Scuola universitaria professionale della Svizzera italiana
Dipartimento tecnologie innovative

SUPSI

Progetto compressore

Huffman Canonico

Studenti:

Mellace Simone

Data:

22.01.2017

Anno accademico:

2016-2017

Corso di laurea:

Ing. Informatica

Modulo:

Algoritmi

Docente responsabile/ Assistenti:

Montemanni Roberto

Huber David

Vermes Nicola

SUPSI

Indice

SUPSI

Obiettivo

L'obiettivo di questo progetto è quello di riuscire a sviluppare un programma per la compressione e la decompressione dei dati implementando un algoritmo di tipo lossless, ovvero senza perdite di dati nelle due fasi; il tutto dovrà essere sviluppato in C.

SUPSI

Huffman canonico

Introduzione

Inizialmente il mio progetto avrebbe dovuto essere fuso con un LZ77 in modo da poter abbozzare una semplice implementazione del Deflate. Purtroppo però, questo non è stato possibile. Per questo motivo ho implementato un solo algoritmo, ovvero Huffman canonico, il quale si occupa di comprimere e decomprimere dati direttamente su file, senza necessità di appoggiarsi ad algoritmi intermedi.

Nel proseguimento del documento userò più volte il termine “carattere” per indicare gli otto bit, letti in un certo istante da file, i quali in un file di testo potrebbero essere tradotti in un carattere ASCII, da qui “caratteri”. Il motivo che mi spinge a fare questa precisazione è che il byte analizzato perde il suo significato caratteriale se quello che si sta analizzando non è un file di testo.

Descrizione algoritmo e funzionamento

Fase di compressione

Il programma inizia analizzando il file ricevuto e contando le frequenze di ogni carattere presente; una volta eseguito questo lavoro si è pronti per creare l’albero di *Huffman*. L’albero viene creato unendo volta per volta i nodi con le relative frequenze, i quali sono contenuti in un array appositamente creato. In questo modo alla fine del processo, la testa dell’albero di *Huffman* sarà contenuta alla prima posizione dell’array iniziale.

L’albero risulta utile per ricavare la lunghezza binaria di ogni codifica, perciò attraversando l’albero e contando i livelli che si scorrono prima di arrivare a trovare un certo carattere si ottengono le lunghezze delle codifiche.

Una volta trovate le lunghezze delle varie codifiche si può risalire alla codifica binaria di ogni carattere attraverso la canonizzazione. Questo è necessario al fine di poter scrivere su file dei dati compressi.

Ottenute le informazioni necessarie si passa alla fase di scrittura su file. Inizialmente si scrive la *header*, dove sono contenuti 256 byte che riportano le lunghezze delle codifiche di ogni carattere in ordine crescente e un byte che indica quanti bit si devono leggere dall’ultimo byte.

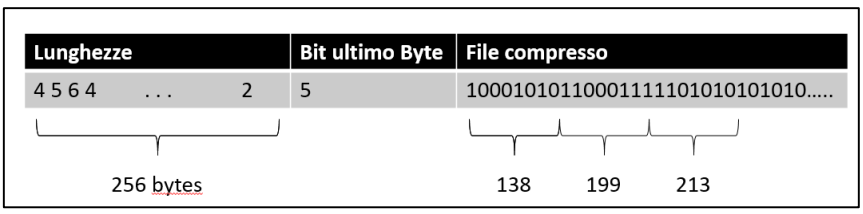

Figura 1: Struttura file compresso

Scrivendo le varie lunghezze seguendo l'ordine crescente delle sequenze di bit da 0 a 255 non è necessario scrivere quale carattere viene codificato a quale posizione. In questo modo la *header* risulta più snella ed ordinata.

Dopo la *header* è possibile iniziare a codificare il documento secondo le codifiche trovate, questo si effettua tramite l'appoggio ad un buffer di otto bit, il quale una volta pieno viene scritto su file in formato decimale.

Fase di decompressione
Grazie alla *header* scritta in precedenza, si possono leggere le lunghezze dei codici e quindi ricostruire, in modo canonico, le varie codifiche. In una fase primitiva del programma si confrontavano poi i bit ricevuti da file con ogni codifica fino al raggiungimento di una valida possibilità; questo però richiedeva molto tempo e non era risultato particolarmente efficiente. Per questo motivo si è deciso di creare un albero contenente tutte le codifiche trovate; in questo modo la ricerca dei bit ricevuti nell'albero risulta più fluida ed efficiente.

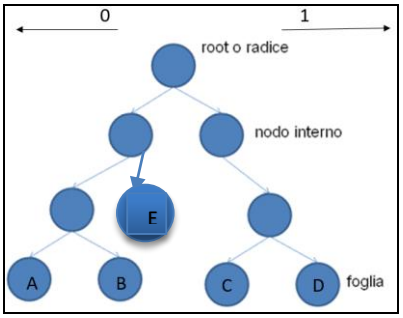

Figura 2: struttura albero per decompressione

Nella decompressione, a questo punto si leggono i numeri decimali trasformandoli in binario e si scorre l'albero creato per cercare il carattere codificato, quando lo si trova viene scritto su file il carattere identificato e si continua così fino al completamento del file.

Descrizione struttura codice sorgente

Il codice si distingue in alcune parti essenziali. Inizialmente (linee 10-21) sono riportate le struct utilizzate durante il progetto, subito dopo le funzioni di debug (23-62) che sono state utili a stampare alberi, array o matrici. Nella terza parte sono presenti le funzioni di inizializzazione delle strutture dati utilizzate (64-115) ed in seguito le varie funzioni di appoggio che eseguono le singole operazioni dell'algoritmo (117-504).

Il flusso del programma viene gestito dal main, che a dipendenza delle immissioni da terminale decide se comprimere o decomprimere, chiamando le rispettive funzioni (572-588). Il cuore dell'applicazione sono proprio le funzioni chiamate, huffmanCod (527-570) e huffmanDec (506-525) si occupano di gestire il programma in fase di compressione e decompressione, richiamando a loro volta le varie funzioni di appoggio.

SUPSI

Descrizione strutture dati

Le strutture presenti nel progetto sono due. La prima è denominata *node* e viene usata per creare l'albero di Huffman, generato in base alle frequenze. Per questo motivo nella struttura sono presenti: il campo *freq*, che indica la frequenza di un certo carattere nel file analizzato; il campo *caracter* che indica appunto il carattere - inteso come gli otto bit letti -; ed infine i relativi puntatori per poter creare un albero.

```
17 typedef struct node {
18     int freq;
19     int caracter;
20     struct node *right;
21     struct node *left;
22 } Node;
```

Figura 3: struttura nodi albero Huffman

Questa struttura viene anche utilizzata nella fase di decompressione per creare l'albero di decodifica. Per questa parte di codice nel campo *freq* non viene inserita una frequenza, bensì un indicatore che prende il valore -1 se il nodo è intermedio o 1 se il nodo è una foglia e quindi contiene un carattere.

La seconda è denominata *canonicalNode* e contiene tutte le informazioni che sono essenziali sia in fase di compressione come anche in fase di decompressione. In particolare questa seconda *struct* contiene: un campo che indica la lunghezza del codice binario – *numBitCodifica* -, il carattere per il quale le informazioni nella *struct* sono valide – *caracter* – e infine la codifica effettiva – *bits* - che nel caso peggiore potrebbe arrivare ad essere lunga fino a 256 bit, dimensione effettiva dell'array.

```
10 typedef struct canonicalNode {
11     int numBitCodifica;
12     int caracter;
13     int bits[256];
14 } CanonicalNode;
```

Figura 4: struttura informazioni e codifiche

Inizialmente le struct presentavano delle differenze, ad esempio il campo del carattere era memorizzato come *char*, oppure in fasi successive come *unsigned char*, questo però portava dei problemi. Leggendo il file 8 bit alla volta e memorizzando ciò che si legge in un carattere si hanno problemi quando quello che si legge da file ha lo stesso valore di *EOF,* ovvero -1, in questo modo si rischiava di non riuscire a leggere il file fino in fondo, ma memorizzando il tutto in un intero il problema non si pone.

È anche importante notare che tenendo in memoria un array di 256 interi per ogni carattere, come viene fatto nella seconda struttura riportata, si occupa in totale 256*256*4 B = 260 kB di memoria RAM. Questa dimensione è trascurabile innanzitutto perché è relativamente piccola, e in seguito perché quando si va a scrivere i dati su file, si scrive solo ciò che è utile lasciando cadere i valori che in precedenza erano inutilizzati.

SUPSI

Esecuzione del programma

Il programma una volta compitalo - gcc huffmanC.c - è invocabile da linea di comando con la seguente sintassi:

nomeFileCompilato opzioni file-input file-output

Dove le opzioni ammesse sono:

-c per comprimere
-d per decomprimere

Ad esempio:

nomeFileCompilato –c file.input outCompresso

nomeFileCompilato –d outDecompresso file.output

Descrizione test effettuati e problemi noti

Testando il programma ci si può accorgere di come esso sia particolarmente efficiente quando si cerca di comprimere file di testo o immagini – salvate in tipi di file non precompressi -. Quando invece si cerca di comprimere file che sono salvati in formati precompressi, l'algoritmo non riesce a comprimere in modo significativo. Questo accade perché nei formati precompressi non si riescono a trovare le ripetizioni necessarie per far funzionare Huffman in modo ottimale; ripetizioni che si trovano in grandi quantità se si analizza un file testuale o ad esempio un immagine salvata nel formato *bmp*. Le ripetizioni trovate fanno aumentare le frequenze dei caratteri e questi riescono a venir codificati in modo ottimale.

Di seguito si riporta una tabella che indica i risultati ottenuti cercando di comprimere diversi tipi di file:

Tipo di file	Dimensione iniziale	Dimensione finale
File .txt	4.6 MB	2.6 MB
File .bmp	380.2 kB	132.1 kB
File .exe	524.3 kB	439.3 kB
File .tiff	3.4 MB	3.2 MB
File .pdf	6.3 MB	6.2 MB
File vuoto	0 B	257 B

Tabella 1: file testati

In genere quando l'algoritmo non riesce a comprimere, il risultato finale migliora o peggiora solamente del 1% rispetto alla dimensione iniziale. Per quanto riguarda invece i casi dove si riesce a comprimere, l'algoritmo effettua una compressione del 50% circa rispetto alle dimensioni iniziali.

SUPSI

Se si cerca di comprimere file di piccole dimensioni l'algoritmo non apporta una compressione, ma ingrandisce il file. Questo accade perché nel file compresso si sfruttano sempre i primi 257 bytes per scrivere le lunghezze delle varie codifiche e il numero di bit da leggere dell'ultimo byte. Perciò se il file da comprimere sarà vuoto o conterrà pochi caratteri, la dimensione minima del file compresso sarà di almeno 257 bytes. Questo non risulta un grosso problema, visto che se si cerca di comprimere un file, solitamente questo sarà di medie/grandi dimensioni.

SUPSI

Sitografia

Generale

https://en.wikipedia.org/wiki/Huffman_coding, consultato il 11 ottobre 2016

https://pineight.com/mw/index.php?title=Canonical_Huffman_code, consultato il 29 novembre 2016

Come funziona

https://www.siggraph.org/education/materials/HyperGraph/video/mpeg/mpegfaq/huffman_tutorial.html, consultato il 11 ottobre 2016

https://mitpress.mit.edu/sicp/full-text/sicp/book/node41.html , consultato il 18 ottobre 2016