Optimal Parsing
In Dictionary-Symbolwise
Compression Algorithms

Il candidato
Alessio Langiu

Il relatore
Prof. Antonio Restivo

Anno Accademico 2007-2008
Contents

1 Introduction 1

I Theoretical Background 7

2 Information Theory 9
  2.1 Channel Theory 9
  2.2 Entropy 10
    2.2.1 Empirical Entropy 10
    2.2.2 Algorithmic Entropy 11

3 Data Compression 13
  3.1 Dictionary-Based Compression 14
    3.1.1 Parsing 15
    3.1.2 Static Dictionary 16
    3.1.3 Semi-Adaptive Dictionary 17
    3.1.4 Adaptive Dictionary 18
  3.2 Statistical Compression 19
    3.2.1 A Classical Example: Huffman Coding 20

4 LZ Examples 23
  4.1 LZ77 23
  4.2 LZR 26
  4.3 LZSS 27
  4.4 LZB 28
  4.5 LZH 35
  4.6 LZMA 35
  4.7 LZ78 36
## CONTENTS

### 5 Arithmetic Coding

5.1 A Brief Introduction ........................................ 41
5.2 Starting Static Model ...................................... 42
5.3 Encoding and Decoding a Generic Source .................. 43
  5.3.1 Encoding Phase ........................................ 43
  5.3.2 Decoding Phase ........................................ 43
5.4 Pseudo Code for an Arithmetic Coding .................... 44
  5.4.1 Encoding Function ...................................... 44
  5.4.2 Decoding function ...................................... 45
5.5 Incremental Transmission and Receiving ................... 45
  5.5.1 Pseudo Code ........................................... 46
5.6 Simple Implementation and Example ....................... 47

### II Theoretical Contribution

### 6 Dictionary-Symbolwise algorithms

6.1 Dictionary and Symbolwise Compression Algorithms ....... 51
  6.1.1 Dictionary-Symbolwise Algorithms and Schemes ...... 52
6.2 Optimal Parsing ............................................. 54
6.3 Flexible Parsing ............................................. 60
  6.3.1 On on-line, space optimal, flexible compressors .... 67
6.4 Dictionary-Symbolwise Can Have Better Ratio .............. 67
6.5 Optimal Parsing in Linear Time ............................. 71
  6.5.1 Optimality ............................................. 73
6.6 DAWGs and LZ77 Algorithms ................................ 74
6.7 An Online Solution .......................................... 76
6.8 Further Improvements ....................................... 77

### III Practical contribution

### 7 Algorithms Description

7.1 What is Large GZip ......................................... 81
  7.1.1 Building the Graph .................................... 81
  7.1.2 Efficient Dictionary Data Structure .................... 82
  7.1.3 Dictionary-Complete Graph ............................ 82
CONTENTS

7.1.4 Linear Size Graph .................................. 83
7.1.5 Flag Information .................................. 84
7.1.6 Arcs Weight ...................................... 84
7.1.7 Shortest Path ...................................... 84
7.1.8 Coding Details .................................... 85
7.2 Optimal LZ78 ......................................... 85
7.2.1 Parsing of the Input File ......................... 85
7.3 Shortest Path ........................................ 87
7.4 Writing the Output in Bits .......................... 87
7.5 Decompression Phase ................................ 88
7.5.1 A Simulation with Arithmetic Coding .......... 88
7.6 Why Using C ........................................ 88

8 Experimental Results of Large GZip ................. 91
8.1 Introduction ......................................... 91
8.2 Parsing Optimality ................................ 92
8.3 Gain At Dictionary Growing ...................... 95
8.4 Comparison with Other Compressors ............ 95
8.5 Corpus Results ..................................... 97
8.5.1 Calgary Corpus ................................ 97
8.5.2 Canterbury Corpus ............................. 98
8.5.3 Packing English Wikipedia File .............. 99
8.6 Pure Dictionary Simulation ....................... 100

9 Experimental Results of Optimal LZ78 .......... 101
9.1 Calgary Corpus .................................... 101
9.2 Canterbury Corpus ................................ 102
9.3 Test English Wikipedia File ................... 104
9.4 Gain With Symbolwise Ability .................. 105
9.5 Simulation of Optimal Parsing Using Arithmetic Coding .......... 105
9.6 Simulation’s Idea and Results .................. 106
9.6.1 Results on Calgary Corpus .................... 106
9.6.2 Results on Canterbury Corpus ............... 107
9.6.3 English Wikipedia ............................. 108
9.7 Comparison Between Most Famous Compressors .... 109

10 Conclusions .......................................... 111
A Summary of work in Italian 113
List of Tables

4.1 Encoding examples with $C_\gamma$ and $C_{\gamma'}$ ........................................ 31
4.2 Encoding examples on short strings ................................................................. 34
4.3 Encoding examples on long strings ................................................................. 34
4.4 LZ78: Complete example on Input Text “aaabbabaabab” ................................. 37

5.1 Compression and Decompression time (in seconds) with Gzip and Rar .......... 42
5.2 Static Model for Arithmetic Coding ................................................................. 42
5.3 Output progressive on message eaii! ............................................................... 43

8.1 Lgzip 32K win on gzip -9 ............................................................................. 93
8.2 Lgzip 32K very close to 7z gzip .................................................................. 93
8.3 Dictionary-Complete Lgzip 32K equal to 7z gzip ......................................... 94
8.4 Lgzip 64K very close to 7z zip .................................................................. 94
8.5 Dictionary Complete Lgzip 64K equal to 7z zip .......................................... 95
8.6 Results at Dictionary grow on book1 (770KB) ............................................. 96
8.7 Results at Dictionary grow on bible.txt (4MB) ............................................ 96
8.8 Results at Dictionary grow on enwik (100MB) ............................................ 96
8.9 Comparison with Other Compressors ............................................................ 97
8.10 Results of lgzip on Calgary Corpus .............................................................. 98
8.11 Results of lgzip on Canterbury Corpus ........................................................ 99
8.12 Results of lgzip on enwik files ................................................................. 99
8.13 Gain at Symbolwise Improvement on enwik files ...................................... 100

9.1 Results on Calgary Corpus ............................................................................. 102
9.2 Results on Canterbury Corpus ....................................................................... 104
9.3 Results on n bytes of English Wikipedia ..................................................... 104
9.4 Gain on English Wikipedia with Optimal Parsing ...................................... 105
9.5 Simulation on Calgary Corpus ....................................................................... 106
LIST OF TABLES

9.6 Simulation on Canterbury Corpus ........................................ 107
9.7 Simulation on \( n \) bytes of English Wikipedia ...................... 108
9.8 Comparison on Calgary Corpus ........................................ 110
9.9 Comparison on Canterbury Corpus ...................................... 110
A.1 Risultati di lgzip su enwik file ........................................ 117
## List of Figures

3.1 General view of compression schemes ................................. 13
3.2 Optimal parsing using a shortest path algorithm .................. 15
3.3 *LZ* example on string *abbaabbabab* ............................. 18
3.4 Binary tree on input string *s* ...................................... 21

4.1 The sliding window of *LZ77* ......................................... 24
4.2 Modulo $N$ numbering of the window ($N = 11$) .................... 28
4.3 *LZB* coding for string “abababbbabcaaaaa” ....................... 30
4.4 Trie data structure for *LZ78* coding on input *aaabbabaabab* .... 37

6.1 Classical parsing graph ............................................ 55
6.2 Locally but not globally optimal parsing .......................... 59
6.3 Greedy parsing graph with base path .............................. 72
6.4 Creation of the parsing graph $G'_T$ using DAWGs ................. 75
Chapter 1

Introduction

Nowadays Data Compression programs are used everywhere. They are used, more or less explicitly, in every transmission channel to save transmission band, from web pages transmission to satellite video broadcasting, from urban phone calls to on-board bus communication between hardware chips or in data storage systems to save space. Less direct is their use in encryption systems to enforce them due to the fact that a compressed message has less redundancy that its original version and this makes it very similar to a random string, which is the core of encryption security. But, which is the best compressor? The first topic of the Frequently Asked Question in data compression (FAQ in short, cf FAQ in dataCompression [18]) has no single answer. It depends on the problem and on the the application it is meant to achieve. In some problems the aim is to achieve a compression as good as possible, without considering compression time and decompression speed. In others it could prevail decompression speed, while in others, furthermore, it could prevail a trade-off among some parameters.

The main parameters to evaluate a compressor are:

- Compression ratio.
- Compression speed.
- Decompression speed.
- System requirements.

This last parameter is often dependent on the other three. Other parameters could be the possibility of creating hardware implementations for specialized devices, the opportunity of being able of answering to specific queries on compressed data,
and so on. Over Internet and transmission channels compressors that achieve good (non necessarily excellent) compression ratio but a very low decompression time, are typically preferred. And this, without considering the compression time. In fact a file is usually compressed only once, but it will be certainly decompressed a lot of times, possibly in real time.

Among the compressors that, in actual systems (i.e. personal computers), grant excellent decompression speed, principal class is constituted by dictionary-based compressors. In fact, decompression substantially takes place copying portion of files just decompressed. This “copy and paste” is performed very quickly because its low calculus complexity. If our intention is, instead, to obtain the best compression, the most of actual compressors are based on statistical methods, sometimes called symbolwise methods. Indeed in [4] authors say that their main result “gives added weight to the idea that research aimed at increasing compression should concentrate on symbolwise methods, while parsing methods should be chosen for speed or temporary storage considerations.”

In this thesis I resume and continue work that other and I did in [38], starting to develop a theory of dictionary-symbolwise data compression schemes, contributing to clarify classification of algorithms in the compression set. Several viable algorithms and most of the commercial data compression programs are, following this definition, dictionary-symbolwise.

Many of them have been classified up to now just as “dictionary” algorithms or programs. But, this classification was unsatisfactory for several reasons and our new subdivision allows to improve and/or generalize previous results and to obtain new ones. Famous programs, such as gzip, zip or cabarc are dictionary-symbolwise, according to this work definitions.

On the other hand, in the scientific literature there are many data compression algorithms that are just catalogued as context mixing, CM in short as are called in the “large text compression benchmark” web page [33]. They are a sort of “cocktail” in which they are mixed many kinds of compression techniques to improve compression rates. Some of them fall now in dictionary-symbolwise class.

In particular, there are two famous compression methods that can work together: the symbolwise and the dictionary encoding, which are sometimes referred to as statistical encoding and parsing (or macro) encoding, respectively. The fact that these methods can work together is commonly accepted in practice but no theory, to our best knowledge, has been developed on this subject up to now.

In [4] it is possible to find a survey on these two methods and a description of deep
relationships among them (see also [48]). In particular in [4] it is stated that:

“We conclude that in principle, nonadaptive symbolwise and greedy parsing schemes are equivalent in their potential for compression performance. However, in practice, superior compression will more easily be achieved by symbolwise schemes. In particular, the compression achieved by parsing schemes suffers because of the loss of context at phrase boundaries. The principal advantages of parsing encoding schemes are that they can be fast, and require a more modest amount of memory.” And “... research aimed at increasing compression should concentrate on symbolwise methods, while parsing methods should be chosen for speed or temporary storage considerations.”

Indeed, about fifteen years later, as predicted in [4], the best compression program reported in [33] concerning compression ratio for ewik8 and ewik9 is durilca_4_linux that uses a symbolwise method, while the fastest in decompression is cabarc that is based on the LZ77 dictionary method.

From now on, a data compression algorithm that uses both a dictionary and a symbolwise encoding belongs to the class of dictionary-symbolwise compressors. Bell, in his 1986 paper ([3]) implements a suggestion made earlier by Storer and Szymanski in 1982 (see [46]), i.e. to use a free mixture of dictionary pointers and literal characters in the Lempel-Ziv algorithms, literals only being used when a dictionary pointer takes up more space than the characters it codes. Bell’s implementation of this scheme is called LZSS, and adds an extra bit to each pointer or character to distinguish between them. LZSS uses no symbolwise compression but, since the identity can be viewed as a particular symbolwise compressor, from a purely formal point of view we say that LZSS is dictionary-symbolwise.

So, why should we use dictionary-symbolwise compressors? There are several reasons, some practical and some theoretical.

From a practical point of view, a fast symbolwise compressor coupled with a dictionary compressor allows some more freedom degrees to a parsing that can increase compression ratios without slowing up the process. Or, at the other extreme, a dictionary compressor coupled with a powerful symbolwise compressor can speed up the decompression without decreasing the compression ratio. Indeed, as we have already mentioned, many viable programs are actually dictionary-symbolwise following our definition. More details will be given in the section that concerns experimental results.

From a theoretical point of view Ferragina et al. (cf. [17]) proved that the compression ratio of the classic greedy-parsing of an LZ77 pure dictionary compressor may be far from the bit-optimal pure dictionary compressor by a multiplicative fac-
tor $\Omega(\log(n)/\log\log(n))$, which is indeed unbounded asymptotically. In section 6.4 it is showed a similar result between the bit-optimal pure dictionary compressor and a symbolwise compressor. Therefore a bit optimal dictionary-symbolwise compressor can use the symbolwise compressor to avoid some pathological situation and be provably better than a simple bit optimal pure dictionary compressor.

Dictionary algorithms obtain compression substituting fragments of the starting message with a reference to a word in a dictionary. Skipping dictionary provenance problem or its formation and maintenance, the key step of this kind of algorithms is the parsing choice, i.e. to choose the message fragmentation that leads to a good compression.

Many papers deal with optimal parsing in dictionary compressions (see for instance [46, 36, 27, 4, 9, 28, 34, 31, 50, 30, 21, 17]), that is a way of parsing the input string that minimizes the cost of the encoded output. Sometimes the cost of a pointer is considered constant, and in this case an optimal parsing splits the text in the smallest number of dictionary phrases. Under certain conditions (which actually apply to LZW for instance) this phrase optimality translates into bit optimality.

A classic way for obtaining an optimal parsing is by reduction to a well-known graph theoretical problem (see [44]). This holds true also for dictionary-symbolwise compression (cf. Theorem 1), if we consider a natural extension of the optimality notion to such kind of algorithms and if we formalize a natural extension of the graph considered in [44] that here is called Schuegraf’s graph in honour to the first author.

This approach, however, was not recommended in [44] because it is too time consuming, i.e. quadratic worst case and $O(n\log(n)/H)$ in average for memoryless sources for LZ77 schemes. The average result was not considered in the original paper but it is an easy consequence of classical results (cf. [37]). In order to get over the quadratic worst case problem, one has to limit the size of the graph. In order to make this, it is possible to use different approaches. In [31] a pruning mechanism is described, that allows a more efficient computation and that still enables the evaluation of an optimal solution. The pruning process may be applied in all cases for which the cost function satisfies the triangle inequality.

The approach described in this thesis it use appropriate subgraphs of the Schuegraf’s graph. Indeed it is showed a novel method that allows to obtain some optimal parsing of any input in linear time under a natural hypothesis on the encoding algorithm and on the cost function. In particular, the constraint on the cost function is weaker than the triangle inequality. It is just required that the cost of dictionary point-
ers must be constant. This hypothesis is natural in LZ78 schemes and it has been used in the main recent optimal-parsing results (cf. for instance [9, 36]). In LZ77-derived scheme, using additional arcs, this method obtain an optimal parsing also when the cost of dictionary pointers depends logarithmically on the dictionary phrase size, and it is linear for all practical purposes.

The results for the LZ78 case can also be seen as a generalization to the dictionary-symbolwise case of the result presented in [9] (see Theorem 4). Similarly it is generalized the main result obtained in [36] and further studied in [34] (see Theorem 2).

The proposed technique works for any kind of dictionary, static or dynamically changing, for any pointer-coding function and for any symbolwise compression. This also includes all cases considered in [31].

A novel approach about subgraph creation is considered in [17], and the ideas of this new approach are compatible to ours and allow further improvements (see Section 10).

An extended experimentation work was made to support dictionary-symbolwise theory and optimal parsing theory, showed in experimentation report sections of this thesis. Many of the experiments was based on LZ77 and LZ78 dynamic dictionary algorithms and Huffman statistical encoding as symbolwise. We especially focused on GZip [20] algorithms, based on LZ77, that naturally offer a platform for dictionary-symbolwise implementation and optimal parsing research.

A very interesting implication of optimal parsing research was to make an extended edition of GZip, called LGZip for Large GZip, thanks to the subgraph used as near optimal parsing solution. This subgraph allow us to overcome the limits imposed by resource requirement due to quadratic complexity of complete graph handling in parsing optimization used in many LZ-based program, like GZip modern implementations, eg. 7Zip [1] or others, or Zip program and library, when maximum compression is expected to be done. This is just one of the potential practical implications of this theory that can be applied to almost all the LZ-based compression algorithms and their specific field-based versions. More exhaustive applied research could be subject of a broader research project.
Part I

Theoretical Background
Chapter 2

Information Theory

2.1 Channel Theory

Information theory was born in 1948 as mathematical theory of communication with papers of Claude E. Shannon [45] which contained the basic results for simple memoryless sources and channels and introduced more general communication systems models. In the Shannon transmission model there is a source that generates messages according to a probability distribution of symbols over an alphabet. A channel with an alphabet of its own, with a mapping function that changes every source symbol in the channel a code, that may be longer than one channel symbol, and a cost function for the channel alphabet.

At this point the theory is divided in

- noisy channel, where the compression, removing redundancy, contrasts with the necessity of making the message resistant to noise
- noiseless channel, where the compression is just bounded by the need of leaving the message being decoded uniquely.

So, in the noiseless channel model, we found that the lower bound for the compression, low cost of channel coding, is the entropy of the source, i.e. the mean information of one source symbol, or, in other words, the amount of information generated by the source in a unit of time. Obviously there are many types of source.

- the memoryless, where each symbol is independent from its previous one,
- the iid, independent and identically distributed source
• the memory one, i.e. the Markov model distributed and the Hidden Markov Model, like the human language is.

All of these are ergodic sources, the most important and used ones. Results providing unbeatable bounds on performance are known as converse coding theorems or negative coding theorems. Shannon also showed us a way to code the source message, that, even if suboptimal, is an upper bound to compression. This if the Shannon coding, that is shown to approach the entropy of the source asymptotically at the limit of \( n \) that tends to infinity where \( n \) is the length of a block of source symbols. This result describes the performance that is actually achievable, at least in the limit of unbounded complexity and time, is known as positive coding theorems. Other coding methods are known to be optimal, as the Huffman one, that is guaranteed to reach the entropy compression level in the hypothesis that we could assign a real cost to the code word transmission. When the same quantity is given by both positive and negative coding theorems, one has exactly the optimal performance theoretically achievable by the given communication systems model. This coding theorem is known as the noiseless source coding theorem.

### 2.2 Entropy

Based on the observation that more little is the probability of an event, more information brings its happening and on the additivity of information function, we define the information event \( E \) of probability \( p_E \), as

\[
I(E) = \log_2 \left( \frac{1}{p_E} \right).
\]

Given \( S \) a memoryless source over the alphabet \( \Sigma = \{s_1, ..., s_n\} \) of probability \( \Pi = \{p_1, ..., p_n\} \), the mean information of the source \( S \) is the weighed mean of the symbols information, defined as the entropy of the source. \( H(S) = - \sum_{i \in \Sigma} p_i \log(p_i) \). The logarithm base determines the measure unit, due to logarithms propriety \( \log_a(x) = \frac{\log_b(x)}{\log_b(a)} \).

It is possible to define the entropy of the other type of sources such as MM, HMM, and ergodic sources.

#### 2.2.1 Empirical Entropy

Let us consider a string \( w \in \Sigma^n \), with \( \Sigma = \{\sigma_1, ..., \sigma_q\} \) we call \( n_i \) the number of occurrences of symbol \( \sigma_i \) in \( w \). Let us define entropy of zero order the quantity \( H_0(w) = - \sum_{q} \left( \frac{n}{n} \log \frac{n}{n} \right) \) and the \( k \) order entropy as \( H_k(w) = - \sum_{v \in A^k} \left( \frac{n}{n} \log \frac{n}{n} \right) \).
2.2.2 Algorithmic Entropy

Another measure of message information is its Kolmogorov complexity. More information has a message, more complex it is. The Kolmogorov complexity of a sequence of symbols is the shortest computer program which will generate that sequence as its output.

So, the complexity of a string is the length of the string’s shortest description in some fixed description or programming language. It is easy to see that the description of any string cannot be much larger than the length of the string itself. Strings whose Kolmogorov complexity is small relative to the string’s size are not considered to be complex. Kolmogorov randomness (also called algorithmic randomness) defines a string as being random if and only if it is shorter than any computer program that can produce that string. A random string is also an "incompressible" string, in the sense that it is impossible to give a representation of the string using a program whose length is shorter than the length of the string itself. For certain classes of sources, the Kolmogorov complexity converges, on average, to the entropy.
Chapter 3

Data Compression

As shown in Figure 3.1, many approaches exist to text compression. Among them two of the most important are:

1. Statistical methods.

2. Dictionary methods.

Figure 3.1: General view of compression schemes

In this chapter we are going to explain the main characteristics of these two classes, pointing out their features, their advantages and disadvantages. Moreover, we will
describe some possible implementations of the methods described.

3.1 Dictionary-Based Compression

Formally a dictionary $D=(M,C)$ is a finite set of phrases $M$ and a function $C$ that maps $M$ into a set of codes. Dictionary coding achieves compression by replacing groups of consecutive characters (phrases) with indexes into a dictionary. The dictionary is a list of phrases that are expected to occur frequently. Indexes are chosen so that on average they take less space than the phrase they encode, thereby achieving compression.

Dictionary encoding has enjoyed popularity for several reasons:

1. It is intuitive, and the same principle is used in context different from computers. For example, think about the best way to write today’s date: you’ll never write the complete name of the current month, it is better to write a number among 1 and 12. Moreover, we would never write “see chapter entitled Dictionary-based compression”, it is better to write “see Chapter 4”.

2. The size of indexes to the dictionary can be chosen to align with machine words, obtaining an efficient implementation. This contrast with statistical coding, which inevitably requires manipulation of groups of bits of different size within a machine word. On the contrary, the codes for many dictionary schemes are multiples of 4 or 8 bits.

3. Dictionary schemes can achieve good compression using only a modest amount of computer time and memory because one dictionary reference may encode several characters. The compression achieved by the better ones is outperformed only by statistical schemes using high-order context models, and these require considerably more computing power because effort must be put into coding each character rather than each phrase.

The most important decision in the design of a dictionary scheme is the selection of entries in the coding dictionary. This choice can be made by using three schemes: static, semi-adaptive or adaptive algorithms. The simplest dictionary scheme uses static dictionary, generally containing only short phrases. However, the best compression is achieved by adaptive schemes, that allow large phrases. Lempel-Ziv coding is a general class of compression methods that fit the latter description. Although we obtain the best results with the adaptive schemes, also static and semi-adaptive have
their advantages. In fact, they are especially suited to the coding, for example, of bibliographic database, where records are to be decoded at random but the same phrase often appears in different records. Let’s consider a database containing information about some periodicals. The phrase “Journal” or “love” will certainly be common. If you insert these phrases into a dictionary, you can encode every their occurrence in a few bits on the output file. However, we will widely describe these techniques ahead.

3.1.1 Parsing

Once a dictionary has been chosen, there is more than one way to choose which phrase in the input text will be replaced by indexes to the dictionary. The task of splitting the text into phrases is called parsing. Obviously, the most practical, easiest and cheapest approach is greedy parsing, where at each step the encoder searches for the longest string $m \in M$ that matches the next characters in the text, and uses $C(m)$ to encode them. For example, if $M = \{a, b, ba, abb\}$ and $C(a) = 00$, $C(b) = 010$, $C(ba) = 0110$, $C(bb) = 0111$ and $C(abb) = 1$, then $babb$ is coded in 8 bits as $C(ba)C(bb) = 01100111$.

As we could expect, greedy parsing is not necessarily optimal. In fact, the string $babb$ could have been coded in only 4 bits as $C(b)C(abb) = 0101$. This is particularly explicit in Figure 3.2.

![Diagram](image)

<table>
<thead>
<tr>
<th>$m$</th>
<th>$C(m)$</th>
<th>$L(m)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>00</td>
<td>2</td>
</tr>
<tr>
<td>$b$</td>
<td>010</td>
<td>3</td>
</tr>
<tr>
<td>$ba$</td>
<td>0110</td>
<td>4</td>
</tr>
<tr>
<td>$bb$</td>
<td>0111</td>
<td>4</td>
</tr>
<tr>
<td>$abb$</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 3.2: Optimal parsing using a shortest path algorithm
However, determining an optimal parsing can be difficult in practice, because there is no limit to how far ahead the encoder may have to look. In fact, if you are parsing the character \( i \) of a length \( n \) file, you could have to look to all the \( n - i \) bytes of the file for establishing in an optimal way which is the best thing to do now. The task of optimal parsing can be transformed to a shortest-path problem, which can be solved by existing algorithms. This solution is widely described in Section ??, while in next paragraph it is described the solution adopted by Deflate algorithm. This latter is the method used by gzip when passing it “-8” as parameter that stated how much the program has to effort for achieving compression. We describe it because in a first step of our parsing (before discovering the optimal solution) we used a method very similar to this one.

**Deflate’s Greedy Solution: Lazy Matching**

To improve overall compression, the compressor optionally defers the selection of matches ("lazy matching"): after a match of length \( N \) has been found, the compressor searches for a longer match starting at the next input byte. If it finds a longer match, it truncates the previous match to a length of one (thus producing a single literal byte) and then emits the longer match. Otherwise, it emits the original match, and, as described above, advances \( N \) bytes before continuing.

Run-time parameters also control this "lazy match" procedure. If compression ratio is more important, the compressor attempts a complete second search regardless of the length of the first match. In the normal case, if the current match is "long enough", the compressor reduces the search for a longer match, thus speeding up the process. If speed is more important, the compressor inserts new strings in the hash table only when no match was found, or when the match is not "too long". This degrades the compression ratio but saves time since there are both fewer insertions and fewer searches. However, in [42] it is possible to take vision of a widely discussion.

### 3.1.2 Static Dictionary

Static dictionary encoders are useful for achieving a small amount of compression with very little effort. One fast algorithm that has been proposed several times in different forms is digraph coding, which maintains a dictionary of commonly used digraph, but we are not going to explaining it in a deep way.

We would like to point out the main idea under a static dictionary-based method works. You have to identify the words or the phrase that probably will appear in
the text you are going to encode. In this way, when your program recognizes them during the compression, it can substitute them with a code that costs few bits on the output file. Following another example: it could be very useful for the database of a biblioteque, because, certainly, there are a lot of words and phrases that often appear, like “printed in”, “editor”, “author”, and so on. You could obtain a fast compression and decompression, using small memory requirements too.

The problem is that for every file you’d like to compress, you must always serve of the same dictionary. This great problem caused this approach to be not widely diffused. In fact, obviously, a C-source file would never achieve a good compression with the same dictionary. For this reason, in a semi-adaptive approach, it is possible to select the dictionary respect to the input file, and then to perform compression.

3.1.3 Semi-Adaptive Dictionary

Let’s proceed our work by analysing the semi-adaptive dictionary encoding. A natural development of the static approach is to generate a dictionary specific to the particular text being encoded. The dictionary will accumulate phrases that occur frequently in the text, such as technical terms that would not be found in other texts. For example, in this chapter we are writing a lot of times the word *dictionary*, and, in this context, if we should compress this chapter, it would certainly be useful to insert this word in the dictionary. We want to point out that in another text, like a tale, probably this action would not be a good choice. So, it is a choice made respect to the input, in other words, adaptive.

Problem: the task of determining an optimal dictionary for a given text at our best knowledge is NP-complete in the size of the text. The proof of what just written shows that the problem of finding an optimal dictionary can be solved using a form of the *vertex cover problem*. However, many heuristics have found out that find near-optimal solutions to the problem is possible. Usually a maximum number of entries in the dictionary is chosen. We won’t describe in depth how to find out the most suitable words to insert in the dictionary. But, it is important to underline that an algorithm that achieves this objective has to consider a trade-off between compression ability and coding speed. In fact, if you need an optimum dictionary, the algorithm must spend a lot of time in this choice. On the contrary, it can obtain a bad dictionary but in less time. Consequentially, the compression ratio depends on this choice.
3.1.4 Adaptive Dictionary

Lempel-Ziv’s Idea

A 1967 paper described a semi-adaptive dictionary coding technique based on the idea that better compression could be achieved by replacing a repeated string by a reference to an earlier occurrence. This idea had no great success until 1977, when Jacob Ziv and Abraham Lempel described an adaptive dictionary encoder in which they employed the concept of encoding next phrase of the input using a pointer to the longest substring seen in past that matches with it, if it exists. In other words. Let $T$ be an input text, $i$ an index, $s$ the portion of $T$ on the left of $i$ and $t$ the portion of $T$ on the right of $i$. The idea is to find out the longest prefix of $t$ that is factor of $s$. Great idea. At our best knowledge, almost all adaptive dictionary encoders fall into a family of algorithms derived from Lempel and Ziv’s work. This family is called LZ-coding.

Let’s show a little example. $s =$ abbaabbbabab. The output will be: abba(1,3)(3,2) (8,3). Figure 3.3 helps the comprehension of this example.

![Diagram of LZ example on string abbaabbbabab]

Figure 3.3: LZ example on string abbaabbbabab

The decoding phase simply replaces an index by an already decoded text to which it points. It is extremely easy, fast and cheap under the point of view of system resources requirements. This is probably the main reason thanks to which LZ-based algorithms are the most used in particular for compressing files to send via Internet (that require
almost real-time decompression speed) and for storing great archives, where a great
amount of data has to be obtained without time loss.

**Lempel-Ziv Variations**

As we have just written, LZ is a growing family of algorithms, with each member
reflecting different design decisions. The main two factors that differ between versions
of LZ-coding are:

1. Whether there is a limit to how far back a pointer can reach.
2. Which substrings within this limit may be the target of a pointer.

The reach of a pointer into earlier text may be unrestricted (growing unbounded
window, the case of our program’s actual implementation), or it may be restricted to a
fixed-size window of the previous N characters, where N is typically several thousand
(32K in the case of the famous gzip).

The choice of substrings can be either unrestricted, or limited to a set of phrases
chosen according to some heuristics. It is important to underline that the combination
of these choices represents some compromise between speed, memory requirements,
and compression. The growing window offers better compression by making more
substring available. However, as the window becomes larger, the encoding may slow
down because of the time taken to search for matching substrings, compression may get
worse because pointers could be larger, and memory requirements increase; moreover,
even if today’s technology offers us high capacity memory benches, we have to consider
the case in which memory could run out: the window may have to be discarded, giving
poor compression until it grows again. A fixed size window certainly solves all these
problems, but has fewer substrings that may be the target of pointers. Moreover,
limiting the size of the window, the size of pointers is smaller.

We will describe in detail some examples in the next chapter.

### 3.2 Statistical Compression

As in [4], compressors in this class could even be called Symbolwise methods. They
achieve compression by generating a code for input symbols based on their estimated
probabilities; the task of estimating probabilities is called prediction. Given these
probabilities, a coding method such as arithmetic coding or Huffman coding can be
used to code the text in a number of bits that is close to the entropy of the probability
distribution. Since arithmetic coding is essentially optimal for this task, attention is focused on the more difficult question of creating good models of message sources. Let’s focus on the Huffman code technique, that represents a classical example. The main idea is this: you have to keep trace of the number of times that a character appears in the input. Next, the more a character appears, the less you have to spend (in terms of number of bit on the output file) for encoding it. This is a very useful technique, widely used in many other compressors. However, refer to next paragraph for a more detailed description. Statistical algorithms fall into three classes: static, semi-adaptive and adaptive, but we are not going to explain them exhaustively.

3.2.1 A Classical Example: Huffman Coding

Huffman is a classical example to give an instance of a statistical compression algorithm. It is widely used in a lot of famous compression algorithms, such as gzip. It could be found in two main variants: static and dynamic.

We take its advantage for encoding a part of our program’s output, selecting runtime, respect to the input file, if it is better using the static version or the dynamic one. We are going to purely describe them in the next two paragraphs.

Static Compression

Huffman coding is a way of constructing codes from a set of message probabilities which gives greater compression. The main operations it executes are the following:

- List all possible messages with their probabilities.
- Locate the two messages with the smallest probabilities.
- Replace these by a single set containing the both, whose probability is the sum of the individual probabilities.
- Repeat until the list contains only one member.

This procedure produces a recursively structured set of sets, each of which contains exactly two members. It can therefore be represented as a binary tree with the original message at the leaves. Then to form the code for any particular message it only needs to traverse the tree from the root to that message, recording 0 for a left branch and 1 for a right branch. The prefix property assures that the resulting code string can be parsed unambiguously into its constituents without the need for any end markers.
This coding scheme gives short codes to high-probability messages and long codes to low probability messages. Moreover, it can be shown that this scheme generates minimum-redundancy codes that produce the shortest possible average code length given the message set’s probability distribution, stated that redundancy is defined as the average code length less the entropy. Let’s take in exam the following example.

- Input string $s$: “abcdadecabedcabdeacebadbceadbad”.  
- Original string length: 95 bits.  
- Compression string length: 85 bits.  
- Compression ratio: 70.48.

You can refer to Figure 3.4 for convincing yourself that the output string will be:

```
1001001101111011111010001001110111001001110111110001001001101111001001111011010001100110111010001.
```

![Figure 3.4: Binary tree on input string $s$](image)

**Dynamic Compression**

It is trivial to perform Huffman coding adaptively by recalculating the coding tree after each change to the model. However, this is inefficient. Recomputing the model
for a $q$-character alphabet takes $q \log q$ operations. This is a high price to pay if the model is updated frequently, such as one for every character transmitted.

The simplest solution is just to recreate the tree from scratch whenever counts are rescaled. But this takes time and because rescaling happens periodically and perhaps at unpredictable intervals, the resulting algorithm could be unsatisfactory for real-time coding. Another method is to use statistics based on a window containing the last $N$ characters only. This can be done by retaining these characters in a circular buffer, incrementing the count associated with a character when it joins the buffer and decrementing when it leaves. However, we won’t take it into deep consideration.
Chapter 4

LZ Examples

In this chapter we are going to explain the most important LZ-based algorithms. In particular, we will focus on LZ77 (Lempel and Ziv, 1977) and some of its principal variations, namely LZR (Rodeh, 1981), LZSS (Bell, 1986), LZB (Bell, 1987), LZH (Brent, 1987).

To sake of completeness we will also describe the other Lempel and Ziv’s main algorithm called LZ78 (Lempel and Ziv, 1978), but we will not describe its variations.

4.1 LZ77

LZ77 was the first form of LZ coding to be published, in 1977. In this scheme pointers denote phrases in a fixed-size window that precedes the coding position. There is a maximum length for substrings that may be replaced by a pointer, given by the parameter $F$ (typically 10 to 20). These restrictions allow LZ77 to be implemented using a “sliding window” of $N$ character. Of these, the first $N - F$ characters have already been encoded and the last $F$ characters constitute a lookahead buffer. For example, if the string “abcabcbababcabc” is being encoded with the parameters $N = 11$ and $F = 4$, and character number 12 is to be encoded next, the window is as in Figure 4.1.

Initially, the first $N - F$ characters of the window are (arbitrarily) spaces, and the first $F$ characters of the text are loaded into the lookahead buffer. To encode the next character, the first $N - F$ characters of the window are searched to find the longest match with the lookahead buffer. The match may overlap the buffer, but obviously cannot be the buffer itself. in Figure 4.1 the longest match for “babc” is “bab”, which starts at character 10.
The longest match is then coded into a triple \(<i, j, a>\), where \(i\) is the offset of the longest match from the lookahead buffer, \(j\) is the length of the match, and \(a\) is the first character that did not match the substring in the window. In the example, the triple would be \(<2, 3, c>\). The window is then shifted right \(j + 1\) characters, ready for another coding step. Attaching the explicit character to each pointer ensures that coding can proceed even if no match is found for the first character of the lookahead buffer. A window of moderate size, typically \(N \leq 8192\) can work well for a variety of texts for the following reasons:

- Common words and fragments of words occur regularly enough in a text to appear more than once in a window. Some English examples are “the”, “of”, “pre-”, “-ing”; while a source program may use keywords such as “while”, “if”, “then”.
- Special words tend to occur in clusters: for example, words in a paragraph on a technical topic, or local identifiers in a procedure of a source program.
- Less common words may be made up of fragments of other words.
- Runs of characters are coded compactly. For example, \(k\) bytes may be coded with only two triples, the first one representing the character, the second one indicating
the number of time that the character has to be written in decompression, i.e. k-1 times.

The amount of memory required for encoding is bounded by the size of the window. The offset (i) in a triple can be represented in \(\lceil \log(N - F) \rceil\) bits, and the number of characters (j) covered by the triple in \(\lceil \log F \rceil\) bits. The time taken at each step is bounded to \(N - F\) substring comparison, which is constant, so the time used for encoding is \(O(n)\) for a text of \(n\) characters. Decoding is very simple and fast. The decoder maintains a window in the same way as the encoder, but instead of searching it for a match it copies the match from the window using the triple given by the encoder.

Lempel and Ziv showed that **LZ77** could give at least as good compression as any semi-adaptive dictionary designed specifically for the string being encoded, if \(N\) is sufficiently large. This result is confirmed by intuition, since a semi-adaptive scheme must include the dictionary with the coded text, while for **LZ77** the dictionary and text are the same thing. The space occupied by an entry in a semi-adaptive dictionary is no less than that consumed by its first (explicit) occurrence in the **LZ77** coded text.

The main disadvantage of **LZ77** is that although each encoding step requires a constant amount of time, the constant can be large, and a straightforward implementation can require up to \((N - F)F\) character comparison for fragment produced. As fragments are typically only a few characters in length, this represents a vast number of comparisons for character coded.

This property of slow encoding and fast decoding is common to many LZ schemes. The encoding speed can be increased using the right data structure, but the amount of memory required also increases. This type of coding is therefore best for situations where a file is to be encoded once (preferably on a fast computer with plenty of memory) and decoded many times, possibly on a small machine. This occurs frequently in practice, for example, on-line help files, manuals, news and electronic books.

The dictionary \(M\) implied by the design of **LZ77**, changes at each coding step. It includes the set of strings in the window with length up to the size of the lookahead buffer, concatenated with the alphabet. The code for each phrase can be viewed in two ways:

1. As an index to the dictionary, it must select one of the \((N - F)Fq\) phrases in \(M\).
2. As a pointer, it has three components, which specify one of the \(N - F\) positions in the window, a length from 0 to \(F - 1\), and a character from the alphabet. the length of the code works out to be the same in both cases.
Formally, the dictionary and length function for \( LZ77 \) are as follows:

\[
M = \{ v | v \text{ begins in the previous } N - F \text{ characters of length } 0 \text{ to } F - 1 \}
\]

\[
L(m) = \lceil \log(N - F) \rceil + \lceil \log F \rceil + \lceil \log q \rceil
\]

### 4.2 LZR

\( LZR \) is the same as the \( LZ77 \) algorithm, except that it allows pointers to denote any position in the already encoded part of the text. This is the same as setting the \( LZ77 \) parameter \( N \) to exceed the size of the input text.

Because the values of \( i \) and \( j \) in the \(< i, j, a >\) triple can grow arbitrarily large, they are represented by a variable-length coding for integers. The method used (\( C_\omega \)) is capable of coding any positive integer, with the length of the code growing logarithmically with the size of the number being represented. For example, the codes for 1, 8 and 16 are, respectively, 0010, 10010000 and 101100000.

There are drawbacks to \( LZR \), principally because the dictionary grows without bound:

1. More and more memory is required as encoding proceeds. Once the memory becomes full either no more of the input can be remembered, or the memory must be cleared and coding started from scratch.

2. The size of the text in which matches are sought increases continually. If a linear search is used, the time taken to code a text of \( n \) characters will be \( O(n^2) \). Data structures have been developed to achieve coding in \( O(n) \) time and \( O(n) \) memory, but other LZ schemes offer similar compression to \( LZR \) for much less effort.

The coding dictionary implied by \( LZR \) at each step includes every substring of the input seen so far. The length of the code for phrase \( m \) is a function of the position of the previous occurrence of the substring \( m \) in the input, which decreases with the distance from the coding position and increases with the length of the substring.
4.3 LZSS

In the next lines we are going to explain the main idea of the Lempel-Ziv-Storer-Szymanski algorithm. As we have just seen, *LZ77* encodes the input file into a series of triples, which can also be viewed as a series of strictly alternating pointers and characters.

The use of the explicit character following every pointer is wasteful in practice because it could often be included as part of the next pointer. LZSS addresses this problem by using a mixture of pointers and characters, the latter being included whenever a pointer would take more space than the characters it codes. A window of $N$ characters is used in the same way as for *LZ77*, so the pointer is fixed. Suppose that a pointer occupies the same space as $p$ encoded characters. The *LZSS* algorithm, in pseudo-code, is the following one.

```
while (lookahead buffer not empty) {
  get a pointer (offset, length) to the longest
  match in the window for the lookahead buffer
  if (length > p) {
    output the pointer (offset, length)
    shift window length characters
  }
  else {
    output first characters in lookahead buffer
    shift window one character
  }
}
```

An extra bit is added to each pointer or character to distinguish between them, and the output is packed to eliminate unused bits. Implementation of *LZSS* encoders and decoders (and other schemes that use a window) can be simplified by numbering the input text character modulo $N$ (see Figure 4.2).

The window is an array of $N$ character. To shift in character number $r$ (modulo $N$), it is simply necessary to overwrite element $r$ of the array, which implicitly shifts out character $r - N$ (modulo $N$). Instead of an offset, the first element of an $(i,j)$ pointer can be a position in the array $(0...N-1)$.

This means that $i$ is capable of indexing substrings that start in the lookahead buffer. The first element of a pointer can be coded in $\lceil \log N \rceil$ bits. As the second
Figure 4.2: Modulo $N$ numbering of the window ($N = 11$)

Element can never have any of the values $0, 1, \ldots, p$, it can be coded in $\lceil \log(F - p) \rceil$.
Including the flag bit to distinguish pointers from characters, a pointer requires $1 + \lceil \log N \rceil + \lceil \log(F - p) \rceil$ bits. Experiments show that using $N = 8192$ and $F = 18$ typically yields good compression for a range of texts, and that the choice of $N$ and $F$ is not too critical.

The dictionary that LZSS uses is:

$$M = A \cup \{ v | v \text{ begins in the previous } N \text{ characters of length } p \text{ to } p + F - 1 \}$$

where:

$$L(m) = \begin{cases} 1 + \lceil \log q \rceil & \text{if } m \in A \\ 1 + \lceil \log N \rceil + \lceil \log(F - p) \rceil & \text{otherwise} \end{cases}$$

4.4 LZB

Studying LZSS algorithm, we can notice that every pointer it uses for representing a phrase has the same size regardless of the length of the substring matched. But it could often happen that some phrase lengths are much more likely to occur than
others, and better compression can be achieved by allowing different-sized pointers. This is exactly the idea of LZB algorithm. It is, in fact, the result of experiments that evaluated a variety of methods for encoding pointers, as well as explicit characters, and the flag that distinguish between them.

It turns out that alternative for the characters and flags do not offer much improvement in compression, but by using a different coding for both components of a pointer, LZB achieves significantly better compression than LZSS and has the added virtue of being less sensitive to the choice of parameters.

The first component of a pointer is the location in the window of the beginning of the match. If fewer than \( N \) characters have been read from the input, it is wasteful to reserve \( \log N \) bits to encode this component. Since the number of bits used is rounded up the next higher integer, it is actually wasteful only if less that \( N/2 \) characters have been read. In particular, if the whole input string contains fewer than \( N/2 \), the full capacity of a pointer is never used. The size starts at 1 bit until there are two characters in the window, then increases to 2 bits until the window contains four characters, and so on, until it is up to full stream with \( N \) characters in the window.

In general, when encoding a substring beginning at character \( n \), this component of the pointer can be transmitted in the minimum of \( \lceil \log n \rceil \) and \( \lceil \log N \rceil \). With this modification, there is no penalty in compression efficiency for choosing \( N \) to be much larger than the number of characters in the input text. LZB uses the variable-length coding scheme \( C_\gamma \) to code the \( j \) component of an \((i, j)\) pointer. If the minimum number of characters represented by a pointer (i.e., minimum value of \( j \)) is \( p \), a match of \( j \) can be coded as \( \gamma(j - p + 1) \). \( C_\gamma \) codes \( j - p + 1 \) in \( 2^\lfloor \log(j - p + 1) \rfloor \).

Since any length of match can be represented by this code, LZB does not impose the limit of \( F \) on the size of a match, although for practical purposes it might be helpful to restrict it to some sufficiently large bound, such as \( N \). The parameter \( F \) is no longer necessary, although a value must be obtained for \( p \), the minimum useful match length. Although the minimum length of a pointer is \( 2 + k \) bits (where \( k = \lceil \log n \rceil \), the current size of the first component of the pointer), suggesting that \( p \) should be \( \lfloor (2 + k)/8 \rfloor \) for 8-bit characters, this is not the case in practice.

For example, if the smallest pointer is 15 bits and characters are 8 bits, it would appear worthwhile to encode two characters as a pointer \((p = 2)\). However, if we set \( p = 3 \) then although character pairs may be coded using one more bit, the code for triples is shorter, and for the distribution of phrase lengths found in practice the latter choice gives slightly better compression.

The best value for \( p \) must be determined by experiment, but experience has shown
that it is around \( [(3 + k)/8] + 1 \) for 8-bit characters, and that this value is not critical. Figure 4.3 shows how LZB coding works for the first 12 characters of the string “ababbbabcaaa...” with \( N = 8 \). For such a small window, \( p \) is always equal to 1. A pointer is represented as \((i, j)\), where \( i \) is the location of the match (coded in binary) and \( j \) is the adjusted length of the match (coded using \( C_\gamma \)). The “match location” and “match length” always specify \( i \) and \( j \), while the output column shows their encoding.

<table>
<thead>
<tr>
<th>Number of characters encoded</th>
<th>Input string and window</th>
<th>Match location</th>
<th>Match length</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ababbbabbc aa...</td>
<td>?</td>
<td>0</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>b c b abbbabbc aa...</td>
<td>?</td>
<td>0</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td>b c b abbbabbc aa...</td>
<td>0</td>
<td>2</td>
<td>&lt;0,1&gt;</td>
</tr>
<tr>
<td>4</td>
<td>b c b abbbabbc aa...</td>
<td>1</td>
<td>4</td>
<td>&lt;01,011&gt;</td>
</tr>
<tr>
<td>8</td>
<td>b c b abbbabbc a...</td>
<td>1</td>
<td>3</td>
<td>&lt;001,010&gt;</td>
</tr>
<tr>
<td>11</td>
<td>a b b b b b b b b a...</td>
<td>?</td>
<td>0</td>
<td>c</td>
</tr>
<tr>
<td>12</td>
<td>a b b b b b b b b b a...</td>
<td>?</td>
<td>0</td>
<td>c</td>
</tr>
</tbody>
</table>

Figure 4.3: LZB coding for string “ababbbabcaaaa”

The dictionary that LZB is using after coding \( n \) character has:

\[
M = A \cup \{ v \mid v \text{ begins in the previous } N \text{ characters of more than } p \text{ characters} \}
\]

\[
L(m) = \begin{cases} 
1 + \lceil \log q \rceil & \text{if } m \in A \\
1 + \lceil \log \min(n, N) \rceil + 2\lceil \log m \rceil + 1 & \text{otherwise}
\end{cases}
\]

\( C_\gamma \) and \( C_{\gamma'} \)

\( C_{\gamma'} \) is a unary code for the number of bits in the binary coding of the integer, followed by the binary coding of the integer with the most significant bit removed. We can say that \( C_{\gamma'} \) can be viewed as \( \lceil \log i \rceil \) zeros followed by a 1, and then \( \beta(i) \).
$C_{\gamma}$ is a rearrangement of $C_{\gamma'}$, with each of the $\lceil \log i \rceil$ zeros followed by a bit from $\beta(i)$, and ending with a 1. The only difference between $C_{\gamma}$ and $C_{\gamma'}$ is that the first one is easier to implement. You can refer to Table 4.1 for a detailed example of this encoding technique.

<table>
<thead>
<tr>
<th>$i$</th>
<th>$\beta(i)$</th>
<th>$C_{\gamma'}(i)$</th>
<th>$C_{\gamma}(i)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>01:0</td>
<td>001</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>01:1</td>
<td>001</td>
</tr>
<tr>
<td>4</td>
<td>00</td>
<td>001:00</td>
<td>00001</td>
</tr>
<tr>
<td>5</td>
<td>01</td>
<td>001:01</td>
<td>00011</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>001:10</td>
<td>01001</td>
</tr>
<tr>
<td>7</td>
<td>11</td>
<td>001:11</td>
<td>01011</td>
</tr>
<tr>
<td>8</td>
<td>000</td>
<td>0001:000</td>
<td>0000001</td>
</tr>
<tr>
<td>9</td>
<td>001</td>
<td>0001:001</td>
<td>0000011</td>
</tr>
<tr>
<td>12</td>
<td>100</td>
<td>0001:100</td>
<td>010001</td>
</tr>
<tr>
<td>16</td>
<td>0000</td>
<td>00001:0000</td>
<td>000000001</td>
</tr>
<tr>
<td>31</td>
<td>1111</td>
<td>00001:1111</td>
<td>010101011</td>
</tr>
<tr>
<td>32</td>
<td>000000</td>
<td>000001:00000</td>
<td>00000000001</td>
</tr>
<tr>
<td>64</td>
<td>0000000</td>
<td>00000001:000000</td>
<td>00000000000001</td>
</tr>
<tr>
<td>128</td>
<td>00000000</td>
<td>0000000001:000000</td>
<td>0000000000000001</td>
</tr>
<tr>
<td>256</td>
<td>000000000</td>
<td>000000000001:00000</td>
<td>000000000000000001</td>
</tr>
</tbody>
</table>

Table 4.1: Encoding examples with $C_{\gamma}$ and $C_{\gamma'}$

**Encoding Commas**

Throughout this paragraph it will be useful to identify non-negative integers with character strings. The following two functions will be used to perform such identifications:

- $INT_\Sigma(s)$: let $\Sigma = \{a_0, ..., a_{k-1}\}$ be an alphabet. Given a string $s$ over $\Sigma$, $INT_\Sigma(s)$ denotes the unique integer that is represented by $s$ in base $k$ notation, assuming that $a_0$ plays the role of 0, $a_1$ the role of 1, etc. When $\Sigma$ is understood, we may drop the subscript $\Sigma$ from $INT$.

- $STR_\Sigma(i)$: given an alphabet $\Sigma$ and a non-negative integer $i$, $STR_\Sigma(i)$ denotes $a_0$ if $i$ is 0, otherwise it denotes the unique string $s$ such that $i = INT_\Sigma(s)$ and the first character of $s$ is not $a_0$. When $\Sigma$ is understood, we may drop the subscript $\Sigma$ from $STR$. In addition, $BIN(i)$ is an abbreviation $STR_{\{0,1\}}(i)$ (the binary representation of $i$).
Note that when discussing string, string concatenation will be denoted by simply writing the two strings together, that is, if $s$ and $t$ are strings, $st$ denotes the string consisting of $s$ followed by $t$.

Given a code $f$ from a set $S$ to an alphabet $\Sigma = \{a_0, \ldots, a_{k-1}\}$, one way to view the problem of decoding is as the process of parsing a sequence of characters over $\Sigma$; that is, start with a string of characters over $\Sigma$ (that has been produced by $f$) and insert “commas” to delimit the codeword boundaries. One way to guarantee that a code is uniquely decipherable is to explicitly include the commas in the code. But, we are going to show a construction technique for converting any code to a uniquely decipherable code by a process that is tantamount to attaching commas to codewords. This construction works even if the size of the set $S$ is not known or is infinite and no bound on the length of codewords produced by $f$ is known.

Before proceeding to the main construction, it is worth noting that there are at least three brute force approaches to "re-writing" a codeword $w$ (using only that character of $\Sigma$) to effectively add a comma to $w$. but, we are going to explain the encoding commas technique.

The idea is to prepend to each codeword a compact description of its length. However, if this description is simply the base $k$ representation of its length, then we are faced with a new instance of the same problem: how do we identify the end of the bit field that contains the number of bits representing the next character? This line of reasoning does however lead to a well known method for handling this problem which we shall henceforth refer to as the cascading length technique, that comprehends the two functions of encode and decode written in pseudo-code in the next two figures.

Moreover, we will use two functions.

- **Concat(s,t)** produces the concatenation of strings $s$ and $t$, returning $st$.
- **Substring(s,i,j)** returns the substring of $s$ included between indexes $i$ and $j$.

Alphabet has to be considered as an array of characters where each element is a member of the alphabet.

### Figure 4.1: Cascade Encoding

```plaintext
Cascade_encode(alphabet a, string s) {
    string t;
    s = concat(a[1], s);
    t = s;
}
```
while (length(s) > 1) {
    s = STR(length(s) - 2);
    t = concat(s, t);
}
return (concat(t, a[0]));

Figure 4.2: Cascade Decoding

Cascade_decode(alphabet a, string s) {
    t = s[0];
    s++;
    while (s != a[0]) {
        i = INT(t) + 2;
        t = substring(s, 0, i);
        s = s + i;
    }
    t++;
    return t;
}

However, not even the decode function presents particular difficulties. It takes in input the alphabet and the encoded string. The result of the execution of lines 2 and 3 makes sure that we have in t the first character of s, while, instead, the same character has been deleted from s. In line 4 there is a loop in which almost the whole decoding phase takes place. The result of INT function’s call is stored in i after adding 2 to the returning value of STR function’s call. As the length s becomes 1, the loop terminates and only two instructions have to be executed. In these latter the first character of t is deleted and t is returned.
Executing the algorithm still described on a binary string of length $n$, this is converted in a string that, counting from right to left, has a structure as follows:

$$1 + (n + 1) + [\lg_k(n - 1)] + [\lg_k(\lg_k(n - 1) - 2)] + \ldots + 1 = n + O(\lg_k(n))$$

Moreover, it is important to point out some observations. In fact, for short strings, the overhead caused by this algorithm becomes so unacceptable to suggest not to use this technique.

For example, in Table 4.2 there are some numeric examples. In the column encoded string some commas have been added for simplifying reader’s distinction of the different fields. In the column relationship between lengths the percentage is approximated to the superior next integer. This is valid for the Table 4.3 too.

<table>
<thead>
<tr>
<th>string $s$</th>
<th>encoded string</th>
<th>relationship between lengths</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0,1,1,0</td>
<td>400%</td>
</tr>
<tr>
<td>00</td>
<td>1,1,00,0</td>
<td>250%</td>
</tr>
<tr>
<td>1000</td>
<td>0,11,1,1000,0</td>
<td>225%</td>
</tr>
<tr>
<td>000000</td>
<td>1,101,1,000000,0</td>
<td>200%</td>
</tr>
</tbody>
</table>

Table 4.2: Encoding examples on short strings

It is evident that there is really a lot of lost space. But, for long strings, results are to be considered very good (see Table 4.3).

<table>
<thead>
<tr>
<th>string $s$</th>
<th>encoded string</th>
<th>relationship between lengths</th>
</tr>
</thead>
<tbody>
<tr>
<td>01350</td>
<td>1,110,10010110,1,10'1350',0</td>
<td>109%</td>
</tr>
<tr>
<td>01025</td>
<td>0,10,1000,10'10''25',0</td>
<td>102%</td>
</tr>
<tr>
<td>00248</td>
<td>0,10,1000,11',1,02048,0</td>
<td>101%</td>
</tr>
<tr>
<td>0131072</td>
<td>0,11,1111,11',1,0131072',0</td>
<td>100%</td>
</tr>
</tbody>
</table>

Table 4.3: Encoding examples on long strings

As input’s size increases, relationship between this last and encoded string is 1.

$$\lim_{\text{length}(\text{input}) \to \infty} \frac{\text{length}(\text{input})}{\text{length}(\text{codificata})} = 1$$

In fact in Table 4.3, in its last entry, the next integer from the real value is 100. It certifies an almost optimal encoding.
4.5 LZH

*LZB* uses some simple codes to represent pointers, but the best representation of pointers can only be determined from their probability distributions using arithmetic or Huffman coding.

It turns out to be difficult to improve compression by applying one of these statistical coders to *LZ* pointers because of the cost of transmitting the codes, and, in addition, the resulting scheme has not the same speed and simplicity of *LZ* coding.

*LZH* is one of these schemes that combine the Lempel-Ziv and Huffman techniques. Coding is performed in two passes:

1. The first is essentially the same as *LZSS*.
2. The second uses statistics measured in the first step to code pointers and explicit characters using Huffman coding.

This means that for *LZH*, the dictionary $M$ is the same as that of *LZSS*, but the length function $L(m)$ is determined by Huffman’s algorithm.

The difficulty is that the encoder must transmit the Huffman codes to the decoder, but there could be up to $N + F + q$ codes, typically several thousands.

*LZH* avoids having a large code table by splitting the first component of pointers into two numbers (with values now ranging to the square root of $N$) and then using a single code table with only a few hundred entries for these two codes and the length and character codes. Considerably less space is needed for the code table, but the codes are now only approximations to the optimum since they are influenced by four different probability distributions. The transmission of the code tables could be eliminated altogether with an adaptive Huffman or arithmetic code, but it is difficult to do this work efficiently because the large range of values accentuates the zero-frequency problem.

4.6 LZMA

Lempel Ziv Marcov chain Algorithm is default and general compression method of 7-zip format. This is because it has a lot of advantages that llow 7-zip to achieve very good results.

- Variable dictionary size (up to 1 GB)
- Estimated compressing speed: about 1 MB/s on 1 GHz CPU
• Estimated decompressing speed (declared):
  - 8-12 MB/s on 1 GHz Intel Pentium 3 or AMD Athlon
  - 500-1000 KB/s on 100 MHz ARM, MIPS, PowerPC or other simple RISC

• Small memory requirements for decompressing (8-32 KB + DictionarySize)

• Small code size for decompressing: 2-8 KB (depending from speed optimizations)

LZMA decoder uses only integer operations and can be implemented in any modern 32-bit CPU (or on 16-bit CPU with some conditions).

Some critical operations that affect to speed of LZMA decompression:

1. 32*16 bit integer multiply
2. Misspredicted branches (penalty mostly depends from pipeline length)
3. 32-bit shift and arithmetic operations

Speed of LZMA decompressing mostly depends from CPU speed. Memory speed has no big meaning. But if CPU has small data cache, overall weight of memory speed will slightly increase.

4.7 LZ78

LZ78 is a fresh approach to adaptive dictionary compression and is important from both theoretical and practical points of view.

Instead of allowing pointers to reference any string that has appeared previously, the text seen so far is parsed into phrases, where each phrase is the longest matching phrase seen previously plus one character. Each phrase is encoded as an index to its prefix, plus the extra character. The new phrase is then added to the list of phrases that may be referenced. For example, the input “aaabbabaababab” is divided into seven phrases. Each one is coded as a phrase that has occurred previously, followed by an explicit character. For instance, the last three characters are coded as phrase number 4 (“ba”) followed by the character “b”. Phrase number 0 is the empty string. The complete example is shown in Table 4.4.

There is no restriction on how far back a pointer may reach (i.e. no window), so more and more phrases are stored as encoding proceeds. To allow an arbitrarily large number of them, the size of a pointer grows as more the input text is parsed. When \( p \) phrases have been parsed, a pointer is represented in \( \lceil \log p \rceil \) bits. In practice
the dictionary cannot continue to grow indefinitely. When the available memory is exhausted, it is simply cleared, and coding continues as if starting on a new text.

An attractive practical feature of LZ78 is that the searching can be implemented efficiently by inserting each phrase in a trie data structure. A trie is a multiway tree where the node associated with each phrase is the one at the end of the path from the root described by the phrase. Each node in the trie contains the number of the phrase it represents. The process of inserting a new phrase will yield the longest phrase previously seen, so for each input character the encoder must traverse just one arc down the trie. Figure 4.4 shows the data structure generated while parsing the string from the last example shown.

Table 4.4: LZ78: Complete example on Input Text “aaabbabaabaabab”

<table>
<thead>
<tr>
<th>input:</th>
<th>a</th>
<th>aa</th>
<th>b</th>
<th>ba</th>
<th>baa</th>
<th>baaa</th>
<th>bab</th>
</tr>
</thead>
<tbody>
<tr>
<td>phrase number:</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>output:</td>
<td>(0,a)</td>
<td>(1,a)</td>
<td>(0,b)</td>
<td>(3,a)</td>
<td>(4,a)</td>
<td>(5,a)</td>
<td>(4,b)</td>
</tr>
</tbody>
</table>

Figure 4.4: Trie data structure for LZ78 coding on input aaabbabaabaabab

The last phrase to be inserted was “bab”, and this process has identified node 4 as the longest match and caused the creation of node 7.

An important theoretical property of LZ78 is that when the input text is generated
by a stationary, ergodic source, compression is asymptotically optimal as the size of the
input increases. That is, \( LZ78 \) will code an indefinitely long string in the minimum size
dictated by the entropy of the source. Very few coding methods enjoy this property.

A source is ergodic if any sequence it produces becomes entirely representative of
the source as its length grows longer and longer. Since this is a fairly mild assumption,
it would appear that \( LZ78 \) is the best solution to the text compression problem, but
we have to notice that optimality occurs as the size of the input tends to infinity,
and most texts are considerably shorter than this. It relies on the size of the explicit
character being significantly less than the size of the phrase code. Since the former
is about 8 bits, it will be consuming 20\% of the output when \( 2^{40} \) phrases have been
constructed. Even if a continuous input were available, we would run out of memory
long before compression became optimal.

The real issue is how fast \( LZ78 \) converges toward this limit. In practice convergence
is relatively slow, and performance is comparable to that of \( LZ77 \). The reason why LZ
techniques enjoy so much popularity in practice is not because they are asymptotically
optimal, but because some variants have a very efficient implementation. For \( LZ78 \)
the dictionary has

\[
M = \{ \text{all phrases parsed so far} \} \\
L(m) = \lceil \log M \rceil + \lceil \log q \rceil
\]
Summary of the Techniques Described in this Chapter

*LZ77*
- Pointers and characters alternate.
- Pointers indicate a substring in the previous $N$ characters.

*LZR*
- Pointers and characters alternate.
- Pointers indicate a substring anywhere in the previous characters.

*LZSS*
- Pointers and characters are distinguished by a flag bit.
- An index is formed if and only if its length is at least equal to $p$.
- Pointers indicate a substring in the previous $N$ characters.

*LZB*
- Same as *LZSS*, except a different coding (based on $C_\gamma$) is used for pointers.

*LZH*
- The dictionary $M$ is the same as that of *LZSS*, but the length function $L(m)$ is determined by Huffman’s algorithm.

*LZ78*
- Pointers and characters alternate.
- Pointers indicate a previous parsed substring.
Chapter 5

Arithmetic Coding

5.1 A Brief Introduction

The aim of this chapter is to show a brief description of the main concepts that characterize the arithmetic coding. Arithmetic coding is today widely used, because of the optimal results it obtains. But, there are also some problems. In fact, compression time is generally quite high, but the great problem is connected to the fact that the decompression phase requires the same time of the compression time. This is probably the main reason thanks to which programs LZ77 based, like WinZip or GZip, are still widely used although they don’t have the compression ratio of an arithmetic coding based program, like WinRar. In tables 5.1 and 5.2 it is possible to notice what just written. The test is performed on four files (of some MB, in way to let the difference be clear), and compression and decompression time for Rar and GZip is shown. The parameter given to the two programs make them perform the best possible compression. It is interesting to see the difference of the two decompression times, especially respect to the last file, of size 60MB. It is easy to understand that an archive of some tens of GB, compressed with Rar, would require a great amount of time for decompression. The reason why GZip has a lower time of compression is due to the option “-9”, anyway, with the default option “-6”, the compression is lightly worst, while the compression time goes down to, about, the 20%.

An arithmetic coding does not strictly require that the symbols of the alphabet have to be translated in an integer number of bits, and this is one of the reasons why it allows to obtain a more efficient coding.

Actually, an arithmetic coding is able to reach the theoretical limit of the entropy, in other words, it is able to compress efficiently any i.i.d. input source. Coupled
with a statistical model of any source, it reaches the theoretical limit of the source, provided that the statistical model correctly correspond to the source. Most of viable best compressor (PPM*, 7zip, Durilca4linus) in terms of compression ratio couple a statistical model with an arithmetic.

It is possible to encode even if all the messages have not been listed, on the contrary, for example, of what static Huffman coding requires. But, in general, programming an arithmetic coding is certainly much more difficult than programming a Huffman coding.

In an arithmetic coding a message is represented by a range of real numbers included between 0 and 1. As the size of the message grows, a greater number of bits is needed for representing it. Moreover, for each step, the reduction of the range for a given symbol of the alphabet is inversely proportional to the probability that the same symbol has to appear. Following, it is treated, a static model.

### 5.2 Starting Static Model

Before that any symbol is transmitted, the range of the message is included between 0 and 1. While the program runs, this range gets smaller after each step of coding. For example, let the alphabet be composed by \{a, e, i, o, u, !\}, and let the model be the one in table 5.2.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Probability</th>
<th>range</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0.2</td>
<td>[0, 0.2]</td>
</tr>
<tr>
<td>e</td>
<td>0.3</td>
<td>[0.2, 0.5]</td>
</tr>
<tr>
<td>i</td>
<td>0.1</td>
<td>[0.5, 0.6]</td>
</tr>
<tr>
<td>o</td>
<td>0.2</td>
<td>[0.6, 0.8]</td>
</tr>
<tr>
<td>u</td>
<td>0.1</td>
<td>[0.8, 0.9]</td>
</tr>
<tr>
<td>!</td>
<td>0.1</td>
<td>[0.9, 1.0]</td>
</tr>
</tbody>
</table>

Table 5.2: Static Model for Arithmetic Coding
5.3 Encoding and Decoding a Generic Source

In this section it is shown a possible pseudo code for coding and decoding phase of a generic arithmetic encoder.

5.3.1 Encoding Phase

Let $eaii!$ be the input message. At the beginning, the range is fixed in $[0, 1)$. After the first symbol, $e$, the encoder sets the new interval in $[0, 0.5)$. The second symbol, $a$, will generate a new range, restricting the range of $\frac{1}{5}$. This action will produce $[0.2, 0.26)$, because previous range was 0.3 and his $\frac{1}{5}$ is 0.06.

Going on with this procedure, the output will be the one shown in table 5.3. Anyway, this algorithm will be described in a best way in the next sections.

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varepsilon$</td>
<td>(0, 1)</td>
</tr>
<tr>
<td>$e$</td>
<td>(0.2, 0.5)</td>
</tr>
<tr>
<td>$a$</td>
<td>(0.2, 0.26)</td>
</tr>
<tr>
<td>$i$</td>
<td>(0.23, 0.236)</td>
</tr>
<tr>
<td>$i$</td>
<td>(0.233, 0.2336)</td>
</tr>
<tr>
<td>$!$</td>
<td>(0.23354, 0.2336)</td>
</tr>
</tbody>
</table>

Table 5.3: Output progressive on message $eaii!$

5.3.2 Decoding Phase

It is clear that the only information given to the decoder is the final range obtained by the coding phase, in the example $[0.23354, 0.2336)$. It is possible immediately to deduce that the first character is $e$, because the range is entirely included in the space of probability of table 5.2.

Now, it is possible to simulate the operation previously executed by the encoder. Initially, the range is $[0, 1)$. After deducing the first character, $e$, the range is reduce to $[0.2, 0.5)$. So, it is possible to decode unambiguously the second character of the original string: $a$. Thus, it follows the restriction of the range to $[0.2, 0.26)$.

Going on in this way, by intuition it is possible to grant that the original string is obtained.

An observation: the output of the encoding phase could be any number included between the range computed at the end of the input source. In this example it is used the whole range, but it isn’t really necessary. Moreover, another problem arises, that is...
how to let the decoder understand that the end of the file is reached. In the example, a special character is used: !. Anyway, this is not a reasonable technique that can be used in reality. Another way to solve the problem is represented by the possibility of writing in the encoded string the size of the input source.

The entropy related to table 5.2 is

$$- \log 0.3 - \log 0.2 - \log 0.1 - \log 0.1 = - \log 0.00006 \approx 4.22$$

Five decimal numbers have been used for encoding the string, there is no gain. But, first of all no bit compression is performed. But this is a banal observation, while, instead it is interested to notice that a probability distribution like \( \{e(0.2), a(0.2), i(0.4), !(0.2)\} \), would have caused an entropy equal to 2.89. So, different starting models lead to a different behavior of the encoder.

### 5.4 Pseudo Code for an Arithmetic Coding

In this section it is shown the pseudo-code of encoding and decoding function.

Symbols are denoted with natural numbers greater than 0. Array \( \text{Prob} \) keeps trace of the probability that a symbol of the alphabet has to appear. It is assumed that a simple preprocessing function calculates the distribution probability of the symbols of the alphabet. The values of the limits of the range related to each symbol are stored in the two arrays \( \text{range1} \) and \( \text{range2} \).

#### 5.4.1 Encoding Function

The function has to be called for each symbol of the input source.

\[
\text{EncodeSymbol(Symbol, Prob)}\{
range = \text{high} - \text{low};
\]

\[
\text{high} = \text{low} + \text{range} \times \text{range2}[\text{symbol}];
\]

\[
\text{low} = \text{low} + \text{range} \times \text{range1}[\text{symbol}];
\]

\}
5.4.2 Decoding function

Following the pseudo code for a decoding function. The input is composed by

- the distribution of probability related to the symbols of the alphabet of the input source, that has been written by the encoder in the output file
- obviously, the range obtained by the execution of the encoder. It is worth to remember that any number included in this range was good.

\[
\text{DecodeSymbol}(\text{Prob}, \text{input}) \{
\]

find a symbol such that 
\[
\text{range1}[\text{symbol}] \leq (\text{input} - \text{low})/(\text{high} - \text{low}) \leq \text{range1}[\text{symbo2}];
\]
\[
\text{range} = \text{high} - \text{low};
\]
\[
\text{high} = \text{low} + \text{range} \times \text{range2}[\text{symbol}];
\]
\[
\text{low} = \text{low} + \text{range} \times \text{range1}[\text{symbol}];
\]
\[
\text{return symbol};
\]

Pseudo code is always so simple! There are a lot of problems that arise when programming the arithmetic coding. The more difficult to solve is bit writing. In fact, if bits are not treated in an efficient way, first of all the size of the final file will be not so good as it would be with a optimal bit writing. Moreover, also the precision of a long double type will not soon be enough for a source for some tens of bytes, in particular if the alphabet contains a lot of symbols. Thus, it is useful to treat incremental transmission and receiving.

5.5 Incremental Transmission and Receiving

Main Idea

In the example shown in table 5.3 it is clear that it is possible to decode unambiguously the message throughout, for example, the number 0.23355. After decoding the second character, the range has been restricted to [0.2, 0.26]. Given that the final code will be certainly included between those two values, it is possible to grant that it will start with 0.2. By the same way, after the third character, it is possible to be sure that it will begin with 0.23, and, after the fourth character, with 0.233. Thanks to this
concept, it is possible to perform the incremental transmission. That is, each cipher is transmitted at the same moment in which its value is known. Decoder too can work with this new technique.

**Example**

For example once that starting value 0.23 is given, the decoder can restrict the range enough to allow to determine the first tree character. If the encoding is incremental, it can be implemented using arithmetic with finite precision, because, as a cipher is transmitted, it will no more influence next computations.

For example, if value 0.23 of range [0.23, 0.236), has been transmitted, next output won’t be influenced if the range will be modified in [0.000, 0.006) or in [0.0, 0.6). The most important consequence of this technique is due to the fact that, using it, it is possible to work with arithmetic with finite precision

**5.5.1 Pseudo Code**

First of all, some introductive notes.

- *high* and *low*, that appear also in previous example, will be represented with an integer number, and not with real number.

- array *CumFreq* keeps trace of the probability of the symbols of the alphabet.

- Probability of symbol *i* is given by *CumFreq[i] / CumFreq[0]*.

As the range becomes stricter, bits of *low* and *high* tend to be equal. When it happens, it is immediately possible to transmit them, because they won’t be no more modified going on with the coding.

**Coding Pseudo Code**

Given that, certainly, it is always true that *low ≤ high*, what follows could be a good pseudo code for the encoding phase.

```plaintext
while high < half or low ≥ half do;

    if high < half then
        OutputBit(0);
        low = 2 * low;
```
\[
high = 2 \times high + 1;
\]

\[
if \ low \geq half
\]

\[
OutputBit(1);
\]

\[
low = 2 \times (low - half);
\]

\[
high = 2 \times (high - half) + 1;
\]

**Decoding Pseudo Code**

\[
while \ high < half \ or \ low \geq half \ do
\]

\[
if \ high < half
\]

\[
value = 2 \times value + InputBit();
\]

\[
low = 2 \times low;
\]

\[
high = 2 \times high + 1;
\]

\[
if \ low \geq half
\]

\[
value = 2 \times (value - half) + InputBit();
\]

\[
low = 2 \times (low - half);
\]

\[
high = 2 \times (high - half) + 1;
\]

### 5.6 Simple Implementation and Example

At the end of this work it is possible to read the C-Code written with the aim of showing in practice how does arithmetic work. In this section they are reported some screenshots, on a simple input string. In particular, thanks to this simple program, it is possible, for example, to observe an execution on a generic string, as following.

**Example on a Simple Binary String**

This subsection shows an example of execution on string \textit{abbababbababbababab}.

At the begininning, the program calculates, for each symbol of the alphabet of the input file

- probability to appear
- range
- number of occurrences

as follows:

a, p:0.4210526315, [0.000000000, 0.4210526315], 8
b, p:0.5789473684, [0.4210526315, 1.000000000], 11

Next, in correspondence of each input character, respect to the space of probability still calculated, range is restricted, and a possible final result is shown, as follows.

a 0.00000000000000000000 0.42105263157894734505
b 0.17728531855955675933 0.42105263157894734505
b 0.27992418719930017845 0.42105263157894734505
a 0.27992418719930017845 0.33934669009599371936
b 0.30494418841896059336 0.33934669009599371936
a 0.30494418841896059336 0.31942945228297453530
b 0.31104324688801909815 0.31942945228297453530
b 0.31457428073852666817 0.31942945228297453530
a 0.31457428073852666817 0.31661856349408368239
b 0.31543503137244538470 0.31661856349408368239
a 0.31543503137244538470 0.31593336068681943107
b 0.31564485424165550365 0.31593336068681943107
b 0.31576633063961928594 0.31593336068681943107
a 0.31576633063961928594 0.31583665908054564531
b 0.31579594261474619454 0.31583665908054564531
b 0.31579594261474619454 0.31581308638981964165
b 0.31580316104635608454 0.31581308638981964165
a 0.31580316104635608454 0.31580734013834071972
b 0.31580492066403381690 0.31580734013834071972

A possible final result: 0.31580492066403381690.

It is interested to observe that there are some underlined ciphers. They are underlined with the aim to show that there are some ciphers that from the moment in which they are underlined they won’t no more influence the computation, and, thus, they could be transmitted to the output file. As written previously, this is very important because it is possible to perform a generic compression with finite arithmetic precision.
Part II

Theoretical Contribution
Chapter 6

Dictionary-Symbolwise algorithms

6.1 Dictionary and Symbolwise Compression Algorithms

A dictionary compression algorithm, as authors of [4] point out, achieve compression by replacing consecutive groups of symbols (phrases) with a code. Such code can be regarded as a dictionary pointer, i.e. index into a dictionary. Dictionaries are lists of frequently used phrases and range from an ad-hoc collection of letter pairs to the adaptive dictionaries used for Lempel-Ziv coding (LZ77 [52] and LZ78 [53]). In particular, for LZ77 schemes, the dictionary is a part of the previously parsed text and sometimes it is called “history”. A dictionary compression algorithm can be fully described by the following characteristics: (1) the dictionary description, i.e. a complete algorithmic description on how the compression dictionary is built and updated, (2) the encoding of dictionary pointers in the compressed data, (3) the parsing method, i.e. the algorithm that splits the uncompressed data in dictionary phrases.

Generally, dictionary compression algorithms use a greedy parsing which, at any point, chooses the longest matching element from the dictionary that starts in the current position. Hence, most dictionary compression algorithms concentrate on the dictionary description and on the pointer encoding, tacitly assuming that a greedy parsing is applied.

As an example, a simple dictionary compression algorithm with fixed dictionary $D = \langle ab, cbb, ca, bcb, abc \rangle$ could parse the text $abccacbbabbcbcb$ as $abc/ca/cbb/ab/bcb/cbb$ and encode it as 532142.

A symbolwise compression algorithm (sometimes referred as statistical compression
algorithm) replaces the letters of the text to be compressed with a code that can
depend, e.g., on the letter itself, on its position or on its context. Obviously, in this case
the compression is achieved by choosing appropriately the codes so that, for example,
the most frequent letters have a smaller code. As T. C. Bell and I.H. Witten say in
[4], given the estimated probabilities of symbols ”a coding method such as arithmetic
coding or Huffman coding can be used to code the text in a number of bits that is
close to the entropy of the probability distribution”.

A symbolwise compressor can be described just by giving the symbolwise encoding
algorithm, that is the algorithm that decides the code to assign to each letter. For
instance, such kind of compressor can decide the static code assignment $a = 0$, $b = 10$,
$c = 111$ and encode the text $abccacbbabbcbbb$ as 01011111001001011110111110.

6.1.1 Dictionary-Symbolwise Algorithms and Schemes

In this short subsection we give the main definitions of this thesis.

A dictionary-symbolwise algorithm uses both a dictionary and a symbolwise com-
pression. Such compressors parse the text as a free mixture of dictionary phrases and
literal characters, which are substituted by the corresponding pointers or literal codes,
respectively. Therefore, the description of a dictionary-symbolwise algorithm should
also include the so called flag information, that is the technique used to distinguish
the actual compression method (dictionary or symbolwise) used for each segment of
the parsed text. Often, as in the case of LZSS [46], an extra bit is added to each
pointer or encoded character to distinguish between them. For instance, a dictionary-
symbolwise compressor algorithm with a fixed dictionary $D = \langle ab, cbb, ca, bc, abc \rangle$
and the static symbolwise code assignment $a = 1$, $b = 2$, $c = 3$ could compress the text
$abccacbbabbcbbb$ as $F1C3F3F2F1F4F2$, where $F$ is the information flag for dictionary
pointers and $C$ is the information flag for the symbolwise code.

More formally, a parsing of a text $T$ in a dictionary-symbolwise algorithm is a pair
$\langle \text{parse}, Fl \rangle$ where parse is a sequence $(u_1, \cdots, u_s)$ of words such that $T = u_1 \cdots u_s$ and
where $Fl$ is a boolean function that, for $i = 1, \ldots, s$ indicates whether the word $u_i$ has
to be coded as a dictionary pointer or as a symbol. Obviously, if $u_i$ has to be coded as
a symbol, then it has length equal to one. Note that a single-character word may be
also coded as a pointer, but this is usually not convenient. In dictionary-symbolwise
compressors, the parsing algorithm is generally more complex to implement, since it
must be able to decide which combination of symbolwise encodings and dictionary
pointers gives the best compression.
To sum up, a dictionary-symbolwise compression algorithm is specified by: (1) the dictionary description, (2) the encoding of dictionary pointers, (3) the symbolwise encoding algorithm, (4) the encoding of the flag information, (5) the parsing method.

We call *dictionary-symbolwise scheme* a set of algorithms having in common the same first four specifics (i.e., the parsing method is not described). We notice that any of the specifics from 1 to 5 above can depend on all others, i.e. they can be mutually interdependent. This fact, however, does not affect the definition of a dictionary-symbolwise scheme: whenever the specifics of the parsing method are given, exactly one algorithm is completely described.

The word *scheme* has been used by other authors with other related meaning. For us the meaning is rigorous, i.e. it is a set of algorithms.

Let now define three classes of dictionary-symbolwise schemes.

1. The first class includes all schemes containing algorithms where for any text $T$ the dictionary and the encoding of symbols, dictionary pointers and flag information do not depend on the specific parsing of the text $T$.

   For instance, a scheme in this class can contain all algorithms which use a given static dictionary, a fixed encoding of dictionary pointers and flag information, and where, for any position $n$, the encoding of the $n$-th symbol depends only on the symbol itself and on the previous one, if it exists (no matter if such previous symbol is part of a parsed dictionary word).

   As another example, a scheme in this class can contain all algorithms as in previous example but such that the encoding of symbols is done by a static Huffman algorithm on all the text. Furthermore, schemes in this class can contain algorithms that have dynamic dictionaries. Indeed the dictionary can dynamically depends on the position of the text. For instance a scheme in this class can contain exactly all pure dictionary algorithms that decide that at any position of the text the dictionary is the one that LZ78 would have built up that position. The encoding of dictionary pointers has fixed cost. But the parsing can be different from the LZ78's parsing. Flexible parsing (cf. [36, 34]) belongs to this scheme and indeed it is optimal within this scheme (cf. Definition 1, Proposition 1, Subsection 6.3 and Theorem 2).

2. The second class includes all schemes not belonging to the first class and containing algorithms where for any text $T$ the dictionary and the encoding of symbols,
dictionary pointers and flag information, at any position of the text \( i \), depend on the text up to position \( i \) and only on the parsing up to position \( i \). In other words they depend on the previous portion of the text that has already been processed, or (as in the previous class) do not depend on the specific parsing of the text \( T \).

For instance, an a scheme in this class can contain all algorithms that use a given static dictionary, a fixed encoding of dictionary pointers and flag information and that perform a dynamic Huffman encoding on the literals already generated by the parsing.

3. The third class includes all the remaining dictionary-symbolwise schemes. For instance, a scheme in this class can contain all algorithms that use a given static dictionary, a fixed encoding of dictionary pointers and flag information and that perform a static Huffman encoding on all the literals of the parsing. We notice that we used here the most common theoretical meaning of “static Huffman”, i.e. the Huffman code is calculated starting from the specified text. There exists another practical meaning of these terms, used for instance in \([41, 42, 43]\), where the meaning is that exists a precalculated Huffman code that is used and that does not depend on the text.

Further explanations on these classes will be given when we will describe Schuegraph’s graphs in next section.

Even if other subdivisions can also be done, our choice depends on this particular subdivision that is a natural one and on each scheme-class that presents some peculiar characteristics that will be studied later on.

### 6.2 Optimal Parsing

We start this section with some definitions.

The cost of an encoding is a function \( C \) that associates to any coded text a real number greater than or equal to 0. Typically \( C \) is chosen to be the bit length of the coded text but, in general, it is a real valued function whose domain is the set of all possible strings over the final encoding alphabet. In this work we will also assume that \( C \) is non negative and additive, as it is usually done.

In a dictionary-symbolwise algorithm, function \( C \) is obtained by defining the cost of the encodings for dictionary pointers, symbols and flag information. The cost of an encoded text is the sum of all the costs for the parsed symbols, of all the flag information and of the parsed dictionary pointers.
Definition 1. Fixed a cost function $C$ and a text $T$, a dictionary-symbolwise algorithm is optimal within a set of algorithms if the cost of encoded text is minimal with respect to all other algorithms in the same set.

A parsing is optimal for a text $T$ within a scheme if the unique algorithm that uses this parsing in the scheme is optimal within the same scheme.

Notice that definition above depends on the text $T$. A parsing can be optimal for a text and not for another one. Clearly, we are mainly interested on parsings that are optimal for all texts over an alphabet or for classes of texts. Whenever it is not explicitly written, from now on when we talk about optimal parsing we mean optimal parsing for all texts.

For static dictionary compressions, the problem of finding an optimal parsing is classically reduced to find a shortest path in a graph, as proved by Schuegraf and other authors in [44].

![Classical parsing graph](Image)

More precisely, given the text $T = a_1a_2a_3 \cdots a_n$ of length $n$, a directed, labeled (or weighted) graph $G_T = (V, E, L)$ is defined. The set of vertices is $V = \{1, \ldots, n, n+1\}$, with vertex $i$ corresponding to the character $a_i$ for $i \leq n$, and $n+1$ corresponding to the end of the text. $E$ is the set of directed edges where an ordered pair $(i, j)$, with $i < j$, belongs to $E$ if and only if the corresponding substring of the text, that is the sequence of characters $v = a_i \cdots a_{j-1}$, is a member of the dictionary. The label $L_{i,j}$ is defined for every edge $(i, j) \in E$ as the cost of encoding the pointer to $a_i \cdots a_{j-1}$ for the given encoding method that is supposed to not depend on the parsing. Figure 6.1 shows an example of such graph, where the edge labels have been omitted for clarity.

In this setting, the problem of finding the optimal parsing of $T$, relative to the given static dictionary description and encoding method, reduces to the well-known problem of finding the shortest path in $G_T$ from vertex 1 to vertex $n + 1$ (see [44]). Since $G_T$ contains no cycles and it is already naturally ordered in a topological order, the shortest path can be found in $O(|E|)$ through simple dynamic programming method. In the example of Figure 6.1, if we assume that the edge cost is constant, then the
optimal parsing is given by the bold edges.

Unfortunately, the number of edges in $E$ can be quadratic in the size of the text, thus motivating the search of suboptimal alternatives ([27]) or pruning techniques ([31]) or else, quite recently, techniques that exploit the discreteness of cost functions ([17]).

Generally, the dictionary and the labels of $G_T$ cannot be always precisely defined without knowing the parsing.

For example, consider a LZ77 [52] scheme where the encodings of dictionary pointers are subsequently compressed by a static Huffman algorithm, and where the cost function is the final encoded text length. It is clear that the label of a pointer edge in this case depends on all other pointers in the chosen parsing, so it cannot be defined “a priori” in a way that does not depend on the parse.

Indeed both the definition of optimality of a parsing and the result in [44] is stated for static dictionaries, but some authors sometimes incorrectly skip this hypothesis. This error is supported by the set of edges of $G_T$, that is always well defined, with or without the labels. In this thesis we will extend the definition and the optimality result described in [44] from the static dictionaries case to the first dictionary-symbolwise class schemes. We want here to recall that the first class includes all pure static dictionary cases and also some schemes containing pure dynamic dictionary algorithms such as the example we gave of a scheme containing the flexible parsing. We will also show how to apply these results in some cases of algorithms and schemes belonging to the second and third class. Particularly we will study the GZIP’s case as main example.

We can naturally extend $G_T = (V, E, L)$ to the dictionary-symbolwise case, where for any $i \leq n + 1$ an arc $(i, i + 1)$ is added to the set $E$ to represent the symbolwise compressor. The cost of all arcs (dictionary arcs and symbolwise arcs) must now include the cost of coding the flag information. Moreover, since usually dictionary arcs having length one are not included in $E$, $G_T = (V, E, L)$ is not a multigraph. In the following, we will suppose that $G_T$ is not a multigraph. All the results we will state, naturally and trivially extend to the multigraph case.

Next remark is very important.

Remark 1 1. If an algorithm belongs to a scheme in the first class, the graph $G_T$ can be built before the whole algorithm processes the text, and, moreover graph $G_T$ is the same for all the algorithms belonging to the same scheme. We will say in this case that the graph $G_T$ is associated to the scheme.
2. The labeled graph $G_T$ can be built on-line for algorithms belonging to schemes in the second class. For on-line we mean that, if the algorithm has processed the text up to position $i$, then all arcs outgoing $i$ are well defined together their costs.

3. For algorithms belonging to schemes in the third class, i.e. in the general case, the graph is not necessarily well defined. If the algorithm has processed the text up to position $i$, then all arcs outgoing $i$ are well defined but not necessarily their costs. Moreover, if some arc outgoing $i$ is not used in the parse, perhaps its cost will never be assigned. Therefore in the general case, for some text $T$ and some algorithm $A$ it is possible that the graph $G_T$ is well defined or not defined (i.e. some cost is not defined), not even after that the encoded text is created.

In what follows we will use the first two properties described in previous remark to design some parsing strategies for all schemes in the first and for some schemes in the second class respectively.

We will further extend our results also for some schemes in the third class. For algorithms belonging to schemes in the third class, we will show a technique that will allow us to build a sequence of labeled graphs $G_{T,s}$ $s \in \mathbb{N}$ that experimentally seems to fast converge to a graph which can be considered “analogue” to graph $G_T$.

In next theorem we formalize a result analogous to the one of [44]. The proof is straightforward.

**Theorem 1** Let us suppose to have a text $T = a_1a_2a_3 \cdots a_n$, a function cost $C$ and a scheme $SC$ that belongs to the first class. Then any parsing which is induced by a path of minimal cost from vertex 1 to vertex $n+1$ in the graph $G_T$ associated to the scheme $SC$ is an optimal parsing within the given scheme.

**Remark 2** We want here to notice that previous result is commonly considered to holds only for static pure dictionary compressor (or it is misunderstood). Developing a theory, we can find the largest “environment” for which this simple theorem in the first class. This “environment” includes many dynamic dictionary compressors.

We want also to notice that $G_T$ and all its subgraphs are Directed and Acyclic Graphs (DAG in short) and that they are naturally ordered in a topological order. Therefore it is well known that in these cases there exists an easy and standard linear time (in the size of the graph) algorithm for finding a path of minimal cost.

Next proposition is easy to prove but it is somewhat surprising and shows that the boundaries among classes are less neat than intuitively supposed. It generalizes the
example concerning the flexible parsing, given after the definition of the first class of schemes. Indeed in next section we will show that flexible parsing turns out to be a parsing induced by a path of minimal cost from vertex 1 to vertex \( n+1 \) in the graph \( G_T \) associated to the scheme \( SC_{T,A} \) considered in next proposition and described explicitly in the proof, where \( A \) is the LZ78 algorithm.

**Proposition 1** Let us consider a text \( T \) and any dictionary-symbolwise algorithm \( A \) such that the graph \( G_T \) is well defined after that the algorithm has processed the text. Then a scheme \( SC_{T,A} \) exists in the first class such that the graph of this scheme is \( G_T \).

For any algorithm \( A \) and for any text \( T \) such that its associated graph \( G_T \) is well defined there exists an algorithm \( A'_T \) in \( SC_{T,A} \) that has the same behavior of \( A \) with respect to the text \( T \), i.e. the restrictions of \( A \) and of \( A'_T \) to the text \( T \) have the same behavior.

For any algorithm \( A \) there exists a scheme \( SC_A \) in the first class such that for all texts \( T \) the graph associated to this scheme is \( G_T \) whenever the same graph \( G_T \) is the one associated to \( A \) and is well defined. Moreover there exists an algorithm \( A'_T \) that has the same behavior of \( A \) on all texts \( T \) such that graph \( G_T \) is well defined and that belongs to \( SC_A \).

**Proof 1** Algorithms in \( SC_{T,A} \) firstly simulate \( A \), then build \( G_T \) and use it to describe the dictionary \( D \) and the function cost \( C \) in the following way. At any position of the text \( i \) the dictionary \( D \) is given (and fixed) by the arcs outgoing node \( i \) and the costs are the same as the costs induced by \( A \), that are, in turns, the labels of these arcs. The parsing is not specified. It is easy to see that this set of algorithms forms a scheme in the first class. The graph of this class of algorithm for text \( T \) is \( G_T \). These algorithms are not defined for texts that are different from \( T \). If, among these we choose as an alternative dictionary in position \( i \) the dictionary that \( A \) has in that position. Notice also that the hypothesis of previous theorem are verified for all texts \( T \) whenever \( A \) belongs to a scheme in the second class, because if \( A \) is in the second class then graph \( G_T \) is always
well defined. As another explicit example we can consider $A$ to be the classical LZ78 algorithm. In this case Flexible parsing will be optimal within the scheme described in the proof of above theorem (cf. Theorem 2 and its consequences).

When an algorithm belongs to a scheme in the second class for a given text, graph $G_T$ is well defined but another algorithm in the same scheme for the same text can have a different graph. This motivates next definition.

**Definition 2.** Let us consider an algorithm $A$ and a text $T$ and suppose that graph $G_T$ is well defined. We say that $A$ is locally optimal if its parsing induces a shortest path in $G_T$ from node 1 to node $n + 1$, with $n = |T|$. In this case we say that its parsing is locally optimal.

Notice that the notion of local optimality is a property that depends on the single considered algorithm (and parsing), not on a scheme. We can connect this fact with Proposition 1 and Theorem 1 to obtain the next proposition whose proof is straightforward.

**Proposition 2** Let us consider an algorithm $A$ and a text $T$ and suppose that graph $G_T$ is well defined. Suppose that the parsing of $A$ is locally optimal. Then the parsing of $A$ is (globally) optimal within the scheme $SC_A$, where $SC_A$ is the scheme in the first class defined in the proof of Proposition 1. This contain an algorithm $A'$ that has the same behavior of $A$ over text $T$.

![Greedy Parsing](image1.png) ![Anti-Greedy Parsing](image2.png)

**Figure 6.2:** Locally but not globally optimal parsing

We have simple examples (see Figure 6.2), where a parsing of a text is locally optimal and the corresponding algorithm belongs to a scheme in the second class but it is not (globally) optimal within the same scheme. For instance let us define a pure
dictionary scheme where the dictionary is composed by \(< a, ab >\) if the parsing of processed text has reached an even position (starting from position 0) with costs 10 and 20 respectively. The dictionary is \(< a, b >\) if the parsing of processed text has reached an odd position with costs 5 each. Notice that now the dictionary phrase “a” has a different cost than before. The dictionary and the costs are changing as a function of the parsing and therefore this is a scheme in the second class. Let us now consider the text \( T = ab \). As first parsing let us choose the greedy parsing, that at any reached position chooses the longest match between text and dictionary. The graph \( G_T \) for this greedy algorithm has three nodes, 0, 1, 2, and only two edges, both outgoing 0, one to node 1 and cost 10 and another to node 2 and cost 20. The greedy parsing reaches the end of the text with this second arc which has global cost 20 and then it is locally optimal. As second parsing we choose the anti-greedy that at any reached position chooses the shortest match between text and dictionary. The graph \( G_T \) for this anti-greedy algorithm has three nodes, 0, 1, 2, and three edges, two outgoing 0, one to node 1 and cost 10 and another to node 2 and cost 20 and a third outgoing 1 to node 2 and cost 5. The parsing of the anti-greedy algorithm is \((a)(b)\) with cost 15. Therefore the greedy parsing is locally optimal but not optimal.

Anyhow, we believe that these cases are somehow pathological and that a locally optimal parsing is often a very good approximation of an optimal parsing within schemes in the second class and, in a more complex way, even in the third class. We have experimental evidence of this fact that will be discussed in the section devoted to experiments.

### 6.3 Flexible Parsing

In this section we extend the flexible parsing to all pure dictionary schemes in the first class, where the dictionary is prefix closed and the cost of dictionary pointers is constant. We also prove that the generalized flexible parsing is optimal within the scheme. For pure dictionary scheme in the first class we mean a dictionary-symbolwise scheme in the first class where all algorithms in the scheme are pure dictionary (or almost pure dictionary as noticed in Remark 4). Essentially we obtain a new proof of the main result concerning flexible parsing (cf. [36, 34]) revisited in our formalism.

In our formalism we further prove that the algorithm of [22], which uses flexible dictionary construction, denoted here as FPA analogously as in [34], is locally optimal. Since Proposition 2 can be applied to FPA, it turns out that FPA is optimal within the scheme \( SC_A \) considered in this proposition.
Notice that the author of [22] was looking for a kind of optimality for FPA, while subsequently the authors of [34] say that in [22] the notion of optimality is not clear. In our formalism all problems concerning this point are settled and we can willingly say that the authors of both [22] and [34] were right in some sense.

In the final part of this section we extend the notion of flexible parsing to the dictionary-symbolwise case. We then give a generalization of our previous result to the dictionary-symbolwise case by proving that our generalized flexible parsing is still optimal within a scheme in the first class.

We want to emphasize what we do not consider in this section. Here we do not consider the important problem of finding an efficient data structure to implement flexible parsing in most realistic cases. This problem is studied and settled for the pure dictionary case in [36, 34] and we do not consider this problem for the extension of flexible parsing to the dictionary-symbolwise case. We leave this problem open. Despite this, we made extensive experiments using a linear space implementation, that we are sure can be improved (cf. Subsection 6.3.1).

We start with two easy lemmas that can belong to folklore, but that we were not able to discover in the literature. The first is indeed an immediate consequence of our definitions and the proof is omitted because it is straightforward.

**Lemma 1** Let us consider a pure dictionary (resp. a dictionary-symbolwise) algorithm where the dictionary can dynamically change. Suppose further that the cost of dictionary pointers (resp. and of symbols) is constant. Then for any text $T$ the graph $G_T$ is well defined. Therefore it is also well defined for any node $i$ of the graph its distance $d_T(i)$ from the origin.

**Lemma 2** Let us consider a pure dictionary algorithm (where the dictionary can dynamically change) where the cost of dictionary pointers is constant. If the dictionary is always (at any moment) prefix-closed for the text $T$ then the function $d_T$ is non-decreasing monotone.

**Proof 2** Without loss of generality we can consider that the constant cost of a pointer is 1, and, so, $d_T$ assume integer values. We have just to prove that for any $i > 1$,

$$d_T(i - 1) \leq d_T(i).$$

If $d_T(i) = h$ then there exists an arc in $G_T$ from a vertex $j < i$ to $i$ such that $d_T(j) = h - 1$. If $j = i - 1$ then the thesis follows. If $j < i - 1$ then, since the dictionary is prefix closed, there is an arc in $G_T$ from $j$ to $i - 1$ and, consequently, $d_T(i - 1) \leq h$. 

61
Remark 4 We have used in previous proposition the phrase “pure dictionary algorithm”. We have to be a bit careful using this terminology.

Indeed, LZ78, LZ77 and related algorithms often parse the text with a dictionary pointer and then add a symbol, i.e. the parse phrase is composed by a dictionary pointer and a symbol. In these cases all arcs of $G_T$ denote parse phrases labelled by the cost of the dictionary pointer plus the cost of the symbol. The reader can easily check that previous theorem and Theorem 1 still holds with analogous proofs. From now on in this work, we consider these cases included in the class of “pure dictionary” algorithms and schemes.

In previous lemma we used the hypothesis of constant cost of the pointers just to grant the fact that $G_T$ is defined. If we include this stronger hypothesis in the statement, then previous lemma holds also in the dictionary-symbolwise case. The proof is analogous to the previous one, just a bit more tedious, and it is left to the reader.

Lemma 3 Let us consider an additive cost function, a dictionary-symbolwise algorithm and a text $T$ such that the graph $G_T$ is well defined. If the dictionary is always (at any moment) prefix-closed then the function $d$, that represent the distance of the nodes of $G_T$ from the origin, is non-decreasing monotone.

We now define the flexible parsing. Flexible parsing can be defined starting from a graph $G_T$.

Definition 3 Given a pure dictionary algorithm $A$ and a text $T$ such that $G_T$ is defined, we define the flexible parsing on the graph $G_T$ to be the greedy parsing with a single step lookahead. In other words, at any position of the text, flexible parsing chooses the parse phrase such that the next one will be the longest possible.

Remark 5 In above definition, that is classical, there is a non-determinism. It is possible that there exist two or more prefixes of the text that has to be parsed, that gives the same longest advancement in the next iteration. To avoid this non-determinism in this work, and in particular in the proof of next theorem, we decide to pick in the parse the longest of such prefixes.

Theorem 2 Let us consider a pure dictionary algorithm $A$ and a text $T$ such that $G_T$ is defined. Then flexible parsing represents a path of minimal cost from vertex 1 to vertex $n + 1$ in the graph $G_T$. 
Proof 3  By Lemma 2, supposing the cost of pointers equal to 1, the distance \( d(i) \) from vertex 1 to vertex \( n+1 \) of the text \( T \) is organized in run of equal numbers. That is, vertex 1 has distance 0 from itself. Vertex \( n+1 \) has distance \( k \leq n \) and, in general, for any \( p, 0 < p \leq k \) there exists at least one vertex having distance \( p \) and the set of vertices that have exactly distance \( p \) is a set of consecutive vertices \( i_{p-1}+1, \ldots, i_p \). In our notation \( i_p \) is the greatest vertex that has distance \( p \). Let us call \( x_p \) the greatest vertex that has distance \( p-1 \) and such that there is an arc in the graph \( G_T \) from \( x_p \) to \( i_p \). Since \( x_{p+1} \) has distance \( p \), it is smaller than \( i_p \). Since the dictionary is prefix closed and there is an arc from \( x_p \) to \( i_p \), there is also an arc from \( x_p \) to \( x_{p+1} \). It is obvious that the sequence of vertex \( 0 = x_1, x_2, \ldots, x_k, i_k = n+1 \) represents a path of minimal cost from vertex 1 to vertex \( n+1 \) in the graph \( G_T \). Obviously, thanks to the previous definition and remark, we can notice that this parse is that obtained by the flexible parsing, and this concludes the proof.

Corollary 1  Consider a pure dictionary scheme in the first class where the cost of dictionary pointers is constant. Then flexible parsing on the graph \( G_T \) of the scheme is an optimal parse in this scheme.

Proof 4  By previous theorem and by Theorem 1, flexible parsing is an optimal parsing within the given scheme.

If we consider the algorithm LZ78 and the scheme \( SC_{LZ78} \) in the first class associated to it (following the proof of Proposition 1) then previous theorem applied to \( SC_{LZ78} \) is exactly the main result of [36, 34].

If instead we consider the algorithm FPA described in [22] then, since FPA uses flexible parsing, we have the following obvious corollary that derives directly from Definition 2.

Corollary 2  The parse of FPA is locally optimal.

Since Proposition 2 can be applied to FPA, it turns out that FPA is optimal within the scheme \( SC_{FPA} \), that is the scheme in the first class defined in the proof of Proposition 1.

We want now extend flexible parsing to the dictionary-symbolwise case. In order to do this, we make several hypotheses. First of all we assume the usual assumption that pointers have a constant cost. In a LZ78 compressor this is locally true in most
classical implementation. Indeed in many cases the pointer is represented by a number \( x \) and a letter where the number \( x \) is upper bounded by the total number \( n \) of nodes in the trie that represent the LZ78 dictionary at that moment. Therefore in many implementations, \( x \) is written using \( \lfloor \log(n - 1) \rfloor \) bits and this number of bits is locally constant and changes only when \( n-1 \) is a power of two. When we use a dictionary-symbolwise compressor, one must be a bit more careful with the cost of the flag information that must be added to the cost of a pointer. Nevertheless, the assumption that pointers have a constant cost (that includes the cost of the flag information) is quite realistic, i.e. it is a very close approximation of real implementations, also in the dictionary-symbolwise case.

A second assumption is that both the cost \( c_1 \) and \( c_2 \) of respectively pointers and letters are integers greater than or equal to 0. They are constant and \( c_2 < c_1 \). This is obviously a realistic hypothesis in the case of classical implementation of LZ78 and letters coded by the identity (i.e. not coded), but it does not always fit with arithmetical coding. Notice that if both the cost of pointers and letters (where these costs includes the cost of the flag information) are real numbers such that their ratio is a rational number, then a trivial scaling technique will reduce this problem to another equivalent one that fits this second assumption.

We call these assumption as A1 and A2 respectively.

Before defining our generalization of the flexible parsing to the dictionary-symbolwise case, we start splitting and stating in a “stand alone” way the ideas contained in the proof of Theorem 2.

**Lemma 4** If assumption A1 and A2 hold true, for any text \( T \) the distance \( d \) takes only values that are multiple of \( \gcd(c_1, c_2) \).

**Proof 5** The distance \( d \) at any node \( v \) is of the form \( d(v) = k_1c_1 + k_2c_2 \) with \( k_1 \) and \( k_2 \) integers, \( k_1 \geq 0, k_2 \geq 0 \) that is a multiple of \( \gcd(c_1, c_2) \).

By scaling, i.e. by using a new cost function that is equal to the previous divided by \( \gcd(c_1, c_2) \), we can suppose that \( \gcd(c_1, c_2) = 1 \).

At this point, by of Lemma 3, the distance \( d(i) \) from vertex 1 to vertex \( n + 1 \) of the text \( T \) is organized in run of equal numbers. That is, vertex 1 has distance 0 from itself. Vertex \( n + 1 \) has distance \( k \) and, in general, for any \( 0 < p \leq k \) the set of vertices that have exactly distance \( p \) is a set of consecutive vertices \( i_{p-1} + 1, \ldots, i_p \).

This run can be empty, i.e. it is possible that for some \( p \) there is no vertex having distance \( p \). This is one main difference with respect to the proof of Theorem 2.
In the rest of this section we suppose that assumption A1 and A2 hold true.

We denote by $i_p$ the greatest vertex that has distance $p$ (if such a vertex exists) and by $f(p)$ the greatest vertex that is a father of $i_p$. To be a father means that there is an arc $e = (f(p), i_p)$ in the graph $G_T$ from $x_p$ to $i_p$ and that $p = d(i_p) = d(f(p)) + C(e)$. Clearly $f(p)$ has distance smaller than $p$. Moreover, since we are in the dictionary-symbolwise case, the function $f$ is not necessarily injective. Indeed vertex 0 is equal to $f(c_2)$ and it can be in some special cases also equal to $f(c_1)$.

**Proposition 3** There is a subsequence of the sequence of all vertices of the form $f(p)$ for some integer $p$ plus $n + 1$,

$$0 = f(c_2), \ldots, f(k), i_k = n + 1$$

that is a path of minimal cost from vertex 1 to vertex $n + 1$ in the graph $G_T$.

**Proof 6** Obviously $i_k = n + 1$ is the last vertex of a path of minimal cost and therefore $(f(k), i_k = n + 1)$ is an edge of a path of minimal cost. For any $j$ with $2 \leq j \leq k$ there exists a number $p$ such that $d(f(j)) = p$. Therefore $f(p)$ is a father of $i_p$, and since the dictionary is prefix closed, $f(p)$ is also a father of $f(j)$. In other words, for any $j$ with $2 \leq j \leq k$ there exists a number $p$ such that $f(p)$ is a father of $f(j)$. Starting from $f(k)$ and going backwards we obtain a sequence of fathers that is then a path of minimal cost that is included in the sequence $0 = f(c_2), \ldots, f(k), i_k = n + 1$.

The main difference with the pure dictionary case is that in that case the sequence $0 = f(c_2), \ldots, f(k), i_k = n + 1$ is a path of minimal cost while now this sequence contains a path of minimal cost. In the original case flexible parsing was able to recover from left to right the sequence. If we want to generalize it, we must be able to recover not just this sequence but also a minimal path somehow.

For any integer $Q \geq 0$, $Q \leq \frac{k}{c_1}$ we want to find the set $F_Q$ of all the elements in the sequence $0 = f(c_2), \ldots, f(k), i_k = n + 1$ that have distance greater than or equal to $QC_1$ and that are smaller than $(Q + 1)c_1$. To each element $f(j)$ in $F_Q$ we also associate and memorize the corresponding vertex $i_j$.

We suppose that one can find $F_0$ in constant time. This fact is not so evident in the case of algorithms that uses some variation of LZ77 dictionaries with unbounded windows and allowing overlaps. We skip the description on how to obtain a constant costs in these particular cases, even if we were able to do it in all cases we considered, because flexible parsing can be replaced by a simpler and more efficient algorithm, as it
will be discussed in the following of this paper. In order to obtain $F_0$ in constant time in the case of LZ78 and derived dictionary-symbolwise schemes, we simply evaluate the distance in the graph $G_T$ up to distance $2c_1$, and then search for nodes $f(p)$ in $F_0$. Evaluating the distance $d$ can be done by making a classical RELAX on the nodes of the graph $G_T$ starting from node 0 up to the first node that has distance greater than or equal to $2c_1$.

Let us now to inductively obtain $F_Q$ starting from $F_{Q-1}$. For any element $f(j)$ in $F_{Q-1}$ we consider all the possible path $\alpha$ to a vertex $v_{f(j),\alpha}$ such that $d(f(j) + C(\alpha))$ is greater than or equal to $Qc_1$ and is smaller than $(Q+1)c_1$. Here for $C(\alpha)$ we mean the cost of the path $\alpha$ that, since $C$ is additive, is equal to the sum of the cost of all symbols and all pointers (including the cost of the flag information since assumptions A1 and A2 hold). If two vertices $v_{f(j),\alpha}$, $v_{f(j'),\alpha'}$ are such that $d(f(j) + C(\alpha)) = d(f(j') + C(\alpha'))$ we choose the one that gives the longest dictionary advancement in the next iteration. If this longest advancement in the next iteration is still equal we choose the greatest among $v_{f(j),\alpha}$ and $v_{f(j'),\alpha'}$. We call $S$ the set of all vertices $v_{f(j),\alpha}$ chosen in this way.

Moreover for any element $f(j)$ in $F_{Q-1}$ we consider all the possible path $\beta$ to a vertex $v_{f(j),\beta}$ such that $d(f(j) + C(\beta))$ is greater than or equal to $Qc_1 + 1$ and is smaller than $(Q+1)c_1 + 1$. We make a RELAX$(f(j), \beta)$, i.e. the RELAX is done with respect to the path $\beta$. If the father restricted to these paths of $v_{f(j),\beta}$ is $v_{f(j),\beta-1}$, then add $v_{f(j),\beta-1}$ to a set $T$ that was initially the empty set. For “father restricted to these paths” we mean that we perform the relax just on these paths and not over all possible paths that reaches the considered vertex.

**Proposition 4** The set $S \cup T$ is equal to $F_Q$. Moreover for any element $v_{f(j),\alpha}$ in $S$ its distance is $d(f(j) + C(\alpha))$.

**Proof 7** First of all, for any element $f(j)$ in $F_{Q-1}$ we consider all the possible path $\gamma$ to a vertex $v_{f(j),\gamma}$ such that $d(f(j) + C(\gamma))$ is greater than or equal to $Qc_1 - c_2$ and is smaller than $(Q+1)c_1$ or if it is equal to $(Q+1)c_1$ then the last element of $\gamma$ is a symbol. If we make a RELAX$(f(j), \gamma)$, i.e. the RELAX is done with respect to the path $\gamma$, then the greatest father restricted to these paths of $d(f(j) + C(\gamma))$ is the true unrestricted father. We prove this fact by induction on the length of $\gamma$. If $\gamma$ has length 1 then it is a symbol. If $d(v_{f(j),\gamma})$ is smaller than $d(f(j) + C(\gamma))$ then the father of $v_{f(j),\gamma}$ reached $v_{f(j),\gamma}$ by using a pointer, that is, in turn, a path $\gamma'$ of length 1 and it is not difficult to show that this father belong to $F_{Q-1}$.

66
6.3.1 On on-line, space optimal, flexible compressors

In this subsection we discuss some relationships among on-line compressors, space optimal compressors and compressors that use flexible parsing. On-line optimal compression is explicitly addressed by several papers (cf. for instance [30, 25, 26]) and the same it happens for space optimal compressors (cf. for instance [27, 17, 50]). But, since the efficient design and implementation of a compressor is a basic and fundamental research, the percentage of papers that consider these problems is quite high.

For “space-optimal” compressor generally it is meant that the considered compressor uses “linear space” with respect some parameter. Usually, in data compression algorithm, this parameter is the length of the text but rarely it has been considered the size of a subgraph of \( G_T \) (cf. [17, ?]).

For “on-line” it is usually meant an algorithm that has linear overall worst case time and that when a letter \( a \) is added to the text \( T \), it can use the work done for text \( T \) to produce an output for text \( Ta \). For each added letter the time spent in this process is not necessarily constant, but the overall time is linear. Therefore on-line algorithms must use linear space and are optimal also in space.

A flexible parsing is certainly space-optimal, supposing that the data structure used for the matching is linear in space. Flexible parsing can be very space-thrifty if it is coupled with an intelligent data structure. Indeed, thanks to flexible parsing there is no need even to keep an array (linear in space) where keeping track of ancestors of nodes in order to recover a shortest path in the graph \( G_T \) and turns out to a linear time algorithm. Intelligent data structures have been shown to exist in [36, 34] for the pure dictionary case and we believe that this fact also holds in the dictionary-symbolwise case. But we leave this problem open up to now.

Flexible parsing and its generalizations can also be applied to LZ77 dictionary based compressors, but, thanks to the fact that LZ77 dictionary are suffix closed and not just prefix closed, it is possible to give some even more efficient parsing algorithms.

6.4 Dictionary-Symbolwise Can Have Better Ratio

It is not difficult to give some "artificial" and trivial example where coupling a dictionary and a symbolwise compressor give rise to a better optimal solution. Such an example is given in what follows.

Example 1 Let us consider the following static dictionary
< a, b, ba, bb, abb >

together a cost function \( C \) that could represents the number of bits of a possible code:

\[
\{ C(a) = 8, C(b) = 12, C(ba) = 16, C(bb) = 16, C(abb) = 4 \}.
\]

A greedy parsing of the text babb is (ba)(bb) and the cost of this parsing is 32. An optimal parsing for this dictionary is (b)(abb) that has cost 16. This example also shows that a greedy parsing is not always an optimal parsing in dictionary compressors.

Let us consider further the following static symbolwise compressor that associate to the letter “a” a code of cost 8 and associate to the letter “b” a code of cost 4 that could represents the number of bits of this code. The cost of coding babb following this symbolwise compressor is 20.

If we connect them in a dictionary-symbolwise compressor then an optimal parsing is \( S(b)D(abb) \) where the flag information is represented by the letter \( S \) for symbolwise of next parse phrase or \( D \) that stands for dictionary. The cost of the flag information is one bit for each letter. Therefore the cost of this parsing is 10.

In this subsection, however, we will prove something more profound than artificial examples such the one above. Indeed, from a theoretical point of view Ferragina et al. (cf. [17]) proved that the compression ratio of the classic greedy-parsing of an LZ77 pure dictionary compressor may be far from the bit-optimal pure dictionary compressor by a multiplicative factor \( \Omega(\log(n)/\log \log(n)) \), which is indeed unbounded asymptotically. The family of strings that is used in [17] to prove this result, is a variation of a family that was used in [?] to prove that the compression ratio of the classic greedy-parsing of an LZ77 pure dictionary compressor may be far from the first order empirical entropy by a multiplicative factor \( \Omega(\log(n)/\log \log(n)) \).

We show in this subsection a similar result between the bit-optimal pure dictionary compressor and a dictionary-symbolwise compressor. Therefore a bit optimal dictionary-symbolwise compressor can use, in some pathological situation, the symbolwise compressor to avoid them and be provably better than a simple bit optimal pure dictionary compressor.

Let us define these two compressors. The first is a LZ77 based compressor that allows overlaps with unbounded windows as dictionary and with an Huffman coding on the lengths and an optimal parser. Since we are using an Huffman coding, this scheme without a parser belongs to the third class, but in this section we do not care
on how to obtain an optimal parser but we just suppose we have one of these optimal parsers. The encoding of pointers can be any of the classical intelligent encoding. We just impose an Huffman coding on the lengths.

We further denote by \( \text{OPT-LZH}(s) \) the bit length of the output of this compressor on the string \( s \).

The same LZ77 is used as dictionary compressor in the dictionary-symbolwise compressor. Clearly we do not include the parser in the dictionary-symbolwise compressor, but, analogously as above, we suppose we have an optimal parser for the dictionary-symbolwise compressor, no matter about the description. The flag information \{ \( D, S \) \} is coded by a run-length encoder. The cost of a run is subdivided over all symbolwise arcs of the run, i.e. if there is a sequence of \( n \) consecutive symbolwise arcs in the optimal parsing then the cost of these \( n \) flag information letters \( S \) (for Symbolwise) will be in total \( O(\log(n)) \) and the cost of each single flag information in this run will be \( O\left(\frac{\log(n)}{n}\right) \). It remains to define a symbolwise compressor. In the next result we could have used a ppm* compressor but, for simplicity, we use a longest match symbolwise. That is, the simbolwise at position \( k \) of the text searches for the closest longest block of consecutive letters in the text up to position \( k - 1 \) that is equal to a suffix ending in position \( k \). This compressor predicts the \( k + 1 \)-th character of the text to be the character that follows the block. It writes a symbol ”\( y \)” (that is supposed not to be in the text) if this is the case. For otherwise it uses an escape character \( n \) (that is supposed not to be in the text) and then write down the correct character plainly. A temporary output alphabet has therefore two characters more than the characters in the text. This temporary output will be subsequently encoded by a run-length encoder.

This is not a very smart symbolwise compressor but it fits our purposes, and it is simple to analyze.

We further denote by \( \text{OPT-DS}(s) \) the bit length of the output of this Dictionary-symbolwise compressor on the string \( s \).

**Theorem 3** There exists a constant \( c > 0 \) such that for every \( n' > 1 \) there exists a string \( s \) of length \( |s| \geq n' \) satisfying

\[
\frac{\text{OPT-LZH}(s)}{|s|} \geq c \frac{\log(|s|)}{\log \log(|s|)} \text{OPT-DS}(s).
\]

**Proof 8** For every \( n' \) let us pick a binary word \( w \) of length \( 2n, n \geq n' \), \( w = a_1 a_2 \cdots a_{3n} \) that has the following properties.
1. For any \( i, 1 = 1, 2 \ldots n \) compressor \( \text{OPT-LZH}(s) \) cannot compress the word \( a_i a_{i+1} \ldots a_{2i+n-1} \) of length \( n + i \) with a compression ratio greater than 2.

2. every factor (i.e. every block of consecutive letters) of \( w \) having length \( 3 \log(3n) \) of \( w \) is unique, i.e. it appears in at most one position inside \( w \).

Even if it could be hard to explicitly show such a word, it is relatively easy to show that such a word exists. Indeed, following the very beginning of the Kolgomorov’s theory, the vast majority of words are not compressible. A simple analogous counting argument can be used to prove that property 1) is satisfied by the vast majority of strings of length \( 2n \), where, for vast majority we mean that the percentage of strings not satisfying 1) decreases exponentially in \( n \). Here, to be safer, we allowed a compression “two to one”.

A less known result (cf. [2, 47, 15, 19, 16, 11]) says for random strings and for any \( \epsilon > 0 \) that the percentage of strings of length \( n \) such that every its factor of length \( 2 \log(n) + \epsilon \) is unique grows exponentially to 1 (i.e. the percentage of strings not having this property decreases exponentially). Here we took as \( \epsilon \) the number 1. Therefore such a string \( a_1 \cdots a_{3n} \) having both properties surely exists for some \( n \geq n' \).

Let us now define the word \( s \) over the alphabet \( \{0, 1, c\} \) in the following way.

\[
s = a_1 a_2 \cdots a_n c^{2^n} a_2 a_3 \cdots a_{n+3} c^{2^n} \cdots a_i a_{i+1} \cdots a_{2i+n-1} c^{2^n} \cdots a_{n+1} a_{n+2}.
\]

Let us now evaluate \( \text{OPT-LZH}(s) \). By property 1) each binary word that is to the left or to the right of a block of \( 2^n \) \( c \)'s cannot be compressed in less than \( \frac{1}{2} n \) bits in a “stand-alone” manner. If one such a string is compressed by a pointer to a previous string then the offset of this pointer will be greater than \( 2^n \) and, so, its cost in bit is \( O(n) \). We defined the string \( s \) in such a manner that all ”meaningful” offsets are different, so that even an Huffman code on offsets (that we do not use, because we use an Huffman code only on lengths) cannot help. Therefore there exists a constant \( c' \) such that \( \text{OPT-LZH}(s) \geq c'n^2 \).

Let us now evaluate \( \text{OPT-DS}(s) \). We plan to show a parse that will give a string of cost \( P\text{-DS}(s) \leq \hat{c} n \log(n) \) as output. Since \( \text{OPT-DS}(s) \leq P\text{-DS}(s) \) then also \( \text{OPT-DS}(s) \leq \hat{c} n \log(n) \).

The blocks of \( 2^n \) \( c \)'s have all the same length. We parse them with the dictionary compressor as \( (c)(c^{2^n} - 1) \). The dictionary compressor is not used in other positions in the parse \( P \) of the string \( s \). The Huffman code on lengths of the dictionary compressor
would pay \( n \) bits for the table and a constant number of bits for each occurrence of a block of \( 2^n \) c’s. Hence the overall cost in the parse \( P \) of all blocks of letters c is \( O(n) \). And this includes the flag information that consists into two bits \( n \) times.

Parse \( P \) uses the symbolwise compressor to parse all the binary strings. The first one \( a_1a_2 \cdots a_{n+1} \) costs \( O(n) \) bits. Starting from the second \( a_2a_3 \cdots a_{n+3} \) till the last one, the symbolwise will pay \( O(\log(n)) \) bits for the first \( 3\log(3n) \) letters and then, by property 1), there is a long run of \( y \) that will cover the whole string up to the last two letters. This run will be coded by the run-length code of the symbolwise. The overall cost is \( O(\log(n)) \) and this includes the flag information that is a long run of \( S \) coded by the run-length of the flag information. The cost of the symbolwise compressor included the flag information over the whole string is then \( O(n\log(n)) \), that dominates the cost of the dictionary-symbolwise parse \( P \).

The length of the string \( s \) is \( O(n2^n + n^2) \) and therefore \( \log(|s|) = n + o(n) \) and the thesis follows.

Remark 6 In the theorem above it is possible to improve the constants in the statement. This can be done simply using for instance a word \( a_1 \cdots a_{n^2} \) instead of \( a_1 \cdots a_{3n} \).

We want to underline that the Huffman coding on the lengths is essential in this statement. At the moment we were not able to find a sequence of strings \( s \) where the dictionary-symbolwise compressor is provably better than the optimal pure dictionary version without using an Huffman coding. It is an open question whether this is possible. It is also an open question to obtain an analogous result with an empirical entropy of some order \( k \) instead of the dictionary-symbolwise compressor.

We finally notice that if the dictionary is coupled with a ROLZ technique then the optimal solution of the pure dictionary compressor reaches the same level of the dictionary symbolwise compressor. This is not surprising because the ROLZ technique is sensible to context and do not ”pay” for changing the source of the text.

### 6.5 Optimal Parsing in Linear Time

Overcoming the complexity limitations of the classical graph used to find an optimal parsing for dictionary compressions, while adapting it to the dictionary-symbolwise schemes, we looked for some hypothesis allowing to build a linear size subgraph of the classical one while maintaining the correspondence between the optimal parsing of the text and shortest in the graph.

Given a text \( T = a_1 \cdots a_n \), for any dictionary-symbolwise scheme of the first class
(see Section 6.1), we can define a subgraph $G'_T = (V, E')$ of the natural generalization of $G_T = (V, E)$ (see Section 6.2) in the following way. The edges in $E'$ can be conceptually distinguished in two classes, so that $E' = E'_1 \cup E'_2$, where $E'_1 = \{(i, i + 1) | i < n\}$ and $\forall i < j < n, (i, j) \in E'_2$ iff $a_i \cdots a_{j-1}$ is the longest matching element from the dictionary starting from $a_i$. Hence $E'_1$ represents the base path, following the definition in [28], and for these edges the label $L_{i,i+1}$ is the cost of encoding the symbol $a_i$ using the symbolwise compression plus the cost of the corresponding information flag. The edges in $E'_2$ come from the dictionary, and more precisely from a greedy parsing. In this case, the label $L_{i,j}$ is the cost of encoding the pointer to the dictionary word plus the cost of the corresponding information flag.

![Text: abccacbbabbcbcb, Dictionary: < ab, cbb, ca, bcb, abc >](image)

Figure 6.3: Greedy parsing graph with base path

Note that, from a purely formal point of view, from vertex $i$ to vertex $i + 1$, we could have both edges of the first and of the second kind, and therefore $G'_T$ should be a labeled multigraph. Since the cost of pointer is usually much bigger that the cost of a symbol, to avoid these situations in this paper we do not include in $G'_T$ edges of the second kind having length equal to one. However, all the results presented do not change if these edges are added to $G'_T$. As an example, Figure 6.3 shows an example parsing where the dashed edges are in the base path $E'_1$ and the solid edges are the ones in $E'_2$.

The graph described above is always well defined, even for dictionary-symbolwise schemes that include algorithms belonging to the second or third class, except for the label function. If a scheme includes an algorithm of the third or second classes, it is possible to define labels by using approximations of the expected costs.

Graph $G'_T$ contains no cycles and it is already naturally ordered in a topological order. Thus we can find the shortest path in $O(|E'|)$, as on the classical graph. However, in this case $O(|E'|) = O(n)$, because each vertex $i$ has at most two outgoing edges (one from $E'_1$ and possibly one from $E'_2$).
In this section we will show that, under a set of reasonable restrictions on the dictionary-symbolwise algorithm used, the shortest path in $G'_T$ from vertex 0 to vertex $n$ corresponds to an optimal parsing of the text $T$.

Let $S$ be a fixed dictionary-symbolwise algorithm and $C$ a cost function. Moreover, let us suppose that the following properties hold for $S$ and $C$: (1) $S$ belongs to the first class, (2) the dictionary of $S$ is suffix closed, (3) the cost of encoding any dictionary pointer together with the corresponding flag information is a positive constant $c$, (4) the cost of encoding a symbol and the corresponding flag information is a (possibly non constant) value greater than or equal to zero and always smaller than $c$. We can state the following

**Theorem 4** The problem of finding the optimal parsing for a text $T$, relative to the dictionary-symbolwise algorithm $S$ that satisfies above constrains (1-4), reduces to the problem of finding the shortest path in $G'_T$ from vertex 0 to vertex $n$.

**Proof 9** Under the hypotheses of the theorem, the graph $G'_T$ is fully defined, including the labels. Suppose now that we have an optimal parsing $(u_1, \ldots, u_s, Fl)$ for $T$. If all the parsing phrases of this optimal parsing are either symbols or greedy choices of elements from the dictionary, then this parsing corresponds to a path in $G'_T$ and, since it is optimal, it must be the shortest path from vertex 0 to vertex $n$. If there are some phrases that are not greedy choices, let $u_i = a_j \cdots a_{j+k}$ be the one with the smallest index. A greedy choice of an element in the dictionary would therefore give a dictionary word $v = a_j \cdots a_{j+k'}$ with $k' > k$. We have two possibilities.

If $j + k'$ is the end position of a phrase $u_{i'}$ (with $i' > i$) in the optimal parsing, we replace $u_i \cdots u_{i'}$ with $v$ obtaining a new parsing with a cost less than or equal to the previous one, since $C(v) = c \leq C(u_i \cdots u_{i'})$ ($u_i \cdots u_{i'}$ contains at least one dictionary pointer).

Otherwise, if $j + k'$ is not the end position of a phrase in the optimal parsing, a parsing phrase $u_{i'} = a_{j'} \cdots a_{j'+h}$ with $j' \leq j + k'$ and $j' + h > j + k'$ has to exist. Since the dictionary is suffix closed, it must also contain the word $v' = a_{j'+1} \cdots a_{j'+h}$. Thus, we can replace the parsing phrases $u_i \cdots u_{i'}$ with $vv'$, obtaining a new parsing with a cost less than or equal to the previous one, since $C(vv') = 2c \leq C(u_i \cdots u_{i'})$ ($u_i \cdots u_{i'}$ contains at least two dictionary pointers).

In both cases we obtained a new parsing whose cost is less than or equal to the previous one and where the first phrase that is not a greedy choice is strictly closer to
the end of the text than in the original parsing. Thus, by iterating this argument (more formally by induction on the number \( j \)) we after all obtain an optimal parsing where all phrases are either symbols or greedy dictionary choices.

Note that the constraints on \( S \) and \( C \) are not very restrictive. Indeed, the condition on the cost \( c \) of pointers is considered in several research papers (see, e.g., [9, 34] and references therein) and has been usually considered for LZ78 and similar schemes. Indeed previous theorem can be seen as an extension to the dictionary-symbolwise case of the main result in [9]. Theorem above can also be extended to the case when the algorithm belongs to the second class, i.e. when the graph can be dynamically built. For second class algorithms using an online parsing, the labels of the edges outgoing vertex \( i \) can be precisely defined once all previous labels and the parsing up to \( i \) have been created, since the shortest path algorithm proceeds left to right following the topological order. Analogously it is possible to extend the main result of [34] to algorithms belonging to the second class.

Regarding LZ77 and derived schemes, that use non constant costs of pointers, we notice that the algorithm for shortest path in \( G'_T \) from vertex 0 to vertex \( n \) induces a parsing algorithm that strictly improves the not greedy parsing considered in [22]. Moreover, in Section 6.8 we will describe an improved algorithm that outputs an approximation of an optimal parsing also when the cost of pointer encoding is non constant.

At the end, it is important to notice that the algorithm for the shortest path in \( G'_T \) could also be defined when considering algorithms belonging to the third classes. Indeed, for these algorithms it is possible to define the edge labels using approximations of the expected costs (see Section 6.5). In this case the obtained parsing will be an approximation of the optimal one as well.

6.6 DAWGs and LZ77 Algorithms

In this section we will show a way to build the graph \( G'_T \) in linear time. The described technique is applied to a variation of the LZ77 algorithm where the longest match in position \( i \) can start in any position smaller than \( i \). However, it may be adapted to work on different compression algorithms.

For building the graph without the labels it is possible to use classical data structures such as suffix trees or truncated suffix trees (see [35, 32] and references therein). These structures allow us to find the position of the longest dictionary match for a given text.
Factor automata (DAWGs) have also been used to this aim ([10]) but, to our best knowledge, compact factor automata (CDAWGs) have never been applied to this problem, even if they have some interesting properties that favorably compare to suffix trees. Indeed, CDAWGs require in average less memory than suffix trees or truncated suffix trees over the same texts. Moreover, there exist on-line linear algorithms for building them, even using a sliding window (see [25, 24]).

Given a text $T = a_1 a_2 \cdots a_n$, to build $G'_T$ it is sufficient to know, for each position $i \in [1, n]$, the length $L(i)$ of the longest prefix of $a_i \cdots a_n$ that also appears in another previous position $j < i$ (obviously, $L(1) = 0$ always holds). Indeed, we can define the set of dictionary edges $E'_{2}$ of $G'_T$ as $E'_{2} = \{(i, i + j)|L(i) = j\}$.

For instance, if $L(4) = 2$, there is an edge outgoing vertex 4 and reaching vertex 6 = 4 + 2. Indeed, $L(4) = 2$ means that the longest dictionary match starting from $a_4$ has length 2, that is $a_4 a_5$. Therefore, as described in Section 6.5, $E'_{2}$ contains the edge $(4, 6)$.

<table>
<thead>
<tr>
<th>$i$</th>
<th>$a_i$</th>
<th>$S(i) \in V'$</th>
<th>$L(i)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>b</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>c</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>c</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>a</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>c</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>b</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>b</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>a</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>b</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>b</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>c</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>c</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>13</td>
<td>b</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>14</td>
<td>c</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>15</td>
<td>b</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>16</td>
<td>b</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 6.4: Creation of the parsing graph $G'_T$ using DAWGs

As an example, the table in Figure 6.4 shows the values of $L$ for a sample text, and the corresponding parsing graph.

If we apply the classical algorithm described in [24] for building on-line in linear time the CDAWG for $T$, after reading the first $i > 1$ symbols we can retrieve in constant time two information: the length $S(i)$ of the longest suffix of $a_1 \cdots a_i$ that appears in a position $j < i$ ($S(1) = 0$ always holds), and the position $P(i) = j$ ($P(i)$ is undefined if $S(i) = 0$).

After building the CDAWG for $T$, we can define in linear time $L(i)$, $i \in [1, n]$ by means of $S$ with the following algorithm. Notice that a similar algorithm, working on
different data structures, was also independently developed by Maxime Crochemore et al. and described in [13, Ch. 1.6].

Let $V' = \{1 \leq i \leq n | S(i + 1) \neq S(i) + 1\}$. Intuitively, $V'$ contains $i$ if the longest suffix of $a_1 \cdots a_{i+1}$ that appears before $a_{i+1}$ does not contain the whole $a_1 \cdots a_i$. Note that $V'$ always contains $n$.

Moreover, let $V'' = \{(i, j) | i, j \in V' \land i < j \land (\nexists k < j | k \in V')\}$. In other words, $V''$ is the set of pairs created by consecutive elements of $V'$. Then, $L$ can be defined through $S$ as follows:

$$\forall (i, j) \in V'', \forall (i - S(i) + 1) \leq k \leq (j - S(j)) \quad L(k) = i - k + 1$$

$$\forall (n - S(n) + 1) \leq k \leq n \quad L(k) = n - k + 1$$

For instance, Figure 6.4 shows the values of $S$ and the corresponding members of $V'$ for a sample text. The pairs in $V''$ for this example are $\{(1, 2), (2, 4), (4, 5), (5, 6), (6, 7), (7, 8), (8, 10), (10, 11), (11, 14), (15, 16)\}$. By applying the algorithm above, we obtain the correct definition of $L$, as reported in the figure. The reader can refer to [38] for a more complete example of this algorithm.

In this algorithm, if the longest suffix that appears in a previous position of $a_1 \cdots a_i$ has length $l$ and the longest suffix that appears in a previous position of $a_1 \cdots a_{i+1}$ has length smaller than $l + 1$, then the longest prefix of $a_{i-l} \cdots a_n$ that appears in a position smaller than $i - l$ has length $l$. The formal proof of the correctness of this algorithm is left to the reader.

6.7 An Online Solution

Another interesting feature of the algorithm described above is that it can be computed online. This allows to optimally parse data streams of any length.

Indeed, the CDAWG construction algorithm given in [24] can be feed progressively with text and, at each step, allows to read the values of $S$ for the given text segment. So we can progressively build $V'$, $V''$ and $L$ (and the parsing graph), since the corresponding algorithm proceeds from left to right following the incremental definition of $S$. Some technicality must be added to handle the last elements of $S$, which allow to define the edges pointing to the last node of the graph $G'_T$ (in topological order). This can be accomplished by exploiting the same techniques used in the online algorithms described in [24, 25], which in turn exploit the technique used by the Ukkonen’s online suffix-tree construction algorithm [49].
In particular, the algorithm works when the cost of encoding for dictionary pointers is constant and also when it depends logarithmically from the pointer length.

Moreover, the online solution also holds for dictionary-symbolwise algorithms belonging to the second class (see Section ??). Indeed, in this case, the cost of pointers outgoing vertex $i$ is always defined when the text preceding $a_i$ has been parsed, thus allowing to perform the subsequent step of the shortest path algorithm.

### 6.8 Further Improvements

In contrast to the hypothesis given in Section 6.2, in the case of LZ77 and similar algorithms, the cost of a dictionary pointer is not constant but usually depends logarithmically on the size of the parse phrase that the pointer is referring to. Indeed, the pointer is an \((offset, length)\) pair that indicates the distance of the matched phrase from the current position on the text and its length. However, the first member has usually a \(fixed\) bit length that allows to express offsets of reasonable (and realistic) size. So, any optimization should be applied to the \(length\) member.

If a portion of a text can be parsed in more than one way by dictionary pointers, usually the smaller cost is obtained by the most asymmetrical parsing, i.e. when the lengths of the parse phrases are as far as possible from an uniform length. This because of a property of the logarithm function. For example, consider a text of length 10 that can be matched by two different combinations of dictionary words. The first parsing contains two dictionary pointers of uniform length 5, whereas in the second the pointers have length 2 and 8. The costs of these parsings are, respectively, \(\lceil 2 \log_2 (5) \rceil = 3\) and \(\lceil \log_2 (2) + \log_2 (8) \rceil = 2\) bits, i.e., the most asymmetrical combination of lengths requires less bits to be represented.

This fact can be exploited to approximate an optimal parsing when the cost of dictionary pointers in not constant. Indeed, we can add some new heuristic in the construction algorithm for graph $G_T'$ shown in Section 6.5 as follows.

Let $G_T' = (V, E')$ be the parsing graph for a text $T$ created as described in Section 6.5. We pick all the pairs of \(adjacent\) edges in $E'_2$, i.e. edges of the form \((i, j)\) and \((j, t)\). Then, for each pair, we look for the smallest $i < j' < j$ such that $(j', t) \in E'$. If $j'$ exists, we add $(i, j')$ to $E'$. In this way, we add at most a new edge from each vertex in $V$. If we call $G''_T$ the obtained graph, we have that $O(|G''_T|) = O(n)$, where $n$ is the size of the text $T$. The new edges represents words of the dictionary if the dictionary is prefix-closed. This is always the case in dictionary-symbolwise schemes that use as dictionary a portion of the text previously seen, analogously to the LZ77
and similar algorithms.

These new edges can be efficiently created without increasing the complexity of the graph construction algorithm. To this aim, for \( t = 1, \ldots, n \) we define \( V(t) \) as the smallest \( j' < t \) s.t. \( (j', t) \in E' \). Clearly \( V \) can be defined as soon as the edges are added to \( E' \), and can be used to create the new edges above in constant time.

At the end, after that this information has been added to the graph, we can approximate the optimal parsing by simply looking for the shortest path in \( G''_T \).
Part III

Practical contribution
Chapter 7

Algorithms Description

The aim of this chapter is to describe, at high level, the two programs that perform the optimal parsing mixing Huffman and LZ-dictionary technics, i.e. Lgzip (LZ77) and OptLZ78 (LZ78).

7.1 What is Large GZip

Lgzip is a version of gzip that implements an optimal parsing and a large dictionary. As gzip does, lgzip uses a mixture of characters and LZ77-dictionary pointers encoded with Huffman coding. The dictionary size grows from 32KB of gzip to 16MB and it is able to represent text fragments up to 514 byte long. According to our theoretical results on 6.5.1, lgzip compute an optimal parsing finding the shortest path on the graph $G'_T$ (see Theorem 4), where arc weights are Huffman code lengths. Obviously, lgzip belongs to the third class of dictionary-symbolwise scheme, so we have to use some practical work-around to dress the graph weights definition problem (see 2).

7.1.1 Building the Graph

The first pass of the lgzip algorithm is to build the graph of “ahead” or “forward” factors that will be involved in the parsing process. The node $i$ of the graph corresponds to the $(i - 1)$-th byte of the input text, let’s say the $(i - 1)$-th character. So, we insert a symbolwise edge in the graph representing the $i$-th character starting from node $i$ to the node $i + 1$. The potential dictionary arcs are all the forwarding factors that match with a fragment of text already read, according to LZ77 dictionary definition. Matches are limited in size and in relative position (offset). In gzip standard the length are bounded to 258 and the offset distances to 32K that is the dictionary size. In our
program, as stated before, the dictionary is of 16MB, i.e. we use a sliding window on the input, also called history, and lengths are bounded to 514 characters. The dictionary is supposed to be updated with the rightmost occurrence of each forward match while the algorithm processes sequentially the input text.

7.1.2 Efficient Dictionary Data Structure

Dictionary data structure is required to be able to found, for each text position \( i \) and for each length \( l \), the position \( j \) that fragments \( < a_j \ldots a_j + l > \) is the closest match (rightmost occurrence) of \( < a_i \ldots a_i + l > \) in the history. This objective was first reached by a CDAWG, as showed in section 6.6. This structure requires linear construction time and size and search time linear in the occurrence length \( O(\log(n)) \) for longest occurrence and its suffix. This approach is very interesting theoretically speaking, but it was replaced with a faster and more widely used hash table. This structure guarantees constant lookup time for fixed length matches plus \( O(\log(n)) \) time to extend these in length. This approach results faster in practice, due to some constant. Lgzip uses a 3byte hash table.

7.1.3 Dictionary-Complete Graph

The simplest methods to build the graph is to add an edge for each forward factor that matches in the dictionary. We say that the resulting graph is dictionary-complete. In this configuration we create an edge in the graph at node \( i \) of length \( l \) for each position \( i \) in the text and for each forwarding match of length \( l \), from node \( i \) to node \( i + l \), that represents an occurrence of the forwarding factor \( (i .. i+1) \) somewhere in the history. Let us recall that in LZ77-based algorithms, a dictionary pointer is defined by a couple \((l,m)\), i.e. length and offset. Let’s say that the longest match starting from position \( i \) is of length \( L \) and it matches with a factor in the history at position \( i-m \) in the text, i.e. at relative position (offset) \( m \). Then, for each \( l \), from \( L \) down to \( \text{min}\_length \) (3 in gzip and lgzip), matches exist at positions less or equal to \( m \), due to the prefix close property of the LZ77 dictionary. The weight used for these arcs depends from the length of the Huffman code associated to the character ascii value, in the symbolwise case, or it depends from the sum of the lengths of the Huffman codes associated to the dictionary pointer length and to dictionary pointer offset.
7.1.4 Linear Size Graph

Lgzip does not create the dictionary-complete graph, the size of which is $O(n \log(n))$ in the average case and $O(n^2)$ in the worst case, but it uses just a selection of arcs that leads to a linear size $(k + c)n$ graph, i.e. $O(n)$ average case. In other word we reducing dictionary size, but our theory still guaranteed optimality of parsing solution. This is what allows us to use a large history for our compressor with respect to gzip certainly, but especially with respect to other gzip-based compressor like PKZip, 7Zip and other that use Deflate 64 parsing algorithm to produce .gz or .zip compressed file. Deflate64 is a PKWARE [40] proprietary parsing method, based on LZ77, that uses a dictionary up to 64KB large [29] and, how we and many authors suppose, it computes an optimal parsing through the dictionary-complete graph.

We use two criteria to select factors to be inserted into the graph. The first idea is partially based on [17] and on the observation that many of the matches at position $i$ occur at the same position in the history, and so they have the same offset in their pointer representation. Indeed, they are just prefix of the same occurrence. So, the weight of these arcs have a common component given by the offset, that let us suppose that their weight is close to each other. We consider meaningless arcs that are so close in length (they are prefix each other) and in weight. The trick is to create in the graph, for each position $i$, just the first $k$ arcs that represent “different” factors in the position (offset) sense. Actually, already they differ each other for the length.

The second idea is based on something similar to flexible parsing intuition [34]. Let $i$ be a node of a graph whose longest arc is of length $m$, starting from $i$ and leading to $i + m$. All the arcs starting from $i$ are shorter than $m$, leading to node between $i$ and $i + m$. We know that at this point the greedy parsing would choose the longest one (of length $m$), while the flexible parsing is supposed to choose the one of length $n$ that ends to node $j$, with $j = i + n <= i + m$, iff the longest arc starting at node $j$ is not a suffix of $i..i + m$, so its length is greater than $m - n$. Starting from this consideration we put in our graph the first $c$ arcs starting from $i$ of length $l < L$ only if the longest match at position $i + l$ is longer than those at $i + l + 1$. It is possible to prove that for each position $i$ these arcs are $c'$ in average.

We could develop a clearer and more detailed theory to support these prune technics as extension of what done in 6.5 about optimal parsing of a graph with character and greedy arcs, but we prefer to leave it in draft because for practical purpose the results in section 6.2, about the extension of flexible parsing to dictionary-symbolwise compression algorithms 6.3 and its optimality, are more interesting and even the dictionary-
complete approach is sustainable.

7.1.5 Flag Information

As already gzip does, lgzip does not represent explicitly flag information, because it includes this information in dictionary pointer lengths and characters representation using two different huffman codes to represent the same value distinguishing between a character ascii code and a length of a length-offset couple. Gzip-based algorithms use one Huffman tree to code the characters, as symbolwise compression, and the same Huffman tree to code the lengths of the couple \((l, m)\) of the dictionary pointers. This allows these algorithms to hide the flag information, or, to say better, to include this information in this first Huffman coding tree. Another Huffman coding tree is used just for pointers offsets.

7.1.6 Arcs Weight

As mentioned at the very top of this chapter, the weights of the graph arcs are their Huffman codings. Consequently, we use the length of the Huffman coding of the character as cost for the symbolwise arcs on the graph and the sum of the length of the Huffman coding for the pointer length and the pointer offset (actually, some more extra bits are used for the dictionary pointers due to a not injective mapping to the Huffman word table).

7.1.7 Shortest Path

At this point we have to deal with a problem of weight definition and weight changing. The arc weights of our graph depend on the parsing, as the cost of Huffman coding. Arcs belonging to the chosen parse have been chosen basing on their cost that depends on the still forming parsing.

How to initialize to cost without a parsing? And more, how to find an optimal parsing, a shortest path in the graph, if the cost of the arcs changes as soon as we choose any parsing? Our solution is based on a theoretical assumption and a practical result. The theoretical assumption is that the cost of the length and the cost of the offsets should follow the normal statistic distribution around the average length and offset expected values, due to statistical property of Huffman coding, and so, they have to converge to some values. We initialize the graph costs with the Huffman coding of a simple parse, we use something between the greedy and flexible parse and we use the worst cost in the Huffman code where code actually is undefined. This means that
we weigh unused arc equal to used only one arc, more exactly, used only one length or literal or offset values. At this point, graph $G'_T$ is well defined and it is possible to find the shortest path. We update the graph costs basing on the previous parse and reiterate the whole process. According to the theoretical assumption, we experimentally proved that the arc costs converge quickly and they get stable in very few iterations (less than 4) for encoded blocks 1MB long. We suppose that exist multiple fixed point for this cost function and we rich just a local minimum. We made some quick experiment of weight perturbation, some other forcing initial parsing and other promoting some kind of arcs instead of others, let us suppose that parsing found in this way is stable enough to be considered a good minimum approximation, but we let this as open problem. Blocks are pieces of encoded data that have their own Huffman coding. Indeed, Huffman trees are included in the blocks header. For practical purpose and theoretical needs, lgzip start with a block long 7KB and it double block size up to 1MB for the next blocks, to follow changes of the length and offset expected size at history growing. Anyway, every about 1MB of precessed data, Huffman code get completely updated accordingly to data source statistical model. This bound memory usage, too.

7.1.8 Coding Details

Many encoding details follow .gz and .zip main philosophy. Files name, files size, files time, CRC values [39] and directory structure information are treated in a way very similar to gzip standard explained in [43]. Decompressor is a simple extension of a .gz decoder, able to treat with 16M of history and other minor details.

7.2 Optimal LZ78

Optimal LZ78 use a LZ78-based dictionary, Huffman coding and optimal parsing. We collect, after the parsing, all the dictionary pointers in one file, all the characters in another one and another file with directive (flag) information and we encode them using huffman coding. Since the work described in this thesis has been developed by a team of people, this section has been mainly developed by Andrea Ulisse. We report it here for sake of completeness.

7.2.1 Parsing of the Input File

First of all, the program copies all the input file on the RAM, in way to manipulate it in a easier and more efficient way. Next, it is carried on the construction of the
dictionary for LZ78 and the graph on which it will performed the computation of the shortest path that will give the optimal parsing.

It is not described how to build the tree related to LZ78 algorithm because it has already been widely explained in section 4.7.

But, it is useful to describe how to build the auxiliary graph for performing the optimal parsing. In fact, the aim of this auxiliary graph is to take trace of all the possible ways for encoding a certain portion of file. That is, in correspondence of each byte of the input file, the program must be able to choose if carrying on the compression by using the symbolwise technique selected or by using a entry of LZ78 dictionary.

This objective is reached in this way. First of all, it is created an auxiliary graph, that has a node for each byte of the input file, including EOF character. Let’s suppose that LZ78 parsing, from byte $i$, given the LZ78 dictionary built previously, has been able to make 3 steps, until byte $i + 3$. In the auxiliary graph it has to be inserted an edge from node $i$ to node $i + 3$, stating that from byte $i$ it is possible to compress with an entry of LZ78 dictionary. Now, the auxiliary graph must be filled with other edges. First of all it has to be inserted an edge for all the generic bytes $i$, in way to allow the optimal parser to walk towards byte $i + 1$. Moreover, it has to be performed the transitive closure. That is, for each byte $j$ included between $i$ and $i + 3$ it has to be stated if it is possible to insert an edge due to the LZ78 dictionary built until character $i$, starting from byte $j$. Relating to this specific example, it is possible, rather it is frequent, that from a generic byte $j$ an edge allows the optimal parser to “jump” to a byte $k > i + 3$.

**Weight of the Edges**

A question obviously arises: how are the edges of the auxiliary graph weighed? They have to be distinguished two kinds of edges.

1. edge from a node $i$ to a node $i + 1$: this kind of edge is weighed with an integer number corresponding to the number of bits that Huffman coding would use for encoding character $i$. This value is obtained thanks to a preprocessing function that calculates the number of bits needed by Huffman coding for each character of the input file.

2. generic edge from $i$ to $k > i + 1$: this edge is weighed with an integer value corresponding to the number of bits that a pointer to an entry of the dictionary
would require if that edge were chosen, that is, the result of the logarithm base 2 of the number of nodes inserted previously in LZ78 dictionary.

In both cases, it has to be added one, because it needs a directive file (that will be part of the final output file) for keeping trace of the point in which it has been chosen to encode with dictionary or with symbolwise technique.

7.3 Shortest Path

Once that the auxiliary graph has been built, it has to be computed the shortest path from the first byte to the last one. An algorithm for calculating the shortest path will return it in $O(|E|)$ time, because the graph is a Directed Acyclic Graph topologically sorted.

Dijkstra algorithm will return a path. This path is composed by edges of length one and by edges of length greater than one. For the first kind of edge, generically from $i$ to $i + 1$, the corresponding character $i$ has to be transmitted “in clear”. For the second kind of edge, it has to be transmitted the index of the corresponding entry of LZ78 dictionary. Since the auxiliary path potentially has a great number of edges, for an input file of some tens of MB the RAM of a normal PC could not be enough. So, the shortest path is computed on line, and all the edges that do not need no more are eliminated, for freeing some bytes of RAM. This is possible because the auxiliary graph is a Directed Acyclic Graph, and it is sure that there are no edges $(i, j)$ with $j < i$. So, once that a edge is visited, it could be eliminated. During the implementation of the program, this little detail was very important.

7.4 Writing the Output in Bits

Once that the optimal path is available, the output has to be written. Three files are created:

1. a directive file, written as specified in the next two points.

2. a file for the characters. Whenever the optimal parser chooses an arc $(i, i + 1)$, the related byte $i$ is written to a specific file, and a bit 0 is written in the directive file.

3. a file for the indexes to LZ78 dictionary. This value is certainly smaller than the number of bytes seen until generic byte $i$. So, $\log_2 i$ bits are certainly enough. A
but 1 has to be written to the directive file.

Next, those three files are compressed with Huffman, static or dynamic respect to the option selected. In general it is possible to obtain better results if the choice between static or dynamic is performed. The choice is written in the header of the output file.

7.5 Decompression Phase

Decompression phase was not so easy to implement, in particular because, while decompressing, it has to be built the dictionary of LZ78. It was quite difficult. Anyway, the first phase of the decompression requires to decompress the three files with Huffman, if they have been previously compressed. Next, bits are extracted from the three files in way to be easily manipulated, and the directive file is read. When a 1 is found it has to be written to the output file (i.e. the future original file) the string corresponding at that specific index in dictionary previously built, and a “piece” of dictionary has to be added. When a 0 is read in the directive file, a character has to be read from the file with “clear” characters and it has to be written to the output.

7.5.1 A Simulation with Arithmetic Coding

The program that performs compression and decompression mixing LZ78 and Huffman perfectly works. But, according to us it is interesting to simulate what could happen if the symbolwise technique were arithmetic. That is, in general, given the flexibility of our approach, it only needs to weigh the edges \((i, i + 1)\) in a different way. In the simulation we performed, we chose to use an order 1 arithmetic coding. Each edge \((i, i + 1)\) is, in fact, weighed with a float number equal to \(-\log_2 p(i/i - 1)\). In other words, the weight of each edge of length one reflects the probability that a generic byte \(i\) has to appear given byte \(i - 1\). The simulation, as it is possible to see in chapter ??, returned very good results. The only great disadvantage is connected to the fact that an arithmetic decoder takes much more time than a statistical one. But, as it always happens, it would be a choice tied to the context and to the objective it is wished to reach, that is: better compression ratio or (exclusive) decompression speed.

7.6 Why Using C

The program was implemented in C language for several reasons:
• It is very efficient, on the contrary, for example, of a language like JAVA.

• They are available a lot of low-level instructions that allow to treat in an efficient way the memory.

• We did not think that for realizing this program it was absolutely necessary an object-oriented language.

• Thanks to the capabilities of the GCC compiler, it is possible to compile the source code respect to the hardware it will run on.
Chapter 8

Experimental Results of Large GZip

8.1 Introduction

In this chapter we’d like to show the results that we obtained by compressing the files of some corpus with our programs. We will underline some results supporting theoretical predictions.

A corpus is a collection of files that has the purpose of pointing out the performance of a compression algorithm. These files exist because it wouldn’t be reasonable neither to examine performance on a few file chosen by programmers nor on a set of files of the same type. Every algorithm can measure its capability on a corpus, and in [6] it is possible to find the results of years and years of experimentation.

In particular we will focus on two corpus: Calgary and Canterbury (see [6] and [5]) and of enwik file set, a newly used benchmark file collection, based on English Wikipedia data base (see [51] and [33]). However, our results will be also given on other files that, according to us, represent a good benchmark for testing a compression algorithm, even in its extreme cases. It is important to underline that, although in the tests we will show space and time used by the program, we have a lot of points to be improved, that will be listed at the end of the chapter. For every file we will focus on, we will show also a comparison with gzip, with its maximum compression capability, i.e. with the parameter -9.

We didn’t take in consideration formats like mp3, mpeg, jpeg, and so on, because they are lossy, i.e. the decompression phase doesn’t succeed in obtaining the original file. Thus the compression leads to the loss of some information. We focused instead
on lossless techniques. Moreover, it would be quite devoid of sense to compress files in format cited so far, because, probably, the compressed file could even be bigger than the original one!

8.2 Parsing Optimality

A dictionary limited version of lgzip was built to appreciate compression gain on gzip original algorithm due by parsing optimality. Table 8.1 shows goodness of our optimal parsing with respect to Lazy Parsing done by Deflate in gzip algorithm. Obviously, our optimal parsing gain up to 4% of compression respect to gzip suboptimal parsing solution. Table 8.2 confirm optimality of our parsing, showing that lgzip acts very similarly to new implementation of gzip based on Deflate64 parsing [29] invented by Phil Katz, owner of PKWARE [40]. Biggest different is below 1%. This parsing methods appeared with PKZip compressor that produce very famous .zip archives, but it was also used by many gzip and zip compatible compressors suite like 7zip [1]. Deflate64 use a 64KB long dictionary, and optimal parsing though dictionary-complete graph path minimization, even if this statement are documented not so much because of royalty. Source code analysis support our declarations. Table 8.3 shows that lgzip 32KB limited version using dictionary-complete graph 7.1.3 is approaching more closely to Deflate64 based gzip of 7zip and that lgzip linear subgraph is not too far from graph complete solution. Lgzip 32k is below 0.8% at most respect to our lgzip 32K dictionary-complete version and lgzip 32K dictionary-complete is below 0.2% at most respect to Deflate64. Similar consideration may be done about Table 8.4 and Table 8.5 that compare lgzip 64k version to zip compressor of 7zip suite, implementing full size Deplate64 algorithm. After these results is evident that lgzip would be a good substitute candidate to gzip or zip, very most part of fast compression field quote, simply because does not worsens their competitors performance. What if we use all 16MB dictionary power?
<table>
<thead>
<tr>
<th>file</th>
<th>size (bytes)</th>
<th>lgzip 32K ratio</th>
<th>gzip -9 ratio</th>
<th>lgzip 32K/gzip</th>
</tr>
</thead>
<tbody>
<tr>
<td>bible.txt</td>
<td>4047392</td>
<td>27.63%</td>
<td>29.07%</td>
<td>95.05%</td>
</tr>
<tr>
<td>book1</td>
<td>768771</td>
<td>39.05%</td>
<td>40.62%</td>
<td>96.13%</td>
</tr>
<tr>
<td>E.coli</td>
<td>4638690</td>
<td>26.72%</td>
<td>28.01%</td>
<td>95.39%</td>
</tr>
<tr>
<td>pi.txt</td>
<td>1000000</td>
<td>45.57%</td>
<td>47.04%</td>
<td>96.88%</td>
</tr>
<tr>
<td>world192.txt</td>
<td>2473400</td>
<td>28.16%</td>
<td>29.17%</td>
<td>96.54%</td>
</tr>
<tr>
<td>enwik 0.5M</td>
<td>500000</td>
<td>33.9%</td>
<td>35.15%</td>
<td>96.44%</td>
</tr>
<tr>
<td>enwik 1M</td>
<td>1000000</td>
<td>34.23%</td>
<td>35.54%</td>
<td>96.31%</td>
</tr>
<tr>
<td>enwik 2M</td>
<td>2000000</td>
<td>34.76%</td>
<td>36.1%</td>
<td>96.29%</td>
</tr>
<tr>
<td>enwik 4M</td>
<td>4000000</td>
<td>35.08%</td>
<td>36.44%</td>
<td>96.27%</td>
</tr>
<tr>
<td>enwik 8M</td>
<td>8000000</td>
<td>35.47%</td>
<td>36.84%</td>
<td>96.28%</td>
</tr>
<tr>
<td>enwik 16M</td>
<td>16000000</td>
<td>35.48%</td>
<td>36.86%</td>
<td>96.26%</td>
</tr>
<tr>
<td>enwik 32M</td>
<td>32000000</td>
<td>35.23%</td>
<td>36.6%</td>
<td>96.26%</td>
</tr>
<tr>
<td>enwik 64M</td>
<td>64000000</td>
<td>35.12%</td>
<td>36.5%</td>
<td>96.22%</td>
</tr>
<tr>
<td>enwik 100M</td>
<td>100000000</td>
<td>35.08%</td>
<td>36.45%</td>
<td>96.24%</td>
</tr>
</tbody>
</table>

Table 8.1: Lgzip 32K win on gzip -9

<table>
<thead>
<tr>
<th>file</th>
<th>size (bytes)</th>
<th>lgzip 32K ratio</th>
<th>7z gzip ratio</th>
<th>lgzip 32K/7z gzip</th>
</tr>
</thead>
<tbody>
<tr>
<td>bible.txt</td>
<td>4047392</td>
<td>27.63%</td>
<td>27.44%</td>
<td>100.69%</td>
</tr>
<tr>
<td>book1</td>
<td>768771</td>
<td>39.05%</td>
<td>38.98%</td>
<td>100.18%</td>
</tr>
<tr>
<td>E.coli</td>
<td>4638690</td>
<td>26.72%</td>
<td>26.65%</td>
<td>100.26%</td>
</tr>
<tr>
<td>pi.txt</td>
<td>1000000</td>
<td>45.57%</td>
<td>45.61%</td>
<td>99.91%</td>
</tr>
<tr>
<td>world192.txt</td>
<td>2473400</td>
<td>28.16%</td>
<td>28.11%</td>
<td>100.18%</td>
</tr>
<tr>
<td>enwik 0.5M</td>
<td>500000</td>
<td>33.9%</td>
<td>33.86%</td>
<td>100.12%</td>
</tr>
<tr>
<td>enwik 1M</td>
<td>1000000</td>
<td>34.23%</td>
<td>34.18%</td>
<td>100.15%</td>
</tr>
<tr>
<td>enwik 2M</td>
<td>2000000</td>
<td>34.76%</td>
<td>34.74%</td>
<td>100.06%</td>
</tr>
<tr>
<td>enwik 4M</td>
<td>4000000</td>
<td>35.08%</td>
<td>35.07%</td>
<td>100.03%</td>
</tr>
<tr>
<td>enwik 8M</td>
<td>8000000</td>
<td>35.47%</td>
<td>35.47%</td>
<td>100%</td>
</tr>
<tr>
<td>enwik 16M</td>
<td>16000000</td>
<td>35.48%</td>
<td>35.48%</td>
<td>100%</td>
</tr>
<tr>
<td>enwik 32M</td>
<td>32000000</td>
<td>35.23%</td>
<td>35.23%</td>
<td>100%</td>
</tr>
<tr>
<td>enwik 64M</td>
<td>64000000</td>
<td>35.12%</td>
<td>35.11%</td>
<td>100.03%</td>
</tr>
<tr>
<td>enwik 100M</td>
<td>100000000</td>
<td>35.08%</td>
<td>35.06%</td>
<td>100.06%</td>
</tr>
</tbody>
</table>

Table 8.2: Lgzip 32K very close to 7z gzip
<table>
<thead>
<tr>
<th>file</th>
<th>size (bytes)</th>
<th>lgzip 64K ratio</th>
<th>7z zip ratio</th>
<th>lgzip 64K/7z zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>bible.txt</td>
<td>4047392</td>
<td>26,2%</td>
<td>25,99%</td>
<td>100,81%</td>
</tr>
<tr>
<td>book1</td>
<td>768771</td>
<td>37,46%</td>
<td>37,41%</td>
<td>100,13%</td>
</tr>
<tr>
<td>E.coli</td>
<td>4638690</td>
<td>26,56%</td>
<td>26,7%</td>
<td>99,48%</td>
</tr>
<tr>
<td>pi.txt</td>
<td>1000000</td>
<td>45,81%</td>
<td>45,69%</td>
<td>100,26%</td>
</tr>
<tr>
<td>world192.txt</td>
<td>2473400</td>
<td>26,01%</td>
<td>25,97%</td>
<td>100,15%</td>
</tr>
<tr>
<td>enwik 0.5M</td>
<td>500000</td>
<td>32,59%</td>
<td>32,56%</td>
<td>100,09%</td>
</tr>
<tr>
<td>enwik 1M</td>
<td>1000000</td>
<td>32,81%</td>
<td>32,77%</td>
<td>100,12%</td>
</tr>
<tr>
<td>enwik 2M</td>
<td>2000000</td>
<td>33,37%</td>
<td>33,33%</td>
<td>100,12%</td>
</tr>
<tr>
<td>enwik 4M</td>
<td>4000000</td>
<td>33,7%</td>
<td>33,67%</td>
<td>100,09%</td>
</tr>
<tr>
<td>enwik 8M</td>
<td>8000000</td>
<td>34,11%</td>
<td>34,09%</td>
<td>100,06%</td>
</tr>
<tr>
<td>enwik 16M</td>
<td>16000000</td>
<td>34,19%</td>
<td>34,16%</td>
<td>100,09%</td>
</tr>
<tr>
<td>enwik 32M</td>
<td>32000000</td>
<td>33,93%</td>
<td>33,9%</td>
<td>100,09%</td>
</tr>
<tr>
<td>enwik 64M</td>
<td>64000000</td>
<td>33,82%</td>
<td>33,78%</td>
<td>100,12%</td>
</tr>
<tr>
<td>enwik 100M</td>
<td>100000000</td>
<td>33,77%</td>
<td>33,72%</td>
<td>100,15%</td>
</tr>
</tbody>
</table>

Table 8.4: Lgzip 64K very close to 7z zip
<table>
<thead>
<tr>
<th>file</th>
<th>size (bytes)</th>
<th>dc_lgzip 64K ratio</th>
<th>7z ratio</th>
<th>dc_lgzip 64K/7z zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>bible.txt</td>
<td>4047392</td>
<td>26,05%</td>
<td>25,99%</td>
<td>100,23%</td>
</tr>
<tr>
<td>book1</td>
<td>768771</td>
<td>37,38%</td>
<td>37,41%</td>
<td>99,92%</td>
</tr>
<tr>
<td>E.coli</td>
<td>4638690</td>
<td>26,51%</td>
<td>26,7%</td>
<td>99,29%</td>
</tr>
<tr>
<td>pi.txt</td>
<td>1000000</td>
<td>45,81%</td>
<td>45,69%</td>
<td>100,26%</td>
</tr>
<tr>
<td>world192.txt</td>
<td>2473400</td>
<td>26%</td>
<td>25,97%</td>
<td>100,12%</td>
</tr>
<tr>
<td>enwik 0.5M</td>
<td>500000</td>
<td>32,56%</td>
<td>32,56%</td>
<td>100%</td>
</tr>
<tr>
<td>enwik 1M</td>
<td>1000000</td>
<td>32,78%</td>
<td>32,77%</td>
<td>100,03%</td>
</tr>
<tr>
<td>enwik 2M</td>
<td>2000000</td>
<td>33,33%</td>
<td>33,33%</td>
<td>100%</td>
</tr>
<tr>
<td>enwik 4M</td>
<td>4000000</td>
<td>33,66%</td>
<td>33,67%</td>
<td>99,97%</td>
</tr>
<tr>
<td>enwik 8M</td>
<td>8000000</td>
<td>34,07%</td>
<td>34,09%</td>
<td>99,94%</td>
</tr>
<tr>
<td>enwik 16M</td>
<td>16000000</td>
<td>34,14%</td>
<td>34,16%</td>
<td>99,94%</td>
</tr>
<tr>
<td>enwik 32M</td>
<td>32000000</td>
<td>33,89%</td>
<td>33,9%</td>
<td>99,97%</td>
</tr>
<tr>
<td>enwik 64M</td>
<td>64000000</td>
<td>33,78%</td>
<td>33,78%</td>
<td>100%</td>
</tr>
<tr>
<td>enwik 100M</td>
<td>100000000</td>
<td>33,73%</td>
<td>33,72%</td>
<td>100,03%</td>
</tr>
</tbody>
</table>

Table 8.5: Dictionary Complete Lgzip 64K equal to 7z zip

8.3 Gain At Dictionary Growing

What follow answers to last question. Table 8.6, Table 8.7, and Table 8.8 shows as gain grows following dictionary growing. This definitely confirm lgzip performance is better than LZ-based principal algorithms, like gzip and zip packers. Doubling dictionary size lead to 5%..2% improvement, each time it is done. Overall gain is around 8.5% with respect to 32k version on Wikipedia file, that mean about 25% better than gzip and zip compressors. Secondarily, Table 8.7 shows that there is room enough for future enhancements in dictionary growing direction.

8.4 Comparison with Other Compressors

This table shows that lgzip large dictionary and optimal parsing stay always on top of zip compression rate and overpass bzip2 performance on big files. LZMA approaching enforce our dictionary strategy, considering that LZMA use a dictionary of 32MB. Accordingly with our analysis in follow Section 8.6, low symbolwise impact factor results in similar compression ratio even if LZMA symbolwise compression is stronger than Huffman coding. We do not say nothing about LZMA parsing, as we would like to perform further experimentation exactly in aritmetic symbolwise compression direction. ***
<table>
<thead>
<tr>
<th>dic. size</th>
<th>lgzip rate</th>
<th>lgzip/gzip</th>
<th>Δ</th>
<th>lgzip/zip</th>
<th>Δ</th>
</tr>
</thead>
<tbody>
<tr>
<td>32k</td>
<td>39.05%</td>
<td>96.13%</td>
<td>104.4%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>64k</td>
<td>37.46%</td>
<td>92.22%</td>
<td>4.25%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>128k</td>
<td>36.18%</td>
<td>89.97%</td>
<td>3.43%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>256k</td>
<td>35.2%</td>
<td>86.66%</td>
<td>2.62%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>512k</td>
<td>34.64%</td>
<td>85.28%</td>
<td>1.49%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1M</td>
<td>34.58%</td>
<td>85.13%</td>
<td>0.16%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2M</td>
<td>34.58%</td>
<td>85.13%</td>
<td>0%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4M</td>
<td>34.58%</td>
<td>85.13%</td>
<td>0%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8M</td>
<td>34.58%</td>
<td>85.13%</td>
<td>0%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16M</td>
<td>34.58%</td>
<td>85.13%</td>
<td>0%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tot. Gain</td>
<td>4.47%</td>
<td>11%</td>
<td>11.95%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 8.6: Results at Dictionary grow on book1 (770KB)

<table>
<thead>
<tr>
<th>dic. size</th>
<th>lgzip rate</th>
<th>lgzip/gzip</th>
<th>Δ</th>
<th>lgzip/zip</th>
<th>Δ</th>
</tr>
</thead>
<tbody>
<tr>
<td>32k</td>
<td>27.63%</td>
<td>95.04%</td>
<td>106.33%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>64k</td>
<td>26.2%</td>
<td>90.12%</td>
<td>4.51%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>128k</td>
<td>25.01%</td>
<td>86.03%</td>
<td>4.58%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>256k</td>
<td>24.02%</td>
<td>82.62%</td>
<td>3.81%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>512k</td>
<td>23.35%</td>
<td>80.32%</td>
<td>2.57%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1M</td>
<td>22.89%</td>
<td>78.74%</td>
<td>1.77%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2M</td>
<td>22.59%</td>
<td>77.7%</td>
<td>1.16%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4M</td>
<td>22.51%</td>
<td>77.43%</td>
<td>0.31%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8M</td>
<td>22.51%</td>
<td>77.43%</td>
<td>0%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16M</td>
<td>22.51%</td>
<td>77.43%</td>
<td>0%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tot. Gain</td>
<td>5.12%</td>
<td>17.61%</td>
<td>19.71%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 8.7: Results at Dictionary grow on bible.txt (4MB)

<table>
<thead>
<tr>
<th>dic. size</th>
<th>lgzip rate</th>
<th>lgzip/gzip</th>
<th>Δ</th>
<th>lgzip/zip</th>
<th>Δ</th>
</tr>
</thead>
<tbody>
<tr>
<td>32k</td>
<td>35.08%</td>
<td>96.25%</td>
<td>104.03%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>64k</td>
<td>33.77%</td>
<td>92.66%</td>
<td>3.89%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>128k</td>
<td>32.62%</td>
<td>89.5%</td>
<td>3.41%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>256k</td>
<td>31.55%</td>
<td>86.57%</td>
<td>3.17%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>512k</td>
<td>30.58%</td>
<td>83.91%</td>
<td>2.88%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1M</td>
<td>29.68%</td>
<td>81.44%</td>
<td>2.66%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2M</td>
<td>28.82%</td>
<td>79.08%</td>
<td>2.29%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4M</td>
<td>28.05%</td>
<td>76.96%</td>
<td>2.29%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8M</td>
<td>27.3%</td>
<td>74.91%</td>
<td>2.22%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16M</td>
<td>26.59%</td>
<td>72.96%</td>
<td>2.11%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tot. Gain</td>
<td>8.49%</td>
<td>23.29%</td>
<td>25.18%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 8.8: Results at Dictionary grow on enwik (100MB)
8.5 Corpus Results

For sake of completeness we report traditional Corpus results. Results on these files are limited by their little sizes. We can anyway state that lgzip gain is around 5% on little file, like web pages are, and gain rise to 27% on 100MB file as Wikipedia results shown.

8.5.1 Calgary Corpus

This corpus is used to evaluate the practical performance of various text compression schemes. Several other researchers are now using the corpus to evaluate text compression schemes.

Nine different types of text are represented, and to confirm that the performance of schemes is consistent for any given type, many of the types have more than one that is representative. Normal English, both fiction and non-fiction, is represented by two books and papers (labeled book1, book2, paper1, paper2, paper3, paper4, paper5, paper6). More unusual styles of English writing are found in a bibliography (bib) and a batch of unedited news articles (news). Three computer programs represent artificial languages (progc, progl, progp). A transcript of a terminal session (trans) is included to indicate the increase in speed that could be achieved by applying compression to a slow line to a terminal. All of the files mentioned so far use ASCII encoding. Some non-ASCII files are also included: two files of executable code (obj1, obj2), some geophysical data (geo), and a bit-map black and white picture (pic). The file geo is particularly difficult to compress because it contains a wide range of data values, while the file pic is highly compressible because of large amounts of white space in the picture, represented by long runs of zeros.

<table>
<thead>
<tr>
<th>file name</th>
<th>size</th>
<th>zip</th>
<th>bzip2</th>
<th>lgzip</th>
<th>LZMA</th>
<th>PPMd</th>
</tr>
</thead>
<tbody>
<tr>
<td>bible.txt</td>
<td>4M</td>
<td>25.99%</td>
<td>20.87%</td>
<td>22.51%</td>
<td>21.88%</td>
<td>18.47%</td>
</tr>
<tr>
<td>book1</td>
<td>770K</td>
<td>37.41%</td>
<td>30.24%</td>
<td>34.58%</td>
<td>34%</td>
<td>27.31%</td>
</tr>
<tr>
<td>enwik</td>
<td>100M</td>
<td>33.72%</td>
<td>29.01%</td>
<td>26.59%</td>
<td>25.73%</td>
<td>24.85%</td>
</tr>
</tbody>
</table>

Table 8.9: Comparison with Other Compressors
<table>
<thead>
<tr>
<th>file</th>
<th>size (bytes)</th>
<th>lgzip ratio</th>
<th>gzip -9 ratio</th>
<th>lgzip/gzip</th>
</tr>
</thead>
<tbody>
<tr>
<td>bib</td>
<td>111261</td>
<td>28.85%</td>
<td>31.37%</td>
<td>91.97%</td>
</tr>
<tr>
<td>book1</td>
<td>768771</td>
<td>34.58%</td>
<td>40.62%</td>
<td>85.13%</td>
</tr>
<tr>
<td>book2</td>
<td>610856</td>
<td>28.71%</td>
<td>33.75%</td>
<td>85.07%</td>
</tr>
<tr>
<td>geo</td>
<td>102400</td>
<td>64.12%</td>
<td>66.81%</td>
<td>95.97%</td>
</tr>
<tr>
<td>news</td>
<td>377109</td>
<td>32.68%</td>
<td>38.29%</td>
<td>85.35%</td>
</tr>
<tr>
<td>obj1</td>
<td>21504</td>
<td>47.83%</td>
<td>47.99%</td>
<td>99.67%</td>
</tr>
<tr>
<td>obj2</td>
<td>246814</td>
<td>30.63%</td>
<td>32.85%</td>
<td>93.24%</td>
</tr>
<tr>
<td>paper1</td>
<td>53161</td>
<td>33.23%</td>
<td>34.88%</td>
<td>95.27%</td>
</tr>
<tr>
<td>paper2</td>
<td>82199</td>
<td>33.7%</td>
<td>36.09%</td>
<td>93.38%</td>
</tr>
<tr>
<td>paper3</td>
<td>46526</td>
<td>37.11%</td>
<td>38.85%</td>
<td>95.52%</td>
</tr>
<tr>
<td>paper4</td>
<td>13286</td>
<td>40.44%</td>
<td>41.65%</td>
<td>97.09%</td>
</tr>
<tr>
<td>paper5</td>
<td>11954</td>
<td>40.94%</td>
<td>41.79%</td>
<td>97.97%</td>
</tr>
<tr>
<td>paper6</td>
<td>38105</td>
<td>33.54%</td>
<td>34.68%</td>
<td>96.71%</td>
</tr>
<tr>
<td>pic</td>
<td>513216</td>
<td>9.51%</td>
<td>10.21%</td>
<td>93.14%</td>
</tr>
<tr>
<td>progc</td>
<td>39611</td>
<td>32.58%</td>
<td>33.48%</td>
<td>97.31%</td>
</tr>
<tr>
<td>progl</td>
<td>71646</td>
<td>21.54%</td>
<td>22.56%</td>
<td>95.48%</td>
</tr>
<tr>
<td>progp</td>
<td>49379</td>
<td>21.59%</td>
<td>22.65%</td>
<td>95.32%</td>
</tr>
<tr>
<td>trans</td>
<td>93695</td>
<td>18.87%</td>
<td>20.13%</td>
<td>93.74%</td>
</tr>
</tbody>
</table>

Table 8.10: Results of lgzip on Calgary Corpus

### 8.5.2 Canterbury Corpus

The Canterbury Corpus file set has been developed specifically for testing new compression algorithms. The files, as in the case of Calgary corpus, were selected basing on their ability to provide representative performance results.

This set of files is designed to replace the Calgary Corpus, which is now over ten years old. However, criteria for choosing the file are the same of Calgary’s one.
<table>
<thead>
<tr>
<th>file</th>
<th>size (bytes)</th>
<th>lgzip ratio</th>
<th>gzip -9 ratio</th>
<th>lgzip/gzip</th>
</tr>
</thead>
<tbody>
<tr>
<td>alice29.txt</td>
<td>152089</td>
<td>32.49%</td>
<td>35.63%</td>
<td>91.19%</td>
</tr>
<tr>
<td>asyoulik.txt</td>
<td>125179</td>
<td>36.07%</td>
<td>39.01%</td>
<td>92.46%</td>
</tr>
<tr>
<td>cp.html</td>
<td>24603</td>
<td>31.7%</td>
<td>32.44%</td>
<td>97.72%</td>
</tr>
<tr>
<td>fields.c</td>
<td>11150</td>
<td>27.6%</td>
<td>28.13%</td>
<td>98.12%</td>
</tr>
<tr>
<td>grammar.lsp</td>
<td>3721</td>
<td>32.63%</td>
<td>33.49%</td>
<td>97.43%</td>
</tr>
<tr>
<td>kennedy.xls</td>
<td>1029744</td>
<td>17.29%</td>
<td>20.37%</td>
<td>84.88%</td>
</tr>
<tr>
<td>lcet10.txt</td>
<td>426754</td>
<td>28.69%</td>
<td>33.84%</td>
<td>84.78%</td>
</tr>
<tr>
<td>plrabn12.txt</td>
<td>481861</td>
<td>34.9%</td>
<td>40.32%</td>
<td>86.56%</td>
</tr>
<tr>
<td>ptt5</td>
<td>513216</td>
<td>9.51%</td>
<td>10.21%</td>
<td>93.14%</td>
</tr>
<tr>
<td>sum</td>
<td>38240</td>
<td>31.85%</td>
<td>33.4%</td>
<td>95.36%</td>
</tr>
<tr>
<td>xargs.1</td>
<td>4227</td>
<td>40.67%</td>
<td>41.54%</td>
<td>97.91%</td>
</tr>
</tbody>
</table>

Table 8.11: Results of lgzip on Canterbury Corpus

### 8.5.3 Packing English Wikipedia File

In this section they are shown the results obtained by executing the program with input file English Wikipedia. In particular, for observing the behavior of the program on different size and for appreciating the improvement of the program as the size of the input file grows, the test has been performed with the first size bytes of English Wikipedia, also simply called enwik.

<table>
<thead>
<tr>
<th>file</th>
<th>size (bytes)</th>
<th>lgzip ratio</th>
<th>gzip -9 ratio</th>
<th>lgzip/gzip</th>
</tr>
</thead>
<tbody>
<tr>
<td>enwik0</td>
<td>1</td>
<td>3100%</td>
<td>2800%</td>
<td>110.71%</td>
</tr>
<tr>
<td>enwik1</td>
<td>10</td>
<td>400%</td>
<td>370%</td>
<td>108.11%</td>
</tr>
<tr>
<td>enwik2</td>
<td>100</td>
<td>108%</td>
<td>101%</td>
<td>106.93%</td>
</tr>
<tr>
<td>enwik3</td>
<td>1K</td>
<td>34.2%</td>
<td>34.3%</td>
<td>99.71%</td>
</tr>
<tr>
<td>enwik4</td>
<td>10K</td>
<td>36.77%</td>
<td>37.22%</td>
<td>98.79%</td>
</tr>
<tr>
<td>enwik5</td>
<td>100K</td>
<td>33.81%</td>
<td>36.07%</td>
<td>93.73%</td>
</tr>
<tr>
<td>enwik6</td>
<td>1M</td>
<td>30.24%</td>
<td>35.54%</td>
<td>85.09%</td>
</tr>
<tr>
<td>enwik7</td>
<td>10M</td>
<td>28.29%</td>
<td>36.85%</td>
<td>76.77%</td>
</tr>
<tr>
<td>enwik8</td>
<td>100M0</td>
<td>26.59%</td>
<td>36.45%</td>
<td>72.95%</td>
</tr>
<tr>
<td>enwik9</td>
<td>1G</td>
<td>23.23%</td>
<td>32.26%</td>
<td>72.01%</td>
</tr>
</tbody>
</table>

Table 8.12: Results of lgzip on enwik files
8.6 Pure Dictionary Simulation

The aim of this section is to show how a pure LZ77 Dictionary algorithm will get benefit from Dictionary-Symbolwise improved version. We can just run a simulation on lgzip giving an huge cost to symbolwise arcs in the graph, forcing it to chose anytime is possible a dictionary arc rather than a symbolwise one. This because the intrinsic dictionary-symbolwise nature of gzip and lgzip too. 1% better compression ratio, gain of 2.5% on gzip compression is measure of compression enhancement switching from “pure” dictionary to dictionary-symbolwise algorithm. This still important result suffer for LZ77 power, i.e. to add one freedom degree to an algorithm with a so rich dictionary lead to little but important gain over compression. A rich dictionary cause that dictionary arcs are frequently preferred during parsing phase, showed by symbolwise average impact of 1.5% on experiment reported in Table 8.13. This thesis is supported by following results on Optimal-LZ78, where a poor dictionary, with respect to LZ77 one, it leaves symbolwise impact rise to 5%. Symbolwise impact is number of symbolwise arcs selected in the coding parse (one character each in lgzip) on input file size ratio.

<table>
<thead>
<tr>
<th>file</th>
<th>pd_lgzip</th>
<th>lgzip</th>
<th>pd_lgzip/gzip</th>
<th>lgzip/gzip</th>
<th>Δ</th>
<th>S. I.F.</th>
</tr>
</thead>
<tbody>
<tr>
<td>enwik 0.5M</td>
<td>32.04%</td>
<td>30.9%</td>
<td>91.13%</td>
<td>87.89%</td>
<td>3.24%</td>
<td>3.02%</td>
</tr>
<tr>
<td>enwik 1M</td>
<td>31.32%</td>
<td>30.25%</td>
<td>88.13%</td>
<td>85.13%</td>
<td>3%</td>
<td>2.88%</td>
</tr>
<tr>
<td>enwik 2M</td>
<td>30.84%</td>
<td>29.78%</td>
<td>85.42%</td>
<td>82.48%</td>
<td>2.94%</td>
<td>2.54%</td>
</tr>
<tr>
<td>enwik 4M</td>
<td>30.26%</td>
<td>29.31%</td>
<td>83.05%</td>
<td>80.44%</td>
<td>2.61%</td>
<td>2.27%</td>
</tr>
<tr>
<td>enwik 8M</td>
<td>29.48%</td>
<td>28.62%</td>
<td>80.03%</td>
<td>77.68%</td>
<td>2.35%</td>
<td>2.01%</td>
</tr>
<tr>
<td>enwik 16M</td>
<td>28.66%</td>
<td>27.77%</td>
<td>77.76%</td>
<td>75.34%</td>
<td>2.42%</td>
<td>1.81%</td>
</tr>
<tr>
<td>enwik 32M</td>
<td>27.99%</td>
<td>27.08%</td>
<td>76.47%</td>
<td>73.98%</td>
<td>2.49%</td>
<td>1.59%</td>
</tr>
<tr>
<td>enwik 64M</td>
<td>27.64%</td>
<td>26.72%</td>
<td>75.73%</td>
<td>73.21%</td>
<td>2.52%</td>
<td>1.42%</td>
</tr>
<tr>
<td>enwik 100M</td>
<td>27.5%</td>
<td>26.59%</td>
<td>75.47%</td>
<td>72.95%</td>
<td>2.52%</td>
<td>1.20%</td>
</tr>
</tbody>
</table>

Table 8.13: Gain at Symbolwise Improvement on enwik files
Chapter 9

Experimental Results of Optimal LZ78

9.1 Calgary Corpus

In general we can notice that best results are achieved with files that have a size more than some hundreds of KB. This happens because the window has a dimension equal to file’s size and allows the program to individuate matches in every just parsed portion of the input. On “short” (some tens of KB) files, gzip -9 often wins, in particular thanks to its very good bit-optimization, but also because of the use of LZ77. Our program could take the advantages of an arithmetic coding (like PPM) that would certainly improve the actual results, as it is possible to see next in simulation’s results.

From a theoretical point of view, pure LZ77 (that is, the version in which there is a greedy parsing) converges to entropy slower than LZ78 in its standard version. This means that, from a theoretical point of view, LZ78 should obtain better compression ratio respect to LZ77 on every file.

LZ77 uses a dictionary that is greater than LZ78’s one. In fact, LZ77 uses all the possible substrings of the portion of input text parsed previously, while LZ78’s dictionary has got only some of the substring used by LZ77. So, if both LZ77 and LZ78 schemes can take the advantage of optimal parsing, the LZ77 scheme has a greater gain with respect to greedy parsing. Indeed, even if the LZ78 scheme improves its performance using an optimal parsing, this improvement is not even comparable with the one obtained by the algorithm in the LZ77 scheme that uses an optimal parsing. When it is added the possibility to make an optimal parsing using also a statistical technique (mixing, for example, dictionary and Huffman) in a dictionary-
symbolwise scheme, the scheme that takes the main advantages of this flexibility is LZ78. LZ77 scheme, that had already a great advantage from optimal parsing without symbolwise, has not a great gain when adding a symbolwise technique in optimal parsing but still has a better compression ratio with respect to the LZ78 one. So, we can affirm that, thanks to the symbolwise technique, the “distance” between LZ77 and LZ78 decreases. Anyway, given the results observed, LZ77 with optimal parsing (dictionary-symbolwise) continues to have a certain advantage on LZ78 with optimal parsing (dictionary-symbolwise). It has to be noticed that, in this tests, the statistical technique used is Huffman. With a more powerful symbolwise, in general, overall compression ratio decreases significantly and the two schemes could get even closer than now.

<table>
<thead>
<tr>
<th>file</th>
<th>size (bytes)</th>
<th>lz78 ratio</th>
<th>gzip -9 ratio</th>
<th>lz78 / gzip</th>
</tr>
</thead>
<tbody>
<tr>
<td>bib</td>
<td>111261</td>
<td>38.920%</td>
<td>31.375%</td>
<td>124.049%</td>
</tr>
<tr>
<td>book1</td>
<td>768771</td>
<td>39.524%</td>
<td>40.622%</td>
<td>97.298%</td>
</tr>
<tr>
<td>book2</td>
<td>610856</td>
<td>37.648%</td>
<td>33.750%</td>
<td>111.549%</td>
</tr>
<tr>
<td>geo</td>
<td>102400</td>
<td>66.453%</td>
<td>66.818%</td>
<td>99.453%</td>
</tr>
<tr>
<td>news</td>
<td>377109</td>
<td>44.017%</td>
<td>38.293%</td>
<td>114.947%</td>
</tr>
<tr>
<td>obj1</td>
<td>21504</td>
<td>59.510%</td>
<td>48.028%</td>
<td>123.906%</td>
</tr>
<tr>
<td>obj2</td>
<td>246814</td>
<td>46.597%</td>
<td>32.857%</td>
<td>141.818%</td>
</tr>
<tr>
<td>paper1</td>
<td>53161</td>
<td>45.232%</td>
<td>34.896%</td>
<td>129.621%</td>
</tr>
<tr>
<td>paper2</td>
<td>82199</td>
<td>42.250%</td>
<td>36.101%</td>
<td>117.031%</td>
</tr>
<tr>
<td>paper3</td>
<td>46526</td>
<td>46.037%</td>
<td>38.864%</td>
<td>118.455%</td>
</tr>
<tr>
<td>paper4</td>
<td>13286</td>
<td>49.782%</td>
<td>41.713%</td>
<td>119.343%</td>
</tr>
<tr>
<td>paper5</td>
<td>11954</td>
<td>51.849%</td>
<td>41.852%</td>
<td>123.886%</td>
</tr>
<tr>
<td>paper6</td>
<td>38105</td>
<td>46.640%</td>
<td>34.696%</td>
<td>134.423%</td>
</tr>
<tr>
<td>pic</td>
<td>513216</td>
<td>11.200%</td>
<td>10.208%</td>
<td>109.720%</td>
</tr>
<tr>
<td>progc</td>
<td>39611</td>
<td>45.977%</td>
<td>33.498%</td>
<td>137.252%</td>
</tr>
<tr>
<td>progl</td>
<td>71646</td>
<td>35.949%</td>
<td>22.572%</td>
<td>159.263%</td>
</tr>
<tr>
<td>progp</td>
<td>49379</td>
<td>37.202%</td>
<td>22.670%</td>
<td>164.106%</td>
</tr>
<tr>
<td>trans</td>
<td>93695</td>
<td>37.519%</td>
<td>20.140%</td>
<td>186.290%</td>
</tr>
</tbody>
</table>

Table 9.1: Results on Calgary Corpus

9.2 Canterbury Corpus

The Canterbury Corpus file set has been developed specifically for testing new compression algorithms. The files, as in the case of Calgary corpus, were selected basing on their ability to provide representative performance results.
This set of files is designed to replace the Calgary Corpus, which is now over ten years old. However, criteria for choosing the file are the same of Calgary’s one.

**Results**

In Canterbury corpus file, we can notice the same tendency that is observable in Calgary corpus. In fact, on short files (some tens of KB), gzip achieves better compression. We must wait that file size increase to appreciate a better performance of our program. We repeat that this is caused by the unbounded window that characterizes our match-research.

From a theoretical point of view, pure LZ77 (that is, the version in which there is a greedy parsing) converges to entropy slower than LZ78 in its standard version. This means that, from a theoretical point of view, LZ77 should obtain better compression ratio respect to LZ77 on every file.

LZ77 uses a dictionary that is greater than LZ78’s one. In fact, LZ77 uses all the possible substrings of the portion of input text parsed previously, while LZ78’s dictionary has got only some of the substring used by LZ77. So, if both LZ77 and LZ78 schemes can take the advantage of optimal parsing, the LZ77 scheme has a greater gain with respect to greedy parsing. Indeed, even if the LZ78 scheme improves its performance using an optimal parsing, this improvement is not even comparable with the one obtained by the algorithm in the LZ77 scheme that uses an optimal parsing. When it is added the possibility to make an optimal parsing using also a statistical technique (mixing, for example, dictionary and Huffman) in a dictionary-symbolwise scheme, the scheme that takes the main advantages of this flexibility is LZ78. LZ77 scheme, that had already a great advantage from optimal parsing without symbolwise, has not a great gain when adding a symbolwise technique in optimal parsing but still has a better compression ratio with respect to the LZ78 one. So, we can affirm that, thanks to the symbolwise technique, the “distance” between LZ77 and LZ78 decreases. Anyway, given the results observed, LZ77 with optimal parsing (dictionary-symbolwise) continues to have a certain advantage on LZ78 with optimal parsing (dictionary-symbolwise). It has to be noticed that, in this tests, the statistical technique used is Huffman. With a more powerful symbolwise, in general, overall compression ratio decreases significantly and the two schemes could get even closer than now.
### Table 9.2: Results on Canterbury Corpus

<table>
<thead>
<tr>
<th>file</th>
<th>size (bytes)</th>
<th>LZ78 ratio</th>
<th>gzip -9 ratio</th>
<th>LZ78 / gzip</th>
</tr>
</thead>
<tbody>
<tr>
<td>alice29.txt</td>
<td>152089</td>
<td>39.880%</td>
<td>35.634%</td>
<td>111.914%</td>
</tr>
<tr>
<td>asyoulik.txt</td>
<td>125179</td>
<td>42.636%</td>
<td>39.011%</td>
<td>109.291%</td>
</tr>
<tr>
<td>cp.html</td>
<td>24603</td>
<td>43.141%</td>
<td>32.459%</td>
<td>132.908%</td>
</tr>
<tr>
<td>fields.c</td>
<td>11150</td>
<td>43.193%</td>
<td>28.170%</td>
<td>153.327%</td>
</tr>
<tr>
<td>grammar.lsp</td>
<td>3721</td>
<td>46.385%</td>
<td>33.620%</td>
<td>137.970%</td>
</tr>
<tr>
<td>kennedy.xls</td>
<td>1029744</td>
<td>25.276%</td>
<td>20.368%</td>
<td>124.094%</td>
</tr>
<tr>
<td>lct10.txt</td>
<td>426754</td>
<td>36.888%</td>
<td>33.845%</td>
<td>108.990%</td>
</tr>
<tr>
<td>plrabn12.txt</td>
<td>481861</td>
<td>39.751%</td>
<td>40.319%</td>
<td>98.592%</td>
</tr>
<tr>
<td>pt5</td>
<td>513216</td>
<td>11.200%</td>
<td>10.208%</td>
<td>109.724%</td>
</tr>
<tr>
<td>sum</td>
<td>38240</td>
<td>48.394%</td>
<td>33.413%</td>
<td>144.838%</td>
</tr>
<tr>
<td>xargs.1</td>
<td>4227</td>
<td>53.158%</td>
<td>41.661%</td>
<td>127.598%</td>
</tr>
</tbody>
</table>

### 9.3 Test English Wikipedia File

This test if very useful for noticing the influence of the unbounded window on the final compression ratio of the input files. In fact, the size of the input file is inversely proportional to the compression ratio. In particular, it is interesting looking at the final compression ratio on the whole 100 MB of English Wikipedia, that is, in absolute, a very good result, also given that GZip performs an excellent bit compression. But, GZip pays a lot the size of the window, that is of some tens of KB.

### Table 9.3: Results on \( n \) bytes of English Wikipedia

<table>
<thead>
<tr>
<th>size (bytes)</th>
<th>GZip bytes</th>
<th>LZ78 with OP bytes</th>
<th>GZip / LZ78 OP</th>
</tr>
</thead>
<tbody>
<tr>
<td>500000</td>
<td>177692</td>
<td>175774</td>
<td>101.091%</td>
</tr>
<tr>
<td>1000000</td>
<td>395963</td>
<td>355365</td>
<td>111.424%</td>
</tr>
<tr>
<td>2000000</td>
<td>775662</td>
<td>775662</td>
<td>107.42%</td>
</tr>
<tr>
<td>4000000</td>
<td>1515520</td>
<td>1457605</td>
<td>103.973%</td>
</tr>
<tr>
<td>8000000</td>
<td>2928828</td>
<td>2947445</td>
<td>99.368%</td>
</tr>
<tr>
<td>16000000</td>
<td>5664281</td>
<td>5897725</td>
<td>96.042%</td>
</tr>
<tr>
<td>32000000</td>
<td>11018001</td>
<td>11018001</td>
<td>94.068%</td>
</tr>
<tr>
<td>64000000</td>
<td>21375386</td>
<td>23358735</td>
<td>91.509%</td>
</tr>
<tr>
<td>100000000</td>
<td>32793312</td>
<td>36445258</td>
<td>89.980%</td>
</tr>
</tbody>
</table>

104
9.4 Gain With Symbolwise Ability

In this section a comparison between the normal program and the program with a variant of the program are shown. In the variant of the program, the weight of the characters is set to 50000, in way to not make the program choose a single character during the optimal parsing. In other words, it is performed a compression with something of very similar to standard LZ78 pure dictionary. From the results obtained it is possible to evince that the gain due to the optimal parsing mixing a statistical method and LZ78 is appreciable. In particular, it is interesting to notice that the best gain is obtained on an input if a little size, 500 KB, while it diminishes with the growth of the size of the input file. This is because on little files the dictionary of LZ78 has not a great number of entries, and, thus, it results advantageous to encode a lot of characters with Huffman.

<table>
<thead>
<tr>
<th>size</th>
<th>LZ78 / GZip</th>
<th>Optimal Parsing / GZip</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000000</td>
<td>121.819%</td>
<td>101.091%</td>
</tr>
<tr>
<td>1000000</td>
<td>117.149%</td>
<td>111.424%</td>
</tr>
<tr>
<td>2000000</td>
<td>112.976%</td>
<td>107.428%</td>
</tr>
<tr>
<td>4000000</td>
<td>109.291%</td>
<td>103.973%</td>
</tr>
<tr>
<td>8000000</td>
<td>104.408%</td>
<td>99.368%</td>
</tr>
<tr>
<td>1600000</td>
<td>100.879%</td>
<td>96.042%</td>
</tr>
<tr>
<td>3200000</td>
<td>98.682%</td>
<td>94.068%</td>
</tr>
<tr>
<td>6400000</td>
<td>95.924%</td>
<td>91.509%</td>
</tr>
<tr>
<td>10000000</td>
<td>94.280%</td>
<td>89.980%</td>
</tr>
</tbody>
</table>

Table 9.4: Gain on English Wikipedia with Optimal Parsing

9.5 Simulation of Optimal Parsing Using Arithmetic Coding

The aim of this chapter is to show the results obtained by a simulation in which it is used an arithmetic coding, and not Huffman as statistical technique. There is not a real implementation of this, although it would be very interesting. They are proposed also some results obtained by testing a new version of GZip, called Large GZip.
9.6 Simulation’s Idea and Results

The main idea of the simulation is based on the fact that all the edges of length pair to 1, are weighed in a different way. In fact, in all the previous tests, and in the working version of the program, those generic edges \((i, i + 1)\) are weighed with the number of bits that character \(i\) would require if it were encoded with Huffman. In this simulation, those edges are weighed in this way: generic edge \((i, i + 1)\) has a weight equal to the probability of \(i\) given \(i - 1\), that is, order one arithmetic coding. It is even possible to make other experimentations, with order 2, 3, 4 and 5.

As expected, results are very encouraging. Results obtained are shown in the tables in the following subsections.

9.6.1 Results on Calgary Corpus

In this section they are shown the results obtained by simulating the optimal parsing with LZ78 and arithmetic coding on files of Calgary Corpus. See table 9.5.

<table>
<thead>
<tr>
<th>file</th>
<th>size (bytes)</th>
<th>lz78 ratio</th>
<th>gzip -9 ratio</th>
<th>simulation ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>bib</td>
<td>111261</td>
<td>38.920%</td>
<td>31.375%</td>
<td>30.536%</td>
</tr>
<tr>
<td>book1</td>
<td>768771</td>
<td>39.524%</td>
<td>40.622%</td>
<td>32.680%</td>
</tr>
<tr>
<td>book2</td>
<td>610856</td>
<td>37.648%</td>
<td>33.750%</td>
<td>31.565%</td>
</tr>
<tr>
<td>geo</td>
<td>102400</td>
<td>66.453%</td>
<td>66.818%</td>
<td>44.835%</td>
</tr>
<tr>
<td>news</td>
<td>377109</td>
<td>44.017%</td>
<td>38.293%</td>
<td>36.156%</td>
</tr>
<tr>
<td>obj1</td>
<td>21504</td>
<td>59.510%</td>
<td>48.028%</td>
<td>36.566%</td>
</tr>
<tr>
<td>obj2</td>
<td>246814</td>
<td>46.597%</td>
<td>32.857%</td>
<td>36.039%</td>
</tr>
<tr>
<td>paper1</td>
<td>53161</td>
<td>45.232%</td>
<td>34.896%</td>
<td>34.330%</td>
</tr>
<tr>
<td>paper2</td>
<td>82199</td>
<td>42.250%</td>
<td>36.101%</td>
<td>32.837%</td>
</tr>
<tr>
<td>paper3</td>
<td>46526</td>
<td>46.037%</td>
<td>38.864%</td>
<td>35.086%</td>
</tr>
<tr>
<td>paper4</td>
<td>13286</td>
<td>49.782%</td>
<td>41.713%</td>
<td>35.094%</td>
</tr>
<tr>
<td>paper5</td>
<td>11954</td>
<td>51.849%</td>
<td>41.852%</td>
<td>35.501%</td>
</tr>
<tr>
<td>paper6</td>
<td>38105</td>
<td>46.640%</td>
<td>34.696%</td>
<td>34.531%</td>
</tr>
<tr>
<td>pic</td>
<td>513216</td>
<td>11.200%</td>
<td>10.208%</td>
<td>7.924%</td>
</tr>
<tr>
<td>progc</td>
<td>39611</td>
<td>45.977%</td>
<td>33.498%</td>
<td>34.288%</td>
</tr>
<tr>
<td>progl</td>
<td>71646</td>
<td>35.949%</td>
<td>22.572%</td>
<td>27.986%</td>
</tr>
<tr>
<td>progp</td>
<td>49379</td>
<td>37.202%</td>
<td>22.670%</td>
<td>27.281%</td>
</tr>
<tr>
<td>trans</td>
<td>93695</td>
<td>37.519%</td>
<td>20.140%</td>
<td>29.509%</td>
</tr>
</tbody>
</table>

Table 9.5: Simulation on Calgary Corpus

Results obtained are really eloquent. In fact, using LZ78 with arithmetic coding grants a great gain respect to LZ78 with Huffman. Anyway, with arithmetic coding, decompression time will certainly significantly grow.
9.6.2 Results on Canterbury Corpus

In this section they are shown the results obtained by simulating the optimal parsing with LZ78 and arithmetic coding on files of Canterbury Corpus. See table 9.6.

<table>
<thead>
<tr>
<th>file</th>
<th>size (bytes)</th>
<th>lz78 ratio</th>
<th>gzip -9 ratio</th>
<th>simulation ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>alice29.txt</td>
<td>152089</td>
<td>39.880%</td>
<td>35.634%</td>
<td>31.385%</td>
</tr>
<tr>
<td>asyoulik.txt</td>
<td>125179</td>
<td>42.636%</td>
<td>39.011%</td>
<td>33.060%</td>
</tr>
<tr>
<td>cp.html</td>
<td>24603</td>
<td>43.141%</td>
<td>32.459%</td>
<td>32.371%</td>
</tr>
<tr>
<td>fields.c</td>
<td>11150</td>
<td>43.193%</td>
<td>28.170%</td>
<td>28.329%</td>
</tr>
<tr>
<td>grammar.lsp</td>
<td>3721</td>
<td>46.385%</td>
<td>33.620%</td>
<td>28.670%</td>
</tr>
<tr>
<td>kennedy.xls</td>
<td>1029744</td>
<td>25.276%</td>
<td>20.368%</td>
<td>17.048%</td>
</tr>
<tr>
<td>lcet10.txt</td>
<td>426754</td>
<td>36.888%</td>
<td>33.845%</td>
<td>30.506%</td>
</tr>
<tr>
<td>plrabn12.txt</td>
<td>481861</td>
<td>39.751%</td>
<td>40.319%</td>
<td>32.313%</td>
</tr>
<tr>
<td>ptt5</td>
<td>513216</td>
<td>11.200%</td>
<td>10.208%</td>
<td>7.924%</td>
</tr>
<tr>
<td>sum</td>
<td>38240</td>
<td>48.394%</td>
<td>33.413%</td>
<td>31.945%</td>
</tr>
<tr>
<td>xargs.1</td>
<td>4227</td>
<td>53.158%</td>
<td>41.661%</td>
<td>33.741%</td>
</tr>
</tbody>
</table>

Table 9.6: Simulation on Canterbury Corpus

Results obtained are really eloquent. In fact, using LZ78 and arithmetic coding grants a great gain respect to LZ78 with Huffman. Anyway, with arithmetic coding decompression time will certainly significantly grow.
9.6.3 English Wikipedia

In this section they are shown the results obtained by simulating the optimal parsing with LZ78 and arithmetic coding on some large files. See table 9.7.

<table>
<thead>
<tr>
<th>size (bytes)</th>
<th>lz78 bytes</th>
<th>gzip -9 bytes</th>
<th>simulation bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000000</td>
<td>395963</td>
<td>355365</td>
<td>321056</td>
</tr>
<tr>
<td>2000000</td>
<td>775662</td>
<td>775662</td>
<td>636346</td>
</tr>
<tr>
<td>4000000</td>
<td>1515520</td>
<td>1457605</td>
<td>1260793</td>
</tr>
<tr>
<td>8000000</td>
<td>2928828</td>
<td>2947445</td>
<td>2461178</td>
</tr>
<tr>
<td>16000000</td>
<td>5664281</td>
<td>5897725</td>
<td>4746174</td>
</tr>
<tr>
<td>32000000</td>
<td>11018001</td>
<td>11712750</td>
<td>9247198</td>
</tr>
<tr>
<td>64000000</td>
<td>21375386</td>
<td>23358735</td>
<td>18211196</td>
</tr>
<tr>
<td>100000000</td>
<td>32793312</td>
<td>36445258</td>
<td>27865166</td>
</tr>
</tbody>
</table>

Table 9.7: Simulation on $n$ bytes of English Wikipedia

Results obtained are really eloquent. In fact, using LZ78 and arithmetic coding grant a great gain respect to LZ78 with Huffman. Anyway, with arithmetic coding decompression time will certainly significantly grow.
9.7 Comparison Between Most Famous Compressors

Calgary Corpus

In Table 9.8 the results following the compression of the files of Calgary corpus are reported. These values are obtained launching, in order, the following processes from a Unix Shell:

- 7za a file.7z5 file.
- bzip2 -9 file.
- advzip -4 -a file.def4 file.
- gzip -9 file.
- rar a -m5 file.rar5 file.

It is important to underline that the programs are exploited at their best capability. You can notice that best performance are almost always achieved by *rar*, but this happens because it is an arithmetic compressor, that needs a lot of time both for compression and decompression phase. This its characteristic does not make *rar* very useful if you have to compress great archives or you need a very low decompression time.

In fact, another interesting test that could be performed, with the aim of observing in particular the decompressin time of the programs selected. Even if this test was not made, we can surely affirm that best decompression time would widely be reached by GZip.

According to us, it is not so important to perform test for observing time and memory requirements, because, respectively, compression is generally made only once and not often as for decompression, and available RAM for a normal PC is really enough for all the five programs selected.

Canterbury Corpus

In Table 9.9 the results following the compression of the files of Canterbury corpus are reported. Comments are the same of those made for Calgary corpus.
<table>
<thead>
<tr>
<th>file</th>
<th>size</th>
<th>7-zip</th>
<th>bzip2</th>
<th>deflate</th>
<th>gzip</th>
<th>rar</th>
</tr>
</thead>
<tbody>
<tr>
<td>bib</td>
<td>111261</td>
<td>27.618%</td>
<td>24.687%</td>
<td>30.563%</td>
<td>31.368%</td>
<td>21.718%</td>
</tr>
<tr>
<td>book1</td>
<td>768771</td>
<td>33.970%</td>
<td>30.258%</td>
<td>39.106%</td>
<td>40.621%</td>
<td>27.412%</td>
</tr>
<tr>
<td>book2</td>
<td>610856</td>
<td>27.816%</td>
<td>25.774%</td>
<td>32.369%</td>
<td>33.749%</td>
<td>22.988%</td>
</tr>
<tr>
<td>geo</td>
<td>102400</td>
<td>51.922%</td>
<td>55.587%</td>
<td>64.371%</td>
<td>66.811%</td>
<td>61.264%</td>
</tr>
<tr>
<td>news</td>
<td>377109</td>
<td>31.677%</td>
<td>31.450%</td>
<td>37.195%</td>
<td>38.291%</td>
<td>27.572%</td>
</tr>
<tr>
<td>obj1</td>
<td>21504</td>
<td>44.117%</td>
<td>50.163%</td>
<td>48.038%</td>
<td>47.991%</td>
<td>45.657%</td>
</tr>
<tr>
<td>obj2</td>
<td>246814</td>
<td>25.203%</td>
<td>30.971%</td>
<td>31.843%</td>
<td>32.853%</td>
<td>28.899%</td>
</tr>
<tr>
<td>paper1</td>
<td>53161</td>
<td>32.607%</td>
<td>31.147%</td>
<td>33.811%</td>
<td>34.881%</td>
<td>27.586%</td>
</tr>
<tr>
<td>paper2</td>
<td>82199</td>
<td>33.219%</td>
<td>30.464%</td>
<td>34.588%</td>
<td>36.092%</td>
<td>27.360%</td>
</tr>
<tr>
<td>paper3</td>
<td>46526</td>
<td>36.794%</td>
<td>34.039%</td>
<td>37.532%</td>
<td>38.847%</td>
<td>30.834%</td>
</tr>
<tr>
<td>paper4</td>
<td>13286</td>
<td>41.066%</td>
<td>39.049%</td>
<td>41.066%</td>
<td>41.653%</td>
<td>35.150%</td>
</tr>
<tr>
<td>paper5</td>
<td>11954</td>
<td>41.359%</td>
<td>40.463%</td>
<td>41.584%</td>
<td>41.785%</td>
<td>36.155%</td>
</tr>
<tr>
<td>paper6</td>
<td>38105</td>
<td>32.975%</td>
<td>32.258%</td>
<td>34.064%</td>
<td>34.675%</td>
<td>28.521%</td>
</tr>
<tr>
<td>pic</td>
<td>513216</td>
<td>8.552%</td>
<td>9.696%</td>
<td>9.600%</td>
<td>10.206%</td>
<td>9.135%</td>
</tr>
<tr>
<td>progc</td>
<td>39611</td>
<td>31.847%</td>
<td>31.668%</td>
<td>32.890%</td>
<td>33.478%</td>
<td>27.904%</td>
</tr>
<tr>
<td>progl</td>
<td>71646</td>
<td>21.052%</td>
<td>21.744%</td>
<td>21.772%</td>
<td>22.561%</td>
<td>18.438%</td>
</tr>
<tr>
<td>progp</td>
<td>49379</td>
<td>21.132%</td>
<td>21.689%</td>
<td>22.007%</td>
<td>22.653%</td>
<td>18.889%</td>
</tr>
<tr>
<td>trans</td>
<td>93995</td>
<td>18.051%</td>
<td>19.103%</td>
<td>19.616%</td>
<td>20.131%</td>
<td>15.794%</td>
</tr>
</tbody>
</table>

Table 9.8: Comparison on Calgary Corpus

<table>
<thead>
<tr>
<th>file</th>
<th>size</th>
<th>7z</th>
<th>bzip2</th>
<th>deflate</th>
<th>gzip</th>
<th>rar</th>
</tr>
</thead>
<tbody>
<tr>
<td>alice29.txt</td>
<td>152089</td>
<td>31.936%</td>
<td>28.406%</td>
<td>34.143%</td>
<td>35.631%</td>
<td>25.572%</td>
</tr>
<tr>
<td>asyoulik.txt</td>
<td>125179</td>
<td>35.644%</td>
<td>31.610%</td>
<td>37.389%</td>
<td>39.007%</td>
<td>29.007%</td>
</tr>
<tr>
<td>cp.html</td>
<td>24603</td>
<td>31.500%</td>
<td>30.988%</td>
<td>31.862%</td>
<td>32.439%</td>
<td>27.029%</td>
</tr>
<tr>
<td>fields.c</td>
<td>11150</td>
<td>27.650%</td>
<td>27.256%</td>
<td>28.143%</td>
<td>28.126%</td>
<td>23.883%</td>
</tr>
<tr>
<td>grammar.lsp</td>
<td>3721</td>
<td>36.496%</td>
<td>34.480%</td>
<td>35.017%</td>
<td>33.486%</td>
<td>30.153%</td>
</tr>
<tr>
<td>kennedy.xls</td>
<td>1029744</td>
<td>4.909%</td>
<td>12.652%</td>
<td>17.181%</td>
<td>20.367%</td>
<td>3.827%</td>
</tr>
<tr>
<td>lcet10.txt</td>
<td>426754</td>
<td>28.043%</td>
<td>25.238%</td>
<td>32.388%</td>
<td>33.844%</td>
<td>22.564%</td>
</tr>
<tr>
<td>plrabn12.txt</td>
<td>481861</td>
<td>34.341%</td>
<td>30.211%</td>
<td>38.419%</td>
<td>40.318%</td>
<td>27.628%</td>
</tr>
<tr>
<td>ppt5</td>
<td>513216</td>
<td>8.552%</td>
<td>9.696%</td>
<td>9.601%</td>
<td>10.207%</td>
<td>9.135%</td>
</tr>
<tr>
<td>sum</td>
<td>38240</td>
<td>25.008%</td>
<td>33.758%</td>
<td>32.168%</td>
<td>33.400%</td>
<td>30.324%</td>
</tr>
<tr>
<td>xargs.1</td>
<td>4227</td>
<td>44.216%</td>
<td>41.684%</td>
<td>42.654%</td>
<td>41.542%</td>
<td>37.071%</td>
</tr>
</tbody>
</table>

Table 9.9: Comparison on Canterbury Corpus
Chapter 10

Conclusions

In this thesis was presented a graph-based parsing algorithm that allows to obtain an optimal parsing of an arbitrary text in linear time, under some reasonable hypotheses. Then, we showed how to build such graph in linear time and possibly online using LZ77-based algorithms.

From a theoretical point of view, pure LZ77 (that is, the version in which there is a greedy parsing) converges to entropy slower than LZ78 in its standard version. This means that, from a theoretical point of view, LZ78 should obtain better compression ratio respect to LZ77 on every file.

LZ77 uses a dictionary that is greater than LZ78’s one. In fact, LZ77 uses all the possible substrings of the portion of input text parsed previously, while LZ78’s dictionary has got only some of the substring used by LZ77. So, if both LZ77 and LZ78 schemes can take the advantage of optimal parsing, the LZ77 scheme has a greater gain with respect to greedy parsing. Indeed, even if the LZ78 scheme improves its performance using an optimal parsing, this improvement is not even comparable with the one obtained by the algorithm in the LZ77 scheme that uses an optimal parsing. When it is added the possibility to make an optimal parsing using also a statistical technique (mixing, for example, dictionary and Huffman) in a dictionary-symbolwise scheme, the scheme that takes the main advantages of this flexibility is LZ78. LZ77 scheme, that had already a great advantage from optimal parsing without symbolwise, has not a great gain when adding a symbolwise technique in optimal parsing but still has a better compression ratio with respect to the LZ78 one. So, we can affirm that, thanks to the symbolwise technique, the “distance” between LZ77 and LZ78 decreases. Anyway, given the results observed, LZ77 with optimal parsing (dictionary-symbolwise) continues to have a certain advantage on LZ78 with optimal
parsing (dictionary-symbolwise). It has to be noticed that, in this tests, the statistical

\textit{\text{technique used is Huffman. With a more powerful symbolwise, in general, overall
compression ratio decreases significantly and the two schemes could get even closer
than now.}}

A further improvement may come from making cocktails of different dictionary-
symbolwise algorithms. For instance, in real texts, some symbols can be totally pre-
dictable from the previous one. A symbolwise algorithm could potentially erase these
symbols, whereas a dictionary-symbolwise algorithm has always to pay the flag infor-
mation. Therefore, if before applying a dictionary-symbolwise algorithm one uses a
compressor such as the \textit{antidictionary} described in \cite{14, 12} or some variation of PPM*,
which erases predictable symbols, there are chances to further improve the compression
ratio. But even using an Huffman coding we made some experiments that by increas-
ing the cictionary to one or two megabyte, the compression ratio seems to overpass
the Bzip2 compression ratio while keeping an extremely fast decompression speed.

We also made a simulation on Calgary, Canterbury and Large corpus for observing
the behavior of our dictionary symbolwise technique with arithmetic coding as sta-
tistical technique. Results were very encouraging, and let us say that it reasonable,
in future, to implement a public program (open source or commercial) that mixes a
dictionary technique with an arithmetic one.

It is useful to underline that the whole theory and experimentation here conduced
are strongly motivated by widely diffusion of fast compressors. Gzip and Zip are Inter-
net preferred packers, used invisibly on many communication sockets. Cabarc is used
in installations package by Microsoft programs and it has the fastest decompression,
with just 15ns for byte on a Pentium 4 cpu, compared to the 3627ns of durilca4linux,
the compression up to date winner. We are try to enhance compression of gzip while
keeping compression speed comparable to cabarc one.
Appendix A

Summary of work in Italian

Sintesi del lavoro in lingua Italiana

Al giorno d’oggi le tecniche di compressione sono utilizzate praticamente ovunque. Si usano nei sistemi di immagazzinamento dati per risparmiare spazio e nei canali di trasmissione per risparmiare banda. Meno diretto è il loro utilizzo nei sistemi crittografici al fine di irrobustirli, sfruttando la proprietà dei messaggi compressi di avere meno ridondanza rispetto alla loro versione originale, cosa che li rende molto simili ad una stringa casuale, nocciolo della sicurezza nella crittografia. Si utilizzano tecniche di compressione anche in innovativi processi di clustering [7].

Ma qual è il miglior compressore? Questa, che è la prima domanda delle Frequently Asked Question sulla compressione dati, non ha una risposta unica. Ovviamente dipende da ciò che ci proponiamo di ottenere. In alcune applicazioni l’obiettivo sarà quello di ottenere il più alto tasso di compressione possibile, trascurando il tempo di compressione o la velocità di decompressione. In altre si preferirà la velocità di decompressione a discapito della compressione stessa o si cerchera un compromesso tra risorse e risultato.

I principali parametri di valutazione per un compressore sono:

- Tasso di compressione.
- Velocità di compressione.
- Velocità di decompressione.
- Risorse utilizzate.
Dove il tasso di compressione, dato dal rapporto tra la dimensione dei dati non compressi e la dimensione dei dati compressi, e le velocità, che si misurano in megabyte al secondo, sono i principali indici di prestazione. Le risorse utilizzate si riferiscono alla potenza di calcolo e alla quantità di memoria massima utilizzate durante i processi di codifica e di decodifica che determinano le specifiche di eventuali dispositivi hardware integrati capaci di effettuare tali operazioni o semplicemente i requisiti di un PC.

Per esempio per il trasferimento dati in Internet si preferisce utilizzare metodi di compressione che garantiscono un’elevata velocità di decompressione, preferibilmente in tempo reale, trascurando il tempo e le risorse utilizzate nel processo di compressione, dal momento che in questo scenario i dati compressi una volta vengono poi trasferiti e decompressi molte volte.

Tra i compressori che offrono una buona velocità di decompressione, la classe principale è quella dei compressori basati sui dizionari, primi tra i quali LZ77 [52] e LZ78 [53]. In questo tipo di compressori la decompressione consiste essenzialmente nel riproporre parte del file appena decompresso, operazione molto semplice e veloce da effettuare nel moderni calcolatori. Se l’obiettivo è quello di raggiungere la compressione massima, la classe dei compressori che attualmente svolge meglio questo compito è quella dei compressori statistici, chiamati anche symbolwise, per esempio Huffmann [23] o PPM [8]. In essi la compressione sfrutta le diverse probabilità relative alla presenza di un simbolo piuttosto che un altro dopo un certo contesto, probabilità calcolate per via statistica. In questa classe di compressori ottiene una migliore compressione al costo di un maggior tempo di decompressione e di maggior risorse impiegate, dal momento che gli stessi calcoli statistici che fa il compressore devono essere fatti anche dal decompressore. Questo panorama è assolutamente coerente con quanto dichiara Bell in [4], dicendo che la ricerca mirata ad incrementare il livello di compressione deve concentrarsi sui metodi symbolwise, mentre i metodi basati sui dizionari devono essere scelti per la velocità di decompressione.

Come preveduto da [4], a quindici anni di distanza, secondo le prove effettuate da Mahoney e pubblicate nella sua pagina web [33] dedicata all’analisi delle prestazioni dei compressori su grandi testi (”large text compression benchmark”), la migliore compressione su ewik8 and ewik9, rispettivamente i primi 100 Megabyte ed il primo Gigabyte della versione in lingua inglese di wikipedia, enciclopedia libera su Internet [51], è ottenuta da durilca_4_linux che utilizza un metodo a dizionario, mentre il minor tempo di decompressione è di cabarc, il programma di compressione di Microsoft basato sul metodo a dizionario LZ77 [52].
In questa tesi riprendo e continuo il lavoro fatto da me e altri in [38] ottenendo fondamentalmente quattro risultati:

- 1) Definizione di compressore dictionary-symbolwise.
- 2) Definizione di ottimalità basata sul parsing.
- 3) Metodo per trovare un parsing ottimo in modo lineare.
- 4) Presentazione del compressore Large GZip.

Per quanto riguarda i compressori dictionary-symbolwise si è cominciato a sviluppare una teoria per questo tipo di compressori molto diffusi, a partire dalle definizioni di algoritmo dictionary-symbolwise e di schema dictionary-symbolwise. Molti risultati estendono dei risultati classici della teoria dei compressori basati sui dizionari e ne offrono una generalizzazione a questa famiglia più ampia. Nei compressori dictionary-symbolwise sono presenti sia tecniche di compressione tramite dizionari che tecniche statistiche, miscelate in una sorta di “cocktail”, ma non è presente nella letteratura scientifica una trattazione soddisfacente di questo tipo di compressori che vengono generalmente catalogati come context mixing, abbreviato CM.

Sinteticamente, un algoritmo dictionary-symbolwise è specificato da: (1) la descrizione del dizionario, (2) la codifica dei puntatori al dizionario, (3) il metodo di compressione symbolwise, (4) la codifica dell’indicazione dictionary o symbolwise, chiamata flag information, e (5) il metodo usato per il parsing. Noi chiamiamo schema dictionary-symbolwise una classe di algoritmi dictionary-symbolwise che abbiano in comune i primi quattro punti della definizione, cioè differiscano solo per il metodo usato per il parsing.

A questo punto è possibile individuare tre classi distinte di schemi dictionary-symbolwise: La prima classe include tutti gli schemi contenenti gli algoritmi dove per qualsiasi testo $T$ il dizionario, la codifica dei simboli, la codifica dei puntatori al dizionario e la codifica della flag information sono indipendenti dal parsing. La seconda classe include tutti gli schemi non appartenenti alla prima classe nei quali il dizionario o le codifiche per ogni posizione del testo $i$ dipendono dal parsing della porzione di testo precedente che è già stata processata.

La terza classe include gli schemi rimanenti.

Possiamo dire che il primo compressore dictionary-symbolwise fu LZSS, presentato da Bell con un articolo nel 1986 ([3]). In quest’ultimo Bell raccoglie il suggerimento di Storer e Szymanski del 1982 (vedi [46]), di rilassare il vincolo dell’algoritmo di
Lempel-Ziv per cui si ha un’alternanza stretta tra un puntatore al dizionario ed un carattere letterale, proponendo di usare l’uno piuttosto che l’altro liberamente, senza uno schema predefinito, in modo tale da usare i caratteri letterali solo quando un puntatore al dizionario prende più spazio degli stessi caratteri che esso rappresenta.

Ma perché utilizzare un compressore dictionary-symbolwise?

Dal punto di vista pratico, perché è un buon compromesso tra compressione e velocità, dato che accoppiando un symbolwise ad un dizionario aggiunge, in un certo senso, un grado di libertà in più al parsing di quest’ultimo, migliorandone la compressione senza comprometterne la velocità.

Dal punto di vista teorico Ferragina et al. (cf. \[17\]) hanno provato che la compressione di LZ77, puro dizionario, data dal parsing greedy, può essere lontana dalla bit ottimalità di un fattore moltiplicativo Ω(\(\frac{\log(n)}{\log \log(n)}\)) che è asintoticamente illimitato. In questa tesi in 6.4 viene mostrato un risultato simile tra la bit ottimalità dei metodi a puro dizionario verso i symbolwise. Da questo segue che i dictionary-symbolwise, che possono comportarsi come i puri symbolwise, anche se solo in una eccezione estrema, possono comprimere meglio dei compressori a puro dizionario.

Negli algoritmi che utilizzano i dizionari la compressione viene realizzata sostituendo frammenti del testo con puntatori, cioè riferimenti, al dizionario, dove la rappresentazione del puntatore occupa meno spazio che il frammento stesso. In questo metodo un passo fondamentale detto parsing è quello di decomporre i dati o il testo da comprimere in frammenti da collezionare nel dizionario. Trovare il parsing ottimo, cioè quello che massimizza la compressione, è uno dei punti cruciali di questi compressori.

Esiste già un’abbondante letteratura sul problema del parsing ottimo (optimal parsing).

Un approccio classico è quello di rappresentare il dizionario come un grafo e ridurre, sotto l’ipotesi di chiusura per prefisso e di costo degli archi logaritmico sulla loro taglia, il problema del parsing ottimo a trovare un cammino minimo sul grafo(see [44]). Questo vale per estensione anche per i compressori dictionary-symbolwise, come mostrato in cf. Theorem 1, dopo aver ridefinito il problema in termini di ottimalità locale e globale per classi di algoritmi.

Fissata una funzione costo \(C\), un algoritmo dictionary-symbolwise è ottimo all’interno di una classe di algoritmi, se il costo della codifica è minimo all’interno della classe di algoritmi.

Un parsing si dice ottimo rispetto ad un testo \(T\) e ad uno schema di algoritmi se l’unico algoritmo dello schema che utilizza quel parsing risulta essere l’ottimo all’interno
dello schema.

Se un algoritmo appartiene ad uno schema nella prima classe il grafo $G_T$ indotto dal dizionario può essere costruito prima che l’algoritmo scelga un parsing per il testo, quindi il grafo $G_T$ è lo stesso per tutti gli algoritmi all’interno dello stesso schema. In questo caso diciamo che il grafo $G_T$ è associato allo schema.

Il grafo $G_T$ può essere costruito on-line per gli algoritmi degli schemi nella seconda classe. Significa che se l’algoritmo ha processato il testo fino alla posizione $i$ allora tutti gli archi uscenti dal nodo $i$ nel grafo sono ben definiti ed anche il loro costo è ben definito. Per gli algoritmi appartenenti alla terza classe il grafo $G_T$ non è necessariamente ben definito.

**Theorem 5** Il parsing indotto dal cammino minimo su un grafo associato ad uno schema è un parsing ottimo per quello schema.

L’approccio al problema del parsing ottimo tramite cammino minimo è sconsigliato in [44] perché richiede troppo tempo, cioè tempo quadratico nel caso pessimo e $O(n\log(n)/H)$ in media per sorgenti senza memoria nel caso di LZ77, che è accettabile in pratica.

In questa tesi abbiamo applicato questa teoria a Large GZip, un compressore dictionary-symbolwise evoluzione di gzip o zip, che implementa un metodo di parsing ottimo, con risultati migliorativi fino al 10% su file grandi rispetto a gzip.

<table>
<thead>
<tr>
<th>file</th>
<th>size (bytes)</th>
<th>lgzip ratio</th>
<th>gzip -9 ratio</th>
<th>lgzip/gzip</th>
</tr>
</thead>
<tbody>
<tr>
<td>enwik0</td>
<td>1</td>
<td>3100%</td>
<td>2800%</td>
<td>110.71%</td>
</tr>
<tr>
<td>enwik1</td>
<td>10</td>
<td>400%</td>
<td>370%</td>
<td>108.11%</td>
</tr>
<tr>
<td>enwik2</td>
<td>100</td>
<td>108%</td>
<td>101%</td>
<td>106.93%</td>
</tr>
<tr>
<td>enwik3</td>
<td>$1K$</td>
<td>34.2%</td>
<td>34.3%</td>
<td>99.71%</td>
</tr>
<tr>
<td>enwik4</td>
<td>$10K$</td>
<td>36.77%</td>
<td>37.22%</td>
<td>98.79%</td>
</tr>
<tr>
<td>enwik5</td>
<td>$100K$</td>
<td>33.81%</td>
<td>36.07%</td>
<td>93.73%</td>
</tr>
<tr>
<td>enwik6</td>
<td>$1M$</td>
<td>30.24%</td>
<td>35.54%</td>
<td>85.09%</td>
</tr>
<tr>
<td>enwik7</td>
<td>$10M$</td>
<td>28.29%</td>
<td>36.85%</td>
<td>76.77%</td>
</tr>
<tr>
<td>enwik8</td>
<td>$100M0$</td>
<td>26.59%</td>
<td>36.45%</td>
<td>72.95%</td>
</tr>
<tr>
<td>enwik9</td>
<td>$1G$</td>
<td>23.23%</td>
<td>32.26%</td>
<td>72.01%</td>
</tr>
</tbody>
</table>

Table A.1: Risultati di lgzip su enwik file
Bibliography


[29] Katz, P.


[40] Pkware home page.


