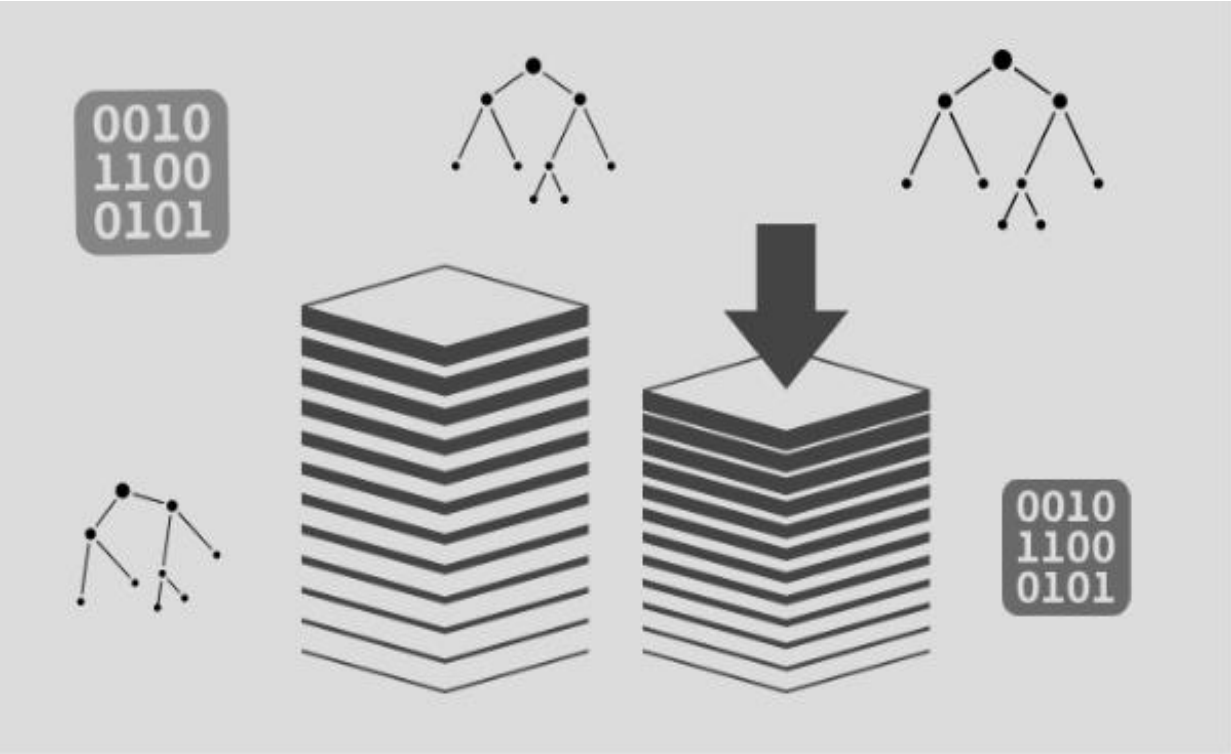


LZS – SHANNON FANO

A HYBRID-LOSSLESS DATA COMPRESSION ALGORITHM

Studenti: Eric Palmas – Mohammad Halabi | Algoritmi e strutture dati | Gennaio 2019

Contenuti

Introduzione 2

Istruzioni per la compilazione 3

- Gerarchia del progetto 3
- Compilazione: 3
- Esecuzione 3
  - Compressione: 3
  - Decompressione: 3

LZS 4

- Strutture dati utilizzate 4
- Logica di funzionamento 5
  - Compressione 5
  - Decompressione 6

Shannon Fano 8

- Strutture dati utilizzate 8
- Logica di funzionamento 9
  - Compressione 9
  - Decompressione 9

Risultati Finali 10

Introduzione

LZS – Shannon Fano è un algoritmo di compressione dati che sfrutta, come si nota appunto dal nome, due algoritmi , LZS e Shannon Fano.

Ognuno dei due algoritmi appartiene ad una categoria di compressione dati chiamata **lossless**, ciò significa che l'algoritmo non porta alla perdita di alcuna parte dell'informazione originale durante la fase di compressione/ decompressione dei dati stessi.

Prima di iniziare la compressione del file vengono effettuati dei controlli sulla correttezza dei parametri passati e sulla dimensione del file, che per essere compresso deve avere una dimensione minore di **260 byte**. Questo specifico numero di byte minimo è legato al fatto che Shannon Fano, usa un header di 257 byte, esso viene necessariamente incluso all'inizio del file compresso, il motivo verrà spiegato più avanti nella parte di logica di funzionamento.

I due algoritmi vengono eseguiti seconda la seguenza:

- Compressione: LZS --> Shannon Fano
- Decompressione: Shannon Fano --> LZS

Istruzioni per la compilazione

Gerarchia del progetto

La directory del progetto contiene le seguenti cartelle:

sources:

- LZSFunctions.c
- ShannonFanoFunctions.c
- Main.c
- makefile

headers:

- LZSFunctions.h
- ShannonFanoFunctions.h

obj: conterrà i files *.o dopo la compilazione

Compilazione:

Per compilare i files sorgenti su **Linux**, posizionarsi nella cartella sources, da un terminale scrivere il comando: **make**

Per assicurarsi che tutto sia andato a buon fine, un file eseguibile chiamato **run** deve essere visibile nella cartella sources.

Esecuzione

I comandi per eseguire la compressione/decompressione sono:

Compressione:

./run -c [fileDaComprimere] [nomeFileCompresso]

Decompressione:

./run -d [nomeFileCompresso] [nomeFileDecompresso]

LZS

Strutture dati utilizzate

```
typedef struct match{
    int length;
    int offset;
} Match;

typedef struct slidingWindow{
    int tail;
    int head;
} SlidingWindow;
```

Match è la struttura che rappresenta una corrispondenza, così come in LZ77 questa è composta da un da un campo offset, che mi indica quanto devo tornare indietro nella finestra per trovare un match e un campo length che mi indica quanto è grande la corrispondeza trovata.

SlidingWindow invece è la struttura che ho utilizzato per rappresentare la finestra scorrevole, la sua dimensione è pari a 2048 elementi ed è composta da una head che indica il primo elemento della finestra e da una tail che indica invece l'ultimo.

Etrambe queste strutture vengono utilizzate in compressione.

Logica di funzionamento

Compressione

Si inizia a scorrere il file e a cercare i match utilizzando la stessa logica di LZ77.
Quello che faccio è confrontare il byte alla posizione corrente con il byte alla posizione relativa ad un contatore, che inizialmente è settato a zero ma che ogni volta che viene trovata un uguaglianza viene incrementato insieme alla posizione corrente. Questo confronto viene effettutato fino a quando il contatore non raggiunge la posizione corrente.

Una volta trovato un match viene scritto sul buffer che poi verrà passato all'algoritmo di shannon fano. Se questo match trovato è minore di due viene scritto un bit pari a zero e in seguito il byte del match, altrimenti si scrive un bit pari a uno ed in seguito un altro bit di controllo per l'offset, se l'offset trovato è minore di 128 viene scritto un bit pari ad uno ed in seguito 7 bit rappresentanti l'offset vero e proprio. Se l'offset dovesse essere maggiore di 128 invece si scrive un bit con valore 0 ed in seguito vengono scritti 11 bit rappresentanti l'offset.

Una volta scritto l'offset mi devo occupare di scrivere la lunghezza del match utilizzando le seguenti regole:

Length	Bit encoding
2	00
3	01
4	10
5	1100
6	1101
7	1110
length > 7	(1111 repeated N times) xxxx, where N is integer result of (length + 7) / 15, and xxxx is length - (N*15 − 7)

Si scrivono i bit relativi alla lunghezza come specificato in tabella, se la lunghezza dovesse essere maggiore di 7 verrà applicata la formula soprastante per trovare la sequenza di bit da scrivere.

Una volta finito di scrivere il match sul buffer sposteremo avanti la nostra finestra scorrevole in base alla lunghezza trovata. Quando il buffer su cui vengono scritti i byte compressi viene riempito al massimo li viene raddoppiata la dimensione, la gestione è molto simile a quella degli arraylist di java.

Una volta arrivati alla fine del file viene scritto l'end marker 110000000, ossia una sequenza di bit che sarebbe impossibile da trovare in decompressione con LZS, in questo modo in decompressione sappiamo che quando incontriamo quella sequenza siamo giunti al termine.

Ultimato quest'ultimo passaggio si può passare alla compressione tramite Shannon Fano.

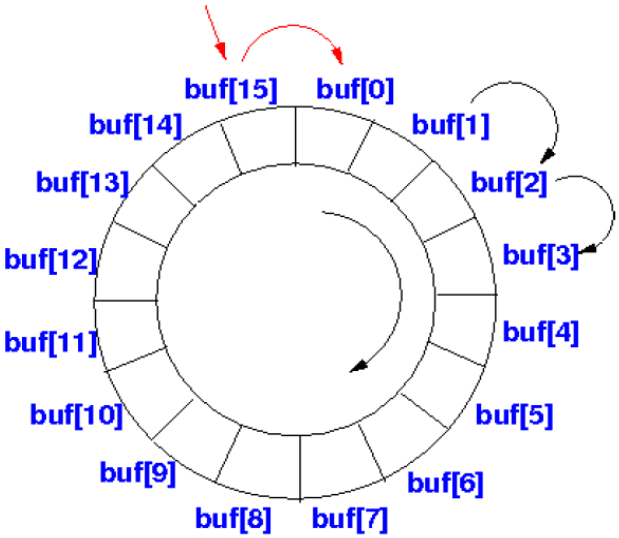
Decompressione

Una volta finita la decompressione con shannon Fano, quest'ultimo passerà ad LZS il buffer con i byte compressi fino a quel momento, e si potrà quindi iniziare a leggere tutti i byte del buffer bit per bit.

per prima cosa viene letto il bit di controllo che come specificato in compressione serve per capire se i bit in seguito rappresentano un singolo byte o un match.

Se il bit di controllo è pari a zero vuol dire che dovrò leggere altri otto bit rappresentanti il carattere, altrimenti se è pari a uno dovrò leggere un'altro bit di controllo che mi indica se l'offset in seguito è contenuto nei successivi 7 o 11 bit, dopodichè leggerò altri 2 bit se questi sono diversi da 11 allora ho finito di leggere il match altrimenti ne leggo altri due, se anche questa volta leggo due bit con valore 1 allora vuol dire che il match ha lunghezza maggiore di 7 e che dovrò continuare a leggere nible fino a quando questi saranno uguali a 1111, l'ultimo nible che leggerò, quello diverso da 1111 sarà la mia lunghezza.

Una volta riconosciuti i byte e le corrispondenze scritte posso aggiugerle al circular bufffer, il quale risulta molto comodo per l'aggiunta dei match, in quanto è possibile avere un accesso diretto ai byte precedenti, cosa che non avevo precedentemente quando utilizzavo la linked list.



Dovrò infatti accedere in modo diretto all'elemento n volte indietro rispetto alla posizione corrente, dove n è pari all'offset, a questo punto basterà leggere tanti elementi quanto è la lunghezza e riscriverli alla fine del buffer.

Shannon Fano

Strutture dati utilizzate

```
typedef struct element {
    unsigned char word;
    int frequency;
    char code[256];
    unsigned char codeLength;
} Element;
```

```
typedef struct node {
    int start, end;
    struct node *rightChild;
    struct node *leftChild;
} Node;
```

Element è la struttura intesa come una rappresentazione di un byte, quanto è frequente nel file da comprimere, la sua codifica e la lunghezza di quest'ultima.

questa struttura viene usata sia in compressione sia in decompressione, i dati vengono aggregati in un array di 256 elementi.

Node è la struttura che rappresenta il nodo dell'albero di codifica di shannon fano.

```
typedef struct codeBits {
    unsigned long long bits;
} CodeBits;
```

```
typedef struct code {
    char code[256];
} Code;
```

CodeBits è la struttura tramite la quale viene generata la codifica canonica, l'utilizzo di un numero a 64 bits facilita le operazioni di codifica come il shift e la somma di 1.

Code è una struttura usata per l'immagazzinamento temporaneo delle codifiche.

Logica di funzionamento

Compressione

Nella fase di compressione, una volta l'algoritmo LZS ha terminato il suo processo, l'algoritmo Shannon Fano parte come segue:

- legge i bytes contenuti nel buffer passato da LZS.
- Una funzione calcola le frequenze per ogni byte.
- Viene eseguito un sort per frequenze decrescenti.
- Dopo di che viene generato l'albero di codifica di shannon fano basandosi sulle frequenze, i bytes più frequenti avranno una codifica corta.
- Le lunghezze delle codifiche vengono utilizzati per generare la codifica canonica.
- All'inizio del file compresso viene inviato un header che contiene:

  1 byte di flag, che indica il numero degli ultimi bits da trascurare durante la fase di decompressione.

  256 bytes, che rappresentano le lunghezze delle codifiche.

- Per ogni byte presente nel buffer viene scritta la codifica corrispondente sul file compresso.

Decompressione

L'algoritmo Shannon Fano inizia la decompressione come segue:

- Legge il header di 257 bytes all'inizio del file compresso.
- A partire dai 256 bytes di lunghezze ordinati in modo crescente, viene ricostruita la codifica canonica.
- Ora che si hanno le codifiche, viene costruito l'albero di decodifica che serve ad identificare il byte da scrivere in output leggendo una sequenza di bits in input.
- I dati contenuti nel file compresso (dal byte 258 alla fine del file) vengono letti e messi in un buffer, che percorrendolo bit per bit, e tramite l'albero di decodifica, i dati decompressi vengono immagazzinati in un altro buffer di output che sarà passato all'algoritmo LZS.

Risultati Finali

La tabella sotto, mostra le informazioni riguardanti i test effettuati su alcuni files.

I tempi di esecuzione dipendono dalla velocità del processore e dalla tecnologia del disco fisso HDD/SSD, possono quindi variare leggermente da una macchina all'altra.

File	Peso	Compresso	Peso ridotto* del	Tempo compressione in secondi			Tempo decompressione in secondi		
				LZS	SHF	LZS-SHF	SHF	LZS	LZS-SHF
empty	0	Il compressore non si avvia perché il file è vuoto o troppo piccolo							
32k_ff	32,768	422	-98.71%	0.312	0.007	**0.319**	0.001	0.012	**0.013**
32k_random	32,768	37,153	13.38%	0.259	0.018	**0.277**	0.016	0.020	**0.036**
alice.txt	167,518	84,018	-49.85%	0.459	0.028	**0.488**	0.053	0.063	**0.116**
immagine.tiff	3,352,968	3,462,637	3.27%	19.510	0.736	**20.246**	1.200	0.419	**1.619**
10mb.tiff	10,151,438	5,301,536	-47.78%	28.404	1.265	**29.669**	2.016	1.156	**3.172**
enwik8	100,000,000	49,803,273	-50.20%	224.298	10.499	**234.797**	14.668	11.156	**25.824**

$$* \, Peso\ ridotto = \left(1 - \frac{File\ compresso}{File\ originale}\right) \times 100$$