Linguaggi di Programmazione
Modulo di Laboratorio di Linguaggi di Programmazione
AA 2024-2025
Progetto febbraio 2025
Consegna 7 marzo 2025

Marco Antoniotti e Fabio Sartori

Compressione e decompressione via codifica di Huffman

Lo scopo di questo progetto è di implementare una libreria per la compressione e la decompressione di "documenti". Il metodo di compressione e decompressione è basato sul metodo di Huffman. Questo progetto è ispirato a un capitolo di [AS84]. Dovrete implementare una versione Prolog ed una versione o Common Lisp, o Julia.

Normalmente i *simboli* (di solito *caratteri*) usati per comporre un messaggio (o *testo*) sono codificati mediante una sequenza di bit. Se abbiamo n simboli, in generale abbiamo bisogno di $\log(n)$ bit per poterli distinguere. E.g., la codifica ASCII usa codici di 7 bits e quindi ci permette di distinguere $2^7 = 128$ simboli diversi. Se tutti i nostri messaggi sono costituiti dai soli simboli A, B, C, D, E, F, G e H, allora possiamo usare un codice con solo 3 bits per carattere, ad esempio

A	B	C	D	E	F	G	H
000	001	010	011	100	101	110	111

Con questo codice, il messaggio

BACADAEAFABBAAAGAH

è codificato in una stringa di 54 bits

```
001000010000011000100000101000001001000000000110000111
```

I codici ASCII e quello a 3 bit per i simboli A-H sono detti codici a *lunghezza fissa* (*fixed length codes*). A volte è però utile usare dei codici dove ogni simbolo può essere codificato mediante una sequenza di bit di lunghezza diversa. Questi codici sono detti a *lunghezza variabile* (*variable length codes*). Ad esempio, il codice Morse usa un solo simbolo (il punto) per rappresentare la lettera E, la più frequente in un testo Inglese. In generale, se alcuni simboli appaiono più frequentemente nei nostri messaggi, allora possiamo codificarli con una sequenza più corta di bit; il risultato sarà una codifica più "efficiente" dei nostri messaggi. Considerate questa codifica per il nostro alfabeto A-H.

A	B	C	D	E	F	G	H
0	100	1010	1011	1100	1101	1110	1111

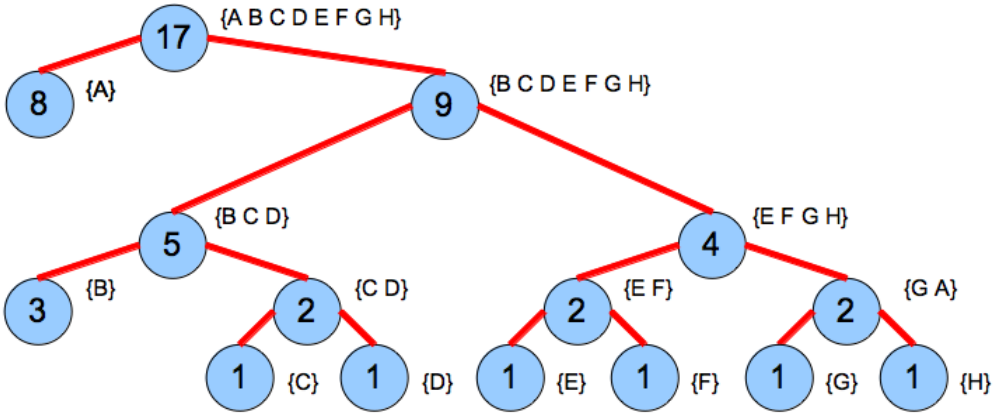
Con questo codice il messaggio precedente è codificato dalla stringa

```
10001010010110110001101010010000011100111
```

Questa stringa contiene solo 42 bits, con un risparmio del 20% rispetto al codice a lunghezza fissa precedente.

Una delle difficoltà nell'uso di codici a lunghezza variabile sta nel decidere quando si sono letti abbastanza 0 e 1 per poter decidere di aver riconosciuto un simbolo. Il codice Morse introduce un terzo simbolo (lo spazio) per *separare* le sequenze di punti e linee. Un'altra soluzione consiste nel disegnare il codice in modo che nessuna codifica completa per un simbolo sia un *prefisso* della codifica di un altro (*prefix codes*). Nell'esempio precedente A è codificato da 0 e B da 100; nessun altro simbolo può avere una codifica che inizia per 0 o per 100.

In generale possiamo ottenere notevoli risparmi di spazio usando dei codici a lunghezza variabile che tengano presente la frequenza relativa dei simboli nei messaggi da codificare. Uno schema particolare che ci permette di ottenere questo risparmio è la *codifica di Huffman* (dal nome del suo inventore). Un codice di Huffman è un *prefix code* che può essere rappresentato da un albero binario le cui foglie contengono i simboli da codificare. Ogni nodo interno dell'albero "contiene" l'insieme di tutti i simboli contenuti nei due sottoalberi. Inoltre, ad ogni foglia è assegnato un "peso" (funzione della sua frequenza relativa) e ad ogni nodo interno la somma dei pesi sottostanti. I pesi non sono usati nelle operazioni di codifica e decodifica; sono usati per costruire l'albero di Huffman.



La figura qui sopra mostra un albero di Huffman per il codice A-H mostrato in precedenza. I pesi delle foglie ci dicono che ad A è stata assegnata una frequenza assoluta pari a 8, a B pari a 3 ed a tutti gli altri simboli pari ad 1 (tali frequenze sono state ricavate dal messaggio di esempio riportato a pagina 1).

Dato un albero di Huffman possiamo trovare la codifica di ogni simbolo partendo dalla radice ed arrivando alla foglia corrispondente al simbolo in questione. Ogni volta che scendiamo a **sinistra** aggiungiamo **0** al codice ed ogni volta che scendiamo a **destra** aggiungiamo **1**. Per ogni nodo interno scegliamo la strada da percorrere controllando a quale sottoinsieme associato alle radici dei sottoalberi appartiene il simbolo da codificare. Per esempio, la codifica di D è destra (1), sinistra (0), destra (1), destra (1): 1011.

Per decodificare un messaggio codificato (una sequenza di bit), partiamo dalla radice e seguiamo il percorso fino ad una foglia a seconda degli 0 (sinistra) ed 1 (destra) che incontriamo. Quando siamo arrivati ad una foglia, abbiamo decodificato un simbolo e possiamo ricominciare dalla radice per decodificare il successivo. Ad esempio, consideriamo la sequenza 10001010. Partendo dalla radice andiamo a destra poiché il primo bit è un 1; successivamente abbiamo uno 0 e quindi andiamo a sinistra; poi ancora a sinistra, dove troviamo una foglia. Abbiamo decodificato B e ripartiamo dalla radice con uno 0; andiamo a sinistra e troviamo la foglia A. Ripartendo dalla radice con il resto del messaggio binario 1010, seguiamo il percorso appropriato nell'albero e ci ritroviamo alla lettera C. Il messaggio decodificato è BAC.

Costruzione di alberi di Huffman

Dato un alfabeto di simboli con le loro frequenze relative, come costruiamo il codice "migliore", ossia l'albero di Huffman che codifica un messaggio con il minor numero di bits?

L'algoritmo è molto semplice. L'idea è di costruire l'albero in maniera tale da avere i simboli meno "frequenti" il più lontano possibile dalla radice. L'algoritmo mantiene come struttura dati un insieme di nodi. I passi dell'algoritmo sono i seguenti.

1- All'inizio l'insieme di nodi contiene tutti i nodi foglia dell'albero. Questo insieme viene costruito creando un nodo per ogni simbolo dell'alfabeto, simbolo con il quale il nodo viene etichettato. Ad ogni nodo viene assegnato un peso pari alla frequenza, relativa o assoluta, associata al simbolo che lo etichetta.

2- A questo punto vengono scelti ad arbitrio due nodi foglia nell'insieme con peso minimo, e viene costruito un nuovo nodo, interno all'albero, che ha le due foglie scelte come figli destro e sinistro. Il nuovo nodo ha come peso la somma dei pesi dei suoi due figli e come etichetta l'insieme di simboli ottenuto dall'unione dei simboli dei suoi due figli. Le due foglie vengono rimosse dall'insieme ed il nuovo nodo viene inserito nello stesso.

3- Il processo si ripete ricorsivamente. Ad ogni passo si rimuovono dall'insieme due nodi dal peso minimo e viene aggiunto un nodo interno allo stesso.

4- Il processo termina quando l'insieme contiene un solo nodo (la radice). Tale nodo sarà etichettato con tutto l'alfabeto, ed avrà come peso la somma delle frequenze di tutti i simboli (1 nel caso di frequenze relative).

Qui di seguito s'illustra come l'albero di Huffman precedente è stato generato a partire dalle informazioni di frequenza assoluta delle lettere A-H ricavata dal messaggio di esempio riportato a pagina 1.

Foglie iniziali	{(A 8) (B 3) (C 1) (D 1) (E 1) (F 1) (G 1) (H 1)}
Fusione	{(A 8) (B 3) ({C D} 2) (E 1) (F 1) (G 1) (H 1)}
Fusione	{(A 8) (B 3) ({C D} 2) ({E F} 2) (G 1) (H 1)}
Fusione	{(A 8) (B 3) ({C D} 2) ({E F} 2) ({G H} 2)}
Fusione	{(A 8) (B 3) ({C D} 2) ({E F G H} 4)}
Fusione	{(A 8) ({B C D} 5) ({E F G H} 4)}
Fusione	{(A 8) ({B C D E F G H} 9)}
Fusione finale	{({A B C D E F G H} 17)}

Attenzione. L'algoritmo è non deterministico, ossia può produrre più di un albero di Huffman. Questo può avvenire poiché non è detto che ad ogni passo ci siano esattamente due elementi dal peso minimo nell'insieme; inoltre la scelta di quale nodo figlio mettere a destra e quale a sinistra al momento della creazione di un nodo padre è arbitraria.

Operazione di decodifica

Come **esempio** d'implementazione di una delle operazioni fondamentali della libreria si presenta la funzione Common Lisp `decode`. La funzione `decode` prende come argomenti una lista di 0 ed 1, ed un albero di Huffman.

```
(defun decode (bits code-tree)
  (labels ((decode-1 (bits current-branch)
             (unless (null bits)
                 (let ((next-branch (choose-branch (first bits)
                                                   current-branch)))
                   (if (leaf-p next-branch)
                       (cons (leaf-symbol next-branch)
                             (decode-1 (rest bits) code-tree))
                       (decode-1 (rest bits) next-branch)))
                 ))
           )
    (decode-1 bits code-tree)))
```

A parte la `labels`, che introduce una (o più) funzione locale, le uniche funzioni che richiedono una spiegazione sono `choose-branch`, `leaf-p` e `leaf-symbol`. Le ultime due dovrebbero essere auto-esplicanti, la prima è implementata come segue.

```
(defun choose-branch (bit branch)
  (cond ((= 0 bit) (node-left branch))
        ((= 1 bit) (node-right branch))
        (t (error "Bad bit ~D." bit))))
```

o, per i più avventurosi...

```
(defun choose-branch (bit branch)
  (ecase (bit)
    (0 (node-left branch))
    (1 (node-right branch))))
```

Dati queste indicazioni (e riferimenti) siete ora in grado di affrontare l'implementazione delle funzioni e predicati che serviranno a costruire la libreria di codifica.

Progetto

L'obiettivo del progetto è di costruire due librerie (Common Lisp e Prolog) che implementino le primitive di codifica e decodifica oltre ad alcune operazioni di utilità.

Common Lisp

Dovete implementare le seguenti funzioni:

hucodec-decode `bits huffman-tree` → `message`

hucodec-encode `message huffman-tree` → `bits`

hucodec-encode-file `filename huffman-tree` → `bits`

hucodec-generate-huffman-tree `symbols-n-weights` → `huffman-tree`

hucodec-generate-symbol-bits-table `huffman-tree` → `symbol-bits-table`

hucodec-print-huffman-tree `huffman-tree &optional (indent-level 0)` → `NIL`

Nella specifica precedente abbiamo i seguenti vincoli

`bits` è una sequenza (*lista*) di 0 ed 1;

`message` è una lista di "simboli" (*simboli*, *caratteri* o altre *strutture dati* Common Lisp; e.g., *liste*);

`file` è una *stringa* (o un *pathname*) che denota un documento nel File System;

`huffman-tree` è un albero di Huffman (la sua radice);

`symbols-n-weigths` è una lista di coppie simbolo-peso

```
( <simbolo> . <peso> );
```

`symbol-bits-table` è una lista di coppie

```
( <simbolo> . <bits> ).
```

La funzione **`hucodec-encode-file`** legge un testo da un file e poi richiama **`hucodec-encode`** su quanto letto.

Le funzioni devono generare degli errori (con la funzione `error`) se codifica e/o decodifica non sono possibili.

La funzione **`hucodec-print-huffman-tree`** stampa a terminale un Huffman Tree e serve essenzialmente per debugging. Non è richiesto un formato perticolare, ma le informazioni stampate devono ben rappresentare la struttura dell'albero di Huffman.

Julia

Dovete implementare le seguenti funzioni:

`hucodec_decode` `bits huffman-tree` → `message`

`hucodec_encode` `message huffman-tree` → `bits`

`hucodec_encode_file` `filename huffman-tree` → `bits`

`hucodec_generate_huffman_tree` `symbols-n-weights` → `huffman-tree`

`hucodec_generate_symbol_bits_table` `huffman-tree` → `symbol-bits-table`

`hucodec_print_huffman_tree` `huffman-tree indent-level = 0` → `Nothing`

Nella specifica precedente abbiamo i seguenti vincoli

`bits` è una sequenza (*vettore*) di 0 ed 1;

`message` è una lista di "simboli" (*simboli*, *caratteri* o altre *strutture dati* Julia; e.g., *vettori*);

`file` è una *stringa* che denota un documento nel File System;

`huffman_tree` è un albero di Huffman (la sua radice);

`symbols_n_weigths` è un vettore di coppie (tuple) simbolo-peso

```
( <simbolo>, <peso> );
```

`symbol_bits_table` è un vettore di coppie

```
( <simbolo>, <bits> ).
```

La funzione **`hucodec_encode_file`** legge un testo da un file e poi richiama **`hucodec_encode`** su quanto letto.

Le funzioni devono generare degli errori (con la funzione `error`) se codifica e/o decodifica non sono possibili.

La funzione **`hucodec_print_huffman_tree`** stampa a terminale un Huffman Tree e serve essenzialmente per debugging. Non è richiesto un formato perticolare, ma le informazioni stampate devono ben rappresentare la struttura dell'albero di Huffman.

Nota Bene. L'implementazione in Julia deve essere "piatta". Ovvero **non** usate moduli.

Prolog

Dovete implementare i seguenti predicati:

hucodec_decode/3 Bits HuffmanTree Message

hucodec_encode/3 Message HuffmanTree Bits

hucodec_encode_file/3 Filename HuffmanTree Bits

hucodec_generate_huffman_tree/2 SymbolsAndWeights HuffmanTree

hucodec_generate_symbol_bits_table/2 HuffmanTree SymbolBitsTable

hucodec_print_huffman_tree/1 HuffmanTree

I vincoli sono gli stessi di cui sopra (ovviamente rimodulati in Prolog). In particolare, le coppie simbolo-peso e simbolo-bits sono rappresentate come termini due elementi:

`sw( <simbolo>, <peso> )` e `sb( <simbolo>, <bits> )`.

I predicati devono *fallire* se ci sono errori o se codifica e/o decodifica non possono essere completate.

Il predicato **hucodec_print_huffman_tree** stampa a terminale un Huffman Tree.

Esempi

L'esempio fondamentale da tener presente (la specifica, secondo una terminologia più corretta) è il seguente.

In Common Lisp:

```
cl-prompt> (defparameter ht
             (hucodec-generate-huffman-tree '<symbols-n-weights>))
HT

cl-prompt> (defparameter message '<some-message>)
MESSAGE

cl-prompt> (equal message
                  (hucodec-decode (hucodec-encode message ht) ht))
T
```

In Prolog:

```
?- assert(symbols_n_weights(<symbols-n-weights>)).
true.

?- assert(message(<some-message>)).
true.


?- symbol_n_weights(SWs),
|  message(M),
|  hucodec_generate_huffman_tree(SWs, HT),
|  hucodec_encode(M, HT, Bits),
```

```
|  hucodec_decode(Bits, HT, M).
true ...
```

Suggerimenti

Come avrete potuto notare non è stata specificata la struttura dell'implementazione di un albero di Huffman.

Un problema che avrete sarà nella gestione d'insiemi ordinati di elementi (foglie e nodi dell'albero in costruzione); dovrete implementare una struttura e/o funzioni che mantengano questi insieme ordinati.

L'implementazione delle varie funzioni e predicati è relativamente semplice una volta che si sfrutti l'ordinamento degli insiemi di nodi e foglie. Se vi trovate a scrivere funzioni o predicati molto lunghi o complessi allora siete probabilmente sulla strada sbagliata.

Note (LEGGERE ATTENTAMENTE)

Una ricerca in rete propone moltissime varianti sul tema. Attenzione! Nessuno vi vieta di ricercare ispirazione e illuminazione in Rete o da parte di qualche AI; però è bene che pensiate attentamente a quello che state implementando; anche perché i motori di ricerca sono usabili da tutti, anche (e soprattutto) dai docenti che vi valuteranno. Inoltre, le AI in circolazione producono codice sub-ottimale alla prima passata, e renderlo compatibile con le specifiche richieste richiede tempo e attenzione.

Da consegnare (LEGGERE ATTENTAMENTE)

Dovrete consegnare un file `.zip` (i files `.tar` o `.rar` **non sono accettabili**!!!) dal nome

```
Cognome_Nome_Matricola_LP_E2P_HUFFMAN_202502.zip
```

`Nome` e `Cognome` devono avere solo la prima lettera maiuscola, `Matricola` deve avere lo zero iniziale se presente. Cognomi e nomi multipli vanno inframmezzati con il carattere '_'; ad esempio: `Pravettoni_Brambilla_Gian_Giac_Pier_Carluca`.

Questo zip file deve contenere **una sola directory con lo stesso nome**.

Ripetiamo: questo zip file deve contenere **una sola directory con lo stesso nome**. In altre parole, prima di consegnare accertatevi che lo zip file si espanda come richiesto.

Al suo interno ci devono essere due sottodirectory chiamate '`Prolog`' (e '`Lisp`' o '`Julia`' per il progetto Lisp associato a questo). Al loro interno queste cartelle devono contenere i files caricabili e interpretabili, più tutte le istruzioni che riterrete necessarie. Il file Common Lisp si deve chiamare '`huffman-codes.lisp`'. Il file Julia si deve chiamare '`huffman-codes.jl`'. Il file Prolog si deve chiamare '`huffman-codes.pl`'. La cartella deve contenere un file chiamato '`README.txt`'. Infine, nella cartella principale dovete avere anche un file chiamato 'Group.txt' che contiene, uno per linea, i nomi dei componenti del gruppo; il file deve essere sempre presente.

In altre parole, questa è la struttura della directory (folder, cartella) una volta spacchettata.

```
Cognome_Nome_Matricola_LP_E4P_HUFFMAN_202107/
       Group.txt
       Lisp/
              huffman-codes.lisp
              README.txt
       Prolog/
              huffman-codes.pl
              README.txt
```

Potete anche aggiungere altri files, ma il loro caricamento dovrà essere eseguito automaticamente al momento del caricamento ("loading") dei files sopra citati.

Il termine ultimo per la consegna è il **7 marzo 2025 alle ore 23:59**.

Valgono le solite regole per lo svolgimento dei progetti in gruppo. Massimo tre persone per gruppo. Ogni file deve contenere – come primo elemento – un commento con l'indicazione di tutti i membri del gruppo. Tutte le persone devono consegnare individualmente.

Valutazione

Valgono naturalmente le seguenti note accessorie.

I progetti di gruppo non implicano una divisione netta del lavoro. Ovvero, un gruppo di due persone non dovrà avere una persona che svolge il progetto Prolog ed una che svolge quello Common Lisp. Il lavoro si può dividere, ma esigiamo che sia collaborativo.

Abbiamo a disposizione una serie di esempi standard che saranno usati per una valutazione oggettiva dei programmi. Se i files sorgente non potranno essere letti/caricati nell'ambiente Prolog (nb.: SWI-Prolog, ma non necessariamente in ambiente Windows, Linux, Mac), nell'ambiente Common Lisp (Lispworks, ma non necessariamente in ambiente Windows, Linux, Mac), o nell'ambiente Julia, il progetto non sarà ritenuto sufficiente.

Il mancato rispetto dei nomi indicati per funzioni e predicati, o anche delle strutture proposte e della semantica esemplificata nel testo del progetto, oltre a comportare ritardi e possibili fraintendimenti nella correzione, può comportare una diminuzione nel voto ottenuto o l'insufficienza. N.B., le funzioni ed i predicati di I/O sono ritenuti *conditio-sine-qua-non*: se non funzionano, il progetto risulterà insufficiente.

Riferimenti

[AS84] H. Abelson, G. J. Sussman, Structure and Interpretation of Computer Programs, MIT Press 1984 (`http://mitpress.mit.edu/sicp`).