Corso di Laboratorio di Algoritmi - AA 2000 - 2001
Compressione dei Dati
Arithmetic Coding

Lorenzo Demattè
2SI

July 25, 2001

1 Introduzione alla compressione

Tutti coloro che posseggono un computer hanno usato almeno una volta un programma di compressione dei dati. In qualunque file non compresso i singoli caratteri (o i singoli pixel, nel caso di un'immagine, oppure i singoli byte nel caso più generico possibile) compaiono con frequenze diverse e quindi la loro rappresentazione standard che usa per ogni simbolo un numero fisso di bit (8, 16, 24 ...) non è certamente ottimale. È sicuramente piú conveniente costruire un alfabeto a lunghezza variabile in cui a ciascun simbolo viene associato un opportuno codice, cioè una sequenza di bit la cui lughezza è proporzionale alla frequenza del particolare simbolo in esame.

Verrà presentato un algoritmo di codifica molto generale e malleabile, l' *Arithmetic Coding*, che permette di raggiungere l'entropia della sorgente (il massimo livello di compressione teoricamente raggiungibile per un dato alfabeto, vedi paragrafo seguente), seguendo alcuni accorgimenti. Verrà poi mostrato come, modellando opportunamente la sorgente, sia possibile trarre il massimo vantaggio dai pattern e dalle strutture ricorrenti nei file per diminuire localmente l'entropia e quindi aumentare il rapporto di compressione.

Ho scelto di focalizzare l'attenzione su due tipi di files molto comuni: testo e immagini bitmap. Ho quindi sviluppato un modello per files di testo, due per le immagini bitmap e uno "general purpose". Non tratterò altri tipi di dati, sebbene con poco sforzo sia possibile adattare i modelli presentati alle esigenze specifiche (per esempio, i files audio WAVE con qualità CD potrebbero essere compressi discretamente con un modello adattivo differenziale che lavora su simboli di 16bit (una classe `differential_adaptive_model<WORD>`)).

Non tratterò neppure la compressione lossy (con perdita di informazione), per cui sono necessarie conoscenze e algoritmi specifici e una matematica molto più complessa.

2 Cenni di Teoria dell'Informazione

Vediamo ora di introdurre alcuni concetti necessari alla trattazione, come quello di entropia e quello di decifrabilità del codice.

Consideriamo l'alfabeto X costituito da M simboli (ad es. per i caratteri ASCII $M = 256$) e indichiamo con H la sua entropia, cioè il contenuto informativo medio associato a ciascun simbolo. Risulta evidente che H dipende dalle frequenze con cui compaiono i vari simboli, cioè dalle loro probabilità di presentazione; infatti se un simbolo è molto probabile, convoglia una scarsa informazione, mentre se è poco probabile la sua presenza è assai significativa. In definitiva, si definisce entropia (o contenuto informativo medio) dell'alfabeto x la quantità

$$H \;=\; \Sigma_{i=1}^{M}\; p_i \; log_2 \frac{1}{p_i} \quad [bit/simbolo]$$

dove p_i rappresenta la probabilità di occorrenza del simbolo x_i.

Come si può verificare facilmente, H è massima quando tutti i simboli sono equiprobabili -nel qual caso $H = log_2 M$-, mentre è minima (e pari a 0) nel caso limite in cui un solo simbolo abbia probabilità unitaria e tutti gli altri abbiano invece probabilità nulla (è evidente che ad una tale situazione non è associata alcuna informazione). Se indichiamo con n_i il numero di bit necessari per codificare x_i, diremo che la lunghezza media del codice è pari a

$$N \;=\; \Sigma_{i=1}^{M}\; p_i n_i \quad [bit/simbolo]$$

Un risultato fondamentale della Teoria dell'Informazione è che non è possibile costruire nessun codice univocamente decifrabile la cui lunghezza media sia inferiore all'entropia dell'alfabeto, ossia si ha sempre $N \geq H$.
Va sottolineato che l'entropia puó essere raggiunta, e che quindi se un algoritmo è ben progettato $N = H$.
Consideriamo ad esempio l'alfabeto quaternario mostrato in tabella

Simbolo	Probabilita	Codice
x_1	$\frac{1}{2}$	0
x_2	$\frac{1}{4}$	01
x_3	$\frac{1}{8}$	10
x_4	$\frac{1}{8}$	100

In questo caso, applicando le definizioni, si ha:

$$H \;=\; \frac{1}{2}\, log_2 2 + \frac{1}{4}\, log_2 4 + \frac{1}{8}\, log_2 8 + \frac{1}{8}\, log_2 8 \;=\; 1.75 \; bit/simbolo$$

mentre la lunghezza media del codice è

$$N \;=\; \frac{1}{2} * 1 + \frac{1}{4} * 2 + \frac{1}{8} * 2 + \frac{1}{8} * 3 \;=\; 1.625 \; bit/simbolo$$

Sembra che siamo riusciti a ottenere un codice per ciu $N < H$.
Tuttavia il codice così costruito risulta del tutto inutilizzabile; infatti la successione 010010 può corrispondere indifferentemente a 5 possibili messaggi: $x_1x_3x_2x_1$, $x_2x_1x_1x_3$, $x_1x_3x_1x_3$, $x_2x_1x_2x_1$ e $x_1x_4x_3$, per cui il codice è detto ambiguo o non univocamente decifrabile.

Una condizione sufficiente per assicurare l'univocità è che nessun codice sia il prefisso di un altro più lungo. In tal caso il codice si dice essere un *prefix code*. Nell'esempio invece si viola questa condizione sia per x_1 che per x_3. Seguendo l'algoritmo di Huffman è possibile costruire un *prefix code* che sia univocamente decifrabile e la cui lunghezza media sia pari all'entropia H. In questo caso, infatti,

$$N \;=\; \frac{1}{2}*1+\frac{1}{4}*2+\frac{1}{8}*3+\frac{1}{8}*3 \;=\; 1.75 \; bit/simbolo$$

Codice di Huffman per l'alfabeto preso in esame

Simbolo	Probabilita	Codice
x_1	$\frac{1}{2}$	0
x_2	$\frac{1}{4}$	10
x_3	$\frac{1}{8}$	110
x_4	$\frac{1}{8}$	111

Va detto che é comunque possibile fare di meglio.
Infatti fino a qui abbiamo sfruttato solo una parte dell'informazione a nostra disposizione; lavorando solo sulle frequenze dei singoli simboli si sfrutta esclusivamente il fatto che questi non sono equiprobabili; in realtà, in qualunque file, esiste sempre una qualche correlazione tra elementi successivi, che può essere sfruttata per aumentare ulteriormente il fattore di compressione. Ad esempio, nel caso di testi italiani, se si considerano separatamente i vari caratteri, sono necessari mediamente circa 4.1 bit/simbolo, mentre lo sfruttamento dell'interdipendenza fra i simboli (ad esempio il fatto che la lettera *'q'* é quasi sempre seguita da una *'u'*) consente di scendere a meno di 2 bit/simbolo.

Per sfruttare appieno queste ridondanze bisogna complicare la procedura di codifica considerando opportune estensioni dell'alfabeto, i cui elementi non sono più i singoli caratteri ma stringhe di simboli scelte opportunamente. Questa operazione (nota come modellizzazione della sorgente che, nel caso di simboli indipendenti, si riduce alla determinazione delle probabilità p_i) è molto delicata e pesante dal punto di vista computazionale, ma consente di ottenere fattori di compressione particolarmente elevati. (vedi paragrafo 5 *"Modellizzazione della sorgente"*)

3 Codifica di Huffman

La tecnica di compressione più nota è la codifica di Huffman, una tecnica statistica basata sulla combinazione successiva dei codici dei simboli con probabilità più bassa.

L' algoritmo di Huffman puo' essere schematizzato come segue:

- disporre tutti i simboli con ordinamento non crescente in base alle corrispondenti probabilità a priori;
- combinare successivamente i 2 simboli con probabilità più bassa per formare un nuovo simbolo composto a cui si associa come probabilità la somma delle due precedenti;
- ripetere fino a formare un albero binario che comprenda tutti i simboli;

- associare ad ogni coppia di rami un numero binario 1/0 e ricostruire il codice seguendo il cammino radice-foglia.

Come si può capire analizzando l'algoritmo, il codice è univoco (vedi paragrafo 2 *"Cenni di Teoria dell'informazione"*) e ottimo nella classe dei prefix code. Inoltre raggiunge il limite dell'entropia nel caso in cui le probalità siano potenze di 2.
Nel caso pessimo il coding rate è maggiore di un **1 bit** rispetto all'entropia (ricordo che il coding rate è il numero medio di bits usato per rappresentare un simbolo della sorgente). Se l'alfabeto è piccolo e le probabilità sono molto differenti, il codice di Huffman può risultare piuttosto inefficiente. Questo, insieme alla difficoltà di adattare il processo di codifica a modelli differenti, fa preferire un differente metodo di codifica basato sull' ***Arithmetic Coding***.

Per aumentare l'efficienza della codifica di Huffman si possono raggruppare i simboli per formare un nuovo alfabeto che sia più esteso e abbia probabilità più livellate, che abbia quindi caratteristiche più congeniali alla codifica.

Facciamo un esempio.
L'entropia dell'alfabeto mostrato nella tabella sotto è di 0.335 *bit/simbolo*. Il codice invece ha una lunghezza di ben 1.05 *bit/simbolo*. Se raggruppiamo i simboli a due a due, otteniamo il codice della tebella successiva, che ha una lunghezza media di 1.222 *bit/simbolo* (0.611 *bit/simbolo* in termini dell'alfabeto originale).

Codice di Huffam per l'alfabeto di tre lettere

Simbolo	Probabilita	Codice
x_1	0.95	0
x_2	0.02	11
x_3	0.03	10

Codice di Huffam per l'alfabeto esteso

Simbolo	Probabilita	Codice
x_1x_1	0.9025	0
x_1x_2	0.0190	111
x_1x_3	0.0285	100
x_2x_1	0.0190	1101
x_2x_2	0.0004	110011
x_2x_3	0.0006	110001
x_3x_1	0.0285	101
x_3x_2	0.0006	110010
x_3x_3	0.0009	110000

Le cose già migliorano, ma se vogliamo avvicinare l'entropia, dobbiamo raggruppare ancora... arrivando a risultati accettabili quando riuniamo blocchi di 8 simboli. L'alfabeto diventa così di ben 6561 simboli!

La grandezza dell'alfabeto cresce esponenzialmente con il numero di simboli ragruppati. E quando cerchiamo di trovare il codice di Huffman per una sequanze lunga **m**, dobbiamo generare e calcolare la probabilità congiunta di tutte le sequenze lunghe **m**.

Noi vorremmo solo assegnare il codice ad una particolare sequenza, quella considerata, senza esser obbligati a generarle tutte.

4 Arithmetic Coding

Qui entra in gioco l'*Arithmetic Coding.* Sappiamo che è necessario assegnare un tag univoco ad ogni simbolo o sequenza di simboli per ottenere un codice univocamante decifrabile. L'insieme di tag scelto per l'arithmetic coding é l'insieme dei reali $[0,\ 1)$.

Abbiamo quindi bisogno di una funzione che mappi i simboli su numeri compresi nell'intervallo considerato.

La funzione F di distribuzione cumulativa delle probabilità fornisce una corrispondenza biunivoca tra una variabile casuale e i reali compresi tra 0 e 1.

Ricordo che una variabile casuale è una variabile che fa corrispondere alle uscite di un esperimento (o, nel nostro caso, ai simboli di un alfabeto) un numero intero o reale.

Ad ogni simbolo a_i dell'alfabeto sorgente corrisponde una probabilità $P(a_i)$ Se associamo una variabile casuale X al nostro alfabeto, tramite la corrispondenza

$$X(a_i) = i$$

la funzione cumulativa di probabilità è definita come

$$F_x(i) = \Sigma_{k=1}^{i}\ P(X = k)$$

dove $P(X = k) = P(a_k)$

Ad ogni simbolo a_i con probabilità non nulla corrisponde un valore distinto di F_x; anzi, per essere più precisi, corrisponde un intervallo di valori compresi tra $[F_x(i-1),\ F_x(i))$. L'intervallo $[0,\ 1)$ è stato così partizionato completamente in i sotto-intervalli, a cui facciamo corrispondere gli i simboli del nostro alfabeto. Basta quindi scegliere un valore compresso nell'intervallo *i-esimo* per avere un tag univoco e univocamente decifrabile per il simbolo a_i.

Per generare un tag corrispondente ad una qualsiasi successione di simboli, basta suddividere ricorsivamente l'intervallo considerato in partizioni che hanno esattamente le stesse proporzioni dell'intervallo originale. Ogni simbolo successivo restringe il sottointervallo in cui possiamo scegliere il tag.
Un esempio puo' essere di aiuto.

Tabella A: Simboli usati negli esempi sull'arithmetic coding

Simbolo(a_i)	Probabilita(p_i)	$F_x(i)$
1	0.8	0.8
2	0.02	0.82
3	0.18	1

Se la sequenza è **1 3 2 1**, il processo di codifica procede come segue.

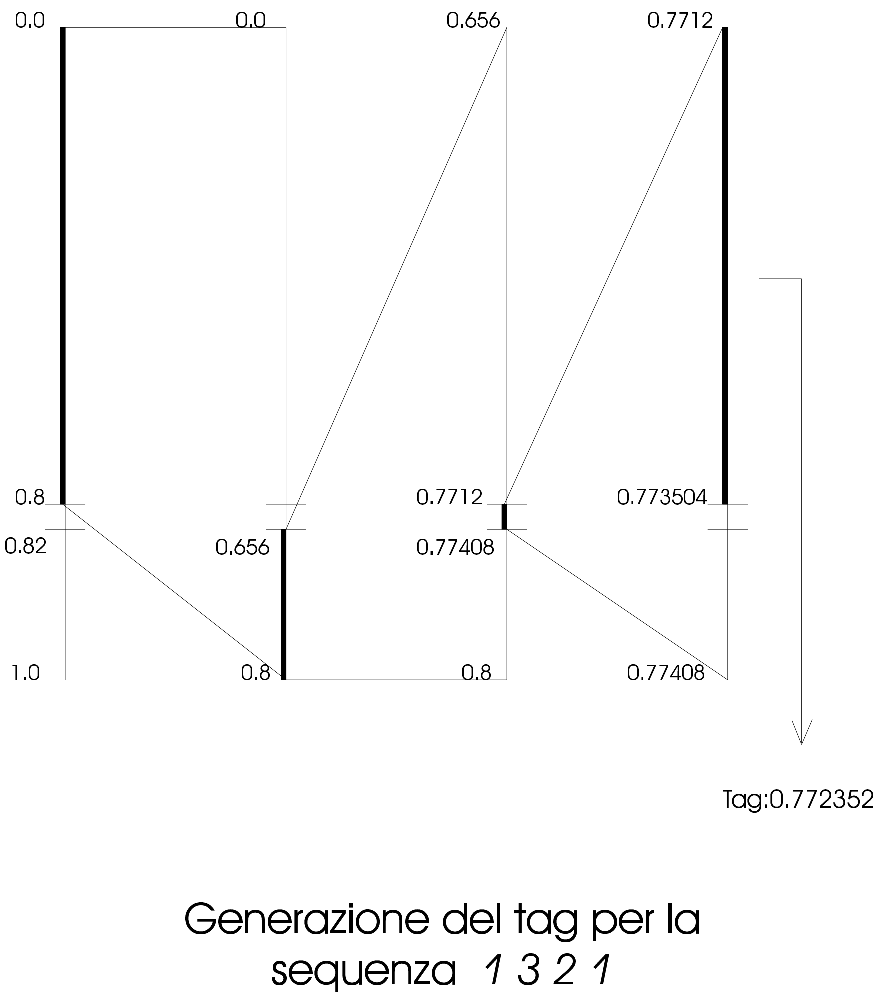


Figure 1: Generazione Tag

All'arrivo del primo simbolo (**1**) la partizione in cui il tag risiede viene ristretta a [0, 0.8), e questo intervallo viene ripartizionato mantenedo le proporzione dell'intervallo unitario, come mostrato nella Figura 1.

È importante notare che la codifica di un nuovo simbolo della sequenza restringe l'intervallo del tag in modo che questo sia disgiunto da ogni altro intervallo generato con questo processo. Quindi, anche nel caso di sequenze di simboli, è sufficiente scegliere un valore qualunque compreso nell'intervallo generato per avere un tag univoco e univocamente decifrabile per la sequenza considerata. Una scelta ricorrente è quella di usare il punto mediano dell'intervallo. Inoltre è conveniente dal punto di vista computazionale calcolare solo gli estremi inferiore e superiore dell'intervallo: questi ultimi bastano per identificarlo univocamente.

Potremmo vedere come tutto questo funziona matematicamente, tramite il calcolo delle probabilità, ma personalmente ritengo che questo appesantisca eccessivamente la trattazione. Chi volesse dettagli e dimostrazioni, può vedere **[1]**

E' molto più produttivo pensare in termini di estremi dell'intervallo. Indichiamo con $l^{(k)}$ l'estremo inferiore e con $u^{(k)}$ l'estremo superiore dopo l'arrivo del k-esimo simbolo della sequenza.

All'inizio si ha

$$l^{(0)} = 0 \qquad u^{(0)} = 1$$

Ora osserviamo la Figura 1 e pensiamo a quello che dobbiamo fare quando arriva il prossimo simbolo. Si tratta semplicemente di una proporzione tra l'intervallo attuale (la barra verticale, nella figura) e la funzione cumulativa di probabilità.

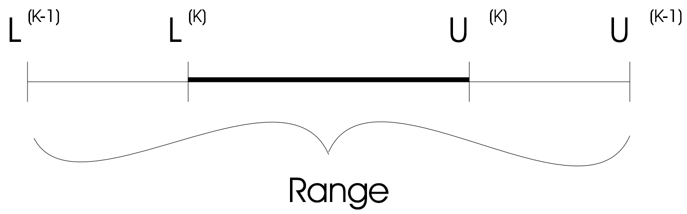
Ricordo che ad ogni simbolo a_i con probabilità non nulla corrisponde un intervallo di valori compresi tra $[F_x(i-1),\ F_x(i))$ e che per trovare il nuovo intervallo del tag devo ripartizionare l'intervallo attuale mantenendo queste proporzioni.

Quindi, all'arrivo di un nuovo simbolo a_i della nostra sequenza effettuiamo la ripartizione tenendo conto che l'estremo inferiore del nuovo sotto-intervallo sta alla grandezza dell' intervallo al passo $k-1$ come l'estremo inferiore dell'intervallo della F_x relativo al simbolo considerato (cioè $F_x(i-1)$) sta alla grandezza dell'intervallo totale della F_x (che é per definizione 1).
Quindi

$$l^{(k)} : range = F_x(i-1) : 1$$

Analogamente, per l'estremo superiore abbiamo

$$u^{(k)} : range = F_x(i) : 1$$



$l^{(k)}$ e $u^{(k)}$ così calcolati sono relativi all'inizio dell'intervallo, quindi aggiungiamo loro $l^{(k-1)}$ per avere i valori relativi all'intervallo unitario [0, 1). Otteniamo:

$$l^{(k)} = l^{(k-1)} + range * F_x(i-1) \qquad (a)$$
$$u^{(k)} = l^{(k-1)} + range * F_x(i) \qquad (b)$$

Dove $range = u^{(k-1)} - l^{(k-1)}$

Decifrare un tag è semplice come generarlo. Dopo aver inizializzato gli estremi dell' intervallo a

$$l^{(0)} = 0 \qquad u^{(0)} = 1$$

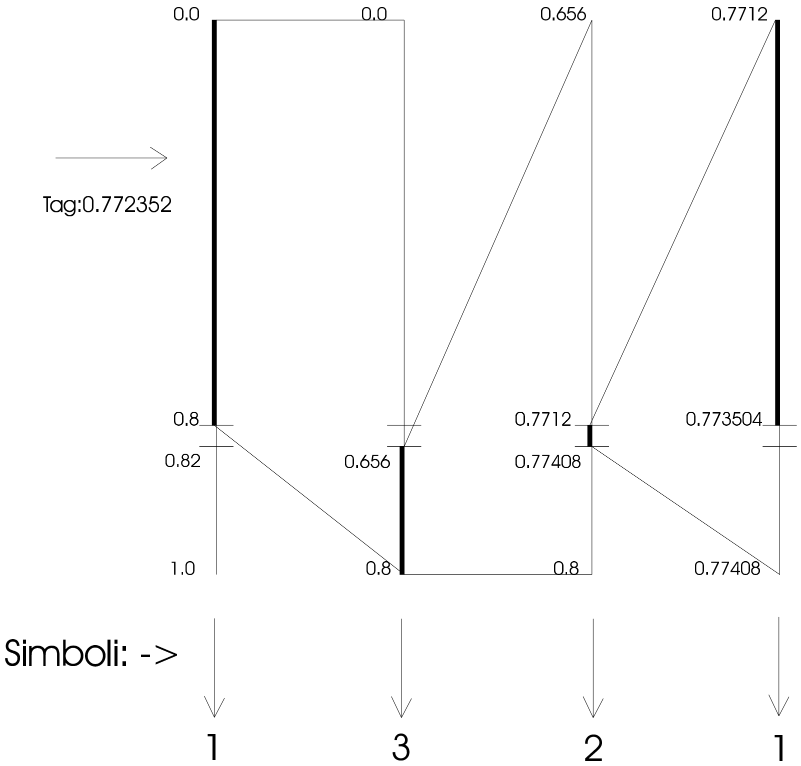


Figure 2: Decifrazione Tag

è sufficiente confrontare il tag con gli estremi dell'intervallo corrispondente a ogni simbolo.

Trovato l'intervallo abbiamo il primo simbolo della sequenza. Si procede come illustrato nella Figura 2, spostandosi da sinistra verso destra.
Si modificano gli estremi e si aggiornano gli intervalli corrispondenti ad ogni simbolo specularmente a quanto fatto durante la codifica, e si procede cercando nuovamente l'intervallo in cui è contenuto il tag. Ci si arresta quando abbiamo tutti i simboli che ci interessano. Questa informazione deve essere preventivamante mandata al decoder (p.e. scrivendola nell'intestazione del file compresso, oppure trasmettendola prima di cominciare la trasmissione dei dati veri e propri).

Fin qui abbiamo trattato i tag come numeri reali, mentre a noi serve un codice binario. Niente di più immediato: basta prendere la rappresentazione binaria della parte decimale del tag (ricordo che il tag compreso tra 0 e 1, con 1 escluso) e troncarla alla lunghezza di

$$l(x) = \left\lceil log\frac{1}{P(x)}\right\rceil + 1bit$$

dove x è il simbolo del quale vogliamo ottenere il tag.

Sebbene troncato, il codice è univoco. Infatti, ricordo che il tag può essere uno qualunque dei valori nell'intervallo $[F_x(x-1),\ F_x(x))$, per cui basta mostrare che il tag di x troncato a $l(x)$ bit è compreso in tale intervallo.

Inoltre il codice é anche univocamente decifrabile, in quanto é un *prefix code* (vedi paragrafo 2 *"Cenni di teoria dell'informazione"*).

Per la dimostrazione di queste due proprietà, vi rimando a **[1]** .

4.1 Aritmetica a precisione finita

Il metodo è sicuramente interessante, tuttavia balza subito agli occhi un difetto che lo rende impraticabile così com'è: necessita di un'aritmatica a precisione infinita. Infatti, man mano che la sequenza si allunga, l'intervallo in cui è compreso il tag diventa sempre più piccolo e quando i due estremi convergono, a causa della precisione finita della macchina, perdiamo ogni informazione. È comunque possibile evitare ciò con un ingegnoso metodo: la scalatura.

Quando, durante la codifica o la decodifica, ridimensioniamo l'intervallo contenete il tag a causa dell'arrivo di un nuovo simbolo, si presentano 4 casi:

1. l'intervallo è compreso in $[0,\ 0.5)$
2. l'intervallo è compreso in $[0.5,\ 1)$
3. l'intervallo è compreso in $[0.25,\ 0.75)$
4. l'intervallo è più grande della metà dell'intevallo $[0,\ 1)$

Nel caso 4 abbiamo un intervallo abbastanza grande da permetterci di continuare senza difficoltà.

Nei casi 1 e 2 provvediamo a una scalatura ragionando come segue.

Se l'intervallo è compreso in $[0,\ 0.5)$, il tag è per sempre confinato nella metà inferiore dell'intervallo unitario, e possiamo notificare ciò al decodificatore trasmettendo subito (o salvando su file) una cifra binaria 0.

Possiamo ora raddoppiare la grandezza dell'intervallo e renderlo più grande e meno soggetto ad errori di underflow: così facendo perdiamo l'informazione sulla cifra più significativa, ma questa informazione é già stata mandata al decodificatore. Si ricade dunque nei 4 casi considerati in precedenza. Possiamo applicare ricorsivamente il procedimento fino a quando la condizione 4 non soddisfatta.

Per ridimensionare l'intervallo, si applica il seguente mapping:

$$E_1 \ = \ [0,\ 0.5) \to [0,\ 1) \qquad E_1(x) = 2x$$

Analogamente, se l'intervallo è compreso in $[0.5,\ 1)$, il tag è per sempre confinato nella metà superiore dell'intervallo, e possiamo notificare ciò al decodificatore trasmettendo una cifra binaria 1.

Per ridimensionare l'intervallo, si applica il seguente mapping:

$$E_2 \ = \ [0.5,\ 1) \to [0,\ 1) \qquad E_2(x) = 2(x - 0.5)$$

Leggermente diverso è il terzo caso. Se ricadiamo in questo caso, i 2 bit + significativi della parte decimale del tag, in binario, devono essere $(0.01...)_2$ per l'estremo inferiore e $(0.10...)_2$ per quello superiore.

L'intervallo viene raddoppiato tramite il seguente mapping:

$$E_3 \ = \ [0.25,\ 0.75) \rightarrow [0,\ 1) \qquad E_3(x) = 2(x - 0.25)$$

Rimane il compito di stabilire come trasmettere al decodificatore l'informazione che è avvenuto un mapping E_3.
Supponiamo che dopo la scalatura E_3 avvenga una scalatura E_2 perchè l'intervallo del tag è confinato nella metà superiore dell'intervallo unitario; se non ci fosse stata la mappatura E_3, l'intervallo del tag sarebbe stato compreso tra $[0.5,\ 0.75)$, così , dopo la E_2, il tag si sarebbe trovato confinato nella metà inferiore dell'intervallo unitario e ci sarebbe stata una scalatura E_1.
La Figura 3 illustra e chiarisce questo procedimento.

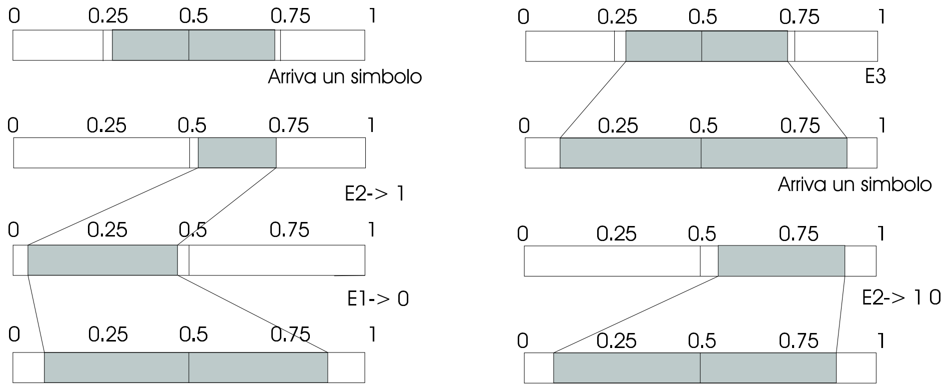


Figure 3: Scalatura senza mapping E_3 e con

Se invece dopo la scalatura E_3 ha luogo un mapping E_1, accade esattamente il contrario: dopo la E_1 ha luogo sicuramente una E_2.

La scalatura E_3 può quindi venire simulata tramite E_1 e E_2. Ma se la terza condizione si verifica più volte consecutivamente? Basta tener traccia del numero di volte che questa avviene, quindi alla prima E_1 o E_2 si spediscono *0* oppure *1* seguiti da tanti *1* o tanti *0* quante sono le mappature avvenute.

4.2 Implementazione

Quella descritta finora è un'implementazione dell'arithmetic coding per aritmetica a virgola mobile.

Sebbene i computer odierni abbiamo FPU molto avanzate, sono ancora più veloci per quanto riguarda l'aritmetica intera.

Per questo motivo si è scelto di implementare, nel progetto, una variante ad aritmetica intera del precedente algoritmo. I meccanismi sono gli stessi, semplicemente l'intervallo $[0,\ 1)$ fino ad ora utilizzato viene mappato sull'intervallo $[00...0, 11...1]$ dove 0 e 1 sono ripetuti m volte se la lunghezza delle parola scelta è di m bit.

Nel progetto, la parola scelta è una double-word, cioè una parola di lunghezza pari a *32 bit*. Questo perché la maggior parte dei calcolatori odierni ha un'architettura a 32 bit, e quindi opera alla massima velocità se gli operandi sono di tale lunghezza. L'implementazione intera dell'algoritmo porta anche altri vantaggi dal punto di vista computazionale: la verifica delle condizioni in cui effettuare i vari mapping si riduce a un confronto fra bit e le moltiplicazioni per due a operazioni di shifting.

Per verificare che valga la condizione ***1.*** o ***2.***, basta infatti confrontare il bit più significativo (MSB, most significant bit) dell'estremo inferiore dell'intervallo (l nel codice) con l'MSB dell'estremo superiore (u nel codice): se sono entrambi pari a 1, si ha la seconda condizione, se sono pari a 0, si ha la prima.

Per esempio, scegliendo la lunghezza di 8 bit per la parola, l'intervallo [0.0625, 0.2) diventa

$$\begin{array}{rcc} \mathrm{l} = & 0.0625 & (00001000)_2 \\ \mathrm{u} = & 0.2 & (00110011)_2 \end{array}$$

I 2 MSB sono uguali, quindi vale la condizione ***1.*** e posso applicare il mapping E_1.

Per quanto riguarda la terza condizione, basta verificare che i bit più significativi di l e u siano rispetivamente 0 e 1, e che i secondi più significativi siano 1 e 0.

Un'ultima considerazione va fatta relativamente alla funzione di probabilità. Essa va sostituita in modo da poter operare totalmente in aritmetica intera.
Definendo n_j come il numero di volte che il simbolo j appare nella sequenza di lunghezza *Total Count*, F_x puó essere espressa come:

$$F_x(k) = \frac{\Sigma_{i=1}^{k} \quad n_i}{Total\ Count}$$

Definendo

$$Cum\_Count(k) = \Sigma_{i=1}^{k} \quad n_i$$

possiamo riscrivere le equazioni a e b come

$$l_{(k)} = \left\lfloor \frac{l_{(k-1)} + (u_{(k-1)} - l_{(k-1)} + 1) * Cum\_Count(x_k - 1)}{Total\ Count} \right\rfloor$$

$$u_{(k)} = \left\lfloor \frac{l_{(k-1)} + (u_{(k-1)} - l_{(k-1)} + 1) * Cum\_Count(k_k)}{Total\ Count} \right\rfloor - 1$$

I vari + 1 e -1 introdotti sono necessari in quanto stiamo lavorando in aritmetica intera.

5 Modellizzazione della sorgente

Per modello si intende semplicemente il modo di calcolare, in un dato contesto, le probabilità di presentazione relative al simbolo successivo. La soluzione più semplice consiste nell'impiego di un modello fisso, ad esempio una tabella con le frequenze standard di presentazione delle lettere in un testo scritto, da cui estrarre le probabilità a priori.

Un miglioramento di questa tecnica è l'uso di un modello adattivo, cioè di un modello che si adatti ai dati da comprimere durante la compressione stessa. Tale soluzione risulta, in generale, non meno efficiente di un modello esatto del messaggio da codificare (ossia di un modello in cui le probabilità di presentazione coincidano con le esatte frequenze nel messaggio), e inoltre non richiede la trasmissione del modello stesso al decodificatore. Un modello fisso si può trasformare in adattivo semplicemente modificando la frequenza di presentazione dopo la codifica di ogni nuovo simbolo.

In realtà si può fare di meglio: infatti l'uso delle probabilità di simbolo non rappresenta una scelta particolarmente buona per la stima dell'effettiva entropia, in quanto prescinde completamente dalle probabilità intersimboliche. Solo sfruttando tali correlazioni è possibile creare modelli particolarmente efficienti.

Tali modelli sfruttano il contesto (cioè un intorno dei simboli precedenti) per ridurre le probabilità priori, e prendono il nome di *modelli predittivi.*

5.1 Impementazione dei modelli

Ho scelto di implementare i seguenti modelli:

- `adaptive_model`:

  Un semplice modello adattivo

- `difference_adaptive_model`:

  Una rivisitazione del modello adattivo; viene codificata la differenza fra due simboli consecutivi e viene mantenuta la funzione distribuzione di questa grandezza. Adatta per sorgenti che presentano forte regolarit locale (immagini e suono, non testo)

- `ppm_model`

  Modello che implementa il ***"prediction with partial match"*** (predizione con corrispondenza parziale), algoritmo impegnativo dal punto di vista computazionale ma che garantisce ottimi risultati (vedi paragrafo 6.1 ***"Prediction with Partial Match"***)

- `jpeg_ls_model` e `jpeg_ls_24_model`

  Modelli che implementano l'algoritmo predittivo per la compressione delle immagini jpeg-ls (jpeg lossless, parente del ben più noto jpeg lossy), rispettivamente per immagini a *8bpp* basate su palette e per immagini true-color a *24bpp* (bits per pixel)

6 Algoritmi predittivi

Come specificato più volte in precedenza, si ottiene compressione maggiore se le probabilità di presentazione dei simboli sono fortemente differenti, cioè se nella sequenza di simboli da codificare alcuni di questi sono molto più probabili di altri. Ci sono differenti modi di ottenere questa propietà, per esempio trasformando la sequenza in una avente la proprità richiesta, oppure usando per ogni simbolo una distribuzione di probabilità differente. E' però necessario che il procedimento sia reversibile e interpretabile dal decoder (non dimentichiamo che dopo aver compresso i nostri dati è desiderabile poterli decodificare...).

Se la trasformazione o le varie distribuzioni sono basate sulla storia passata della sequenza, che sia conosciuta sia al codificatore che al decodificatore, non c'è la necessità di trasmette informazioni aggiuntive.

Algoritmi che si basano sulla storia passata per determinare il codice del nuovo simbolo, prevedendo la sua probabilità, si chiamano schemi di codifica predittivi.

Quando si codifica un testo, è possibile prevedere la probabilità dei caratteri in maniera molto semplice. Quando codifichiamo un particolare simbolo, vorremmo utilizzare per questo una distribuzione di probabilità nella quale il simbolo è molto probabile. Comunque, se utilizziamo una distribuzione differente per ogni simbolo, dobbiamo mettere a conoscenza il decodificatore di quale ddp è stata utilizzata. Un modo di farlo senza costo aggiuntivo è di usare i simboli precedenti per prevedere il modello di probabilità.

I simboli precedenti formano il ***contesto*** nel quale il simbolo è codificato. Pensiamo per esempio di codificare la parola *"colazione"*.

La **"e"** finale, se considerata da sola, ha probabilità pari a circa 0.06 di presentarsi. Se usiamo l'informazione che la lettera precedente è **"n"** questo riduce la probabilità di molte lettere (come la **"b"**, la **"p"**, la **"q"** e la **"m"**) a tutto vantaggio delle altre, compresa la **"e"**.

In questo esempio, **"n"** è il contesto di ordine uno di **"e"**, **"on"** quello di ordine due e così via. Più lettere usiamo per definire il contesto, più forte sarà (nel caso medio) la probabilità del simbolo da codificare (basti pensare alla probabilità della nostra **"e"** nel contesto di **"zion"**).

Lo schema di codifica qui esposto è molto potente, ma così esposto non è utilizzabile; se dovessimo mantenere in memoria tutti i possibili contesti, avremmo una quantità incredibile di informazioni.

Se la grandezza dell'alfabeto fosse di 256 simboli (come nel caso della codifica ASCII) e usassimo un contesto di ordine cinque, avremmo bisogno di 256^5 distribuzioni di probabilità, più di *mille miliardi*! Questo problema è risolto in modo semplice ed elegante dall'approccio *"Prediction with Partial Match"*

6.1 Prediction with partial match

L'idea dietro al *ppm* è semplice: ci piacerebbe avere contesti grandi per avere probabilità più alte per il simbolo da decodificare, ma non possiamo sopportare l'onere di stimare e conservare tutte le ddp.

Invece di stimare le probabilità in anticipo, possiamo ridurre il carico di lavoro stimandole via via che si presentano, memorizzando solo quei contesti che sono realmente apparsi nella sequenza codificata. Dobbiamo però considerare il fatto che, specie all'inizio del processo, ci capiti di dover codificare lettere che non sono mai apparse.

L'alfabeto deve dunque includere un simbolo di *escape* per far fronte a questa situazione. L'algoritmo ha così a disposizione contesti di vari ordini (la dimensione del più grande deve essere comunque predeterminata), tra i quali poter scegliere.

Supponiamo che la dimensione del contesto più grande sia 5: l'algoritmo cercherà la lettera attuale nel contesto attuale di dimensione 5. Se la lettera non è presente in tale contesto, viene mandato un *escape* e la dimensione del contesto viene ridotta (si cerca nel contesto di dimensione 4).

Si continua così finché non si trova un contesto nel quale il simbolo era già stato incontrato in precedenza o si riscontra che il simbolo non è presente in nessun contesto. In questo caso, viene usato un particolare contesto (detto di *ordine -1*) in cui tutti i simboli sono equiprobabili.

Naturalmente, dopo la codifica, i vari contesti devono essere aggiornati incrementando il conteggio delle lettere già presenti ed inserendo la nuova lettera nei contesti che non la contenevano.

Un piccolo esempio può aiutare a chiarire il procedimento. Per semplicità prendiamo un ordine massimo di due (altrimenti l'esempio rischierebbe di diventare confusivo!)

La sequenza da codificare è:

cala la lana

Supponiamo di aver codificato le lettere fino a *"cala la"*, ci ritroviamo con le seguenti tabelle per ogni ordine.

Array dei conteggi per il contesto di ordine -1

lettera	frequenza	frequenza marginale
c	1	1
a	1	2
l	1	3
ø	1	4
n	1	5
totale:		5

Array dei conteggi per il contesto di ordine 0

lettera	frequenza	frequenza marginale
c	1	1
a	3	4
l	2	5
ø	1	6
totale:		6

Array dei conteggi per il contesto di ordine 1

contesto	lettera	frequenza	frequenza marginale
c	a	1	1
	$< esc >$	1	2
totale:			2
a	l	1	1
	ø	1	2
	$< esc >$	1	3
totale:			3
l	a	2	2
	$< esc >$	1	3
totale:			3
ø	l	1	1
	$< esc >$	1	2
totale:			2

Array dei conteggi per il contesto di ordine 2

contesto	lettera	frequenza	frequenza marginale
ca	l	1	1
	$< esc >$	1	2
totale:			2
al	a	1	1
	$< esc >$	1	2
totale:			2
la	ø	1	1
	$< esc >$	1	2
totale:			2
aø	l	1	1
	$< esc >$	1	2
totale:			2
øl	a	1	1
	$< esc >$	1	2
totale:			2

La prossima lettera da codificare è *"ø"*. Partiamo dal contesto di ordine maggiore, cioè 2. Il contesto di ordine 2 della lettera corrente è **"la"**, quindi cerco nell'array dei conteggi per il contesto di ordine 2 se questo è presente. Lo è, quindi posso codificare la lettera *"ø"* con probabilità di ben 0.5, contro il circa 0.04 che si avrebbero con un modello fisso o adattivo!
Gli array aggiornati hanno i seguenti conteggi:

Array dei conteggi per il contesto di ordine -1

lettera	frequenza	frequenza marginale
c	1	1
a	1	2
l	1	3
ø	1	4
n	1	5
totale:		5

Array dei conteggi per il contesto di ordine 0

lettera	frequenza	frequenza marginale
c	1	1
a	3	4
l	2	5
ø	1	7
totale:		7

Array dei conteggi per il contesto di ordine 1

contesto	lettera	frequenza	frequenza marginale
c	a	1	1
	$< esc >$	1	2
totale:			2
a	l	1	1
	ø	2	3
	$< esc >$	1	4
totale:			4
l	a	2	2
	$< esc >$	1	3
totale:			3
ø	l	1	1
	$< esc >$	1	2
totale:			2

Array dei conteggi per il contesto di ordine 2

contesto	lettera	frequenza	frequenza marginale
ca	l	1	1
	$< esc >$	1	2
totale:			2
al	a	1	1
	$< esc >$	1	2
totale:			2
la	ø	2	2
	$< esc >$	1	3
totale:			3
aø	l	1	1
	$< esc >$	1	2
totale:			2
øl	a	1	1
	$< esc >$	1	2
totale:			2

Nell'esempio è stata usata la frequenza di 1 per il simbolo di escape.

Non c'è un metodo unico, ne tantomeno un metodo ovvio, per scegliere il conteggio di tale simbolo. Quello da me scelto nell'implementazione del *ppm* è stato di conteggiare il numero di lettere finora inserite nel contesto e di aumentarlo di uno per ottenere la frequenza dell'escape (questa è una variazione del cosiddetto *metodo C*, che consente di ottenere in media prestazioni migliori. Per la cronaca, il metodo usato nell'esempio è detto *metodo A*).

7 Prove comparative di performance

Ho fatto alcune prove con i vari modelli per misurare le loro performance e confrontarli con alcuni metodi di compressione conosciuti. Di seguito sono elencati i risultati.

Figure 4: L'immagine di prova lena.bmp

Compressione dell'immagine lena

Immagine	Dimensione (in bytes)
originale	263,222
adaptive	250,203
differential adaptive	199,642
jpeg_ls	**166,705**
gif (algoritmo LZW)	278,592
zip (algoritmo LZW)	235,373

Compressione del file di testo part1.txt

File	Dimensione (in bytes)
originale	169,778
adaptive	106,977
ppm	**59,017**
zip (algoritmo LZW)	61,908

Compressione del file di testo divina.txt

File	Dimensione (in bytes)	Ratio
originale	562,935	1.0
adaptive	316,834	1.78
ppm	**188,824**	**2.98**
zip (algoritmo LZW)	227,869	2.47

Qualche parola va spesa su questi risultati. Innanzitutto, l'immagine *lena.bmp* è usata da anni per misurare le performance dei compressori perché è piuttosto difficile da comprimere (vedere `comp.compression.faq`). Presenta forte entropia e scarse zone omogenee. I compressori basati su dizionario (come l'algoritmo LZW) danno il loro meglio con files che presentano grandi zone omogenee o pattern (come nelle immagini di computer graphics o il testo).

E' da sottolieneare l'ottimo risultato ottenuto con il modello *ppm*, che batte il ben piú conosciuto *zip*.

Va comunque detto che dalla parte del secondo c'è la maggiore velocità di esecuzione.

Compressione dell'immagine 8bpp mountain

Immagine	**Dimensione (in bytes)**
originale	307,694
jpeg-ls	**214,190**
gif (algoritmo LZW)	237,790

Compressione dell'immagine 24bpp tulip

Immagine	**Dimensione (in bytes)**
originale	1,179,702
jpeg-ls	**725,590**

Figure 5: L'immagine di prova tulip.bmp

Si può notare che l'algoritmo batte i metodi tradizionali basati su dizionario in tutte le occasioni.
Nel caso di immagini true color a 24 bit, i risultati (come si vede nelle seguenti tabelle) sono ancora migliori.

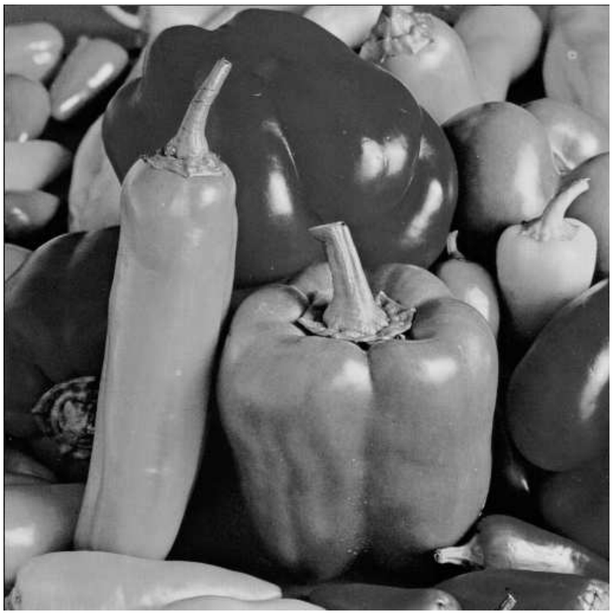


Figure 6: L'immagine di prova peppers.bmp

Compressione dell'immagine 8bpp peppers

Immagine	Dimensione (in bytes)	Ratio
originale	263,118	1.0
jpeg-ls	**160,514**	**1.64**
zip (algoritmo LZW)	230,091	1.14
gif (algoritmo LZW)	265,732	0.99

Compressione dell'immagine 24bpp peppers

Immagine	Dimensione (in bytes)	Ratio
originale	786,486	1.0
jpeg-ls	**447,792**	**1.76**
zip (algoritmo LZW)	678,219	1.15

8 Descrizione delle classi

8.1 Classe model

Nella fase di design del progetto sono state definite le varie classi necessarie a formare il modello a oggetti del sistema. L'accento è stato messo sulla classe descrivente la modellizzazione dei dati (**class model**$< T >$, file *model.h*), che rappesenta il punto di etendibilità del modello e che fornisce attraverso le sue funzioni ***encode_symbol()*** e ***decode_symbol()*** il "motore" dell'intero sistema.

$\langle$*model.h*$\rangle\equiv$

```
#ifndef MODEL_H
#define MODEL_H

#include "common.h"

template <typename symbolT>
class model
{
public:
  virtual void encode_symbol(symbolT symbol) = 0;
  virtual symbolT decode_symbol() = 0;
};

#endif
```

E' questa classe (e le sue derivate) che richiamano e usano l'arithmetic coder e l'arithmetic decoder (classi **class aritmetic_encoder** e **class aritmetic_decoder**, files *arithmetic_encoder.h* e *arithmetic_encoder.h*).

8.2 Classi arithmetic_encoder e arithmetic_decoder

Queste due classi forniscono pochi semplici metodi per la compressione/decompressione usando l'arithmetic coding sono totalmente indipendenti dal modello, garantendo la massima espandibilità e riusabilità.

⟨*arithmetic-encoder.h*⟩≡

```
#ifndef ARITHMETIC_ENCODER_H
#define ARITHMETIC_ENCODER_H

#include <math.h>

#include "common.h"
#ifdef WINDOWS
#include "file_writer.h"
#else
#include "u_file_writer.h"
#endif

/////////////////////////////////////////////////////////

class arithmetic_encoder
{
private:

  file_writer<bit>* m_output;
  const DWORD top_value;
  DWORD l, u;
  int scale3;

  arithmetic_encoder& operator=(const arithmetic_encoder&);
  arithmetic_encoder(const arithmetic_encoder& );

public:

  arithmetic_encoder(file_writer<bit>* fw) : top_value(0xFFFFFFFF),
                                             l(0),
                                             u(top_value),
                                             scale3(0)
  {
    m_output = fw;
```

```
}

inline void encode(DWORD lower_bound, DWORD upper_bound, DWORD total)
{
  bool e3_cond = false;
  bool eql_cond;

  QWORD range = (QWORD)(u - l) + 1;

  ASSERT(l < u);
  ASSERT((lower_bound <= upper_bound) && (upper_bound <= total));

  u = l + (DWORD)(((range * upper_bound) / total) - 1);
  l = l + (DWORD)((range * lower_bound) / total);

  while( (eql_cond = (MSB(u) == MSB(l))) ||
         (e3_cond  = (M2SB(l) == 1) && (M2SB(u) == 0)) )
  {
    if(eql_cond)
    {
      bit bit_to_send = ((MSB(u) == 0) ? 0 : 1);
      m_output->send(bit_to_send);

      l <<= 1;
      u <<= 1;
      u |= 1;

      while (scale3 > 0)
      {
        m_output->send(!bit_to_send);
        --scale3;
      }
    }
    else if (e3_cond)
    {
      l <<= 1;
      u <<= 1;
      u |= 1;

      l += MSB_MASK;
      u += MSB_MASK;

      ++scale3;
    }
  }
```

```
    ASSERT(l < u);
  }

  inline void encode_eof()
  {
    for (size_t i = 0; i < sizeof(DWORD) * 8; ++i)
    {
      bit bit_to_send = ((MSB(l) == 0) ? 0 : 1);
      m_output->send(bit_to_send);

      l <<= 1;
      while (scale3 > 0)
      {
        m_output->send(!bit_to_send);
        --scale3;
      }
    }
  }
};

////////////////////////////////////////////////////////

#endif //ARITHMETIC_ENCODER_H
```

⟨*arithmetic-decoder.h*⟩≡

```
#ifndef ARITHMETIC_DECODER_H
#define ARITHMETIC_DECODER_H

#include <math.h>

#include "common.h"

#ifdef WINDOWS
#include "file_map.h"
#else
#include "u_file_map.h"
#endif

#include "model.h"

//#pragma warning(push, 4)

/////////////////////////////////////////////////////////

class arithmetic_decoder
{
private:
  file_map<bit>* m_input;
  const DWORD top_value;
  DWORD l, u, tag;

  arithmetic_decoder& operator=(const arithmetic_decoder&);
  arithmetic_decoder(const arithmetic_decoder& );

public:
  arithmetic_decoder(file_map<bit>* fm): top_value(0xFFFFFFFF),
                                         l(0),
                                         u(top_value),
                                         tag(0)
  {
    m_input = fm;
    for (size_t i = 0; i < (sizeof(tag) * 8); ++i)
    {
      tag <<= 1;
      tag += m_input->read_next();
    }
  }
```

```
inline void decode(DWORD lower_bound, DWORD upper_bound, DWORD total)
{
  bool eql_cond, e3_cond;
  QWORD range = (QWORD)u - l + 1;

  ASSERT(l < u);
  ASSERT((lower_bound <= upper_bound) && (upper_bound <= total));

  u = l + (DWORD)(((range * upper_bound) / total) - 1);
  l = l + (DWORD)((range * lower_bound) / total);

  while( (eql_cond = (MSB(u) == MSB(l))) ||
         (e3_cond  = ((M2SB(l) == 1) && (M2SB(u) == 0))) )
  {
    if(eql_cond)
    {
      l <<= 1;
      u <<= 1;
      tag <<= 1;

      u += 1;

      bit b = m_input->read_next();
      ASSERT (((tag + b)  == tag + 1) || ((tag + b) == tag));
      tag += b;
    }
    else if (e3_cond)
    {
      l <<= 1;
      u <<= 1;
      tag <<= 1;
      u += 1;

      bit b = m_input->read_next();
      ASSERT (((tag + b) == tag + 1) || ((tag + b) == tag));
      tag += b;

      l += MSB_MASK;
      u += MSB_MASK;
      tag += MSB_MASK;
    }
  }
  ASSERT(l < u);
}
```

```
  inline DWORD decode_target(DWORD total)
  {
    DWORD ret =  (DWORD)((((QWORD)(tag - l) + 1) * total - 1) / ((QWORD)u - l + 1));
    return (ret);
  }
};

#endif //ARITHMETIC_DECODER_H

////////////////////////////////////////////////////////
```

Tutto quello che deve fare un modello è di trovare il range della F, la funzione cumulativa di probabilità, corrispondente al simbolo in esame, e poi l'arithmetic coder si preoccupo di codificare tale range al fine di produrre un tag univoco.

È possibile così usare ddp diverse per codificare lo stesso file. Questa caratteristica permette di implementare in modo semplice ed efficiente il modello *ppm* (Prediction with Partial Match, vedi sopra)

8.3 Gestione I/O

È stata separata dal modello e dal coder anche la gestione I/O, implementata in 6 classi a parte. Questo permette portabilità ed efficienza (sono state realizzate versioni specifiche per POSIX/LINUX (classi **class u_file_map**$< T >$ e **class u_file_writer**$< T >$, files *u_file_map.h* e *u_file_writer.h*) e per Win32 (classi **class file_map**$< T >$ e **class file_writer**$< T >$, files *file_map.h* e *file_writer.h*).

⟨*u-file-map.h*⟩≡

```
#ifndef FILE_MAP_H
#define FILE_MAP_H

#define _POSIX_MAPPED_FILE

#include <unistd.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

  class bad_file { };

  template<class T>
  class file_map
  {
  protected:
    int m_fdFile;

    T* m_DataMap;
    BYTE* m_FileMap;

    DWORD m_dwFileSize;
    DWORD m_iCurrentElem;
    DWORD m_dwSizeOfHeader;
    bool m_bEof;
    bool m_bHand;

    file_map<T>& operator=(const file_map<T>&);
```

```
  file_map(const file_map<T>&);

public:

  file_map(DWORD dwSizeOfHeader = 0): m_fdFile(-1),
                                      m_dwFileSize(0),
                                      m_iCurrentElem(0),
                                      m_dwSizeOfHeader(dwSizeOfHeader),
                                      m_bEof(false)
  {

  }

  file_map(char* szFileName, DWORD dwSizeOfHeader = 0, DWORD dwFileSize = 0):

                              m_fdFile(-1),
                              m_dwFileSize(dwFileSize),
                              m_iCurrentElem(0),
                              m_dwSizeOfHeader(dwSizeOfHeader),
                              m_bEof(false)
  {
    if(!this->create(szFileName, dwFileSize))
      throw new bad_file();
  }

  bool create(char* szFileName, DWORD dwFileSize = 0)
  {
    m_fdFile = open(szFileName, O_RDWR);

    ASSERT(m_fdFile != -1);
    if (m_fdFile != -1)
    {
      struct stat stFile;
      fstat(m_fdFile, &stFile);

      if (dwFileSize == 0)
      {
        m_dwFileSize = stFile.st_size;
        m_FileMap = (BYTE*)mmap(0, m_dwFileSize, PROT_READ | PROT_WRITE,
                                MAP_SHARED, m_fdFile, 0);
        m_bHand = false;
      }
      else
      {
        m_dwFileSize = dwFileSize;
```

```
      m_FileMap = (BYTE*)malloc(m_dwFileSize);

      ASSERT(m_FileMap != NULL);
      if (m_FileMap == NULL)
        m_FileMap = (BYTE*)MAP_FAILED;

      /*m_FileMap = (BYTE*)mmap(0, m_dwFileSize, PROT_READ | PROT_WRITE,
                                MAP_ANON, -1, 0);*/

      read(m_fdFile, m_FileMap, stFile.st_size);
      m_bHand = true;
    }

    m_DataMap = (T*)(m_FileMap + m_dwSizeOfHeader);

    ASSERT(m_FileMap != MAP_FAILED);
    if (m_FileMap == MAP_FAILED)
    {
      throw new bad_alloc();
    }
    else
    {
      m_iCurrentElem = 0;
      return true;
    }
  }
  return false;
}

inline T read_next()
{
  return(m_DataMap[m_iCurrentElem++]);
}

inline T operator[](size_t i)
{
  return (m_DataMap[i]);
}

inline DWORD get_size()
{
  struct stat stFile;

  fstat(m_fdFile, &stFile);
```

```
    return(stFile.st_size);
  }

  inline bool eof()
  {
    return (m_dwFileSize <= ((m_iCurrentElem + 1) * sizeof(T) + m_dwSizeOfHeader));
  }

  inline BYTE* get_header()
  {
    return (m_FileMap);
  }

  ~file_map()
  {
    if (m_bHand)
    {
      lseek(m_fdFile, 0, SEEK_SET);
      write(m_fdFile, m_FileMap, m_dwFileSize);
      free(m_FileMap);
    }
    else
    {
      munmap(m_FileMap, m_dwFileSize);
    }
    close(m_fdFile);
  }
};

template<>
class file_map<bit>
{
private:
  int m_fdFile;

  BYTE* m_DataMap;
  BYTE* m_FileMap;

  DWORD m_dwFileSize;

  DWORD m_iCurrentByte;
  DWORD m_nBitsInCurrentByte;
  BYTE m_CurrentByte;

  DWORD m_dwSizeOfHeader;
```

```
  bool m_bEof;

  file_map<bit>& operator=(const file_map<bit>&);
  file_map(const file_map<bit>&);

public:

  class bad_file { };

  file_map(DWORD dwSizeOfHeader = 8): m_fdFile(-1),
                                      m_dwFileSize(0),
                                      m_iCurrentByte(0),
                                      m_nBitsInCurrentByte(0),
                                      m_CurrentByte(0),
                                      m_dwSizeOfHeader(dwSizeOfHeader),
                                      m_bEof(false)
  {

  }

  file_map(char* szFileName, DWORD dwSizeOfHeader = 8, DWORD dwFileSize = 0):

                                m_fdFile(-1),
                                m_dwFileSize(dwFileSize),
                                m_iCurrentByte(0),
                                m_nBitsInCurrentByte(0),
                                m_CurrentByte(0),
                                m_dwSizeOfHeader(dwSizeOfHeader),
                                m_bEof(false)
  {
    if(!this->create(szFileName, dwFileSize))
      throw new bad_file();
  }

  bool create(char* szFileName, DWORD dwFileSize = 0)
  {
    m_fdFile = open(szFileName, O_RDWR);

    ASSERT(m_fdFile != -1);
    if (m_fdFile != -1)
    {
      if (dwFileSize == 0)
      {
        struct stat stFile;
```

```
      fstat(m_fdFile, &stFile);
      m_dwFileSize = stFile.st_size;
    }
    else
    {
      m_dwFileSize = dwFileSize;
    }

            m_FileMap = (BYTE*)mmap(0, m_dwFileSize, PROT_READ | PROT_WRITE,
                                    MAP_PRIVATE, m_fdFile, 0);
    m_DataMap = m_FileMap + m_dwSizeOfHeader;

    ASSERT(m_FileMap != MAP_FAILED);
    if (m_FileMap != MAP_FAILED)
    {
      m_iCurrentByte = 0;
      m_nBitsInCurrentByte = 0;
      m_CurrentByte = 0;
      return true;
    }
  }
  return false;
}

inline bit read_next()
{
  if (m_nBitsInCurrentByte == 0)
  {
    m_CurrentByte = m_DataMap[m_iCurrentByte++];
    m_nBitsInCurrentByte = 8;
  }

  bit retval = (MSB(m_CurrentByte) ? 1 : 0);
  m_CurrentByte <<= 1;
  --m_nBitsInCurrentByte;

  return (retval);
}

inline bit operator[](size_t i)
{
  BYTE TargetByte = m_DataMap[i / 8];
  bit TargetBit = ( ( TargetByte & (1 << ((i % 8) - 1)) ) == 0 ? 0 : 1);
  return TargetBit;
}
```

```
    inline DWORD get_size()
    {
      struct stat stFile;
      fstat(m_fdFile, &stFile);
      return(stFile.st_size);
    }

    inline BYTE* get_header()
    {
      return (m_FileMap);
    }

    inline bool eof()
    {
      return (m_dwFileSize == (m_iCurrentByte + 1 + m_dwSizeOfHeader));
    }

    ~file_map()
    {
      munmap(m_FileMap, m_dwFileSize);
      close(m_fdFile);
    }
  };


#endif //FILE_MAP_H
```

⟨*u-file-writer.h*⟩≡

```
#ifndef FILE_WRITER_H__
#define FILE_WRITER_H__

#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>


  template<class T>
  class file_writer
  {
    private:
      int m_fdFile;
      T* m_tBuffer;

      DWORD m_dwCurrentElem;
      DWORD m_dwElemsInBuffer;

    public:
      file_writer() : m_fdFile(-1)
      {
        m_tBuffer = (T*)malloc(4 * 1024);
        if(m_tBuffer)
        {
          m_dwCurrentElem = 0;
          m_dwElemsInBuffer = (4 * 1024) / sizeof(T);
        }
      }

      inline bool is_good()
      {
        return (m_fdFile != -1);
      }

      inline bool create(char* szFilename)
      {
        m_fdFile = creat(szFilename, 0666);
        return(is_good());
      }
```

```
    inline bool send(T t)
    {
      m_tBuffer[m_dwCurrentElem] = t;
      if (++m_dwCurrentElem == m_dwElemsInBuffer)
      {
        DWORD nBytesWritten = write(m_fdFile, m_tBuffer,
                                    m_dwElemsInBuffer * sizeof(T));

        m_dwCurrentElem = 0;
        ASSERT(nBytesWritten == (m_dwElemsInBuffer * sizeof(T)));
        if (nBytesWritten != (m_dwElemsInBuffer * sizeof(T)))
          return false;
      }
      return true;
    }

    inline void write_header(BYTE* Buffer, int cbSize)
    {
      int nBytesWritten = write(m_fdFile, Buffer, cbSize);

      ASSERT(nBytesWritten == cbSize);
    }

    ~file_writer()
    {
      DWORD nBytesWritten = write(m_fdFile, m_tBuffer, m_dwCurrentElem * sizeof(T));

      ASSERT(nBytesWritten == (m_dwCurrentElem * sizeof(T)));
      close(m_fdFile);
      free(m_tBuffer);
    }
};

template<>
class file_writer<bit>
{
  private:
    int m_fdFile;
    BYTE* m_Buffer;
    BYTE m_bitBuffer;

    DWORD m_dwCurrentByte;

    DWORD m_dwBytesInBuffer;
    DWORD m_dwBitsInByte;
```

```
public:
  file_writer() : m_fdFile(-1)
  {
    m_Buffer = (BYTE*)malloc(4 * 1024);
    if(m_Buffer)
    {
      m_dwCurrentByte = 0;
      m_dwBytesInBuffer = 4 * 1024;
      m_dwBitsInByte = 8;
      m_bitBuffer = 0;
    }
  }

  inline bool is_good()
  {
    return (m_fdFile != -1);
  }

  inline bool create(char* szFilename)
  {
    m_fdFile = creat(szFilename, 0666);
    return(is_good());
  }

  inline bool send(bit b)
  {
    if (m_dwBitsInByte == 0)
    {
      m_Buffer[m_dwCurrentByte] = m_bitBuffer;

      if (++m_dwCurrentByte == m_dwBytesInBuffer)
      {
        DWORD nBytesWritten = write(m_fdFile, m_Buffer, m_dwBytesInBuffer);

        m_dwCurrentByte = 0;

        ASSERT(nBytesWritten == m_dwBytesInBuffer);
        if (nBytesWritten != m_dwBytesInBuffer)
          return false;
      }

      m_dwBitsInByte = 8;
      m_bitBuffer = 0;
    }
```

```
        --m_dwBitsInByte;
        m_bitBuffer <<= 1;

        if (b) m_bitBuffer += 0x1;

        return true;
      }

      inline void write_header(BYTE* Buffer, int cbSize)
      {
        int nBytesWritten = write(m_fdFile, Buffer, cbSize);
        ASSERT(nBytesWritten == cbSize);
      }

      ~file_writer()
      {
        DWORD nBytesWritten;

        nBytesWritten = write(m_fdFile, m_Buffer, m_dwCurrentByte);
        ASSERT(nBytesWritten == m_dwCurrentByte);

        while(m_dwBitsInByte)
        {
          m_bitBuffer <<= 1;
          --m_dwBitsInByte;
        }

        nBytesWritten = write(m_fdFile, &m_bitBuffer, sizeof(BYTE));
        ASSERT(nBytesWritten == sizeof(BYTE));

        close(m_fdFile);
        free(m_Buffer);
      }
  };

#endif //FILE_WRITER_H
```

⟨*file-map.h*⟩≡

```
#ifndef FILE_MAP_H
#define FILE_MAP_H

#ifdef WINDOWS

#include <windows.h>

  class bad_file { };


  template<class T>
  class file_map
  {
  protected:
    HANDLE m_hFile;
    HANDLE m_hFileMapping;

    T* m_DataMap;
    BYTE* m_FileMap;

    DWORD m_dwFileSize;
    DWORD m_iCurrentElem;
    DWORD m_dwSizeOfHeader;
    bool m_bEof;

    file_map<T>& operator=(const file_map<T>&);
    file_map(const file_map<T>&);

  public:

    file_map(DWORD dwSizeOfHeader = 0): m_hFile(INVALID_HANDLE_VALUE),
                                        m_hFileMapping(0),
                                        m_dwFileSize(0),
                                        m_iCurrentElem(0),
                                        m_dwSizeOfHeader(dwSizeOfHeader),
                                        m_bEof(false)
    {

    }

    file_map(TCHAR* szFileName, DWORD dwSizeOfHeader = 0, DWORD dwFileSize = 0):

                                        m_hFile(INVALID_HANDLE_VALUE),
```

```
                                    m_hFileMapping(0),
                                    m_dwFileSize(dwFileSize),
                                    m_iCurrentElem(0),
                                    m_dwSizeOfHeader(dwSizeOfHeader),
                                    m_bEof(false)
{
  if(!this->create(szFileName, dwFileSize))
    throw new bad_file();
}

bool create(TCHAR* szFileName, DWORD dwFileSize = 0)
{
  m_hFile = CreateFile(szFileName, GENERIC_READ | GENERIC_WRITE,
                       0, NULL, OPEN_EXISTING,
                       FILE_FLAG_SEQUENTIAL_SCAN | FILE_ATTRIBUTE_NORMAL,
                       NULL);
  ASSERT(m_hFile != INVALID_HANDLE_VALUE);
  if (m_hFile != INVALID_HANDLE_VALUE)
  {
    m_hFileMapping = CreateFileMapping(m_hFile, NULL, PAGE_READWRITE,
                                       0, dwFileSize, NULL);
    ASSERT(m_hFileMapping != NULL);
    if (m_hFileMapping != NULL)
    {
      m_FileMap = (BYTE*)MapViewOfFile(m_hFileMapping,
                                       FILE_MAP_ALL_ACCESS, 0, 0, 0);

      m_DataMap = (T*)(m_FileMap + m_dwSizeOfHeader);

      ASSERT(m_FileMap != NULL);
      if (m_FileMap != NULL)
      {
        m_iCurrentElem = 0;
        m_dwFileSize = GetFileSize(m_hFile, NULL);
        return true;
      }
    }
  }
  return false;
}

inline T read_next()
{
  return(m_DataMap[m_iCurrentElem++]);
}
```

```
  inline T operator[](size_t i)
  {
    return (m_DataMap[i]);
  }

  inline DWORD get_size()
  {
    return(GetFileSize(m_hFile, NULL));
  }

  inline bool eof()
  {
    return (m_dwFileSize <= ((m_iCurrentElem + 1) * sizeof(T) + m_dwSizeOfHeader));
  }

  inline BYTE* get_header()
  {
    return (m_FileMap);
  }

  ~file_map()
  {
    UnmapViewOfFile(m_FileMap);
    CloseHandle(m_hFileMapping);
    CloseHandle(m_hFile);
  }
};

template<>
class file_map<bit>
{
protected:
  HANDLE m_hFile;
  HANDLE m_hFileMapping;

  BYTE* m_DataMap;
  BYTE* m_FileMap;

  DWORD m_dwFileSize;

  DWORD m_iCurrentByte;
  DWORD m_nBitsInCurrentByte;
  BYTE m_CurrentByte;
```

```
  DWORD m_dwSizeOfHeader;
  bool m_bEof;

  file_map<bit>& operator=(const file_map<bit>&);
  file_map(const file_map<bit>&);

public:

  class bad_file { };

  file_map(DWORD dwSizeOfHeader = 8): m_hFile(INVALID_HANDLE_VALUE),
                                    m_hFileMapping(0),
                                    m_dwFileSize(0),
                                    m_iCurrentByte(0),
                                    m_nBitsInCurrentByte(0),
                                    m_CurrentByte(0),
                                    m_dwSizeOfHeader(dwSizeOfHeader),
                                    m_bEof(false)
  {

  }

  file_map(TCHAR* szFileName, DWORD dwSizeOfHeader = 8, DWORD dwFileSize = 0):

                                    m_hFile(INVALID_HANDLE_VALUE),
                                    m_hFileMapping(0),
                                    m_dwFileSize(dwFileSize),
                                    m_iCurrentByte(0),
                                    m_dwSizeOfHeader(dwSizeOfHeader),
                                    m_bEof(false)
  {
    if(!this->create(szFileName, dwFileSize))
      throw new bad_file();
  }

  bool create(TCHAR* szFileName, DWORD dwFileSize = 0)
  {
    m_hFile = CreateFile(szFileName, GENERIC_READ | GENERIC_WRITE,
                         0, NULL, OPEN_EXISTING,
                         FILE_FLAG_SEQUENTIAL_SCAN | FILE_ATTRIBUTE_NORMAL,
                         NULL);
    ASSERT(m_hFile != INVALID_HANDLE_VALUE);
    if (m_hFile != INVALID_HANDLE_VALUE)
    {
      m_hFileMapping = CreateFileMapping(m_hFile, NULL, PAGE_READWRITE,
```

```
                                    0, dwFileSize, NULL);
    ASSERT(m_hFileMapping != NULL);
    if (m_hFileMapping != NULL)
    {
      m_FileMap = (BYTE*)MapViewOfFile(m_hFileMapping,
                                 FILE_MAP_ALL_ACCESS, 0, 0, 0);

      m_DataMap = m_FileMap + m_dwSizeOfHeader;

      ASSERT(m_FileMap != NULL);
      if (m_FileMap != NULL)
      {
        m_iCurrentByte = 0;
        m_nBitsInCurrentByte = 0;
        m_CurrentByte = 0;
        m_dwFileSize = GetFileSize(m_hFile, NULL);
        return true;
      }
    }
  }
  return false;
}

inline bit read_next()
{
  if (m_nBitsInCurrentByte == 0)
  {
    m_CurrentByte = m_DataMap[m_iCurrentByte++];
    m_nBitsInCurrentByte = 8;
  }

  bit retval = (MSB(m_CurrentByte) ? 1 : 0);
  m_CurrentByte <<= 1;
  --m_nBitsInCurrentByte;

  //TRACE("bit in: %i\n", retval);

  return (retval);
}

inline bit operator[](size_t i)
{
  BYTE TargetByte = m_DataMap[i / 8];
  bit TargetBit = ( ( TargetByte & (1 << ((i % 8) - 1)) ) == 0 ? 0 : 1);
  return TargetBit;
```

```
    }

    inline DWORD get_size()
    {
      return(GetFileSize(m_hFile, NULL));
    }

    inline bool write_header(BYTE* pBuffer)
    {
      ASSERT(pBuffer);
      if (pBuffer)
      {
        for (int cbBytes = 0; cbBytes < m_dwSizeOfHeader; ++cbBytes)
                m_FileMap[cbBytes] = pBuffer[cbBytes];
        return true;
      }
      return false;
    }*/

    inline BYTE* get_header()
    {
      return (m_FileMap);
    }

    inline bool eof()
    {
      return (m_dwFileSize == (m_iCurrentByte + 1 + m_dwSizeOfHeader));
    }

    ~file_map()
    {
      UnmapViewOfFile(m_FileMap);
      CloseHandle(m_hFileMapping);
      CloseHandle(m_hFile);
    }
  };

#endif //WINDOWS

#endif //FILE_MAP_H
```

⟨*file-writer.h*⟩≡

```
#ifndef FILE_WRITER_H__
#define FILE_WRITER_H__

#ifdef WINDOWS

#include <windows.h>

  template<class T>
  class file_writer
  {
    private:
      HANDLE m_hFile;
      T* m_tBuffer;

      DWORD m_dwCurrentElem;
      DWORD m_dwElemsInBuffer;

    public:
      file_writer() : m_hFile(INVALID_HANDLE_VALUE)
      {
        m_tBuffer = (T*)VirtualAlloc(NULL, 4 * 1024, MEM_COMMIT, PAGE_READWRITE);
        if(m_tBuffer)
        {
          m_dwCurrentElem = 0;
          m_dwElemsInBuffer = (4 * 1024) / sizeof(T);
        }
      }

      inline bool is_good()
      {
        return (m_hFile != INVALID_HANDLE_VALUE);
      }

      inline bool create(TCHAR* szFilename)
      {
        m_hFile = CreateFile(szFilename, GENERIC_WRITE,
                             0, NULL, CREATE_ALWAYS,
                             FILE_FLAG_SEQUENTIAL_SCAN | FILE_ATTRIBUTE_NORMAL,
                             NULL);
        return(is_good());
      }

      inline bool send(T t)
```

```
    {
      m_tBuffer[m_dwCurrentElem] = t;
      if (++m_dwCurrentElem == m_dwElemsInBuffer)
      {
        DWORD dwBytesWritten;
        BOOL bRet = WriteFile(m_hFile,
                              m_tBuffer,
                              m_dwElemsInBuffer * sizeof(T),
                              &dwBytesWritten, NULL);

        m_dwCurrentElem = 0;
        ASSERT(bRet);
        if (bRet == FALSE)
          return false;
      }
      return true;
    }

    inline void write_header(BYTE* Buffer, int cbSize)
    {
      DWORD dwBytesWritten;

      BOOL bRet = WriteFile(m_hFile,
                            Buffer,
                            cbSize,
                            &dwBytesWritten,
                            NULL);
      ASSERT(bRet);
    }

    ~file_writer()
    {
      DWORD dwBytesWritten;

      BOOL bRet = WriteFile(m_hFile, m_tBuffer,
                            m_dwCurrentElem * sizeof(T),
                            &dwBytesWritten, NULL);

      ASSERT(bRet);
      CloseHandle(m_hFile);
      VirtualFree(m_tBuffer, 0, MEM_RELEASE);
    }
};

template<>
```

```
class file_writer<bit>
{
  private:
    HANDLE m_hFile;
    BYTE* m_Buffer;
    BYTE m_bitBuffer;

    DWORD m_dwCurrentByte;

    DWORD m_dwBytesInBuffer;
    DWORD m_dwBitsInByte;

  public:
    file_writer() : m_hFile(INVALID_HANDLE_VALUE)
    {
      m_Buffer = (BYTE*)VirtualAlloc(NULL, 4 * 1024, MEM_COMMIT, PAGE_READWRITE);
      if(m_Buffer)
      {
        m_dwCurrentByte = 0;
        m_dwBytesInBuffer = 4 * 1024;
        m_dwBitsInByte = 8;
        m_bitBuffer = 0;
      }
    }

    inline bool is_good()
    {
      return (m_hFile != INVALID_HANDLE_VALUE);
    }

    inline bool create(TCHAR* szFilename)
    {
      m_hFile = CreateFile(szFilename, GENERIC_WRITE, 0,
                           NULL, CREATE_ALWAYS,
                           FILE_FLAG_SEQUENTIAL_SCAN | FILE_ATTRIBUTE_NORMAL,
                           NULL);
      return(is_good());
    }

    inline bool send(bit b)
    {
      if (m_dwBitsInByte == 0)
      {
        m_Buffer[m_dwCurrentByte] = m_bitBuffer;
```

```
    if (++m_dwCurrentByte == m_dwBytesInBuffer)
    {
      DWORD dwBytesWritten;

      BOOL bRet = WriteFile(m_hFile,
                            m_Buffer,
                            m_dwBytesInBuffer,
                            &dwBytesWritten,
                            NULL);

      TRACE("Buffer flushed: %d bytes written\n", dwBytesWritten);

      m_dwCurrentByte = 0;
      ASSERT(bRet);
      if ((bRet == 0) || (dwBytesWritten != m_dwBytesInBuffer))
      {
        return false;
      }
    }

    m_dwBitsInByte = 8;
    m_bitBuffer = 0;
  }

  --m_dwBitsInByte;
  m_bitBuffer <<= 1;

  //TRACE("bit_out %i\n", b);

  if (b) m_bitBuffer += 0x1;

  return true;
}

inline void write_header(BYTE* Buffer, int cbSize)
{
  DWORD dwBytesWritten;

  BOOL bRet = WriteFile(m_hFile,
                        Buffer,
                        cbSize,
                        &dwBytesWritten,
                        NULL);
  ASSERT(bRet);
}
```

```
    ~file_writer()
    {
      DWORD dwBytesWritten;

      BOOL bRet = WriteFile(m_hFile,
                            m_Buffer,
                            m_dwCurrentByte,
                            &dwBytesWritten,
                            NULL);
      ASSERT(bRet);

      while(m_dwBitsInByte)
      {
        m_bitBuffer <<= 1;
        --m_dwBitsInByte;
      }

      bRet = WriteFile(m_hFile, &m_bitBuffer,
                       sizeof(BYTE), &dwBytesWritten, NULL);
      ASSERT(bRet);

      CloseHandle(m_hFile);
      VirtualFree(m_Buffer, 0, MEM_RELEASE);
    }
  };
#endif //WINDOWS


#endif //FILE_WRITER_H
```

8.4 Gestione Bitmap

Sono state realizzate anche due classi specifiche per la gestione delle bitmap palettizzate e true color (classi **class bitmap8bpp** e **class bitmap_true**$< T >$, file *bitmap.h*) con le relative classi di supporto (**bitmap_header, rgb_color24, rgb_color32**, ecc., nei files *bitmap.h* e *rgb_color.h*).

⟨*bitmap.h*⟩≡

```
#ifndef BIT_FILE_MAP_H
#define BIT_FILE_MAP_H

#ifdef WINDOWS
#include <windows.h>
#include "file_map.h"
#include "file_writer.h"
#else
#include "u_file_map.h"
#include "u_file_writer.h"
#include "bmpDEF.h"
#endif

#include "rgb_color.h"

//per gli algoritmi di sort
#include <algorithm>

using namespace std;

class wrong_bpp {  };

#pragma pack(1)
class bitmap_header: public BITMAPFILEHEADER, public BITMAPINFOHEADER
{
public:
  inline DWORD height()
  {
    return (biHeight);
  }

  inline DWORD width()
  {
    return (biWidth);
  }

  inline int get_bpp()
  {
```

```
    return (biBitCount);
  }

  inline int get_palette_size()
  {
    if (biBitCount >= 16)
      return 0;
    return (biClrUsed);
  }

  inline int get_struct_size()
  {
    return (biSize + sizeof(BITMAPFILEHEADER));
  }

  inline int get_header_size()
  {
    return (bfOffBits);
  }

  inline DWORD get_image_size()
  {
    return (biHeight * biWidth * biBitCount / 8);
  }

  inline DWORD get_file_size()
  {
    return (bfSize);
  }

  inline DWORD get_compresion_flag()
  {
    return (biCompression);
  }

  inline void set_compression_flag(DWORD dwFlag)
  {
    biCompression = dwFlag;
  }
};
#pragma pack()

class bitmap8bpp: private file_map<BYTE>
{
private:
```

```
/////////////////////////// Dichiarazione classi ausiliarie /////////////////////////////////

struct indexed_rgb
{
  rgb_color32 rgbValue;
  BYTE iValue;
};

class compare_indexed_rgb
{
public:
  int operator() (const indexed_rgb& a, const indexed_rgb& b)
  {
    return (a.rgbValue.to_dword() < b.rgbValue.to_dword());
  }
};

//////////////////////////////// Variabili private //////////////////////////////////////////

bitmap_header* m_pBitmapHeader;

rgb_color32* m_aRgb;
BYTE*        m_aColorIndex;

indexed_rgb m_IndexedPalette[256];
int         m_IndexToGrey[256];

////////////////////////////////// Metodi privati /////////////////////////////////////////

void translate_simil_grey()
{
  //create the rgb_color32 - index array
  for (int i = 0; i < m_pBitmapHeader->get_palette_size(); ++i)
  {
    m_IndexedPalette[i].rgbValue = m_aRgb[i];
    m_IndexedPalette[i].iValue = i;
  }

  //ordina con shell-sort
  stable_sort(m_IndexedPalette,
              m_IndexedPalette + m_pBitmapHeader->get_palette_size(),
                          compare_indexed_rgb());

  for(int j = 0; j < m_pBitmapHeader->get_palette_size(); ++j)
```

```
    {
      m_IndexToGrey[m_IndexedPalette[j].iValue] = j;
    }
  }

  //////////////////////////////// Disbilitazione copia ////////////////////////////////////

  bitmap8bpp(const bitmap8bpp&);
  bitmap8bpp& operator=(const bitmap8bpp&);

public:

  bitmap8bpp(char* szFileName, DWORD dwFileSize = 0)
  {
    if (!this->create(szFileName, dwFileSize))
      throw new bad_file();

    m_pBitmapHeader = (bitmap_header*)m_FileMap;

    if (m_pBitmapHeader->get_bpp() != 8)
      throw new wrong_bpp();

    m_aRgb = (rgb_color32*)(m_FileMap + m_pBitmapHeader->get_struct_size());
    m_aColorIndex = (m_FileMap + m_pBitmapHeader->get_header_size());

    translate_simil_grey();
  }

  inline DWORD height()
  {
    return(m_pBitmapHeader->height());
  }

  inline DWORD width()
  {
    return(m_pBitmapHeader->width());
  }

  inline BYTE get_greyed_pixel(DWORD x, DWORD y)
  {
    if ( (x > (height() - 1)) || (y > (width() - 1)) )
     return 0; //pixel fuori immagine

    return (m_IndexToGrey[m_aColorIndex[x * width() + y]]);
  }
```

```
  inline void set_greyed_pixel(DWORD x, DWORD y, BYTE pixel_value)
  {
    ASSERT((x <= (height() - 1)) && (y <= (width() - 1)));
    if ((x <= (height() - 1)) && (y <= (width() - 1)))
      m_aColorIndex[x * width() + y] = m_IndexedPalette[pixel_value].iValue;
  }

  inline rgb_color32 get_pixel(DWORD x, DWORD y)
  {
    if ( (x > (height() - 1)) || (y > (width() - 1)) )
    {
      //pixel fuori immagine
      rgb_color32 ret;
      return (ret);
    }
    return (m_aRgb[m_aColorIndex[x * width() + y]]);
  }

  inline rgb_color32* get_palette()
  {
    return m_aRgb;
  }

  inline bitmap_header* get_header()
  {
    return m_pBitmapHeader;
  }

  inline int get_palette_size()
  {
    return (m_pBitmapHeader->get_palette_size());
  }
};

template<class RGB_TYPE>
class bitmap_true: private file_map<BYTE>
{
private:

  bitmap_header* m_pBitmapHeader;
  RGB_TYPE*     m_aColorIndex;

  //////////////////////////// Disabilitazione copia //////////////////////////////
```

```
  bitmap_true(const bitmap_true<RGB_TYPE>&);
  bitmap_true<RGB_TYPE>& operator=(const bitmap_true<RGB_TYPE>&);

public:

  bitmap_true(char* szFileName, DWORD dwFileSize = 0)
  {
    if (!this->create(szFileName, dwFileSize))
      throw new bad_file();

    m_pBitmapHeader = (bitmap_header*)m_FileMap;

    if (m_pBitmapHeader->get_bpp() != (sizeof(RGB_TYPE) * 8))
      throw new wrong_bpp();

    if (m_pBitmapHeader->bfType != 0x4d42) // 0x42 = "B" 0x4d = "M"
      throw new bad_file();

    m_aColorIndex = (RGB_TYPE*)(m_FileMap + m_pBitmapHeader->get_header_size());
  }

  inline DWORD height()
  {
    return(m_pBitmapHeader->height());
  }

  inline DWORD width()
  {
    return(m_pBitmapHeader->width());
  }

  inline RGB_TYPE get_pixel(DWORD x, DWORD y)
  {
    if ( (x > (height() - 1)) || (y > (width() - 1)) )
    {
      //pixel fuori immagine
      RGB_TYPE ret;
      return (ret);
    }
    return (m_aColorIndex[x * width() + y]);
  }

  inline void set_pixel(DWORD x, DWORD y, RGB_TYPE rgbColor)
  {
    ASSERT ((x <= (height() - 1)) && (y <= (width() - 1)));
```

```
    if ( (x <= (height() - 1)) && (y <= (width() - 1)) )
      m_aColorIndex[x * width() + y] = rgbColor;
  }

  inline bitmap_header* get_header()
  {
    return m_pBitmapHeader;
  }
};

#endif //BIT_FILE_MAP_H
```

⟨*rgb-color.h*⟩≡

```
#ifndef RGB_COLOR_CLASS_H__
#define RGB_COLOR_CLASS_H__

#ifdef WINDOWS
#include <windows.h>
#else
#include "bmpDEF.h"
#endif

class rgb_color32: public RGBQUAD
{
public:
  rgb_color32(BYTE cRed = 0, BYTE cGreen = 0, BYTE cBlue = 0, BYTE cAlpha = 0)
  {
    rgbRed = cRed;
    rgbGreen = cGreen;
    rgbBlue = cBlue;
    rgbReserved = cAlpha;
  }

  rgb_color32(RGBQUAD rgbNew)
  {
    rgbRed = rgbNew.rgbRed;
    rgbGreen = rgbNew.rgbGreen;
    rgbBlue = rgbNew.rgbBlue;
    rgbReserved = rgbNew.rgbReserved;
  }

  rgb_color32(DWORD dwRGB)
  {
    rgbBlue     = (BYTE)(dwRGB & 0xFF);
    rgbGreen    = (BYTE)((dwRGB >> 8) & 0xFF);
    rgbRed      = (BYTE)((dwRGB >> 16) & 0xFF);
    rgbReserved = (BYTE)((dwRGB >> 24) & 0xFF);
  }

  inline DWORD to_dword() const
  {
    DWORD dwValue = (rgbReserved << 24) +
                    (rgbRed      << 16) +
                    (rgbGreen    << 8) +
                     rgbBlue;
    return dwValue;
```

```
  }
};

inline rgb_color32 operator+(rgb_color32 a, rgb_color32 b)
{
  return rgb_color32(a.rgbRed + b.rgbRed,
                     a.rgbGreen + b.rgbGreen,
                     a.rgbBlue + b.rgbBlue,
                     a.rgbReserved + b.rgbReserved);
}

inline rgb_color32 operator-(rgb_color32 a, rgb_color32 b)
{
  return rgb_color32(a.rgbRed - b.rgbRed,
                     a.rgbGreen - b.rgbGreen,
                     a.rgbBlue - b.rgbBlue,
                     a.rgbReserved - b.rgbReserved);
}

inline rgb_color32 operator*(rgb_color32 a, int b)
{
  return rgb_color32(a.rgbRed * b,
                     a.rgbGreen * b,
                     a.rgbBlue * b,
                     a.rgbReserved * b);
}

inline rgb_color32 operator/(rgb_color32 a, int b)
{
  ASSERT(b != 0);
  return rgb_color32(a.rgbRed / b,
                     a.rgbGreen / b,
                     a.rgbBlue / b,
                     a.rgbReserved / b);
}

inline bool operator==(const rgb_color32 a, const rgb_color32 b)
{
  return (a.to_dword() == b.to_dword());
}

inline bool operator<(const rgb_color32 a, const rgb_color32 b)
{
  return (a.to_dword() < b.to_dword());
}
```

```
inline bool operator>(const rgb_color32 a, const rgb_color32 b)
{
  return (a.to_dword() > b.to_dword());
}

#pragma pack(1)
class rgb_color24
{
public:
  BYTE rgbBlue;
  BYTE rgbGreen;
  BYTE rgbRed;

  rgb_color24(BYTE cRed = 0, BYTE cGreen = 0, BYTE cBlue = 0)
  {
    rgbRed = cRed;
    rgbGreen = cGreen;
    rgbBlue = cBlue;
  }

  rgb_color24(RGBTRIPLE rgbNew)
  {
    rgbRed = rgbNew.rgbtRed;
    rgbGreen = rgbNew.rgbtGreen;
    rgbBlue = rgbNew.rgbtBlue;
  }

  rgb_color24(DWORD dwRGB)
  {
    rgbBlue     = (BYTE)(dwRGB & 0xFF);
    rgbGreen    = (BYTE)((dwRGB >> 8) & 0xFF);
    rgbRed      = (BYTE)((dwRGB >> 16) & 0xFF);
  }

  inline DWORD to_dword() const
  {
    DWORD dwValue = (rgbRed      << 16) +
                    (rgbGreen    << 8) +
                     rgbBlue;
    return dwValue;
  }

  inline operator RGBTRIPLE() const
  {
```

```
    RGBTRIPLE rgbRetVal = { rgbBlue, rgbGreen, rgbRed, };
    return rgbRetVal;
  }
};

inline rgb_color24 operator+(rgb_color24 a, rgb_color24 b)
{
  return rgb_color24(a.rgbRed + b.rgbRed,
                     a.rgbGreen + b.rgbGreen,
                     a.rgbBlue + b.rgbBlue);
}

inline rgb_color24 operator-(rgb_color24 a, rgb_color24 b)
{
  return rgb_color24(a.rgbRed - b.rgbRed,
                     a.rgbGreen - b.rgbGreen,
                     a.rgbBlue - b.rgbBlue);
}

inline rgb_color24 operator*(rgb_color24 a, int b)
{
  return rgb_color24(a.rgbRed * b,
                     a.rgbGreen * b,
                     a.rgbBlue * b);
}

inline rgb_color24 operator/(rgb_color24 a, int b)
{
  ASSERT(b != 0);
  return rgb_color24(a.rgbRed / b,
                     a.rgbGreen / b,
                     a.rgbBlue / b);
}

inline bool operator==(const rgb_color24 a, const rgb_color24 b)
{
  return (a.to_dword() == b.to_dword());
}

inline bool operator<(const rgb_color24 a, const rgb_color24 b)
{
  return (a.to_dword() < b.to_dword());
}

inline bool operator>(const rgb_color24 a, const rgb_color24 b)
```

```
{
  return (a.to_dword() > b.to_dword());
}

#pragma pack()

#endif //RGB_COLOR_CLASS_H__
```

Tali classi permettono la manipolazione delle bitmap da parte degli algoritmi predittivi come *jpeg-ls*, specie per quanto riguarda quelle palettizzate: le bitmap a 8bppbasate su palette, infatti, sono memorizzate come una serie di indici che si riferiscono a una tavolozza di colori, la palette appunto.

L'algoritmo *jpeg-ls* trae vantaggi dalla similitudine fra pixel adiacenti per stimare il valore del pixel successivo, codificando poi la differenza. Se la differenza si mantiene bassa ovunque, abbiamo una buona compressione.

In una bitmap con palette, pixel adiacenti con colori simili possono puntare ad elementi distanti nell'array di colori, e quindi se presi cosí come sono non vanno bene. La classe **class bitmap8bpp** supplisce a questo con i metodi *get_greyed_pixel()* e *set_greyed_pixel()* che utilizzano una mappa di traduzione calcolata al caricamento dell'immagine per restituire il colore dei pixel normalizzati in scala di grigio.

8.5 Modelli

I vari modelli, descritti in prfecedenza nelle sezione *"Modellizzazione della sorgente"*, sono implementati nelle rispettive classi, una per modello.

⟨*jpeg-ls-model.h*⟩≡

```
#ifndef JPEG_LS_MODEL_H
#define JPEG_LS_MODEL_H

#include <vector>

using namespace std;

#include "common.h"
#include "model.h"
#include "arithmetic_encoder.h"
#include "arithmetic_decoder.h"
#include "index.h"
#include "adaptive_model.h"
#include "bitmap.h"
#include <algorithm>

using namespace std;

class jpeg_ls_model : public adaptive_model<BYTE>
{
private:
  BYTE m_PredictedValue;
  bitmap8bpp* m_Bmp;

  jpeg_ls_model(const jpeg_ls_model&);
  jpeg_ls_model& operator=(const jpeg_ls_model&);
```

```
public:
  jpeg_ls_model(arithmetic_encoder* encoder, bitmap8bpp* bmp);
  jpeg_ls_model(arithmetic_decoder* decoder, bitmap8bpp* bmp);

  void predict5(DWORD i, DWORD j);

  virtual void encode_symbol(BYTE symbol);
  virtual void encode_eof();
  virtual BYTE decode_symbol();
  virtual ~jpeg_ls_model()
  {
  }
};

///////////////////////////////////////////////////////////

jpeg_ls_model::jpeg_ls_model(arithmetic_encoder* encoder, bitmap8bpp* bmp) :
  adaptive_model<BYTE>(encoder), m_Bmp(bmp)
{
  m_PredictedValue = 0;
}

jpeg_ls_model::jpeg_ls_model(arithmetic_decoder* decoder, bitmap8bpp* bmp) :
  adaptive_model<BYTE>(decoder), m_Bmp(bmp)
{
  m_PredictedValue = 0;
}

void jpeg_ls_model::predict5(DWORD i, DWORD j)
{
  BYTE N =  m_Bmp->get_greyed_pixel(i - 1, j);
  BYTE W =  m_Bmp->get_greyed_pixel(i, j - 1);
  BYTE NW = m_Bmp->get_greyed_pixel(i - 1, j - 1);

  if (!(NW < _MAX(W, N)))
  {
    m_PredictedValue = _MAX(W, N);
  }
  else
  {
    if (!(NW > _MIN(W, N)))
      m_PredictedValue = _MIN(W, N);
    else
      m_PredictedValue = W + N - NW;
```

```
  }
}

void jpeg_ls_model::encode_symbol(BYTE symbol)
{
  if (m_encoder)
  {
    BYTE cbDistance = symbol - m_PredictedValue;
    DWORD iDistance = idx<BYTE>::from_symbol(cbDistance);

    m_encoder->encode(cum_count(iDistance - 1),
                      cum_count(iDistance),
                      total_count());

    update(cbDistance);
  }
}

void jpeg_ls_model::encode_eof()
{
  m_encoder->encode_eof();
}

BYTE jpeg_ls_model::decode_symbol()
{
  BYTE symbol;
  if (m_decoder)
  {
    DWORD target = m_decoder->decode_target(total_count());
    DWORD index = search_index(target);

    m_decoder->decode(cum_count(index - 1),
                      cum_count(index),
                      total_count());

    BYTE cbDistance = idx<BYTE>::to_symbol(index);
    symbol = m_PredictedValue + cbDistance;
    update(cbDistance);
  }

  return symbol;
}

/////////////////////////// The model for true color BMPs ///////////////////////////////////
```

```
class jpeg_ls_24_model : public model<rgb_color24>
{
private:
  rgb_color24 m_rgbPredictedValue;

  bitmap_true<rgb_color24>* m_Bmp;

  enum { dwSize = 257 };

  DWORD m_aRedCumCount  [dwSize];
  DWORD m_aGreenCumCount[dwSize];
  DWORD m_aBlueCumCount [dwSize];
  //DWORD m_aAlphaCumCount[dwSize];

  arithmetic_encoder* m_encoder;
  arithmetic_decoder* m_decoder;

  jpeg_ls_24_model(const jpeg_ls_24_model&);
  jpeg_ls_24_model& operator=(const jpeg_ls_24_model&);

  void  init(DWORD aColorCumCount[]);

  DWORD cum_count(size_t iSymbol, DWORD aColorCumCount[]);
  DWORD search_index(DWORD dwTarget, DWORD aColorCumCount[]);
  DWORD total_count(DWORD aColorCumCount[]);

  void update(BYTE bColorChannel, DWORD aColorCumCount[]);
  void update(rgb_color24 rgbNewColor);

  void encode_channel(DWORD iColorDistance, DWORD aColorCumCount[]);

public:
  jpeg_ls_24_model(arithmetic_encoder* encoder, bitmap_true<rgb_color24>* bmp);
  jpeg_ls_24_model(arithmetic_decoder* decoder, bitmap_true<rgb_color24>* bmp);

  void predict5(DWORD i, DWORD j);

  virtual void encode_symbol(rgb_color24 symbol);
  virtual void encode_eof();
  virtual rgb_color24 decode_symbol();
  virtual ~jpeg_ls_24_model()
  {
  }
};
```

```
////////////////////////////// ctor ////////////////////////////

jpeg_ls_24_model::jpeg_ls_24_model(arithmetic_encoder* encoder, bitmap_true<rgb_color24>* bmp
  m_Bmp(bmp),
  m_encoder(encoder),
  m_decoder(NULL)
{
  init(m_aRedCumCount);
  init(m_aGreenCumCount);
  init(m_aBlueCumCount);
  //init(m_aAlphaCumCount);
}

jpeg_ls_24_model::jpeg_ls_24_model(arithmetic_decoder* decoder, bitmap_true<rgb_color24>* bmp
  m_Bmp(bmp),
  m_encoder(NULL),
  m_decoder(decoder)
{
  init(m_aRedCumCount);
  init(m_aGreenCumCount);
  init(m_aBlueCumCount);
  //init(m_aAlphaCumCount);
}

////////////////////////// private /////////////////////////////

void jpeg_ls_24_model::init(DWORD aColorCumCount[])
{
  aColorCumCount[0] = 0;
  for (DWORD i = 1; i < dwSize; ++i)
    aColorCumCount[i] = i;
}

inline DWORD jpeg_ls_24_model::cum_count(size_t iSymbol, DWORD aColorCumCount[])
{
  return(aColorCumCount[iSymbol]);
}

inline DWORD jpeg_ls_24_model::search_index(DWORD dwTarget, DWORD aColorCumCount[])
{
  register DWORD k;
  for (k = 1; k < dwSize; ++k)
    if ((aColorCumCount[k - 1] <= dwTarget) && (dwTarget < aColorCumCount[k]))
      break;
  return k;
```

```
}

inline DWORD jpeg_ls_24_model::total_count(DWORD aColorCumCount[])
{
  ASSERT(aColorCumCount[dwSize - 1] > aColorCumCount[dwSize - 2]);
  return(aColorCumCount[dwSize - 1]);
}

void jpeg_ls_24_model::update(BYTE bColorChannel, DWORD aColorCumCount[])
{
  for (DWORD j = idx<BYTE>::from_symbol(bColorChannel); j < dwSize; ++j)
  {
    ++(aColorCumCount[j]);
  }

  if (aColorCumCount[dwSize - 1] >= ((DWORD)UPPER_LIMIT >> 2) )
  {
    for (DWORD i = 1; i < dwSize; ++i)
    {
      aColorCumCount[i] >>= 1;
      while (aColorCumCount[i] <= aColorCumCount[i - 1])
        ++aColorCumCount[i];
    }
  }
}

void jpeg_ls_24_model::update(rgb_color24 rgbNewColor)
{
  update(rgbNewColor.rgbRed,      m_aRedCumCount  );
  update(rgbNewColor.rgbGreen,    m_aGreenCumCount);
  update(rgbNewColor.rgbBlue,     m_aBlueCumCount );
  //update(rgbNewColor.rgbReserved, m_aAlphaCumCount);
}

void jpeg_ls_24_model::predict5(DWORD i, DWORD j)
{
  rgb_color24 rgbN = m_Bmp->get_pixel(i - 1, j);
  rgb_color24 rgbW = m_Bmp->get_pixel(i, j - 1);
  rgb_color24 rgbNW = m_Bmp->get_pixel(i - 1, j - 1);

  if (!(rgbNW < _MAX(rgbW, rgbN)))
  {
    m_rgbPredictedValue = _MAX(rgbW, rgbN);
  }
  else
```

```
  {
    if (!(rgbNW > _MIN(rgbW, rgbN)))
      m_rgbPredictedValue = _MIN(rgbW, rgbN);
    else
      m_rgbPredictedValue = rgbW + rgbN - rgbNW;
  }
}

void jpeg_ls_24_model::encode_channel(DWORD iColorDistance, DWORD aColorCumCount[])
{
  m_encoder->encode(cum_count(iColorDistance - 1, aColorCumCount),
                    cum_count(iColorDistance, aColorCumCount),
                    total_count(aColorCumCount));
}

/////////////////////////// publics ////////////////////////////////////

void jpeg_ls_24_model::encode_symbol(rgb_color24 rgbSymbol)
{
  if (m_encoder)
  {
    DWORD iDistance;

    rgb_color24 rgbDistance = rgbSymbol - m_rgbPredictedValue;

    iDistance = idx<BYTE>::from_symbol(rgbDistance.rgbRed);
    encode_channel(iDistance, m_aRedCumCount);

    iDistance = idx<BYTE>::from_symbol(rgbDistance.rgbGreen);
    encode_channel(iDistance, m_aGreenCumCount);

    iDistance = idx<BYTE>::from_symbol(rgbDistance.rgbBlue);
    encode_channel(iDistance, m_aBlueCumCount);

    //iDistance = idx<BYTE>::from_symbol(rgbDistance.rgbReserved);
    //encode_channel(iDistance, m_aAlphaCumCount);

    update(rgbDistance);
  }
}

void jpeg_ls_24_model::encode_eof()
{
  m_encoder->encode_eof();
}
```

```
rgb_color24 jpeg_ls_24_model::decode_symbol()
{
  rgb_color24 rgbSymbol;
  if (m_decoder)
  {
    rgb_color24 rgbDistance;

    //red
    DWORD dwRedTarget = m_decoder->decode_target(total_count(m_aRedCumCount));
    DWORD iRed = search_index(dwRedTarget, m_aRedCumCount);

    m_decoder->decode(cum_count(iRed - 1, m_aRedCumCount),
                      cum_count(iRed, m_aRedCumCount),
                      total_count(m_aRedCumCount));

    BYTE cbRedDistance = idx<BYTE>::to_symbol(iRed);
    rgbDistance.rgbRed = cbRedDistance;

    //green
    DWORD dwGreenTarget = m_decoder->decode_target(total_count(m_aGreenCumCount));
    DWORD iGreen = search_index(dwGreenTarget, m_aGreenCumCount);

    m_decoder->decode(cum_count(iGreen - 1, m_aGreenCumCount),
                      cum_count(iGreen, m_aGreenCumCount),
                      total_count(m_aGreenCumCount));

    BYTE cbGreenDistance = idx<BYTE>::to_symbol(iGreen);
    rgbDistance.rgbGreen = cbGreenDistance;

    //blue
    DWORD dwBlueTarget = m_decoder->decode_target(total_count(m_aBlueCumCount));
    DWORD iBlue = search_index(dwBlueTarget, m_aBlueCumCount);

    m_decoder->decode(cum_count(iBlue - 1, m_aBlueCumCount),
                      cum_count(iBlue, m_aBlueCumCount),
                      total_count(m_aBlueCumCount));

    BYTE cbBlueDistance = idx<BYTE>::to_symbol(iBlue);
    rgbDistance.rgbBlue = cbBlueDistance;

    //alpha
    /*DWORD dwAlphaTarget = m_decoder->decode_target(total_count(m_aAlphaCumCount));
    DWORD iAlpha = search_index(dwAlphaTarget, m_aAlphaCumCount);
```

```
    m_decoder->decode(cum_count(iAlpha - 1, m_aAlphaCumCount),
                      cum_count(iAlpha, m_aAlphaCumCount),
                      total_count(m_aAlphaCumCount));

    BYTE cbAlphaDistance = idx<BYTE>::to_symbol(iAlpha);
    rgbDistance.rgbReserved = cbAlphaDistance;*/

    rgbSymbol = m_rgbPredictedValue + rgbDistance;
    update(rgbDistance);
  }

  return rgbSymbol;
}

#endif //JPEG_LS_MODEL_H
```

⟨*adaptive-model.h*⟩≡

```
#ifndef ADAPTIVE_MODEL_H
#define ADAPTIVE_MODEL_H

#include <vector>

using namespace std;

#include "common.h"
#include "model.h"
#include "arithmetic_encoder.h"
#include "arithmetic_decoder.h"
#include "index.h"

template <typename symbolT>
class adaptive_model : public model<symbolT>
{
protected:
  DWORD dwSize;
  vector<DWORD> m_cum_count;

  arithmetic_encoder* m_encoder;
  arithmetic_decoder* m_decoder;

  adaptive_model(const adaptive_model<symbolT>&);
  adaptive_model<symbolT>& operator=(const adaptive_model<symbolT>&);

  void update(symbolT new_symbol);

  inline DWORD cum_count(size_t iSymbol)
  {
    return(m_cum_count[iSymbol]);
  }

  inline DWORD search_index(DWORD target)
  {
    register DWORD k;
    for (k = 1; k < dwSize; ++k)
      if ((m_cum_count[k - 1] <= target) && (target <m_cum_count[k]))
        break;
    return k;
  }

  inline DWORD total_count()
```

```
  {
    ASSERT(m_cum_count[dwSize - 1] > m_cum_count[dwSize - 2]);
    return(m_cum_count[dwSize - 1]);
  }

  inline void init()
  {
    m_cum_count[0] = 0;
    for (DWORD i = 1; i < dwSize; ++i)
      m_cum_count[i] = i;
  }

public:
  adaptive_model(arithmetic_encoder* encoder);
  adaptive_model(arithmetic_decoder* decoder);

  virtual void encode_symbol(symbolT symbol);
  virtual void encode_eof();
  virtual symbolT decode_symbol();
  virtual ~adaptive_model() {}
};

////////////////////////////////////////////////////////////

template<typename symbolT>
adaptive_model<symbolT>::adaptive_model(arithmetic_encoder* encoder) :

  dwSize((idx<symbolT>::max()) + 2/*4*/),
  m_cum_count(dwSize),
  m_encoder(encoder),
  m_decoder(NULL)

{
  init();
}

template<typename symbolT>
adaptive_model<symbolT>::adaptive_model(arithmetic_decoder* decoder) :

  dwSize(idx<symbolT>::max() + 2/*4*/),
  m_cum_count(dwSize),
  m_encoder(NULL),
  m_decoder(decoder)

{
```

```
  init();
}

template<typename symbolT>
void adaptive_model<symbolT>::update(symbolT new_symbol)
{
  for (DWORD j = idx<symbolT>::from_symbol(new_symbol); j < dwSize; ++j)
  {
    ++(m_cum_count[j]);

  }

  if (m_cum_count[dwSize - 1] >= ((DWORD)UPPER_LIMIT >> 2) )
  {
    for (DWORD i = 1; i < dwSize; ++i)
    {
      m_cum_count[i] >>= 1;
      while (m_cum_count[i] <= m_cum_count[i - 1])
        ++m_cum_count[i];
    }
  }
}

template<typename symbolT>
void adaptive_model<symbolT>::encode_symbol(symbolT symbol)
{
  if (m_encoder)
  {
    m_encoder->encode(cum_count(idx<symbolT>::from_symbol(symbol) - 1),
                      cum_count(idx<symbolT>::from_symbol(symbol)), //symbol_to_index(symbol)
                      total_count());
    update(symbol);
  }
}

template<typename symbolT>
void adaptive_model<symbolT>::encode_eof()
{
  m_encoder->encode_eof();
}

template<typename symbolT>
symbolT adaptive_model<symbolT>::decode_symbol()
{
  symbolT symbol;
```

```
  if (m_decoder)
  {
    DWORD target = m_decoder->decode_target(total_count());
    DWORD index = search_index(target);

    m_decoder->decode(cum_count(index - 1),
                      cum_count(index),
                      total_count());

    symbol = idx<symbolT>::to_symbol(index);
    update(symbol);
  }

  return symbol;
}

#endif
```

⟨*difference-adaptive-model.h*⟩≡

```
#ifndef DIFF_ADAPTIVE_MODEL_H
#define DIFF_ADAPTIVE_MODEL_H

#include <vector>

using namespace std;

#include "common.h"
#include "model.h"
#include "arithmetic_encoder.h"
#include "arithmetic_decoder.h"
#include "index.h"
#include "adaptive_model.h"

template <typename symbolT>
class difference_adaptive_model : public adaptive_model<symbolT>
{
private:
  symbolT m_tLastSymbol;

  difference_adaptive_model(const difference_adaptive_model<symbolT>&);
  difference_adaptive_model<symbolT>& operator=(const difference_adaptive_model<symbolT>&);

  inline void init()
  {
    adaptive_model<symbolT>::init();
    m_tLastSymbol = 0;
  }

public:
  difference_adaptive_model(arithmetic_encoder* encoder);
  difference_adaptive_model(arithmetic_decoder* decoder);

  virtual void encode_symbol(symbolT& symbol);
  virtual void encode_eof();
  virtual symbolT decode_symbol();
  virtual ~difference_adaptive_model()
  {
  }
};

/////////////////////////////////////////////////////////
```

```
template<typename symbolT>
difference_adaptive_model<symbolT>::difference_adaptive_model(arithmetic_encoder* encoder) :
  adaptive_model<symbolT>(encoder)
{
  init();
}

template<typename symbolT>
difference_adaptive_model<symbolT>::difference_adaptive_model(arithmetic_decoder* decoder) :
  adaptive_model<symbolT>(decoder)
{
  init();
}

template<typename symbolT>
void difference_adaptive_model<symbolT>::encode_symbol(symbolT& symbol)
{
  if (m_encoder)
  {
    symbolT tDistance = symbol - m_tLastSymbol;
    DWORD iDistance = idx<symbolT>::from_symbol(tDistance);

     m_encoder->encode(cum_count(iDistance - 1),
                       cum_count(iDistance), //symbol_to_index(symbol)),
                       total_count());

    m_tLastSymbol = symbol;

    update(tDistance);
  }
}

template<typename symbolT>
void difference_adaptive_model<symbolT>::encode_eof()
{
  m_encoder->encode_eof();
}

template<typename symbolT>
symbolT difference_adaptive_model<symbolT>::decode_symbol()
{
  symbolT symbol;
  if (m_decoder)
  {
    DWORD target = m_decoder->decode_target(total_count());
```

```
    DWORD index = search_index(target);

    m_decoder->decode(cum_count(index - 1),
                      cum_count(index),
                      total_count());

    symbolT tDistance = idx<symbolT>::to_symbol(index);
    symbol = m_tLastSymbol + tDistance;
    m_tLastSymbol = symbol;
    update(tDistance);
  }

  return symbol;
}

#endif
```

⟨*ppm-model.h*⟩≡

```
#ifndef PPM_MODEL_H
#define PPM_MODEL_H

#include <map>
#include <list>
#include <deque>
#include <string>

using namespace std;

#include "common.h"
#include "model.h"
#include "arithmetic_encoder.h"
#include "arithmetic_decoder.h"
#include "index.h"

const char escape = '\0';

class context_entry
{
public:
  char cLetter;
  DWORD dwCount;
  DWORD dwCumCount;
};

class context_entry_list
{
private:
  list<context_entry> m_List;
  context_entry m_ceEscape;
public:
  typedef list<context_entry>::iterator iterator;

  context_entry_list()
  {
    m_ceEscape.cLetter = escape;
    m_ceEscape.dwCount = 1;
    m_ceEscape.dwCumCount = 1;
  }

  inline DWORD total_count()
  {
```

```
  return(m_ceEscape.dwCumCount);
}

void insert(char cNewLetter)
{
  context_entry ceNew;
  ceNew.cLetter = cNewLetter;
  ceNew.dwCount = 1;
  ceNew.dwCumCount = m_ceEscape.dwCumCount - m_ceEscape.dwCount + 1;

  m_List.push_back(ceNew);

  m_ceEscape.dwCount += 1;
  m_ceEscape.dwCumCount += 2;
}

void update(iterator i)
{
  i->dwCount += 1;
  for (; i != end(); ++i)
  {
    i->dwCumCount += 1;
  }

  m_ceEscape.dwCumCount += 1;
}

iterator begin()
{
  return(m_List.begin());
}

iterator end()
{
  return(m_List.end());
}

inline DWORD escape_lower_bound()
{
  return(m_ceEscape.dwCumCount - m_ceEscape.dwCount);
}

inline DWORD escape_upper_bound()
{
  return(m_ceEscape.dwCumCount);
```

```
  }

  inline context_entry& escape_entry()
  {
    return(m_ceEscape);
  }
};

template<size_t S = 5>
class ppm_model : public model<char>
{
protected:
  typedef map<string, context_entry_list> context_order;

  context_order aContextOrders[S + 1];
  string sHistory;
  DWORD m_dwNegativeContextCumCount;

  arithmetic_encoder* m_encoder;
  arithmetic_decoder* m_decoder;

  ppm_model(const ppm_model&);
  ppm_model& operator=(const ppm_model&);

  context_entry search_symbol(DWORD dwTarget, context_entry_list& ContextEntryList);
  char rec_decode(string sContext, int iContext);

public:
  ppm_model(arithmetic_encoder* encoder): m_encoder(encoder),
                                          m_decoder(NULL)
  {
  }

  ppm_model(arithmetic_decoder* decoder): m_encoder(NULL),
                                          m_decoder(decoder)
  {
  }

  virtual void encode_symbol(char cNewSymbol);
  virtual void encode_eof();
  virtual char decode_symbol();

  virtual ~ppm_model()
  {
  }
```

```
};

////////////////////////////////////////////////////////////

template<size_t S>
context_entry ppm_model<S>::search_symbol(DWORD dwTarget, context_entry_list& ContextEntryLis
{
  context_entry ceRet;

  if ((ContextEntryList.escape_lower_bound  <= dwTarget) &&
      (dwTarget < ContextEntryList.escape_upper_bound))
  return ContextEntryList.escape_entry();

  DWORD dwPrevCumCount = 0;

  for(context_entry_list::iterator i = ContextEntryList.begin();
      i != ContextEntryList.end();
      ++i)
  {
    if ((dwPrevCumCount <= dwTarget) && (dwTarget < i->dwCumCount))
      return (*i);

    dwPrevCumCount = i->dwCumCount;
  }
  return ceRet;
}

template<size_t S>
void ppm_model<S>::encode_symbol(char cNewSymbol)
{
  if (m_encoder)
  {
    bool bContextFound = false;
    int j = 0;

    for ( int iContext = sHistory.length();
         (iContext >= 0) && !bContextFound;
         --iContext)
    {
      string sContext(sHistory, j, iContext);
      ++j;

      context_entry_list& ContextEntryList = aContextOrders[iContext][sContext];

      for(context_entry_list::iterator i = ContextEntryList.begin();
```

```
        (i != ContextEntryList.end()) && !bContextFound;
        ++i)
      {
        if(i->cLetter == cNewSymbol)
        {
          //codifica il simbolo
          m_encoder->encode(i->dwCumCount - i->dwCount,
                            i->dwCumCount,
                            ContextEntryList.total_count());

          //update (existing)
          ContextEntryList.update(i);

          bContextFound = true;
        }
      }

      if(!bContextFound)
      {
        //codifica un escape
        m_encoder->encode(ContextEntryList.escape_lower_bound(),
                          ContextEntryList.escape_upper_bound(),
                          ContextEntryList.total_count());
        //update_new
        ContextEntryList.insert(cNewSymbol);
      }
    }

    if(!bContextFound)
    {
      //-1 context order
      m_encoder->encode((BYTE)cNewSymbol - 1,
                        (BYTE)cNewSymbol,
                        0xFF);
    }

    if (sHistory.size() < S)
      sHistory += cNewSymbol;
    else
      sHistory = sHistory.substr(1) + cNewSymbol;
  }
}

template<size_t S>
void ppm_model<S>::encode_eof()
```

```
{
  m_encoder->encode_eof();
}

template<size_t S>
char ppm_model<S>::rec_decode(string sContext, int iContext)
{
  context_entry ceSymbol;
  context_entry_list::iterator i;

  if (iContext >= 0)
  {
    context_entry_list& ContextEntryList = aContextOrders[iContext][sContext];

    DWORD dwTarget = m_decoder->decode_target(ContextEntryList.total_count());

    //search symbol
    if ((ContextEntryList.escape_lower_bound()  <= dwTarget) &&
        (dwTarget < ContextEntryList.escape_upper_bound()))
    {
      ceSymbol= ContextEntryList.escape_entry();
    }
    else
    {
      DWORD dwPrevCumCount = 0;

      for(i = ContextEntryList.begin();
          i != ContextEntryList.end();
          ++i)
      {
        if ((dwPrevCumCount <= dwTarget) && (dwTarget < i->dwCumCount))
        {
          ceSymbol = *i;
          break;
        }
        dwPrevCumCount = i->dwCumCount;
      }
    }
    //fine ricerca simbolo

    m_decoder->decode(ceSymbol.dwCumCount - ceSymbol.dwCount,
                      ceSymbol.dwCumCount,
                      ContextEntryList.total_count());

    if (ceSymbol.cLetter != escape)
```

```
    {
      ContextEntryList.update(i);
      return (ceSymbol.cLetter);
    }
    else
    {
      string sNewContext;

      if (sContext.size() == 0)
        sNewContext = sContext;
      else
        sNewContext = string(sContext, 1, iContext - 1);

      char cSymbol = rec_decode(sNewContext, iContext - 1);

      ASSERT(cSymbol != escape);

      ContextEntryList.insert(cSymbol);
      return (cSymbol);
    }
  }
  else
  {
    //-1 context order

    DWORD dwTarget = m_decoder->decode_target((DWORD)0xFF);
    char cSymbol = (char)(dwTarget + 1);

    m_decoder->decode((BYTE)cSymbol - 1,
                      (BYTE)cSymbol,
                      0xFF);
    return (cSymbol);
  }
}

template<size_t S>
char ppm_model<S>::decode_symbol()
{
  char cSymbol;

  if (m_decoder)
  {
    cSymbol = rec_decode(sHistory, sHistory.length());

    if (sHistory.size() < S)
```

```
      sHistory += cSymbol;
    else
      sHistory = sHistory.substr(1) + cSymbol;
  }
  return cSymbol;
}

#endif //PPM_MODEL_H
```

Si noti l'utilizzo esteso dei template, che permette di aver codice più efficiente e di menutenzione meno problematiche. Inoltre permette di avere modelli operanti su più tipi di dati (nell'implementazione attuale, ci sono modelli basati su **char, BYTE** e **rgb_color24**).

9 Appendice A: codice di supporto

⟨*common.h*⟩≡

```
#ifndef COMMON_H__
#define COMMON_H__

//#define NDEBUG
#define X86
//#define WINDOWS

#include "types.h"
#include <assert.h>
#include <stdio.h>

#ifdef WINDOWS

#include <windows.h>
#pragma warning (disable: 4786)
#include "G:\\Debug\\include\\BugslayerUtil.h"

#else

#define _MAX max
#define _MIN min

#ifdef NDEBUG
#define ASSERT(x) { if (!(x)) perror("Last error");  assert(x); }
#define TRACE printf
#else
#define ASSERT(x)
#endif //NDEBUG

#endif //WINDOWS

#define MSB_MASK (0x80000000)
#define UPPER_LIMIT (~0x0)

const DWORD g_dwMagic = 0x68746542;

#define MSB(x)  (x >> ((sizeof(x) * 8) - 1))
```

```
#define M2SB(x) ((x >> ((sizeof(x) * 8) - 2)) & 0x1)
#define LSB(x)  (x & 0x0001)

#ifdef max
#undef max
#endif

#ifdef min
#undef min
#endif

#endif
```

⟨*types.h*⟩≡

```
#ifndef TYPES_H___
#define TYPES_H___

#ifdef ALPHA
typedef unsigned long QWORD;
typedef unsigned int DWORD;
#endif

#if defined X86 && defined WINDOWS
typedef __int64 QWORD;
#endif

#if defined X86 && !defined WINDOWS
typedef unsigned long long int QWORD;
typedef unsigned int DWORD;
typedef long LONG;
#endif

typedef unsigned short WORD;

typedef bool bit;
typedef unsigned char BYTE;

#endif
```

$\langle bmpDEF.h \rangle \equiv$

```
#ifndef BMP_DEF_H
#define BMP_DEF_H

#include "common.h"

#define BI_RGB        0L
#define BI_RLE8       1L
#define BI_RLE4       2L
#define BI_BITFIELDS  3L
#define BI_JPEG       4L
#define BI_PNG        5L

#pragma pack(1)
typedef struct tagRGBQUAD {
        BYTE    rgbBlue;
        BYTE    rgbGreen;
        BYTE    rgbRed;
        BYTE    rgbReserved;
} RGBQUAD, *PRGBQUAD;


typedef struct tagRGBTRIPLE { // rgbt
    BYTE rgbtBlue;
    BYTE rgbtGreen;
    BYTE rgbtRed;
} RGBTRIPLE;

typedef struct tagBITMAPINFOHEADER{
        DWORD      biSize;
        LONG       biWidth;
        LONG       biHeight;
        WORD       biPlanes;
        WORD       biBitCount;
        DWORD      biCompression;
        DWORD      biSizeImage;
        LONG       biXPelsPerMeter;
        LONG       biYPelsPerMeter;
        DWORD      biClrUsed;
        DWORD      biClrImportant;
} BITMAPINFOHEADER, *PBITMAPINFOHEADER;

typedef struct tagBITMAPINFO {
    BITMAPINFOHEADER    bmiHeader;
```

```
    RGBQUAD             bmiColors[1];
} BITMAPINFO, *PBITMAPINFO;

typedef struct tagBITMAPFILEHEADER {
        WORD    bfType;
        DWORD   bfSize;
        WORD    bfReserved1;
        WORD    bfReserved2;
        DWORD   bfOffBits;
} BITMAPFILEHEADER, *PBITMAPFILEHEADER;

#pragma pack()

#endif //BMP_DEF_H
```

⟨*index.h*⟩≡

```
#ifndef INDEX_H__
#define INDEX_H__

template<class T>
class idx
{
public:
  static inline DWORD max()
  {
    return ((1 << (sizeof(T) * 8)) - 1);
  }

  static inline size_t from_symbol(T symbol)
  {
    return ((size_t)symbol + 1);
    //return ((symbol) - '0');
  }

  static inline T to_symbol(DWORD index)
  {
    //return ((char)index + '0');
    return ((T)index - 1 );
  }
};

template<>
class idx<bit>
{
public:
  static inline DWORD max()
  {
    return (2);
  }

  static inline size_t from_symbol(bit symbol)
  {
    return (symbol ? 2 : 1);
  }

  static inline bit to_symbol(DWORD index)
  {
    return ((index == 0) ? 0 : 1);
  }
```

```
};

#endif //INDEX_H__
```

⟨*encode.cpp*⟩≡

```
#include "common.h"
#include "arithmetic_encoder.h"

#include "difference_adaptive_model.h"
#include "jpeg_ls_model.h"
#include "bitmap.h"
#include "adaptive_model.h"
#include "ppm_model.h"

#ifdef WINDOWS
#include "file_map.h"
#include "file_writer.h"
#else
#include "u_file_map.h"
#include "u_file_writer.h"
#endif

#include <iostream>

using namespace std;

void print_usage()
{
  cout << "\nUsage: encode -enc=x input output\n\n"
       << "       -enc=x x: modello da utilizzare\n"
       << "              1: modello ppm\n"
       << "              2: modello jpeg-ls per bitmap a 24bpp\n"
       << "              3: modello jpeg-ls per bitmap a 8bpp\n"
       << "              4: modello adattivo per BYTE (generale)\n"
       << "              5: modello differenziale per BYTE\n"
       << "\nCopyright (C) 2001 Lorenzo Dematte'. Tutti i diritti riservati\n";

}

void encode_ppm(char* szInputName, char* szOutputName)
{
  file_map<char> map_input(szInputName);
  file_writer<bit> file_comp_out;
  file_comp_out.create(szOutputName);

  arithmetic_encoder enc(&file_comp_out);
  ppm_model<5> mdl(&enc);
```

```
  DWORD adwHeader[2];

  adwHeader[0] = g_dwMagic;
  adwHeader[1] = map_input.get_size();

  file_comp_out.write_header((BYTE*)adwHeader, sizeof(adwHeader));

  for(;;)
  {
    char c = map_input.read_next();
    if (map_input.eof())
    {
      mdl.encode_eof();
      break;
    }
    mdl.encode_symbol(c);
  }
}

void encode_ls24(char* szInputName, char* szOutputName)
{
  bitmap_true<rgb_color24> bmpFileInput(szInputName);

  file_writer<bit> file_comp_out;
  file_comp_out.create(szOutputName);

  file_comp_out.write_header((BYTE*)(bmpFileInput.get_header()),
                                   bmpFileInput.get_header()->get_header_size());

  arithmetic_encoder enc(&file_comp_out);
  jpeg_ls_24_model mdl(&enc, &bmpFileInput);

  for(DWORD i = 0; i < bmpFileInput.height(); ++i)
    for(DWORD j = 0; j < bmpFileInput.width(); ++j)
    {
      rgb_color24 rgbPixel = bmpFileInput.get_pixel(i, j);

      mdl.predict5(i, j);
      mdl.encode_symbol(rgbPixel);
    }

  mdl.encode_eof();
}

void encode_ls8(char* szInputName, char* szOutputName)
```

```
{
  bitmap8bpp bmpFileInput(szInputName);

  file_writer<bit> file_comp_out;
  file_comp_out.create(szOutputName);

  file_comp_out.write_header((BYTE*)(bmpFileInput.get_header()),
                                   bmpFileInput.get_header()->get_header_size());

  arithmetic_encoder enc(&file_comp_out);
  jpeg_ls_model mdl(&enc, &bmpFileInput);

  for(DWORD i = 0; i < bmpFileInput.height(); ++i)
    for(DWORD j = 0; j < bmpFileInput.width(); ++j)
    {
      BYTE bPixel = bmpFileInput.get_greyed_pixel(i, j);

      mdl.predict5(i, j);
      mdl.encode_symbol(bPixel);
    }

  mdl.encode_eof();
}

void encode_BYTE(char* szInputName, char* szOutputName)
{
  file_map<BYTE> map_input(szInputName);
  file_writer<bit> file_comp_out;
  file_comp_out.create(szOutputName);

  arithmetic_encoder enc(&file_comp_out);
  adaptive_model<BYTE> mdl(&enc);

  DWORD adwHeader[2];

  adwHeader[0] = g_dwMagic;
  adwHeader[1] = map_input.get_size();

  file_comp_out.write_header((BYTE*)adwHeader, sizeof(adwHeader));

  for(;;)
  {
    BYTE b = map_input.read_next();
    if (map_input.eof())
    {
```

```
      mdl.encode_eof();
      break;
    }
    mdl.encode_symbol(b);
  }
}

void encode_diff_BYTE(char* szInputName, char* szOutputName)
{
  file_map<BYTE> map_input("lenna.bmp");
  file_writer<bit> file_comp_out;
  file_comp_out.create("lenna1.comp");

  arithmetic_encoder enc(&file_comp_out);
  difference_adaptive_model<BYTE> mdl(&enc);

  DWORD adwHeader[2];

  adwHeader[0] = g_dwMagic;
  adwHeader[1] = map_input.get_size();

  file_comp_out.write_header((BYTE*)adwHeader, sizeof(adwHeader));

  for(;;)
  {
    BYTE b = map_input.read_next();
    if (map_input.eof())
    {
      mdl.encode_eof();
      break;
    }
    mdl.encode_symbol(b);
  }
}

int main (int argc, char* argv[])
{
  char* szInputName = NULL;
  char* szOutputName = NULL;
  DWORD dwEncoderType = 0;

  //parse the command line

  if (argc != 4)
  {
```

```
  print_usage();
  return -1;
}

if(strncmp(argv[1], "-enc=", 5) == 0)
{
  dwEncoderType = atoi(&(argv[1][5]));
}
else
{
  print_usage();
  return -1;
}

szInputName = argv[2];
szOutputName = argv[3];

if ((!szInputName) || (!szOutputName))
{
  print_usage();
  return -1;
}

try
{
  switch(dwEncoderType)
  {
  case 1:
    encode_ppm(szInputName, szOutputName);
    break;

  case 2:
    encode_ls24(szInputName, szOutputName);
    break;

  case 3:
    encode_ls8(szInputName, szOutputName);
    break;

  case 4:
    encode_BYTE(szInputName, szOutputName);
    break;

  case 5:
    encode_diff_BYTE(szInputName, szOutputName);
```

```
      break;

    default:
      cout << "Modello " << dwEncoderType << " non riconosciuto!" << endl;
      break;
    }

  }
  catch(bad_file*)
  {
    cout << "\nERRORE:\n"
         << "Impossibile aprire o creare i files necessari.\n"
         << "Controllare che i files esistano, non siano bloccati "
         << "da altri processi e che si abbiano i permessi necessari.\n\n";
  }
  catch(wrong_bpp*)
  {
    cout << "\nERRORE:\n"
         << "Il file bitmap specificato ha una profondita' di colore differente "
         << "da quella indicata: provare a specificare un formato diverso\n\n";
  }
  catch(...)
  {
    cout << "\nERRORE:\n"
         << "Allocazione memoria fallita!\n\n";
  }

  return 0;
}
```

⟨*decode.cpp*⟩≡

```
#include "common.h"
#include "arithmetic_decoder.h"

#include "difference_adaptive_model.h"
#include "jpeg_ls_model.h"
#include "bitmap.h"
#include "adaptive_model.h"
#include "ppm_model.h"

#ifdef WINDOWS
#include "file_map.h"
#include "file_writer.h"
#else
#include "u_file_map.h"
#include "u_file_writer.h"
#endif //WINDOWS

#include <iostream>

using namespace std;

void print_usage()
{
  cout << "\nUsage: decode -enc=x input output\n\n"
       << "       -enc=x x: modello da utilizzare\n"
       << "              1: modello ppm\n"
       << "              2: modello jpeg-ls per bitmap a 24bpp\n"
       << "              3: modello jpeg-ls per bitmap a 8bpp\n"
       << "              4: modello adattivo per BYTE (generale)\n"
       << "              5: modello differenziale per BYTE\n"
       << "\nCopyright (C) 2001 Lorenzo Dematte'. Tutti i diritti riservati\n";

}

void decode_ls24(char* szInputName, char* szOutputName)
{
  DWORD dwFileSize, dwHeaderSize;

  {
    bitmap_header* bhCompInput;

    file_map<bit> mapCompressedInput(szInputName, sizeof(bitmap_header));
    bhCompInput = (bitmap_header*)(mapCompressedInput.get_header());
```

```
    dwHeaderSize = bhCompInput->get_header_size();
    dwFileSize = bhCompInput->get_file_size();

    file_writer<bit> bmpWriter;
    bmpWriter.create(szOutputName);
    bmpWriter.write_header((BYTE*)bhCompInput, bhCompInput->get_header_size());
  }

  file_map<bit> mapCompressedInput(szInputName, dwHeaderSize);
  bitmap_true<rgb_color24> bmpFileOut(szOutputName, dwFileSize);

  arithmetic_decoder decoder(&mapCompressedInput);
  jpeg_ls_24_model model(&decoder, &bmpFileOut);

  for (DWORD i = 0; i < bmpFileOut.height(); ++i)
    for (DWORD j = 0; j < bmpFileOut.width(); ++j)
    {
      model.predict5(i, j);
      rgb_color24 rgb = model.decode_symbol();
      bmpFileOut.set_pixel(i, j, rgb);
    }
}

void decode_ppm(char* szInputName, char* szOutputName)
{
  file_map<bit> map_comp_input(szInputName);
  file_writer<char> file_out;
  file_out.create(szOutputName);

  arithmetic_decoder dec(&map_comp_input);
  ppm_model<5> mdl(&dec);

  DWORD* dwBuffer = (DWORD*)map_comp_input.get_header();

  DWORD dwExpSize = dwBuffer[1];

  if (dwBuffer[0] == g_dwMagic)
  {
    for(DWORD i = 0; i < (dwExpSize / sizeof(BYTE)); ++i)
    {
      char c = mdl.decode_symbol();
      file_out.send(c);
    }
  }
```

```
}


void decode_ls8(char* szInputName, char* szOutputName)
{
  DWORD dwFileSize, dwHeaderSize;

  {
    bitmap_header* bhCompInput;

    file_map<bit> mapCompressedInput(szInputName, sizeof(bitmap_header));
    bhCompInput = (bitmap_header*)(mapCompressedInput.get_header());

    dwHeaderSize = bhCompInput->get_header_size();
    dwFileSize = bhCompInput->get_file_size();

    file_writer<bit> bmpWriter;
    bmpWriter.create(szOutputName);
    bmpWriter.write_header((BYTE*)bhCompInput, bhCompInput->get_header_size());
  }

  file_map<bit> mapCompressedInput(szInputName, dwHeaderSize);
  bitmap8bpp bmpFileOut(szOutputName, dwFileSize);

  arithmetic_decoder decoder(&mapCompressedInput);
  jpeg_ls_model model(&decoder, &bmpFileOut);

  for (DWORD i = 0; i < bmpFileOut.height(); ++i)
    for (DWORD j = 0; j < bmpFileOut.width(); ++j)
    {
      model.predict5(i, j);
      BYTE b = model.decode_symbol();
      bmpFileOut.set_greyed_pixel(i, j, b);
    }
}

void decode_BYTE(char* szInputName, char* szOutputName)
{
  file_map<bit> map_comp_input(szInputName);
  file_writer<BYTE> file_out;
  file_out.create(szOutputName);

  arithmetic_decoder dec(&map_comp_input);
  adaptive_model<BYTE> mdl(&dec);
```

```
  DWORD* dwBuffer = (DWORD*)map_comp_input.get_header();

  DWORD dwExpSize = dwBuffer[1];

  if (dwBuffer[0] == g_dwMagic)
  {
    for(DWORD i = 0; i < (dwExpSize / sizeof(BYTE)); ++i)
    {
      BYTE b = mdl.decode_symbol();
      file_out.send(b);
    }
  }
}

void decode_diff_BYTE(char* szInputName, char* szOutputName)
{
  file_map<bit> map_comp_input(szInputName);
  file_writer<BYTE> file_out;
  file_out.create(szOutputName);

  arithmetic_decoder dec(&map_comp_input);
  difference_adaptive_model<BYTE> mdl(&dec);

  DWORD* dwBuffer = (DWORD*)map_comp_input.get_header();

  DWORD dwExpSize = dwBuffer[1];

  if (dwBuffer[0] == g_dwMagic)
  {
    for(DWORD i = 0; i < (dwExpSize / sizeof(BYTE)); ++i)
    {
      BYTE b = mdl.decode_symbol();
      file_out.send(b);
    }
  }
}

int main (int argc, char* argv[])
{
  char* szInputName = NULL;
  char* szOutputName = NULL;
  DWORD dwEncoderType = 0;

  //parse the command line
```

```
if (argc != 4)
{
  print_usage();
  return -1;
}

if(strncmp(argv[1], "-enc=", 5) == 0)
{
  dwEncoderType = atoi(&(argv[1][5]));
}
else
{
  print_usage();
  return -1;
}

szInputName = argv[2];
szOutputName = argv[3];

if ((!szInputName) || (!szOutputName))
{
  print_usage();
  return -1;
}

try
{
  switch(dwEncoderType)
  {
  case 1:
    decode_ppm(szInputName, szOutputName);
    break;

  case 2:
    decode_ls24(szInputName, szOutputName);
    break;

  case 3:
    decode_ls8(szInputName, szOutputName);
    break;

  case 4:
    decode_BYTE(szInputName, szOutputName);
    break;
```

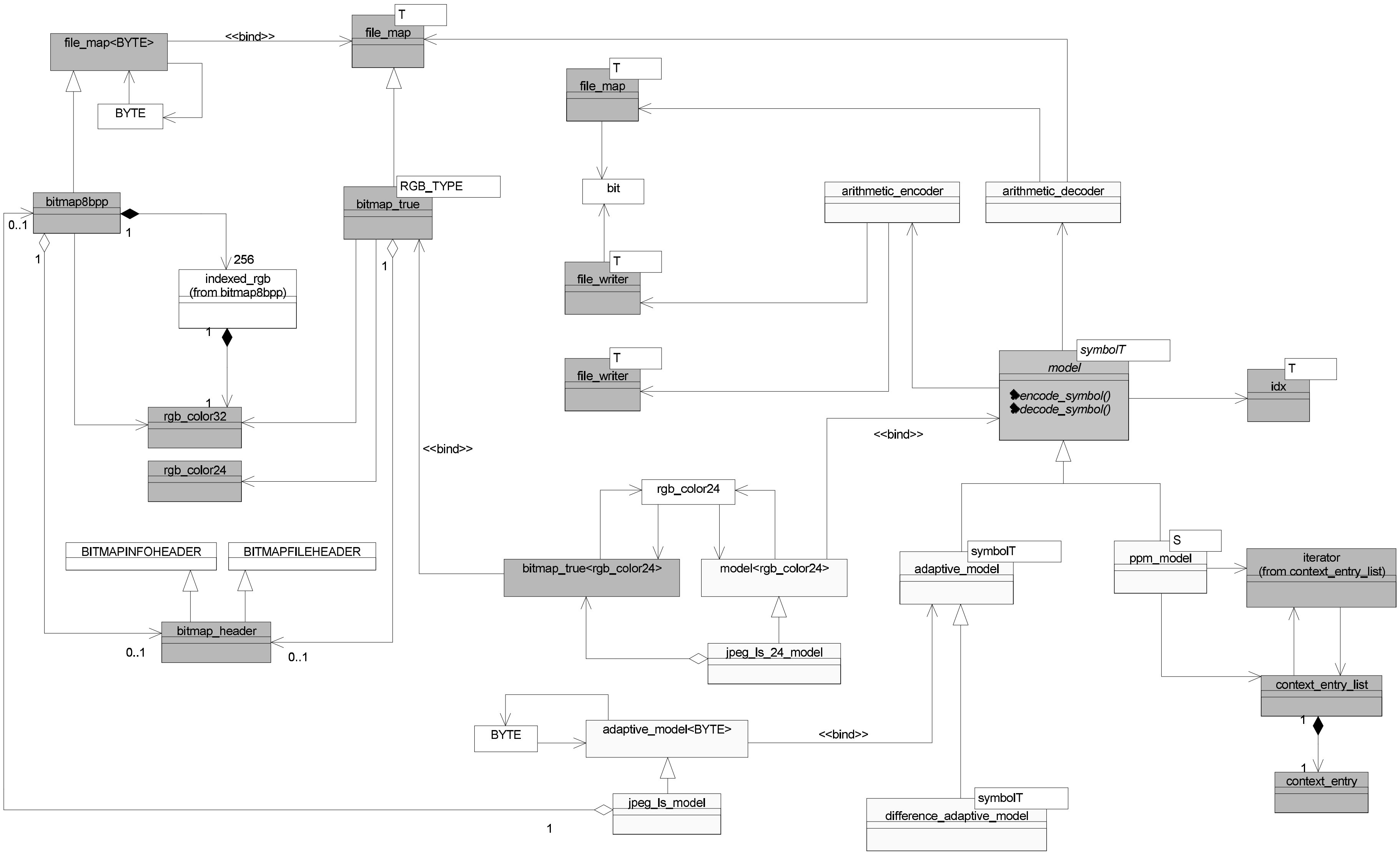
```
    case 5:
      decode_diff_BYTE(szInputName, szOutputName);
      break;
    }
  }
  catch(bad_file*)
  {
    cout << "\nERRORE:\n"
         << "Impossibile aprire o creare i files necessari.\n"
         << "Controllare che i files esistano, non siano bloccati "
         << "da altri processi e che si abbiano i permessi necessari.\n\n";
  }
  catch(wrong_bpp*)
  {
    cout << "\nERRORE:\n"
         << "Il file bitmap specificato ha una profondita' di colore differente "
         << "da quella indicata: provare a specificare un formato diverso\n\n";
  }
  catch(...)
  {
    cout << "\nERRORE:\n"
         << "Allocazione memoria fallita!\n\n";
  }

  return 0;
}
```

10 Appendice B: diagramma delle classi

Accludo anche il diagramma delle classi UML, che può aiutare alla comprensione del progetto.

11 Appendice C: bibliografia

1. Introduction to Data Compression, Second Edition – Khalid Sayood;
2. Newsgroup *comp.compression.faq*;
3. Computer Programming 34 “Compressione dati”



File: C:\WINNT\Profiles\studente\Desktop\ac2-2.mdl   8.37.02 martedì 24 luglio 2001   Class Diagram: ac2 / Reverse Engineered Page