LZS (Lempel Ziv Stac) con Hash Table

Mohamed Boutaleb

Scuola Universitaria Professionale della Svizzera Italiana, Manno, Svizzera

mohamed.boutaleb@student.supsi.ch

Abstract. Spesso quando si parla di trasferimento dati si parla di dimensioni dei dati da trasferire. Difatti le dimensioni stabiliscono la velocità di trasferimento e i mezzi da adottare[1]. Comprimere dati dunque risulta molto importante. Oggigiorno sono in molti gli utenti che usano software di compressione dati. LZS acronimo dei creatori dell'algoritmo (Lempel, Ziv e Stac Electronics), è un algoritmo lossless che sfrutta la finestra scorrevole di LZ77 e la codifica di Huffman[2]. L'algoritmo in precedenza è stato sviluppato per essere usato a livello hardware, nei protocolli di trasferimento poi successivamente per scopi generali e per sistemi di registrazione di informazioni su supporti intercambiabili[3]. Questa implementazione rispetto a quella standard usa hash table per la ricerca del matching e una finestra scorrevole di 256 byte per migliorare i tempi d'esecuzione. In questo paper si mostrerà il design dell'algoritmo e come sono state affrontate alcune scelte implementative.

Keywords: Compressione dati, load factor, trasferimento dati, LZ77, Huffman fixed code, hash table, sliding window, linear probing, collisioni

1 Definizioni

Di seguito alcune definizioni utili in ordine alfabetico definite dallo standard[3], [4]:

- **Compressing Ratio** (Rapporto di compressione): Il rapporto tra il numero di byte dati in input all'algoritmo di codifica e il numero di byte in output dall'algoritmo[3].
- **End marker**: Uno schema di bit unico presente nel flusso di uscita che rappresenta il fine di un blocco di dati compressi. Rappresentato da 9 bit (110000000)[3].
- **History buffer** (Buffer cronologico): Il metodo di codifica della compressione richiede il riferimento ai più recentemente elaborati 2048 byte di dati di ingresso. Questi 2048 byte sono definiti nella finestra scorrevole[3].
- **Lossless compression** (Compressione senza perdita dati): Una compressione tecnica che permette la ricostruzione completa del flusso di dati di input originale senza l'introduzione di errori[3], [4].
- **Matching pattern**: Un pattern a byte multipli che risiede nel buffer di storia che è identico al source pattern[3].
- **Raw byte token**: Uno schema a 9 bit nel flusso di uscita che rappresenta un singolo byte dal flusso di ingresso più il nono (MSB) bit settato a 0[3].
- **Source pattern**: Uno schema a byte multipli in entrata all'algoritmo di codifica che è identico ad un match pattern[3].
- **String token**: Uno schema di bit di lunghezza variabile nel flusso di uscita che rappresenta un source pattern[3].

2 Struttura progetto

Il design pattern che si è cercato di simulare è simile all'architettura multilivello, dove ogni layer ha le sue responsabilità. In questo progetto ci sono in totale 3 layer. Ogni layer svolge un'operazione specifica. In totale ci sono 7 file sorgente. Di seguito le responsabilità di ogni file:

- **Algorithm_controller** (application layer)*:* si occupa di fare i controlli sull'operazione da effettuare (-c per comprimere e -d per decomprimere) e di chiamare il modulo *encoding_service* o *decoding_service* a seconda dell'opzione.
- **Encoding_service** (business e service layer)*:* E' il modulo che si occupa di comprimere i dati e tenere aggiornati l'history e l'holding buffer.
- **Decoding_service** (business e service layer)*:* si occupa di decomprimere e aggiornare solamente l'history_buffer.
- **Buffer_service** (business e service layer)*:* scrive su un buffer di 1 byte i bit generati durante la codifica e la decodifica.
- **Hash_service** (business e service layer)*:* usato per cercare i match pattern e gestire le hash table. Definisce una hash function alloca memoria per i match e gestisce tutte le collisioni generate dalla hash function.
- **Data_repository** (layer repository): usato per interfacciarsi con i file di input e output e leggere o stampare il buffer di input e output di dimensioni di 1<<20 (circa 1MB). Questo per diminuire le richieste di lettura e scrittura al sistema operativo ottenendo una migliore performance.
- **Utils***:* usato per alcune funzioni utili ad alcuni moduli come il metodo per copiare una stringa o compararla con un'altra.

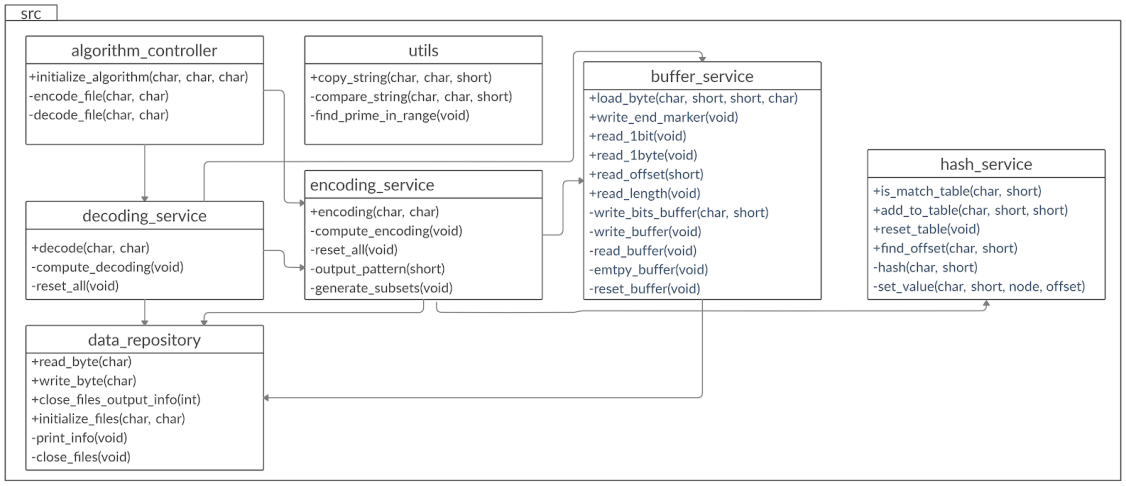

Figura 1. Le frecce indicano le dipendenze con gli altri file

3 Hash Table e Hash Function

Questa sezione risulta più importante discutere prima delle altre. Invero è stata la prima ad essere stata considerata sia durante la fase di analisi preliminare del progetto e che durante la fase conclusiva.

Una buona funzione hash ricordiamo essere una funzione che dovrebbe mappare gli input previsti nel modo più uniforme possibile nel suo output previsto, essere veloce ed evitare collisioni[5]. Facendo dei test e consultando diverse fonti si è deciso di adottare ***Murmur2*** (la stessa usata da unordered_set di C++). Creata da Austin Appleby nel 2008 è una funzione di hash non crittografica adatta per la ricerca generale basata su

hash[6]. Il codice è stato preso in riferimento dalla seguente fonte[7]. Questa ha generato valori di collisioni molto basse rispetto alle altre funzioni di hashing.

Per quanto riguarda le dimensioni della hash table sono state scelte in base al numero massimo di input che la hash table può avere. Siccome la dimensione dell'history buffer è di 256 (WINDOW_SIZE) caratteri, allora il massimo numero di pattern che si possono avere nella hash table è di **(WINDOW_SIZE*(WINDOW_SIZE/2)-WINDOW_SIZE/2)**. Un buon load factor di una hash table dovrebbe essere di 0.75 il quale dovrebbe offrire un buon compromesso tra tempi costi e memoria[8], quindi la dimensione dovrebbe essere: MAX_ENTRIES*1.33 e si prende il primo numero primo maggiore del risultato. Ma siccome il numero di collisioni risultava essere ancora alto, allora si è deciso di aumentare le dimensioni della hash table per diminuire il numero di collisioni a fronte di un load factor più basso. La dimensione è quindi il primo numero primo maggiore di (1024*(512)-512)*1.33 ovvero **696623.**

Le collisioni sono state gestite utilizzando linear probing. Questo risulta essere più veloce rispetto ad altre tecniche come chaining con open hash[9].

All'inizio la struttura dell'hash table era strutturata come un array di nodi, dove la struttura dei nodi era configurata in questo modo:

```
struct node{
   uc * value; // Rappresentava la il match pattern salvato
   us pos_hb; // Rappresentava la posizione nel history_buffer
   us value_length; // Rappresentava la lunghezza della stringa
};
```

Successivamente dopo un'analisi sui tempi di esecuzione (tempo di allocazione e deallocazione memoria di ogni singolo nodo) e sulla quantità di memoria allocata sullo heap si è deciso di suddividere l'hash table in 3 array. Il primo array non è altro che un array di stringhe di tutti i matching pattern (allocati nella memoria heap), il secondo array contiene tutte le posizioni nell'history buffer dei matching pattern e infine l'ultimo contiene le lunghezze di questi matching pattern.

4 Compressione

Per comprendere al meglio la procedura della compressione risulterebbe più efficace mostrare un flow chart di tutta la procedura di acquisizione.

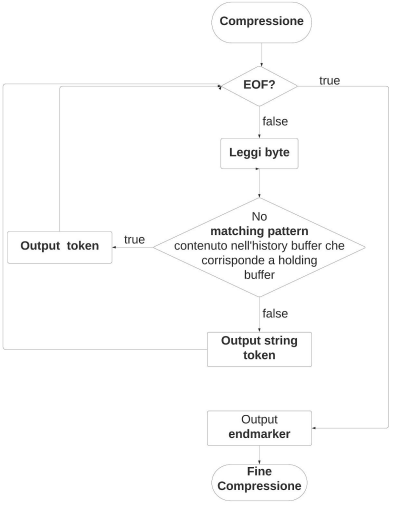

Figura 2. Flow chart compressione

Ognuna di queste procedure (vedi Figura 2) viene poi suddivisa in operazioni più specifiche. L'operazione "Leggi byte" in Figura 2 immette il byte letto sia nell'history buffer che nel holding buffer. L'history buffer come già definito nel paragrafo 1 contiene gli ultimi 256 byte letti, mentre l'holding buffer contiene il source pattern di lunghezza minima di 1. Per quanto riguarda il controllo del matching si genera una hash del source pattern e se si trova un valore uguale nella hash table allora si ritorna un valore minore delle dimensioni della hash table, altrimenti uno maggiore. Si ricorda che l'hash table si resetta ogni qualvolta che lo fa l'history buffer. Di seguito lo pseudo codice delle altre due operazioni, ossia, output token e output length:

```
Process = Output_token.
If (number of bytes in holding_buffer ≤ 2).
   Put single 0 bit to output bit stream.
   Put oldest byte in holding_buffer to output bit stream.
   Clear the oldest byte from the holding_buffer.
Elseif.
   Put single 1 bit to output bit stream.
   If (Offset ≤ 127).
      Put single 1 bit to output bit stream.
      Put 7-bit binary value of offset to output stream.
   Elseif.
      Put single 0 bit to output bit stream.
      Put 11-bit binary value of offset to output stream.
   Endif.
   Output_length.
   Clear all bytes from the holding_buffer except the newest byte.
Endif.
Endprocess.
```

```
Process = Output_length.
Set X to (number of bytes in holding_buffer – 1).
If (X ≤ 4).
   Put 2-bit binary value of (X – 2) to output stream.
Elseif.
   If (X ≤ 7).
      Put 2-bit pattern with all bits set to a 1 bit to output stream.
      Put (2-bit binary value of (X – 5) to output stream.
   Elseif.
      Put 4-bit pattern with all bits set to a 1 bit to output stream.
      Set X to (X – 8).
      While (X ≥ 15).
         Put 4-bit pattern with all bits set to a 1 bit to output stream.
         Set X to (X – 15).
      Endwhile.
      Put 4-bit binary value of X to output stream.
   Endif.
Endif.
Endprocess.
```

Figura 3. Pseudo codice output token e output length

Fonte: da[3]

Per quanto riguarda le operazioni finali, queste comprendono la deallocazione della memoria e infine la scrittura dell'end marker seguito da tanti zeri quanti sono i bit restanti per completare il byte. Di seguito lo pseudo codice:

```
Process = Flush.
While (number of bytes in holding_buffer > 0).
   Output_token.
Endwhile.
Put 9-bit pattern with b8 and b9 set to 1s and bits b1 through b7 set to 0s to output stream.
If (desired to clear the history).
   Clear all bytes from the history_buffer.
Endif.
Endprocess.
```

Figura 4. Pseudo codice flush operation

Fonte: da[3]

5 Decompressione

La decompressione una volta compresa la compressione risulta essere molto intuitiva. Difatti è stato il modulo che ha preso meno tempo per lo sviluppo. La sua esecuzione è molto rapida anche per file di grosse

dimensioni, ciò è dovuto al fatto che questo modulo non usa hash table e non alloca/dealloca memoria nello heap. Inoltre necessita soltanto dell'history buffer per ricostruire il dato.

Come risulta evidente dalla codifica se il primo bit letto è zero ciò vuol dire che bisogna leggere un *literal byte* altrimenti si tratta di uno string token o dell'end marker, ricordando che la differenza tra i due è che l'end marker è costituito da 9 bit con il nono e l'ottavo settati a 1 e il resto a zero, mentre lo string token è seguito da 8(se offset<=127) o 12(offset>127) bit che rappresentano l'offset seguito dalla lunghezza che può essere rappresentata da 2, 4, 8, 12 ecc bit[4].

6 Ottimizzazioni

Per ridurre al minimo il numero di richieste al sistema operativo per la lettura e scrittura dei byte elaborati dall'algoritmo, si è deciso di utilizzare un buffer di scrittura e lettura di **1 MB** (gestito dal data_repository)**.** Inoltre dato che le funzioni di copia o uguaglianza tra stringhe messe a disposizione dal linguaggio non sono adatte per confrontare o copiare stringhe contenenti '\0' in mezzo alla stringa, allora si è deciso di sviluppare queste due funzioni basandosi sulla lunghezza delle due stringhe (funzioni contenute nel sorgente utils.c). Infine considerata la dimensione della hash table e del suo load factor si è deciso di immagazzinare tutte le posizione realmente occupate in un array; questo utile per risparmiare tempo durante la deallocazione della memoria delle stringhe contenenti dati della hash table.

7 Compilazione esecuzione e distribuzione

Per eseguire il file è necessario usare i target del Makefile. Siccome non si possono passare argomenti al target di esecuzione (run), è necessario prima fare lo static client con il seguente comando (da eseguire nello stesso percorso del Makefile. Creerà tutti i file oggetto e anche la libreria statica nella cartella *lib/*):

```
$ make static_LZS_1.0.0
```

Successivamente verrà creato il client statico nella cartella *dist/* e si potrà eseguire il client statico (sempre all'interno della cartella *dist/*) con il seguente comando:

```
$ ./static_LZS_1.0.0 <option> <input_file> <output_file>
```

- <option> accetta i valori "-c" o "-d" che specificano se effettuare la compressione o decompressione del file rispettivamente.
- <input_file> specifica il nome dell'input file presente nella directory corrente
- <output_file> specifica il nome dell'output file (non per forza esistente nella directory)

8 Test e Debugging

I test all'inizio sono stati effettuati principalmente su file di testo con dimensioni ridotte fino ad arrivare con immagini di grandi dimensioni. Con il sostegno dell'IDE Clion sono state testate tutte le operazioni di codifica, decodifica e ricerca di matching patterns. Il tutto con l'uso dei break point messi a disposizione dall'IDE. Inoltre è stato adottato l'uso di variabili per tenere conto di quante collisioni genera la funzione di

hash. Il tutto cronometrato in secondi e stampato in output nel terminale insieme al compressing ratio. Sulla base quindi di questi dati è stato fatto un compromesso tra tempi di compressione e compressing ratio. Difatti per migliorare le tempistiche a fronte di un fattore di compressione più basso è stato scelto di usare al posto di 2048 byte di holding e history buffer una dimensione di 256 byte. Questo per diminuire di conseguenza anche le dimensioni della hash table e quindi delle collisioni. Di seguito i risultati che testano la differenza in termini di tempo e compressing ratio tra l'uso dell'history buffer da 2048 byte e quello da 256 byte in una macchina i7-6500 CPU 2.5 GHz:

	tempo con window size di 2048	compressio n ratio con window size 2048	tempo con window size di 256	compressio n ratio con window size 256
immagine.tiff	**8430 s**	**1.2%**	**392 s**	**-7.23%**
ff_ff_ff	**0 s**	**-33.33%**	**0 s**	**-33.33%**
empty	**0 s**	**-5%**	**0 s**	**-5%**
alice.txt	**667 s**	**39.26%**	**19 s**	**19.49%**
32k_random	**135 s**	**-11.81%**	**4 s**	**-12.37%**
32k_ff	**87 s**	**96.54%**	**2 s**	**95.60%**

[1] "[No title]." https://amslaurea.unibo.it/9150/1/Andrea_Rinaldi_tesi.pdf (accessed Aug. 26, 2020).

[2] Contributors to Wikimedia projects, "Lempel–Ziv–Stac," Jul. 21, 2010. https://en.wikipedia.org/wiki/Lempel%E2%80%93Ziv%E2%80%93Stac (accessed Aug. 26, 2020).

[3] AMERICAN NATIONAL STANDARD ANSI X3. 241-, "American National Standard for Information Systems – Data Compression Method – Adaptive Coding with Sliding Window for Information Interchange," Accessed: Aug. 23, 2020. [Online]. Available: http://masters.donntu.org/2003/fvti/boykov/library/lzs.pdf.

[4] American National Standards Institute, *American National Standard for Information Systems: Data Compression Method: Adaptive Coding with Sliding Window for Information Interchange*. 1994.

[5] C. S. Jutla and A. C. Patthak, "Provably Good Codes for Hash Function Design," *Selected Areas in Cryptography*. pp. 376–393, doi: 10.1007/978-3-540-74462-7_26.

[6] Contributors to Wikimedia projects, "MurmurHash," Apr. 09, 2009. https://en.wikipedia.org/wiki/MurmurHash (accessed Aug. 25, 2020).

[7] "murmurhash." http://murmurhash.googlepages.com/ (accessed Aug. 26, 2020).

[8] T. Mailund, "Collision Resolution, Load Factor, and Performance," *The Joys of Hashing*. pp. 21–47, 2019, doi: 10.1007/978-1-4842-4066-3_3.

[9] jephos, "What are some advantages of linear probing over separate chaining?," Oct. 19, 2018. https://discuss.codecademy.com/t/what-are-some-advantages-of-linear-probing-over-separate-chaining/370071 (accessed Aug. 26, 2020).