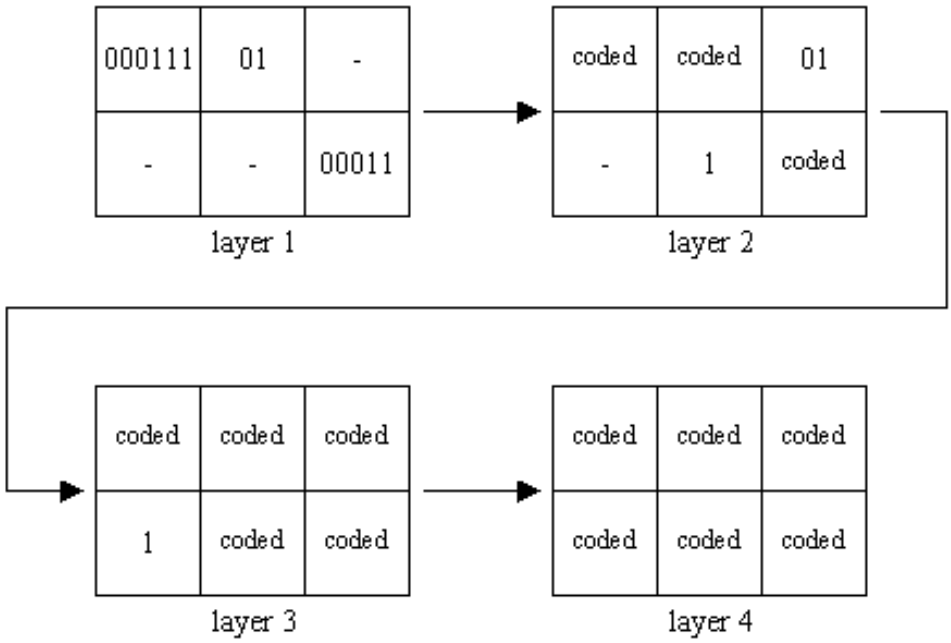

Figura 59: Rappresentazione mediante tag tree del numero di bit-planes più significativi nulli.

Si noti che in entrambi gli usi dei tag trees, l'algoritmo rappresenta efficientemente le informazioni per il sotto-insieme di blocchi che appaiono per la prima volta in un certo layer, facendo uso dell'informazione codificata in precedenza per i blocchi inclusi nei layer precedenti.
Per ogni blocco nel pacchetto bisogna trasmettere il nuovo punto di troncamento, indicando il numero di passi di codifica inclusi, n_i. Questa informazione è trasmessa usando un semplice codice a lunghezza variabile:

- se $n_i = 1$, si codifica un solo bit "0"
- se $n_i = 2$, si codificano i bit "10"
- se $3 \le n_i \le 5$, si trasmette un bit "1", seguito dalla rappresentazione a 2 bit di n_i
- se $6 \le n_i \le 36$, la codifica consiste di una serie di quattro bit "1", seguita dalla rappresentazione a 5 bit di n_i
- se $n_i \ge 37$, si trasmettono nove bit "1", seguiti dalla rappresentazione a 7 bit di n_i

Questo metodo permette di codificare fino a un massimo di 164 passi di codifica, che corrispondono a $M_b = 55$ bit-planes. Questa quantità dovrebbe essere sufficiente per ogni applicazione pratica. La tabella seguente riassume quanto detto.

Number of coding passes	Codeword in Packet Header
1	0
2	10
3	11 00
4	11 01
5	11 10
6 - 36	1111 00000 - 1111 11110
37 - 164	1111 11111 0000000 - 1111 11111 1111111

Tabella 16: Codeword codificata per indicare il numero di passi di codifi inclusi.

Per segnalare la lunghezza dei dati codificati inclusi nel pacchetto corrente per ogni blocco, si utilizza un codice composto da due parti. La prima serve a indicare la lunghezza in bit della seconda, che contiene una rappresentazione binaria del numero di bytes di dati codificati, presenti nel corpo del pacchetto per il blocco corrente. Il numero di bit, utilizzati per rappresentare la seconda parte del codice, è calcolato mediante la formula:

$$bits = \beta_i + \lfloor \log_2(p_i) \rfloor \qquad \text{(E.33)}$$

dove p_i è il numero di nuovi passi di codifica aggiunti e β_i è una variabile di stato del blocco. Il valore di β_i è inizializzato a 3. Il codice inizia con una sequenza di k bit "1" seguita da un bit "0". Questa è una rappresentazione unaria del numero di bit che devono essere aggiunti a quelli calcolati mediante la precedente formula, per calcolare la lunghezza della seconda parte del codice, che quindi è data di ($bits + k$) bit. Supponiamo che un blocco contribuisca a layers successivi con 6, 39 e 34 bytes e il numero di passi di codifica sia, rispettivamente, 1, 5 e 3. Supponiamo, inoltre, che il valore di β_i sia 3 in tutti e tre gli esempi. La codifica è riportata nella seguente tabella.

	length	coding passes	*bits*	*k*	code
layer 1	6	1	3	0	0 110
layer 2	39	5	5	1	10 100111
layer 3	34	3	4	2	110 1000010

Tabella 17: Esempio di codifica della dimensione (in bytes) del contributo di un blocco a diversi layer.

Concludiamo questa sezione con un esempio completo di codifica dell'intestazione di alcuni pacchetti, relativi all'area di sei blocchi di Figura 55. Nell'esempio, si considerano solo le informazioni relative a quest'area. I tag trees che rappresentano l'inclusione dei blocchi e il numero di bit-planes più significativi vuoti sono quelli nelle figure 51 e 53. Il numero di nuovi passi di codifica e il numero di nuovi bytes di codice sono mostrati nelle seguenti figure.

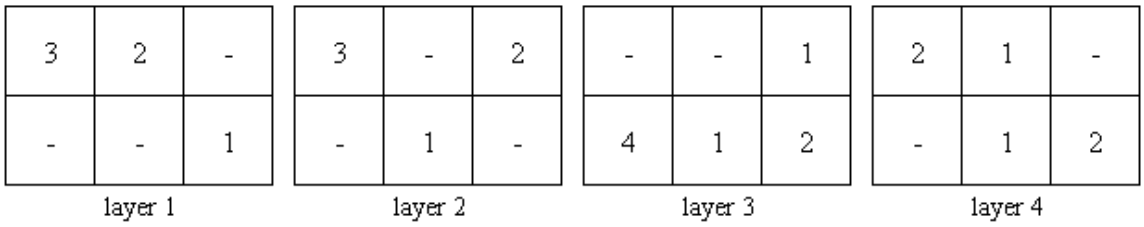

Figura 60: Numero di nuovi passi di codifica per ciascun blocco in ogni layer.

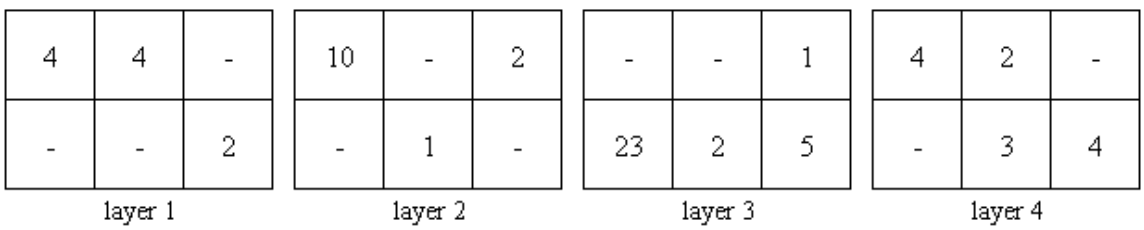

Figura 61: Lunghezza del bit-stream di ciascun blocco in ogni layer.

Le intestazioni dei pacchetti, relativi all'esempio in Figura 55, sono tabulate di seguito.

	bit-stream (in order)	meaning
	1	Packet non-zero in length
Block 0, 0	111	Block included (for the first time)
	000111	Block insignificant for 3 bit-planes
	1100	coding passes included: 3
	0	Length indicator is unchanged -> 0 + 3+log(3) = 4 bits
	0100	number of code bytes: 4
Block 0, 1	1	Block included (for the first time)
	01	Block insignificant for 4 bit-planes
	10	coding passes included: 2
	10	Length indicator increased by 1 -> 1 + 3+log(2) = 5 bits
	00100	number of code bytes: 4
0, 2	10	Block 0, 2 not yet included
1, 0	0	Block 1, 0 not yet included
1, 1	0	Block 1, 1 not yet included
Block 1, 2	1	Block included (for the first time)
	00011	Block insignificant for 6 bit-planes
	0	coding passes included: 1
	0	Length indicator is unchanged -> 0 + 3+log(1) = 3 bits
	010	number of code bytes: 2

Tabella 18: Parte dell'intestazione del pacchetto relativo al layer 1 della tile dell'esempio in Figura 55.

	bit-stream (in order)	meaning
	1	Packet non-zero in length
Block 0, 0	1	Block included (again)
	-	insignificant bit-planes already signalled in previous layers (1)
	1100	coding passes included: 3
	0	Length indicator is unchanged -> 0 + 3+log(3) = 4 bits
	1010	number of code bytes: 10
0, 1	0	Block not included in this layer (2)
Block 0, 2	1	Block included (for the first time)
	01	Block insignificant for 7 bit-planes
	10	coding passes included: 2
	0	Length indicator is unchanged -> 0 + 3+log(2) = 4 bits
	0010	number of code bytes: 2
1, 0	0	Block 1, 0 not yet included
Block 1, 1	1	Block included (for the first time)
	1	Block insignificant for 3 bit-planes
	0	coding passes included: 1
	0	Length indicator is unchanged -> 0 + 3+log(1) = 3 bits
	001	number of code bytes: 2
1, 2	0	Block 1, 2 not included in this layer (2)

Tabella 19: Parte dell'intestazione del pacchetto relativo al layer 2 della tile dell'esempio in Figura 55.

	bit-stream (in order)	meaning
	1	Packet non-zero in length
0, 0	0	Block not included in this layer (3)
0, 1	0	Block not included in this layer (3)
Block 0, 2	1	Block included (again)
	-	insignificant bit-planes already signalled in previous layers (2)
	0	coding passes included: 1
	0	Length indicator is unchanged -> 0 + 3+log(1) = 3 bits
	001	number of code bytes: 1
Block 1, 0	1	Block included (for the first time)
	1	Block insignificant for 3 bit-planes
	1101	coding passes included: 4
	0	Length indicator unchanged -> 0 + 3+log(4) = 5 bits
	10111	number of code bytes: 23
Block 1, 1	1	Block included (again)
	-	insignificant bit-planes already signalled in previous layers (2)
	0	coding passes included: 1
	0	Length indicator is unchanged -> 0 + 3+log(1) = 3 bits
	010	number of code bytes: 2
Block 1, 2	1	Block included (again)
	-	insignificant bit-planes already signalled in previous layers (1)
	10	coding passes included: 2
	0	Length indicator is unchanged -> 0 + 3+log(2) = 4 bits
	0101	number of code bytes: 5

Tabella 20: Parte dell'intestazione del pacchetto relativo al layer 3 della tile dell'esempio in Figura 55.

	bit-stream (in order)	meaning
	1	Packet non-zero in length
Block 0, 0	1	Block included (again)
	-	insignificant bit-planes already signalled in previous layers (1)
	10	coding passes included: 2
	0	Length indicator is unchanged -> 0 + 3+log(2) = 4 bits
	0100	number of code bytes: 4
Block 0, 1	1	Block included (for the first time)
	-	insignificant bit-planes already signalled in previous layers (1)
	0	coding passes included: 1
	0	Length indicator unchanged -> 0 + 3+log(1) = 3 bits
	010	number of code bytes: 2
0, 2	0	Block not included in this layer (4)
1, 0	0	Block not included in this layer (4)
Block 1, 1	1	Block included (again)
	-	insignificant bit-planes already signalled in previous layers (2)
	0	coding passes included: 1
	0	Length indicator is unchanged -> 0 + 3+log(1) = 3 bits
	011	number of code bytes: 3
Block 1, 2	1	Block included (again)
	-	insignificant bit-planes already signalled in previous layers (1)
	10	coding passes included: 2
	0	Length indicator is unchanged -> 0 + 3+log(2) = 4 bits
	0100	number of code bytes: 4

Tabella 21: Parte dell'intestazione del pacchetto relativo al layer 4 della tile dell'esempio in Figura 55.

Capitolo 6

Estensioni

In questo capitolo saranno presentate alcune delle caratteristiche aggiunte nella Part II dello standard JPEG2000.

6.1. Region Of Interest (ROI)

Una delle caratteristiche più importanti dello standard JPEG2000 è la possibilità di definire aree dell'immagine che saranno codificate con qualità più alta rispetto al resto dell'immagine. Queste aree sono dette *Region Of Interest* (ROI), mentre le parti rimanenti sono indicate come *background* (sfondo).
Un metodo per implementare facilmente la regione di interesse è quello di codificare i blocchi appartenenti alla ROI con qualità maggiore rispetto ai blocchi del background, affidandosi all'algoritmo di ottimizzazione di EBCOT. Allo stesso modo, l'ordine di progressione dei pacchetti può essere aggiustato in maniera tale da codificare prima i pacchetti contenenti informazioni sulla ROI. Con un'accurata distribuzione dei chunks ("pezzi" di blocchi) nei layers, questo metodo è efficace e, soprattutto, trasparente al decoder. Tuttavia, questo metodo impone che la geometria della ROI si adatti alla forma e dimensione dei blocchi. Questo problema è ancora maggiore per immagini di piccole dimensioni o quando si utilizzano blocchi grandi.
Per codificare regioni di interesse di forma arbitraria, è necessario segnalare al decoder quali sono i campioni dell'immagine che appartengono alla ROI, evitando l'overhead dovuto alla trasmissione esplicita di questa informazione. JPEG2000 risolve questo problema mediante il cosiddetto metodo *MaxShift*. Per comodità, il discorso che segue farà riferimento ad una sola componente dell'immagine, tuttavia il metodo può essere facilmente esteso a tutte le componenti.
Siano $x_c[i,j]$ i valori della componente *c* dei pixel nel dominio spaziale e R_c^r le ROI che vogliamo codificare per la stessa componente, con *r* = 1,2,…. Diciamo $S_b[i,j]$ l'insieme dei coefficienti wavelet della sotto-banda *b* che contribuiscono alla ricostruzione del pixel $x_{c(b)}[i,j]$, dove *c*(*b*) è la componente cui la sotto-banda *b* appartiene. A ogni regione assegniamo uno shift β_c^r.

Per rendere più semplice la trattazione, per il momento consideriamo il caso in cui dobbiamo codificare una sola ROI. Sia $M_{\max}$, il massimo numero di bit nella magnitudo dei coefficienti di ogni sotto-banda dell'immagine, che non appartengono alla ROI. L'idea è quella di shiftare i coefficienti della ROI a sinistra di $\beta_c^1 = M_{\max}$ bit, prima di quantizzarli. In questo modo, non solo saranno quantizzati meno grossolanamente, ma saranno trovati significativi nei primi passi di codifica, diversamente dai coefficienti del background che possono diventare significativi solo negli ultimi β_c^1 bit-planes. La Figura 62 mostra una sotto-banda con i coefficienti appartenenti alla ROI e nella Figura 63 gli stessi coefficienti shiftati di β_c^1 bit.

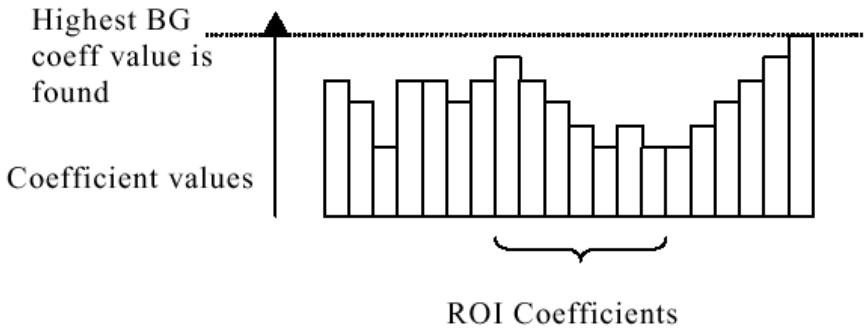

Figura 62: Esempio di una sotto-banda. I coefficienti della ROI sono evidenziati.

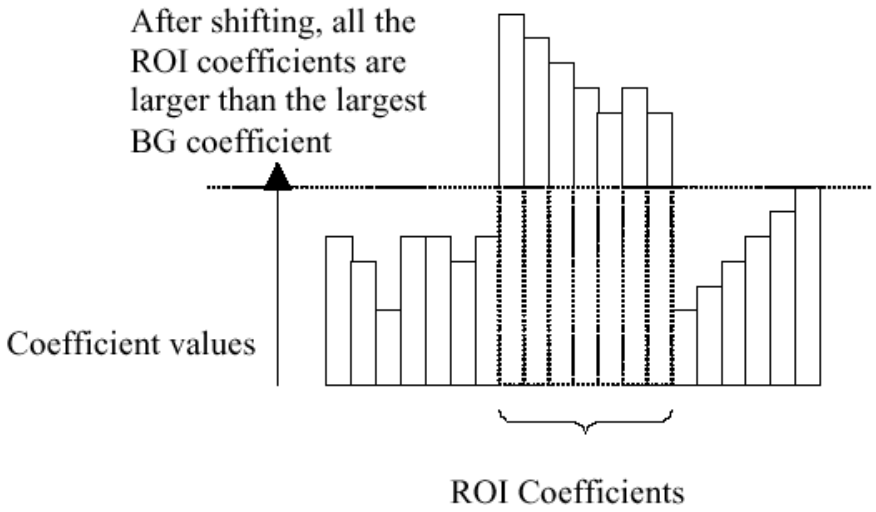

Figura 63: La stessa sotto-banda, dopo che i coefficienti della ROI sono stati shiftati.

La cosa interessante di questo tipo di codifica è che il decoder, per poter distinguere i coefficienti della ROI da quelli appartenenti al background, deve conoscere il valore di β_c^1. Infatti, se il decoder trova che un coefficiente diventa significativo prima degli ultimi β_c^1 bit-planes, allora quel coefficiente non può che appartenere alla ROI. A questo punto, per trovare il valore corretto del coefficiente basta shiftarlo a destra di β_c^1 bit dopo la fase di dequantizzazione. In questo modo siamo riusciti a codificare una ROI di forma arbitraria, segnalando al decoder solo il valore β_c^1.

Dovendo codificare più di una ROI, bisogna considerare il fatto che in uno stesso blocco possono trovarsi coefficienti appartenenti a ROI diverse, ciascuna con una propria priorità. Analogamente al caso della codifica di una sola ROI, bisogna shiftare i coefficienti di una ROI di un numero di bit sufficiente a superare il numero massimo di bit nella magnitudo dei coefficienti della ROI a priorità immediatamente più bassa, a loro volta shiftati. Cioè, per ogni ROI $R_c^{\,r}$, deve valere la relazione

$$\beta_c^{\,r} \geq \beta_c^{\,r-1} + M_{\max}^{r-1} \tag{E.34}$$

dove $M_{\max}^{r}$ è il massimo numero di bit nella magnitudo dei coefficienti appartenenti alla ROI $R_c^{\,r}$, $M_{\max}^{0} = M_{\max}$ e $\beta_c^{\,0} = 0$. Per poter decodificare la ROI, il decoder deve conoscere solo l'insieme degli shift $\left\{\beta_c^{\,1}, \beta_c^{\,2}, \ldots\right\}$.
Finora abbiamo spiegato come, data una ROI di forma arbitraria, sia possibile codificare i coefficienti wavelet che contribuiscono ai pixel della ROI, cioè quelli che appartengono agli insiemi $S_b[i,j]$. Tuttavia, rimane da chiarire come questi coefficienti siano associati ai campioni che costituiscono la regione di interesse. Data una ROI $R_c^{\,r}$, ciò che vogliamo fare è calcolare una *maschera* binaria del tipo:

$$M_c^{\,r}[m,n] = \begin{cases} 1 & \exists x_{c(b)}[i,j] \in R_c^{\,r} : q[m,n] \in \bigcup_b S_b[i,j] \\ 0 & \text{otherwise} \end{cases} \tag{E.35}$$

che indichi quali coefficienti wavelet sono utili alla ricostruzione dei pixel di $R_c^{\,r}$. Per costruire la maschera $M_c^{\,r}[m,n]$, si studia la trasformata wavelet inversa. Poiché i filtri usati nella trasformata sono separabili, possiamo concentrarci su un segnale mono-dimensionale. Allora, per ogni pixel $x[n]$ in $R_c^{\,r}$, sono inseriti nella maschera i coefficienti wavelet necessari per ricostruirlo. In particolare, per derivare la maschera per il filtro 5/3, le equazioni che descrivono la trasformata inversa sono:

$$x(2n) = L(n) - \frac{H(n-1) + H(n)}{4} \quad \text{e} \tag{E.36}$$

$$x(2n+1) = \frac{L(n) + L(n+1)}{2} + \frac{-H(n-1) + 6H(n) - H(n+1)}{8} \tag{E.37}$$

dove $L(\cdot)$ e $H(\cdot)$ sono i segnali delle sottobande inferiori, rispettivamente, a bassa e ad alta frequenza. Allora, i coefficienti necessari per ricostruire $x(2n)$ e $x(2n+1)$ sono, rispettivamente, $L(n)$, $H(n\text{-}1)$, $H(n)$ e $L(n)$, $L(n\text{+}1)$, $H(n\text{-}1)$, $H(n)$ e $H(n\text{+}1)$. Quindi, se $x(2n)$ e $x(2n+1)$ sono nella ROI, i precedenti coefficienti sono aggiunti alla maschera. Il procedimento deve essere iterato per tutti i coefficienti appena inseriti nella maschera, che sono decomposti nelle sotto-bande di livello più basso.

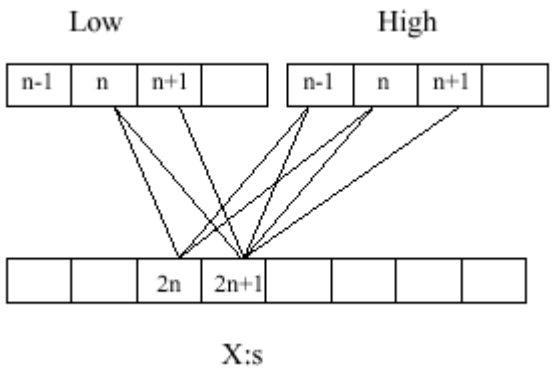

Figura 64: Trasformata 5/3 inversa. I coefficienti necessari per la ricostruzione di $x(2n)$ e $x(2n+1)$ sono, rispettivamente, $L(n)$, $H(n-1)$, H(n) e $L(n)$, $L(n+1)$, $H(n-1)$, H(n) , H(n+1).

Analogamente, studiamo la trasformata 9/7 inversa:

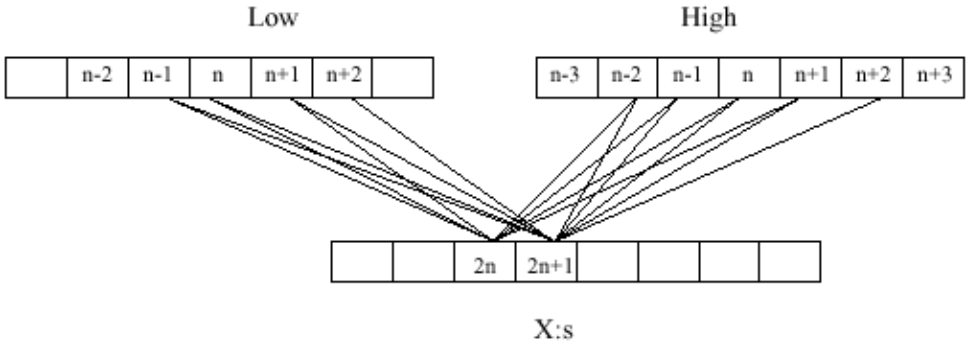

Figura 65: Trasformata 9/7 inversa. I coefficienti necessari per la ricostruzione di $x(2n)$ e $x(2n+1)$ sono, rispettivamente, $L(n-1)$, $L(n)$, $L(n+1)$, $L(n+1)$, $H(n-2)$, $H(n-1)$, $H(n)$, $H(n+1)$ e $L(n-1)$, $L(n)$, $L(n+1)$, $L(n+1)$, $L(n+2)$, $H(n-2)$, $H(n-1)$, $H(n)$, $H(n+1)$, $H(n+2)$.

Se i coefficienti $x(2n)$ e $x(2n+1)$ appartengono alla ROI, i coefficienti delle sotto-bande più basse che vanno inseriti nella maschera sono tutti i coefficienti nell'intervallo $L(n-1)$, $L(n+2)$ e quelli nell'intervallo $H(n-2)$, $L(n+2)$.

I vantaggi del metodo MaxShift possono essere riassunti come segue:
- Supporto per ROI di forma arbitraria con complessità aggiuntiva minima
- Non devono essere trasmesse informazioni sulla forma della ROI, né sulla maschera
- Supporto per la progressione sia per qualità che per risoluzione

Nonostante i notevoli vantaggi, tuttavia, il metodo risulta poco pratico per codificare più di una ROI a causa dell'impatto che la "distanza" tra i coefficienti β_c^r ha sulla precisione dell'implementazione (implementazioni tipiche hanno una precisione di 32 bit e i coefficienti wavelet solitamente hanno una precisione di 16 bit).

6.2. Codifica di Immagini Indicizzate

Un'immagine digitale è rappresentata mediante una matrice di punti colorati. Tipicamente ogni punto è costituito da tre componenti di colore, ciascuna di 8 bit, quindi il numero di colori rappresentabili è 2^{24}. Nella pratica, questo numero è molto maggiore del numero di pixel contenuti in molte immagini. Allora, per immagini a pochi colori può essere conveniente mappare in una tabella i colori usati e rappresentare ciascun pixel come un indice agli elementi della tabella. L'immagine può essere allora descritta da una coppia (I, Map), dove *I* è una matrice di indici e *Map* è una tabella, detta *palette*, che mappa gli indici di *I* nello spazio dei colori.
Per comprimere immagini di questo tipo, la codifica entropica risulta spesso inefficiente. Infatti, mentre le immagini naturali mostrano una certa ridondanza tra i pixel vicini, ciò non accade nelle immagini indicizzate perché i pixel sono semplicemente degli indici agli elementi di una tabella. Per avere un'intuizione del perché questo è vero, si pensi al modo in cui EBCOT assegna i contesti ai coefficienti. Per mettere in evidenza la ridondanza tra coefficienti vicini, ad ogni coefficiente è associato un contesto in base alla significanza dei coefficienti che appartengono al suo intorno. In immagini naturali è molto probabile che le variazioni tra pixel vicini siano graduali, quindi questo schema è efficiente. Per immagini indicizzate, invece, non possiamo dire nulla sul valore degli indici.
L'idea è, allora, quella di disporre gli indici in maniera tale che le variazioni tra indici vicini siano il più possibile graduali. Questo è l'approccio utilizzato da diversi algoritmi di compressione, tra i quali GIF e PNG.
Data un'immagine indicizzata rappresentata dalla coppia (I, M), vogliamo cambiare l'ordine degli indici nella tabella *M* in modo tale da ottenere una rappresentazione (I', M') che permetta di raggiungere un fattore di compressione maggiore. Si noti che, purché le associazioni nella palette cambino di conseguenza, un'immagine può essere reindicizzata senza perdita di informazione.
Dato un criterio di reindicizzazione, la soluzione ottimale può essere trovata cercando in maniera esaustiva tra tutte le possibili soluzioni. Tuttavia, il numero di permutazioni di *N* elementi è $N!$, quindi questa ricerca non è pratica. Poiché il problema è intrattabile, può essere risolto solo in maniera approssimata. L'obiettivo è, allora, quello di individuare un criterio di reindicizzazione appropriato e trovare un'euristica che fornisca una soluzione sub-ottimale in maniera efficiente.
Di seguito sarà presentato l'algoritmo di reindicizzazione suggerito dal comitato JPEG.
Data un'immagine indicizzata, siano S_0, S_1, …, S_{M-1} i colori usati, disposti secondo l'ordine in cui appaiono nella palette. Dobbiamo creare una nuova tabella che mappi ciascun colore in uno ed uno solo degli indici nell'intervallo $[0, M-1]$. Il metodo che descriveremo assegna un nuovo indice ad un colore alla volta, secondo un'euristica di tipo *greedy*[7]. Ogni assegnamento è ottimizzato in base alle statistiche collezionate in una fase di pre-processing sull'immagine iniziale. Sia, allora, $C(S_i, S_j)$ il numero di occorrenze in cui il colore S_i è adiacente al colore S_j. Evidentemente, $C(S_i, S_j) = C(S_j, S_i)$. L'osservazione qui è che è vantaggioso assegnare indici vicini a colori che sono frequentemente vicini tra loro. Questo tende a ridurre la differenza globale tra i pixel adiacenti, producendo una matrice di indici con variazioni più graduali.

L'algoritmo che descrive lo schema di reindicizzazione è il seguente:

[7] Un'euristica greedy è una sequenza di scelte localmente ottimali. Si noti che, come in questo caso, una sequenza di scelte localmente ottimali non porta necessariamente ad una soluzione globalmente ottimale.

1. Si calcolino le occorrenze $C(S_i, S_j)$ per ogni coppia di colori S_i ed S_i.
2. Si calcolino le occorrenze cumulative $C_i = \sum_{j=0, j\neq i}^{M-1} C(S_i, S_j)$ per ogni colore S_i.
3. Si trovi il colore $S_{\max}$ che ha il massimo numero di occorrenze, ovvero il valore C_i più grande. Siano $L_0 = S_{\max}$ e $P = \{L_0\}$. P conterrà la sequenza ordinata dei colori. Sia N la dimensione di P e si ponga $N = 1$. I nuovi colori possono essere inseriti solo alle estremità di P. Ogni colore inserito in P è marcato come assegnato.
4. Per ogni colore non assegnato S_i, si calcoli la funzione potenziale $D_i = \sum_{j=0}^{N-1} w_{(N,j)} \cdot C(S_i, L_j)$, dove i pesi $w_{(N,j)}$ controllano l'impatto delle occorrenze $C(S_i, L_j)$ sulla funzione potenziale e dipendono dalla distanza tra l'estremo corrente di P e la posizione in P di L_j. La funzione potenziale è calcolata per ogni colore non assegnato, da ciascuno degli estremi di P. Siano $S_{L_{\max}}$ ed $S_{R_{\max}}$ i colori che hanno il potenziale D_i maggiore, rispettivamente, da sinistra e da destra.
5. Si ponga $L_N = \max\{S_{L_{\max}}, S_{R_{\max}}\}$ e lo si inserisca nella corrispondente posizione di P. Si incrementi il contatore N dei colori in P.
6. Se $N < M$ si torni al passo (4)
7. Si assegnino gli indici 0, 1, …, M-1 ai colori di P, seguendo l'ordine spaziale, da sinistra verso destra o da destra verso sinistra. L'immagine reindicizzata è ottenuta sostituendo l'indice iniziale i con il nuovo indice assegnato al colore S_i.

Poiché ciò che importa è la differenza tra indici vicini e non il valore degli indici stessi, la scelta iniziale $S_{\max}$ ha senso perché tale colore è quello più frequentemente vicino agli altri. La decisione di quale colore inserire in P ad ogni passo è fatta in base alla funzione potenziale D_i, che misura il numero di volte in cui pixel del colore S_i sono vicini ai pixel di colori già in P. Il valore dei pesi è $w_{(N,j)} = \log_2\left(1 + \frac{1}{d_{(N,j)}}\right)$, dove $d_{(N,j)}$ indica la distanza del colore L_j dall'estremo corrente di P.

Per ulteriori informazioni sull'argomento il lettore può riferirsi a [13].

Capitolo 7

Analisi Qualitativa

In questo capitolo sarà presentata un'analisi del sistema di compressione JPEG2000, dal punto di vista qualitativo, volta a dare un'idea intuitiva delle caratteristiche e delle potenzialità del nuovo standard. Sarà fornito, inoltre, un confronto con lo standard JPEG. Alla fine del capitolo sarà svolta una breve analisi sulla complessità computazionale del sistema di compressione.

7.1. Scalabilità e Ordine di Progressione

L'eliminazione delle ridondanze nell'informazione inevitabilmente induce una forma di dipendenza tra pixel vicini. Questo rende l'immagine compressa poco flessibile e incapace di adattarsi alle esigenze del nuovo mercato. Si pensi, ad esempio, all'internet browsing. La grande varietà di applicazioni impone che l'immagine compressa sia capace di adattarsi a condizioni di visualizzazione notevolmente diverse. Una pagina web potrà essere consultata mediante un PC, con un monitor a 17", o dal display a bassa risoluzione di un telefono cellulare. La prima applicazione richiede un'immagine ad alta risoluzione e di buona qualità, mentre la seconda non necessita né dell'una né dell'altra. Entrambe, inoltre, richiedono la minima occupazione possibile del canale di comunicazione. È in applicazioni come questa che la scalabilità acquista valore. In questo modo, infatti, è possibile trasmettere solo le informazioni utili a ricostruire l'immagine con risoluzione e qualità volute. Inoltre, il browsing interattivo impone la capacità di accedere a porzioni arbitrarie dell'immagine.
JPEG2000 è in grado di rispondere a tutte queste esigenze, offrendo fattori di compressione elevati, pur conservando una notevole scalabilità del bit-stream compresso. Inoltre, in fase di codifica è possibile stabilire l'ordine di progressione. Nell'esempio precedente, i dati compressi potrebbero essere ordinati in modo tale che il decoder possa visualizzare l'immagine progressivamente, dalla risoluzione più bassa a quella più alta, ricostruendo ogni livello alla qualità massima.

7.1.1. Progressione per Risoluzione

Di seguito è mostrato un esempio di progressione per risoluzione.

Figura 66: Esempio di progressione per risoluzione.

Si noti che, poiché la scalabilità per risoluzione è una caratteristica che deriva dalla decomposizione wavelet, il fattore di incremento nella risoluzione è 2 per ciascuna dimensione.

7.1.2. Progressione per Qualità

Le immagini che seguono mostrano un esempio di progressione per qualità.

Figura 67: Immagine decodificata a 0.0625 bpp.

Figura 68: Immagine decodificata a 0.125 bpp

Figura 69: Immagine decodificata a 0.25 bpp

Figura 70: Immagine decodificata a 0.5 bpp

7.2. Region Of Interest (ROI)

La ricostruzione di porzioni dell'immagine con qualità maggiore rispetto al resto dell'immagine è utile in molte applicazioni. Si consideri, per esempio, uno scenario in cui

la regione di interesse deve essere codificata senza perdite, mentre il resto può essere compresso in maniera *visually lossless* (senza perdite nel senso della qualità percepita). Questo permette un notevole risparmio in termini di memorizzazione e trasmissione dell'immagine compressa. Le immagini che seguono mostrano un esempio di codifica con *Region Of Interest* (Regione di interesse). Le immagini sono compresse con un fattore, rispettivamente, di 0.0625, 0.125, 0.25, 0.5, 1.0 e 2.0 bpp.

Figura 71: Esempio di codifica della ROI. Le immagini sono codificate, rispettivamente, a 0.0625, 0.125, 0.25, 0.5, 1.0, 2.0 bpp.

Si noti che la codifica della ROI è fatta in maniera progressiva e, come per il resto dell'immagine, anche la ROI è scalabile per risoluzione e qualità.

7.3. Confronto tra JPEG2000 e JPEG

Riportiamo, infine, un esempio di un'immagine codificata con JPEG baseline e JPEG2000. A parità di bit-rate, la qualità dell'immagine codificata con JPEG2000 è visibilmente più alta della qualità di quella compressa con JPEG.

Figura 72: Entrambe le immagini sono compresse a 0.125 bpp. L'immagine a sinistra è compressa con JPEG baseline, quella a destra con JPEG2000.

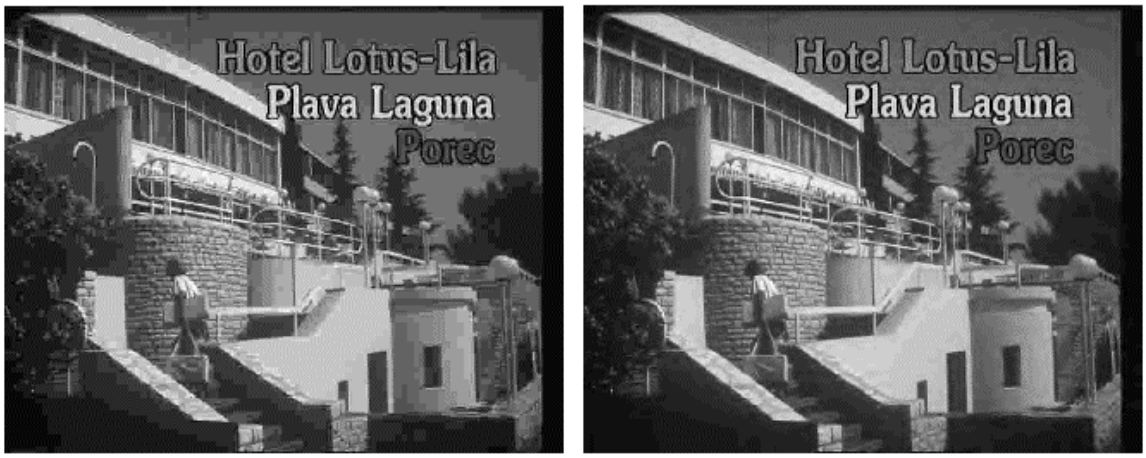

Figura 73: Entrambe le immagini sono compresse a 0.25 bpp. L'immagine a sinistra è compressa con JPEG baseline, quella a destra con JPEG2000.

7.4. Complessità Computazionale

L'obiettivo di questa sezione è quello di fornire un'analisi della complessità computazionale dell'algoritmo di compressione JPEG2000. Lo scopo dell'analisi svolta è determinare in numero massimo di operazioni elementari (o accessi alla memoria) eseguite dall'algoritmo, piuttosto che calcolarne un valore medio.

Poiché le dimensioni delle immagini da codificare possono essere estremamente variabili, perché la valutazione abbia senso, il numero di operazioni eseguite deve essere calcolato per ogni pixel. La compressione è fatta in maniera indipendente per ogni componente di colore dell'immagine, quindi la discussione seguente si concentrerà sui coefficienti di una sola componente. Inoltre, essendo l'algoritmo composto da una sequenza di fasi distinte, è conveniente effettuare il calcolo separatamente per ciascuna fase.

7.4.1. Trasformazione dello Spazio dei Colori

Questo è l'unico passo che necessita di essere eseguito tenendo conto di tutte le componenti di colore, quindi il numero di operazioni eseguite in questo passo, diversamente che negli altri passi, sarà calcolato per tutte le componenti.
Il numero di operazioni effettuate può essere facilmente dedotto dalle equazioni E.1 ed E.2, per la trasformazione irreversibile, e dalle equazioni E.3 ed E.4 per quella reversibile.
La trasformazione reversibile consiste del prodotto tra una matrice 3x3 e una 3x1, quindi il numero di operazioni è 5 (tre prodotti e due somme) per ogni riga della matrice. In totale, la trasformazione effettua 15 operazioni in virgola mobile per ogni pixel.
La trasformazione reversibile, invece, consiste di quattro operazioni (due somme, un prodotto e un'operazione di arrotondamento) per la luminanza e una ciascuno per le altre componenti; in totale, sei operazioni intere per pixel.
Lo stesso numero di operazioni è svolto anche per la trasformazione inversa.

7.4.2. Trasformata Wavelet

Per calcolare il numero di operazioni effettuate durante questa fase, dobbiamo distinguere i filtri wavelet usati. Considereremo solo i filtri ammessi dalla Part I dello standard, implementati mediante il lifting scheme.
Il calcolo è immediatamente derivato delle formule E.5 ed E.6 per il filtro di analisi 5/3, E.7 ed E.8 per il filtro di sintesi 5/3 e E.9 ed E.10 per il filtro 9/7, rispettivamente, di analisi e sintesi.
Il numero di operazioni eseguite usando il filtro di analisi 5/3 è di:

- 4 (2 somme, 1 moltiplicazione e 1 operazione di arrotondamento) per i coefficienti dispari
- 5 (3 somme, 1 moltiplicazione e 1 arrotondamento) per i coefficienti pari

In totale, nove operazioni per ogni coppia di coefficienti, cioè 4.5 operazioni intere per ogni coefficiente per ogni componente di colore. Lo stesso numero di operazioni si ottiene per il filtro di sintesi.
Il numero di operazioni eseguite per ogni passo per il filtro di analisi 9/7 è di:

- step 1: 3 (2 somme e 1 prodotto)
- step 2: 3 (2 somme e 1 prodotto)
- step 3: 3 (2 somme e 1 prodotto)
- step 4: 3 (2 somme e 1 prodotto)
- step 5: 1 (1 prodotto)
- step 6: 1 (1 prodotto)

In totale, 14 operazioni per ogni coppia di coefficienti, cioè sette operazioni in virgola mobile per ogni coefficiente per ogni componente di colore. Lo stesso numero di operazioni si ottiene per il filtro di sintesi.

Entrambi i numeri calcolati devono essere moltiplicati per il numero di componenti di colore nell'immagine.

7.4.3. Quantizzazione

Il numero di operazioni eseguite nella fase di quantizzazione può essere facilmente dedotto dalla formula E.12. Per ogni coefficiente è eseguita una somma e un arrotondamento. In totale, dunque, due operazioni per pixel, per ogni componente di colore.
La fase di dequantizzazione è simmetrica a quella di dequantizzazione ed esegue lo stesso numero di operazioni per pixel.

7.4.4. EBCOT Tier 1

Nella codifica dei blocchi, nel tier 1 di EBCOT, non è eseguita alcuna operazione, ma vengono raccolte delle statistiche sui coefficienti di ciascun blocco. Anziché calcolare il numero di operazioni, allora, valuteremo il numero di accessi alle variabili di stato usate durante la scansione. La decodifica agisce allo stesso modo della codifica, quindi codifica e decodifica non saranno distinte ai fini di quest'analisi.
Ogni bit del coefficiente $q_i[m,n]$ è scandito tre volte, una per ogni passo di codifica. Per ogni scansione si testano le due variabili di stato $\sigma_i[m,n]$ e $\eta_i[m,n]$; in totale, sei operazioni per ogni bit. Inoltre, indipendentemente dal tipo di passo di codifica, per stabilire il contesto del coefficiente, devono essere letti i valori delle variabili $\sigma_i[i,j]$ dei coefficienti nell'intorno di $q_i[m,n]$, che sono otto. Infine, durante la scansione, è studiato l'intorno di ogni bit, quindi devono essere aggiunti altri otto accessi alle variabili $\sigma_i[i,j]$.
In totale, quindi, sono eseguiti al più 22 accessi alla memoria per ogni bit nel coefficiente, cioè circa 350 accessi per coefficiente, nel caso peggiore (16 bit per coefficiente). Questo numero va moltiplicato per il numero di componenti di colore dell'immagine.

7.4.5. EBCOT Tier 2

La fase più impegnative del tier 2 è il calcolo dei punti di troncamento ottimali. Quest'operazione è eseguita al livello dei blocchi, quindi il numero di operazioni per pixel è ammortizzato sul numero di coefficienti nel blocco.
Analizziamo l'algoritmo A.1. Per ogni esecuzione, il numero dei cicli eseguiti è al più uguale al numero dei punti di troncamento legali, cioè $3 \cdot M_b - 2$, dove M_b è il numero di bit-planes nel blocco *b*. Allora, poiché $M_b \leq 16$, il numero di cicli per ogni chiamata è al più 46. Il numero totale di iterazioni è pari al numero di chiamate all'algoritmo per il numero iterazioni per chiamata. Evidentemente, dato il numero relativamente elevato di coefficienti in un blocco, il numero di operazioni elementari eseguite in questa fase è trascurabile.

7.4.6. Codifica Aritmetica

Per calcolare il numero di operazioni eseguite durante questa fase, considereremo la codifica di un singolo bit. Il calcolo è facilmente esteso a tutto il coefficiente. La decodifica è simmetrica alla codifica, perciò il numero di operazioni eseguite è uguale a quello della codifica.
Quando un nuovo bit è passato al MQ-Coder, esso sarà codificato come MPS o LPS. In entrambi i casi sono necessarie due somme per aggiornare i registri C ed A. In più, l'operazione di rinormalizzazione è eseguita ogni volta che viene codificato un simbolo LPS e in alcuni casi di codifica del MPS. Il costo della rinormalizzazione è pari al numero di bit di cui A deve essere shiftato per rientrare nel range legale, per il numero di operazioni eseguite per ciascun shift. Per ogni ciclo effettuiamo due shift (A e C) e una somma (CT), più due operazioni di test. Il numero massimo di shift del registro A e, quindi, di cicli è al più 15. Quindi una rinormalizzazione costa al più tre operazioni e due test per 15 cicli, cioè 45 operazioni elementari e 30 test.
Rimane da calcolare il peso delle rinormalizzazioni sulla codifica di un coefficiente. Indichiamo con P_R la probabilità che un simbolo in input provochi una rinormalizzazione e siano P_{MPS} e $P_{R_{MPS}}$, rispettivamente, la probabilità del simbolo più probabile e la probabilità di una rinormalizzazione MPS. Analogamente, P_{LPS} e $P_{R_{LMPS}}$ sono la probabilità del simbolo meno probabile e la probabilità di una rinormalizzazione LPS. Allora, la probabilità di una rinormalizzazione è data da $P_R = P_{R_{MPS}} \cdot P_{MPS} + P_{R_{LPS}} \cdot P_{LPS}$.
Per l'algoritmo A.3, la rinormalizzazione viene eseguita dopo la codifica di un LPS, quindi $P_{R_{LMPS}} = 1$. Valutiamo, ora, la probabilità di una rinormalizzazione MPS. In [12], si trova che tale probabilità vale:

$$P_{R_{MPS}} = (1-q)\frac{Qe}{0.75} \quad \text{(E.38)}$$

dove $q = P_{LPS}$ e Qe è la stima corrente della probabilità P_{LPS}. Per semplificare, possiamo supporre che la stima sia corretta, cioè $Qe \cong q$. La precedente equazione, allora, diventa:

$$P_{R_{MPS}} = \frac{(1-q)\cdot q}{0.75} \quad \text{(E. 39)}$$

Il massimo di questa funzione è in corrispondenza di $q = 1/2$, che equivale all'incertezza massima (i simboli sono ritenuti equiprobabili). In definitiva, nella situazione di massima incertezza, gli assegnamenti di probabilità sono i seguenti:

$$\begin{cases} P_{LPS} = 1/2 \\ P_{R_{LPS}} = 1 \\ P_{MPS} = 1/2 \\ P_{R_{MPS}} = 1/3 \end{cases} \quad \text{(E. 40)}$$

quindi la probabilità che un simbolo in input causi una rinormalizzazione è, nel caso peggiore, $P_R = P_{R_{MPS}} \cdot P_{MPS} + P_{R_{LPS}} \cdot P_{LPS} = \frac{1}{3}\cdot\frac{1}{2} + 1\cdot\frac{1}{2} = \frac{2}{3}$.
Il costo di una rinormalizzazione è, allora, di al più $45 \cdot 2/3 = 30$ operazioni e $30 \cdot 2/3 = 20$ test per bit. Poiché nel caso peggiore il numero di bit per coefficiente è di 16, il costo totale

per coefficiente è di al più 480 operazioni e 320 test per coefficiente. Per ricavare il numero di operazioni per pixel, bisogna moltiplicare per il numero di componenti di colore nell'immagine.

La tabella seguente mostra il costo per pixel di ciascuna operazione di codifica.

stage		operations/pixel	memory accesses/pixel	type of operations
Colour Transform	reversible	6	-	integer
	irreversible	15	-	floating point
Wavelet Transform	5/3 analysis	13,5	-	integer
	5/3 syntesis	13,5	-	integer
	9/7 analysis	21	-	floating point
	9/7 syntesis	21	-	floating point
Quantization	quantization	6	-	floating point
	dequantization	6	-	floating point
EBCOT	tier 1	-	< 1050	-
	tier 2	-	neglegible	-
Arithmetic Coding	coding	< 1440	< 960	integer
	decoding	< 1440	< 960	integer

Tabella 22: numero di operazioni/accessi alla memoria per pixel nei passi di codifica di JPEG2000.

Come risulta evidente dalla precedente tabella, il grosso della complessità si concentra nella fase di codifica aritmetica. C'è da dire, comunque, che la precedente valutazione della complessità di quest'ultima fase prende in considerazione un caso estremo che nella pratica non dovrebbe mai verificarsi. In realtà, il caso medio è molto più efficiente di quello considerato perché la previsione del MPS è accurata e raramente si verificano situazioni di incertezza come quella descritta. Esiste una certa proporzionalità tra numero di bit di codice emessi e complessità della codifica aritmetica. Infatti, i bit sono messi in output dalla procedura BYTEOUT che è invocata durante la rinormalizzazione. Quindi, minore è la lunghezza della codifica, minore è il numero di rinormalizzazioni occorse e, di conseguenza, il numero di operazioni eseguite. Poiché i risultati sperimentali mostrano che JPEG2000 raggiunge buoni fattori di compressione, dobbiamo aspettarci che il numero di operazioni elementari eseguite dall'algoritmo di codifica aritmetica sia molto inferiore a quello indicato nella precedente tabella.

7.5. Conclusioni

Lo standard JPEG2000, oltre a fornire un alto fattore di compressione, genera un bit-stream dotato di caratteristiche che lo rendono molto flessibile e capace di rispondere alle nuove esigenze del mercato per una notevole varietà di applicazioni, che vanno dalla telefonia mobile alla navigazione interattiva in remoto. La sua architettura aperta, inoltre, permette di ottimizzare tanto processo di codifica, quanto la qualità delle immagini compresse.
Alcune delle caratteristiche salienti dello standard JPEG2000 sono riassunte qui di seguito:

- fattore di compressione alto e buona qualità a bassi bit-rate
- scalabilità per risoluzione e accesso random
- trasmissione progressiva per qualità o risoluzione
- controllo del bit-rate (cioè: controllo della dimensione finale dei dati compressi)

- supporto per compressione lossy e lossless
- supporto per immagini a toni continui e grafiche
- regione di interesse
- error resilience (protezione da errori di trasmissione)
- richiesta ridotta di memoria

La seguente tabella mette a confronto le caratteristiche offerte da JPEG2000 con quelle di JPEG.

	JPEG	JPEG2000
image quality at high bit-rates	excellent	excellent
image quality at low bit-rates	poor	good
natural imagery	yes	yes
graphics imagery	no	yes
bi-level imagery	no	yes
resolution scalability	no	yes
SNR scalability	no	yes
random access	yes	yes
bit-rate control	no	yes
lossy compression	yes	yes
lossless compression	no	yes
error resilience	no	yes
ROI	no	yes

Tabella 23: Confronto delle caratteristiche di JPEG e JPEG2000.

Capitolo 8

Ottimizzazione Content-Dependent

In questo capitolo vogliamo presentare alcune idee per cercare di migliorare, a parità di bit-rate, la qualità visiva di un'immagine compressa mediante un encoder JPEG2000. L'algoritmo descritto è completamente trasparente per il decoder, quindi le immagini elaborate sono conformi allo standard.

8.1. Introduzione

Nel capitolo precedente si è discusso di come l'encoder sia capace di limitare la dimensione dei dati compressi ad un certo bit-rate $R_{\max}$. Dopo la compressione entropica dei blocchi, il tier 2 di EBCOT, in base alle informazioni sui punti di troncamento legali fornite dal tier 1, "seleziona" i pezzi di blocchi compressi da inserire nel bit-stream finale, cercando di ottimizzare la qualità dell'immagine ricostruita, limitatamente al bit-rate $R_{\max}$.
Lo standard, tuttavia, non specifica a quale livello deve essere fatta l'ottimizzazione bitrate/distorsione per trovare i punti di troncamento ottimali. Cioè, non è chiaro se l'algoritmo di ottimizzazione deve essere eseguito dopo la codifica di tutti i blocchi di una tile o dell'intera immagine. La formula che descrive il problema di ottimizzazione vincolato, che deve essere risolto da EBCOT, è la seguente:

$$D_\lambda + \lambda \cdot R_\lambda = \sum_i \left(D_i^{n_i^\lambda} + R_i^{n_i^\lambda} \right) \qquad \text{(E.41)}$$

Il problema è quello di trovare il parametro λ e i punti di troncamento ottimali n_i^λ che minimizzino la sommatoria E.41, soggetti al vincolo $R_\lambda \le R_{\max}$. Ciò che lo standard non specifica è l'insieme dei blocchi B_i che contribuiscono alla somma. Infatti, se l'ottimizzazione è fatta su tutta l'immagine, nella sommatoria appaiono i contributi di tutti i blocchi dell'immagine e il parametro λ sarà unico per tutte le tiles. Se, invece, l'algoritmo di ottimizzazione è eseguito separatamente per ogni tile, solo i blocchi appartenenti alla tile corrente contribuiranno alla sommatoria e il parametro λ potrà variare da una tile all'altra. In quest'ultimo caso la precedente sommatoria può essere riscritta nel modo seguente:

$$D_{\lambda_T} + \lambda_T \cdot R_{\lambda_T} = \sum_i \left(D_i^{n_i^{\lambda_T}} + R_i^{n_i^{\lambda_T}} \right) \quad \text{(E.42)}$$

dove T è l'indice della tile corrente.
Nella pratica, il software di riferimento esegue l'ottimizzazione sull'intera immagine; tuttavia, alcuni esperimenti che abbiamo condotto su un insieme di immagini di test, compresse utilizzando entrambi i metodi di ottimizzazione, hanno dimostrato che non c'è una sostanziale differenza tra i due metodi. Il SNR (Signal/Noise Ratio) delle immagini compresse con un metodo, calcolato rispetto all'immagine originale, differisce di pochi centesimi di decibel da quello calcolato per le immagini compresse con l'altro metodo. Inoltre, nessuno dei due metodi prevale sull'altro.

8.2. L'algoritmo

L'obiettivo è ottimizzare la distribuzione dei bit disponibili tra le tiles, in modo tale da ottenere un'immagine che esibisca una maggiore qualità visiva rispetto a quella compressa direttamente mediante JPEG2000. L'idea è quella di ridurre il numero di bit utilizzati nella codifica di zone in cui la distorsione è meno visibile, a vantaggio di aree più ricche di dettagli visibili. Tutto questo deve essere fatto, ovviamente, lasciando inalterata la dimensione finale dei dati compressi.
Indichiamo con R_I e R_T, rispettivamente, il fattore di compressione globale e il fattore di compressione per una certa tile T, misurati in bit per pixel (bpp). Allora deve valere la seguente uguaglianza:

$$R_I = \frac{\sum_T R_T}{N_T} \quad \text{(E.43)}$$

dove N_T è il numero di tile nell'immagine. Cioè, R_I è la media dei fattori di compressione delle singole tile. Il nostro obiettivo è quello di ridurre la distorsione globale, pesando diversamente i contributi di ogni tile, avendo cura, però, di mantenere costante R_I.
Nel software di riferimento di JPEG2000, l'encoder fa l'assunzione $R_I = R_T$ e lascia ad EBCOT il compito di distribuire i bit tra le tile, in maniera opportuna. Tuttavia, in base alle osservazioni fatte sopra, i risultati ottenuti mediante l'ottimizzazione al livello di tile sono uguali a quelli ottenuti mediante un'ottimizzazione globale. Questa osservazione suggerisce che la distribuzione dei bit alle tile avviene in maniera pressoché omogenea. Ciò può portare ad uno scenario in cui le tiles più ricche di particolari, dovendosi uniformare al bit-rate massimo fissato, sono molto distorte, mentre bit importanti vengono "sprecati" in zone in cui il rumore è meno visibile. Questa tendenza è ancora più accentuata a bassi bit-rate.
Per ogni tile T, poniamo $R_T = R_I \cdot w_T$, dove w_T è un peso positivo, centrato intorno a 1. L'equazione E.43, allora, può essere riscritta nel modo seguente:

$$R_I = \frac{\sum_T (R_I \cdot w_T)}{N_T} = R_I \frac{\sum_T w_T}{N_T} \quad \text{(E.44)}$$

Il problema di ottimizzare la distribuzione dei bit alle tiles diventa, dunque, quello di trovare i pesi w_T . Dall'equazione precedente segue un primo vincolo sui valori dei pesi:

$$\frac{\sum_T w_T}{N_T} = 1 \qquad \text{(E.45)}$$

Ciò riflette la condizione che abbiamo posto all'inizio, per la quale il bit-rate globale deve rimanere invariato. Un'altra ovvia condizione è che i pesi siano tutti positivi.
Per calcolare correttamente i pesi delle tiles, è necessario saperne riconoscere il contenuto. Per questo il nostro algoritmo avrà bisogno di un modulo, che chiameremo "*riconoscitore*", che si occupi di classificare le tiles secondo determinati criteri visivi.
Una questione importante, qui, è decidere a quale livello nella catena di codifica di JPEG2000 deve operare il riconoscitore. Poiché i valori R_T sono utilizzati durante la fase di ottimizzazione bit-rate/distorsione, l'algoritmo deve essere eseguito prima del tier 2 di EBCOT. Deve essere eseguito anche prima del tier 1 perché dopo la codifica entropica è molto difficile riconoscere i dati codificati. Infine, è conveniente catalogare le tiles prima della quantizzazione dei coefficienti perché, in presenza di una forte quantizzazione, potremmo perdere informazioni importanti. Allora la scelta si riduce a decidere se eseguire l'algoritmo prima o dopo la trasformata wavelet. La nostra prima scelta è stata quella di eseguirlo prima della trasformata, ma è nostra intenzione valutare anche l'altra possibilità.
Dopo aver riconosciuto la classe di appartenenza di una tile *T*, siamo in condizione di assegnare il peso w_T . Abbiamo bisogno, dunque, di un criterio per associare, a ciascuna delle classi del riconoscitore, un valore che indichi se una tile, appartenente ad una data classe, sia disposta a cedere bit o, viceversa, ne richiede in più. Facendo uso di questo criterio, associamo ad ogni tile il peso corretto. Tuttavia, dobbiamo stare attenti a non violare il vincolo E.45. Inoltre, dobbiamo evitare di assegnare pesi troppo bassi o troppo alti. Infatti, pesi troppo bassi creerebbero rumore in zone a basso contenuto di informazione, il che peggiorerebbe la qualità dell'immagine, anziché migliorarla; pesi troppo alti potrebbero non dare il guadagno sperato e, ancora una volta, potremmo ottenere un'immagine di qualità più bassa. La seguente figura schematizza i passi dell'algoritmo.

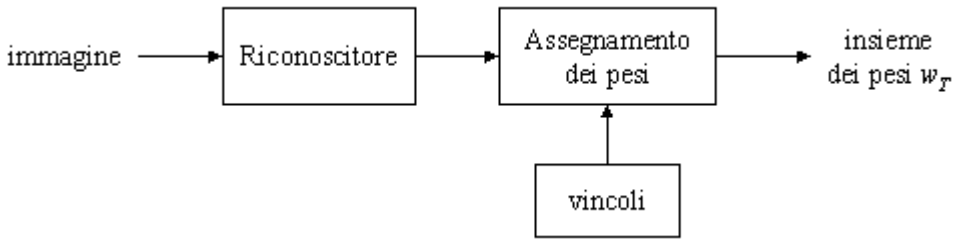

Figura 74: Passi dell'algoritmo per il calcolo dei pesi w_T .

L'immagine in input è analizzata dal riconoscitore che classifica le tiles in funzione di determinati criteri visivi. Il blocco successivo, in base a questa classificazione e ad un certo insieme di vincoli, stabilisce l'assegnamento dei pesi w_T .
Analizzeremo queste fasi separatamente.

8.2.1. Il Riconoscitore

Il riconoscitore lavora sulla componente di luminanza dell'immagine originale, prima della trasformata wavelet e subito dopo la trasformazione dello spazio dei colori.
La scelta di usare solo la luminanza ai fini della classificazione deriva dal fatto che le componenti di crominanza hanno, rispetto alla componente di luminanza, un valore percettivamente minore, nel senso che le variazioni di luminanza sono percepite in maniera più accurata rispetto alle variazioni di crominanza. Dunque, l'uso della sola componente di luminanza non pregiudica la bontà della classificazione, perché tale componente contiene tutte le informazioni che ci sono utili per la classificazione, e inoltre ci permette di ridurre notevolmente il numero dei calcoli necessari.
La classificazione delle tiles è fatta in due passi consecutivi. Inizialmente l'immagine è filtrata con un opportuno filtro di *edge enhancing*, per metterne in evidenza i contorni. Poi, l'immagine filtrata è suddivisa in tiles e per ogni tile si ricavano due misure: valore medio e varianza. Il valor medio fornisce una misura di quanta superficie della tile è occupata da lati. La varianza, invece, indica quanto netti sono i lati presenti nella tile. La combinazione di queste due misure permette di classificare le tile come "*plain*", "*edge*" o "*texture*".
Fissate le soglie T_M e T_V, rispettivamente per la media e la varianza, una semplice classificazione è mostrata nella seguente tabella.

Threshold		class
mean	variance	
<	<	plain
<	>	edge
>	>	texture
>	<	-

Tabella 24: Classificazione delle tiles.

Una difficoltà nella classificazione è la dimensione delle tiles. Classificare blocchi piccoli, per esempio 8x8, è relativamente semplice perché il contenuto è chiaramente riconoscibile. Insiemi di campioni grandi come le tiles di JPEG2000 (64x64 o più), invece, in generale contengono elementi diversi, il che rende difficile decidere a quale classe assegnare una tile che non ha un contenuto immediatamente riconoscibile. Un'idea potrebbe essere quella di rendere più granulare la classificazione, ammettendo un maggior numero di soglie, con un conseguente aumento delle classi nella tabella precedente.
La classificazione è ancora oggetto di studio. Alcuni risultati sono presentati più avanti.

8.2.2. Assegnamento dei Pesi

I pesi sono assegnati utilizzando un sistema basato su "*crediti*". Le zone che possono cedere bit, aggiungono crediti in un deposito. Le zone che necessitano di crediti, li chiedono al deposito. A ciascuna classe è assegnato un numero intero di crediti, distribuiti secondo la tabella seguente.

class	credits
plain	-2
edge	2
texture	-1
other	0

Tabella 25: Assegnamento dei crediti alle classi.

I valori negativi indicano che la tile è disposta a versare i crediti corrispondenti nel deposito, mentre i valori positivi indicano che la tile ha necessità di crediti. I valori nella precedente tabella sono scelti in maniera totalmente arbitraria e richiedono un'accurata fase di verifica.
Un problema che sorge nell'assegnamento dei crediti è il modo in cui devono essere trattate le zone texturizzate. Mentre per le zone omogenee e per le zone con lati marcati l'assegnamento dei crediti è piuttosto naturale, le zone con tessiture presentano una peculiarità che rende difficile stabilire il valore corretto di crediti da assegnare. Infatti, il contenuto di informazione di queste zone è sicuramente molto alto perché sono caratterizzate da una notevole variabilità; tuttavia, a causa di un effetto di mascheramento visivo (*visual masking*), in queste zone è più difficile percepire il rumore. La figura seguente mostra due immagini a cui è stato aggiunto rumore casuale. La stessa quantità di rumore è stata aggiunta ad un'area rettangolare, in alto nell'immagine a sinistra e in basso nell'immagine a destra. Le due immagini hanno PSNR identico, ma il rumore è molto più visibile nel cielo (zona omogenea) piuttosto che nelle rocce (tessitura).

Figura 75: Due immagini con PSNR identico (31.7db). La stessa quantità di rumore casuale è stata aggiunta ad un'area rettangolare, in alto nell'immagine a sinistra e in basso nell'immagine a destra. Il rumore è molto più visibile nel cielo piuttosto che nelle rocce a causa del forte effetto di mascheramento, di cui il PSNR non tiene conto.

Per questo motivo alle tile che contengono grosse zone texturizzate è stato assegnato un peso negativo. Tuttavia, questa scelta si rivela poco adeguata quando trattiamo con tiles texturizzate che contengano, però, lati netti. Questo è un problema di difficile soluzione e richiede notevole attenzione sia in fase di assegnamento dei pesi che, soprattutto, nella fase di riconoscimento.
Come appare evidente dalla Tabella 25, l'assegnamento dei crediti è strettamente legato alla classificazione delle tile, quindi un miglioramento nell'algoritmo di classificazione dovrebbe portare a migliori risultati in questa fase.

Dopo avere assegnato i crediti, dobbiamo assegnare i pesi alle tiles. Ciò è fatto fissando un valore Δ_{out}, che rappresenta il valore di un credito in uscita. Il valore dei crediti in entrata è calcolato mediante l'equazione:

$$\Delta_{in} = \frac{\Delta_{out} \cdot N_{out}}{N_{in}} \qquad \text{(E.46)}$$

dove N_{in} e N_{out} sono, rispettivamente, il numero di crediti in entrata e il numero di crediti in uscita. Detto c_T il numero di crediti associato alla tile *T*, ricavato dalla Tabella 25, i pesi sono assegnati mediante le formule seguenti:

$$w_T = 1 + \Delta_{out} \cdot c_T \quad \text{per } c_T > 0 \qquad \text{(E.47a)}$$
$$w_T = 1 + \Delta_{in} \cdot c_T \quad \text{per } c_T < 0 \qquad \text{(E.47b)}$$
$$w_T = 1 \quad \text{per } c_T = 0 \qquad \text{(E.47c)}$$

Si noti che, con gli assegnamenti precedenti, i pesi w_T rispettano sempre il vincolo E.45. Infatti, per la E.46 si ha $\Delta_{in} \cdot N_{in} = \Delta_{out} \cdot N_{out}$.
Tuttavia, esiste ancora la possibilità che Δ_{out} e Δ_{in} siano troppo grandi. Come detto sopra, variazioni eccessive di questi due parametri possono portare ad una perdita di qualità, piuttosto che a un guadagno. È necessario, allora, limitarne il valore.
Questo può essere fatto imponendo che $w_T \in [m_T, M_T]$, con $0 < m_T \leq 1 \leq M_T$. Allora Δ_{out} e Δ_{in} devono essere scelti in maniera tale da rispettare i vincoli:

$$m_T < -\min(c_T) \cdot \Delta_{in} \quad \text{e} \qquad \text{(E.48a)}$$
$$\max(c_T) \cdot \Delta_{out} < M_T \qquad \text{(E.48b)}$$

In questo modo, scegliendo accuratamente gli estremi dell'intervallo $[m_T, M_T]$, si evita che la qualità dell'immagine possa peggiorare. Il valore di m_T e M_T è calcolato, per ogni tile *T*, in funzione del target bit-rate e del contenuto di informazione di *T*.

8.2.3. Complessità Computazionale

Per le osservazioni fatte in precedenza, anche qui il numero di operazioni eseguite dall'algoritmo esaminato sarà calcolato in funzione di ogni pixel. Tuttavia, l'algoritmo presentato lavora con due insiemi diversi di dati. In particolare, le fasi di filtraggio e la raccolta delle statistiche sui campioni di ciascuna tile (media e varianza) sono eseguite sui campioni dell'immagine, mentre le successive fasi di classificazione e assegnazione dei pesi lavorano sulle statistiche raccolte, il cui numero è proporzionale a quello delle tiles. Il numero di pixel per ogni tile, considerando una tile di dimensioni tipiche (64x64) è di 4096. Allora, l'insieme di dati elaborati durante le fasi di classificazione e assegnazione dei pesi è di oltre tre ordini di grandezza più piccolo rispetto a quello utilizzato durante le prime fasi. In altre parole, queste due ultime fasi hanno un peso trascurabile nella complessità totale dell'algoritmo.
Poiché la tale complessità si concentra nelle prima fasi dell'algoritmo, l'analisi che segue si concentrerà su queste.

Calcoliamo innanzi tutto il costo del calcolo di media e varianza. Se diciamo μ la media e σ^2 la varianza e il numero di pixel è *N*, dalle formule:

$$\mu = \frac{\sum_{i=1}^{N} x_i}{N} \qquad \text{(E.49)}$$

$$\sigma^2 = \frac{\sum_{i=1}^{N} (x_i - \mu)^2}{N} \qquad \text{(E.50)}$$

risulta evidente che sono necessarie poche operazioni per ogni pixel. La media richiede $N-1$ somme e una divisione, quindi solamente una operazione per pixel. Per il calcolo della varianza, per ogni elemento della sommatoria dobbiamo effettuare una differenza e un prodotto (quadrato), poi dobbiamo calcolare la media dei valori calcolati ($N-1$ somme e una divisione); in totale, tre operazioni per pixel. Tuttavia, la E.50 può essere riscritta nella seguente forma:

$$\sigma^2 = \frac{\sum_{i=1}^{N} x_i^{\,2}}{N} - \mu^2 \qquad \text{(E.51)}$$

che evita di ricalcolare la media μ. Quest'ultima costa solo due operazioni per pixel, uno per l'operazione di quadrato per ogni elemento della sommatoria, l'altro per motivi analoghi a quelli illustrati per il calcolo della media. Allora, in totale la raccolta delle statistiche sui campioni delle tile costa tre operazioni per pixel.
Il costo del filtraggio dell'immagine dipende dal tipo di filtro usato. In particolare, due caratteristiche sono rilevanti ai fini del calcolo della complessità computazionale:

1. se il filtro è lineare o non-lineare
2. dimensione del supporto del filtro.

Se il filtro è lineare, il numero di operazioni per pixel dipende direttamente dalla dimensione del kernel. In particolare, se il kernel ha dimensioni $M \times N$, il numero di operazioni eseguite per la convoluzione sono $2 \cdot (M \times N) - 1$ ($M \times N$ prodotti e $(M \times N) - 1$ somme). Dimensioni tipiche dei kernel di convoluzione sono 3x3, 5x5, 7x7, quindi il numero di operazioni eseguite per un filtraggio lineare può variare tra 17 e 97.
Se il filtro applicato è non-lineare, il numero di operazioni eseguite è molto variabile. In particolare, se tale filtro è composizione di più filtri lineari, il numero di operazioni è dato dalle operazioni eseguite per l'applicazione dei singoli filtri lineari, più il costo per la composizione. Evidentemente, per filtri complessi il costo può crescere notevolmente.
La tabella seguente mostra il costo di ogni passo dell'algoritmo presentato.

stage	operations/pixel
filtering	17-97 +
mean	1
variance	2
classification	neglegible
weighting	neglegible

Tabella 26: Numero di operazioni per pixel eseguite in ogni fase dell'algoritmo proposto.

Come risulta evidente dalla tabella precedente, il grosso del costo si concentra nella fase di filtraggio. L'obiettivo, dunque, è quello di trovare un filtro efficace, che sia il più possibile semplice. Con il filtro attualmente in fase di studio, il numero totale di operazioni per pixel eseguite dall'algoritmo presentato è di circa 50.
Infine, un'ultima osservazione riguarda la quantità di memoria richiesta per la computazione. Poiché la valutazione dei pesi può essere fatta solo dopo aver raccolto le statistiche necessarie per ogni tile, l'algoritmo presentato non può essere eseguito on-line, cioè non può operare sull'input man mano che arriva. Questo impone che tutta l'immagine sia bufferizzata prima della seconda fase dell'algoritmo. Il sistema proposto, dunque, non è adeguato per applicazioni che hanno a disposizione risorse di memoria limitate.

8.3. Risultati Sperimentali

8.3.1. Il Software Usato

Per effettuare gli esperimenti, è stato usato il software di riferimento JJ2000, un'implementazione di encoder e decoder JPEG2000 scritta in Java. Il software è stato opportunamente modificato per eseguire l'ottimizzazione Bitrate/Distorsione separatamente per ogni tile T, con target bit-rate R_T .

8.3.2. Risultati

In questa sezione sono riportati i risultati di alcuni degli esperimenti condotti per verificare la validità del sistema presentato.
Le figure seguenti mostrano due esempi dell'output del riconoscitore su immagini due classiche immagini di test. La Figura 78, invece, mostra il miglioramento nella qualità visiva dell'immagine compressa con i pesi calcolati dall'algoritmo presentato, rispetto a quella compressa direttamente con JPEG2000.

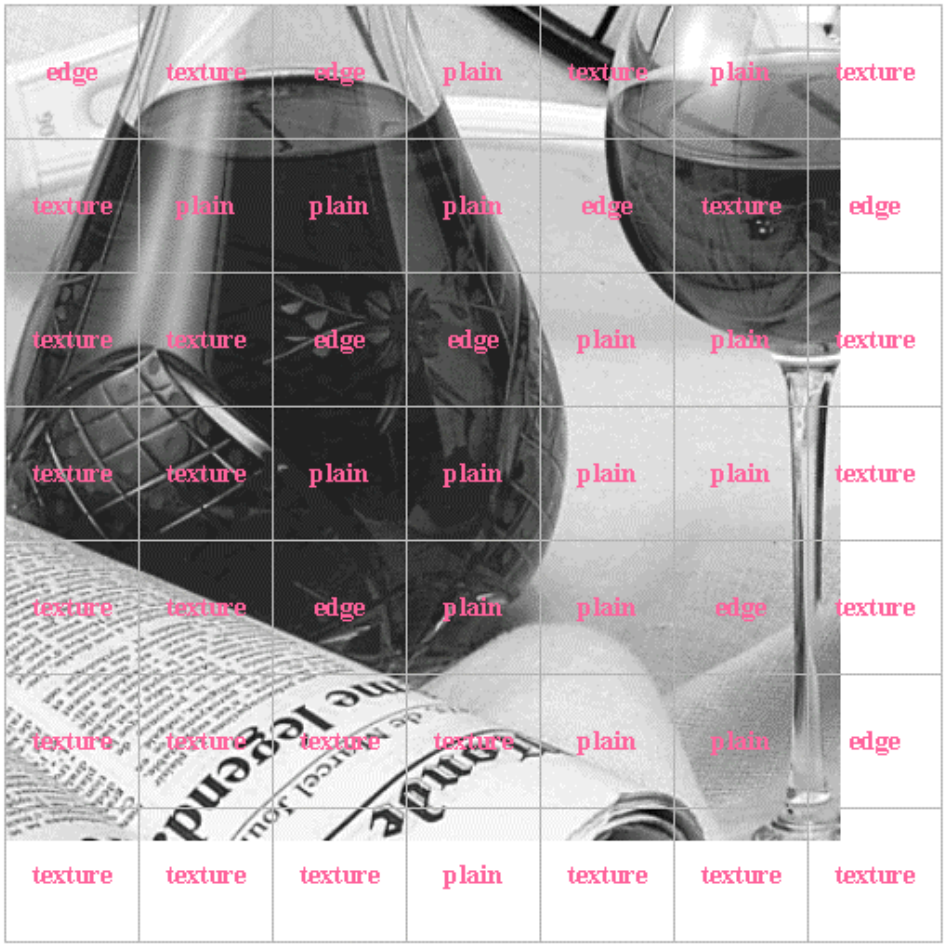

Figura 76: Classificazione delle tiles di una porzione dell'immagine di test "bike".

Figura 77: Classificazione delle tiles dell'immagine di test "lena".

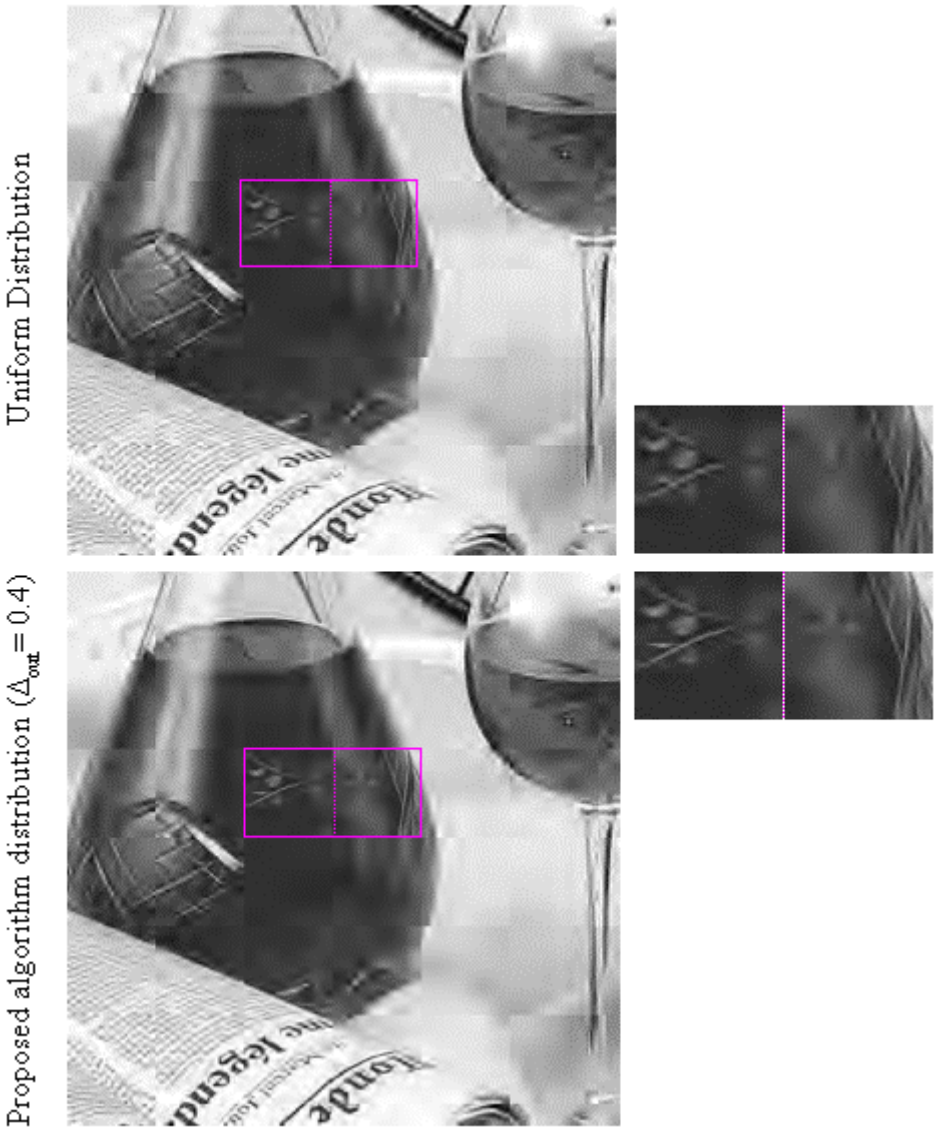

Figura 78: Esempio di miglioramento nella qualità visiva di due tiles dell'immagine di test "bike", compressa a 0.5 bpp.

Di seguito riportiamo tre tabelle comparative, che mostrano il confronto tra la compressione con pesi uniformi e quella effettuata utilizzando i pesi ottenuti mediante l'analisi presentata. La qualità è valutata utilizzando il PSNR (Peak Signal/Noise Ratio) dell'immagine codificata, rispetto all'originale. La formula per il calcolo del PSNR è la seguente:

$$\mathrm{PSNR} = 10 \cdot \log_{10}\left(\frac{M^2}{\mathrm{MSE}}\right) \qquad \text{(E.52)}$$

dove M è il massimo valore che un pixel può assumere e MSE è l'errore quadratico medio (*mean square error*), espresso come:

$$\mathrm{MSE} = \frac{\sum_{i=1}^{N} (x_i - \bar{x}_i)^2}{N} \qquad \text{(E.53)}$$

con x_i valore di un pixel dell'immagine originale, $\bar{x}_i$ valore del pixel corrispondente nell'immagine elaborata ed N numero di campioni nell'immagine.
Per ogni tabella sono presentati i risultati di tre test, effettuati su otto immagini a toni di grigio, con bit-rate di 0.25, 0.5 e 1.0 bit/pixel. I valori di Δ_{out} sono: 0.2 nella prima tabella, 0.4 nella seconda e 1.0 nella terza.
A causa degli effetti visivi cui si accennava nella sezione 8.2.2, un aumento del PSNR non sempre corrisponde ad un miglioramento nella qualità percepita. Questo è il destino di tutte le misure oggettive che non tengano conto delle caratteristiche del sistema percettivo dell'osservatore. Per ovviare a questo problema, nel futuro affiancheremo al PSNR una misura studiata per catturare le peculiarità del sistema visivo umano.

Tabella 27: Risultati dell'algoritmo presentato. $\Delta_{out} = 0.2$

	image	PSNR uniform	PSNR our	difference	%	
0.25 bpp	baboon	22.6900	22.7139	0.0240	0.1055	OK
	bike1	26.1698	26.9815	0.8117	3.0084	OK
	bike2	24.2237	24.3296	0.1060	0.4357	OK
	bike3	25.0412	25.1490	0.1078	0.4287	OK
	lena	25.8908	26.0542	0.1635	0.6274	OK
	woman1	29.1506	29.1506	0.0000	0.0000	same
	woman2	19.3333	19.3370	0.0038	0.0196	OK
	woman3	31.6798	31.7100	0.0302	0.0951	OK
0.5 bpp	baboon	24.6094	24.6490	0.0396	0.1607	OK
	bike1	32.9062	33.7235	0.8173	2.4235	OK
	bike2	28.0985	28.3504	0.2520	0.8888	OK
	bike3	28.9080	29.0531	0.1452	0.4997	OK
	lena	30.2149	30.3290	0.1141	0.3762	OK
	woman1	31.7665	31.7665	0.0000	0.0000	same
	woman2	21.7922	21.8008	0.0086	0.0395	OK
	woman3	35.6982	35.7175	0.0193	0.0542	OK
1.0 bpp	baboon	27.5260	27.2003	-0.3257	-1.1975	-
	bike1	38.9264	39.6608	0.7344	1.8516	OK
	bike2	33.3640	33.6535	0.2896	0.8604	OK
	bike3	33.9892	34.1465	0.1573	0.4606	OK
	lena	35.6460	36.0292	0.3831	1.0634	OK
	woman1	35.6312	35.6312	0.0000	0.0000	same
	woman2	25.7984	25.8154	0.0170	0.0658	OK
	woman3	40.0137	40.0011	-0.0126	-0.0315	-
	mean	29.5445	29.7064	0.1619	0.5098	OK
	max			0.8173	3.0084	
	min			-0.3257	-1.1975	

Tabella 28: Risultati dell'algoritmo presentato. $\Delta_{out} = 0.4$

	image	PSNR uniform	PSNR our	difference	%	
0.25 bpp	baboon	22.6900	22.7139	0.0240	0.1055	OK
	bike1	26.1698	26.9815	0.8117	3.0084	OK
	bike2	24.2237	24.3296	0.1060	0.4357	OK
	bike3	25.0412	25.1490	0.1078	0.4287	OK
	lena	25.8908	26.0542	0.1635	0.6274	OK
	woman1	29.1506	29.1506	0.0000	0.0000	same
	woman2	19.3333	19.3370	0.0038	0.0196	OK
	woman3	31.6798	31.7100	0.0302	0.0951	OK
0.5 bpp	baboon	24.6094	24.6490	0.0396	0.1607	OK
	bike1	32.9062	33.7235	0.8173	2.4235	OK
	bike2	28.0985	28.3504	0.2520	0.8888	OK
	bike3	28.9080	29.0531	0.1452	0.4997	OK
	lena	30.2149	30.3290	0.1141	0.3762	OK
	woman1	31.7665	31.7665	0.0000	0.0000	same
	woman2	21.7922	21.8008	0.0086	0.0395	OK
	woman3	35.6982	35.7175	0.0193	0.0542	OK
1.0 bpp	baboon	27.5260	27.2297	-0.2964	-1.0884	-
	bike1	38.9264	40.0056	1.0791	2.6974	OK
	bike2	33.3640	33.8402	0.4762	1.4071	OK
	bike3	33.9892	34.2054	0.2162	0.6321	OK
	lena	35.6460	36.0596	0.4135	1.1468	OK
	woman1	35.6312	35.6312	0.0000	0.0000	same
	woman2	25.7984	25.8031	0.0047	0.0182	OK
	woman3	40.0137	40.0040	-0.0097	-0.0242	-
	mean	29.5445	29.7331	0.1886	0.5813	OK
	max			1.0791	3.0084	
	min			-0.2964	-1.0884	

Tabella 29: Risultati dell'algoritmo presentato. $\Delta_{out} = 1.0$

	image	PSNR uniform	PSNR our	difference	%	
0.25 bpp	baboon	22,6900	22,6505	-0,0395	-0,1743	-
	bike1	26,1698	27,4626	1,2928	4,7075	OK
	bike2	24,2237	24,5230	0,2993	1,2206	OK
	bike3	25,0412	25,2339	0,1927	0,7638	OK
	lena	25,8908	25,9254	0,0346	0,1334	OK
	woman1	29,1506	29,1506	0,0000	0,0000	same
	woman2	19,3333	19,3113	-0,0219	-0,1135	-
	woman3	31,6798	31,6872	0,0074	0,0234	OK
0.5 bpp	baboon	24,6094	24,5335	-0,0759	-0,3094	-
	bike1	32,9062	34,2557	1,3495	3,9395	OK
	bike2	/	/	/	/	/
	bike3	28,9080	29,0615	0,1536	0,5285	OK
	lena	30,2149	30,1321	-0,0828	-0,2749	-
	woman1	31,7665	31,7665	0,0000	0,0000	same
	woman2	21,7922	21,7440	-0,0482	-0,2218	-
	woman3	35,6982	35,6373	-0,0609	-0,1709	-
1.0 bpp	baboon	27,5260	27,0551	-0,4709	-1,7406	-
	bike1	38,9264	40,4163	1,4899	3,6863	OK
	bike2	33,3640	33,8578	0,4938	1,4586	OK
	bike3	33,9892	34,2116	0,2224	0,6501	OK
	lena	35,6460	35,7189	0,0729	0,2040	OK
	woman1	35,6312	35,6312	0,0000	0,0000	same
	woman2	25,7984	25,6951	-0,1033	-0,4020	-
	woman3	40,0137	39,9335	-0,0803	-0,2010	-
	mean	29,6074	29,8085	0,2011	0,5960	OK
	max			1,4899	4,7075	
	min			-0,4709	-1,7406	

Riferimenti Bibliografici

[1] D. Taubman, "High performance scalable image compression with EBCOT", IEEE *Trans. Image Proc*. July, 2000.

[2] "JPEG2000 Image Coding System", ISO/IEC FCD 15444-1. March, 2000

[3] "JPEG2000 Verification Model 7.0", ISO/IEC JTC 1/SC 29/WG1 N1684. April, 2000.

[4] D. Taubman, E.Ordentlich, M.Weinberger, G.Seroussi, I.Ueno, F.Ono, "Embedded block coding in JPEG2000", *Proc*. IEEE *Int. Conf. Image Proc*.

[5] D. Taubman, E.Ordentlich, M.Weinberger, G.Seroussi, "Embedded block coding in JPEG2000", *Proc*. IEEE *Int. Conf. Image Proc*.

[6] H. Printz, "Tutorial on arithmetic coding", February, 1994.

[7] I. Daubechies, W.Sweldens, "Factoring wavelet into lifting steps", *Technical report*, *Bell Laboratiories*, *Lucent Technologies*, 1996

[8] A. Ortega, K. Ramchandran, "Rate-Distortion Methods for image and video compression"" IEEE *Signal Proc. Magazine*, November 1998

[9] W. Zeng, S. Daly, S. Lei, "Visual optimization tools in JPEG2000", Proc. IEEE *Int. Conf. Image Proc.*, September 2000

[10] S. Daly, W. Zeng, J. Li, S. Lei, "Visual masking in wavelet compression for JPEG2000", SPIE/IS&T *Electronic Imaging, Image and Video Comm. And Proc.*, January 2000

[11] M. J. Nadenau, J. Reichel, M. Kunt, "Wavelet-based Color Image Compression: Exploiting the Contrast Sensitivity Function", 2001

[12] W. B. Pennebaker and J. L. Mitchell, "JPEG: Still Image Compression Standard", *Van Nostrand Reinhold*, NY, 1993.

[13] W. Zeng, J. Li, S. Lei, "An efficient color re-indexing scheme for palette-based compression", Proc. IEEE *Int. Conf. Image Proc.*, September 2000