Scuola universitaria professionale della Svizzera italiana
Dipartimento tecnologie innovative

SUPSI

CONFRONTOLZ*

LZ77 & LZ78

Autori: Ivan Pavic (LZ77), Elia Perrone (LZ78)

Data inizio progetto: ottobre 2017
Data fine progetto: 19.01.2018

Sommario

1. Introduzione

Di giorno in giorno la mole di dati scambiati cresce e i file sono elementi sempre più sofisticati e pesanti in termini di spazio di archiviazione.
Gli algoritmi di compressione dati hanno il compito di individuare ripetizioni all'interno di un qualsiasi file e di sostituire queste ripetizioni con codifiche rappresentabili con il numero di bit più piccolo possibile mantenendo però la possibilità di ricavarne lo stato originale.

L'obiettivo di questo progetto è quello di mettere a confronto due algoritmi di compressione della famiglia LZ.
I due algoritmi di compressione lossless che abbiamo scelto sono l'LZ77 e l'LZ78.
Questa scelta è stata fatta per analizzare le differenze e le prestazioni di due algoritmi che stanno alla base di molti software di compressione dati uttilizzati tutt'oggi, verificando in che modo si comportano la velocità di compressione e il fattore di compressione a seconda del tipo di dizionario che viene usato: esplicito nel caso dell'LZ78 e implicito in quello dell'LZ77.

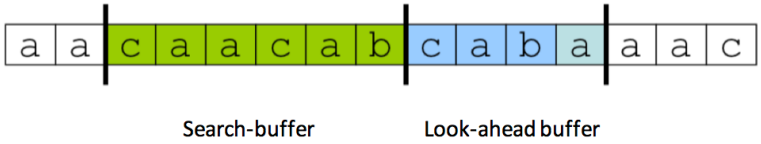
2. Algoritmi

a) LZ77 (Ivan Pavic)

1. Descrizione dell'algoritmo e funzionamento

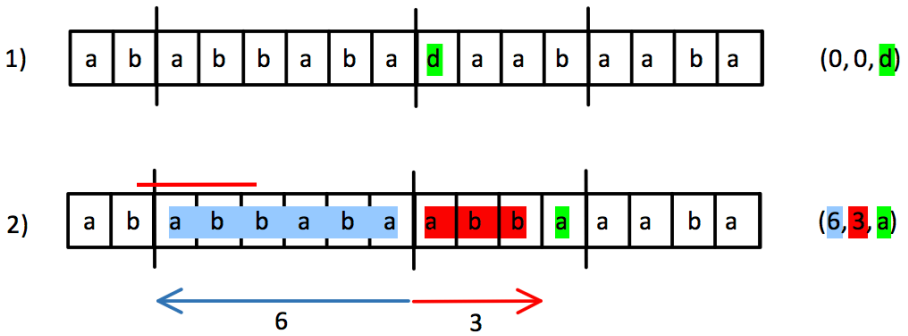
L'algoritmo LZ77 è un algoritmo di compressione a dizionario implicito che fa parte della famiglia dei compressori senza perdita di informazioni.
Questo algoritmo cerca sequenze che risultano ridondanti sostiduendole con delle codifiche. Per generare queste codifiche viene effettuata una ricerca all'interno dei dati che sono già stati codificati e le sequenze che sono già state incontrate vengono sostituite con dei codici che permettono al processo di decompressione di espanderle e risalire nuovamente alle parti che compongono il file originale.

La ricerca delle ridondanze avviene facendo scorrere una finestra composta da due elementi: un search buffer che contiene i dati che sono già stati codificati e un look-ahead buffer che contiene i dati che non sono ancora stati compressi.



La codifica che deriva da tale ricerca ha 3 elementi:
-**offset (o)**: rappresenta di quanto è necessario muoversi indietro nel search-buffer per identificare l'inizio dell'occorrenza più lunga.
-**length (l)**: rappresenta la lunghezza del prefisso più lungo trovato. Se il look-ahead ha grandezza n, il valore di questo parametro non può superare n-1 poiché la codifica include il byte che segue il prefisso.
-**next char (a)**: è il carattere che compare all'interno del look-ahead buffer subito dopo il prefisso.

Alcuni esempi di codifica:



Quando l'algoritmo non trova nessun prefisso nel searchbuffer, viene generato il codice con offset nullo, lunghezza nulla e primo carattere nel look-ahead buffer. (vedi esempio 1)
Altrimenti il codice generato indica dove è stato trovato il prefisso, quanto è lungo e quale carattere lo segue. (vedi esempio 2)

2. Strutture dati utilizzate

Poiché il principio di implementazione scelto è quello classico iterativo, la struttura dati scelta per la compressione e la decompressione è un array all'interno del quale vengono spostati dei puntatori per la ricerca delle occorenze.

Per tenere memorizzati i codici è stata usata una stuttura:

```
struct code
{
   int o;                  //offset
   int l;                  //length
   unsigned char a;        //next char
};
```

Per la compressione:

-Array:

```
unsigned char bytes_from_file[STREAM_SIZE];        //  bytes letti dal file di input
```

-Puntatori:

```
unsigned char *endOfBuffer;                        //puntatore a ultimo elemento di bytes_from_file
unsigned char *lookahead;                          //inizio look-ahead
unsigned char *window;                             //inizio search-buffer
unsigned char *l_cursor;                           //puntatore che si muove nel look-ahead buffer
unsigned char *w_cursor;                           //puntatore che si muove nel search-buffer
unsigned char *w_cursor_last;                      //inizio dell'ultimo prefisso trovato
```

Per la decompressione:

-Array:

```
unsigned char decompressed[STREAM_SIZE];           //bytes decompressi
struct code d[STRUCT_ARRAY_SIZE];                  //array di strutture che contenente le codifiche
```

-Puntatori:

```
unsigned char *last_element=&decompressed[STREAM_SIZE-1]; //fine bytes decompressi
unsigned char *d_lookahead = decompressed;                //inizio look-ahead buffer
unsigned char *d_window = d_lookahead;                    //inizio search-buffer
unsigned char *w_cursor;                                  //inizio prefisso
```

Per la scrittura/lettura bufferizzata:

```
int buffer[BUFFER_SIZE];                   //Buffered Writing/Reading Array
```

3. Implementazione

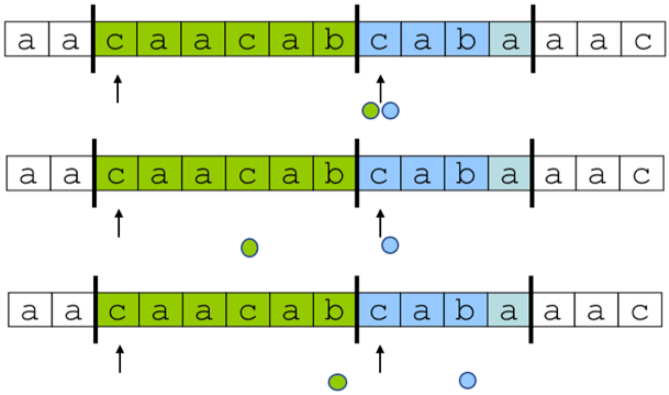
COMPRESSIONE

Ricerca delle occorrenze:

Per sviluppare questo algoritmo mi sono basato sulla ricerca iterativa delle occorrenze. L'obiettivo è stato quello di sviluppare un compressore funzionante con delle performance discrete e successivamente di migliorarne le prestazioni.

Per fare ciò ho deciso di implementare la ricerca delle occorrenze con il metodo classico iterativo usando come base di ricerca un semplice **array** all'interno del quale vengono caricati ciclicamente i byte letti dal file di input e sul quale vengono fatti scorrere dei puntatori.

Il programma si occupa di caricare ciclicamente a blocchi i byte letti dal file di input all'interno di un array di 1'000'000 di byte e di aggiornare la posizione dei puntatori all'interno di quest'ultimo.
Nelle immagini seguenti viene mostrato com'è strutturata la ricerca dei prefissi:



I **puntatori** usati sono sostanzialmente quattro:
- 2 puntatori di riferimento: il primo segna l'inizio del search-buffer mentre il secondo l'inizio del look-ahead.
- 2 puntatori di appoggio: questi due puntatori si muovono all'interno dei due buffer per effettuare la vera e propria ricerca dei prefissi.
Il puntatore verde si occupa della ricerca dell'inizio del prefisso.
Una volta trovato, i puntatori di appoggio verde e blu vengono spostati in parallelo fino a quando il loro valore risulta diverso o quando viene raggiunta la posizione lookahead-1.
La ricerca di prefissi continua fino a quando il puntatore verde ha raggiunto l'inizio del search-buffer.
L'occorrenza più lunga del search-buffer attuale viene tenuta salvata e aggiornata ogni qual volta si incontra una sequenza di lunghezza ulteriormente maggiore.

PROBLEMATICA:

La tripla che forma le codifiche risulta troppo onerosa in termini di spazio occupato poiché:

(offset, length, next-char) → (int, int, unsigned char) → (4 byte, 4byte, 1byte) = 9 byte

Nel caso peggiore, quando la codifica deve comprimere un unico byte, ne vengono occupati 8 aggiuntivi, se questa problematica viene estesa a tutto il file, il fattore di compressione diventa nullo.

A questo proposito ho sviluppato delle funzioni di scrittura bufferizzata.

Scrittura Bufferizzata:

Lo scopo della scrittura bufferizzata è quello di ridurre il numero di bit necessari per rappresentare le codifiche generate dall'algoritmo di ricerca che ho sviluppato.

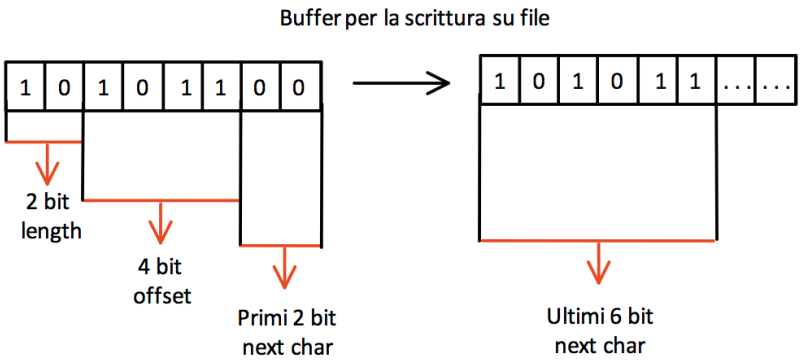
Supponendo che *m* sia la grandezza del search-buffer e *n* sia la grandezza del lookahead buffer, è noto che l'offset massimo che si può raggiungere è *m* mentre la lunghezza del prefisso più lungo arriva al massimo a *n-1*. Il numero minimo di bit per rappresentare un char è 8 per cui non è possibile risparmiare bit su questo elemento della codifica.
Con il variare della grandezza di search-buffer e lookahead buffer variano anche i bit necessari per rappresentarli.

Per calcolare quanti bit sono necessari per rappresentare offset e lunghezza è sufficiente fare:

$$\log_2(m)+1 \quad / \quad \log_2(n)$$

Con questo calcolo però perdo la possibilità di rappresentare l'offset di zero, cioè quando non esiste l'occorrenza.
È noto però che quando l'offset è nullo, anche la lunghezza del prefisso lo è, per cui per risparmiare ulteriori bit, la lunghezza viene bufferizzata per prima, permettendo così al decompressore di capire se ciò che segue i bit della lunghezza sono i bit dell'offset o direttamente quelli che rappresentano il next-char.

Grandezza finestra:	16 byte
Grandezza lookahead:	4 byte



Poiché la scrittura su file richiede al minimo 8 bit, i bit che rappresentano gli elementi di codifica vengono concatenati uno ad uno all'interno di un buffer di 8 celle che rappresenta 1 byte. Per fare ciò ho implementato questa semplice funzione che smista i bit:

```
void decToBin(int buffer[], int n, int b_size, FILE *outfile)
{
   for(int i=b_size-1; i>=0; i--)
   {
      if(n >=((int)pow(2,i)) )
      {
         fillBuffer(buffer, 1, outfile);
         n = n - ((int) pow(2, i));
      }else
         fillBuffer(buffer, 0, outfile);
   }
}
```

n = elemento di codifica da convertire in binario
b_size = numero di byte necessari per rappresentarlo

La funzione concatena gli elementi di codifica nel formato "big-endian".

Grazie a una variabile globale il programma tiene conto della posizione in cui è stato inserito l'ultimo bit e qualora venisse raggiunta la fine del buffer durante l'inserimento dei bit, il buffer viene convertito in decimale e scritto sul file compresso sotto forma di unsigned char; la variabile globale viene azzerata e l'inserimento può continuare. Applicando questo processo a tutte le codifiche la compressione risulta molto più efficiente.

```
void fillBuffer(int buffer[], int bit, FILE *outfile){
   if(bits<BUFFER_SIZE){
      buffer[bits| = bit;
      bits++;                                    //contatore di posizione
   }else if(bits==BUFFER_SIZE){
      decimal = binToDec(buffer);   //conversione binario-decimale
      fputc(decimal, outfile);                   //scrittura codifica sul file
      inizializeArray(buffer, BUFFER_SIZE);
      bits=0;                                    //azzeramento contatore
      buffer[bits]=bit;
      bits++;
   }
}
```

DECOMPRESSIONE:

Lettura bufferizzata:

Il processo per reperire le codifiche dai codici generati con la scrittura bufferizzata è esattamente l'inverso.
I codici vengono estratti dal file compresso uno ad uno in base alla necessità.
Quando un codice viene estratto dal file, le funzioni decToBin e fillBuffer, già implementate nella scrittura bufferizzata, lo convertono in binario riempiendo il buffer di bit.

Il programma all'inizio della decompressione sa già di dover estrarre per primi i bit della lunghezza. Una volta estratti, vengono convertiti in decimale e viene effettuata la seguente verifica: se la lunghezza risulta nulla, allora richiedo i bit del nextchar altrimenti richiedo prima

quelli dell'offset a cui sommo 1 (per la ragione vista nella scrittura bufferizzata) e poi quelli del next-char.
Ogni volta che i bit del buffer vengono esauriti, l'algoritmo richiede un nuovo codice e il ciclo continua fino all'esaurimento dei codici.

```
int extractCodes(int buffer[]){
   n_bits--;                          //numero di bits da leggere, variabile globale
   int decimal = 0, c=0;
   while(n_bits>=0){
      if(buffer_position==0) {
         buffer_position=8;
         c=getNextCode(&infile);
         //se i codici sono finiti ritorna EOF
         if(c!=EOF) {
            decToBin(buffer, c, BUFFER_SIZE);
         }else {
            return EOF;
         }
      }
      //conversione binario/decimale
      decimal = decimal + ( (buffer[8-buffer_position]) * ((int) pow(2,n_bits)));
      n_bits--;
      buffer_position--;              //contatore bit estratti dal buffer
   }
   return decimal;
}
```

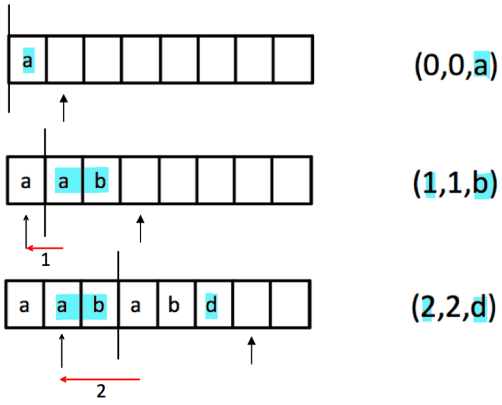
Tutte le codifiche ricavate da queste conversioni vengono salvate in un array di strutture che rappresentano la tripla.

Decompressione:

Quando si riempie completamente, l'array di strutture viene passato alla vera e propria funzione di decompressione che prende le codifiche e come il compressore sfrutta dei puntatori per riempire un secondo array con i byte decompressi.
I puntatori sono grossomodo 2:

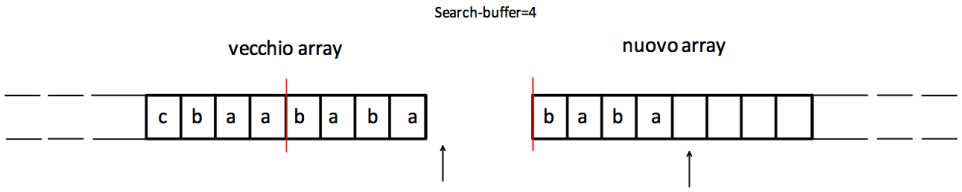
-**1 puntatore di riferimento** che tiene la posizione sull'ultimo byte decompresso
-**1 puntatore di appoggio** che si muove indietro nei file decompressi per trovare l'inizio del prefisso da espandere.

Una volta trovato l'inizio del prefisso, i 2 puntatori si muovono in parallelo e viene effettuata una copia di x byte, elemento per elemento, dove x è la lunghezza del prefisso.



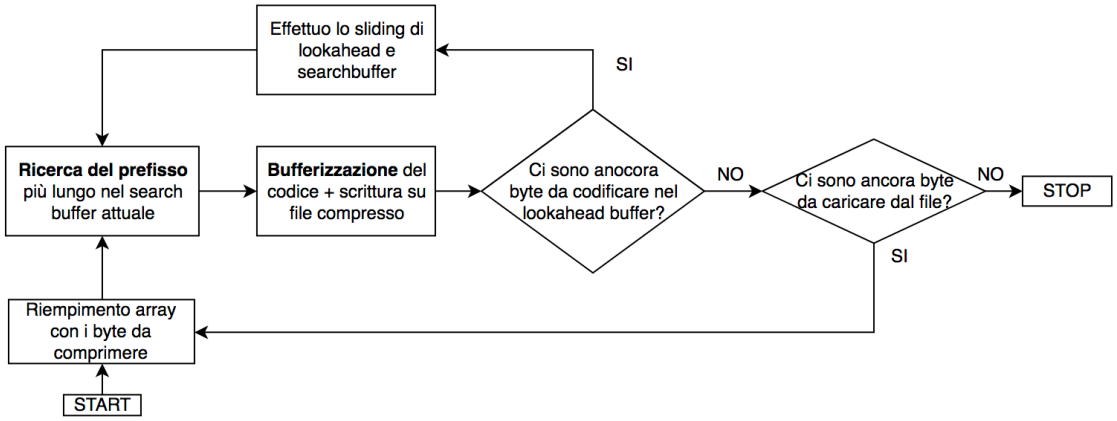
Quando l'array dei byte decompressi viene riempito, viene scritto sul file decompresso con la funzione di libreria fread e il processo continua fino a quando la lettura bufferizzata fornisce i codici.

Per avere di nuovo a disposizione tutto il search-buffer anche dopo la scrittura su file, è necessario copiare gli n byte finali all'inizio del nuovo array e riposizionare il puntatore di riferimento alla posizione n-1. (n=grandezza search-buffer)

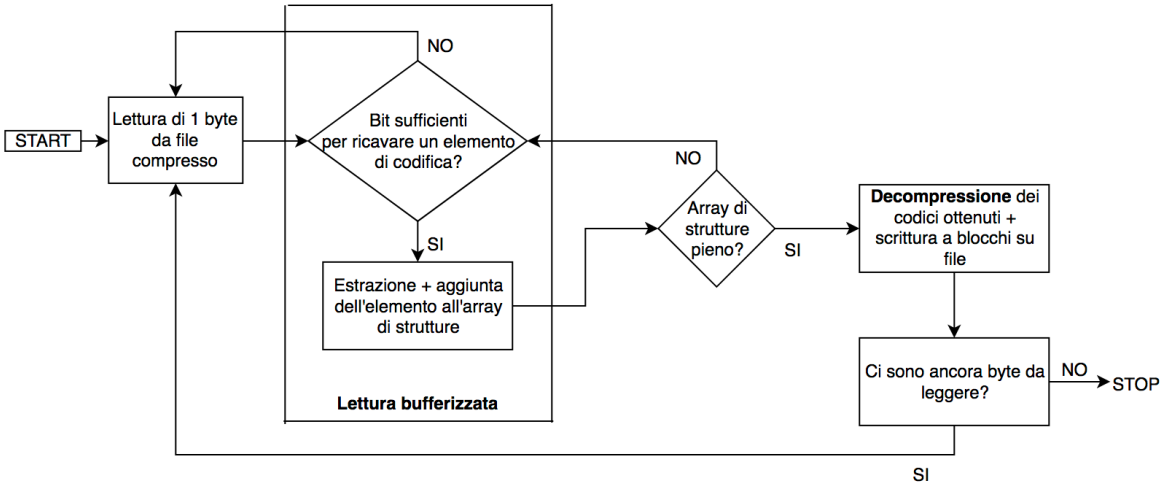


4. Struttura del codice sorgente

Compressione:



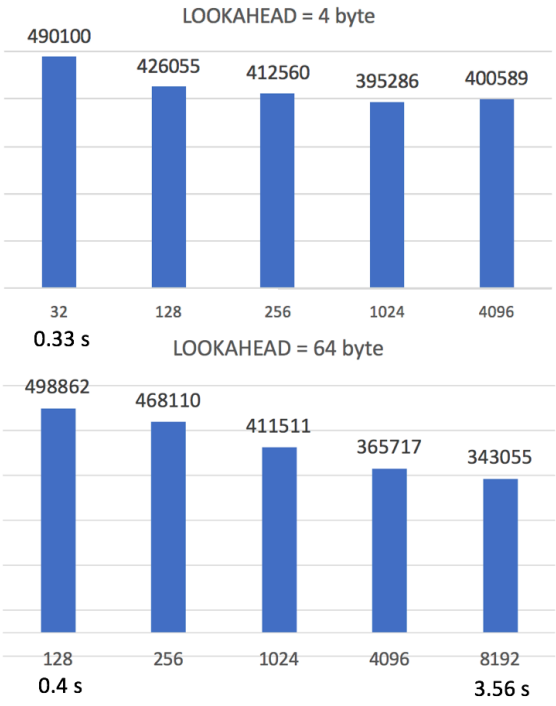
Decompressione:

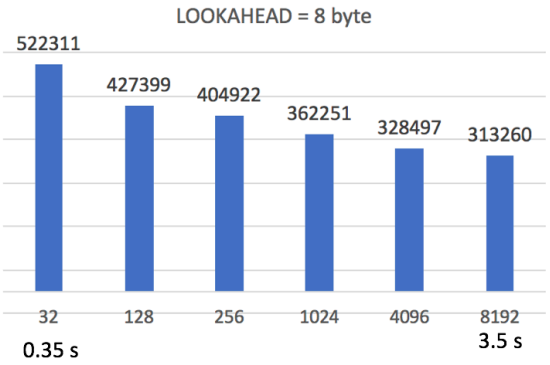


5. Risultati ottenuti

L'algoritmo è stato testato sottoponendo inizialmente delle semplici stringhe mantenendo lookahead buffer di grandezza 4 e searchbuffer di grandezza 8, per verificare se i codici generati sono corretti.
Una volta ottenuti i codici giusti ho sottoposto il programma a diversi test modificando le grandezze dei buffer e degli array di appoggio usati per la compressione e la decompressione.

Questi sono i risultati ottenuti con la Divina Commedia (551'846 byte):





Come si può notare il fattore di compressione migliore è stato ottenuto con un search buffer di 8kbyte e un lookahead buffer di 8byte.
Con le stesse grandezze ho testato i seguenti file:

	TEMPO (s)	PESO
32k_ff (32768 byte)	1	12 kbyte
32k_random (32768 byte)	0.4	47 kbyte
empty (0 byte)	0	0 byte
ff_ff_ff (3 byte)	0.00317	5 byte
alice.txt (167518 byte)	1.07	93 kbyte
immagine.bmp (1.6MB)	36	1.2MB

32k_random contiene byte casuali il che rende difficile trovare prefissi molto lunghi e per via del searchbuffer di 8kbyte le codifiche generate per prefissi di offset 0 risultano pesanti. Il risultato è un file che non è stato compresso.
Allo stesso modo il file ff_ff_ff è un file di soli 3 byte, con un searchbuffer così grande è impossibile comprimerlo.

6. Problemi riscontrati e future migliorie

Il problema più grande è stato senz'altro la scelta della struttura dati da utilizzare. Non avendo mai utilizzato alberi di ricerca o liste concatenate, ho optato per lo sviluppo iterativo passando da un semplice array.

Sviluppare l'algoritmo di ricerca dei prefissi mi ha preso moltissimo tempo poiché i casi da considerare erano molteplici e ogni volta che mi sembrava di averli considerati tutti, venivano a galla nuove problematiche.

Una volta risolta la ricerca dei prefissi, mi sono accorto che il fattore di compressione non era per niente ottimale per cui ho dovuto implementare la scrittura bufferizzata.
Avendo implementato correttamente il processo di bufferizzazione, lo sviluppo del decompressore è risultato molto più rapido del previsto.

BUG 1:

Un problema che ho riscontrato e che è **tutt'ora irrisolto** riguarda il decompressore.

```
#define STRUCT_ARRAY_SIZE 600000
#define STREAM_SIZE 40000000
```

Queste due definizioni permettono di impostare la grandezza dell'array di strutture e dell'array che contiene i caratteri decodificati da scrivere su file.
In questo caso, quando la compressione genera meno di 600'000 codici il decompressore è funzionante.
Invece quando viene superata questa soglia, dal momento che l'array di strutture viene sovrascritto e caricato con nuovi codici, l'algoritmo estrae alcuni bit in più dal file compresso e genera codifiche sfasate.
Di conseguenza la funzione di decompressione, che opera sui byte già decompressi, entra in ridondanza e non decomprime più nel modo corretto.
Il risultato di questo errore sono dei file decompressi a pezzi.

Soluzione possibile:

Avendo sviluppato la decompressione estraendo solo un byte alla volta dal file compresso, il programma risulta meno stabile aumentando la possibilità di bug.
Una modifica che si potrebbe apportare è leggere il file compresso a blocchi e gestendo più facilmente il concatenamento tra un blocco di codici e l'altro.

BUG 2:

In certi file alla fine della decompressione, viene inserito un byte di troppo oppure un byte di meno.
L'errore deriva probabilmente dal fatto che come nel BUG1 la lettura bufferizzata è gestita male e viene letto un codice in più/meno alla fine della decompressione.
La soluzione che propongo è la medesima.

MIGLIORIE:

Come spiegato nel capitolo dei risultati ottenuti, dopo aver testato il compressore impostando diverse grandezze per il search-buffer e il look-ahead buffer ho riscontrato che con l'aumentare della grandezza del search-buffer l'algoritmo comprime meglio, ma è penalizzato sulla velocità di compressione siccome la ricerca dei prefissi diventa molto lenta.

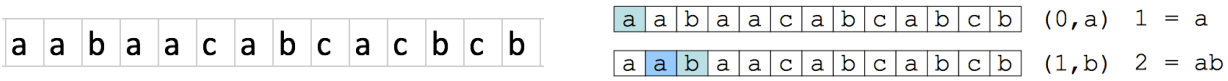
La chiave per migliorare questo aspetto è la scelta di una struttura dati che sia in grado di effettuare la ricerca con una complessità e un numero di iterazioni minore.

b) LZ78 (Elia Perrone)

1. Descrizione dell'algoritmo e funzionamento

A differenza dell'algoritmo LZ77, che lavora su dati precedentemente analizzati, l'algoritmo LZ78 lavora sulla gestione dei dati futuri, ovvero sui dati che ancora devono passare in analisi. LZ78 si basa su uno schema ben definito implementando l'utilizzo di un dizionario, che è uno strumento utilizzato dall'algoritmo per salvare delle occorrenze già analizzate.
Questo algoritmo esegue l'aggiunta di valori in un dizionario per poi rilevarne nuovamente un'occorrenza già riscontrata, ovvero con lo stesso valore.
Quando viene verificata un'occorrenza medesima ad una già precedentemente salvata nel dizionario, l'algoritmo emette l'indice dell'elemento all'interno del dizionario invece del valore.
L'LZ78 genera, nel file di compressione, codifiche di due elementi: un indice, che rappresenta l'indice associato al valore di un elemento nel dizionario, e il seguente simbolo che interrompe la corrispondenza con il valore.

Esempio di codifica applicato su una stringa:



Notiamo che il dizionario viene aggiornato costantemente durante lo scorrimento della stringa.
Sul file compresso verranno inserite le codifiche generate dall'algoritmo.

2. Strutture dati utilizzate

All'inizio del progetto è stato deciso di approcciarsi ad un'implementazione lineare dell'algoritmo tramite un sistema iterativo, ovvero con l'utilizzo di strutture basi, con tipi di dato semplici.
Per questo motivo sono state progettate le seguenti due strutture:

Codifica	Elemento dizionario

`typedef struct _output{`
`  unsigned int index;`
`  unsigned char next_value;`
`}Output;`
	`typedef struct _element{`
	`  unsigned int index;`
	`  unsigned char value[VALUE_SIZE];`
	`}Element;`

Questa struttura rappresenta i codici che vengono generati dalla compressione. Il codice è composto da due elementi: un indice, che rappresenta l'indice dell'elemento all'interno del dizionario, e un carattere successivo, che rappresenta il carattere che interrompe la corrispondenza con un valore all'interno del dizionario. Questa struttura è utilizzata semplicemente per avere una struttura ed una lettura del codice più semplificata ed ordinata.	Questa struttura rappresenta un elemento all'interno del dizionario. Questi elementi vengono man mano creati durante la compressione e la decompressione. L'elemento del dizionario è composto anch'esso da due componenti: un indice, che rappresenta l'indice univoco dell'elemento, e un valore, che può essere composto da più caratteri.

Variabili utilizzate per la compressione:

FILE:
FILE *input_file: è il file di input che deve venire compresso.
FILE *output_file: è il file compresso (dove sono inserite le codifiche).

BUFFER:
unsigned char buffer[BUFFER_SIZE]: è il buffer che viene utilizzato per il popolamento con il file da comprimere.
unsigned char subbuffer[BUFFER_SIZE]: è il buffer in cui vengono inserite delle sottostringhe estrapolate da buffer.
unsigned int bitbuffer[BITBUFFER_SIZE]: è il buffer utilizzato per ridurre la grandezza degli indici degli elementi all'interno del dizionario (miglioria per la compressione).

VARIABILI:
Output *output = malloc(sizeof(Output)): è l'oggetto output in quale verranno inserite le codifiche durante la compressione, e che poi verranno scritte su file.
Element dictionary[DICTIONARY_SIZE]: è il dizionario in cui verranno aggiunti gli elementi.

Scrittura bufferizzata:
Analizzando la grandezza della nostra struttura Output notiamo che essa è costituita da due elementi: un indice, che è un integer, che nel linguaggio C viene allocato con una grandezza di 4 byte, e un carattere, che viene allocato con una grandezza di 1 byte.
In totale, per ogni codifica, avremmo un oggetto che pesa 5 byte e dunque esageratamente grande in confronto alla memoria che bisogna allocare per una codifica.
Questo perché sappiamo che, la grandezza dell'indice degli elementi all'interno del dizionario varia in base alla grandezza del file da comprimere; infatti più sarà grande il file da comprimere più sarà grande lo spazio di allocazione di memoria per la rappresentazione di una codifica.
In conclusione possiamo constatare che la grandezza di una codifica è strettamente dipendente dalla grandezza dell'indice che prende l'elemento del dizionario.

La scrittura bufferizzata si occupa proprio di risolvere questo problema.
L'idea era di implementare un sistema logico che mi gestisse la grandezza dell'indice degli elementi del dizionario in base al valore associato. Questo è comodo perché conosciamo a priori la grandezza degli indici dato che sono dei numeri naturali e sono aggiunti in sequenza.

Es. Data una codifica e la quantità di codifiche già analizzate, possiamo estrapolare la grandezza, in bit, che servirà per rappresentare un nuovo elemento del dizionario.

Es. Se la codifica è (n,c) sappiamo che per rappresentare l'indice n bastano log2(n) bit.

Variabili utilizzate per la decompressione:

FILE:
FILE *output_file: è il file di input che deve venire decompresso.
FILE *output2_file: è il file decompresso (uguale al file che si voleva comprimere inizialmente).

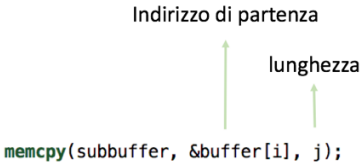
BUFFER:
unsigned char debuffer[BUFFER_SIZE]: è il buffer che viene utilizzato per il popolamento con il file da decomprimere.
unsigned char next_char_value[VALUE_SIZE]: è il buffer utilizzato per l'inserimento del carattere successivo.
unsigned char value[VALUE_SIZE]: è il buffer utilizzato per l'inserimento del valore dell'elemento del dizionario.

VARIABILI:
unsigned int index_value: è utilizzato per l'inserimento dell'indice della codifica.
unsigned int k: utilizzato per il popolamento di debuffer.

3. Implementazione

Funzioni rilevanti compressione:

memcpy(subbuffer, &buffer[i], j);
È una funzione della libreria standard string.h. Tramite questa funzione posso, dato l'indirizzo di partenza di un array e una lunghezza, estrarre una sottostringa dal buffer principale.
Utilizzo memcpy per estrapolare parti di file input e ricercarne i valori all'interno del dizionario, questo per crearne le codifiche.



search_element_by_value(dictionary, subbuffer, DICTIONARY_SIZE, VALUE_SIZE);
È una funzione realizzata appositamente per la gestione del dizionario. Dato un valore da ricercare questa funzione verifica, di tutti gli elementi presenti nel dizionario, se esiste una corrispondenza.
Se esiste un elemento con il medesimo valore, allora la funzione ritorna il valore dell'indice dell'elemento all'interno del dizionario che ha lo stesso valore del valore richiesto, mentre, se non viene trovato nessun elemento corrispondente la funzione ritorna 0.

```
unsigned int search_element_by_value(Element d[], unsigned char value[], unsigned int dictionary_size, unsigned int value_size){
    unsigned int index=0,count=0;
    for (int i = 0; i < dictionary_size; i++) {
        for (int j = 0; j < value_size; j++) {
            if(d[i].value[j]==value[j]) {
                count++;
            }
            else{
                count=0;
                break;
            }
        }
        if(count==value_size){
            index = d[i].index;
        }
    }
    return index;
}
```

Se ritorna 0 → non è stato trovato l'elemento nel dizionario
Se ritorna un valore → il valore è l'indice dell'elemento ricercato

add_element(dictionary, global_index, DICTIONARY_SIZE, VALUE_SIZE);
È una funzione realizzata appositamente per la gestione del dizionario. Dato un valore da aggiungere al dizionario la funzione non esegue altro che la copia del valore all'interno di un nuovo elemento del dizionario. Gli elementi sono aggiunti in fila (in coda) e per ogni nuovo elemento viene assegnato un indice globale.

```
void add_element(Element d[], unsigned int global_index, unsigned char value[], unsigned int value_size){
    d[global_index].index = global_index+1;
    for (int i = 0; i < value_size; i++) {
        d[global_index].value[i] = value[i];
    }
}
```

Funzioni rilevanti decompressione:

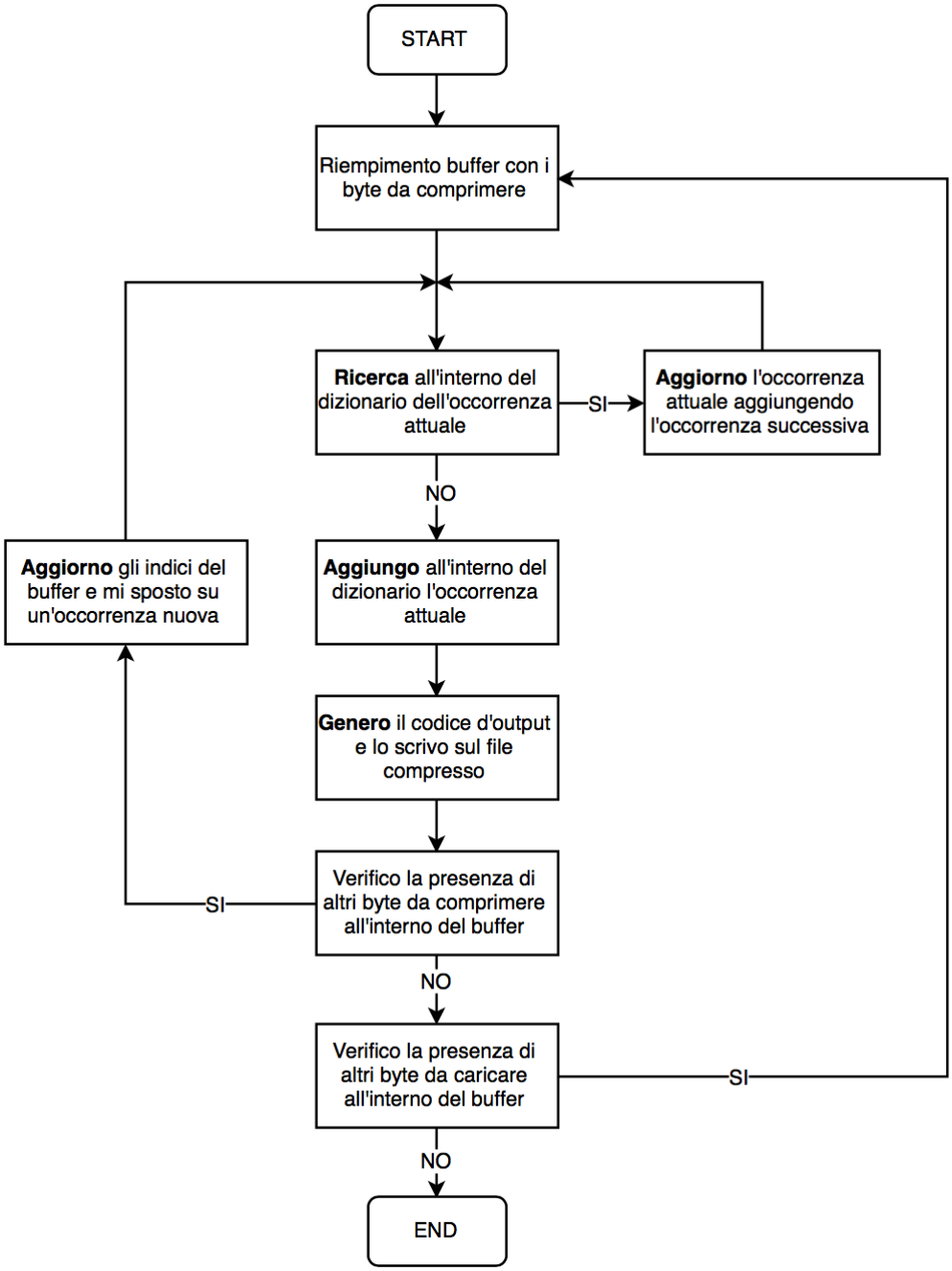
search_element_by_index(dictionary, index_value-'0', value, DICIONTARY_SIZE, VALUE_SIZE);

È una funzione realizzata appositamente per la gestione del dizionario. Dato un indice da ricercare questa funzione paragona, per tutti gli elementi presenti nel dizionario, se esiste una corrispondenza di indici. Se viene trovato l'indice richiesto allora la funzione associato all'array value il valore dell'elemento trovato.

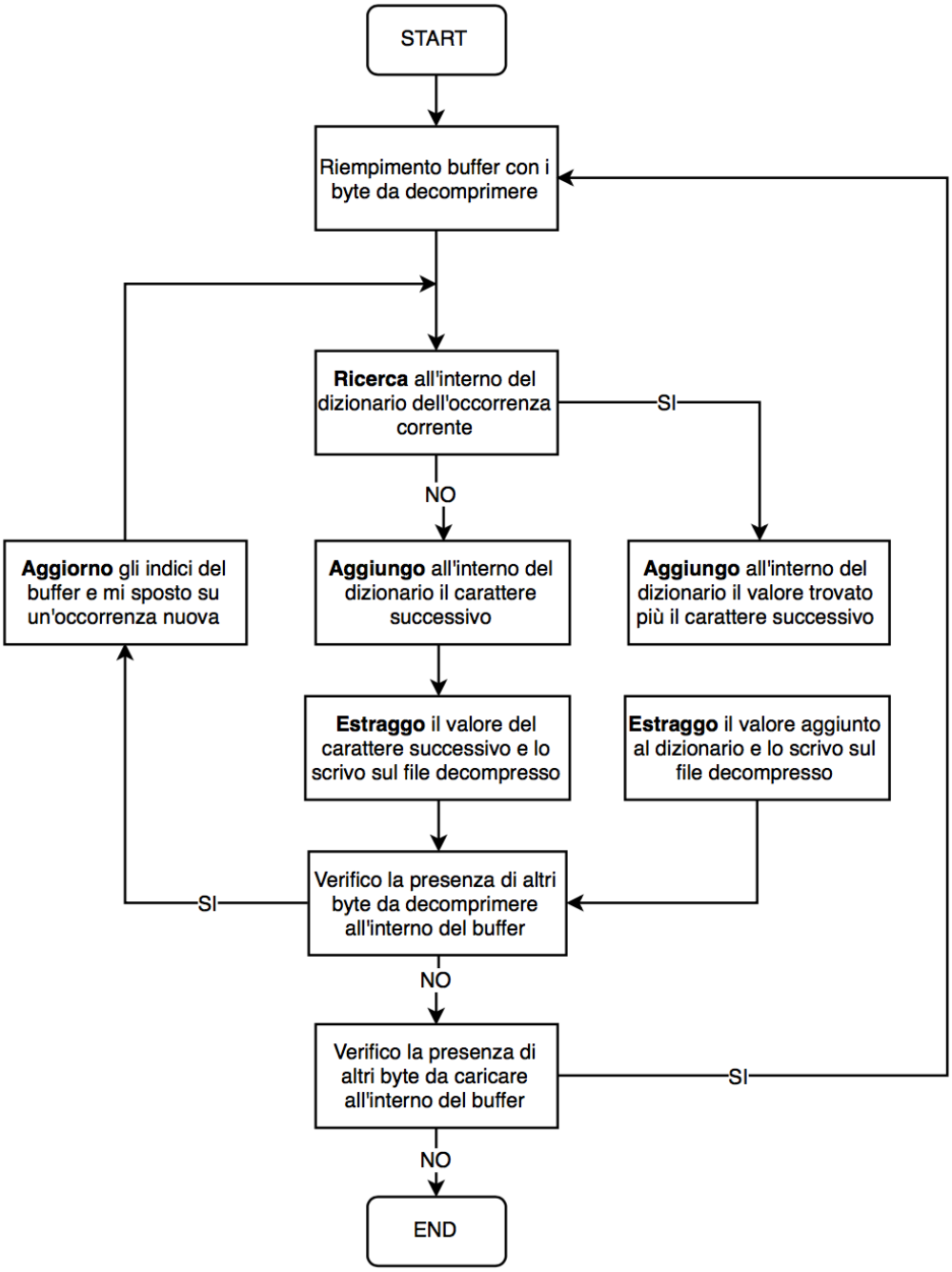
```
void search_element_by_index(Element d[], unsigned int index, unsigned char value[], unsigned int dictionary_size, unsigned int value_size){
    for (int i = 0; i < dictionary_size; i++) {
        if(d[i].index==index) {
            for (int j = 0; j < value_size; j++) {
                value[j]=d[i].value[j];
            }
        }
    }
}
```

4. Struttura del codice sorgente

Compressione:



Decompressione:



5. Risultati ottenuti

Dato l'utilizzo di funzioni appartenenti unicamente al mondo delle stringhe (string.h) l'algoritmo funziona unicamente con file di testo, che sono i file con cui ho testato il funzionamento.

FILE	SIZE	FILE COMPRESSO	TEMPO DI COMPRESSIONE
Prova1.txt	20 kbyte	16 kbyte	1.2 s
Prova2.txt	40 kbyte	32 kbyte	2.4 s
Prova3.txt	60 kbyte	34 kbyte	3.5 s
Divina_Commedia.txt	550 kbyte	330 kbyte	56 s
Promessi_Sposi.txt	1800 kbyte	1500 kbyte	106 s

Notiamo che i tempi di compressione sono estremamente alti. Questo è dato dall'utilizzo di strutture dati non efficienti (array e array di strutture) che all'interno delle funzioni di ricerca e di aggiunta nel dizionario impiegano un lasso di tempo enorme per completare le operazioni; infatti sia la funzione di ricerca che di aggiunta utilizzano due cicli for annidati che mi permettono di scorrere tutto l'array di strutture (dizionario). Questo metodo ovviamente non è efficiente.

Dati i problemi riscontrati non è stato ancora implementato un utilizzo del programma da terminale. Per eseguire dei test basta semplicemente inserire il percorso del file che si vuole comprimere all'interno del programma quando si esegue la funzione fopen sui file.

6. Problemi riscontrati e future migliorie

Strutture dati inefficienti:
La prima grande problematica è stata, una volta implementata la compressione, la gestione delle grandezze della struttura Element.
Questa struttura infatti nella compressione è stata utilizzata per la realizzazione di un dizionario, che non è altro che un array di elementi.
Per la gestione del dizionario, in compressione, vengono utilizzate principalmente due funzioni, add_element() e search_element_by_value(). Queste due funzioni scorrono, tramite due cicli for annidati, tutti i valori degli elementi nel dizionario. Questo limita molto la compressione rendendola estremamente lenta.

Una miglioria sulle strutture dati sarebbe utilizzare un tipo di struttura ad albero, perché permettono una gestione del dizionario più veloce e flessibile e dunque delle prestazioni migliori sia in compressione che in decompressione.

Scrittura e lettura bufferizzata:
Dato il problema delle grandezze degli indici degli elementi del dizionario (vedi capitolo scrittura bufferizzata) all'interno del programma manca l'implementazione di scrittura e lettura bufferizzata, che sarebbe utilizzate unicamente per migliorare la compressione.

3. Confronto LZ*

È necessario sottolineare che il confronto che abbiamo fatto è basato solamente sotto il punto di vista teorico poiché l'algoritmo LZ78 è risultato più lento del previsto e questo ha reso difficili dei confronti realistici.
Anche per quanto riguarda la decompressione, non è stato possibile effettuare un confronto pratico visto che entrambi gli algoritmi presentano ancora qualche problema di decompressione: in particolare l'LZ77 non è ancora in grado di decomprimere nel modo corretto file di grosse dimensione mentre l'LZ78 presenta dei problemi nella lettura delle codifiche dal file compresso.

Confronto teorico:
Come sottolineato nei capitoli precedenti l'algoritmo LZ77 e l'algoritmo LZ78 si differenziano su un fattore principale: l'LZ77 lavora basandosi su un dizionario implicito contente i byte già compressi che viene aggiornato dopo ogni ricerca del prefisso più lungo, l'LZ78 invece costruisce un dizionario esplicito operando sui dati che devono ancora essere compressi.
Se prendiamo per esempio una stringa (di 14 byte)

a	a	b	a	a	c	a	b	c	a	c	b	c	b

e generiamo le codifiche con entrambi gli algoritmi (LZ77: search buffer = 6byte, lookahead = 4 byte).

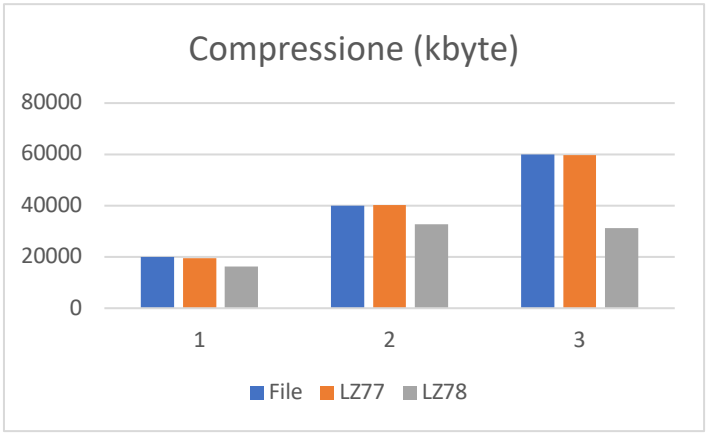
LZ78	
Codice	**Dizionario**
(0,a)	1=a
(1,b)	2=ab
(1,a)	3=aa
(0,c)	4=c
(2,c)	5=abc
(1,c)	6=ac
(0,b)	7=b
(4,b)	8=cb

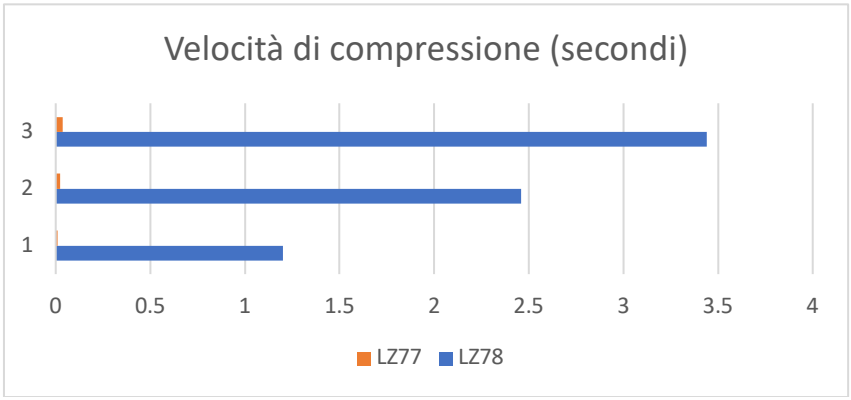
LZ77
Codice
(0,0,a)
(1,1,b)
(3,2,c)
(5,2,c)
(5,2,b)
(4,1,b)

Notiamo che tramite l'algoritmo LZ78 vengono generate 8 codifiche mentre con LZ77 6.
Senza scrittura bufferizzata le codifiche avrebbero un peso di:

- LZ77: 8*4byte (indice) + 8*1byte (char) = 40byte
- LZ78: 12*4byte (offset + length) + 6*1byte (char) = 54byte

Applicando la scrittura bufferizzata, per entrambi gli algoritmi, constatiamo che le compressioni totali hanno il peso rispettivamente di 10byte e 9 byte sui 14byte di partenza.
Dal punto di vista teorico, la scrittura bufferizzata, se implementata correttamente, a lungo termine ha un impatto molto più forte sulle codifiche dell'LZ77 poiché la grandezza del lookahead buffer e del searchbuffer rimangono invariate durante tutta la compressione.
Per LZ78 invece questo fattore cambia poiché se aumenta il numero di indici del dizionario, aumentano anche i bit necessari per rappresentarli.
Chiaramente tutte queste osservazioni sono da prendere con le pinze poiché l'implementazione dinamica dell'LZ77 permette di modificare la grandezza di offset e length permettendo di trovare prefissi più lunghi rappresentabili con pochi bit nel caso migliore, rischiando però di diminuire il fattore di compressione in caso di file con prefissi molto corti. È importante sottolineare che l'algoritmo LZ77 ci da la possibilità di modificare la grandezza dei due buffer permettendo un adattamento molto più dinamico rispetto all'LZ78. Anche questo fattore rende difficile la comparazione tra i due algoritmi perché possono essere considerati molti parametri.





I file presi in esame sono tre differenti file di testo con peso rispettivamente di: 20, 40, 60 Kbyte. Notiamo che l'algoritmo LZ78 impiega molto più tempo rispetto al LZ77. Questo è spiegato appunto per l'utilizzo di strutture dati poco efficienti(vedi capitolo procedure di test per LZ78).

Teoricamente LZ78 ha una ricerca molto più rapida se implementata per esempio tramite alberi. La ricerca tramite albero permette di saltare numerosi prefissi che non combaciano con i byte da comprimere. L'algoritmo di ricerca che è stato implementato per l'LZ78 invece scorre tutti gli elementi di un array controllando ogni singolo byte e questo in termini di performance risulta molto oneroso.

4. Istruzioni per l'utilizzo

LZ77:

Compilare il file main.c su terminale con il comando:

gcc main.c -lm -o main

Per usare il **compressore** usare il comando:

./main -c inputfile outputfile

Per usare il **decompressore** usare il comando:

./main -d inputfile outputfile

Per modificare grandezza di searchbuffer e look-ahead buffer aprire il file main.c e modificare le seguenti definizioni:

```
#define LOOKAHEAD 8
#define WINDOW 8192
```