Algoritmo di Huffman

Perché si studia?

Alla fine degli anni 40, agli albori della Teoria dell'Informazione, nuove ed efficienti tecniche di codifica cominciavano ad essere scoperte.

L'**entropia** è una misura dell'incertezza con la quale un testo viene prodotto da una sorgente (per esempio un fax, un modem ecc.). In quegli anni si proponeva di realizzare degli algoritmi che riuscissero a codificare testi in modo che questi potessero essere inviati da una sorgente ad una destinazione senza che nessuno, tranne il destinatario, potesse capirne il contenuto oppure algoritmi che potessero permettere al destinatario 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.

Codifica di un testo

Codificare un testo significa sostituire ciascuna parola (o carattere) del testo originario con la corrispondente parola di codice presente nella tabella del codice.

Un **codice è prefisso** se è possibile decodificare correttamente una parola di codice non appena essa è riconosciuta nel testo codificato, poiché nessuna parola di codice è prefisso di un'altra parola di codice

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

Con l'avvento dei moderni calcolatori elettronici si è sviluppato parallelamente alla teoria dei codici il concetto di **compressione dati**. Supponiamo di avere a che fare con un

testo su di un file: 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

E' importante ricordare che, fissato un codice, affinché esso sia **efficiente**, deve accadere che parole di codice associate a caratteri meno frequenti siano costituite da più bit, mentre parole di codice associate a caratteri più frequenti siano costituite da meno bit.

Codifica di Huffman

La codifica di Huffman usa un metodo specifico per scegliere la rappresentazione di ciascun simbolo, esprimendo il carattere più frequente nella maniera più breve possibile.

- Affrontiamo brevemente il problema con un esempio

  Vogliamo comprimere un file testo contenente la stringa: **CIAO_MAMMA**

  Ogni carattere necessita di almeno 3 bit in questo esempio. Le lettere verrebbero quindi codificate nel modo seguente:

  A = 000

  C = 001

  I = 010

  O = 011

  M = 100

  _ = 101

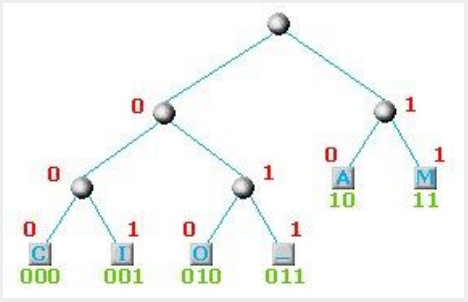
  Soluzione banale: Il nostro file salvato sarà composto così da 30 bits, ovvero 10 lettere * 3 bit.

  001 010 000 011 101 100 000 100 100 000

  Adesso applichiamo l'algoritmo di **Huffman** per calcolare dei nuovi codici da assegnare alle lettere dell'esempio. Si deve costruire un **albero** in cui le lettere più frequenti siano posizionate più vicino alla radice rispetto a quelle con minore frequenza.

  Per prima cosa è necessario contare la frequenza di ogni lettera nella nostra stringa: C(1) I(1) A(3) M(3) O(1) _(1)

  Ancora non sappiamo come, ma supponiamo che Huffman restituisca un albero fatto in questo modo



1. Ogni foglia dell'albero è un carattere
2. ogni arco padre-figlio è associato al valore 0 o 1
3. La parola di codice associata a un carattere è la sequenza di bit lungo il cammino dalla radice alla foglia

Volendo scrivere a questo punto il nostro **CIAO_MAMMA** con i nuovi codici otterremo la seguente sequenza di bits:

000001100100111110111110

Solamente 24 bits invece dei 30 usati in precedenza.

Immaginiamo lo stesso algoritmo applicato ad un file testo di migliaia di parole per capirne la vera potenza. Il guadagno di spazio al termine della compressione è dovuto al fatto che gli elementi che si ripetono frequentemente sono identificati da un codice breve, che occupa meno spazio di quanto ne occuperebbe la loro codifica normale. Viceversa gli elementi rari nel file originale ricevono nel file compresso una codifica lunga, che può richiedere, per ciascuno di essi, uno spazio anche notevolmente maggiore di quello occupato nel file non compresso.

- Spiegazione dell'algoritmo

Pseudo-codice:

```
Huffman(C)

  n = |C|

  Q = C

  for i = 1 to n

       x = node (C[i]) //alloca un nuovo nodo

       Q.insert(x)


  while Q.size is not equal to 1

       z = new Node()

       z.left = Q.extractMin()

       z.right = Q.extractMin()
```

```
        z.f = z.left.f + z.right.f

        Q.insert(z)

end while

return Q
```

L'algoritmo di Huffman è un algoritmo goloso. La scelta riguarda coppie di caratteri e in particolare quelli che hanno frequenza minore. Questi ultimi vengono sostituiti con un carattere fittizio avente come frequenza la somma delle frequenze dei due caratteri. La strategia riconduce a un sottoproblema con dimensione minore. Sul sottoproblema si continua in modo ricorsivo fino alla sua risoluzione.

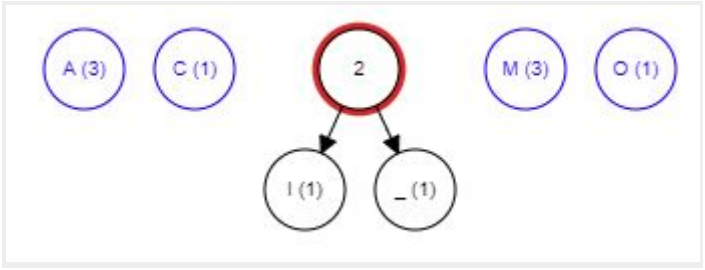
Osservazione: ripetutamente si estraggono i due caratteri con frequenza minore, quindi posso pensare di ordinare in base alla frequenze. L'aggiunta del carattere fittizio mi fa saltare questo ragionamento, poichè l'insieme cambia dinamicamente. Lo strumento per gestire questa strategia è la coda di priorità, usando la frequenza come chiave.

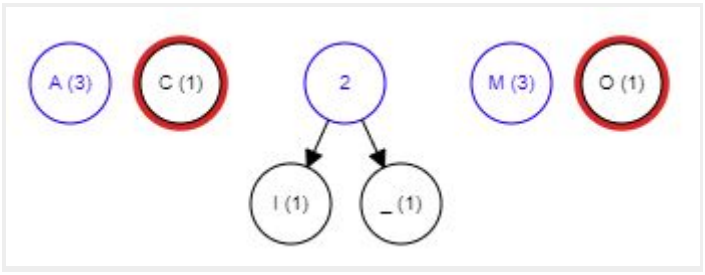
L'implementazione della coda di priorità in questo caso è realizzata con heap binari. Vediamo ora il codice:

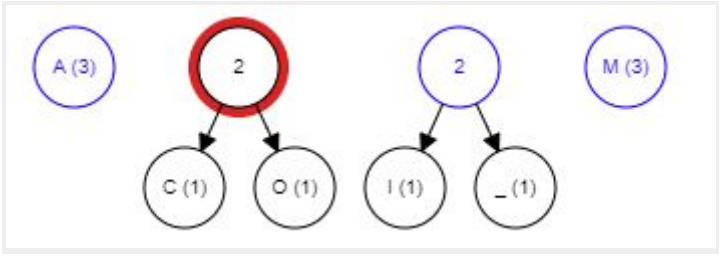
1. Dopo aver creato la coda, si alloca un nodo per ogni carattere e viene inserito in coda. La coda è tenuta in ordine crescente basata sulla chiave, ovvero la frequenza dei nodi
2. Ad ogni ciclo viene allocato un nuovo nodo z "fittizio". Si estraggono dalla coda i due nodi x e y con frequenza minima che diventano figli del nodo z. Il campo frequenza di z viene impostato con la somma delle frequenze di x e y. Z viene inserito nella coda
3. Alla fine estraggo il il nodo radice che mi rimane e l'albero risulta essere costruito

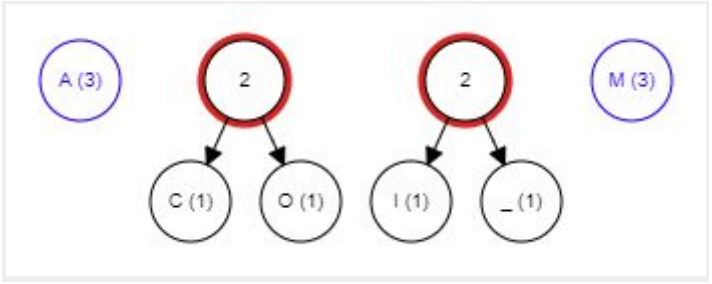
L'algoritmo di Huffman produce un codice prefisso ottimo e quindi l'albero a costo minimo.

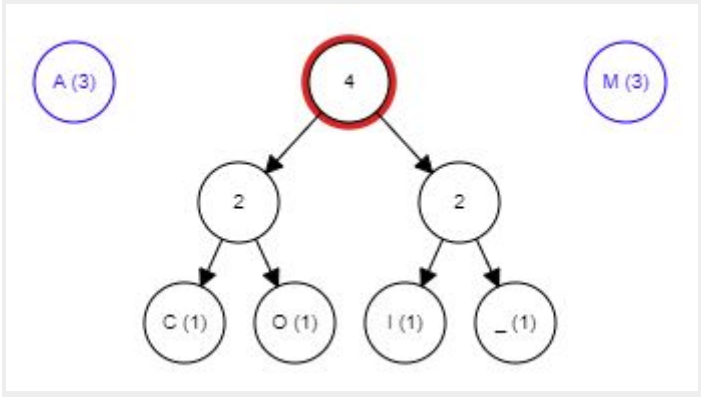
Vediamo passo passo dal punto vista grafico la costruzione dell'albero.

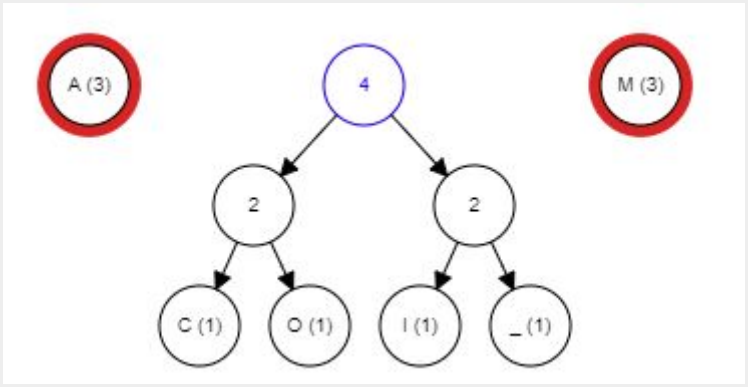

A (3)
C (1)
2
M (3)
O (1)
I (1)
_ (1)

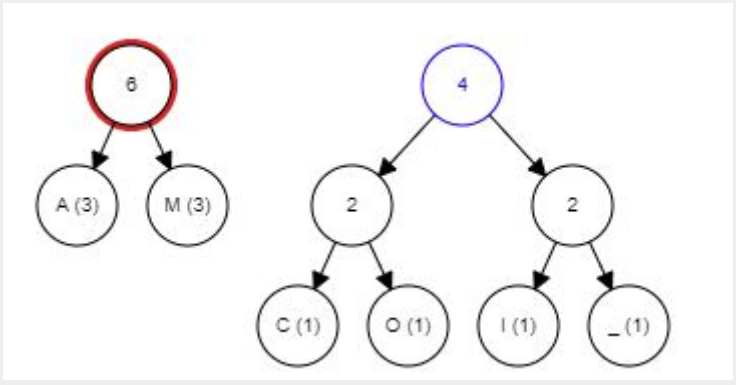

A (3)
C (1)
2
M (3)
O (1)
I (1)
_ (1)

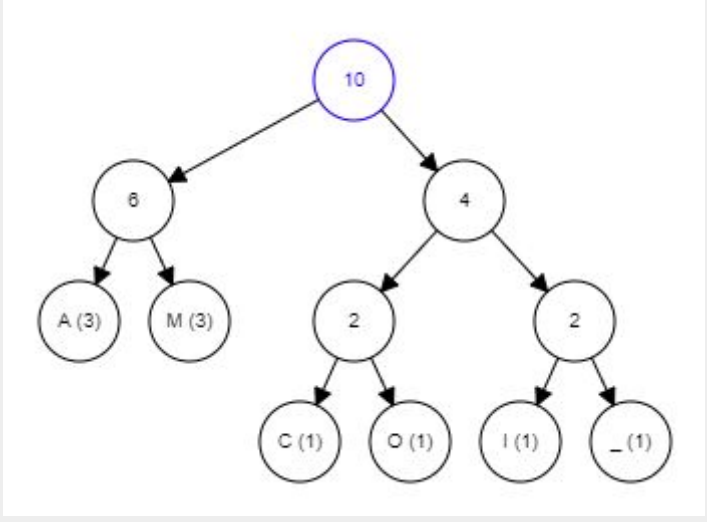

A (3)
2
2
M (3)
C (1)
O (1)
I (1)
_ (1)


A (3)
2
2
M (3)
C (1)
O (1)
I (1)
_ (1)


A (3)
4
M (3)
2
2
C (1)
O (1)
I (1)
_ (1)







- Assumendo che la coda Q venga realizzata con un heap binario, le operazioni **Insert** ed **ExtractMin** richiedono tempo O(log n). Pertanto l'intero algoritmo richiede tempo O(n log n), dove n è il numero di caratteri dell'alfabeto.

  Insert(A, key)

  //inserisce il nodo contenente il valore nella posizione più
  //sinistra del livello più basso dell'albero binario

```
A.heap-size=A.heap-size+1

A.heap-size=key

i=A.heap-size

//Ripristino proprietà heap order

while i>1 and A[PARENT(i)] > A[i]

        scambia A[i] con A[PARENT(i)]

        i=PARENT[i]
```

La chiave viene spinta in alto fino a trovare la posizione giusta, quindi la complessità è nel caso peggiore O(lg n)

Extract-min(A)

```
if A.heap-size==0

        error

min=A[1]

//si scambia il minimo in posizione 1 con l'ultimo elemento che verrà rimosso

A[1]=A[A.heap-size]

A.heap-size=A.heap-size-1

//Ripristino proprietà heap order

MIN-HEAPIFY(A,1)

return min
```

La chiave in posizione 1 potrebbe essere maggiore di quella dei suoi figlio e violare la proprietà heap order. Il nuovo elemento viene spinto in basso fino alla posizione giusta. Nel caso peggiore si discende tutto l'albero con con complessità pari all'altezza dell'albero, quindi O(lg n)

- ❏ Esperimenti

```
run:
taking file as input
USAGE:java MainClass [input.csv]
------------------------------------
String to encode
il gatto prende al volo la palla
------------------------------------
Charcter        Frequency
------------------------------------
i               1
l               6
g               1
a               5
t               2
o               3
p               2
r               1
e               2
n               1
d               1
v               1
------------------------------------
Coding of all single characters
a:00
l:01
o:100
p:1010
n:10110
d:10111
t:1100
e:1101
g:11100
v:11101
i:11110
r:11111
------------------------------------
------------------------------------
String to encode:
il gatto prende al volo la palla
Encoding: 1111001111000011001100100101011111111011011010111111010001111011000110001001010000101000
------------------------------------
------------------------------------
Decoding: ilgattoprendealvololapalla
104.0
Compression Ratio: 81.73076923076923
Il temp di esecuzione è: 0.01
```

L’esempio mostra i risultati dell’algoritmo di Huffman, cioè la codifica con numero variabile di bit dei singoli caratteri. Il testo viene codificato e poi decodificato per un riscontro. Il tempo di esecuzione aumenta con l’aumentare della lunghezza dell’input.