Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica (8009)

COMPRESSION OF LABELED SPINE TREES

Relatore
Chiar.mo Prof.
Ugo Dal Lago

Corelatore
Gabriele Vanoni

Presentata da
Oscar Barreca

Sessione Straordinaria
2020—2021

I have made this letter longer than usual
because I lack the time to make it shorter.
—Blaise Pascal

Contents

Contents iii

List of Figures v

List of Tables vii

1 Introduction 1
- 1.1 Historical Notes of Information Theory . . . 2
- 1.2 Overview of Information Theory . . . 3
  - 1.2.1 Entropy of a Random Variable . . . 3
  - 1.2.2 Conditional Entropy . . . 6
  - 1.2.3 Relative Entropy and Mutual Information . . . 7
- 1.3 Overview of Data Compression . . . 9
  - 1.3.1 Models . . . 9
  - 1.3.2 Codes . . . 10
  - 1.3.3 Compression Algorithms . . . 12

2 Introduction to Spines 17
- 2.1 Spine Trees . . . 17
- 2.2 Basic Concepts and Notation . . . 18
  - 2.2.1 Mathematical Notation . . . 18
  - 2.2.2 XBW Transform . . . 18

3 Spine Detection 21
- 3.1 XBW- Transform . . . 21
- 3.2 Informal Detection Description . . . 22
- 3.3 Properties of XBW- Transforms on Spines . . . 23
- 3.4 An Algorithm for Spine Detection . . . 30

4 Storage of Spines 35
- 4.1 Pruning of Spine Trees . . . 35
- 4.2 Context Manipulation of Spine Trees . . . 36

5 Spine Compression 39
- 5.1 Logarithmic Compression . . . 39
- 5.2 Problem Statement . . . 40
- 5.3 Candidate Algorithms . . . 41

6 Conclusions 47

7 Acknowledgements 49

List of Figures

1.1 Entropy $H(X) = -p \log p - (1-p)\log(1-p)$ of the random variable X as the value of p goes from 0 to 1. 5
1.2 Binary tree representing the Huffman encoding of the alphabet of Table 1.2. 13

2.1 Two spine trees. 18
2.2 An example XBW transform. 19

3.1 An example XBW$^-$ transform. 22
3.2 The left spine of Figure 2.1a along with its XBW$^-$ transform. . 23
3.3 The right spine of Figure 2.1b along with its XBW$^-$ transform. 23
3.4 A tree T with a left spine of height $h = 1$. 24
3.5 A tree T with a left spine of height $h > 1$. 25
3.6 A left spine of height $h = 1$. 26
3.7 A left spine of height $h = 1$. 27
3.8 A tree T with a right spine of height $h > 1$. 28
3.9 The left-spine automaton. 31
3.10 The right-spine automaton. 32

4.1 Pruning of a tree T. 35
4.2 The tree of Figure 4.1, along with its XBW$^-$ transform. 36
4.3 The tree of Figure 4.1, along with its XBW transform that has undergone a context manipulation. Here, $\xi = \mathrm{s}$. 37

5.1 An example of a LCF (Logarithmically Compressible File) of 1KiB. The name for these files is a convenience, since LCFs aren't necessarily logarithmically compressible, and if they are, they are with respect to one algorithm, without necessarily being with respect to another. 43
5.2 Compression of $\texttt{AB}^{2^i}$ for $i \in \{1, \ldots, 16\}$. 46
5.3 Compression of $\texttt{ABCDE}^{2^i}$ for $i \in \{1, \ldots, 16\}$. 46

List of Tables

1.1 A four-letter alphabet, along with three possible codes. 10
1.2 Example probability distribution of a source alphabet Σ for the Huffman algorithm. The entropy of this distribution is 1.88 bits per symbol. 12
1.3 Codewords for the alphabet Σ of Table 1.2. 13

3.1 The $S[a, a+2]$ portion of the XBW$^-$ transform S. 25
3.2 The $S[a, a+2]$ portion of the XBW$^-$ transform S. 26
3.3 The $S[a, a+2]$ portion of the XBW$^-$ transform S. 27
3.4 The $S[a, a+2]$ portion of the XBW$^-$ transform S. 29

5.1 Maximum number N of allowable splits of x as its length n increases. As always, m denotes the number of bits needed to encode one single symbol of x uniformly. In this case, m = 8bits = 1byte; all other measures are likewise in bytes. d denotes the size of a single block out of N, and c the multiplicative constant from (5.2). We expressed c in terms of d and set $c = 10d$. 42
5.2 LZ77 performance on a series of exponentially-growing LCFs (Logarithmically Compressible Files). Having set a logarithmic constant $c = 10$, LZ77 achieves logarithmic compression up to a file of 32KiB, reflecting the fact that being able to do so for infinitely growing inputs becomes harder and harder. . . . 44

Sommario

Sviluppi recenti nell'implementazione di linguaggi di programmazione funzionali [ALV21] si affidano a una forma particolare di albero etichettato al fine di rappresentare la gerarchia di chiamate di funzioni. Tali alberi, da noi denominati *alberi spina*, o più semplicemente soltanto *spine*, sono caratterizzati da una struttura assai regolare, che li rende idonei al raggiungimento di un alto fattore di compressione, ovvero una riduzione nello spazio di memoria necessario alla loro memorizzazione. L'introduzione di un metodo di compressione efficiente per gli alberi spina garantirebbe perciò un miglioramento prestazionale nell'implementazione di linguaggi funzionali che si affidano ad essi.

Nella presente opera, ci prefiggiamo di indagare le possibilità di compressione di alberi spina. Poiché un albero spina può presentarsi all'interno di un qualunque altro albero genitore, più esteso e possibilmente privo di struttura, la loro "forma" necessita di essere riconosciuta da una procedura algoritmica. Avviamo dunque lo studio esaminando il problema del *rilevamento di spine* (Capitolo 3), proponendo metodi di rilevamento asintoticamente efficienti, sia in tempo che in spazio. Procediamo affrontando il problema della loro *memorizzazione* (Capitolo 4), vale a dire il modo in cui è possibile memorizzare alberi spina all'interno, o al di fuori, dei loro alberi genitori, e terminiamo dedicandoci al problema della loro *compressione* (Capitolo 5).

Lo studio della compressione di spine viene svolto traendo risultati già noti da un sotto ambito della teoria dell'informazione, noto come *compressione del dato* (*data compression*). Anziché focalizzarci sull'introduzione di nuovi algoritmi di compressione, cerchiamo di capire quali, tra quelli già esistenti, si presta meglio al compito della compressione di spine. Nello studiare la compressione di alberi spina cerchiamo, in particolare, di determinare quando una cosiddetta *compressione logaritmica* (Sezione 5.1), ossia una riduzione al logaritmo della dimensione originale di una spina, può essere ottenuta. Introduciamo dapprima un criterio teorico (*dominio di restrizione*, Sezione 5.2) per classificare l'efficacia delle possibili soluzioni, e progrediamo in maniera incrementale nelle proposte, fino a giungere ad una soluzione (l'approccio BWT–RLE, Sezione 5.3) che reputiamo molto promettente, sia nella capacità di ottenere una compressione logaritmica, sia nelle risorse (tempo e spazio) necessarie a realizzare questa riduzione.

Chapter 1

Introduction

Recent developments in the implementation of functional programming languages [ALV21] rely on a specific form of labeled tree to represent the call hierarchy of functions. Such trees, that we refer to as *spine trees*, or simply *spines* for short, possess a very regular structure that makes them suitable for high compression, i.e. a reduction in the space needed to represent them internally. The discovery of an efficient method of compression for spines would directly impact to the implementation of functional programming languages that rely on them.

In this work, we set out to investigate the possibilities of compression of labeled spine trees. Since we hypothesize spines to occur anywhere within a larger, possibly unstructured labeled tree, their "shape" needs to be recognized by an algorithmic procedure. We start therefore by addressing the problem of *spine detection* (Chapter 3), proposing asymptotically efficient methods, both in time and in space. We then progress to the problem of their *storage* (Chapter 4), namely the way in which they can be stored within or apart the tree they were originally embedded in, and finally turn to the problem of their own *compression* (Chapter 5).

The study of spine compression is done by applying already established results from the subfield of information theory known as *data compression*. Rather than devise a new compression algorithm from scratch, we try to understand which, among the already available ones, is fit to our own interests. In studying the compression of spine trees, we try in particular to determine when a so-called *logarithmic compression* (Section 5.1), namely a reduction of the size of the spine to the logarithm of its original size, can be achieved. We first introduce a theoretical criterion (*restriction domain*, Section 5.2) for classifying the adequacy of our solutions, and then progress incrementally to our findings, until we reach a solution (the BWT–RLE approach, Section 5.3) that is very promising both in the capacity of achieving logarithmic compression and in the computational resources (time and space) needed to obtain this storage reduction.

Since a relevant part of this work is centered on information theory, we would be interested to provide an elementary overview of the area before proceeding to the actual topics. This is what we will be doing in the next few sections.

1.1 Historical Notes of Information Theory

Information theory is the study of the quantification and transmission of information in the abstract and mathematical sense. It lies at the intersection of several scientific disciplines, including mathematics (probability theory and statistics), computer science (Kolmogorov complexity), electrical engineering (communication theory) and physics (thermodynamics) [CT06]. Important concepts of the field include the self-information and the entropy of a random variable, the mutual information between two random variables, and the unit of digital information itself, the bit. Having grown in size, today it encompasses a variety of subdisciplines, one of it being source coding, in turn divided into data compression (of a certain relevance to our study), error-correcting codes and cryptographic codes.

The advent of information theory as an acknowledged academic discipline is classically attributed to Claude Elwood Shannon's (1916–2021) foundational paper *A mathematical theory of communication* [Sha48] (1948), where he stated the source coding theorem, introduced the notion of entropy of a random variable, coined the term *bit* (partly credited to John Tukey) and devised the Shannon-Fano algorithm. Prior to that, related work had been conducted by scholars like Harry Nyquist (1889–1976) in *Certain Factors Affecting Telegraph Speed* [Nyq24] (1924) and Ralph Hartley's (1888–1970) *Transmission of Information* [Har28] (1928). Perhaps the earliest known contribution to information theory, classifiable as a technique of data compression, is due to Samuel Morse's (1792–1872) code in the 1830s. Morse code can be regarded as a data compression technique, for more frequents letters are associated to shorter codes, while infrequent letters are associated to longer ones.

Shannon's cornerstone work gave rise to a surge in publications short time after the time of its publication. In 1951, while completing his PhD, David Huffman (1925–1999) from MIT introduced the Huffman coding (1952) [Huf52]. Nasir Ahmed (1940–), in 1972, proposed the DCT (Discrete Cosine Transform), that has since become one of the most widely-used lossy algorithms for data compression, and the basis for modern digital compression standards (H.261, MPEG, JPEG, MP3, AAC). In 1977, Abrahm Lempel and Jacob Ziv jointly developed the dictionary-based compression scheme LZ77 [ZL77], soon followed by LZ78 [ZL78] one year later; in 1984, Terry Welch made some adjustments to the LZ78 algorithm, leading to the LZW (Lempel-Ziv-Welch) [Wel84] compression algorithm. In the course of the years, both LZ77 and LZ78 have inspired a whole family of compression algorithms.

Information theory has been at the center of many technological innovations. In 1986, the TIFF file format for high colour-depth images was introduced; shortly thereafter, in 1987, the GIF (Graphics Interchange Format) by CompuServe was created, replacing the use of run-length coding with LZW coding. In 1989, the German company Fraunhofer–Gesellshaft received a patent for MP3, an audio coding standard, and in the same year, Phil Kats published the `.zip` file format, including the DEFLATE algorithm (consisting of LZ77 followed by a run of Huffman coding). 1992 saw the birth of

the JPEG (Joint Photographic Experts Group) image format, offering a compromise between storage size and image quality, followed by PNG (Portable Network Graphics) in 1996, an image file format supporting lossless compression.

1.2 Overview of Information Theory

We will dedicate this section to an overview of the core concepts of information theory, so as to set the context for the subsequent section, dedicated to data compression, and the remainder of the thesis.

When possible, we will provide proof of our own statements. For a more in-depth and detailed discussion of the same concepts, you are invited refer to [CT06]. Please note that a basic knowledge of probability theory is assumed throughout this section.

1.2.1 Entropy of a Random Variable

Definition 1.2.1 — Entropy of a Random Variable

Let $X : \Omega \to S_X$ be a discrete random variable over the support set S_X, and let $p(x) = \mathbb{P}(X = x)$ be the PMF (Probability Mass Function) of X. We define the *entropy of* X as

$$H(X) = \sum_{x \in S_X} p(x) \log_2 \frac{1}{p(x)} \tag{1.1}$$

The entropy of a random variable is a scalar, real-valued quantity. The base of the logarithm is usually 2, in which case one simply writes log to mean $\log_2$. Otherwise, one writes $\log_b$ to specify a logarithm in a base b other than 2, and $H_b(X)$ for the entropy of X in base b. We call *bit* the unit of measurement of the entropy when $b = 2$, *nat* when $b = e$ and *dits* when $b = 10$.

As can be seen from (1.1), the entropy of X is the value of $\log \frac{1}{p(x)}$, averaged over the probabilities of X. This corresponds to the definition of an expected value, and therefore

$$H(X) = \mathbb{E}\left[\log \frac{1}{p(x)}\right] \tag{1.2}$$

It is often convenient to express the entropy in an equivalent, but simpler, form

$$H(X) = -\sum_{x} p(x) \log p(x) \tag{1.3}$$

The quantity $\log \frac{1}{p(x)}$ is termed *self-information* of an event. In other words, let $A \subseteq \Omega$ be an event. Then the self-information, measured in bits, related to the event A is

$$i(A) = \frac{1}{\log \mathbb{P}(A)} = -\log \mathbb{P}(A) \tag{1.4}$$

where $\mathbb{P}(A)$ denotes the probability of A occurring.

The entropy $H(X)$ of a random variable X denotes the average number of bits we need to describe an outcome of X.

Example 1.2.1 — Head vs. Tail Experiment

Let X represent a head-tossing experiment, so that $S_X = \{0, 1\}$. Assume that $p(0) = p$, while $p(1) = 1 - p$. Then

$$\begin{aligned} H(X) &= -\sum_x p(x) \log p(x) && (1.5)\\ &= -\left(p(0) \log p(0) + p(1) \log p(1)\right) && (1.6)\\ &= -\left(p \log p + (1-p) \log(1-p)\right) && (1.7)\\ &= -p \log p - (1-p) \log(1-p) && (1.8) \end{aligned}$$

Consider Figure 1.1 for a representation of the entropy of X as the value of p ranges from 0 to 1. If we set $p = 1/4$ then (1.8) becomes 0.81, meaning that we need, *on average*, 0.81 bits to encode the value of X. The minimum value of this expression is 0 when $p \in \{0, 1\}$ (in that case, X is a constant-valued random variable, and we need convey no information to tell the next outcome); the maximum value of it is 1 when $p = \frac{1}{2}$, that is when either head or tail are equiprobable.

Example 1.2.2

Consider a discrete random variable X, on a support set $S_X = \{0, 1, 2, 3, 4\}$, such that

$$p(x) = \begin{cases} \frac{1}{2} & \text{if } x = 0 \\ \frac{1}{4} & \text{if } x = 1 \\ \frac{1}{8} & \text{if } x = 2 \\ \frac{1}{16} & \text{if } x = 3 \\ \frac{1}{16} & \text{if } x = 4 \end{cases} \tag{1.9}$$

The entropy of X is

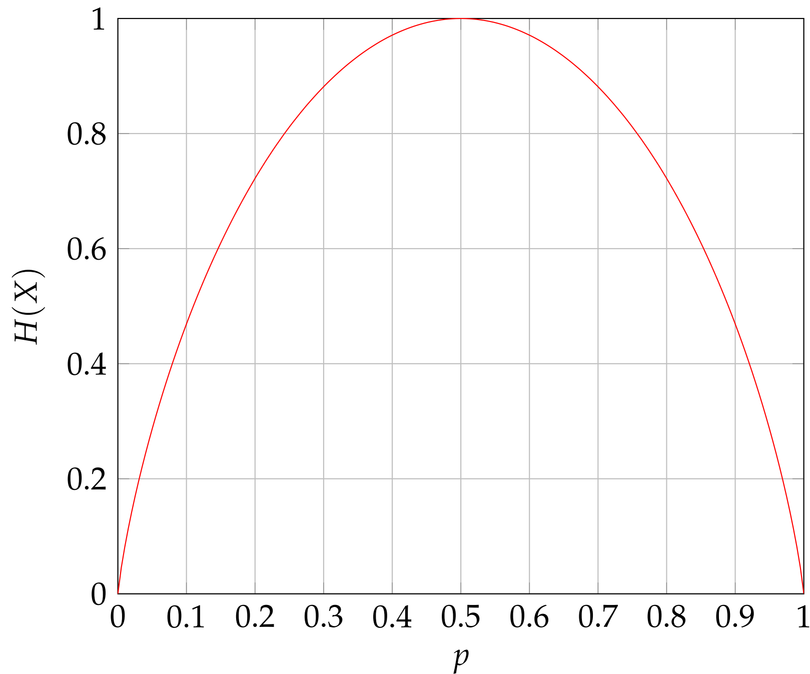


FIGURE 1.1: Entropy $H(X) = -p\log p - (1-p)\log(1-p)$ of the random variable X as the value of p goes from 0 to 1.

$$H(X) = \frac{1}{2}\log 2 + \frac{1}{4}\log 4 + \frac{1}{8}\log 8 + \frac{1}{16}\log 16 + \frac{1}{16}\log 16 \tag{1.10}$$
$$= \frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{4}{16} + \frac{4}{16} \tag{1.11}$$
$$= \frac{8+8+6+4+4}{16} \tag{1.12}$$
$$= 1.875 \text{ bits} \tag{1.13}$$

Observation 1.2.1

The entropy of a random variable X is maximum when it is uniformly distributed, that is when $X \sim \mathrm{U}(S_X)$, for a support set S_X of finite size.

Example 1.2.3

Let $X \sim \mathrm{U}(\{0, 1, \ldots, 7\})$, so that $\forall x,\ p(x) = \frac{1}{8}$. Then

$$H(X) = \underbrace{\frac{1}{8}\log 8 + \cdots + \frac{1}{8}\log 8}_{8} \tag{1.14}$$

$$= 8 \cdot \frac{1}{8}\log 8 \tag{1.15}$$

$$= 3 \text{ bits} \tag{1.16}$$

In general, the entropy of a uniform discrete random variable X on a support set S_X is $\log |S_X|$ bits.

1.2.2 Conditional Entropy

Just as we have defined the entropy for the case of a single random variable, it is also possible to define it for the case of a vector-valued random variable. We will give some examples, restricting, for simplicity, to the case of a pair of random variables (X_1, X_2).

Definition 1.2.2 — Join Entropy

Let $X = (X_1, X_2)$ be a pair of random variables. We define its entropy $H(X)$ as

$$H(X) = \sum_{x_1 \in S_{X_1}} \sum_{x_2 \in S_{X_2}} p(x_1, x_2) \log \frac{1}{p(x_1, x_2)} \tag{1.17}$$

where $p(x_1, x_2) = \mathbb{P}\left(X_1 = x_2, X_2 = x_2\right)$ is the PMF of X.

A very important concept is the *conditional entropy* of a random variable X, defined with regard to another random variable Y.

Definition 1.2.3 — Conditional Entropy

Given two random variables X and Y, the *conditional entropy of X given Y* is

$$H(X|Y) = \sum_{x \in S_X} p_X(x) H(Y|X = x) \tag{1.18}$$

$$= \sum_{x \in S_X} p_X(x) \sum_{y \in S_Y} p_Y(y|x) \log p_Y(y|x) \tag{1.19}$$

$$\tag{1.20}$$

where p_X and p_Y are the PMFs of X and Y, respectively[1].

[1] We take $p_Y(y|x)$ to be a shorthand notation for $\mathbb{P}\left(Y = y \mid X = x\right)$.

The two notions of joint entropy and conditional entropy are related in an important way, going under the name of *chain rule*.

Theorem 1.2.1

For any two random variables X and Y

$$H(X,Y) = H(X) + H(Y|X) \tag{1.21}$$

Proof. Denote $\mathbb{P}(X = x, Y = y)$ simply by $p(x,y)$, then

$$\begin{aligned}
H(X,Y) &= -\sum_{x\in S_X}\sum_{y\in S_Y} p(x,y)\log p(x,y) && \text{(By definition)}\\
&= -\sum_{x\in S_X}\sum_{y\in S_Y} p(x,y)\log\left(p_X(x)\cdot p_Y(y|x)\right) && (1.22)\\
&= -\sum_{x\in S_X}\sum_{y\in S_Y} p(x,y)\log p_X(x) - \sum_{x\in S_X}\sum_{y\in S_Y} p(x,y)\log p_Y(y|x) && (1.23)\\
&= -\sum_{x\in S_X} p_X(x)\log p_X(x) - \sum_{x\in S_X}\sum_{y\in S_Y} p(x,y)\log p_Y(y|x) && (1.24)\\
&= H(X) + H(Y|X) && (1.25)
\end{aligned}$$

□

Corollary 1.2.1

For any three random variables X, Y and Z

$$H(X,Y|Z) = H(X|Z) + H(Y|X,Z) \tag{1.26}$$

1.2.3 Relative Entropy and Mutual Information

We will briefly hint to a measure known as *mutual information*, defined between a pair of random variables. Informally, mutual information expresses the amount of information that one random variables conveys about another.

Before actually defining mutual information, we have to step through another quantity.

Definition 1.2.4 — Relative Entropy

Let X and Y be two random variables with PMFs p_X and p_Y. We define the relative entropy of X and Y as

$$D\left(p_X \,||\, p_Y\right) = \sum_{x \in S_X} p_X(x) \log \frac{p_X(x)}{p_Y(x)} \tag{1.27}$$

Note that p_Y is fed values $x \in S_X$, so that the denominator in (1.27) may equal 0. However, due to analytical considerations, it is assumed by convention that $\forall p, p \log \frac{p}{0} = \infty$, so that if $\exists x \in S_X | p_X(x) > 0 \wedge p_Y(x) = 0$, $D\left(p_X \,||\, p_Y\right) = \infty$.

Proposition 1.2.1

For any two random variables X and Y

1. $D\left(p_X \,||\, p_Y\right) \geq 0$
2. $D\left(p_X \,||\, p_Y\right) = 0$ if and only if $p_X = p_Y$

Definition 1.2.5 — Mutual Information

The *mutual information* $I(X;Y)$ between two random variables X and Y is the relative entropy of the joint distribution $p_{(X,Y)}$ of X and Y and the product distribution $p_X(x)p_Y(y)$

$$\begin{aligned} I(X;Y) &= \sum_{x \in S_X} \sum_{y \in S_Y} p_{(X,Y)}(x,y) \log \frac{p_{(X,Y)}(x,y)}{p_X(x)p_Y(y)} && (1.28) \\ &= D\left(p_{(X,Y)}(x,y) \,||\, p_X(x) \cdot p_Y(y)\right) && (1.29) \end{aligned}$$

The mutual information of two random variables can be expressed as

$$I(X;Y) = H(X) - H(X|Y) \tag{1.30}$$

In fact

$$\begin{aligned} I(X;Y) &= \sum_{x \in S_X} \sum_{x \in S_Y} p_{(X,Y)}(x,y) \log \frac{p_{(X,Y)}(x,y)}{p_X(x)p_Y(y)} && (1.31) \\ &= \sum_{x,y} p_{(X,Y)}(x,y) \log \frac{p_X(x|y)}{p_X(x)} && (1.32) \\ &= \sum_{x,y} p_{(X,Y)}(x,y) \log p_X(x|y) - \sum_{x,y} p_{(X,Y)} \log p_X(x) && (1.33) \\ &= -\left(\sum_{x,y} p_{(X,Y)}(x,y) \log p_X(x|y)\right) - \sum_{x,y} p_{(X,Y)} \log p_X(x) && (1.34) \\ &= -H(X|Y) + H(X) && (1.35) \end{aligned}$$

In this form, $I(X;Y)$ tells the amount of information of X we spare by observing Y. Note that, due to symmetry, it also holds that $I(X;Y) = H(Y) - H(Y|X)$.

Finally, since $I(X;X) = H(X) - H(X|X) = H(X)$, we get that the mutual information of a variable with itself is the entropy. For this reason, the entropy of a random variable is also called *self-information*.

1.3 Overview of Data Compression

We progress into our overview of information theory with a few words to data compression, an applied subfield of information theory, more pertinent to the topics of this study. Some of what we are going to introduce here will be considered again in Chapter 5.

Data compression is concerned with the design and analysis of compression algorithms. On a macro level, it can be subdivided into two major areas: lossless data compression, and lossy data compression.

By lossless compression, we refer to algorithms that compress their data without loss of information. If we were to revert the output of a lossless compression algorithm, we would reconstruct, bit-by-by, the exact input we started from.

Lossy data compression, on the other hand, compromises on the quality of its data, as long as this leads to higher compression ratios. That is, lossy data compression algorithms allow some loss of the unessential data (subtle traces of audio records, tiny details of pixel images etc.), in an attempt to reduce the space requirements of the input data even further.

While both approaches are important on their own, each has its area of applicability. For the purposes of our study, only lossless data compression will be relevant.

In the remainder of the section, we will provide some theory behind compression codes and describe some of the more important compression algorithms.

1.3.1 Models

When designing a compression algorithm, we sometimes construct a model for the data. One such model is the *probabilistic model*, where a probability distribution is used to reason about the data. When the assumption of a probabilistic model for the data isn't available a-priori, we can generate it on the basis of the input we see. For example, the Huffman coding assumes the existence of a probability distribution on the alphabet it works on; if this distribution isn't available, a construction can be used that determines it on the basis of the data that is received by the algorithm. This variation of Huffman coding is named adaptive Huffman coding.

An alternative way of modeling a data source is to use a *Markov model*, specifically a discrete time Markov chain.

Note, however, that not all algorithms have to make use of an explicit mathematical model. Dictionary-based techniques for text compression are

Letter	Code 1	Code 2	Code 3
a_0	0	0	0
a_1	1	1	10
a_2	0	00	110
a_3	01	10	111

TABLE 1.1: A four-letter alphabet, along with three possible codes.

an example, where data is seen like a series of repeated patterns, rather than the manifestation of a probabilistic outcome.

1.3.2 Codes

The algorithms we will be considering work by mapping the symbols of an input alphabet A to so-called *codewords*, strings built on top of an alphabet B, possibly equal to A itself. These algorithms, when scanning its input, associate a codeword to a single symbol $a \in A$, or to more symbols of A packed together. We call an association of codewords to strings of symbols $s \in A^*$ a *code*.

One remark concerns how we pack more codewords together to form the compressed output. We may use a separator character between each codeword, but this would be far from being an efficient method. In a data compression scenario, we strive to squeeze our output as much as possible. Instead, we will simply pack our codewords one next to the other, without any padding in the middle or any header information before the output. What is required is to be able to recognize a codeword on the basis of its symbols alone.

And now, for the actual codes. Consider Table 1.1. In that table, we included an alphabet of four symbols, along with three possible codes. We'll introduce some common code classifications on the basis of these examples. They are all variable-length codes, meaning that, if the probability of an individual symbol was higher than that of another, we may be able to achieve compression[2].

Let's start with Code 1. Code 1 is noticeably flawed: the same codeword, `0`, is mapped to two different symbols, a_0 and a_2. If we met a 0 in the output stream, we wouldn't be able to tell whether it came from a_0 or a_2. We call codes like Code 1 *ambiguous codes*.

We then have a look at Code 2. Although Code 2 is not ambiguous, it is not free of defects. Imagine we had to decode the string $\omega =$ `0100`. One way to decode this string would be to split it as `0 1 00`, and interpret its inverse image as $a_0 a_1 a_2$. However, if we split it as `0 10 0`, we would be getting the equally plausible inverse image $a_0 a_3 a_0$. This, clearly, is an undesirable situation to us. What we would like to have is a code that, unlike Code 2, admits

[2]We assume that the least advantageous way of compressing the $\{a_0, a_1, a_2, a_3\}$ alphabet is by assuming an uniform distribution, and using $\log_2 4 = 2$ bits per each symbol.

one and only one possible interpretation for each combination of its codewords. Such a code is termed *uniquely decodable code*. Code 2 is not uniquely decodable.

Let us finally take a look at Code 3. Clearly, this code is unambiguous. Further, it exhibits an important property: no codeword is the prefix of any other codeword. Codes with this property go under the name of *prefix codes*. Prefix codes are also *instantaneous codes*, that is codes whose codewords can be interpreted as soon as they are read, with no need to read the whole string they are embedded in. What we are most pressed to remark though is that, Code 3, like any other prefix code, is uniquely decodable. This is true because, no matter how we mix up the codewords together, the union of two codewords ω_1, ω_2 never produces a third distinct codeword ω_3. (If this did happen, then ω_1 would be a prefix for ω_3.) Being a uniquely decodable code, Code 3 is a viable code for the $\{a_0, a_1, a_2, a_3\}$ alphabet of Table 1.1.

On more abstract terms, we may wonder: if every prefix code is a uniquely decodable code, what can we say about uniquely decodable codes in general? Are there non-prefix, uniquely decodable codes that are shorter than some prefix code? Fortunately for us, this is not the case, as the following two statements demonstrate.

Theorem 1.3.1 — Kraft-McMillan Inequality

Let C be a uniquely decodable code with N codewords of length $l_1, l_2, \ldots, l_N$. Then

$$\sum_{i=1}^{N} 2^{l_i} \leq 1 \tag{1.36}$$

The relation in (1.36) is known as *Kraft-McMillan inequality*. Next we have

Theorem 1.3.2

For every set of integers $l_1, l_2, \ldots, l_N$ satisfying the inequality

$$\sum_{i=1}^{N} 2^{-l_i} \leq 1 \tag{1.37}$$

it is possible to find a prefix code whose codewords have lengths $l_1, l_2, \ldots, l_N$.

These two statements allow us to deduce the following fact: if we are given a uniquely decodable code C, then C satisfies the Kraft-McMillan inequality; and if it satisfies the Kraft-McMillan inequality, then a prefix code exists whose codewords have the same length as the codewords of C. Thus, we lose nothing when focusing on prefix codes.

Symbol	Probability
c	0.5
e	0.25
a	0.1
d	0.1
b	0.05

TABLE 1.2: Example probability distribution of a source alphabet Σ for the Huffman algorithm. The entropy of this distribution is 1.88 bits per symbol.

1.3.3 Compression Algorithms

In the next few paragraphs, we describe some of the more important compression algorithms. The purpose of this explanation is twofold. First, although limited to just a few examples, we would like to provide the reader with a sense of what a compression algorithm actually is. Second, we will need this explanation later in Section 5.1, when we consider the problem of logarithmic compression.

Huffman Coding The Huffman algorithm is an prefix code developed by David A. Huffman in 1952 [Huf52], that assumes a probability distribution over a source alphabet Σ. For our own purposes, we will assume the algorithm to work on the binary alphabet $\{0,1\}$ as a codomain, so that, $\forall s \in \Sigma^*, C(s) \in \{0,1\}^*$, where C denotes the application of the Huffman encoding.

The central idea of the method is to construct a mapping for each symbol $\sigma \in \Sigma$ to a sequence of bit strings in $\{0,1\}^*$, so that, if the probability of a symbol $\sigma_1 \in \Sigma$ is higher than that of another symbol $\sigma_2 \in \Sigma$, then the codeword $C(\sigma_1)$ for σ_1 is shorter than the one for σ_2. In other terms

$$\mathbb{P}(\sigma_1) > \mathbb{P}(\sigma_2) \implies |C(\sigma_1)| < |C(\sigma_2)| \tag{1.38}$$

To simplify understanding, we provide a small example for the construction of a Huffman code for the alphabet $\Sigma = \{\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}, \mathrm{e}\}$. There are various ways to depict the construction of a Huffman code. The one we utilise relies on a binary tree.

Consider Table 1.2, where the symbols from Σ have been sorted according to their own probability $\mathbb{P}(\sigma)$. We build the binary tree starting from the symbols with the lowest probability. This tree satisfies the following properties

- Every edge connecting two nodes is labeled with either 0 or 1
- Each symbol $\sigma \in \Sigma$ is represented by one and only one leaf node
- Internal nodes of the tree correspond to no symbols from Σ

The tree building procedure, whose complete formal steps we omit for brevity, applied to the probability distribution of Table 1.2, results in the tree

Symbol	Codeword
c	0
e	10
a	110
d	1111
b	1110

TABLE 1.3: Codewords for the alphabet Σ of Table 1.2.

of Figure 1.2. The codewords thus obtained for the Σ alphabet are shown in Table 1.3.

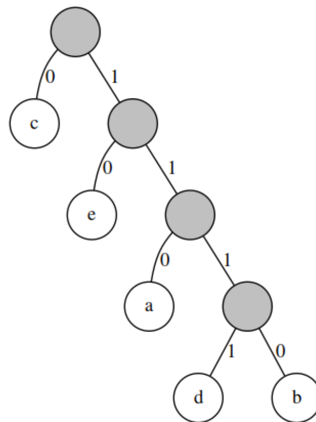


FIGURE 1.2: Binary tree representing the Huffman encoding of the alphabet of Table 1.2.

With a mapping between symbols of Σ and binary codewords, we can translate a string $s \in \Sigma$ into one from $\{0,1\}^*$. Assume, for example, that $s =$ cedcea. Then

$$C(s) = 0 \cdot 10 \cdot 1111 \cdot 0 \cdot 10 \cdot 110 \tag{1.39}$$

where $\cdot$ is just a visual aid with no implications for the representation of $C(s)$.

Recall that the entropy of the probability distribution of Table 1.2 is 1.88 bits per symbol, meaning that, for a string of 6 characters, we would need, on average, $6 \times 1.88 = 11.28$ bits for transmitting it, provided we had an optimal encoding method.

While, in general, Huffman coding is not perfectly optimal with respect to the theoretical entropy of a source, it performs pretty close to it.

For instance, if we were to represent Σ with a uniform encoding method, using $\lceil \log_2 5 \rceil = 3$ bits per symbol, we would need $6 \times 3 = 18$ bits to transmit s. With Huffman encoding, instead, we only need 13, since $|C(s)| = 13$.

In general, the higher the number of symbols in s with high probability, the higher also the space saving of its Huffman-encoded version $C(s)$.

LZ77 LZ77 is a classical lossless textual compression algorithm, introduced by Abraham Lempel and Jacob Ziv in 1977 [ZL77]. Together with LZ78 [ZL78], published in 1978, they form the basis for a series of modifications, such as LZW, LZMA, LZSS, introduced by other researchers throughout the years.

LZ77 is a dictionary coder. Dictionary coders are compression algorithms that translate strings of symbols from an input string into "keywords" stored within the dictionary as a mapping for these sequences. If the content of the dictionary is not allowed to change during the execution, then we are dealing with a static dictionary coder, otherwise, just as with LZ77 and LZ78, we are dealing with a dynamic one.

To describe the functioning of LZ77, we start from the underlying idea. Take a string of text $s = s_1 s_2 \dots s_n$, built from a series of symbols from an alphabet Σ. Then we can split s into two halves s' and s'', such that $s = s's''$. s' is considered to be the part of the input stream that is already compressed (also referred as *sliding window* in the literature), while s'' the part of the input that is to be yet compressed. In compressing s'', we take a look back at s', to see whether some string of symbols $k \in \Sigma^*$ is occurring there that also occurs in s''. If such a k exists, then we take the longest, and output its location in s', plus its length in characters, to the output stream. In addition, the character $c \in \Sigma$ occurring right after k in s'' is also output in the output stream[3].

This approach is effective the longer k is in characters, since its compressed representation (that is, the triple of values indicating the position of k in s', its length and the character c) is constant in size. However, other factors also determine the effectiveness of LZ77. To understand them, we have to take a look at the fuller picture of LZ77.

The crucial bit of information that we have omitted from this informal explanation is that LZ77 has limited buffer, that is a limited memory capacity, and as a consequence also a limited view into both s' and s''. Let B be the number of symbols that LZ77 can hold in its buffer, and F the number of symbols that it is allowed to read from s''. The number of symbols readable from s' will consequently be $B - F$. Therefore, the performance of LZ77 depends not only on the concrete input string it is run with, but also on the parameters B and F it is configured with. The larger these values, the greater the length of a possible k will be allowed to be (although, then, running times will also tend to increase).

This tuning aspect is important, and will be considered again in Section 5.1, when we study LZ77 under the problem of logarithmic compression.

Another relevant aspect we want to mention is the size, in bits, of the triple output by the algorithm. The size of the triple is a function of the parameters of LZ77, and therefore constant with respect to the input string s. It is expressed as

$$d = m + \lceil \log (B - F) \rceil + \lceil \log F \rceil \tag{1.40}$$

[3] This is true in the pure LZ77 version. Other variations, such as LZSS, allow this character c to be omitted when it is possible to save some space.

where $m > 0$ is the number of bits needed to represent a character $c \in \Sigma$ ($m = \lceil \log |\Sigma| \rceil$ if Σ is given a uniform memorization scheme), $\lceil \log (B - F) \rceil$ is the number of bits needed to express the position of k in s' (as a relative offset starting at s'' and going backwards for up to $B - F$ symbols) and $\lceil \log F \rceil$ the number of bits needed to encode the length of k.

If LZ77 parses its input string s into N distinct triples, the size of its output will therefore be dN.

Chapter 2

Introduction to Spines

In the previous chapter, we introduced many of the concepts of information theory and data compression that will be relevant during the course of the work. This chapter is also an introduction, but to spines, and to the few other technical concepts that we will directly make use of, such as the XBW transform for trees from [Fer+09]. After these pages, we are going to deal with the problem of spine detection.

2.1 Spine Trees

There is not an univocal definition of spine tree. In fact, there are two, a *left spine tree* and *right spine tree* one. However, their difference is minimal, in that one is symmetrical to the other. Therefore, it only suffices to formally state what a left spine tree is.

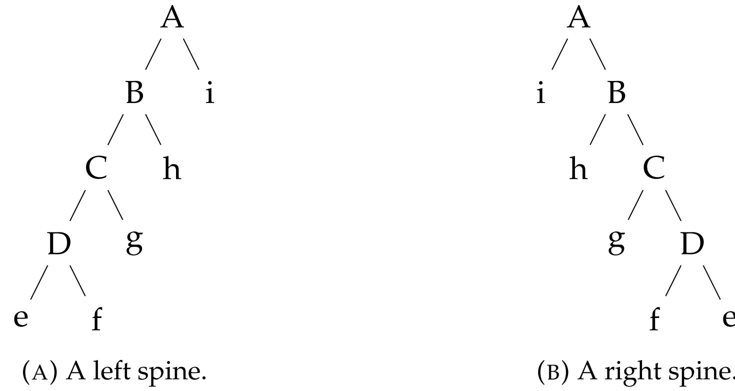
Definition 2.1.1 — Left Spine Tree

A *left spine tree*, or simply *left spine*, is an ordered and labeled tree such that

- The root node, r, has two child nodes c_1 and c_2, with c_1 preceding c_2
- c_2 is a leaf node
- c_1 is either the root of a left spine, or a leaf node

To get the definition of a right spine tree, one only needs to invert the roles of the c_1 and c_2 nodes. We give a graphical example of two spine trees, a left and a right one, in Figure 2.1.

In dealing with the detection and storage of spines, we mainly rely on the work of [Fer+09], where the XBW transform, a highly-compressible vectorized representation of labeled trees, is discussed. The study of [Fer+09] is useful to us because it reduces the problem of the compression of a labeled tree into the compression of a pair of strings. We will tweak the concept of the XBW transform to suit our needs by introducing the XBW^- transform, a type of XBW transform whose final, ordering step hasn't yet been performed.



(A) A left spine.

(B) A right spine.

FIGURE 2.1: Two spine trees.

2.2 Basic Concepts and Notation

2.2.1 Mathematical Notation

Integer Interval We write $[i, j]$ for the set of all integers between i and j, that is $[i, j] = \{x \in \mathbb{Z} \mid i \leq x \leq j\}$. In those rare cases—if any—where we want to designate the real-valued interval, we will write $[i, j]_{\mathbb{R}}$, and more generally, $\forall Q \subseteq \mathbb{R}$, we let $[i, j]_Q = \{x \in Q \mid i \leq x \leq j\}$.

Congruence Modulo n We will denote the congruence modulo n of two numbers a and b as $a \equiv_n b$. This is a departure from the more common notation $a = b \mod n$ of some authors, but we prefer it for its succinctness.

Internal and Leaf Node Labels As in [Fer+09], we adopt the convention of representing the internal nodes of a labeled tree with values drawn from an alphabet Σ_N, and leaf nodes with values drawn from an alphabet Σ_L, so that $\Sigma = \Sigma_N \cup \Sigma_L$. This convention only affects the presentational aspect. Internally, an additional bit for each node may be used to distinguish internal nodes from leaf ones.

Vector Indexing Let V be a vector. We write $V[i]$ for the ith element of V, and $V[i, j]$ for all elements of V such that their index $k \in [i, j]$.

2.2.2 XBW Transform

In this section, we summarize some of the key aspects of the XBW transform. If interested to get the fuller picture, please refer to [Fer+09].

Take an ordered and labeled tree T, of arbitrary shape and depth. In order to define the XBW transform $\text{xbw}[T]$ of T, we need the following elements. Let u be any node of T, then we consider

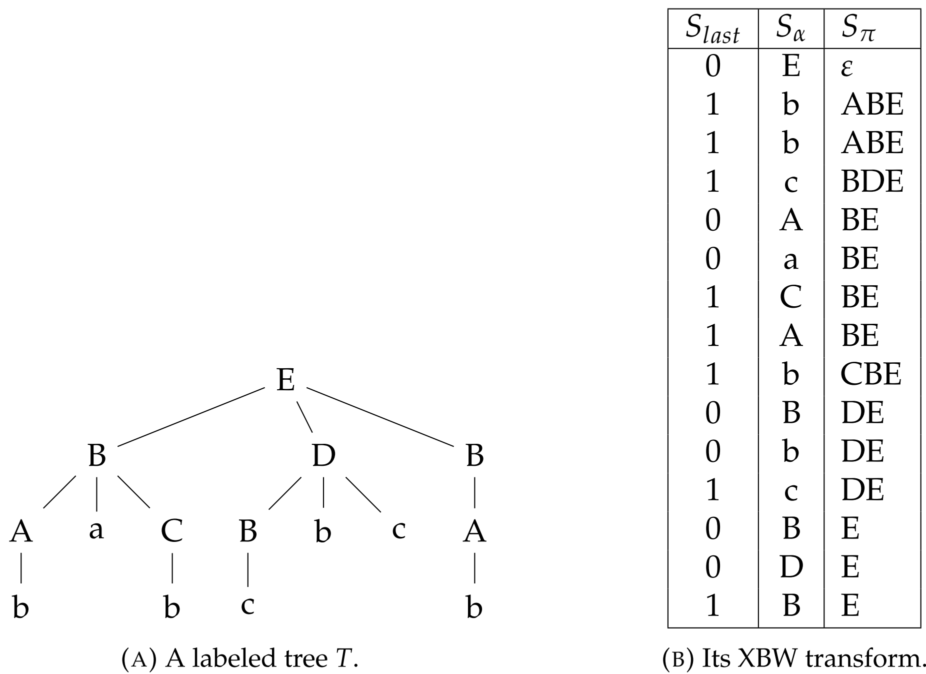
- $\text{last}[u]$, a binary value equal to 1 if u is the rightmost (i.e. last) child of its parent

- $\alpha[u]$, the label, or value, associated to u
- $\pi[u]$, the string obtained by concatenating all the labels from u's parent up to the root. If u is the root node, $\pi[u] = \varepsilon$, the empty string

The XBW transform of T consists then of the multi-set S of t triplets $\langle \text{last}[u], \alpha[u], \pi[u] \rangle$, one for each node of T. The transform is constructed according to the following procedure

1. Initially, set $S = \emptyset$
2. Perform a pre-order visit of T
3. For each visited node u, store $\langle \text{last}[u], \alpha[u], \pi[u] \rangle$ into S
4. Stably sort S with respect to the lexicographic order of $\pi[u]$

To simplify understanding, Figure 2.2 depicts a labeled tree T, along with its XBW transform xbw$[T]$.



S_{last}	S_α	S_π
0	E	ε
1	b	ABE
1	b	ABE
1	c	BDE
0	A	BE
0	a	BE
1	C	BE
1	A	BE
1	b	CBE
0	B	DE
0	b	DE
1	c	DE
0	B	E
0	D	E
1	B	E

(A) A labeled tree T.

(B) Its XBW transform.

FIGURE 2.2: An example XBW transform.

Given xbw$[T]$, we sometimes identify each node $u \in T$ with an integer i, that is $u = S[i]$. In such a case, we denote last$[u]$, $\alpha[u]$ and $\pi[u]$ with $S_{last}[i]$, $S_\alpha[i]$ and $S_\pi[i]$ respectively, or even last$_i$, α_i and π_i for the sake of brevity.

Note that the vector $\langle S_{last}, S_\alpha \rangle$, consisting only of S_{last} and S_α, is sufficient for recovering the whole structure of T. In fact, S_π does not need to be stored in a compressed representation of xbw$[T]$, since it can be reconstructed during the decompression process.

[Fer+09] discusses different results concerning the XBW transform on trees. We report a few of the more important ones, omitting their proof.

Theorem 2.2.1

Let T be a labeled tree with t nodes and labels drawn from an alphabet Σ. Then the transform $\mathrm{xbw}[T]$ can be computed in $O(t)$ time and $O(t \log t)$ bits of working space.

Theorem 2.2.2

A labeled tree T of t nodes can be reconstructed from its XBW transform $\mathrm{xbw}[T]$ in optimal $O(t)$ time and $O(t \log t)$ bits of working space.

The XBW transform is interesting for the task of tree compression because of the locality principle. The locality principle for a string $s \in \Sigma^*$ states that the surrounding elements of a value in s closely depend from the predecessor and successor values. Due to the sorting step of $\mathrm{xbw}[T]$, S_α satisfies the locality principle and can be compressed efficiently by BWT-based compressors [BW94; Fer+05].

Chapter 3

Spine Detection

Spine detection—the problem of locating a spine tree within a labeled tree of arbitrary shape—is the first actual problem we will be facing. Even though it is not the main objective of our work, it nonetheless forms a very important part of it.

We will be building our solution for spine detection by means of the XBW transform on trees, that we introduced in the earlier chapter. Although a detection algorithm for spines could be easily implemented in any procedural programming language through an ADT (Abstract Data Type) for trees, doing so wouldn't necessarily result in an efficient detection procedure. In fact, the use of an ADT would not make any guarantee on the underlying data structure utilized in the representation of the tree. This data structure might be efficient or inefficient for our purposes, depending on the case. To ensure an efficient implementation, we must choose ourselves this data structure, and the one we chose is, as said, the XBW transform.

The use of the XBW transform has two primary known advantages. First, since it consists of nothing more than a pair of vectors stored in a contiguous memory location, navigating the tree that it represents is fast, also thanks to many performance optimizations available for the different programming languages. Second, due to the same reason, the whole cache-memory hierarchy is guaranteed to work more smoothly, as opposed to, say, a pointer-based implementation of a tree, since the memory content of the tree is not dislocated in distant areas of central memory, but is concentrated in a single memory block.

3.1 XBW$^-$ Transform

What we have just said in the introduction to this chapter was only true in part. In fact, rather than working with an ordinary XBW transform, we will be mostly relying on a derived concept of it, the *XBW$^-$ transform*.

The XBW$^-$ transform is a derived concept from the ordinary XBW transform that we introduce for our own purposes. When applied to left and right spine trees, it encodes peculiar patterns of these within the tree they are embedded in, allowing for a very fast—in fact, optimal—identification of spines. We will introduce the concept of the XBW$^-$ transform informally, and then proceed to formalize it mathematically.

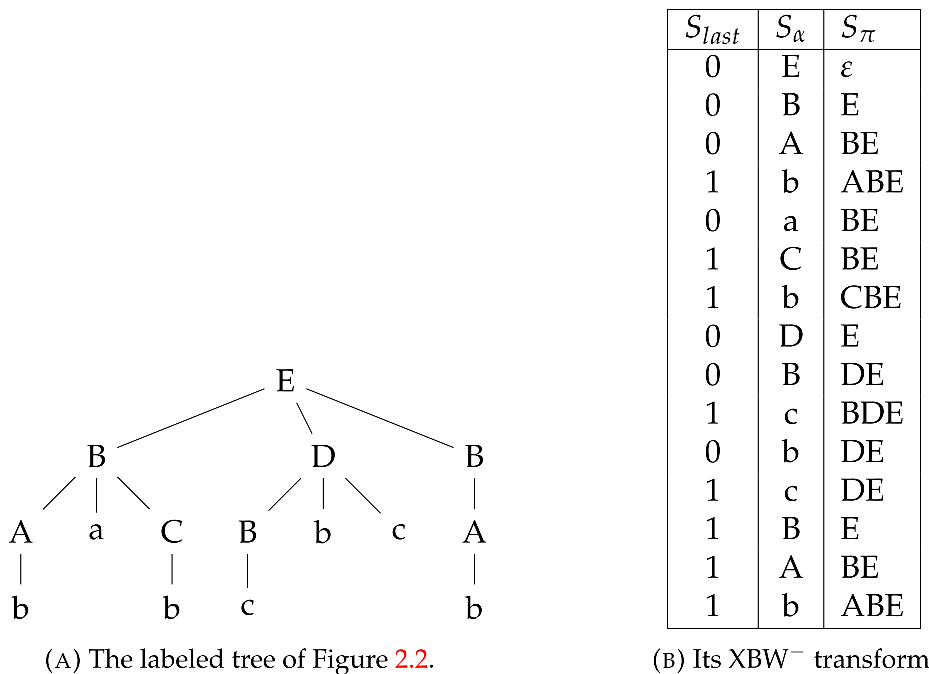
Definition 3.1.1 — XBW$^-$ Transform

An *XBW$^-$ transform* of a tree T, denoted by xbw$^-[T]$, is the XBW transform of T deprived of the sorting step with respect to S_π.

Observation 3.1.1

Computing an XBW$^-$ transform demands no computational overhead, if we expect to compute the full XBW transform at a later time.

In Figure 3.1, we give the example of an XBW$^-$ transform, taken from the tree of Figure 2.2.



S_{last}	S_α	S_π
0	E	ε
0	B	E
0	A	BE
1	b	ABE
0	a	BE
1	C	BE
1	b	CBE
0	D	E
0	B	DE
1	c	BDE
0	b	DE
1	c	DE
1	B	E
1	A	BE
1	b	ABE

(A) The labeled tree of Figure 2.2.

(B) Its XBW$^-$ transform.

FIGURE 3.1: An example XBW$^-$ transform.

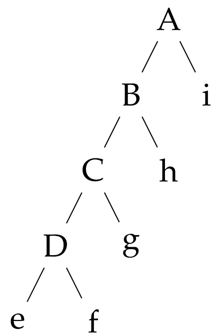
Observation 3.1.2

Since no sorting is performed, the order of the entries in an XBW$^-$ transform reflects that of a pre-order visit.

3.2 Informal Detection Description

The idea behind the identification of left and right spines is inspecting the S_{last} and S_π subvectors of the XBW$^-$ transform. Consider, by way of example, the left spine of Figure 2.1a, along with its XBW$^-$ transform, that we show in Figure 3.2.

Clearly, some visible patterns emerge from Figure 3.2b. Let $h = 4$ be the height of the spine. Then S_{last} exposes a series of h contiguous 0s, followed by



(A) The left spine of Figure 2.1a.

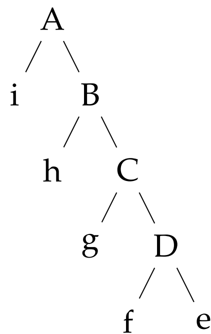
S_{last}	S_α	S_π
0	A	ε
0	B	A
0	C	BA
0	D	CBA
0	e	DCBA
1	f	DCBA
1	g	CBA
1	h	BA
1	i	A

(B) Its XBW$^-$ transform.

FIGURE 3.2: The left spine of Figure 2.1a along with its XBW$^-$ transform.

another consecutive h 1s, while S_π is characterized by a kind of "symmetry" for all the nodes except the root.

Albeit dissimilar, even right spines display their own distinctive pattern. Consider Figure 3.3 for a reference. In this case, an alternating series of 0s and 1s takes place of the contiguous series of 0s and 1s found in the left-spine case. And S_π is characterized by a sort of "increasing monotonicity", rather than a symmetry.



(A) The right spine of Figure 2.1b.

S_{last}	S_α	S_π
0	A	ε
0	i	A
1	B	A
0	h	BA
1	C	BA
0	g	CBA
1	D	CBA
0	f	DCBA
1	e	DCBA

(B) Its XBW$^-$ transform.

FIGURE 3.3: The right spine of Figure 2.1b along with its XBW$^-$ transform.

Having introduced informally the detection of left and right spines by means of their XBW$^-$ transform, we make our discourse more precise by stating some propositions in the mathematical language.

3.3 Properties of XBW$^-$ Transforms on Spines

In some of the following statements, we will make use of the `is_leaf(i)` predicate, evaluating to true when the node indexed i in the tree T under consideration is a leaf node, and to false otherwise. Although it is not our

concern here, it is possible to offer a constant-time implementation of the predicate, so that its use does not result in any performance penalty.

Proposition 3.3.1

Let S be a spine tree, and $|S|$ the number of its nodes. Then $|S| = 2h + 1$, where h is the height of S.

Proposition 3.3.2

Let S be a spine of height $h \geq 1$ rooted under a tree T. Enumerate the nodes of T in a pre-order visit fashion, and let a and b be the indices of the first and last elements of S, respectively. Then

$$b - a = 2h \tag{3.1}$$

Lemma 3.3.1 — Left Spine to XBW$^-$ Transform

Let L be a left spine of height $h \geq 1$, rooted in a tree T possibly equal to L itself. Also let a be the index of the root node of L, $b = a + 2h$ the index of the last node of L and S the XBW$^-$ transform of T. Then we have that

$$S_{last}[i] = \begin{cases} 0, & a+1 \leq i \leq a+h \\ 1, & a+h+1 \leq i \leq b \end{cases} \tag{3.2}$$

$$S_\pi[i] = \begin{cases} S_\alpha[i-1]S_\pi[i-1], & a+1 \leq i \leq a+h \\ S_\pi[k-i], & a+h+1 \leq i \leq b \end{cases} \tag{3.3}$$

where $k = 2(a+h)+1$.

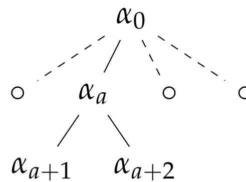


FIGURE 3.4: A tree T with a left spine of height $h = 1$.

Proof. We proceed by induction on h.

Base Case We have $h = 1$, and the tree T is shown in Figure 3.4. Clearly, nodes of T whose index i is such that $i < a$ or $a + 2 < i$ have no impact on $S[a, a+2]$, the portion of XBW$^-$ transform whose entries are indexed from a to $a + 2$.
From Table 3.1, where we represent $S[a, a+2]$, it is obvious that

i	S_{last}	S_α	S_π
0	0	α_0	ε
1	?	?	α_0
$\vdots$	$\vdots$	$\vdots$	$\vdots$
a	?	α_a	π_a
$a+1$	0	α_{a+1}	$\alpha_a\pi_a$
$b=a+2$	1	α_{a+2}	$\alpha_a\pi_a$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\|T\|-1$	?	?	?

TABLE 3.1: The $S[a, a+2]$ portion of the XBW$^-$ transform S.

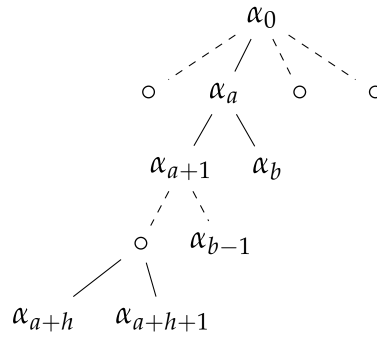


FIGURE 3.5: A tree T with a left spine of height $h > 1$.

$$S_{last}[a+1] = 0 \tag{3.4}$$
$$S_{last}[a+2] = 1 \tag{3.5}$$
$$S_\pi[a+1] = S_\pi[a+2] = S_\alpha[a]S_\pi[a] \tag{3.6}$$

Inductive Step In this case, $h > 1$, and a left spine of height $h' = h - 1$ is rooted at node $a+1$ (Figure 3.5).
Therefore, by the inductive hypothesis

$$S_{last}[i] = \begin{cases} 0, & a+2 \leq i \leq a+1+h' \\ 1, & a+2+h' \leq i \leq b-1 \end{cases} \tag{3.7}$$

$$S_\pi[i] = \begin{cases} S_\alpha[i-1]S_\pi[i-1], & a+2 \leq i \leq a+1+h' \\ S_\pi[k-i], & a+2+h' \leq i \leq b-1 \end{cases} \tag{3.8}$$

with $k = 2(a+1+h') = 2(a+h)+1$. We finally have to consider nodes $a+1$ and b. It is evident that $S_{last}[a+1] = 0$, $S_{last}[b] = 1$ and $S_\pi[a+1] = S_\pi[b] = S_\alpha[a]S_\pi[a]$. The inductive hypothesis and this last point lead us to

$$S_{last}[i] = \begin{cases} 0, & i = a+1 \vee a+2 \leq i \leq a+1+h' \\ 1, & i = b \vee a+2+h' \leq i \leq a+1+2h' \end{cases} \tag{3.9}$$

$$= \begin{cases} 0, & a+1 \leq i \leq a+h \\ 1, & a+1+h \leq i \leq b \end{cases} \tag{3.10}$$

$$S_\pi[i] = \begin{cases} S_\alpha[i-1]S_\pi[i-1], & i = a+1 \vee a+2 \leq i \leq a+h \\ S_\pi[k-i], & i = b \vee a+2+h \leq i \leq b-1 \end{cases} \tag{3.11}$$

$$= \begin{cases} S_\alpha[i-1]S_\pi[i-1], & a+1 \leq i \leq a+h \\ S_\pi[k-i], & a+1+h \leq i \leq b \end{cases} \tag{3.12}$$

□

Lemma 3.3.2 — XBW^- Transform to Left Spine

Let S be an XBW^- transform of length n. Then if $\exists a, b \mid b - a = 2h$, with $h \geq 1$, conditions (3.2) and (3.3) hold and $\forall i \mid a+h \leq i \leq b$

$$\text{is_leaf}(i) \tag{3.13}$$

the subtree U encoded by the subportion $S[a,b]$ of S is a left spine of height h.

i	S_{last}	S_α	S_π
0	0	α_0	ε
1	?	?	α_0
⋮	⋮	⋮	⋮
a	?	α_a	π_a
$a+1$	0	α_{a+1}	$\alpha_a \pi_a$
$b = a+2$	1	α_{a+2}	$\alpha_a \pi_a$
⋮	⋮	⋮	⋮
$n-1$	?	?	?

TABLE 3.2: The $S[a, a+2]$ portion of the XBW^- transform S.

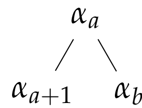


FIGURE 3.6: A left spine of height $h = 1$.

Proof. By induction on h.

Base Case If $h = 1$, then we have the XBW^- transform from Table 3.2

i	S_{last}	S_α	S_π
0	0	α_0	ε
1	?	?	α_0
$\vdots$	$\vdots$	$\vdots$	$\vdots$
a	?	α_a	π_a
$a+1$	0	α_{a+1}	$\alpha_a\pi_a$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$a+h$	0	?	$\alpha_{a+h-1}\pi_{a+h-1}$
$a+h+1$	1	?	$\alpha_{a+h-1}\pi_{a+h-1}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$b-1$	1	?	$\alpha_{a+1}\pi_{a+1}$
b	1	?	$\alpha_a\pi_a$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$n-1$	?	?	?

TABLE 3.3: The $S[a, a+2]$ portion of the XBW$^-$ transform S.

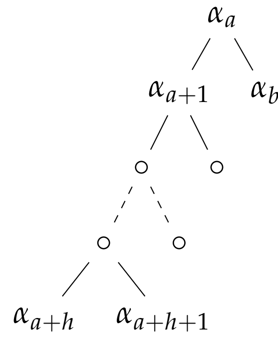


FIGURE 3.7: A left spine of height $h = 1$.

Given a generic node indexed i, its parent node has index j, where

$$j = \max\{j' \mid 0 \le j' < i \wedge \pi_i = \alpha_j\pi_j\} \tag{3.14}$$

Nodes $a+1$ and b have node a as their parent, since $\pi_{a+1} = \pi_b = \alpha_a\pi_a$, $a < a+1 < b$ and a is the greatest index satisfying these conditions. Furthermore, $a+1$ and b are the only child nodes of a (because $S_{last}[b] = 1$). To complete the base case, consider nodes i such that $i < a$. These nodes can have no influence over the structure of the subtree rooted at a. However, it may happen that nodes $i \mid b < i$, are descendants of b. This case is ruled out by the fact that is_leaf(b). Therefore the structure of the subtree rooted at a matches that of Figure 3.6

Inductive Step In this case, we have $b - a = 2h$, with $h > 1$, and $S[a, b]$ satisfies conditions (3.2), (3.3) and (3.13). Of course, these conditions still hold for $a' = a+1$, $b' = b-1$, and since $b' - a' = a - b - 2 = 2(h-1) = 2h'$, we can apply the inductive hypothesis to (3.2) and claim that $S[a', b']$ encodes a left spine of height h'.

To conclude the proof, observe nodes $a+1$ and b. By a reasoning analogous to that for the base case, $a+1$ and b are the only children of node a, and so they form the subtree of Figure 3.7
Since node $a+1$ is the root of a left spine of height h', a is the root of a left spine of height $h'+1 = h$. □

Lemma 3.3.3 — Right Spine to XBW$^-$ Transform

Let R be a right spine of height $h \geq 1$, rooted in a tree T possibly equal to R itself. Also let a be the index of the root node of R, b the index of the last node of R and S the XBW$^-$ transform of T. Then we have that $\forall i \mid a+1 \leq i \leq a+2h = b$

$$S_{last}[i] = \begin{cases} 0, & i \not\equiv_2 a \\ 1, & i \equiv_2 a \end{cases} \tag{3.15}$$

$$S_\pi[i] = \begin{cases} S_\alpha[i-1]S_\pi[i-1], & i \not\equiv_2 a \\ S_\pi[i-1], & i \equiv_2 a \end{cases} \tag{3.16}$$

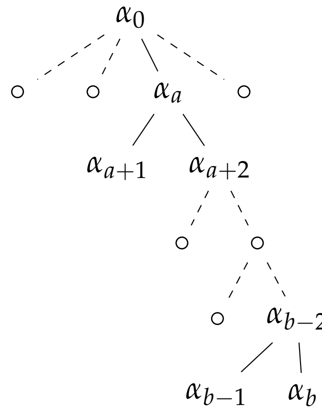


FIGURE 3.8: A tree T with a right spine of height $h > 1$.

Proof. By induction on h.

Base Case For $h = 1$, we assume to have a tree and an XBW$^-$ transform identical to those of Figure 3.4 and Table 3.1, respectively. Verifying equations (3.15) and (3.16) is thus straightforward.

Inductive Step For the tree of Figure 3.8, it is easy to observe that the subtree rooted at $a+2$ is a right spine of height $h' = h-1$.
By the inductive hypothesis, we have that for all i such that

$$a' + 1 \leq i \leq a' + 2h' \quad (3.17)$$
$$a' + 1 \leq i \leq a + 2 + 2h - 2 \quad (3.18)$$
$$a + 3 \leq i \leq a + 2h = b \quad (3.19)$$

conditions (3.15) and (3.16) hold. By an easy check, they are verified even for nodes $a+1$ and $a+2$, and therefore for every node i such that $a+1 \leq i \leq a+2h$. □

Lemma 3.3.4 — XBW$^-$ Transform to Right Spine

Let S be an XBW$^-$ transform of length n. Then if $\exists a, b \mid b - a = 2h$, with $h \geq 1$, conditions (3.15) and (3.16) hold and also

$$\text{is_leaf}(b) \quad (3.20)$$

the subtree U encoded by the subportion $S[a, b]$ of S is a right spine of height h.

i	S_{last}	S_α	S_π
0	0	α_0	ε
1	?	?	α_0
$\vdots$	$\vdots$	$\vdots$	$\vdots$
a	?	α_a	π_a
$a+1$	0	?	$\alpha_a \pi_a$
$a+2$	1	?	$\alpha_a \pi_a$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$b-1$	0	?	$\alpha_{b-2}\pi_{b-2}$
b	1	?	$\alpha_{b-2}\pi_{b-2}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$n-1$	?	?	?

TABLE 3.4: The $S[a, a+2]$ portion of the XBW$^-$ transform S.

Proof. By induction on h.

Base Case For a right spine of height $h = 1$, we have an XBW$^-$ transform portion identical to that of Table 3.2. The reasoning follows similarly for the proof of the left-spine case: nodes a, $a+1$ and $a+2 = b$ form a potential right spine of height $h = 1$. For it to be effectively so, it must be is_leaf(b), which is true by hypothesis.

Inductive Step Table 3.4 shows the XBW$^-$ transform for the inductive step. The $S[a, b]$ portion of S satisfies properties (3.15), (3.16) and (3.20), with $b - a = 2h$ and $h > 1$. These properties still hold for the portion

$S[a', b']$ of S, where $a' = a + 2$ and $b' = b$. In this case, $b' - a' = 2h'$ and $h' = h - 1$, so we can apply the inductive hypothesis to the subtree rooted in a' and conclude that it represents a right spine of height h'.
If we finally consider entries $a + 1$ and $a + 2$ we immediately notice that, with a reasoning similar to that of previous proofs, they represent two nodes whose parent is a, and that they are the only children. Since the subtree rooted at $a + 2$ is a right spine of height $h - 1$, the subtree rooted at a is a right spine of height h. □

This marks the end of the formal statement of properties of the XBW$^-$ transform on spines. With these conceptual tools at our disposal, we can introduce and justify a detection algorithm for spines, the main objective of this chapter.

3.4 An Algorithm for Spine Detection

Theorem 3.4.1

Let T be a tree of arbitrary shape, and S its XBW$^-$ transform. Then an algorithm exists that is able to detect all left and right spines of T in $\Theta(|T|)$ time and $\Theta(1)$ space.

```
1  function spines (automaton)
2      spines = ∅
3
4      while automaton.has_next()
5              automaton.next()
6
7              if automaton.state == ACCEPTED
8                      spines = spines ∪ automaton.spine_range()
9                      automaton.reset_state()
10
11     return spines
```

ALGORITHM 1: Spine detection algorithm.

Proof. The proof is constructive, and deploys an algorithm that performs the detection of all left and right spines of T with a single scan of S in constant space. We give the pseudocode of this algorithm in Algorithm 1. The algorithm receives as input a particular finite-state automaton, and returns `spines`, the set of all (a, b) pairs such that a and b are the first- and last-node index of a spine in T, respectively. The finite-state automaton that is fed into the algorithm does not define a language, but maintains the state of the current scan of the vector S. It comes in two varieties, a left and a right one.

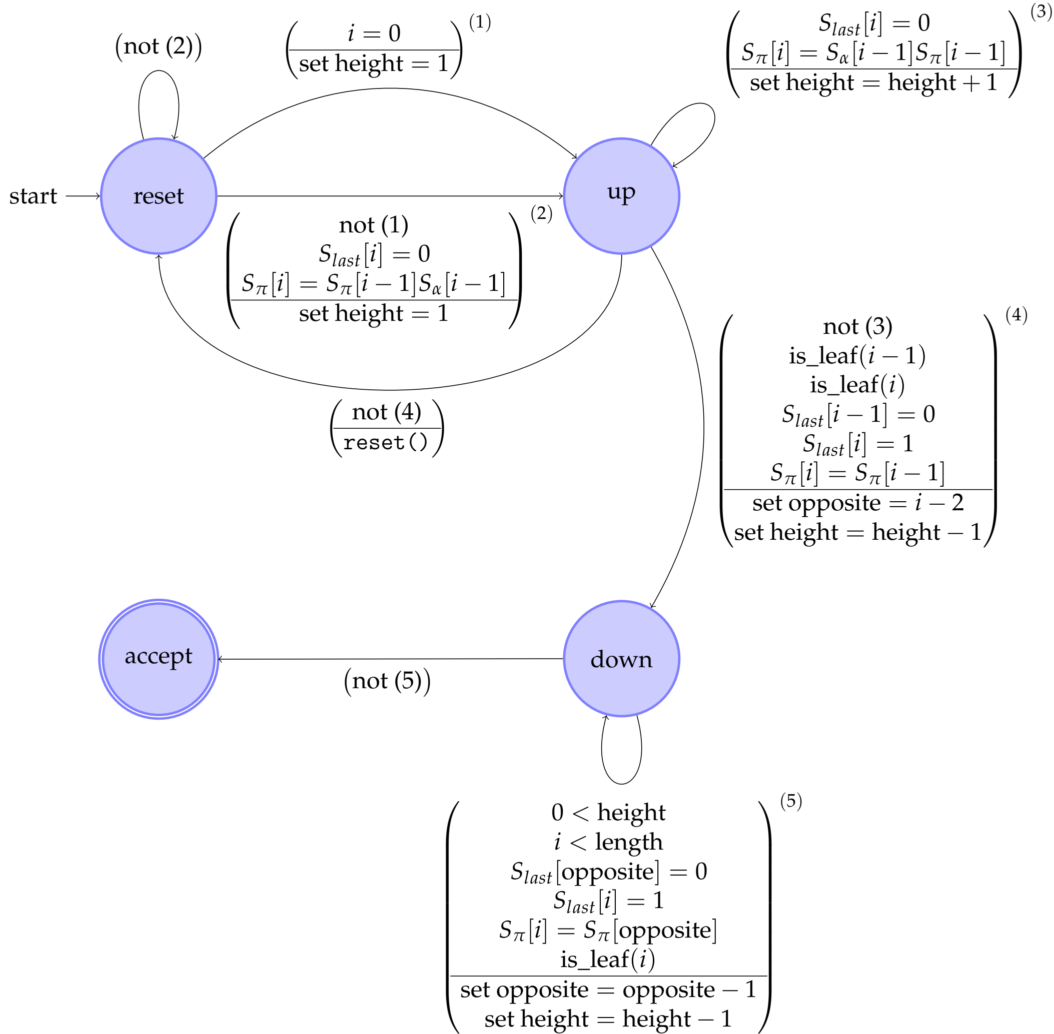


FIGURE 3.9: The left-spine automaton.

This automaton is initialized with the vector S and is stepped forward by the `next()` operation. The `has_next()` operation returns `true` as long as the automaton hasn't finished iterating on S. Whenever the automaton detects a new spine, `automaton.state` equals the special valute `ACCEPTED`, and the (a, b) pair of this spine can be obtained by the `spine_range()` operation. Algorithm 1 simply manipulates this automaton until it comes to the end of vector S.

To explain the left and right automata in more detail, consider Figure 3.9 and Figure 3.10. The graphical notation can be interpreted as follows: each node represents a state in the ordinary sense of finite-state automata, but each edge connects two nodes a and b if and only if the conditions enclosed by the parentheses are satisfied when the state is a. Sometimes, a horizontal rule separates this list of conditions from a set of instructions, that are executed on the transition from a to b. The aggregate of conditions is identified, when necessary, by an (x) at the top-right corner, where x is a number.

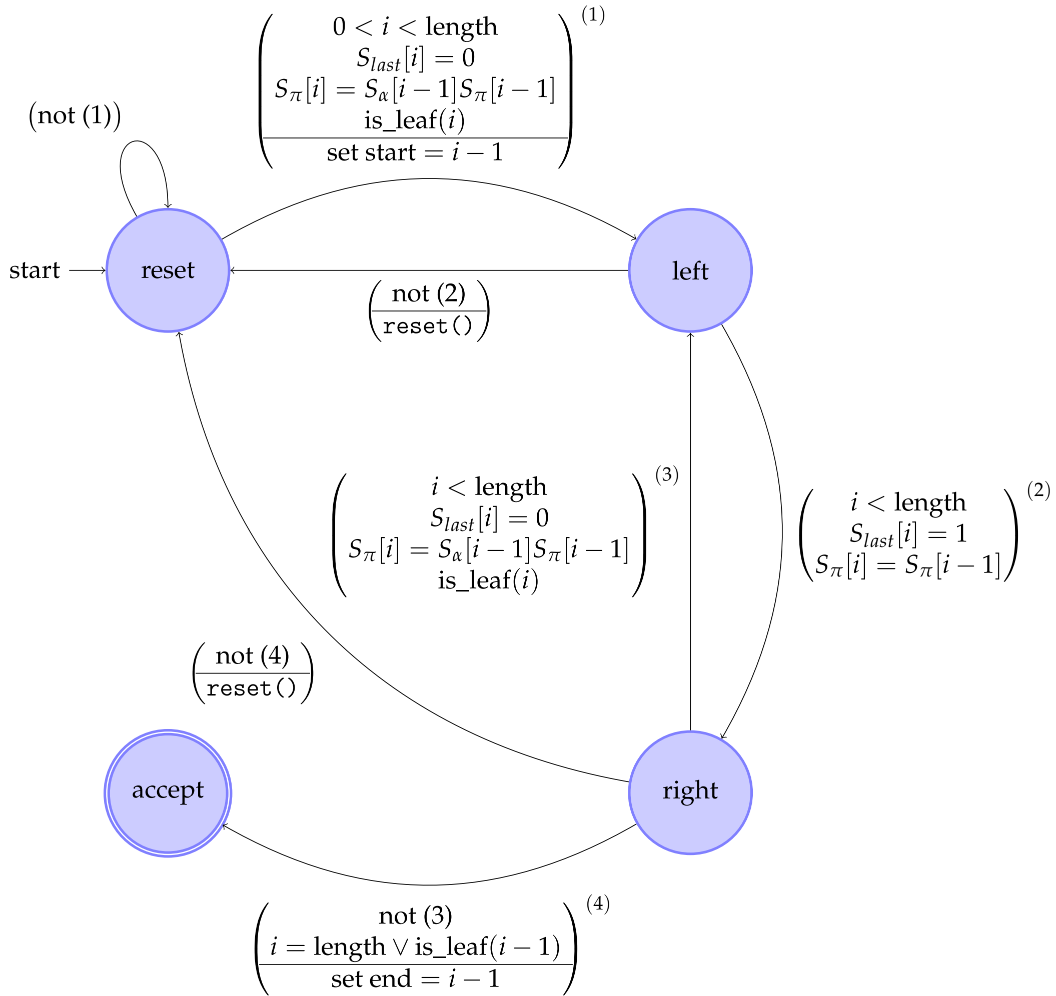


FIGURE 3.10: The right-spine automaton.

We describe the logic of the left automaton in detail, and only mention that of the right one, which is similar. What the automaton does is to simply follow the pattern described by Lemma 3.3: it looks for an "upward" series of h S_π values, followed by a "downward" series of other h values that are "symmetric" to the first group. In the meanwhile, it also checks for the correct series of S_{last} values, that is, a contiguous run of h 0s followed by a contiguous run of h 1s. This is done by maintaining some internal variables, like i, which is automatically increased at each step, and a few others.
The automaton starts at the `reset` state, and may eventually enter the `up` state. If it does, but then realizes that any condition is not satisfied for the detection of a left spine, it falls back to `reset`. Alternatively, it may detect the end of an "upward" phase, and make a transition to the `down` state. Once there, the detection of a left spine, of height $h = 1$ or greater, is guaranteed. The automaton greedily iterates itself until some condition in the `down` state is no longer satisfied, and then terminates in the `accept` state. The calling algorithm, when notified of the acceptance, collects the (a, b) range and resets the state of the automaton with a call to `reset()`.

`reset()` does not bring i back to 0, but merely resets the state and a few other internal variables.
The explanation for the right-spine automaton is analogous, except that the automaton looks for a pattern of alternating 0 and 1 pairs in S_{last}. Two sibling nodes of a right spine i and $i+1$ are such that $S_\pi[i] = S_\pi[i+1] = S_\alpha[i-1]S_\pi[i-1]$.

The correctness of Algorithm 1 follows almost directly by the two automata, and the correctness of these can be verified in detail by Figure 3.9 and Figure 3.10.
To analyze the complexity of Algorithm 1, consider its `while` loop. Clearly, this runs as long as `automaton.has_next()` which, by definition, returns `true` at most $|T|$ times. The space complexity depends from $|$`spines`$|$. This is not constant, but it can be made so by the use of *generator functions* that generate, rather than store in memory, the values of `spines` on a one-by-one fashion. If Algorithm 1 is implemented on a language without generator functions support, then the space complexity is $\Theta(|\texttt{spines}|)$.

Finally, to prove that Algorithm 1 is able to detect all left *and* right spines in a single run, note that the left and right automata maintain an internal state that is independent from that of the other. It is therefore possible to modify Algorithm 1 in such a way that, on each iteration, it updates both automata. They will terminate in the same number of steps, as long as they are initialized with the same XBW$^-$ transform S. □

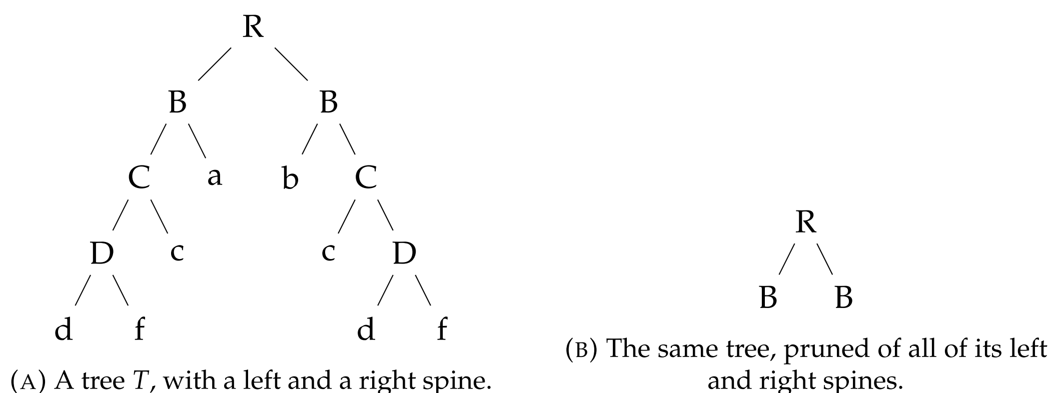
Chapter 4

Storage of Spines

Now that we have introduced a practical algorithm for the identification of spine trees, we turn to the problem of what to do with them once they have been detected, specifically, how to store spines so that they can benefit from enhanced compression ratios with respect to the remaining portions of the tree. We have devised two primary and alternate solutions to this problem: a prune-based storage technique, and a so–called context-manipulation one. We'll start with the simpler.

4.1 Pruning of Spine Trees

As the name itself suggests, the *pruning storage technique* consists of detecting all spine trees of a tree T and moving them out of the tree. Formally, let i be the index of a spine subtree S within T, then the pruning of S has the effect of replacing the whole subtree of S with its root node, namely the node indexed i. Figure 4.1 depicts this process for a tree hosting a left and a right spine.



(A) A tree T, with a left and a right spine.

(B) The same tree, pruned of all of its left and right spines.

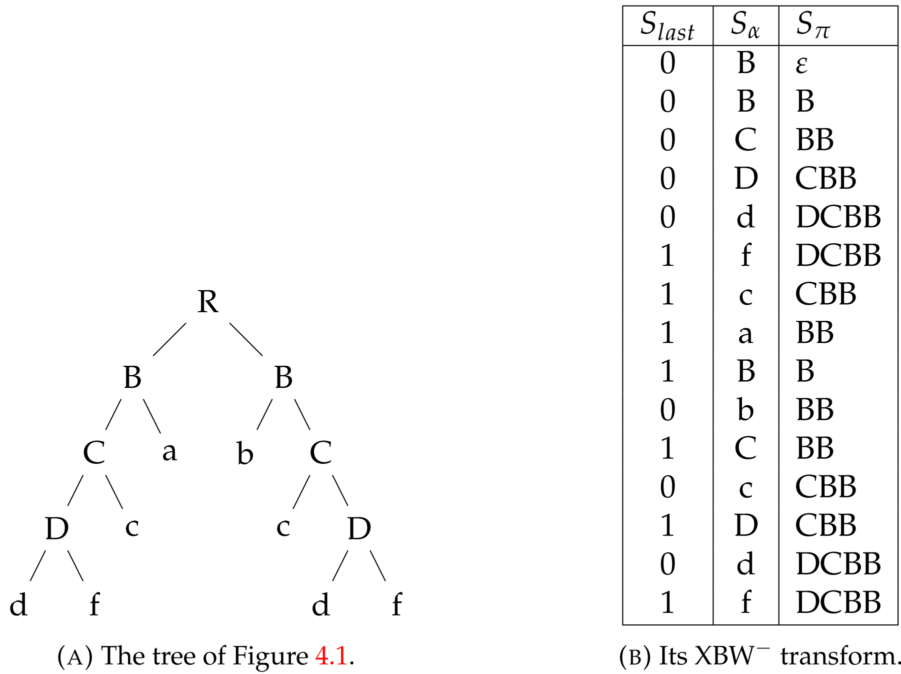
FIGURE 4.1: Pruning of a tree T.

Once the spines have been pruned out of their host tree, they can be safely stored and compressed as strings, due to their linear structure. For example, the two arms of a left spine can be concatenated to each other; for the case of Figure 4.1a, this would produce the string `BCDdacf`. Or the same spine may be turned into string by a level-wide visit; for the same left spine of of Figure 4.1a, this would give the string `BCaDcdf`.

4.2 Context Manipulation of Spine Trees

For the second technique of spine storage, consider the properties of the XBW transform. As you may be able to recall from the introductory section, the construction of the XBW transform is comprised of a final sorting step, that orders the tuples within the XBW transform S by the lexicographic order of their S_π component. The S_π subvector is often also referred to as *context* [Fer+09].

We may take advantage of this ordering to devise an alternate storage scheme for spines. We will explain it with reference to the tree of Figure 4.1, that we newly report in Figure 4.2, along with its XBW$^-$ transform.



(A) The tree of Figure 4.1.

S_{last}	S_α	S_π
0	B	ε
0	B	B
0	C	BB
0	D	CBB
0	d	DCBB
1	f	DCBB
1	c	CBB
1	a	BB
1	B	B
0	b	BB
1	C	BB
0	c	CBB
1	D	CBB
0	d	DCBB
1	f	DCBB

(B) Its XBW$^-$ transform.

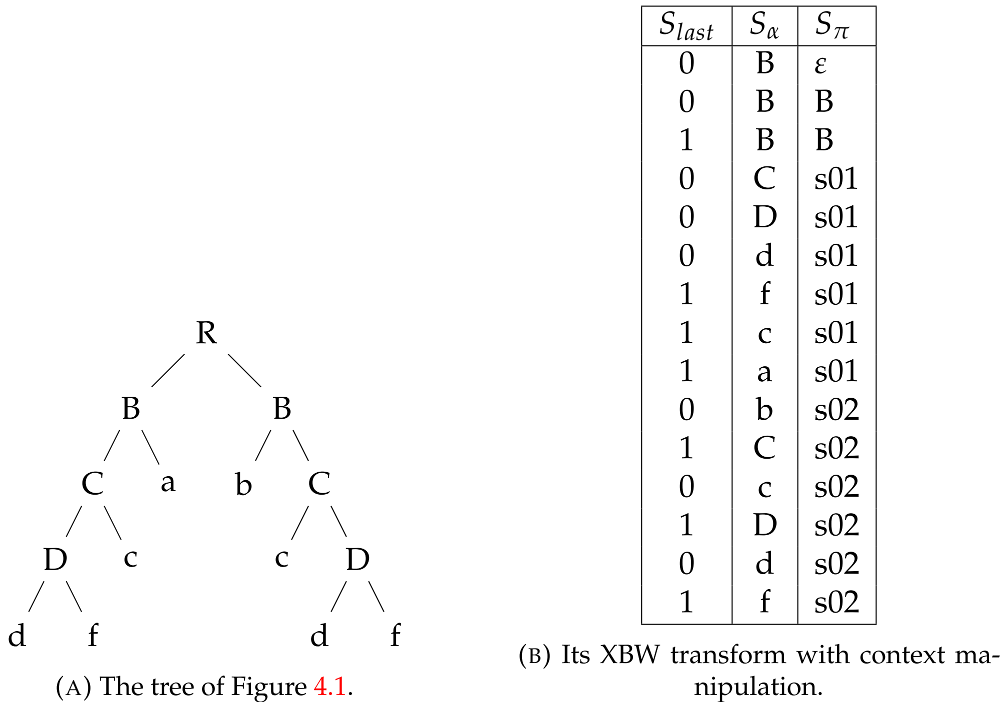
FIGURE 4.2: The tree of Figure 4.1, along with its XBW$^-$ transform.

The idea of this scheme is to manipulate the S_π subvector (that is, the context–hence the name *context manipulation*) in such a way as to cluster all the nodes of a spine together. The way we do this is the following. First, we detect all the spines of the input tree T, and associate a simple numeric index to them. We then consider the alphabet A used for the labeling of T, and pick a symbol $\xi \notin A$. Then for each spine i and for each of its nodes j that is not the root, we alter $S_\pi[j]$ so as to set it to

$$S_\pi[j] = \xi i \tag{4.1}$$

We then proceed to order the XBW$^-$ vector according to S_π, waiting for the sorting process to do the rest. Since this sorting procedure is a stable one by hypothesis, we are guaranteed that the local order of nodes within their spine tree remains unaffected.

This procedure, applied to the XBW$^-$ transform of Figure 4.2, results in the XBW transform of Figure 4.3b.



(A) The tree of Figure 4.1.

S_{last}	S_α	S_π
0	B	ε
0	B	B
1	B	B
0	C	s01
0	D	s01
0	d	s01
1	f	s01
1	c	s01
1	a	s01
0	b	s02
1	C	s02
0	c	s02
1	D	s02
0	d	s02
1	f	s02

(B) Its XBW transform with context manipulation.

FIGURE 4.3: The tree of Figure 4.1, along with its XBW transform that has undergone a context manipulation. Here, $\xi = \text{s}$.

In addition to the manipulation of context of the nodes of a spine, we also have to memorize which nodes of the transformed XBW belong to the root of a spine, like the nodes B and B in Figure 4.3. There are multiple ways to do this, from picking a special value for the label of the root nodes to alterate their own context in peculiar ways. Since this is more of an implementational aspect, we do not detail it futher here.

The choice of which storage scheme to adopt may depend on the concrete application. The context-manipulation technique is easier to implement, since it just requires to alter the context of some nodes, leaving the rest to the XBW transform procedure, but it does not offer a way to compress separately the spines from the tree they were originally embedded in; with this storage scheme, only one compression algorithm can be used for the whole tree. On the other hand, with the pruning technique, we can adopt a general-purpose compression algorithm for the input tree, and use a specialized one for the spine trees, which may be simpler (i.e. low–entropy) and able to be compressed with a higher compression ratio; on the other hand, the pruning technique requires one to devise a way of storing the pruned tree and the extracted spines apart, which may be a little bit trickier to realize in practice.

Chapter 5

Spine Compression

Having described techniques for the detection and storage of spines, we turn to the problem of spine compression. Utilizing the technique of pruning from the previous section for storing spines, we can utilise two distinct algorithms for compressing a tree: one dedicated to the compression of spines, and the other to the compression of the pruned tree. If the spines we deal with are regular and long enough, we can produce an actual saving in space.

In recent times, spine trees have made an appearance in the implementation of functional programming languages [ALV21]. We have already seen that the detection of spines is asymptotically optimal, producing neither time nor space overhead, and the implementation we can realize is performant in practice too. Therefore, if we discovered ways to efficiently compress spines other than detect them, we would have found ways to improve the compilation or the runtime environment of these functional programming languages. To that account, we dedicate the next section to tackling one problem, sitting in the topic of data compression, that we refer to as *logarithmic compression*.

5.1 Logarithmic Compression

Shannon's contribution to the study of data communication [Sha48] teaches us that there is a limit to how far we can losslessly compress a string. Specifically, for a random variable (also referred to as *source)* X drawing symbols from an alphabet $A = \{a_1, \ldots, a_n\}$ with individual probabilities $p_1, \ldots, p_n$ we can't, on average, compress a string $x = x_1 x_2 \cdots x_t \in A^t$ with a number of bits per symbol less than

$$H(X) = \sum_{x \in A} P(x) \log \left(\frac{1}{P(x)} \right) \tag{5.1}$$

where $P(x)$ denotes the probability of the source X emitting the symbol x. The quantity expressed in (5.1) is called *entropy* of the source X and is a characterization independent of the particular string x originated from X. For a message x of length t we therefore need, *on average*, about $tH(X)$ bits to convey it from one sender to one receiver. The specific number of bits required will depend on the encoding method, or algorithm.

Remember that a spine is nothing more than a pair of strings, or just one resulting from the concatenation of the two. Therefore, the usual limitations of data compression apply to spines as to any other textual string.

The problem we want to tackle is the following. Consider a compression algorithm C, capable of operating on a variety of sources X. As we know, C can't compress, on average, any better than Shannon's lower bound. However, is there a limited subset of strings for which C can achieve a compression rate lower than $H(X)$? (Of course, it might also be the case that for a given compression algorithm C there exist groups of strings which are expanded, rather than compressed, by a given factor.) Specifically, can we achieve some sort of *logarithmic compression* for a proper subset of all possible inputs of C?

5.2 Problem Statement

Consider a compression algorithm C, and denote by dom C the domain upon which compression by C is possible. Suppose, for the sake of generality, that dom $C = A^*$, where A is an alphabet set of arbitrary size. We state the problem of logarithmic compression by C as finding a constant $c \geq 1$ and a subset $D \subseteq \text{dom } C$ such that, $\forall x \in D$

$$l(C(x)) \leq c \log(l(x)) \tag{5.2}$$

Here, $C(x)$ denotes the application of C to x to yield a compressed version of x, while $l(x)$ denotes the number of bits needed to represent x with some coding convention. For example, if $|x| = t$ and every $x_i \in A$ is encoded uniformly with $m = \lceil \log |A| \rceil$ bits, then $l(x) = |x|m = tm$.

One objection that could be moved against this definition is that we are being too lax by allowing the choice of a constant c. In fact if we, say, were to pick $c = 10^6$, we would consent to the expansion, rather than compression, of an input string x.

The rationale for considering logarithmic compression minus a constant factor, however, is that the log function is a slowly-increasing one, and for inputs x of very large size, logarithmic compression would no longer be possible. Imagine to have a string x of 10^9 bytes: if we had no regard for constant factors (corresponding to implicitly set $c = 1$), and worked with logarithms in base 2, then we would force ourselves to compress x with no more than 29 bytes ($\log_2 10^9 \approx 29.9$). On the other hand, by even setting c to an as low a value as 2 or 3, we expand the set of strings x that can be logarithmically compressed by C, without sacrificing much in terms of space performance.

The choice of c is up to the algorithm designer, and is of course fixed before picking any $D \subseteq \text{dom } C$ and $x \in D$.

We can generalize the problem beyond the logarithm function, by considering an arbitrary function $f : \mathbb{R}^+ \to \mathbb{R}^+$. We denote by $\text{dom}_{f,c}\, C$ and call it the *restriction domain of* C *by* f the set of all inputs x which C compresses up to $cf(l(x))$ bits. Formally, given a $c \geq 1$

$$\text{dom}_{f,c}\, C = \{x \in \text{dom } C \mid l(C(x)) \leq cf(l(x))\} \tag{5.3}$$

When we don't want to be overly precise, or can deduce the value of c from the context, we'll simply write $\text{dom}_f\, C$ instead of $\text{dom}_{f,c}\, C$.

5.3 Candidate Algorithms

Rather than devise a new compression algorithm from scratch, we will examine some of the existing ones. The approach we follow is a gradual one, transitioning from inapplicable solutions to more promising ones.

Huffman Coding To start with, consider the Huffman coding [Huf52; Say18a], that is, let C be a compression algorithm that compresses according to the Huffman method. In such a case, $\text{dom}_{\log,1} C = \emptyset$. In fact, for the Huffman algorithm to work, each symbol x_i is mapped to another symbol y_i, possibly occupying fewer bits. That is, if $y = C(x)$ then $|y| = |x| = t$, and since we need at least one bit for each y_i (and more than one for at least a given y_j, $j \neq i$), $c \log(l(y)) \leq t \lneq l(y)$.

If we wanted to stick with the Huffman coding, we could consider the extended Huffman coding [Say18a], where each chunk of k symbols from x gets encoded as a single symbol in the output string y. While this could increase the chances of achieving logarithmic compression, for large values of k we would need to construct a correspondingly large table of $|A|^k$ entries associating every possible sequence $s \in A^k$ to its own codeword. In a networked setting, transmitting this entire table would soon cancel out any advantage derived from the text compression.

LZ77 Recall, from section 1.3.3, when we talked about Lempel-Ziv compression methods. In the current paragraph, we will investigate the performance of LZ77 [ZL77] with reference to logarithmic compression. As we are going to see, LZ77 acts as a solution of intermediate effectiveness, since it is able to perform well only for inputs of a limited size.

We will denote by C the LZ77 compression algorithm, and by x an arbitrary input string of n symbols; if the alphabet of x is A, then the binary size of x, $l(x)$, is given by mn, where $m = \lceil \log |A| \rceil$.

As we have seen, LZ77 has a limited view into the compressible sequence, being limited to scan no more than F symbols of its input string. Therefore, the best it is able to perform is scan its whole input, split it up into $\frac{n}{F}$ "chunks" and compress those into an equal number of blocks of size d each. Clearly, as the length n of the sequences we expect to compress increases, we have to adapt the algorithm parameters accordingly.

To achieve logarithmic compression with LZ77 means to find a constant $c \geq 1$ and a subset $D \subseteq \text{dom}\, C$ such that, $\forall x \in D$, $l(C(x)) \leq c \log(mn)$ (see (5.2)). LZ77 works by scanning the input string x, dividing it into N chunks x_i of possibly distinct length, and compressing them into fixed-sized blocks of length d each (recall the definition of d from (1.40))

The binary length, expressed in bits, $l(C(x))$ of the algorithm output is therefore given by dN. Since we want it to be no longer than $c \log(nm)$ we have that

$$dN \leq c \log(nm) \tag{5.4}$$

mn (bytes)	chunks num.	chunks length	output size %
2^{10}	130	8	41.26
2^{11}	140	15	22.22
2^{12}	150	28	11.90
2^{13}	160	52	6.35
2^{14}	170	97	3.37
2^{15}	180	183	1.79
2^{16}	190	345	0.94
2^{17}	200	656	0.50
2^{18}	210	1249	0.26
2^{19}	220	2384	0.14
2^{20}	230	4560	0.07

TABLE 5.1: Maximum number N of allowable splits of x as its length n increases. As always, m denotes the number of bits needed to encode one single symbol of x uniformly. In this case, $m = 8\text{bits} = 1\text{byte}$; all other measures are likewise in bytes. d denotes the size of a single block out of N, and c the multiplicative constant from (5.2). We expressed c in terms of d and set $c = 10d$.

or

$$N \leq \frac{c}{d} \log(nm) \tag{5.5}$$

(5.5) tells us how many chunks x can be split into at most. Since mn grows more quickly than $\log(mn)$ for greater values of n, the greater n, i.e. the longer x, the more difficult it will be to compress x down to a logarithmic factor. To get a sense of how this is true, take a look at Table 5.1.

In Table 5.1, we set $m = 8$ bits, $B = 1024$, $F = 512$; each row of the table reports an input length, in bytes, as a power of two, the maximum number of chunks N that an input of that length can be split into by the LZ77 algorithms, the average number of symbols per chunk, and the output size, in percentage, of a compressed version of the string if the LZ77 algorithm was able to parse its input in exactly N chunks. As we can see, although the size of these inputs increases exponentially, the maximum number of allowed chunks for these inputs is only incremented by a constant of 10 per row. To be able to consistently achieve logarithmic compression as the input size increases is a feat getting harder and harder as we progress toward greater input sizes.

To experiment with this situation, we set up a compression experiment for LZ77. We considered files of exponentially growing size, from 1 KiB = 2^{10} bytes up to 1 MiB = 2^{20} bytes. These files—denoted as LCFs (Logarithmically Compressible Files)—are formed by a concatenation of a limited number of chunks, each composed by the repetition of a single character. In the experiment we conducted, we fixed the number of distinct chunks to 10 and their length equal to $\frac{B-F}{5}$, so that about 5 chunks can fit into the window buffer of LZ77. An example LCF of 1KiB is shown in Figure 5.1.

```
eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee
↪  eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee
↪  eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee
↪  eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee
↪  eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee
↪  eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee
↪  eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeHHHHHHHHH
↪  HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH
↪  HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH
↪  HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH
↪  HHHHHHHHHHHHHHHHHHsssssssssssssssssssssssssssssssssssssssss
↪  sssssssssssssssssssssssssssssssssssssssssssssssssssssssssss
↪  sssssssssssssssssssssssssssssssssssssssssssssssssssssssssss
↪  sssssssssssssssssssssssssssssssssssssssssssssssssssssssssss
↪  sssssssssssssssssssssssssssssssssssssssssssssssssssssssssss
↪  sssssssssssssssssssssssssssssssssssssssssssssssssssssssssss
↪  sssssssssssssssssssssssssssssssssssssssssssssssssssssssssss
↪  sssssssssssssHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH
↪  HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH
↪  HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH
↪  HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH
```

FIGURE 5.1: An example of a LCF (Logarithmically Compressible File) of 1KiB. The name for these files is a convenience, since LCFs aren't necessarily logarithmically compressible, and if they are, they are with respect to one algorithm, without necessarily being with respect to another.

The outcome of our experiment is visualized in Table 5.2, where the first column denotes the number of bytes for a given LCF and the second the number of blocks that LZ77 divided its input into. Knowing that each of these blocks is encoded in d bits, LZ77 would be able to compress these files with just dN bits. The halfway horizontal line splitting Table 5.2 in two halves demarcates the extent by which LZ77 is able to achieve logarithmic compression with $c = 10$, i.e. about 32KiB. This conclusion is derived from Table 5.1, where the maximum number of allowed chunks N for an input of 2^{15} bytes is 180. If our requirements aren't excessive and our expected input is not overly complex, we can put LZ77 to good use and expect to receive a logarithmic compression factor.

The BWT–RLE Approach We finally take a look at a candidate algorithm for the problem of logarithmic compression, given by the combination of the Burrows-Wheeler Transform [BW94] and run-length encoding, that is fast and has a wide restriction domain $\mathrm{dom}_{\log}C$. To the best of our knowledge, this is the best solution we could come up with.

LCF length (bytes)	N = parsed blocks num.
2^{10}B = 1KiB	5
2^{11}	11
2^{12}	20
2^{13}	37
2^{14}	73
2^{15}B = 32KiB	147
2^{16}	293
2^{17}	552
2^{18}	1158
2^{19}	2243
2^{20}B = 1MiB	4514

TABLE 5.2: LZ77 performance on a series of exponentially-growing LCFs (Logarithmically Compressible Files). Having set a logarithmic constant $c = 10$, LZ77 achieves logarithmic compression up to a file of 32KiB, reflecting the fact that being able to do so for infinitely growing inputs becomes harder and harder.

Run-length encoding is a compression scheme acting on runs of repeated characters. When an input string is formed by a series of identical symbols, they are substituted by a shorter description, indicating the character that is being repeated and the number of times it appears in the source text. For example, if our input string $x = \texttt{AAAAABBB}$, then $C(x) = \texttt{5A3B}$. With this algorithm, we find it easy to characterize, although in part, the restriction domain $\text{dom}_{\log} C$ of C. For example, for sufficiently large values of n

$$0^n \in \text{dom}_{\log,1} C \tag{5.6}$$
$$0^n 1^n \in \text{dom}_{\log,2} C \tag{5.7}$$
$$0^n 10^n \in \text{dom}_{\log,2} C \tag{5.8}$$

Run-length encoding is a simple compression algorithm, but its scope of application may seem limited at first. However, consider the following. On one hand, the set of logarithmically-compressible strings is known to be "limited" and somehow "simple", from the point of view of information theory. On the other hand, run-length encoding can find application beyond the set of inputs described in (5.8). For this, it suffices to consider the BWT (Burrows-Wheeler Transform) [BW94].

The BWT is not a compression algorithm per se, but a transformation between strings from the same alphabet. Given an input string x, the BWT produces an output string x' containing the same symbols as x, but rearranged in such a way as to form runs of repeated characters. This process improves the compression ratio of algorithms that apply the MTF (Move to Front) transform, or directly apply run-length encoding, as in our case. Although hard to analyze mathematically, the combination of run-length encoding and BWT

makes for a promising practical solution to our problem of logarithmic compression.

As an example, take the string $x = \mathtt{ABABABAB}\ldots\mathtt{AB}$. If $\forall c \in A, C(c) = c$, then $C(x) = x$, meaning that we obtain no compression at all. However, consider the application of the BWT to x to obtain a string x'. In this case, we have that

$$x' = \underbrace{\mathtt{AAA}\ldots\mathtt{A}}_{\frac{|x|}{2}} \cdot \underbrace{\mathtt{BBB}\ldots\mathtt{B}}_{\frac{|x|}{2}} \tag{5.9}$$

We can now apply the run-length encoding to x'; if x' is long enough, we get a logarithmic compression; otherwise, an equally good compression factor.

Run-length encoding would also achieve a better compression ratio, with lower computational resources consumption, with the type of LCFs (Logarithmically Compressible Files) described in the LZ77 paragraph, since these files are constituted by a run of repeated characters.

Experimental Outcomes To validate the choice of the BWT plus RLE approach (BWT–RLE), some experiments were run. As a first attempt, consider an input string sequence of the form $x = \mathtt{AB}^{2^i}$, with $i \in \{1, \ldots, 16\}$, like the one just mentioned (higher values of i weren't considered due to computational overhead and lack of performance optimization). The experiment of compressing this first type of sequence is illustrated in Figure 5.2. With a tendency such as the one shown in the picture, it also emerges from an experimental basis that for sufficiently large values of i, $\mathtt{AB}^i \in \mathrm{dom}_{\log,2} C$, and possibly, even that $\mathtt{AB}^i \in \mathrm{dom}_{\log,1} C$, where C denotes the BWT–RLE combination.

Similar results are obtained if we extend the iterated character pair AB to a longer sequence, like ABCDE. In fact, thanks to the BWT, we have that

$$\mathrm{BWT}(\mathtt{ABCDE}\cdot\mathtt{ABCDE}\cdot\ldots\cdot\mathtt{ABCDE}) = \underbrace{\mathtt{AAA}\ldots\mathtt{A}}_{\frac{n}{5}} \cdot \underbrace{\mathtt{BBB}\ldots\mathtt{B}}_{\frac{n}{5}} \cdots \underbrace{\mathtt{EEE}\ldots\mathtt{E}}_{\frac{n}{5}} \tag{5.10}$$

Figure 5.3 shows how the BWT–RLE algorithm performed with $\mathtt{ABCDE}^{2^i}$.

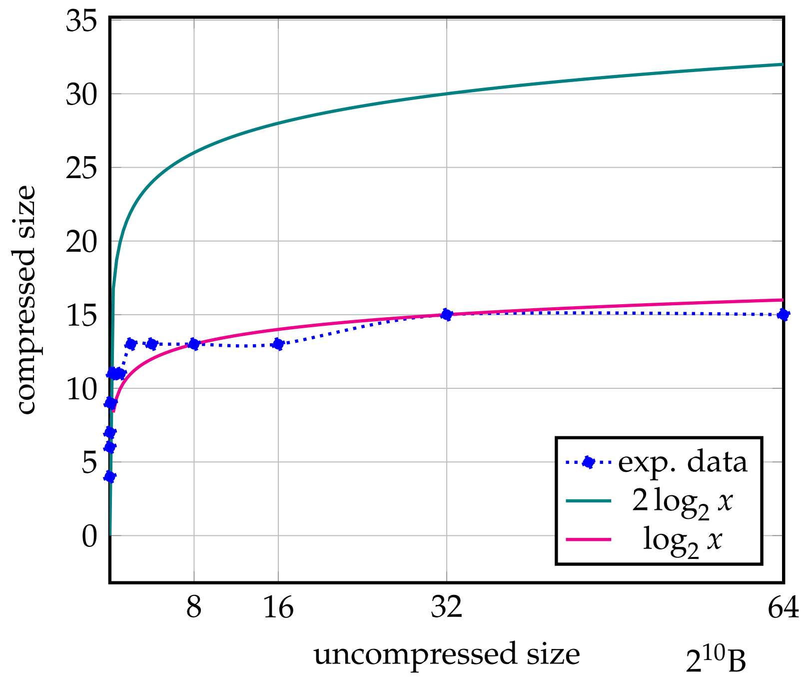


FIGURE 5.2: Compression of $\mathtt{AB}^{2^i}$ for $i \in \{1, \ldots, 16\}$.

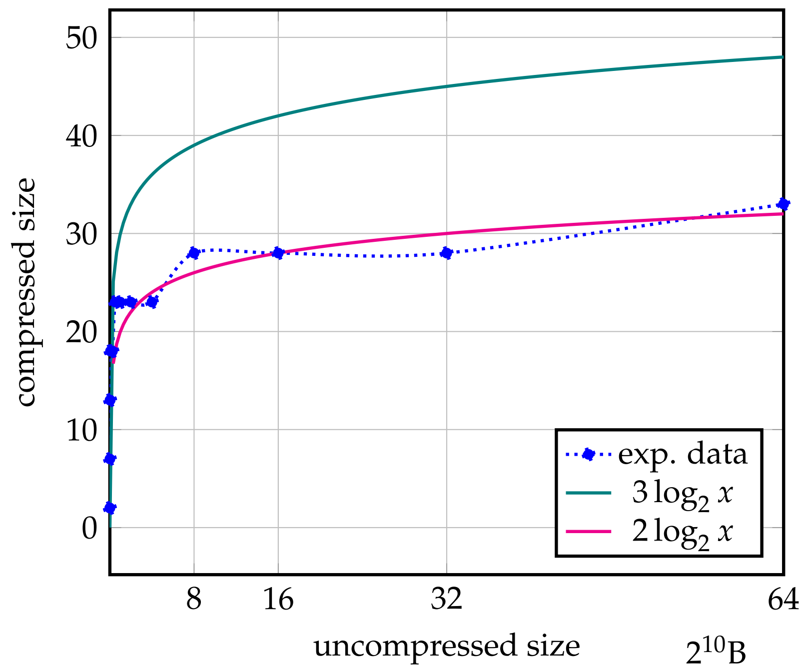


FIGURE 5.3: Compression of $\mathtt{ABCDE}^{2^i}$ for $i \in \{1, \ldots, 16\}$.

Chapter 6

Conclusions

Of all the major topics that we addressed in this work—detection, storage and compression of spines—the one more likely to benefit from extended research is spine compression. As we have seen, spine detection on XBW transforms can be solved efficiently, both in time and space, and it is hard to see any possibility of improvement, since the detection of all spines of a tree requires a single iteration of the XBW transform. Spine storage has been given a couple of satisfying solutions, and one would hardly expect any improvement in the study of spines to originate from here. Still, novel ideas might still be proposed.

For spine compression, and even more specifically logarithmic compression, there is still a plethora of different compression algorithms that could be evaluated and proposed as an alternative. One of these is arithmetic coding [Say18b], that we haven't considered just for lack of more time.

An additional problem that was originally proposed by my thesis advisor, but was not given complete attention, is the study of a couple of operations on labeled trees—of any type, not just spine trees—that were named *construction* and *destruction*.

Formally, the construction operation on trees is an algorithmic procedure that is specified to receive a tuple of n trees $(T_1, \ldots, T_n)$, and produce as output a tree T with a single root r and n direct child nodes c_i, such that each c_i is the root of the tree T_i. The peculiarity of this operation is that it is required to operate on the compressed representation of each tree T_i, and produce the output tree T without decompressing it. This is a bit like homomorphic encryption, where we try to perform some operations on encrypted data without decrypting it, except that we have compression, and not encryption.

Analogously, the destruction operation on trees is a procedure that is specified to take the compressed representation of a tree T, formed by a single root r and n direct child nodes c_i, and output a tuple $(T_1, \ldots, T_n)$, where each T_i is the *compressed* sub tree obtained by the child node c_i of r. Again, the implementation specifications require this operation not to decompress any of the individual trees T_i.

These two problems, construction and destruction of compressed trees, were considered since the beginning, but never managed to reach a full, mature solution, mostly because of time limitations. It is likely that an adequate solution for them would benefit to the same area that motivated the study of the efficient compression of spines.

Chapter 7

Acknowledgements

The following persons have been involved, directly or indirectly, in the preparation of this work, and I feel an obligation to acknowledge their role in it.

First, my thesis advisor, professor Ugo Dal Lago[1], whose mere presence was already a merit. I want to thank him for giving me the possibility to delve in a topic—information theory—that I had longed to study for a long time, but had not managed to do, either because no formal lectures were offered for it in my computer science curriculum, or because I never found the time to self-study it. I hope that someday our department will be able to equip itself with such a course.

My co-advisor Gabriele Vanoni[2] deserves no less gratitude for keeping a constant pressure on my work, and soliciting for my drafts, even when I was delaying for them. I thank him for taking the time to read the work and give direct feedback on it, and also for providing precious indications in the relevant literature.

My family played an important role throughout all my whole undergraduate (*laurea triennale*) degree, not only during the time of my thesis, so I want to also express gratitude for them. Thank you for providing me with the resources and the time needed to focus on my studies.

[1] https://www.unibo.it/sitoweb/ugo.dallago/.

[2] https://www.unibo.it/sitoweb/gabriele.vanoni2/.

Bibliography

[Nyq24] H. Nyquist. "Certain Factors Affecting Telegraph Speed". In: *Transactions of the American Institute of Electrical Engineers* XLIII (1924), pp. 412–422. DOI: 10.1109/T-AIEE.1924.5060996.

[Har28] R. V. L. Hartley. "Transmission of information". In: *The Bell System Technical Journal* 7.3 (1928), pp. 535–563. DOI: 10.1002/j.1538-7305.1928.tb01236.x.

[Sha48] C. E. Shannon. "A mathematical theory of communication". In: *The Bell System Technical Journal* 27.3 (1948), pp. 379–423. DOI: 10.1002/j.1538-7305.1948.tb01338.x.

[Huf52] David A. Huffman. "A Method for the Construction of Minimum-Redundancy Codes". In: *Proceedings of the IRE* 40.9 (1952), pp. 1098–1101. DOI: 10.1109/JRPROC.1952.273898.

[ZL77] J. Ziv and A. Lempel. "A universal algorithm for sequential data compression". In: *IEEE Transactions on Information Theory* 23.3 (1977), pp. 337–343. DOI: 10.1109/TIT.1977.1055714.

[ZL78] J. Ziv and A. Lempel. "Compression of individual sequences via variable-rate coding". In: *IEEE Transactions on Information Theory* 24.5 (1978), pp. 530–536. DOI: 10.1109/TIT.1978.1055934.

[Wel84] Welch. "A Technique for High-Performance Data Compression". In: *Computer* 17.6 (1984), pp. 8–19. DOI: 10.1109/MC.1984.1659158.

[BW94] Michael Burrows and David Wheeler. *A Block-Sorting Lossless Data Compression Algorithm*. Tech. rep. DIGITAL SRC RESEARCH REPORT, 1994.

[Fer+05] Paolo Ferragina et al. "Structuring Labeled Trees for Optimal Succinctness, and Beyond". In: *Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science*. FOCS '05. USA: IEEE Computer Society, 2005, pp. 184–196. ISBN: 0769524680. DOI: 10.1109/SFCS.2005.69. URL: https://doi.org/10.1109/SFCS.2005.69.

[CT06] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory (Wiley Series in Telecommunications and Signal Processing)*. USA: Wiley-Interscience, 2006. ISBN: 0471241954.

[Fer+09] Paolo Ferragina et al. "Compressing and Indexing Labeled Trees, with Applications". In: *J. ACM* 57.1 (Nov. 2009). ISSN: 0004-5411. DOI: 10.1145/1613676.1613680. URL: https://doi.org/10.1145/1613676.1613680.

[Say18a] Khalid Sayood. "Chapter 3 - Huffman Coding". In: *Introduction to Data Compression (Fifth Edition)*. Ed. by Khalid Sayood. Fifth Edition. The Morgan Kaufmann Series in Multimedia Information and Systems. Morgan Kaufmann, 2018, pp. 41–88. ISBN: 978-0-12-809474-7. DOI: https://doi.org/10.1016/B978-0-12-809474-7.00003-3. URL: https://www.sciencedirect.com/science/article/pii/B9780128094747000033.

[Say18b] Khalid Sayood. "Chapter 4 - Arithmetic Coding". In: *Introduction to Data Compression (Fifth Edition)*. Ed. by Khalid Sayood. Fifth Edition. The Morgan Kaufmann Series in Multimedia Information and Systems. Morgan Kaufmann, 2018, pp. 89–130. ISBN: 978-0-12-809474-7. DOI: https://doi.org/10.1016/B978-0-12-809474-7.00004-5. URL: https://www.sciencedirect.com/science/article/pii/B9780128094747000045.

[ALV21] Beniamino Accattoli, Ugo Dal Lago, and Gabriele Vanoni. "The Space of Interaction". In: *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. 2021, pp. 1–13. DOI: 10.1109/LICS52264.2021.9470726.