UNIVERSITÀ DEGLI STUDI DI PADOVA
DIPARTIMENTO DI INGEGNERIA ELETTRONICA
E INFORMATICA

DOTTORATO DI RICERCA IN INGEGNERIA INFORMATICA
ED ELETTRONICA INDUSTRIALI
X ciclo

Off-Line Data Compression By Textual Substitution

Stefano Lonardi

Tutor
Ch.mo Prof. Alberto Apostolico

Coordinatore
Ch.mo Prof. Luigi Malesani

31 Dicembre 1998

Sommario

Un aspetto centrale nella gestione digitale dell'informazione è lo sviluppo di metodi per la rappresentazione efficiente dell'informazione stessa. La compressione dati si interessa di algoritmi, protocolli e codici capaci di produrre rappresentazioni "concise" dell'informazione.

In questa Tesi ci concentriamo sulla compressione dati senza perdita di informazione (*lossless*) ed in particolare su alcuni metodi di sostituzione testuale (*textual substitution*) che traggono origine da idee di Ziv e Lempel [178, 179, 180] in seguito generalizzate da Storer e Szymanski [149, 150, 152]. Tali metodi sono oggi alla base di numerosi programmi di compressione, quali ad esempio COMPRESS, GZIP, ZIP, PKZIP, STUFFIT, STACKER, ed altri, comunemente utilizzati per ridurre le dimensioni di archivi in memoria secondaria e/o per diminuire i tempi di trasmissione sulle reti di comunicazione digitale. Sono anche realizzati in hardware nei protocolli di trasmissione di FAX, MODEM, e *router* di rete, e sono usati in crittografia per ridurre le possibilità di criptoanalisi.

Questa Tesi studia un approccio alla compressione dati chiamato OFF-LINE costruito intorno al seguente paradigma. L'analisi fuori linea (cioé dopo che i dati sono stati caricati in memoria centrale) identifica nel testo da comprimere una sottostringa particolarmente ridondante e ne sostituisce tutte le possibili occorrenze non sovrapposte con altrettanti puntatori ad una copia comune. La sottostringa viene scelta in modo da ottenere a quello stadio la massima compressione possibile. Il processo viene ripetuto sul testo compresso finché non è più possibile trovare una parola in grado di produrre ulteriore compressione.

A dispetto della apparente semplicità dello schema, la sua realizzazione efficiente presenta interessanti problemi algoritmici, scelte progettuali riguardanti strutture dati avanzate, e lo sviluppo di euristiche atte a ridurre i tempi di calcolo. Il lavoro descritto in questa tesi riguarda specificatamente questi aspetti e tralascia deliberatamente gli aspetti algoritmici e combinatoriali che sottendono questi sviluppi, aspetti che formano oggetto e di cui si da resoconto nelle pubblicazioni allegate in Appendice.

Nella maggior parte dei casi, le realizzazioni risultanti dal paradigma OFF-LINE attingono a compressioni superiori a quelle raggiunte dai più noti programmi di compressione di tipo sostituzionale. Su dati non strutturati, quali ad esempio le sequenze genetiche, la prestazione raggiunta è migliore perfino di quella di BZIP, l'attuale "stato dell'arte" basato sull'algoritmo di *context-sorting* di Burrows e Wheeler [30].

Dipartimento di Ingegneria Elettronica ed Informatica
Via Gradenigo 6/a – 35131 Padova – Italy
E-MAIL: `stelo@dei.unipd.it`

Abstract

One crucial concern of digital information processing consists precisely of developing increasingly efficient methods for the representation of information itself. Data compression studies algorithms, protocols, schemes, and codes capable to produce "concise" representations of information.

In this Thesis, we focus the attention on *lossless* data compression and in particular on some *textual substitution* methods that were pioneered by Lempel and Ziv [178, 179, 180] and generalized later by Storer and Szymanski [149, 150, 152]. These methods are implemented today in many compression programs such as COMPRESS, GZIP, ZIP, PKZIP, STUFFIT, STACKER, etc., commonly used to reduce the size of archives on secondary memory and/or to speed up transmissions on computer networks. They are also realized in hardware in many communication protocols of FAX machines, MODEMS, and network routers, and also in cryptography to make the encryption stronger.

This Thesis studies an approach to data compression called OFF-LINE based on the following paradigm. An analysis conducted *off-line* (i.e., assuming that the entire text-string to be compressed is already available) identifies in the text a substring which is highly redundant. Next, the occurrences of this substring in a maximal set of non-overlapping occurrences are substituted with as many pointers to a suitable reference copy. The substring itself is chosen so as to maximize the expected contraction it would induce on the text once substituted as above. The process is then iterated until no word capable of producing further compression can be found.

Despite of the apparent simplicity of the scheme, its efficient realization raises challenging and interesting algorithmic problems. Furthermore, advanced data structures and heuristics have to be designed and implemented in order to reduce the computation time. The work described in this Thesis is concerned specifically with these aspects and disregards the algorithmic and combinatorial issues that subtend these developments, issues that form the object of the already published records reported as here in the References and as an Appendix.

In most of the cases, the implementations of the OFF-LINE paradigm attain a compression efficiency higher than other textual substitution programs. When input files are unstructured, notably, with genetic sequences, the performance is even higher that BZIP, the current "state of the art" based on the *context-sorting* algorithm by Burrows and Wheeler [30].

Acknowledgments

First and foremost, I would like to thank Alberto Apostolico for his advice, interest, and encouragement during the collaboration that led to this Thesis. He first sparked my interest in the theory of algorithms on strings, computational biology and lossless data compression.

At Purdue University, thanks go to Vernon Rego for granting use of the PC-cluster on which many experiments were performed, to Mike Atallah for an enlightening course on algorithms, and to Valerio Pascucci for introducing me to STL.

At University of Padova, I am indebted to Giorgio Satta for many stimulating discussions on suffix trees and algorithms on strings.

Special thanks goes to James Storer who contributed some interesting ideas; to Andreas Dress, Stefan Kurtz, Gene Myers, and Bernhard Balkenhol for a fruitful period in Bielefeld; to Jean Loup Gailly, Mark Adler, and Mark Nelson for having produced and made public helpful code on data compression.

During the time spent doing my research I feel particularly lucky to have had the opportunity to meet and exchange ideas with Gianfranco Bilardi, Mary E. Bock, John J. Cruz, Concettina Guerra, Graziano Pesole, Andrea Pietracaprina, Geppino Pucci, Guglielmo Rabbiolo, James F. Reid, Paolo Sommaruga, and Giorgio Valle. All of them have helped me in various ways and contributed to this work.

My family deserves a warm and special thank for their loving support and encouragement during these years.

Last but not least, I want to thank Paola — for everything.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

I have made this letter longer than usual because I lack the time to make it shorter.
– *Blaise Pascal (1623–1662)*

Eventually, all information about physical objects including humans, buildings, processes, and organizations will be on-line. This trend is both desirable and inevitable. In the near future grand scientific challenges of molecular biology, physics, aerospace, etc. will involve very-large distributed databases of textual documents, whose effective storage and communication requires a major research and development effort. From Genome Studies to the World Wide Web, from Digital Libraries to Satellite Communications, the ability to handle efficiently information is crucial.

The goal of data compression is to find succinct representations of data. The reason is twofold. First, a shorter representation takes less time to be transmitted over a network. When sitting at the computer, waiting for a Web page to come in, or for a file to download, we naturally feel that anything longer than a few seconds is a long time to wait. Second, a compact file takes less space to be stored in primary or secondary memory. No matter how big a storage device one has, sooner or later it is going to overflow. Data compression delays this inevitability.

The importance of data compression in information technology cannot be over-emphasized. The design of algorithms and methods to build a compact representation of data has been crucial in the process of developing devices like FAX machines, digital television, Internet routers and fast modems [157].

An abstract model of a transmission system is shown in Figure 1.1 where compression is performed in the encoder, decompression in the decoder. The general assumption is that the source emits symbols based on some unknown probabilistic model. Data can be compressed whenever some patterns of data are more likely to occur than others. In most applications the channel is assumed to be noiseless, although some methods pose the problem of devising an encoding resilient to errors (see e.g., [154]).

In general, two broad classes of data compression schemes can be defined: (1) *lossless* (or *reversible*, *textual*) and (2) *lossy* (or *non-reversible*).

The lossless class contains all the methods where the data recovered by the decoder is identical to the data which was compressed by the encoder. By contrast, in a system of lossy compression, the decompressed data is just an approximation of the original. By allowing some errors in the recovered data much greater compression can be achieved. In particular, small differences between the original and the reconstructed data are not noticed when we compress images and sounds. We can say that lossy methods "exploit" the weaknesses of human senses and perception to achieve better compression. Some of the lossy methods in the for image/video compression are JPEG (see e.g., [164]), MPEG (see e.g., [66]), vector quantization (VQ) (see e.g., [69]), wavelet based (see e.g., [47]), and fractal based (see e.g., [104, 17]).

In the present Thesis we restrict our attention to lossless techniques. Unfortunately, any lossless compression method is inherently limited. It cannot compress *all* files of size n since some of these files are "random". For the sake of the contradiction, let us assume that an algorithm exists that can compress by at least one bit without loss all files of size n bits. Use this algorithm to process all the 2^n files of size n bits. All compressed files have at most $n-1$ bits, so there are at most $\sum_{i=0}^{i=n-1} 2^i = 2^n - 1$ different compressed files. So at least two different input files must compress to the same output file, then the method is not lossless.

An analogous counting argument can be used to show that there cannot exist a single dictionary construction scheme which is superior to other schemes for all inputs. If a compression algorithm performs well for one set of input strings, it is likely that it will not perform well for others. The advantage of one dictionary construction scheme over another can only apply with regard to restricted class of input texts.

In spite of the above fundamental limitations, many successful approaches have been devised and implemented in software products. A partial classification is shown in Figure 1.2, along with relevant references.

Although in this Thesis we concentrate on the class of *substitutional* (or *dictionary-based* or *macro schemes*) methods, we briefly describe *statistical* and *predictive* encoders.

Statistical encoders usually consists of a modeling stage followed by a coding stage. The model assigns probabilities to the source symbols, the coding stage

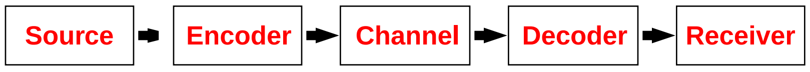


Figure 1.1: The general model of a transmission system: compression and decompression are respectively integrated in the encoder and the decoder

actually codes the symbols based on those probabilities. *Static* methods need two passes over the file, one to gather the statistics and the other to compress the file and therefore they are not *on-line*. *Dynamic* methods build and adapt the model as they process the file.

Predictive encoders are based on finite-context models and use the preceding few symbols to predict (or estimate the probability of) the next one. The main idea is that, since not all short substrings may be uniformly represented in the text, then the knowledge of a short substring constitutes a good basis for guessing the next character in the stream of symbols.

Statistical and predictive methods usually give better compression because they locally adapt better to the structure of the text. Textual substitution schemes are usually faster because they process a whole text fragment at a time.

A systematic classification of substitutional techniques has been developed by Storer and Szymanski in the late Seventies [149, 155, 68, 156, 151, 152]. They characterized several different *macro* schemes with regard to the structure of the pointers. Unfortunately, for most such schemes, the problem to find the shortest representation of a given text is $\mathcal{NP}$-complete.

Only a couple of notable macro schemes turn out to be tractable: Lempel and Ziv (LZ) introduced these schemes in the late Seventies, originally to characterize the "complexity", or information contents of a string [178, 179, 180]. Later, they were shown to have linear time implementations (see e.g., [131]).

Their moderate requirements in terms of time and memory gave them widespread implementations in wired hardware, like for modems [157], FAX machines and network routers, as well as in software, like Java applets, image formats and cryptography [133, 117].

In the next Chapters we will see that the nature of the pointers in Lempel-Ziv schemes is highly constrained to be unidirectional and non-recursive. In fact, the typical implementation of LZ-77 [179] processes data *on-line* as it is read, i.e., it parses the file sequentially *left to right* and looks through a fixed size window of past symbols to find the longest match of the substring starting at the current position. This substring is then substituted with a *pointer* to the previous occurrence, e.g., a pair of integers such as displacement and length, which yields a shorter representation.

There are many variations on the "theme by Lempel and Ziv" (more than twenty are described in the books [20, 140]), but probably the most important implementations are COMPRESS, which is found on all UN*X installations (based on a variant of LZ-78 [166, 180]) and the exemplary GZIP from GNU (based on LZ-77 [179], with addition of many tricks and improvements). The relative performance of such schemes depends on many factors, including the often subtle interplay between pointer size and dictionary parameters. The fine tuning of these parameters is extremely time consuming but always critical for the final performance.

Lempel-Ziv schemes are asymptotically optimal both in terms of compression achieved and algorithmic complexity. The uni-directional nature of pointers is crucial in determining their computational efficiency. Unfortunately, the only tradeoff that can be adjusted in these methods is the size of sliding window or the size of the dictionary.

Our proposed method may be regarded as introducing a tradeoff in Lempel-Ziv schemes, by relaxing the on-line constraint which is naturally suggested by the sequential nature of the data to be compressed. The approach, called OFF-LINE, can in principle handle the entire file or large portions of it, and exploits random access to issue pointers in either direction if this brings increase in compression.

A prominent property of this paradigm is the possibility for the user to adjust the tradeoff between compression and time/space complexity. We would expect that, within some reasonable bounds, the larger is the computational effort, the more compact the representation obtained at the outset. There are applications, e.g., storing data on CD-ROMS, in which one can afford a very efficient time-consuming compression if the decoding is kept fast.

Off-line heuristics introduce a time complexity overhead that can be heavy. In this Thesis we describe and explore some strategies devised to cope with the problem. Their possible implementation in parallel [153, 42, 70, 43, 147, 39], perhaps by dedicated architectures, may be expected to achieve the speed to process streams of large files in real-time.

The main contribution of this work is to have carried out a systematic analysis of a family of data compression methods previously largely unexplored. In terms of results we obtain uniform improvements over PACK (Huffman), COMPRESS (LZ-78). We outperform GZIP (LZ-77) in most of the cases. When compared with context-modeling encoders, like BZIP [30], we obtain better results only for highly random inputs (notably, genetic sequences).

The textual substitution scheme described here shows promise of interesting applications to the inference of hierarchical structures or grammars (see e.g., [63, 64, 125]) an aspect not studied in detail in this Thesis.

Related works

It was surprising to learn that the idea of iterating textual substitution based on the number of occurrences of each word in the text dates back to the Seventies. The following passages are from a landmark paper by Frank Rubin [132].

> ... A seemingly ideal way to develop the input groups[1] would be to choose the most frequent sequence in the input string, substitute a code for each occurrence of this group, choose the next most frequent sequence in the recorded input string, and so forth ... It is not a priori obvious that the most frequent groups are the most valuable, and it may be desirable to take the length of the group into account. That is a frequent long group would be considered more valuable than a frequent short group ... The

[1] i.e., to parse the input in phrases

second open question in the accretion technique is what measure of "best" input group should be used. The measure should reflect the amount of saving obtained by using that particular input group ...

Rubin attacks the problem of finding a compact representation of a given text by means of an iterative algorithm. In the first iteration he assigns a code to each symbol of the alphabet and he parses the text. In the following iterations he groups together parsed substrings to create a new entry in the dictionary only if it brings an increase of compression in the encoding. The evaluation of a function based on the length and the number of occurrences of the phrases indicates the "best" phrases to be merged. The process of grouping is iterated until the length of the representation of the source cannot be improved further.

The work by Storer and Szymanski [149, 155, 156, 151, 152] shows that the general problem of finding the representation of minimal size of a given text using general macro schemes, even assuming that the pointers are encoded with a fixed length code, is $\mathcal{NP}$-complete. Our approach can be thought as an approximation of the optimal parsing by steepest descent of some of those macro schemes. Chapter 2 describes Storer's macro schemes in detail and introduces some complexity issues connected with data compression via textual substitution.

As an improvement of LZ-77, Horspool discusses the effect of non-greedy parsing and shows that some gain can be obtained relaxing the usual longest match strategy [78]. This may suggest that off-line could be advantageous in terms of the compression achieved. A non-greedy strategy is used effectively in GZIP and is called *lazy evaluation* mechanism.

The paper by Apostolico and Preparata [13] suggests that the augmented suffix tree could be used in the implementation of a data compression scheme.

Some theoretical studies on the complexity of the off-line *optimum parsing* appeared recently [46, 44], but as far as we know no off-line technique has been implemented to date (see Section 2.8). These results suggest that the power of an off-line is encoder is needed to approximate the optimal parsing.

Finally, Nevill-Manning and Witten [125, 123, 126, 124] describe SEQUITUR an on-line algorithm capable to infer a hierarchical grammar from the text. However, the grammar they produce is much more constrained than ours. We report some comparisons in the final Chapter.

Organization of the Thesis

The structure of the Thesis is as follows. In Chapter 2 we introduce the problem and we review most of the known textual substitution methods, in particular Lempel-Ziv schemes. We give an overview of the theoretical studies and results on macro schemes to date.

Chapter 3 introduces the general idea behind the approach we propose. We cover three variations on the general scheme connected to Storer's macro scheme described in Chapter 2.

In Chapter 4 we delve into the details of the implementation. We describe data structures, implementation issues, heuristics, and performance of our encoders.

Some conclusive remarks and future directions are addressed in the final Chapter.

- Substitutional
  - Macro schemes or dictionary-based [149, 155, 150, 156, 68, 151, 152]
  - LZ-78 [180] and LZW (e.g., COMPRESS and GIF) [166, 115, 174, 35, 111, 109]
  - LZ-77 (e.g., GZIP) [179, 131, 60, 170, 78, 22, 98]
- Statistical
  - Huffman (e.g., PACK) [80]
  - Dynamic Huffman (e.g., COMPACT) [67, 89, 161]
  - Arithmetic coding [97, 96, 171, 127]
  - Markov model (DMC) [36]
- Context Modeling
  - Prediction by partial matching (e.g., PPM) [34, 33, 116]
  - Context tree weighting (CTW) [168, 169, 167, 159, 158, 1, 162, 2]
  - Burrows-Wheeler Transform (e.g., BZIP) [30, 55, 136, 135, 15, 99, 177, 175, 176, 110]
  - Symbol ranking [56, 57]
  - Hash table based [128]
  - Induction of hierarchical grammars (e.g., SEQUITUR) [125, 123, 126, 124, 16]
- Misc.
  - Run-Length (RLE) (see e.g., [20])
  - Differential or delta (DPCM) [81]
  - Move To Front (MTF) [134, 79, 50, 84, 21, 71]

Figure 1.2: A partial classification of lossless compression methods

Chapter

2

Macro schemes

The input to a lossless data compression algorithm is a text on a fixed alphabet Σ. It is denoted by x and called the *source*. The output of the algorithm is a word $\hat{x} \in \{0,1\}^*$, the *encoded* text. The efficiency of the compression is measured by the *compression ratio* $|\hat{x}| / |x|$.

The method is defined by means of a functions $f : \Sigma \to \{0,1\}^*$ and its inverse $f^{-1} : \{0,1\}^* \to \Sigma$ such that $\hat{x} = f(x)$ and $x = f^{-1}(\hat{x})$. The function f is called the *encoder*, f^{-1} is the *decoder*. Since no distortion is allowed and the original sequence must be fully recoverable from its compressed description we have $x = f^{-1}(f(x))$.

In general, a textual substitution scheme (or macro scheme) defines f as follows

- find a dictionary D (it could be part of x);
- parse the text x using the dictionary D;
- compute a code to encode the parsing.

In Section 2.2 we describe some instances on the above scheme. For most of them the problem of computing the optimal encoding of a given text is $\mathcal{NP}$-complete (see Section 2.3). In two very special cases, however, the problem is tractable. The schemes by Ziv and Lempel, covered in Section 2.4, 2.5, and 2.6, have linear time implementations. A polynomial algorithm that computes the optimal encoding when the dictionary is known in advance is described in Section 2.7. Analytic properties of on-line/off-line encoding and decoding are discussed in Section 2.8.

2.1 Notations

We shall treat the source data as a finite *string* over some fixed *alphabet* Σ of *symbols*. Given a string w in Σ^*, the number of consecutive symbols in w is the *length* of the string, $|w|$.

For convenience of notation we assume n to be the length of the source string x and m_w the length of a substring w. We set B equal to $\log_2 |\Sigma|$ that is the number of bits per symbol in the plain text. Unless otherwise stated, $\log k$ means $\log_2 k$.

We denote the i-th symbol of w by $w[i]$, $1 \leq i \leq |w|$. A *substring* (or a *word*) of w, denoted by $w[i,j]$, is composed by $w[i] \cdot w[i+1] \cdot \ldots \cdot w[j]$ where $\cdot$ is the concatenation operation and $1 \leq i \leq j \leq |w|$ (for all other choices of i and j, $w[i,j]$ is the empty string). Substrings of the form $w[1,j]$ for $1 \leq j \leq |w|$ are called the *prefixes* of w, while the substrings $w[j,|w|]$ are called the *suffixes* of w.

We define $\mathcal{X}$ to be the set of all the distinct substrings that we can choose from x, that is $\mathcal{X} = \{w |\ \exists\ i,j \in [1,n]$ such that $w = x[i,j]\}$. If $|x| = n$, then the cardinality of $\mathcal{X}$ is $O(n^2)$.

We say that a *pattern w occurs* in x if there exists a *position* $i \in [1, |x| - |w| + 1]$ such that $w = x[i, i + |w| - 1]$. Equivalently, w occurs in x if either (1) w is a prefix of a suffix of x or (2) w is a suffix of a prefix of x.

2.2 The class of macro schemes

In his 1979 Ph.D. Thesis [150], James Storer describes some complexity issues related to lossless data compression. In particular, he introduces a rather exhaustive classification of *macro schemes*. Macro schemes are systems which factor out duplicate occurrences of data, replacing the repeated elements with some sort of *pointer* identifying the data to be replaced at that point.

More precisely, Storer defines a pointer as a pair (d, l) where d is the position of the first character in the target and l is the length of the target. The *size* of the pointer is denoted by $|(d, l)|$.

A pointer is called *internal* if it refers to positions inside the original string. With internal macro schemes, a string is compressed by replacing duplicate occurrences of a substring with a pointer to another occurrence of the same substring. The final result is a single string of symbols interspersed with pointers.

A pointer is called *external* if it points to an external structure, called *dictionary*. With external macro schemes, a string is compressed by replacing duplicates with pointers to common words stored in the dictionary. In general, the dictionary is allowed to contain pointers to other entries in the dictionary. The result is a pair of strings, the dictionary and the string of symbols interspersed with pointers.

Other considerations on the nature of the pointers lead to several variations on the above basic scheme. A scheme is *recursive* if a string that is the target of a pointer is allowed to contain inside other pointers. While cycles cannot occur in compressed representation using compressed pointers, using original pointers cycles may be useful. If we allow recursion but not cycles, then the scheme has *topological recursion*.

An *original* pointer refers to a substring of the original source string, whereas a *compressed* pointer denotes a substring of the compressed representation itself. A *left (right)* pointer points only to substrings occurring earlier (later) in the text. If all the pointers point in the same direction then we say that the pointers are *unidirectional.* Finally, we say that two pointers *overlap* if their target substrings overlaps.

The above classification led Storer to define four basic macro schemes: (compressed) *external* pointer macro (EPM), *compressed* (internal) pointer macro (CPM), *original* (internal) pointer macro (OPM) and *original external* pointer macro (OEPM).

Example 1 *Let* $x = \textit{\textbf{abaababaabaababaababa\$}}$ *and assume for simplicity that the size of a pointer is one. D is the external dictionary. Then we can encode x using the CPM scheme, and non-recursive, unidirectional pointers as*

$$\hat{x} = \textit{\textbf{aba}}(1,3)(2,5)(4,8)(2,2)\textit{\textbf{\$}}$$

and we obtain a ratio of $|\hat{x}| \,/\, |x| = \frac{8}{22}$*. If we allow pointers to be bidirectional and overlapping we can express the string under the OPM scheme as*

$$\hat{x} = (9,6)(10,2)\textit{\textbf{abaaba}}(13,2)(12,5)\textit{\textbf{\$}}$$

and we obtain $\frac{11}{22}$*. Using EPM, and non-recursive, unidirectional pointers*

$$\hat{x} = (1,3)(1,3)(2,2)(1,3)(1,3)(2,2)(1,3)(2,2)\textit{\textbf{\$}} \quad D = \textit{\textbf{aba}}$$

we get $\frac{12}{22}$*. Using a bigger dictionary*

$$\hat{x} = (1,8)(1,8)(4,6) \quad D = \textit{\textbf{abaababa\$}}$$

we get again $\frac{12}{22}$*.*

2.3 Shortest representation

An important theoretical concept related to lossless data compression is the notion of *Kolmogorov complexity* of a string [100]. Loosely speaking, it is defined as the length of the shortest program by which an universal machine produces such string at the outset from scratch. Unfortunately, the problem of telling if a string has a Kolmogorov complexity less than a given k is undecidable.

When we restrict the descriptions in terms of macro schemes then the problem of finding the shortest representation becomes decidable, although not tractable. In fact, Storer and Szymanski [149, 155, 156, 151, 152] and Gallant [68] prove that the problem of optimal encoding with most of the macro schemes is intractable.

Theorem 1 *The following problems are* $\mathcal{NP}$*-complete*

- *given a string x and an integer k, determine whether $|\hat{x}| < k$ using EPM scheme, in any of the following situations*
  - *both overlapping pointers and recursion is allowed*
  - *overlapping pointers is allowed and recursion is forbidden*
  - *overlapping pointers is forbidden and recursion is allowed*
  - *both overlapping pointers and recursion is forbidden*
  - *restricting pointers to be unidirectional and any of the above*
- *given a string x and an integer k, determine whether $|\hat{x}| < k$ using CPM scheme when any combination of the restriction to unidirectional pointers, no recursion, and no overlapping is made and regardless whether the pointer size is part of the problem*
- *given a string x and an integer k, determine whether $|\hat{x}| < k$ using OPM scheme with either or both of the recursion and overlapping restriction (with unidirectional or bidirectional pointers)*
- *given a string x and an integer k, determine whether $|\hat{x}| < k$ using OEPM scheme when both recursion and overlapping are forbidden*

The situation for OPM with unidirectional pointers is the only case not shown to be intractable. In fact, the scheme by Lempel and Ziv (Sections 2.4 and 2.5) falls within the framework of OPM with left pointers and topological recursion [178, 179]. Rodeh *et al.* present a linear time algorithm for that scheme [131].

As we will see, the algorithms in Section 2.7 can be regarded as an instance of the EPM scheme where the dictionary is specified in advance. The running time is quadratic in the size of the source.

2.4 Lempel-Ziv 77

The typical implementation of LZ-77 [179] processes data *on-line* as it is read, i.e., parses the file sequentially *left to right* and looks through a fixed size window of past symbols to find the longest match of the string starting at the current position. The string is then substituted with a *pointer* (e.g., the triplet of displacement, length, last symbol) which should result in a shorter representation.

Formally, suppose that we have the prefix of a string w already parsed in $p-1$ phrases as $w[1,i-1] = y_1 y_2 \ldots y_{p-1}$, and we want to define the p-th phrase $y_p = w[i,j], j \geq i$.

The LZ-77 scheme looks for the longest prefix of $w[i,n]$ that matches a substring of $w[1,i-1]$. In other words, the length of the prefix can be expressed as $\max_{k \in [1,i-1]}\{\, l \mid w[i,i+l] = w[k,k+l-1] \cdot c, c \in \Sigma\}$. The prefix $w[i,i+l]$

becomes the p-th phrase which is encoded as a pointer (k, l, c). The current position i is updated to $i + l + 1$.

A variation of the previous scheme, called LZSS, avoids to send always the new character c adding a bit to the encoding to distinguish a pointer from a literal. Specifically, it encodes a match, called a *citation*, with $(1, k, l)$ (if $l > 0$) and a literal, called an *innovation*, with $(0, c)$ (if $l = 0$).

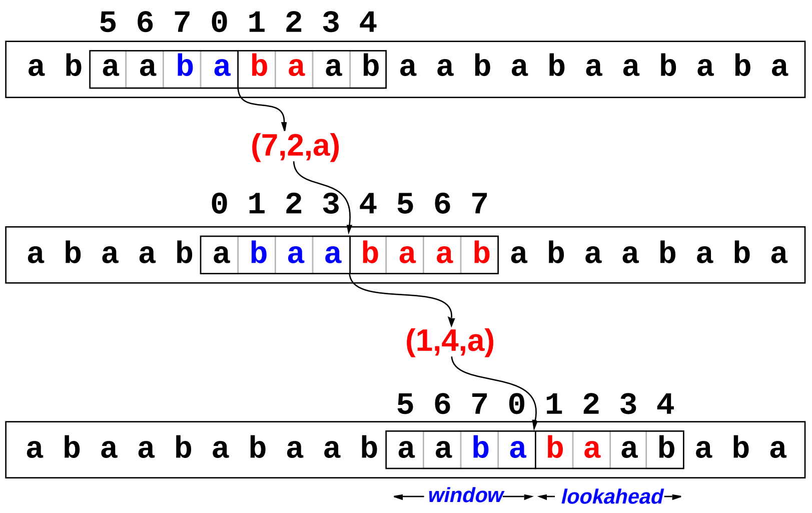

5 6 7 0 1 2 3 4
a b a a b a b a a b a a b a b a a b a b a
(7,2,a)
0 1 2 3 4 5 6 7
a b a a b a b a a b a b a a b a b a
(1,4,a)
5 6 7 0 1 2 3 4
a b a a b a b a a b a a b a b a a b a b a
window
lookahead

Figure 2.1: An example of LZ-77 encoding (sliding window)

Example 2 ***(see Figure 2.1)*** *The sliding window stores the $N = 4$ more recently coded symbols, and the lookahead buffer stores the next N symbols about to be coded. The window and the lookahead form a circular buffer of size $2N$. The first six symbols of the text have just been encoded. The encoder searches the window for the occurrence of the longest prefix of the lookahead and then issues the pointer $(7, 2, \texttt{a})$ which suggests that the next two symbols can be copied from position 7 in the window, and that they are followed by the symbol* `a`*. Then the window is moved to the right three symbols simply overwriting symbols 5, 6 and 7 with the incoming symbols and changing the beginning of the lookahead buffer to position 4.*

The central point for determining the complexity of the algorithm is the search for longest match. Many approaches have been analyzed in [19] ranging from the Boyer-Moore string matching algorithm [26], to digital search trees [7, 131, 60], and from splay trees [146] to hash tables (see for example [62, 48]). In particular if one chooses the suffix tree to create an index of the words in the sliding window then the algorithm runs in amortized linear time [131].

2.5 Lempel-Ziv 78

Instead of allowing pointers to reference any string that appeared previously, the text seen so far is parsed in phrases, where each phrase is the longest phrase already used previously plus one character [180]. The list of phrases that may be references is the *dictionary*.

Initially, the dictionary $y_1, \dots y_{|\Sigma|}$ is initialized with all the symbols of the alphabet. Suppose we have already parsed $w[1, i-1]$ in $p-1$ phrases from the dictionary, and we want to define the p-th phrase $y_p = w[i, j], j \geq i$. At this point the dictionary contains $|\Sigma| + p - 1$ entries.

The LZ-78 encoder looks for the longest prefix of $w[i, n]$ that matches a string in the dictionary y_q for any $q \in [1, |\Sigma|+p-1]$. In other words, the length of the prefix is $\max_{q \in [1,|\Sigma|+p-1]}\{|y_q|$ such that $w[i, i+|y_q|-1] = y_q\}$. The string $w[i, i+|y_q|-1]$ becomes the p-th phrase that is encoded as (q).

The dictionary is updated with the new phrase $y_p = w[i, i+|y_q|]$ and the current position is moved to i to $i+|y_q|$. The dictionary built in this fashion satisfies the *prefix property*: for any given phrase in the dictionary, all its prefixes are also phrases in the dictionary.

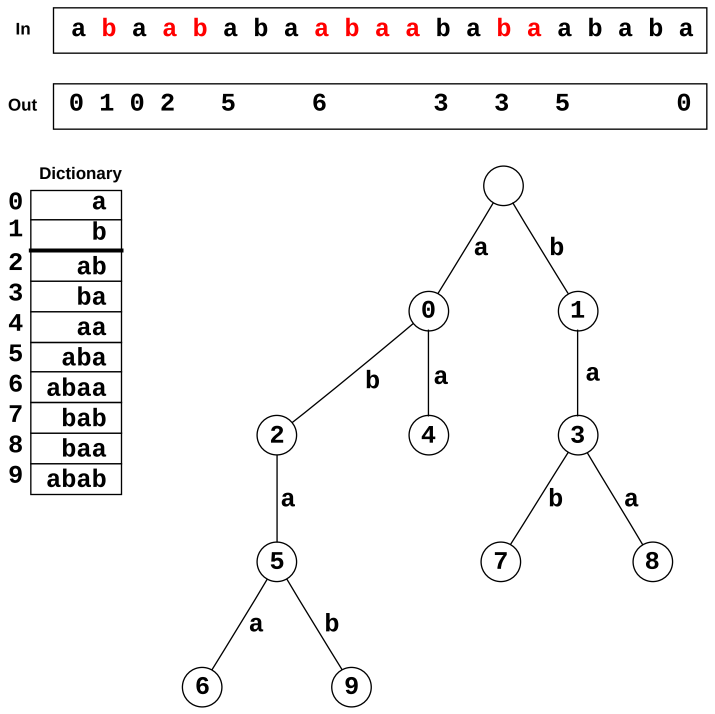


Figure 2.2: An example of LZ-78 encoding

Example 3 ***(see Figure 2.2)*** *At the beginning the dictionary is initialized with the symbols* **a** *and* **b***, respectively with index* 0 *and* 1*. Then the input text is parsed by looking for the longest match of a word in the dictionary: the compressed output is composed by the sequence of indices in the dictionary. At each step, a new string composed by the longest match plus the innovation is added to the dictionary.*

LZ-78 encoding can be implemented in linear time by use of simple trie data structure [7] with space complexity proportional to the number of codewords in the output.

The decoding is symmetric to the encoding. The dictionary is initialized with all the symbols in Σ. The dictionary is recovered while the decompression process runs: indeed, the dictionary is not transmitted. When the decoder reads a symbol (q) from the encoded file it outputs y_q and it adds the word $y_q \cdot c$ to the dictionary where c is the first symbol of the next phrase.

Because we need the first symbol of the next phrase, the dictionary is updated only after the new phrase is decoded. A problem occurs if the next symbol of the encoding is (r) which is precisely the index of $y_q \cdot c$. Indeed, this happens only in a very special case, when the symbol c is also the first symbol of y_q. This arises if the text contains the string $awawa$ where $a \in \Sigma$ and $w \in \Sigma^*$, such that aw belongs to the dictionary and awa does not. In the decoding we have $y_q = aw$ and $y_r = awa$. However, since $y_q a$ is not yet in the dictionary the entry y_r does not exists. The decoder must recognize this situation and add awa to the dictionary immediately after reading (q). This observation has been made by Welch [166].

2.6 LZ-77 vs. LZ-78

Although apparently different, LZ-77 and LZ-78 share some important properties. Both methods are asymptotically optimal in the information-theoretic sense, i.e., the average code length of a symbol tends asymptotically to the entropy of the source [178, 179, 180], even for sliding window schemes [173].

However, it is not always clear how fast these schemes converge to the best achievable compression. There are several studies on the *redundancy* (defined as the difference between the average code rate and the source entropy, when the memory size for the algorithm is limited) of the Lempel-Ziv code.

It has been recently shown that LZ-78 approaches asymptotic optimality faster than LZ-77. The average number of bits generated by LZ-78 for the first n symbols of an input string from an i.i.d. source is only $O(1/\log n)$ more than its entropy [83, 105, 141]. However, for the LZ-77 algorithm this redundancy is as much as $O(\log\log n/\log n)$ [172, 106, 107].

Another recent result by Kosaraju and Manzini [92] states that for low entropy strings, the worst case compression ratio obtained by LZ-78 is better by a factor 8/3 than the compression ratio of the LZ-77 algorithm.

2.7 Optimal encoding given a static dictionary

A dictionary can be constructed in static or dynamic fashion. In *static* schemes the whole dictionary is constructed before the input is compressed: both the encoder and the decoder share a fixed, given dictionary. By contrast, the encoder may transmit to the decoder the dictionary before transmitting the pointers;

or the compressor and the decompressor may evolve a common *dynamic* dictionary through the coding process. In the latter case, as in LZ-77 and LZ-78 schemes, the dictionary is initially empty and is constructed incrementally.

In the Section 2.3 we mentioned that the selection of the best possible dictionary is a $\mathcal{NP}$-complete problem. But what if we are *given* a static dictionary?

We define the *optimal parsing* relative to an *arbitrary* static dictionary, the parsing that produces the least number of phrases (and pointers). A more complex problem is finding the *optimal encoding* in the case of non-uniform pointer size.

The optimal parsing can be found in $O(n^2)$ time via dynamic programming as proposed by Wagner [163]. It is an off-line algorithm that requires the entire input to be stored in main memory. Alternatively, the optimal parsing can be rephrased as the problem of finding a shortest path between a given pair of vertices in a directed network [142].

For prefix dictionaries, the on-line algorithm by Hartman and Rodeh [77] runs in $O(nd)$ time where d is the longest string in the dictionary. If the prefix dictionary comes from a process like Lempel-Ziv parsing where successive insertions can grow by at most one symbol, then $d = O(\sqrt{n})$ and the algorithm runs in time $O(n^{3/2})$. The result has been improved to linear time by Cohn and Khazan [35].

In general, the optimal encoding has been considered impractical because of its computation cost. Therefore, several heuristic algorithms have been developed, e.g., the longest match heuristic, the longest fragment first heuristic (LFF), and the greedy heuristic [70, 87, 88, 18, 35]. In particular Katajainen and Raita [87] propose an approximation algorithm for the optimal parsing for a given static dictionary.

2.8 Off-line/on-line encoding and decoding

The theoretical studies on the performance of LZ schemes in the case of one-way head machines (on-line algorithms) or two-way head machines (off-line algorithms) can be divided in two classes. The studies that address the problem asymptotically and the papers that consider the finite case.

Ziv and Lempel investigate the encoding power of a finite state one-way head machine with an unrestricted decoder and show that the lower bound on the achievable compression ratio can be attained asymptotically [179, 180]. The same bound is achieved when both the encoder and the decoder are one-way head machines. Later, Sheinwald, Lempel and Ziv [144, 145] prove that the bound can be attained when the encoder is unrestricted and the decoder is a finite state one-way machine. As far as asymptotic results are concerned, the power of off-line coding is not useful if we want to be able to decode on-line.

In the finite case, De Agostino and Storer study the performance of on-line and off-line coding and decoding. If the decoding has to be on-line, then the problem of finding, off-line, the dictionary with prefix property that parses the string in the minimum number of phrases is $\mathcal{NP}$-complete [46]. Furthermore, they prove that a sub-logarithmic factor approximation algorithm cannot be realized on-line. Later, De Agostino and Silvestri [44] show that LZ-78 and two widely used greedy strategies produce in the worst case an $O(n^{1/4})$ approximation of the optimal parsing. These results suggest that the power of an off-line encoder is needed in order to approximate the optimal parsing.

Chapter

3

Off-line

In the previous Chapter we mentioned that the problem of finding the optimal representation in the world of macro schemes is $\mathcal{NP}$-complete. Notably, only a couple of schemes have polynomial time algorithms.

Our interest in the approach described in this Thesis can be regarded as an "exploration of the algorithmic space" between the linear time algorithms for textual substitution (LZ-77, LZ-78) and the exponential time algorithms for optimal encoding for general macro schemes (see Figure 3.1). As far as we know, that space is largely unexplored. With the exception of the quadratic-time algorithm for the optimal encoding with static dictionary no other textual substitution algorithm "inhabit" that space.

We wonder if one could design an approximation scheme of the optimal encoding that runs in polynomial time. Or if one could extend Lempel-Ziv schemes to achieve greater compression at the expenses of higher time complexity. In other words, we would like to understand if one could "trade" time complexity for compression efficiency.

3.1 Steepest descent

The problem of finding the shortest representation of a given string can be formulated as an optimization problem. The feasible solutions are all the possible pointer-based representations of the original string x. The neighborhood $N(\hat{x})$ of a compressed description $\hat{x}$ is the set of all the descriptions obtained from $\hat{x}$ by (1) substituting some of the occurrences of a substring $w \in \mathcal{X}$ with pointers (*contraction*) or (2) substituting one of the pointers with its target (*expansion*). The objective function corresponds to the length of the compressed description, $|\hat{x}|$. The goal is to find, among all the $\hat{x}$ in the solution space, a representation $\hat{x}_{OPT}$ which attains the minimum of the objective function.

Steepest descent is a general approach for the minimization of complex functions. Its strategy is to move in the space of the feasible solutions from the current $\hat{x}$ to one of the elements in $N(\hat{x})$ based on the largest improvement

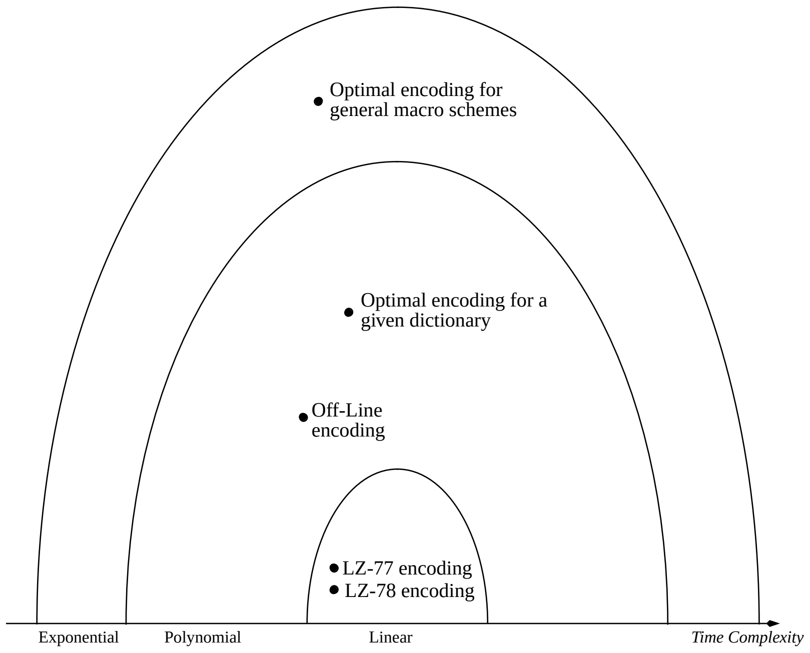


Figure 3.1: The hierarchy of time-complexity for macro schemes

in the objective function. Formally, given the current $\hat{x}$ the algorithm moves to the element $\mathrm{argmax}_{y \in N(\hat{x})}\{|\hat{x}| - |y|\}$. The direction of the move is towards the highest contraction in the neighborhood. Therefore, it will never consider expansions and it will always try to substitute the maximal number of occurrences with pointers. However, expansion moves could be necessary to escape from a local minimum. Indeed, the method is not guaranteed to converge to the shortest representation $\hat{x}_{OPT}$.

Contraction steps involve checking all the substrings in $\mathcal{X}$ to find the most "convenient" substitution. Let us suppose to have a function $G : \mathcal{X} \rightarrow \mathbf{R}$ that computes the difference in bits between the plain text representation of a substring $w \in \mathcal{X}$ and its pointer-based equivalent. As the first move, steepest descent suggests to substitute, among all the words in $\mathcal{X}$, the word w_1 which maximizes G.

The value of $G(w)$ is computed upon the length m_w and the number of *non-overlapping* occurrences f_w. We distinguish the number of occurrences of w from the number of non-overlapping occurrences of w. Two occurrences of w are said to be *non-overlapping* if their respective starting positions, say i and j, satisfy $|i - j| \geq |w|$. In general, f_w may be different from the total number of occurrences of w.

We are interested in counting non-overlapping occurrences because we substitute these occurrences with pointers to a common copy. If two occurrences overlap there is no way to substitute both with pointers. The maximal number of substitutions of w with pointers is therefore given by f_w.

In general, the maximal set of non-overlapping occurrences is not unique.

The set we consider is the set obtained by scanning the text left to right and greedily choosing the next non-overlapping occurrence.

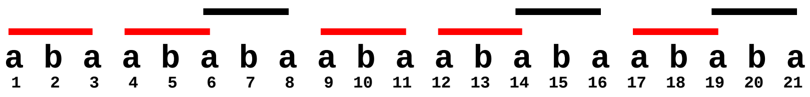


Figure 3.2: Overlapping and non-overlapping occurrences

Example 4 ***(Figure 3.2)*** *The substring $w = \textbf{aba}$ occurs eight times in $x = \textbf{abaababaabaababaababa\$}$. However the occurrences at position 4 and 6, 12 and 14, 17 and 19 overlap with each other. Since there are no more than five non-overlapping occurrences of w in x, then $f_{\textbf{aba}} = 5$. The maximal number of pointers that could substitute the occurrences of w is $f_{\textbf{aba}}$. The sets of positions of maximal cardinality are* {1, 4, 9, 12, 17}, {1, 4, 9, 12, 19}, {1, 4, 9, 14, 17}, {1, 4, 9, 14, 19}, {1, 6, 9, 12, 17}, {1, 6, 9, 12, 19}, {1, 6, 9, 14, 17} *and* {1, 6, 9, 14, 19}. *We choose* {1, 4, 9, 12, 17}.

Once that w_1 has been substituted with pointers, the substring w_2 which maximizes G is considered. Note that G has changed after the substitution of w_1. All the words which occurrences overlap with any of the occurrences of w_1 have their statistics changed. Some of them could not be represented anymore in $\mathcal{X}$. Therefore, the parameters on which G is based should be "updated" after each substitution.

The entire process is iterated until no words in x capable of producing further compression can be found. The general paradigm considered in this Thesis is shown in Figure 3.3.

```
OFF-LINE (string $x$)
repeat
    $D$ = < index containing $f_w$, for every substring $w$ of the text $x$ >
    $s$ = < the substring in $D$ which maximizes $G$ >
    $x$ = < $x$ where the occurrences of $s$ are substituted with pointers >
until < no further compression of $x$ can be obtained >
```

Figure 3.3: The general paradigm

In order to fully specify OFF-LINE we need to define: (1) the structure of the pointer-based representation, that is, the specific macro scheme (2) the function G and (3) the substitution algorithm, that is, the implemention of (1).

3.2 Choosing the gain function

By *gain measure* $G : \mathcal{X} \to \mathbf{R}$ we refer to the function which evaluates the "convenience" of a particular substring substitution. $G(w)$ computes the number of bits saved by transforming w in a pointer-based representation, that is, by substituting the occurrences of the w in the text with pointers to a common copy. We assume to know the statistics of every word in $\mathcal{X}$, in particular to know the number f_w and the positions of a maximal set of non-overlapping occurrences of w.

Unfortunately, it is not easy to define precisely G because, when we want to predict the contraction that the substitution of a particular word will induce, we lack the actual costs associated to the encoding of the pointers which can be computed only at the end of the entire process.

Letting $l(z)$ be the number of bits needed to encode an integer z, we assume for simplicity that $l(z) = \lceil \log z \rceil$, even if we could choose more accurate measures[1] for the encoding of integers in an unknown range [10, 51, 131]. However, since the final encoding of the compressed string is not based on any such representation, but rather on some statistical encoding (i.e., Huffman or arithmetic coding) there is no way to compute accurately G at this stage.

In the rest of the Chapter we describe three possible measures of the gain respectively associated with three different substitution schemes. In terms of Storer's classification of Section 2.2, our pointers are always bidirectional. Schemes 1 and 3 use external and non-overlapping pointers, while Scheme 2 uses internal and overlapping pointers. Pointers are always compressed because of the iterative steepest descent strategy.

3.2.1 Scheme 1

Let us suppose that w is the word that maximizes G at the i-th iteration and thus it is selected to be substituted by the algorithm in Figure 3.3. Scheme 1 removes from the text *all* the f_w occurrences of the word w and compacts the text. The word w is saved in the dictionary with its length, its non-overlapping frequency f_w and the positions where it appeared in the text. In order to use small integers the positions of the occurrences of w are not saved as absolute values. They are stored instead as consecutive increments each relative to the end of the preceding occurrence of w.

Specifically, the items of information we need to save in order to be able to recover the original string are

- the string w, that requires Bm_w bits, where B is $\log |\Sigma|$ and Σ is the alphabet;
- the length of the string, m_w, that requires $l(m_w)$ bits;
- the number of occurrences, f_w, that requires $l(f_w)$ bits;
- the f_w positions of w in x, that require $f_w l(n)$ bits.

Figure 3.4 shows the two representations and their associated costs. The figure at the top illustrates the cost of representing the occurrences of w as

[1] specifically, we can represent an integer z with $l(z) = \log z + O(\log(1 + \log z))$ bits

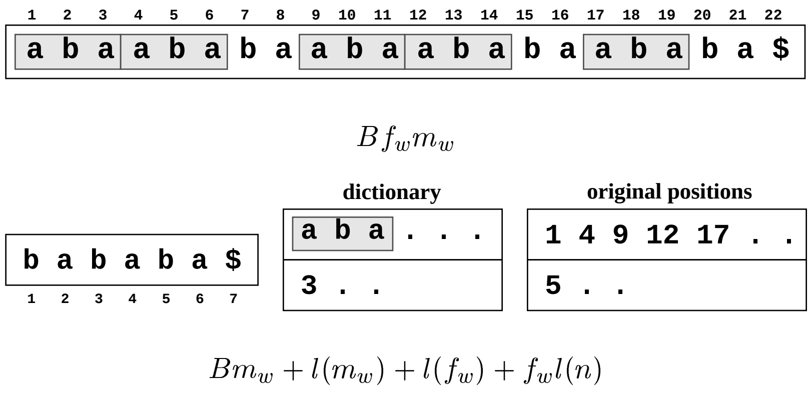


Figure 3.4: Scheme 1. The word $w =$ aba is selected to be substituted. The difference of the two expressions gives the gain in bits of transforming w to a pointer-based representation

plain text. The f_w copies of w require a total of Bf_wm_w bits in the original string. The bottom figure shows the cost of representing w using the external dictionary. The difference of the two expressions defines $G_1(w)$ as follows

$$\begin{aligned} G_1(w) &= Bf_wm_w - l(m_w) - Bm_w - l(f_w) - l(n)f_w \\ &= (f_w - 1)Bm_w - l(m_w) - l(f_w) - l(n)f_w \end{aligned}$$

Note that $G_1(w)$ depends only from f_w and m_w, while n and B are constants.

The final encoding of Scheme 1 is then composed by

- the text with all the redundant words removed;
- the dictionary;
- the length of the words in the dictionary,
- their original (relative) positions,
- their maximum number of non-overlapping occurrences.

Note that in this scheme we do not need extra bits for distinguishing literal from pointers.

A graphical representation of the values assigned by the function G_1 to the words of the fifth Fibonacci string is shown in Figure 3.5 (compare with Figure 3.7). The picture has been obtained with VERBUMCULUS [8, 9], a tool for the detection of over- and under-represented words in sequences. The tree captures all the substrings in the original strings. The substrings are represented by the labels spelled out on the paths from the root to any internal node. The font size of the labels is proportional to the value of G_1. The negative values of G_1 are shown with an italic font.

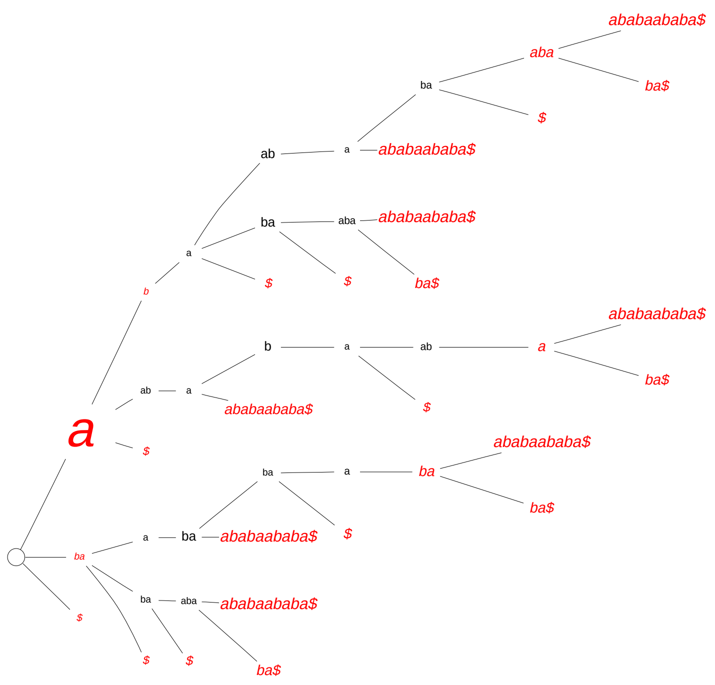


Figure 3.5: A graphical representation of the value of $G_1(w)$ for all the words in the Fibonacci string abaababaabaababaababa$ (computed on the augmented suffix tree)

3.2.2 Scheme 2

If w is the word selected at the i-th iteration, Scheme 2 substitutes all the occurrences of w *except one* with pointers to the original copy. The pointer is a pair in the form (*position*, *length*). In order to keep the size of the pointers short, we store the position as a displacement. A bit-vector is needed to distinguish pointers from literals in the text. On the other hand, Scheme 2 does not need an external dictionary.

This substitution strategy is an iterative "original internal pointer macro" scheme (OPM) in terms of Storer's classification (see Section 2.2). We forbid pointer recursion, that is, we assume that we cannot substitute words that contains other pointers inside[2]. The pointers are bidirectional.

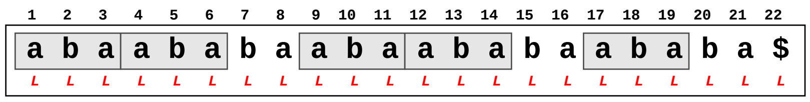
The plain text representation of all the occurrences of w requires now $(B+1)f_w m_w$ bits, because of the additional bit. The pointer-based representation costs are as follows (see Figure 3.6)

- $(B+1)m_w$ bits for the original copy of w
- $(f_w-1)(l(n)+l(m_w)+1)$ bits for the f_w-1 pointers

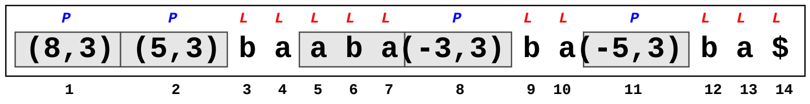
The difference of these expressions defines G_2 as follows

$$G_2(w) \quad = \quad (B+1)f_w m_w - (B+1)m_w - (f_w-1)(l(n)+l(m_w)+1)$$

[2]Thus, no bit-vector is needed for the dictionary


1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
a b a a b a b a a b a a b a b a a b a b a $
L L L L L L L L L L L L L L L L L L L L L L

$$(B+1)f_w m_w$$


P P L L L L L P L L P L L L
(8,3) (5,3) b a a b a (-3,3) b a (-5,3) b a $
1 2 3 4 5 6 7 8 9 10 11 12 13 14

$$(B+1)m_w + (f_w-1)(l(n)+l(m_w)+1)$$

Figure 3.6: Scheme 2. The word $w =$aba is selected to be substituted. The difference of the two expressions gives the gain in bits of transforming w to a pointer-based representation. (L means literal, P means pointer; the pair (p,l) is a pointer to a reference copy that is p symbols distant and has length l)

$$\begin{aligned} &= (f_w-1)(B+1)m_w - (f_w-1)(l(n)+l(m_w)+1) \\ &= (f_w-1)((B+1)m_w - l(n) - l(m_w) - 1) \end{aligned}$$

With respect to Scheme 1 we have one additional degree of freedom. We can choose which one of the original copies should remain untouched in the text or we could even leave more than one copy in the text. The problem to select the copy that gives the best compression has not been addressed. We always choose to leave one reference copy in the middle of the sequence of occurrences, in order to try to minimize the size of the pointers.

The final encoding is therefore composed by

- the text with the redundant words substituted with pointers,
- a bit-vector of the same size of the text, indicating pointers and literals.

Figure 3.7 displays the value of the gain function on the suffix tree for our running example. The font size of the labels is proportional to the value of G_2. The negative values of G_2 are shown with an italic font.

3.2.3 Scheme 3

In Scheme 3, words are substituted with pointers to entries in an external dictionary. If the substring w is selected at the i-th iteration, we append w to the dictionary and we replace *all* the occurrences of w with pointers to the entry of w in the dictionary. The dictionary is composed by the concatenation of words and their lengths. Again, a bit-vector is required to distinguish pointers from literals in the text.

We forbid pointer recursion, that is, we assume that words in the dictionary do not contain pointers. Therefore, we do not need a bit-vector for the dictionary.

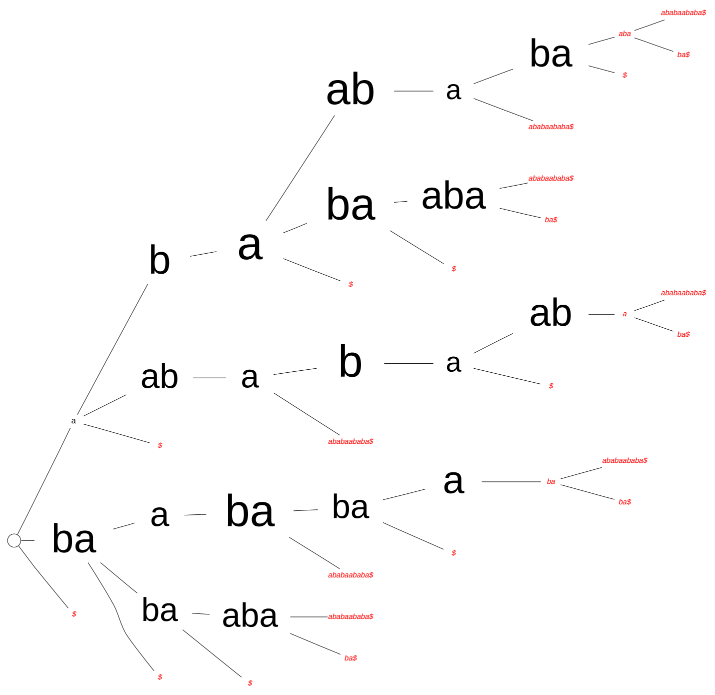


Figure 3.7: A graphical representation of $G_2(w)$ on the augmented suffix tree for the Fibonacci string abaababaabaababaababa$

The plain text representation of all the occurrences of w requires $(B+1)f_w m_w$ bits. The costs of the pointer-based representation (see Figure 3.8) are the following

- Bm_w bits for the string w in the dictionary,
- $l(m_w)$ to store the length m_w,
- $l(d)f_w$ for the f_w pointers inside the text, where d is the size of the dictionary.

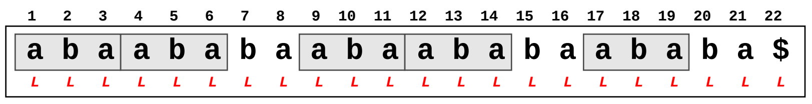
The difference of the expressions defines $G_3(w)$ as follows

$$\begin{aligned} G_3(w) &= (B+1)f_w m_w - Bm_w - l(d)f_w - l(m_w) \\ &= B(f_w - 1)m_w + f_w m_w - l(d)f_w - l(m_w) \end{aligned}$$

Note that at the time of evaluating G_3 we cannot predict the value of d. This problem will be addressed in Section 4.5.3.

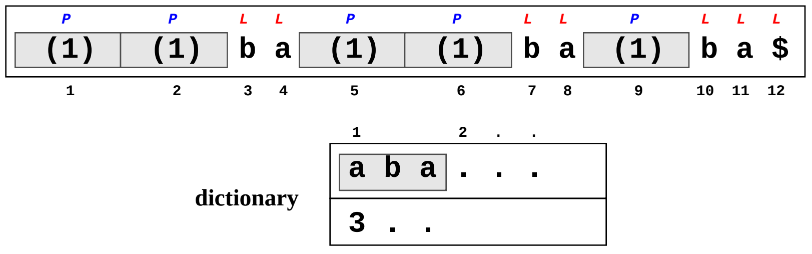
The final encoding is composed by

- the text with the redundant words substituted with pointers,
- a bit-vector of the same size of the text, indicating pointers and literals,



1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
a b a a b a b a a b a a b a b a a b a b a $
L L L L L L L L L L L L L L L L L L L L L L

$$(B+1)f_w m_w$$



$$Bm_w + l(d)f_w + l(m_w)$$

Figure 3.8: Scheme 3. The word w = aba is selected to be substituted. The difference of the two expressions gives the gain in bits of transforming w to a pointer-based representation, where d is the size of the dictionary (L means literal, P means pointer and (1) refers to the first entry in the dictionary)

- the dictionary,
- the lengths of the words in the dictionary.

Chapter 4

Implementation

In first part of the Chapter we describe how to build and maintain two data structures that play a fundamental role in the implementation of OFF-LINE. The first is the *fragment tree*, a balanced tree of text fragments allowing one to search and substitute efficiently all the occurrences of the words. The second is the *augmented suffix tree*, a text index used to allocate and compute the maximal number of non-overlapping occurrences for each substring in the text.

In the second part, we analyze the type and performance of various statistical encodings applied to the internal representation produced by the three versions of OFF-LINE. We propose some heuristics to speed up the compression process. Finally, we describe the software implementing our scheme, and report the results.

4.1 The fragment tree

In Chapter 3 we saw that our compression paradigm requires efficient string searching [90, 26] over dynamically updates of the text. Since these two operations are somewhat contrasting the problem becomes quite involved.

The dynamic text indexing problem has been studied by Gu *et al.* [74] by introducing a data structure which permits insertion and deletion of a single character in $O(\log n)$ time and searching for all occ_w occurrences of a string w (after i update operations in the data structure) in time $O(|w| + occ_w \log i + i \log |w|)$. Other recent papers address the dynamic text indexing problem [58, 59].

A naive implementation keeps two data structures, a dynamic array and a single linked list, that both contains the same text. The array is used to search, the list to update the text. The two structure should be maintained synchronized by copying one into the other at the end of each series of updates. The cost of the synchronization is $\Theta(n)$ and the list requires at least $5n$ bytes of memory.

A better solution is to build and maintain a *fragment tree*, a balanced search tree of text fragments. *Text fragments* are substrings of the original text left untouched after the removal of some words. The concatenation of these fragments represents the text after the update operation.

We use a red-black tree where each node is associated with a text fragment. Each node contains the fields *key*, *color*, *p*, *left*, *right*, *begin*, and *end*. The *key* corresponds to the ending position of the fragment. The fields *color* and *p* are used to maintain balanced the red-black tree during insertions and deletions (see e.g., [37]). The pointers *left* and *right* point respectively to the left and right child of the node. The fields *begin* and *end* are two pointers to the beginning and the ending position of the text fragment represented in that node.

The operations defined on fragment tree are the following

- CREATE (**string** x), creates a fragment tree which consists of a single node encompassing the entire text-string x that constitutes the only fragment
- SEARCH (**string** s), returns the maximal set of non-overlapping occurrences of s in the text
- REMOVE (**string** s), deletes all the occurrences of s from the text
- SUBSTITUTE (**string** s_1, **string** s_2), replaces all the occurrences[1] of s_1 with s_2

The access to the i-th element of the text is obtained as follows. We use the position i as a query in the search tree. Since the tree is kept balanced, the search takes time logarithmic in the number of fragments. Once the fragment containing the position i is found, the symbol $x[i]$ is retrieved in constant time.

The SEARCH operation is based on the KMP linear time string searching algorithm [90]. The operations REMOVE and SUBSTITUTE split the array in several fragments. The effect of these operations is reflected by introducing new nodes in the tree, but the array in which the text x is stored is not modified. Only the structure of the tree and the content of the nodes are updated.

Example 5 *Initially, the fragment tree consists of a single node (see Figure 4.1). The deletion of the occurrences of the substring* **aba** *splits the array into three text fragments (see Figure 4.2). The nodes for the fragments* **ba**, **ba**, *and* **ba$** *are created. Now, if we want to read $x[5]$ we use "5" as a query. We start the search from the root of the tree. Since $5 > 4$ we move to the node at its right. We stop there because $5 < 7$. We access the array through the pointers, and we finally get* **b**.

If the size of the tree becomes too large then the SEARCH operation becomes too expensive. To counteract excessive fragmentation the fragment tree is compacted periodically in a single array by refresh cycles.

[1]we assume $|s_1| \geq |s_2|$ because our substitutions are aimed to compress the text

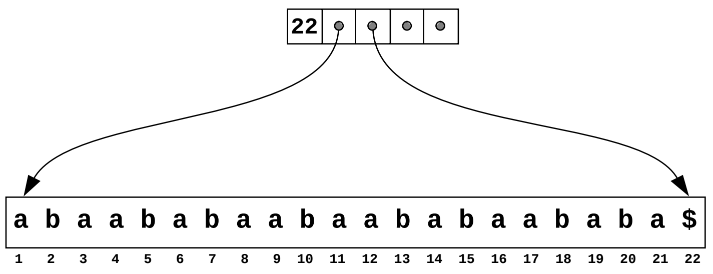


Figure 4.1: The initial fragment tree for abaababaabaababaababa$. Only the fields *key*, *begin*, *end*, *left* and *right* are displayed

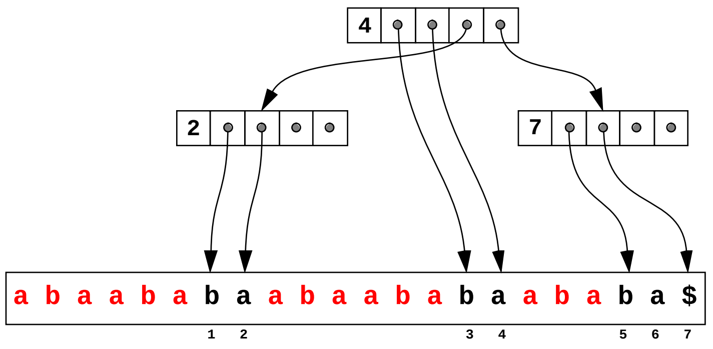


Figure 4.2: The fragment tree after the removal of the occurrences of aba

The performance of this data structure has been compared with our first naive implementation. The new data structure improves the overall performance of the compression algorithm by at least 20% and uses a fraction of the memory.

4.2 Statistical indexing of the text

To compute and store the statistics of the words in the text we choose the *suffix tree*. The suffix tree is a digital search tree that embodies a compact index of all distinct, non-empty substrings of the text (see e.g., [7, 148, 40, 75]).

The suffix tree $T(x)$ for the string $x =$ abaababaabaababaababa$ is shown in Figure 4.3. The paths of the tree are *compressed*, i.e., only nodes with more than one outgoing edge are represented. The tree $T(x)$ has $n+1$ leaves, labeled from 1 to $n+1$. Since each node has at least two children (and at most $|\Sigma|+1$), the tree has $O(n)$ nodes overall.

The i-th suffix of x, that is, the suffix that starts at position $i \in [1, n+1]$, may be read on the tree by concatenating the words on the edges of a path from the root to the leaf labeled by i. The end-marker $\$ \notin \Sigma$ ensures that no suffix of $x\$$ can be a prefix of another suffix, and hence there is a one-to-one correspondence between the leaves and the non-empty suffixes of x.

On the other hand, the substrings of x can be obtained by spelling out the words from the root to any internal node of $T(x)$ or to any position in the

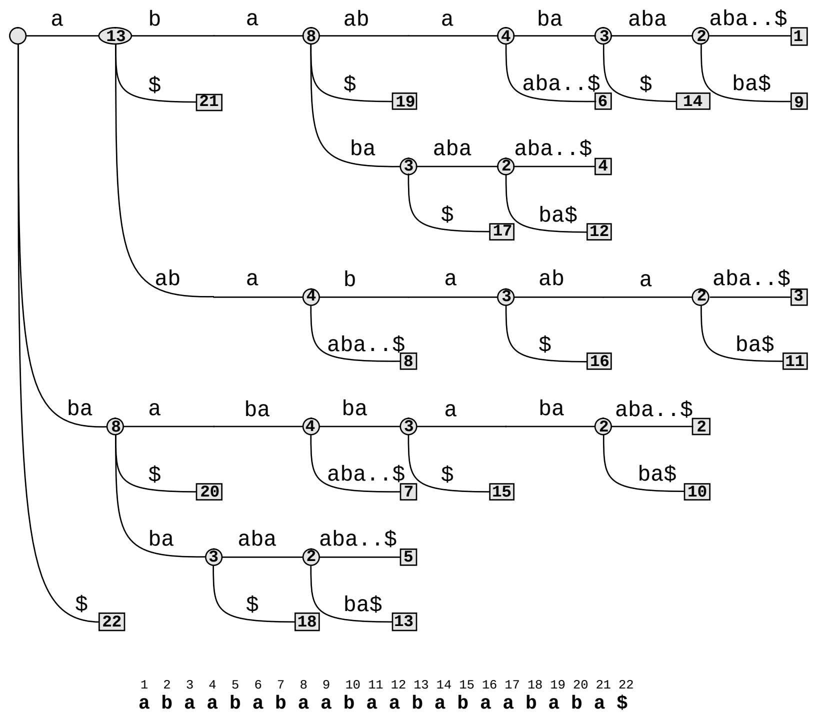


Figure 4.3: The suffix tree T(abaababaabaababaababa$), with internal nodes storing the number of (overlapping) occurrences

middle of an edge. For any two substrings of x, if y is a prefix they have in common, then the path in $T(x)$ relative to y is the same for the two substrings.

To achieve overall linear-space allocation, the words on the edges are not stored explicitly. For each word, it suffices to save an ordered pair of integers indexing one of the occurrences of the word in the text. Each edge label requires thus constant space, which, in conjunction with the fact that total number of nodes is bounded by $O(n)$, results in the overall linear space for the tree.

The computation of the statistics of all substrings of a string is a direct application of suffix trees. We denote by $<w>$ the node, if it exists, precisely at the end of the path in $T(x)$ labeled with w. If instead w ends in the middle of an arc then $<w>$ denotes the node corresponding to the shortest extension of w that ends in a node. Then, the number of occurrences of a string w is given by the number of leaves in the subtree rooted at $<w>$. In Figure 4.3, the number of occurrences is shown inside the internal nodes.

Many algorithms exist for the construction of suffix trees. A "brute force" algorithm, for instance, would consist of inserting the suffixes of x one at a time starting with the longest and progressing to the shortest (see Figure 4.4). In the worst case it requires quadratic time while the average time complexity is $O(n \log n)$ [14] (cf. also [31]).

Several clever constructions are available. The algorithm by McCreight [112] and the one by Chen and Seiferas [32] are variation of the Weiner's algorithm [165] and require only linear time for finite alphabet. These algorithms are off-

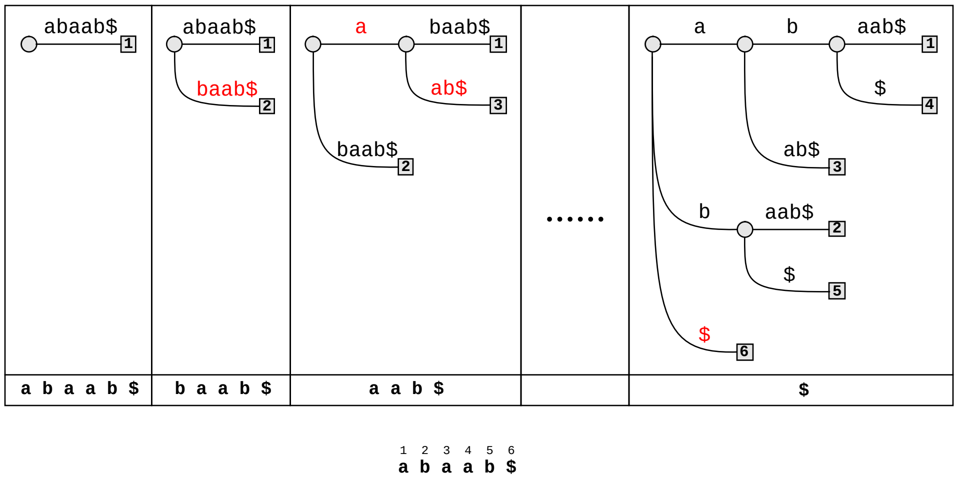


Figure 4.4: The direct construction of the suffix tree T(abaab$)

line. A more recent construction by Ukkonen [160] achieves linear time on-line. Recently, Farach [52] proposes an optimal construction for large alphabets.

4.3 Augmented suffix tree

If we want to build an index similar to a weighted version of $T(x)$, only this time for the statistics without overlap then the problem becomes more involved. A perusal of Figures 4.3 and 4.5 shows that this transition induces a twofold change in the tree: the number of occurrences in each internal node does not longer necessarily coincide with the number of leaves; moreover, extra nodes must now be introduced to account for changes in the statistics which occur in the middle of the arcs.

The efficient construction of the augmented index in minimal form (i.e., with the minimum possible number of unary nodes) is quite elaborate. The resulting structure, called *minimal augmented* suffix tree, is denoted with $\hat{T}(x)$. The algorithm by Apostolico and Preparata [13] requires a post processing to *augment* the tree with the extra unary nodes (at most $O(n \log n)$) that results in a worst-case complexity of $O(n \log^2 n)$. However, the brute force construction of $T(x)$ can be adapted to produce $\hat{T}(x)$ directly in $O(n \log n)$ time on average, and $O(n^2)$ time in the worst case. The number of auxiliary nodes was bounded in [13] by $O(n \log n)$, but a tighter, $O(n)$ bound, was claimed recently [27].

4.4 Implementing the augmented suffix tree

The allocation in main memory of the suffix tree presents a spectrum of choices which are related to the actual structure of the nodes. We are interested in structures that allow to find for each node and symbol of the alphabet the corresponding outgoing edge as quickly as possible. Unfortunately, the fastest

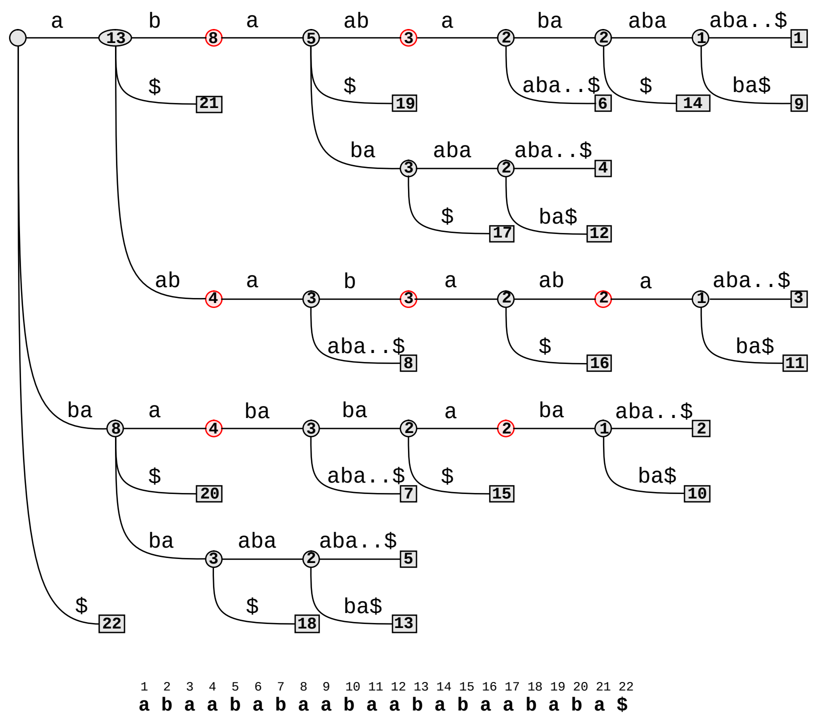


Figure 4.5: The augmented suffix tree $\hat{T}$(abaababaabaababaababa$) with internal nodes (original & auxiliary) storing the number of non-overlapping occurrences

ways are the most expensive in terms of memory usage.

We can arrange the outgoing pointers of each node as an array of size $|\Sigma|$. If we index directly the array with the first character of the word labeling the outgoing edge, we achieve $O(1)$ time to find the next node to visit. The array of pointers yields fast searches, but it introduces an amount of waste space even for small alphabets because many entries will not be used.

At the other end of the spectrum, we could arrange the pointers to the descendants of a node as a linked list (or as a balanced search tree). This keeps the space to a minimum, but introduce an overhead on the search ($O(|\Sigma|)$ time in the case of the list, $O(\log|\Sigma|)$ time for the balanced tree). If the alphabet is ordered we can keep the list of the outgoing pointers sorted by the edge labels with no extra cost. This somewhat reduces the time to search for a given character and thus speeds up in practice the construction of the tree.

Finally, the adjacency of a node could be realized as part of a global hash table. This yields worst case linear time searching [62, 48], but the space for the table could be considerably large if we want to keep collisions low.

In actual applications the best design is probably a clever mix of the above choices. Nodes near the root tend to have $|\Sigma|$ children, so the array would be the best choice. Moreover, if the first l levels of the tree are very dense we could "compress" and eliminate those level by having an l-dimensional array indexed by a prefix of length l instead of using a single character [113]. Nodes near the leaves have usually few descendants so lists can be attractive. For the nodes in the middle of the tree, hashing or balanced trees may be the best option.

In our case the space is of high practical concern, so that we use the linked list. Specifically, the structure of a node $<w>$ contains: (1) two indices $[i, j]$ which identify the beginning and the the end of the last non-overlapping occurrence of a suffix of w of size $j - i + 1$ in x [2], (2) one pointer to the list of children, (3) one to the list of siblings, and (4) one counter for the number of non-overlapping occurrences, f_w.

As said, the suffix trees require linear space. However, in practice even the multiplicative constant involved matters. At 20 bytes per node and with an average $1.5n$ number of nodes as observed in our experiments, a text of size n needs about $30n$ bytes of storage space.

Although the size of the suffix tree depends on the particular implementation, one might expect it to be never smaller than 20 bytes per symbols (*bps*) in the worst case [94]. Other related or alternative data structures have been devised to relieve space requirements, such as the PATRICIA tree (12 bps) [118], the suffix-array (9 bps) [108], the suffix-cactus (10 bps) [85, 86], the level compressed trie (12 bps) [4], the suffix binary search tree (10 bps) [82] and the compact dyrected acyclic word graphs (36 bps) [23, 24, 25, 41]. Generally space savings are achieved at the expense of a higher time complexity in the construction or in the query: for instance, the suffix array and the PAT tree need $O(n \log n)$ time for the construction ($O(n)$ on average for the suffix array) and $O(|w| + \log n)$ time when searching for a string w.

4.4.1 Computing the gain function

The function G discussed in Section 3.2 has to be computed for each substring in the text x. However the cardinality of $\mathcal{X}$ is $\Theta(n^2)$ in the worst case, and that would expose us to a quadratic time algorithm.

Fortunately we can prove we can neglect words that end in the middle of an arc. If we compute G only at the internal nodes of $\hat{T}(x)$ we are not missing any potential substitution. Therefore the annotation of the augmented suffix tree with the value of the gain takes time proportional to the number of nodes in the tree, that is $O(n \log n)$ or better.

The latter corresponds to the time complexity for each iteration of the encoder. The overall complexity is therefore $O(mn \log n)$ where m is the number of iterations. The parameter m depends on many factors, such as the alphabet size, and the length and the redundancy of the text.

The proof is based on the two following observations.

Property 1 *$G(w)$ is a monotone increasing function of m_w, that is if we choose two strings s and t such that $m_s < m_t$ and $f_s = f_t$ then $G(s) < G(t)$*

Indeed the family of functions $G(w)$ described in Section 3.2 have the form $cm_w - d \log(m_w) - e$ where c, d and e are independent from m_w. If we define a

[2]i.e., such that $w[|w| - j + i, |w|] = x[i, j]$

new function $g(m) = cm - d\log m - e$, then its derivative $g'(m) = c - d/m$ is always positive for $m > d/c$.

Property 2 *The number of occurrences of w does not change in the middle of an arc.*

This property comes directly from the structure of the suffix tree (see Section 4.2).

We can now prove our statement. Suppose, for the sake of contradiction, that a word w which attains the maximum $G(w)$ ends in the middle of an arc. By Property 1 if we move downward on the arc the number of occurrences does not change, but the length of the string increases. By Property 2, if the length of the string increases then the gain increases as well. If G increases, then w was not the word that maximized G.

We conclude that is safe to disregard the words that end in the middle of an arc and compute the gain only at the internal nodes. A more complete and general study on this somewhat surprising property can be found in [8, 9].

4.4.2 Maintaining the augmented suffix tree

Recall that, according to our scheme, we compute $G(w)$ while visiting the tree, and the word w which maximizes $G(w)$ is selected. The substitution of w with appropriate pointers is performed and the process is repeated. The suffix tree is updated and searched again for the next best substitution. These iterations stop as soon as the optimal $G(w)$ becomes zero or other conditions occur.

Repeatedly building the suffix tree at each iteration results in a considerable computational effort, irrespective of the method adopted. Ideally, we would like to build the tree once and then have an efficient way to maintain it, together with updated statistics, following every substring deletion or substitution.

A linear time algorithm to maintain the tree dynamically under deletion of a string was originally proposed by McCreight [112]. A similar problem, restricted to a single character update, has been studied by Fiala and Green in the context of sliding window compression [60]. More recently, Larsson showed that the algorithm by Ukkonen can be easily extended to accommodate the "sliding window update" of the suffix tree in amortized linear time [98].

However, we did not find any existing solution to the problem of dynamic maintenance of the entire augmented statistical index so to reflect the deletion of *all* the occurrences of a given word. Updating dynamically the tree seems to be much more complicated and expensive than rebuilding the entire tree on the new text.

We then decided to build the suffix tree from scratch at each iteration. Some strategies to speed up the process were also devised, and they will be discussed in Section 4.7.

4.5 Statistical encoders

A crucial point in the design of compression schemes is the coding of the stream of data which is produced by the textual substitution process. It is well-known from many experiments (see e.g., [60]) that the encoding of the pointers is critical and that a fixed length code [49, 51] is unlikely to allocate optimally the bits for displacement and length.

id	*encoding type*
0	Plain
1	Huffman
2	Arithmetic
3	Deflate (ZLIB)
4	RLE
5	RLE + Huffman
6	RLE + Arithmetic
7	RLE + Deflate (ZLIB)

Table 4.1: Statistical encoders available to OFF-LINE. Three bits suffice to describe the type of encoding for each array

At present, OFF-LINE uses the statistical encoders shown in Table 4.1. The Huffman and the Deflate encoder are part of the ZLIB library written by M. Adler and J. L. Gailly [65]. The interface to ZLIB and the arithmetic encoder are adapted versions of programs by M. Nelson [122, 121]. The last four options are composed by a cascade of run-length encoding followed by a statistical encoder.

OFF-LINE picks the most efficient statistical encoding for each one of the arrays generated by the substitution process by comparing their respective final sizes. Three bits for each array are needed to indicate the type of compressor employed. The specific choice depends on the nature of the file and on the type of scheme we are using.

These considerations should now make it clear why a precise definition of G *a priori* is very difficult. Indeed, the parameters of G should depend on entropy estimates of the arrays as produced by the most appropriate statistical encoders. Unfortunately, we do not know such estimates until the substitution process it terminated. But the function G dictates when to stop substituting and what to choose. Moreover, the values assigned to parameters such as the minimum/maximum match length and minimum number of occurrences also have some impact on the entire process. The interplay between the parameters of G and the statistical encoders is very complex and not completely understood at this moment.

In our experiments, the value of B that appears as a parameter in G is estimated by computing the average bit length of a symbol after the compression of the source with the best statistical encoder available in Table 4.1. Indeed, the initial assumption to assign $B = \log |\Sigma|$ bits per symbol is an often incorrect estimate, because it assumes that all the symbols occur with the same probability. To evaluate correctly the cost the original representation, we must take into account the actual entropy of the source.

4.5.1 Off-Line$_1$

An example of output produced by Off-Line$_1$ (i.e., the program implementing Scheme 1) is shown in Figure 4.6. The first iteration results in the choice of `aba`; the second of `ba`.

```
1: abaababaabaababaababa$       Substituted substring: "aba"
2: bababa$                      Substituted substring: "ba"
----------------------------    Final encoding:
sublen = [3 2]
substr = [ababa]
abspol = [0 0]                  abspoh = [0 0]
relpol = [0 2 0 2 0 0]          relpoh = [0 0 0 0 0 0]
occurr = [5 3]
text   = [$]
```

Figure 4.6: A run of Off-Line$_1$ on the string abaababaabaababaababa$

The actual encoding $\hat{x}$ consists of some arrays that contain all the information we need to reconstruct the text. At the end of the i-th iteration, resulting in the choice of the substring w, such arrays are as follows

- `sublen[i]` contains the length of the word w and more precisely m_w − `min_length`; the latter term represents the minimum acceptable length, and is 0 in the previous example but 2 or more in our experiments;

- `substr[k,k+sublen[i]+min_length-1]` records w where `k-1` is the ending position of substring substituted in iteration $i-1$; `substr` is the dictionary;

- `occurr[i]` stores the number of non-overlapping occurrences of w and more precisely f_w − `min_occ`; the latter term represents the minimum acceptable length, and is 0 in the previous example but 2 in our experiments;

- `abspoh[i]` and `abspol[i]` contains the higher and the lower byte of the absolute position of the first occurrence of w;

- `relpoh[j]` and `relpol[j,j+occur[i]+min_occ-1]` record the higher and the lower byte of the consecutive displacements of the other occurrences of w;

File	`text`	`sublen`	`substr`	`occurr`	`abspol`	`abspoh`	`relpol`	`relpoh`
`bib`	Z	R+H	Z	R+H	N	A	N	H
`book1`	A	R+H	Z	R+A	A	A	A	H
`book2`	Z	R+H	Z	R+H	H	H	H	H
`geo`	Z	R+A	Z	R+A	N	A	Z	Z
`news`	Z	R+H	Z	R+H	A	A	H	A
`obj1`	Z	Z	Z	Z	N	A	A	H
`obj2`	Z	R+H	Z	R+H	A	A	Z	H
`paper1`	Z	Z	Z	R+H	N	A	N	H
`paper2`	Z	Z	Z	R+H	N	A	N	H
`pic`	Z	A	Z	A	A	A	R+H	Z
`progc`	Z	Z	Z	R+H	N	A	N	H
`progl`	Z	Z	Z	H	N	A	N	H
`progp`	Z	Z	Z	H	N	A	A	H
`trans`	Z	Z	Z	H	N	A	Z	H

Table 4.2: Statistical encoders used in OFF-LINE$_1$ for the Calgary Corpus: "N" is no encoding,"A" is arithmetic coding, "R" is run-length encoding, "H" is Huffman encoding, "Z" is Deflate encoding (GZIP)

- `text[]` stores whatever may be left of the original x at the end of the compression process.

In general, the number 255 is reserved as an escape code to indicate that a current datum overflows the standard space so that an additional byte is allocated for its storage. For example, if the length of a word is 297 we store $(255, 42)$ in two consecutive positions of the array `sublen`. The value 255 is represented as $(255, 0)$.

As one should expect, the bulk of the output is represented by the lower byte of the relative positions `relpol` and by the array `text`. The experiments shows that although the former is practically incompressible, in principle `text` can be compressed again with other methods.

The choices for the encoding for each array are summarized in Table 4.3 and 4.2. Table 4.2 refers to the Calgary corpus, a standard benchmark for textual compression techniques. Table 4.3 is concerned with the fourteen chromosomes of the yeast *S. cerevisiae* and the *mitochondrial* DNA. The experimental results for these datasets will be discussed in Section 4.9.

As already noted, some arrays exhibit a high entropy (e.g., those containing the absolute and relative positions) and are usually encoded as plain numbers (N) or with arithmetic coding (A) [171] or Huffman (H). Others show long runs of equal numbers (for example, those storing substring length and the number of occurrences) and are significantly compressed by a cascade of run-length and Huffman encoding (R+H).

File	text	sublen	substr	occurr	abspol	abspoh	relpol	relpoh
chrI	A	A	H	H	A	A	A	A
chrII	A	Z	A	H	A	H	A	A
chrIII	A	Z	H	Z	A	A	A	A
chrIV	A	Z	A	R+H	A	A	A	A
chrV	A	Z	A	H	A	A	A	A
chrVI	A	H	A	A	A	A	A	A
chrVII	A	Z	A	R+H	A	A	A	A
chrVIII	A	Z	H	H	A	A	A	A
chrIX	A	A	A	A	A	A	A	A
chrX	A	Z	A	H	A	A	A	A
chrXI	A	A	H	A	A	A	A	A
chrXII	A	Z	A	R+H	A	A	A	A
chrXIII	A	Z	A	R+H	A	A	A	A
chrXIV	A	A	H	R+A	A	Z	A	A
chrXV	A	Z	A	H	A	A	A	A
chrXVI	A	Z	A	R+H	A	A	A	A

Table 4.3: Statistical encoders used in OFF-LINE$_1$ for the DNA dataset: "A" is arithmetic coding, "R" is run-length encoding, "H" is Huffman encoding, "Z" is Deflate encoding (GZIP)

4.5.2 OFF-LINE$_2$

An example of output produced by the second scheme is shown in Figure 4.7. The first iteration results in the choice of aba. One reference copy of aba remains in the text. However in the second iteration, the word ba is not substituted because the size of the pointer is more expensive than the plain text representation (i.e., $G_2(\texttt{ba}) < 0$).

```
1: abaababaabaababaababa$       Substituted substring: "aba"
-----------------------------   Final encoding:
text = [baabababa$]
text_bit_vector = [0 0 1 1 1 1 1 0 1 1 0 1 1 1]
text_low = [8 5 -3 -8]          text_high = [0 0 0 0]
sublen = [3 3 3 3]
```

Figure 4.7: A run of OFF-LINE$_2$ on the string abaababaabaababaababa$

During the iterative process the text is composed of a sequence of literals interspersed by pointers that can be distinguished by a bit-vector. At the end of the process the text appears as decomposed in two parts. The leftover of literals in the text forms the array text. The pointers are saved in the arrays text_low, text_high and sublen. Specifically, text_low stores the lower byte of the displacement, text_high the higher byte of the displacement, and sublen the length of the target of the pointer.

Therefore, Off-Line$_2$ produces five arrays that contains all the information we need in order to retrieve the original text. In particular, the original linear ordering of pointers and literals is preserved during the decomposition so we can reverse the process using the `text_bit_vector`.

Figure 4.7 shows the contents of the arrays after the decomposition. A "0" in the bit-vector indicates a pointer, a "1" indicates a literal. In `text_low` we can read the displacements of the four pointers that substituted the substring aba.

The specific statistical encoders chosen for the two datasets are summarized in Tables 4.4 and 4.5. Additional experimental results are reported in Section 4.9.

File	`text`	`text_bit_vector`	`text_low`	`text_high`	`sublen`
`bib`	Z	A	Z	Z	H
`book1`	Z	A	Z	Z	Z
`book2`	Z	A	Z	Z	H
`geo`	Z	A	Z	Z	Z
`news`	Z	A	Z	Z	Z
`obj1`	Z	Z	Z	Z	Z
`obj2`	Z	Z	Z	Z	Z
`paper1`	Z	A	A	Z	H
`paper2`	Z	A	A	A	H
`pic`	Z	R+Z	Z	Z	Z
`progc`	Z	A	Z	Z	H
`progl`	Z	A	Z	Z	Z
`progp`	Z	A	Z	Z	Z
`trans`	Z	A	Z	Z	Z

Table 4.4: Statistical encoders used in Off-Line$_2$ for the Calgary Corpus: "A" is arithmetic coding, "R" is run-length encoding, "H" is Huffman encoding, "Z" is Deflate encoding (Gzip)

4.5.3 Off-Line$_3$

An example of the output produced by the third scheme is shown in Figure 4.8. This time, the string abaab is selected. This substring becomes the first word in the dictionary (indexed by "0"), and all the occurrences of abaab are replaced by the pointer (0). After the substitution of abaab, all the substrings in the text result in $G_3(w) < 0$, so the process is terminated. The dictionary is augmented by the words a with index "1", b with index "2", and \$ with index "3" as explained in detail later.

A critical issue in this scheme is the size of the pointer used in the text. Unfortunately, only at the end of the substitution process we know the exact value of d (see the definition of G_3 in Section 3.2.3). We solve the problem as follows. We assign some tentative size to the pointers during the substitution process, say, by allocating two bytes for each one. Once the iterative process

File	text	text_bit_vector	text_low	text_high	sublen
chrI	A	Z	A	A	H
chrII	A	R+Z	A	A	H
chrIII	A	Z	A	A	H
chrIV	A	Z	A	A	H
chrV	A	R+Z	A	A	H
chrVI	A	Z	A	A	H
chrVII	A	Z	A	A	H
chrVIII	A	R+Z	A	A	H
chrIX	A	Z	A	A	H
chrX	A	R+Z	A	A	H
chrXI	A	R+Z	A	A	H
chrXII	A	R+Z	A	A	H
chrXIII	A	Z	N	A	H
chrXIV	A	R+Z	A	A	H
chrXV	A	Z	N	A	H
chrXVI	A	Z	A	A	H

Table 4.5: Statistical encoders used in OFF-LINE$_2$ for the DNA dataset: "N" is no encoding, "A" is arithmetic coding, "R" is run-length encoding, "H" is Huffman encoding, "Z" is Deflate encoding (GZIP)

is finished we decompose the pointers in lower and higher byte in the arrays `text_low` and `text_high` respectively.

At this point, the final encoding should have a bit-vector to distinguish literal from pointers in `text`. To get rid of it, we add to the dictionary the words composed by each symbol that appears in the leftover. Then we are entitled to substitute all the symbols with pointers. As a result the encoding is composed only by a sequence of pointers (see Figure 4.8). It means that the original string can be expressed by means of a new alphabet of cardinality d, the symbols of which are the words in the dictionary.

Note that the "name" of the pointers, that is, the index associated with

```
1: abaababaabaababaababa$        Substituted substring: "abaab"
-------------------------------
Appending to dictionary "a" - 1
Appending to dictionary "b" - 2
Appending to dictionary "$" - 3
-------------------------------  Final encoding:
text_low  = [0 0 1 1 2 0 1 2 1 3]
text_high = [0 0 0 0 0 0 0 0 0 0]
dict      = [abaabab$]
sublen    = [5 1 1 1]
```

Figure 4.8: A run of OFF-LINE$_3$ on the string abaababaabaababaababa$

File	`text_l`	`text_h`	`substr`	`sublen`
`bib`	Z	A	Z	Z
`book1`	Z	Z	Z	Z
`book2`	Z	Z	Z	Z
`geo`	Z	Z	Z	R+A
`news`	Z	Z	Z	Z
`obj1`	Z	Z	Z	Z
`obj2`	Z	Z	Z	Z
`paper1`	Z	Z	Z	Z
`paper2`	Z	A	Z	Z
`pic`	Z	R+Z	Z	Z
`progc`	Z	Z	Z	Z
`progl`	Z	Z	Z	Z
`progp`	Z	R+Z	Z	Z
`trans`	Z	Z	Z	R+H

Table 4.6: Statistical encoders used in OFF-LINE$_3$ for the Calgary Corpus: "A" is arithmetic coding, "R" is run-length encoding, "H" is Huffman encoding, "Z" is Deflate encoding (GZIP)

each word in the dictionary, is arbitrary. We can exploit this observation by reshuffling the names of the pointers in such a way that the words in the dictionary are sorted in order of increasing length, so that smaller indices correspond to shorter words. Then the array `sublen` contains a sequence of monotonically increasing numbers which can be highly compressed by a statistical encoder if represented in terms of consecutives increments.

The statistical encoders used for each array are summarized in Table 4.6 and 4.7. In the case of Calgary Corpus, most of the arrays retain some redundancy that is conveniently compressed by a LZ-77 encoding. Final results are discussed in Section 4.9.

4.6 Decoding

Decoding a string given in the OFF-LINE$_2$ and OFF-LINE$_3$ compressed representation can be done in linear time, because these schemes forbid pointer recursion. In particular, OFF-LINE$_3$ has an on-line, linear time decoding algorithm.

For Scheme 2 we first reconstruct the original sequence of literals and pointers using the bit-vector. We scan the sequence left to right. A literal is copied in the decompressed representation. A pointer suggests to copy the corresponding string. Since pointers are bidirectional then the decoding has to be off-line.

Scheme 3 is even easier to decode because the final compressed string is composed only by pointers. We examine the sequence of pointers left to right. For each pointer we copy the corresponding word from the dictionary in the decompressed string. The algorithm takes linear time in the size of the output

File	text_l	text_h	substr	sublen
chrI	A	R+Z	H	H
chrII	A	R+Z	A	H
chrIII	A	R+Z	H	H
chrIV	A	R+Z	A	Z
chrV	A	R+Z	A	H
chrVI	A	R+Z	H	A
chrVII	A	R+Z	A	H
chrVIII	A	R+Z	H	H
chrIX	A	R+Z	H	A
chrX	A	R+Z	A	H
chrXI	A	R+Z	H	H
chrXII	A	R+Z	A	Z
chrXIII	A	R+Z	A	H
chrXIV	A	R+Z	H	H
chrXV	A	R+Z	A	H
chrXVI	A	R+Z	A	H

Table 4.7: Statistical encoders used in OFF-LINE$_3$ for the DNA dataset: "A" is arithmetic coding, "R" is run-length encoding, "H" is Huffman encoding, "Z" is Deflate encoding (GZIP)

and it is on-line.

Scheme 1 is slightly more complicated. We allocate an empty array of size n and fill the array starting from the first string in the dictionary. Once the first word has been stored in the appropriate positions, we are left with a number of empty fragments that should be filled by the next words. The problem is that the positions of the occurrences of the i-th word in the dictionary are expressed with respect to the "white space" in the array after the insertion of the first $i-1$ words. For example, it is entirely possibile that the some occurrences of the i-th word could span multiple fragments. Even if we were keeping the fragments in a linked list, the time complexity of one iteration the decoding would be more than linear. We should resort to a data structure called finger tree (see e.g., [91, 29, 28]). A finger search tree is a data structure which stores a sorted list of elements in such a way that searches are fast in the vicinity of a *finger*, where a finger is a pointer to an arbitrary element of the list. If we organize the empty fragments in a finger tree, then the decoding of one iteration takes linear time, the reason being that the algorithm visits a linear number of nodes in the finger tree.

4.7 Heuristics

The most time-consuming activity of the compression phase is the construction of the augmented suffix tree and its annotation with the values of the gain. We employed three heuristics to overcome the high computational demands of the original "full-fledged" version of the compressor [12].

Instead of selecting only one word from a freshly built suffix tree and then quickly dispose of the tree, we could extract several candidate substitutions. In the latter case, we could utilize more than one substring between any two consecutive updates of the statistical index.

Another way to cut the computation time is to build a pruned version of the tree instead of a complete version. In principle, we could miss some advantageous substitution of a very long repeated string, but we could use much less storage space and time.

Finally, we could try to use a standard suffix tree instead of an augmented tree to store the statistics of the subwords.

In the following Subsections, we will evaluate the effect on the compression efficiency of each strategy separately on two files, using OFF-LINE$_1$. The two test files are `paper2` from the Calgary Corpus and the `mito`chondrial DNA of the yeast *S. cerevisiae*. These two texts have been selected because of their radically different nature. The file `paper2` contains English text, it is structured and it has a large alphabet, while `mito` could be regarded as a pseudo-random sequence of five symbols (A,T,C,G,N).

The overall speed-up of the above heuristics combined together is impressive: our original implementation took several hours to compress each of those files while now it runs in the order of some minutes. What is even better, the corresponding loss of efficiency in terms of compression is almost negligibile.

4.7.1 Priority queue

To store multiple candidates for substitution we use a queue that can grow to a maximum size `queue` and that contains the words that result in the maximal gain. The priority is the value of the gain: at the front of the queue we find the most profitable substitutions.

The strings are retrieved one at time from the queue and used in the contraction step. It is entirely possible that at some point a string from the queue could not be found in the contracted text because of previous deletions. In fact, part of the words in the queue end up unused. In any case, as soon as all the words in the queue have been considered, a new augmented suffix tree is built on the contracted text.

To avoid filling up the queue with words that will not be used because of the effect of previous substitutions, we check whether the word we are inserting in the queue is a suffix or a prefix of some other word already in the queue. If it is not, we insert the word and possibly drop the word in position `queue`+1. If it is, we keep only the one which gives the best gain and discard the other.

The speed-up factor of this heuristic is proportional to the value of `queue` until the size of the queue becomes too large and the cost of searching for words which will not be substituted overcomes the benefit of building less trees (see Figure 4.8). The loss in compression is negligible for sizes in the range $5 \ldots 20$.

	paper2		mito	
queue	*size*	*time*$_{[\text{min}]}$	*size*	*time*$_{[\text{min}]}$
1	30,773	19.70	16,326	7.06
2	30,780	10.36	16,367	4.06
5	30,785	5.06	16,405	2.24
10	30,787	3.21	16,446	1.66
20	30,826	2.39	16,476	1.36
50	30,904	1.97	16,632	1.28
100	30,923	1.86	16,702	1.37
1,000	30,923	1.98	16,702	1.47

Table 4.8: Comparing the performance of OFF-LINE$_1$. We fixed `min_occ` $= 2$, `min_length` $= 2$, `max_length` $= 100$. The tree is augmented

We usually employ `queue` $= 10$, but a careful study of the optimal value has not been carried out.

To summarize, the overall result is a considerable speed up with respect to the eager version without a substantial penalty in the compression performance.

4.7.2 Pruned tree

The observation that it is highly unlikely that very long words occur frequently in a text suggests that building the statistics for *all* the substrings can be a waste of resources. Pruning the tree speeds up considerably the implementation and saves large amounts of memory.

Pruning the tree does not mean that we could completely miss the word involved in a long substitution. If the current best substitution is a word w longer than the threshold `max_length` then the encoder will eventually choose some substring of w of length `max_length` because that substring occurs without overlap at least as many times as w.

	paper2		mito	
max_length	*size*	*time*$_{[\text{min}]}$	*size*	*time*$_{[\text{min}]}$
10	30,986	2.58	17,044	0.29
50	30,664	2.62	16,491	1.32
100	30,636	2.68	16,470	1.38
∞	30,636	19.39	16,470	10.34

Table 4.9: Comparing the performance of OFF-LINE$_1$. We fixed `min_occ` $= 4$, `min_length` $= 4$, `queue` $= 10$. The tree is augmented

Table 4.9 shows that the pruned version of OFF-LINE$_1$ at `max_length` $= 100$ performs almost ten time faster and achieves exactly the same compression as the version that builds the complete tree. Again, the speed up is an order of magnitude without any penalty in the efficiency.

4.7.3 Suffix tree

The knowledge of the statistics of all the words in the text is a fundamental prerequisite of our algorithm. In Section 4.3 we describe how to use the augmented suffix tree, a data structure that compactly embodies all the words in a text with their non-overlapping frequencies.

A "standard" suffix tree (see Section 4.2) gives the number of overlapped occurrences of w that is not what we really need. In principle, using the incorrect statistics could degrade the performance of the encoder.

However, the standard suffix tree can be built in linear time (see [165, 112, 160]), while the best known result for the augmented suffix tree is $O(n \log^2 n)$ [13].

type of tree	`min_length`	`min_occ`	`paper2`	`mito`
augmented	2	2	30798	16445
augmented	2	4	30661	16487
augmented	4	2	30798	16445
augmented	4	4	30660	16487
augmented	10	10	29913	16659
standard	2	2	30837	16521
standard	2	4	30680	16462
standard	4	2	30833	16521
standard	4	4	30680	16462
standard	10	10	29996	16826
standard - augmented	2	2	39	76
standard - augmented	2	4	19	-25
standard - augmented	4	2	35	76
standard - augmented	4	4	20	-25
standard - augmented	10	10	83	167

Table 4.10: Comparing the performance of OFF-LINE$_1$ between standard suffix tree and augmented suffix tree. We fixed `max_length` $= 40$, `queue` $= 10$

Table 4.10 shows the performance of OFF-LINE$_1$ using augmented and standard trees and the difference in the final sizes. The maximum loss in performance is around 1%. Surprisingly, there are two cases where the overlapping statistics gives a better result.

4.8 Software

The whole project is written in C++ and is based on the Standard Template Library (STL). STL is a clean template library of containers and generic functions endowing C++ with some features of higher-order imperative languages [119]. We mainly used containers for dynamic arrays of characters and boolean, priority queues, and red-black trees.

The three versions of the program for the general schemes described in Sections 3.2.1, 3.2.2 and 3.2.3 can be obtained through conditional compilation. The general structure of OFF-LINE is shown in the pseudo-code of Figure 4.9.

```
$i \leftarrow 0$;
$x \leftarrow$ text;
**while** (forever) {
      **if** (queue $Q$ is empty) {
            **if** (no substitution has been performed)
                  **break;**
            $T \leftarrow$ **create_min_augm_suffix_tree**($x$);
            $Q \leftarrow$ **compute_gain**($T$);
      } **else**
      $(w, occ) \leftarrow Q$.**pop**();
#     **ifdef** OFF-LINE$_1$
      **if** (($G_1(w) > 0$) **and** ($occ >$ **min_occ**)) {
            save $w$ in the dictionary;
            save the positions of the occurrences;
            $x \leftarrow$ delete all the occurrences of $w$ from $x$;
      }
#     **endif**
#     **ifdef** OFF-LINE$_2$
      **if** (($G_2(w) > 0$) **and** ($occ >$ **min_occ**))
            $x \leftarrow$ substitute all the occurrences of $w$, except one, with
                  pointers to the reference copy;
#     **endif**
#     **ifdef** OFF-LINE$_3$
      **if** (($G_3(w) > 0$) **and** ($occ >$ **min_occ**)) {
            append $w$ in the dictionary at position $i$;
            $x \leftarrow$ substitute all the occurrences of $w$ with the pointer $i$;
            $i \leftarrow i + 1$;
      }
#     **endif**
};
run a statistical compressor on the encoding;
```

Figure 4.9: The top level structure of OFF-LINE

The main data structures are `class symbol`, `class Text`, and `class Tree`. The class `symbol` defines the structure and the methods to handle a symbol from the alphabet. It is composed by a `char` which stores the symbol itself, and a `bool` that indicates if the symbol should be interpreted as a literal or as a pointer.

`Text` is the class the instances of which store and process the text. It is composed by the fragment tree, a red-black tree of nodes representing fragments of text created during the substitution process and by a vector of `symbols` (the text itself). The most frequent operations are `erase(vector<symbol> pattern)`, that searches and deletes all the non-overlapping occurrences of some given pattern and `substitute(vector<symbol> old, vector<symbol> new)` that searches and substitutes all the non-overlapping occurrences of the old pattern with the new one.

The suffix tree is built from the `Text` once it has been compacted to a single contiguous array. The tree is stored in an instance of the class `Tree`. The definition of the class `Tree` is shown in Figure 4.10. There are two pointers to the beginning and to the end of the string represented by that node, two pointers to siblings and children and the counter of non-overlapping occurrences. The prominent methods defined in the class `Text` are the function that builds the suffix tree and the procedure that annotates the tree with the gain values.

```
class Tree {
   vector<symbol>::iterator
     begin,    // pointer to the starting position
     end;      // pointer to the ending position
   Tree
     *sibling, // sibling pointer
     *child;   // child pointer
   int
     counter;  // number of non-overl occurrences
   ...
}
```

Figure 4.10: The definition of the class `Tree`

4.9 Results

The encoders described in the previous Sections have been subject to extensive experimentation and tuning. The files on which we performed most of the experiments are `paper2` from the Calgary Corpus and `mito`, the mitochondrial DNA sequence of the yeast. Table 4.11 compares directly the three encoders on these two files.

The running times are in the order of 2-3 minutes for files of the size of 80 KB on a 300 Mhz machine running Solaris. When comparing compression, the best encoder is Off-Line_3, followed by Off-Line_1 and, at some distance, Off-Line_2.

Table 4.12 displays the same ranking among the three encoders. The data set is the Calgary Corpus, a standard benchmark for testing lossless compression programs. Off-Line_3 outperforms the other two encoders on most of the files.

	paper2 (82,199)		mito (78,521)	
encoder	*size*	*time*$_{[\text{min}]}$	*size*	*time*$_{[\text{min}]}$
OFF-LINE$_1$	30,848	3.21	16,426	1.66
OFF-LINE$_2$	33,757	3.01	17,741	2.24
OFF-LINE$_3$	**30,219**	2.38	**16,086**	2.38

Table 4.11: Comparing the performance of OFF-LINE encoders. We fixed `min_occ` $= 2$, `min_length` $= 2$, `max_length` $= 100$, `queue` $= 10$ and the tree is augmented. Running times are for a 300Mhz Solaris machine.

When compared with other textual substitution programs, the OFF-LINE family performs better on most of input files. In all the other cases, GZIP is the winner: however, the results of OFF-LINE are still comparable with GZIP. Moreover, a faithful comparison to GZIP is made difficult by the many heuristics employed in the latter.

File	*Size* (bytes)	Huffman PACK	LZ-78 COMPRESS	LZ-77 GZIP	OFF-LINE$_1$	OFF-LINE$_2$	OFF-LINE$_3$
`bib`	111,261	72,868	46,528	35,063	36,145	39,226	**34,442**
`book1`	768,771	438,487	332,056	313,376	305,185	323,007	**298,735**
`book2`	610,856	368,423	250,759	206,687	**203,249**	216,494	204,703
`geo`	102,400	72,836	77,777	68,493	**68,229**	69,983	68,726
`news`	377,109	246,516	182,121	144,840	**141,257**	150,462	143,246
`obj1`	21,504	16,330	14,048	**10,323**	10,845	11,271	11,088
`obj2`	246,814	194,378	128,659	**81,631**	88,179	93,915	87,574
`paper1`	53,161	33,457	25,077	**18,577**	19,994	21,607	19,289
`paper2`	82,199	47,731	36,161	**29,753**	30,848	33,757	30,219
`pic`	513,216	106,737	62,215	56,442	52,036	55,427	**50,885**
`progc`	39,611	26,030	19,143	**13,275**	14,758	15,527	14,127
`progl`	71,646	43,093	27,148	16,273	18,508	18,919	**16,153**
`progp`	49,379	30,328	19,209	11,246	12,890	13,282	**11,160**
`trans`	93,695	65,343	38,240	**18,985**	21,170	21,170	19,662

Table 4.12: Comparing OFF-LINE with other compression programs via textual substitution on the Calgary Corpus

If we cross the boundary of textual substitution methods, the block-sorting technique BZIP and BZIP2 based on [30] outperform GZIP and OFF-LINE on the whole Calgary Corpus (see Table 4.13). A different scenario is shown by with other datasets, as explained in the following Section.

4.9.1 DNA sequences

A class of data where we extensively tested our encoders are DNA sequences. The deoxyribonucleic acid (DNA) constitutes the physical medium in which all properties of living organisms are encoded. The knowledge of its sequence is fundamental in molecular biology. Important molecular biology databases (e.g., EMBL, Genbank, DDJB, Entrez, SwissProt, etc.) have been developed

File	*Size* (bytes)	BWT BZIP	BWT BZIP2	OFF-LINE$_1$	OFF-LINE$_2$	OFF-LINE$_3$
`bib`	111,261	**27,097**	27,467	36,145	39,226	34,442
`book1`	768,771	**230,247**	232,598	305,185	323,007	298,735
`book2`	610,856	**155,944**	157,443	203,249	216,494	204,703
`geo`	102,400	57,358	**56,921**	68,229	69,983	68,726
`news`	377,109	**118,112**	118,600	141,257	150,462	143,246
`obj1`	21,504	**10,409**	10,787	10,845	11,271	11,088
`obj2`	246,814	**76,017**	76,441	88,179	93,915	87,574
`paper1`	53,161	**16,360**	16,558	19,994	21,607	19,289
`paper2`	82,199	**24,826**	25,041	30,848	33,757	30,219
`pic`	513,216	**49,422**	49,759	52,036	55,427	50,885
`progc`	39,611	**12,379**	12,544	14,758	15,527	14,127
`progl`	71,646	**15,387**	15,579	18,508	18,919	16,153
`progp`	49,379	**10,533**	10,710	12,890	13,282	11,160
`trans`	93,695	**17,561**	17,899	21,170	21,170	19,662

Table 4.13: Comparing OFF-LINE with context-sorting encoders on the Calgary Corpus

to collect hundreds of thousand of sequences of nucleotides and amino-acids from biological laboratories all over the world.

The size of these databases, that is currently in the order of thousands of gigabytes, increases exponentially fast. Unluckily, DNA files have shown to be difficult to compress with Lempel-Ziv schemes because of their unstructured nature. The compression of genetic sequences constitutes, therefore, a very challenging task [73].

In other contexts, the amount of compression achievable on genetic sequences has been used as a possible measure of biological significance [54, 129, 130] or as a classifier [103, 101, 102].

Due to mutations, errors in the sequencing, and other biological events, the redundancy in the DNA should be modeled mainly as consecutive (*tandem*) repeats of the same word (*motif*) and palindromes. However, tandem repeats and palindromes are not exact but they can occur with substitutions, insertions or deletions of symbols. Moreover, palindromes are complemented, that is, the word is reversed and the base `A` is substituted with `T` (and vice-versa), while `C` is substituted with `G` (and vice-versa).

A corpus of novel approaches for compressing DNA has been established in the past few years. The first attempt by Grumbach and Tahi [72, 73], called BIOCOMPRESS2, extends LZ-77 to catch very distant repeats and complementary palindromes.

Loewenstern and Yianilos [101] attack the problem of computing good estimates of the entropy of DNA sequences by building a predictive model PPM-style [34, 33, 116]. With respect to the original PPM, they extend the context model by allowing mismatches. Their algorithm estimates the parameters of the model, called CDNA, via a learning process that tries to optimize a complex objective function. The general problem is known to be $\mathcal{NP}$-complete, but they devise some approximation schemes.

Allison, Edgoose and Dix propose the most time expensive approach to DNA compression [3]. They search for approximate repeats and approximate palindromes. Their primary purpose is not to compress the text, but rather to model the statistical property of the data as accurately as possible and to find patterns and structures within them. They build a model with parameters such as the probability of repeats, the probability of the length of repeats, and the probability of mismatches within repeats. The parameters of the model are estimated by an expectation maximization algorithm that takes $O(n^2)$ at each iteration. Their results are the current "state of the art", but the algorithm is extremely slow (the computation is in the order of days).

We compare the performance of OFF-LINE encoders with standard compression programs in Table 4.14. The encoder OFF-LINE$_3$ outperforms each and every general purpose encoder on the fourteen chromosomes and the mitochondrial DNA of the yeast (*Saccharomyces Cerevisiae strain S288*).

Even comparing our encoders with programs designed to compress only DNA, our performance is not very distant (see Table 4.15). We believe that a specialized version of OFF-LINE augmented with some "biological knowledge" could come very close to the best DNA compressors.

File	*Size* (bytes)	Huffman PACK	LZ-78 COMPRESS	LZ-77 GZIP	BWT BZIP	BWT BZIP2	OFF-LINE$_1$	OFF-LINE$_2$	OFF-LINE$_3$
`chrI`	230,195	63,144	62,935	66,264	61,674	62,373	57,098	58,631	**56,915**
`chrII`	813,137	222,597	219,845	236,837	218,463	221,032	201,617	203,456	**201,180**
`chrIII`	315,344	86,281	86,009	91,827	84,809	85,705	77,916	78,983	**77,764**
`chrIV`	1,522,191	416,516	409,957	440,056	407,799	411,250	371,230	374,413	**370,796**
`chrV`	574,860	157,415	155,944	167,749	154,580	155,731	142,364	143,775	**141,919**
`chrVI`	270,148	74,077	73,873	78,925	72,838	73,651	67,451	68,151	**67,391**
`chrVII`	1,090,936	298,680	294,417	317,282	293,079	296,245	270,051	272,972	**269,265**
`chrVIII`	562,638	154,110	152,265	163,135	151,240	152,992	139,588	140,924	**139,271**
`chrIX`	439,885	120,669	118,965	127,805	118,182	119,553	109,507	110,871	**109,303**
`chrX`	745,443	204,152	201,783	216,148	200,325	202,223	184,709	186,471	**184,287**
`chrXI`	666,448	182,377	180,100	194,119	179,306	180,901	165,780	166,752	**165,478**
`chrXII`	1,078,171	295,441	291,754	305,653	288,112	290,800	260,172	261,346	**259,898**
`chrXIII`	924,430	253,176	249,099	267,127	248,450	250,735	228,233	231,474	**227,610**
`chrXIV`	784,328	215,020	212,219	228,757	210,988	212,816	195,291	196,719	**194,947**
`chrXV`	1,091,282	298,762	294,921	317,971	293,838	297,279	270,626	273,366	**269,921**
`chrXVI`	948,061	286,579	264,113	278,651	254,947	257,590	234,099	237,365	**233,150**
`mito`	78,521	18,149	17,890	19,369	17,962	18,094	16,426	17,741	**16,086**

Table 4.14: Comparing OFF-LINE with other compression programs on the chromosomes of the yeast

Encoder	*Size*	*bpc*
GZip	91,827	2.33
Pack	86,281	2.19
Compress	86,009	2.18
BZip2	85,705	2.17
BZip	84,809	2.15
Off-Line$_3$	77,764	1.97
CDNA [101]	76,471	1.94
Biocompress2 [73]	75,682	1.92
AED [3]	75,407	1.913

Table 4.15: Comparing Off-Line with DNA-specific compression programs on third chromosome (`chrIII`) of the yeast (315,344 bps). bpc is the average number of bits per character in the compressed representation

Chapter

5

Conclusions

In this Thesis, we showed that off-line textual substitution can be a feasible approach to lossless data compression when the user can afford to spend time and memory for the encoding while the decoding should be still fast. Situations such as distribution of data on CD-ROMs could take advantage of the higher efficiency of compression shown by our programs.

The results of our experiments are encouraging. OFF-LINE encoders outperformed all other general purpose compression programs on DNA benchmarks, and textual substitution programs on the Calgary Corpus in most of the cases. A specialized version of OFF-LINE for compressing genetic sequences is expected soon.

When compared with the context-sorting encoders on general files, the performance of OFF-LINE seems to fade. We believe that the gap will be closed when we exploit more the "context". The files of the Calgary Corpus contains natural language, Pascal code, Lisp code, object code, etc. In these files there is always a strong correlation between adjacent symbols, that is the probability of occurrence of a symbols is strongly tied on the preceding/following context. OFF-LINE removes repetitions in a new and clever way, but it does not remove redundancy hidden in the correlation of a symbol with its context. More work should be done in this direction.

Many interesting questions have been answered by the discussions and experiments reported in this Thesis. Some of these answers have raised new and more challenging questions which would warrant additional effort.

These include more precise definitions of gain functions G or alternative approaches to the minimization problem, a clever procedure for the selection of the string to store in the queue, the usefulness of reiteration of treatment following the first application of OFF-LINE, and several issues pertaining to the computational efficiency achievable by sequential and parallel implementations. Among the latter, a prominent concern would be to devise efficient algorithms that avoid building the statistical index from scratch, and better storage and matching algorithms for the data structures.

We conclude by describing some of the research directions that we are planning to pursue in the near future.

5.1 Bounds

In order to be entitled to call our steepest descent strategy an *approximation scheme* of the optimal encoding we should prove formally bounds on the approximation. Specifically, we would like to prove that

$$\frac{|\hat{x}_{OPT}|}{|\hat{x}_{\textsc{Off-Line}}|} < C$$

for some constant C.

This line of research could converge towards some of the recent works by De Agostino, Storer, and Silvestri [46, 44].

5.2 Parallel implementation

Several papers studied parallel algorithms for data compression via textual substitution on the PRAM (see e.g., [153, 45, 42, 70, 43, 147, 39, 120]).

In a parallel architecture with a sufficient number of processor, we could imagine to assign to each processor a textual substitution. Tables 5.1 and 5.2 show the modest number of iterations of the main loop of Off-Line encoders. They correspond to the number of substitutions in the text.

The problem of performing the substitutions in parallel is that the each substitution is tied to the history of the contraction steps[1], so we cannot precompute all the substitutions from the tree that was built on the original string.

However, we have shown that using a careful selection of the words from the tree we can get a significant speed up without substantial penalty. Moreover, the suffix tree can be built in parallel very efficiently [95, 11, 76]. Therefore, we believe that a parallel implementation of Off-Line might result in relatively clean and very fast real-time application.

5.3 Grammatical inference

Grammatical inference is an inductive inference problem where the target domain is a formal language and the representation class is a family of grammars. The learning task is to identify a “correct” grammar for the (unknown) target language, given a finite number of examples of the language. This has been investigated within many research fields, including machine learning, computational learning theory, pattern recognition, computational linguistics, neural networks, formal language theory, information theory, and many others. There are several surveys on the field of grammatical inference [63, 64, 5, 114, 137].

[1]Recall that G changes after each substitution

File	*Size*	OFF-LINE$_1$	OFF-LINE$_2$	OFF-LINE$_3$
`bib`	111,261	504	634	465
`book1`	768,771	2997	2857	2990
`book2`	610,856	2305	2408	2378
`geo`	102,400	407	473	503
`news`	377,109	1789	1634	1619
`obj1`	21,504	125	111	337
`obj2`	246,814	1219	1207	1055
`paper1`	53,161	373	475	342
`paper2`	82,199	506	717	505
`pic`	513,216	94	125	222
`progc`	39,611	255	261	308
`progl`	71,646	312	267	273
`progp`	49,379	208	210	252
`trans`	93,695	340	253	318

Table 5.1: Iterations of the main loop of OFF-LINE for the Calgary corpus files

Recently, grammatical inference has been applied to computational biology [93, 143, 138, 139].

It is interesting to examine the behavior of OFF-LINE$_1$ when used as a tool for inferring hierarchical grammar in sequences. The rationale to build grammar based on some measure of compression can be justified by the "Occam's Razor" paradigm. *When more than one explanation is possible, choose the simplest.* The grammar that our encoder is looking for is the shortest "explanation" of the original string in terms of information content.

Another algorithm called SEQUITUR developed by Nevill-Manning *et al.* [125, 123, 126, 124], is based on the inference of grammar to achieve compression. SEQUITUR implements an on-line algorithm capable to infer a hierarchical grammar from the text, looking for the longest repeated substring as in Lempel-Ziv schemes. Except for the starting production, the inferred grammar is constrained to have the right hand sides of the productions composed of digrams.

As a preliminary experiment we analyzed the first paragraph of "Alice's Adventures in Wonderland" by Lewis Carroll, which is part of the new Canterbury Corpus (see Figure 5.1).

```
Alice was beginning to get very tired of sitting by her sister
on the bank, and of having nothing to do:  once or twice she had
peeped into the book her sister was reading, but it had no
pictures or conversations in it, 'and what is the use of a book,'
thought Alice 'without pictures or conversation?'
```

Figure 5.1: The text analyzed

Figure 5.2 shows the grammar generated by OFF-LINE$_1$, while in Figure 5.3 the grammar is created by SEQUITUR. Note that OFF-LINE$_1$ generates a

File	*Size*	OFF-LINE$_1$	OFF-LINE$_2$	OFF-LINE$_3$
`chrI`	230,195	78	603	80
`chrII`	813,137	112	474	128
`chrIII`	315,344	61	309	68
`chrIV`	1,522,191	383	1297	441
`chrV`	574,860	109	276	118
`chrVI`	270,148	22	226	30
`chrVII`	1,090,936	144	1009	162
`chrVIII`	562,638	91	264	102
`chrIX`	439,885	54	543	63
`chrX`	745,443	108	376	123
`chrXI`	666,448	49	302	58
`chrXII`	1,078,171	444	1443	499
`chrXIII`	924,430	187	706	212
`chrXIV`	784,328	24	441	72
`chrXV`	1,091,282	128	924	147
`chrXVI`	948,061	193	755	217

Table 5.2: Iterations of the main loop of OFF-LINE for the chromosomes of the yeast

more compact representation than SEQUITUR. However, because of the iterated substitutions it is entirely possible that the target of some production could overlap.

```
Al<LL><II>beg<D<<NN>E>>get_very_t<B>d<KK>sitt<NN>by<OOOOOOOOOO>on<MMM>bank,
<A><KK>hav<NN>noth<NN>to_do:__once_or_tw<LL>sh<<JJJ>F><F>d_<<E><MMM><HH>
<OOOOOOOOOO><II>B>ading,_bu<C>t<JJJ>no<PPPPPPPPPPPPPPPPP>s_<D>it,
_‘<A>_wha<C>s<MMM>use<KK>a_<HH>,’<GG>ght_Al<LL>‘wi<GG>t<PPPPPPPPPPPPPPPPP>?’

<A>   --> "and"
<B>   --> "ire"
<C>   --> "t_i"
<D>   --> "in_"
<E>   --> "nto"
<F>   --> "epe"
<GG>  --> "thou"
<HH>  --> "book"
<II>  --> "was_"
<JJJ> --> "_had_"
<KK>  --> "_of_"
<LL>  --> "ice_"
<MMM> --> "_the_"
<NN>  --> "ing_"
<OOOOOOOOOO> --> "_her_sister_"
<PPPPPPPPPPPPPPPPPPP> --> "_pictures_or_conversation"
```

Figure 5.2: The inferred grammar produced by OFF-LINE$_1$ (blanks are substituted by _)

```
A100 --> A136 A116 A123 A104  "eg" A101 "n" A103 A102 A113 "ge" A126
         A129 "y" A102 "i" A124  A110 A106 "tt"  A103 A104 "y"  A122
         A114 A102  A117 "b" A109 "k" A125  A109 A110 A111 "av" A112
         A128 A121 A112 A120 "do:_" A114 A115 "or" A102 "w" A116 "s"
         A117 A127 A118  "e" A118 A119  A101 A120 A134 "b" A135 A122
         "_" A123  "_" A124 "ad" A103 A125  "bu" A132 A126 A127 A128
         "_" A140 A133 A101  "_" A138  A125  "‘" A109 A119  "w" A131
         A132 A133  A134 "us" A108 "of_a"  A104 A135  ",’" A102 A139
         "gh" A126 A136 A137 A108 "‘w" A138 A139 A126 A140 "?’"

A101 --> "in"                   A102 --> "_t"
A103 --> A101 "g"               A104 --> "_b"
A105 --> "er"                   A106 --> "_si"
A107 --> "_o"                   A108 --> "e_"
A109 --> "an"                   A110 --> d A107 f
A111 --> "_h"                   A112 --> A103 "_"
A113 --> "o_"                   A114 --> A107 "n"
A115 --> "c" A108               A116 --> "i" A115
A117 --> "h" A108               A118 --> "pe"
A119 --> "d_"                   A120 --> "t" A113
A121 --> "th"                   A122 --> A111 A105 A106 "st" A105
A123 --> "was"                  A124 --> "re"
A125 --> ",_"                   A126 --> "t_"
A127 --> A131 A119              A128 --> "no"
A129 --> "v" A105               A130 --> "on"
A131 --> "ha"                   A132 --> A126 i
A133 --> "s_"                   A134 --> A121 A108
A135 --> "ook"                  A136 --> "Al"
A137 --> "ic"                   A138 --> "it"
A139 --> "hou"
A140 --> "p" A137 "tu" A124 "s" A107 "r_c" A130 A129 "sati" A130
```

Figure 5.3: The inferred grammar produced by Sequitur (blanks are substituted by _)

Bibliography

[1] Aberg, J., and Shtarkov, Y. M. Text compression by context tree weighting. In *Data Compression Conference* (Snowbird, Utah, 1997), J. A. Storer and M. Cohn, Eds., IEEE Computer Society Press, TCC, pp. 337–386.

[2] Aberg, J., Shtarkov, Y. M., and Smeets, B. J. M. Non-uniform PPM and contex tree models. In *Data Compression Conference* (Snowbird, Utah, 1998), J. A. Storer and M. Cohn, Eds., IEEE Computer Society Press, TCC, pp. 279–288.

[3] Allison, L., Edgoose, T., and Dix, T. I. Compression of strings with approximate repeats. *Intell. Sys. in Mol. Biol. '98* (June 1998), 8–16.

[4] Andersson, A., and Nilsson, S. Efficient implementation of suffix trees. *Softw. Pract. Exp. 25*, 2 (Feb. 1995), 129–141.

[5] Angluin, D., and Smith, C. H. Inductive inference: Theory and methods. *ACM Computing Surveys 15*, 3 (Sept. 1983), 237–269.

[6] Apostolico, A. Linear pattern matching and problems of data compression. In *IEEE International Symposium on Information Theory* (1979).

[7] Apostolico, A. The myriad virtues of suffix trees. In *Combinatorial Algorithms on Words*, A. Apostolico and Z. Galil, Eds., vol. 12 of *NATO Advanced Science Institutes, Series F.* Springer-Verlag, Berlin, 1985, pp. 85–96.

[8] Apostolico, A., Bock, M. E., and Lonardi, S. Efficient detection of unusual words. Tech. Rep. TR-98, Purdue University, 1998.

[9] Apostolico, A., Bock, M. E., and Lonardi, S. Linear global detectors of redundant and rare substrings. In *to appear in the Proceedings of Data Compression Conference* (Snowbird, Utah, 1999), J. A. Storer and M. Cohn, Eds., IEEE Computer Society Press, TCC.

[10] Apostolico, A., and Fraenkel, A. Robust transmission of unbounded strings using Fibonacci representations. *IEEE Trans. Inf. Theory 33*, 2 (1987), 238–245.

[11] Apostolico, A., Iliopoulos, C., Landau, G. M., Schieber, B., and Vishkin, U. Parallel construction of a suffix tree with applications. *Algorithmica 3* (1988), 347–365.

[12] Apostolico, A., and Lonardi, S. Some theory and practice of greedy off-line textual substitution. In *Data Compression Conference* (Snowbird, Utah, 1998), J. A. Storer and M. Cohn, Eds., IEEE Computer Society Press, TCC, pp. 119–128.

[13] Apostolico, A., and Preparata, F. P. Data structures and algorithms for the strings statistics problem. *Algorithmica 15*, 5 (May 1996), 481–494.

[14] Apostolico, A., and Szpankowski, W. Self-alignment in words and their applications. *J. Algorithms 13*, 3 (1992), 446–467.

[15] Arnavut, Z., and Magliveras, S. S. Block sorting and compression. In *Data Compression Conference* (Snowbird, Utah, 1997), J. A. Storer and M. Cohn, Eds., IEEE Computer Society Press, TCC, pp. 181–190.

[16] Balkenhol, B., Kurtz, S., and Shtarkov, Y. Modification of the burrows and wheeler data compression algorithm. In *Data Compression Conference* (Snowbird, Utah, 1999), J. A. Storer and M. Cohn, Eds., IEEE Computer Society Press, TCC, pp. 124–136.

[17] Barnsley, M. *Fractals Everywhere.* Academic Press, New York, NY, USA, 1988.

[18] Békési, J., Galambos, G., Pferschy, U., and Woeginger, G. J. Greedy algorithms for on-line data compression. *J. Algorithms 25*, 2 (Nov. 1997), 274–289.

[19] Bell, T., and Kulp, D. Longest-match string searching for Ziv-Lempel compression. *Softw. Pract. Exp. 23*, 7 (July 1993), 757–771.

[20] Bell, T. C., Cleary, J. G., and Witten, I. H. *Text Compression.* Prentice Hall, 1990.

[21] Bentley, J. L., Sleator, D. D., Tarjan, R. E., and Wei, V. K. A locally adaptive data compression scheme. *Commun. ACM 29*, 4 (Apr. 1986), 320–330.

[22] Bloom, C. LZP: A new data compression algorithm. In *Data Compression Conference* (Snowbird, Utah, 1996), J. A. Storer and M. Cohn, Eds., IEEE Computer Society Press, TCC, pp. 425–434.

[23] Blumer, A., Blumer, J., Ehrenfeucht, A., Haussler, D., and McConnel, R. Linear size finite automata for the set of all subwords of a word: an outline of results. *Bull. Eur. Assoc. Theor. Comput. Sci. 21* (1983), 12–20.

[24] Blumer, A., Blumer, J., Ehrenfeucht, A., Haussler, D., and McConnel, R. Complete inverted files for efficient text retrieval and analysis. *J. Assoc. Comput. Mach. 34*, 3 (1987), 578–595.

[25] Blumer, A., Ehrenfeucht, A., and Haussler, D. Average size of suffix trees and DAWGS. *Discret. Appl. Math. 24* (1989), 37–45.

[26] Boyer, R. S., and Moore, J. S. A fast string searching algorithm. *Commun. ACM 20*, 10 (1977), 762–772.

[27] Breslauer, D. Personal Communication. .

[28] Brodal, G. S. Finger search trees with constant insertion time. In *ACM-SIAM Annual Symposium on Discrete Algorithms* (San Francisco, California, 25–27 Jan. 1998), pp. 540–549.

[29] Brown, M., and Tarjan, R. E. Design and analysis of a data structure for representing sorted lists. *SIAM J. Comput. 9* (1980), 594–614.

[30] Burrows, M., and Wheeler, D. J. A block-sorting lossless data compression algorithm. *TR Digital Equipments Corporation*, 124 (May 1994).

[31] Chang, W. I., and Lawler, E. L. Sublinear approximate string matching and biological applications. *Algorithmica 12*, 4/5 (Oct./Nov. 1994), 327–344.

[32] Chen, M. T., and Seiferas, J. Efficient and elegant subword tree construction. In *Combinatorial Algorithms on Words*, A. Apostolico and Z. Galil, Eds., vol. 12 of *NATO Advanced Science Institutes, Series F.* Springer-Verlag, Berlin, 1985, pp. 97–107.

[33] Cleary, J. G., Teahan, W. J., and Witten, I. H. Unbounded length contexts for PPM. In *Data Compression Conference* (Snowbird, Utah, 1995), J. A. Storer and M. Cohn, Eds., IEEE Computer Society Press, TCC, pp. 52–61.

[34] Cleary, J. G., and Witten, I. H. Data compression using adaptive coding and partial string matching. *IEEE Trans. Comput. 32* (1984), 396–402.

[35] Cohn, M., and Khazan, R. Parsing with suffix and prefix dictionaries. In *Data Compression Conference* (Snowbird, Utah, 1996), J. A. Storer and M. Cohn, Eds., IEEE Computer Society Press, TCC, pp. 180–189.

[36] Cormack, G. V., and Horspool, R. N. S. Data compression using dynamic Markov modelling. *Comput. J. 30*, 6 (Dec. 1987), 541–550.

[37] Cormen, T. H., Leiserson, C. E., and Rivest, R. L. *Introduction to Algorithms.* MIT Press, 1990.

[38] Cover, T. M., and Thomas, J. A. *Elements of Information Theory.* Wiley Series in Telecommunication. John Wiley & Son, New York, NY, USA, 1991.

[39] Crochemore, M., and Rytter, W. Efficient parallel algorithms to test square-freeness and factorize strings. *Inf. Process. Lett. 38*, 2 (1991), 57–60.

[40] Crochemore, M., and Rytter, W. *Text Algorithms.* Oxford University Press, 1994.

[41] Crochemore, M., and Verin, R. Direct construction of compact directed acyclic word graphs. *Lecture Notes in Computer Science 1264* (1997), 116–??

[42] De Agostino, S. P-complete problems in data compression. *Theor. Comput. Sci. 127*, 1 (May 1994), 181–186.

[43] De Agostino, S. A parallel decoding algorithm for LZ2 data compression. *Parallel Comput. 21*, 12 (Dec. 1995), 1957–1961.

[44] De Agostino, S., and Silvestri, R. A worst-case analysis of the LZ2 compression algorithm. *Information and Computation 139* (1997), 258–268.

[45] De Agostino, S., and Storer, J. A. Parallel algorithms for optimal compression using dictionaries with the prefix property. In *Data Compression Conference* (Snowbird, Utah, 1992), J. A. Storer and M. Cohn, Eds., IEEE Computer Society Press, TCC, pp. 52–61.

[46] De Agostino, S., and Storer, J. A. On-line versus off-line computation in dynamic text compression. *Inf. Process. Lett. 59*, 3 (1996), 169–174.

[47] DeVore, R. A., Jawerth, B., and Lucier, B. J. Image compression through wavelet transform coding. *IEEE Trans. Inf. Theory 38*, 2 (Mar. 1992), 719–746.

[48] Dietzfelbinger, M., Karlin, A., Mehlhorn, K., Heide, F. M. A. D., Rohnert, H., and Tarjan, R. E. Dynamic perfect hashing: Upper and lower bounds. *SIAM J. Comput. 23*, 4 (1994), 738–761.

[49] Elias, P. Universal codeword sets and representations of the integers. *IEEE Trans. Inf. Theory 21* (1975), 194–202.

[50] Elias, P. Interval and recency rank source coding: two on-line adaptive variable-length schemes. *IEEE Trans. Inf. Theory 33*, 1 (1987), 3–10.

[51] Even, S., and Rodeh, M. Economical encoding of commas between strings. *Commun. ACM 21*, 4 (Apr. 1978), 315–317.

[52] Farach, M. Optimal suffix tree construction with large alphabets. In *38th Annual Symposium on Foundations of Computer Science* (Miami Beach, Florida, 20–22 Oct. 1997), IEEE, pp. 137–143.

[53] Farach, M., and Muthukrishnan, S. Optimal logarithmic time randomized suffix tree construction. *Lecture Notes in Computer Science 1099* (1996), 550.

[54] Farach, M., Noordewier, M., Savari, S., Shepp, L., Wyner, A., and Ziv, J. On the entropy of DNA: Algorithms and measurements based on memory and rapid convergence. In *ACM-SIAM Annual Symposium on Discrete Algorithms* (San Francisco, California, 22–24 Jan. 1995), pp. 48–57.

[55] Fenwick, P. M. The Burrows-Wheeler transform for block sorting text compression: Principles and improvements. *Comput. J. 39*, 9 (1996), 731–740.

[56] Fenwick, P. M. Symbol ranking text compression with Shannon recodings. *J. Universal Computer Science 3*, 2 (1997), 70–85.

[57] Fenwick, P. M. Symbol ranking text compressors: Review and implementation. *Softw. Pract. Exp. 28*, 5 (1998), 547–559.

[58] Ferragina, P. Dynamic text indexing under string updates. *J. Algorithms 22*, 2 (Feb. 1997), 296–328.

[59] Ferragina, P., and Grossi, R. A fully-dynamic data structure for external substring search. In *Proceedings of the 27th ACM Symposium on the Theory of Computing* (Las Vegas, Nevada, 1995), ACM Press.

[60] Fiala, E. R., and Greene, D. H. Data compression with finite windows. *Commun. ACM 32*, 4 (Apr. 1989), 490–505.

[61] Fraenkel, A. S., Simpson, J., and Paterson, M. S. On weak circular squares in binary words. In *Annual Symposium on Combinatorial Pattern Matching* (Aarhus, Denmark, 30 June–2 July 1997), A. Apostolico and J. Hein, Eds., vol. 1264 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, pp. 76–82.

[62] Fredman, M. L., Komlos, J., and Szemeredi, E. Storing a sparse table with $O(1)$ worst case access time. *J. Assoc. Comput. Mach. 31*, 3 (1984), 538–544.

[63] Fu, K. S., and Booth, T. L. Grammatical inference: Introduction and survey – Part I. *IEEE Trans. on Systems, Man and Cybernetics 5* (1975), 95–111.

[64] Fu, K. S., and Booth, T. L. Grammatical inference: Introduction and survey – Part II. *IEEE Trans. on Systems, Man and Cybernetics 5* (1975), 112–127.

[65] Gailly, J. L., and Adler, M. ZLib compression library. Available at `http://www.cdrom.com/pub/infozip/zlib/`.

[66] Gall, D. L. MPEG: a video compression standard for multimedia applications. *Commun. ACM 34, 4* (1991), 46–58.

[67] Gallager, R. G. Variations on a theme by Huffman. *IEEE Trans. Inf. Theory 24*, 6 (Nov. 1978), 668–674.

[68] Gallant, J. *String Compression Algorithms.* PhD thesis, Dept. of Electrical Engineering and Computer Sciences, Princeton University, Princeton, NJ, 1982.

[69] Gersho, A., and Gray, R. M. *Vector Quantization and Signal Compression.* Kluwer Academic Publishers, Boston, 1992.

[70] Gonzalez Smith, M. E., and Storer, J. A. Parallel algorithms for data compression. *J. Assoc. Comput. Mach. 32*, 2 (Apr. 1985), 344–373.

[71] Grinberg, D., Rajagopalan, S., Venkatesan, R., and Wei, V. K. Splay trees for data compression. In *ACM-SIAM Annual Symposium on Discrete Algorithms* (San Francisco, California, 22–24 Jan. 1995), pp. 522–530.

[72] Grumbach, S., and Tahi, F. Compression of DNA sequences. In *Data Compression Conference* (Snowbird, Utah, 1993), J. A. Storer and M. Cohn, Eds., pp. 340–350.

[73] Grumbach, S., and Tahi, F. A new challenge for compression algorithms: genetic sequences. *Inf. Proc. and Mngm. 30*, 6 (1994), 875–886.

[74] Gu, M., Farach, M., and Beigel, R. An efficient algorithm for dynamic text indexing. In *ACM-SIAM Annual Symposium on Discrete Algorithms* (Arlington, VA, 1994), pp. 697–704.

[75] Gusfield, D. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.

[76] Hariharan, R. Optimal parallel suffix tree construction. In *Proceedings of the 26th ACM Symposium on the Theory of Computing* (Montréal, Canada, 1994), ACM Press, pp. 290–299.

[77] Hartman, A., and Rodeh, M. Optimal sparsing of strings. In *Combinatorial Algorithms on Words*, A. Apostolico and Z. Galil, Eds., vol. 12 of *NATO Advanced Science Institutes, Series F.* Springer-Verlag, Berlin, 1985, pp. 155–167.

[78] Horspool, R. N. The effect of non-greedy parsing in Ziv-Lempel compression methods. In *Data Compression Conference* (Snowbird, Utah, 1995), J. A. Storer and M. Cohn, Eds., IEEE Computer Society Press, TCC, pp. 302–311.

[79] Horspool, R. N., and Cormack, G. V. Data compression based on token recognition. manuscript, Oct. 1983.

[80] Huffman, D. A. A method for the construction of minimum-redundancy codes. *Proc. I.R.E. 40*, 9 (1952), 1098–1101.

[81] Hunt, J. J., Vo, K.-P., and Tichy, W. F. An empirical study of delta algorithms. In *IEEE Soft. Config. and Maint. Workshop* (1996).

[82] Irving, R. Suffix binary search trees. Tech. rep., University of Glasgow, Computing Science Department, April 1996. http://www.dcs.gla.ac.uk/ rwi/papers/.

[83] Jacquet, P., and Szpankowski, W. Asymptotic behavior of the Lempel-Ziv parsing scheme and digital search trees. *Theor. Comput. Sci. 144*, 1–2 (June 1995), 161–197.

[84] Jones, D. W. Application of splay trees to data compression. *Commun. ACM 31*, 8 (Aug. 1988), 996–1007.

[85] Käerkkäeinen, J. Suffix cactus: A cross between suffix tree and suffix array. In *Proceedings of the 6th Annual Symposium on Combinatorial Pattern Matching* (Espoo, Finland, 1995), Z. Galil and E. Ukkonen, Eds., vol. 937 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, pp. 191–204.

[86] Käerkkäeinen, J., and Ukkonen, E. Sparse suffix trees. *Lecture Notes in Computer Science 1090* (1996), 219–230.

[87] Katajainen, J., and Raita, T. An approximation algorithm for space-optimal encoding of a text. *Comput. J. 32*, 3 (June 1989), 228–237.

[88] Katajainen, J., and Raita, T. An analysis of the longest match and the greedy heuristics in text encoding. *J. Assoc. Comput. Mach. 39*, 2 (Apr. 1992), 281–294.

[89] Knuth, D. E. Dynamic Huffman coding. *J. Algorithms 6*, 2 (June 1985), 163–180.

[90] Knuth, D. E., Morris, Jr, J. H., and Pratt, V. R. Fast pattern matching in strings. *SIAM J. Comput. 6*, 1 (1977), 323–350.

[91] Kosaraju, S. R. Localized search in sorted lists. In *Conference Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computation* (Milwaukee, Wisconsin, 11–13 May 1981), pp. 62–69.

[92] Kosaraju, S. R., and Manzini, G. Compression of low entropy strings with Lempel-Ziv algorithms. In *Compression and Complexity of Sequences* (1998), B. Carpentieri, A. D. Santis, U. Vaccaro, and J. A. Storer, Eds.

[93] Krogh, A., Brown, M., Mian, I. S., Sjolander, K., and Haussler, D. Hidden markov models in computational biology: applications to protein modelling. *J. Mol. Biol. 235* (1994), 1501–1531.

[94] Kurtz, S. Reducing the space requirments of suffix trees. Tech. Rep. 98-03, Technischen Fakultät, Universität Bielefeld, 1998.

[95] Landau, G. M., Schieber, B., and Vishkin, U. Parallel construction of a suffix tree. In *Proceedings of the 14th International Colloquium on Automata, Languages and Programming* (Karlsruhe, Germany, 1987), T. Ottmann, Ed., no. 267 in Lecture Notes in Computer Science, Springer-Verlag, Berlin, pp. 314–325.

[96] Langdon, Jr., G. G. Erratum: "An introduction to arithmetic coding". *IBM Journal of Research and Development 28*, 4 (1984), 498–498.

[97] Langdon, Jr., G. G. An introduction to arithmetic coding. *IBM Journal of Research and Development 28*, 2 (Mar. 1984), 135–149.

[98] Larsson, N. J. Extended application of suffix trees to data compression. In *Data Compression Conference* (Snowbird, Utah, 1996), J. A. Storer and M. Cohn, Eds., IEEE Computer Society Press, TCC, pp. 190–199.

[99] Larsson, N. J. The context tree of block sorting compression. In *Data Compression Conference* (Snowbird, Utah, 1998), J. A. Storer and M. Cohn, Eds., IEEE Computer Society Press, TCC, pp. 189–198.

[100] Li, M., and Vitanyi, P. *Introduction to Kolmogorov Complexity and its Applications.* Springer-Verlag, Aug. 1993.

[101] Loewenstern, D., and Yianilos, P. N. Significant lower entropy estimates for natural DNA sequences. In *Data Compression Conference* (Snowbird, Utah, 1997), J. A. Storer and M. Cohn, Eds., pp. 151–160.

[102] Loewenstern, D. M., Berman, H. M., and Hirsch, H. Maximum a posteriori classification of DNA structure from sequence information. *Pacific Symp. Biotech.* (Jan. 1998).

[103] Loewenstern, D. M., Hirsh, H., Yianilos, P., and Noordewier, M. DNA sequence classification using compression-based induction. Tech. Rep. 95-04, DIMACS, Apr. 1995.

[104] Lonardi, S., and Sommaruga, P. Fractal image approximation and orthogonal bases. *Image Communications 15*, 4 (March 1999), 413–423.

[105] Louchard, G., and Szpankowski, W. Average profile and limiting distribution for a phrase size in the Lempel-Ziv parsing algorithm. *IEEE Trans. Inf. Theory 41* (1995).

[106] Louchard, G., and Szpankowski, W. Averaged redundancy rate of the Lempel-Ziv code. In *Data Compression Conference* (Snowbird, Utah, 1996), J. A. Storer and M. Cohn, Eds., IEEE Computer Society Press, TCC.

[107] Louchard, G., and Szpankowski, W. On the average redundancy rate of the Lempel-Ziv code. *IEEE Trans. Inf. Theory 43* (1997).

[108] Manber, U., and Myers, G. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing 22* (1993).

[109] Matias, Y., and Cenk Sahinalp, S. On the optimality of parsing in dynamic dictionary based data compression. In *to appear in ACM-SIAM Annual Symposium on Discrete Algorithms* (1999).

[110] Matias, Y., Muthukrishnan, S., Sahinalpk, S. C., and Ziv, J. Augmenting suffix trees with applications. In *Proceedings of the 6th Annual European Symposium* (Venice, Italy, 1998), G. Bilardi, G. F. Italiano, A. Pietracaprina, and G. Pucci, Eds., no. 1461 in Lecture Notes in Computer Science, Springer-Verlag, Berlin, pp. 67–78.

[111] Matias, Y., Rajpoot, N., and Cenk Sahinalp, S. Implementation and experimental evaluation of flexible parsing for dynamic dictionary based data compression. In *Workshop on Algorithm Engineering* (Saarbrucken, Germany, 1998), pp. 49–61.

[112] McCreight, E. M. A space-economical suffix tree construction algorithm. *J. Assoc. Comput. Mach. 23*, 2 (Apr. 1976), 262–272.

[113] Mewes, H. W., and Heumann, K. Genome analysis: pattern search in biological macromolecules. In *Proceedings of the 6th Annual Symposium on Combinatorial Pattern Matching* (Espoo, Finland, 1995), Z. Galil and E. Ukkonen, Eds., no. 937 in Lecture Notes in Computer Science, Springer-Verlag, Berlin, pp. 261–285.

[114] Miclet, L. Grammatical inference. In *Syntactic and Structural Pattern Recognition; Theory and Applications*, H. Bunke and A. Sanfeliu, Eds. World Scientific, Singapore, 1990, ch. 9.

[115] Miller, V. S., and Wegman, M. N. Variations on a theme by Ziv and Lempel. In *Combinatorial Algorithms on Words*, A. Apostolico and Z. Galil, Eds., NATO Advanced Science Institutes, Series F. Springer-Verlag, 1985, pp. 131–140.

[116] Moffat, A. Implementing the PPM data compression scheme. *IEEE Trans. Comput. 38*, 11 (1990), 1917–1921.

[117] Mohtashemi, M. On the cryptanalysis of Huffman codes. Thesis (m.s.), Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, USA, May 1992. Also published as Technical report MIT/LCS/TR-617.

[118] Morrison, D. R. PATRICIA - practical algorithm to retrieve coded in alphanumeric. *J. Assoc. Comput. Mach. 15*, 4 (1968), 514–534.

[119] Musser, D. R., and Stepanov, A. A. Algorithm-oriented generic libraries. *Softw. Pract. Exp. 24*, 7 (July 1984), 623–642.

[120] Nagumo, H., Li, M., and Watson, K. Parallel algorithms for the static dictionary compression. In *Data Compression Conference* (Snowbird, Utah, 1995), J. A. Storer and M. Cohn, Eds., IEEE Computer Society Press, TCC, pp. 162–171.

[121] Nelson, M. Arithmetic coding and statistical modeling. *Dr. Dobb's Journal* (February 1991).

[122] Nelson, M. Examining the zlib compression library. *Dr. Dobb's Journal* (January 1997).

[123] Nevill-Manning, C., and Witten, I. H. Linear-time, incremental hierarchy inference for compression. In *Data Compression Conference* (Snowbird, Utah, 1997), J. A. Storer and M. Cohn, Eds., IEEE Computer Society Press, TCC, pp. 3–11.

[124] Nevill-Manning, C., and Witten, I. H. Phrase hierarchy inference and compression in bounded space. In *Data Compression Conference* (Snowbird, Utah, 1998), J. A. Storer and M. Cohn, Eds., IEEE Computer Society Press, TCC, pp. 179–188.

[125] Nevill-Manning, C., Witten, I. H., and Maulsby, D. Compression by induction of hierarchical grammars. In *Data Compression Conference* (Snowbird, Utah, 1994), J. A. Storer and M. Cohn, Eds., IEEE Computer Society Press, TCC, pp. 244–253.

[126] Nevill-Manning, C. G., and Witten, I. H. Compression and explanation using hierarchical grammars. *Comput. J. 40*, 2/3 (1997), 103.

[127] Pennebaker, W. B., Mitchell, J. L., Langdon, Jr., G. G., and Arps, R. B. An overview of the basic principles of the Q-coder adaptive binary arithmetic coder. *IBM Journal of Research and Development 32*, 6 (Nov. 1988), 717–726.

[128] Raita, T., and Teuhola, J. Predictive text compression by hashing. In *Research and Development in Information Retrieval: Proceedings of the Tenth Annual International ACM SIGIR Conference* (New York, NY 10036, USA, 1987), C. T. Yu and C. J. Rijsbergen, Eds., Storage/Retrieval Techniques I, ACM Press, pp. 223–233.

[129] Rivals, E., Delahaye, J. P., Dauchet, M., and Delgrange, O. A guaranteed compression scheme for repetitive DNA sequences. In *Data Compression Conference* (Snowbird, Utah, 1996), J. A. Storer and M. Cohn, Eds., p. 453.

[130] Rivals, E., Delgrange, O., Delahaye, J. P., Dauchet, M., Delorme, M. O., Henaut, A., and Ollivier, E. Detection of significant patterns by compression algorithms: the case of approximate tandem repeats in DNA sequences. *CABIOS 13*, 2 (1997), 131–136.

[131] Rodeh, M., Pratt, V. R., and Even, S. Linear algorithm for data compression via string matching. *J. Assoc. Comput. Mach. 28*, 1 (Jan. 1981), 16–24.

[132] Rubin, F. Experiments in text file compression. *Commun. ACM 19*, 11 (Nov. 1976), 617–623.

[133] Rubin, F. Cryptographic aspects of data compression codes. *Cryptologia 3*, 4 (Oct. 1979), 202–205.

[134] Ryabko, B. Y. Data compression by means of a 'book stack'. *Problemy Peredachi Informatsii 16*, 3 (1980).

[135] Sadakane, K. A fast algorithm for making suffix arrays and for Burrows-Wheeler transformation. In *Data Compression Conference* (Snowbird, Utah, 1998), J. A. Storer and M. Cohn, Eds., IEEE Computer Society Press, TCC, pp. 129–138.

[136] Sadakane, K. Text compression using recency rank with context and relation to context sorting, block sorting and PPM. In *Compression and Complexity of Sequences*, B. Carpentieri, A. D. Santis, U. Vaccaro, and J. A. Storer, Eds. IEEE Computer Society Press, TCC, 1998.

[137] Sakakibara, Y. Recent advances of grammatical inference. *Theor. Comput. Sci. 185*, 1 (Oct. 1997), 15–45.

[138] Sakakibara, Y., Brown, M., Hughey, R., Mian, I. S., Sjölander, K., Underwood, R. C., and Haussler, D. Stochastic context-free grammars for tRNA modeling. *Nucleic Acids Research 22* (1994), 5112–5120.

[139] Sakakibara, Y., Brown, M., Hughey, R., Mian, I. S., Sjölander, K., Underwood, R. C., and Haussler, D. Recent methods for RNA modeling using stochastic context-free grammars. In *Annual Symposium on Combinatorial Pattern Matching* (Asilomar, California, 5-8 June 1994), M. Crochemore and D. Gusfield, Eds., vol. 807 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, pp. 289–306.

[140] Salomon, D. *Data Compression: The Complete Reference.* Springer-Verlag, 1998.

[141] Savari, S. A. Redundancy of the Lempel-Ziv incremental parsing rule. *IEEE Trans. Inf. Theory 43*, 1 (June 1997), 9–21.

[142] Schuegraf, E., and Heaps, H. S. A comparison of algorithms for data base compression by use of fragments as language elements. *Information Storage and Retrieval 10* (1974), 309–319.

[143] Searls, D. B. The linguistics of DNA. *American Scientist. 80*, 6 (Nov.-Dec. 1992), 579–591.

[144] Sheinwald, D., Lempel, A., and Ziv, J. On compression with two-way head machines. In *Data Compression Conference* (Snowbird, Utah, 1991), J. A. Storer and M. Cohn, Eds., IEEE Computer Society Press, TCC.

[145] Sheinwald, D., Lempel, A., and Ziv, J. On encoding and decoding with two-way head machines. *Inf. Comput. 116*, 1 (Jan. 1995), 128–133.

[146] Sleator, D. S., and Tarjan, R. E. Self-adjusting binary search trees. *J. Assoc. Comput. Mach. 32*, 3 (1985), 652–686.

[147] Stauffer, L. M., and Hirschberg, D. S. PRAM algorithms for static dictionary compression. In *Proceedings of the 8th International Symposium on Parallel Processing* (Los Alamitos, CA, USA, Apr. 1994), H. J. Siegel, Ed., IEEE Computer Society Press, pp. 344–348.

[148] Stephen, G. A. *String Searching Algorithms.* Lecture-Notes-Series-on-Computing. World-Scientific-Publishing, Oct. 1994.

[149] Storer, J. A. Np-completeness results concerning data compression. Report 234, Princeton University, 1977.

[150] Storer, J. A. *Data Compression: Methods and Complexity Issues.* PhD thesis, Dept. of Electrical Engineering and Computer Sciences, Princeton University, Princeton, NJ, 1979.

[151] Storer, J. A. Textual substitution techniques for data compression. In *Combinatorial Algorithms on Words*, A. Apostolico and Z. Galil, Eds., NATO Advanced Science Institutes, Series F. Springer-Verlag, Berlin, 1985.

[152] Storer, J. A. *Data Compression: Methods and Theory.* Computer Science Press, 1988.

[153] Storer, J. A., and Reif, J. H. A parallel architecture for high-speed data compression. *Journal of Parallel and Distributed Computing 13*, 2 (Oct. 1991), 222–227.

[154] Storer, J. A., and Reif, J. H. Error-resilient optimal data compression. *SIAM J. Comput. 26*, 4 (Aug. 1997), 934–949.

[155] Storer, J. A., and Szymanski, T. G. The macro model for data compression. In *Proceedings of the 10th ACM Symposium on the Theory of Computing* (San Diego, CA, 1978), ACM Press, pp. 30–39.

[156] Storer, J. A., and Szymanski, T. G. Data compression via textual substitution. *J. Assoc. Comput. Mach. 29*, 4 (Oct. 1982), 928–951.

[157] Thomborson, C. The V.42bis standard for data-compressing modems. *IEEE Micro 12*, 5 (Oct. 1992), 41–53.

[158] Tjalkens, T., Volf, P., and Willems, F. A context-tree weighting method for text-generating sources. In *Data Compression Conference* (Snowbird, Utah, 1997), J. A. Storer and M. Cohn, Eds., IEEE Computer Society Press, TCC, p. 472.

[159] Tjalkens, T., and Willems, F. Implementing the context-tree weighting method: Arithmetic coding. In *International Conference on Combinatorics, Information Theory & Statistics* (Portland, Maine, 1997).

[160] Ukkonen, E. On-line construction of suffix trees. *Algorithmica 14*, 3 (1995), 249–260.

[161] Vitter, J. S. Design and analysis of dynamic Huffman codes. *J. Assoc. Comput. Mach. 34*, 4 (Oct. 1987), 825–845.

[162] Volf, P. A. J., and Willems, F. Switching between two universal source coding algorithms. In *Data Compression Conference* (Snowbird, Utah, 1998), J. A. Storer and M. Cohn, Eds., IEEE Computer Society Press, TCC, pp. 491–500.

[163] Wagner, R. A. Common phrases and minimum-space text storage. *Commun. ACM 16*, 3 (1973), 148–152.

[164] Wallace, G. K. The JPEG still picture compression standard. *Commun. ACM 34*, 4 (Apr. 1991), 30–44.

[165] Weiner, P. Linear pattern matching algorithm. In *Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory* (Washington, DC, 1973), pp. 1–11.

[166] Welch, T. A. A technique for high-performance data compression. *IEEE Computer 17*, 6 (June 1984), 8–19.

[167] Willems, F., and Tjalkens, T. Complexity reduction of the context-tree weighting method. In *18th Benelux Symposium on Information Theory* (Veldhoven, The Netherlands, 1997).

[168] Willems, F. M. J., Shtarkov, Y. M., and Tjalkens, T. J. The context-tree weighting method: basic properties. *IEEE Trans. Inf. Theory* (May 1995), 653–664.

[169] Willems, F. M. J., Shtarkov, Y. M., and Tjalkens, T. J. Context weighting for general finite context sources. *IEEE Trans. Inf. Theory* (Sept. 1996), 1514–1520.

[170] Williams, R. N. An extremely fast Ziv-Lempel data compression algorithm. In *Data Compression Conference* (Snowbird, Utah, 1991), J. A. Storer and M. Cohn, Eds., IEEE Computer Society Press, TCC, pp. 362–371.

[171] Witten, I. H., Neal, R. M., and Cleary, J. G. Arithmetic coding for data compression. *Commun. ACM 30*, 6 (June 1987), 520–540.

[172] Wyner, A. J., and Wyner, A. D. Improved redundancy of a version of the Lempel-Ziv algorithm. *IEEE Trans. Inf. Theory 41* (1995).

[173] Wyner, A. J., and Ziv, J. The sliding window Lempel-Ziv algorithm is asymptotically optimal. *Proceedings of the IEEE 82* (June 1994), 872–877.

[174] Yokoo, H. Improved variations relating the Ziv-Lempel and Welch-type algorithms for sequential data compression. *IEEE Trans. Inf. Theory 38* (1992).

[175] Yokoo, H. An adaptive data compression method based on context sorting. In *Data Compression Conference* (Snowbird, Utah, 1996), J. A. Storer and M. Cohn, Eds., IEEE Computer Society Press, TCC.

[176] Yokoo, H. Data compression using a sort-based similarity measure. *Comput. J. 40*, 2/3 (1997), 94–100.

[177] YOKOO, H. Context tables: a tool for describing text compression algorithms. In *Data Compression Conference* (Snowbird, Utah, 1998), J. A. Storer and M. Cohn, Eds., IEEE Computer Society Press, TCC, pp. 299–308.

[178] ZIV, J., AND LEMPEL, A. On the complexity of finite sequences. *IEEE Trans. Inf. Theory 22* (1976), 75–81.

[179] ZIV, J., AND LEMPEL, A. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory 23*, 3 (May 1977), 337–343.

[180] ZIV, J., AND LEMPEL, A. Compression of individual sequences via variable-rate coding. *IEEE Trans. Inf. Theory 24*, 5 (Sept. 1978), 530–536.