SUPSI

LZ*-Comparison
LZ77 & LZ78

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

Start date: october 2017
End date: 19.01.2018

Sommario

1. Introduction

Day by day the amount of data exchanged is growing and files are increasingly sophisticated and heavy in terms of storage space.
The data compression algorithms have the task of identifying repetitions within any file and replacing these repetitions with codings that can be represented with the smallest possible number of bits while maintaining the possibility of obtaining the original state.

The aim of this project is to compare two compression algorithms of the LZ family.
The two lossless compression algorithms we have chosen are LZ77 and LZ78.
This choice was made to analyze the differences and performance of two algorithms that are the basis of many data compression software used today, checking how they behave, what is the compression speed and what is the compression factor depending on the type of dictionary that is used: explicit in the case of LZ78 and implicit in that of LZ77.

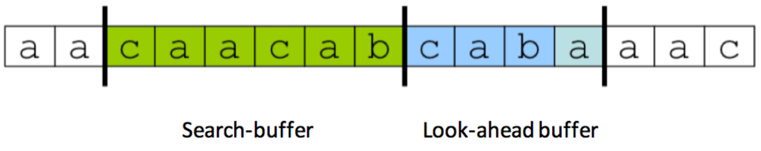
2. Algorithms

a) LZ77 (Ivan Pavic)

1. Description of the algorithm

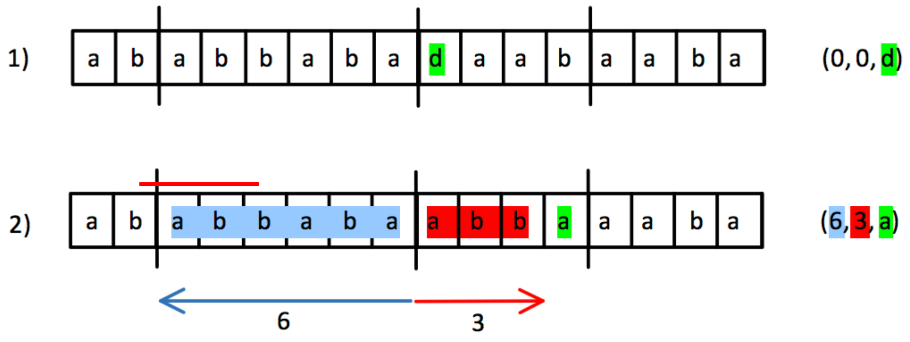
Algorithm LZ77 is an implicit dictionary compression algorithm that belongs to the family of lossless compressors.
This algorithm searches for redundant sequences and replaces them with encodings. To generate these encodings a search is made inside the data that have already been encoded and the sequences that have already been encountered are replaced with codes that allow the decompression process to expand them and go back to the parts that make up the original file.

The search for redundancies is done by scrolling a window consisting of two elements: a search buffer that contains data that has already been encoded and a look-ahead buffer that contains data that has not yet been compressed.



The encoding resulting from this research has 3 elements:
-**offset (o)**: represents how much you need to move back in the search-buffer to identify the beginning of the longest occurrence.
-**length (l)**: represents the length of the longest prefix found. If the look-ahead is size n, the value of this parameter cannot exceed n-1 because the encoding includes the byte following the prefix.
-**next char (a):** is the character that appears inside the look-ahead buffer just after the prefix.

Some examples of the encoding:


1)
a b a b b a b a d a a b a a b a
(0, 0, d)
2)
a b a b b a b a a b b a a a b a
(6, 3, a)
6
3

When the algorithm does not find any prefix in the search-buffer, the code with null offset, null length and first character in the look-ahead buffer is generated. (see example 1)
Otherwise, the generated code indicates where the prefix was found, how long it is and which character follows it. (see example 2)

2. Used data structures

Since the chosen implementation principle is the classic iterative one, the data structure chosen for compression and decompression is an array within which pointers are moved to search for occurrences.

A structure has been used to store the codes:

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

For the compression:

-**Array**:

```
unsigned char bytes_from_file[STREAM_SIZE];        // bytes read from the input file
```

-**Pointers**:

```
unsigned char *endOfBuffer;                        //pointer on the last element of bytes_from_file
unsigned char *lookahead;                          //start of look-ahead
unsigned char *window;                             //start of search-buffer
unsigned char *l_cursor;                           //pointer that is moved inside the look-ahead buffer
unsigned char *w_cursor;                           //pointer that is moved inside the search-buffer
unsigned char *w_cursor_last;                      //start of the last found prefix
```

For the decompression:

-**Array:**

```
unsigned char decompressed[STREAM_SIZE];           //decompressed bytes
struct code d[STRUCT_ARRAY_SIZE];                  //array of data structures which contains the encodings
```

-**Pointers:**

```
unsigned char *last_element=&decompressed[STREAM_SIZE-1]; //end of the decompressed bytes
unsigned char *d_lookahead = decompressed;                //start of the look-ahead buffer
unsigned char *d_window = d_lookahead;                    //start of the search-buffer
unsigned char *w_cursor;                                  //start of the prefix
```

for the bufferized writing and reading:

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

3. Implementation

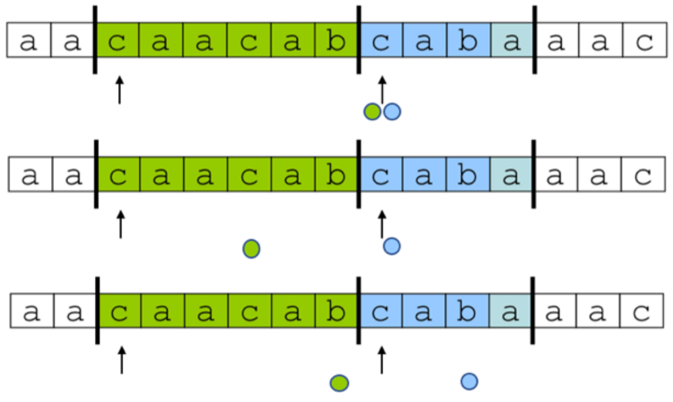
<u>COMPRESSION</u>

<u>Occurrencies search:</u>

To develop this algorithm I relied on iterative search for occurrences. The goal was to develop a compressor that works with discrete performance and then to improve its performance.

To do this I decided to implement the search for occurrences with the classic iterative method using as a search basis a simple array in which the bytes read from the input file are loaded cyclically and over which the pointers are scrolled.

The program takes care of cyclically loading in blocks of the bytes read by the input file within an array of 1'000'000 bytes and to update the position of the pointers within the latter.
The following images show how the prefix search is structured:


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

There are basically **four pointers** used:
- <u>2 reference pointers</u>: the first marks the beginning of the search-buffer while the second marks the beginning of the look-ahead.
- <u>2 support pointers</u>: these two pointers move within the two buffers to perform the actual search for prefixes.

<u>The green pointer</u> searches for the beginning of the prefix.
Once found, the green and blue support pointers are moved in parallel until their value is different or when the lookahead-1 position is reached.
The search for prefixes continues until the green pointer has reached the beginning of the search-buffer.
The longest occurrence of the current search buffer is kept saved and updated whenever a further longer sequence is encountered.

PROBLEMATIC:

The triple that forms the encodings is too expensive in terms of space occupied because:

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

In the worst case, when the encoding has to compress a single byte, 8 additional bytes are occupied, if this problem is extended to the whole file, the compression factor becomes null.

For this purpose, I have developed the buffered writing function.

Buffered writing:

The purpose of buffered writing is to reduce the number of bits needed to represent the encodings generated by the search algorithm I developed.

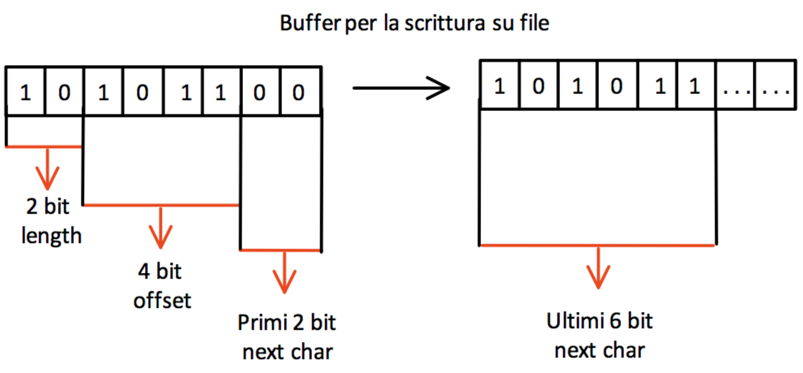
Assuming that m is the size of the search-buffer and n is the size of the lookahead buffer, it is known that the maximum offset that can be achieved is m while the length of the longest prefix reaches a maximum of n-1. The minimum number of bits to represent a char is 8 so it is not possible to save bits on this encoding element.
As the size of the search-buffer and lookahead buffer varies, so do the bits needed to represent them.

To calculate how many bits are needed to represent offsets and length we just do:

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

With this calculation, however, I lose the possibility of representing the zero offset, i.e. when the occurrence does not exist.
It is known, however, that when the offset is zero, even the length of the prefix is zero, so to save additional bits, the length is buffered first, thus allowing the decompressor to understand if what follows the bits of the length are the bits of the offset or directly those that represent the next-char.

Grandezza finestra:	16 byte
Grandezza lookahead:	4 byte



Since writing to files requires a minimum of 8 bits, the bits representing the encoding elements are linked one by one within an 8-cell buffer representing 1 byte. To do this I implemented this simple function that sorts the bits:

```
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 = encoding element to be converted in binary format
b_size = number of bytes necessary to represent it

The function concatenates the encoding elements in the "big-endian" format.

Thanks to a global variable, the program takes into account the position in which the last bit was inserted and if the end of the buffer is reached during the insertion of the bits, the buffer is converted into decimal and written to the compressed file in the form of unsigned char; the global variable is reset and the insertion can continue. Applying this process to all encodings makes compression much more efficient.

```
void fillBuffer(int buffer[], int bit, FILE *outfile){
   if(bits<BUFFER_SIZE){
      buffer[bits] = bit;
      bits++;  //position counter
   }else if(bits==BUFFER_SIZE){
      decimal = binToDec(buffer);   //binary to decimal conversion
      fputc(decimal, outfile);                //write the encoding to the file
      inizializeArray(buffer, BUFFER_SIZE);
      bits=0;                                 //reset the counter
      buffer[bits]=bit;
      bits++;
   }
}
```

<u>**DECOMPRESSION**</u>:

<u>**Buffered reading:**</u>

The process for retrieving encodings from codes generated with buffered write is exactly the reverse.
Codes are extracted from the compressed file one by one as needed.
When a code is extracted from the file, the functions decToBin and fillBuffer, already implemented in the buffered writing, convert it to binary by filling the bit buffer.

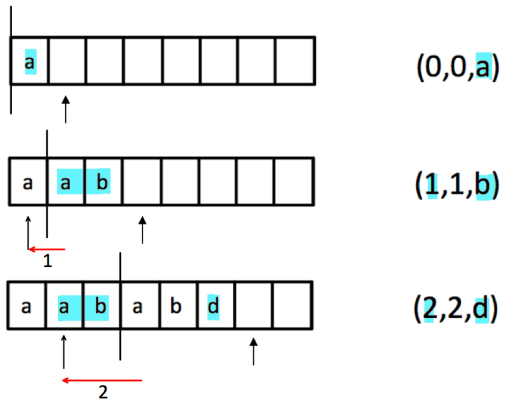
The program already knows that at the beginning of the decompression, it has to extract the bits of the length first. Once extracted, they are converted to decimal and the following check is performed: if the length is null, then I ask for the bits of nextchar otherwise I ask first those of the offset to which I add 1 (for the reason seen in the buffered writing) and then those of next-char. Each time the buffer bits are exhausted, the algorithm requests a new code and the cycle continues until the codes are exhausted.

```
int extractCodes(int buffer[]){
   n_bits--; //number of bits to be read, global variable
   int decimal = 0, c=0;
   while(n_bits>=0){
       if(buffer_position==0) {
          buffer_position=8;
          c=getNextCode(&infile);
          //if there are no nore encodings return EOF
          if(c!=EOF) {
             decToBin(buffer, c, BUFFER_SIZE);
          }else {
             return EOF;
          }
       }
       //decimal-binary conversion
       decimal = decimal + ( (buffer[8-buffer_position]) * ((int) pow(2,n_bits)));
       n_bits--;
       buffer_position--;  //extracted bits counter
   }
   return decimal;
}
```

All the encodings obtained from these conversions are saved in an array of structures representing the triple.

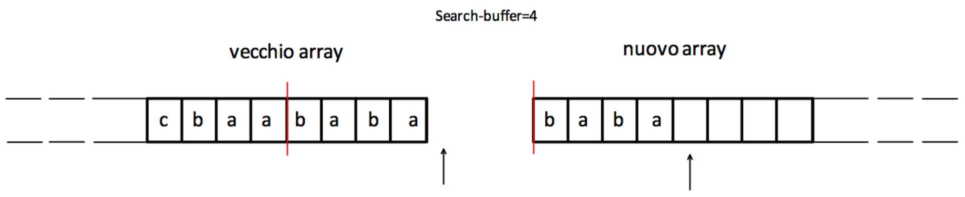
Decompression:

When it fills up completely, the array of structures is passed to the actual decompression function that takes the encodings and like the compressor uses pointers to fill a second array with the decompressed bytes.
The pointers are roughly 2:
-1 **reference pointer** that holds the position on the last decompressed byte
-1 **support pointer** moving backwards in unzipped files to find the beginning of the prefix to expand.
Once the beginning of the prefix is found, the 2 pointers move in parallel and a copy of x bytes is made, element by element, where x is the length of the prefix.



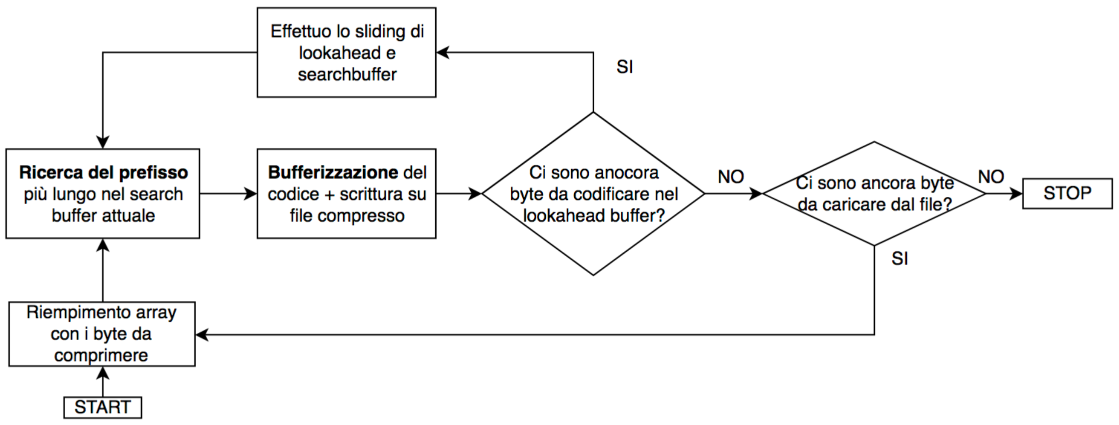
When the array of unzipped bytes is filled, it is written to the unzipped file with the fread library function and the process continues until the buffered reading provides the codes.

To have all the search-buffer available again even after writing to file, you need to copy the final n bytes to the beginning of the new array and reposition the reference pointer at the n-1 position. (n=search-buffer size).

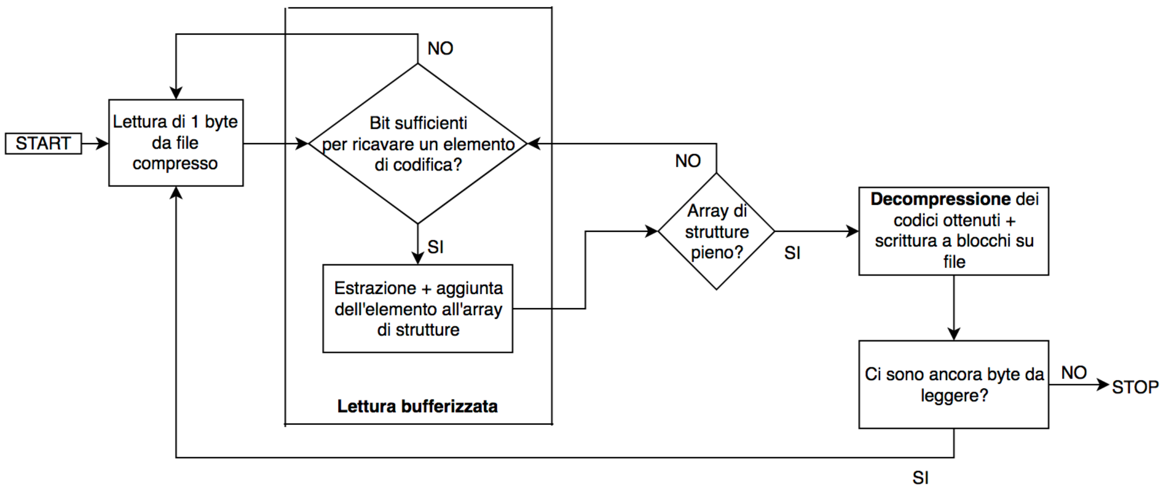


4. Source code structure

Compression:


Effettuo lo sliding di lookahead e searchbuffer
SI
Ricerca del prefisso più lungo nel search buffer attuale
Bufferizzazione del codice + scrittura su file compresso
Ci sono anocora byte da codificare nel lookahead buffer?
NO
Ci sono ancora byte da caricare dal file?
NO
STOP
SI
Riempimento array con i byte da comprimere
START

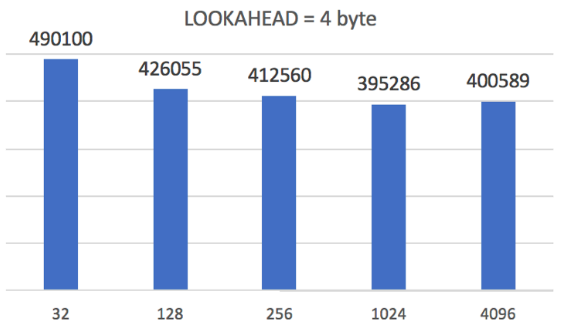
Decompression:


NO
START
Lettura di 1 byte da file compresso
Bit sufficienti per ricavare un elemento di codifica?
NO
Array di strutture pieno?
SI
Decompressione dei codici ottenuti + scrittura a blocchi su file
SI
Estrazione + aggiunta dell'elemento all'array di strutture
Ci sono ancora byte da leggere?
NO
STOP
Lettura bufferizzata
SI

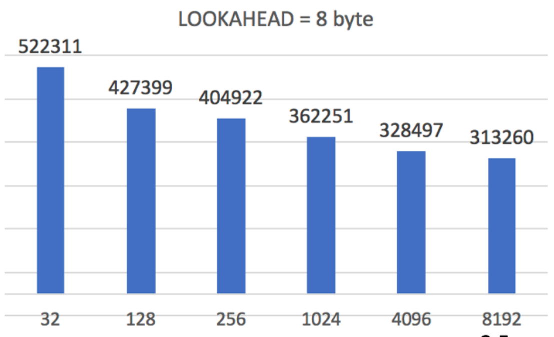
5. Results

The algorithm has been tested by initially submitting simple strings with a size 4 head buffer and a size 8 searchbuffer, to check if the generated codes are correct.
Once I got the right codes, I tested the program several times and modified the buffer and support array sizes used for compression and decompression to search for the best combination of the two.

These are the results obtained with the Divine Comedy (551'846 bytes):

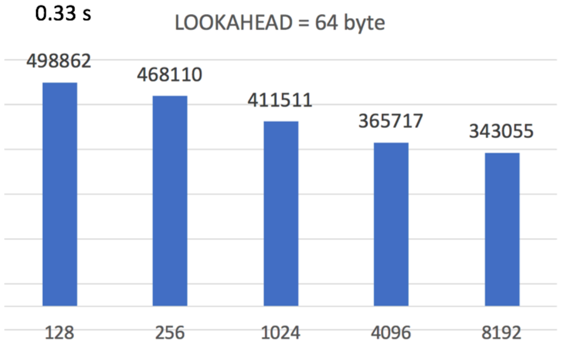


0.33 s



0.35 s

3.5 s



0.4 s

3.56 s

As you can see the best compression factor was obtained with an 8kbyte search buffer and an 8byte lookahead buffer.
With the same sizes I tested the following files:

	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 contains random bytes which makes it difficult to find very long prefixes and because of the 8kbyte searchbuffer the encodings generated for 0 offset prefixes are heavy. The result is a file that has not been compressed.
Similarly, the file ff_ff_ff is a file of only 3 bytes, with such a large searchbuffer it is impossible to compress it.

6. Encountered issues and future improvements

The biggest problem was undoubtedly the choice of the data structure to be used. Having never used search trees or linked lists, I opted for iterative development with a simple array.
Developing the prefix search algorithm took me a long time because there were many cases to consider and every time I felt I had considered them all, new problems came up.
Once the prefix search was resolved, I realized that the compression factor was not at all optimal so I had to implement buffered writing.
Having implemented the buffering process correctly, the development of the decompression was much faster than expected.

BUG 1:

One problem that I have encountered and which is still unresolved concerns the decompressor.

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

These two definitions allow you to set the size of the array of structures and the array that contains the decoded characters to write to files.
In this case, when the compression generates less than 600,000 codes, the decompression is working.
Instead, when this threshold is exceeded, since the array of structures is overwritten and loaded with new codes, the algorithm extracts a few more bits from the compressed file and generates out-of-phase encodings.
As a result, the decompression function, which operates on already decompressed bytes, goes into redundancy and no longer decompresses correctly.
This error result in partially uncompressed files.

Possible solution

Having developed decompression by extracting only one byte at a time from the compressed file, the program is less stable, increasing the possibility of bugs.
One change that could be made is to read the compressed file in blocks and manage more easily the concatenation between a block of codes and the other.

BUG 2:

In some files at the end of the decompression, one byte more or one byte less gets inserted.
The error is probably due to the fact that, as in BUG1, the buffered reading is misshandled and an extra/minus code is read at the end of the decompression.
The solution I propose is the same.

IMPROVEMENTs:

As explained in the results chapter, after testing the compressor by setting different sizes for the search-buffer and the look-ahead buffer I found that as the size of the search-buffer increases the algorithm compresses better, but is penalized on the compression speed as the search for prefixes becomes very slow.

The key to improving this is the choice of a data structure that is able to search with less complexity and fewer iterations.

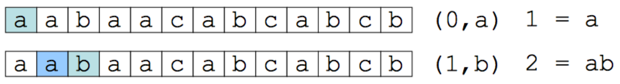
b) LZ78 (Elia Perrone)

1. Description of the algorithm

Unlike the LZ77 algorithm, which works on previously analyzed data, the LZ78 algorithm works on the management of future data, that is, on the data that still need to be analyzed.
LZ78 is based on a well defined scheme implementing the use of a dictionary, which is a tool used by the algorithm to save occurrences already analyzed.
This algorithm adds values to a dictionary and then detects again an occurrence already detected, i.e. with the same value.
When the same occurrence is verified to an occurrence previously saved in the dictionary, the algorithm outputs the index of the element inside the dictionary instead of the value.
The LZ78 generates, in the compression file, encodings of two elements: an index, which represents the index associated with the value of an element in the dictionary, and the following symbol that interrupts the correspondence with the value.

Example of coding applied to a string:

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



Note that the dictionary is constantly updated as the string scrolls.
The encodings generated by the algorithm will be inserted on the compressed file.

2. Used data structures

At the beginning of the project it was decided to approach a linear implementation of the algorithm through an iterative system, that is with the use of basic structures, with simple data types.
For this reason, the following two structures have been designed:

Encoding	Dictionary element

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

This structure represents the codes that are generated by compression. The code consists of two elements: an index, which represents the index of the element within the dictionary, and a subsequent character, which represents the character that interrupts the correspondence with a value within the dictionary. This structure is used simply to have a more simplified and ordered structure and code reading.	This structure represents an element within the dictionary. These elements are gradually created during compression and decompression. The dictionary element also consists of two components: an index, which represents the element's unique index, and a value, which can be composed of multiple characters.

Variables used for the compression:

<u>FILE:</u>
FILE *input_file: is the file to be compressed.
FILE *output_file: is the compressed file.

<u>BUFFER</u>:
unsigned char buffer[BUFFER_SIZE]: is the buffer that is used for populating, with the file to be compressed inside.
unsigned char subbuffer[BUFFER_SIZE]: is the buffer in which sub-strings extrapolated from buffers are inserted.
unsigned int bitbuffer[BITBUFFER_SIZE]: is the buffer used to reduce the size of the element indices within the dictionary (compression enhancement).

<u>VARIABLES</u>:
Output *output = malloc(sizeof(Output)): is the output object in which the encodings will be inserted during compression, and then written to files.
Element dictionary[DICTIONARY_SIZE]: is the dictionary in which the elements will be added.

<u>**Buffered writing**</u>**:**
Analyzing the size of our Output structure, we notice that it consists of two elements: an index, which is an integer, which in the language C is allocated with a size of 4 bytes, and a character, which is allocated with a size of 1 byte.
In total, for each encoding, we would have an object that weighs 5 bytes and therefore exaggeratedly large in comparison to the memory that must be allocated for a encoding.
This is because we know that the size of the index of the elements inside the dictionary varies according to the size of the file to be compressed; in fact, the larger the file to be compressed, the larger the memory allocation space for the representation of an encoding will be.
In conclusion, we can see that the size of an encoding is strictly dependent on the size of the index that takes the element of the dictionary.

The buffered writing deals with this problem.
The idea was to implement a logical system that would manage the size of the index of the dictionary elements according to the associated value. This is convenient because we know in advance the size of the indexes since they are natural numbers and are added in sequence.

E.g. Given a coding and the quantity of codings already analyzed, we can extrapolate the size, in bits, that will serve to represent a new element of the dictionary.

E.g. if the encoding is (n,c) we know that to represent the index n log2(n) bit is enough.

Variables used for the decompressione:

<u>**FILE:**</u>
FILE *output_file: is the input file that must be unzipped.
FILE *output2_file: is the unzipped file (equal to the file you wanted to compress initially).

<u>**BUFFER:**</u>
unsigned char debuffer[BUFFER_SIZE]: is the buffer that is used to populate the file to be decompressed.
unsigned char next_char_value[VALUE_SIZE]: is the buffer used for inserting the next character.
unsigned char value[VALUE_SIZE]: is the buffer used to enter the value of the dictionary element.

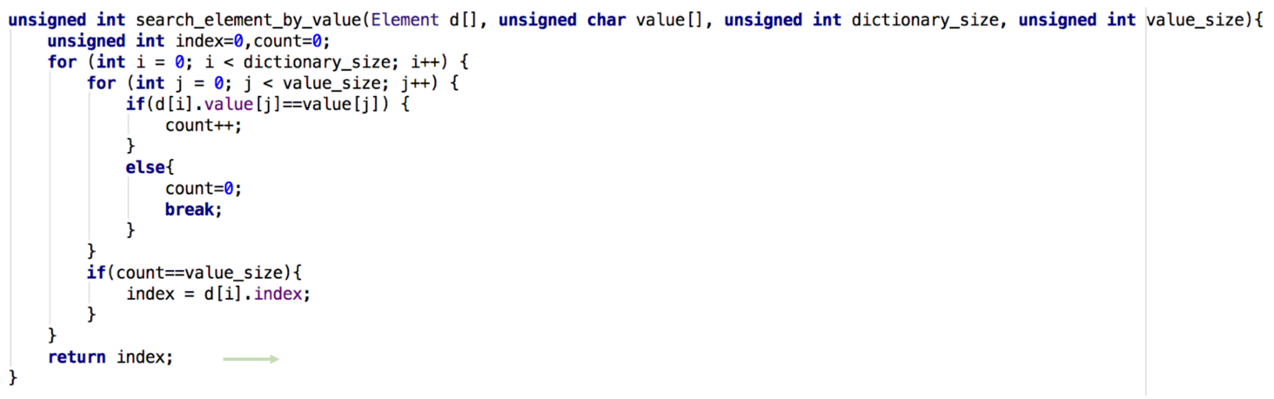
<u>**VARIABLES:**</u>
unsigned int index_value: is used to enter the coding index.
unsigned int k: used for debuffer population.

3. Implementation

<u>**Relevant compression functions:**</u>

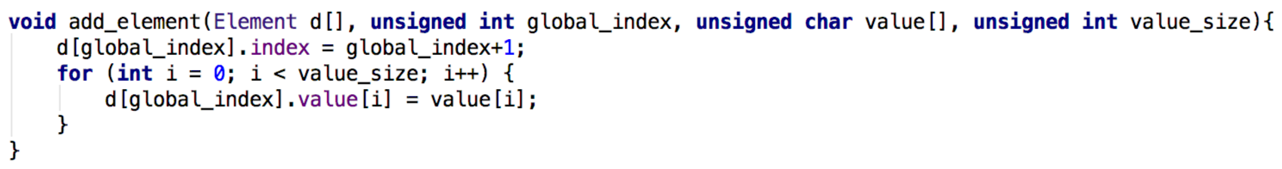
memcpy(subbuffer, &buffer[i], j);
It is a function of the standard string.h library. With this function I can, given the starting address of an array and a length, extract a substring from the main buffer.
I use memcpy to extrapolate parts of input files and search for values in the dictionary to create encodings for them.

search_element_by_value(dictionary, subbuffer, DICTIONARY_SIZE, VALUE_SIZE);
This function is specially designed for dictionary management. Given a value to be searched this function checks, of all the elements present in the dictionary, if there is a match.
If there is an element with the same value, then the function returns the value of the element's index inside the dictionary that has the same value as the requested value, while, if no corresponding element is found, the function returns 0.



```c
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;
}
```

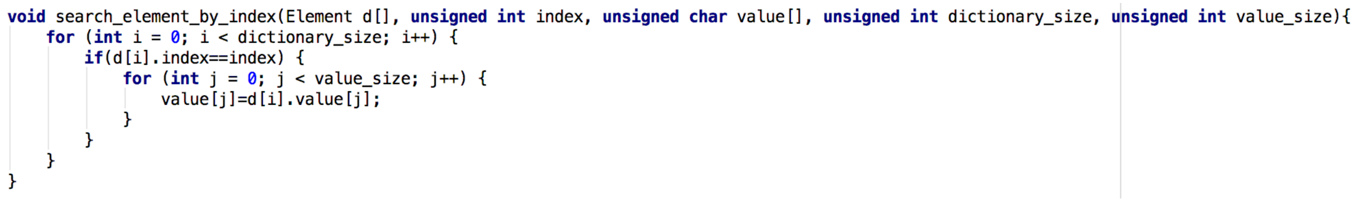
add_element(dictionary, global_index, DICTIONARY_SIZE, VALUE_SIZE);
This function is specially designed for dictionary management. Given a value to be added to the dictionary, the function simply copies the value into a new element of the dictionary. Elements are added in a row (in the queue) and a global index is assigned for each new element.



```c
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];
    }
}
```

<u>Relevant decompression functions:</u>

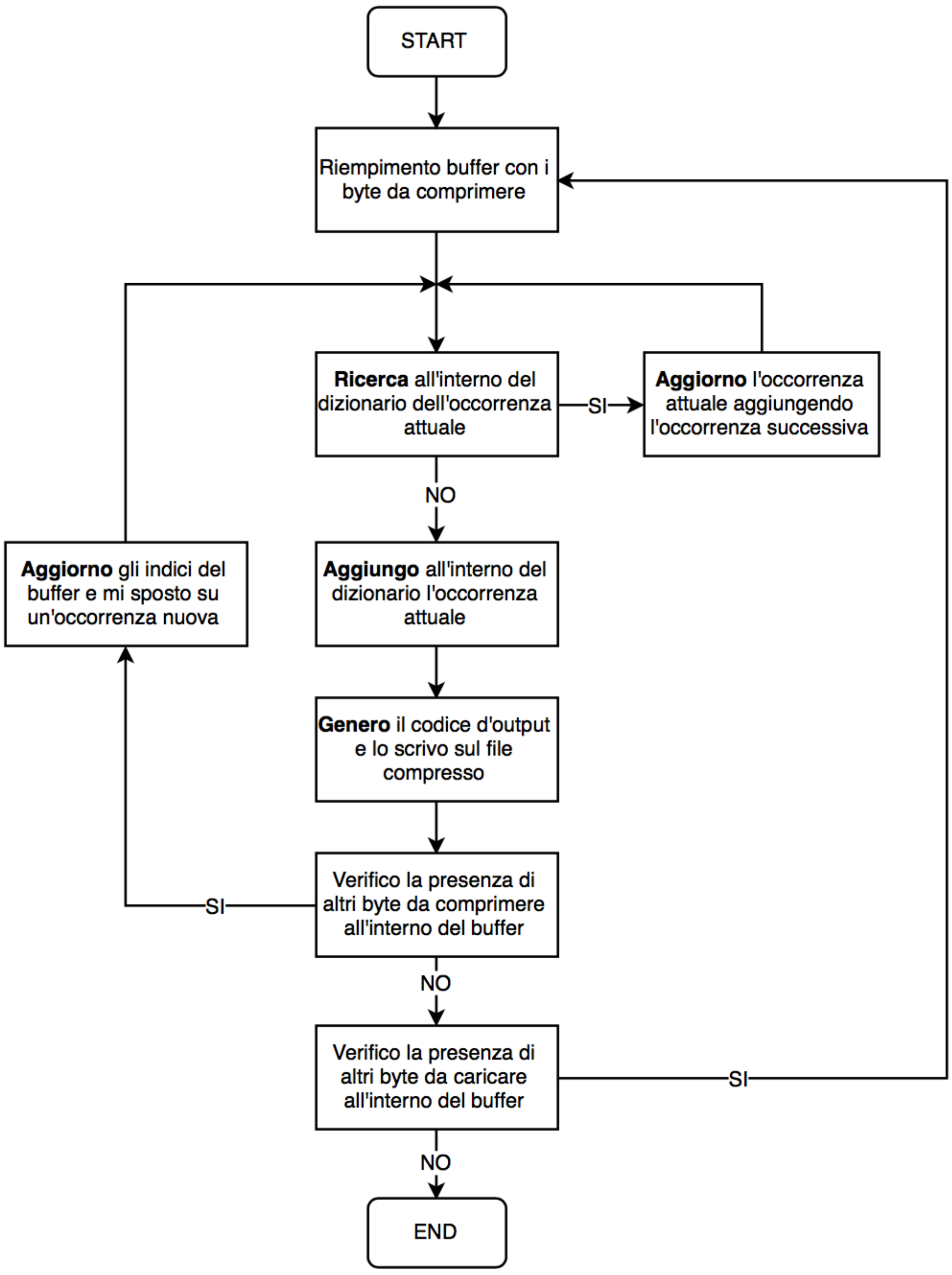
search_element_by_index(dictionary, index_value-'0', value, DICIONTARY_SIZE, VALUE_SIZE);
This function is specially designed for dictionary management. Given an index to be searched, this function compares, for all the elements present in the dictionary, if there is a correspondence of indices. If the required index is found then the function associated with the array value is the value of the found element.



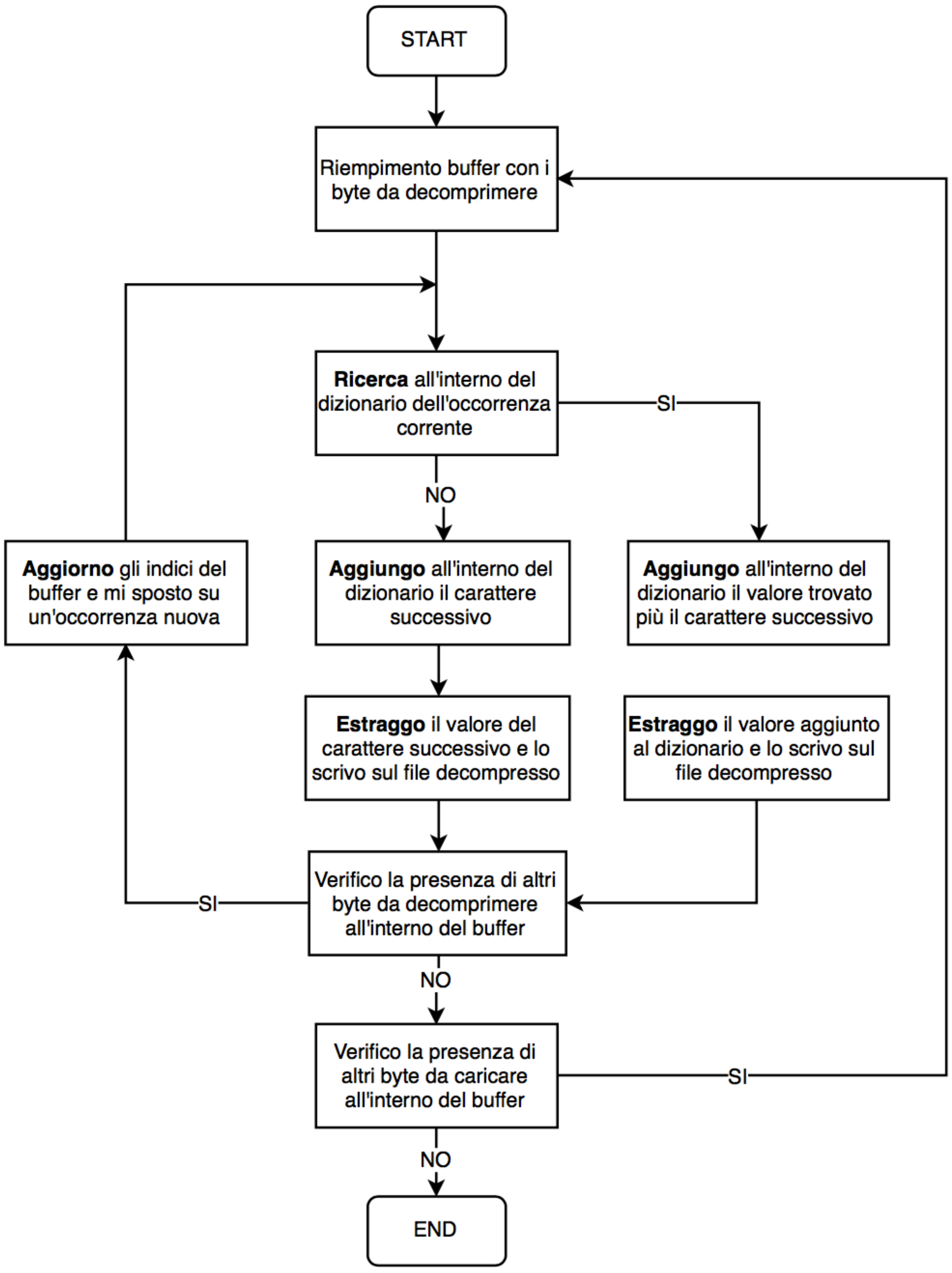
```c
void search_element_by_index(Element d[], unsigned int index, unsigned char value[], unsigned int dictionary_size, unsigned int value_size){
    for (int i = 0; i < dictionary_size; i++) {
        if(d[i].index==index) {
            for (int j = 0; j < value_size; j++) {
                value[j]=d[i].value[j];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

4. Source code structure

Compression:


START
Riempimento buffer con i byte da comprimere
Ricerca all'interno del dizionario dell'occorrenza attuale
SI
Aggiorno l'occorrenza attuale aggiungendo l'occorrenza successiva
NO
Aggiorno gli indici del buffer e mi sposto su un'occorrenza nuova
Aggiungo all'interno del dizionario l'occorrenza attuale
Genero il codice d'output e lo scrivo sul file compresso
SI
Verifico la presenza di altri byte da comprimere all'interno del buffer
NO
Verifico la presenza di altri byte da caricare all'interno del buffer
SI
NO
END

Decompression:


START
Riempimento buffer con i byte da decomprimere
Ricerca all'interno del dizionario dell'occorrenza corrente
SI
NO
Aggiorno gli indici del buffer e mi sposto su un'occorrenza nuova
Aggiungo all'interno del dizionario il carattere successivo
Aggiungo all'interno del dizionario il valore trovato più il carattere successivo
Estraggo il valore del carattere successivo e lo scrivo sul file decompresso
Estraggo il valore aggiunto al dizionario e lo scrivo sul file decompresso
Verifico la presenza di altri byte da decomprimere all'interno del buffer
SI
NO
Verifico la presenza di altri byte da caricare all'interno del buffer
SI
NO
END

5. Results

Given the use of functions belonging only to the world of strings (string.h) the algorithm works only with text files, which are the files with which I tested the operation.

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

We note that the compression times are extremely high. This is due to the use of inefficient data structures (arrays and arrays of structures) that within the search and add functions in the dictionary take an enormous amount of time to complete operations; in fact, both the search and add functions use two nested cycles that allow me to scroll through the entire array of structures (dictionary). This method is obviously not efficient.

Given the problems found, the terminal program has not yet been implemented. To perform tests simply enter the path to the file you want to compress inside the program when you run the fopen function on the files.

6. Encountered problems and future improvements

Inefficient data structures:
The first major problem was, once compression was implemented, the management of the quantities of the Element structure.
This structure in fact in the compression has been used for the realization of a dictionary, which is nothing more than an array of elements.
For the management of the dictionary, in compression, two functions are mainly used, add_element() and search_element_by_value(). These two functions scroll through all the values of the elements in the dictionary through two nested cycles. This limits the compression very much, making it extremely slow.

An improvement on the data structures would be to use a type of tree structure, because they allow a faster and more flexible dictionary management and therefore a better performance both in compression and in decompression.

Buffered reading and writing
Due to the problem of index sizes of dictionary elements (see chapter buffered writing) the implementation of buffered writing and reading, which would be used only to improve compression, is missing within the program.

3. LZ* Comparison

It is necessary to underline that the comparison we made is based only from the theoretical point of view because the algorithm LZ78 was slower than expected and this made realistic comparisons difficult.
Even with regard to decompression, it was not possible to make a practical comparison since both algorithms still present some problems of decompression: in particular the LZ77 is not yet able to decompress correctly large files while the LZ78 presents problems in reading the encodings from the compressed file.

Theoretical comparison :
As pointed out in the previous chapters, the LZ77 algorithm and the LZ78 algorithm differ on one main factor: the LZ77 works on the basis of an implicit dictionary containing already compressed bytes that is updated after each search for the longest prefix, the LZ78 instead builds an explicit dictionary working on the data that have yet to be compressed.
If we take for example a string (of 14 bytes)

a	**a**	**b**	**a**	**a**	**c**	**a**	**b**	**c**	**a**	**c**	**b**	**c**	**b**

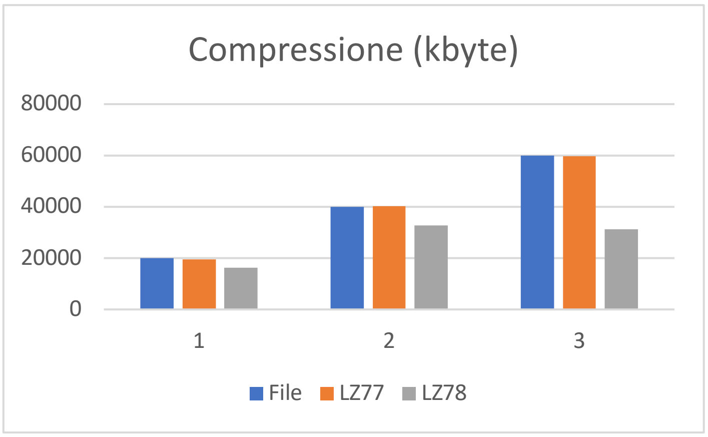
and we generate the encodings with both algorithms (LZ77: search buffer = 6bytes, lookahead = 4 bytes).

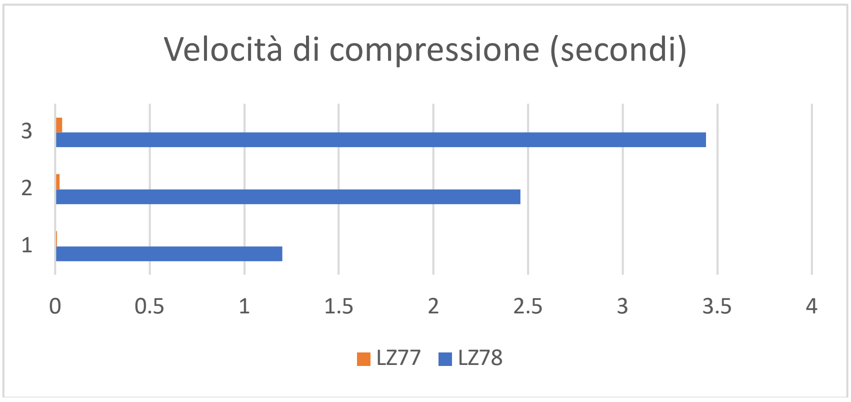
LZ78	
Codice	**Dizionario**
(0,a)	1=a
(1,b)	2=ab
(1,a)	3=aa
(0,c)	4=c
(2,c)	5=abc
(1,c)	6=ac
(0,b)	7=b
(4,b)	8=cb

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

Note that 8 encodings are generated using the LZ78 algorithm and 6 encodings are generated using the LZ77 algorithm.
Without buffered writing the encodings would have a weight of:
- LZ77: 8*4byte (index) + 8*1byte (char) = 40byte
- LZ78: 12*4byte (offset + length) + 6*1byte (char) = 54byte

Applying buffered writing, for both algorithms, we find that the total compressions have the weight of 10bytes and 9 bytes respectively on the starting 14bytes.
From a theoretical point of view, buffered writing, if implemented correctly, has a much stronger long-term impact on LZ77 encodings since the size of the lookahead buffer and searchbuffer remain unchanged throughout the compression.
For LZ78 this factor changes because if the number of indexes in the dictionary increases, the bits needed to represent them also increase.
Clearly all these observations are to be taken with the pliers because the dynamic implementation of the LZ77 allows you to change the size of offset and length allowing you to find longer prefixes that can be represented with a few bits in the best case, but with the risk of reducing the compression factor in the case of files with very short prefixes. It is important to underline that the LZ77 algorithm gives us the possibility to modify the size of the two buffers allowing a much more dynamic adaptation than the LZ78. This factor also makes it difficult to compare the two algorithms because many parameters can be considered.





The files examined are three different text files with weights of respectively: 20, 40, 60 Kbytes.
Note that the LZ78 algorithm takes much longer than the LZ77. This is explained by the use of inefficient data structures (see chapter on testing procedures for LZ78).

Theoretically LZ78 has a much faster search if implemented for example through trees. The tree search allows you to skip many prefixes that do not match the bytes to be compressed.
The search algorithm that has been implemented for the LZ78 instead scrolls through all the elements of an array checking every single byte and this in terms of performance is very expensive.