CONFRONTO LZ

# 1.Introduzione

Per il progetto di laboratorio riguardante gli algoritmi di compressione/decompressione, abbiamo scelto di sviluppare il Confronto LZ in quanto eravamo maggiormente interessati a studiare l’approccio di diversi algoritmi.

I due algoritmi scelti sono LZ77 e LZ78 sviluppati rispettivamente da Luca e Nicol.

La scelta di questi due algoritmi è da ricercarsi nel fatto che seppur sono poco recenti li si ritrovano in più algoritmi che si occupano di compressione(vedi deflate,LZSS); inoltre è curioso sapere come si comportano questi due algoritmi in situazioni simili.

# 2.LZ77

Funzionamento di LZ77

LZ77 è un algoritmo che si basa sulla ricerca di pattern all’interno di un file, nella mia implementazione di questo algoritmo vi è un approccio iterativo nella ricerca di corrispondenze.

Questo algoritmo è un algoritmo detto a finestra scorrevole in quanto si basa sul movimento all’interno del buffer di ricerca (Search buffer) e la dimensione di questo buffer mi indicherà la massima distanza tra la posizione attuale e l’occorrenza da ricercare.

Mentre la dimensione del LookAhead buffer mi permetterà di limitare la lunghezza di un’occorrenza.

Lunghezza Search/LookAhead buffers

Inizialmente ho optato per la scelta di un Search buffer anche detto dictionary di lunghezza 255 byte e impostato anche il LookAhead buffer a 255 byte.

Questa scelta portava il mio algoritmo ad avere degli ottimi tempi di compressione ma come aspetto negativo aveva il rapporto di compressione in quanto risultava minimo.

In seguito, dopo molti tentativi e sotto consiglio di un docente mi sono reso conto che l’ampiamento del mio Search buffer e del LookAhead buffer mi avrebbe portato ad avere rapporti di compressione notevoli ma con il problema del rallentamento dell’esecuzione del codice.

Per avere un buon bilanciamento tempo-compressione ho dichiarato la lunghezza di search e LookAhead buffers di 32767 byte.

Ma visto che ho un problema con la scrittura bit a bit sono dovuto ricorrere a dei buffers di dimensione 65535.

Descrizione della struttura del codice sorgente

Il mio codice sorgente è strutturato in 4 funzioni di cui due sono di appoggio:

-la funzione bool isEmpty():

Questa funzione viene da me utilizzata per verificare se il file in input è un file vuoto.

Questo mi serve per evitare di comprimere un file vuoto.

-la funzione void bufferwork():

In questa funzione viene gestista la scrittura bufferizzata grazie ad un array d’appoggio chiamato app[] e di dimensione MAX che mi permette di salvare gli ultimi byte per poi inserirli in testa al buffer di scrittura in decompressione.

-la funzione void comprimi (unsigned char filedacomprimere[],unsigned char filecompresso[])

Questa funzione si occupa della compressione e gestisce la scrittura sul file compresso che è una terna (offset, lunghezza, prossimo carattere).

Inoltre abbiamo il buffer contenente il file da comprimere.

-la funzione void decomprimi(unsigned char filecompresso[],unsigned char fileoutput[])

Questa funzione si occupa della decompressione e gestisce la scrittura sul file decompresso che viene bufferizzata tramite la funzione bufferwork().

In questa funzione abbiamo un buffer di scrittura che è dichiarato dinamicamente di dimensione 100000 byte ed è del tipo unsigned char.

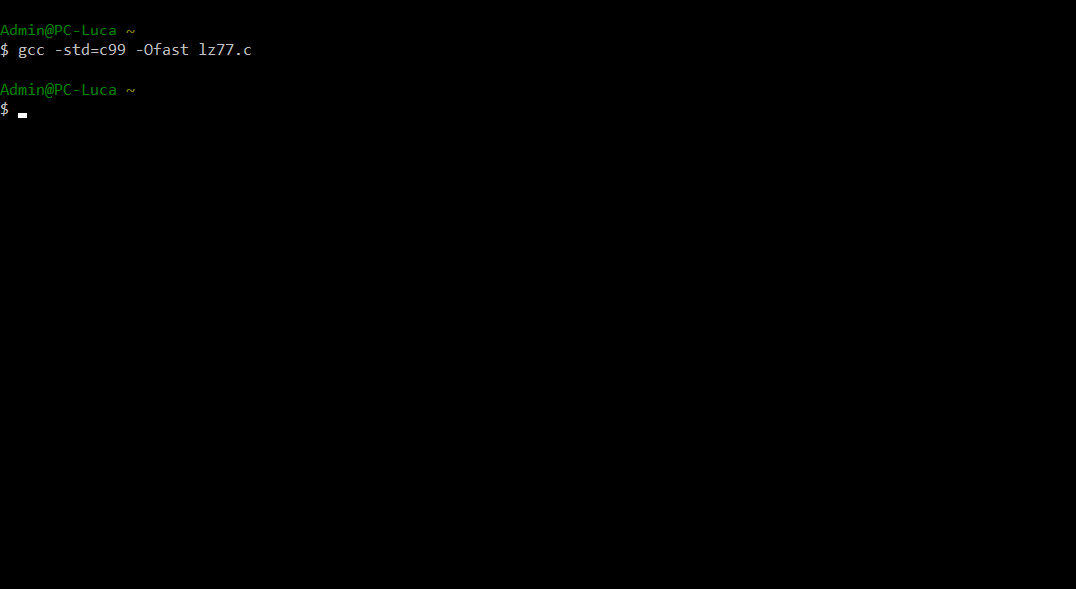
Utilizzo della libreria <time.h>

L’utilizzo di questa libreria mi è stato necessario a fini statistici e di confronto con l’algoritmo LZ78.

Questa libreria mi ha permesso di cronometrare il tempo di esecuzione del mio codice ed in progetto come il nostro è stato davvero utile per vede chi tra i due algoritmi andava meglio sulla base del tempo.

Compilazione

La compilazione del mio algoritmo viene fatta come in figura:



Con il seguente comando:

gcc -std=c99 -Ofast lz77.c

viene utilizzato il comando -std=c99 per essere compatibili con standard ISO/IEC 9899:1999.

Mentre viene utilizzato il comando -Ofast per migliorare le performance di velocità dell’algoritmo.

Lz77.c è il nome del file sorgente.

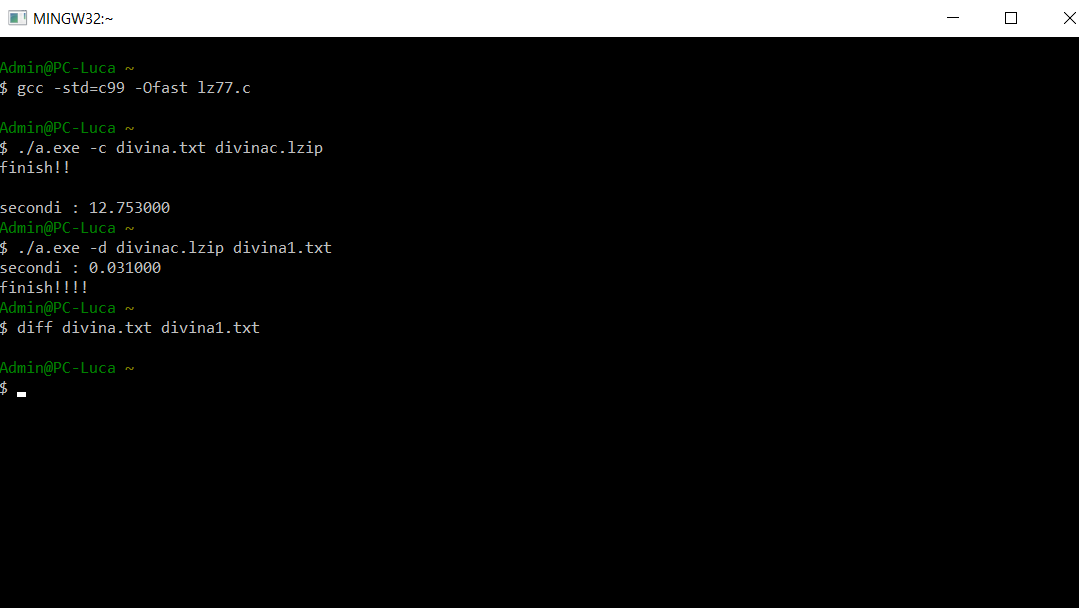
Passaggio dei parametri da riga di comando

In fase di esecuzione il mio algoritmo (attualmente in esecuzione su Windows), accetta tre parametri

-il primo che può essere o -c per la compressione o -d per la decompressione.

-il secondo sarà il file in input sia nella compressione che decompressione.

-il terzo sia in caso di compressione che decompressione sarà l’output dell’esecuzione del codice.



Lista dei problemi noti

Il mio algoritmo seppur funzionante non è ottimale poiché ha un oneroso consumo di risorse causato dalla lettura dell’intero file e dell’approccio brute-force nel trovare le occorrenze. Ho iniziato a ridurre il lavoro della CPU avendo search e lookAhead buffer di 15 bit ma questa implementazione non mi funziona a causa di un problema in decompressione (allego il codice del progetto effetuata con operatori bit a bit) in quanto non viene scritta sul file l’ultima terna(offset-lunghezza-prossimo carattere).La soluzione al utilizzo di molta memoria sarebbe stata quella di leggere il file a blocchi ma in questo ho avuto problemi che causavano l’inserimento di un carattere ogni volta che finivo un blocco di memoria anche di questo ne allego il codice commentato nel progetto funzionante.

# 3.LZ78

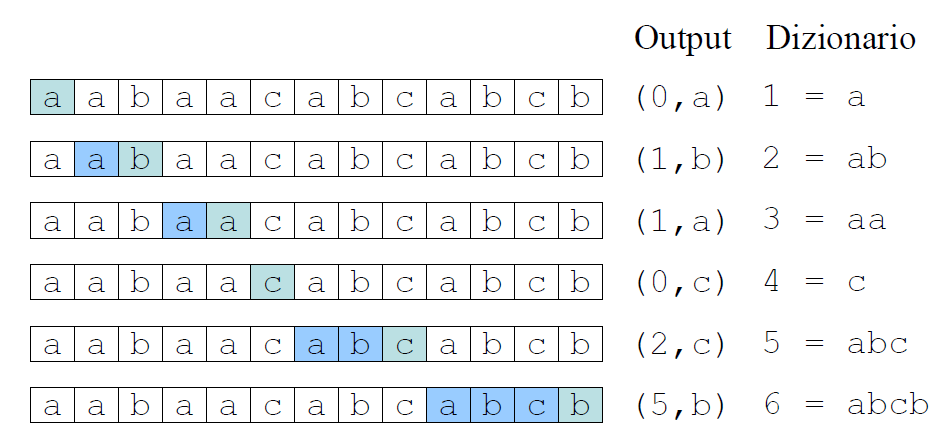
DESCRIZIONE ALGORITMO E FUNZIONAMENTO

L’algoritmo è stato spiegato e visto più volte in classe; rivedendo brevemente i concetti principali LZ7 si basa su:

-Un dizionario esplicito e quindi più sofisticato rispetto alla sliding window di LZ77

-Si trasmettono solo riferimenti al dizionario

-Il dizionario viene creato con identificatore intero+sottostringa individuata all’interno del dizionario+carattere successivo



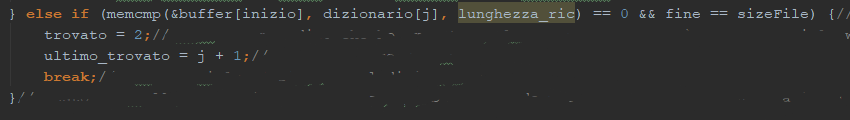
DESCRIZIONE STRUTTURA DEL CODICE SORGENTE

Il codice sorgente è interamente commentato e ben documentato, ogni riga di codice è molto intuitiva.

Inoltre il mio programma è molto semplice e lineare, ho utilizzato un approccio sequenziale e non ci dovrebbero essere incomprensioni.

Di seguito illustro alcuni pezzi di codice che ho utilizzato per risolvere piccoli problemi che ho riscontrato mentre sviluppavo il mio algoritmo:

1-L’algoritmo trovava una corrispondenza all’interno del dizionario e richiedeva il carattere successivo, ma ero alla fine del file quindi non “esisteva” alcun carattere successivo



Con questo piccolo controllo[trovato=2] riesco a capire che ho trovato una corrispondenza all’interno del dizionario e che sono anche alla fine del file.

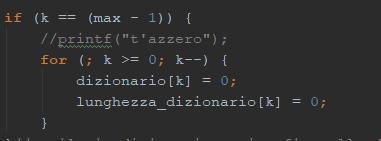


Cosi se non mi sono ritrovata in questa situazione scrivo il mio carattere sul file di output, altrimenti andrò a scrivere solo il codice; ovviamente questo fatto viene gestito in decompressione.

2-Il dizionario si riempiva e bisognava decidere come riutilizzarlo.

In principio avevo pensato di sfruttare l’indice del dizionario, e una volta pieno azzerare solo l’indice e quindi eseguire delle sovrascritture, eseguendo vari test è però emerso il problema del tempo, e cioè una volta pieno il dizionario (nell’algoritmo funzionante utilizzo un dizionario di dimensione 2^!6) il ciclo di ricerca era troppo oneroso e il rapporto di compressione non era per niente ottimale.

Così ho deciso di azzerare drasticamente il dizionario e quindi ricrearlo da zero ogni qualvolta si riempisse, nonostante i miei dubbi iniziali con diverse prove di test è risultata la scelta migliore da implementare.



DOCUMENTAZIONE STRUTTURE DATI UTILIZZATE

L’unica struttura dati interessante e molto semplice che ho utilizzato è quella per il dizionario, avrei preferito e sarebbe stato ottimale un TREE TRIE che avrebbe reso l’algoritmo più performante e molto più veloce grazie alle funzioni di ricerca che non avrebbero richiesto approcci sequenziali.





ISTRUZIONI PER LA COMPILAZIONE

Per la compilazione sul server linux1 eseguire la seguente linea di comando:

* gcc -Ofast -std=c99 lz78.c



Per eseguire la parte di compressione scrivere la seguente linea di comando:

* /a.out -c filedacomprimere filecompresso

Per eseguire la parte di decompressione scrivere la seguente linea di comando:

* /a.out -d filecompresso filedecompresso

Eseguendo il comando per trovare le differenze,dovrebbe dare “nessuna differenza riscontrata”

* Diff filedacomprimere filedecompresso

LISTA PROBLEMI NOTI

Purtroppo il mio algoritmo seppur funzionante non è ottimale; ho cercato di risolvere e provare soluzioni nuove per renderlo più efficace ma non sono stata in grado di implementare le mie idee e ho preferito lasciare il mio algoritmo con alcuni difetti piuttosto che averlo mal funzionante.

Il primo grande problema è la lettura del file; la prima cosa che faccio è leggere l’intero file e caricarlo all’interno di un buffer grande quindi quanto la dimensione del file, si intuisce subito che per file di grandi dimensioni ho un grandissimo utilizzo della memoria e non è detto che il mio algoritmo riesca a far partire l’esecuzione del codice.

Non sono riuscita a risolvere questo problema in quanto leggendo il file a blocchi inciampavo nel problema di aver trovato una corrispondenza ma di non aver alcun carattere dopo da leggere, ho pensato e provato diverse implementazioni che includevano lo shift dal buffer, oppure la lettura carattere per carattere ma risultava un’operazione troppo onerosa, la soluzione ideale sarebbe l’utilizzo di un circolar buffer che non ho concluso in tempo debito.

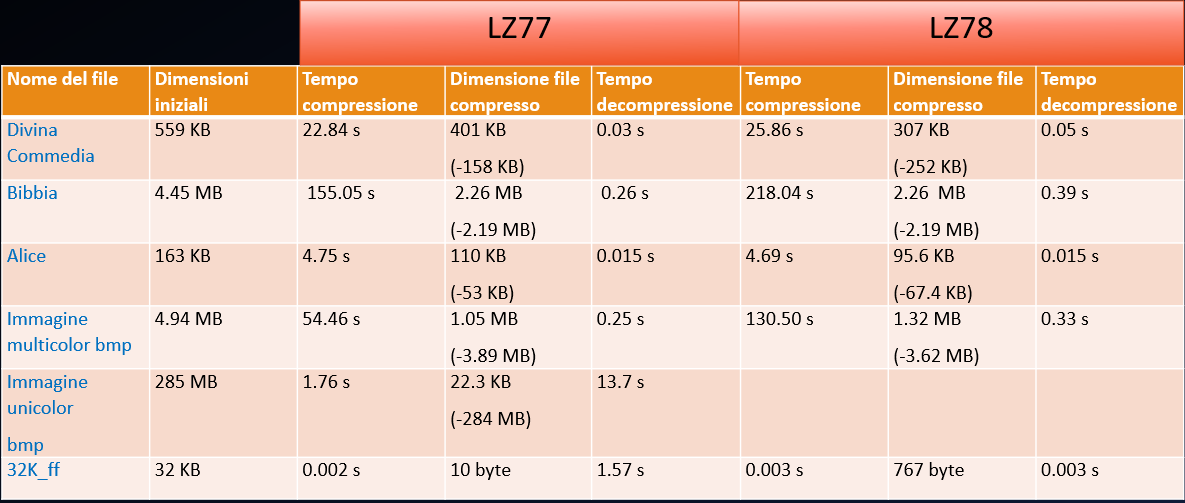
Un altro problema, che più che problema è un’ottimizzazione dell’algoritmo, è l’utilizzo dell’unsigned short che mi fa scrivere 16bit sul file per quanto riguarda il codice e mi fa utilizzare un dizionario di dimensioni 2^16.

La mia idea era quella di andare a risparmiare sui bit che andavo a scrivere per il codice diminuendo così anche la dimensione del dizionario e avendo delle tempistiche nettamente inferiori, rimanendo con un buon rapporto di compressione; il codice da me sviluppato funziona interamente sia in compressione che in decompressione perché con vari test ho visto che i codici che vengono scritti sul file compresso e che vengono poi rilevati dal decompressore sono giusti, il problema che non sono riuscita a risolvere è inerente alla scrittura sul file decompressore, cioè la ricostruzione del file originale.

Il codice di cui parlo è comunque presente come commento sotto il codice sorgente eseguibile.

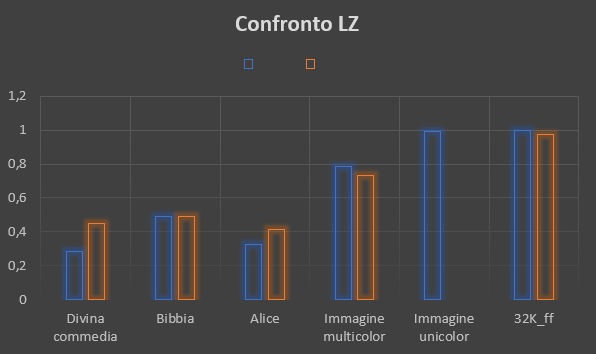
# 4.ALGORITMI A CONFRONTO

Di seguito riportiamo alcuni dati che abbiamo ricavato effettuando vari test su tipi di file diversi e con dimensioni differenti tra loro, interessante guardare come si comportano gli algoritmi da noi implementati



Come ci si poteva aspettare i due algoritmi viaggiano più o meno in simbiosi, a livello di tempo di compressione lZ77 ha molto distacco rispetto all’LZ78 che però si difende comprimendo file di grandi dimensioni a volte anche dimezzandoli.

Qui di seguito è riportato un grafico in cui si può vedere il variare e il confronto del rapporto di compressione tra i due LZ:



Riportiamo comunque la tabella e il grafico dei valori di come risulterebbero gli algoritmi con i loro rispettivi miglioramenti:

