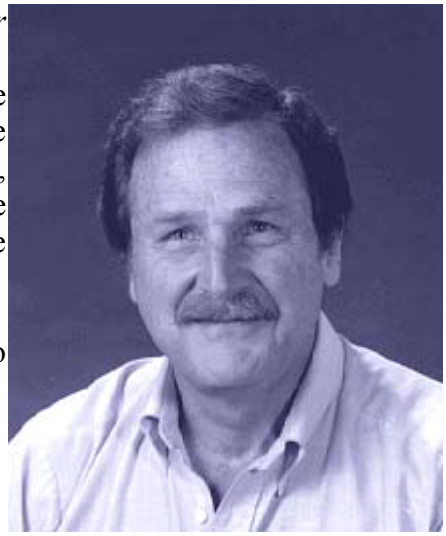
La compressione di Huffman



Oggi ci occupiamo di una delle pagine più belle della *Computer Science*, oggi ci occupiamo di compressione dei dati.
Ognuno di noi avrà usato almeno una volta programmi come WinZIP, o simili. Questi hanno la capacità di prelevare delle informazioni (sotto forma di file, contenenti immagini, suoni, testi, programmi o altro ancora) e ottenere un file di dimensioni ridotte contenente però tutto quello che serve per ripristinare le informazioni originali.

I metodi mediante i quali si realizzano programmi simili sono diversi, e sono stati affinati negli anni.

Ma il pioniere di questa fantastica avventura è sicuramente lui:

DAVID A. HUFFMAN (1925 – 1999)
Professor Emeritus of Computer Science

Eminent UCSC computer scientist David Huffman, muore il 11/10/1999, ma dentro ogni fax, modem e altre macchine che devono trattare grosse quantità di dati continua a vivere, oltre che nel cuore di tutti gli appassionati d'informatica e di milioni di studenti.

Codifica, decodifica e compressione, perché?

Alla fine degli anni 40, agli albori della Teoria dell'Informazione, nuove ed efficienti tecniche di codifica cominciavano ad essere scoperte ed i concetti di entropia e ridondanza venivano introdotti. L'entropia è una misura dell'incertezza con la quale un testo viene prodotto da una sorgente (per esempio un fax, un modem ecc.). Ciò che in quegli anni ci si proponeva di realizzare, vista l'assenza di elaboratori elettronici, era definire algoritmi in gradi di codificare testi in modo che questi potessero essere inviati da una sorgente ad una destinazione senza che nessuno, tranne il destinatario, potesse carpirne il contenuto o che potessero permettere a quest'ultimo di capire se un testo codificato fosse stato trasmesso con errori dovuti alla trasmissione o ad un sabotaggio da parte di terzi. Nasce in tal modo il concetto di codice e di codifica.
Con l'avvento, poi, dei moderni calcolatori elettronici si è sviluppato parallelamente alla teoria dei codici il concetto di compressione dati.
La compressione dei dati è utile per inviare grosse quantità di dati riducendo i costi dell'invio (pensiamo ai primi modem analogici a bassa velocità) e anche in numerose situazioni dove i supporti sui quali memorizzare le informazioni sono di limitate capacità o costosi. Nei grandi archivi dei server, se i dati non venissero compressi, i costi già alti per la loro memorizzazione sicuramente raddoppierebbero, almeno.

Diversi tipi di codifica

Quando si ha a che fare con la compressione delle informazioni c'è subito da fare una precisazione: non esiste un metodo assolutamente migliore di compressione.
La bontà del risultato è infatti fortemente legata al tipo d'informazioni che deve essere trattata.
Ad esempio l'algoritmo di un programma che si propone di fare un backup dei dati e di memorizzarlo compresso su di un nastro è differente dall'algoritmo che comprime immagini in formato JPEG.
Il primo tipo di compressione si chiama *lossless*, il secondo *lossy*.
La compressione *lossless* punta a riottenere dopo la decodifica una **copia esatta** a quella dei dati originali, mentre l'altra tollera alcuni cambiamenti (ne è un esempio una musica in formato mp3, che si può salvare con differenti gradi di compressione, che corrispondono a differenti qualità di riproduzione, maggiore è la compressione meno fedeltà rispetto all'originale forma d'onda si ottiene). Chiaramente quest'ultimo metodo di compressione è specifico per particolari tipi di dati e quando applicabile è quello che fornisce i risultati in termini di prestazioni maggiori (sempre che le imperfezioni siano tollerabili), sfortunatamente la maggior parte delle informazioni richiedono copie esatte alle originali.
Nella categoria di compressione *lossless* (quella di cui d'ora in poi tratteremo) va detto che ci sono due strategie fondamentali per ottenere il risultato e quindi gli algoritmi.
Il primo approccio è quello della **compressione statistica**, l'altro invece è rappresentato dalla **compressione mediante sostituzione di testo**, utilizzando un dizionario.
Gli algoritmi di compressione statistica si chiamano in questo modo perché al fine di produrre un risultato in linea con gli obiettivi, analizzano l'input per trovare la codifica migliore.
Si fanno studi statistici sul formato dell'input per ottenere poi una buona compressione. Supponiamo di avere a che fare con un testo su di un file in formato ASCII: ogni carattere viene rappresentato da una stringa di 8 bit, quindi se riuscissimo a codificare un carattere utilizzando meno di otto bit, sarebbe possibile diminuire lo spazio occupato dal testo originale. Utilizzando i concetti di entropia e ridondanza è stato possibile sviluppare algoritmi capaci di risalire alla codifica di un set di caratteri, conoscendone il numero di occorrenze di ciascun carattere in un dato testo in modo tale da ridurre la taglia complessiva del testo stesso.
Si studia la frequenza relativa di ciascun carattere per associare poi al carattere presente più volte nel testo, il codice più corto, viceversa a caratteri presenti con frequenza bassa la parola codice più lunga.

La compressione di Huffman è la più nota espressione di questa famiglia di algoritmi.

Per quanto riguarda, invece, la compressione mediante sostituzione del testo, questa è basata sull'idea di rimpiazzare, in un file, occorrenze di stringhe ripetute con puntatori a precedenti copie. La compressione è dovuta al fatto che la lunghezza di un puntatore è in genere più piccola della lunghezza della stringa che è rimpiazzata. I capostipiti di questa famiglia sono gli algoritmi Lempel-Ziv, che sono alla base di molti programmi di compressione attualmente in commercio.
Questo articolo non si occupa di questa famiglia di algoritmi sebbene sia molto interessante.
A quasi trent'anni di distanza dai primi algoritmi di compressione, il primo algoritmo LZ (1977), comunque ancora acerbo rispetto alle modifiche che l'hanno reso efficiente come oggi lo abbiamo conosciuto dentro programmi di compressione di successo (LZW, Lemper-Ziv-Welch 1984 – LZSS e LZH, Bell 1986 – 1987).
Oggi tutti i vari zip/gzib ad esempio hanno alla base un algoritmo LZ ricompresso poi con tecniche "statistiche" per migliorarne le prestazioni (sono chiamate compressioni "aritmetiche", e sono attualmente le più efficienti e recenti compressioni statistiche).
Ma quando si parla di Huffman e della sua codifica si tocca un punto fondamentale nella storia della Computer Science e ancora vivo (proprio perché oltre ad essere un argomento realmente affascinante funziona ancora benissimo in tante applicazioni), poi a me, quest'argomento sta particolarmente a cuore…

La ricerca del la codifica migliore, una battaglia tra cervelli

Siamo alla fine degli anni ‘40 e già Shannon e Fano avevano intuito che codificare le informazioni su “parole” di lunghezza fissa (come fanno i computer oggi) è estremamente funzionale ai fini della gestione dell’informazione ma genera ridondanza ed è quindi sconveniente rispetto agli intenti della ricerca (limitare gli errori, ridurre la ridondanza) e proposero un metodo per generare una codifica.
La codifica di Shannon-Fano costruisce una lista L di una struttura che chiamiamo “coppia”, questa è definita come { codice_nella_vecchia_codifica, occorrenza }.
Il primo elemento della coppia è il vecchio codice ad esempio si supponga d’avere una stringa contenente 15 lettere “a”, 7 lettere “b”, 6 lettere “c”, 6 lettere “d” e 5 lettere “e”.
Quindi si ha **L = { { “a”,15 } , { “b”,7 }, { “c”,6 }, { “d”,6 }, { “e”,5 } };**
Questa lista serve per capire quali sono i codici che hanno maggiore probabilità d’essere “estratti” dal testo, in modo che ad essi possa essere associato un nuovo codice di lunghezza minore rispetto al codice assegnato alle occorrenze più rare… (quando si parla di lunghezza minore di un codice si pensi sempre alla codifica che una macchina fa di un carattere, che è sempre una configurazione binaria, nei moderni computer, ad esempio il carattere ‘a’ è codificato su 8 bit nel formato ASCII e addirittura in 16 nel formato UNICODE).
Per poter valutare però la bontà dell’algoritmo Shannon-Fano (sul quale torneremo più avanti) dobbiamo purtroppo introdurre un po’ di teoria dell’informazione.

Dentro l’informazione

A questo punto, senza voler essere rigorosi, senza utilizzare formalismi complessi e dimostrazioni non immediate, diciamo come verificare la bontà di una codifica.
Sia una sorgente S tale da avere una lista di coppie associata
L = { { codice1, q1 }, { codice2,q2 }, … , { codice n, qn } }, si ha

$|S| = \sum_{i=1}^{n} q_i$ che in pratica è la lunghezza del nostro testo in chiaro.

Ognuno degli n simboli presente nel messaggio, ha la probabilità

$p(s) = \frac{1}{q_s}, \forall s \in \mathrm{I} : 1 \le s \le n$ di essere letto dall’applicazione che lo decodifica.

In parole molto povere, si è introdotta una grandezza, chiamata entropia che si indica con I(S) ed è una misura che ci permette di rilevare per un messaggio a quanto può arrivare la bontà dell’informazione stessa in termini di “efficienza”, ovvero ci da l’indicazione di quanto il messaggio potrà essere privo da ridondanza.
Il suo valore si quantifica nel modo seguente

1) $$I(S) = \sum_{s=1}^{n} p(s) \log_2 \frac{1}{p(s)}$$

Il risultato è in unità binaria d’informazione quindi è un bit, come dimostra il logaritmo in base 2 usato nella formula.

Nel caso peggiore, con n = |S| si hanno simboli equiprobabili $\forall s : p(s) = \frac{1}{n}$ e quindi

2) I(S)=log2 n.

Il suo valore è un valore limite! Immaginando un'ipotetica unità d'informazione (come è ad esempio il bit per la microinformatica) l'entropia rivela quante unità d'informazione occorrono mediamente per rappresentare, nel modo migliore possibile (secondo i criteri dell'essenzialità dell'informazione) il messaggio che vogliamo rappresentare.
Se questo è il migliore possibile è chiaro che qualsiasi altro ha un indice maggiore o al limite uguale all'entropia.
Facciamo un esempio pratico: il calcolatore conosce come unità elementari i bit ma per ovvi motivi prestazionali e di semplicità memorizza le informazioni elementari che compongono i testi in formati che sono multipli del byte (che è come tutti sanno un ottetto di bit), ad esempio nella codifica ASCII si usa un byte per rappresentare ciascuno dei 256 caratteri elementari, ognuno di questi byte si combina ad altri per costruire una stringa. Ad esempio la lettera 'A' corrisponde al byte '01000001'.
Siccome il computer non può stabilire la frequenza con la quale incontrerà ognuna delle 256 combinazioni (perché fortunatamente non sa cosa c'è nella mente folle di un programmatore), in pratica per lui i simboli sono equiprobabili e quindi
3) Corollario: **dalla formula 2 si ha che per il codice ASCII il valore d'entropia I(S)=8.**
La rappresentazione di 256 differenti stati equiprobabili si fa su 8 (otto) bit e quindi un byte!
Se però conoscessimo a priori il messaggio da ottimizzare dal punto di vista "entropico", potremmo verificarne l'entropia, confrontarla con il messaggio "in chiaro" e verificare la bontà di una codifica in termini di compressione.
Nel caso appunto del messaggio cui corrisponde la lista L = { { "a",15 } , { "b",7 }, { "c",6 }, { "d",6 }, { "e",5 } }, possiamo verificare subito che il computer impiega ben **|L|=39 byte** (la somma di tutte le frequenza 15+7+6+6+5), che equivalgono a 312 bit (39 [byte]* 8 [bit/byte]).
Applicando la formula, il valore dell'entropia I(S) per questo messaggio è I(S)=2,1858.
Infatti:

Simbolo		p	1/log p	p*(1/log2 p)
A	15	0,384615	1,378512	0,530196778
B	7	0,179487	2,478047	0,44477772
C	6	0,153846	2,70044	0,415452264
D	6	0,153846	2,70044	0,415452264
E	5	0,128205	2,963474	0,37993258
|L| =	**39**		**I(S)=**	**2,185811607**

Mentre il livello d'entropia del messaggio in codice ASCII è 8 ! Ben al di sopra di I(S) e infatti il testo in formato ASCII non è assolutamente compresso, ma è funzionale e ad alto livello (vicino all'uomo, è editabile e leggibile direttamente).
La validità di ogni algoritmo che possiamo creare per comprimere una sorgente d'informazione sarà migliore quanto s'avvicina al valore limite I(S).

Non si possono creare codici di lunghezza inferiore all'entropia (Teorema di Shannon).

Ora possiamo tornare sull'algoritmo Shannon-Fano.

L’algoritmo Shannon-Fano.

Questo è stato uno dei primi algoritmi per ridurre la ridondanza dell’informazione.
L è una lista di coppie simbolo-occorrenza ordinata in modo crescente rispetto a quest’ultimo.

ShannonFanoCode(L)

1. **if** |L| == 1 **then**
2. **return** 0
3. **if** |L| >= 2 **then**
4. "dividere L in due sottoliste L_1 ed L_2 , considerando le coppie nell'ordine in cui si presentano in L_1 , in modo tale che la differenza tra $\sum_{(x,n)} \varepsilon L_1$ n e $\sum_{(x,n)} \varepsilon L_2$n sia minima"
5. "aggiungere uno 0 alla parola di codice w_x per tutti i caratteri x : (x,n) εL_1"
6. "aggiungere uno 1 alla parola di codice w_x per tutti i caratteri x : (x,n) εL_2"
7. **if** $|L_1|$ == 1 **then**
8. **return** w_x
9. **else**
10. ShannonFanoCode(L_1)
11. **if** $|L_2|$ == 1 **then**
12. **return** w_x
13. **else**
14. ShannonFanoCode(L_2)

L’algoritmo di C. Shannon e R. M. Fano non porta ad una codifica ottima, applicandolo a testi si noterebbe coma la lunghezza media delle parole ottenute è superiore ad altri algoritmi introdotti successivamente. In pratica crea liste e le divide in base agli scarti delle frequenza tra i simboli per ciascuna lista e a seconda che il carattere sia nell’una o nell’altra lista aggiunge al codice uno zero oppure un uno, ed è un bel passo avanti rispetto a prima, ma rispetto all’algoritmo di Huffman non considera il peso associato alla somma delle frequenze di più simboli e quindi perde in efficienza.
Si vedrà che nel prossimo esempio, Shannon-Fano codificano il carattere ‘a’ con due bit, ‘b’ e ‘c’ anch’essi su due bit, purtroppo non si accorgono del peso della somma dei due rispetto alla frequenza del carattere ‘a’ e che quindi è più conveniente ridurre la dimensione del codice di ‘a’ e sacrificare gli altri con un bit in più!!
Il nostro scopo non è fare questa dimostrazione diciamo però che non appena Huffman presentò il suo algoritmo, quello presentato qua sopra venne definitivamente abbandonato.
Con L = { { “a”,15 } , { “b”,7 }, { “c”,6 }, { “d”,6 }, { “e”,5 } } si ottiene la seguente codifica

Carattere	Parola codice
A	00
B	01
C	10
D	110
E	111

Con riferimento all’esempio corrente si ha che la lunghezza media delle parole di codice prodotte è pari a **M = (15 * 2 + 7 * 2 + 6 * 2 + 6 * 3 + 5 * 3)/39 = 2.28**, la lunghezza del codice è 89 bit.

L’albero di Huffman.

Abbiamo visto nell’esempio precedente come la lunghezza media delle parole sia 2.28, un’algoritmo che minimizza questo valore (e quindi comprime di più) è l’algoritmo di Huffman.
Alla base dell’algoritmo di compressione c’è la costruzione di un albero che server per ricavare il codice stesso. Ogni nodo ha un peso, che dipende dal numero delle occorrenze di ciascun carattere, ad ogni nodo sono collegati altri nodi oppure foglie, partendo dalla radice dell’albero le prime foglie che si raggiungono appartengono ai simboli più frequenti nel testo.
Con l’algoritmo di Huffman il testo dell’esempio precedente, potrebbe essere codificato nel modo seguente (dipende dall’implementazione):

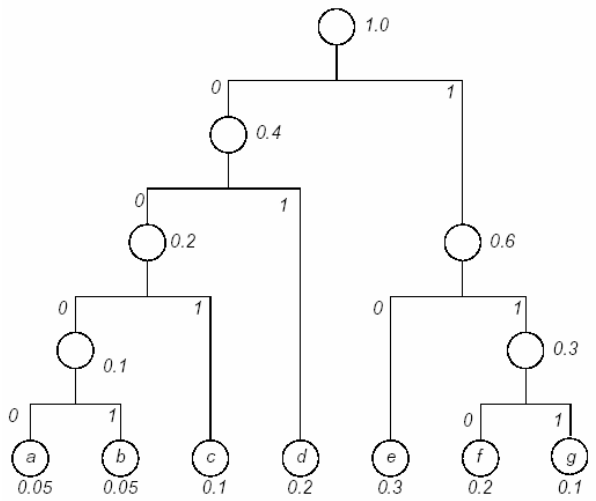
Carattere	Codifica
A	0
B	100
C	101
D	110
E	111

La lunghezza media della parola, applicando la compressione di Huffman al testo di prima risulta **M = (15 * 1 + 7 * 3 + 6 * 3 + 6 * 3 + 5 * 3)/39 = 2.23**.
Avevamo calcolato l’entropia di questo messaggio ed era risultata 2.18, il risultato ottenuto da Huffman è migliore rispettai Shannon-Fano perché è ancora più vicino al valore dell’entropia.
Questo 2.23 contro 2.28 forse non sembra ma è una bella compressione in più, vuol dire che ogni 100 byte di testo in chiaro risparmiamo mediamente 5 bit. Su file di qualche megabyte si fa presto a sprecare kilobyte di memoria.
Se vi ricordate, mentre l’algoritmo di Shannon-Fano non ha capito che il peso di ‘a’ era ben superiore a quello di ‘b’ e ‘c’ messi assieme (15>7+6), l’algoritmo di Huffman controlla sempre il peso di un carattere durante la codifica e lo rimette continuamente in discussione, così da accorgersi che diminuendo il codice di ‘a’ di un bit, a scapito dell’aggiunta di un bit a ‘b’ e di uno a ‘c’, si riduce il codice di 13 bit in meno ma si perdono poi 15 bit, infatti il codice di Huffman recupera questi due bit, rispetto alla codifica di Shannon-Fano, e alla fine la stessa informazione viene codificata in 87 bit, contro 89 !
La codifica di Huffman è superiore alla Shannon-Fano proprio in questo: riesce a stimare meglio il peso che ha la codifica sulla lunghezza totale del testo codificato.

La “comprensione” di Huffman

Per comprendere come lavora la compressione di Huffman è necessario capire com’è fatto e come si forma l’albero.

L’albero di Huffman dispone i simboli (le foglie) lungo i rami in modo che la distanza tra ciascuna foglia e la radice sia maggiore quanto più la probabilità d’incontrare quel simbolo è minore.



Viceversa la distanza tra la radice e la foglia è minore quando c’è una grande probabilità d’incontrare quel dato simbolo.

Alla radice corrisponde la probabilità 1.0 d’incontrare uno dei caratteri contenuti nel testo, infatti la possibilità che nel testo compresso esca un carattere qualsiasi tra quelli codificati è certa.

Ciascuna foglia è disposta nell’albero tenendo conto del peso di ciascun simbolo nel testo in chiaro.

I nodi che legano le foglie sommano le probabilità dei nodi figli (siano essi foglie oppure nodi), cosa che invece Shannon-Fano non fanno!

La via che porta dalla radice ad una foglia terminale (notare lo zero a sinistra e l’uno a destra) è il codice del simbolo, di lunghezza variabile.

Ad esempio in questo caso (l’esempio non è più quello di prima) a=0000, e=10 , g=111, è giusto infatti che il simbolo ‘e’ abbia una lunghezza di codice inferiore, dato che è più probabile incontrarlo nel testo.

Un file codificato secondo questa codifica è per tanto una serie di bit che corrispondono ai percorsi necessari per giungere alle foglie corrispondenti ai caratteri presenti nel file in chiaro. Quindi secondo lo schema dell’esempio la parola “gaffe” viene codificata come 111000011011010 (soli 15 bit contro i 40 della normale codifica ASCII).

A questo punto però sorgono due problemi (per altro comuni a molti tipi di codifica).

1) Ciascuna sorgente di dati ha una propria distribuzione di caratteri al proprio interno, e quindi un albero caratteristico che va creato ogni volta.

2) Il decodificatore deve assolutamente possedere gli strumenti di conversione, altrimenti la serie di bit ricevuta dal codificatore non ha alcun senso.

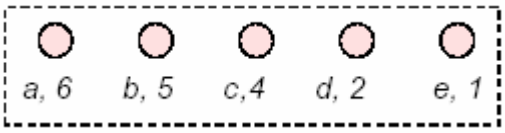
Di questi problemi si fa carico il codificatore stesso, quindi, per ciascuna sorgente dati da codificare deve creare l’albero opportuno, quindi deve includere nel file codificato tutte le informazioni necessarie per una corretta decodifica.

Queste informazioni aggiunte in testa al file, vengono definite col termine header (intestazione).

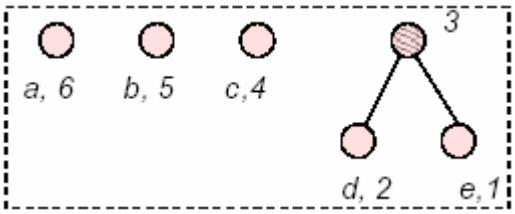
Non è necessario passare l’intero albero al decodificatore, basta passare o una tabella di conversione per convertire direttamente le sequenze di bit in caratteri in chiaro oppure passare le informazioni necessarie alla costruzione di un albero identico.

Come si costruisce un albero di Huffman.

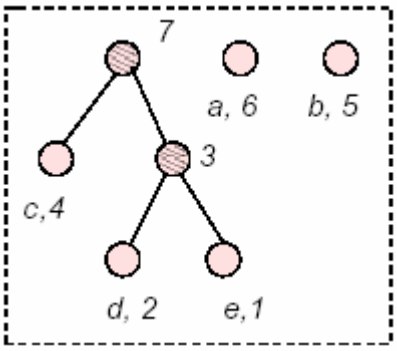
Osservando l'albero nella figura precedente si nota che è costituito da elementi, i quali possono essere foglie o nodi.
Ciascun elemento ha una valore di **probabilità o frequenza** mentre ogni foglia ha un **carattere** (che è l'unità minima d'informazione che costituisce il messaggio che deve essere codificato, lasciando stare il fatto che un carattere è composto da bit… un bit non è comprimibile).
Questi due elementi sono gli unici parametri necessari alla costruzione di un albero di Huffman.
Facciamo un esempio pratico: abbiamo un file che contiene sei lettere 'a', cinque 'b', quattro 'c', due 'd' e una 'e'., ad esempio potrebbe essere file = { *accababaccaddebbabe* }.
Consideriamo ciascun simbolo presente nel file con la propria frequenza d'uscita nel testo e ordiniamoli dal simbolo più frequente al più infrequente.
Otteniamo quindi cinque nodi (in questo caso foglie, ovvero nodi terminali, che contengono un simbolo).



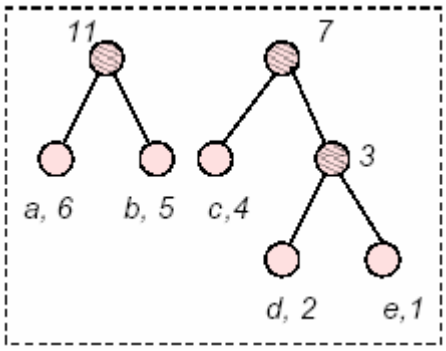
Consideriamo i due simboli meno frequenti, e li sommiamo in un nodo.

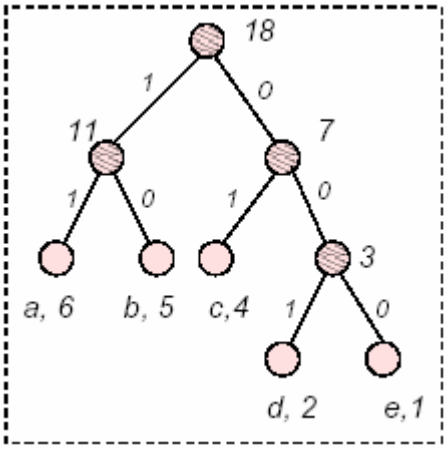


ora il nodo è composto da due foglie e complessivamente il suo peso (o frequenza) è la somma dei pesi, è importante mantenere l'ordinamento tra i nodi.
Consideriamo ancora i due nodi con frequenza minore e creiamo con questi un nodo somma, e riordiniamo subito la sequenza.



Cosa è successo? Semplice, ora la somma dei nodi (c,4) e 3 ha assunto un peso superiore a quello degli altri nodi, e quindi è andato in testa all'ordinamento!
Per costruire il prossimo nodo, consideriamo sempre gli ultimi due nodi in ordine di peso, ora con i nodi (a,6) e (b,5) si costruisce un nodo somma dei due, dato che il loro peso è 11, che è più grande di sette, il nodo passa in testa.



Ancora una volta andiamo a considerare i due nodi con peso minore nella lista, sono ovviamente 11 e 7 (sono rimasti solo loro).
La loro unione in un nodo costituisce la radice dell'albero di Huffman!


A partire dalla radice, qualsiasi spostamento verso destra genera un bit 0 (zero), altrimenti genera un bit 1 (uno), così il simbolo d è associato al codice 001 mentre il codice a è 11, com'è giusto che sia perché è più frequente quindi se è più corto del meno frequente ci fa risparmiare più bit.
A questo punto è possibile sostituire alla sequenza di caratteri (18) del messaggio in chiaro (18*8 bit) la sequenza di bit **11**0**1**0**1**11**10**11**10**11…**000**, corrispondente ad **a**cc**a**b**a**b**a**…**e**, con notevole risparmio di bit.
Il fatto d'assegnare un uno a sinistra e uno zero a destra è del tutto arbitrario, quindi non è detto che un programmatore decida in un senso o nell'altro, per questo ogni encoder deve possedere un proprio decoder.

Decodificare Huffman

Mettiamoci nei panni di un decodificatore che riceve la sequenza codificata appena vista…
Come fa a sapere che 11 corrisponde al carattere **a** e così via tutti gli altri?
Si potrebbe pensare che il codificatore salva anche il codice perché poi il decodificatore lo legga.
L'idea è quella buona, però salvare il codice non è sufficiente: i caratteri dei computer sono a lunghezza fissa (e sono multipli di 8 bit), mentre Huffman codifica a lunghezza variabile, come fare?
Il codificatore non si limita a scrivere i codici, ma fornisce tutti gli strumenti al decodificatore per costruirsi un'albero di Huffman equivalente a quello usato per la codifica.
Queste informazioni costituiscono l' **header**, ovvero l'intestazione di ogni file codificato con questo metodo. Ogni implementazione dell'algoritmo ha il proprio sistema per costruirsi l'header.
E' importante però che codificatore e decodificatore condividano le stesse convenzioni per la memorizzazione dello header e del codice.
Il decodificatore deve ricostruire l'albero di Huffman e per ogni bit che si trova nel file, ripercorrere l'albero fino ad arrivare ad un nodo terminale che identifica il carattere decodificato, tutto questo per ogni carattere presente nel file.

L'implementazione in C++ relativa a quest'articolo

Merita due parole l'implementazione riguardo al contenuto di quest'articolo.
Ci sono tantissime implementazioni, anche molto efficienti e tante varianti, dell'algoritmo di compressione di Huffman. Quella che presento io è sicuramente la peggiore in termini di efficienza, quello che infatti vorrei trasmettere è l'aspetto puramente didattico, una volta compreso il metodo sarà possibile scrivere codice più efficiente, più veloce e meno pesante. Tuttavia dovendosi preoccupare troppo dei dettagli sfuggono poi le linee guida, e così ho deciso di utilizzare contenitori standard del C++ e di non forzare troppo con l'algoritmo e di favorire la linearità dello stesso, quindi il codice non è certo da prendere ad esempio per scrivere un programma come WinZip, tuttavia se non avete mai visto un codice che comprime dati, questo è particolarmente leggibile e comunque funziona.
Il listato è sufficientemente commentato e non starò quindi qui a descriverne tutte le varie fasi, anche perché l'articolo è molto generale e quindi è naturalmente destinato anche a persone che non lo svilupperanno mai in C++ né in altri linguaggi ma sono soltanto interessate a conoscere le tecniche di compressione dei dati (e che quindi non leggeranno nemmeno il codice), mentre le persone che intendono provare l'algoritmo in C++ proposto in queste pagine le suppongo sufficientemente esperte nella programmazione dato che, lo sviluppo di programmi di compressione dei dati è un argomento molto avanzato e inadatto ai principianti. Queste persone sicuramente capiranno i vari passaggi che realizzano ne più ne meno i passaggi descritti nell'articolo.
Il programma è in standard C++ e può quindi funzionare su tutti i calcolatori dotati di un compilatore per tale linguaggio di programmazione.
Praticamente, il programma, sfrutta la codifica di Huffman per comprimere un file e poi consente la sua decodifica, restituendo una copia esatta del file prima della codifica.

Il codice sorgente

```
/*************************************************************************
        Compressione di Huffman
        =======================

        file: main.cpp

        Questo semplice programma usa le classi proposte per creare un semplice
        programma di compressione file.

        huf [ -c | -d ] file_in file_out
        -c comprime
        -d decomprime

        Programma scritto da Paolo Ferraresi (C)2003
        a scopo didattico per comprendere la tecnica di compressione
*************************************************************************/

#include <iostream>
#include <fstream>
#include <cstdlib>
#include "huffman.h"

using namespace std;

void istruzioni(char* bin_name,int returnCode)
{
        cout << "Uso: " << bin_name << " [options] infile outfile\n";
        cout << "-c : applica compressione di Huffman\n";
```

```
        cout << "-d : decodifica\n";
        exit(returnCode);
}

int main(int argc, char* argv[])
{
        ifstream* in;
        ofstream* out;
        if (argc!=4) istruzioni(argv[0],-1);
        char *opt=argv[1];
        if (*opt++!='-') istruzioni(argv[0],-1);
        in=new ifstream(argv[2],ios::binary);
        if (!*in) {
                cout << "Il file non esiste\n";
                delete in;
                exit(-1);
        }
        out=new ofstream(argv[3],ios::binary | ios::trunc);

        Huffman* H=new Huffman(*in,*out);

        switch(*opt) {
                case 'c':       H->Encode();
                                break;
                case 'd':       H->Decode();
                                break;
                default:        istruzioni(argv[0],-1);
        }
        in->close();
        out->close();
        delete in;
        delete out;
        return 0;
}
//------------------------------------------------------------------------------
```

```
/*************************************************************************
        Compressione di Huffman
        =======================

        file: huffman.h

        David. A. Huffman (1925-1999) ha scoperto nel 1953
        un sistema per compattare i dati riducendone la ridondanza.
        La tecnica consiste nella creazione di un albero (Albero di Huffman)
        sul quale vengono disposti, secondo una certa logica data dalla
        distribuzione dei simboli nel testo da comprimere, i vari simboli.
        Partendo dalla radice dell'albero andando verso i codici finali
        si ottiene il codice secondo codifica di Huffman.

        Programma scritto da Paolo Ferraresi (C)2003
        a scopo didattico per comprendere la tecnica di compressione
*************************************************************************/


#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include <map>

using namespace std;

typedef unsigned long ULONG;
typedef unsigned char TDATA; // TDATA potrebbe anche essere un altro tipo!
typedef string bitseq;     // usa una stringa per memorizare sequenze di bit
#define TDATA_MAXC 256

typedef struct { // Symb e' una coppia valore-frequenza
        public:
        TDATA   valore; // e' il valore (di solito e' un byte).
        ULONG   prob;   // prob e' la freq. (o la probabilita') che valore esca
} Symb;
class nodo { // una foglia e' un nodo che ha simbolo impostato, sx e dx = 0
        public:
        nodo(int,ULONG);        // crea una foglia, una coppia valore-frequenza
        nodo(nodo*,nodo*);      // crea un nodo con due figli
        nodo(const nodo&);      // crea un nodo come copia di un'altro
        Symb& Simbolo();        // ritorna la ref a simbolo
        bool isNode();          // e' true se e' un nodo, atrimenti e' foglia
        nodo* Sx();             // ritorna un puntatore al nodo di Sx
        nodo* Dx();             // ritorna un puntatore al nodo di Dx
        bitseq& bitSeq();
        private:
        Symb    simbolo;
        bool    fit;            // fit e' false se e' una foglia, true se e'
nodo
        nodo    *sx,*dx,*father;        // puntatori ai figli e a un padre
        bitseq  bits;   // bits e' il codice di Huffman per quel valore
};

/*************************************************************************
        classe HuffmanTree
        ======================

        lo scopo di questa classe e' quello di fornire un'infrastruttura
        tra la creazione dell'albero  e una classe che materialmente
        va a comprimere oppure a decomprimere un file.
*************************************************************************/
```

```
class HuffmanTree {
        friend class Huffman;
        public:
        HuffmanTree(istream&);
        ~HuffmanTree();
        void    BuildHuffman();         // costruisce la codifica di Huffman
        void    View();                 // mostra la codifica
        bool    EncodeSymbol(const TDATA&, bitseq&);  // codifica un simbolo
        bool    DecodeSymbol(TDATA&,bitseq&);  // decodifica un simbolo

        protected:
        nodo*   root;   // radice dell'albero di Huffman
        istream& in;
        void    huffsize(nodo*,ULONG&); // calcola le dimensioni dell'oggetto
        ULONG   _orig_size;     // dimensioni del testo originale
        short   ns;             // numero di simboli

        // Una mappa viene usata per la codifica, per passare
        // dal codice non compresso, all'albero nel punto dove
        // c'è la foglia codificata
        map<TDATA,nodo*>        EncodeMAP;

        private:

        // make_tree costruisce l'albero a partire dal vettore delle frequenze
        nodo*   make_tree(vector<nodo*>&);       // crea l'albero di Huffman
        void    DestroyRoot(nodo*);     // Distrugge l'albero dalla memoria




        // costruisce la mappa e di fatto codifica secondo Huffman
        void    buildEncodeMAP(nodo*,bitseq);
        void    _init();
        bool    _built;
};

/*********************************************************************
        classe Huffman
        ======================

        Grazie ai servizi offerti dalla classe HuffmanTree, senza
        preoccuparsi di come e' costruito l'albero o come di (de)codifica
        ciascun simbolo, codifica file o dati in memoria (grazie a IOB)
        secondo l'algoritmo di Huffman oppure li legge e li decodifica.
*********************************************************************/

class Huffman: public HuffmanTree {
        public:
        Huffman(istream&,ostream&);

        void    Encode();
        void    Decode();
        ULONG   ComputeEncodeLength();

        private:
        ostream& out;
        void    WriteCode(bitseq&);     // scrive una sequenza di bit
        void    putbit(int);
        int     getbit(void);
};
```

```
/*************************************************************************
        Compressione di Huffman
        =======================

        file: huffman.cpp

        David. A. Huffman (1925-1999) ha scoperto nel 1953
        un sistema per compattare i dati riducendone la ridondanza.
        La tecnica consiste nella creazione di un albero (Albero di Huffman)
        sul quale vengono disposti, secondo una certa logica data dalla
        distribuzione dei simboli nel testo da comprimere, i vari simboli.
        Partendo dalla radice dell'albero andando verso i codici finali
        si ottiene il codice secondo codifica di Huffman.

        Programma scritto da Paolo Ferraresi (C)2003
        a scopo didattico per comprendere la tecnica di compressione
*************************************************************************/

#include <vector>
#include <algorithm>

#include "huffman.h"

using namespace std;
/*
ostream& operator<<(ostream& os,const nodo& N) {
        return os << "'" << (char)N.simbolo.valore << "' --- freq. " <<
N.simbolo.prob;
}
*/
// questa routine serve a std::sort per ordinare il vettore dei nodi
bool sort_routine(const nodo* n1,const nodo* n2) {
        nodo *c1=const_cast<nodo*>(n1),*c2=const_cast<nodo*>(n2);
        return c2->Simbolo().prob<c1->Simbolo().prob;
}
// costruttore di un nodo dal suo valore e dal peso (nodo foglia)
nodo::nodo(int x,ULONG Prob=0) : fit(false),sx(0),dx(0),father(0),bits("") {
        simbolo.valore=static_cast<TDATA>(x);
        simbolo.prob=Prob;
}
// costruttore di un nodo dai puntatore ai figli
nodo::nodo(nodo* Sx,nodo* Dx)
        : fit(true),sx(Sx),dx(Dx),father(0),bits("")
{
        simbolo.valore=0;
        simbolo.prob=Sx->simbolo.prob+Dx->simbolo.prob;
        // fa puntare ai nodi Sx e Dx questo.
        Sx->father=Dx->father=this;
}
// costruttore di copia del nodo
nodo::nodo(const nodo& N)
        : bits(N.bits),fit(N.fit),sx(N.sx),dx(N.dx),father(N.father) {
        simbolo.valore=N.simbolo.valore;
        simbolo.prob=N.simbolo.prob;
}
// restituisce il simbolo di un nodo
Symb& nodo::Simbolo() { return simbolo; }
// dice se il nodo non è una foglia
bool nodo::isNode() { return fit; }
// restituisce il nodo figlio di sinistra
nodo* nodo::Sx() { return sx; }
// restituisce il nodo figlio di destra
nodo* nodo::Dx() { return dx; }
```

```
// restituisce la stringa di bit associata al codice, una volta calcolata
string& nodo::bitSeq() { return bits; }

// costruisce l'albero di Huffman a partire da uno stream di input
HuffmanTree::HuffmanTree(istream& source)
        : root(0),in(source) { _init(); }

// inizializza
void HuffmanTree::_init() {
        _orig_size=0;   // il file è al momento lungo 0 bytes.
        _built=false;   // l'albero non è ancora stato costruito.
}
// il distruttore della classe distrugge l'albero se c'è.
HuffmanTree::~HuffmanTree() {
        if (root) DestroyRoot(root);
}
// costruisce la lista di nodi iniziali (foglie) con un vettore di coppie
// simbolo-frequenza, poi chiama make_tree che costruisce l'albero
void HuffmanTree::BuildHuffman()
{
        // calcola le distribuzione delle frequenze (quante volte ciascun
        // carattere compare nel testo originale, questo serve
        // per ottimizzare la codifica).
        int f[TDATA_MAXC];      // crea un array
        fill(f,f+TDATA_MAXC,0); // azzera tutti i suoi elementi
        int x;
        // scansiona il file e aggiorna le frequenze nell'array
        for (_orig_size=0;(x=in.get())!=EOF;_orig_size++) f[x]++;
        // un array e' efficiente ma non e' flessibile come un vettore e allora
        // mettiamo nel seguente vettore tutti gli elementi comparsi almeno
        // una volta, creiamo nel vettore un puntatore a una foglia
        // che contiene il simbolo e il numero di volte che compare nel file!
        vector<nodo*> v;
        for (x=0;x<TDATA_MAXC;x++) if (f[x]) v.push_back(new nodo(x,f[x]));
        ns=(short)v.size();
        // Ora fa costruire l'albero di Huffman a make_tree e ne ottiene
        // il puntatore!
        root=make_tree(v);

        // Adesso costruiamo la mappa per accelerare la fase di codifica
        buildEncodeMAP(root,"");
        _built=true;    // questo flag indica che la tua CPU e' stata quì!!
}
// distrugge l'albero
void HuffmanTree::DestroyRoot(nodo* HuffmanTree_node) {
        if (HuffmanTree_node->isNode()) {
                DestroyRoot(HuffmanTree_node->Sx());
                DestroyRoot(HuffmanTree_node->Dx());
        }
        delete HuffmanTree_node;
}
// crea l'albero a partire dalla lista di nodi foglia iniziali
// questo è l'algoritmo descritto nell'articolo.
nodo* HuffmanTree::make_tree(vector<nodo*>& V) {
        // Possiamo ora costruire l'albero di Huffman.
        // Per creare l'albero si ordina il vettore delle frequenze (1)
        // si considerano i due elementi con frequenze minori (2)
        // bisogna eliminarli dal vettore delle frequenze.    (3)
        // Ora si mette nel vettore un nuovo nodo (quindi con fit=true) che
        // ha come figli i due elementi precedentemente considerati (punto 4).
        // Questa procedure riparte dal punto 1 e si ferma quando rimane
        // un solo nodo nel vettore (5)
        while (V.size()>1) {
```

```
                // ordina le frequenze, solo se sono sempre ordinate funziona...
                sort(V.begin(),V.end(),sort_routine); // punto (1)
                // preleva i puntatori degli ultimi due nodi (punto 2)
                vector<nodo*>::iterator it=V.end();
                nodo* pdx=*(--it);      // salva temporaneamente il valore
                nodo* psx=*(--it);      // dei nodi
                // li cancella dal vettore (punto 3)
                V.erase(it);    // se non li salvavo in pdx e psx
                V.erase(it);    // ora avrei perso quei valori...
                V.push_back(new nodo(psx,pdx));  // aggiungo il nodo (p. 4)
        }
        // adesso abbiamo un solo nodo nel vettore (punto 5)
        // che e' la radice del nostro albero di Huffman.
        return V[0]; // consegna il puntatore alla storia!!
}

// stampa la codifica di Huffman
void HuffmanTree::View() {
        if (!_built) BuildHuffman();
        map<TDATA,nodo*>::iterator it;
        cout << "Sono presenti " << EncodeMAP.size() << endl;
        cout << "================================================\n";
        for (it=EncodeMAP.begin();it!=EncodeMAP.end();it++) {
                cout    << "Symb: " << it->first << " >>>CODE<<< "
                        << it->second->bitSeq() << endl;
        }
}
// Costruisce una mappa che associa ad ogni valore la giusta foglia nell'albero
// di Huffman
// Se non esistesse la mappa bisognerebbe cercare nell'albero ogni codice prima
// di codificarlo mentre così, l'accesso alla foglia e' diretto!
// In pratica la ricerca ricorsiva all'interno dell'albero viene fatta una volta
// e non ad ogni codifica.
// Infatti, durante la ricorsione dell'albero di Huffman viene fatto il percorso
// che associa ad ogni nodo un 1 per ogni spostamento a sinistra e uno 0
// per ogni spostamento a destra, in  modo che alla fine del percorso
// quando quindi s'arriva ad una foglia, si ha, grazie all'albero
// che abbiamo creato prima, il codice di Huffman
void HuffmanTree::buildEncodeMAP(nodo* nod,bitseq enc_str)
{
        if (nod->isNode()) {
                buildEncodeMAP(nod->Sx(),enc_str+"1");
                buildEncodeMAP(nod->Dx(),enc_str+"0");
        } else {
                nod->bitSeq()=enc_str;
                EncodeMAP[nod->Simbolo().valore]=nod;
        }
}

// Tramite la mappa creata, direttamente dall'albero di Huffman
// codifica un carattere, secondo la codifica.
// se il carattere non e' nella mappa allora ritorna un puntatore
// ai bit nullo!
bool HuffmanTree::EncodeSymbol(const TDATA& x,bitseq& codice) {
        if (!_built) BuildHuffman();
        map<TDATA,nodo*>::iterator it;
        it=EncodeMAP.find(x);
        if (it!=EncodeMAP.end()) {
                codice=it->second->bitSeq();
                return true;
        }
        return false;
}
```

```
// Questa routine decodifica una sequenza di bit.
// Si potrebbe fare ancora in questo caso con la mappa
// confrontando la sequenza da decodificare con una presente nella mappa
// ma tutti quei confronti appesantiscono l'elaborazione (forse...)
// E' probabile che il sistema migliore, che porta al risultato in pochi passi
// e' seguire l'albero dalla radice alla foglia
// L'algoritmo e' volutamente senza ricorsione, perche' e' piu' efficiente
bool HuffmanTree::DecodeSymbol(TDATA& dest,bitseq& bit_sequence) {
        if (!_built) BuildHuffman();
        nodo* huff_tree=root;   // punta alla radice dell'albero
        // punta all'inizio della sequenza di bit da decodificare
        bitseq::iterator it=bit_sequence.begin();
        bitseq::iterator fi=bit_sequence.end(); // e punta alla fine
        // Esce dal ciclo quando la sequenza e' finita o quando
        // si arriva ad una foglia
        while (it!=fi && huff_tree->isNode()) {
                if (*(it++)=='0') huff_tree=huff_tree->Dx();
                        else  huff_tree=huff_tree->Sx();
        }
        // solo se la sequenza di bit termina (it==fi)
        // su di una foglia (huff_tree->fit==false)
        // il risultato e' corretto, altrimenti c'e' un errore
        // probabilmente dovuto ad una sequenza di bit errata!
        if (it==fi && huff_tree->isNode()==false) {
                dest=huff_tree->Simbolo().valore;
                return true;
        }
        return false;
}

// calcola la dimensione (in bit) dell'albero di Huffman
void HuffmanTree::huffsize(nodo* n,ULONG& dimensioni_in_bit) {
        if (n->isNode()) {
                huffsize(n->Sx(),dimensioni_in_bit);
                huffsize(n->Dx(),dimensioni_in_bit);
        } else  dimensioni_in_bit+=n->Simbolo().prob * n->bitSeq().size();
}



/*********************************************************************
        classe Huffman
        ======================

*********************************************************************/

// La classe Huffman accetta un file di input e da un file in output
Huffman::Huffman(istream& source,ostream& dest)
        : out(dest),HuffmanTree(source)
{

}

// routine principale di codifica ad alto livello
// prende un input, crea la codifica e ne scrive tutti i dati codificati
void Huffman::Encode() {
        TDATA x;
        bitseq code;
        // Se l'albero e la mappa di codifica non sono
        // ancora state create le crea
        if (!_built) BuildHuffman();
```

```
        // scrive l'header sull'output
        out.write((char*)&_orig_size,sizeof(_orig_size));
        out.write((char*)&ns,sizeof(ns));
        // scorre la mappa dei simboli e li aggiunge allo header
        map<TDATA,nodo*>::iterator it;
        for (it=EncodeMAP.begin();it!=EncodeMAP.end();it++) {
                Symb b;
                b.valore=it->second->Simbolo().valore;
                b.prob=it->second->Simbolo().prob;
                out.write((const char*)&b,sizeof(Symb));
        }

        // nella creazione dell'albero si legge l'input per fare le
        // statistiche necessarie alla creazine dell'albero
        // ora bisogna riavvolgere l'input per la rilettura
        in.clear();
        in.seekg(0,ios::beg);
        // ora legge, codifica e scrive tutti i caratteri del testo
        while (!in.eof()) {
                x=in.get();
                EncodeSymbol(x,code);
                WriteCode(code);
        }
        // dato che la codifica di Huffman lavora bit a bit mentre invece
        // i file si misurano in bytes, bisogna completare la scrittura
        // di bit affiche' venga scritto l'ultimo byte.
        for (int i=0;i<7;i++) putbit(0);
}

// scrive una sequenza di bit sull'output
void Huffman::WriteCode(bitseq& bit_sequence) {
        bitseq::iterator it;
        for (it=bit_sequence.begin();it!=bit_sequence.end();it++)
                if (*it=='0') putbit(0); else putbit(1);
}
// calcola la lunghezza del file codificato (comprendendo l'header)
ULONG   Huffman::ComputeEncodeLength() {
        if (!_built) BuildHuffman();
        ULONG   dim=0;
        ULONG header_size=sizeof(_orig_size)+sizeof(ns)+ns*sizeof(Symb);
        huffsize(root,dim);
        return (header_size+(dim>>3)+1);
}

void Huffman::Decode() {
        // La prima parte legge l'header
        ULONG   fsize;  // dimensioni del file originale
        short   ns;     // numero di simboli

        in.read((char*)&fsize,sizeof(fsize));   // legge le dim. originali
        in.read((char*)&ns,sizeof(ns));          // legge il numero di simboli

        // costruisce il vettore dei nodi, necessario alla creazione
        // dell'albero, mediante i simboli salvati nell'header
        vector<nodo*> v;
        Symb h;
        for (int i=0;i<ns;i++) {
                in.read((char*)&h,sizeof(Symb));
                // crea un nodo (foglia) e lo inserisce nel vettore
                nodo* n=new nodo(h.valore,h.prob);
                v.push_back(n);
        }
        // Ora fa costruire l'albero di Huffman a make_tree
```

```
        root=make_tree(v);

        // adesso che l'albero e' pronto si possono decodificare i bit
        // del file sorgente, per tutta la lunghezza del file originale (fsize)
        for (unsigned long i=0;i<fsize;i++) {
                // ora ci prepariamo a correre lungo l'albero alla ricerca
                // dei simboli
                nodo* n=root;
                // fino a quando non ho trovato il codice...
                // cioe' finche fit=true (nodo fittizio)
                // legge un bit alla volta se e' 1 va verso sinistra
                // se e' 0 va verso destra, alla fine s'arriva al nodo
                // con il codice decodificato!
                while (n->isNode()) if (getbit()) n=n->Sx();else n=n->Dx();
                out.put((int)n->Simbolo().valore);
        }
}
void Huffman::putbit(int bit)  /* scrive un bit (bit = 0,1) */
{
        static unsigned int buffer=0;
        static unsigned int wmask=128;
        unsigned char cbuf;

      if (bit) buffer |= wmask;
      if ((wmask >>= 1) == 0) {
                out.put((char)buffer);
           buffer = 0;  wmask = 128;
      }
}
int Huffman::getbit(void)  /* legge (0 or 1) */
{
      static unsigned int buffer;
        static unsigned int rmask=0;
      if ((rmask >>= 1) == 0) {
                buffer=in.get();
                rmask = 128;
      }
      return ((buffer & rmask) != 0);
}
```

Occorre mettere i tre file in una stessa directory.
Per la compilazione, sotto **Windows:**
Se avete il *Dev-C++* (http://www.bloodshed.net) ottimo IDE gratuito, aprire il progetto, huf.dev e compilare.
Se avete il Gnu C Compiler GCC, basta entrare da shell nella directory e dare il comando
C:\huf>**make -f Makefile.win**
Altrimenti con qualsiasi altro compilatore basta aprire un progetto C++ da terminale (non GUI), includere i file main.cpp e huffman.cpp quindi salvare il progetto e compilare.

Per compilare e installare il progetto su calcolatori con sistema operativo **Unix** o **GNU/Linux**, basta, come sempre, fare:
$ ./configure
$ make
$ make install

Paolo Ferraresi.
E-mail: micro.byte@mondohobby.it