1 Introduction

This final project has the aim of developing a dictionary-based compression coding technique and comparing the gained performances with those of softwares present in a typical PC. In particular, we are required to develop our own version of the LZ77 and LZSS algorithms.

In the following we will give a brief overview of such algorithms and we will explain how our versions have been implemented, focusing on the implementation choices. Eventually, we will provide a comparison between the two algorithms and their commercial versions, when they work to compress several kinds of file.

The last section gathers all the further developments which would complete or improve the performances of our implementation.

1.1 Dictionary-based coding

A *dictionary-based coding* is a coding technique which processes a file as a sequence of symbols to build a dictionary, that is a sequence of pairs (key, value), where the value can be either a symbol or a sequence itself. The dictionary, properly formatted, is the message to send to the receiver. The latter has to build back the original sequence exploiting the information contained in the dictionary entries. Actually, for the dictionaries we will be using, we do not care very much about the keys, since the entries are stacked with the same order the receiver will use at its side, therefore we will not talk about a key for a dictionary entry, rather than about a position.

There are two kinds of dictionary-based coding technique, depending on the dictionary form. The first is the *implicit dictionary coding*, which is the case of the LZ77 algorithm, in which the entries are generated and read sequentially, without the need of specifying a key for them. On the other hand, the *explicit dictionary coding* creates some entries which are referred to with their keys and, in the decoding process, are used also non-sequentially. This is the case of the LZ78 algorithm, published by the same authors of the LZ77 one year later.

2 The LZ77 Algorithm

In this section we provide a short overview of the dictionary-based compression algorithm presented by Abrham Lempel and Jacob Ziv in 1977 [1]. The aim of this algorithm is trying to reduce the redundancy due to similar sequence repeating along the message. What it does is basically checking if a certain piece of the sequence has already been found in the past and, if it is the case, we code the whole piece with a reference to the previous alike sequence.

2.1 The algorithm

The algorithm works with two adjacent windows shifting on the right during the coding of the message. The first window is the *search window* and has length L_s; the second one is the *coding window* and has length L_c. We want to find a match between any prefix of the *coding window* and a sequence contained in the overall window given by the juxtapposition of the *search window* and the *coding window*, i.e. it is sufficient that only the first symbol of the match belongs to the *search window*, while the matched pattern can stretch in through the *coding window* itself. The window lengths L_s and L_c are two parameters of the algorithm and their choice strongly influences the compression performances.

Once we have found a match between a prefix of the *coding window* and another piece of message, we code it as a triplet (*offset*, *length*, *symbol*) where:

- **offset** denotes the number of back hops we have to do, starting from the first symbol of the *coding sequence*, to find the beginning of the matched string; it is clear that, for how it is defined, the *offset* cannot exceeds the *search window* dimension;

- **length** is the length of the matched string or, equivalently, of the prefix of the *coding window* which has been matched. This value cannot exceed $L_s + L_c - 1$ and it is exactly $L_s + L_c - 1$ when the matched string starts from the first symbol of the *search window* and ends in the penultimate symbol of the *coding window*;
- **symbol** is the first symbol after the matched prefix of the *coding window*. We encode assure the functioning of the algorithm also in case of no matches.

After the generation and the storage of a triplet inside the dictionary, we shift the windows such that the first symbol of the *coding window* corresponds to the first non encoded symbol, that is the oen encoded in the field *symbol* of the last triplet.

2.2 LZ77 Implementation

In the following paragraphs we will present our own implementation in Matlab of the algorithm, spanning the different versions we produced. As a matter of fact, several editions of the same program have been written, adopting different approaches in some parts of the code. Actually the versions differ above all for the pattern matching that has been adopted, while the rest of the code reamins basically the same.

2.2.1 Basic Coding Algorithm

First of all we choose a file and write it as a sequence of bytes (`uint8` in Matlab). We assume this is the original message, whose alphabet is made of all the possible combinations of 8 bit and has size $M = 2^8 = 256$. The first symbol of the sequence is encoded as a single symbol directly, because placing the *coding window* in the first positions makes no sense, since we would have no *search window* to exploit. Then, the *coding window* starts from the second position of the sequence and, until there are at least L_s symbols before it, the *search window* is shrunk to fit the available positions. The same thing happen to the *coding window* when we are reaching the end of the file: if less than L_c symbols are left, we use a shorter *coding window*. Note that, in order to have always a last symbol to insert in the *symbol* field of the triplet, near the end of the file the *coding window* is shrunk such that the last symbol is left out and, in case of a full match of the window, we still have one more symbol to complete the triplet.

Once the dictionary has been created, we want to write it using the least amount of memory we can. Since the values of L_s and L_c are fixed and determine the maximum values for the fields *offset* and *length* and we know that a symbol is represented with a single byte, we can estimate a maximum number of bytes to use for each triplet as:

$$l_{offset} = \left\lceil \frac{\lceil log_2 L_s \rceil}{8} \right\rceil \text{bytes} \quad (1)$$

$$l_{length} = \left\lceil \frac{\lceil log_2 (L_s + L_c - 1) \rceil}{8} \right\rceil \text{bytes} \quad (2)$$

We use exactly $l_{offset} + l_{length} + 1$ bytes for each triplet and encode separately its three components. Then we create the message to be sent by concatenating the bytes related to each entry of the dictionary, starting from the first one. Since we use a constant number of bytes for each entry and the receiver knows how many bytes are employed for each triplet component, it is easy to build the dictionary at the receiver side and perform the decoding to restore the original file.

2.2.2 Version 1: Basic

The first, basic, version of the program used to execute the pattern matching by brute force: we compare the first character of the *coding window* with each element of the *search window*. When a match occurs, we compare the second element of the *coding window* with the following symbol of the *search window* and so on until we find two different symbols. Then, if the length of the match is longer than the longest match found so far, we replace it and save also the starting point of the match in the *search window*. If, in the end, the longest match has zero length, this means that no match has been found and we have to encode one only symbol. This basic algorithm is very wasteful, because we have to try every possible position.

2.2.3 Version 2: KMP

A first improvement has been brought by using the KMP (Knuth-Morris-Pratt) pattern mathcing algorithm [2]. This algorithm is optimum for the research of a pattern inside a string: it takes a time $\mathcal{O}(n+k)$, where n is the length of the pattern and k is the length of the string. We though about using the KMP algorithm where the pattern $\mathbf{c}$ is the *coding window* and the overall string $[\mathbf{s}, \mathbf{c}]$ is the *search window* concatenated with the *coding window*.

The KMP algorithm starts with the brute force approach, comparing the first character of the pattern with each character of the string, but it allows to improve performances in pattern matching because, thanks to previous failed matches, it realize when it is possible to jump ahead and discard some trials, which will surely fail. We used the "jump mechanism" of KMP to realize a second version of the program, but the time performances keep remaining very poor, due to the many nested cycle we have to use, which are not very fast to be executed in Matlab. Moreover, each different pattern needs a *matching function* to be computed, which increases the number of necessary computations.

2.2.4 Version 3: optimistic `strfind()`

The third version uses the Matlab function `strfind()` for pattern matching, which is optimized to work in Matlab. We simply invoke this method using the *search window* as string and the *coding window* as pattern to be found. If no matches occurs we discard the last symbol of the *coding window* and repeat the procedure until either a match is found or we reduce the length of the pattern to zero, that means that we have to encode the symbol individually.

The term *optimistic* refers to the order we take different pattern: we are confident to find the longest match, and then the best one, quite soon and therefore we start by looking for the entire pattern. Of course, if there is little redundancy and there is often no match found, we have to repeat the cycle L_c times for each different *coding window*.

2.2.5 Version 4: pessimistic `strfind()`

From an empirical point of view we found that, often, we have not many long matches and hence the *optimistic* approach is not very suitable. The *pessimistic* approach consists in starting to look for a match with the shortest possible prefix of the *coding window* and executing the pattern matching through `strfind()` until we find no more matches. In this case we consider the previous match, which is the longest one, to build the current triplet. If there are no matches we know it from the first iteration and this allows us to save a lot of time.

This fourth version results to be the faster one; anyway, we must be aware that performances are stringly influenced by what kind of file we are trying to compress. Our experiments involved most of all text files or human-written files, which do not present a large redundancy. On one hand this makes the *pessimistic* approach the best one in terms of time performances; on the other hand, the LZ77 algorithm is not the best choice for compression for files with low autocorrelation.

All the results presented in the following has been obtained with this version of the program.

3 The LZSS Algorithm

The LZ77 algorithm has a lack of performances due to the constraint on the triplet to send: whenever we have to encode a single symbol, we have to waste $l_{offset} + l_{length}$ bytes because the *offset* and the *length* of the match are both zero. In fact the triplet to send is $[0, 0, \alpha]$. In 1982 Storer and Szymanski [3] proposed a different version of the LZ77 algorithm which improves the compression performances through a better management of the size of the dictionary entries.

We can distinguish two kinds of entries: those which contain the information related to an only symbol not find in the *search window* and those which refer to a previous pattern. We can use a flag bit to indicate in which case we are. If the bit is 0 the entry is made only by the code of a symbol. If the bit is 1 the entry is made of a pair (*offset*, *length*). This convention allows us to save a lot of space in memory in both cases.

3.1 LZSS Implementation

The main difference with the LZ77 implementation lies on the dictionary construction, all the rest of the code is quite the same used in in the version 4 of the LZ77 exposed in section 2.2.5, with a pessimistic usage of the function `strfind()` for pattern matching.

The problem with the usage of flag bits is that every computer works with bytes as fundamental memory unit. Since we cannot simply add a surplus bit to each sequence of bytes forming a dictionary entries, we decided to spend a whole byte to gather the flags for 8 consecutive entries. The sequence of these 8 bits is read as a only value and is transmitted as part of the message in front of the entries it is referring to. At the receiver side the knowledge of the number of bytes used to encode *offset*, *length* and *symbol* fields allows a unique interpretation of the message. Once the receiver has build the dictionary, it can proceed with the reconstruction of the original message.

4 Comparison with existing softwares

In this section we report some results obtained by the comparison between our implementations and other softwares usually installed in common PCs. We use the so called *Canterbury Corpus* as set of files to perform our examination, which is a collection of several types of file whose principal aim is being used to try compression algorithms. I want to mention right now that our implementations have no chance to behave as well as any of the professional software employed by default by a PC. The reasons are mainly two: the first one is that programs that compress files or folders into archives .ZIP use an advanced algorithm, where the simple LZ77 is enhanced by further coding, for example by a Huffman coding (this combined coding is called *LZ77 deflate* and is used by most of the .ZIP compressors). The second reason is about time: since we execute our program inside Matlab, performances are lowered by the infrastructure of the environment and, moreover, `for` cycles are not very convenient in this particular language. Hence, rather than observing performances in an absolute way, we will investigate the types of files which LZ77 and LZSS are favorable for and we will observe when the implicit dictionary-based coding are a suitable choice or not.

5 Results

In this section we illustrate some considerable results we found during the experience. The first part of the section shows the different performances reached by using the *LZ77* algorithm with different lengths of the *searching win-*

dow, the second part focuses on the combined effect of changing both *searching window* and *coding window* and the last part compares our *LZ77* and *LZSS* implementations with commercial softwares.

All the experiments we performed have been executed on seven specific types of files, part of them taken from the *Canterbury Corpus*, a collection of files with several formats to be used as sample files for compression testing. The used files are here reported (during the experimentation and the report itself we refer to each file by its index number):

- **`file_1`**: simple text file containing a sequence of character `A` repeated 10000 times (10000 bytes);
- **`file_2`**: simple text file containing a repetition of the latin alphabet (13000 bytes);
- **`file_3`**: file generated by concatening 10000 random bytes (10000 bytes);
- **`file_4`**: simple text file reporting a piece of a Shakespeare poem (13741 bytes);
- **`file_5`**: html format page (24603 bytes);
- **`file_6`**: piece of `c` code (11150 bytes);
- **`file_7`**: SPARC executable file (38240 bytes).

We chose these files to have a wide span among different kinds of files and show, in this way, the several behaviours the algorithm can assume for different scenarios. In particular, we use the file `file_1` to obtain the maximum compression capability of the algorithm, the file `file_2` was included to show the behaviours in presence of periodicity and the file `file_3`, which, being randomly generated has no redundancy to exploit and a high entropy, represents the worst scenario for compression purposes. The last four files, on the other hand, do not represent borderline or special cases, but more common examples of files one can deals with during his ordinary life.

5.1 *Searching window* variations

A big problem with implicit dictionary based compression algorithms is that periodicity of the message could not be properly exploited and thus we could be wasting compression resources. This unfavorable event occurs if the *searching window* is shorter than the period of the message, such that there is no way the algorithm can become aware of the existance of periodicity in the message. We ran the algorithm on the seven chosen files using different lengths for the *searching window*. Hereafter we report the obtained curves:

It is easy to see that performances tends to improve with the growing of the *searching window*. We could expect such a result: with shorter windows it is more difficult to find repeated pattern to refer to and, in some cases, we could neither realize that such patterns are present in the message. For a highly redundant input stream a long window allows to encode at the same time much more repetitions; even though the number of bits needed to represent a wide range of values grows with the increasing of the window, this happens with a logarithmic trend, while the amount of space used to encode a sequence byte per byte grows linearly. Clearly the usage of large windows implies a drop of performances from a computational time point of view. In the best compression scenario, we would like to have a *search window* as long as the message itself, but, since this is not suitable for time reason, commercial softwares usually work with block of symbols 32 Kbyte long with windows of

5.2 *Searching window* and *coding window* variations

The *LZ77* algorithm has, in fact, two parameters we can act to: the lengths of the *searching window* and of the *coding window*. It could be interesting wondering what happens if we stretch both the windows. From the following 2D graphs we can see that the length of the *coding win-*

dow has, actually, a low impact on coding performances. The reason is that, for a redundant message,

References

[1] J. ZIV and A. LEMPEL *A Universal Algorithm for Sequential Data Compression*, 1977, IEEE Transactions on Information Theory, vol.23.

[2] D. KNUTH, J.H. MORRIS and V. PRATT *Fast Pattern Matching in Strings*, 1977, SIAM Journal on Computing, vol.6.

[3] J.A. STORER and T.G. SZYMANSKI *Data Compression via Textual Substitution*, 1982, J.ACM, vol.29.