Università degli Studi di Salerno

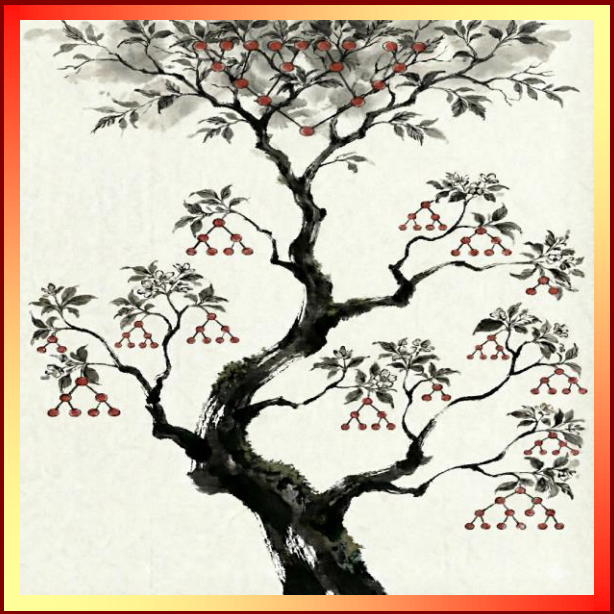
Dipartimento di Informatica

Dottorato di Ricerca in Informatica – XXXVIII Ciclo

Tesi di Dottorato/Ph.D. Thesis

From Search to Coding (and Back): Two Sides of the Same Coin

Roberto Bruno



Supervisor: **Prof. Ugo Vaccaro**

Ph.D. Program Director: **Prof. Andrea De Lucia**

AA 2024/2025

Curriculum Computer Science and Information Technology

Università degli Studi di Salerno

Dipartimento di Informatica

Dottorato di Ricerca in Informatica
Curriculum Computer Science and Information Technology
XXXVIII Ciclo

Tesi di Dottorato / Ph.D. Thesis

From Search to Coding (and Back): Two Sides of the Same Coin

Roberto BRUNO

Supervisor: **Prof. Ugo VACCARO**

PhD Program Director: **Prof. Andrea DE LUCIA**

A.A 2024/2025

To my family, my friends, and all those who stood by me and believed in me, even when I was the first to doubt myself.

"The first rule of all problems is to find out what you have to find out."

— George Pólya

"You miss 100% of the shots you don't take."

— Michael Scott

"The whole purpose of computation is insight, not numbers."

— Richard W. Hamming

"We have seen that computer programming is an art, because it applies accumulated knowledge to the world, because it requires skill and ingenuity, and especially because it produces objects of beauty."

— Donald E. Knuth

ACKNOWLEDGMENTS

This section is dedicated to the amazing people who have supported me-not only through these years of study, but through life itself. Thank you, truly.

My heartfelt gratitude goes first and foremost to my family. You have always had my back, not just over these past eight years, but for my entire life. I especially thank you for carrying me through the worst moments of my life, even when I wanted to give up or I struggled with the most basic things, like simply reaching for a cup of water. This small token of thanks will never be enough to express my love and appreciation. I hope that through your example I might become a person who is at least a bit like you all.

A special thanks goes to my uncles as well. You have always believed in me and celebrated all my achievements. You were the first to encourage me to pursue this long academic journey.

Next, I thank my "buddies": Jacopo, Paolo, and Piero (in alphabetical order). Thank you for your support, even when I was (and often am) at my worst—I still marvel at your ability to endure me. We've shared some of the best moments of my life (and, yes, some of the worst too, especially with Jacopo, yet somehow, we are both still here). I know that I am terrible at showing my emotions, but I genuinely cherished all the afternoons and evenings we spent together simply watching random things, playing video games, or even playing Magic. Thank you for never giving up on our friendship.

I need to give a special thank also to all the friends that I made along these eight years: Ciccio, Giusy, Lorenzo, Marco, and Sabato (again, in alphabetical order). We lived through all the anxieties, difficulties, and endless questions about why we chose this path in the first place, thinking about giving up so many times. But we stuck together, drinking an almost uncountable amount of coffee during our long coffee breaks. I don't know how some of you managed to endure so much my countless complaints. Thank you for choosing to be my friends (a problem still undecidable for me). Moreover, I want to extend my gratitude to all the other friends

encountered along the way whom I haven't explicitly mentioned here. Your presence and support truly meant the world to me.

A very special, huge thanks goes to my Advisor, Ugo Vaccaro. Your title truly reflects your role; you advised me exceptionally well during these "long-short" three years. Your passion for Science is inspiring. I thank you for everything you taught me and for the interesting and funny stories you shared with me during our long sessions gazing at nearly empty blackboards. You truly showed me the beauty of research when it's done right.

Another special thank you goes to the wonderful people I met in Okinawa. I learned so much from you and, most importantly, had a lot of fun together. A special acknowledgment goes to Amedeo (or, perhaps more appropriately, Professor Esposito). You taught me so many new things and opened my eyes to aspects of research I hadn't experienced at all in Italy. I want to really thank you for the massive effort you put into helping me figure out what I wanted to do in my life——a question I am still not entirely sure I have answered.

I am certain there are many more people I should thank, because without all the help I received, I would be nothing. However, as many who know me can attest, I am not good with words Thus, I will simply conclude by saying: Thank You.

Questa sezione è dedicata alle persone straordinarie che mi hanno sostenuto, non solo durante questi anni di studio, ma nella vita stessa. Grazie, davvero.

La mia più profonda gratitudine va innanzitutto alla mia famiglia. Mi avete sempre supportato, non solo in questi ultimi otto anni, ma per tutta la vita. Vi ringrazio soprattutto per avermi sostenuto nei momenti peggiori, anche quando volevo arrendermi o faticavo nelle cose più basilari, come riuscire a sollevare un bicchiere d'acqua. Questo piccolo ringraziamento non sarà mai abbastanza per esprimere il mio affetto e la mia stima. Spero solo che, seguendo il vostro esempio, io possa diventare una persona che vi somigli almeno un po'.

Un ringraziamento speciale va anche ai miei zii. Avete sempre creduto in me ed eravate felici per ogni mio traguardo. Siete stati i primi a spingermi a intraprendere questo lungo percorso di studi.

Ringrazio poi i miei "Buddies": Jacopo, Paolo e Piero (in ordine alfabetico). Grazie per il vostro supporto, anche quando ero (e spesso sono) intrattabile: mi stupisco ancora della vostra capacità di sopportarmi. Abbiamo condiviso alcuni dei momenti migliori della mia vita (e sì, anche alcuni dei peggiori, specialmente con Jacopo, eppure in qualche modo siamo ancora qui entrambi). So di essere terribile nel mostrare le emozioni, ma ho sinceramente apprezzato tutti i pomeriggi e le serate passate insieme a guardare cose a caso, giocare ai videogiochi o a Magic. Grazie per non aver mai rinunciato alla nostra amicizia.

Un grazie speciale va anche a tutti gli amici incontrati lungo questi otto anni: Ciccio, Giusy, Lorenzo, Marco e Sabato (sempre in ordine alfabetico). Abbiamo vissuto insieme tutte le ansie, le difficoltà e le infinite domande sul perché avessimo scelto questa strada, pensando tante volte di mollare. Ma siamo rimasti uniti, bevendo una quantità quasi incalcolabile di caffè durante le nostre lunghe pause. Non so come alcuni di voi abbiano fatto a sopportare così a lungo le mie continue lamentele. Grazie per aver scelto di essermi amici (un problema che per me rimane indecidibile). Inoltre, voglio estendere la mia gratitudine a tutti gli altri amici incontrati lungo il cammino che non ho menzionato esplicitamente. La vostra presenza e il vostro supporto hanno significato tantissimo per me.

Un ringraziamento speciale va al mio Relatore, Ugo Vaccaro. Il titolo riflette davvero il ruolo: mi ha consigliato eccezionalmente bene durante questi "lunghi-brevi" tre anni. La sua passione per la Scienza è fonte di ispirazione. La ringrazio per tutto ciò che mi ha insegnato e per le storie interessanti e divertenti condivise durante le nostre lunghe sessioni a fissare lavagne quasi vuote. Mi ha mostrato davvero la bellezza della ricerca quando viene fatta nel modo giusto.

Un altro ringraziamento speciale va alle persone meravigliose che ho conosciuto a Okinawa. Ho imparato tantissimo da voi e, cosa più importante, ci siamo divertiti molto insieme. Una menzione particolare va ad Amedeo (o, forse più appropriatamente, Professor Esposito). Mi hai insegnato tantissime cose nuove e mi hai aperto gli occhi su aspetti della ricerca che in Italia non avevo ancora toccato con mano. Voglio ringraziarti davvero per l'enorme sforzo che hai fatto per aiutarmi a capire cosa volessi fare della

mia vita——una domanda a cui non sono ancora sicuro di aver risposto del tutto.

Sono certo che ci siano molte altre persone che dovrei ringraziare, perché senza tutto l'aiuto che ho ricevuto non sarei nulla. Tuttavia, come ben sa chi mi conosce, non sono bravo con le parole. Quindi, concluderò semplicemente dicendo: Grazie.

ABSTRACT

This thesis investigates the deep connection between deterministic search procedures and *variable-length codes* (VLCs), showing that these two domains—traditionally treated as distinct—are fundamentally equivalent. Any search algorithm based on membership or comparison queries naturally induces a binary *variable-length code* (VLC), and conversely, every VLC implicitly represents a search strategy. This correspondence allows classical search problems to be reformulated through the mathematical language of Coding Theory and analyzed using tools from Information Theory.

Within this framework, the thesis investigates several families of alphabetic and prefix codes arising from different structural or cost constraints. First, it establishes new and tighter upper bounds for optimal alphabetic codes and introduces linear-time algorithms that achieve them, yielding improved bounds for *binary search trees* (BSTs) as well. Second, it proposes and analyzes a novel class of codes, denoted as (α, β)-constrained codes, motivated by search problems with asymmetric test outcome costs, and provides polynomial-time algorithms for their construction. Third, it studies prefix codes that use a symbol only as a termination character, deriving linear-time near-optimal construction methods and new entropic bounds. Moreover, it explicitly connects these codes to search procedures using comparison and equality tests.

Overall, the thesis shows that search can be understood as an act of compression: efficient information acquisition and efficient information representation are governed by the same principle of minimizing uncertainty. This perspective provides a powerful, general methodology for translating search problems into code problems.

SOMMARIO

Questa tesi indaga la profonda connessione tra le procedure di ricerca deterministiche e i codici a lunghezza variabile, dimostrando come questi due ambiti — apparentemente distinti — siano in realtà equivalenti. Ogni algoritmo di ricerca basato su test di appartenenza o di confronto induce naturalmente un codice a lunghezza variabile e, viceversa, ogni codice a lunghezza variabile rappresenta implicitamente una strategia di ricerca. Tale corrispondenza consente di riformulare i problemi classici di ricerca nel linguaggio matematico della Teoria dei Codici e di analizzarli mediante strumenti della Teoria dell'Informazione.

All'interno di questo quadro, la tesi analizza diverse famiglie di codici alfabetici e prefissi, caratterizzate da differenti vincoli strutturali o di costo. In primo luogo, stabilisce nuove limitazioni superiori sulla lunghezza media di codici alfabetici ottimi ed introduce algoritmi lineari che ottengono tali limitazioni, usandoli poi anche per derivare nuove limitazioni anche sulla lunghezza media di Alberi Binari di Ricerca. In secondo luogo, propone e analizza una nuova classe di codici, motivata da problemi di ricerca con costi asimmetrici degli esiti dei test, e fornisce algoritmi polinomiali per la loro costruzione. In terzo luogo, esamina codici prefisso che utilizzano un simbolo esclusivamente come carattere di terminazione, derivando metodi di costruzione quasi ottimali in tempo lineare ed anche nuove limitazioni sulla loro lunghezza in termini di entropia. Inoltre, mette in evidenza l'esplicita connessione tra questa nuova classe di codici e le procedure di ricerca che impiegano test di confronto ed eguaglianza.

Nel complesso, la tesi mostra come la ricerca possa essere interpretata come un processo di compressione: l'efficienza nell'acquisizione dell'informazione e l'efficienza nella sua rappresentazione sono governate dallo stesso principio di minimizzazione dell'incertezza. Questa prospettiva fornisce una metodologia generale e potente per tradurre problemi di ricerca in problemi di costruzione di codici.

CONTENTS

LIST OF TABLES

ACRONYMS

VLCs *variable-length codes*

VLC *variable-length code*

BSTs *binary search trees*

BST *binary search tree*

Part I

INTRODUCTION

1

INTRODUCTION

> *"Searching is the most time-consuming part of many programs, and the substitution of a good search method for a bad one often leads to a substantial increase in speed."*
>
> ***— Donald E. Knuth***

In almost every domain of science and human activity, we face problems that require the identification of an unknown object. This act of *searching* — be it for a missing value, a hidden structure, or a correct configuration — takes many forms. It may be a physical search, such as locating a faulty component in a machine, the correct route in a network, or a document in an archive. Similarly, it may be a more abstract search, like a solution to a combinatorial problem, or a correct strategy in a game.

The common objective of such searches is to discover the unknown target as efficiently as possible, where efficiency may be measured in time, cost, number of queries, or simply computational effort. Moreover, depending on the nature of the queries we are allowed to make and on the structure of the search space, we encounter a wide variety of search problems. In one of the most classical settings, the elements are linearly ordered and the allowed queries are binary comparisons: given an unknown element x, we ask whether it is smaller or larger than a known element from the list. This model, which gives rise to the well-known binary search algorithm when all elements are equally likely, is simple but remarkably powerful.

Another interesting variant is the so-called Group testing problem. This problem was originally formulated during World War II by Dorfman in the context of screening soldiers for syphilis and has found applications in areas ranging from quality control in manufacturing to data compression and computational biology. More recently, it gained renewed and unfortunate relevance during the COVID-19 pandemic. The central objective is to identify a small number of defective or infected individuals within a

large population, using as few tests as possible. Instead of testing each individual separately — which may be costly— group testing allows membership queries: we select arbitrary subsets of individuals and ask whether the unknown elements (e.g., infected persons) belong to that subset. Here too, the answer to each query is binary: positive if the subset contains at least one target element, and negative otherwise.

Since in the following our focus is primarily on search procedures based on *comparison* and *membership queries*, each yielding binary outcomes, it is essential to highlight the natural connection that emerges between such search procedures and binary variable-length codes. In the literature, Search Theory and the Theory of Variable-Length Codes are strongly linked together; see [5, 6, 65, 85, 93, 111], as a few examples. A *variable-length code* (VLC) is a mapping that assigns to the symbols of a source alphabet codewords of varying lengths over a target (typically binary) alphabet. Such codes are especially useful when some symbols occur more frequently than others, as shorter codewords can be assigned to more probable symbols to minimize the average code length, i.e. the average cost of the encoding.

In the context of search procedures, *any* search procedure that sequentially executes suitable tests to identify objects within a given search space *inherently* produces a binary variable-length encoding for elements in that space. Specifically, one can represent each potential test outcome using a distinct symbol from a finite code alphabet, and by concatenating these (encoded) test outcomes, one obtains a legitimate encoding of any object within the search domain. In particular, the types of encodings that are most relevant to our settings—those that naturally correspond to membership and comparison queries—are the binary *prefix* codes and the binary *prefix and alphabetic* codes, respectively.

Crucially, there is a deep connection between the *average cost* of such code—measured as the expected codeword length—and the *computational* (or *query*) *complexity*, of the associated search algorithms. In essence, an efficient search strategy corresponds to a code with low average length, and vice versa, a compact code corresponds to a highly efficient search strategy. This connection enables the application of tools from Coding Theory, and,

by extension, Information Theory, to analyze and optimize search procedures.

For example, Shannon's source coding theorem [125] states that the average length of any *uniquely decodable* binary code - including all prefix codes - must be at least the entropy of the source distribution. This provides a universal lower bound on the average number of queries required to identify an unknown element, whether using membership or comparison queries. As a result, entropy serves not only as a measure of information content, but also as a benchmark for the efficiency of search algorithms built on binary queries.

STRUCTURE OF THE THESIS

This thesis is organized as follows.

Chapter 2 provides a comprehensive *Literature Review* that supplies the theoretical foundation for the main topics of the thesis. It begins by defining in Section 2.1 the notations and the preliminary definitions required throughout this work. The chapter then analyzes alphabetic codes, which are central to the main results of this thesis, particularly recalling their strict equivalence to comparison-based search procedures. The chapter surveys the classical algorithms for constructing optimal codes, including those by Gilbert and Moore, Knuth, Hu and Tucker, and Garsia and Wachs (Section 2.3). It concludes by reviewing the conditions for the existence of alphabetic codes (Section 2.4), known upper bounds on their average length (Section 2.5), and various generalizations and variants addressed in the literature (Section 2.6.1).

Chapters 3 to 5 present the novel contributions of the thesis.

Chapter 3 presents the main and most relevant contribution. Specifically, it addresses the classical and well-known problem of constructing almost-optimal alphabetic codes, establishing new and improved upper bounds on the average cost with respect to the current state of the art. Crucially, it also provides linear-time algorithms for constructing codes that achieve these bounds. Finally, the chapter introduces a framework to construct almost-optimal binary search trees from almost-optimal alphabetic codes, improving upon the best-known results in the literature.

Chapter 4 maintains the focus on alphabetic codes but introduces a novel variant of the problem: the so-called (α, β)-constrained codes. This variant is motivated by search problems where test outcomes may have different costs. The chapter provides an efficient dynamic programming algorithm for constructing optimal (α, β)-constrained codes. Then, it focuses on the special case of $(0, 1)$-constrained codes, providing a necessary and sufficient condition for the existence of both alphabetic and prefix codes.

Chapter 5, instead, moves the focus to the more general class of prefix codes. More specifically, it considers the natural problem of constructing codes in which one character is used exclusively as a word terminator to achieve the prefix condition, following the idea of Jaynes [80]. This chapter establishes a fundamental relationship between this novel class of prefix codes and one-to-one codes, providing a linear-time procedure for constructing almost optimal prefix codes using such a delimiter, starting from optimal one-to-one codes. Finally, it provides and derives new entropic lower and upper bounds for the average length of these codes, showing that the imposed constraint becomes negligible as the size of the source alphabet increases.

Finally, Chapter 6 presents the final considerations of the thesis and concludes by outlining a series of interesting open questions and problems in the field of VLCs.

Part II

BACKGROUND

2

LITERATURE REVIEW

This chapter provides the background information readers need to understand our work. We will start by defining the notations used throughout this thesis in Section 2.1, then survey relevant prior research to offer a comprehensive understanding of the main problems we will address later. Part of the information presented herein is drawn from [25].

Since we focus mostly on alphabetic codes, this brief survey is organized as follows. In Section 2.2, we describe the various motivations that led researchers to investigate alphabetic codes. More in particular, in Section 2.2.1, we illustrate in detail the basic correspondence between alphabetic codes and comparison-based search procedures. Historically, this correspondence was the first incentive for the study of alphabetic codes and their properties. In Section 2.2.2, we describe several additional application scenarios where alphabetic codes play an important role.

In Section 2.3, we review the known algorithms to construct optimal alphabetic codes (that is, of minimum average length). We also explain in detail the structure of the most efficient known algorithms with worked examples.

In Section 2.4, we present three necessary and sufficient conditions for the existence of alphabetic codes. These conditions represent, in a sense, the generalizations of the classical Kraft condition for the existence of prefix codes.

In Section 2.5 we describe explicit upper bounds on the average length of optimal alphabetic codes. Interestingly, these upper bounds are often accompanied by *linear* time algorithms to construct alphabetic codes whose average lengths are within such bounds.

In Section 2.6 and Section 2.7 we conclude the chapter by recalling some results about variations and generalizations of the classical problem of constructing alphabetic codes of minimum average length.

2.1 PRELIMINARIES

Let us introduce some notations and concepts that we use often throughout the thesis. Let $S = \{s_1, \dots, s_n\}$ denote a finite set of symbols or arbitrary elements. We represent a probability distribution over n elements as a vector $p = (p_1, \dots, p_n)$, where p_i denotes the probability of the i-th element. When a total order $\prec$ exists on the set of symbols S, i.e., $s_1 \prec \cdots \prec s_n$, we can also denote the associated probability distribution as $p = \langle p_1, \dots, p_n \rangle$ to highlight the inherent order among the symbols. Given a probability distribution $p = (p_1, \dots, p_n)$, we define its *Shannon entropy* as:

$$H(p) = -\sum_{i=1}^{n} p_i \log p_i, \tag{2.1}$$

where, unless otherwise specified, the logarithm is taken base 2.

Let us recall some definitions of binary variable length encodings that we will use later. Let $S = \{s_1, \dots, s_n\}$ be a set of source symbols distributed according to a probability distribution $p = (p_1, \dots, p_n)$. A *binary code* for S is a mapping $w : S \to \{0,1\}^*$ that assigns a binary codeword to each symbol. We denote by $C = \{w(s) : s \in S\}$ the set of all codewords. The *average length* of the code is given by:

$$\mathbb{E}[C] = \sum_{i=1}^{n} p_i \ell(w(s_i)), \tag{2.2}$$

where $\ell(w(s_i))$ denotes the length of the codeword for symbol s_i, that is, the number of bits in $w(s_i)$. Moreover, when the relation is implicit, for simplicity, we denote $\ell(w(s_i))$ by ℓ_i.

We now recall several important classes of binary codes:

Definition 1 (One-to-one Code)**.** *A code $w : S \to \{0,1\}^*$ is a* one-to-one *code if it is a one-to-one mapping, meaning $w(s_i) \neq w(s_j)$ for all $i \neq j$, i.e., it associates a different codeword to each symbol.*

Definition 2 (Prefix Code)**.** *A code $w : S \to \{0,1\}^+$ is a* prefix code (*or prefix-free code*) *if:*

1. *w is a one-to-one mapping,*

2. *for each pair of symbols s_i, s_j with $i \neq j$, the codeword $w(s_i)$ is no prefix of $w(s_j)$.*

Definition 3 (Alphabetic Code)**.** *Given a total order $s_1 \prec \cdots \prec s_n$ on S, a code $w : S \to \{0,1\}^+$ is a* prefix and alphabetic code *if:*

1. *it is a prefix code, i.e., no codeword $w(s_i)$ is a prefix of another $w(s_j)$, for any $s_i, s_j \in S$, $i \neq j$,*

2. *the mapping $w : S \to \{0,1\}^+$ is order-preserving, where the order relation on the set of all binary strings $\{0,1\}^+$ is the standard alphabetical order.*

For the sake of brevity, from this point on a *prefix and alphabetic code* will be simply referred to as an *alphabetic code*. Moreover, we recall that we can view an alphabetic code (and consequently also a prefix code) as a *binary tree T* where each edge is labeled either by bit 0 or 1, and each leaf of the tree corresponds to a different symbol $s \in S$. The codeword $w(s)$ for the symbol $s \in S$ is equal to the concatenation of all the 0's and 1's in the path from the root of T to the leaf associated with s. While the order-preserving property of alphabetic codes implies that the leaves of T, read from the leftmost leaf to the rightmost one, appear in the order $s_1 \prec \cdots \prec s_m$. In the following, we will use the correspondence above described between trees and codes, in the sense that we will freely switch between the terminology of codes and trees, according to which is more suitable for the scenario we will be considering.

2.2 MOTIVATIONS

In the following sub-sections, we present the key motivations and application examples for alphabetic codes. We begin by exploring their fundamental connection to Search Theory, as previously anticipated in Chapter 1. This relationship not only underscores the theoretical importance of alphabetic codes but also highlights their practical utility across various domains.

2.2.1 *Alphabetical Codes and Search Procedures*

Search Theory and the Theory of VLCs are strongly linked [5, 6, 65, 85, 93, 111]. Indeed, *any* search process that sequentially executes suitable tests to identify objects within a given search space *inherently* produces a variable-length encoding for elements in that space. Specifically, one can represent each potential test outcome using a distinct symbol from a finite code alphabet, and by concatenating these (encoded) test outcomes, one obtains a legitimate encoding of any object within the search domain.

More in particular, binary *alphabetic* codes emerge as fundamental combinatorial structures in Search Theory; indeed, alphabetic codes are mathematically *equivalent* to search procedures that operate via binary comparison queries in totally ordered sets. To explain this equivalence, we first recall the formal definition of binary *alphabetic codes* (3).

Definition. *Let $S = \{s_1, \dots, s_m\}$ be a set of symbols, ordered according to a given total order relation $\prec$, that is, for which $s_1 \prec \cdots \prec s_m$ holds. A binary alphabetic code is a mapping $w : \{s_1, \dots, s_m\} \mapsto \{0,1\}^+$, enjoying the following two properties*

- *no codeword $w(s)$ is prefix of another $w(s')$, for any $s, s' \in S$, $s \neq s'$.*
- *the mapping $w : \{s_1, \dots, s_m\} \mapsto \{0,1\}^+$ is order-preserving, where the order relation on the set of all binary strings $\{0,1\}^+$ is the standard alphabetical order,*

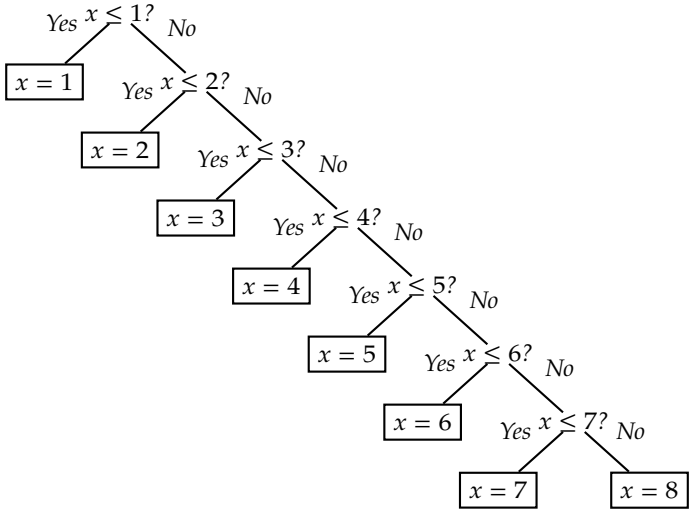
We denote by C the set of codewords

$$C = \{w(s) : s \in S\}.$$

We illustrate how alphabetic codes arise from search algorithms through some examples.

Example 1. *Let $S = \{1, 2, \dots, 8\}$ be the search space. We consider a search algorithm that attempts to determine an unknown element $x \in S$ by asking queries of the form "is $x \leq j$?" for $j = 1, 2, \dots, 8$. The algorithm (and the corresponding answers to queries) can be represented by the following binary tree. Each internal node of the tree corresponds to a query "is $x \leq j$?", and each branch emanating from a node corresponds*

either to the Yes answer to the node query or to the No answer. Each leaf f of the tree corresponds to the (unique) element of S that is consistent with the sequence of Yes/No answers (to the node questions) from the root of the tree to the leaf f.



By encoding the Yes answer to each test with the symbol 0 and the No answer with 1, we get a binary coding $c : \{1, \dots, 8\} \mapsto \{0,1\}^+$, *namely:* $c(1)=0$, $c(2)=10$, $c(3)=110$, $c(4)=1110$, $c(5)=11110$, $c(6)=111110$, $c(7)=1111110$, $c(8)=1111111$, *that is clearly alphabetic. Note that the length of the* i^{th} *codeword corresponds to the number of tests required to determine whether or not the unknown element x is equal to i, for each* $i \in S$.

Conversely, if one had the binary alphabetic coding $c : \{1, \dots, 8\} \mapsto \{0,1\}^+$ *defined above, it would be easy to design an algorithm A that searches successfully in the space* $\{1, \dots, 8\}$. *More precisely, one could partition the search space* $S = \{1, \dots, 8\}$ *in*

$$S_0 = \{i \in S : \text{ the first bit of } c(i) \text{ is } 0\}$$

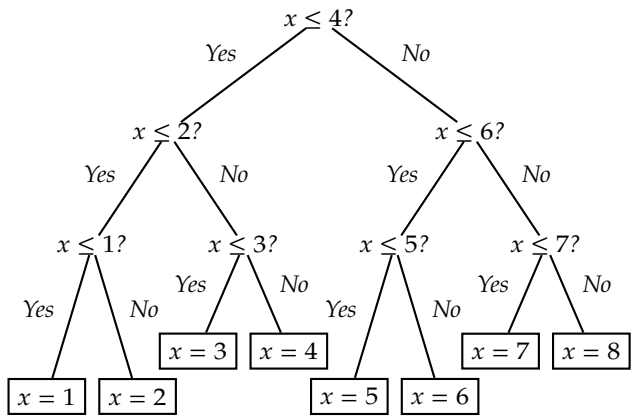
and

$$S_1 = \{i \in S : \text{ the first bit of } c(i) \text{ is } 1\}.$$

Let m be the maximum of the set S_0. *The first query of the algorithm A is "is* $x \leq m$?", *where x is the unknown element in S we are trying to determine. Since the encoding c is alphabetic, we know that both* S_0 *and*

S_1 are made by consecutive elements of S. Therefore, the answer to the query "is $x \leq m$?" allows one to identify the first bit of the encoding of the unknown x. This way to proceed can be iterated either in S_0 or S_1 (according to the query's response) until one gets all bits of the encoding $c(x)$. From the knowledge of $c(x)$, one gets the value of the unknown element x.

Example 2. *One could use a different algorithm to determine an unknown element $x \in S$. For example, a binary search that performs, at each step, the query "is $x \leq j$?", where j is the middle point of the interval that contains x. In this case, the tree representing the algorithm is:*



Again, by encoding the Yes answer for each test with the symbol 0 and the No answer with 1, we get the (different) encoding of $1, \dots, 8$, given by: $b(1)$=000, $b(2)$=001, $b(3)$=010, $b(4)$=011, $b(5)$=100, $b(6)$=101, $b(7)$=110, $b(8)$=111. Also in this case one can see that the obtained encoding $b(\cdot)$ is order-preserving, and therefore alphabetic. As before, from the encoding $b(\cdot)$ one can easily design an algorithm that successfully searches in the space $\{1, \dots, 8\}$. The idea is always the same: The search space $S = \{1, \dots, 8\}$ can be partitioned in

$$S_0 = \{i \in S : \text{ the first bit of } b(i) \text{ is } 0\}$$

and

$$S_1 = \{i \in S : \text{ the first bit of } b(i) \text{ is } 1\}.$$

Since the encoding b is alphabetic, we know that both S_0 and S_1 are made by consecutive elements of S (in our case, $S_0 = \{1, 2, 3, 4\}$ and

$S_1 = \{5, 6, 7, 8\}$). *Let m be the maximum of the set S_0. The first query of the algorithm A is "is $x \leq m$?", where x is the unknown element in S we are trying to determine. According to the answer to the query, the algorithm A will recursively iterate in S_0 or in S_1.*

In general, it holds the following basic result.

Theorem 1 ([5, 6])**.** *Let $S = \{s_1, \ldots, s_m\}$ be a set of elements, ordered according to a given total order relation $\prec$, that is, for which it holds that $s_1 \prec \cdots \prec s_m$.* Any *algorithm A that successfully determines the value of an arbitrary unknown $x \in S$, by means of the execution of tests of the type "is $x \prec s$?", for given $s \in S$, gives rise to a prefix and alphabetic binary encoding of the elements of S.*

Conversely, from any *prefix and alphabetic binary encoding of the elements of S one can construct an algorithm A that successfully determines the value of an arbitrary unknown $x \in S$, by means of the execution of tests of the type "is $x \prec s$?".*

Theorem 1 highlights a deep connection between the efficiency of search algorithms on ordered data and the properties of variable-length encodings. Specifically, it suggests that studying optimal encodings can lead to insights into optimal search strategies, and conversely, well-designed search algorithms reveal characteristics of efficient encodings. For instance, when the search space is embedded in a probability distribution, the theorem implies that the structure of an optimal comparison-based search algorithm directly corresponds to the properties of a binary alphabetic encoding that minimizes the average codeword length (and thus, the average number of comparisons). This equivalence extends to more general classes of search algorithms as well, as shown in [5]. Specifically, membership-based search procedures, which involve queries of the form "$x \in A$?", where A denotes an arbitrary subset of the search space, correspond directly to prefix encodings.

2.2.2 *Additional applications of Alphabetic Codes*

In this section, we highlight other interesting scenarios beyond search theory in which alphabetic codes arise. Gupta *et al.* [63] used alphabetic codes to provide efficient algorithms for the routing lookup problem. Indeed, standard *classless interdomain routing*

requires that a router perform a "longest prefix match" to determine the next hop of a packet. Therefore, given a packet, the lookup operation consists of finding the longest prefix in the routing table that matches the first few bits of the destination address of the packet. In [63], Gupta *et al.* show how alphabetic codes allow one to speed up this kind of lookup operation. Subsequently, Nagaraj [114] improved their analysis, always using alphabetic codes as a basic tool.

In the paper [129], Vaishnav and Pedram apply (variants of) alphabetic trees to problems arising in efficient VLSI (Very Large Scale Integration) design. More specifically, they consider the problem of *fan-out optimization*, whereby one tries to design logical circuits with bounded fan-out. Interestingly, they show that, after appropriate technology-independent optimization, the fan-out optimization problem essentially becomes a tree optimization problem; subsequently, they develop suitable alphabetic tree generation and optimization algorithms, and apply them to the fan-out optimization problem.

In [107, 118, 141] the authors apply ideas, techniques and results about alphabetic codes to the problem of designing efficient algorithms for noiseless fault diagnosis. Similarly, Garey [52] considered the application of alphabetic codes to binary identification procedures that go from machine fault location to medical diagnosis and more. In the paper [62], Graefe discusses the use of alphabetic codes for order-preserving data compression in the implementation of database systems, to the purpose of saving space and bandwidth at all levels of the memory hierarchy. In [49], Gagie exploits the properties of alphabetic codes to design efficient algorithms for the compression of probability distributions. Finally, in [10] Arafat applies alphabetic codes to the problem of efficient design of encryption algorithms.

Alphabetic codes are also useful for the implementation of arithmetic coding. In fact, since binary arithmetic coding is much faster than other types of arithmetic coding, a decision tree (representing an alphabetic code) can be used to reduce an infinite alphabet source into a binary source for fast arithmetic coding, as shown by Marpe *et al.* in [108]. In addition, the basic order preservation property of alphabetic codes is necessary for the ordered repre-

sentation of rational numbers as integers in continued fractions (e.g., see [66, 109]).

Moreover, in [119] Pezza *et al.* used alphabetic codes as a tool for the construction of variable-length unidirectional error-detecting codes with few check symbols.

Alphabetic codes are also used in computational geometry; more precisely, in [120] Preparata and Shamos used alphabetic codes for efficiently locating a point on a line when the query point does not coincide with any of the points dividing the line.

We conclude this section by highlighting the strict relationship between alphabetic codes and binary search trees [93, 113], a widely used and important data structure in computer science. In essence, binary search trees can be considered a generalization of alphabetic codes. This is because the search algorithms underlying binary search trees typically operate using both *comparison and equality tests*. Furthermore, binary search trees account for both *successful* and *unsuccessful* searches, whereas alphabetic codes can be viewed as a specific instance of search trees where the probability of successful searches is implicitly zero. However, we will delve into this relationship in more detail in Section 3.3. For a comprehensive survey on binary search trees and their numerous applications, we refer the reader to the work by Nagaraj [113].

2.3 ALGORITHMS FOR CONSTRUCTING OPTIMAL ALPHABETIC CODES

In this section, we recall the most well-known and widely used algorithms for constructing optimal alphabetic codes, i.e., minimum average length alphabetic codes. We begin by recalling the problem. Let $S = \{s_1, \dots, s_n\}$ be a set of symbols over which we have a total order relation $\prec$, for which it holds $s_1 \prec \cdots \prec s_n$. We also assume that the set S is endowed with a probability distribution $p = \langle p_1, \dots, p_n \rangle$, that is, p_i is the probability of symbol s_i, for $i = 1, \dots, n$. We use the notation $\langle \cdot \rangle$ to emphasize that we are dealing with ordered lists. Moreover, given an alphabetic code

$w : s \in \{s_1, \ldots, s_n\} \mapsto w(s) \in \{0,1\}^+$, we denote the average code length of the code $C = \{w(s) : s \in S\}$ by

$$\mathbb{E}[C] = \sum_{i=1}^{m} p_i \, \ell_i, \tag{2.3}$$

where ℓ_i is the length of the codeword $w(s_i)$. The fundamental problem that we discuss in this section is to find *efficient algorithms* for constructing alphabetic codes for which (2.3) is *minimum*. We recall that the minimum possible value of (2.3) is lower bounded by the Shannon entropy $H(p) = -\sum_i p_i \log p_i$ of p.

The quest for efficient algorithms to construct optimal alphabetic codes has a rich history. Gilbert and Moore, in their classic paper [56], designed a dynamic programming algorithm, of time complexity $O(n^3)$, for the construction of optimal alphabetic codes. Subsequently, Knuth [91] gave an improved $O(n^2)$ algorithm. Significant advancements were made by Hu and Tucker [74], who provided an algorithm of time complexity $O(n \log n)$, with an initial fairly complicated correctness proof that was later slightly simplified by Hu [69]. At the same time, Garsia and Wachs [54] gave a similar algorithm, later shown to be equivalent to the Hu-Tucker algorithm by Nagaraj [113]. Further clarity on the Garsia-Wachs algorithm's analysis was provided by Kingston [87]. More recently, Karpinski, Larmore, and Rytter [84] gave new correctness proofs for both the Garsia-Wachs algorithm and the Hu-Tucker algorithm. Finally, it is worth mentioning also the work [15], in which Belal *et al.* proposed a different algorithm and claimed that it produces optimal alphabetic codes.

Unlike the Huffman method [77], which constructs optimal prefix codes with a relatively straightforward proof of optimality, most of the algorithms mentioned above do not have simple proofs of optimality. In the following, we will focus on explaining the logic and the intuition behind them.

Thus, in the rest of this section, we describe various algorithms, and to aid in the explanation, we use a practical example. Specifically, we will use $n = 11$ and the following probability distribution

$$p = \langle 0.24, 0.12, 0.09, 0.08, 0.04, 0.02, 0.03, 0.06, 0.14, 0.11, 0.07 \rangle \tag{2.4}$$

on the set of symbols S.

2.3.1 *Gilbert and Moore's algorithm*

The Gilbert-Moore algorithm for the construction of optimal alphabetic codes is a dynamic programming algorithm. The algorithm breaks down the problem into smaller subproblems. Therefore, we define the subproblem $S(i,j)$ as the construction of an optimal alphabetic code (or tree) for the contiguous sequence of symbols $s_i, \dots, s_j$, with $1 \leq i \leq j \leq n$, and similarly with $C(i,j)$ we define its cost, i.e., its average length. The overall problem is $S(1,n)$, that is, the one that has the minimum average length, $C(1,n)$. The base cases are the following two:

- Subproblem with a single symbol ($i = j$): For a single symbol s_i, the cost $C(i,i)$ is 0, since there is no need to encode;
- Subproblem with two symbols ($j = i+1$): For those subproblems, the optimal code assigns the codewords "0" and "1" to the symbols s_i and s_{i+1}. This corresponds to an optimal tree structure consisting of a root node with two children, which are leaves representing s_i and s_{i+1}. Thus, the cost for these subproblems are $C(i,i+1) = p_i + p_{i+1}$, for $i = 1, 2, \dots, n-1$.

These initial costs for $n = 11$ are shown in Table 2.1.

$i\backslash j$	1	2	3	4	5	6	7	8	9	10	11
1	0	p_1+p_2									
2		0	p_2+p_3								
3			0	p_3+p_4							
4				0	p_4+p_5						
5					0	p_5+p_6					
6						0	p_6+p_7				
7							0	p_7+p_8			
8								0	p_8+p_9		
9									0	p_9+p_{10}	
10										0	$p_{10}+p_{11}$
11											0

Table 2.1: Initial matrix for the dynamic programming algorithm ($n = 11$, costs are multiplied by 100 for readability).

For the general case, the optimal tree/code for the subproblem $S(i,j)$ can be found by considering all possible ways of splitting the sequence of symbols $s_i, \dots, s_j$ into two. Each split divides

the sequence into a *left subtree* (containing $s_i, \ldots, s_k$) and a *right subtree* (containing $s_{k+1}, \ldots, s_j$), ensuring at least one element in each subtree. Thus, there are exactly $j - i$ ways to perform such a split, namely $s_i, \ldots, s_k$ on the left and $s_{k+1}, \ldots, s_j$ on the right, for $k = i, i+1, \ldots, j-1$. The cost of the tree produced by the split for a given value k is

$$C(i,j) = \sum_{\ell=i}^{j} p_\ell + C(i,k) + C(k+1,j).$$

As an illustration, consider the computation of $C(1,3)$ for our instance (2.4). Here, we must consider the two possible splits of the sequence s_1, s_2, s_3:

1. Split at $k = 1$: This forms a left subtree with s_1, and a right subtree with s_2 and s_3. The cost of such a split is:

$$(p_1 + p_2 + p_3) + C(1,1) + C(2,3) = p_1 + 2p_2 + 2p_3.$$

2. Split at $k = 2$: This forms a left subtree with s_1, s_2, and a right subtree with s_3. The cost in this case is:

$$(p_1 + p_2 + p_3) + C(1,2) + C(3,3) = 2p_1 + 2p_2 + p_3.$$

The minimum cost for $S(1,3)$ is then determined by selecting the minimum among the two. This dynamic programming approach allows us to fill the cost table with an $O(n^3)$-time algorithm. Furthermore, straightforward bookkeeping allows one to reconstruct the optimal tree/code, as illustrated in Algorithm 1.

$i\backslash j$	1	2	3	4	5	6	7	8	9	10	11
1	0	36	66	99	123	135	152	182	236	283	322
2		0	21	46	66	76	90	116	162	209	242
3			0	17	33	43	57	78	120	160	192
4				0	12	20	31	51	88	124	156
5					0	6	14	29	58	94	123
6						0	5	16	41	77	102
7							0	9	32	66	91
8								0	20	51	76
9									0	25	50
10										0	18
11											0

$i\backslash j$	1	2	3	4	5	6	7	8	9	10	11
1		1	1	1	1	1	1	2	3	3	3
2			2	2	2	3	3	3	4	4	7
3				3	3	3	3	4	5	8	8
4					4	4	4	5	7	8	8
5						5	5	7	8	8	9
6							6	7	8	8	9
7								7	8	9	9
8									8	9	9
9										9	9
10											10
11											

Table 2.2: Costs (left) and roots indexes (right) matrices. (Costs are multiplied by 100)

Algorithm 1: Gilbert-Moore algorithm

Input: Symbols $S = \{s_1, \dots, s_n\}$ and $p = \langle p_1, \dots, p_n \rangle$ the associated probability distribution.

$C(1,1) = 0$
for $i \leftarrow 2$ ***to*** n **do**
  $C(i,i) = 0$
  $C(i-1,i) = p_{i-1} + p_i$
for $s \leftarrow 2$ ***to*** $n-1$ **do**
  for $i \leftarrow 1$ ***to*** $n-s$ **do**
    $j = i + s$; `// Proceed by diagonals`
    $min = \infty$
    $minindex = -1$
    for $k \leftarrow i$ ***to*** $j-1$ **do**
      if $C(i,k) + C(k+1,j) < min$ **then**
        $min = C(i,k) + C(k+1,j)$
        $minindex = k$
    $C(i,j) = \sum_{k=i}^{j} p_k + min$
    $R(i,j) = minindex$

Output: $C(1,n)$ and R

Table 2.2 shows the values obtained for the probability distribution p. For enhanced readability, all costs within the table are scaled by a factor of 100. The cost of an optimal tree is 3.22. The matrix R is crucial for reconstructing the optimal tree/code since it keeps track of the indices of the optimal splits. For example, the entry $R(1,11) = 3$ indicates that the first split of the optimal tree occurs after the third symbol, i.e. $s_1, \dots, s_3 : s_4, \dots, s_{11}$.

$i\backslash j$	1	2	3	4	5	6	7	8	9	10	11
1		1	1-2	1-3	1-4	1-5	1-6	1-7	1-8	1-9	1-10
2			2	2-3	2-4	2-5	2-6	2-7	2-8	2-9	2-10
3				3	3-4	3-5	3-6	3-7	3-8	3-9	3-10
4					4	4-5	4-6	4-7	4-8	4-9	4-10
5						5	5-6	5-7	5-8	5-9	5-10
6							6	6-7	6-8	6-9	6-10
7								7	7-8	7-9	7-10
8									8	8-9	8-10
9										9	9-10
10											10
11											

$i\backslash j$	1	2	3	4	5	6	7	8	9	10	11
1		1	1-2	1-2	1-2	1-3	1-3	1-3	2-4	2-4	3-7
2			2	2-3	2-3	2-3	3-3	3-4	3-5	4-8	4-8
3				3	3-4	3-4	3-4	3-5	4-7	5-8	8-8
4					4	4-5	4-5	4-7	5-8	7-8	8-9
5						5	5-6	5-7	7-8	8-8	8-9
6							6	6-7	7-8	8-9	8-9
7								7	7-8	8-9	9-9
8									8	8-9	9-9
9										9	9-10
10											10
11											

Table 2.3: Search intervals of root indexes of Gilbert and Moore's algorithm (left) and of Knuth's improvement on the input of the previous example (right).

2.3.2 *Knuth's algorithm*

Knuth demonstrated that the search for the root of an optimal subtree, a process which initially requires $j - i$ iterations for each subproblem $S(i, j)$ (corresponding to the **for** loop at line 9 in Algorithm 1), can be improved. This optimization is achieved through the application of an inequality, subsequently formalized as the *Knuth-Yao Quadrangle inequality*[91, 137], which we will discuss further in Chapter 2.4.

Specifically, Knuth proved that the root $R(i, j)$ of an optimal tree for the subproblem $S(i, j)$ is guaranteed to lie within the interval defined by the roots of the previously computed smaller optimal subtrees. Thus, instead of searching from i to $j - 1$, it is sufficient to search only from $R(i, j - 1)$ through $R(i + 1, j)$. This means that the for loop at line 9 can be changed to

$$\textbf{for } k = R(i, j - 1) \textbf{ to } R(i + 1, j) \textbf{ do}$$

with savings on the total execution time that lowers the time complexity of the algorithm to $O(n^2)$.

To better visualize this saving, we report in Table 2.3 the search intervals for the Gilbert-Moore dynamic programming algorithm and the search intervals of Knuth's improvement on the input of the previous example. For the Gilbert-Moore algorithm, the size of the interval is fixed (because it depends only on the indexes i and j and in each iteration the size grows by 1), while for the Knuth's algorithm it depends on the input that determines the roots of the subtrees, and it is smaller.

2.3.3 *Hu and Tucker's algorithm*

The Hu-Tucker algorithm constructs an optimal alphabetic code/tree through a two-phase process. It first builds an intermediate tree T', which does not necessarily preserve the original alphabetical order of the symbols. Subsequently, it leverages the structure of T' to build the final tree T, which preserves the original order.

First phase

Let us start by describing the first phase in which the tree T' is built. The construction of T' is somewhat similar to the construction of a Huffman tree, for which the two smallest probabilities are repeatedly merged together. However, there are two crucial differences. First, the merging operation is subject to an order-preserving constraint: two probabilities (or nodes) can be merged together only if, in the ordered list maintained during the construction, there are no nodes that correspond to single symbols, i.e., leaves that have not yet been merged, in between the two probabilities; this constraint is vacuously satisfied for consecutive probabilities in the list. Moreover, we will say that two probabilities (or nodes) are *joinable* if they satisfy the condition. Second, since we are dealing with an ordered list, it is crucial to specify where the new element is placed; when joining two probabilities, the resulting probability takes the place of the "left" one, while the "right" one gets deleted. Finally, in case of a tie, that is, when there are two pairs of joinable nodes with the same minimal sum, the algorithm always chooses the leftmost one.

The construction starts by initializing a forest of n single-node trees, where each leaf corresponds to a symbol/probability. To clarify the construction, we will use an example alongside the description of the steps. Figure 2.1 shows the initial forest for the probability distribution p. To easily visualize the constraint that makes two nodes joinable, nodes that correspond to leaves are depicted as squares and internal nodes as circles: two nodes are joinable if in the list there are no squares in between them.

Figure 2.1: Initial forest. Probabilities are multiplied by 100 to ease the drawing and the reading.

In the first step, all pairs of consecutive leaves, and only those pairs, are joinable due to the first constraint. Thus, the nodes that will be joined are [2] and [3] because they give the smallest sum. Figure 2.2 shows the resulting forest. In the figures, we show both the ordered list of nodes that have to be joined (the top row), and the nodes that have already been joined, by attaching to each node in the list the subtree created by the joining process.

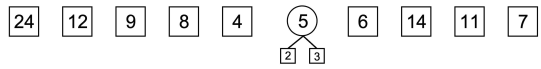


Figure 2.2: List of nodes after step 1 of the Hu-Tucker algorithm.

Node ⑤ takes the place of node [2], while node [3] is deleted from the sequence of nodes that have to be joined. Recall that the newly created node takes the place of the leftmost node among the joined nodes (in this case, it does not make a difference since the two joined nodes are adjacent). The list of nodes that have to be joined is now ⟨[24], [12], [9], [8], [4], ⑤, [6], [14], [11], [7]⟩. In the second step, the pairs of nodes that are joinable are all consecutive pairs of nodes, but now nodes [4] and [6] are joinable too, since in between them there are no squares (leaves). The smallest sum is given by the pair [4] and ⑤ whose joining gives the forest shown in Figure 2.3.

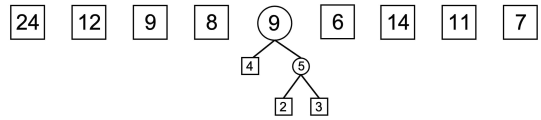


Figure 2.3: List of nodes after step 2 of the Hu-Tucker algorithm.

The newly created node takes the position of the node [4] in the list of remaining nodes.

For the next step, the list of nodes is ⟨[24], [12], [9], [8], ⑨, [6], [14], [11], [7]⟩ and thus the joinable nodes are all the consecutive pairs of nodes and the pair [8] and [6]. And exactly this last pair is the one that gives the smallest sum. Their joining produces the forest shown in Figure 2.4, with the new node taking the place of node [8]. Notice that this step causes the order of the leaves to be disrupted, as node [6] has moved to the left of node [4].

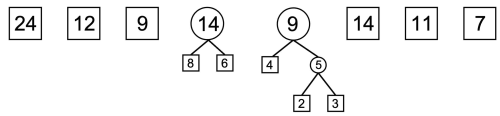


Figure 2.4: List of nodes after step 3 of the Hu-Tucker algorithm.

In step 4, the joinable nodes are all the consecutive pairs of nodes, and the pairs $\boxed{9}$ and $\textcircled{9}$, $\boxed{9}$ and $\boxed{14}$ and, $\textcircled{14}$ and $\boxed{14}$. In this case, we have that the smallest sum is 18 and is achieved by $\boxed{9}$ and $\textcircled{9}$ and by $\boxed{11}$ and $\boxed{7}$. The algorithm chooses the leftmost pair, which is $\boxed{9}$ and $\textcircled{9}$, producing the forest shown in Figure 2.5.

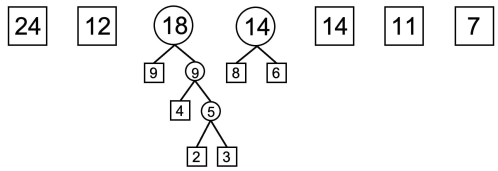


Figure 2.5: List of nodes after step 4 of the Hu-Tucker algorithm.

For the next step, the set of joinable pairs is again all the consecutive pairs of nodes and the pairs $\boxed{12}$ and $\textcircled{14}$, $\boxed{12}$ and $\boxed{14}$ and, $\textcircled{18}$ and $\boxed{14}$. The smallest sum is given by the nodes $\boxed{11}$ and $\boxed{7}$, and their joining produces the forest shown in Figure 2.6.

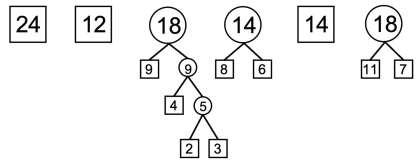


Figure 2.6: List of nodes after step 5 of the Hu-Tucker algorithm.

For step 6, the joinable pairs are all consecutive nodes, and the pairs $\boxed{12}$ and $\textcircled{14}$, $\boxed{12}$ and $\boxed{14}$, and $\textcircled{18}$ and $\boxed{14}$. And the leftmost smallest sum is obtained by joining the two nodes $\boxed{12}$ and $\textcircled{14}$, resulting in the forest shown in Figure 2.7.

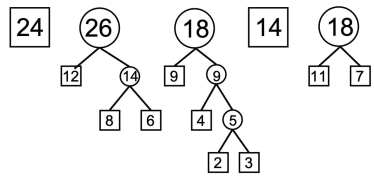


Figure 2.7: List of nodes after step 6 of the Hu-Tucker algorithm.

For step 7, the joinable pairs are all consecutive nodes, and the pairs [24] and (18), [24] and [14], and (26) and [14]. And the leftmost smallest sum is obtained by joining the two nodes (18) and [14], resulting in the forest shown in Figure 2.8.

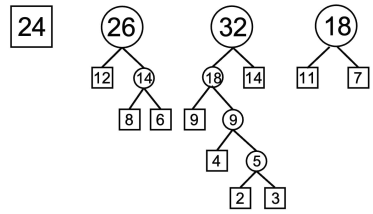


Figure 2.8: List of nodes after step 7 of the Hu-Tucker algorithm.

Now, all pairs of nodes are joinable, and thus the construction proceeds as in the construction of a Huffman tree, joining first [24] and (18), then (26) and (32), and finally (42) and (58). The resulting intermediate tree T' is shown in Figure 2.9.

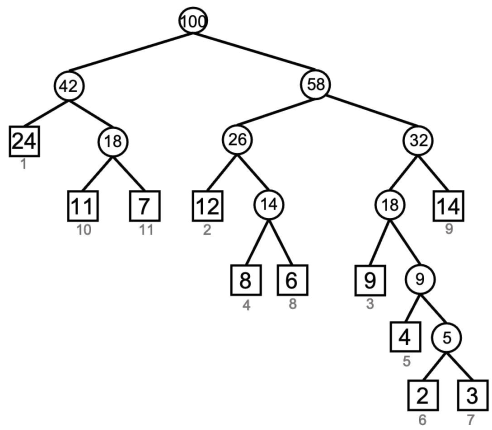


Figure 2.9: Intermediate tree built T' by the Hu-Tucker algorithm.

Now, from the intermediate tree T', we can proceed with the second phase. A direct consequence of allowing the merging of non-adjacent nodes in the construction of T' is that the order of the leaves may no longer correspond to the initial alphabetic order (as shown in our example). However, the crucial information that we need from T' is not its topology, but rather the set of root-to-leaf path lengths, specifically the lengths $\langle \ell_1, \ell_2, \dots, \ell_n \rangle$, where ℓ_i denotes the length of the root-to-leaf path for symbol s_i.

Second phase

Indeed, a fundamental result by Hu and Tucker [74] demonstrates that, despite T' being non-alphabetic, there *exists* an alphabetic tree T whose codeword lengths are precisely $\langle \ell_1, \ell_2, \dots, \ell_n \rangle$. Consequently, the cost of this alphabetic tree T is equal to that of T' and thus optimal.

In our running example, by re-ordering the lengths to match the initial ordering of the symbols, we obtain $\langle \ell_1, \ell_2, \dots, \ell_{11} \rangle = \langle 2, 3, 4, 4, 5, 6, 6, 4, 3, 3, 3 \rangle$. The corresponding optimal alphabetic tree, constructed by using these lengths, is shown in Figure 2.10. It is worth noting that knowing the lengths $\langle \ell_1, \ell_2, \dots, \ell_n \rangle$, one can easily build the tree in linear time, just by using always the leftmost available path.

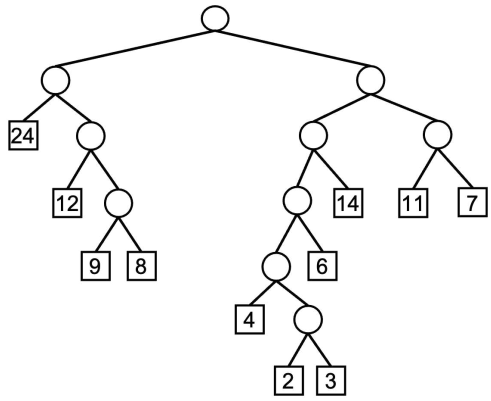


Figure 2.10: Final optimal alphabetic tree built by the Hu-Tucker algorithm.

For the reader's convenience, we summarise the *modus operandi* of the Hu-Tucker algorithm in Algorithm 2.

Algorithm 2: Hu-Tucker algorithm

Input: Symbols $S = \{s_1, \dots, s_n\}$ and $p = \langle p_1, \dots, p_n \rangle$ the associated probability distribution.

1 Build the intermediate tree T' as follows. Start with an ordered forest of one-node trees corresponding to the n probabilities. Consider two nodes of the list to be *joinable* if there are no nodes that correspond to leaves in between them.

2 **repeat**

3 Find the leftmost pair of joinable nodes that gives the smallest sum;

4 Merge the two nodes, keeping the merged node in the position of the left node of the pair

5 **until** *there is only one tree*;

6 Let ℓ_i be the length of the root-to-leaf paths in T' for symbol s_i

7 Build the final tree T with leaves at levels $\langle \ell_1, \ell_2, \dots, \ell_n \rangle$, with leaf i associated to symbol s_i.

Output: The tree T

We conclude this section by mentioning that a detailed implementation of the Hu-Tucker algorithm was given in [28, 142]. Moreover, the procedure given in [142] finds a minimal cost tree whose longest path length and total path length are minimal.

2.3.4 *Garsia and Wachs' algorithm*

Similar to the Hu-Tucker algorithm, the Garsia-Wachs algorithm is divided into two phases: it first builds an intermediate tree (which does not necessarily preserve the alphabetic property), and then utilizes the path lengths of the intermediate tree to build the final alphabetic tree. While the overall structure is similar, the construction of the intermediate tree in Garsia-Wachs diverges from Hu-Tucker. Perhaps the construction of the intermediate tree is somewhat simpler since there is no need to distinguish between joinable and non-joinable nodes. The second phase remains identical between the two approaches.

As in the Hu-Tucker algorithm, we start from the initial (ordered) list of probabilities, and we perform $n - 1$ steps. In each

step, we join two probabilities and move the resulting node to an appropriate position in the new list. The rule used to select the two nodes to be joined is what differentiates the Garsia-Wachs algorithm from the Hu-Tucker algorithm. The Garsia-Wachs algorithm joins the two rightmost consecutive nodes for which the sum of the probabilities is the smallest. Then, the newly created element, which in the tree will be the parent of the two nodes that have been joined, is moved to the right of the current position, placing it just before the first node whose probability is greater than or equal to its probability; or at the end of the list if there is no such node. More formally, let

$$\langle p_1, p_2, \cdots \cdots \cdots, p_m \rangle$$

be the ordered list of probabilities for a generic step of the algorithm (where $m = n$ initially and will decrease by 1 at each iteration). Let p_i, p_{i+1} be the rightmost consecutive probabilities whose sum is the minimum possible over all the consecutive pairs. Let p_k, $k \geq i+2$, be the first probability such that $p_k \geq p_i + p_{i+1}$. If such a probability exists, the new list of $m-1$ probabilities is

$$\langle p_1, \cdots, p_{i-1}, p_{i+2}, \cdots, p_{k-1}, (p_i + p_{i+1}), p_k, p_{k+1}, \cdots, p_m \rangle.$$

If p_k does not exist, the new probability is moved to the end of the list, that is, the new list is

$$\langle p_1, \cdots, p_{i-1}, p_{i+2}, \cdots, p_{m-1}, p_m, (p_i + p_{i+1}) \rangle.$$

After $n-1$ steps, the intermediate tree is built.

Let us clarify the construction of the intermediate tree according to the Garsia-Wachs algorithm with an example. We consider the same probability distribution p that we have used for the previous one. The initial list is the same as for the Hu-Tucker algorithm, that is, the one depicted in Figure 2.1. The consecutive pair of probabilities with the smallest sum is $\boxed{2}$ and $\boxed{3}$, and thus they get joined. Moreover, the first probability on the right side of the joined probability is $\boxed{6}$; thus, the new node $\textcircled{5}$ will be placed right before $\boxed{6}$, leading to the same list of the Hu-Tucker algorithm depicted in Figure 2.2. For the next step, the Garsia-Wachs algorithm behaves differently. Indeed, the consecutive pair of probabilities whose sum is minimum is $\boxed{4}$ and $\textcircled{5}$. This creates the new probability $\textcircled{9}$

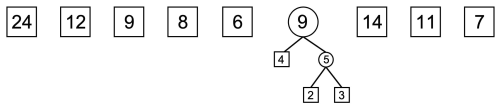


Figure 2.11: List of nodes after step 2 of the Garsia-Wachs algorithm

and the first probability greater than 9 is 14. Thus, the new list is the one shown in Figure 2.11.

Now the smallest sum is given by 8 and 6, and the first probability greater than or equal to their sum is 14, so the newly created node (14), will be placed right before 14, as shown in Figure 2.12.

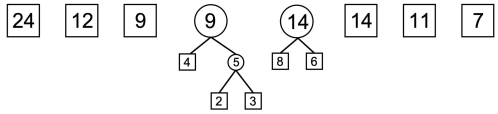


Figure 2.12: List of nodes after step 3 of the Garsia-Wachs algorithm

The rightmost smallest sum is now given by 11 and 7 and the new node will be placed at the end of the list, as shown in Figure 2.13 (notice that also (9) and 9 give 18 as sum, but the algorithm takes the rightmost pair).

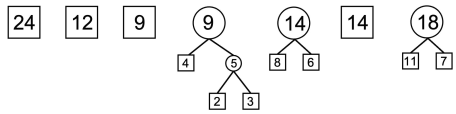


Figure 2.13: List of nodes after step 4 of the Garsia-Wachs algorithm

For the next step, the smallest sum is obtained by joining 9 and (9), an operation that creates the node (18). The first node to their right with a probability equal to or greater than 18 is the last node of the list (18), thus the newly created node will be placed just before the last one, as depicted in Figure 2.14.

The next step will join 12 and (14) creating a new node (26) that will be placed at the end of the list as shown in Figure 2.15.

The next step will join 14 and (18) creating a new node (32) that will be placed at the end of the list as shown in Figure 2.16.

The subsequent step will join 24 and (18) creating a new node (42) that will be placed at the end of the list as shown in Figure 2.17.

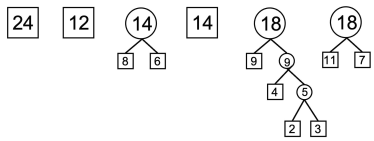


Figure 2.14: List of nodes after step 5 of the Garsia-Wachs algorithm

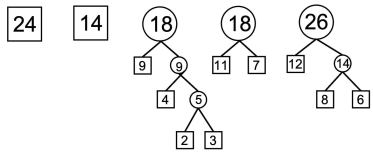


Figure 2.15: List of nodes after step 6 of the Garsia-Wachs algorithm

Step 9 joins (26) and (32), as shown in Figure 2.18, and the final step gives the intermediate tree shown in Figure 2.19.

From this intermediate tree, following the same procedure as in the Hu-Tucker algorithm, we extrapolate the lengths of the codewords associated with each symbol. We then reorder them to match the initial ordering of the symbols.

For example, in our specific intermediate tree, node [24] has length 2, node [12] has length 3, node [9] has length 4, and so forth. This process yields the vector of lengths $\langle \ell_1, \ldots \ell_{11} \rangle = \langle 2, 3, 4, 4, 5, 6, 6, 4, 3, 3, 3 \rangle$. This is the same length vector obtained by the intermediate tree of the Hu-Tucker algorithm (although the intermediate trees are slightly different); hence, the final optimal alphabetic tree is the same as that of the Hu-Tucker algorithm, already shown in Figure 2.10.

As done for the Hu-Tucker algorithm, we summarise the *modus operandi* of the Garsia-Wachs algorithm in the pseudocode of Algorithm 3.

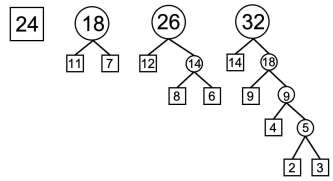


Figure 2.16: List of nodes after step 7 of the Garsia-Wachs algorithm

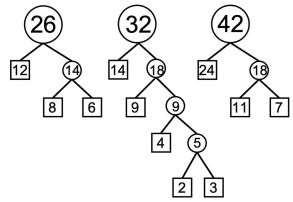


Figure 2.17: List of nodes after step 8 of the Garsia-Wachs algorithm

Algorithm 3: Garsia-Wachs algorithm

Input: Symbols $S = \{s_1, \dots, s_n\}$ and $p = \langle p_1, \dots, p_n \rangle$ the associated probability distribution.

1 Build the intermediate tree T' as follows.
2 Start with an ordered forest of trees with one node corresponding to the n probabilities.
3 **repeat**
4 Find the rightmost pair of consecutive nodes p_i, p_{i+1} that gives the smallest sum in the current list $p_1, \dots, p_m$;
5 Let $k \geq i + 2$ be such that p_k is the first probability satisfying $p_k \geq p_i + p_{i+1}$;
6 If such k exists, the new list is $p_1, \dots, p_{i-1}, p_{i+2}, \dots, p_{k-1}, (p_i + p_{i+1}), p_k, p_{k+1}, \dots, p_m$.
7 If such k does not exist, the new list is $p_1, \dots, p_{i-1}, p_{i+2}, \dots, p_{m-1}, p_m, (p_i + p_{i+1})$.
8 **until** *there is only one tree*;
9 Let ℓ_i be the length of the root-to-leaf paths in T' for symbol s_i
10 Build the final tree T with leaves at levels $\langle \ell_1, \ell_2, \dots, \ell_n \rangle$, with leaf i associated with the symbol s_i.

Output: The tree T

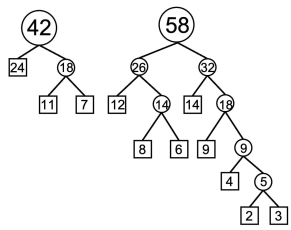


Figure 2.18: List of nodes after step 9 of the Garsia-Wachs algorithm

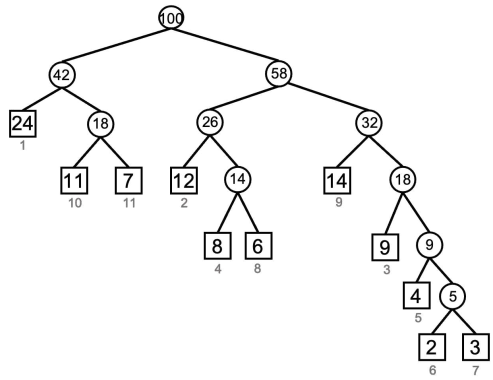


Figure 2.19: The intermediate tree built by the Garsia-Wachs algorithm

We conclude this section by mentioning that Bird [19] provided an $O(n \log n)$ implementation of the Garsia-Wachs algorithm in the framework of functional programming.

2.4 CONDITIONS FOR THE EXISTENCE OF ALPHABETIC CODES

Having discussed algorithms for constructing optimal alphabetic code in the previous section, we now shift our focus to the fundamental question of their existence. This section explores the sufficient and necessary conditions for an alphabetic code to exist, drawing a parallel with the well-known Kraft inequality for general prefix codes. Indeed, given a multiset of integers $\{\ell_1, \dots, \ell_n\}$, the Kraft inequality states that there exists a binary prefix code with codeword lengths $\ell_1, \dots, \ell_n$ if and only if it holds that

$$\sum_{i=1}^{n} 2^{-\ell_i} \leq 1. \tag{2.5}$$

It is natural to ask whether similar conditions hold also for prefix and alphabetic codes. As expected, the answer is positive, but the conditions are considerably more complicated than (2.5).

Since the ordering of the codeword-to-symbol association is a hard constraint on alphabetic codes, we recall the general setup. Let $S = \{s_1, \dots, s_m\}$ be a set of symbols and $\prec$ be a total order relation on S, that is, for which we have $s_1 \prec \cdots \prec s_m$. Given a list of integers $L = \langle \ell_1, \dots, \ell_n \rangle$, we ask under which conditions there exists an alphabetic code $w : S \mapsto \{0,1\}^+$ that assigns a codeword of length ℓ_i to the symbol s_i, for $i = 1, \dots, n$.

For the sake of completeness, it is worth recalling a preliminary, albeit weaker, condition established by Ahlswede and Wegener in [5]. They showed that for a given a list of positive integers $L = \langle \ell_1, \dots, \ell_n \rangle$, if

$$\sum_{i=1}^{n} 2^{-\ell_i} \leq \frac{1}{2}, \tag{2.6}$$

then there exists an alphabetic code with codeword lengths L. However, this condition is sufficient but not necessary. Intuitively, the reason why such a condition is not necessary is that it does not take into account the ordering of lengths imposed by the alphabetic property.

The first necessary and sufficient condition for the existence of alphabetic codes was provided by Yeung [139]. To properly describe Yeung's result, we need to introduce some preliminary definitions. Let $\mathsf{c} : (\mathbb{R}_+, \mathbb{R}_+) \to \mathbb{R}_+$ be a mapping defined as

$$\mathsf{c}(a,b) = \left\lceil \frac{a}{b} \right\rceil b.$$

Definition 4 ([139]). *For any list of positive integers $L = \langle \ell_1, \dots, \ell_n \rangle$, define the numbers $\mathsf{s}(L,k)$ as*

$$\mathsf{s}(L,k) = \begin{cases} 0 & \text{if } k = 0, \\ \mathsf{c}(\mathsf{s}(L,k-1), 2^{-\ell_k}) + 2^{-\ell_k} & \text{if } 1 \leq k \leq n. \end{cases}$$

Yeung proved the following result:

Theorem 2 ([139]). *There exists a binary alphabetic code with codeword lengths $L = \langle \ell_1, \dots, \ell_n \rangle$ if and only if $\mathsf{s}(L,n) \leq 1$.*

Intuitively, Yeung's inequality views the unit interval $[0, 1]$ (or equivalently, the interval $[0, 2^{\ell_{\max}}]$, with $\ell_{\max} = \max_i \ell_i$) as the total "coding space" available for constructing an alphabetic prefix code. Each codeword k is assigned a "width" $w_k = 2^{-\ell_k}$, and the function $\mathtt{s}(L, k)$ represents the cumulative ending position of the k^{th} codeword. The key idea is that the alphabetic constraint forces the codewords to appear in order, while the prefix constraint forces each codeword to begin at a position in the interval that is aligned with its own width-that is, at a point of the form $d \cdot 2^{-\ell_k}$ for some integer d. The function $\mathtt{c}$ models this by "rounding up" the starting position of each new codeword to the earliest allowed point, a process that often introduces unavoidable gaps between codewords (gaps that we will exploit in the improvement presented in Chapter 3). The final condition $\mathtt{s}(L, n) \leq 1$ checks whether all codewords, together with any such forced gaps, fit within the whole interval.

For example, the list $L = \langle 3, 1, 3 \rangle$ satisfies the standard Kraft inequality

$$\frac{1}{8} + \frac{1}{2} + \frac{1}{8} \leq 1,$$

but fails the Yeung's inequality. The first codeword (width $1/8$) ends at $\mathtt{s}(L, 1) = 1/8$. The second codeword (width $1/2$) must start at the next available multiple of $1/2$, namely $1/2$, ending at $\mathtt{s}(L, 2) = 1$. The third codeword (width $1/8$) must then start at 1, ending at $\mathtt{s}(L, 3) = 9/8$. Since $9/8 > 1$, the construction overflows the available space, and thus no such alphabetic code exists.

A similar and equivalent condition, following the same idea, was later provided by Nakatsu in [115]. We first recall the following definitions.

Definition 5. *For a binary fraction x and integer $i \geq 1$, let the function* `trunc` *be defined as*

$$\mathtt{trunc}(i, x) = \frac{\lfloor 2^i x \rfloor}{2^i}, \tag{2.7}$$

that is, $\mathtt{trunc}(i, x)$ *is the fraction obtained by considering only the first i bits in the binary representation of x.*

Definition 6 ([115]). *Let $L = \langle \ell_1, \dots, \ell_n \rangle$ be a list of positive integers. Let $\alpha_i = \min(\ell_{i-1}, \ell_i)$, for $i = 2, \dots, n$. Define the following recursive function* `sum` *as*

$$\texttt{sum}(L, i) = \begin{cases} \texttt{trunc}(\alpha_i, \texttt{sum}(L, i-1)) + 2^{-\alpha_i} & \text{if } i \geq 2, \\ 0 & \text{if } i = 1. \end{cases} \tag{2.8}$$

Nakatsu proved the following result, which we will make use of in Chapter 3.

Theorem 3 ([115]). *There exists a binary alphabetic code with codeword lengths $L = \langle \ell_1, \dots, \ell_n \rangle$ if and only if* $\texttt{sum}(L, n) < 1$.

Subsequently, a different necessary and sufficient condition was introduced by Sheinwald in [126]. As for the previous conditions, we need to introduce some preliminary definitions.

Definition 7 ([126]). *Let $L = \langle \ell_1, \dots, \ell_n \rangle$ be a list of positive integers. For a binary fraction x and integer $i \geq 1$, let the function* `t` *be defined as*

$$\texttt{t}(i, x) = \texttt{trunc}(i, x) + \frac{\lceil x - \texttt{trunc}(i, x) \rceil}{2^i},$$

that is, $\texttt{t}(i, x)$ *is equal to x if* $\texttt{trunc}(i, x) = x$, *and to* $\texttt{trunc}(i, x) + 2^{-i}$ *otherwise. Moreover, let φ be the following function*

$$\varphi(L, i) = \begin{cases} \texttt{t}(\ell_i, \varphi(L, i-1)) + 2^{-\ell_i} & \text{if } 2 \leq i \leq n, \\ 2^{-\ell_1} & \text{if } i = 1. \end{cases}$$

Sheinwald provided the following result.

Theorem 4 ([126]). *There exists a binary alphabetic code with codeword lengths $L = \langle \ell_1, \dots, \ell_n \rangle$ if and only if $\varphi(L, n) \leq 1$.*

Although clearly equivalent, there might be scenarios where one of the conditions stated in Theorems 2, 3, and 4 could be more easily applicable than the others, depending on the specific context.

We observe that Theorem 3 is the only one whose condition must be strictly less than 1. This difference arises from how the condition maps the interval $[0, 1]$ to binary codewords.

We conclude this section with an extension of the above results to the case where the codewords of the alphabetic code must respect some additional constraints [44]. Namely, by looking at the lengths of the codewords ℓ_i as the lengths of the root-to-left paths in the tree that represents the code, one can consider the number of "left" edges, l_i and the number of "right" edges r_i, whose sum gives $\ell_i = l_i + r_i$. We can define the path vector $\bar{v}$ as the ordered set of pairs of $\langle (l_1, r_1), (l_2, r_2), ..., (l_n, r_n) \rangle$. The question is: Determine whether or not an alphabetic code with a given path vector $\bar{v} = \langle (l_1, r_1), (l_2, r_2), ..., (l_n, r_n) \rangle$ exists.

As for the previous condition, also in this case, we need to introduce a specific notation. Let $N = \max\{l_1 + r_1, l_2 + r_2, ..., l_n + r_n\}$ and let F be a full tree of order N. Number the leaves of F from 0 through $2^N - 1$. Define the *projection* of a node u of F as the set of leaves descendant of u. A projection is identified by the pair of indices corresponding to the leftmost and the rightmost leaf of the projection.

Consider the set of binary strings belonging to $\{0, 1\}^{l+r}$, that is, the set of strings consisting of l bits equal to zero and r bits equal to one; denote such a set by $S(l, r)$.

Each element of $\gamma \in S(l, r)$ represents an (l, r)-node of F: the node whose path, encoded with a 0 for a left edge and with a 1 for a right edge, gives γ.

Let u be the node of F identified by some element α of $S(l_u, r_u)$. The projection of u is given by (a, b) where a and b are the integers whose binary representations (with possible leading zeros) are respectively $\gamma \underbrace{00...00}_{N-(l_u+r_u)}$ and $\gamma \underbrace{11...11}_{N-(l_u+r_u)}$.

A natural way to construct a binary tree with a given path vector is the following: for $k = 1, 2, ..., n$, choose the leftmost available (l_k, r_k)-node of F to be the k^{th} leaf of the binary tree.

This strategy can be formalized as follows. Let $B_0 = 0$ and for each $i = 1, 2, ..., n$, let γ_i be the smallest element of $S(l_i, r_i) \cup \{\infty\}$ such that $\gamma_i 2^{n-(l_i+r_i)} > B_{i-1}$. Then define $A_i = \gamma_i 2^{N-(l_i+r_i)}$ and $B_i = 2^{N-(l_i+r_i)}(\gamma_i + 1) - 1$. Notice that the binary representation of A_i is $\gamma_i \underbrace{00...00}_{N-(l_i+r_i)}$ and the one of B_i is $\gamma_i \underbrace{11...11}_{N-(l_i+r_i)}$ and thus (A_i, B_i) is the projection of a (l_i, r_i)-node (provided that a tree with path vector $\bar{v}$ exists). From the definition, it follows that $B_k < A_{k+1}$, that

is, the projections are disjoint and in increasing order. In paper [44], De Prisco and Persiano proved the following result.

Theorem 5 ([44]). *Let $\bar{v} = \langle (l_1, r_1), (l_2, r_2), ..., (l_n, r_n) \rangle$ be a vector of pairs of positive integers. A binary tree with path vector $\bar{v}$ exists if and only if*

$$B_n < 2^N.$$

We notice that if B_n cannot be defined, then the condition of the theorem is not satisfied. Conversely, if B_n can be defined, then it is guaranteed to be strictly less than 2^N, thus a binary tree with path vector $\bar{v}$ exists if and only if B_n can be defined.

2.5 UPPER BOUNDS ON THE AVERAGE LENGTH OF OPTIMAL ALPHABETIC CODES

For practical and theoretical reasons, it is often important to know an estimate of the minimum average length of alphabetic codes *before* building them, that is, in terms of a closed formula of the symbol probabilities alone. In this section, we review the most relevant literature on the topic.

As discussed in Section 2.3, optimal alphabetic codes can be constructed in $O(n \log n)$ time. However, for real-time applications where faster processing is crucial, linear-time approaches are often preferred. Therefore, most of the studies that provide upper bounds on the average length of optimal alphabetic codes operate as follows:

- first, they design linear-time construction algorithms for suboptimal codes,
- subsequently, they compute explicit upper bounds on the constructed codes, in terms of some partial information on the probability distribution of the set of symbols.

Clearly, the derived bounds constitute upper bounds on the length of *optimal* alphabetic codes, as well.

The first result in this area was obtained by Gilbert and Moore [56]. They proposed a linear-time algorithm to construct an alphabetic code for a set of symbols S with an associated probability

distribution p, such that the code's average length is less than $H(p) + 2$. Let us briefly recall the algorithm's core idea:

Algorithm 4: Gilbert and Moore's algorithm

1 Let $S = \{s_1, \dots, s_n\}$ be a set of symbols and $p = \langle p_1, \dots, p_n \rangle$ the associated probability distribution.
2 Compute the values

$$r_i = \sum_{j=1}^{i-1} p_j + \frac{p_i}{2}, \quad \forall i = 1, \dots, n.$$

3 For each $i = 1, \dots, n$, take the first $\lceil -\log p_i \rceil + 1$ bit of the binary expansion of r_i to construct the codeword of the symbol s_i.

The algorithm is straightforward. Intuitively, its correctness follows from the strictly increasing values of r_i, for $i = 1, \dots, n$, which ensure both the prefix and alphabetic properties of the constructed codewords.

To illustrate this better, let us consider an example. Consider a set of source symbols $S = \{s_1, s_2, s_3, s_4\}$, which are ordered as $s_1 \prec s_2 \prec s_3 \prec s_4$. Let $p = \langle 0.25, 0.5, 0.125, 0.125 \rangle$ be its associated probability distribution. Let us apply the Gilbert and Moore algorithm.

We first compute the values of r_i for each symbol:

$$\begin{aligned}
r_1 &= \frac{p_1}{2} = \frac{0.25}{2} = 0.125 \\
r_2 &= p_1 + \frac{p_2}{2} = 0.25 + \frac{0.5}{2} = 0.5 \\
r_3 &= p_1 + p_2 + \frac{p_3}{2} = 0.25 + 0.5 + \frac{0.125}{2} = 0.8125 \\
r_4 &= p_1 + p_2 + p_3 + \frac{p_4}{2} = 0.25 + 0.5 + 0.125 + \frac{0.125}{2} = 0.9375.
\end{aligned}$$

Next, for each $i = 1, \dots, 4$ we have to take the first $\lceil -\log p_i \rceil + 1$ bits of the binary expansion of r_i. For s_1, the first $\lceil -\log_2(0.25) \rceil + 1 = 3$ bits of the binary expansion of $r_1 = 0.125$, which is 0.001, are 001. Thus, the codeword for s_1 is 001. For s_2 the first $\lceil -\log_2(0.5) \rceil + 1 = 2$ bits of the binary expansion of $r_2 = 0.5$, that is 0.10, are 10.

Thus, the codeword for s_2 is 10. For s_3 the first $\lceil -\log(0.125) \rceil + 1 = 4$ bits of the binary expansion of $r_3 = 0.8125$, that is 0.1101, are 1101. Thus, the codeword for s_3 is 1101. Finally, for s_4 the first $\lceil -\log_2(0.125) \rceil + 1 = 4$ bits of the binary expansion of $r_4 = 0.9375$, that is 0.1111, are 1111. Thus, the codeword for s_4 is 1111. The resulting code is $C = \{001, 10, 1101, 1111\}$. As one can see, this code satisfies both the prefix and alphabetic properties.

Moreover, since the codeword lengths are explicitly defined ($\lceil -\log p_i \rceil + 1$), one can see that they satisfy the existence conditions for an alphabetic code presented in Section 2.4. Formally, we can summarize the result as follows.

Theorem 6 ([56]). *For any set of symbols $S = \{s_1, \dots, s_n\}$, $s_1 \prec \cdots \prec s_n$, with associated probabilities $p = \langle p_1, \dots, p_n \rangle$, Algorithm 4 constructs an alphabetic code C for S whose average length $\mathbb{E}[C]$ satisfies*

$$\mathbb{E}[C] = \sum_{i=1}^{n} p_i \ell_i < H(p) + 2, \tag{2.9}$$

where ℓ_i is the length of the i^{th} codeword. Algorithm 4 runs in linear time.

It is worth noticing that since the average length of any prefix code is lower bounded by the Shannon entropy, $H(p)$, (and, therefore, *a fortiori*, the average length of any alphabetic code is also lower bounded by $H(p)$) Gilbert and Moore's codes are at most two bits away from the optimum.

It is interesting to remark that the upper bound of Theorem 6 is tight and cannot be improved unless one has some additional information about the probability distribution p of the symbols. To illustrate this, consider a set of three symbols $s_1 \prec s_2 \prec s_3$, with the probability distribution $p = \langle \epsilon, 1 - 2\epsilon, \epsilon \rangle$. For this instance, one can see that the average length of the optimal alphabetical code is $2 - \epsilon$. On the other hand, the entropy $H(p)$ of the distribution p can be arbitrarily small, as $\epsilon \to 0$.

Successively, Horibe [68] provided a better upper bound than that of Gilbert and Moore. He achieved this by providing an algorithm to construct alphabetic codes of average length less than

$$H(p) + 2 - (n + 2)p_{\min}, \tag{2.10}$$

where $p_{\min}$ is the smallest probability of p. The initial naive implementation of the algorithm given in [68] has a quadratic time complexity, $O(n^2)$. Walker and Gottlieb [132] later designed a more efficient implementation, reducing the time complexity to $O(n \log n)$. Subsequently, Fredman [47] provided a clever method that further reduced the time complexity to $O(n)$. We will delve deeper into Fredman's approach in Chapter 3, as it constitutes a crucial component of our linear-time procedure.

Let us now describe the idea of Horibe's algorithm. Unlike Algorithm 4, Horibe's algorithm does not explicitly specify the codeword lengths. Instead, it is a *weight-balancing* algorithm, similar to the classical Fano algorithm for sub-optimal prefix codes [125, p.17]. Horibe's algorithm constructs a binary tree whose root-to-leaf paths represent the codewords of the alphabetic tree (as illustrated in Section 2.2.1). The construction process begins by associating the root of the tree with the whole probability distribution $p = \langle p_1, \ldots, p_n \rangle$. Successively, one computes the index k that partitions the probabilities into the two sequences $\langle p_1, \ldots, p_k \rangle$ and $\langle p_{k+1}, \ldots, p_n \rangle$, where k is chosen in such a way that it minimizes the absolute difference between the sums of the probabilities in the two partitions, i.e.,

$$\left| \sum_{z=1}^{k} p_z - \sum_{z=k+1}^{n} p_z \right| = \min_{1 \le \ell < n} \left| \sum_{z=1}^{\ell} p_z - \sum_{z=\ell+1}^{n} p_z \right|. \tag{2.11}$$

The sequence $\langle p_1, \ldots, p_k \rangle$ is associated to the left child of the root, and the sequence $\langle p_{k+1}, \ldots, p_n \rangle$ is associated to the right child of the root.

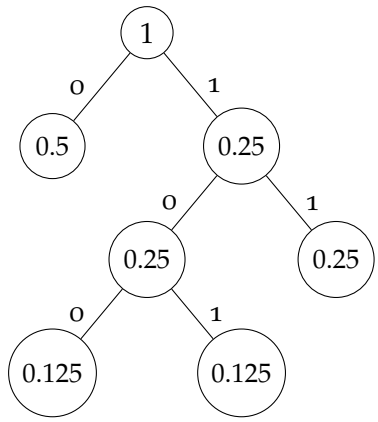
This partitioning process is recursively iterated in $\langle p_1, \ldots, p_k \rangle$ and $\langle p_{k+1}, \ldots, p_n \rangle$. In general, for any consecutive sequence of probabilities $\langle p_i, \ldots, p_j \rangle$, associated to a node x in the tree, one computes the index k_{ij} such that

$$\left| \sum_{z=i}^{k_{ij}} p_z - \sum_{z=k_{ij}+1}^{j} p_z \right| = \min_{i \le \ell < j} \left| \sum_{z=i}^{\ell} p_z - \sum_{z=\ell+1}^{j} p_z \right|.$$

Successively, the sequence $\langle p_i, \ldots, p_{k_{ij}} \rangle$ is associated to the left child of the node x, and $\langle p_{k_{ij}+1}, \ldots, p_j \rangle$ is associated to the right child of the node x. The process is iterated until one obtains sequences made up of just *one* single element (or symbol).

One can see that the binary tree built by such an algorithm is, by construction, a valid alphabetic code.

To make it clearer, let us illustrate through a specific example. Consider a set of source symbols $S = \{s_1, s_2, s_3, s_4\}$, which are ordered as $s_1 \prec s_2 \prec s_3 \prec s_4$. Let $p = \langle 0.5, 0.125, 0.125, 0.25 \rangle$ be its associated probability distribution. According to Horibe's weight-balancing algorithm, we begin with the whole probability distribution $\langle 0.5, 0.125, 0.125, 0.25 \rangle$. We need to partition it into two subsequences that minimize the absolute difference of the sums of their probabilities. Thus, for $\langle 0.5, 0.125, 0.125, 0.25 \rangle$, an optimal partition is $\langle 0.5 \rangle$ and $\langle 0.125, 0.125, 0.25 \rangle$, whose absolute difference is 0. Then, since $\langle 0.5 \rangle$ contains only one element, we need to repeat the operation only on $\langle 0.125, 0.125, 0.25 \rangle$. For $\langle 0.125, 0.125, 0.25 \rangle$, an optimal partition is $\langle 0.125, 0.125 \rangle$ and $\langle 0.25 \rangle$. We reiterate again the procedure on $\langle 0.125, 0.125 \rangle$ obtaining $\langle 0.125 \rangle$ and $\langle 0.125 \rangle$. The procedure now stops since all subsequences contain only one element. This process constructs the balanced binary shown below.



By assigning the bit 0 to the left branches and the bit 1 to the right branches, from the paths from the root to the leaves, one obtains the following final code $C = \{0, 100, 101, 11\}$.

Furthermore, Horibe proved the following upper bound on the average length of the obtained binary tree.

Theorem 7 ([68]). *For any set of symbols $S = \{s_1, \dots, s_n\}$, $s_1 \prec \cdots \prec s_n$, with associated probabilities $p = \langle p_1, \dots, p_n \rangle$, the alphabetic code C*

for S constructed by Horibe's weight balancing algorithm has an average length $\mathbb{E}[C]$ upper bounded by

$$\begin{aligned}\mathbb{E}[C] &\leq H(p) + \sum_{i=1}^{n-1} \max(p_i, p_{i+1}) - p_{\min} \\ &\leq H(p) + 2 - (n+2)p_{\min},\end{aligned}$$

where $p_{\min} = \min_i p_i$.

Later, Yeung [139], following the same approach of Gilbert and Moore, improved their upper bound (2.9). He achieved this by designing an algorithm that produces an alphabetic code whose average length is upper-bounded by

$$H(p) + 2 - p_1 - p_n,$$

where p_1 and p_n are the probabilities of the first and the last symbol of the ordered set of symbols S, respectively. To demonstrate the existence of a code with such an average length, Yeung proved that for a given probability distribution $p = \langle p_1, \ldots, p_n \rangle$, the codeword lengths $L = \langle \ell_1, \ldots, \ell_n \rangle$ defined as follows

$$\ell_i = \begin{cases} \lceil -\log p_i \rceil & \text{if } i = 1 \text{ or } i = n, \\ \lceil -\log p_i \rceil + 1 & \text{otherwise,} \end{cases} \tag{2.12}$$

satisfy the condition stipulated in Theorem 2. Therefore, it is established that an alphabetic code with lengths $L = \langle \ell_1, \ldots, \ell_n \rangle$ exists. Moreover, following an implementation strategy similar to Gilbert and Moore's or Horibe's algorithms, such an alphabetic code can be constructed in linear time. Let us briefly summarize the core idea of Yeung's approach.

Algorithm 5: Yeung's algorithm

1 Let $S = \{s_1, \dots, s_n\}$ be a set of symbols and $p = \langle p_1, \dots, p_n \rangle$ the associated probability distribution.
2 Compute the lengths $L = \langle \ell_1, \dots, \ell_n \rangle$ as follows

$$\ell_i = \begin{cases} \lceil -\log p_i \rceil & \text{if } i = 1 \text{ or } i = n, \\ \lceil -\log p_i \rceil + 1 & \text{otherwise,} \end{cases}$$

3 For each $i = 1, \dots, n$, choose the leftmost available leaf at the level ℓ_i to be the codeword of the symbol s_i.

As for the others, let us build some intuition through a specific example. Consider a set of source symbols $S = \{s_1, s_2, s_3, s_4\}$, which are ordered as $s_1 \prec s_2 \prec s_3 \prec s_4$. Let $p = \langle 0.25, 0.5, 0.125, 0.125 \rangle$ be its associated probability distribution. We start by computing the lengths ℓ_i for $i = 1, \dots, 4$:

$$\begin{aligned} \ell_1 &= \lceil -\log p_1 \rceil = \lceil -\log(0.25) \rceil = 2 \\ \ell_2 &= \lceil -\log p_2 \rceil + 1 = \lceil -\log(0.5) \rceil + 1 = 2 \\ \ell_3 &= \lceil -\log p_3 \rceil + 1 = \lceil -\log(0.125) \rceil + 1 = 4 \\ \ell_4 &= \lceil -\log p_4 \rceil = \lceil -\log(0.125) \rceil = 3. \end{aligned}$$

Then, sequentially for each $i = 1, \dots, n$, we take the leftmost available leaf at the level ℓ_i of a full binary tree to be the leaf associated with the symbol s_i, i.e., we take, according to the alphabetical order, the first available binary string of length ℓ_i to be the codeword of s_i. Thus, the codeword for s_1 is 00 because it is the first available binary string of length 2. Similarly, for s_2 we take the codeword 01 that which is the first available binary string of length 2 that comes after 00. Then, for s_3 we take 1000, which is the first available binary string of length 4 that comes after 01. And finally, for s_4 we take 101, which is the first available binary string of length 3 that comes after 1000. Thus, the resulting code is $C = \{\, 00, 01, 1000, 101\}$.

Like Gilbert and Moore's approach, since the lengths in the above algorithm are explicitly defined, one can quite easily get the following result.

Theorem 8 ([139]). *For any set of symbols $S = \{s_1, \dots, s_n\}$, $s_1 \prec \cdots \prec s_n$, with associated probabilities $p = \langle p_1, \dots, p_n \rangle$, there exists an*

alphabetic code C for S, that can be constructed in linear time and whose average length satisfies

$$\begin{aligned}\mathbb{E}[C] \leq & H(p) + 2 - p_1 \left(2 - \log p_1 - \lceil -\log p_1 \rceil\right) \\ & - p_n \left(2 - \log p_n - \lceil -\log p_n \rceil\right) \\ \leq & H(p) + 2 - p_1 - p_n.\end{aligned}$$

In [139], Yeung also established a significant relationship between alphabetic codes and Huffman codes (or, more generally, optimal prefix codes) [77].

Theorem 9 ([139]). *For any set of symbols $S = \{s_1, \ldots, s_n\}$, $s_1 \prec \cdots \prec s_n$, with associated probabilities $p = \langle p_1, \ldots, p_n \rangle$, if the probabilities p are in ascending or descending order, then the average length of an optimal alphabetic code for S is equal to the average length of the Huffman code for S.*

A significant consequence of this equivalence is that existing upper bounds on the average length of Huffman codes (e.g., [3]) can be directly applied to obtain upper bounds on the average lengths of optimal alphabetic codes, *when* the probability distribution p is ordered. In general, these upper bounds are much tighter than the known upper bounds for alphabetic codes that hold for arbitrary probability distributions (i.e., not necessarily ordered). Bounds on the length of Huffman codes as functions of partial knowledge of the probability distribution have been widely studied, e.g., [3, 20, 29–31, 42, 43, 51, 81, 112, 138, 140].

Several other significant contributions to the literature on upper bounds for the average length of alphabetic codes have followed. Nakatsu [115] claimed a new upper bound on the minimum average length of alphabetical codes. Nakatsu's method requires the construction (as a preliminary step) of a Huffman code for the probability distribution p, which is then suitably modified to obtain an alphabetic code for the same probability distribution p. Unfortunately, Sheinwald [126] later pointed out a gap in the analysis carried out in [115]. In a related but distinct approach, Fariña *et al.* [46] provided a multiplicative factor approximation. Indeed, they designed an algorithm to build an alphabetic code whose average length is at most a factor of $1 + O(1/\sqrt{\log |S|})$ more than the optimal one, where $|S|$ is the cardinality of the set of symbols S.

Another new upper bound on the minimum average length of optimal alphabetic codes was proposed by De Prisco and De Santis [40]. However, recently Dagan *et al.* [37] pointed out an issue in the analysis in [40] and, building on the original idea, proposed a modified upper bound. The core idea in [37] can be summarized as follows:

- from the initial probability distribution $p = \langle p_1, \dots, p_n \rangle$, compute the *extended* distribution $q = \langle 0, p_1, 0, \dots, 0, p_n, 0 \rangle$;
- apply the classic algorithm of Gilbert and Moore [56] (see Algorithm 4) to construct an alphabetic code C for the distribution q, whose average length is less than $H(q) + 2 = H(p) + 2$ since $H(q) = H(p)$;
- prune the binary tree representing the alphabetical code C by eliminating the leaves associated with the null probabilities in q, and re-adjust the resulting tree.

Dagan *et al.* proved that the average length $\mathbb{E}[C]$ of the resulting alphabetic code satisfies the following inequality:

$$\begin{aligned} \mathbb{E}[C] &\leq H(p) + 2 - p_1 - p_n - \sum_{i=1}^{n-1} \min(p_i, p_{i+1}) \\ &= H(p) + 1 - \frac{p_1 + p_n}{2} + \frac{1}{2} \sum_{i=1}^{n-1} |p_i - p_{i+1}|. \end{aligned} \tag{2.13}$$

In Chapter 3, we re-examine the approach presented in [37]. We first point out an issue in the analysis in [37], and subsequently we refine the idea to improve the upper bound (2.13). Furthermore, we design a new linear-time procedure for constructing these near-optimal alphabetic codes, whose average is upper-bounded by a quantity smaller than (2.13). Finally, in Section 3.3, by leveraging the intrinsic connection between binary search trees and alphabetic codes, we utilize our results on alphabetic codes to also improve the current best-known upper bound on the average length of optimal binary search trees.

2.6 VARIATIONS AND GENERALIZATIONS

In this section, we will survey the known results about variations and generalizations of the classical problem of constructing alphabetic codes of minimum average length.

2.6.1 *Alphabetic codes optimum under different criteria*

In the classical formulation of the problem, one is given a sequence of positive weights $w = \langle w_1, \dots, w_n \rangle$, which is usually assumed to be a probability distribution, and the objective is to construct an alphabetic code for w of minimum average cost, that is,

$$\min \sum_{i=1}^{n} w_i \ell_i, \tag{2.14}$$

where ℓ_i is the length of the codeword associated with the weight w_i. However, in some cases, it is more useful to consider other criteria for the construction of the optimal code.

Hu, Kleitman and Tamaki in [70] considered a variant of the problem (2.14), in which instead of minimizing the average length of the tree representing the alphabetic code, they seek to minimize suitable cost functions that they call *regular cost functions*. As an example of a regular cost function in [70], they consider the following minimax criterion:

$$\min \max_i w_i 2^{\ell_i}. \tag{2.15}$$

Alphabetic trees optimized according to (2.15) are also called *alphabetic minimax* trees. Other examples of regular cost functions are described in [70], together with their justifications. Moreover, Hu *et al.* [70] observed that through a suitable modification, the classic Hu-Tucker algorithm (see Section 3) can be adapted to compute an optimal alphabetic tree for any arbitrary regular cost function in $O(n \log n)$ time. Successively, for the specific case of alphabetic minimax trees, Kirkpatrick and Klawe [88] improved the result given in [70]. Indeed, they proposed a linear time algorithm for the problem when the weights are integers (actually, their algorithm minimizes the quantity $\max_i\{w_i + \ell_i\}$, but one can see this minimization is equivalent to (2.15), as discussed in [55]).

Later, Gagie [50] provided a $O(nd \log\log n)$ time algorithm for the same problem considered in [88], where d is the number of distinct integers in the set $\{\lceil w_1 \rceil, \ldots, \lceil w_n \rceil\}$. A more efficient algorithm has been designed by Gawrychowski [55], who gave an $O(nd)$ algorithm for the construction of the optimal alphabetic minimax tree, where d is the number of distinct integers in the set $\{\lfloor w_i \rfloor, \ldots, \lfloor w_n \rfloor\}$.

In the paper [36], Cun-Quan introduced a new variant of the problem. Given a sequence of positive weights $w = \langle w_1, \ldots, w_n \rangle$, and an alphabetic tree T_n (i.e., a tree representing an alphabetic code for w), one defines the cost $w(T_n)$ of T_n in the following way:

- the cost of the i^{th} leaf (read from left to right) is equal to w_i;
- the cost $w(u)$ of any internal node u of T_n is given by

  $$w(u) = \max\{w(u_\ell), w(u_r)\},$$

  where u_ℓ is the left child and u_r is the right child of u, respectively;
- the cost $w(T_n)$ is

  $$w(T_n) = \sum w(u), \tag{2.16}$$

  where the summation is over all internal nodes of T_n.

One can see that the kind of minimization problem described above does *not* fit in the framework of regular cost functions defined by Hu, Kleitman, and Tamaki in [70]. Furthermore, Cun-Quan [36] provides an $O(n \log n)$ algorithm to compute an alphabetic tree whose cost (as defined in (2.16)) is minimum.

Fujiwara and Jacobs [48] analyzed a generalization of the classical alphabetic-tree problem, called *general cost alphabetic* tree, where instead of associating a weight to each leaf, they associate an arbitrary function. More formally, the problem can be described as follows: Given n arbitrary functions $f_1, \ldots, f_n : \mathbb{N} \to \mathbb{R}_+$, the objective is to construct an alphabetic tree such that

$$\sum_{i=1}^{n} f_i(\ell_i) \tag{2.17}$$

is minimized, where ℓ_i is the depth of the i^{th} leaf from left to right. The authors of [48] showed that the dynamic programming approach for the classical alphabetic tree problem can be extended to arbitrary cost functions, obtaining an $O(n^4)$ time and $O(n^3)$ space algorithm for the construction of an optimal general cost alphabetic tree. They also extended their findings to Huffman codes with general costs. However, unlike the case of alphabetical trees, in the Huffman scenario, they showed that the problem becomes NP-hard in the general prefix scenario.

In a separate study, Baer [12] considered minimizing a different cost function: Given a sequence of positive weights $w = \langle w_1, \ldots, w_n \rangle$, the goal is to minimize

$$\log_a \left(\sum_{i=1}^{n} w_i a^{\ell_i} \right),$$

for a given $a \in (0, 1)$, over the class of all alphabetic codes with n codewords. Baer provided an $O(n^3)$ time and $O(n^2)$ space dynamic programming algorithm to find the optimal solution. Moreover, he noted that methods traditionally used to improve the speed of optimizations in related problems, such as the Hu–Tucker procedure, fail for this specific variant of the problem. In the same paper, Baer proposed two algorithms that find suboptimal solutions in linear time or $O(n \log n)$ time, respectively, and provided redundancy bounds to guarantee their coding efficiency.

2.6.2 *Height-limited alphabetic trees*

In contexts of routing lookups [63, 114], it emerges the necessity to minimize the average packet lookup time while keeping the worst-case lookup time *within* a fixed bound. This requirement directly translates into considering a specific class of alphabetic trees known as *height-limited* alphabetic trees. In this scenario, the main problem is to find alphabetic codes of minimum average length, under the constraint that no word in the code has a length above a certain input parameter. More formally, given an ordered set of symbols $S = \{s_1, \ldots, s_n\}$, and a probability distribution $p =$

$\langle p_1, \dots, p_n \rangle$ on S, one seeks to solve the following optimization problem:

$$\min_{C \text{ alphabetic}} \mathbb{E}[C] = \min_{C \text{ alphabetic}} \sum_{i=1}^{n} p_i \ell_i,$$
$$\text{subj. to } \ell_i \leq L, \quad \forall i = 1, \dots, n,$$

where ℓ_i is the length of the codeword associated with the symbol s_i. These codes are also called L-restricted alphabetic codes.

In [73], Hu and Tan presented an algorithm for constructing an optimal binary tree with the restriction that its height does not exceed a given integer L. However, the time complexity of the algorithm is exponential in L. Garey [53] introduced a polynomial-time solution with complexity $O(n^3 \log n)$ for the construction of an optimal height-limited alphabetic tree. Successively, Itai [78] and Wessner [133] independently reduced this time to $O(n^2 L)$. Similarly, Larmore [99] also designed an $O(n^2 L)$ time algorithm for the construction of an optimal height-limited alphabetic tree by refining the previous Hu and Tan's algorithm [73].

Hassin and Henig [64] extended a monotonicity theorem of Knuth [91] to hold under weaker assumptions and applied this new result to reduce the complexity of several optimization scenarios, including height-limited alphabetic trees.

Successively, Larmore and Przytycka [101] provided a more efficient algorithm with a complexity of $O(nL \log n)$ for constructing an optimal alphabetic tree with height restricted to L.

Gupta *et al.* [63] focused their attention on the construction of nearly optimal L-restricted alphabetic codes. They provided an $O(n \log n)$ time algorithm for constructing an alphabetic tree whose average length differs from the optimal value by at most 2. Similarly, Laber *et al.* [98] suggested a simple approach to construct sub-optimal L-restricted alphabetic codes, comparing their average length with the average length of the Huffman code for the same distribution.

In Chapter 4, we will turn our attention to another variant of alphabetic codes that generalizes the L-restricted alphabetic codes. Furthermore, beyond exploring this new framework, we will also see how some of our findings can be extended to the broader and more challenging domain of prefix codes.

2.6.3 *Binary trees that are alphabetic with respect to given partial orders*

In the classic alphabetic tree problem, a total order $\prec$ is defined on the set of symbols $S = \{s_1, \ldots, s_n\}$, and the left-to-right reading of the leaves in the tree must preserve the same ordering of the elements in S, according to the relation $\prec$.

However, in some scenarios (see, e.g., [106]) there might be known only a *partial* order on the set of symbols $S = \{s_1, \ldots, s_n\}$, and the challenge is to construct a tree (or a code) in which the left-to-right reading of the leaves of the tree is *consistent* with the partial order on S.

Lipman and Abrahams [106] studied this variant, motivated by the problem of detecting defects in a pipeline. They proposed an indirect solution to solve the problem. Specifically, they first decompose the given partial order into a set of linear total orders. Then, they apply the Hu-Tucker algorithm (see Algorithm 3) to each of these subproblems, and finally construct the final tree by applying, again, the Hu-Tucker algorithm to the partial solutions previously obtained. In [106], the authors left open the problem of providing explicit upper bounds on the average length of the trees produced by this construction.

Later, Barkan and Kaplan [13] addressed a problem similar to the one considered by Lipman and Abrahams [106]. Specifically, Barkan and Kaplan [13] considered a generalized version of the classic problem, which they called *partial alphabetic* tree problem. In the partial alphabetic tree problem, given a multiset of non-negative weights $W = \{w_1, \ldots, w_n\}$, partitioned into $m \leq n$ blocks $B_1, \ldots, B_m$, the aim is to build a tree T, where the elements of W reside in its leaves, satisfying the following property: If we traverse the leaves of T from left to right, then all leaves of B_i precede all leaves of B_j for every $i < j$. Furthermore, among all such trees, it is required that T has minimum average length

$$\sum_{i=1}^{n} w_i \ell_i,$$

where ℓ_i is the depth of w_i in T. For $n = m$, the problem reduces to the classical one. In [13], Barkan and Kaplan developed a pseudo-polynomial time algorithm for the construction of an optimal tree,

whose complexity depends on the weight values. Moreover, the technique developed in [13] is general enough to apply to several other objective functions, possibly different from the average length of the tree. However, the problem of whether there exists or not an algorithm for the partial alphabetic tree problem that runs in *polynomial time*, for any set of weights, remains open.

2.6.4 *Alphabetic AIFV codes*

Prefix codes, which include alphabetic codes, are an example of *uniquely decodable* codes that allow *instantaneous decoding*. This means a codeword can be uniquely identified and decoded as soon as its last bit is received, without needing to see subsequent bits.

Yamamoto *et al.* [136] introduced binary *Almost Instantaneous Fixed-to-Variable length* (AIFV) codes. These are uniquely decodable codes that might have a small decoding delay, specifically, at most a two-bit decoding delay. This means that a codeword can be uniquely decoded after its bits and, at most, two additional bits are received. Yamamot *et al.* [136] demonstrated that AIFV codes can sometimes provide better compression for stationary memoryless sources. Later, Hiraoka and Yamamoto [67] defined the alphabetic version of the AIFV codes, using three code trees for the decoding process to achieve at most a two-bit decoding delay. They also proposed an algorithm for the construction of almost optimal binary alphabetic AIFV codes by modifying Hu-Tucker codes [74]. Furthermore, despite their non-optimality, the constructed codes still attain a better compression rate than classical Hu-Tucker codes. A further natural extension of the alphabetic AIFV codes, called alphabetic AIFV-m, was proposed by Iwata and Yamamoto [79]. These codes use $2m - 1$ code trees for decoding and have a decoding delay of at most m-bits for any integer $m \geq 2$. Moreover, they also designed a polynomial-time algorithm for the construction of an optimal binary AIFV-m code.

2.6.5 *Linear time algorithms for special cases*

In Section 2.3, we mentioned that the best-known algorithms for constructing optimal alphabetic codes have an $O(n \log n)$ time complexity in the general case. However, under specific circumstances, it is possible to achieve linear-time algorithms for the problem.

Klawe and Mumey [89] extended the ideas and techniques of Hu and Tucker, creating an $O(n)$ time algorithm to construct optimal alphabetic codes either when all the input weights are within a constant factor of each other or when they are exponentially separated. A sequence $w_1, \ldots, w_n$ of weights is said to be *exponentially separated* if there exists a constant C such that it holds that

$$|\{i : \lfloor \log w_i \rfloor = k\}| < C, \forall k \in \mathbb{Z}.$$

Later, Larmore and Przytycka [102] considered the *integer alphabetic* tree problem, where the weights are integers in the range $[0, n^{O(1)}]$. They provided an $o(n \log n)$ time algorithm for the construction of an optimal integer alphabetic tree. Moreover, by relating the complexity of the optimal alphabetic tree problem to the complexity of sorting, they gave an $O(n\sqrt{\log n})$-time algorithm for the cases in which the weights can be sorted in linear time, or, equivalently, the weights are all integers in a small range [71]. Successively, Hu *et al.* [71] further improved the results of [102], designing an $O(n)$ time algorithm for the construction of an optimal integer alphabetic tree.

Following a different line of research, Hu and Morgenthaler [75] analyzed several classes of inputs on which the Hu-Tucker algorithm [74] runs in linear time. For instance, they showed that for *almost uniform sequences* of weights, which are sequences W of weights for which

$$\forall w_i, w_j, w_k \in W, \quad w_i + w_j \geq w_k,$$

and also for *bi-monotonal increasing sequences*, i.e., sequences of weights for which

$$w_1 + w_2 \leq w_2 + w_3 \leq \cdots \leq w_{n-1} + w_n,$$

holds, the Hu-Tucker algorithm requires only linear time.

In [76], Hu *et al.* introduced the concept of *valley sequence* for the purpose of understanding the computational complexity of constructing optimal alphabetic codes. We recall that a sequence $w_1, \ldots, w_n$ of weights is a valley sequence if

$$w_1 > w_2 > \cdots > w_{j-1} \leq w_j \leq w_{j+1} \leq \cdots \leq w_n.$$

In other words, the weights are first decreasing and then increasing. They [76] showed that if the weight sequence W is a valley sequence, then the cost of the optimal alphabetic tree for W is the same as the cost of the Huffman tree for W. In addition, one can construct an optimal alphabetic tree for a valley sequence in linear time. Moreover, since an ordered sequence is just a special case of a valley sequence, one can also construct an optimal alphabetic tree for an ordered sequence in linear time. A similar result for ordered sequences also derives from the well-known fact that the minimum average length of an alphabetic code for an ordered sequence W, is equal to the minimum average length of a prefix code for W (see Theorem 9). Therefore, since Huffman codes for ordered sequences can be computed in linear time [130], it follows that minimum average length alphabetic codes for ordered sequences can as well.

2.6.6 *k-ary alphabetic trees*

Most of the literature on alphabetic codes focuses on **binary** alphabetic codes. In this section, we will describe the known results for general k-ary alphabetic codes, where $k \geq 2$ is an arbitrary integer. The papers containing results on k-ary alphabetic codes use the terminology of search trees. Since we have already seen that there is an equivalence between alphabetic codes and search algorithms (search trees) that operate through comparison tests, we will stick to the search-tree terminology.

The first author to study the problem of constructing optimal (i.e., minimum average length) k-ary alphabetic trees was Itai [78]. He claimed an $O(n^2 L \log k)$ time algorithm for constructing optimal k-ary alphabetic trees of maximum depth L, a scenario similar to the one considered in Section 2.6.2. Furthermore, he also claimed an $O(n^2 \log k)$ time algorithm for constructing *unrestricted* optimal k-ary alphabetic trees. However, Gotlieb and Wood [61]

later pointed out a gap in the analysis of [78], invalidating its claims. Moreover, they designed an $O(n^3 L \log k)$ time algorithm for constructing optimal k-ary alphabetic trees of maximum depth L, and an $O(n^3 \log k)$ time algorithm for constructing *unrestricted* optimal k-ary alphabetic trees.

Ben-Gal [16] considered the problem of constructing *almost* optimal k-ary alphabetic trees. Specifically, he generalized Horibe's weight-balancing algorithm [68] (that we have described in Section 2.5) to the arbitrary case of $k \geq 2$, and provided some upper bounds on the average length of the k-ary alphabetic trees one obtains.

Kirkpatrick and Klawe [88] considered the k-ary alphabetic minimax tree problem, which is a generalization of the problem discussed in Section 2.6.1, providing a linear time algorithm when the weights are integers and an $O(n \log n)$ time algorithm for the general case. Subsequently, Coppersmith *et al.* [33] considered a variant of the problem in [88], in which each internal node of the tree has degree at most k (not exactly k as in [88]). They provided a linear-time algorithm when the input weights are integers, and an $O(n \log n)$ time algorithm for real weights. They also provided a tight upper bound for the cost of the constructed solution.

Finally, Gagie [50] developed an $O(nd \log \log n)$ time algorithm for the same problem considered in [88], where d is the number of distinct integers in the set $\{\lceil w_1 \rceil, \ldots, \lceil w_n \rceil\}$. The algorithm improves upon the previous result of Kirkpatrick and Klawe when d is small.

2.7 MISCELLANEA

In this section, we review a few interesting results on alphabetic codes that deal with distinct and unrelated problems, not classifiable under a unified theme.

Let $S = \{s_1, \ldots, s_n\}$ be an ordered set of symbols and $p = \langle p_1, \ldots, p_n \rangle$ be the probabilities of the symbols in S. The minimum average length of an alphabetic code for S, regarded as a function of p, is *not* invariant with respect to permutations of the probabilities $p_1, \ldots, p_n$. By contrast, if S is unordered, then one has that the minimum average length of a prefix encoding of S (i.e., the average length of a Huffman code for S), is *invariant* with respect to permutations of the probabilities $p_1, \ldots, p_n$.

This leads to the following natural question: for a given ordered set of symbols $S = \{s_1, \ldots, s_n\}$, and an arbitrary probability distribution $p_1, \ldots, p_n$, what is the permutation of the probabilities $p_1, \ldots, p_n$ that forces the average length of an optimal alphabetic code for S to assume its *maximum* value? Kleitman and Saks [90] studied this problem and found an elegant answer. Given a probability distribution $p = (p_1, \ldots, p_n)$, such that $p_1 \leq p_2 \leq \ldots, \leq p_n$, then the permutation of the elements of p that produces the costliest optimal alphabetic code is an alternating sequence of the smallest and largest probabilities: $p_1, p_n, p_2, p_{n-1}, \ldots,$. Moreover, they showed that the average length of such costliest optimal alphabetic code can be expressed in terms of the average length of a Huffman code for a related probability distribution q, easily computable from p.

Related problems were explored in [72, 122], whose results can be seen as corollaries of the main finding in [90]. Later, Yung-chin [143] extended the main result of the work of Kleitman and Saks [90] to the alphabetic codes with a hard limit on the maximum codeword length, a setting similar to the one discussed in Section 2.6.2.

Ramanan [121] considered the important problem of efficiently testing whether or not a given alphabetic code is optimal for an ordered set of symbols $S = \{s_1, \ldots, s_n\}$, and its associated probability distribution p. Based on the correctness proof of the Hu-Tucker algorithm [74], Ramanan provided necessary and sufficient conditions on the probability sequence $p = \langle p_1, \ldots, p_n \rangle$, for a given code tree to be optimal. From this result, Ramanan also demonstrated that the optimality of two specific types of alphabetic code trees can be tested in linear time: *very skewed* trees (trees in which the number of nodes in each level is bounded by some constant) and *well-balanced* trees (trees in which the maximum difference between the levels of any two leaves is bounded by some constant). The problem of testing the optimality of an *arbitrary* code tree in linear time, however, remains one of the main open problems in the field.

In their paper, Anily and Hassin [9] considered the problem of ranking the *best K* trees for a given sequence of weights $w_1, \ldots, w_n$. More precisely, given the weights $w_1, \ldots, w_n$, the problem is that of computing the binary tree with the smallest average length, the

binary tree with the second smallest average length, and so on, up to the binary tree with the K^{th} smallest average length. They studied both the alphabetic and non-alphabetic cases, providing an $O(Kn^3)$ time algorithm for ranking both the K-best binary alphabetic trees and the K-best binary non-alphabetic trees.

In [100], Larmore introduced the concept of *minimum delay* codes. A minimum delay code is a prefix code in which, instead of minimizing the average length, the aim is to minimize the *expected delay* of the code. Expected delay is defined as the expected time between a request to transmit the symbol and the completion of that transmission, assuming a channel with fixed capacity, where requests are queued. Larmore formally defined the expected delay as a particular nonlinear function of the average code length of the code. Furthermore, he presented an $O(n^5)$ time and $O(n^3)$ space algorithm to find a prefix code of minimum expected delay. The algorithm can also be adapted to build an alphabetic code of minimum expected delay. However, its time complexity is not guaranteed to be polynomial, as it depends heavily on the monotonicity of the weights. Therefore, the question of whether a polynomial-time algorithm exists for constructing an alphabetic code of minimum delay remains open.

In [1], Abrahams considered the problem of constructing optimal *monotonic codes*, which are codes with monotonic codeword lengths (i.e. the codeword lengths are either increasing or decreasing). She compared optimal monotonic codes with optimal alphabetic codes, establishing bounds between their lengths. Additionally, she also provided sufficient conditions under which the Hu-Tucker algorithm can be used to build optimal monotonic codes.

In [2], Abrahams considered a parallelized version of the classical search problem described in Section 2.2.1. Specifically, the idea of the parallelized version is to distribute the set of n items, with a given probability distribution $p_1, \dots, p_n$, into k subsets. These subsets are then searched simultaneously (i.e., in parallel) to find the single item of interest. The aim remains the one to minimize the average search time. The case with $k = 1$ corresponds exactly to the classical case of alphabetic codes. While, the case with $k = n$ is trivial: one item is placed in each subset. Thus, the main challenge is to solve the intermediate cases, where $1 < k < n$. The goal is

to decide which items should be grouped together into a common subset. Within each subset, the search is performed using the Hu-Tucker algorithm, which, as shown in Section 2.3, constructs the optimal alphabetic code/tree. In [2], Abrahams provided an algorithm for the problem and established upper bounds on the resulting average search time.

In [11], Baer focused on constructing alphabetic codes optimized for power law distributions, that is, when the probability of the i^{th} symbol p_i is of the form $p_i \sim ci^{-\alpha}$, where c and $\alpha > 1$ are constant, and $f(i) \sim g(i)$ means that the ratio of the two functions, $f(i)/g(i)$, approaches 1 as i increases.

Kosaraju *et al.* introduced the *Optimal Split Tree* problem in [96]. This problem generalizes several classic tree-building problems. Let $A = \{a_1, \ldots, a_n\}$ be a set of elements where each element a_i has an associated weight $w_i > 0$. A partition of A into two subsets $B, B \setminus A$ is called a *split* of A. A set S of splits of A is a *complete set of splits* if for each pair $a_i, a_j \in A$ there exists a split $B, B \setminus A$ in S such that $a_i \in B$ and $a_j \in B \setminus A$. A *split tree* for a set A and a complete set S of splits of A is a binary tree in which the leaves are labeled with the elements of A and the internal nodes correspond to splits in S. Specifically, for any node v of a binary tree, let $L(v)$ be the set of labels of the leaves of the subtree rooted at v. A split tree is a full binary tree such that for any internal node v with children v_1, v_2 there exists a split $\{B_1, B_2\} \in S$ such that $B_1 \cap L(v) = L(v_1)$ and $B_2 \cap L(v) = L(v_2)$. Hence, given a set A with its associated weights and a complete set of splits of A, the optimal split tree problem is to compute a split tree with minimum average length. One can see that the problem is a generalization of many others, including the Huffman tree problem and the optimal alphabetic tree problem. Indeed, when the set S of splits contains all possible splits of A we get the classic Huffman tree problem, while when the set S of splits contains $\{B_1, A \setminus B_1\}, \ldots, \{B_{n-1}, A \setminus B_{n-1}\}$ splits where $B_i = \{a \in A : a \prec a_i\}$, the problem reduces to the classic alphabetic tree problem. In [96], the authors showed that the optimal split tree problem, in the general case, is NP-complete. An equivalent proof of NP-completeness was previously provided by Laurent and Rivest [103]. Moreover, in [96] the authors provided an $O(\log n)$ approximation algorithm for the problem, providing also an example for which the algorithm achieves an $\Omega(\log n / \log \log n)$

approximation ratio. In addition, they adapted their algorithm, obtaining an $O(1)$ approximation algorithm for the partially ordered alphabetic tree problem (the same problem considered in Section 2.6.3).

In [105], Levcopoulos *et al.* presented an interesting and powerful result: they demonstrated that for any arbitrarily small $\epsilon > 0$, one can construct, in $O(n)$ time, an alphabetic tree whose cost is within a factor of $(1 + \epsilon)$ from the optimum. This means that their algorithm can efficiently produce almost optimal alphabetical codes that are very close to the optimal ones.

Part III

THE SHOWCASE

This part presents the original contributions of this Thesis, demonstrating the deep interplay between *Search Theory* and *Variable-Length Codes* (VLCs).

It begins with new results for classical alphabetic codes and binary search trees, establishing tighter bounds and novel linear-time constructions. The perspective is then extended to codes with asymmetric symbol costs—a framework that models search problems with unequal test outcome costs—for which polynomial-time construction procedures are provided. Finally, the analysis broadens to prefix codes, specifically those using one symbol as a delimiter, providing new linear-time constructions and bounds on their average cost.

Taken together, these contributions showcase how the framework of Variable-Length Codes can be used to reformulate and solve complex search problems, yielding new insights, more efficient algorithms, and tighter analytical bounds.

3

NEW RESULTS ON ALPHABETIC CODES AND BINARY SEARCH TREES

As anticipated previously, this chapter presents the primary results of this thesis, focusing on the classical binary alphabetic code problem. We establish novel upper bounds on the cost of optimal binary alphabetic codes and optimal BSTs. For both structures, these bounds significantly improve on the best bounds currently known in the literature. Additionally, we provide linear-time algorithms for their construction. Finally, we introduce a framework that, in linear time, can transform an alphabetic code into a binary search tree. This framework ensures that any future advancements in the field of alphabetic codes can be directly and efficiently applied to the field of BSTs, too. The majority of the results presented in this chapter originate from [24].

3.1 NOTATION AND PRELIMINARIES

Before presenting our findings, let us recall some notations for the sake of completeness. Let $S = \{s_1, \dots, s_m\}$ be a set of m symbols, ordered according to a given total order relation $\prec$, for which it holds that $s_1 \prec \cdots \prec s_m$, and let $\phi = \langle \phi_1, \dots, \phi_m \rangle$ be the probability distribution on the set of symbols S. We use ϕ instead of the more common p to avoid confusion with the notation typically used to describe the probability distribution of a *binary search tree* (BST). A binary alphabetic code, as already seen in Definition 3, is an order-preserving mapping $w : S \mapsto \{0,1\}^+$, where the order relation on the set of all binary strings $\{0,1\}^+$ is the standard alphabetical order. Moreover, we denote by $C = \{w(s) : s \in S\}$ the set of codewords of w.

In the following sections, as already argued in Section 2.2.1, we will interchangeably view, depending on the context, an alphabetic code as a binary tree and vice-versa. Moreover, as said before, alphabetic codes, in their binary tree representation, naturally

correspond to search procedures that make use of comparison queries with binary outcomes.

Thus, BSTs represent, in a sense, a generalization of binary trees associated with alphabetic codes, since the former can take into account both *successful* and *unsuccessful* searches. More precisely, any comparison-based search algorithm on an ordered list of elements $x_1 \prec \cdots \prec x_n$ can be represented as a BST, with an internal node associated with each element x_i and a leaf associated to each of the $n+1$ gaps $(-\infty, x_1), (x_1, x_2), \ldots, (x_{n-1}, x_n), (x_n, +\infty)$ among the x_i's. Let x be an element that we want to seek for in the list. Each internal node represents a comparison operation between x and the content x_i of the node. The outcome of the comparison can be either $x \prec x_i$, $x = x_i$ or $x \succ x_i$ (see Section 6.2 of [93]). Thus, the internal nodes correspond to successful searches, and the leaves to the unsuccessful ones.

The cost of a BST T is defined as the average number of questions needed to identify an arbitrary input element x. More precisely, given a probability distribution $\sigma = \langle \sigma_1, \ldots, \sigma_{2n+1} \rangle = \langle p_0, q_1, p_2, \ldots, q_n, p_n \rangle$, where the probabilities of successful searches are $q_1, \ldots, q_n$, and $p_0, \ldots, p_n$ are the probabilities of unsuccessful searches, the cost of the binary search tree T, for the probability distribution σ, is equal to

$$\mathbb{E}[T] = \sum_{i=1}^{n} q_i \, \ell(q_i) + \sum_{i=0}^{n} p_i \, \ell(p_i), \tag{3.1}$$

where $\ell(q_i)$ denotes the level of q_i in the tree, i.e., the number of nodes from the root of T to q_i, whereas $\ell(p_i)$ is the level of the parent of p_i in the tree [91]. We assume that the level of the root of T is equal to 1.

We observe that when the probabilities of successful searches, $q_1, \ldots, q_n$, are all 0's, the problem of finding an optimal binary search tree for the probability distribution σ is equivalent to the problem of finding an optimal alphabetic code for the probability distribution $\phi = (p_0, \ldots, p_n)$.

3.2 IMPROVED BOUNDS AND ALGORITHMS FOR ALMOST-OPTIMAL ALPHABETIC CODES

Some of our key findings consist of a series of improvements to the upper bound (2.13) on the average length of an optimal alphabetic code, denoted as $L_{\min}$, established by Dagan *et al.* [37]. For instance, we show that

$$\begin{aligned} L_{\min} \leq H(\phi) + 2 - \phi_1 \left(2 - \log_2 \phi_1 - \lceil -\log_2 \phi_1 \rceil\right) \\ - \phi_m \left(2 - \log_2 \phi_m - \lceil -\log_2 \phi_m \rceil\right) \\ - \sum_{i=1}^{m-1} \min(\phi_i, \phi_{i+1}). \end{aligned} \tag{3.2}$$

We also point out and then fix a gap in their original analysis. But more importantly, we design a linear-time algorithm to construct alphabetic codes of average length no greater than the value on the right-hand side of (3.2). This efficient algorithm will be crucial, as it serves as the foundation for the results we present later on BSTs in Section 3.3.

Before we dive into our findings, let us develop some useful machinery and recall some tools and results that will be necessary.

Given a probability distribution $\phi = \langle \phi_1, \ldots, \phi_m \rangle$, we denote by

$$\nu = \langle \nu_1, \ldots, \nu_{2\,m-1} \rangle = \langle \phi_1, 0, \phi_2, \ldots, 0, \phi_m \rangle$$

its *partially extended* distribution, and by

$$\nu' = \langle \nu_1, \ldots, \nu_{2\,m+1} \rangle = \langle 0, \phi_1, 0, \phi_2, \ldots, 0, \phi_m, 0 \rangle$$

its *fully extended* distribution.

For the sake of completeness, we also recall some auxiliary tools and results from Nakatsu [115] that have already been introduced and discussed in Section 2.4.

Definition 8 ([115])**.** *Let $L = \langle \ell_1, \ldots, \ell_m \rangle$ be a list of positive integers. For a binary fraction x and integer $i \geq 1$, let the function* `trunc` *be defined as*

$$\texttt{trunc}(i, x) = \frac{\lfloor 2^i x \rfloor}{2^i} \tag{3.3}$$

that is, $\texttt{trunc}(i, x)$ *is the fraction obtained by considering only the first* i *bits in the binary representation of* x.

Let $\alpha_i = \min(\ell_{i-1}, \ell_i)$, *for* $i = 2, \ldots, m$. *The recursive function* sum *is defined as*

$$\texttt{sum}(L, i) = \begin{cases} \texttt{trunc}(\alpha_i, \texttt{sum}(L, i-1)) + 2^{-\alpha_i} & \text{if } i \geq 2, \\ 0 & \text{if } i = 1. \end{cases} \tag{3.4}$$

To gain some intuition about the functions $\texttt{trunc}(\cdot, \cdot)$ and $\texttt{sum}(\cdot, \cdot)$ defined above, let us examine a small numerical example.

Let $L = \langle 4, 2, 3, 3 \rangle$ be a list of integers. We have $\alpha_2 = \min(4, 2) = 2$, $\alpha_3 = \min(2, 3) = 2$ and $\alpha_4 = \min(3, 3) = 3$. Now, we can compute the value of sum for L. By definition $\texttt{sum}(L, 1) = 0$. It follows that

$$\begin{aligned} \texttt{sum}(L, 2) &= \texttt{trunc}(\alpha_2, \texttt{sum}(L, 1)) + 2^{-\alpha_2} \\ &= \texttt{trunc}(2, 0) + 2^{-2} = 2^{-2}, \end{aligned}$$

since $\texttt{trunc}(2, 0)$ takes the value whose binary expansion is the same as the first 2 bits of the binary representation of 0 (that are all 0's). Similarly, we have

$$\begin{aligned} \texttt{sum}(L, 3) &= \texttt{trunc}(\alpha_3, \texttt{sum}(L, 2)) + 2^{-\alpha_3} \\ &= \texttt{trunc}(2, 2^{-2}) + 2^{-2} = 2^{-2} + 2^{-2} = 2^{-1}, \end{aligned}$$

since $\texttt{trunc}(2, 2^{-2})$ takes the value whose binary expansion is the same as the first 2 bits of the binary representation of $2^{-2} = \frac{1}{4}$, which corresponds to the same value 2^{-2} since the binary representation of $\frac{1}{4}$ is $0\,\frac{1}{2} + 1\,\frac{1}{4}$. Finally, we have

$$\begin{aligned} \texttt{sum}(L, 4) &= \texttt{trunc}(\alpha_4, \texttt{sum}(L, 3)) + 2^{-\alpha_4} \\ &= \texttt{trunc}(3, 2^{-1}) + 2^{-3} = 2^{-1} + 2^{-3}. \end{aligned}$$

Nakatsu [115] proved the following important result (an equivalent result was independently given by Yeung [139], see Section 2.4 for further details).

Theorem 10 ([115]). *Let* $L = \langle \ell_1, \ldots, \ell_m \rangle$ *be a list of integers, associated with the ordered symbols* $s_1 \prec \cdots \prec s_m$. *There exists an alphabetical code for which the codeword assigned to symbol* s_i *has length* ℓ_i, *for each* $i = 1, \ldots, m$, *if and only if* $\texttt{sum}(L, m) < 1$.

Nakatsu proved the sufficient part of Theorem 10 by showing that the first ℓ_i digits of the binary representation of $\mathtt{sum}(L, i)$, for $i = 1, \ldots, m$, (under the hypothesis that $\mathtt{sum}(L, m) < 1$) provide an alphabetic code for the ordered set of symbols $s_1 \prec \cdots \prec s_m$. Intuitively, when $\mathtt{sum}(L, i) < 1$, one can see that $\mathtt{sum}(L, i)$ enjoys the property of being *the smallest value* strictly greater than $\mathtt{sum}(L, i-1)$ that differs from $\mathtt{sum}(L, i-1)$ on at least the α_i^{th} bit of their binary expansion, where $\alpha_i = \min(\ell_{i-1}, \ell_i)$. This property has important implications for their binary representations. Indeed, by denoting with w_i the first ℓ_i bits of the binary expansion of $\mathtt{sum}(L, i)$ and with w_{i-1} the first ℓ_{i-1} bits of the binary expansion of $\mathtt{sum}(L, i-1)$, the property guarantees that w_{i-1} *comes before* w_i in the standard lexicographic order among binary sequences. Moreover, the property ensures that neither w_i is a prefix of w_{i-1} nor w_{i-1} is a prefix of w_i.

In Lemma 2, we extend its result by showing that an alphabetic code, whose existence is guaranteed by Nakatsu's condition (Theorem 10), can be constructed in time $O(m)$, where m is the number of symbols.

First, we need the following technical result, whose proof is provided in Appendix A.

Lemma 1. *Let $L = \langle \ell_1, \ldots, \ell_m \rangle$ be a list of integers such that $\mathtt{sum}(L, m) < 1$, let i, j be integers such that $1 \leq i < j \leq m$, and let $\{\mathtt{sum}(L, i), \mathtt{sum}(L, i+1), \ldots, \mathtt{sum}(L, j)\}$ be a subset of consecutive elements of $\{\mathtt{sum}(L, 1), \ldots, \mathtt{sum}(L, m)\}$. For each $k = i, \ldots, j-1$, let t_k be the* smallest *integer s for which the binary expansion of $\mathtt{sum}(L, k)$ differs from the binary expansion of $\mathtt{sum}(L, k+1)$ on the s^{th} bit, and define $t_{ij} = \min_{i \leq k < j} t_k$. Then, it holds that:*

1. $$t_{ij} = \lceil -\log_2(\mathtt{sum}(L, i) \oplus \mathtt{sum}(L, j)) \rceil, \tag{3.5}$$

   where $\mathtt{sum}(L, i) \oplus \mathtt{sum}(L, j)$ is the numerical *value whose binary expansion is the result of the XOR operation between the binary expansions of $\mathtt{sum}(L, i)$ and $\mathtt{sum}(L, j)$, and*

2. *there exists an index $z \in \{i, \ldots, j-1\}$ such that $\mathtt{sum}(L, z) < \mathtt{trunc}(t_{ij}, \mathtt{sum}(L, i)) + 2^{-t_{ij}}$ and $\mathtt{sum}(L, z+1) \geq \mathtt{trunc}(t_{ij}, \mathtt{sum}(L, i)) + 2^{-t_{ij}}$.*

For completeness, we recall that the XOR $\mathtt{sum}(L,i) \oplus \mathtt{sum}(L,j)$ is computable, in most programming languages, with a single built-in instruction. Using Lemma 1, we can now prove our constructive result. To avoid overburdening the notation, when the pair of integers (i,j) in (3.5) is clear from the context, we will simply use t instead of t_{ij}.

Lemma 2. *Let $L = \langle \ell_1, \dots, \ell_m \rangle$ be a list of integers, associated with the ordered symbols $s_1 \prec \cdots \prec s_m$. If $\mathtt{sum}(L,m) < 1$, then one can construct in $O(m)$ time an alphabetic code C for which the codeword assigned to symbol s_i has length at most $\min(\ell_i, m-1)$, for each $i = 1, \dots, m$.*

Proof. To prove the lemma, we provide an $O(m)$ time algorithm to construct the binary tree associated with the code C, that is, the binary tree whose root-to-leaves paths give the codewords of C.

We recall that Nakatsu [115, Theorem 10] showed that the code constructed by taking the first ℓ_i bits of the binary expansion of $\mathtt{sum}(L,i)$ as the codeword associated to the symbol s_i, for each $i = 1, \dots, m$, is alphabetic (or more precisely, prefix and alphabetic). Our algorithm constructs the binary codewords by using a *subset* of the first ℓ_i bits of the binary expansion of $\mathtt{sum}(L,i)$, for each $i = 1, \dots, m$. The algorithm builds the binary tree in a top-down fashion, proceeding by iterated bisections of the set

$$S = \{\mathtt{sum}(L,1), \mathtt{sum}(L,2), \dots, \mathtt{sum}(L,m)\}. \tag{3.6}$$

Initially, the root of the tree is associated with the whole set S. To perform the first bisection, we proceed as follows.

For each $a = 1, \dots, m-1$, let t_a be the smallest integer for which the binary expansions of the pair $\mathtt{sum}(L,a)$ and $\mathtt{sum}(L,a+1)$ differ on the t_a^{th} bit, and let $t = t_{1m} = \min_{1 \le a < m} t_a$. From Lemma 1, we have that $t = \lceil -\log_2(\mathtt{sum}(L,1) \oplus \mathtt{sum}(L,m)) \rceil$. We now compute the index $k \in \{1, \dots, m-1\}$ for which it holds that

$$\mathtt{sum}(L,k) < \mathtt{trunc}(t, \mathtt{sum}(L,1)) + 2^{-t} \tag{3.7}$$

and

$$\mathtt{sum}(L,k+1) \ge \mathtt{trunc}(t, \mathtt{sum}(L,1)) + 2^{-t}. \tag{3.8}$$

We associate the left child of the root with the subset

$$\{\mathtt{sum}(L,1), \dots, \mathtt{sum}(L,k)\},$$

and the right child of the root with the subset

$$\{\mathtt{sum}(L, k+1), \ldots, \mathtt{sum}(L, m)\}.$$

By 2) of Lemma 1, both $\{\mathtt{sum}(L,1), \ldots, \mathtt{sum}(L,k)\}$ and $\{\mathtt{sum}(L, k+1), \ldots, \mathtt{sum}(L,m)\}$ are non empty. The subset $\{\mathtt{sum}(L,1), \ldots, \mathtt{sum}(L,k)\}$ represents the nodes that are in the left subtree of the root, and by the definition of the function `trunc` and from (3.7) and (3.8), each numerical value $\mathtt{sum}(\cdot,\cdot) \in \{\mathtt{sum}(L,1), \ldots, \mathtt{sum}(L,k)\}$ has a binary expansion with 0 in the t^{th} position. Analogously, the subset $\{\mathtt{sum}(L,k+1), \ldots, \mathtt{sum}(L,m)\}$ represents the nodes that are in the right subtree of the root and each value $\mathtt{sum}(\cdot,\cdot) \in \{\mathtt{sum}(L, k+1), \ldots, \mathtt{sum}(L,m)\}$ has a binary expansion with 1 in the t^{th} position. The algorithm proceeds recursively on the *left* subset $\{\mathtt{sum}(L,1), \ldots, \mathtt{sum}(L,k)\}$ and *right* subset $\{\mathtt{sum}(L,k+1), \ldots, \mathtt{sum}(L,m)\}$. It stops when the cardinality of the subset is equal to 1, in such a case, we have just a leaf, or when it is equal to 2, where we have a final subtree with two sibling leaves. The complete pseudo-code of the algorithm is presented in **Algorithm 7**.

The procedure presented above for the construction of the binary tree can be implemented to work in $O(m)$ time and space, by exploiting a technique devised in [47]. In fact, the crucial operation in Line 9 of Algorithm 6 can be implemented in $O(\log \min(s, j-s))$ time, where $s = k - i + 1$, by first checking if $\mathtt{sum}(L,r) < \mathtt{trunc}(t, \mathtt{sum}(L,i)) + 2^{-t}$ with $r = \lceil (j-1+i)/2 \rceil$ to discover whether the index k belongs to $\{i, \ldots, r-1\}$ or to $\{r, \ldots, j-1\}$. Successively, we use an exponential search to find the interval in which the value k is located, and, finally, a binary search in such an interval to find k. More precisely, we proceed in the following way: if $k \in \{i, \ldots, r-1\}$, we check the validity of the inequality $\mathtt{sum}(L, i+2^v) < \mathtt{trunc}(t, \mathtt{sum}(L,i)) + 2^{-t}$ in Line 9 for increasing values $v = 0, 1, 2, 3, \ldots$, till we find the first value of v that satisfies the inequality $\mathtt{sum}(L, i+2^v) \geq \mathtt{trunc}(t, \mathtt{sum}(L,i)) + 2^{-t}$. After $\log s$ step, we shall have located the value of k we seek within an interval of size at most $2s$. A successive binary search will exactly determine the value of k. Altogether, this procedure takes $O(\log s)$ steps. Similarly, if $k \in \{r, \ldots, j-1\}$, we apply the same procedure with the caveat that we start the exponential search from the right end of the interval $\{r, \ldots, j-1\}$. In this way, the procedure will

Algorithm 6: SubTree(i,j)

if $i = j$ **then**
  Construct the leaf v associated with the symbol s_i.
  return v
else if $i + 1 = j$ **then**
  Construct a node v whose left child is the leaf s_i and whose right child is the leaf s_j.
  return v
else
  $t = \lceil -\log_2(\mathtt{sum}(L,i) \oplus \mathtt{sum}(L,j)) \rceil$
  Let $k \in \{i, \dots, j-1\}$ be the index such that $\mathtt{sum}(L,k) < \mathtt{trunc}(t, \mathtt{sum}(L,i)) + 2^{-t}$ and $\mathtt{sum}(L,k+1) \geq \mathtt{trunc}(t, \mathtt{sum}(L,i)) + 2^{-t}$.
  $l = \text{SubTree}(i,k)$
  $r = \text{SubTree}(k+1,j)$
  Construct a node v whose left child is the tree l and whose right child is the tree r.
  return v

Algorithm 7: ConstructTree(L)

Input: The list of lengths $L = \langle \ell_1, \dots, \ell_m \rangle$
Output: An alphabetic code tree with codeword lengths $\ell'_i \leq \min(\ell_i, m-1)$, for each $i = 1, \dots, m$.

Compute $\mathtt{sum}(L,i)$ for each $i = 1, \dots, m$.
return SubTree$(1,m)$

take $O(\log(j-s))$ time. Putting the two cases together, the whole operation takes $O(\log \min(s, j-s))$ time.

Therefore, by denoting with $T(m)$ the number of operations performed by the algorithm to construct the tree with m leaves, because of Line 9, 10, and 11, we have that

$$\begin{aligned} T(m) &\leq \max_{1\leq k\leq m-1} \{T(m-k) + T(k) \\ &\quad + \alpha \log_2 \min(k, m-k) + \beta\} \\ &= \max_{1\leq k\leq m/2} \{T(m-k) + T(k) + \alpha \log_2 k + \beta\}, \end{aligned} \tag{3.9}$$

for suitable constant values α and β.

One can see that the right-hand side of (3.9) grows at most linearly with m (e.g., Theorem 11, p. 185 of [111]); therefore Algorithm 7 (ConstructTree(L)) requires total time $O(m)$.

The tree produced by this algorithm is alphabetic by construction since the recursive bisections of the set

$$\{\mathtt{sum}(L,1), \mathtt{sum}(L,2), \dots, \mathtt{sum}(L,m)\}$$

preserve the symbol order $s_1 \prec \cdots \prec s_m$. What is left to prove is that the procedure constructs a tree whose root-to-leaves paths (i.e., codewords) have length $\ell'_a \leq \min(\ell_a, m-1)$, for each $a = 1, \dots, m$.

The inequality $\ell'_a \leq m-1$ holds because the constructed binary tree is *full*, that is, each node internal node of the tree has two children. This is due to 2) of Lemma 1 and Line 9,10 and 11 of **Algorithm 6**.

We now prove that $\ell'_a \leq \ell_a$, for each $a = 1, \dots, m$. Consider first the cases $a \in \{2, \dots, m-1\}$. By definition of $\mathtt{sum}(\cdot,\cdot)$, we have that

$$\mathtt{sum}(L,a) = \mathtt{trunc}(\alpha_a, \mathtt{sum}(L, a-1)) + 2^{-\alpha_a}, \tag{3.10}$$

where $\alpha_a = \min(\ell_{a-1}, \ell_a)$. Therefore, since $\mathtt{trunc}(b,x)$ is the binary fraction one gets by considering only the first b bits of the binary expression of x (cfr. (3.3)), from (3.10) one obtains that the binary expansion of $\mathtt{sum}(L,a)$ differs from the binary expansion of $\mathtt{sum}(L,a-1)$ on at least the α_a^{th} bit. Because of Line 9,10 and 11 of **Algorithm 6**, this implies that after at most α_a bisection steps, the value of $\mathtt{sum}(L,a)$ can no longer be in the same subset of $\mathtt{sum}(L,a-1)$. By symmetry, after at most α_{a+1} steps, the value $\mathtt{sum}(L,a)$ can no longer be in the same subset of $\mathtt{sum}(L,a+1)$. All

together, after at most $\max(\alpha_a, \alpha_{a+1})$ bisection steps, the value $\texttt{sum}(L, a)$ does not belong to the same subset of either $\texttt{sum}(L, a-1)$ or $\texttt{sum}(L, a+1)$, that is, $\texttt{sum}(L, a)$ must correspond to a leaf in the code-tree. Therefore, the corresponding root-to-leaves path has length

$$\begin{aligned}\ell'_a &\leq \max(\alpha_a, \alpha_{a+1}) = \max(\min(\ell_{a-1}, \ell_a), \min(\ell_a, \ell_{a+1})) \\ &\leq \ell_a.\end{aligned}$$

Analogously, for $a = 1$ it holds that

$$\ell'_1 \leq \alpha_2 = \min(\ell_1, \ell_2) \leq \ell_1,$$

and for $a = m$ it holds that

$$\ell'_m \leq \alpha_m = \min(\ell_{m-1}, \ell_m) \leq \ell_m.$$

□

Let us clarify the algorithm's idea with an example. Figure 3.1 displays two alphabetic trees for the list of lengths

$$L = \langle 6, 6, 5, 2, 9, 9, 8, 6, 3, 2 \rangle.$$

The first one is constructed by taking the first ℓ_i bits of the binary expansion of $\texttt{sum}(L, i)$ (as done by Nakatsu [115]). The second one is obtained by the application of our linear-time algorithm. When comparing the two trees, one can see how the codewords of the code tree built by our algorithm are subsequences of that built through Nakatsu's procedure. For instance, one can see how the codeword associated with the symbol s_7 in our code tree, which is 10001, is only a subsequence of the codeword associated with the symbol s_7 in the first tree, that is, 10000001, where the underlined bits are those that correspond to the codeword of our code tree. This property is crucial: it means our algorithm can efficiently construct a compact code tree, even if some lengths in the input list are exponentially large. Indeed, in our code tree, all codeword lengths remain at most linear with respect to the input size.

Before we present our improved results on the average length of optimal alphabetic codes (as anticipated in Section 2.5), we will briefly address a problematic issue in the analysis of the bound (2.13) claimed in [37].

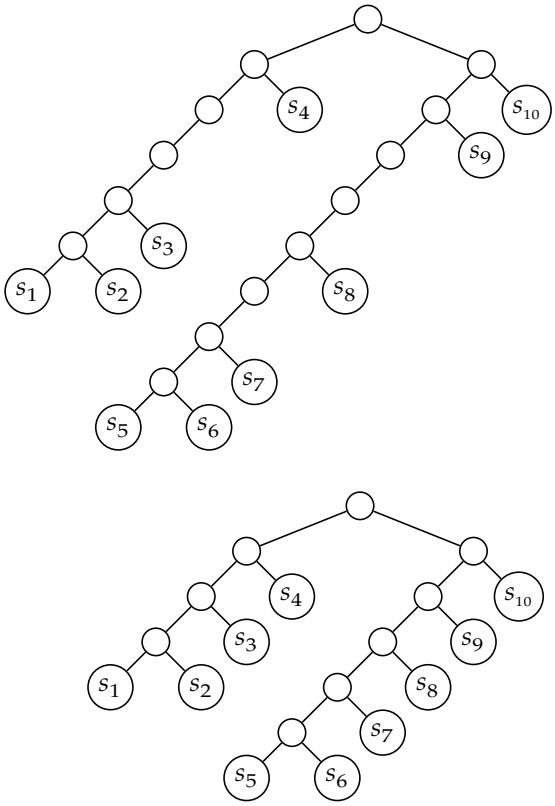


Figure 3.1: The alphabetic trees constructed on the list of lengths $L = \langle 6, 6, 5, 2, 9, 9, 8, 6, 3, 2 \rangle$. On the left, the tree constructed by taking the first ℓ_i bits of the binary expansion of $\texttt{sum}(L, i)$, (according to Nakatsu [115]), on the right, the tree constructed by our algorithm ConstructTree(L).

Remark 1. *Let us first of all recall the idea of the proof of the upper bound (2.13) presented in [37]. The authors proceed as follows: Starting from an input probability distribution $\phi = \langle \phi_1, \dots, \phi_m \rangle$, they first construct the fully extended distribution $\nu' = \langle 0, \phi_1, 0, \dots, 0, \phi_m, 0 \rangle$ and, subsequently, apply the classic Gilbert and Moore's algorithm [56] (see Algorithm 4) to construct an alphabetic code C for the distribution $\nu' = \langle 0, \phi_1, 0, \dots, 0, \phi_m, 0 \rangle$. The average length of the constructed code is less than $H(\nu') + 2 = H(\phi) + 2$. Successively, they prune the binary tree representing the alphabetical code C and show that the average length of the new code so obtained satisfies the upper bound (2.13). In [37], the codeword lengths associated with the null probabilities are left unspecified, while for non-null probabilities, the codeword lengths are*

$$\lceil -\log_2 \phi_i \rceil + 1,$$

as required by the Gilbert-Moore algorithm.

Next, we illustrate the problematic issue in the analysis in [37] through a numeric example. Successively, we generalize the example to infinitely many cases in Appendix A.2.

Let $\phi = \langle \phi_1, \phi_2, \phi_3 \rangle = \left\langle \frac{1}{2}, \frac{1}{4}, \frac{1}{4} \right\rangle$ and let ν' be its fully extended distribution

$$\nu' = \left\langle 0, \frac{1}{2}, 0, \frac{1}{4}, 0, \frac{1}{4}, 0 \right\rangle.$$

As recalled above, the Gilbert-Moore algorithm assigns codeword lengths $\lceil -\log_2 \phi_1 \rceil + 1 = 2$, $\lceil -\log_2 \phi_2 \rceil + 1 = 3$ and $\lceil -\log_2 \phi_3 \rceil + 1 = 3$ to the symbols of probabilities ϕ_1, ϕ_2 and ϕ_3, respectively. However, we will show that there does not exist any alphabetic code for ν' with codeword lengths

$$L = \langle f_1, \dots, f_7 \rangle = \langle x_1, 2, x_2, 3, x_3, 3, x_4 \rangle, \tag{3.11}$$

for any choice of the codeword lengths $x_1, x_2, x_3, x_4 \in \mathbb{N}_+$ associated to the symbols of null probability.

Let us start with the choice $x_1 = 2$ and $x_2 = x_3 = x_4 = 3$. It is immediate to verify that in such a case $\mathtt{sum}(L, 7) = 1$. Therefore, by Theorem 10 there cannot exist an alphabetic code with the chosen lengths. In fact:

$$\mathtt{sum}(L, 2) = \mathtt{trunc}(2, \mathtt{sum}(L, 1)) + 2^{-2} = 2^{-2}$$

$$\begin{aligned}
\mathtt{sum}(L,3) &= \mathtt{trunc}(2,\mathtt{sum}(L,2)) + 2^{-2} = 2^{-1}\\
\mathtt{sum}(L,4) &= \mathtt{trunc}(3,\mathtt{sum}(L,3)) + 2^{-3} = 2^{-1} + 2^{-3}\\
\mathtt{sum}(L,5) &= \mathtt{trunc}(3,\mathtt{sum}(L,4)) + 2^{-3} = 2^{-1} + 2^{-2}\\
\mathtt{sum}(L,6) &= \mathtt{trunc}(3,\mathtt{sum}(L,5)) + 2^{-3} = 2^{-1} + 2^{-2} + 2^{-3}\\
\mathtt{sum}(L,7) &= \mathtt{trunc}(3,\mathtt{sum}(L,6)) + 2^{-3} = 2^{-1} + 2^{-2} + 2^{-2}\\
&= 1.
\end{aligned}$$

Similarly, if we increase the lengths of any x_i, the value of the function $\mathtt{sum}(L,7)$ *remains the same, since the values of* $\alpha_i = \min(f_{i-1}, f_i)$ *do not change. Even if we decrease the lengths x_i, considering all possible combinations, we can also verify that the value of* $\mathtt{sum}(L,7)$ *remains greater than or equal to 1. In fact, for instance, let us consider* $x_1 = 1$, $x_2 = x_3 = 2$ *and* $x_4 = 3$. *It follows that:*

$$\begin{aligned}
\mathtt{sum}(L,2) &= \mathtt{trunc}(1,\mathtt{sum}(L,1)) + 2^{-1} = 2^{-1}\\
\mathtt{sum}(L,3) &= \mathtt{trunc}(2,\mathtt{sum}(L,2)) + 2^{-2} = 2^{-1} + 2^{-2}\\
\mathtt{sum}(L,4) &= \mathtt{trunc}(2,\mathtt{sum}(L,3)) + 2^{-2} = 2^{-1} + 2^{-2} + 2^{-2}\\
&= 1\\
\mathtt{sum}(L,5) &= \mathtt{trunc}(2,\mathtt{sum}(L,4)) + 2^{-2} = 2^{-1} + 2^{-2} + 2^{-2}\\
&\quad + 2^{-2} = 1.25\\
\mathtt{sum}(L,6) &= \mathtt{trunc}(2,\mathtt{sum}(L,5)) + 2^{-2} = 2^{-1} + 2^{-2} + 2^{-2}\\
&\quad + 2^{-2} + 2^{-2} = 1.5\\
\mathtt{sum}(L,7) &= \mathtt{trunc}(3,\mathtt{sum}(L,6)) + 2^{-3} = 2^{-1} + 2^{-2} + 2^{-2}\\
&\quad + 2^{-2} + 2^{-2} + 2^{-3} = 1.625.
\end{aligned}$$

Therefore, by Theorem 10, there cannot exist an alphabetic code with lengths (3.11).

We can now proceed to present our new results for alphabetic codes. We divide our analysis into two cases. In the first case, we provide an improved upper bound on the average length of alphabetic codes for *dyadic distributions*. A dyadic distribution $\phi = \langle \phi_1, \ldots, \phi_m \rangle$ is a probability distribution where each ϕ_i is equal to 2^{-k_i}, for suitable integers $k_i > 0$.

In the second case, we focus on providing an upper bound on the average length of alphabetic codes for all other probability distributions.

Theorem 11. *For any dyadic probability distribution $\phi = \langle \phi_1, \dots, \phi_m \rangle$ on the symbols $s_1 \prec \cdots \prec s_m$, we can construct in linear time an alphabetic code C with*

$$\mathbb{E}[C] \leq H(\phi) + 1 - \phi_1 - \phi_m. \tag{3.12}$$

Proof. The idea of the proof is simple and is based on Yeung's analysis [139, Thm. 5]. Yeung proved that, given a probability distribution $\phi = \langle \phi_1, \dots, \phi_m \rangle$, there exists an alphabetic code C for ϕ with lengths of the codewords $L = \langle \ell_1, \dots, \ell_m \rangle$ equal to:

$$\ell_i = \begin{cases} \lceil -\log_2 \phi_i \rceil & \text{if either } i = 1 \text{ or } i = m, \\ \lceil -\log_2 \phi_i \rceil + 1 & \text{otherwise.} \end{cases}$$

From Lemma 2, the code C can be constructed in $O(m)$ time.

Using the additional hypothesis that ϕ is a dyadic distribution, it follows that $\lceil -\log_2 \phi_i \rceil = -\log_2 \phi_i$ for each $i = 1, \dots, m$. Therefore, the average length of the alphabetic code C with lengths L is

$$\begin{aligned} \mathbb{E}[C] &\leq \sum_{i=1}^{m} \phi_i \ell_i \\ &= \phi_1 (-\log_2 \phi_1) + \phi_m (-\log_2 \phi_m) \\ &\quad + \sum_{i=2}^{m-1} \phi_i (-\log_2 \phi_i + 1) \\ &= -\sum_{i=1}^{m} \phi_i \log_2 \phi_i + \sum_{i=2}^{m-1} \phi_i \\ &= H(\phi) + 1 - \phi_1 - \phi_m. \end{aligned}$$

□

We now turn our attention to the general case of non-dyadic probability distributions $\phi = \langle \phi_1, \dots, \phi_m \rangle$.

Let us first consider the case in which the first and last probabilities, ϕ_1 and ϕ_m, are positive. For our analysis, we also need the following property of the function $\texttt{sum}(\cdot, \cdot)$, whose proof is given in Appendix A.3.

Lemma 3. *Let $\ell_1, \dots, \ell_m$ be a sequence of positive integers, and let $k \geq \max_i \ell_i$. The function* sum *on the list $L = \langle f_1, f_2, \dots, f_{2m-1} \rangle =$*

$\langle \ell_1, k, \ell_2, k, \dots, k, \ell_m \rangle$ of $2m-1$ elements satisfies the following inequality:

$$\mathtt{sum}(L, 2m-1) \leq 2^{-\ell_1} + 2^{-\ell_m} + 2 \sum_{i=2}^{m-1} 2^{-\ell_i}. \tag{3.13}$$

We can now present one of our main improvements. Before doing so, let us briefly and intuitively describe the idea behind it. As discussed in Section 2.5, the codeword lengths defined by Yeung's approach satisfy both Yeung's and Nakatsu's inequalities, and therefore an alphabetic code with these lengths indeed exists. However, one can see that the corresponding codes in their tree representation typically contain gaps between consecutive leaves, leaving room for improvement. Therefore, through the additional *null* probabilities into our extended distribution, we implicitly encode these gaps between the leaves. In this way, we capture more structural information about the code/tree, which ultimately yields an improvement on its average length.

Theorem 12. *For any non-dyadic probability distribution $\phi = \langle \phi_1, \dots, \phi_m \rangle$ on the symbols $s_1 \prec \cdots \prec s_m$ with $\phi_1, \phi_m > 0$, we can construct in linear time an alphabetic code C with*

$$\begin{aligned} \mathbb{E}[C] \leq & H(\phi) + 2 - \phi_1 \left(2 - \log_2 \phi_1 - \lceil -\log_2 \phi_1 \rceil\right) \\ & - \phi_m \left(2 - \log_2 \phi_m - \lceil -\log_2 \phi_m \rceil\right) \\ & - \sum_{i=1}^{m-1} \min(\phi_i, \phi_{i+1}) \qquad (3.14) \\ < & H(\phi) + 2 - \phi_1 - \phi_m - \sum_{i=1}^{m-1} \min(\phi_i, \phi_{i+1}). \qquad (3.15) \end{aligned}$$

Proof. We proceed as follows. We first show that there exists an alphabetic code C' for the partially extended distribution

$$\nu = \langle \nu_1, \nu_2, \dots, \nu_{2m-1} \rangle = \langle \phi_1, 0, \dots, 0, \phi_m \rangle,$$

whose average length is upper-bounded by

$$\begin{aligned} & H(\phi) + 2 - \phi_1 \left(2 - \log_2 \phi_1 - \lceil -\log_2 \phi_1 \rceil\right) \\ & \quad - \phi_m \left(2 - \log_2 \phi_m - \lceil -\log_2 \phi_m \rceil\right). \end{aligned} \tag{3.16}$$

Successively, we prune the binary tree representing C' by removing the additional $m-1$ leaves (i.e., the leaves corresponding to the

$m-1$ null probabilities in ν). Then, we show that this pruning reduces the quantity (3.16) to (3.14).

Let k be an integer greater than $\max_{j:\phi_j>0}\lceil -\log_2 \phi_j\rceil + 1$ (the value of k will be chosen later, in order to satisfy necessary inequalities), and let $L = \langle \ell_1, \dots, \ell_{2\,m-1}\rangle$ be the lengths of the codewords for the distribution $\nu = \langle \phi_1, 0, \dots, 0, \phi_\rangle$), defined as follows:

$$\ell_i = \begin{cases} k & \text{if } \nu_i = 0, \\ \lceil -\log_2 \nu_i\rceil & \text{if } i = 1 \text{ or } i = 2m-1, \\ \lceil -\log_2 \nu_i\rceil + 1 & \text{if } \nu_i > 0. \end{cases} \tag{3.17}$$

From Lemma 3 we have that

$$\begin{aligned} \mathtt{sum}(L, 2\,m-1) &\le 2^{-\lceil -\log_2 \phi_1\rceil} + 2^{-\lceil -\log_2 \phi_m\rceil} \\ &\quad + 2 \sum_{i\neq 1, m:\phi_i>0} 2^{-(\lceil -\log_2 \phi_i\rceil+1)} + 2 \sum_{i:\phi_i=0} 2^{-k} \\ &= \sum_{i:\phi_i>0} 2^{-\lceil -\log_2 \phi_i\rceil} + 2 \sum_{i:\phi_i=0} 2^{-k} < 1, \end{aligned}$$

where the last inequality holds because $\sum_{i:\phi_i>0} 2^{-\lceil -\log_2 \phi_i\rceil} < 1$ since ϕ is *a non-dyadic* distribution[1], and we can choose a sufficiently large value for k.

Since $\mathtt{sum}(L, 2m-1) < 1$, by Theorem 10, there exists an alphabetic code with lengths ℓ_i defined in (3.17), and from Lemma 2 we can construct in linear time ($O(2m-1) = O(m)$) a code C' whose average length satisfies

$$\begin{aligned} \mathbb{E}[C'] &\le \phi_1\lceil -\log_2 \phi_1\rceil + \phi_m\lceil -\log_2 \phi_m\rceil \\ &\quad + \sum_{i\neq 1, m:\phi_i>0} \phi_i(\lceil -\log_2 \phi_i\rceil + 1) \\ &\le \phi_1\lceil -\log_2 \phi_1\rceil + \phi_m\lceil -\log_2 \phi_m\rceil \\ &\quad + \sum_{i\neq 1, m:\phi_i>0} \phi_i(-\log_2 \phi_i + 2) \\ &\quad (\text{since } \lceil x\rceil < x+1) \end{aligned}$$

1 Notice that if the distribution is dyadic, this inequality might not hold. However, we will see later that the theorem can be extended to dyadic distributions as well.

$$
\begin{aligned}
&= -\phi_1(-\log_2 \phi_1 + 2 - \lceil -\log_2 \phi_1 \rceil) \\
&\quad - \phi_m(-\log_2 \phi_m + 2 - \lceil -\log_2 \phi_m \rceil) \\
&\quad + \sum_{i:\phi_i>0} \phi_i(-\log_2 \phi_i + 2) \\
&= -\sum_{i:\phi_i>0} \phi_i \log_2 \phi_i + 2 \\
&\quad - \phi_1(-\log_2 \phi_1 + 2 - \lceil -\log_2 \phi_1 \rceil) \\
&\quad - \phi_m(-\log_2 \phi_m + 2 - \lceil -\log_2 \phi_m \rceil) \\
&= H(\phi) + 2 - \phi_1(2 - \log_2 \phi_1 - \lceil -\log_2 \phi_1 \rceil) \\
&\quad - \phi_m(2 - \log_2 \phi_m - \lceil -\log_2 \phi_m \rceil). \qquad (3.18)
\end{aligned}
$$

We can now eliminate the $m - 1$ leaves associated with the *additional null* probabilities in ν, proceeding similarly to [37, 40]. First, observe that the leaves associated with the probabilities ν_i appear, from left to right, in the same order as the probabilities ν_i (this is due to the fact that the tree represents an alphabetic code). Moreover, observe that the binary tree of the code C' is full by construction, i.e., each node has exactly zero or two children. Thus, when we prune a leaf x associated with an additional null probability, we bring up one level the whole sub-tree T whose root r is the sibling of x. Since the binary tree is alphabetic, the tree T contains either the leaf associated with a probability in ν that immediately follows the additional null probability associated with x, or it contains the leaf associated with a probability in ν that immediately precedes x. Hence, if x is the "left" child of a, its pruning causes a bump up of one level of the leaf that follows x, and if x is the "right" child of a, it causes a bump up of one level of the leaf that precedes x. Figure 3.2 explains the procedure pictorially.

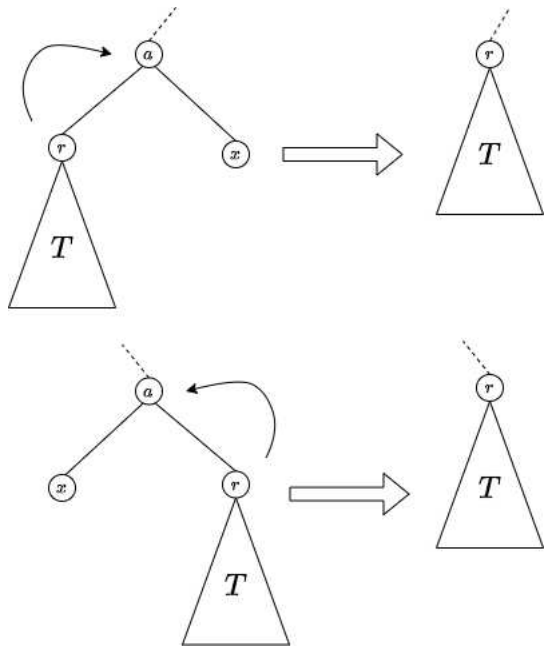


Figure 3.2: The figure shows the two cases discussed above: the elimination of x bumps up of one level the leaf that precedes it (above), and the leaf that follows it (below).

Hence, the elimination of each leaf x associated with an additional null probability in ν bumps up one level at least one leaf among the two leaves that precede and follow x. In total, we bump up leaves whose total probability is at least

$$\sum_{i=1}^{m-1} \min(\phi_i, \phi_{i+1}). \tag{3.19}$$

We recall that the construction of the alphabetic code with average cost (3.18) requires linear time. Similarly, the elimination of the $m-1$ leaves associated with the additional *null* probabilities in ν requires linear time too. Indeed, the operation can be implemented through a simple tree visit.

Thus, putting together (3.18) and (3.19), it follows that the resulting alphabetic code has an average length upper bounded by

$$H(\phi) + 2 - \phi_1 \left(2 - \log_2 \phi_1 - \lceil -\log_2 \phi_1 \rceil\right)$$

$$- \phi_m \left(2 - \log_2 \phi_m - \lceil -\log_2 \phi_m \rceil\right) - \sum_{i=1}^{m-1} \min(\phi_i, \phi_{i+1}).$$

□

Let us now consider the more general case where the first and last probabilities, ϕ_1 and ϕ_m, are not necessarily positive but can be zero.

Theorem 13. *For any non-dyadic probability distribution $\phi = \langle \phi_1, \dots, \phi_m \rangle$ on the symbols $s_1 \prec \cdots \prec s_m$, we can construct in linear time an alphabetic code C with*

$$\begin{aligned}
\mathbb{E}[C] &\leq H(\phi) + 2 - \phi_\ell \left(2 - \log_2 \phi_\ell - \lceil -\log_2 \phi_\ell \rceil\right) \\
&\quad - \phi_k \left(2 - \log_2 \phi_k - \lceil -\log_2 \phi_k \rceil\right) \\
&\quad - \sum_{i=\ell}^{k-1} \min(\phi_i, \phi_{i+1}) + \phi_\ell + \phi_k \\
&= H(\phi) + 2 - \phi_\ell \left(1 - \log_2 \phi_\ell - \lceil -\log_2 \phi_\ell \rceil\right) \\
&\quad - \phi_k \left(1 - \log_2 \phi_k - \lceil -\log_2 \phi_k \rceil\right) - \sum_{i=\ell}^{k-1} \min(\phi_i, \phi_{i+1}),
\end{aligned} \tag{3.20}$$

where $\ell = \arg\min_i \{\phi_i > 0\}$ and $k = \arg\max_i \{\phi_i > 0\}$.

Proof. Let $\phi' = \langle \phi_\ell, \dots, \phi_k \rangle$ be the probability distribution obtained by considering only the elements of ϕ between ϕ_ℓ and ϕ_k. From Theorem 12, we know that we can construct in linear time an alphabetic code C' for ϕ' whose average length satisfies

$$\begin{aligned}
\mathbb{E}[C'] &\leq H(\phi') + 2 - \phi_\ell \left(2 - \log_2 \phi_\ell - \lceil -\log_2 \phi_\ell \rceil\right) \\
&\quad - \phi_k \left(2 - \log_2 \phi_k - \lceil -\log_2 \phi_k \rceil\right) - \sum_{i=\ell}^{k-1} \min(\phi_i, \phi_{i+1}).
\end{aligned} \tag{3.21}$$

We need to consider four cases according to the values of ℓ and k:

1. $\ell = 1$ and $k = m$: in this case $\phi = \phi'$ and the result follows directly from Theorem 12.
2. $\ell > 1$ and $k < m$: then $\phi = \langle \underbrace{0, \dots, 0}_{\ell-1}, \phi_\ell, \dots, \phi_k, \underbrace{0, \dots, 0}_{m-k} \rangle$. We have to create the leaves for the symbols $s_1, \dots, s_{\ell-1}$ and

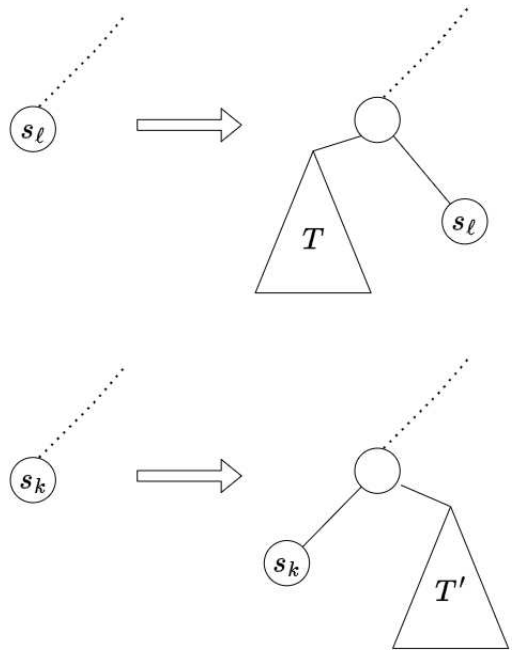


Figure 3.3: The figure shows the insertion of the subtrees T and T', with leaves $s_1, \ldots, s_{\ell-1}$ and $s_{k+1}, \ldots, s_n$, respectively.

$s_{k+1}, \ldots, s_n$. To do so, we let the leaves associated with s_ℓ and s_k descend by one level, and we then insert the subtrees T and T' containing the leaves for $s_1, \ldots, s_{\ell-1}$ and $s_{k+1}, \ldots, s_n$, respectively. This operation is shown pictorially in Figure 3.3.

Thus, from (3.21) and since $H(\phi) = H(\phi')$, it follows that the average length of the new code C obtained satisfies

$$\begin{aligned}\mathbb{E}[C] \leq H(\phi) + 2 - \phi_\ell \left(2 - \log_2 \phi_\ell - \lceil -\log_2 \phi_\ell \rceil\right) \\ - \phi_k \left(2 - \log_2 \phi_k - \lceil -\log_2 \phi_k \rceil\right) \\ - \sum_{i=\ell}^{k-1} \min(\phi_i, \phi_{i+1}) + \phi_\ell + \phi_k.\end{aligned}$$

3. $\ell = 1$ and $k < m$: then $\phi = \langle \phi_1, \dots, \phi_k, \underbrace{0, \dots, 0}_{m-k} \rangle$. Here, we have to create the leaves only for the symbols $s_{k+1}, \dots, s_n$. Therefore, proceeding similarly to case 2), we obtain that

$$\begin{aligned}
\mathbb{E}[C] &\leq H(\phi) + 2 - \phi_\ell \left(2 - \log_2 \phi_\ell - \lceil -\log_2 \phi_\ell \rceil\right) \\
&\quad - \phi_k \left(2 - \log_2 \phi_k - \lceil -\log_2 \phi_k \rceil\right) \\
&\quad - \sum_{i=\ell}^{k-1} \min(\phi_i, \phi_{i+1}) + \phi_k \\
&< H(\phi) + 2 - \phi_\ell \left(2 - \log_2 \phi_\ell - \lceil -\log_2 \phi_\ell \rceil\right) \\
&\quad - \phi_k \left(2 - \log_2 \phi_k - \lceil -\log_2 \phi_k \rceil\right) \\
&\quad - \sum_{i=\ell}^{k-1} \min(\phi_i, \phi_{i+1}) + \phi_\ell + \phi_k.
\end{aligned}$$

4. $\ell > 1$ and $k = m$: then $\phi = \langle \underbrace{0, \dots, 0}_{\ell-1}, \phi_\ell, \dots, \phi_m \rangle$. We have to create the leaves only for the symbols $s_1, \dots, s_{\ell-1}$. Therefore, again proceeding similarly to case 2), we get a new code C with

$$\begin{aligned}
\mathbb{E}[C] &\leq H(\phi) + 2 - \phi_\ell \left(2 - \log_2 \phi_\ell - \lceil -\log_2 \phi_\ell \rceil\right) \\
&\quad - \phi_k \left(2 - \log_2 \phi_k - \lceil -\log_2 \phi_k \rceil\right) \\
&\quad - \sum_{i=\ell}^{k-1} \min(\phi_i, \phi_{i+1}) + \phi_\ell \\
&< H(\phi) + 2 - \phi_\ell \left(2 - \log_2 \phi_\ell - \lceil -\log_2 \phi_\ell \rceil\right) \\
&\quad - \phi_k \left(2 - \log_2 \phi_k - \lceil -\log_2 \phi_k \rceil\right) \\
&\quad - \sum_{i=\ell}^{k-1} \min(\phi_i, \phi_{i+1}) + \phi_\ell + \phi_k.
\end{aligned}$$

□

We have proved Theorem 12 (and subsequently Theorem 13) under the hypothesis that the probability distribution $\phi = \langle \phi_1, \dots, \phi_m \rangle$ is non-dyadic. However, the result can be generalized to include dyadic distributions as well. This generalization is shown in the following corollary, whose proof is postponed in Appendix A.4.

Corollary 13.1. *For any dyadic probability distribution $\phi = \langle \phi_1, \dots, \phi_m \rangle$ on the symbols $s_1 \prec \cdots \prec s_m$, we can construct in linear time an alphabetic code C whose average length satisfies*

$$\begin{aligned}\mathbb{E}[C] &\leq H(\phi) + 2 - \phi_1 \left(2 - \log_2 \phi_1 - \lceil -\log_2 \phi_1 \rceil\right) \\ &\quad - \phi_m \left(2 - \log_2 \phi_m - \lceil -\log_2 \phi_m \rceil\right) - \sum_{i=1}^{m-1} \min(\phi_i, \phi_{i+1}) \\ &= H(\phi) + 2 - 2\phi_1 - 2\phi_m - \sum_{i=1}^{m-1} \min(\phi_i, \phi_{i+1}). \quad (3.22)\end{aligned}$$

Furthermore, Theorem 12 can be explicitly improved under a slightly stronger hypothesis, specifically, that the first two and the last two probabilities are positive. This improvement is presented in the following corollary, whose proof is provided in Appendix A.5.

Corollary 13.2. *For any probability distribution $\phi = \langle \phi_1, \dots, \phi_m \rangle$ on the symbols $s_1 \prec \cdots \prec s_m$, with $\phi_1, \phi_2, \phi_{m-1}, \phi_m > 0$, we can construct in linear time an alphabetic code C with*

$$\begin{aligned}\mathbb{E}[C] \leq & H(\phi) + 2 - \phi_1 (2 - \log_2 \phi_1 - \lceil -\log_2 \phi_1 \rceil) \\ & - \phi_m (2 - \log_2 \phi_m - \lceil -\log_2 \phi_m \rceil) \\ & - \sum_{i=1}^{m-1} \min(\phi_i, \phi_{i+1}) \\ & - \phi_1 \max(0, \lceil -\log_2 \phi_1 \rceil - \lceil -\log_2 \phi_2 \rceil - 2) \\ & - \phi_m \max(0, \lceil -\log_2 \phi_m \rceil - \lceil -\log_2 \phi_{m-1} \rceil - 2). \quad (3.23)\end{aligned}$$

Before concluding this section on alphabetic codes, we will present a few final remarks on our results.

By considering our specific upper bounds for dyadic distributions, we remark that the bounds (3.12) and (3.22) are not comparable, in the sense that there are dyadic probability distributions ϕ for which (3.12) is better than (3.22), and others for which it holds the opposite. For instance, if a dyadic $\phi = \langle \phi_1, \dots, \phi_m \rangle$ is ordered in non decreasing fashion, then (3.22) becomes

$$\mathbb{E}[C] \leq H(\phi) + 1 - 2\phi_1 - \phi_m,$$

and one can see that it is better than the bound (3.12). On the other hand, if a dyadic $\phi = \langle \phi_1, \dots, \phi_m \rangle$ is ordered according to the "saw-tooth" order, that is,

$$\phi_1 < \phi_2 > \phi_3 < \phi_4 > \dots > \phi_{m-1} < \phi_m,$$

with m even, then (3.22) becomes

$$\mathbb{E}[C] \leq H(\phi) + 2\phi_2 + 2\phi_4 \dots + 2\phi_{m-2} - \phi_1. \tag{3.24}$$

Thus, one can easily verify that there are dyadic probability distributions $\phi = \langle \phi_1, \dots, \phi_m \rangle$ for which (3.12) is better than (3.24).

Remark 2. *It might be worthwhile to point out a way to potentially improve the upper bound (3.14) of Theorem 12. For simplicity, let us assume that all probabilities ϕ_i have distinct values (but the reasoning holds in general).*

The core idea is to refine the analysis of which symbols get bumped up during the pruning phase described in Theorem 12. Let us consider the lengths defined in (3.17). After the pruning phase, we know that between ϕ_i and ϕ_{i+1} it is the smaller probability to be bumped up, in the worst case. Therefore, let S_1 be the set of all indexes $i \in \{2, \dots, m-1\}$ for which holds that $\phi_i = \min(\phi_{i-1}, \phi_i)$ and $\phi_i = \min(\phi_i, \phi_{i+1})$, i.e., the set of the symbols that in the worst case are bumped up at least two times. Similarly, let S_2 be the set of all indexes $i \in \{2, \dots, m-1\}$ for which holds that $\phi_i = \min(\phi_{i-1}, \phi_i)$ or $\phi_i = \min(\phi_i, \phi_{i+1})$ (but not both), i.e., the set of all symbols that in the worst case are bumped up at least one time (except the first and last symbol). Whereas, for the first and last symbol, let $S_3 \subseteq \{1, m\}$ be a set such that $1 \in S_3$ if $\min(\phi_1, \phi_2) = \phi_1$, and similarly $m \in S_3$ if $\min(\phi_{m-1}, \phi_m) = \phi_m$. Finally, let S_4 be the set of the remaining indexes (except the first and the last one) that in the worst case are not bumped up, i.e., $S_4 = \{2, \dots, m-1\} \setminus (S_1 \cup S_2)$. By partitioning the symbols in this way, we can upper bound the average length of the obtained alphabetic code in the following way:

$$\begin{aligned}
\mathbb{E}[C] &\leq \sum_{i=2}^{m-1} \phi_i(\lceil -\log_2 \phi_i \rceil + 1) + \phi_1(\lceil -\log_2 \phi_1 \rceil) \\
&\quad + \phi_m(\lceil -\log_2 \phi_m \rceil) - 2 \sum_{i \in S_1} \phi_i - \sum_{i \in S_2} \phi_i - \sum_{i \in S_3} \phi_i \\
&= \sum_{i \in S_4} \phi_i(\lceil -\log_2 \phi_i \rceil + 1)
\end{aligned}$$

$$+ \sum_{i \in S_1 \cup S_2 \cup \{1,m\}} \phi_i(\lceil -\log_2 \phi_i \rceil) - \sum_{i \in S_1} \phi_i - \sum_{i \in S_3} \phi_i \tag{3.25}$$

$$< \sum_{i \in S_4} \phi_i(-\log_2 \phi_i + 2) + \sum_{i \in S_1 \cup S_2 \cup \{1,m\}} \phi_i(\lceil -\log_2 \phi_i \rceil) - \sum_{i \in S_1} \phi_i - \sum_{i \in S_3} \phi_i \qquad (\textit{since } \lceil x \rceil < x + 1)$$

$$= \sum_{i \in S_4} \phi_i(-\log_2 \phi_i + 2) + \sum_{i \in S_1 \cup S_2 \cup \{1,m\}} \phi_i(\lceil -\log_2 \phi_i \rceil) - \sum_{i \in S_1} \phi_i - \sum_{i \in S_3} \phi_i - \sum_{i \in S_1 \cup S_2 \cup \{1,m\}} \phi_i(-\log_2 \phi_i + 2) + \sum_{i \in S_1 \cup S_2 \cup \{1,m\}} \phi_i(-\log_2 \phi_i + 2)$$

$$= \sum_{i=1}^{m} \phi_i(-\log_2 \phi_i + 2) - \sum_{i \in S_1 \cup S_2 \cup \{1,m\}} \phi_i(2 - \log_2 \phi_i - \lceil -\log_2 \phi_i \rceil) - \sum_{i \in S_1} \phi_i - \sum_{i \in S_3} \phi_i$$

$$= H(\phi) + 2 - \sum_{i \in S_1 \cup S_2 \cup \{1,m\}} \phi_i(2 - \log_2 \phi_i - \lceil -\log_2 \phi_i \rceil) - \sum_{i \in S_1} \phi_i - \sum_{i \in S_3} \phi_i \tag{3.26}$$

$$< H(\phi) + 2 - \phi_1(2 - \log_2 \phi_1 - \lceil -\log_2 \phi_1 \rceil) - \phi_m(2 - \log_2 \phi_m - \lceil -\log_2 \phi_m \rceil) - \sum_{i \in S_1 \cup S_2} \phi_i - \sum_{i \in S_1 \cup S_3} \phi_i$$

$$(\textit{since } 2 - \log_2 \phi_i - \lceil -\log_2 \phi_i \rceil > 1)$$

$$= H(\phi) + 2 - \phi_1 (2 - \log_2 \phi_1 - \lceil -\log_2 \phi_1 \rceil) - \phi_m (2 - \log_2 \phi_m - \lceil -\log_2 \phi_m \rceil) - \sum_{i=1}^{m-1} \min(\phi_i, \phi_{i+1})$$

$$(\textit{since } \textstyle\sum_{i \in S_1 \cup S_2} \phi_i + \sum_{i \in S_1 \cup S_3} \phi_i = \sum_{i=1}^{m-1} \min(\phi_i, \phi_{i+1})).$$

Therefore, it follows that (3.25) *and* (3.26) *cannot be worse than the upper bound* (3.14) *of Theorem 12. But we also notice that the right-hand side of* (3.25) *and* (3.26) *depends on the content of the sets* S_1, S_2, S_3 *and* S_4*. Thus, working out an explicit estimate of the sums appearing on the right-hand side of* (3.25) *and* (3.26) *would lead to further improvements*

to (3.14). However, doing so seems challenging. For this reason, we leave it as an open problem for future investigations in the field.

3.3 LINEAR-TIME FRAMEWORK FOR ALMOST-OPTIMAL BINARY SEARCH TREES

In this section, we turn our attention to BSTs. Since in Chapter 2 we primarily focused on alphabetic codes, in Section 3.3.1, we will recall some of the main and most relevant results on BSTs from the literature for our purpose. Then, in Section 3.3.2, we proceed to present our new results. Specifically, we provide a framework that, in linear time, allows the construction of a BST from an alphabetic code. Subsequently, we demonstrate how this framework, combined with the new results discussed in Section 3.2, enables us to provide new and improved upper bounds on the average length of BSTs as well.

3.3.1 *Background*

As previously anticipated, BSTs can be seen as a generalization of alphabetic codes. In fact, while alphabetic codes are equivalent to comparison-based search procedures that employ binary queries, BSTs are equivalent to comparison-based search procedures that operate through queries that have a ternary outcome. This means each query can result in one of three possibilities: the searched element is equal to the current node, smaller than it, or larger than it. Due to their fundamental role in Computer Science, BSTs have been extensively investigated in the literature (e.g., [40, 91, 93, 110, 111, 116]).

Knuth [91] gave an $O(n^2)$ time and $O(n^2)$ space complexity algorithm to construct an optimum BST, i.e., a BST with minimum cost, as defined in (3.1). We notice that the algorithm provided by Knuth is a variation of the one we discussed in Section 2.3 for constructing optimal alphabetic codes. This underlines, again, the strict relationship between the two combinatorial structures. However, the quadratic complexity of the algorithm designed in [91] may be prohibitive in many applications, as Mehlhorn remarked in

[110]. Therefore, Mehlhorn [110] proposed a linear-time algorithm to construct a BST for an arbitrary probability distribution

$$\sigma = \langle \sigma_1, \dots, \sigma_{2n+1} \rangle = \langle p_0, q_1, p_1, q_2, \dots, q_n, p_n \rangle,$$

where the p_i's are probabilities of unsuccessful searches and the q_i's are probabilities of successful searches. Furthermore, Mehlhorn demonstrated that the cost of the constructed tree in this way is not greater than

$$H(\sigma) + 1 + \sum_{i=0}^{n} p_i. \tag{3.27}$$

Similarly, De Prisco and De Santis [40] proposed a different linear-time algorithm for constructing almost optimal BSTs, and claimed an upper bound on the cost of their tree given by

$$H(\sigma) + 1 - p_0 - p_n + p_{\max}, \tag{3.28}$$

where $p_{\max}$ is the maximum value among $p_0, \dots, p_n$. Unfortunately, the bound (3.28) does not hold in general, as pointed out later in [37].

It is also worth recalling the following work of De Prisco and De Santis [41] in which they provided a lower bound on the average length of an optimal BST. Specifically, they demonstrated that the cost of *any* BST for a probability distribution σ is lower bounded by

$$H(\sigma) + \sum_{i=1}^{n} q_i + H(\sigma) \log_2(H(\sigma)) - (H(\sigma) + 1) \log_2(H(\sigma) + 1). \tag{3.29}$$

3.3.2 *New Bounds on Almost-Optimal Binary Search Trees*

In this section, we exploit the results on alphabetic codes we have derived in Section 3.2 to establish new bounds on the average cost of optimal binary search trees. Specifically, we provide a linear-time algorithm that takes an almost optimal alphabetic code for a given probability distribution σ and transforms it into an almost optimal BST. We also show upper bounds on the cost of the binary trees constructed in this way. It is clear that our upper

bounds apply to optimal trees as well; we will also see that our bounds improve on the best previous results. Moreover, since our algorithms to construct almost-optimal search trees take in input almost-optimal alphabetic trees, in order to get linear time complexity, it is crucial that we can construct almost-optimal alphabetic trees in linear time as well. This is one of the main reasons why we have stressed this feature of the algorithms presented in Section 3.2.

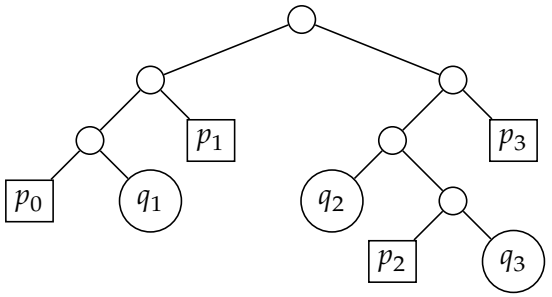
We begin by presenting our linear-time framework for constructing BSTs from alphabetic codes. Our method is a refinement of the algorithm proposed by De Prisco and De Santis [40]. Indeed, by fixing the gap pointed out in their analysis by Dagan [37], we manage to derive a slightly more general result than that of [40].

Theorem 14. *Let $\sigma = \langle \sigma_1, \ldots, \sigma_{2n+1} \rangle = \langle p_0, q_1, p_1, \ldots, q_n, p_n \rangle$ be a probability distribution that contains the probabilities $q_1, \ldots, q_n$ and $p_0, \ldots, p_n$ for successful and unsuccessful searches, respectively, on an ordered sequence $x_1 \prec \cdots \prec x_n$. Let C be* any *alphabetic code for σ, and let $\mathbb{E}[C]$ be its average length. Then, one can construct in linear time, i.e., $O(n)$, a binary search tree T for σ whose average length satisfies*

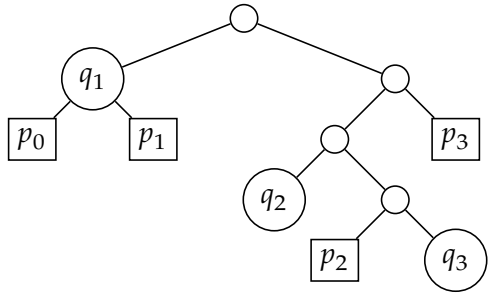
$$\mathbb{E}[T] \leq \mathbb{E}[C] - \sum_{i=1}^{n} q_i - \sum_{i=0}^{n-1} \min(p_i, p_{i+1}). \tag{3.30}$$

Proof. Let us start from the binary tree B corresponding to the alphabetic code C. The tree B has $2n+1$ leaves, which we identify with the ordered sequence of probabilities $p_0, q_1, p_1, \ldots, q_n, p_n$. We transform the leaves corresponding to the q_i's into *internal* nodes. As in [40], we raise each q_i to the *lowest common ancestor* of p_{i-1} and p_i, which is at least two levels above q_i. Hence, $\ell(q_i)$, i.e., the number of nodes from the root to q_i, decreases by at least one unit, for each i. In addition, when we bring up q_i, the whole sub-tree whose root is the sibling of q_i, goes up by at least one level. Hence, at least one between p_{i-1} and p_i goes up one level. [2] Figure 3.4 illustrates an example of the described process.

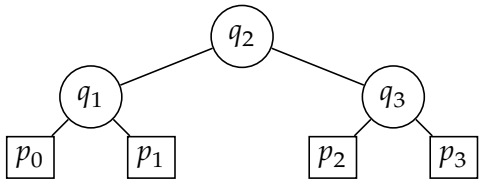
2 The analysis of [40] is incorrect since it claims that all but only one probability goes up one level. One can see that this is not the case, as also observed by the authors of [37].



(a) The initial alphabetic tree



(b) The tree after transforming q_1 into an internal node



(c) The final binary search tree

Figure 3.4: Figure (a) shows the initial alphabetic tree; Figure (b) shows the tree after performing the first step; Figure (c) shows the tree after the application of the algorithm.

Therefore, in total, we bump up leaves whose total probability is at least

$$\sum_{i=1}^{n} q_i + \sum_{i=0}^{n-1} \min(p_i, p_{i+1}). \tag{3.31}$$

Let us prove that the tree T, constructed according to the above procedure, is a valid binary search tree for the distribution $\sigma = \langle \sigma_1, \dots, \sigma_{2n+1} \rangle = \langle p_0, q_1, p_1, \dots, q_n, p_n \rangle$.

To that purpose, let us observe that the search property holds in a binary tree if and only if its in-order visit gives the nodes in the order $p_0, q_1, p_1, \dots, q_n, p_n$. To show that such a property holds in our tree T, we first point out that each probability q_i necessarily occupies a distinct internal node in T. By the sake of contradiction, let us suppose that it is not the case, that is, there exist q_i and q_j, with $i < j$, that have been assigned to the same node x. Then, x is the lowest common ancestor both of p_{i-1} and p_i, and of p_{j-1} and p_j. This implies that p_{i-1} and p_{j-1} are in the left sub-tree of x, while p_i and p_j are in the right sub-tree of x. This flagrantly contradicts the fact that in the starting tree B the leaves appear in the order $p_0, q_1, p_1, \dots, q_n, p_n$ (i.e., that B represents an alphabetic code).

Thus, each q_i is assigned to a distinct lowest common ancestor, and the order of the p_i's is not affected by the transformation of the q_i's in internal nodes. Therefore, an in-order visit of the tree traverses the nodes in the order $p_0, q_1, p_1, \dots, q_n, p_n$. Therefore, we can conclude that T is a correct BST for σ.

The manipulation of the q_i's to transform them from leaves of B to internal nodes of T can be implemented in linear time, as argued in [40].

Finally, we evaluate the cost of the obtained binary search tree T. From (3.31) the cost of T satisfies the inequality

$$\mathbb{E}[T] \leq \mathbb{E}[C] - \sum_{i=1}^{n} q_i - \sum_{i=0}^{n-1} \min(p_i, p_{i+1}).$$

This concludes the proof of the theorem. □

Let us now see how one can use the result of Theorem 14 to derive new upper bounds for optimal BSTs. By plugging into (3.30) either formula (3.12) of Theorem 11, or formula (3.14) of Theorem 12, or formula (3.23) of Corollary 13.2 (according to the hypothesis that holds), one obtains a series of upper bounds on the average length of BSTs that considerably improve on the bound (3.27) of [110]. For instance, by using formula (3.15) of Theorem 12 in formula (3.30), we can construct a BST T whose average length satisfies

$$\mathbb{E}[T] \leq H(\sigma) + 2 - \sigma_1 - \sigma_{2n+1} - \sum_{i=1}^{2n} \min(\sigma_i, \sigma_{i+1})$$

$$
\begin{aligned}
&- \sum_{i=1}^{n} q_i - \sum_{i=0}^{n-1} \min(p_i, p_{i+1}) \\
&= H(\sigma) + 1 + \sum_{i=0}^{n} p_i - \Bigg(p_0 + p_n + \sum_{i=0}^{n-1} \min(p_i, q_{i+1}) \\
&\quad + \sum_{i=1}^{n} \min(q_i, p_i) + \sum_{i=0}^{n-1} \min(p_i, p_{i+1}) \Bigg) \\
&= H(\sigma) + 1 + \sum_{i=0}^{n} p_i - \Bigg(1 - \sum_{i=0}^{n-1} \frac{|p_i - q_{i+1}|}{2} \\
&\quad - \sum_{i=1}^{n} \frac{|q_i - p_i|}{2} + \sum_{i=0}^{n} p_i - \sum_{i=0}^{n-1} \frac{|p_i - p_{i+1}|}{2} \Bigg).
\end{aligned} \tag{3.32}
$$

One can easily verify that (3.32) strictly improves with respect to (3.27).

It is worth noting that the BSTs constructed according to Theorem 14 are not too far from being optimal, and this is due to the lower bounds (3.29) on the cost of *any* binary search tree given in [41]. One can see that (3.32) and (3.29) are not too far apart. In fact, the difference between the upper bound (3.32) and the lower bound (3.29) is smaller than

$$
\begin{aligned}
&2 - \sigma_1 - \sigma_{2n+1} - \sum_{i=1}^{2n} \min(\sigma_i, \sigma_{i+1}) - 2 \sum_{i=1}^{n} q_i \\
&\quad - \sum_{i=0}^{n-1} \min(p_i, p_{i+1}) \\
&\quad - H(\sigma) \log_2(H(\sigma)) + (H(\sigma) + 1) \log_2(H(\sigma) + 1) \\
&\quad < 2 + \log_2(H(\sigma) + 1) + \log_2 e - \sigma_1 - \sigma_{2n+1} \\
&\qquad - \sum_{i=1}^{2n} \min(\sigma_i, \sigma_{i+1}) - 2 \sum_{i=1}^{n} q_i - \sum_{i=0}^{n-1} \min(p_i, p_{i+1}).
\end{aligned} \tag{3.33}
$$

Additionally, other results in [41] strengthen the lower bound (3.29) for a particular range of values of the entropy $H(\cdot)$ and thus can be used to reduce the gap between the upper bound and lower bound given in (3.33). Therefore, in situations where the optimal but onerous $O(n^2)$ time algorithm by Knuth [91] cannot be used, our $O(n)$ time algorithm represents a valid alternative.

We conclude this section with a final remark on Theorem 14. The primary contribution of Theorem 14 is the direct relationship it establishes between alphabetic codes and BSTs. This means that any future improvements to the bounds on the average length of alphabetic codes will immediately translate into better bounds for BSTs as well. This provides a powerful, forward-looking benefit: advancements in one area of data structures can be directly leveraged to enhance the theoretical understanding and practical performance of another.

4

ALPHABETIC AND PREFIX CODES WITH ASYMMETRIC SYMBOL COSTS

In this chapter, we remain in the field of binary VLCs, focusing on a specific variant of such codes, similar to the one discussed in Section 2.6.2. Specifically, we consider a constraint on the codewords, i.e., given a "budget", we associate with each codeword a cost, and only those codewords that stay within this budget are allowed. We present a novel, efficient polynomial-time algorithm for constructing optimal codes.

More specifically, Section 4.1 introduces the problem setting, and Section 4.2 the formal definition of what we term (α, β)-*constrained codes*. In Section 4.3, we deal with (α, β)-constrained alphabetic codes and provide an efficient algorithm for their construction. Furthermore, in Section 4.3.2, we establish a necessary and sufficient condition for the existence of the special case of $(0, 1)$-constrained alphabetic codes, building upon the inequalities discussed in literature (see Section 2.4). Successively, in Section 4.4, we address the more general and complex challenge of prefix codes. For the specific case of $(0, 1)$-constrained prefix codes, we derive a Kraft-like inequality for their existence. The majority of the results presented in the following chapter are taken from [26].

4.1 PROBLEM INTRODUCTION

We introduce the problem through a well-known and funny mathematical puzzle (see [94], p. 53). The puzzle goes as follows:

> *Suppose we wish to know which windows in a 36-story building are safe to drop eggs from, and which will cause the eggs to break on landing. We make a few assumptions:*
>
> - *An egg that survives a fall can be used again.*
> - *A broken egg must be discarded.*
> - *The effect of a fall is the same for all eggs.*

- *If an egg breaks when dropped, then it would break if dropped from a higher window.*
- *If an egg survives a fall, then it would survive a shorter fall.*
- *It is not ruled out that the first-floor windows break eggs, nor is it ruled out that the 36th-floor windows do not cause an egg to break.*

If only one egg is available and we wish to be sure of obtaining the right result, the experiment can be carried out in only one way. Drop the egg from the first-floor window; if it survives, drop it from the second-floor window. Continue upward until it breaks. In the worst case, this method might require 36 droppings. Suppose two eggs are available. What is the least number of egg droppings that is guaranteed to work in all cases?

Puzzles of this kind are nowadays commonly used in job interviews. However, the generalized version of such a problem has also entered the halls of academia through several papers[17, 18, 45, 104]. For instance, van Leeuwen *et al.* [104] employed a generalization of the problem as a central tool in their work on the problem of efficient leader election in asynchronous anonymous networks. We recall their formulation of the problem (quoted *verbatim*, apart from the names of the parameters):

> "Consider the case when the unknown number is a positive integer in the interval $[1, n]$, and the total amounts D of admissible overestimates is predetermined. The $\langle n, *, D\rangle$–game consists of determining the unknown number by asking as few questions as possible".

In [104] with the term *overestimate* it is intended that the search algorithm asks a question of the form: "Is the unknown number greater than $j \in [1, n]$?", *and* the answer received is NO. In fact, in this case, the value j is *greater* than the value of the unknown number one seeks, hence, j constitutes an overestimate of it. Analogously, one can also operate with questions of the form:"Is the unknown number smaller than $j \in [1, n]$?", where an overestimate corresponds instead to a YES answer. Therefore, the problems of limiting the number of YES or NO answers are equivalent. Nevertheless, in order to follow the work [104], we stick for the moment with the first kind of questions.

Moreover, one can see that the egg-dropping puzzle of [94] can be viewed as a special case, namely, it corresponds to the $\langle 36, *, 2\rangle$–game of [104]. Moreover, van Leeuwen *et al.* [104] pointed out that their $\langle n, *, D\rangle$–game corresponds to the bounded-searching problem with variable cost comparisons studied by Bentley and Brown in [17]. They also remarked that their results about $\langle n, *, D\rangle$–games lead to improved solutions for the algorithmic problems studied in [17]; this, in turn, offers an improvement to some of the scenarios where the solutions by Bentley and Brown had been applied: sensitivity analysis, broadcasting to points on a line, and linear recursion (see [17, 104] for more details).

Formally, the $\langle n, *, D\rangle$–search game problem consists in determining the best *worst-case* comparison-based search algorithm A that determines the unknown element in the search space $[n]$, under the constraints that A never performs a sequence of comparisons with more than D overestimates of the unknown number.

Using the well-known correspondence between comparison-based search procedures and binary alphabetic codes/trees (see Section 2.2.1), the $\langle n, *, D\rangle$–search game problem of [104] can be reformulated as the problem of constructing an alphabetic code for the search space $[n]$ that minimizes the maximum codeword length, i.e., the worst-case number of comparisons, subject to the constraint that each codeword contains at most D ones, where an 1 encodes an overestimate.

A related but more general setting was studied by Dolev *et al.* [45], motivated by the goal of quantifying the time–energy tradeoff in message transmissions between mobile hosts and mobile support stations. They formulated the problem as a guessing game in which the mobile support station tries to guess the message that the mobile host intends to transmit, while the host replies either by sending an acknowledgment or by simply remaining silent. This formulation naturally reduces to the problem of constructing optimal binary prefix codes in which each codeword does not contain more than D ones. Here, the ones encode the explicit replies of the host, whose number must be controlled to optimize the energy consumption. More specifically, they assumed that the search space $[n]$ is endowed with a probability distribution $p = (p_1, \ldots, p_n)$, where p_i is the probability that the unknown element one wants to determine is equal to $i \in [n]$. Thus, the objective

is to construct a binary prefix code of minimum average length for the search space $[n]$, under the additional constraint that no codeword contains more than D ones. One can see that such codes achieve the shortest expected transmission time, while obeying a required bound on the energy. Additionally, Dolev *et al.* [45] provided an $O(n^{2+D})$ time complexity Dynamic Programming algorithm to construct an optimal code under these constraints.

Furthermore, by recalling the strict relationship between codes and search procedure, we notice that *any* minimum average length prefix code, satisfying the restriction on the number of ones per codeword, also naturally corresponds to an average-case optimal search strategy in which:

- the queries take the form of arbitrary membership queries, i.e., "Does the unknown number belong to a subset $A \subseteq [n]$?"(e.g., $[5, 6]$),
- the number of queries that may receive a No is upper bounded by D.

Problems of this kind arise in different scenarios [117, 128], providing additional motivations to study the problem introduced in [45].

It is also worth noting the contrast with the setting of van Leeuwen *et al.* [104]: while their $\langle n, *, D\rangle$–game focuses on minimizing the worst-case number of comparisons, the formulation of Dolev *et al.* instead targets the average-case performance, optimizing the expected number of queries under a given probability distribution.

In Section 4.3, we introduce and solve a generalized search problem inspired by the $\langle n, *, D\rangle$–search games studied in [104]. Indeed, we consider the search space $[n]$ endowed with a probability distribution $p = \langle p_1, \dots, p_n\rangle$, and p_i is the probability that the unknown element is equal to $i \in [n]$. Moreover, in our scenario, we assume that both underestimates and overestimates have a cost expressed through non-negative integer values α and β. Thus, we consider comparison-based search algorithms that are capable of determining the unknown element in $[n]$, under the constraint that they never perform a sequence of comparison with an overall cost, $\alpha u + \beta o$, more than D, where u and o denote the number of

underestimates and overestimates in the sequence, respectively, and $\alpha, \beta \in \mathbb{N}$. We term such algorithms (α, β)-*constrained* search algorithms (or equivalently, (α, β)-*constrained* alphabetic codes). For this problem, we design an $O(n^2 D)$ time algorithm to find the *best average case* (α, β)-constrained comparison-based search algorithm A that is capable of determining the unknown element in $[n]$, under the constraints that A never performs a sequence of comparisons with a cost greater than D. Our main technical tool will be the celebrated Knuth-Yao Dynamic Programming–speedup through quadrangle–inequality [91, 137].

Successively, in Section 4.4, we focus on the challenging case of 1-constrained prefix codes, which are binary prefix codes where each codeword contains no more than D ones. In this scenario, our contribution is an extension of the classical Kraft inequality (see, e.g., Thm. 5.1 of [5]). Indeed, we establish a necessary and sufficient condition for the existence of a 1-constrained prefix code with a given set of codeword lengths $\{\ell_1, \ldots, \ell_n\}$.
We conclude this introduction by mentioning that our results can be seen as an extension of classical results, in the sense that the 1-constrained alphabetic case corresponds to the study of binary alphabetic codes [74] in which all codewords have a bounded number of 1's, and the 1-constrained non-alphabetic case corresponds to the extension of the Huffman encoding [77] to the constrained scenario just mentioned. As such, our work fits into the long line of research devoted to the construction of minimum average height binary trees satisfying given structural properties (see, e.g., [53, 57, 58, 83, 133] and references quoted therein).

4.2 PRELIMINARIES

This section formally defines the problems previously introduced in the language of binary codes and trees. We leverage the well-known equivalence between comparison-based search procedures, binary alphabetic codes, and binary search trees that we have already discussed in Section 2.2.1. The core of this translation maps an "overestimate" in a search algorithm to traversing a "right-pointing" edge (or, symmetrically a left-pointing edge) in the corresponding binary tree, which in turn corresponds to a '1' (or a '0') in a binary codeword.

Let $S = \{s_1, \ldots, s_n\}$ be a set of symbols, ordered according to a given total order relation $\prec$, that is, for which $s_1 \prec \cdots \prec s_n$ holds. Given a probability distribution $p = \langle p_1, \ldots, p_n \rangle$, where p_i is the probability of symbol s_i, for $i = 1, \ldots, n$, we recall that the average length of an alphabetic code $C = \{w(s) : s \in S\}$ for S is equal to

$$\mathbb{E}[C] = \sum_{i=1}^{n} p_i \ell_i,$$

where ℓ_i is the length of the codeword $w(s_i)$. Then, we can translate the $\langle n, *, D \rangle$–search games of [104] into the language of binary alphabetic codes as follows. For any binary string $w \in \{0,1\}^+$, let us denote with $|w|_1$ the number of 1's that appear in w. The problem of designing the *best average case* 1-constrained comparison-based search algorithm A to determine the unknown element in $[n]$, under the constraints that A never performs a sequence of comparisons with more than D overestimates of the unknown number, is equivalent to solving the following optimization problem:

$$\min_{C \text{ alphabetic}} \mathbb{E}[C] = \min_{C \text{ alphabetic}} \sum_{i=1}^{n} p_i \, \ell_i, \qquad (4.1)$$
$$\text{subj. to } |w(s_i)|_1 \leq D, \quad \forall i = 1, \ldots, n,$$

where ℓ_i is the length of $w(s_i)$. In other words, one seeks an alphabetic code of minimum average length in which *each* codeword does not contain more than D ones. For brevity, we will refer to such codes as *1-constrained alphabetic codes*. Furthermore, we observe, for the sake of completeness, that in this translation process, depending on the kind of question used to determine an overestimate in the initial search procedure (and thus whether it is encoded as a '0' or a '1'), it may be necessary to impose a different total order relation on the set of symbols when constructing the corresponding alphabetic code. For instance, for the type of question defined in [104], when an overestimate is encoded as a '1' bit, naturally, translate into alphabetic codes in which the order of the leaves is reversed with respect to the original ordering. Conversely, encoding an overestimate as a '0' bit corresponds naturally to alphabetic codes in which the order of the leaves preserves the original one. In any case, the final solutions are equivalent, but simply symmetric.

Finally, we can also define the more general (α,β)-constrained alphabetic codes as follows. For any binary string $w \in \{0,1\}^+$, let us denote with $|w|_{\alpha,\beta} = \alpha|w|_0 + \beta|w|_1$ the cost of 0's and 1's that appear in w, where $|w|_0$ and $|w|_1$ denote the number of zeros and ones in w respectively. The problem of designing the *best average case* (α,β)-constrained comparison-based search algorithm A, able to determine the unknown element in $[n]$ under the constraints that A never performs a sequence of comparisons with a total cost greater than D, is equivalent to solve the following optimization problem:

$$\min_{C \text{ alphabetic}} \mathbb{E}[C] = \min_{C \text{ alphabetic}} \sum_{i=1}^{n} p_i \ell_i, \qquad (4.2)$$
$$\text{subj. to } |w(s_i)|_{\alpha,\beta} \leq D, \quad \forall i = 1,\dots,n,$$

where ℓ_i is the length of $w(s_i)$. Notice that when both α and β are equal to zero, i.e. $\alpha = \beta = 0$, the problem reduces to the classical one without any constraint, while when they are both equal to 1, i.e. $\alpha = \beta = 1$, it yields the well-known length-limited alphabetic code problem (already discussed in Section 2.6.2).

Finally, if we relax the alphabetic constraint in Problem (4.1) to a prefix constraint, we arrive at the exact problem studied by Dolev *et al.* [45], which was discussed in the previous section.

4.3 (α,β)-CONSTRAINED ALPHABETIC CODES

In this section, we provide a straightforward Dynamic Programming algorithm to compute a solution to the optimization problem (4.2) and therefore also to (4.1). Subsequently, we employ the Knuth-Yao Dynamic Programming– speed up to get a better algorithm.

Let $C(i,j,w)$ denote the *minimum* average length of any (α,β)-constrained alphabetic code (tree) for the symbols $s_i,\dots,s_j$, with $1 \leq i \leq j \leq n$, in which each codeword has a cost of at most w. We say that $C(i,j,w)$ is the *cost of an optimal* (α,β)*-constrained alphabetic tree*. The solution of the optimization problem (4.2) corresponds to the problem of determining the value $C(1,n,D)$. Let $c(i,j) = p_i + \dots + p_j$ denote the sum of the probabilities associated with

the symbols $s_i, \dots, s_j$. We express the cost $C(i, j, w)$ as a function of the cost of its sub-trees. By considering all the possible ways of splitting the sequence of symbols $s_i, \dots, s_j$ into a left and a right sub-tree (with at least a node in each sub-tree), we get the following recurrence relation:

$$C(i,j,w) = \begin{cases} 0 & \text{if } i = j, \\ \infty & \text{if } i < j \text{ and} \\ & \quad w < \max\{\alpha, \beta\}, \\ \mathrm{c}(i,j) + \min_{i<k\leq j}\{C(i,k-1,w-\alpha) & \\ \qquad +C(k,j,w-\beta)\} & \text{otherwise.} \end{cases} \tag{4.3}$$

We recall that the right sub-tree is reached from a right-pointing edge of the root; therefore, all corresponding codewords one obtains begin with symbol 1, and this accounts for the value $w - \beta$ in the parameters of the right sub-tree cost. Similarly, all the codewords one obtains from the left sub-tree begin with 0, and this accounts for the value $w - \alpha$.

From the expression of (4.3), one can easily see that $C(1, n, D)$, can be computed in $O(n^3 D)$ time and $O(n^3 D)$ space. Algorithm 8 does that. Moreover, to compute also the optimal alphabetic code (and not only its cost), the algorithm keeps track of the indices on which the *optimal* partition occurs. This is done with the values $R(i, j, w)$'s, i.e., $R(i, j, w)$ corresponds to the biggest index k for which $C(i, j, w) = \mathrm{c}(i, j) + C(i, k-1, w-\alpha) + C(k, j, w-\beta)$.

4.3.1 *An improved algorithm*

The time complexity of Algorithm 8 can be improved by a factor of n by using a technique introduced by Knuth in [91] and generalized by Yao [137]. For such a purpose, let us recall some preliminary definitions and results. Let $R_{i,j,w}$ be the largest index where the minimum is achieved in the definition of $C(i, j, w)$. We recall that $R_{i,j,w}$ corresponds exactly to the value $R(i, j, w)$ computed in **Algorithm 8**. Moreover, let us use $C_k(i, j, w)$ to denote the

Algorithm 8: Dynamic Programming Approach

Input: Symbols $S = \{s_1, \dots, s_n\}$, $p = \langle p_1, \dots, p_n \rangle$ the associated probability distribution and an integer value D

```
1  for i ← 1 to n do
2      for j ← i + 1 to n do
3          for w ← 0 to max{α, β} − 1 do
4              C(i, j, w) = ∞
5  for w ← 0 to D do
6      for i ← 1 to n do
7          C(i, i, w) = 0
8          R(i, i, w) = i
9  for w ← max{α, β} to D do
10     for i ← 1 to n − 1 do
11         C(i, i + 1, w) = p_i + p_{i+1}
12         R(i, i + 1, w) = i
13 for w ← max{α, β} to D do
14     for s ← 2 to n − 1 do
15         for i ← 1 to n − s do
16             j = i + s ;            // Proceed by diagonals
17             min = ∞
18             mindex = −1
19             for k ← i + 1 to j do
20                 if C(i, k − 1, w − α) + C(k, j, w − β) ≤ min then
21                     min = C(i, k − 1, w − α) + C(k, j, w − β)
22                     mindex = k
23             C(i, j, w) = c(i, j) + min
24             R(i, j, w) = mindex
```

Output: $C(1, n, D)$ and R

cost of an optimal (α, β)-constrained alphabetic code under the constraint that the first splitting of symbols $s_i, \ldots, s_j$ is performed into $s_i, \ldots, s_{k-1}$ and $s_k, \ldots, s_j$, i.e, $C_k(i, j, w)$ satisfies

$$C_k(i, j, w) = \mathrm{c}(i, j) + C(i, k-1, w-\alpha) + C(k, j, w-\beta). \quad (4.4)$$

We recall that a two-argument function $f(\cdot, \cdot)$ satisfies the *Quadrangle Inequality* [137] if it holds that

$$f(i, j) + f(i', j') \leq f(i', j) + f(i, j'), \ \forall 1 \leq i \leq i' \leq j \leq j' \leq n.$$

A function $f(\cdot, \cdot)$ is said to be *monotone* if it holds that

$$f(i, j') \geq f(i', j), \quad \forall 1 \leq i \leq i' \leq j \leq j' \leq n.$$

One can easily see that $\mathrm{c}(i, j) = p_i + \cdots + p_j$ is both monotone and satisfies the Quadrangle Inequality (actually, with the equality sign).

Let us now recall the work of Borchers and Gupta [22] in which they showed that the Quadrangle Inequality holds for a large class of functions.

Let $F(i, j, r)$ be the minimum cost of solving a problem on inputs $i, i+1, \ldots, j$ with "resource" r (in our case, the resource will be the maximum cost that a codeword cannot exceed). Borchers and Gupta showed that the Quadrangle Inequality holds for all functions $F(i, j, r)$ defined as follows:

$$F(i, j, r) = \begin{cases} w(i), & \text{if } i = j \text{ and } r \geq 0, \\ \min_{i \leq k < j} \{ aF(i, k, f(r)) & \\ \qquad + bF(k+1, j, g(r)) + h(i, k, j) \}, & \text{if } f(r) \geq 0 \text{ and} \\ & \quad g(r) \geq 0, \\ \text{undefined}, & \text{otherwise}, \end{cases} \quad (4.5)$$

where $w(i)$ denotes the cost of $F(i, i, r)$, $f(\cdot)$ and $g(\cdot)$ are two functions such that $r \geq f(r)$ and $r \geq g(r)$, i.e., they are assuming that the resource does not increase, a and b are two positive integers,

and the function $h(i,k,j)$ is the cost of dividing the problem $F(i,j,r)$ at the splitting point k. Moreover, the function h must satisfy the following conditions that are similar to the quadrangle inequality: for $i \leq j \leq t < s \leq l$ and $i \leq s < l$, if $t \leq k$ it holds that

$$h(i,t,s) - h(j,t,s) + h(j,k,l) - h(i,k,l) \leq 0 \tag{4.6}$$

and

$$h(j,k,l) - h(i,k,l) \leq 0. \tag{4.7}$$

While if $k < t$, it holds that

$$h(j,t,l) - h(j,t,s) + h(i,k,s) - h(i,k,l) \leq 0 \tag{4.8}$$

and

$$h(i,k,s) - h(i,k,l) \leq 0. \tag{4.9}$$

To summarize, Borchers and Gupta [22] proved the following result.

Lemma 4 ([22]). *For each $r \geq 0$, the function F defined in (4.5) satisfies the Quadrangle Inequality, that is, $\forall 1 \leq i \leq i' \leq j \leq j' \leq n$ it holds that*

$$F(i,j,r) + F(i',j',r) \leq F(i,j',r) + F(i',j,r). \tag{4.10}$$

Using Lemma 4 we can prove the following result. The proof is provided in Appendix B.1.

Lemma 5. *For each $0 \leq w \leq D$, our function C defined in (4.3) satisfies the Quadrangle Inequality, that is, $\forall 1 \leq i \leq i' \leq j \leq j' \leq n$, it holds that*

$$C(i,j,w) + C(i',j',w) \leq C(i,j',w) + C(i',j,w). \tag{4.11}$$

By using Lemma 5 we can show the following result, which is a generalization of the crucial property called "monotonicity of the roots" in [91]. This result is the key step for improving the algorithm.

Theorem 15. *For each $0 \leq w \leq D$ it holds that*

$$\forall 1 \leq i \leq j \leq n \quad R_{i,j,w} \leq R_{i,j+1,w} \leq R_{i+1,j+1,w}. \tag{4.12}$$

Proof. Let $\max\{\alpha,\beta\} \le w \le D$ (for $w < \max\{\alpha,\beta\}$ the theorem is trivially true). The claim is trivially true also when $i = j$. Let us assume that $i < j$. We will show that $R_{i,j,w} \le R_{i,j+1,w}$, the argument for $R_{i,j+1,w} \le R_{i+1,j+1,w}$ follows by symmetry. Since $R_{i,j,w}$ is the largest index where the minimum is achieved in the definition of $C(i,j,w)$, it is sufficient to show that $\forall i < k \le k' \le j$ it holds that

$$C_{k'}(i,j,w) \le C_k(i,j,w) \implies C_{k'}(i,j+1,w) \le C_k(i,j+1,w). \tag{4.13}$$

This is sufficient because the implication (4.13) ensures that the set of indices achieving the minimum does not shift to the left as j increases. Specifically, if k' is the largest index that minimizes $C(i,j,w)$, the implication $C_{k'}(i,j+1,w) \le C_k(i,j+1,w)$ for all $k < k'$ guarantees that the new optimal index $R_{i,j+1,w}$ cannot be smaller than k'.

We will prove that (4.13) holds by showing the following inequality, for all $i < k \le k' \le j$,

$$C_k(i,j,w) - C_{k'}(i,j,w) \le C_k(i,j+1,w) - C_{k'}(i,j+1,w). \tag{4.14}$$

In fact, let us assume that (4.14) is true. Then, whenever $C_{k'}(i,j,w) \le C_k(i,j,w)$ of (4.13) holds, from (4.14) we get that

$$0 \le C_k(i,j,w) - C_{k'}(i,j,w) \le C_k(i,j+1,w) - C_{k'}(i,j+1,w). \tag{4.15}$$

From (4.15), since $0 \le C_k(i,j+1,w) - C_{k'}(i,j+1,w)$, we obtain that $C_{k'}(i,j+1,w) \le C_k(i,j+1,w)$, that is, (4.13) holds too. To prove (4.14) let us rewrite it as follows

$$C_k(i,j,w) + C_{k'}(i,j+1,w) \le C_k(i,j+1,w) + C_{k'}(i,j,w). \tag{4.16}$$

By expanding all four terms of (4.16) we get (cfr., see (4.4))

$$\begin{aligned} &\mathrm{c}(i,j) + C(i,k-1,w-\alpha) + C(k,j,w-\beta) \\ &\qquad + \mathrm{c}(i,j+1) + C(i,k'-1,w-\alpha) + C(k',j+1,w-\beta) \\ &\quad \le \mathrm{c}(i,j+1) + C(i,k-1,w-\alpha) + C(k,j+1,w-\beta) \\ &\qquad + \mathrm{c}(i,j) + C(i,k'-1,w-\alpha) + C(k',j,w-\beta). \end{aligned}$$

Grouping the terms of the above inequality, we obtain

$$C(k,j,w-\beta) + C(k',j+1,w-\beta) \le C(k,j+1,w-\beta) + C(k',j,w-\beta).$$

(4.17)

This is the quadrangle inequality for the function C with parameter $w - \beta$, with $k \leq k' \leq j < j + 1$, and from Lemma 5 we know that it holds true. Thus, this concludes the proof. □

Now, according to Theorem 15, the loop in **Algorithm 8** (line 19) can be optimized. Instead of searching from $i+1$ to j, it is sufficient to search only from $R(i, j-1, w)$ to $R(i+1, j, w)$. As argued by Knuth [91, p. 19], with these modifications, the number of cases to examine is only $R(i+1, j, w) - R(i, j-1, w) + 1$. Summing this expression for a fixed difference $j - i$ gives a telescoping series, showing that the total amount of work of each iteration of the loop at line 13 is proportional to $O(n^2)$, at worst. This decreases the total time complexity of the algorithm from $O(n^3 D)$ to $O(n^2 D)$.

We conclude this section by showing in the following remark that the problem remains polynomial in n even if the parameters α, β and D are not.

Remark 3. *We observe that no assumption is required regarding the magnitude of α, β and D with respect to n to guarantee that the problem can be solved in polynomial time in n. This is because the algorithm can be implemented to maintain a time complexity that is polynomial in n, regardless of the size of the other parameters. Indeed, the recurrence in (4.3) implies that the third parameter w assumes only values of the form $D - (x\alpha + y\beta)$, with $x, y \in \mathbb{N}$. Moreover, since the depth of the constructed tree is bounded by $n-1$, we have the constraint $x+y \leq n-1$. Consequently, the set of effectively reachable values for w is restricted to*

$$\{D - (x\alpha + y\beta) \mid x, y \in \mathbb{N}, x + y \leq n - 1\} \cap [0, D].$$

The cardinality of this set is bounded by the number of integer points (x, y) satisfying the length constraint, which is at most $O(n^2)$. Therefore, regardless of the size of D, the algorithm can be implemented to run in polynomial time in n.

4.3.2 *A condition for the existence of* 1*-constrained alphabetic codes*

In this section, we briefly focus on the special case of $(0, 1)$-constrained alphabetic codes, also referred to as 1-constrained alphabetic codes.

We provide a necessary and sufficient condition for the existence of these codes. Specifically, we naturally extend the classical condition introduced by Nakatsu [115] (see Section 2.4 for the classical setting). We begin with some preliminary definitions.

Definition 9. *Let x be a binary fraction and let $i \geq 1$ be an integer.*
Define the function `trunc` *as follows*

$$\texttt{trunc}(i, x) = \frac{\lfloor 2^i x \rfloor}{2^i}. \tag{4.18}$$

In other words, $\texttt{trunc}(i, x)$ *is the fraction obtained by considering only the first i bits in the binary representation of x.*

Definition 10. *Let x be a binary fraction and let $i \geq 1$ and $D \geq 1$ be integers.*
Define the function ϕ_D as follows:
$\phi_D(i, x)$ returns x if the first i bits in the binary expansion of x contain at most D ones.
Otherwise, it returns y, where y is the smallest value greater than x whose binary expansion has at most D ones in its first i bits.

We can now define our condition.

Definition 11. *Let $L = \langle \ell_1, \ldots, \ell_n \rangle$ be a list of positive integers, and let $\alpha_i = \min(\ell_{i-1}, \ell_i)$, for $i = 2, \ldots, n$. The recursive function* $\texttt{sum}_D$ *is defined as*

$$\texttt{sum}_D(L, i) = \begin{cases} \phi_D(\ell_i, \texttt{trunc}(\alpha_i, \texttt{sum}_D(L, i-1)) + 2^{-\alpha_i}) & \text{if } i \geq 2, \\ 0 & \text{if } i = 1. \end{cases} \tag{4.19}$$

Following the same approach of Nakatsu, we can now present our necessary and sufficient condition in the following theorem, whose proof is deferred to Appendix B.2.

Theorem 16. *Let $L = \langle \ell_1, \ldots, \ell_n \rangle$ be a list of integers, associated with the ordered symbols $s_1 \prec \cdots \prec s_n$, and let D be a positive integer. There exists a 1-constrained alphabetic code for which each codeword contains at most D ones and for which the codeword assigned to symbol s_i has length ℓ_i, for each $i = 1, \ldots, n$, if and only if $\texttt{sum}_D(L, n) < 1$.*

Intuitively, our condition extends the same idea underlying Nakatsu's classical inequality [115]. As in the classical setting, we interpret the unit interval $[0, 1]$ as our total coding space for constructing an alphabetic code. Therefore, each codeword is associated with a value in $[0, 1]$. Under this point of view, $\mathtt{sum}_D(L, i)$ represents the smallest value greater than $\mathtt{sum}_D(L, i-1)$ that can be used to build a codeword of length ℓ_i while preserving both the prefix property and the constraint on the maximum number of ones.

4.4 1-constrained prefix codes

Let us now focus on the more complex scenario of (α, β)-constrained prefix codes. Specifically, let us consider the special case of 1-constrained binary prefix codes. As discussed in Section 4.1, the best-known algorithm for constructing such codes has a time complexity of $O(n^{2+D})$ [45]. This high complexity arises because, without a predefined order of the symbols (like in the alphabetic scenario), one should take into account all possible permutations. Furthermore, the common technique of reducing the prefix coding problem to the simpler alphabetic one-applicable when the probability distribution p of the symbols is ordered (in a non-increasing or non-decreasing fashion), as in the classical setting [139] or in the length-limited case [53, 60] (discussed in Section 2.6.2)-does *not* carry over to general (α, β)-constrained prefix codes with $\alpha \neq \beta$. This reduction works only in the special symmetric case $\alpha = \beta = 1$, which coincides with the length-limited model. To illustrate this point, consider the following instance of the 1-constrained binary prefix coding problem. Let $n = 11, D = 2$ and

$$p_1 = p_2 = \cdots = p_8 = \frac{1}{16}, \quad p_9 = p_{10} = \frac{1}{8}, \quad p_{11} = \frac{1}{4}.$$

An optimal 1-constrained prefix code for this instance is shown in Figure 4.1; its average length is equal to $H(p) = 3.25$.

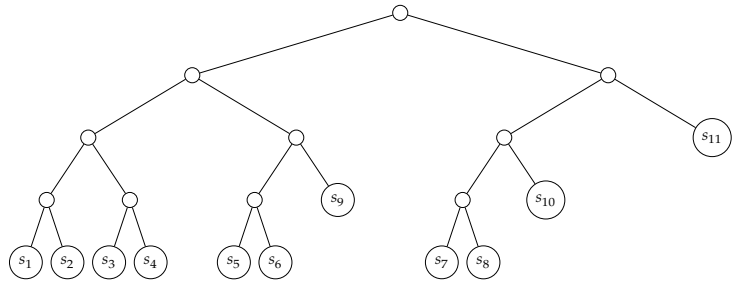


Figure 4.1: An optimal 1-constrained binary prefix code for $p = (1/16, \dots, 1/16, 1/8, 1/8, 1/4)$.

Consider now the alphabetic version of the same instance. If we impose the alphabetic order on the set of symbols $S = \{s_1, \dots, s_n\}$- whether the probabilities are ordered in a non-decreasing fashion

$$\left\langle \frac{1}{16}, \cdots, \frac{1}{16}, \frac{1}{8}, \frac{1}{8}, \frac{1}{4} \right\rangle$$

or in a non-increasing fashion

$$\left\langle \frac{1}{4}, \frac{1}{8}, \frac{1}{8}, \frac{1}{16}, \cdots, \frac{1}{16} \right\rangle$$

the optimal 1-constrained binary alphabetic codes obtained in either case have an average length that is strictly greater than 3.25. Moreover, by applying the condition for the existence of 1-constrained alphabetic codes established in Section 4.3.2, it can be shown that there does not exist a 1-constrained alphabetic code for the symbols $s_1 \prec \cdots \prec s_n$ with codeword lengths equal to the ones used in Figure 4.1.

Given that in this context the prefix coding problem cannot be reduced to the alphabetic problem, in the following we proceed to establish a Kraft-like condition, i.e., a necessary and sufficient criterion for the existence of 1-constrained binary prefix codes. Subsequent sections present further results and considerations for this class of codes.

4.4.1 *A Kraft-like condition for the existence of 1-constrained binary prefix codes*

In this section, we derive a necessary and sufficient condition for the existence of an 1-constrained prefix code/tree, given a

multiset $\{\ell_1, \dots, \ell_n\}$ of positive integers, and a positive integer D. It represents a natural extension of the classical Kraft inequality to the 1-constrained setting. We point out that there are many extensions of the Kraft inequality to situations where the codes have to satisfy additional constraints (besides that of being prefix), see [14, 32, 44, 59, 82, 86, 123, 131]. However, none of the above quoted papers deal with the extension we derive in the following.

Let us start by presenting the sufficient direction of our Kraft-like condition, whose proof is deferred to Appendix B.3.

Lemma 6. *Let $\{\ell_1, \dots, \ell_n\}$ be a multiset of positive integers, D a positive integer and $L = \max_i \ell_i$. Let N_j count the number of integers in $\{\ell_1, \dots, \ell_n\}$ that are equal to j, for $j = 1, \dots, L$. Then, there exists a 1-constrained binary prefix code with codeword lengths $\ell_1, \dots, \ell_n$, such that each codeword does not contain more than D ones,* if *the following inequalities hold:*

$$N_j \leq \sum_{i=0}^{D-1} \binom{j}{i} + \binom{j-1}{D-1} - M_{j+1}, \quad \forall j = 1, \dots, L, \quad \textit{where} \tag{4.20}$$

$$M_{j+1} = \begin{cases} 0 & \textit{if } j \geq L, \\ \left\lceil \dfrac{N_{j+1} + M_{j+2}}{2} \right\rceil & \textit{if } 1 \leq j < L. \end{cases} \tag{4.21}$$

To build intuition for the construction method presented in the proof of the previous lemma, let us walk through a concrete example.

Example 3. *Let $\langle 4, 2, 3, 4, 3, 3, 3 \rangle$ be the list of lengths associated with the symbols $s_1, \dots, s_7$ and $D = 2$. By the definitions, we get the values: $N_4 = 2, N_3 = 4, N_2 = 1$ and $N_1 = 0$. Let us first of all see that for such values the inequalities (4.20) hold. Indeed:*

$$N_4 = 2 \leq \sum_{i=0}^{1} \binom{4}{i} + \binom{3}{1} = 8, \tag{4.22}$$

$$N_3 = 4 \leq \sum_{i=0}^{1} \binom{3}{i} + \binom{2}{1} - \left\lceil \frac{2}{2} \right\rceil = 5, \tag{4.23}$$

$$N_2 = 1 \leq \sum_{i=0}^{1} \binom{2}{i} + \binom{1}{1} - \left\lceil \frac{4+1}{2} \right\rceil = 1, \tag{4.24}$$

$$N_1 = 0 \leq \sum_{i=0}^{1} \binom{1}{i} + \binom{0}{1} - \left\lceil \frac{1+3}{2} \right\rceil = 0. \tag{4.25}$$

Since the inequalities hold, we can construct the 1-constrained prefix code with the required list of lengths $\langle 4, 2, 3, 4, 3, 3, 3 \rangle$. *For such a purpose, let us start from the complete binary tree T of depth* 4 *in which, for any node v, the path from the root to the node v does not contain more than* $D = 2$ *edges with label* 1 (*Figure 4.2*).

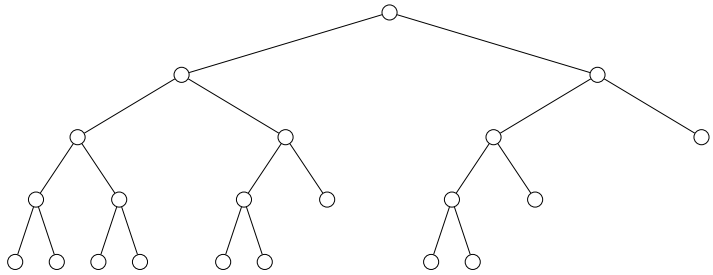


Figure 4.2: The complete binary tree T of depth 4

Since $N_4 = 2$ *we need two nodes at the level* 4 *of T to construct the codewords for the symbols* s_1 *and* s_4, *respectively. As described in Lemma 6, we take the left-most ones, and we get the tree in Figure 4.3. Note that it is the inequality* (*4.22*) *that assures us that we can choose such nodes at the level* 4.

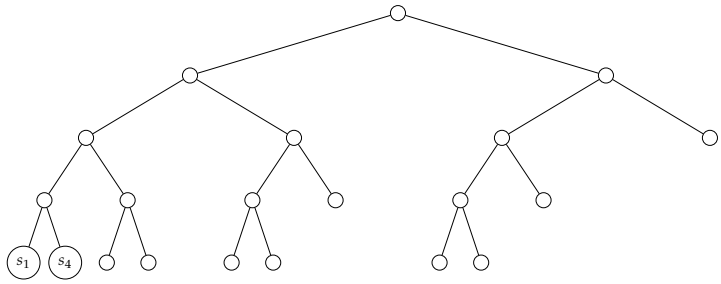


Figure 4.3: The tree T after choosing the nodes for the codewords of length 4

Now, since $N_3 = 4$, *we need to take four nodes at level* 3 *for the construction of the codewords for the symbols* s_3, s_5, s_6 *and* s_7. *However,*

we have to ignore the nodes that are prefixes of the chosen codewords of length 4*. Note that we can do this since the values* N_i*'s satisfy the required inequalities.*

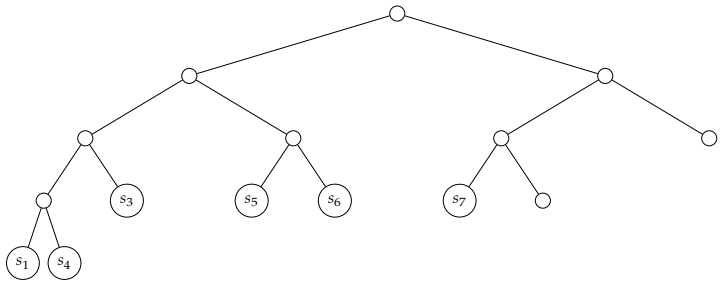


Figure 4.4: The tree after choosing the nodes for the codewords of length 4 and 3

Since $N_2 = 1$*, there is only one codeword of length* 2 *that we need to choose. Therefore, we take the left-most node at level* 2 *that is still "available", as shown in Figure 4.5.*

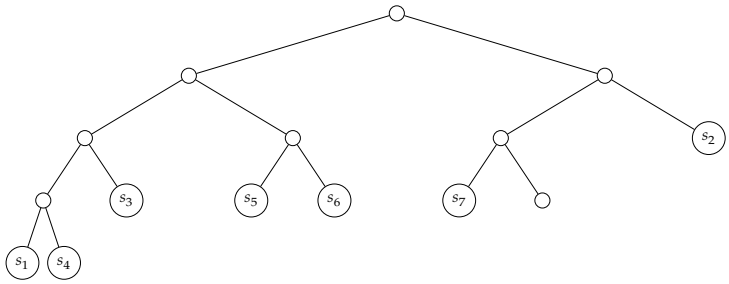


Figure 4.5: The tree after choosing the nodes for the codewords of length 4, 3 and 2

Finally, since $N_1 = 0$ *we do not need to take any node at level* 1*. So, the tree that we get in the end is shown in Figure 4.6.*

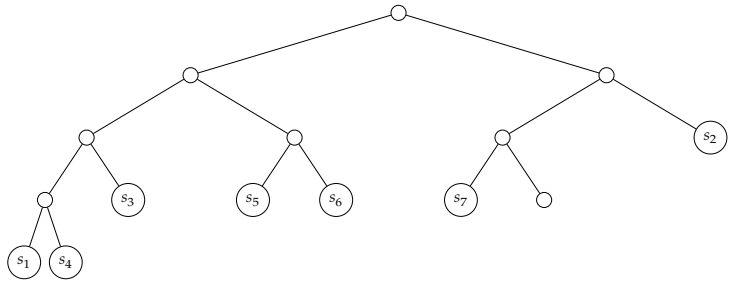


Figure 4.6: The final tree after the choice of all codewords

The 1-constrained prefix code $w : \{s_1, \dots, s_7\} \to \{0,1\}^+$ *that we get from the tree in Figure 4.6 is the following:* $w(s_1) = 0000$, $w(s_2) = 11$, $w(s_3) = 001$, $w(s_4) = 0001$, $w(s_5) = 010$, $w(s_6) = 011$ *and* $w(s_7) = 100$.

□

The condition presented in Lemma 6 is *only* a sufficient condition for the existence of a 1-constrained binary prefix code with at most D ones in each codeword. This is because the values M_{j+1} (4.21) do not necessarily represent the *minimal* number of nodes that must be removed at each level j of the complete binary tree to ensure the prefix property is maintained.

However, if we restrict our analysis to codeword lengths of 1-constrained prefix codes that correspond to *full* binary trees (i.e., for which it holds that $\sum_{i=1}^{n} 2^{-\ell_i} = 1$ or, equivalently, any internal node has *exactly* two children [5]), then the condition of Lemma 6 becomes necessary as well. The following lemma, whose proof is deferred to Appendix B.4, formalizes this result.

Lemma 7. *Let* $\ell_1, \dots, \ell_n$ *be the codeword lengths of a 1-constrained prefix code* C *in which each codeword does not contain more than* D *ones and whose tree representation corresponds to a full binary tree, i.e., for which* $\sum_{i=1}^{n} 2^{-\ell_i} = 1$. *Then the following inequalities hold:*

$$N_j \leq \sum_{i=0}^{D-1} \binom{j}{i} + \binom{j-1}{D-1} - M_{j+1}, \quad \forall j = 1, \dots, L, \tag{4.26}$$

where M_j *and* N_j *are defined as in Lemma 6 and* $L = \max_i \ell_i$.

4.4.2 *Additional remarks*

Let us maintain our focus on the simpler class of 1-constrained codes. The constraint that *at most D* ones must appear in each codeword might prevent the construction of the (otherwise unrestricted) optimal code. Thus, the following natural question arises: under what conditions does this constraint actually limit the choice of an optimal code? That is, for what value of D is the optimal constrained code identical to the optimal unrestricted code?

For the case of alphabetic trees, establishing the smallest value of D that does not prevent the construction of the optimal code is straightforward. Indeed, it is easy to identify probability distributions (for example, a decreasing dyadic distribution, i.e., $p = \langle 1/2, 1/4, 1/8, ... \rangle$) for which *any* optimal alphabetic code must have a codeword with $n - 1$ ones. Hence, any value of D strictly less than $n - 1$ is a constraint that *prevents* the construction of a code with the same cost of an *optimal unrestricted* one.

For the case of prefix codes, establishing a similar result is not immediate. In the next theorem (whose proof is deferred to Appendix B.5), we prove that the smallest value of D that does not prevent the construction of a tree whose cost is as the cost of an optimal unrestricted tree is $D = \lfloor \log_2 n \rfloor$.

Theorem 17. *Let $S = \{s_1, \dots, s_n\}$ be a set of symbols and $p = \langle p_1, \dots, p_n \rangle$ a probability distribution on S with $p_1 \leq \cdots \leq p_n$. There exists an optimal 1-constrained prefix tree/code T^* for S, in which each codeword does not contain more than $\log_2 n$ ones.*

In Theorem 17, we have shown that for $D = \log_2 n$, the restriction on the maximum number of ones per codeword does not represent a restriction anymore. This naturally leads to the question of whether, for this specific value of D, our proposed Kraft-like condition is equivalent to the classical Kraft inequality. In the following theorem, whose proof is deferred to Appendix B.6, we demonstrate that our Kraft-like condition is indeed equivalent to the Kraft Inequality when $D = \lceil \log_2 n \rceil$.

Theorem 18. *For $D = \lceil \log_2 n \rceil$, our Kraft-like condition (4.20) is equivalent to the classical Kraft-inequality.*

4.5 OPEN PROBLEMS

In this chapter, we studied the problem of constructing minimum average-length VLCs derived from search problems with *asymmetric* test outcome costs. We formalized this by introducing (α, β)-constrained codes, where distinct outcomes of a binary test incur different costs, α and β, respectively. The central challenge was to devise search strategies—or equivalently, codes—that minimize expected query complexity while ensuring the cumulative cost of any single search path remains within a predefined budget, D.

Our primary contribution was the development of an efficient, polynomial-time dynamic programming algorithm for constructing optimal (α, β)-constrained alphabetic codes. Successively, for the specific case of $(0, 1)$-constrained alphabetic codes, we also established a necessary and sufficient condition for their existence. Then, we considered the more complex prefix code variant, specifically focusing on 1-constrained prefix codes. For this class, we established a novel Kraft-like inequality that characterizes their existence.

However, the question of whether or not, like in the alphabetic setting, a polynomial time algorithm exists for constructing (α, β)-constrained prefix codes, or even 1-constrained prefix codes, remains open. Furthermore, the problem of whether our Kraft-like condition for 1-constrained binary prefix codes can be extended to general (α, β)-constrained binary prefix codes also remains open. This generalization is substantially more complex, as the combinatorial analysis that underpins our Kraft-like inequality in Section 4.4.1 does not readily extend to arbitrary, non-negative costs for both symbols.

However, for the special case of $(0, \beta)$-constrained prefix codes, we can see that our condition is still applicable. This is achieved by showing that the problem of constructing $(0, \beta)$-constrained prefix codes can be reduced to that of constructing 1-constrained prefix codes, for which we have already established the existence condition. Let us briefly see how this is possible. In the $(0, \beta)$-constrained prefix code problem, one seeks to construct a prefix code in which for each codeword w holds that

$$|w|_\beta = \beta|w|_1 \leq D.$$

In other words, for each codeword w it holds that

$$|w|_1 \leq D/\beta. \tag{4.27}$$

Since $|w|_1$ is an integer, (4.27) is equivalent to

$$|w|_1 \leq \lfloor D/\beta \rfloor.$$

But, this final expression is precisely the definition of a 1-constrained prefix code with a budget $D' = \lfloor D/\beta \rfloor$. Therefore, establishing the existence of $(0, \beta)$-constrained prefix codes with budget D is equivalent to establishing the existence of 1-constrained prefix codes with budget $D' = \lfloor D/\beta \rfloor$, allowing our established Kraft-like condition to be directly applied. The exact same reasoning can be applied to the condition for 1-constrained alphabetic codes too. Therefore, our necessary and sufficient condition for the existence of 1-constrained alphabetic codes can be extended to $(0, \beta)$-constrained alphabetic codes as well.

Finally, another interesting open question is to derive good upper and lower bounds on the average length of such (α, β)-constrained alphabetic and prefix codes, in terms of an appropriate, easy-to-compute function of the probability distribution $p = (p_1, \ldots, p_n)$ and the parameter D, as done in terms of the Shannon entropy of p in the unrestricted classical case (see Section 2.5).

5

PREFIX CODES WITH A SPACE DELIMITER

The preceding chapters examined VLCs, focusing mainly on their close relation with binary search procedures (see Chapter 2 for further information). In this chapter, we turn our attention to a more general and distinct setting in which the aim is to construct "efficient" prefix codes. Specifically, we consider the problem of constructing prefix codes in which a designated symbol, a *"space"*, can only appear at the end of codewords. We designate such codes with the name of *prefix codes ending with a space*. This model formalizes the intuitive notion of using a special character exclusively as a word delimiter, as happens in many natural languages.

The chapter is structured as follows: in Section 5.1 we provide further context on the background and the settings in which such codes are applicable; in Section 5.2 we investigate the relationship between one-to-one codes and prefix codes ending with a space, and present a linear-time algorithm to construct *almost*-optimal prefix codes ending with a space, meaning that their average length differs from the minimum possible by at most one; then in Sections 5.3 and 5.4 we derive upper and lower bounds to the average length of optimal prefix codes ending with a space, again leveraging their relationship with one-to-one codes. Finally, we will also highlight an interesting link between our prefix codes ending with a space and particular classes of search algorithms. The main results presented in the following are taken from [27].

5.1 INTRODUCTION AND PRELIMINARIES

Many modern natural written languages typically ensure clarity by separating words with a designated special character, i.e., a *space*, a simple yet effective parsing mechanism [39] (See [134] for a few exceptions to this rule). In contrast, classical Information Theory achieves unique parsability through strict combinatorial rules imposed on the codeword set, e.g., the prefix property, unique decipherability, etc. [35]. The efficiency of such codes, typically

measured by the average number of code symbols per source symbol, as is well-known, is fundamentally lower-bounded by the Shannon entropy of the information source.

On the other hand, what happens if unique parsability is relaxed? Such a relaxation gives rise to *one-to-one codes* (see Definition 1), which assign a unique codeword to each source symbol but do not guarantee that concatenated messages are uniquely decodable. This freedom allows their average length to fall below the Shannon entropy barrier, albeit not substantially (see, e.g., [7, 21]).

This interesting tradeoff between distinct coding paradigms motivates a deeper investigation into alternative models that bridge the gap between them. The first to formally study problems in such a scenario was Jaynes. In [80] he investigated codes in which a designated character of the code alphabet is *exclusively* used as a word delimiter. More precisely, Jaynes studied the possible decrease of the noiseless channel capacity (see [125], p. 8) associated with any code that uses a designated symbol as an end-of-codeword mark, as compared with the noiseless channel capacity of an unconstrained code. Quite interestingly, Jaynes proved that the decrease of the noiseless channel capacity of codes with an end-of-codeword mark becomes negligible as the maximum codeword length increases.

Following Jaynes's idea, we investigate the related problem of constructing prefix codes where a specific symbol (referred to as a "space") can only be positioned at the end of codewords. We refer to this class of prefix codes as *prefix codes ending with a space*. Our analysis reveals a fundamental relationship between this new class of codes and the well-established domain of *one-to-one codes*. By leveraging this connection, we achieve two main results:

- First, we design a linear-time algorithm that constructs almost-optimal codes with this characteristic, guaranteeing that the average length of the constructed codes is at most one unit longer than the *shortest possible* average length of any prefix code in which the space can appear only at the end of codewords.

- We also provide upper and lower bounds on the average length of optimal prefix codes ending with a space, expressed

in terms of the source entropy and the cardinality of the code alphabet.

5.2 RELATION BETWEEN ONE-TO-ONE CODES AND PREFIX CODES ENDING WITH A SPACE

This section establishes the core framework for our study by detailing the relationship between the well-known class of one-to-one codes and that of prefix codes ending with a space. We will demonstrate that this connection is not merely theoretical; it provides a direct, constructive method for building almost-optimal prefix codes ending with a space from optimal one-to-one codes. To ground this discussion, we begin with the formal definitions of these classes of codes.

Let $S = \{s_1, \dots, s_n\}$ be the set of source symbols, $p = (p_1, \dots, p_n)$ be a probability distribution on the set S, and $\Sigma_k = \{0, \dots, k-1\}$ be the code alphabet, with $k \geq 2$.

We denote by Σ_k^+ the set of all non-empty sequences on the code alphabet Σ_k, and by $\Sigma_k^+ \sqcup$ the set of all non-empty k-ary sequences that end with the special symbol $\sqcup$, i.e., the *space* symbol.

A $(k+1)$-ary *prefix code* ending with a space is a one-to-one mapping

$$w : S \longmapsto \Sigma_k^+ \cup \Sigma_k^+ \sqcup$$

that satisfies the prefix condition: for any two distinct source symbols $s, s' \in S$, $s \neq s'$, the codeword $w(s)$ is not a prefix of the codeword $w(s')$. We note that the space symbol $\sqcup$ may appear only as the last symbol of a codeword, and its presence is optional. We denote by $C = \{w(s) : s \in S\}$ the set of all its codewords.

A k-ary *one-to-one code* (see [7, 21, 34, 95, 97, 127] and the references therein quoted) is a one-to-one mapping $f : S \longmapsto \Sigma_k^+$ from S to the set of all non-empty sequences over the alphabet Σ_k, $k \geq 2$. This class of codes does not enforce the prefix property, focusing only on assigning a unique string to each symbol. Similarly, we denote by $D = \{f(s) : s \in S\}$ the set of all its codewords.

For the sake of notation, as done in the previous chapters, we will often refer to a code by its set of codewords, using C for prefix codes ending with a space and D for one-to-one codes.

We recall that given an arbitrary code C for a set of source symbols $S = \{s_1, \ldots, s_n\}$, with probabilities $p = (p_1, \ldots, p_n)$, its average length $\mathbb{E}[C]$ is defined as

$$\mathbb{E}[C] = \sum_{i=1}^{n} p_i \ell_i,$$

where ℓ_i is the length of the codeword assigned to the source symbol s_i.

In the following, without loss of generality, we assume that the probability distribution $p = (p_1, \ldots, p_n)$ of the source symbols is ordered in a non-increasing fashion, that is $p_1 \geq \ldots \geq p_n$. Under this assumption, one can see that the *best* one-to-one code, i.e., the one with the minimum average length, proceeds by assigning the shortest available codewords to the most probable symbols. For example, in the binary case, it assigns the shortest codeword 0 to the highest probability source symbol s_1, the next shortest codeword 1 to the source symbol s_2, the codeword 00 to s_3, the codeword 01 to s_4, and so on. An efficient enumeration of this sequence of codewords has recently been provided in [23].

In the following, however, for our purposes we utilize an equivalent approach for constructing an optimal one-to-one code, which proceeds as follows. Let us consider the first n non-empty k-ary strings according to the *radix* order [93] (that is, the k-ary strings are ordered by length and, for equal lengths, ordered according to the lexicographic order). It assigns the strings to the symbols $s_1, \ldots, s_n$ in S by increasing the string length and, for equal lengths, by inverse order according to the lexicographic order. For example, in the binary case, it assigns the codeword 1 to the highest probability source symbol s_1, the codeword 0 to the source symbol s_2, the codeword 11 to s_3, the codeword 10 to s_4, and so on.

Therefore, in the general case of a k-ary code alphabet, one can see that an optimal one-to-one code of minimal average length assigns a codeword of length ℓ_i to the i-th symbol $s_i \in S$, where ℓ_i is given by:

$$\ell_i = \lfloor \log_k((k-1)\, i + 1) \rfloor. \tag{5.1}$$

Moreover, these codewords can be visualized as nodes of a k-ary tree of maximum depth $h = \lceil \log_k(n - \lceil n/k \rceil) \rceil$, where, for each

node v, the k-ary string (codeword) associated with v is obtained by concatenating all the labels in the path from the root of the tree to v.

A direct consequence of this encoding scheme is that concatenated sequences of codewords are *not* uniquely parsable. Let us see how one can recover unique parsability by appending a space $\sqcup$ to a judiciously chosen subset of codewords.

To gain insight, let us consider the following example. Let $S = \{s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}\}$ be the set of source symbols, and and let us assume that the code alphabet is $\{0, 1\}$. Under the standing hypothesis that $p_1 \geq \ldots \geq p_{10}$, one has that the best prefix code C, with mapping w, one can obtain by the procedure of appending a space $\sqcup$ to codewords of an optimal one-to-one code for S is the following:

$$
\begin{aligned}
C(s_1) &= 1\sqcup \\
C(s_2) &= 0\sqcup \\
C(s_3) &= 11 \\
C(s_4) &= 10 \\
C(s_5) &= 01\sqcup \\
C(s_6) &= 00\sqcup \\
C(s_7) &= 011 \\
C(s_8) &= 010 \\
C(s_9) &= 001 \\
C(s_{10}) &= 000.
\end{aligned}
$$

We observe that we started from the codewords of the optimal one-to-one code constructed according to the *second* procedure previously described. Moreover, the space symbol is appended precisely to only those codewords from the original one-to-one code that are a prefix of another; in this case, the codewords associated with symbols s_1, s_2, s_5, and s_6. While, the remaining codewords associated with symbols s_3, s_4, s_7, s_8, s_9, and s_{10} do not necessitate the space character $\sqcup$. Indeed, the codeword set

$$C = \{1\sqcup, 0\sqcup, 11, 10, 01\sqcup, 00\sqcup, 011, 010, 001, 000\}$$

satisfies the prefix condition (i.e., no codeword is a prefix of any other); therefore, it guarantees the unique parsability.

The idea of the above example can be generalized, as shown in the following lemma.

Lemma 8. *Let $S = \{s_1, \dots, s_n\}$ be the set of source symbols and $p = (p_1, \dots, p_n)$, with $p_1 \geq \dots \geq p_n > 0$, be a probability distribution on S. Let Σ_k be the $k \geq 2$-ary code alphabet. We can construct a prefix code C ending with a space in linear time, $O(n)$, such that its average length $\mathbb{E}[C]$ satisfies*

$$\mathbb{E}[C] = \mathbb{E}[D] + \sum_{i=1}^{\frac{k^{h-1}-1}{k-1}-1} p_i + \sum_{i=\frac{k^h+k^{h-1}-2}{k-1}-\lceil n/k \rceil}^{\frac{k^h-1}{k-1}-1} p_i \tag{5.2}$$

$$\leq \mathbb{E}[D] + \sum_{i=1}^{\lceil n/k \rceil - 1} p_i, \tag{5.3}$$

where $\mathbb{E}[D]$ is the average length of an optimal k-ary one-to-one code D and $h = \lceil \log_k(n - \lceil n/k \rceil) \rceil$.

Proof. Under the standing hypothesis that the probabilities of the source symbols are ordered from the largest to the smallest, we show how to construct a prefix code—by appending the special character $\sqcup$ to the end of (some) codewords of an optimal one-to-one code for S—having the average length upper bounded by (5.2).

Among the class of all the prefix codes that one can obtain by appending the character $\sqcup$ to the end of (some) codewords of an optimal k-ary one-to-one code for S, we aim to construct the one with the minimum average length. Therefore, we have to ensure that, in the k-ary tree representation of the code, the following basic condition holds: For any pair of nodes v_i and v_j, $i < j$, associated with the symbols s_i and s_j, the depth of the node v_j is not smaller than the depth of the node v_i. In fact, if it were otherwise, the average length of the code could be improved.

Therefore, by recalling that $h = \lceil \log_k(n - \lceil n/k \rceil) \rceil$ is the height of the k-ary tree associated with an optimal k-ary one-to-one code D, we have that the prefix code of the minimum average length that one can obtain by appending the special character $\sqcup$ to the end of (some) codewords of D assigns a codeword of length ℓ_i to the i-th symbol $s_i \in S$, where ℓ_i is given by:

$$\ell_i = \begin{cases} \lfloor \log_k((k-1)\,i+1) \rfloor + 1, & \text{if } i \leq \frac{k^{h-1}-1}{k-1} - 1, \\ \lfloor \log_k((k-1)\,i+1) \rfloor + 1, & \text{if } i \geq \frac{k^h+k^{h-1}-2}{k-1} - \lceil n/k \rceil \\ & \text{and } i \leq \frac{k^h-1}{k-1} - 1, \\ \lfloor \log_k((k-1)\,i+1) \rfloor, & \text{otherwise.} \end{cases} \tag{5.4}$$

We stress that the obtained prefix code C is not necessarily a prefix code ending with a space of minimum average length. Let us now justify the expression (5.4). First, since the probabilities $p_1, \ldots, p_n$ are ordered in non-increasing fashion, the codeword lengths ℓ_i of the code are ordered in non-decreasing fashion, that is $\ell_1 \leq \ldots \leq \ell_n$. Therefore, in the k-ary tree representation of the code, it holds the desired basic condition: For any pair of nodes v_i and v_j, $i < j$, associated with the symbols s_i and s_j, the depth of the node v_i is smaller than or equal to the depth of the node v_j.

Furthermore, we need to append the space character only to the k-ary strings that are the prefix of some others. Therefore, let us consider the first n non-empty k-ary strings according to the *radix* order [93], in which, we recall, the k-ary strings are ordered by length and, for equal lengths, ordered according to the lexicographic order. We have that the number of strings that are a prefix of some others is exactly $\lceil \frac{n}{k} \rceil - 1$. One obtains this number by seeing the strings as corresponding to nodes in a k-ary tree with labels $0, \ldots, k-1$ on the edges. The number of strings that are a prefix of some others (among the n strings) is *exactly* equal to the number of internal nodes (except the root) in such a tree. This number of internal nodes is equal to $\lceil \frac{N-1}{k} \rceil - 1$, where N is the total number of nodes that, in our case, is equal to $N = n + 1$ (i.e., N counts also the root of the tree).

Moreover, starting from the optimal one-to-one code constructed according to our second method, that is, by assigning k-ary strings to the symbols by increasing length and, for equal lengths, by inverse order according to the lexicographic order, one can verify that the $\lceil \frac{n}{k} \rceil - 1$ internal nodes are associated with the codewords of the symbols s_i, for i that goes from 1 to $\frac{k^{h-1}-1}{k-1} - 1$, and from $\frac{k^h+k^{h-1}-2}{k-1} - \lceil n/k \rceil$ to $\frac{k^h}{k-1} - 2$.

In fact, since the height of the k-ary tree is $h = \lceil \log_k(n - \lceil n/k \rceil) \rceil$ and since all the levels of the tree, except the last two, are full, we need to append the space to all symbols from 1 to $\frac{k^{h-1}-1}{k-1} - 1$. While on the second-to-last level, we have to append the space only to the remaining internal nodes associated with the symbols s_i, where i goes from $\frac{k^h+k^{h-1}-2}{k-1} - \lceil n/k \rceil$ to $\frac{k^h-1}{k-1} - 1$. Those remaining nodes are exactly, among all the nodes in the second-to-last level, the ones associated with the symbols that have smaller probabilities. Thus, we obtain (5.4).

Summarizing, starting from an optimal k-ary one-to-one code D, we can construct a prefix code C ending with a space with codeword lengths defined as in (5.4). Moreover, this entire construction can be implemented in $O(n)$ time. We simply visit the tree associated with the one-to-one code——which itself is built in linear time——and generate the codewords by appending a space to the identified internal nodes

Thus, the average length of the constructed code satisfies:

$$
\begin{aligned}
\mathbb{E}[C] &= \sum_{i=1}^{n} p_i \ell_i \\
&= \sum_{i=1}^{n} p_i \lfloor \log_k((k-1)\,i + 1) \rfloor + \sum_{i=1}^{\frac{k^{h-1}-1}{k-1}-1} p_i + \sum_{i=\frac{k^h+k^{h-1}-2}{k-1}-\lceil n/k \rceil}^{\frac{k^h-1}{k-1}-1} p_i \\
&= \mathbb{E}[D] + \sum_{i=1}^{\frac{k^{h-1}-1}{k-1}-1} p_i + \sum_{i=\frac{k^h+k^{h-1}-2}{k-1}-\lceil n/k \rceil}^{\frac{k^h-1}{k-1}-1} p_i \\
&\leq \mathbb{E}[D] + \sum_{i=1}^{\lceil n/k \rceil - 1} p_i \\
&\quad \text{(since we are adding } \lceil n/k \rceil - 1 \text{ } p_i\text{'s,} \\
&\qquad \text{and the } p_i\text{'s are ordered).}
\end{aligned}
$$

□

A direct consequence of Lemma 8 is an upper bound on the average length of any *optimal* prefix codes ending with a space. Moreover, the upper bound, given by equation (5.2), is expressed as a function of the average length of an optimal k-ary one-to-one code.

Following a similar line of reasoning, we can also derive a *lower bound* on the average length of optimal prefix codes ending with a space in terms of the average length of optimal one-to-one codes. For such a purpose, we need two intermediate results, whose proofs are deferred in Appendix C.1-C.2. But to establish these results, let us first recall the tree representation of codes. Any k-ary code C can be visualized as a set of nodes in a k-ary tree, where the edges are labeled from 0 to $k-1$. Indeed, the codeword associated with a node v corresponds to the sequence of labels along the unique path from the root to a specific node v. Moreover, a crucial observation that will be useful for our analysis is that in a prefix code, the codewords must correspond exclusively to the leaves of the tree, while in a one-to-one code, they may correspond to any node, including internal ones.

Lemma 9. *Let $S = \{s_1, \dots, s_n\}$ be the set of source symbols, and let $p = (p_1, \dots, p_n)$, $p_1 \geq \dots \geq p_n > 0$, be a probability distribution on S. There exists an* optimal $(k+1)$-ary prefix code ending with a space *C such that for any internal node v (except the root) of the tree representation of C, if we denote by w the k-ary string associated with the node v, then the string $w\sqcup$ belongs to the codeword set of C.*

Lemma 10. *Let C be an arbitrary $(k+1)$-ary prefix code ending with a space, then the k-ary code D that one obtains from C by removing the space $\sqcup$ from each codeword of C is a* one-to-one *code.*

We can now derive a lower bound on the average length of optimal $(k+1)$-ary prefix codes ending with a space in terms of the average length of optimal k-ary one-to-one codes.

Lemma 11. *Let $S = \{s_1, \dots, s_n\}$ be the set of source symbols, and let $p = (p_1, \dots, p_n)$, $p_1 \geq \dots \geq p_n > 0$, be a probability distribution on S, then the average length of an optimal $(k+1)$-ary prefix code C satisfies*

$$\mathbb{E}[C] \geq \mathbb{E}[D] + \sum_{i=1}^{\lceil n/k \rceil - 1} p_{n-i+1}, \tag{5.5}$$

where $\mathbb{E}[D]$ is the average length of an optimal k-ary one-to-one code D on S.

Proof. From Lemma 9, we know that there exists an optimal $(k+1)$-ary prefix code C ending with a space in which exactly $\lceil \frac{n}{k} \rceil - 1$

codewords contain the space character at the end. Let $A \subset \{1, \dots, n\}$ be the set of indices associated with the symbols whose codeword contains the space. Moreover, from Lemma 10, we know that the code D' obtained by removing the space from C is a one-to-one code. Putting it all together, we obtain that

$$\mathbb{E}[D'] = \mathbb{E}[C] - \sum_{i \in A} p_i. \tag{5.6}$$

From (5.6), we have that

$$\begin{aligned}
\mathbb{E}[C] &= \mathbb{E}[D'] + \sum_{i \in A} p_i \\
&\geq \mathbb{E}[D] + \sum_{i \in A} p_i \\
&\quad \text{(since } D' \text{ is a one-to-one code and } D \text{ is an optimal one)} \\
&\geq \mathbb{E}[D] + \sum_{i=1}^{\lceil n/k \rceil - 1} p_{n-i+1} \\
&\quad \text{(since } A \text{ contains } \lceil \tfrac{n}{k} \rceil - 1 \text{ elements)}.
\end{aligned}$$

□

We notice that the gap between the upper bound (5.2) and the lower bound (5.5) is, because of (5.3), less than

$$\sum_{i=1}^{\lceil n/k \rceil - 1} p_i - \sum_{i=1}^{\lceil n/k \rceil - 1} p_{n-i+1} < 1. \tag{5.7}$$

Therefore, the prefix code ending with a space that we construct in Lemma 8 is *almost-optimal*. Indeed, it has an average length that differs from the minimum possible by *at most one*. It is also worth noting that the gap is often significantly smaller than one, since the left-hand side of (5.7) is, usually, much smaller than one. Furthermore, the gap is decreasing in the size of the code alphabet, k.

Having established an efficient method for constructing almost-optimal prefix codes ending with a space, the next natural step is to derive upper and lower bounds on the average length of such codes. Thus, as anticipated before, the following sections are dedicated to deriving upper and lower bounds on the average length of these codes in terms of the k-ary Shannon entropy

$H_k(p) = -\sum_{i=1}^{n} p_i \log_k p_i$ of the source distribution p. Specifically, our approach is twofold. We first establish these entropic bounds for optimal k-ary one-to-one codes. Then, by virtue of the relationships established in Lemma 8 and Lemma 11, these bounds will directly translate into corresponding bounds on the average length of optimal prefix codes ending with a space.

5.3 LOWER BOUNDS

In this section, we provide lower bounds on the average length of optimal one-to-one codes and, subsequently, thanks to Lemma 11, on the average length of optimal prefix codes ending with a space.

For technical reasons, it will be convenient to analyze k-ary one-to-one codes that include the empty word ϵ. These codes correspond to one-to-one mappings of the form $f_\epsilon : S \longmapsto \Sigma_k^+ \cup \{\epsilon\}$.

One can see (cf. (5.1)) that an optimal one-to-one code of this type assigns to the i-th symbol $s_i \in S$ a codeword of length ℓ_i given by:

$$\ell_i = \lfloor \log_k((k-1)i) \rfloor. \tag{5.8}$$

where k is the cardinality of the code alphabet.

For the sake of notation, let L_ϵ and L_+ denote the average length of optimal one-to-one codes that *include* and *exclude* the empty word, respectively. Then, one obtains the following relationship between them:

$$L_+ = L_\epsilon + \sum_{i=1}^{\lfloor \log_k \lceil \frac{n-1}{k} \rceil \rfloor} p_{\frac{k^i-1}{k-1}}. \tag{5.9}$$

Our first result is a generalization of the lower bound on the average length of the optimal one-to-one codes, presented in [21], from the binary case to the general setting of k-ary alphabets, where $k \geq 2$. Our proof technique differs from that of [21] since our analysis deals with a set of source symbols of *bounded* cardinality, while in [21], the authors considered the case of a numerable set of source symbols. The generalization is presented in the following lemma, whose proof is deferred to Appendix C.3.

Lemma 12. *Let $S = \{s_1, \dots, s_n\}$ be the set of source symbols and $p = (p_1, \dots, p_n)$ be a probability distribution on S, with $p_1 \geq \dots \geq p_n$. The average length L_ϵ of an optimal k-ary one-to-one code that includes the empty word satisfies*

$$\begin{aligned} L_\epsilon > & H_k(p) - (H_k(p) + \log_k(k-1)) \\ & \log_k\left(1 + \frac{1}{H_k(p) + \log_k(k-1)}\right) \\ & - \log_k(H_k(p) + \log_k(k-1) + 1), \end{aligned} \tag{5.10}$$

where $H_k(p) = -\sum_{i=1}^{n} p_i \log_k p_i$.

The bound presented in Lemma 12, which relies solely on the entropy, can be tightened by incorporating additional information about the probability distribution—specifically, the value of the largest probability mass. The following result, whose proof is also deferred to Appendix C.4, establishes this improved bound.

Lemma 13. *Let $S = \{s_1, \dots, s_n\}$ be the set of source symbols and $p = (p_1, \dots, p_n)$, $p_1 \geq \dots \geq p_n$, be a probability distribution on S. The average length L_ϵ of an optimal k-ary one-to-one code that includes the empty word satisfies the following:*

1. *If $0 < p_1 \leq 0.5$,*

$$\begin{aligned} L_\epsilon \geq & H_k(p) - (H_k(p) - p_1 \log_k \frac{1}{p_1} + (1-p_1)\log_k(k-1)) \\ & \log_k\left(1 + \frac{1}{H_k(p) - p_1 \log_k \frac{1}{p_1} + (1-p_1)\log_k(k-1)}\right) \\ & - \log_k(H_k(p) - p_1 \log_k \frac{1}{p_1} + (1-p_1)\log_k(k-1) + 1) \\ & - \log_k\left(1 - \left(\frac{1}{kn}\right)^{\log_k\left(1+\frac{1}{1-p_1}\right)}\right), \end{aligned} \tag{5.11}$$

2. ***if*** *$0.5 < p_1 \leq 1$*

$$\begin{aligned} L_\epsilon \geq & H_k(p) - (H_k(p) - \mathcal{H}_k(p_1) + (1-p_1)(1 + \log_k(k-1))) \\ & \log_k\left(1 + \frac{1}{H_k(p) - \mathcal{H}_k(p_1) + (1-p_1)(1+\log_k(k-1))}\right) \end{aligned}$$

$$- \log_k(H_k(p) - \mathcal{H}_k(p_1) + (1-p_1)(1+\log_k(k-1)) + 1)$$
$$- \log_k\left(1 - \left(\frac{1}{kn}\right)^{\log_k\left(1+\frac{1}{1-p_1}\right)}\right), \tag{5.12}$$

where $\mathcal{H}_k(p_1) = -p_1 \log_k p_1 - (1-p_1)\log_k(1-p_1)$ *is the k-ary binary Shannon entropy.*

Thanks to Lemma 11 and to Equation (5.9), we can use the above lower bounds on L_ϵ to derive lower bounds on the average length of optimal prefix codes ending with a space, as shown in the following theorems.

Theorem 19. *The average length of an optimal* $(k+1)$*-ary prefix code ending with a space C satisfies*

$$\begin{aligned}\mathbb{E}[C] >& H_k(p) - (H_k(p) + \log_k(k-1))\\ & \log_k\left(1 + \frac{1}{H_k(p) + \log_k(k-1)}\right)\\ & - \log_k(H_k(p) + \log_k(k-1) + 1)\\ & + \sum_{i=1}^{\lceil \frac{n}{k} \rceil - 1} p_{n-i+1} + \sum_{i=1}^{\lfloor \log_k \lceil \frac{n-1}{k} \rceil \rfloor} p_{\frac{k^i-1}{k-1}}.\end{aligned} \tag{5.13}$$

Proof. From Lemma 11 and the formula (5.9), we have

$$\mathbb{E}[C] \geq L_\epsilon + \sum_{i=1}^{\lceil \frac{n}{k} \rceil - 1} p_{n-i+1} + \sum_{i=1}^{\lfloor \log_k \lceil \frac{n-1}{k} \rceil \rfloor} p_{\frac{k^i-1}{k-1}}. \tag{5.14}$$

By applying the lower bound (5.10) of Lemma 12 to (5.14), we obtain (5.13). □

Analogously, by exploiting (the possible) knowledge of the maximum source symbol probability value, we have the following result.

Theorem 20. *The average length of an optimal* $(k+1)$*-ary prefix code ending with a space C satisfies the following lower bounds:*

1. *If* $0 < p_1 \leq 0.5$*:*

$$\mathbb{E}[C] \geq H_k(p) - (H_k(p) - p_1 \log_k \frac{1}{p_1} + (1-p_1)\log_k(k-1))$$

$$\log_k\left(1+\frac{1}{H_k(p)-p_1\log_k\frac{1}{p_1}+(1-p_1)\log_k(k-1)}\right)$$
$$-\log_k\left(H_k(p)-p_1\log_k\frac{1}{p_1}+(1-p_1)\log_k(k-1)+1\right)$$
$$-\log_k\left(1-\left(\frac{1}{kn}\right)^{\log_k\left(1+\frac{1}{1-p_1}\right)}\right)$$
$$+\sum_{i=1}^{\lceil\frac{n}{k}\rceil-1}p_{n-i+1}+\sum_{i=1}^{\lfloor\log_k\lceil\frac{n-1}{k}\rceil\rfloor}p_{\frac{k^i-1}{k-1}}. \tag{5.15}$$

2. *If* $0.5 < p_1 \leq 1$*:*

$$\mathbb{E}[C] \geq H_k(p)-(H_k(p)-\mathcal{H}_k(p_1)+(1-p_1)(1+\log_k(k-1)))$$
$$\log_k\left(1+\frac{1}{H_k(p)-\mathcal{H}_k(p_1)+(1-p_1)(1+\log_k(k-1))}\right)$$
$$-\log_k\left(H_k(p)-\mathcal{H}_k(p_1)+(1-p_1)(1+\log_k(k-1))+1\right)$$
$$-\log_k\left(1-\left(\frac{1}{kn}\right)^{\log_k\left(1+\frac{1}{1-p_1}\right)}\right)$$
$$+\sum_{i=1}^{\lceil\frac{n}{k}\rceil-1}p_{n-i+1}+\sum_{i=1}^{\lfloor\log_k\lceil\frac{n-1}{k}\rceil\rfloor}p_{\frac{k^i-1}{k-1}}. \tag{5.16}$$

Proof. From Lemma 11 and Equation (5.9), one obtains

$$\mathbb{E}[C]\geq L_e+\sum_{i=1}^{\lceil\frac{n}{k}\rceil-1}p_{n-i+1}+\sum_{i=1}^{\lfloor\log_k\lceil\frac{n-1}{k}\rceil\rfloor}p_{\frac{k^i-1}{k-1}}. \tag{5.17}$$

By applying the lower bounds (5.11)-(5.12) of Lemma 13 to (5.17), one gets (5.15) or (5.16) according to the value of the maximum source symbol probability. □

5.4 UPPER BOUNDS

This section mirrors the structure of the previous one. We first derive *upper bounds* on the average length of optimal k-ary one-to-one codes. Successively, we translate them into the corresponding

upper bounds on the average length of optimal $(k+1)$-ary prefix codes ending with a space.

We begin by generalizing the upper bound obtained in [135], extending the result from the binary case to the general k-ary case, for $k \geq 2$. The result is presented in the following lemma, whose proof is deferred to Appendix C.5.

Lemma 14. *Let $S = \{s_1, \ldots, s_n\}$ be the set of source symbols and $p = (p_1, \ldots, p_n)$, $p_1 \geq \ldots \geq p_n$, be a probability distribution on S. The average length L_ϵ of an optimal k-ary one-to-one code that includes the empty word satisfies*

$$L_\epsilon \leq H_k(p) + \log_k(k-1). \tag{5.18}$$

Following the same principle used for the lower bound in Lemma 13, we can tighten the upper bound by incorporating additional information about the source distribution beyond its entropy—specifically, the maximum probability value.

The following lemma, whose proof is deferred to Appendix C.6, formalizes this improved bound. It generalizes the result established in [21] from the binary case to the arbitrary k-ary setting for $k \geq 2$.

Lemma 15. *Let $S = \{s_1, \ldots, s_n\}$ be the set of source symbols and $p = (p_1, \ldots, p_n)$, $p_1 \geq \ldots \geq p_n$, be a probability distribution on S. The average length L_ϵ of an optimal k-ary one-to-one code that includes the empty word satisfies*

$$L_\epsilon \leq \begin{cases} H_k(p) - p_1 \log_k \frac{1}{p_1} + (1-p_1)\log_k(k-1) & \text{if } 0 < p_1 \leq 0.5, \\ H_k(p) - \mathcal{H}_k(p_1) + (1-p_1)\log_k 2(k-1) & \text{if } 0.5 < p_1 \leq 1, \end{cases} \tag{5.19}$$

where $\mathcal{H}_k(p_1) = -p_1 \log_k p_1 - (1-p_1)\log_k(1-p_1)$ is the k-ary binary Shannon entropy.

We can now use the upper bound for one-to-one codes, presented in Lemma 14 and 15, to derive our results. The following theorems translate these findings into the corresponding upper bounds on the average length of optimal $(k+1)$-ary prefix codes ending with a space.

Theorem 21. *Let $S = \{s_1, \dots, s_n\}$ be the set of source symbols and $p = (p_1, \dots, p_n)$, $p_1 \geq \dots \geq p_n$, be a probability distribution on S. The average length of an optimal $(k+1)$-ary prefix code ending with a space C for S satisfies*

$$\begin{aligned}\mathbb{E}[C] \leq & H_k(p) + \log_k(k-1) + \sum_{i=1}^{\lfloor \log_k \lceil \frac{n-1}{k} \rceil \rfloor} p_{\frac{k^i-1}{k-1}} \\ & + \sum_{i=1}^{\frac{k^{h-1}-1}{k-1}-1} p_i + \sum_{i=\frac{k^h+k^{h-1}-2}{k-1}-\lceil n/k \rceil}^{\frac{k^h-1}{k-1}-1} p_i \qquad (5.20) \\ \leq & H_k(p) + \log_k(k-1) + \sum_{i=1}^{\lfloor \log_k \lceil \frac{n-1}{k} \rceil \rfloor} p_{\frac{k^i-1}{k-1}} + \sum_{i=1}^{\lceil \frac{n}{k} \rceil - 1} p_i, \qquad (5.21)\end{aligned}$$

where $h = \lceil \log_k(n - \lceil n/k \rceil) \rceil$.

Proof. From Lemma 8 and formula (5.9), it follows that

$$\begin{aligned}\mathbb{E}[C] \leq & L_\epsilon + \sum_{i=1}^{\lfloor \log_k \lceil \frac{n-1}{k} \rceil \rfloor} p_{\frac{k^i-1}{k-1}} \\ & + \sum_{i=1}^{\frac{k^{h-1}-1}{k-1}-1} p_i + \sum_{i=\frac{k^h+k^{h-1}-2}{k-1}-\lceil n/k \rceil}^{\frac{k^h-1}{k-1}-1} p_i. \qquad (5.22)\end{aligned}$$

By applying the upper bound (5.18) of Lemma 14 on L_ϵ to (5.22), one gets (5.20). □

Theorem 22. *Let $S = \{s_1, \dots, s_n\}$ be the set of source symbols and $p = (p_1, \dots, p_n)$, $p_1 \geq \dots \geq p_n$, be a probability distribution on S. The average length of an optimal $(k+1)$-ary prefix code ending with a space C for S satisfies*

$$\mathbb{E}[C] \leq \begin{cases} H_k(p) - p_1 \log_k \frac{1}{p_1} + (1-p_1)\log_k(k-1) \\ + \sum_{i=1}^{\lceil \frac{n}{k} \rceil - 1} p_i + \sum_{i=1}^{\lfloor \log_k \lceil \frac{n-1}{k} \rceil \rfloor} p_{\frac{k^i-1}{k-1}} & \text{if } 0 < p_1 \leq 0.5, \\ H_k(p) - \mathcal{H}_k(p_1) + (1-p_1)\log_k 2(k-1) \\ + \sum_{i=1}^{\lceil \frac{n}{k} \rceil - 1} p_i + \sum_{i=1}^{\lfloor \log_k \lceil \frac{n-1}{k} \rceil \rfloor} p_{\frac{k^i-1}{k-1}} & \text{if } 0.5 < p_1 \leq 1. \end{cases} \qquad (5.23)$$

Proof. From Lemma 8 and formula (5.9) and by recalling that $h = \lceil \log_k(n - \lceil n/k \rceil) \rceil$, it follows that

$$\begin{aligned}
\mathbb{E}[C] \leq & L_\epsilon + \sum_{i=1}^{\lfloor \log_k \lceil \frac{n-1}{k} \rceil \rfloor} p_{\frac{k^i-1}{k-1}} \\
& + \sum_{i=1}^{\frac{k^{h-1}-1}{k-1}-1} p_i + \sum_{i=\frac{k^h+k^{h-1}-2}{k-1}-\lceil n/k \rceil}^{\frac{k^h-1}{k-1}-1} p_i \\
\leq & L_\epsilon + \sum_{i=1}^{\lceil \frac{n}{k} \rceil - 1} p_i + \sum_{i=1}^{\lfloor \log_k \lceil \frac{n-1}{k} \rceil \rfloor} p_{\frac{k^i-1}{k-1}}. \qquad (5.24)
\end{aligned}$$

Now, by applying the upper bound (5.19) of Lemma 15 on L_ϵ to (5.24), one gets (5.23), concluding the proof. □

5.5 CONCLUDING REMARKS

This chapter defined and analyzed the class of prefix codes ending with a space, i.e., a restricted class of prefix codes where a specific alphabet character is constrained to serve exclusively as a codeword delimiter. The core of our analysis was based on studying the strict relationship between such codes and the well-known class of one-to-one codes. This connection allowed us to obtain two main contributions. First, we provided a linear-time approach for constructing almost-optimal prefix codes ending with a space starting from optimal one-to-one codes. Second, by leveraging their relationship again, we derived entropic lower bounds and upper bounds on the average length of general optimal $(k+1)$-ary prefix codes ending with a space.

We conclude this chapter with some final observations and open problems.

First, following the line of the work of Jaynes [80], let us see if one can estimate how much the average length of an optimal prefix code ending with a space differs from the one of an optimal *unrestricted* prefix code on the same alphabet, i.e., a prefix code in which the space symbol ␣ is not constrained to appear as a termination symbol.

Let S be the set of source symbols and p be a probability distribution on S. We denote by $C_␣$ an optimal $(k+1)$-ary prefix

code ending with a space for S and by C an optimal unrestricted $(k+1)$-ary prefix code for S. Clearly, the average length of $C_{\sqcup}$ is bounded by $\mathbb{E}[C_{\sqcup}] < H_k(p) + 1$. This is because the set of codes available for $C_{\sqcup}$ includes all unrestricted k-ary prefix codes (by simply not using the space symbol), and the average length of an optimal k-ary prefix code is strictly less than $H_k(p) + 1$. With this, we can bound the difference in *performance* as follows

$$\begin{aligned}\mathbb{E}[C_{\sqcup}] - \mathbb{E}[C] &< H_k(p) + 1 - \mathbb{E}[C] \\ &\leq H_k(p) + 1 - H_{k+1}(p) \\ &\quad (\text{since } \mathbb{E}[C] \geq H_{k+1}(p)) \\ &= H_k(p) \left(1 - \frac{1}{\log_k(k+1)}\right) + 1.\end{aligned}$$

Since $\lim_{k\to\infty} \log_k(k+1) = 1$, we have that, as the cardinality of the code alphabet increases, the constraint that the space can appear only at the end of codewords becomes less and less influential. This is somewhat reminiscent of the classic result by Jaynes [80] that we have discussed in Section 5.1.

In the previous chapters, we emphasized the strong connection between Code Theory and Search Theory. For instance, how binary codes naturally lead to binary search procedures and vice versa. Let us see how, in this case too, this link between codes and search procedures still holds. In fact, prefix codes ending with a space also admit a search-theoretic interpretation, specifically 3-ary prefix codes ending with a space map to search procedures based on *ternary-outcome* queries.

We recall that classical binary prefix codes correspond to search procedures that ask binary membership queries, e.g., "Does the unknown element x belong to a given subset S?". Similarly, one can see how our prefix codes ending with a space correspond to a more powerful search model. They model search procedures that ask queries of the following kind: "Given a subset S and a specific element $y \in S$, is the unknown element x equal to y or does it belong to $S \setminus \{y\}$?".

Practically, the space symbol adds the possibility to ask also for an equality test, which allows for the potential early termination of the search process if a match is found. These kinds of questions with ternary outcomes are very similar to those done in binary

search trees (see Section 3.3.1). In fact, the connection becomes more explicit when we consider the *alphabetic variant* of such codes. Depending on which order we decide on the alphabet $\{0, 1, \sqcup\}$, we get different kinds of search procedures. For example, if we adopt the order $0 \prec \sqcup \prec 1$, we get the same kind of queries used in binary search trees: "Is the unknown element $x < y$, $x = y$, or $x > y$?". Thus, the alphabetic version of 3-ary prefix codes ending with a space provides a somewhat generalization of the comparison-based model related to binary search trees.

Thus, thanks to this similarity, the problem of constructing optimal code in the alphabetic setting can be solved quite efficiently by readapting the classical dynamic programming algorithms originally developed for optimal binary search trees [93].

However, in the general prefix setting, the problem of constructing optimal codes remains open. In fact, we provided only an efficient way for constructing almost-optimal codes. The existence of a polynomial-time algorithm for finding the truly optimal code is currently unknown. Finally, another interesting open path would be that of deriving a new Kraft-like condition for the existence of such a new kind of prefix codes. This could lead to an improvement of the upper bounds that we presented.

6

CONCLUSIONS AND OPEN PROBLEMS

This thesis investigated several problems concerning VLCs, with particular attention to *alphabetic* and *prefix* codes. In many cases, we began with problems formulated in the context of *Search Theory*, and, by exploiting the deep connection between VLCs and search procedures, we reformulated them in terms of *Coding Theory*. This reformulation made it possible to apply the full range of tools and results from *Information Theory*. In essence, we reduced the problem of designing efficient search procedures—those minimizing the *average number of queries*—to the problem of constructing codes with *minimum average length*. Using VLCs as a framework enabled us to develop *optimal* or *near-optimal* search procedures in terms of average complexity across several different settings. Furthermore, this approach allowed us to derive *theoretical lower and upper bounds* on their complexity as well.

To better understand how one can exploit such a framework, let us briefly recall the main results and contributions that we presented in this thesis.

In Chapter 3, we considered the natural and classical setting of search procedures that operate through *comparison queries*, or through *comparison and equality tests*. We showed how such procedures can be naturally represented by binary VLCs, specifically by *binary alphabetic codes* and BSTs, respectively. By exploiting this correspondence, the design of efficient search procedures was reduced to the study of the average length of the associated codes. In this context, we established new and improved *upper bounds* on the average length of optimal alphabetic codes, making a significant contribution to the field and improving upon the well-known bound presented in the seminal work [139]. Furthermore, we also developed an efficient *linear-time* algorithm for constructing near-optimal codes that achieve these bounds.

Focusing on BSTs, we introduced a conceptual framework for building in linear time almost-optimal BSTs directly from almost-

optimal alphabetic codes, thereby deriving new and meaningful bounds in the field of BSTs as well. Finally, this framework makes it possible to translate any future improvement in the theory of alphabetic codes directly into corresponding advances for BSTs.

In Chapter 4, we considered search problems with *asymmetric test outcome costs*. As in the previous chapter, we reformulated the problem within the framework of VLCs, introducing a new class of codes that we denoted as (α, β)-constrained codes. In these codes, each codeword must satisfy a cost constraint where every 0-bit and 1-bit incur a cost of α and β, respectively.

We first considered the simpler case of the *alphabetic* variant of such codes and presented an efficient polynomial-time algorithm for constructing optimal codes, i.e., codes corresponding to search procedures with minimum average complexity. Moreover, for the specific case of $(0, \beta)$-constrained alphabetic codes, we also provided a necessary and sufficient condition for their existence. Then, we turned to the more challenging case of *prefix codes*, where we derived a Kraft-like condition for the existence of $(0, \beta)$-constrained prefix codes, i.e., a necessary and sufficient condition for the existence of $(0, \beta)$-constrained prefix codes given their codeword lengths.

In Chapter 5, we focused on prefix codes. Specifically, we addressed the problem of constructing prefix codes in which one symbol of the alphabet is constrained to serve solely as a termination character, following the line of work introduced in [80]. We exploited the more general class of one-to-one codes to devise a linear-time method for constructing almost-optimal codes of this type. Furthermore, we established both lower and upper bounds on the average length of such codes. Finally, following the main line of the thesis, we connected this class of codes to search problems. In particular, the alphabetic variant of these codes corresponds to search procedures that operate through *comparison and equality tests*, thereby re-obtaining the class of BSTs.

To conclude, this thesis has explored the profound correspondence between search procedures and variable-length codes, showing that these two seemingly distinct domains are, in fact, closely connected. Every deterministic search strategy can be viewed as a code that encodes the information acquired during the search,

while every code implicitly defines a search procedure that progressively reveals information.

This perspective is not only conceptually elegant but also practically powerful. It enables classical search problems to be reformulated, analyzed, and optimized through the mathematical framework of Coding Theory and, by extension, Information Theory. The results presented throughout this work demonstrate that efficiency in information acquisition and efficiency in information representation are governed by the same fundamental principle: both seek to minimize uncertainty as effectively as possible.

Ultimately, the contribution of this thesis lies in establishing a general methodology for translating problems of search optimization into problems of code design, and vice versa. This conceptual bridge proposes a fascinating paradigm for understanding search as an act of compression.

6.1 OPEN PROBLEMS

In this final section of the thesis, we list and highlight several open problems related to the topics studied previously and to the broader area of VLCs.

- In Chapter 4, we provided polynomial-time algorithms for constructing optimal (α, β)-constrained alphabetic codes. However, the existence of a polynomial-time algorithm for the general (α, β)-constrained prefix code problem with $\alpha \neq \beta$ remains an open question.

- While Chapter 5 introduced an efficient algorithm for constructing near-optimal *prefix codes ending with a space*, a method for finding the truly *optimal code* in polynomial time is not yet known and remains an open area for investigation.

- As presented in Section 2.3, the Hu-Tucker algorithm [74] constructs an alphabetic code of minimum average length in $O(n \log n)$ time. To date, there is no non-trivial lower bound on the complexity of algorithms that construct alphabetic codes of minimum average length. This leaves a significant gap in the field, motivating the following open question:

Does an algorithm exist for constructing optimal alphabetic codes with a time complexity below $n \log n$?

- An open problem, strictly related to the previous one, concerns the optimality verification of a code: given a code C and a probability distribution $p = \langle p_1, \ldots, p_n \rangle$, can one check in $O(n)$ time whether C is an optimal alphabetic code for p? This verification problem is significant because its computational complexity serves as a lower bound on the complexity of constructing an optimal alphabetic code. Some partial results have been presented in [121], but the general problem remains open. On the other hand, for a prefix code, the optimality can be checked in linear time [121].

- A different variant of comparison-based search algorithms, introduced by Ambainis *et al.* [8] and Ahlswede and Cai [4], considers a *delayed-query model*. In this model, the outcome of any test is received d time units after the test has been performed. This implies that the algorithm's i^{th} query can only be based on the outcomes of the queries from the 1^{st} to the $(i-d-1)^{th}$. This class of algorithms induces alphabetic codes with a structure that differs from that of classical ones. The studies presented in [4, 8] focus on worst-case analysis of such algorithms. However, the analysis of average-case performance for such delayed-query models remains an open problem.

- In [38], Dagan *et al.* extend the classical setting of alphabetic codes (or, equivalently, comparison-based search procedures) to scenarios in which "lies" may occur—that is, cases where the search procedure must correctly identify the unknown element even though up to a fixed number k of queries may return erroneous answers. They further prove the existence of an algorithm for this problem whose expected number of queries is upper-bounded by

$$H(p) + k \sum_{i=1}^{n} p_i \log\log \frac{1}{p_i} + O\left(k \sum_{i=1}^{n} p_i \log\log\log \frac{1}{p_i} + k \log k \right), \tag{6.1}$$

where $p = \langle p_1, \ldots, p_n \rangle$ denotes the probability distribution over the elements of the search space. Moreover, they establish that *any* algorithm solving the problem must perform an average number of queries lower bounded by

$$H(p) + k \sum_{i=1}^{n} p_i \log \log \frac{1}{p_i} - (k \log k + k + 1). \tag{6.2}$$

Thus, a natural question that follows is whether the gap between the upper bound (6.1) and the lower bound (6.2) can be tightened.

- In Subsection 2.6.1, we reviewed several results concerning alphabetic codes that are optimal under different criteria. For some of these cases, efficient algorithms for constructing optimal alphabetic codes are known. However, the problem of establishing tight upper bounds on the cost of optimal solutions remains largely open. In fact, the class of codes studied in Chapter 4 also falls within this research direction.

- In Subsection 2.6.3, we presented the existing results in the area of alphabetic codes for partially ordered sets. Although some interesting results have been established, this research area remains largely unexplored.

- To the best of our knowledge, no algorithms currently exist that can efficiently update the structure of optimal alphabetic codes as the symbol probabilities change. It would be of great interest to design such algorithms, in the spirit of the dynamic approach proposed by Knuth [92] for classical prefix codes.

- Given a probability distribution $p = \langle p_1, \ldots, p_n \rangle$, there may exist several distinct alphabetic codes that achieve the same minimum average length. In such cases, it would be interesting to modify existing algorithms for constructing minimum average-length alphabetic codes so that they also minimize the maximum codeword length or other relevant parameters. For prefix codes, this problem has been investigated in [124].

Part IV

APPENDIX

A

CHAPTER 3 - NEW RESULTS ON ALPHABETIC CODES AND BINARY SEARCH TREES

A.1 PROOF OF LEMMA 1

Lemma. *Let $L = \langle \ell_1, \dots, \ell_m \rangle$ be a list of integers such that $\texttt{sum}(L, m) < 1$, let i, j be integers such that $1 \leq i < j \leq m$, and let $\{\texttt{sum}(L, i), \texttt{sum}(L, i+1), \dots, \texttt{sum}(L, j)\}$ be a subset of consecutive elements of $\{\texttt{sum}(L, 1), \dots, \texttt{sum}(L, m)\}$. For each $k = i, \dots, j-1$, let t_k be the* smallest *integer s for which the binary expansion of $\texttt{sum}(L, k)$ differs from the binary expansion of $\texttt{sum}(L, k+1)$ on the s^{th} bit, and define $t_{ij} = \min_{i \leq k < j} t_k$. Then, it holds that:*

1. $$t_{ij} = \lceil -\log_2(\texttt{sum}(L, i) \oplus \texttt{sum}(L, j)) \rceil, \tag{A.1}$$

   where $\texttt{sum}(L, i) \oplus \texttt{sum}(L, j)$ is the numerical *value whose binary expansion is the result of the XOR operation between the binary expansions of $\texttt{sum}(L, i)$ and $\texttt{sum}(L, j)$, and*

2. *there exists an index $z \in \{i, \dots, j-1\}$ such that $\texttt{sum}(L, z) < \texttt{trunc}(t_{ij}, \texttt{sum}(L, i)) + 2^{-t_{ij}}$ and $\texttt{sum}(L, z+1) \geq \texttt{trunc}(t_{ij}, \texttt{sum}(L, i)) + 2^{-t_{ij}}$.*

Proof. To prove claim 1) of the lemma, let s be the *smallest* integer for which the binary expansions of the pair $\texttt{sum}(L, i)$ and $\texttt{sum}(L, j)$ differ on the s^{th} bit. We first prove that s and $t = t_{ij}$ (cfr, eq. (A.1)) have the same value.

First of all, by definition of s, we know that the binary expansions of $\texttt{sum}(L, i)$ and $\texttt{sum}(L, j)$ are equal on the first $s-1$ bits. Moreover, since $\texttt{sum}(L, i)$ and $\texttt{sum}(L, j)$ are equal on the first $s-1$ bits and $\texttt{sum}(L, i) < \cdots < \texttt{sum}(L, j)$, it follows that for each $k = i, \dots, j-1$, the binary expansions of $\texttt{sum}(L, k)$ and $\texttt{sum}(L, k+1)$ are equal on the first $s-1$ bits, too. Therefore, it holds that $s \leq t$.

Since $\texttt{sum}(L, i)$ and $\texttt{sum}(L, j)$ differ on the s^{th} bit and $\texttt{sum}(L, i) < \texttt{sum}(L, j)$, we know that the s^{th} bit of $\texttt{sum}(L, i)$ is 0, while the s^{th} bit

of $\mathtt{sum}(L, j)$ is 1. As a consequence, since $\mathtt{sum}(L, i) < \cdots < \mathtt{sum}(L, j)$ and for each $k = i, \ldots, j-1$, *both* the binary expansions of $\mathtt{sum}(L, k)$ *and* of $\mathtt{sum}(L, k+1)$ are equal on the first $s-1$ bits, there must exist an index $k \in \{i, \ldots, j-1\}$ such that the s^{th} bit of $\mathtt{sum}(L, k)$ is 0, and the s^{th} bit of $\mathtt{sum}(L, k+1)$ is 1. Therefore, since $t = t_{ij} = \min_{i \le k < j} t_k$ and there exists k such that $t_k = s$, it holds that $s \ge t$. Thus, we finally get that $s = t$.

One can also see that $s = \lceil -\log_2(\mathtt{sum}(L, i) \oplus \mathtt{sum}(L, j)) \rceil$, where $\mathtt{sum}(L, i) \oplus \mathtt{sum}(L, j)$ is the numerical value whose binary expansion is equal to the XOR operation between the binary expansions of $\mathtt{sum}(L, i)$ and $\mathtt{sum}(L, j)$. Indeed, since the binary expansions of $\mathtt{sum}(L, i)$ and $\mathtt{sum}(L, j)$ are equal on the first $s-1$ bits, it holds that the binary expansion of the value $\mathtt{sum}(L, i) \oplus \mathtt{sum}(L, j)$ contains the first bit equal to 1 exactly in the s^{th} position. Therefore, one gets that s is the smallest integer for which it holds that

$$2^{-s} \le \mathtt{sum}(L, i) \oplus \mathtt{sum}(L, j). \tag{A.2}$$

From (A.2), we obtain that $s = \lceil -\log_2(\mathtt{sum}(L, i) \oplus \mathtt{sum}(L, j)) \rceil$.

To prove claim 2) of the lemma, let $z \in \{i, \ldots, j-1\}$ denote an index for which it holds that $t_z = t_{ij}$. Then, we have that the binary expansion of $\mathtt{sum}(L, z)$ differs from the binary expansion of $\mathtt{sum}(L, z+1)$ on the t_{ij}^{th} bit. In particular, from the definition of t_z, the binary expansions of $\mathtt{sum}(L, z)$ and $\mathtt{sum}(L, z+1)$ are equal on the first $t_{ij} - 1$ bits, and, since $\mathtt{sum}(L, z) < \mathtt{sum}(L, z+1)$, we have that $\mathtt{sum}(L, z)$ has a binary expansion with 0 in the t_{ij}^{th} position, while $\mathtt{sum}(L, z+1)$ has a binary expansion with 1 in the t_{ij}^{th} position. Finally, since $\mathtt{sum}(L, i) < \cdots < \mathtt{sum}(L, j)$ and since the binary expansions of the values in the subset $\{\mathtt{sum}(L, i), \ldots, \mathtt{sum}(L, j)\}$ are equal on the first $t_{ij} - 1$ bits (this follows from the definition of t_{ij}), we obtain that for the index z it holds that

$$\mathtt{sum}(L, z) < \mathtt{trunc}(t_{ij}, \mathtt{sum}(L, i)) + 2^{-t_{ij}}$$

and

$$\mathtt{sum}(L, z+1) \ge \mathtt{trunc}(t_{ij}, \mathtt{sum}(L, i)) + 2^{-t_{ij}}.$$

This concludes the proof.

□

A.2 LEMMA OF REMARK 1

Lemma. *For any non-increasing dyadic distribution $\phi = (\phi_1, \dots, \phi_m)$, let $\nu' = (0, \phi_1, 0, \dots, 0, \phi_m, 0)$ be its fully extended distribution, then there does not exist any alphabetic code with lengths $L = \langle \ell_1, \dots, \ell_{2m+1} \rangle$ defined as follows*

$$\ell_i = \begin{cases} x_i & \text{if } \nu'_i = 0, \\ \lceil -\log_2 \nu'_i \rceil + 1 & \text{if } \nu'_i > 0, \end{cases}$$

where $x_i \in \mathbb{N}_+$ are arbitrarily chosen values.

Proof. Without loss of generality, let us assume that $\phi_m > 0$. Otherwise, we can simply show that the lemma holds for $(\phi_1, \dots, \phi_{m-1})$ and hence for ϕ.

Let $L = \langle \ell_1, \dots, \ell_{2m+1} \rangle$ be the list of lengths as defined in the statement of the lemma. We will consider two cases. The first corresponds to the case in which, for each $i \in \{1, 3, \dots, 2m-1\}$, we have $x_i = \ell_i \geq \ell_{i+1}$ and $x_{2m+1} = \ell_{2m+1} \geq \ell_{2m}$; the second case is for all other possible choices of the x_i.

Case 1: Recall that, for $i \in \{2, \dots, m\}$, we have that $\alpha_i = \min(\ell_{i-1}, \ell_i)$. Since ϕ is non-increasing, for each $i \in \{2, 4, \dots, 2m-2\}$, it follows that $\ell_i \leq \ell_{i+2}$. Thereby, for each $i \in \{2, \dots, 2m\}$, it holds that $\alpha_i \leq \alpha_{i+1}$. In particular, for each $i \in \{2, 4, \dots, 2m\}$, we have that

$$\alpha_i = \alpha_{i+1} = \lceil -\log_2 \phi_{\frac{i}{2}} \rceil + 1 = -\log_2 \phi_{\frac{i}{2}} + 1.$$

Putting it all together we obtain that

$$\begin{aligned} \mathtt{sum}(L, i) &= \mathtt{trunc}(\alpha_i, \mathtt{sum}(L, i-1)) + 2^{-\alpha_i} \\ &= \mathtt{sum}(L, i-1) + 2^{-\alpha_i} \\ &= \mathtt{sum}(L, i-1) + 2^{-\left(-\log_2 \phi_{\lfloor \frac{i}{2} \rfloor} + 1\right)} \\ &= \mathtt{sum}(L, i-1) + \frac{\phi_{\lfloor \frac{i}{2} \rfloor}}{2}, \end{aligned}$$

where the second equality holds because the α_i are non-decreasing, and therefore the binary representation of $\mathtt{sum}(L, i-1)$ contains bits equal to 1 only in the positions less than or equal to α_i.

Since we add up each ϕ_i exactly once without any truncation, $\texttt{sum}(L, 2m+1) = \sum_{i=1}^{m} \phi_i = 1$ and by Theorem 10 there does not exist any alphabetic code for the lengths L.

Case 2: In this case, there is at least one $\ell_i = x_i$ that does not satisfy the condition of Case 1. Let us suppose that $\texttt{sum}(L, 2m+1) < 1$ and show that this leads to a contradiction.

Consider the sub-case in which we increase by 1 x_i, with $1 < i < 2m+1$. Let $L' = \langle \ell_1, \ldots, \ell_{i-1}, \ell_i + 1, \ell_{i+1}, \ldots, \ell_{2m+1} \rangle$ be the list of lengths after increasing such x_i by 1. Let us show that $\texttt{sum}(L', 2m+1)$ can only decrease with respect to $\texttt{sum}(L, 2m+1)$. Since x_i can only affect α_i and α_{i+1}, we have four cases:

- α_i increases by 1 and α_{i+1} does not change:

$$\begin{aligned}
\texttt{sum}(L', i) &= \texttt{trunc}(\alpha_i + 1, \texttt{sum}(L, i-1)) + 2^{-(\alpha_i+1)} \\
&\leq \texttt{trunc}(\alpha_i, \texttt{sum}(L, i-1)) + 2^{-(\alpha_i+1)} \\
&\quad + 2^{-(\alpha_i+1)} \\
&= \texttt{trunc}(\alpha_i, \texttt{sum}(L, i-1)) + 2^{-\alpha_i} \\
&= \texttt{sum}(L, i).
\end{aligned}$$

- α_i does not change and α_{i+1} increases by 1:

$$\begin{aligned}
\texttt{sum}(L', i+1) &= \texttt{trunc}(\alpha_{i+1} + 1, \texttt{sum}(L, i)) \\
&\quad + 2^{-(\alpha_{i+1}+1)} \\
&\leq \texttt{trunc}(\alpha_{i+1}, \texttt{sum}(L, i)) \\
&\quad + 2^{-(\alpha_{i+1}+1)} + 2^{-(\alpha_{i+1}+1)} \\
&= \texttt{trunc}(\alpha_{i+1}, \texttt{sum}(L, i)) + 2^{-\alpha_{i+1}} \\
&= \texttt{sum}(L, i+1).
\end{aligned}$$

- Both α_i and α_{i+1} increase by 1: then $\alpha_i = \alpha_{i+1}$ and

$$\begin{aligned}
\texttt{sum}(L', i+1) &= \texttt{trunc}(\alpha_{i+1} + 1, \texttt{sum}(L', i)) \\
&\quad + 2^{-(\alpha_{i+1}+1)} \\
&= \texttt{trunc}(\alpha_{i+1} + 1, \\
&\qquad \texttt{trunc}(\alpha_i + 1, \texttt{sum}(L, i-1)) \\
&\qquad + 2^{-(\alpha_i+1)}) \\
&\quad + 2^{-(\alpha_{i+1}+1)}
\end{aligned}$$

$$
\begin{aligned}
&= \texttt{trunc}(\alpha_i + 1, \texttt{sum}(L, i-1)) \\
&\quad + 2^{-(\alpha_i+1)} + 2^{-(\alpha_{i+1}+1)} \\
&\quad (\text{since } \alpha_i = \alpha_{i+1}) \\
&= \texttt{trunc}(\alpha_i + 1, \texttt{sum}(L, i-1)) \\
&\quad + 2^{-(\alpha_i+1)} + 2^{-(\alpha_i+1)} \\
&= \texttt{trunc}(\alpha_i + 1, \texttt{sum}(L, i-1)) + 2^{-\alpha_i} \\
&\leq \texttt{trunc}(\alpha_i, \texttt{sum}(L, i-1)) \\
&\quad + 2^{-\alpha_i} + 2^{-(\alpha_i+1)} \\
&< \texttt{trunc}(\alpha_i, \texttt{sum}(L, i-1)) \\
&\quad + 2^{-\alpha_i} + 2^{-\alpha_{i+1}} \\
&= \texttt{trunc}(\alpha_{i+1}, \\
&\qquad \texttt{trunc}(\alpha_i, \texttt{sum}(L, i-1)) + 2^{-\alpha_i}) \\
&\quad + 2^{-\alpha_{i+1}} \\
&= \texttt{trunc}(\alpha_{i+1}, \texttt{sum}(L, i)) + 2^{-\alpha_{i+1}} \\
&= \texttt{sum}(L, i+1).
\end{aligned}
$$

- Both α_i and α_{i+1} do not change: the final sum remains the same.

Therefore, we have that $\texttt{sum}(L', 2m+1) \leq \texttt{sum}(L, 2m+1)$.

Consider now the sub-cases in which we increase by 1 x_1 or x_{2m+1}. In such sub-cases, only α_2 and α_{2m+1} can be affected. Let us consider the case when x_1 is increased and let $L' = \langle \ell_1 + 1, \dots, \ell_{2m+1} \rangle$ be the list of lengths after increasing x_1. We have two possibilities:

- α_2 increases by 1:

$$
\begin{aligned}
\texttt{sum}(L', 2) &= \texttt{trunc}(\alpha_2 + 1, \texttt{sum}(L, 1)) + 2^{-\alpha_2 - 1} \\
&= \texttt{trunc}(\alpha_2, \texttt{sum}(L, 1)) + 2^{-\alpha_2 - 1} \\
&< \texttt{trunc}(\alpha_2, \texttt{sum}(L, 1)) + 2^{-\alpha_2} \\
&= \texttt{sum}(L, 2)
\end{aligned}
$$

- α_2 does not change: then $\texttt{sum}(L', 2) = \texttt{sum}(L, 2)$.

A similar reasoning holds for x_{2m+1} too. By denoting with $L' = \langle \ell_1, \dots, \ell_{2m+1} + 1 \rangle$ the list of lengths after increasing x_{2m+1}, we have two possibilities:

- α_{2m+1} increases by 1:

$$\begin{aligned}\mathtt{sum}(L', 2m+1) &= \mathtt{trunc}(\alpha_{2m+1}+1, \mathtt{sum}(L, 2m)) \\ &\quad + 2^{-(\alpha_{2m+1}+1)} \\ &\leq \mathtt{trunc}(\alpha_{2m+1}, \mathtt{sum}(L, 2m)) \\ &\quad + 2^{-(\alpha_{2m+1}+1)} + 2^{-(\alpha_{2m+1}+1)} \\ &= \mathtt{trunc}(\alpha_{2m+1}, \mathtt{sum}(L, 2m)) \\ &\quad + 2^{-\alpha_{2m+1}} \\ &= \mathtt{sum}(L, 2m+1)\end{aligned}$$

- α_{2m+1} does not change: then $\mathtt{sum}(L', 2m+1) = \mathtt{sum}(L, 2m+1)$.

Therefore, in such cases too, we have that $\mathtt{sum}(L', 2m+1) \leq \mathtt{sum}(L, 2m+1)$.

However, the above conclusions readily lead to a contradiction since, if we repeat the process iteratively for each x_i, we obtain that when $x_i = \ell_{i+1}$ for each $i \in \{1, 3, \ldots, 2m-1\}$, and $x_{2m+1} = \ell_{2m}$, it holds that $\mathtt{sum}(L', 2m+1) < 1$ and by Theorem 10 there exists an alphabetic codes with lengths L'. But, for such lengths L', we are in Case 1 and we know that this is not possible. Thus, even for values of x_i arbitrarily chosen, we have that $\mathtt{sum}(L, 2m+1) \geq 1$ and by Theorem 10 there does not exist any alphabetic code for the list of length L. □

A.3 PROOF OF LEMMA 3

Lemma. *Let $\ell_1, \ldots, \ell_m$ be a sequence of positive integers, and let $k \geq \max_i \ell_i$. The function* sum *on the list $L = \langle f_1, f_2, \ldots, f_{2m-1}\rangle = \langle \ell_1, k, \ell_2, k, \ldots, k, \ell_m\rangle$ of $2m-1$ elements satisfies the following inequality:*

$$\mathtt{sum}(L, 2m-1) \leq 2^{-\ell_1} + 2^{-\ell_m} + 2\sum_{i=2}^{m-1} 2^{-\ell_i}. \tag{A.3}$$

Proof. By the definition of number α_i (see Definition 8), it follows that

$$\alpha_2 = \min(f_1, f_2) = \ell_1,$$

and

$$\alpha_{2m-1} = \min(f_{2m-2}, f_{2m-1}) = \ell_m.$$

Similarly, for each $i \in \{3, 5, \dots, 2m-3\}$, it follows that

$$\alpha_i = \min(f_{i-1}, f_i) = \min(f_i, f_{i+1}) = \alpha_{i+1} = \ell_{\frac{i+1}{2}}.$$

Furthermore, by the definition of the functions $\texttt{sum}(\cdot,\cdot)$ and $\texttt{trunc}(\cdot,\cdot)$, for each index i it holds that

$$\begin{aligned}\texttt{sum}(L, i) &= \texttt{trunc}(\alpha_i, \texttt{sum}(L, i-1)) + 2^{-\alpha_i} \\ &\leq \texttt{sum}(L, i-1) + 2^{-\alpha_i}.\end{aligned}$$

Thus, given that $\texttt{sum}(L, 1) = 0$, iteratively unrolling $\texttt{sum}(L, 2m-1)$ yields to

$$\begin{aligned}\texttt{sum}(L, 2m-1) &\leq \sum_{i=2}^{2m-1} 2^{-\alpha_i} \\ &= 2^{-\ell_1} + 2^{-\ell_m} + 2\sum_{i=2}^{m-1} 2^{-\ell_i}.\end{aligned}$$

□

A.4 PROOF OF COROLLARY 13.1

Corollary. *For any dyadic probability distribution $\phi = \langle \phi_1, \dots, \phi_m \rangle$ on the symbols $s_1 \prec \cdots \prec s_m$, we can construct in linear time an alphabetic code C whose average length satisfies*

$$\begin{aligned}\mathbb{E}[C] &\leq H(\phi) + 2 - \phi_1 \left(2 - \log_2 \phi_1 - \lceil -\log_2 \phi_1 \rceil\right) \\ &\quad - \phi_m \left(2 - \log_2 \phi_m - \lceil -\log_2 \phi_m \rceil\right) - \sum_{i=1}^{m-1} \min(\phi_i, \phi_{i+1}) \\ &= H(\phi) + 2 - 2\phi_1 - 2\phi_m - \sum_{i=1}^{m-1} \min(\phi_i, \phi_{i+1}). \quad \text{(A.4)}\end{aligned}$$

Proof. Let $m \geq 4$. For a given dyadic probability distribution $\phi = \langle \phi_1, \dots, \phi_m \rangle$, let us define

$$\hat{\phi} = \langle \hat{\phi}_1, \hat{\phi}_2, \hat{\phi}_3 \dots, \hat{\phi}_m \rangle = \langle \phi_1, \phi_2 - \epsilon, \phi_3 + \epsilon, \phi_4, \dots, \phi_m \rangle \quad \text{(A.5)}$$

for a sufficiently small $\epsilon > 0$. Since $\hat{\phi}$ is a non-dyadic probability distribution, it follows from Theorem 12 that we can construct in linear time an alphabetic code C with

$$\mathbb{E}[C] \leq H(\hat{\phi}) + 2 - \hat{\phi_1} \left(2 - \log_2 \hat{\phi_1} - \lceil -\log_2 \hat{\phi_1} \rceil\right)$$

$$
\begin{aligned}
&- \hat{\phi}_m \left(2 - \log_2 \hat{\phi}_m - \lceil -\log_2 \hat{\phi}_m \rceil\right) - \sum_{i=1}^{m-1} \min(\hat{\phi}_i, \hat{\phi}_{i+1}) \\
&\leq H(\hat{\phi}) + 2 - \phi_1 \left(2 - \log_2 \phi_1 - \lceil -\log_2 \phi_1 \rceil\right) \\
&\quad - \phi_m \left(2 - \log_2 \phi_m - \lceil -\log_2 \phi_m \rceil\right) \\
&\quad - \min(\phi_1, \phi_2 - \epsilon) - \min(\phi_2 - \epsilon, \phi_3 + \epsilon) \\
&\quad - \min(\phi_3 + \epsilon, \phi_4) - \sum_{i=4}^{m-1} \min(\phi_i, \phi_{i+1}).
\end{aligned}
$$

Since the Shannon entropy $H(\cdot)$ and the operator $\min(\cdot, \cdot)$ are continuous functions for $\epsilon > 0$, by letting $\epsilon \to 0$ we obtain that

$$
\begin{aligned}
\mathbb{E}[C] &\leq H(\phi) + 2 - \phi_1 \left(2 - \log_2 \phi_1 - \lceil -\log_2 \phi_1 \rceil\right) \\
&\quad - \phi_m \left(2 - \log_2 \phi_m - \lceil -\log_2 \phi_m \rceil\right) - \sum_{i=1}^{m-1} \min(\phi_i, \phi_{i+1}).
\end{aligned}
$$

The probability distribution $\hat{\phi}$ is not equal to ϕ for $\epsilon > 0$. However, the code tree constructed for $\hat{\phi}$ can be used for ϕ, provided that $\epsilon > 0$ is sufficiently small. □

A.5 PROOF OF COROLLARY 13.2

Corollary. *For any probability distribution $\phi = \langle \phi_1, \ldots, \phi_m \rangle$ on the symbols $s_1 \prec \cdots \prec s_m$, with $\phi_1, \phi_2, \phi_{m-1}, \phi_m > 0$, we can construct in linear time an alphabetic code C with*

$$
\begin{aligned}
\mathbb{E}[C] \leq & H(\phi) + 2 - \phi_1 (2 - \log_2 \phi_1 - \lceil -\log_2 \phi_1 \rceil) \\
& - \phi_m (2 - \log_2 \phi_m - \lceil -\log_2 \phi_m \rceil) \\
& - \sum_{i=1}^{m-1} \min(\phi_i, \phi_{i+1}) \\
& - \phi_1 \max(0, \lceil -\log_2 \phi_1 \rceil - \lceil -\log_2 \phi_2 \rceil - 2) \\
& - \phi_m \max(0, \lceil -\log_2 \phi_m \rceil - \lceil -\log_2 \phi_{m-1} \rceil - 2).
\end{aligned} \tag{A.6}
$$

Proof. We begin by assuming first that ϕ is a non-dyadic probability distribution. The core idea of the proof is to improve the analysis of Theorem 12 by leveraging specific knowledge about the codeword lengths for the first and last two symbols Indeed, such knowledge in these two particular cases allows us to state

more precisely how many levels we bump up the first and last symbol during the construction of the final alphabetic code. We begin by considering the same code C, constructed in Theorem 12.

Let $\ell_1 = \lceil -\log_2 \phi_1 \rceil$ and $\ell_2 = \lceil -\log_2 \phi_2 \rceil + 1$ be the lengths of the codewords associated with symbols s_1 and s_2, according to (3.17). When the level of the codeword associated to s_1 exceeds the level of the codeword of s_2 by *more* than one unit, that is when $\ell_1 - \ell_2 \geq 2$, it holds that for *each* of these additional levels, there would have been a redundant internal node having only one child in the tree representation of the code. However, these nodes are removed since the intermediate code C' constructed through Lemma 2 is full in its tree representation. The elimination of such nodes bumps up the symbol s_1, since the nodes would have been on the path from the lowest common ancestor of s_1 and s_2 to s_1. Thus, when $\ell_1 - \ell_2 - 1 > 0$ we can bump up ϕ_1 of at least $\ell_1 - \ell_2 - 1$ additional levels. This explains the $-\phi_1 \max(0, \lceil -\log_2 \phi_1 \rceil - \lceil -\log_2 \phi_2 \rceil - 2)$ term in the bound.

A similar reasoning can be made with s_m and s_{m-1}, where $\ell_m = \lceil -\log_2 \phi_m \rceil$ and $\ell_{m-1} = \lceil -\log_2 \phi_{m-1} \rceil + 1$. Note that such reasoning cannot be applied to the other pairs of probabilities, since the knowledge of the codeword lengths alone is not enough to determine which of the symbols will get bumped up. We need to examine the entire set of codewords to gather sufficient information.

The bound (A.6) can be extended to dyadic probability distributions proceeding as in Corollary 13.1. □

B

CHAPTER 4 - ALPHABETIC AND PREFIX CODES WITH ASYMMETRIC SYMBOL COSTS

B.1 PROOF OF LEMMA 5

Lemma. *For each* $0 \leq w \leq D$*, our function* C *defined in* (4.3) *satisfies the Quadrangle Inequality, that is,* $\forall 1 \leq i \leq i' \leq j \leq j' \leq n$*, it holds that*

$$C(i,j,w) + C(i',j',w) \leq C(i,j',w) + C(i',j,w). \tag{B.1}$$

Proof. We need to show that the function C is a special case of (4.5). For such a purpose, let $w(i) = 0$ for each $i = 1, \dots, n$, and let $a = b = 1$. While as functions f and g we take $f(x) = x - \alpha$ and $g(x) = x - \beta$. Finally, we define the function $h(i,k,j) = \mathrm{c}(i,j)$ for each $i \leq k < j$, where $\mathrm{c}(i,j) = p_i + \cdots + p_j$. One can see that the function h defined in this way satisfies the conditions (4.6-4.9). In fact, for $i \leq j \leq t < s \leq l$ and $i \leq s$, if $t \leq k$ it holds that

$$h(i,t,s) - h(j,t,s) + h(j,k,l) - h(i,k,l) \tag{B.2}$$
$$= \mathrm{c}(i,s) - \mathrm{c}(j,s) + \mathrm{c}(j,l) - \mathrm{c}(i,l) = 0 \tag{B.3}$$

and

$$h(j,k,l) - h(i,k,l) = \mathrm{c}(j,l) - \mathrm{c}(i,l) < 0. \tag{B.4}$$

Similarly, if $k < t$, it holds that

$$h(j,t,l) - h(j,t,s) + h(i,k,s) - h(i,k,l) \tag{B.5}$$
$$= \mathrm{c}(j,l) - \mathrm{c}(j,s) + \mathrm{c}(i,s) - \mathrm{c}(i,l) = 0 \tag{B.6}$$

and

$$h(i,k,s) - h(i,k,l) = \mathrm{c}(i,s) - \mathrm{c}(i,l) < 0. \tag{B.7}$$

Since we have shown that the function C is a special case of (4.5), from Lemma 4 we get that the function C satisfies the Quadrangle Inequality. □

B.2 PROOF OF THEOREM 16

Theorem. *Let $L = \langle \ell_1, \dots, \ell_n \rangle$ be a list of integers, associated with the ordered symbols $s_1 \prec \cdots \prec s_n$, and let D be a positive integer. There exists a 1-constrained alphabetic code for which each codeword contains at most D ones and for which the codeword assigned to symbol s_i has length ℓ_i, for each $i = 1, \dots, n$, if and only if $\mathtt{sum}_D(L, n) < 1$.*

Proof. We have to show that whenever a 1-constrained alphabetic code with codeword lengths L exists, then $\mathtt{sum}_D(L, n) < 1$. Conversely, whenever a list of integers L satisfies $\mathtt{sum}_D(L, n) < 1$, then there exists a 1-constrained alphabetic code whose codeword lengths are exactly L.

Let us start by considering the first implication. Let $L = \langle \ell_1, \dots, \ell_n \rangle$ be the codeword lengths of a 1-constrained alphabetic code for which each codeword contains at most D ones, and let $w_1, \dots, w_n$ be the codewords associated with the symbols $s_1, \dots, s_n$. Furthermore, given a codeword w, let $v(w)$ denote the decimal value of the binary expansion that the codeword w represents. Since for any alphabetic code $v(w_i) < 1$ ([115]), to prove the implication, we need to show that $\mathtt{sum}_D(L, n) \leq v(w_n) < 1$. Specifically, we prove by induction that $\mathtt{sum}_D(L, i) \leq v(w_i)$ for $i = 1, \dots, n$.

For $i = 1$, since $sum_D(L, 1) = 0 \leq v(w_1)$, the inequality trivially holds.

Suppose it holds for i. Then, we have two cases:

- $\ell_i < \ell_{i+1}$: in this case $\alpha_{i+1} = \ell_i$, thus we have that

$$\begin{aligned}
\mathtt{sum}_D(L, i+1) &= \phi_D(\ell_{i+1}, \mathtt{trunc}(\alpha_{i+1}, \mathtt{sum}_D(L, i)) + 2^{-\alpha_{i+1}}) \\
&= \phi_D(\ell_{i+1}, \mathtt{trunc}(\ell_i, \mathtt{sum}_D(L, i)) + 2^{-\ell_i}) \\
&\leq \phi_D(\ell_{i+1}, \mathtt{trunc}(\ell_i, v(w_i)) + 2^{-\ell_i}) \\
&\quad \text{(by inductive hypothesis)} \\
&= \phi_D(\ell_{i+1}, v(w_i) + 2^{-\ell_i}) \\
&\leq \phi_D(\ell_{i+1}, v(w_{i+1})) \\
&\quad \text{(since } w_i \text{ and } w_{i+1} \text{ differ on the first } \ell_i \text{ bits)} \\
&= v(w_{i+1}). \\
&\quad \text{(since } w_{i+1} \text{ contains at most } D \text{ ones)}
\end{aligned}$$

- $\ell_i \geq \ell_{i+1}$: in this case $\alpha_{i+1} = \ell_{i+1}$, thus we get that

$$\begin{aligned}
\mathtt{sum}_D(L, i+1) &= \phi_D(\ell_{i+1}, \mathtt{trunc}(\alpha_{i+1}, \mathtt{sum}_D(L, i)) + 2^{-\alpha_{i+1}}) \\
&= \phi_D(\ell_{i+1}, \mathtt{trunc}(\ell_{i+1}, \mathtt{sum}_D(L, i)) + 2^{-\ell_{i+1}}) \\
&\leq \phi_D(\ell_{i+1}, \mathtt{trunc}(\ell_{i+1}, v(w_i)) + 2^{-\ell_{i+1}}) \\
&\quad \text{(by inductive hypothesis)} \\
&\leq \phi_D(\ell_{i+1}, v(w_{i+1})) \\
&\quad \text{(since } w_i \text{ and } w_{i+1} \text{ differ on the first } \ell_{i+1} \text{ bits)} \\
&= v(w_{i+1}). \\
&\quad \text{(since } w_{i+1} \text{ contains at most } D \text{ ones)}
\end{aligned}$$

This proves that $\mathtt{sum}_D(L, i) \leq v(w_i)$ for $i = 1, \ldots, n$; therefore, for any 1-constrained alphabetic code, it holds that $\mathtt{sum}_D(L, n) \leq v(w_n) < 1$.

Let us prove the other implication, i.e., if a list of integers $L = \langle \ell_1, \ldots, \ell_n \rangle$ satisfies $\mathtt{sum}_D(L, n) < 1$, then there exists a 1-constrained alphabetic code in which each codeword contains at most D ones and whose codeword lengths are exactly L. We prove that when $\mathtt{sum}_D(L, n) < 1$, we can construct a 1-constrained alphabetic code with codeword lengths L. We utilize the same method of Nakatsu [115] to construct the code: for every $i = 1, \ldots, n$, we assign the first ℓ_i bits of the binary expansion of $\mathtt{sum}_D(L, i)$ as the codeword for s_i. We need to show that the code constructed in this way is prefix, alphabetic and satisfies the constraint on the number of ones per codeword. The latter two properties are satisfied by construction. Specifically, the strictly increasing nature of the values $\mathtt{sum}_D(L, i)$ ensures the code is alphabetic, while the definition of $\mathtt{sum}_D$ guarantees that the first ℓ_i bits of each $\mathtt{sum}_D(L, i)$ contain at most D ones for each $i = 1, \ldots, n$. It remains to prove that the code is prefix. We observe that because the code is alphabetic (lexicographically ordered), if a codeword w_i were a prefix of w_j for any $i < j$, then w_i would necessarily be a prefix of all intermediate codewords w_k where $i < k < j$. Consequently, the prefix condition for the entire set is guaranteed if the condition holds for every pair of consecutive codewords. To this end, we observe that when $\ell_i \geq \ell_{i+1}$, the definition of $\mathtt{sum}_D$ implies that $\mathtt{sum}_D(L, i)$ and $\mathtt{sum}_D(L, i+1)$ have at least one different bit among their first ℓ_{i+1} bits. Similarly, when $\ell_i < \ell_{i+1}$, it follows that $\mathtt{sum}_D(L, i)$ and $\mathtt{sum}_D(L, i+1)$ have at least

one different bit among their first ℓ_i bits. In both cases, neither the codeword for s_i nor that for s_{i+1} is a prefix of the other. Therefore, the code is prefix. This concludes the proof. □

B.3 PROOF OF LEMMA 6

Lemma. *Let $\{\ell_1, \dots, \ell_n\}$ be a multiset of positive integers, D a positive integer and $L = \max_i \ell_i$. Let N_j count the number of integers in $\{\ell_1, \dots, \ell_n\}$ that are equal to j, for $j = 1, \dots, L$. Then, there exists a 1-constrained binary prefix code with codeword lengths $\ell_1, \dots, \ell_n$, such that each codeword does not contain more than D ones,* if *the following inequalities hold:*

$$N_j \leq \sum_{i=0}^{D-1} \binom{j}{i} + \binom{j-1}{D-1} - M_{j+1}, \quad \forall j = 1, \dots, L, \text{ where} \tag{B.8}$$

$$M_{j+1} = \begin{cases} 0 & \text{if } j \geq L, \\ \left\lceil \frac{N_{j+1} + M_{j+2}}{2} \right\rceil & \text{if } 1 \leq j < L. \end{cases} \tag{B.9}$$

Proof. As per established convention, we assume that $\binom{j}{i} = 0$ if $i > j$. Let T be the binary tree of maximum depth L such that: 1) T is full, 2) T has the maximum number of nodes, and 3) any root-to-leaf path of T has at most D right-pointing edges

We label each left-pointing edge of T with 0 and each right-pointing edge with 1. The set of all binary words one gets by reading the sequences of binary labels from the root of T to each leaf gives a binary prefix code P in which $|w|_1 \leq D$, for each $w \in P$, i.e., P is a 1-constrained binary prefix code. We now show that the number of nodes at the level j of such a tree T is exactly equal to

$$\sum_{i=0}^{D-1} \binom{j}{i} + \binom{j-1}{D-1} \quad \forall j = 1, \dots, L. \tag{B.10}$$

First of all, we recall that each path in the tree can be seen as a binary string. Therefore, (B.10) holds because the first term $\sum_{i=0}^{D-1} \binom{j}{i}$ counts the number of binary strings of length j, each

containing at most $D-1$ ones. The number of binary strings of length j containing exactly D ones that one obtains, by reading the root-to-leaves path in T is equal to $\binom{j-1}{D-1}$. This is true because any binary string of length j that one obtains by reading the root-to-leaves path in T *and* that contains D ones must end with 1. In fact, let us consider a binary string of length j that contains exactly D ones and does not end with 1. In the path from the root to the node associated with such a binary string, there will be at least one internal node that has only exactly *one* child. Therefore, such a node cannot belong to the binary tree T since, by hypothesis, T is full.

We show that for an arbitrary list of positive integers $\ell_1, \ldots, \ell_n, D$ that satisfy the inequalities (B.8), there is a prefix code C with $\ell_1, \ldots, \ell_n$ as codeword lengths, obeying the constraint that $|w|_1 \leq D$, for each $w \in C$. The idea of the proof is to build the code by taking as codewords the binary sequences one gets by following the paths from the root to nodes of T of levels $\ell_1, \ldots, \ell_n$, starting from level $L = \max_i \ell_i$ (i.e., by constructing the longest codewords first). From (B.8) we know that

$$N_L \leq \sum_{i=0}^{D-1} \binom{L}{i} + \binom{L-1}{D-1}. \tag{B.11}$$

Hence, there are enough available nodes in the level L of T to construct N_L codewords of length L. In particular, to construct such codewords one can choose the first N_L nodes, in sequence, starting from the left-most one. The constraint that the obtained codewords w's have to satisfy (i.e., that $|w|_1 \leq D$) is automatically satisfied since in the tree T any root-to-leaf path has at most D right-pointing edges, each labeled with 1.

The nodes at level $L-1$ that are parents of the chosen nodes at level L *cannot* be used anymore in the construction of the code, to comply with the prefix constraint of C. Therefore, we need to ignore such nodes at level $L-1$ and their number is exactly $M_L = \lceil N_L/2 \rceil$. Thus, the number of nodes at level $L-1$ that can be still used is

$$\sum_{i=0}^{D-1} \binom{L-1}{i} + \binom{L-2}{D-1} - M_L. \tag{B.12}$$

From (B.8) and (B.9) we know that

$$N_{L-1} \leq \sum_{i=0}^{D-1} \binom{L-1}{i} + \binom{L-2}{D-1} - M_L, \tag{B.13}$$

therefore we can iterate the same process to construct the desired number of codewords of length $L-1$. We choose the first N_{L-1} available nodes at level $L-1$ (that are all the nodes at level $L-1$ that give rise to non-prefixes of the chosen nodes at level L) in sequence, starting from the left-most one. Again, we have to ignore all the nodes at level $L-2$ that are ancestors of the chosen nodes at level L and $L-1$. This number is exactly

$$M_{L-1} = \left\lceil \frac{N_{L-1} + M_L}{2} \right\rceil. \tag{B.14}$$

From (B.8) and (B.9) we know that

$$N_j \leq \sum_{i=0}^{D-1} \binom{j}{i} + \binom{j-1}{D-1} - M_{j+1}, \quad \forall j = 1, \dots, L. \tag{B.15}$$

We can iterate this operation for each level of T. Thus, we can construct a prefix code with codeword lengths $\ell_1, \dots, \ell_n$ in which each codeword does not contain more than D ones, i.e., an 1-constrained prefix code with codeword lengths $\ell_1, \dots, \ell_n$. □

B.4 PROOF OF LEMMA 7

Lemma 16. *Let $\ell_1, \dots, \ell_n$ be the codeword lengths of a 1-constrained prefix code C in which each codeword does not contain more than D ones and whose tree representation corresponds to a full binary tree, i.e., for which $\sum_{i=1}^{n} 2^{-\ell_i} = 1$. Then the following inequalities hold:*

$$N_j \leq \sum_{i=0}^{D-1} \binom{j}{i} + \binom{j-1}{D-1} - M_{j+1}, \quad \forall j = 1, \dots, L, \tag{B.16}$$

where M_j and N_j are defined as in Lemma 6 and $L = \max_i \ell_i$.

Proof. First of all, let us recall that

$$\sum_{i=0}^{D-1} \binom{j}{i} + \binom{j-1}{D-1} \tag{B.17}$$

is the *maximum* number of nodes that a complete binary tree (in which any root-to-leaf path has at most D right-pointing edges) can have at level $j = 1, \dots, L$. Moreover, since we are considering only full binary trees, one can see that M_j, for each $j = 2, \dots, L$, corresponds to the *minimum* number of nodes at level $j - 1$ that are necessarily prefixes of some nodes associated with codewords of length $\geq j$, regardless of how such nodes are chosen. Putting all together, since by hypothesis, $\ell_1, \dots, \ell_n$ are the codeword lengths of a prefix code C in which each codeword does not contain more than D ones and whose binary tree representation is full, we have that the number of leaves at level j in the binary tree of C, which is N_j, *cannot* be greater than

$$\sum_{i=0}^{D-1} \binom{j}{i} + \binom{j-1}{D-1} - M_{j+1}, \tag{B.18}$$

for each $j = 1, \dots, L$. This holds because (B.18) is obtained by subtracting from the total number of nodes at level j (counted by (B.17)) the minimum number of nodes at level $j - 1$ that are necessarily prefixes of some nodes associated with codewords of length $\geq j$ (counted by M_{j+1}). In other words, (B.18) represents the maximum number of leaves at level $j = 1, \dots, L$, that a complete binary tree — where any root-to-leaf path has at most D right-pointing edges — has still "available" (i.e., they are not prefixes of the nodes associated with codewords of length $\geq j$). □

B.5 PROOF OF THEOREM 17

Theorem. *Let $S = \{s_1, \dots, s_n\}$ be a set of symbols and $p = \langle p_1, \dots, p_n \rangle$ a probability distribution on S with $p_1 \leq \cdots \leq p_n$. There exists an optimal 1-constrained prefix tree/code T^* for S, in which each codeword does not contain more than $\log_2 n$ ones.*

Proof. We prove the theorem by contradiction.

Let T^* be an optimal prefix tree for S whose leaves are ordered from left to right in a non-increasing fashion with respect to their codeword lengths. We observe that an optimal prefix code with leaves ordered in such a way always exists (see Theorem 9). Let us assume that there exists a codeword v in T^* that contains more

than $\log_2 n > \lfloor \log_2 n \rfloor$ ones, i.e., in the path from the root to the leaf associated with v there are more than $\lfloor \log_2 n \rfloor$ right edges. But since by optimality T^* is a full binary tree and its leaves are ordered in non-increasing fashion from left to right, it follows that to each of these right edges corresponds a full sub-tree whose leaves are at depth at least $\lfloor \log_2 n \rfloor + 1$ in the tree. Therefore, there must be at least $2^{\lfloor \log_2 n \rfloor + 1} > n$ leaves in the tree. However, this is not possible, since T^* contains n leaves. Thus, no codeword in T^* can contain more than $\log_2 n$ ones. This concludes the proof. □

B.6 PROOF OF THEOREM 18

Theorem. *For $D = \lceil \log_2 n \rceil$, our Kraft-like condition defined in (4.20) is equivalent to the classical Kraft-inequality.*

Proof. To prove the equivalence, we have to show that our condition implies the Kraft inequality and vice-versa. Since our condition is sufficient and necessary only if we consider the lengths of a full binary prefix tree, for a meaningful comparison we will assume that the lengths $\ell_1, \dots, \ell_n$ correspond to the codeword lengths of a full binary prefix tree and thus satisfy the Kraft inequality with equality, i.e., $\sum_{i=1}^{n} 2^{-\ell_i} = 1$. We also note that when we are dealing with full binary trees, the values M_j, defined in (B.9), for $j = 1, \dots, L-1$ (where L is the maximum length), are precisely equal to $\frac{N_j + M_{j+1}}{2}$. This holds because if $N_j + M_{j+1}$ were not even, it would imply the existence of an internal node at level $j-1$ with only one child, which is not possible in a full binary tree. Let us now prove the equivalence.

($\Rightarrow$) This direction is relatively straightforward because our condition represents a special case of the classical one. However, for the sake of completeness, we provide a direct proof.

Assume our Kraft-like condition holds, that is,

$$N_j \leq \sum_{i=0}^{\lceil \log_2 n \rceil - 1} \binom{j}{i} + \binom{j+1}{\lceil \log_2 n \rceil - 1} - M_{j+1}, \ \forall j = 1, \dots, L. \quad \text{(B.19)}$$

Since we are dealing with lengths of a full binary tree, we have

$$N_1 = 2^1 - M_2 = 2^1 - \frac{N_2 + M_3}{2} \quad \text{(B.20)}$$

$$= 2^1 - \frac{N_2 + \frac{N_3+M_4}{2}}{2} \tag{B.21}$$

$$= 2^1 - \frac{N_2 + \frac{N_3+\frac{N_4+M_5}{2}}{2}}{2} \tag{B.22}$$

$$= \cdots = 2^1 - \frac{2^{L-2}N_2 + 2^{L-3}N_3 + \cdots + 2^{L-L}N_L}{2^{L-1}}, \tag{B.23}$$

from which it holds that

$$\sum_{i=1}^{n} 2^{L-\ell_i} = \sum_{i=1}^{L} 2^{L-i}N_i = 2^L. \tag{B.24}$$

Dividing by 2^L, formula (B.24) yields $\sum_{i=1}^{n} 2^{-\ell_i} = 1$. Thus, our condition implies the Kraft inequality.

($\Leftarrow$) Let us assume now that the classical Kraft inequality holds, that is, $\sum_{i=1}^{n} 2^{-\ell_i} = 1$, which can be rewritten as

$$\sum_{i=1}^{n} 2^{-\ell_i} = \sum_{i=1}^{L} N_i 2^{-i} = 1,$$

that yields

$$\sum_{i=1}^{L} N_i 2^{L-i} = 2^L.$$

We need to show above equality implies our condition (4.20). We note that, for each $j > \lceil \log_2 n \rceil$, the inequality

$$N_j \leq \sum_{i=0}^{\lceil \log_2 n \rceil - 1} \binom{j}{i} + \binom{j-1}{\lceil \log_2 n \rceil - 1} - M_{j+1}, \tag{B.25}$$

is satisfied since $\sum_{i=0}^{\lceil \log_2 n \rceil - 1} \binom{j}{i} + \binom{j-1}{\lceil \log_2 n \rceil - 1} \geq n$ and $N_j + M_{j+1} \leq n$ (because the number of nodes at level j cannot exceed the total number of symbols n). Thus, we only have to prove that the inequalities in (4.20) hold for $j \leq \lceil \log_2 n \rceil$. For each $j \leq \lceil \log_2 n \rceil$, we have

$$N_j = 2^j - \frac{\sum_{i \neq j} 2^{L-i}N_i}{2^{L-j}} \tag{B.26}$$

$$\leq 2^j - \frac{\sum_{i=j+1}^{L} 2^{L-i}N_i}{2^{L-j}} \tag{B.27}$$

$$= 2^j - \frac{N_{j+1} + \frac{N_{j+2} + \frac{N_{j+3} + \cdots}{2}}{2}}{2} \tag{B.28}$$

$$= 2^j - \frac{N_{j+1} + M_{j+2}}{2} = 2^j - M_{j+1} \tag{B.29}$$

$$= \sum_{i=0}^{\lceil \log_2 n \rceil - 1} \binom{j}{i} + \binom{j-1}{\lceil \log_2 n \rceil - 1} - M_{j+1}, \tag{B.30}$$

which concludes the proof. □

C

CHAPTER 5 - PREFIX CODES WITH A SPACE DELIMITER

C.1 PROOF OF LEMMA 9

Lemma. *Let $S = \{s_1, \dots, s_n\}$ be the set of source symbols, and let $p = (p_1, \dots, p_n)$, $p_1 \geq \dots \geq p_n > 0$, be a probability distribution on S. There exists an* optimal $(k+1)$-ary prefix code ending with a space *C such that for any internal node v (except the root) of the tree representation of C, if we denote by w the k-ary string associated with the node v, then the string $w\sqcup$ belongs to the codeword set of C.*

Proof. Let C be an arbitrary optimal $(k+1)$-prefix code ending with a space. Let us assume that, in the tree representation of C, there exists an internal node v whose associated string w is such that $w\sqcup$ *does not* belong to the codeword set of C. Since v is an internal node, there is at least a leaf x, which is a descendant of v, whose associated string is the codeword of some symbol s_j. We modify the encoding, by assigning the codeword $w\sqcup$ to the symbol s_j. The new encoding is still prefix, and its average length can only decrease since the length of the newly assigned codeword to s_j cannot be greater than the previous one. We can repeat the argument for all internal nodes that do not satisfy the property stated in the lemma to complete the proof. □

C.2 PROOF OF LEMMA 10

Lemma. *Let C be an arbitrary $(k+1)$-ary prefix code ending with a space, then the k-ary code D that one obtains from C by removing the space $\sqcup$ from each codeword of C is a* one-to-one *code.*

Proof. The proof is straightforward. Since C is prefix, it holds that, for any pair $s_i, s_j \in S$, with $s_i \neq s_j$, the codeword $C(s_i)$ is not a prefix of $C(s_j)$ and vice versa. Therefore, since D is obtained from C by removing the space, we have four cases:

1. $C(s_i) = D(s_i)$ and $C(s_j) = D(s_j)$: then $D(s_i) \neq D(s_j)$ since $C(s_i) \neq C(s_j)$;
2. $C(s_i) = D(s_i)\sqcup$ and $C(s_j) = D(s_j)\sqcup$: then $D(s_i) \neq D(s_j)$ since $C(s_i)$ is not a prefix of $C(s_j)$ and vice versa;
3. $C(s_i) = D(s_i)\sqcup$ and $C(s_j) = D(s_j)$: then $D(s_i) \neq D(s_j)$ since $C(s_j)$ is not a prefix of $C(s_i)$;
4. $C(s_i) = D(s_i)$ and $C(s_j) = D(s_j)\sqcup$: then $D(s_i) \neq D(s_j)$ since $C(s_i)$ is not a prefix of $C(s_j)$.

Therefore, for any pair $s_i, s_j \in S$, with $s_i \neq s_j$, $D(s_i) \neq D(s_j)$, and D is a one-to-one code. □

C.3 PROOF OF LEMMA 12

Lemma 17. *Let $S = \{s_1, \dots, s_n\}$ be the set of source symbols and $p = (p_1, \dots, p_n)$ be a probability distribution on S, with $p_1 \geq \dots \geq p_n$. The average length L_ϵ of an optimal k-ary one-to-one code that includes the empty word satisfies*

$$L_\epsilon > H_k(p) - (H_k(p) + \log_k(k-1)) \log_k \left(1 + \frac{1}{H_k(p) + \log_k(k-1)}\right) - \log_k(H_k(p) + \log_k(k-1) + 1),$$

where $H_k(p) = -\sum_{i=1}^{n} p_i \log_k p_i$.

Proof. The proof is an adaptation of Alon *et al.*'s proof [7] from the binary case to the $k \geq 2$-ary case.

We recall that the optimal one-to-one code (i.e., whose average length achieves the minimum L_ϵ) has codeword lengths ℓ_i given by:

$$\ell_i = \lfloor \log_k((k-1)i) \rfloor. \tag{C.1}$$

For each $j \in \{0, \dots, \lfloor \log_k n \rfloor\}$, let us define the quantities q_j as

$$q_j = \sum_{i=\frac{k^j-1}{k-1}+1}^{\frac{k^{j+1}-1}{k-1}} p_i.$$

It holds that $\sum_{j=0}^{\lfloor \log_k n \rfloor} q_j = 1$. Let Y be a random variable that takes values in $\{0, \dots, \lfloor \log_k n \rfloor\}$ according to the probability distribution $q = (q_0, \dots, q_{\lfloor \log_k n \rfloor})$, that is

$$\forall j \in \{0, \dots, \lfloor \log_k n \rfloor\} \quad \Pr\{Y = j\} = q_j.$$

From (C.1), we have

$$\begin{aligned} L_\epsilon &= \sum_{i=1}^{n} \lfloor \log_k((k-1)i) \rfloor p_i \\ &= \sum_{j=0}^{\lfloor \log_k n \rfloor} \sum_{i=\frac{k^j-1}{k-1}+1}^{\frac{k^{j+1}-1}{k-1}} \lfloor \log_k((k-1)i) \rfloor p_i \\ &= \sum_{j=0}^{\lfloor \log_k n \rfloor} j q_j = \mathbb{E}[Y]. \end{aligned} \tag{C.2}$$

By applying the entropy grouping rule ([35], Ex. 2.27) to the distribution p, we obtain

$$\begin{aligned} H_2(p) &= H_2(q) + \sum_{j=0}^{\lfloor \log_k n \rfloor} q_j H_2\left(\frac{p_{\frac{k^j-1}{k-1}+1}}{q_j}, \dots, \frac{p_{\frac{k^{j+1}-1}{k-1}}}{q_j} \right) \\ &\le H_2(q) + \sum_{j=0}^{\lfloor \log_k n \rfloor} q_j \log_2 k^j \\ &\quad \left(\text{since } H_2\left(\frac{p_{\frac{k^j-1}{k-1}+1}}{q_j}, \dots, \frac{p_{\frac{k^{j+1}-1}{k-1}}}{q_j} \right) \le \log_2 k^j\right) \\ &= H_2(q) + \sum_{j=0}^{\lfloor \log_k n \rfloor} j q_j \log_2 k \\ &= H_2(q) + \mathbb{E}[Y] \log_2 k. \end{aligned} \tag{C.3}$$

We now derive an upper bound to $H_2(Y) = H_2(q)$ in terms of the expected value $\mathbb{E}[Y]$.

To this end, let us consider an auxiliary random variable Y' with the same distribution of Y, but with values ranging from 1 to $\lfloor \log_k(n) \rfloor + 1$ (instead of from 0 to $\lfloor \log_k(n) \rfloor$). It is easy to verify that $\mu = \mathbb{E}[Y'] = \mathbb{E}[Y] + 1$.

Let α be a positive number, whose value will be chosen later. We obtain that

$$\begin{aligned}
H_k(Y) - \alpha\mu &= \sum_{i=1}^{\lfloor \log_k(n) \rfloor + 1} q_{i-1} \log_k \frac{1}{q_{i-1}} - \alpha \sum_{j=1}^{\lfloor \log_k(n) \rfloor + 1} j q_{j-1} \\
&= \sum_{i=1}^{\lfloor \log_k(n) \rfloor + 1} q_{i-1} \log_k \frac{1}{q_{i-1}} + \sum_{j=1}^{\lfloor \log_k(n) \rfloor + 1} (-\alpha j) q_{j-1} \\
&= \sum_{i=1}^{\lfloor \log_k(n) \rfloor + 1} q_{i-1} \log_k \frac{1}{q_{i-1}} + \sum_{j=1}^{\lfloor \log_k(n) \rfloor + 1} q_{j-1} \log_k (k^{-\alpha j}) \\
&= \sum_{i=1}^{\lfloor \log_k(n) \rfloor + 1} q_{i-1} \log_k \frac{k^{-\alpha i}}{q_{i-1}} \\
&\leq \log_k \sum_{i=1}^{\lfloor \log_k(n) \rfloor + 1} k^{-\alpha i} \qquad \text{(by Jensen's inequality)} \\
&= \log_k \left[\left(\frac{1}{k^\alpha} \right) \left(\frac{1 - k^{-\alpha(\lfloor \log_k(n) \rfloor + 1)}}{1 - k^{-\alpha}} \right) \right] \\
&\leq \log_k \left(\frac{1 - k^{-\alpha(\log_k(n)+1)}}{k^\alpha - 1} \right) \\
&= \log_k \left(\frac{1 - (kn)^{-\alpha}}{k^\alpha - 1} \right).
\end{aligned}$$

By substituting $\log_k \frac{\mu}{\mu-1}$ with α in the obtained inequality

$$H_k(Y) \leq \alpha\mu + \log_k \left(\frac{1 - (kn)^{-\alpha}}{k^\alpha - 1} \right),$$

we obtain

$$H_k(Y) \leq \mu \log_k \frac{\mu}{\mu - 1} + \log_k(\mu - 1) + \log_k \left(1 - \left(\frac{1}{kn} \right)^{\log_k \frac{\mu}{\mu-1}} \right). \tag{C.4}$$

Since $\left(\frac{1}{kn}\right)^{\log_k \frac{\mu}{\mu-1}}$ is decreasing in μ, and because $\mu = \mathbb{E}[Y]+1 > 1$, we obtain:

$$H_k(Y) < \mathbb{E}[Y] \log_k \left(1 + \frac{1}{\mathbb{E}[Y]} \right) + \log_k(\mathbb{E}[Y] + 1). \tag{C.5}$$

By applying (C.5) to (C.3) and since $H_k(Y) = \frac{H_2(Y)}{\log_2 k}$, we obtain

$$H_2(p) < \mathbb{E}[Y] \log_2 k + \mathbb{E}[Y] \log_2 \left(1 + \frac{1}{E[Y]}\right) + \log_2(E[Y] + 1). \tag{C.6}$$

From (C.2), we have that $L_\epsilon = \mathbb{E}[Y]$; moreover, from the inequality (5.18) of Lemma 14, we know that

$$L_\epsilon \leq H_k(p) + \log_k(k-1). \tag{C.7}$$

Hence, since the function $f(z) = z \log_k \left(1 + \frac{1}{z}\right)$ is increasing in z, we can apply (C.7) to upper-bound the term

$$\mathbb{E}[Y] \log_2 \left(1 + \frac{1}{E[Y]}\right),$$

to obtain the following inequality:

$$\begin{aligned} H_2(p) < & L_\epsilon \log_2 k + (H_k(p) + \log_k(k-1)) \log_2 \left(1 + \frac{1}{H_k(p) + \log_k(k-1)}\right) \\ & + \log_2(H_k(p) + \log_k(k-1) + 1). \end{aligned} \tag{C.8}$$

Rewriting (C.8), we finally obtain

$$\begin{aligned} L_\epsilon > & H_k(p) - (H_k(p) + \log_k(k-1)) \log_k \left(1 + \frac{1}{H_k(p) + \log_k(k-1)}\right) \\ & - \log_k(H_k(p) + \log_k(k-1) + 1), \end{aligned}$$

and that concludes our proof. □

C.4 PROOF OF LEMMA 13

Lemma. *Let $S = \{s_1, \dots, s_n\}$ be the set of source symbols and $p = (p_1, \dots, p_n)$, $p_1 \geq \dots \geq p_n$, be a probability distribution on S. The average length L_ϵ of an optimal one-to-one code that includes the empty word satisfies the following:*

1. *If $0 < p_1 \leq 0.5$,*

$$L_\epsilon \geq H_k(p) - (H_k(p) - p_1 \log_k \frac{1}{p_1} + (1 - p_1) \log_k(k-1))$$

$$\log_k\left(1+\frac{1}{H_k(p)-p_1\log_k\frac{1}{p_1}+(1-p_1)\log_k(k-1)}\right)$$
$$-\log_k(H_k(p)-p_1\log_k\frac{1}{p_1}+(1-p_1)\log_k(k-1)+1)$$
$$-\log_k\left(1-\left(\frac{1}{kn}\right)^{\log_k\left(1+\frac{1}{1-p_1}\right)}\right), \tag{C.9}$$

2. ***if*** $0.5 < p_1 \leq 1$

$$L_\epsilon \geq H_k(p)-(H_k(p)-\mathcal{H}_k(p_1)+(1-p_1)(1+\log_k(k-1)))$$
$$\log_k\left(1+\frac{1}{H_k(p)-\mathcal{H}_k(p_1)+(1-p_1)(1+\log_k(k-1))}\right)$$
$$-\log_k(H_k(p)-\mathcal{H}_k(p_1)+(1-p_1)(1+\log_k(k-1))+1)$$
$$-\log_k\left(1-\left(\frac{1}{kn}\right)^{\log_k\left(1+\frac{1}{1-p_1}\right)}\right), \tag{C.10}$$

where $\mathcal{H}_k(p_1) = -p_1\log_k p_1-(1-p_1)\log_k(1-p_1)$ *is the k-ary binary Shannon entropy.*

Proof. The proof is the same as the proof of Lemma 12. However, we change two steps in the demonstration.

First, since

$$\left(\frac{1}{kn}\right)^{\log_k\frac{\mu}{\mu-1}}=\left(\frac{1}{kn}\right)^{\log_k\frac{\mathbb{E}[Y]+1}{\mathbb{E}[Y]}}=\left(\frac{1}{kn}\right)^{\log_k\left(1+\frac{1}{\mathbb{E}[Y]}\right)}$$

is decreasing in μ and $\mathbb{E}[Y] = L_\epsilon = 0p_1 + 1p_2 + \cdots \geq 1 - p_1$, we have

$$\log_k\left(1-\left(\frac{1}{kn}\right)^{\log_k\frac{\mu}{\mu-1}}\right)\leq\log_k\left(1-\left(\frac{1}{kn}\right)^{\log_k\left(1+\frac{1}{1-p_1}\right)}\right). \tag{C.11}$$

Hence, by applying (C.11) to the right-hand side of (C.4), we obtain

$$H_k(Y)\leq\mathbb{E}[Y]\log_k\left(1+\frac{1}{\mathbb{E}[Y]}\right)+\log_k(\mathbb{E}[Y]+1)$$
$$+\log_k\left(1-\left(\frac{1}{kn}\right)^{\log_k\left(1+\frac{1}{1-p_1}\right)}\right). \tag{C.12}$$

Now, by applying (C.12) (instead of (C.5)) to (C.3) and since $H_k(Y) = \frac{H_2(Y)}{\log_2 k}$, we obtain

$$H_2(p) \leq \mathbb{E}[Y]\log_2 k + \mathbb{E}[Y]\log_2\left(1+\frac{1}{E[Y]}\right) + \log_2(E[Y]+1) + \log_2\left(1-\left(\frac{1}{kn}\right)^{\log_k\left(1+\frac{1}{1-p_1}\right)}\right). \tag{C.13}$$

Here, instead of applying the upper bound:

$$L_\epsilon \leq H_k(p) + \log_k(k-1)$$

of Lemma 14 to the right-hand side of (C.13), we apply the improved version:

$$L_\epsilon \leq \begin{cases} H_k(p) - p_1\log_k\frac{1}{p_1} + (1-p_1)\log_k(k-1) \text{ if } 0 < p_1 \leq 0.5, \\ H_k(p) - \mathcal{H}_k(p_1) + (1-p_1)\log_k 2(k-1) \text{ if } 0.5 < p_1 \leq 1, \end{cases}$$

proven in Lemma 15. Then, we simply need to rewrite the inequality, concluding the proof. □

C.5 PROOF OF LEMMA 14

Lemma. *Let $S = \{s_1, \dots, s_n\}$ be the set of source symbols and $p = (p_1, \dots, p_n)$, $p_1 \geq \dots \geq p_n$, be a probability distribution on S. The average length L_ϵ of an optimal one-to-one code that includes the empty word satisfies*

$$L_\epsilon \leq H_k(p) + \log_k(k-1). \tag{C.14}$$

Proof. Under the standing hypothesis that $p_1 \geq \dots \geq p_n$, it holds that

$$p_i \leq \frac{1}{i} \qquad \forall i = 1, \dots, n. \tag{C.15}$$

We recall that the length of the i-th codeword of an optimal one-to-one code D that includes the empty word is equal to

$$\ell_i = \lfloor \log_k((k-1)i) \rfloor. \tag{C.16}$$

Therefore, from (C.15), we can upper bound each length ℓ_i as

$$\ell_i = \lfloor \log_k((k-1)i) \rfloor \leq \log_k((k-1)i) \leq \log_k(k-1) + \log_k \frac{1}{p_i}. \quad \text{(C.17)}$$

Hence, by applying (C.17) to the average length of D, one gets

$$L_\epsilon = \sum_{i=1}^{n} p_i \ell_i \leq \sum_{i=1}^{n} p_i \left(\log_k(k-1) + \log_k \frac{1}{p_i} \right) = H_k(p) + \log_k(k-1).$$

This concludes our proof. □

C.6 PROOF OF LEMMA 15

Lemma. *Let $S = \{s_1, \ldots, s_n\}$ be the set of source symbols and $p = (p_1, \ldots, p_n)$, $p_1 \geq \ldots \geq p_n$, be a probability distribution on S. The average length L_ϵ of an optimal one-to-one code that includes the empty word satisfies*

$$L_\epsilon \leq \begin{cases} H_k(p) - p_1 \log_k \frac{1}{p_1} + (1-p_1)\log_k(k-1) & \text{if } 0 < p_1 \leq 0.5, \\ H_k(p) - \mathcal{H}_k(p_1) + (1-p_1)\log_k 2(k-1) & \text{if } 0.5 < p_1 \leq 1, \end{cases} \quad \text{(C.18)}$$

where $\mathcal{H}_k(p_1) = -p_1 \log_k p_1 - (1-p_1)\log_k(1-p_1)$ is the k-ary binary Shannon entropy.

Proof. Let us prove first that the length of an optimal one-to-one code that includes the empty word satisfies the inequality:

$$L_\epsilon \leq \sum_{i=2}^{n} p_i \log_k(i(k-1)) - 0.5 \sum_{j \geq 2 : \frac{k^j-1}{k-1} \leq n} p_{\frac{k^j-1}{k-1}}. \quad \text{(C.19)}$$

By recalling that $\ell_1 = \lfloor \log_k(k-1) \rfloor = 0$, we can rewrite L_ϵ as follows:

$$
\begin{aligned}
L_\epsilon &= \sum_{i=2}^{n} p_i \lfloor \log_k(i(k-1)) \rfloor \\
&= \sum_{j \geq 1 : \frac{k^j-1}{k-1}+1 \leq n} \; \sum_{i=\frac{k^j-1}{k-1}+1}^{\min(\frac{k^{j+1}-1}{k-1}-1, n)} p_i \lfloor \log_k(i(k-1)) \rfloor \\
&\quad + \sum_{j \geq 2 : \frac{k^j-1}{k-1} \leq n} p_{\frac{k^j-1}{k-1}} \left\lfloor \log_k \left(\frac{k^j-1}{k-1}(k-1) \right) \right\rfloor \\
&= \sum_{j \geq 1 : \frac{k^j-1}{k-1}+1 \leq n} \; \sum_{i=\frac{k^j-1}{k-1}+1}^{\min(\frac{k^{j+1}-1}{k-1}-1, n)} p_i \lfloor \log_k(i(k-1)) \rfloor \\
&\quad + \sum_{j \geq 2 : \frac{k^j-1}{k-1} \leq n} p_{\frac{k^j-1}{k-1}} \log_k(k^j-1) \\
&\quad - \sum_{j \geq 2 : \frac{k^j-1}{k-1} \leq n} p_{\frac{k^j-1}{k-1}} (\log_k(k^j-1) - \lfloor \log_k(k^j-1) \rfloor) \\
&\leq \sum_{i=2}^{n} p_i \log_k(i(k-1)) - \sum_{j \geq 2 : \frac{k^j-1}{k-1} \leq n} p_{\frac{k^j-1}{k-1}} (\log_k(k^j-1) - \lfloor \log_k(k^j-1) \rfloor),
\end{aligned}
$$

where the last inequality holds since

$$
\begin{aligned}
&\sum_{j \geq 1 : \frac{k^j-1}{k-1}+1 \leq n} \; \sum_{i=\frac{k^j-1}{k-1}+1}^{\min(\frac{k^{j+1}-1}{k-1}-1, n)} p_i \lfloor \log_k(i(k-1)) \rfloor \leq \\
&\qquad \sum_{j \geq 1 : \frac{k^j-1}{k-1}+1 \leq n} \; \sum_{i=\frac{k^j-1}{k-1}+1}^{\min(\frac{k^{j+1}-1}{k-1}-1, n)} p_i \log_k(i(k-1)).
\end{aligned}
$$

We observe that the function $f(j) = \log_k(k^j-1) - \lfloor \log_k(k^j-1) \rfloor$ is increasing in j. Therefore, it reaches its minimum at $j=2$, where it takes the value

$$
\log_k(k^2-1) - \lfloor \log_k(k^2-1) \rfloor = 1 + \log_k \left(1 - \frac{1}{k^2}\right) > 0.5,
$$

for any $k \geq 2$. Thus, (C.19) holds as we claimed.

Let us now show that

$$L_\epsilon \leq H_k(p) - p_1 \log_k \frac{1}{p_1} + (1-p_1)\log_k(k-1) - 0.5 \sum_{j\geq 2:\frac{k^j-1}{k-1}\leq n} p_{\frac{k^j-1}{k-1}}. \tag{C.20}$$

Since the distribution p is ordered in a non-increasing fashion, from (C.15) and (C.19), we have that

$$\begin{aligned} L_\epsilon &\leq \sum_{i=2}^{n} p_i \log_k(i(k-1)) - 0.5 \sum_{j\geq 2:\frac{k^j-1}{k-1}\leq n} p_{\frac{k^j-1}{k-1}} \\ &\leq \sum_{i=2}^{n} p_i \log_k \frac{1}{p_i}(k-1) - 0.5 \sum_{j\geq 2:\frac{k^j-1}{k-1}\leq n} p_{\frac{k^j-1}{k-1}} \qquad (\text{since } i \leq \tfrac{1}{p_i}) \\ &= H_k(p) - p_1 \log_k \frac{1}{p_1} + (1-p_1)\log_k(k-1) - 0.5 \sum_{j\geq 2:\frac{k^j-1}{k-1}\leq n} p_{\frac{k^j-1}{k-1}}. \end{aligned}$$

Therefore, (C.20) holds.

To conclude the proof, it remains to prove that

$$L_\epsilon \leq H_k(p) - \mathcal{H}_k(p_1) + (1-p_1)(\log_k 2(k-1)) - 0.5 \sum_{j\geq 2:\frac{k^j-1}{k-1}\leq n} p_{\frac{k^j-1}{k-1}}. \tag{C.21}$$

By observing that for any $i \geq 2$, it holds that

$$p_i \leq \frac{2(1-p_1)}{i}, \tag{C.22}$$

we obtain:

$$\begin{aligned} L_\epsilon &\leq \sum_{i=2}^{n} p_i \log_k(i(k-1)) - 0.5 \sum_{j\geq 2:\frac{k^j-1}{k-1}\leq n} p_{\frac{k^j-1}{k-1}} \\ &\leq \sum_{i=2}^{n} p_i \log_k \left(\frac{2(1-p_1)}{p_i}(k-1)\right) - 0.5 \sum_{j\geq 2:\frac{k^j-1}{k-1}\leq n} p_{\frac{k^j-1}{k-1}} \\ &\qquad (\text{since from (C.22), we have } i \leq \tfrac{2(1-p_1)}{p_i}) \\ &= H_k(p) - p_1 \log_k \frac{1}{p_1} + (\log_k 2 + \log_k(1-p_1) \end{aligned}$$

$$
\begin{aligned}
&+ \log_k(k-1))(1-p_1) - 0.5 \sum_{j\geq 2:\frac{k^j-1}{k-1}\leq n} p_{\frac{k^j-1}{k-1}} \\
=&H_k(p) - \mathcal{H}_k(p_1) + (1-p_1)(\log_k 2(k-1)) - 0.5 \sum_{j\geq 2:\frac{k^j-1}{k-1}\leq n} p_{\frac{k^j-1}{k-1}}.
\end{aligned}
$$

Therefore, (C.21) holds as well.

From (C.20) and (C.21), since

$$
\sum_{j\geq 2:\frac{k^j-1}{k-1}\leq n} p_{\frac{k^j-1}{k-1}} \geq 0,
$$

one gets

$$
L_\epsilon \leq H_k(p) - p_1 \log_k \frac{1}{p_1} + (1-p_1)\log_k(k-1),
$$

and

$$
L_\epsilon \leq H_k(p) - \mathcal{H}_k(p_1) + (1-p_1)(\log_k 2(k-1)).
$$

Now, one can see that $p_1 \log_k \frac{1}{p_1} \geq \mathcal{H}_k(p_1) + (1-p_1)\log_k 2$ for $0 < p_1 \leq 0.5$, concluding the proof. □

PUBLICATIONS

[1] Roberto Bruno. "Hardness and Approximability of Dimension Reduction on the Probability Simplex." In: *Algorithms* 17.7 (2024), p. 296.

[2] Roberto Bruno, Roberto De Prisco, Alfredo De Santis, and Ugo Vaccaro. "Bounds and Algorithms for Alphabetic Codes and Binary Search Trees." In: *IEEE Transactions on Information Theory* 70.10 (2024), pp. 6974–6988.

[3] Roberto Bruno, Roberto De Prisco, and Ugo Vaccaro. "Old and New Results on Alphabetic Codes." In: *Information Theory and Related Fields: Festschrift in Memory of Ning Cai*. Ed. by Christian Deppe, Andreas Winter, Raymond W. Yeung, Holger Boche, Ingo Althöfer, Jens Stoye, Ulrich Tamm, and Rami Ezzine. Cham: Springer Nature Switzerland, 2025, pp. 151–194. ISBN: 978-3-031-82014-4. DOI: 10.1007/978-3-031-82014-4_7. URL: https://doi.org/10.1007/978-3-031-82014-4_7.

[4] Roberto Bruno, Roberto De Prisco, and Ugo Vaccaro. "Optimal Binary Variable-Length Codes with a Bounded Number of 1's Per Codeword: Design, Analysis, and Applications." In: *2025 IEEE International Symposium on Information Theory (ISIT)*. 2025, pp. 1–6. DOI: 10.1109/ISIT63088.2025.11195687.

[5] Roberto Bruno, Roberto De Prisco, and Ugo Vaccaro. "Optimal Binary Variable-Length Codes with a Bounded Number of 1's per Codeword: Design, Analysis, and Applications." In: *arXiv preprint arXiv:2501.11129* (2025).

[6] Roberto Bruno and Ugo Vaccaro. "A note on equivalent conditions for majorization." In: *AIMS Mathematics* 9.4 (2024), pp. 8641–8660.

[7] Roberto Bruno and Ugo Vaccaro. "Entropic Bounds on the Average Length of Codes with a Space." In: *Entropy*

26.4 (2024). ISSN: 1099-4300. DOI: 10.3390/e26040283. URL: https://www.mdpi.com/1099-4300/26/4/283.

[8] Roberto Bruno and Ugo Vaccaro. "NP-Hardness and Approximation Algorithms for Constrained Entropy Maximization." In: *IEEE Transactions on Information Theory* 72.2 (2026), pp. 832–843.

BIBLIOGRAPHY

[1] Julia Abrahams. "Codes with monotonic codeword lengths." In: *Information Processing & Management* 30.6 (1994), pp. 759–764.

[2] Julia Abrahams. "Parallelized huffman and hu-tucker searching." In: *IEEE Transactions on Information Theory* 40.2 (1994), pp. 508–510.

[3] Julia Abrahams. "Code and parse trees for lossless source encoding." In: *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171)* (1997), pp. 145–171.

[4] Rudolf Ahlswede and Ning Cai. "A kraft–type inequality for d–delay binary search codes." In: *General Theory of Information Transfer and Combinatorics*. Springer, 2006, pp. 704–706.

[5] Rudolf Ahlswede and Ingo Wegener. *Search problems*. John Wiley & Sons, Inc., 1987.

[6] Martin Aigner. *Combinatorial search*. John Wiley & Sons, Inc., 1988.

[7] Noga Alon and Alon Orlitsky. "A lower bound on the expected length of one-to-one codes." In: *IEEE Transactions on Information Theory* 40.5 (1994), pp. 1670–1672.

[8] Andris Ambainis, Stephen A. Bloch, and David L. Schweizer. "Delayed binary search, or playing Twenty Questions with a procrastinator." In: *Proceedings of 10th AMC SIAM Symposium on Discrete Algorithms (SODA)*. 1999, pp. 844–845.

[9] Shoshana Anily and Refael Hassin. "Ranking the best binary trees." In: *SIAM Journal on Computing* 18.5 (1989), pp. 882–892.

[10] Shymaa M. Arafat. "An encryption algorithm based on alphabetic trees." In: *The 3rd ACS/IEEE International Conference on Computer Systems and Applications*. IEEE. 2005, p. 92.

[11] Michael B. Baer. "Prefix codes for power laws." In: *2008 IEEE International Symposium on Information Theory*. IEEE. 2008, pp. 2464–2468.

[12] Michael B. Baer. "Alphabetic coding with exponential costs." In: *Information Processing Letters* 110.4 (2010), pp. 139–142.

[13] Arye Barkan and Haim Kaplan. "Partial alphabetic trees." In: *Journal of algorithms* 58.2 (2006), pp. 81–103.

[14] Frédérique Bassino, Marie-Pierre Béal, and Dominique Perrin. "A Finite State Version of the Kraft–McMillan Theorem." In: *SIAM Journal on Computing* 30.4 (2000), pp. 1211–1230.

[15] Ahmed A. Belal, Mohamed S. Selim, and Shymaa M. Arafat. "Building optimal alphabetic trees recursively." In: *WSEAS Transactions Mathematics* 1.1 (2002), pp. 77–82.

[16] Irad Ben-Gal. "An upper bound on the weight-balanced testing procedure with multiple testers." In: *IIE Transactions* 36.5 (2004), pp. 481–493.

[17] Jon Louis Bentley and Donna J. Brown. "A general class of resource tradeoffs." In: *Journal of Computer and System Sciences* 25.2 (1982), pp. 214–238.

[18] Jon Louis Bentley and James B. Saxe. "Decomposable searching problems I. Static-to-dynamic transformation." In: *Journal of Algorithms* 1.4 (1980), pp. 301–358. ISSN: 0196-6774. DOI: https://doi.org/10.1016/0196-6774(80)90015-2. URL: https://www.sciencedirect.com/science/article/pii/0196677480900152.

[19] Richard S. Bird. "An optimal, purely functional implementation of the Garsia–Wachs algorithm." In: *Journal of Functional Programming* 30 (2020), e3.

[20] Ian Blanes, Miguel Hernández-Cabronero, Joan Serra-Sagristà, and Michael W. Marcellin. "Lower bounds on the redundancy of Huffman codes with known and unknown probabilities." In: *IEEE Access* 7 (2019), pp. 115857–115870.

[21] Carlo Blundo and Roberto De Prisco. "New bounds on the expected length of one-to-one codes." In: *IEEE Transactions on Information Theory* 42.1 (1996), pp. 246–250.

[22] Al Borchers and Prosenjit Gupta. "Extending the quadrangle inequality to speed-up dynamic programming." In: *Information Processing Letters* 49.6 (1994), pp. 287–290.

[23] Bella Bose and Luca G. Tallini. "On the proper enumeration of all finite length strings for source coding." In: *Kuwait Journal of Science* 53.2 (2026), p. 100537. DOI: https://doi.org/10.1016/j.kjs.2026.100537.

[24] Roberto Bruno, Roberto De Prisco, Alfredo De Santis, and Ugo Vaccaro. "Bounds and Algorithms for Alphabetic Codes and Binary Search Trees." In: *IEEE Transactions on Information Theory* 70.10 (2024), pp. 6974–6988.

[25] Roberto Bruno, Roberto De Prisco, and Ugo Vaccaro. "Old and New Results on Alphabetic Codes." In: *Information Theory and Related Fields: Festschrift in Memory of Ning Cai*. Ed. by Christian Deppe, Andreas Winter, Raymond W. Yeung, Holger Boche, Ingo Althöfer, Jens Stoye, Ulrich Tamm, and Rami Ezzine. Cham: Springer Nature Switzerland, 2025, pp. 151–194. ISBN: 978-3-031-82014-4. DOI: 10.1007/978-3-031-82014-4_7. URL: https://doi.org/10.1007/978-3-031-82014-4_7.

[26] Roberto Bruno, Roberto De Prisco, and Ugo Vaccaro. "Optimal Binary Variable-Length Codes with a Bounded Number of 1's per Codeword: Design, Analysis, and Applications." In: *arXiv preprint arXiv:2501.11129* (2025).

[27] Roberto Bruno and Ugo Vaccaro. "Entropic Bounds on the Average Length of Codes with a Space." In: *Entropy* 26.4 (2024). ISSN: 1099-4300. DOI: 10.3390/e26040283. URL: https://www.mdpi.com/1099-4300/26/4/283.

[28] J. G. Byrne. "Remark on Algorithm 428: Hu-Tucker Minimum Redundancy Alphabetic Coding Method [Z]." In: *Communications of the ACM* 16.8 (1973), p. 490.

[29] Renato M. Capocelli and Alfredo De Santis. "Tight upper bounds on the redundancy of Huffman codes." In: *IEEE Transactions on Information Theory* 35.5 (1989), pp. 1084–1091.

[30] Renato M. Capocelli and Alfredo De Santis. "New bounds on the redundancy of Huffman codes." In: *IEEE Transactions on Information Theory* 37.4 (1991), pp. 1095–1104.

[31] Renato M. Capocelli, Raffaele Giancarlo, and Inder J. Taneja. "Bounds on the redundancy of Huffman codes (Corresp.)" In: *IEEE Transactions on Information Theory* 32.6 (1986), pp. 854–857.

[32] Larry Carter and John Gill. "Conjectures on uniquely decipherable codes (Corresp.)" In: *IEEE Transactions on Information Theory* 20.3 (1974), pp. 394–396.

[33] Don Coppersmith, Maria M. Klawe, and Nicholas J. Pippenger. "Alphabetic minimax trees of degree at most t." In: *SIAM Journal on Computing* 15.1 (1986), pp. 189–192.

[34] Thomas A. Courtade and Sergio Verdú. "Cumulant generating function of codeword lengths in optimal lossless compression." In: *2014 IEEE International Symposium on Information Theory*. IEEE. 2014, pp. 2494–2498.

[35] Thomas M. Cover. *Elements of information theory*. John Wiley & Sons, 1999.

[36] Zhang Cun-Quan. "Optimal alphabetic binary tree for a nonregular cost function." In: *Discrete applied mathematics* 8.3 (1984), pp. 307–312.

[37] Yuval Dagan, Yuval Filmus, Ariel Gabizon, and Shay Moran. "Twenty (simple) questions." In: *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*. 2017, pp. 9–21.

[38] Yuval Dagan, Yuval Filmus, Daniel Kane, and Shay Moran. "The Entropy of Lies: Playing Twenty Questions with a Liar." In: *12th Innovations in Theoretical Computer Science Conference (ITCS 2021)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. 2021, pp. 1–1.

[39] Peter T. Daniels and William Bright. *The world's writing systems*. Oxford University Press, 1996.

[40] Roberto De Prisco and Alfredo De Santis. "On binary search trees." In: *Information Processing Letters* 45.5 (1993), pp. 249–253.

[41] Roberto De Prisco and Alfredo De Santis. "New lower bounds on the cost of binary search trees." In: *Theoretical computer science* 156.1-2 (1996), pp. 315–325.

[42] Roberto De Prisco and Alfredo De Santis. "On the redundancy achieved by Huffman codes." In: *Information Sciences* 88.1-4 (1996), pp. 131–148.

[43] Roberto De Prisco and Alfredo De Santis. "A new bound for the data expansion of Huffman codes." In: *IEEE Transactions on Information Theory* 43.6 (1997), pp. 2028–2032.

[44] Roberto De Prisco and Giuseppe Persiano. "Characteristic inequalities for binary trees." In: *Information processing letters* 53.4 (1995), pp. 201–207.

[45] Shlomi Dolev, Ephraim Korach, and Dmitry Yukelson. "The sound of silence—guessing games for saving energy in mobile environment." In: *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*. 1999, p. 273.

[46] Antonio Fariña, Travis Gagie, Giovanni Manzini, Gonzalo Navarro, and Alberto Ordónez. "Efficient and compact representations of some non-canonical prefix-free codes." In: *International Symposium on String Processing and Information Retrieval*. Springer. 2016, pp. 50–60.

[47] Michael L. Fredman. "Two applications of a probabilistic search technique: Sorting X+ Y and building balanced search trees." In: *Proceedings of the seventh annual ACM symposium on Theory of computing*. 1975, pp. 240–244.

[48] Hiroshi Fujiwara and Tobias Jacobs. "On the huffman and alphabetic tree problem with general cost functions." In: *Algorithmica* 69.3 (2014), pp. 582–604.

[49] Travis Gagie. "Compressing probability distributions." In: *Information Processing Letters* 97.4 (2006), pp. 133–137.

[50] Travis Gagie. "A new algorithm for building alphabetic minimax trees." In: *Fundamenta Informaticae* 97.3 (2009), pp. 321–329.

[51] Robert G. Gallager. "Variations on a theme by Huffman." In: *IEEE Transactions on Information Theory* 24.6 (1978), pp. 668–674.

[52] Michael R. Garey. "Optimal binary identification procedures." In: *SIAM Journal on Applied Mathematics* 23.2 (1972), pp. 173–186.

[53] Michael R. Garey. "Optimal binary search trees with restricted maximal depth." In: *SIAM Journal on Computing* 3.2 (1974), pp. 101–110.

[54] Adriano M. Garsia and Michelle L. Wachs. "A new algorithm for minimum cost binary trees." In: *SIAM Journal on Computing* 6.4 (1977), pp. 622–642.

[55] Paweł Gawrychowski. "Alphabetic minimax trees in linear time." In: *International Computer Science Symposium in Russia*. Springer. 2013, pp. 36–48.

[56] Edgar N. Gilbert and Edward F. Moore. "Variable-length binary encodings." In: *Bell System Technical Journal* 38.4 (1959), pp. 933–967.

[57] Mordecai J. Golin, Claire Kenyon, and Neal E. Young. "Huffman coding with unequal letter costs." In: *Proceedings of the thirty-fourth annual ACM symposium on theory of computing*. 2002, pp. 785–791.

[58] Mordecai J. Golin, Claire Mathieu, and Neal E. Young. "Huffman coding with letter costs: A linear-time approximation scheme." In: *SIAM Journal on Computing* 41.3 (2012), pp. 684–713.

[59] Mordecai J. Golin and Hyeon-Suk Na. "Generalizing the Kraft-McMillan inequality to restricted languages." In: *Data Compression Conference*. IEEE. 2005, pp. 163–172.

[60] Mordecai J. Golin and Yan Zhang. "A Dynamic Programming Approach to Length-Limited Huffman Coding: Space Reduction With the Monge Property." In: *IEEE Transactions on Information Theory* 56.8 (2010), pp. 3918–3929.

[61] Louis Gotlieb and Derick Wood. "The construction of optimal multiway search trees and the monotonicity principle." In: *International Journal of Computer Mathematics* 9.1 (1981), pp. 17–24.

[62] Goetz Graefe. "Implementing sorting in database systems." In: *ACM Computing Surveys (CSUR)* 38.3 (2006).

[63] Pankaj Gupta, Balaji Prabhakar, and Stephen Boyd. "Near-optimal routing lookups with bounded worst case performance." In: *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (Cat. No. 00CH37064)*. Vol. 3. IEEE. 2000, pp. 1184–1192.

[64] Refael Hassin and Mordecai Henig. "Monotonicity and efficient computation of optimal dichotomous search." In: *Discrete applied mathematics* 46.3 (1993), pp. 221–234.

[65] Refael Hassin and Anna Sarid. "Operations research applications of dichotomous search." In: *European Journal of Operational Research* 265.3 (2018), pp. 795–812.

[66] Yokoo Hidetoshi. "An efficient representation of the integers for the distribution of partial quotients over the continued fractions." In: *Journal of Information Processing* 11.4 (1989), pp. 288–293.

[67] Tomotaka Hiraoka and Hirosuke Yamamoto. "Alphabetic AIFV codes constructed from Hu-Tucker codes." In: *2018 IEEE International Symposium on Information Theory (ISIT)*. IEEE. 2018, pp. 2182–2186.

[68] Yasuichi Horibe. "An improved bound for weight-balanced tree." In: *Information and Control* 34.2 (1977), pp. 148–151.

[69] Te C. Hu. "A new proof of the TC algorithm." In: *SIAM Journal on Applied Mathematics* 25.1 (1973), pp. 83–94.

[70] Te C. Hu, Daniel J. Kleitman, and Jeanne K. Tamaki. "Binary trees optimum under various criteria." In: *SIAM Journal on Applied Mathematics* 37.2 (1979), pp. 246–256.

[71] Te C. Hu, Lawrence L. Larmore, and J. David Morgenthaler. "Optimal integer alphabetic trees in linear time." In: *European Symposium on Algorithms*. Springer. 2005, pp. 226–237.

[72] Te C. Hu and Kung C. Tan. "Least upper bound on the cost of optimum binary search trees." In: *Acta Informatica* 1.4 (1972), pp. 307–310.

[73] Te C. Hu and Kung C. Tan. "Path length of binary search trees." In: *SIAM Journal on Applied Mathematics* 22.2 (1972), pp. 225–234.

[74] Te C. Hu and Alan C. Tucker. "Optimal computer search trees and variable-length alphabetical codes." In: *SIAM Journal on Applied Mathematics* 21.4 (1971), pp. 514–532.

[75] Te Chiang Hu and J. David Morgenthaler. "Optimum alphabetic binary trees." In: *Franco-Japanese and Franco-Chinese Conference on Combinatorics and Computer Science*. Springer. 1995, pp. 234–243.

[76] Te Chiang Hu, Min-Teh Shing, and Yu-Shung Kuo. *Combinatorial Algorithms*. USA: Addison-Wesley Longman Publishing Co., Inc., 1982. ISBN: 0201038595.

[77] David A. Huffman. "A method for the construction of minimum-redundancy codes." In: *Proceedings of the IRE* 40.9 (1952), pp. 1098–1101.

[78] Alon Itai. "Optimal alphabetic trees." In: *SIAM Journal on Computing* 5.1 (1976), pp. 9–18.

[79] Ken-ichi Iwata and Hirosuke Yamamoto. "An algorithm for constructing the optimal code trees for binary alphabetic AIFV-m codes." In: *2020 IEEE Information Theory Workshop (ITW)*. IEEE. 2021, pp. 1–5.

[80] Edwin T. Jaynes. "Note on unique decipherability." In: *IRE Transactions on Information Theory* 5.3 (1959), pp. 98–102.

[81] Ottar Johnsen. "On the redundancy of binary Huffman codes (Corresp.)" In: *IEEE Transactions on Information Theory* 26.2 (1980), pp. 220–222.

[82] Ali Kakhbod and Morteza Zadimoghaddam. "On the construction of prefix-free and fix-free codes with specified codeword compositions." In: *Discrete applied mathematics* 159.18 (2011), pp. 2269–2275.

[83] Richard Karp, Elias Koutsoupias, Christos Papadimitriou, and Scott Shenker. "Optimization problems in congestion control." In: *Proceedings 41st Annual Symposium on Foundations of Computer Science*. IEEE. 2000, pp. 66–74.

[84] Marek Karpinski, Lawrence L. Larmore, and Wojciech Rytter. "Correctness of constructing optimal alphabetic trees revisited." In: *Theoretical Computer Science* 180.1-2 (1997), pp. 309–324.

[85] Gyula O. H. Katona. "Combinatorial search problems." In: *A survey of combinatorial theory*. Elsevier, 1973, pp. 285–308.

[86] Mohammadali Khosravifard, Hassan Halabian, and T. Aaron Gulliver. "A Kraft-type sufficient condition for the existence of D-ary fix-free codes." In: *IEEE Transactions on Information Theory* 56.6 (2010), pp. 2920–2927.

[87] Jeffrey H. Kingston. "A new proof of the Garsia-Wachs algorithm." In: *Journal of Algorithms* 9.1 (1988), pp. 129–136.

[88] David G. Kirkpatrick and Maria M. Klawe. "Alphabetic minimax trees." In: *SIAM Journal on Computing* 14.3 (1985), pp. 514–526.

[89] Maria Klawe and Brendan Mumey. "Upper and lower bounds on constructing alphabetic binary trees." In: *SIAM Journal on Discrete Mathematics* 8.4 (1995), pp. 638–651.

[90] Daniel J. Kleitman and Michael E. Saks. "Set orderings requiring costliest alphabetic binary trees." In: *SIAM Journal on Algebraic Discrete Methods* 2.2 (1981), pp. 142–146.

[91] Donald E. Knuth. "Optimum binary search trees." In: *Acta informatica* 1 (1971), pp. 14–25.

[92] Donald E. Knuth. "Dynamic huffman coding." In: *Journal of algorithms* 6.2 (1985), pp. 163–180.

[93] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. 2nd. Addison-Wesley, 1998. ISBN: 978-0201896855.

[94] Joseph D.E. Konhauser, Dan Velleman, and Stan Wagon. *Which way did the bicycle go?: and other intriguing mathematical mysteries*. 18. Cambridge University Press, 1996.

[95] Ioannis Kontoyiannis and Sergio Verdú. "Optimal lossless data compression: Non-asymptotics and asymptotics." In: *IEEE Transactions on Information Theory* 60.2 (2014), pp. 777–795.

[96] S. Rao Kosaraju, Teresa M. Przytycka, and Ryan Borgstrom. "On an optimal split tree problem." In: *Workshop on Algorithms and Data Structures*. Springer. 1999, pp. 157–168.

[97] Oliver Kosut and Lalitha Sankar. "Asymptotics and non-asymptotics for universal fixed-to-variable source coding." In: *IEEE Transactions on Information Theory* 63.6 (2017), pp. 3757–3772.

[98] Eduardo Sany Laber, Ruy Luiz Milidiú, and Artur Alves Pessoa. "Practical constructions of L-restricted alphabetic prefix codes." In: *6th International Symposium on String Processing and Information Retrieval. 5th International Workshop on Groupware* (*Cat. No. PR00268*). IEEE. 1999, pp. 115–119.

[99] Lawrence L. Larmore. "Height restricted optimal binary trees." In: *SIAM Journal on Computing* 16.6 (1987), pp. 1115–1123.

[100] Lawrence L. Larmore. "Minimum delay codes." In: *SIAM Journal on Computing* 18.1 (1989), pp. 82–94.

[101] Lawrence L. Larmore and Teresa M. Przytycka. "A fast algorithm for optimum height-limited alphabetic binary trees." In: *SIAM Journal on Computing* 23.6 (1994), pp. 1283–1312.

[102] Lawrence L. Larmore and Teresa M. Przytycka. "The optimal alphabetic tree problem revisited." In: *Journal of Algorithms* 28.1 (1998), pp. 1–20.

[103] Hyafil Laurent and Ronald L. Rivest. "Constructing optimal binary decision trees is NP-complete." In: *Information Processing Letters* 5.1 (1976), pp. 15–17.

[104] Jan van Leeuwen, Nicola Santoro, Jorge Urrutia, and Shmuel Zaks. "Guessing games and distributed computations in synchronous networks." In: *International Colloquium on Automata, Languages, and Programming*. Springer. 1987, pp. 347–356.

[105] Christos Levcopoulos, Andrzej Lingas, and Jörg-R Sack. "Heuristics for optimum binary search trees and minimum weight triangulation problems." In: *Theoretical Computer Science* 66.2 (1989), pp. 181–203.

[106] Marc J. Lipman and Julia Abrahams. "Minimum average cost testing for partially ordered components." In: *IEEE Transactions on Information Theory* 41.1 (1995), pp. 287–291.

[107] Yuval Lirov and O-C Yue. "Circuit pack troubleshooting via semantic control. I. goal selection." In: *Proceedings of the International Workshop on Artificial Intelligence for Industrial Applications*. IEEE. 1988, pp. 118–122.

[108] Detlev Marpe, Heiko Schwarz, and Thomas Wiegand. "Context-based adaptive binary arithmetic coding in the H. 264/AVC video compression standard." In: *IEEE Transactions on Circuits and Systems for video technology* 13.7 (2003), pp. 620–636.

[109] David W. Matula and Peter Kornerup. "An order preserving finite binary encoding of the rationals." In: *IEEE 6th Symposium on Computer Arithmetic (ARITH)*. IEEE. 1983, pp. 201–209.

[110] Kurt Mehlhorn. "Nearly optimal binary search trees." In: *Acta Informatica* 5.4 (1975), pp. 287–295.

[111] Kurt Mehlhorn. "Searching, sorting and information theory." In: *International Symposium on Mathematical Foundations of Computer Science*. Springer. 1979, pp. 131–145.

[112] Bruce L. Montgomery and Julia Abrahams. "On the redundancy of optimal binary prefix-condition codes for finite and infinite sources." In: *IEEE Transactions on Information Theory* 33.1 (1987), pp. 156–160.

[113] S. V. Nagaraj. "Optimal binary search trees." In: *Theoretical Computer Science* 188.1-2 (1997), pp. 1–44.

[114] S. V. Nagaraj. "Performance of Routing Lookups." In: *International Conference on Advances in Computing and Information Technology*. Springer. 2011, pp. 482–487.

[115] Narao Nakatsu. "Bounds on the redundancy of binary alphabetical codes." In: *IEEE Transactions on Information Theory* 37.4 (1991), pp. 1225–1229.

[116] Jürg Nievergelt. "Binary search trees and file organization." In: *ACM Computing Surveys (CSUR)* 6.3 (1974), pp. 195–207.

[117] B. S. Pasternack, D. E. Bohning, and J. Thomas. "Group-sequential leak-testing of sealed radium sources." In: *Technometrics* 18.1 (1976), pp. 59–66.

[118] Krishna R. Pattipati and Mark G. Alexandridis. "Application of heuristic search and information theory to sequential fault diagnosis." In: *IEEE Transactions on Systems, Man, & Cybernetics* 20.4 (1990), pp. 872–887.

[119] Laura Pezza, Luca G. Tallini, and Bella Bose. "Variable length unordered codes." In: *IEEE transactions on information theory* 58.2 (2012), pp. 548–569.

[120] Franco P. Preparata and Michael I. Shamos. *Computational geometry: an introduction*. Springer Science & Business Media, 2012.

[121] Prakash Ramanan. "Testing the optimality of alphabetic trees." In: *Theoretical Computer Science* 93.2 (1992), pp. 279–301.

[122] D. Richards. "On the worst-possible analysis of weighted comparison-based algorithms." In: *The Computer Journal* 31.3 (1988), pp. 276–278.

[123] Ron M. Roth and Paul H. Siegel. "Variable-Length Constrained Coding and Kraft Conditions: The Parity-Preserving Case." In: *IEEE Transactions on Information Theory* 67.9 (2021), pp. 6179–6192.

[124] Eugene S. Schwartz. "An optimum encoding with minimum longest code and total number of digits." In: *Information and control* 7.1 (1964), pp. 37–44.

[125] Claude E. Shannon. "A mathematical theory of communication." In: *The Bell system technical journal* 27.3 (1948), pp. 379–423.

[126] Dafna Sheinwald. "On binary alphabetical codes." In: *1992 Data Compression Conference*. IEEE Computer Society. 1992, pp. 112–113.

[127] Wojciech Szpankowski and Sergio Verdú. "Minimum expected length of fixed-to-variable lossless compression without prefix constraints." In: *IEEE Transactions on Information Theory* 57.7 (2011), pp. 4017–4025.

[128] J. Thomas, B. S. Pasternack, S. J. Vacirca, and D. L. Thompson. "Application of group testing procedures in radiological health." In: *Health Physics* 25.3 (1973), pp. 259–266.

[129] Hirendu Vaishnav and Massoud Pedram. "Alphabetic trees-theory and applications in layout-driven logic synthesis." In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 20.1 (2001), pp. 58–69.

[130] Jan Van Leeuwen. "On the Construction of Huffman Trees." In: *ICALP*. 1976, pp. 382–410.

[131] Kristo Visk and Ago-Erik Riet. "Generalisation of Kraft inequality for source coding into permutations." In: *2016 IEEE International Symposium on Information Theory (ISIT)*. IEEE. 2016, pp. 1237–1241.

[132] W. A. Walker and Calvin C. Gotlieb. "A top-down algorithm for constructing nearly optimal lexicographic trees." In: *Graph theory and computing*. Elsevier, 1972, pp. 303–323.

[133] Russell L. Wessner. "Optimal alphabetic search trees with restricted maximal height." In: *Information Processing Letters* 4.4 (1976), pp. 90–94.

[134] Wikipedia. *Writing systems without word boundaries*. Wikimedia Foundation. 2024. URL: `https://en.wikipedia.org/wiki/Category:Writing_systems_without_word_boundaries` (visited on 03/25/2024).

[135] Aaron D. Wyner. "An upper bound on the entropy series." In: *Information and Control* 20.2 (1972), pp. 176–181.

[136] Hirosuke Yamamoto, Masato Tsuchihashi, and Junya Honda. "Almost instantaneous fixed-to-variable length codes." In: *IEEE Transactions on Information Theory* 61.12 (2015), pp. 6432–6443.

[137] F. Frances Yao. "Speed-up in dynamic programming." In: *SIAM Journal on Algebraic Discrete Methods* 3.4 (1982), pp. 532–540.

[138] Chunxuan Ye and Raymond W. Yeung. "A simple upper bound on the redundancy of Huffman codes." In: *IEEE Transactions on Information Theory* 48.7 (2002), pp. 2132–2138.

[139] Raymond W. Yeung. "Alphabetic codes revisited." In: *IEEE Transactions on Information Theory* 37.3 (1991), pp. 564–572.

[140] Raymond W. Yeung. "Local redundancy and progressive bounds on the redundancy of a Huffman code." In: *IEEE Transactions on Information Theory* 37.3 (1991), pp. 687–691.

[141] Raymond W. Yeung. "On noiseless diagnosis." In: *IEEE Transactions on Systems, Man & Cybernetics* 24.7 (1994), pp. 1074–1082.

[142] J. Michael Yohe. "Algorithm 428: Hu-Tucker minimum redundancy alphabetic coding method [Z]." In: *Communications of the ACM* 15.5 (1972), pp. 360–362.

[143] Chu Yung-ching. "An extended result of Kleitman and Saks concerning binary trees." In: *Discrete Applied Mathematics* 10.3 (1985), pp. 255–259.