



Università degli Studi di Napoli “Parthenope”

FACOLTÀ DI SCIENZE E TECNOLOGIE
Corso di Laurea in Informatica e Tecnologie Multimediali

CODIFICA E COMPRESSIONE DI DATI MULTIMEDIALI
PROGETTO D'ESAME

La Codifica Aritmetica

Candidati:

Daniele Iervolino

Matricola LI/1065

Antonio Forghieri

Matricola LI/927

Professore:

Giuliana Ramella

SOMMARIO

La codifica aritmetica é una metodo di compressione lossless. Normalmente, una stringa di caratteri come ad esempio le parole "salve" é rappresentata mediante un numero fisso di bit per carattere, come nel codice ASCII. Come per la codifica di Huffman, la codifica aritmetica crea codici entropici di lunghezza variabile, che converte simboli piú frequentemente utilizzati con un numero inferiore di bit e quelli di rado utilizzati con piú bit, con l'obiettivo di utilizzare un minor numero di bit in totale. A differenza di altre tecniche di codifica entropica che separano componente per componete i simboli in input e sostituiscono ogni simbolo con una parola di codice, la codifica aritmetica codifica l'intero messaggio in un unico numero, una frazione n tale che $(0.0 \leq n < 1.0)$.

ABSTRACT

Arithmetic coding is a method for lossless data compression. Normally, a string of characters such as the words "hello there" is represented using a fixed number of bits per character, as in the ASCII code. Like Huffman coding, arithmetic coding is a form of variable-length entropy encoding that converts a string into another form that represents frequently used characters using fewer bits and infrequently used characters using more bits, with the goal of using fewer bits in total. As opposed to other entropy encoding techniques that separate the input message into its component symbols and replace each symbol with a code word, arithmetic coding encodes the entire message into a single number, a fraction n where $(0.0 \leq n < 1.0)$.

INDICE

1	Codifica e Compressione	1
1.1	I concetti chiave	1
1.2	Teoria dell'informazione	3
2	La Codifica Aritmetica	6
2.1	Introduzione alla codifica aritmetica	6
2.2	Modelli probabilistici	7
2.3	Codifica aritmetica "classica"	9
2.4	Codifica aritmetica incrementale	12
3	La codifica aritmetica e JPEG2000	14
3.1	La compressione aritmetica in JPEG2000	14
3.2	Processo di codifica	16
3.3	Processo di decodifica	20
4	Implementazione	21
4.1	La classe Encoder	22
4.1.1	Il metodo readFreq	22
4.1.2	Il metodo model	22
4.1.3	Il metodo updateMap	23
4.1.4	Il metodo writeOutFile	23
4.1.5	Il metodo encoding	23
4.2	La classe Decoder	24
4.2.1	Il metodo readInfo	24
4.2.2	Il metodo decoding	24
4.3	Esempi di codifica	25
4.4	Conclusioni	28
A	Codice sorgente	29
A.1	Encoder.h	29
A.2	Encoder.cpp	30
A.3	Decoder.h	33
A.4	Decoder.cpp	34
A.5	color.h	36
A.6	main.cpp	37

BIBLIOGRAFIA	38
--------------	----

INDICE ANALITICO	39
------------------	----

ELENCO DELLE FIGURE

Figura 1	Esempio di codifica aritmetica classica.	9
Figura 2	Esempio di codifica aritmetica incrementale.	12
Figura 3	MQ-coder. Suddivisione dell'intervallo in LPS e MPS.	15
Figura 4	MQ-coder. Interfaccia con EBCOT.	17
Figura 5	MQ-decoder. Interfaccia con EBCOT.	20
Figura 6	Codifica del file 1prova.txt.	25
Figura 7	Decodifica del file 1prova.ac.	25
Figura 8	Codifica del file 2prova.txt.	26
Figura 9	Codifica del file 3prova.txt.	26
Figura 10	Errore: Il file 4prova.txt non esiste.	27
Figura 11	Errore: Manca il file di output	27
Figura 12	Errore: Manca il file di output	27

ELENCO DELLE TABELLE

Tabella 1	MQ-coder. Registri interni.	16
Tabella 2	MQ-coder. Registri interni.	19
Tabella 3	La classe Encoder	22
Tabella 4	La classe Decoder	24

CODIFICA E COMPRESSIONE

Prima di entrare nei dettagli della codifica aritmetica, in questo capitolo vengono introdotti i concetti di *codifica e compressione*

1.1 I CONCETTI CHIAVE

I concetti di codifica e compressione sono strettamente correlati. La codifica è un procedimento per ottenere una rappresentazione in simboli di codice delle informazioni di nostro interesse. E' molto utilizzata in informatica per la rappresentazione dei dati in memoria e in crittografia per ottenere messaggi indecifrabili in modo tale che nessuno, ad esclusione del mittente e del ricevente, possa far uso delle informazioni comunicate. La codifica va intesa, dunque, come il procedimento attraverso il quale rappresentiamo le informazioni di partenza come stringhe di bit. Il motivo per il quale si usano sequenze di bit e non sequenze di caratteri o di simboli, è che il computer ragiona in termini di 0 e 1 in quanto basato su logica binaria. Con il processo di codifica si generano sequenze di minore lunghezza. Tenendo conto di queste riflessioni è facile comprendere il perché dell'applicazione della compressione.

Per definizione, la compressione dei dati è un'efficiente codifica in formato digitale delle informazioni di partenza ottenuta col minor numero di bit pur mantenendo una adeguata riproduzione dell'originale. Questo procedimento implica l'eliminazione di bit ridondanti cioè superflui al fine della rappresentazione, ma bisogna essere in grado di recuperarli nella fase di decompressione, quindi al momento della creazione del file compresso bisogna inserirvi anche la legge per la decompressione.

La compressione ha numerose applicazioni: ad esempio ci permette di risparmiare tempo e banda per lo scambio di informazioni in rete e di diminuire lo spazio occupato da un file in memoria. Sono stati fatti passi da gigante con lo studio di algoritmi

per la compressione. Allo stato attuale questi algoritmi vengono fondamentalmente divisi in due categorie:

- Lossless : non vi è alcuna perdita di informazione rispetto al file originario
- Lossy : vi è una parziale perdita di informazione rispetto all'informazione originaria. Tale perdita comporta percentuali di compressione abbastanza elevate. Gli algoritmi che producono perdita di informazione vanno applicato sempre tenendo conto della percezione umana e considerando la banda e lo spazio disponibili.

La compressione lossless è usata per la compressione di dati che devono essere perfettamente ricostruiti nello stato originale (quali ad esempio testi di programmi o testo). La compressione lossy è usata quando abbiamo bisogno di rapporti di compressione un po' più elevati. Un'altra caratteristica fondamentale degli algoritmi di compressione è la tecnica di compressione adottata. Si parla di tecnica di compressione:

- statistica o basata su modello: si costruisce un modello contenente la probabilità dei simboli dell'alfabeto sorgente, in base ai risultati di tale studio statistico della sorgente si determinano le stringhe di bit associate a ciascun simbolo di sorgente, la codifica viene eseguita in maniera istantanea, un simbolo alla volta
- basata su dizionario: si costruisce un dizionario o codebook ovvero una tabella dei simboli e delle stringhe di sorgente; ogni loro occorrenza nel file sorgente è sostituita dall'indice del corrispondente elemento all'interno del dizionario.

Col passare degli anni sono stati fatti notevoli progressi in tale campo e introdotti numerosi standard. Tutti questi algoritmi differiscono per il tipo di tecnica adottata e per la fedeltà rispetto all'originale. La codifica aritmetica e Huffman si collocano nei metodi lossless e basati su modello. Entrambi sono usati nell'ambito della codifica e compressione di dati multimediali. L'algoritmo di Huffman genera un codice prefisso e a lunghezza variabile che associa una codeword ad ogni simbolo di sorgente. La codifica aritmetica o arithmetic coding, invece, associa un solo valore in virgola mobile, a tutto il segnale sorgente. La maggiore efficienza del secondo, rispetto al primo, sta nella possibilità di utilizzare un numero non intero di bit per ogni simbolo di sorgente.

1.2 TEORIA DELL'INFORMAZIONE

Nella teoria dell'informazione *l'entropia* misura la quantità di incertezza o informazione presente in un segnale aleatorio. Da un altro punto di vista l'entropia è la minima complessità descrittiva di una variabile aleatoria, ovvero il limite inferiore della compressione dei dati.

Il nucleo della teoria dell'informazione fu elaborato nel 1948 dall'ingegnere elettrotecnico statunitense Claude E. Shannon e fu il risultato di una lunga e sistematica attività di ricerca dettata dall'esigenza di una solida base teorica per le tecnologie delle comunicazione[1]. L'intenso sfruttamento dei canali di comunicazione, quali le reti telefoniche e i sistemi di radiocomunicazione, iniziato negli anni Trenta e continuato nel dopoguerra, poneva infatti in primo piano la necessità della messa a punto di sistemi sempre più efficaci per la trasmissione dei messaggi.

Si deve a Shannon la prima espressione di un sistema per la generazione e trasmissione delle informazioni in forma di modello, che comprende una sorgente dell'informazione, un codificatore, un canale di trasmissione, un decodificatore e un osservatore, il quale, rilevando, interpretando e utilizzando l'informazione per fini propri, interagisce con essa, contribuendo in parte a determinarne il contenuto trasferito.

Il significato dei vari elementi del modello discende piuttosto intuitivamente dalla loro denominazione: meno intuitiva è la definizione di informazione, che, nel senso della teoria, va intesa come tutto il contenuto di variazione, novità e imprevedibilità di un messaggio che due sistemi interagenti si scambiano. Per essere trasmesso, tutto questo contenuto di "novità" necessita di un supporto materiale, che nella pratica si identifica con un canale di trasmissione (telefono, radio, suono ecc.), il quale porta il messaggio in forma di modificazioni di varie porzioni della propria struttura.

La definizione del sistema ottimale per il trasferimento di un messaggio deve dunque necessariamente passare per la misura del suo contenuto, che richiede la scelta di relazioni matematiche fra le variabili che meglio caratterizzano il concetto di "novità" sopra delineato. Secondo Shannon, il contenuto informativo di un messaggio è legato alla sua probabilità di mostrarsi entro un insieme di messaggi possibili: maggiore è la probabilità di realizzarsi, minore è il contenuto informativo. È abbastanza intuitivo che sarà il messaggio meno probabile, fra diverse alternative, a

portare la massima quantità di informazione quando si verificherà, mentre il contenuto informativo di un messaggio atteso con certezza è 0.

Come esempio si può considerare il lancio di una moneta: il messaggio "testa o croce" descrive il risultato, ma non ha contenuto informativo, mentre i due messaggi distinti "testa" oppure "croce" sono ugualmente possibili, con probabilità pari a $\frac{1}{2}$, e da quanto detto finora dovrebbero portare il massimo dell'informazione. Matematicamente, la misura del contenuto di un'informazione I , si ottiene con la formula di Shannon:

$$I = \log_2 \frac{p'}{p} \quad (1.1)$$

dove p è la probabilità del messaggio di essere trasmesso, p' è legato all'uso che l'osservatore fa del messaggio, e coincide con la probabilità che il contenuto dell'informazione da questi attesa ha di realizzarsi dopo la trasmissione del messaggio, mentre \log_2 indica l'operazione di logaritmo in base 2, ossia la determinazione dell'esponente che deve essere attribuito al numero 2 per ottenere il numero in argomento (ad esempio, $\log_2 b = x \rightarrow 2^x = b$).

Nel caso del lancio della moneta, l'osservatore può essere interessato solo a sapere se è uscito testa o croce, e dunque ciascuna delle due informazioni, per le quali $p = \frac{1}{2}$ e $p' = 1$ porta contenuto informativo 1.

Nella maggior parte delle applicazioni pratiche della teoria dell'informazione bisogna operare una scelta tra i messaggi di un insieme, ciascuno con una sua probabilità di essere trasmesso. Shannon ha dato una definizione di entropia di tale insieme, identificandola con il contenuto informativo che la scelta di uno dei messaggi trasmetterà. Se ciascun messaggio ha probabilità p_i di venire trasmesso, l'entropia si ottiene come la somma su tutto l'insieme delle funzioni $p_i \log_2 p_i$, ciascuna relativa a un messaggio. Il termine entropia, mutuato dalla termodinamica, designa dunque il contenuto informativo medio di un messaggio. Se N messaggi di un insieme hanno uguali probabilità, la formula dell'entropia diventa :

$$H = \log_2 N \quad (1.2)$$

Ad esempio, se si trasmettono messaggi che consistono in combinazioni casuali delle 26 lettere dell'alfabeto, del segno di

spazio e dei cinque segni di interpunzione, e se si assume che la probabilità di ogni messaggio sia la stessa, l'entropia è data dall'espressione

$$H = \log_2 32 = 5 \quad (1.3)$$

In altre parole, sono necessari al massimo cinque bit per codificare ogni carattere, o messaggio: 00000, 00001, 00010, ..., 11111. Può accadere che il numero di simboli necessari per trasmettere e memorizzare in modo efficace l'informazione sia inferiore a quello ottenuto con la valutazione dell'entropia del sistema: è quindi necessario ridurre il numero di bit usati per la codifica. Questo è possibile nell'elaborazione dei testi, ad esempio, poiché la distribuzione delle lettere dell'alfabeto non è completamente casuale (ad esempio, alcune sequenze di lettere sono più probabili di altre).

In tal caso si calcola, con le probabilità modificate, un'entropia relativa del messaggio, rapporto fra l'entropia effettiva e l'entropia massima, quella che si avrebbe se i messaggi dell'insieme fossero equiprobabili. Se questa grandezza viene sottratta al numero uno, si ottiene la "*ridondanza*", ovvero la percentuale di informazione del messaggio che rimane automaticamente definita da regole dell'insieme relative agli accostamenti fra i segni contenuti nel messaggio. L'entropia della lingua scritta generalmente è di circa un bit per lettera, il che significa che il linguaggio è molto ridondante. È proprio questa ridondanza che permette a una persona di capire messaggi lacunosi o di decifrare una calligrafia poco chiara. Nei moderni sistemi di comunicazione si parla di "*ridondanza artificiale*" per indicare quella ridondanza che viene introdotta di proposito nei messaggi codificati per ridurre gli errori di trasmissione.

LA CODIFICA ARITMETICA

Nel capitolo 1, sono stati introdotti i concetti di compressione e codifica e vari aspetti della teoria dell'informazione. In questo capitolo si descriverà la codifica aritmetica, la sua implementazione e le varie problematiche da affrontare per realizzarla.

2.1 INTRODUZIONE ALLA CODIFICA ARITMETICA

La codifica aritmetica è stata inventata da Rissanen e presentata in un articolo nel 1976[5].

Analogamente ad altre codifiche per la compressione di dati, l'idea è di ricodificare l'informazione da trasmettere, assegnando le parole codificate di lunghezza minore ai simboli più frequenti¹. Shannon ha dimostrato che, dato un insieme di eventi distinti mutuamente esclusivi $e_1, e_2, e_3, \dots, e_n$ ed una distribuzione di probabilità P definita su di essi, il minor numero di bit necessari per codificare un evento e_k è pari alla sua entropia

$$H = - \sum_{k=1}^n p_{e_k} \log_2 p_{e_k} \quad (2.1)$$

dove p_{e_k} è la probabilità associata al verificarsi dell'evento e_k , la quantità H si chiama entropia.

Teoricamente i codici aritmetici assegnano una *codeword*² differente per ciascuna combinazione di ingresso. A differenza di codifiche tradizionali, come ad esempio quella di *Huffman*, che utilizzano un prefisso di lunghezza variabile per differenziare le codeword, in questo caso la parola di codice rappresenta un sottointervallo aperto di $[0, 1)$.

¹ Principio di Morse

² Parola codificata

Ogni sottointervallo è rappresentato da un numero binario di precisione opportuna, tale da differenziarlo da tutti gli altri possibili sottointervalli, in modo che a codeword corte corrispondono sottointervalli di ampiezza maggiore e, quindi, associati a simboli più probabili. Il numero di bit necessari per discriminare gli intervalli più ampi è minore di quello necessario per i più piccoli: in questo modo simboli più probabili sono codificati con meno bit.

In pratica l'ampiezza dei sottointervalli è ottenuta suddividendo progressivamente il segmento $[0,1)$, in modo che le ampiezze di ogni ripartizione siano proporzionali alla probabilità dei singoli eventi dell'alfabeto di ingresso.

E' stato mostrato che, in media, i codificatori basati su tale metodo di codifica forniscono prestazioni di codifica superiori a quelle ottenibili con codici a lunghezza variabile. Una delle loro caratteristiche più interessanti, infatti, è la possibilità di ottenere meno di un bit di informazione codificata per ciascun simbolo in ingresso.

Il prezzo da pagare è una maggiore complessità di implementazione dato che si devono manipolare numeri binari di precisione arbitraria. Inoltre è possibile osservare che gli stream compressi con codifiche aritmetiche presentano una minore immunità alla propagazione degli errori. Può essere sufficiente un bit errato nei dati codificati per portare ad un'errata ricostruzione dell'informazione.

2.2 MODELLI PROBABILISTICI

Il processo di codifica necessita di conoscere la distribuzione di probabilità dei simboli da comprimere. Tanto maggiore è la precisione con cui è nota questa distribuzione, tanto migliore sarà la compressione ottenibile. E' chiaro, quindi, che essere in grado di fornire un modello probabilistico accurato prelude la possibilità di ottenere un ottimo sistema di compressione.

Si noti che, affinché la decodifica possa avvenire con successo, il *decoder* deve potere utilizzare lo stesso modello di probabilità dell'encoder. E' possibile classificare i codificatori aritmetici in base al sistema utilizzato per la costruzione del modello di probabilità. Se ne individuano di tre tipi:

- statici o non adattativi : La distribuzione di probabilità è assegnata a priori, scelta opportunamente in base ai dati da codificare. Le prestazioni ottenibili dai codificatori che utilizzano questi modelli sono in genere molto modeste, anche se la loro complessità può risultare molto ridotta.
- semi-adattativi : La codifica avviene in due fasi. Durante la prima si determina la distribuzione di probabilità da utilizzare mediante opportune stime. La codifica aritmetica vera e propria è effettuata nella seconda passata. Le prestazioni di compressione sono molto buone. Lo svantaggio principale è la necessità di trasmettere esplicitamente il modello probabilistico insieme ai dati compressi, alterando svantaggiosamente così il rapporto di compressione. D'altra parte, il decoder non ha alcun modo di determinare la distribuzione di probabilità usata in codifica.
- adattativi : Si effettua una stima dinamica delle probabilità dei simboli $e_1, e_2, e_3, \dots, e_n$, "apprendendo" progressivamente la statistica dei dati durante la codifica. Il processo di compressione può avvenire, in questo caso, in una singola passata e non è necessario segnalare esplicitamente il modello ottenuto. Per il decodificatore sarà sufficiente ripetere "a ritroso" le stime effettuate in codifica, adattando progressivamente il proprio modello di probabilità, in modo analogo a quanto fatto in codifica. Lo svantaggio principale di questi sistemi è il cosiddetto costo di apprendimento: all'inizio la conoscenza della statistica dei dati è molto rozza e inadeguata ad una compressione efficace. Durante il transitorio di adattamento iniziale, pertanto, i simboli codificati non saranno ottimi dal punto di vista del numero di bit compressi per simbolo.

2.3 CODIFICA ARITMETICA "CLASSICA"

L'algoritmo 2.1 presenta la codifica aritmetica nella sua forma più semplice. Solo per privilegiare la chiarezza espositiva si suppone che l'alfabeto di simboli da codificare sia binario. In figura 1 è rappresentato un esempio di codifica aritmetica di una stringa binaria di tre simboli. In particolare, chiamando a il MPS ³ e b il LPS ⁴ ed ipotizzando un alfabeto di ingresso $\mathcal{A} = \{a,b\}$ la figura rappresenta la codifica della sequenza baa.

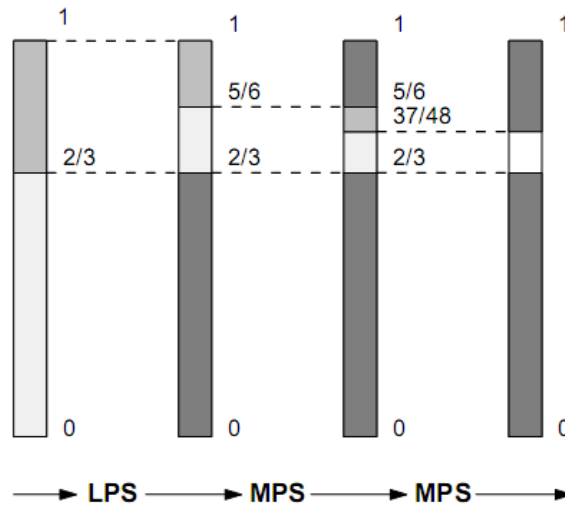


Figura 1: Esempio di codifica aritmetica classica.

Si tenga presente che la probabilità di occorrenza dei simboli non rimane costante nel corso del processo di codifica presentato, al fine di generalizzare l'esempio anche al caso di contesti adattativi. Inizialmente :

$$p(\text{MPS}) = \frac{2}{3} \quad (2.2)$$

$$p(\text{LPS}) = \frac{1}{3} \quad (2.3)$$

Si codifica un LPS e quindi l'intervallo scelto sarà di ampiezza $I = 3$.

³ Acronimo di More Probable Symbol

⁴ Acronimo di Less Probable Symbol

Al secondo passo :

$$p(\text{MPS}) = \frac{1}{2} \quad (2.4)$$

$$p(\text{LPS}) = \frac{1}{2} \quad (2.5)$$

e per un MPS si avrà $I = \frac{1}{2} \frac{1}{3} = \frac{1}{6}$. Infine :

$$p(\text{MPS}) = \frac{5}{8} \quad (2.6)$$

$$p(\text{LPS}) = \frac{3}{8} \quad (2.7)$$

L'intervallo finale diventa pertanto $I = \frac{5}{8} \frac{1}{6} = \frac{5}{48}$. A questo punto è sufficiente emettere una stringa binaria che identifichi completamente l'intervallo. Visto che $I = [\frac{2}{3}, \frac{37}{48})$ e che:

$$\frac{2}{3} \rightarrow 0.101010101\dots_2 \quad (2.8)$$

$$\frac{37}{48} \rightarrow 0.110001011\dots_2 \quad (2.9)$$

per identificare l'intervallo è possibile utilizzare tutti i numeri binari compresi tra i due estremi. Una scelta valida potrebbe essere, ad esempio, la parola binaria 0.1100_2 .

Algorithm 2.1 Codifica aritmetica

procedure ARITHMETICCODING(\mathcal{A}) ▷ Alfabeto binario
 $L \leftarrow 0$
 $H \leftarrow 1$
 $I \leftarrow [L, H)$
for all $k | e_k \in \mathcal{A}$ **do**
 $\bar{I}_L = (H - L) p_{\text{lps}}$
 $\bar{I}_H = (H - L) p_{\text{mps}}$
if $e_k = \text{MPS}$ **then**
 $H \leftarrow H - \bar{I}_L$
else
 $L \leftarrow L + \bar{I}_H$
end if
end for
Emetto bit a sufficienza per identificare l'intervallo finale.
end procedure

In questo algoritmo, al termine della codifica, la lunghezza del sottointervallo risultante rappresenta la probabilità associata ad

una particolare sequenza di eventi. Il valore ottenuto è uguale al prodotto delle probabilità degli eventi codificati, ovvero :

$$P = \prod_k p_k \quad (2.10)$$

La lunghezza finale della parola di codice ottenuta è, al più, pari a $\lfloor -\log_2 p \rfloor + 2$. Inoltre nello stream compresso è opportuno prevedere un sistema per segnalare la fine dei dati. Si possono adottare due strategie:

- Segnalare la fine dei dati mediante una codeword "riservata". Occorre garantire che nessun insieme di simboli in ingresso sia in grado di produrre una parola codificata uguale al simbolo di fine file.
- Aggiungere all'inizio del file un preambolo che contiene la lunghezza totale dei dati codificati.

In ogni caso l'informazione da aggiungere tende a degradare leggermente le prestazioni di compressione del sistema. Il metodo di codifica "classico" presenta due principali difetti: lo scalamento dell'intervallo richiede l'impiego di aritmetica a precisione arbitraria, e non è possibile produrre, in uscita, la rappresentazione compressa dell'informazione fintanto che non si termina l'intera codifica. Per ovviare a questi svantaggi sono state proposte numerose varianti al processo di codifica classico. Una tra le più interessanti è la codifica aritmetica incrementale.

2.4 CODIFICA ARITMETICA INCREMENTALE

Rappresenta la soluzione più diretta dei problemi del metodo tradizionale. L'idea è di spedire in uscita ogni bit codificato non appena è disponibile ⁵ e, in seguito, di raddoppiare l'ampiezza dell'intervallo ottenuto, mantenendola sempre maggiore di $\frac{1}{4}$. Sono stati proposti, in letteratura, vari algoritmi di codifica incrementale che fanno uso di aritmetica a precisione finita.

Di particolare interesse è l'idea di utilizzare un meccanismo di *bit-stuffing* per limitare la propagazione dei riporti nelle operazioni aritmetiche, proposta da ricercatori dell'IBM.

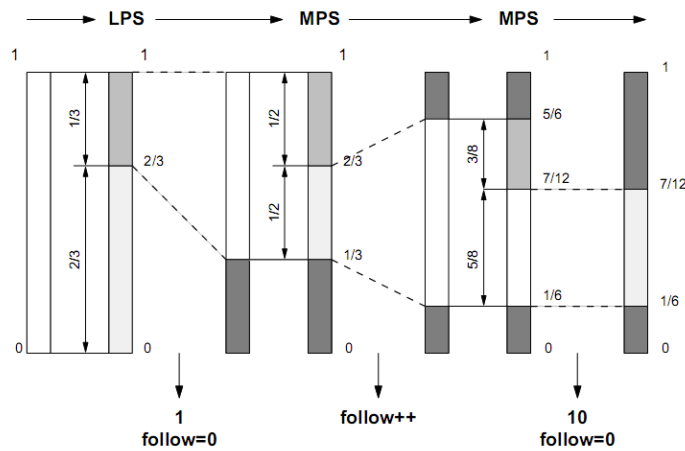


Figura 2: Esempio di codifica aritmetica incrementale.

In figura 2 è riportato un esempio di codifica incrementale, simile a alla figura 1, con gli stessi dati di ingresso utilizzati nell'esempio del paragrafo precedente. Anche la probabilità di occorrenza dei simboli è la stessa dell'esempio precedente.

E' interessante osservare l'espansione dell'intervallo al termine di ciascuna operazione di codifica. L'algoritmo 2.2 propone una possibile implementazione della codifica aritmetica incrementale. Il codificatore emette un 1 se l'intervallo attuale I' appartiene a $[\frac{1}{2}, 1)$, uno 0 se appartiene a $[0, \frac{1}{2})$. Il caso più interessante si ha quando l'intervallo I' giace "a cavallo" di $\frac{1}{2}$. Qui, non sapendo se emettere uno zero o un uno, è sufficiente incrementare il valore del contatore *follow*, che indica quante espansioni sono state effettuate per le quali $I' \in [\frac{1}{4}, \frac{3}{4})$. Quando si emetterà

⁵ Spesso ci si riferisce a queste strategie con il termine ASAP, acronimo di As Soon As Possible

Algorithm 2.2 Codifica aritmetica Incrementale

```

procedure ARITHMETICCODINGINCR( $\mathcal{A}$ )    ▷ Alfabeto binario
   $L \leftarrow 0$ 
   $H \leftarrow 1$ 
   $I \leftarrow [L, H)$ 
  for all  $k | e_k \in \mathcal{A}$  do
     $\bar{I}_L = (H - L) p_{lps}$ 
     $\bar{I}_H = (H - L) p_{mps}$ 
    if  $e_k = \text{MPS}$  then
       $H \leftarrow H - \bar{I}_L$ 
    else
       $L \leftarrow L + \bar{I}_H$ 
    end if
     $I' \leftarrow [L, H)$ 
     $\text{break} \leftarrow 0$ 
    while  $\text{break} \neq 1$  do
      if  $I' \in [0, \frac{1}{2})$  then
        Emetto 0 seguito da 1 specificato da follow
         $I' \leftarrow 2I'$ 
         $\text{follow} \leftarrow 0$ 
      else if  $I' \in [\frac{1}{2}, 1)$  then
        Emetto 1 seguito da 0 specificato da follow
         $I' \leftarrow 2I'$ 
         $\text{follow} \leftarrow 0$ 
      else if  $I' \in [\frac{1}{4}, \frac{3}{4})$  then
         $\text{follow} \leftarrow \text{follow} + 1$ 
      else
         $\text{break} \leftarrow 1$ 
      end if
    end while
     $I \leftarrow I'$ 
  end for
end procedure

```

il prossimo simbolo sarà sufficiente aggiungervi un numero di simboli complementati pari al valore di *follow*.

LA CODIFICA ARITMETICA E JPEG2000

In questo capitolo si descrive il codificatore aritmetico adattativo utilizzato in JPEG2000. Dopo una rapida introduzione in cui si inquadrano le problematiche tipiche dei processi di compressione, si presenterà il codificatore aritmetico adattativo MQ dell'IBM.

In JPEG2000 la codifica entropica dell'informazione è affidata all'algoritmo EBCOT¹. Introdotto nel 1998 da David Taubman [6], possiede interessanti caratteristiche, particolarmente adatte ad essere impiegate nel nuovo ambiente di compressione. Nello schema a blocchi del codificatore JPEG2000 si colloca a valle del quantizzatore. La caratteristica principale di EBCOT è la capacità di produrre stream compressi altamente scalabili e ad accesso diretto.

3.1 LA COMPRESSIONE ARITMETICA IN JPEG2000

Per codificare le informazioni provenienti dal blocco EBCOT in JPEG2000 si impiega un particolare tipo di codificatore aritmetico adattativo, chiamato MQ-coder. La principali peculiarità del codificatore MQ sono l'impiego di aritmetica a precisione finita (numeri interi) e la capacità di compressione senza effettuare moltiplicazioni e divisioni. Ispirato ai codificatori Q e QM sviluppati dalla IBM alla fine degli anni ottanta, è in grado di offrire ottime prestazioni di compressione, richiedendo un carico computazionale non eccessivo.

A differenza di questi ultimi, il codificatore MQ non è protetto da copyright troppo stringenti e si pensa che questo potrà favorire sensibilmente la diffusione.

¹ Acronimo di Embedded Block Coding with Optimized Truncation

Volendo collocare MQ nelle classificazioni presentate nel capitolo 2 alle pagine 9 e 12, si potrebbe affermare che l'approccio seguito, in questo caso, è di tipo ibrido. Infatti mentre per l'intervallo di probabilità è suddiviso in modo "classico", applicando ricorsivamente le partizioni in sottointervalli, le parole di codice vengono emesse appena sono disponibili. Esiste inoltre la possibilità di espandere l'ampiezza del segmento da suddividere, per garantire di restare all'interno di un intervallo di ampiezza opportuna.

Da un punto di vista operativo il codificatore MQ suddivide ricorsivamente un intervallo in parti, di ampiezza proporzionale alla probabilità del simbolo associato. L'alfabeto in ingresso è binario: i simboli, anziché 0 e 1, sono chiamati convenzionalmente MPS e LPS. Va sottolineato che, nel corso del processo di codifica, la corrispondenza tra MPS,LPS e 0,1 può cambiare, in seguito ad operazioni di normalizzazione.

Ad ogni passo di codifica l'intervallo è diviso e la stringa di codice modificata in modo opportuno, facendo in modo che si riferisca sempre alla base del sottointervallo corrente. Nella suddivisione dei sottointervalli si assume, per convenzione, che il simbolo meno probabile (LPS) si trovi "al di sotto" di quello più probabile (si veda, ad esempio, la figura 3).

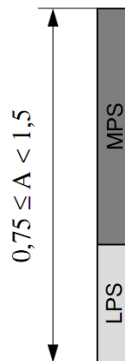


Figura 3: MQ-coder. Suddivisione dell'intervallo in LPS e MPS.

Per questo motivo, ogni volta che si codifica MPS, il valore della probabilità del LPS è sommato al registro di codice; al contrario, la codifica di LPS non richiede somme. Siccome il dato codificato punta sempre alla base dell'intervallo corrente, la decodifica consiste nel determinare quale segmento è indicato nello stream compresso. In modo simmetrico a quanto fatto in codifica il valore della probabilità di LPS è sottratta dal valore che

individua il sottointervallo attualmente in esame.

In tabella 1 si riportano i principali registri impiegati dal codificatore MQ ed il loro significato. Da una prima analisi dei registri è facile notare che

Tabella 1: MQ-coder. Registri interni.

Encoder	Registro	Significato
A	16	Ampiezza dell'intervallo
C	32	Parola codificata

il codificatore utilizza un'aritmetica a precisione finita. L'assenza di registri floating-point è un notevole vantaggio a livello implementativo e risolve, almeno in parte, i problemi di rappresentazione presenti nei codificatori "classici".

Una caratteristica interessante è la rappresentazione interna impiegata, che associa il valore esadecimale 0x8000 al numero decimale 0.75. Questa scelta, anche se potrebbe apparire bizzarra, è giustificata da interessanti considerazioni che saranno esposte tra breve. L'algoritmo, dopo ogni operazione di codifica, controlla sempre il valore del registro A, assicurandosi che risulti $0,75 \leq A < 1.5$.

Nel caso in cui questa condizione non sia rispettata il contenuto di A viene moltiplicato per due, così come C, finché il suo valore non rientrerà nell'intervallo precedente. Questa serie di moltiplicazioni per due produce uno shift verso l'alto del contenuto di entrambi i registri e potrebbe portare, dopo un po' di volte, all'overflow del registro C; per questo motivo si emette periodicamente il byte più significativo del registro di codice.

La tabella 2 è di fondamentale importanza per il codificatore MQ. Tra le informazioni che vi sono riportate quella che attualmente si esamina è la stima della probabilità del LPS, indicato come Q_e .

3.2 PROCESSO DI CODIFICA

Si presenta, di seguito, il processo di codifica aritmetica. Anche se l'analisi delle problematiche riscontrabili in compressione esula dallo scopo del presente lavoro, si è scelto di affrontarlo almeno a livello descrittivo, per favorire una più agevole comprensione del processo di decodifica. In figura 4 è riportato un diagramma a blocchi che evidenzia l'interfaccia offerta del codifi-

catore verso il blocco EBCOT. Ai fini della trattazione seguente è possibile immaginare che gli ingressi prodotti dall'esterno siano soltanto il contesto ed il dato da codificare. Il risultato prodotto dal MQ-coder è la codeword corrispondente.

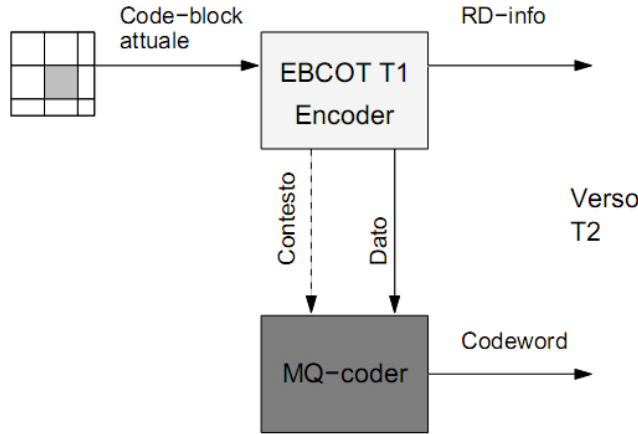


Figura 4: MQ-coder. Interfaccia con EBCOT.

Durante il processo di codifica è necessario determinare il prossimo valore dell'ampiezza del segmento da suddividere. Il valore della probabilità di occorrenza del LPS, $p(\text{LPS}) = Q_e$ può essere letto direttamente dalla tabella 2, noto lo stato in cui si trova attualmente il codificatore. È possibile calcolare l'ampiezza dei nuovi sottointervalli per mezzo delle relazioni:

$$I_{\text{LPS}} = Q_e A \quad (3.1)$$

$$I_{\text{MPS}} = A - (Q_e A) \quad (3.2)$$

Purtroppo per ricavare i valori sopra riportati sembrerebbe necessario eseguire almeno moltiplicazione per ciascun simbolo da codificare. In realtà, a patto di sacrificare un po' di accuratezza del modello probabilistico, è lecito confondere il valore attuale di A con l'unità, ovvero:

$$A \approx 1 \quad (3.3)$$

La condizione posta poc'anzi sull'intervallo dei valori di A serve proprio a garantire la validità di questa approssimazione. Sotto queste ipotesi e avendo effettuato l'approssimazione (3.3) le relazioni (3.1) e (3.2) diventano:

$$I_{\text{LPS}} \approx Q_e \quad (3.4)$$

$$I_{\text{MPS}} \approx A - Q_e \quad (3.5)$$

In questo modo è possibile evitare il calcolo delle due moltiplicazioni, riducendo così la complessità computazionale dell'algoritmo. Quando si codifica un MPS, Q_e va aggiunto al valore corrente del registro C, così come $A = A - Q_e$. Nel caso della codifica di un LPS il registro C rimane invariato ed $A = Q_e$. In entrambi i casi la condizione necessaria riguardo al valore di A può essere soddisfatta con shift successivi che equivalgono, in binario, a moltiplicazioni per potenze di due.

Nel processo descritto esisterebbero dei casi anomali nei quali, per l'approssimazione (3.3), l'ampiezza dell'intervallo associato al MPS potrebbe diventare inferiore a quella del LPS, violando così una delle ipotesi fondamentali della codifica aritmetica. Come esempio, si riporta un caso che verifica quanto sopra affermato.

Qualora si avesse $A = 0.75$ e $Q_e = 0.5$ gli intervalli sarebbero $I_{LPS} = Q_e = 0.5$ e $I_{MPS} = Q_e = 0.25$. Ma chiaramente questo porterebbe ad avere $I_{LPS} > I_{MPS}$ che non avrebbe alcun senso. Al fine di evitare questo inconveniente è opportuno effettuare una rinormalizzazione, scambiando i valori di MPS e LPS per lo stato attuale. A seguito di una rinormalizzazione bisogna determinare una nuova stima della probabilità per il contesto corrente.

Per adattare il modello probabilistico si utilizza la macchina a stati finiti, rappresentata in tabella 2. I campi *NMPS* e *NLPS* indicano a quale nuovo stato associare il contesto attualmente codificato.

Tabella 2: MQ-coder. Registri interni.

I	Q _e		NMPS	NLPS	SWITCH
	Decimale	Esadecimale			
0	0.503937	0x5601	1	1	1
1	0.304715	0x3401	2	6	0
2	0.140650	0x1801	3	9	0
3	0.063012	0x0AC1	4	12	0
4	0.030053	0x0521	5	29	0
5	0.012474	0x0221	38	33	0
6	0.503937	0x5601	7	6	1
7	0.492218	0x5401	8	14	0
8	0.421904	0x4801	9	14	0
9	0.328153	0x3801	10	14	0
10	0.281277	0x3001	11	17	0
11	0.210964	0x2401	12	18	0
12	0.164088	0x1C01	13	20	0
13	0.128931	0x1601	29	21	0
14	0.503937	0x5601	15	14	1
15	0.492218	0x5401	16	14	0
16	0.474640	0x5101	17	15	0
17	0.421904	0x4801	18	16	0
18	0.328153	0x3801	19	17	0
19	0.304715	0x3401	20	18	0
20	0.281277	0x3001	21	19	0
21	0.234401	0x2801	22	19	0
22	0.210964	0x2401	23	20	0
23	0.199245	0x2201	24	21	0
24	0.164088	0x1C01	25	22	0
25	0.140650	0x1801	26	23	0
26	0.128931	0x1601	29	21	0
27	0.117212	0x1401	28	25	0
28	0.105493	0x1201	29	26	0
29	0.099634	0x1101	30	27	0
30	0.063012	0x0AC1	31	28	0
31	0.057153	0x09C1	32	29	0
32	0.050561	0x08A1	33	30	0
33	0.030053	0x0521	34	31	0
34	0.024926	0x0441	35	32	0
35	0.015404	0x02A1	36	33	0
36	0.012474	0x0221	37	34	0
37	0.007347	0x0141	38	35	0
38	0.006249	0x0111	39	36	0
39	0.003044	0x0085	40	37	0
40	0.001671	0x0049	41	38	0
41	0.000847	0x0025	42	39	0
42	0.000481	0x0015	43	40	0
43	0.000206	0x0009	44	41	0
44	0.000114	0x0005	45	42	0
45	0.000023	0x0001	45	43	0
46	0.503937	0x5601	46	46	0

3.3 PROCESSO DI DECODIFICA

I blocchi coinvolti nella decodifica sono gli unici ad essere regolati esplicitamente dello standard JPEG2000.

Il decoder aritmetico è un processo che riceve in ingresso un flusso di byte, che costituiscono la rappresentazione compressa di un quality layer del codeblock corrente.

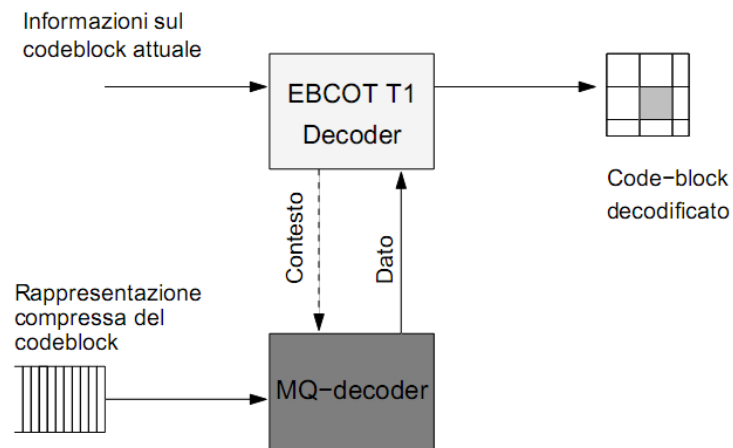


Figura 5: MQ-decoder. Interfaccia con EBCOT.

Inoltre, il decodificatore EBCOT valuta, per ciascun campione appartenente al codeblock, il contesto ad esso associato, fornendolo in ingresso al blocco di decodifica aritmetica. In figura 5 si riporta un possibile schema a blocchi che mostra le interazioni tra il blocco EBCOT e il decodificatore aritmetico adattativo.

IMPLEMENTAZIONE

Nei capitoli precedenti sono stati descritti i concetti chiave per la realizzazione del progetto d'esame. Questa sezione introdurrà la parte pratica, illustrando l'ambiente di sviluppo utilizzato, le scelte progettuali ed i metodi con cui queste sono state realizzate. Si comincia con l'introdurre l'ambiente di sviluppo.

Il codice del progetto è stato sviluppato in ambiente Linux con il compilatore usato è *GCC version 4.3.2*. Il compilatore GCC è stato scelto, non soltanto perchè è gratuito, ma anche perchè è uno dei migliori compilatori che ci siano in circolazione, poichè è il più conforme allo standard C/C++ , ed in più essendo open source è costantemente aggiornato.

Il codice implementato realizza la codifica aritmetica "classica", in quanto per problemi di precisione e rappresentazione dovuti all'epsilon macchina degli attuali processori , pari a

$$2.220446049250313e^{-16} \quad (4.1)$$

pertanto non è possibile effettuare operazioni matematiche di precisione superiore e questo ha limitato l'implementazione della versione incrementale. Sono state implementate apposite classi per la gestione del codificatore e del decodificatore; entrambe le classi gestiscono un flusso di dati in formato testuale e condividono lo stesso modello probabilistico.

Dopo questa breve descrizione degli strumenti utilizzati per la stesura del codice, e delle idee base utilizzate per realizzarlo, il prossimo passo sarà quello di descrivere quest'ultimo in modo approfondito.

4.1 LA CLASSE ENCODER

La classe *Encoder* è la classe che descrive il codificatore aritmetico. Tra i suoi attributi privati troviamo :

Tabella 3: La classe Encoder

Encoder	
map<char,double> lookupTable	contiene gli estremi inferiori
map<char,double> freq	contiene le probabilità
map<char,int> occ	contiene le occorrenze
short cumfreq	frequenze cumulative
void readFreq(string in)	legge la frequenza dei caratteri
void model()	modello statistico adottato
void entropia()	calcola dell'entropia teorica
void updateMap(double low, double high)	aggiornamento mappa
void writeOutFile(string out, double code)	scrittura codifica su file

Tutte le mappe sono indicizzate dallo stesso carattere, la *lookupTable* contiene l'estremo inferiore dell'intervallo relativo alla sua probabilità per ogni carattere, questa mappa è estremamente importante per il continuo aggiornamento degli intervalli. La mappa *occ* contiene le occorrenze dei caratteri, mentre la mappa *freq* contiene la frequenza di probabilità per ogni carattere. La variabile *cumfreq* rappresenta le occorrenze cumulative dei caratteri nonché la dimensione del file di input. Ora analizzeremo in dettaglio i vari metodi della classe

4.1.1 Il metodo readFreq

Il metodo *readFreq* è utilizzato per leggere le informazioni del file di input, ovvero i singoli caratteri presenti e le loro relative occorrenze. Inoltre, inizializza le tre mappe descritte in precedenza e la variabile *cumfreq* incrementata ogni qualvolta viene letto un carattere.

4.1.2 Il metodo model

Il metodo *model* descrive il modello probabilistico usato. La probabilità di ogni carattere viene calcolata come segue :

$$\frac{occ_i}{cumfreq} \quad (4.2)$$

dove occ_i è l'occorrenza di ogni singolo carattere al passo i -esimo e *cumfreq* rappresenta il numero totale di occorrenze.

4.1.3 Il metodo `updateMap`

Il metodo *updateMap* viene utilizzato per l'aggiornamento della mappa *lookupTable*. Il metodo ha come parametri di input gli estremi del nuovo intervallo in base al quale dovrà effettuare un'operazione di scalatura che avviene nel modo seguente :

$$\text{low} = \text{low} + (\text{high} - \text{lowOld}) * \text{freq}_i \quad (4.3)$$

dove *low* rappresenta ad ogni passo l'estremo inferiore dell'intervallo relativo alla sua probabilità per ogni carattere, *high* è l'estremo superiore ricevuto in input, *freq_i* è la probabilità del carattere *i*-esimo.

4.1.4 Il metodo `writeOutFile`

Il metodo *writeOutFile* viene utilizzato per scrivere la codifica elaborata la quale consiste nella scrittura delle frequenze cumulative totali, della mappa delle occorrenze e del codice finale generato dal metodo *encoding*. Per i problemi di implementazione descritti precedentemente la variabile *cumfreq* e la mappa delle occorrenze vengono scritti su file considerando solo i byte meno significativi. Inoltre per risparmiare spazio sul file il codice generato che è di tipo *double* viene indicizzato da un tipo intero a 64 bit per permettere la scrittura di un byte alla volta.

4.1.5 Il metodo `encoding`

Il metodo *encoding* è il cuore del codificatore in essa vengono richiamate tutti gli altri metodi descritti fino ad ora. La parte principale è costituita da un ciclo *while* la cui condizione di terminazione si verificherà quando verranno effettuati *cumfreq* cicli. Ad ogni passo viene letto un carattere, e tramite il metodo *find* della struttura dati *map* si individua l'estremo inferiore dell'intervallo relativo alla sua probabilità per ogni carattere. Per l'estremo superiore verrà considerato l'estremo inferiore del carattere successivo. Solo nel caso in cui il carattere letto si troverà nell'ultima posizione della mappa l'estremo superiore non verrà aggiornato in quanto è fisso.

4.2 LA CLASSE DECODER

La classe *Decoder* è la classe che descrive il decodificatore aritmetico. Tra i suoi attributi privati troviamo :

Tabella 4: La classe Decoder

Decoder	
map<char,double> lookupTable	contiene gli estremi inferiori
map<char,double> freq	contiene le probabilità
map<char,int> occ	contiene le occorrenze
short cumfreq	frequenze cumulative
double code	il codice calcolato dal encoder
void readInfo(string in)	legge l'header del file codificato
void model()	modello statistico adottato
void updateMap(double low, double high)	aggiornamento mappa

Come si evince dalla tabella, il decoder è praticamente speculare all'encoder per cui si descriveranno solo i metodi *readInfo* e *decoding*

4.2.1 Il metodo readInfo

Il metodo *readInfo* legge le informazioni del file di input codificato. Il primo carattere letto è relativo a *cumfreq*, in seguito entra in un ciclo dove leggerà tutti i caratteri con le loro occorrenze. Per non invadere locazioni sul file che non appartengono a questo tipo di informazioni man mano che le occorrenze vengono lette, verrà confrontato il valore con *cumfreq*. Infine per leggere il codice generato dal codificatore viene indicizzata una variabile intera a 64 bit con un double.

4.2.2 Il metodo decoding

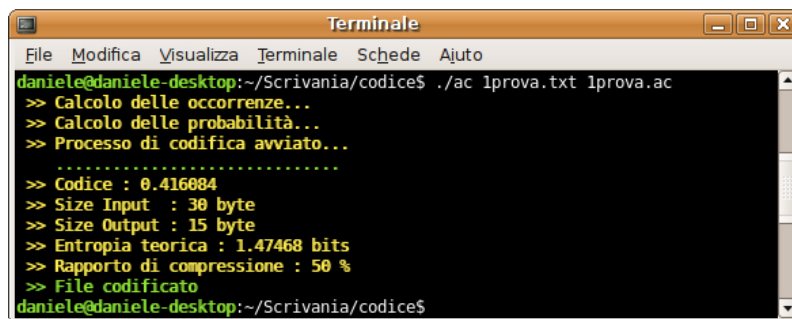
Il metodo *decoding* è il cuore del decodificatore, in esso vengono richiamate tutti gli altri metodi. A differenza del processo di codifica nel quale possiamo fruttare il metodo find della struttura dati map, nel processo di decodifica dobbiamo scandire l'intera mappa fino a che non si individuano gli estremi dell'intervallo che conterrà il codice. Dopo l'individuazione dell'intervallo si scrive sul file di output il carattere relativo all'estremo inferiore dell'intervallo e si prosegue con l'aggiornamento degli intervalli facendo particolare attenzione, come nell'encoding, all'estremo superiore.

4.3 ESEMPI DI CODIFICA

Di seguito vengono riportati alcuni test di funzionamento del progetto. I diversi test ci permettono di considerare i diversi scenari in base alle diverse distribuzioni di probabilità dei vari file di prova. Nel primo esempio il messaggio da codificare è il seguente

cbbcicbbccicbbccbbiibccccciccb

Come possiamo notare il dalla figura 6 processo di codifica produce il codice a precisione finita ed inoltre le informazioni riguardanti l'entropia teorica ed il rapporto di compressione che in questo caso è pari al 50%



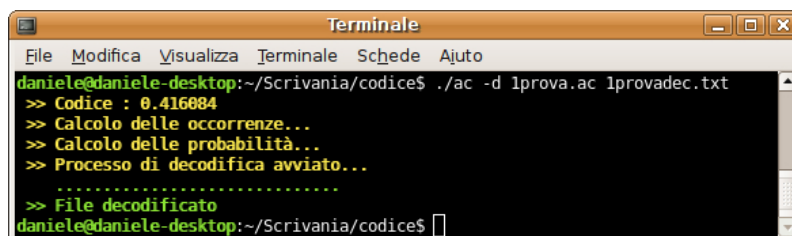
```

Terminale
File Modifica Visualizza Terminale Schede Ajuto
daniele@daniele-desktop:~/Scrivania/codice$ ./ac 1prova.txt 1prova.ac
>> Calcolo delle occorrenze...
>> Calcolo delle probabilità...
>> Processo di codifica avviato...
>> .....
>> Codice : 0.416084
>> Size Input : 30 byte
>> Size Output : 15 byte
>> Entropia teorica : 1.47468 bits
>> Rapporto di compressione : 50 %
>> File codificato
daniele@daniele-desktop:~/Scrivania/codice$

```

Figura 6: Codifica del file 1prova.txt.

Il processo di decodifica, invece, ricostruisce il file originale a partire da quello codificato conoscendo solamente codice e le occorrenze da cui successivamente calcolerà le probabilità.



```

Terminale
File Modifica Visualizza Terminale Schede Ajuto
daniele@daniele-desktop:~/Scrivania/codice$ ./ac -d 1prova.ac 1prova.dec.txt
>> Codice : 0.416084
>> Calcolo delle occorrenze...
>> Calcolo delle probabilità...
>> Processo di decodifica avviato...
>> .....
>> File decodificato
daniele@daniele-desktop:~/Scrivania/codice$

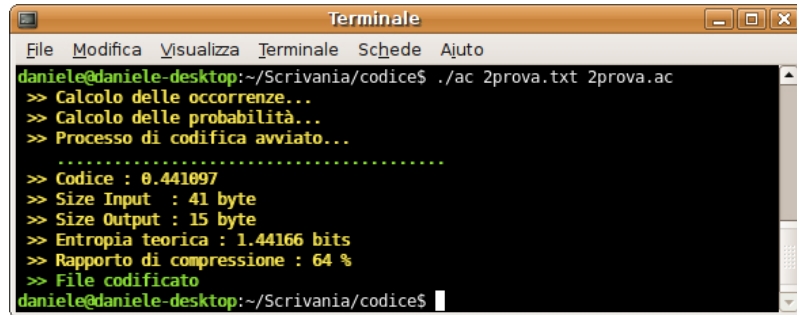
```

Figura 7: Decodifica del file 1prova.ac.

Nel secondo test il file 2prova.txt contiene il messaggio

anna nana na na annanna nanna annaaannnnn

Possiamo notare anche in questo caso di aver ottenuto un'ottima compressione pari al 64%



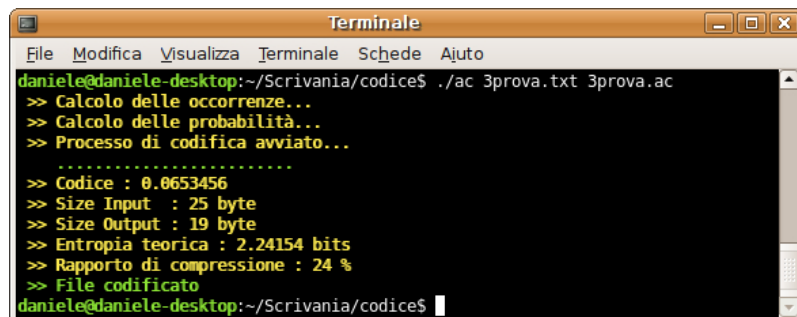
```

Terminale
File Modifica Visualizza Terminale Schede Ajuto
daniele@daniele-desktop:~/Scrivania/codice$ ./ac 2prova.txt 2prova.ac
>> Calcolo delle occorrenze...
>> Calcolo delle probabilità...
>> Processo di codifica avviato...
>> .....
>> Codice : 0.441097
>> Size Input : 41 byte
>> Size Output : 15 byte
>> Entropia teorica : 1.44166 bits
>> Rapporto di compressione : 64 %
>> File codificato
daniele@daniele-desktop:~/Scrivania/codice$

```

Figura 8: Codifica del file 2prova.txt.

Nell'ultimo test il messaggio da codificare è il seguente : AntonioAntonioAntoniotoni. In questo caso abbiamo ottenuto una ottima compressione pari al 24% , una compressione un po' scadente. Questo, dovuto al fatto che le distribuzioni di probabilità dei caratteri presenti nel file, sono molto vicine tra esse.



```

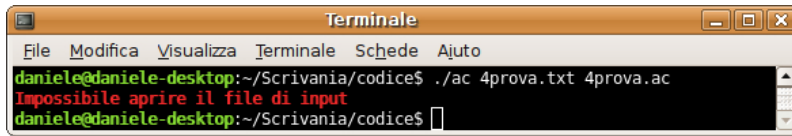
Terminale
File Modifica Visualizza Terminale Schede Ajuto
daniele@daniele-desktop:~/Scrivania/codice$ ./ac 3prova.txt 3prova.ac
>> Calcolo delle occorrenze...
>> Calcolo delle probabilità...
>> Processo di codifica avviato...
>> .....
>> Codice : 0.0653456
>> Size Input : 25 byte
>> Size Output : 19 byte
>> Entropia teorica : 2.24154 bits
>> Rapporto di compressione : 24 %
>> File codificato
daniele@daniele-desktop:~/Scrivania/codice$

```

Figura 9: Codifica del file 3prova.txt.

Di seguito sono riportati i controlli che vengono effettuati dal programma, per consentire il corretto utilizzo. Nella figura 10 è riportato il controllo sui file. In questo caso il file "4prova.txt" è inesistente.

Nella figura 11 è mostrato la schermata d'errore dovuta al mancato inserimento dei parametri da riga di comando, infatti manca il file di output.

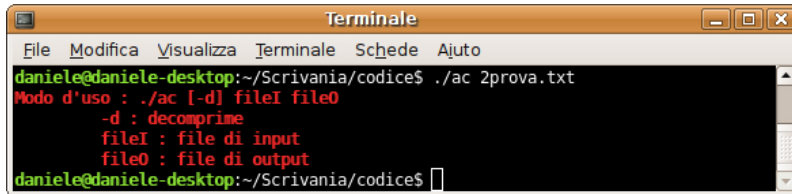


```

Terminale
File Modifica Visualizza Terminale Schede Ajuto
daniele@daniele-desktop:~/Scrivania/codice$ ./ac 4prova.txt 4prova.ac
Impossibile aprire il file di input
daniele@daniele-desktop:~/Scrivania/codice$

```

Figura 10: Errore: Il file 4prova.txt non esiste.



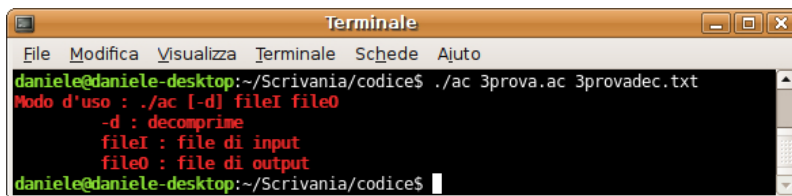
```

Terminale
File Modifica Visualizza Terminale Schede Ajuto
daniele@daniele-desktop:~/Scrivania/codice$ ./ac 2prova.txt
Modo d'uso : ./ac [-d] fileI fileO
-d : decomprime
fileI : file di input
fileO : file di output
daniele@daniele-desktop:~/Scrivania/codice$

```

Figura 11: Errore: Manca il file di output

In questo ultimo caso, invece, manca il parametro “-d” in quanto, serve a specificare l’intenzione di voler decodificare un file, come file di input, però, è stato inserito un file con estensione “.ac”, per cui già codificato.



```

Terminale
File Modifica Visualizza Terminale Schede Ajuto
daniele@daniele-desktop:~/Scrivania/codice$ ./ac 3prova.ac 3prova.dec.txt
Modo d'uso : ./ac [-d] fileI fileO
-d : decomprime
fileI : file di input
fileO : file di output
daniele@daniele-desktop:~/Scrivania/codice$

```

Figura 12: Errore: Manca il file di output

4.4 CONCLUSIONI

Durante l'implementazione e la stesura della tesina si è constatato che l'arithmetic coding rispetto all'algoritmo di Huffman ha delle prestazioni superiori. Tale vantaggio sta nel fatto che, mentre Huffman produce una codifica per ogni singolo simbolo, la codifica aritmetica produce una sola codifica per l'intero messaggio.

Ad esempio supponendo di avere un file composto da 1000 caratteri, uno presente 999 volte su mille e l'altro presente una sola volta su mille, il file compresso con l'algoritmo di Huffman sarà composto da 1000 bit perché viene associato a ciascuno dei due caratteri una codeword di un bit, mentre il file compresso col metodo della codifica aritmetica sarà composto soltanto da 64 bit associati all'intero segnale sorgente. Pur essendo molto più efficiente dell'algoritmo di Huffman, la codifica aritmetica non ha avuto una grande diffusione a causa delle royalty per il suo utilizzo.

Sono già state introdotte, nell'apposita sezione, le difficoltà riscontrate nell'implementazione della codifica aritmetica e i relativi limiti. Tali circostanze non hanno consentito la realizzazione di esempi complessi, dunque le considerazioni fatte a riguardo vanno adattate al contesto con cui si è lavorato.



CODICE SORGENTE

A.1 ENCODER.H

```
1  #ifndef ENCODER_H
2  #define ENCODER_H
3
4  #include <iostream>
5  #include <cstdlib>
6  #include <string>
7  #include <fstream>
8  #include <map>
9  #include <cmath>
10 #include "color.h"
11
12 using namespace std;
13
14 class Encoder
15 {
16 private:
17     map<char, double> lookupTable;
18     map<char, double> freq;
19     map<char, int> occ;
20     short cumfreq;
21
22     //metodo che restituisce la frequenza dei caratteri
23     void readFreq(string in);
24
25     //calcolo probabilit dei singoli caratteri
26     void model();
27
28     //aggiornamento mappa
29     void updateMap(double low, double high);
30
31     //calcola dell'entropia teorica
32     void entropia();
33
34     //scrittura codifica su file
35     void writeOutFile(string out, double code);
36
37 public:
38     Encoder() {cumfreq = 0;}
39
40     //processo di codifica
41     void encoding(string inFile, string outFile);
42
43     ~Encoder() {}
44 };
45
46 #endif //ENCODER_H
```

A.2 ENCODER.CPP

```

1 #include "Encoder.h"
2
3 //metodo che restituisce la frequenza dei caratteri
4 void Encoder::readFreq(string in)
5 {
6     const char* inFile = in.c_str();
7     ifstream ifile(inFile, ios::binary);
8     if (ifile.fail())
9     {
10         cerr<<RED<<"Impossibile aprire il file di input"<<endl
11         ;
12         exit(1);
13     }
14     else
15     {
16         while (ifile.good())
17         {
18             char ch = ifile.get();
19             if ( (ch&255)!=255 && (ch&255)!=10 )
20             {
21                 ++cumfreq;
22                 if (occ.find(ch) != occ.end())
23                     ++occ[ch];
24             }
25             else
26             {
27                 occ.insert(pair<char, int>(ch,1));
28                 freq.insert(pair<char, double>(ch,0.0));
29                 lookupTable.insert(pair<char, double>(ch,0.0)
30                 );
31             }
32         }
33         ifile.close();
34     }
35 }
36 //calcolo occorrenze totali
37 void Encoder::model()
38 {
39     map<char, double>::iterator itFreq = freq.begin();
40     map<char, int>::iterator itOcc = occ.begin();
41     while ( itFreq != freq.end())
42     {
43         itFreq->second = itOcc->second / static_cast<double>(
44             cumfreq);
45         ++itFreq;
46         ++itOcc;
47     }
48 }
49 //aggiornamento mappa
50 void Encoder::updateMap(double low, double high)
51 {
52     map<char, double>::iterator itTable = lookupTable.begin();
53     map<char, double>::iterator itFreq = freq.begin();
54     double lowOld = low;
55     while ( itTable != lookupTable.end())
56     {
57         itTable->second = low;
58         low += (high - lowOld) * itFreq->second;

```

```

59         ++itTable;
60         ++itFreq;
61     }
62 }
63
64 //scrittura codifica su file
65 void Encoder::writeOutFile(string out, double code)
66 {
67     const char* outFile = out.c_str();
68     ofstream ofile(outFile, ios::trunc | ios::binary);
69     if (ofile.fail())
70     {
71         cerr<<RED<<"Impossibile aprire il file "<<endl;
72         exit(1);
73     }
74     ofile.put(cumfreq&255);
75     map<char, int>::iterator itOcc = occ.begin();
76     while (itOcc != occ.end())
77     {
78         ofile.put(itOcc->first);
79         ofile.put(itOcc->second & 255);
80         ++itOcc;
81     }
82     long long iCode =(long long &) code;
83     for (short i = 7; i >= 0; i--)
84     {
85         ofile.put((iCode >> (i*8))&255);
86     }
87     cout<<YELLOW<<"\n >> Codice : '"<<code<<endl;
88     ofile.close();
89 }
90
91 //processo di codifica
92 void Encoder::encoding(string in, string out)
93 {
94     readFreq(in);
95     cout<<YELLOW<<" >> Calcolo delle occorrenze... '"<<endl;
96     model();
97     cout<<YELLOW<<" >> Calcolo delle probabilit ... '"<<endl;
98     updateMap(0.0, 1.0);
99     cout<<YELLOW<<" >> Processo di codifica avviato... '"<<endl
100         <<"'";
101     const char* inFile = in.c_str();
102     ifstream ifile(inFile, ios::binary);
103     if (ifile.fail())
104     {
105         cerr<<RED<<"Impossibile aprire il file "<<endl;
106         exit(1);
107     }
108     else
109     {
110         double high = 1.0, low = 0.0;
111         char ch = 0;
112         int i = 0;
113         while (i < cumfreq)
114         {
115             cout<<GREEN<<".'.";
116             ch = ifile.get();
117             if ( (ch&255)!=255 && (ch&255)!=10 )
118             {
119                 map<char, double>::iterator it = lookupTable.find
120                     (ch);
121                 low = it->second;

```

```

121         ++it;
122         if (it != lookupTable.end())
123         {
124             high = it->second;
125
126         }
127         updateMap(low, high);
128         ++i;
129     }
130 }
131 writeOutFile(out, low);
132 }
133 ifile.close();
134 short sizeOutput = (lookupTable.size()*2 + 9);
135 float ratio = 100 - ( sizeOutput * 100 / cumfreq );
136 cout<<YELLOW<<'>> Size Input : '<<cumfreq<<' byte'<<
137     endl;
138 cout<<YELLOW<<'>> Size Output : '<<sizeOutput<<' byte'<<
139     <<endl;
140 entropia();
141 cout<<YELLOW<<'>> Rapporto di compressione : '<<ratio<<'
142     %'<<endl;
143 cout<<GREEN<<'>> File codificato'<<endl;
144 }
145
146 void Encoder::entropia()
147 {
148     double H = 0.0;
149     map<char, double>::iterator itFreq = freq.begin();
150     while ( itFreq != freq.end())
151     {
152         H += itFreq->second * log2(itFreq->second);
153         ++itFreq;
154     }
155     cout<<YELLOW<<'>> Entropia teorica : '<<-H<<' bits'<<
156         endl;
157 }

```

A.3 DECODER.H

```

1  #ifndef DECODER_H
2  #define DECODER_H
3
4  #include<iostream>
5  #include<cstdlib>
6  #include<string>
7  #include<fstream>
8  #include<map>
9  #include''color.h''
10
11 using namespace std;
12
13 class Decoder
14 {
15 private:
16     map<char, double> lookupTable;
17     map<char, double> freq;
18     map<char, int> occ;
19     short cumfreq;
20     double code;
21
22     //metodo che restituisce le informazioni necessarie per la
23     //decodifica
24     void readInfo(string in);
25
26     //calcolo probabilit dei singoli caratteri
27     void model();
28
29     //aggiornamento mappa
30     void updateMap(double low, double high);
31
32 public:
33     Decoder() {cumfreq = 0; code = 0.0;}
34
35     //processo di codifica
36     void decoding(string inFile, string outFile);
37
38     ~Decoder() {}
39 };
40 #endif //DECODER_H

```

A.4 DECODER.CPP

```

1 #include ''Decoder.h''
2
3 using namespace std;
4
5 //metodo che restituisce la frequenza dei caratteri
6 void Decoder::readInfo(string in)
7 {
8     const char* inFile = in.c_str();
9     ifstream ifile(inFile, ios::binary);
10    if (ifile.fail())
11    {
12        cerr<<RED<<'Impossibile aprire il file di input''<<endl
13        ;
14        exit(1);
15    }
16    else
17    {
18        char ch = 0, c = 0;
19        //Leggo cumfreq
20        ch = ifile.get();
21        cumfreq = (cumfreq<<8)|(ch&255);
22        int z = cumfreq;
23        do
24        {
25            int val = 0;
26
27            //Leggo carattere
28            ch = ifile.get();
29
30            //Leggo occorrenza
31            c = ifile.get();
32
33            val = (val<<8)|(c&255);
34            occ.insert(pair<char, int>(ch, val));
35            freq.insert(pair<char, double>(ch, 0.0));
36            lookupTable.insert(pair<char, double>(ch, 0.0));
37            z -= val;
38        } while (ifile.good() && z > 0);
39
40        //intero a 64 bit
41        long long iCode = 0;
42        for (int i = 0; i < 8; ++i)
43        {
44            ch = ifile.get();
45            iCode = (iCode << 8)|(ch&255);
46        }
47        code = (double &)iCode;
48        cout<<YELLOW<<'>> Codice : ''<<code<<endl;
49    }
50    ifile.close();
51 }
52
53 //calcolo occorrenze totali
54 void Decoder::model()
55 {
56     map<char, double>::iterator itFreq = freq.begin();
57     map<char, int>::iterator itOcc = occ.begin();
58     while (itFreq != freq.end())
59     {
60

```

```

61         itFreq->second = static_cast<double>(itOcc->second) /
            cumfreq;
62         ++itFreq;
63         ++itOcc;
64     }
65 }
66
67 //aggiornamento mappa
68 void Decoder::updateMap(double low, double high)
69 {
70     map<char, double>::iterator itTable = lookupTable.begin();
71     map<char, double>::iterator itFreq = freq.begin();
72     double lowOld = low;
73     while ( itTable != lookupTable.end())
74     {
75         itTable->second = low;
76         low += (high - lowOld) * itFreq->second;
77         ++itTable;
78         ++itFreq;
79     }
80 }
81
82 //processo di decodifica
83 void Decoder::decoding(string in, string out)
84 {
85     readInfo(in);
86     cout<<YELLOW<<'>> Calcolo delle occorrenze... '><<endl;
87     model();
88     cout<<YELLOW<<'>> Calcolo delle probabilit ... '><<endl;
89     updateMap(0.0,1.0);
90     cout<<YELLOW<<'>> Processo di decodifica avviato... '><<
        endl<<'>>';
91     const char* outFile = out.c_str();
92     ofstream ofile(outFile, ios::trunc | ios::binary);
93
94     if (ofile.fail())
95     {
96         cerr<<RED<<'>>Impossibile aprire il file '><<endl;
97         exit(1);
98     }
99     double high = 1.0, low = 0.0, lowOld;
100     char ch = 0;
101     int i = 0;
102     map<char, double>::iterator it, itOld;
103     while ( i < cumfreq)
104     {
105         cout<<GREEN<<'>>';
106         it = lookupTable.begin();
107         while ( code >= it->second && it != lookupTable.end
            () )
108         {
109             itOld = it;
110             ++it;
111         }
112         ch = itOld->first;
113         ofile.put((ch&255));
114         low = itOld->second;
115         if (it != lookupTable.end())
116         {
117             high = it->second;
118         }
119         updateMap(low, high);
120         ++i;
121     }

```



```

122 |     ofile.close();
123 |     cout<<GREEN<<endl<<'\'a >> File decodificato'\'<<endl;
124 | }

```

A.5 COLOR.H

```

1 | /*color.h*/
2 |
3 | #ifndef COLOR_H
4 | #define COLOR_H
5 |
6 | #define BLACK    '\033[1;30m'
7 | #define RED      '\033[1;31m'
8 | #define GREEN    '\033[1;32m'
9 | #define YELLOW   '\033[1;33m'
10 | #define BLUE     '\033[0;34m'
11 | #define MAGENTA  '\033[1;35m'
12 | #define CYAN     '\033[1;36m'
13 | #define WHITE    '\033[1;37m'
14 | #define DEFAULT  '\033[0;00m'
15 |
16 | #endif // COLOR_H

```

A.6 MAIN.CPP

```

1 #include <cstring>
2 #include ''Encoder.h''
3 #include ''Decoder.h''
4
5 using namespace std;
6
7 int main(int argc, char *argv[])
8 {
9
10     if (argc == 4 && strcmp(argv[1], '-d') == 0 )
11     {
12         Decoder decod;
13         string input = argv[2];
14         string output = argv[3];
15         decod.decoding(input, output);
16     }
17     else if (argc == 3 && argv[1][strlen(argv[1])-1] != 'c')
18     {
19         Encoder cod;
20         string input = argv[1];
21         string output = argv[2];
22         cod.encoding(input, output);
23     }
24     else
25     {
26         cerr<<RED<<'Modo d'uso : ./ac [-d] fileI fileO''<<endl;
27         cerr<<RED<<'\\t -d : deprime\\n\\t fileI : file di
28             input''<<endl;
29         cerr<<RED<<'\\t fileO : file di output''<<endl;
30         exit(1);
31     }
32     cout<<DEFAULT;
33     return 0;
34 }

```

BIBLIOGRAFIA

- [1] Microsoft Encarta. Teoria dell'informazione, 2009. URL http://it.encarta.msn.com/text_761577650___6/Teoria_dell'informazione.html. (Citato a pagina 3.)
- [2] Berger Gibson and Lookabaugh. *Digital Compression For Multimedia*. Morgan Kaufmann Publishers, 1998.
- [3] John G. Cleary Ian H. Witten, Radford M. Neal. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, 1987. URL http://www.cs.duke.edu/~jsv/Papers/HoV94.arithmetic_coding.pdf.
- [4] Wikipedia l'enciclopedia libera. Codifica aritmetica, 2008. URL http://it.wikipedia.org/wiki/Codifica_aritmetica.
- [5] Jorma Rissanen. Generalized kraft inequality and arithmetic coding. *IBM Journal of Research and Development*, 20(3): 198–203, 1976. URL <http://domino.watson.ibm.com/tchjr/journalindex.nsf/4ac37cf0bdc4dd6a85256547004d47e1/53fec2e5af172a3185256bfa0067f7a0?OpenDocument>. (Citato a pagina 6.)
- [6] David Taubman. High performance scalable image compression with ebcot. *IEEE Transactions on Image Processing*, 9(7):1158–1170, 2000. URL http://www.ee.unsw.edu.au/~taubman/publications_files/ebcot-icip99.pdf. (Citato a pagina 14.)

INDICE ANALITICO

AC, [6](#)

codice, [29](#)

Codifica Aritmetica, [6](#)

Codifica aritmetica "classica", [9](#)

Codifica aritmetica incrementale, [12](#)

Codifica e Compressione, [1](#)

color.h, [36](#)

Conclusioni, [28](#)

Decoder, [24](#)

Decoder.cpp, [34](#)

Decoder.h, [33](#)

decoding, [24](#)

Encoder, [22](#)

Encoder.cpp, [30](#)

Encoder.h, [29](#)

encoding, [23](#)

Implementazione, [21](#)

lookupTable, [22](#)

lossy, [1](#)

main.cpp, [37](#)

Modelli probabilistici, [7](#)

readInfo, [24](#)

updateMap, [23](#)

writeOutFile, [23](#)