

UNIVERSITY OF VERONA  
DEPARTMENT OF COMPUTER SCIENCE

GRADUATE SCHOOL OF NATURAL AND ENGINEERING SCIENCES  
DOCTORAL PROGRAM IN COMPUTER SCIENCE  
CYCLE XXXII

Algorithms and Data Structures for  
Coding, Indexing, and Mining of  
Sequential Data

S.S.D. INF/01

Coordinator: \_\_\_\_\_  
Prof. Massimo Merro




Tutor: \_\_\_\_\_  
Prof. Ferdinando Cicalese

Co-Tutor: \_\_\_\_\_  
Prof. Zsuzsanna Lipták

Doctoral Student: \_\_\_\_\_  
Massimiliano Rossi

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License, Italy. To read a copy of the licence, visit the web page:

<http://creativecommons.org/licenses/by-nc-nd/3.0/>

-  **Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
-  **NonCommercial** — You may not use the material for commercial purposes.
-  **NoDerivatives** — If you remix, transform, or build upon the material, you may not distribute the modified material.

*for Simona*



## Abstract

In recent years, the production of sequential data has been rapidly increasing. This requires solving challenging problems about how to represent information, how to retrieve information, and how to extract knowledge, from sequential data. These questions belong to the areas of coding, indexing, and mining, respectively. In this thesis, we investigate problems from those three areas.

Coding refers to the way in which information is represented. Coding aims at generating optimal codes, that are codes having a minimum expected length. Codes can be generated for different purposes, from data compression to error detection/correction. The Lempel-Ziv 77 parsing produces an asymptotically optimal code in terms of compression. We study algorithms to efficiently decompress strings from the Lempel-Ziv 77 parsing, using memory proportional to the size of the parsing itself. We provide the first implementation of an algorithm by Bille et al., the only work we are aware of on this problem. We present a practical evaluation of this approach and several optimizations which improve the performance on all datasets we tested.

Through the Ulam-Rényi game, it is possible to provide optimal adaptive error-correcting codes. The game consists of discovering an unknown  $m$ -bit number by asking membership questions the answers to which can be erroneous. Questions are formulated knowing the answers to all previous ones. We want to find an optimal strategy, i.e., a strategy that can identify any  $m$ -bit number using the theoretical minimum number of questions. We studied the case where questions are a union of up to a fixed number of intervals, and up to three answers can be erroneous. We first show that for any sufficiently large  $m$ , there exists a strategy to identify an initially unknown  $m$ -bit number which uses at most four intervals per question. We further refine our main tool to turn the above asymptotic result into a complete characterization of those instances of the Ulam-Rényi game that admit optimal strategies.

Indexing refers to the way in which information is retrieved. An index for texts permits finding all occurrences of any substring, without traversing the whole text. Many applications require to look for approximate substrings. One of these is the problem of jumbled pattern matching, where two strings match if one is a permutation of the other. We study combinatorial aspects of prefix normal words, a class of binary words introduced in this context. These words can be used as indices for the Indexed Binary Jumbled Pattern Matching problem. We present a new recursive generation algorithm for prefix normal words that is competitive with the previous one but allows to list all prefix normal words sharing the same prefix. This sheds lights on novel insights that may help solving the problem of counting the number of prefix normal words

of a given length. We then introduce infinite prefix normal words, and we show that one of the operations used by the algorithm, when repeatedly applied to extend a word, produces an infinite prefix normal word. This motivates the seeking for other operations that produce infinite prefix normal words. We found that one of these operations establishes a connection between prefix normal words and Sturmian words. We also explored the relationship between prefix normal words and Abelian complexity, as well as between prefix normal words and lexicographic order.

Mining refers to the way in which information is converted into knowledge. The process of knowledge discovery covers several processing steps, including knowledge extraction. We analyze the problem of mining assertions for an embedded system from its simulation traces. This problem can be modeled as a pattern discovery problem on colored strings. We present two problems of pattern discovery on colored strings: patterns for one color only, or for all colors at the same time. We present two suffix tree-based algorithms. The first algorithm solves both the one color problem and the all colors problem. We then, introduce modifications which improve performance of the algorithm both on synthetic and on real data. We implemented and evaluated the proposed approaches, highlighting time trade-offs that can be obtained.

A different way of knowledge extraction is based on the information-theoretic perspective of Pearl's model of causality. It has been postulated that the true causality direction between two phenomena A and B is related to the problem of finding the minimum entropy joint distribution between A and B. This problem is known to be NP-hard, and greedy algorithms have recently been proposed. We provide a novel analysis of one of the proposed heuristic showing that this algorithm guarantees an additive approximation of 1 bit. We then, provide a general criterion for guaranteeing an additive approximation factor of 1. This criterion may be of independent interest in other contexts where couplings are used.

## Abstract (Italian)

Negli ultimi anni, la produzione di dati sequenziali è in rapido aumento. Ciò richiede la risoluzione di problemi sempre più impegnativi su come rappresentare, recuperare informazioni ed estrarre conoscenza dai dati sequenziali. Queste domande appartengono rispettivamente alle aree di codifica, indicizzazione ed estrazione. In questa tesi si investigano problemi negli ambiti di queste tre aree.

Per codifica si intende il modo in cui le informazioni sono rappresentate. Questa mira a generare codici ottimali, ovvero codici con una lunghezza minima attesa, che possono essere generati per scopi diversi, dalla compressione dei dati alla rilevazione/correzione degli errori. La fattorizzazione Lempel-Ziv 77 produce un codice asintoticamente ottimale in termini di compressione. In questa tesi studiamo algoritmi per decomprimere in modo efficiente stringhe dalla fattorizzazione Lempel-Ziv 77, usando memoria proporzionale alla dimensione della fattorizzazione stessa. Forniamo la prima implementazione dell'algoritmo di Bille et al., l'unico lavoro di cui siamo a conoscenza su questo problema. Presentiamo una valutazione pratica di questo approccio e diverse ottimizzazioni che migliorano le prestazioni su tutti i dati che abbiamo testato.

Attraverso il gioco di Ulam-Rényi, è possibile costruire codici a correzione di errori adattivi ottimali. Il gioco consiste nello scoprire un numero sconosciuto di  $m$  bit, ponendo domande di appartenenza le cui risposte possono essere errate. Nel caso adattivo, le domande sono formulate conoscendo le risposte a tutte le precedenti domande poste. L'obiettivo è quello di trovare una strategia ottima, ovvero una strategia in grado di identificare qualsiasi numero di  $m$  bit usando il minimo numero teorico di domande. Abbiamo studiato il caso in cui le domande sono l'unione di un numero fissato di intervalli, in cui fino a tre risposte possono essere errate. Mostriamo innanzitutto che per qualsiasi  $m$  sufficientemente grande, esiste una strategia per identificare un numero di  $m$  bit inizialmente sconosciuto che utilizza al più quattro intervalli. Raffiniamo, poi, il nostro principale strumento per trasformare il risultato asintotico in una caratterizzazione completa delle istanze del gioco Ulam-Rényi che ammettono strategie ottime.

Per indicizzazione si intende il modo in cui le informazioni vengono recuperate. Indicizzare un testo, permette di individuare tutte le occorrenze di qualsiasi sua sottostringa, senza necessariamente scorrere tutto il testo. Molte applicazioni richiedono la ricerca di sottostringhe approssimate. Uno di questi è il problema di jumbled pattern matching, dove due stringhe corrispondono se una è una permutazione dell'altra. In questa tesi studiamo gli aspetti combinatori delle parole normali prefisse, una classe di parole binarie introdotte in questo

contesto, che possono essere usate come indici per il problema indicizzazione del jumbled pattern matching nel caso binario.

Presentiamo un nuovo algoritmo di generazione ricorsiva per le parole normali prefisso che è paragonabile con il precedente ma che consente di elencare tutte le parole normali prefisse condividendo lo stesso prefisso. Questo mette in risalto nuovi aspetti che possono essere utili nel risolvere il problema di contare il numero di parole normali prefisse di una certa lunghezza. Introduciamo le parole normali prefisse infinite e mostriamo che una delle operazioni utilizzate dall'algoritmo di generazione, quando viene ripetutamente applicata per estendere una parola, produce un parola normale prefissa infinita. Questo ha motivato la ricerca di altre operazioni che producono parole normali prefisse infinite. Una di queste operazioni stabilisce una connessione tra le parole normali prefisse e le parole Sturmiane. Abbiamo esplorato le relazioni tra parole normali prefisse e la complessità abeliana, come anche tra parole normali prefisse e l'ordine lessicografico.

Per estrazione si intende il modo in cui le informazioni vengono convertite in conoscenza. Il processo di scoperta della conoscenza copre diverse fasi di elaborazione, inclusa l'estrazione della conoscenza. Analizziamo il problema dell'estrazione di asserzioni per un sistema embedded, date le sue tracce di simulazione. Questo problema può essere modellato come un problema di rilevamento di patterns in stringhe colorate. Presentiamo due problemi di scoperta di patterns su stringhe colorate, vale a dire patterns relativi ad un solo colore o relativi a tutti i colori contemporaneamente. Presentiamo due algoritmi basati su alberi dei suffissi. Il primo algoritmo risolve sia il problema nel caso di patterns relativi ad un solo colore, che il caso di patterns relativi a tutti i colori. Abbiamo introdotto delle modifiche che aumentano le prestazioni dell'algoritmo sia sui dati sintetici che sui dati reali. Abbiamo implementato e valutato gli approcci proposti, evidenziando i diversi compromessi sui tempi di esecuzione che possono essere ottenuti.

Un altro modo di estrarre la conoscenza è dalla prospettiva della teoria dell'informazione attraverso il modello di causalità di Pearl. È stato postulato che la direzione corretta della causalità tra due fenomeni A e B sia correlata al problema di trovare la distribuzione congiunta tra A e B di minima entropia. Questo problema è noto essere NP-hard e sono stati proposti recentemente degli algoritmi greedy. In questa tesi forniamo una nuova analisi di una delle euristiche proposte dimostrando che questo algoritmo garantisce un'approssimazione additiva di 1 bit. Quindi, forniamo un criterio generale per garantire un fattore di approssimazione addizionale di 1. Questo criterio può essere di interesse indipendente in altri contesti in cui il coupling viene utilizzato.



## Acknowledgements

First and foremost I would like to thank my advisors Ferdinando Cicalese and Zsuzsanna Lipták. Ferdinando made me take my first steps in the theory of computing introducing me to Paul and Carole. Zsuzsa uncovered the beauty of strings to me. They guided my research work always giving me the freedom to choose what I wanted to do. They are passionating teachers and always available to explain when their students come to ask — and to advise when their students have doubts. They thought me how to be a scientist, showing me how to ask the right questions and how to find the answers. They showed me that doing science is not a job, but it is mostly a way of living, constant research of the truth. I would also like to thank Simon J. Puglisi who led me to the field of data compression and data structures. He hosted me during my period in Helsinki and he showed me that it is always possible to look at the same problem from different points of view, and each different view makes room for improvement. I wish to thank all my present and past officemates and beyond: Vincenzo, Michele, Florenc, Stefano, Alessandro, Enrico, Matteo, Francesca, Stefano, Simone, Nino e Stefano who showed me that science is not something that you can do by yourself and keep it for yourself, but it has to be shared with colleagues and friends. For all lunches, coffees, discussions, and laughs that we had together. I would like to thank all the tutors of the programming challenges course, and my teammates: Andrea, Alessandro, and Eros, with who I have shared the SWERC and Hash Code experiences. I wish to thank all my friends in Verona, in particular: Marco, Paolo, Simone, Francesca, Matteo, Antonella, Angelo, Alice, Francesco, Giada, Stefano, Cristina, Emanuele, Riccardo, because also PhD students may have spare time. They are the living proof that you can always feel at home even if you are far away from it. I want to thank all my friends in Amandola, with which we don't meet very often, but the bound is always strong. In particular, I wish to thank all the boys and girls of the ACR who are always in my thoughts. I wish to thank my brother Marco, his wife Laura, they let me feel as I never went away from home. I don't have a real thanks to my little nephew Elisabetta, but it was so a great joy when she was born in the middle of this journey that I could not leave her out of these acknowledgments. I want to thank my parents who taught me to never give up, they always believed in me and supported me in every decision I made, even if they brought me far from home. I want to thank my grandmother who gave me the strength and the courage to go forward. I wish to thank my dog Tiago, who relieved me of all problems every time I needed it. I thank Simona who bumped in my life like a thunder, showing to me that nothing is impossible as long as you believe in it. She made all this possible.



---

# Contents

<b>List of Figures</b> .....	I
<b>List of Tables</b> .....	III
<b>List of Algorithms</b> .....	V
<b>1 Introduction</b> .....	1
1.1 Coding .....	1
1.1.1 Data compression and Lempel-Ziv 77 .....	1
1.1.2 Reliability and Ulam-Rényi game .....	3
1.2 Indexing .....	5
1.2.1 Approximate patterns and prefix normal words .....	5
1.3 Mining .....	7
1.3.1 Assertion mining and colored strings .....	8
1.3.2 Causality discovery and greedy minimum-entropy coupling ...	9
<b>2 Lempel-Ziv 77 decompression</b> .....	11
2.1 Preliminaries .....	12
2.2 Bille et al.'s algorithm .....	12
2.3 Practical Small Space LZ Decompression .....	13
2.3.1 A Fast, Practical Mergeable Dictionary .....	14
2.3.2 Hybrid Decompression .....	16
2.4 Experimental Results .....	17
2.4.1 Setup .....	17
2.4.2 Data .....	17
2.4.3 Algorithms .....	17
2.4.4 Results .....	19
2.4.5 Effects of cache and in-chunk decompression .....	19
2.4.6 Parameter Tuning .....	19
<b>3 Ulam-Rényi game</b> .....	21
3.1 Basic facts .....	23
3.2 The key result - structure of perfect 4-interval strategies for 3 lies ....	27
3.3 The proof of Theorem 3.1 .....	28
3.4 The non asymptotic strategy .....	37

3.4.1	The proof of the main theorem .....	41
<b>4</b>	<b>Prefix normal words</b> .....	47
4.1	Basics .....	49
4.2	The Bubble-Flip algorithm .....	52
4.2.1	The algorithm .....	52
4.2.2	Listing $\mathcal{L}_n$ as a combinatorial Gray code .....	58
4.2.3	Prefix normal words with given critical prefix .....	59
4.2.4	Practical improvements of the algorithm .....	63
4.3	On infinite extensions of prefix normal words .....	66
4.3.1	Flip extensions and ultimate periodicity .....	67
4.3.2	Lazy flip extensions .....	70
4.4	Prefix normal words and Sturmian words .....	71
4.5	Prefix normal words and lexicographic order .....	73
4.6	Prefix normal words, prefix normal forms, abelian complexity .....	74
4.6.1	Balanced and $c$ -balanced words .....	75
4.6.2	Prefix normal forms and abelian complexity .....	77
4.6.3	Prefix normal forms of Sturmian words .....	81
4.6.4	Prefix normal forms of binary uniform morphisms .....	81
4.7	A characterization of periodic and aperiodic prefix normal words with respect to minimum density .....	83
<b>5</b>	<b>Pattern discovery in colored strings</b> .....	87
5.1	Basics .....	90
5.1.1	Colored strings .....	91
5.1.2	Suffix trees and suffix arrays .....	92
5.1.3	Maximum-oriented indexed priority queue .....	93
5.1.4	Rank, select, and range maximum query .....	93
5.2	A pattern discovery algorithm for colored strings using the suffix tree ..	94
5.2.1	Finding all $(y, d)$ -unique substrings .....	95
5.2.2	Outputting only minimally $(y, d)$ -unique substrings .....	96
5.2.3	An algorithm for all colors .....	98
5.3	Skipping Algorithm .....	99
5.3.1	Right-minimality check .....	105
5.4	Output restrictions and algorithm improvement .....	106
5.5	Experimental results .....	108
5.5.1	Setup .....	108
5.5.2	Data .....	108
5.5.3	Algorithms .....	109
5.5.4	Results .....	109
<b>6</b>	<b>Greedy minimum-entropy coupling</b> .....	115
6.1	Basic facts .....	116
6.1.1	Notation .....	116
6.1.2	Joint Entropy Minimization Algorithm .....	116
6.1.3	Majorization .....	117
6.2	Main Theorem .....	118
6.3	A General approach for additive approximation on couplings .....	120

<b>7 Conclusion</b> .....	123
7.1 Future work .....	125
<b>References</b> .....	127



---

## List of Figures

1.1	International Morse code table	2
1.2	Search tree for an adaptive error-correcting code for 2 bits	3
1.3	Communication scenario with a noiseless feedback channel	4
1.4	Suffix tree of “BANANA\$”	6
1.5	Data mining in knowledge discovery	8
2.1	Space-time consumption trends	18
2.2	Decompression types repartition on data	19
3.1	A well-shaped state of type (3.1)	26
3.2	A well-shaped state of type (3.2)	26
3.3	An example of state dynamics	29
3.4	A well-shaped state like in (3.1) and the cuts of a 4 interval question	36
3.5	A well-shaped state like in (3.2) and the cuts of a 4 interval question	36
4.1	The words in $\mathcal{PN}(11010000)$ represented as a tree	57
4.2	A sketch of the computation tree of Algorithm 2 for the set $w = 110^{n-2}$	60
4.3	he frequency of prefix normal words with given critical prefix length	61
4.4	The area in which there are all prefix normal words with $w$ as prefix and minimum density equals to $\delta(w)$	71
4.5	The Fibonacci word	78
4.6	The Champernowne word	78
4.7	The paperfolding word	79
4.8	The Thue-Morse word	80
5.1	Simulation trace example	88
5.2	Coloring function on the suffix tree example	100
5.3	Example of $h(u, \ell)$ function, <i>left_minimal</i> array, and maximum-oriented indexed priority queue	104
6.1	Example of the execution of Algorithm 11	120





---

## List of Tables

2.1	Files used in the experiments .....	18
2.2	Execution time and memory usage on each data set .....	18
3.1	States of character $\leq 12$ which are 0-typical but non nice .....	44
3.2	First part of the proof of Lemma 3.10 .....	44
3.3	Case $m = 1$ , after the first question and case $m = 4$ , after the first four questions .....	45
3.4	Proof of Corollary 3.1 for $6 \leq m \leq 32$ and proof of [101, Lemma 6] ..	46
4.1	The set of prefix normal words of length $n = 1, 2, 3, 4, 5$ . .....	50
4.2	The size of prefix normal words sharing the same critical prefix .....	61
4.3	The maximum number of 0s and 1s of the Fibonacci word, and its prefix normal forms .....	78
4.4	The border tables of the morphism $\mu(0) = 0101$ and $\mu(1) = 1100$ ....	82
4.5	The minimum number of 0's in a factor of length $n$ of the morphism $\mu(0) = 0101$ and $\mu(1) = 1100$ .....	83
5.1	Dataset of real-world devices .....	109
5.2	Results on syntetic data to mine one color .....	110
5.3	Results on real-world data to mine one color .....	110
5.4	Results on syntetic data to mine all colors .....	112
5.5	Results on real-world data to mine all colors .....	113



---

## List of Algorithms

1	Compute $\varphi$ .....	53
2	Generate $\mathcal{PN}(w)$ .....	56
3	Bubble-Flip algorithm .....	57
4	Generate2 $\mathcal{PN}(w)$ .....	58
5	All unique algorithm .....	96
6	Baseline algorithm .....	98
7	Baseline algorithm for all colors .....	101
8	Computation of $h(u, \ell)$ function .....	103
9	Skipping algorithm .....	105
10	Computation of $fast\_h(u, \ell)$ function .....	107
11	Joint Entropy Minimization Algorithm [83] .....	117
12	Diagonal algorithm .....	120



## Introduction

Sequential data are sequences of objects where the order matter, e.g., text, natural language, DNA sequences. In the last decade, sequential data have become more and more ubiquitous. It has been estimated that between 100 million and as many as 2 billion human genomes could be sequenced by 2025. Compared to astronomy, YouTube, and Twitter, genomics is either on par with, or the most demanding of, data acquisition, storage, distribution, and analysis [128]. This production of textual data at an unprecedented pace requires solving increasingly challenging problems concerning space resources, reliability, searching, and extraction of information.

These problems can be grouped in the three tasks of *coding*, *indexing*, and *mining*.

### 1.1 Coding

Coding refers to the way in which information is represented. For example, the famous Morse code associates each character of the English alphabet to sequences of dots and dashes called *codewords*. See Figure 1.1 for the table of the *International Morse code*. Given a text over the English alphabet, the process of translating characters of the English alphabet into sequences of Morse codewords is called *encoding*, while the reverse process is called *decoding*. A code can be designed for different purposes, e.g., to reduce the total length of the encoded text (e.g., data compression), to detect/correct errors occurred during the transmission of information (e.g., reliability), to obscure the text (e.g., cryptography).

#### 1.1.1 Data compression and Lempel-Ziv 77

The Morse code is an example of a data compression encoding. The length of Morse code codewords depends on the frequency of occurrences in texts of the English language characters, i.e. frequent characters have shorter codewords. The same idea is applied in the Huffman coding procedure which produces optimal codes, in the sense of minimum expected length [46].

Another example is the Lempel-Ziv 77 parsing [139], or briefly LZ77, that has been proved to be asymptotically optimal [46]. It is an adaptive dictionary-based scheme where the text is divided into two types of substrings, literal and repeat phrases, applying the following rules (refer to Chapter 2 for a formal definition). Assuming that we have

**International Morse Code**

1. The length of a dot is one unit.
2. A dash is three units.
3. The space between parts of the same letter is one unit.
4. The space between letters is three units.
5. The space between words is seven units.

A	• —	U	• • —
B	— • • •	V	• • • —
C	— • • —	W	— • • —
D	— • •	X	— • • —
E	•	Y	— • — —
F	• • — •	Z	— — • •
G	— • • •		
H	• • • •		
I	• •		
J	• — — —		
K	— • — —		
L	• — • •	1	• — — — —
M	— — •	2	• • — — —
N	— • —	3	• • • — —
O	— — —	4	• • • • —
P	• — — •	5	• • • • •
Q	— • — —	6	• • • • •
R	• • — •	7	— • • • •
S	• • •	8	— • • • •
T	—	9	— • • • •
		0	— — — — —

**Fig. 1.1:** International Morse code table. *Rhey T. Snodgrass & Victor F. Camp, 1922 [Public domain]*

computed the parsing up to a certain position, the next phrase is a *literal phrase* if the character in the current position never appeared before, otherwise the next phrase is a *repeat phrase*, i.e., the longest substring starting in the current position which appeared before in the string.

*Example 1.* Given the text “abaabababba” its LZ77 parsing is the following:

1	2	3	4	5	6	7	8	9	10	11			
a		b		a		a	b	a	b		b	a	

The vertical bars mark the end of each phrase. The first *a* and the first *b* are *literal* phrases, while *a*, *aba*, *bab*, and *ba* are *repeat* phrases. We write repeat phrases as pairs where the first entry is the position of the previous occurrence of the phrase and the second entry is its length, while we write literal phrases as pairs where the first entry is the new character never seen before and the second is 0.

The final LZ77 encoding of the text “abaabababba” is the following:

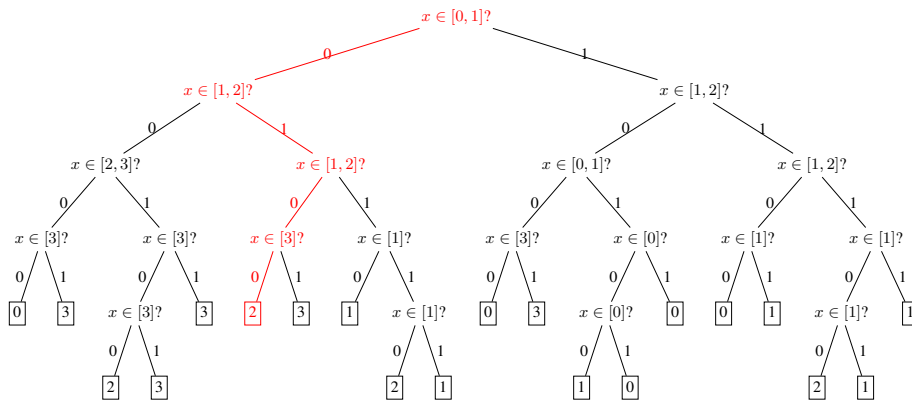
$$(a, 0) (b, 0) (1, 1) (1, 3) (5, 3) (2, 2)$$

The Lempel-Ziv 77 parsing and its later variants such as Lempel-Ziv 78 [140], Lempel-Ziv-Welch [137], Sliding window LZ77, have been widely applied in practical compression schemes and widely-used compression tools (e.g., *gzip*, *LZ4*, *Snappy*). Recently, Bille et al. [16] proposed a decompression algorithm for LZ77 that works in space proportional to the length of the parsing. In Chapter 2 we present a practical evaluation of the algorithm proposed in [16]. We further present a practical solution, asymptotically worse than [16], that reduce the space and time computational bottlenecks.

### 1.1.2 Reliability and Ulam-Rényi game

The capability of a code to carry reliable information during the transmission through a communication system is crucial in applications such as space communications, and teleoperation. Hamming codes are one way of constructing an error-correcting code, i.e., a code that allows to correct up to a given number of errors which may occur during the communication. This can be achieved by inserting redundancy in the code. The underlying idea of Hamming codes is that the number of bits in which any two codewords differ, the so-called Hamming distance, is as big as possible. For example, one possible Hamming code to provide error correction of 1 bit is to encode the value 0 with 000 and the value 1 with 111. In this case, if an error occurs then one out of 3 bits of the codewords will be flipped, i.e. 0 turned into 1 and vice versa. Then, to decode the original bit we apply a majority consensus rule, e.g., if we receive more than two 0s then the decoded bit is 0.

Error-correcting codes can be seen as answers to a search problem, the so-called Ulam-Rényi game. We want to use the minimum number of yes-no questions necessary to uniquely identify an unknown  $m$ -bit number, where up to  $e$  answers can be erroneous. The search problem induces a search tree, in which each node represents a question. The edges connecting a node with its children are labeled with the answers, and the leaves contain the unknown  $m$ -bit numbers, see Figure 1.2.



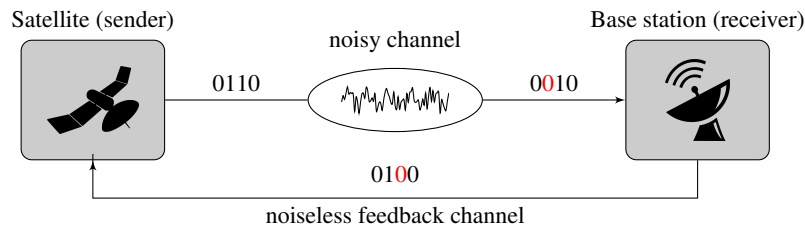
**Fig. 1.2:** A possible optimal search tree for an adaptive error-correcting code for 2 bits. Each node reports the question that has to be asked. If the answer to the question is *no* then we follow the branch labeled with 0, otherwise, we follow the branch labeled with 1. The leaves store the result of the search. The red path is the codeword received in Example 2 by the receiver.

The height of the search tree, i.e., the length of the encoding, is significantly affected by the amount of adaptiveness allowed. If questions can be asked knowing the answers to all previous ones, then we are in a fully *adaptive* setting. On the other hand, if no answer is provided, then we are in a fully *non-adaptive* setting. For example, Hamming codes are *non-adaptive codes*. Adaptiveness is provided, usually, with a noiseless feedback channel. Adaptiveness is necessary to have optimal codes that are able to correct more than 1 error [129], except for Golay codes [31].

A real world scenario is given by the case where transmitter and receiver have different power of transmission, hence one can assume that the bits delivered in one

direction are sent and received reliably, and error correction is required for the other direction. As an example, we have satellites orbiting Earth that communicate with a base station on Earth. Due to issues of power availability, the signal transmitted from the satellite is usually much weaker than that of the base station, and interferences may easily cause errors. On the other hand, the error-rate of transmissions from the base station can be negligible.

*Example 2.* Let us consider the following scenario. There are two actors, a *sender*, and a *receiver*. The sender wants to communicate a piece of information to the receiver through a noisy channel. The receiver, on his side, can transmit information back to the sender through a noiseless feedback channel as in Figure 1.3.



**Fig. 1.3:** Communication scenario with a noiseless feedback channel. The first bit sent from the satellite to the base station is the rightmost one, while the first bit sent from the base station back to the satellite is the leftmost one. The bit highlighted in red is the result of an error occurred due to the noisy channel.

Both the sender and the receiver hold the same search tree, see Figure 1.2. Let  $x = 2$  be the message that the sender wants to send. As in Figure 1.3, the sender first starts from the root of the search tree and answers the question “is  $x \in [0, 1]$ ?”. The answer “no” is encoded as a 0 and it is sent through the noisy channel to the receiver. No error occurs, the receiver reads a 0 from the channel and sends it to the sender back through the noiseless feedback channel. Both the sender and the receiver received a 0 thus they move on the search tree following the left branch of the root. At the second round, the sender answers the question “is  $x \in [1, 2]$ ?”, the answer “yes” is encoded as a 1 and it is sent through the channel. It is correctly received from the receiver who sends it back to the sender. Both the sender and the receiver received a 1, thus they move on the search tree following the right branch (the red path highlighted in Figure 1.2). In the third round, the sender answers the question “is  $x \in [1, 2]$ ?”, the answer “yes” is encoded and it is sent to the receiver. This time an error occurs and the 1 is changed into a 0 on the receiver side, who sends it back to the sender through the feedback channel. Both the sender and the receiver received a 1, thus they move on the search tree following the right branch. Finally, in the last round, the sender sends a 0, which is correctly received by the receiver who sends it back through the feedback channel. Both the sender and the receiver move through the left branch of the search tree leading to a leaf labeled with 2, decoding the message correctly.

The Ulam-Rényi game is studied in many variants. In Chapter 3 we consider the multi-interval version of the problem, in which questions are represented by set of intervals of the search space. Using multi-intervals questions allows to reduce the space to store questions in the nodes of the search tree. In general, we need  $2^m$  bits to store



one question, i.e. the set representing it, while using  $k$  interval-questions we need only  $2k \cdot m$  bits to store the beginning and the end of each interval. We show that if up to 3 errors occur, there exists a sequence of 4-interval questions that uniquely identify all elements of the search space, and the length of the search —equivalently the encoding— is the smallest possible.

## 1.2 Indexing

One of the most frequent tasks on sequential data, and on data in general, is to search where a given pattern occurs in the data. Donald Knuth devoted an entire chapter of Volume 3 of his *The art of Computer Programming* [81] to algorithms for searching.

In texts, we usually want to search for all occurrences of a short string — called *pattern*— inside the text, that is usually much longer. This process is called string matching and it can be performed in a *sequential* or *indexed* form. In the former setting, we sequentially traverse the text to report all occurrences of the pattern. In the latter setting, we preprocess the text to build the index, which allows to find all occurrences of a given pattern, without traversing the text. The choice of which string matching form has to be used, lies on the type of text we have to handle. Indexing is usually chosen when the text is large, it does not change over time, and we have enough space to save the index [100].

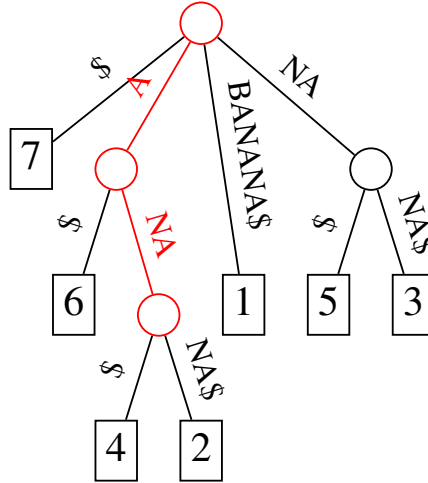
The most well-known index in strings is the *suffix tree*, introduced by Peter Weiner in 1973 [136]. See Figure 1.4 for an example of a suffix tree. The suffix tree allows to locate all substrings of the text using space linear in the length of the text, and it can be built in linear time [8]. This is remarkable because a string of length  $n$  can have  $\mathcal{O}(n^2)$  possible substrings, while the suffix tree requires only  $\mathcal{O}(n)$  space. Furthermore, given a pattern of length  $\mathcal{O}(m)$ , the suffix tree allows finding all occurrences of the pattern in  $\mathcal{O}(m + occ)$ , where  $occ$  is the number of occurrences of the pattern in the text.

### 1.2.1 Approximate patterns and prefix normal words

Depending on the application, one may want to search approximate occurrences of the given pattern. One type of approximate pattern matching is jumbled pattern matching [5, 6, 7, 23, 29, 34, 49, 65, 66, 85, 98]. Here, given a text and a pattern, we want to find all substrings of the text that are permutations of the characters of the pattern, i.e. a substring with the same multiplicity of the characters of the pattern.

*Example 3.* Consider “*The history of how radium came to be*”, clearly “*madame curie*” does not match any substring of this text. However, we have that “*madame curie*” appears in the text as a jumbled pattern. In particular “*radium came*” is a permutation of “*madame curie*”, i.e. it contains the same characters. An index for jumbled pattern matching for the text “*The history of how radium came to be*” would need to store the information that there is a substring of the text that contains 2 *a*’s, 1 *c*, 1 *d*, 2 *e*’s, 1 *i*, 2 *m*’s, 1 *r*, and 1 *u*.

For alphabets of size greater than 3, there are hardness results for the jumbled indexing problem, under different 3-SUM Hardness assumptions [7]. — The 3-SUM problem is the problem of finding three elements  $a \in A$ ,  $b \in B$ , and  $c \in C$  from three integer sets  $A$ ,  $B$ , and  $C$ , such that  $a + b = c$  — The best index for jumbled pattern matching [29]



**Fig. 1.4:** Suffix tree of “BANANAS\$”. The suffix tree is a rooted tree where the concatenation of the labels in each root to leaf path is a suffix of “BANANAS\$”. In particular, a leaf is numbered with  $i$  if its root to leaf path is the  $i$ -th suffix of “BANANAS\$”. The red path represents the substring “ANA” of “BANANAS\$”. The leaves under the lowest red node are the indices of the occurrences of the substring “ANA” in “BANANAS\$”.

has strongly subquadratic construction and strongly sublinear query time, which for larger alphabets approaches the conditional lower bounds shown in [7].

For binary alphabets, there exists an index for the decision version of the jumbled pattern matching problem — decide whether a permutation of the pattern occur in the text — that uses linear space and constant query time [23], and it can be built in strongly subquadratic time, i.e.,  $\mathcal{O}(n^{1.859})$  [29]. This index is based on the *interval property* of binary string [23], where given a binary string, if there are two substrings of length  $\ell$  of the string having  $a$  and  $b$  1s, respectively, then the text contains a substring of length  $\ell$  with all intermediate number of 1s. Thus, storing for each length  $\ell$  of the text, the maximum and the minimum number of 1s in a substring of length  $\ell$ , to answer a query, it is enough to check if the required number of 1s in the query is in between the minimum and the maximum number of 1s for a substring of the length of the query. This information can be encoded in two binary strings called prefix normal forms (See Chapter 4 for formal definitions), and retrieved in constant time, e.g., using a rank data structure.

Those binary strings have special properties, i.e., one of the two string (resp. the other) has the property that the number of 1s (resp. 0s) in the prefix is always greater than or equals to the number of 1s (resp. 0s) in any other substring of the same length. Words with this structure are referred to as prefix normal words (See Chapter 4 for formal definitions). This equivalence between indices and prefix normal words suggested that a combinatorial characterization of these words could shed light on algorithmic and complexity issues of jumbled pattern matching. This has motivated the study of the language of prefix normal words [57].

We will treat *prefix normal words* in Chapter 4. In particular, we present a new algorithm for generating all prefix normal words of a fixed length. The new algorithm runs in worst-case linear time per words, which is competitive with the previous one [24]

that runs in amortized linear time per words. However, it allows in addition to list all prefix normal words sharing the same prefix. Moreover, it gives new insights into properties of prefix normal words. We then introduce *infinite prefix normal words*. We show that one of the operation used by the algorithm, when repeatedly applied to extend a word, produces an infinite prefix normal word. We use a similar extension operation to establish connections between prefix normal words and Sturmian words, a well-known class of infinite binary words. We further show connections between prefix normal words and lexicographic order. We also show that is always possible to compute the abelian complexity function of a word given its prefix normal forms, while the converse is not always possible. We then provide sufficient conditions to compute the prefix normal forms of a word, given its abelian complexity function. We further extend a recent result on computing the abelian complexity function of binary uniform morphisms (See Chapter 4 for formal definitions) to compute the prefix normal forms of binary uniform morphisms. Finally, we provide a characterization of periodicity and aperiodicity of prefix normal words based on their minimum density, a parameter that we introduce in this context.

### 1.3 Mining

The process of *knowledge discovery from data*, also known as KDD, is a workflow composed of 7 steps [71], as shown in Figure 1.5:

- Data cleaning:** Clears the data from noise, inconsistency, and redundancies.
- Data integration:** Combines multiple sources of the data.
- Data selection:** Selects and retrieve data that are important to the analysis.
- Data transformation:** Converts data into the form suitable for mining, usually by aggregation operations.
- Data mining:** Discovers interesting patterns in the data.
- Pattern evaluation:** Sifts the mined patterns using *interestingness measures* to find those representing knowledge.
- Knowledge presentation:** Represents and visualizes the extracted knowledge to the users.

Data cleaning, integration, selection, and transformation are pre-processing steps, in which data are prepared to be mined. The data mining step is the core step in which interesting patterns and knowledge are discovered. Pattern evaluation and knowledge presentation are post-processing steps in which the discovered knowledge is refined through filtering steps. Data mining can be applied in all those data that carry information for a target application, independent of the type of data. The common sources of data are databases and data warehouses, but the mining process can be also applied to streams, graphs, texts, etc. . . . Data mining tasks can be tuned according to the functionalities that we want to use. The functionalities include characterization and discrimination, mining of frequent patterns, classification, clustering, etc. Depending on the chosen functionality, it varies the types of mined patterns. These data mining tasks can be classified into two categories, (i) *descriptive* tasks that characterize properties in the data, and (ii) *predictive* tasks that perform induction to make predictions. Data mining was originally applied in the discovery of frequent itemsets and association rules in

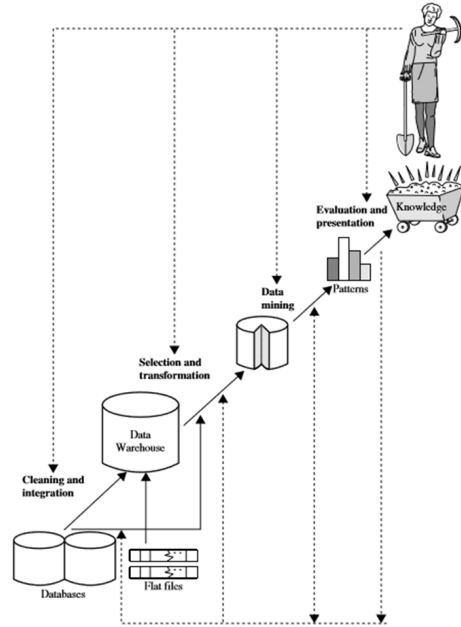


Fig. 1.5: Data mining as a step in the process of knowledge discovery. [71]

basket data, i.e. items that were frequently bought together in a retail store. Nowadays, data mining is applied wherever there are data, e.g., in business intelligence, search engines, software engineering, bioinformatics, and embedded system verification, to cite just a few areas.

### 1.3.1 Assertion mining and colored strings

Assertion mining is a task of the embedded systems verification process where predictive data mining tasks are applied to extract knowledge from simulation traces of a device. Knowledge is represented in the form of assertions<sup>1</sup> that describe behaviors of the device. The aim is to verify that the implemented functionalities of the device are correct with respect to its model. Furthermore, the mined assertions improves the set of assertions used in future steps of the design process of the device, in the context of so-called assertion based design [63].

*Example 4.* Let us consider a device that takes in input two integers  $a$  and  $b$  and produce as output their product  $a \times b$ . A possible assertion on the behavior of this device is that “it is always true that if either  $a$  or  $b$  are 0 then the result is 0”. Another possible assertion is that “it is always true that if  $a$  is 1 then the result is  $b$ ”, or is that “it is always true that if one of  $a$  and  $b$  is negative and one of  $a$  and  $b$  is positive then the result is negative”.

Designers typically write assertions by hand. It might take months to obtain a set of assertions that is small and effective (i.e. it covers all functionalities of the device) [63].

<sup>1</sup> Assertions are logic formulae expressed in temporal logics such as Linear Temporal Logic (LTL) or Computation Tree Logic (CTL).

Assertion mining automatically generates assertions from simulation traces to help designers with the verification process. For this purpose, it is possible to model the assertion mining process as searching for patterns in a colored string (See Chapter 5 for formal definitions).

Motivated by the assertion mining problem, in Chapter 5 we propose two pattern discovery problems on colored strings. We provide upper bounds on the number of the possible mined patterns. We present a baseline algorithmic solution that solves both problems. We refine the baseline solution to specifically solve one of the two problems. Finally, we present an experimental evaluation of the proposed approaches over both synthetic and real datasets.

### 1.3.2 Causality discovery and greedy minimum-entropy coupling

A different view of extracting knowledge from data is from the information-theoretic perspective of Pearl’s model of causality. By causality, we mean the dependency relation between two variables: “A variable  $X$  is a cause of a variable  $Y$  if  $Y$  in any way relies on  $X$  for its value” [106]. Here the knowledge discovery process wants to infer the causality direction between  $X$  and  $Y$ , i.e., if  $X$  causes  $Y$  or vice versa. In Pearl’s model of causality [105], given two random variables  $X$  and  $Y$ , if  $X$  causes  $Y$  then there exists an exogenous random variable  $E$  independent of  $X$  and a function  $f$  such that  $Y$  is function of  $X$  and  $E$ . It is possible to find pairs of functions  $f$  and  $g$  such that both  $Y$  is function of  $X$  and  $E$ , and  $X$  is function of  $Y$  and  $E'$ . In order to identify the correct causal direction between  $X$  and  $Y$ , it has been postulated that in the true causal direction, the entropy of the exogenous random variable  $E$  is small [83]. The problem of evaluating the entropy of the exogenous random variable  $E$ , is equivalent to finding the minimum entropy joint distribution of  $X$  and  $Y$ .

We focus on algorithms for minimum entropy coupling in Chapter 6. Minimum entropy coupling is a central issue not only in causality discovery, but also in the information-theoretic analysis of clustering and channel quantization. Since the problem is known to be NP-hard, several heuristics and approximation algorithms have recently been proposed. We start by giving a novel analysis of a heuristics proposed in [83]. We are able to show that, in fact, this algorithm guarantees a 1-bit additive approximation. Leveraging on the analytic tools employed in the previous result, we then provide a general characterization of a class of algorithms — which does not include the one in [83]— that guarantees 1-bit additive approximation for the problem. This criterion may be of independent interest in other contexts where couplings are used.



## Lempel-Ziv 77 decompression

This chapter is devoted to the Lempel-Ziv 77 decompression algorithm.

Lempel-Ziv (LZ) parsing (or factorization), often referred to as LZ77, has been the subject of hundreds of papers in the past 40 years. The technique is not only foundational to data compression, where it is the basis for several widely-used compression tools (e.g., `gzip`, `LZ4`, `Snappy`), but is also central to many compressed data structures (e.g., [12, 89]) and efficient string processing algorithms (e.g., [10, 86]).

Because obtaining the parsing is a computational bottleneck in these applications, efficient LZ parsing algorithms have been heavily studied, especially recently [4, 13, 68, 80]. There has been considerably less work on decompression — i.e., obtaining the original string given the parsing — most probably because the natural (i.e. naive) decompression algorithm is so simple and fast in practice. The naive algorithm, however, uses  $O(n)$  working space as it may make accesses to all areas of the string it is decompressing. This may be unacceptable in scenarios where RAM is at a premium, such as on small devices, or when the entire decompressed string cannot be stored and is instead processed as it is decompressed in a streaming manner by another process.

In this chapter we study algorithms to efficiently decompress strings from the LZ parsing that use working memory proportional to the size,  $z$ , of the parsing itself, not that of the output string,  $n$ , as in the naive algorithm. The only work we are aware of on this problem is recent and due to Bille et al. [16], who describe an algorithm using  $O(n \log^\delta \sigma)$  time and  $O(z \log^{1-\delta} \sigma)$  space for any  $0 \leq \delta \leq 1$ .

The main contribution of this chapter is an implementation and experimental analysis of Bille et al.'s algorithm. Our results show that when implemented as described, the approach is extremely slow in practice compared to the naive decompression algorithm, and has constant factors in its space usage that lead to no space advantage either. To remedy this we introduce several novel optimizations that drastically improve performance and lead to relevant space-time tradeoffs on all datasets we tested.

The remainder of this chapter is organized as follows. The next section lays down notation and defines basic concepts. Then, in Section 2.2, we describe the approach of Bille et al. Our refinements and practical optimizations to that approach are described in Section 2.3. Section 2.4 is devoted to experimental results and analysis.

The contents of this chapter have been published in [111].

## 2.1 Preliminaries

A string  $S = S[1..n] = S[1]S[2] \cdots S[n]$  of length  $n = |S|$  is a sequence of characters (or symbols) over an alphabet  $\Sigma$  of size  $\sigma = |\Sigma|$ . We denote with  $\varepsilon$  the empty string. We denote with  $S[i..j]$  a substring (or factor) of  $S$  starting at position  $i$  and ending in position  $j$ , with  $S[i..j] = \varepsilon$  if  $i > j$ . We call  $S[1..j]$  the  $j$ -th prefix of  $S$ , and  $S[i..n]$  the  $i$ -th suffix of  $S$ . For ease of exposition we assume, as in [56], that  $S$  is prefixed by  $\Sigma = \{c_1, \dots, c_\sigma\}$  in the negative positions, i.e., for all  $i = 1, \dots, \sigma$ ,  $S[-i] = c_i$  for each  $c_i \in \Sigma$  and  $S[0] = \$ \notin \Sigma$ .

The *Lempel-Ziv (LZ) factorization* of  $S$  is a decomposition of  $S$  into factors  $S[1..n] = S[u_1..u_1+\ell_1-1]S[u_2..u_2+\ell_2-1] \cdots S[u_i..u_i+\ell_i-1] \cdots S[u_z..u_z+\ell_z-1]$ , where  $u_1 = 1$  and  $u_i = u_{i-1} + \ell_{i-1}$  for  $i = 2, \dots, z$ . For each  $i = 1, \dots, z$ , the factor  $S[u_i..u_i + \ell_i - 1]$  is the longest prefix of  $S[u_i..n]$  that occurs at some position  $p_i < u_i$  in  $S$ , and so  $S[u_i..u_i + \ell_i - 1] = S[p_i..p_i + \ell_i - 1]$ . The LZ factorization of  $S$  can be expressed as a sequence of pairs  $(p_1, \ell_1)(p_2, \ell_2) \cdots (p_z, \ell_z) \in (\{-\sigma, \dots, n\} \times \{1, \dots, n\})^z$ . We refer to  $S[u_i..u_i + \ell_i - 1]$  as the  $i$ -th *phrase* of the factorization, to  $S[p_i..p_i + \ell_i - 1]$  as the *source* of the  $i$ -th phrase, and  $\ell_i$  its length. We require that  $p_i + \ell_i \leq u_i$  so that phrases and their sources do not overlap<sup>1</sup>.

## 2.2 Bille et al.'s algorithm

The key device in Bille et al.'s LZ decoding algorithm [16] is the so-called  $\tau$ -context, which, essentially, is the set of positions that are within  $\tau$  of an LZ phrase boundary. Formally, let  $\tau$  be a positive integer. The  $\tau$ -context of a string  $S$  (induced by the LZ factorization  $Z$  of  $S$ ) is the set of positions  $j$  where either  $j \leq 0$  or there is some  $k$  such that  $u_k - \tau < j < u_k + \tau$ . If positions  $i$  through  $j$  are in the  $\tau$ -context of  $S$ , then we simply say “ $S[i..j]$  is in the  $\tau$ -context of  $S$ ”.

We can define a string that contains all the characters at positions in the  $\tau$ -context. Namely, the  $\tau$ -context string of  $S$ , denoted by  $S^\tau$ , is the subsequence of  $S$  that includes  $S[j]$  if and only if  $j$  is in the  $\tau$ -context of  $S$ .

In [16, Lemma 9] it is shown how to map positions from the  $\tau$ -context of  $S$  to  $S^\tau$ .

**Lemma 2.1.** *Let  $Z$  be an LZ factorization of string  $S[1..n]$  with  $z$  phrases and let  $\tau$  be a positive integer. Given  $t$  sorted positions  $1 \leq a_1 \leq \dots \leq a_t \leq n$  in the  $\tau$ -context of  $S$  we can compute the corresponding positions  $b_1, \dots, b_t$  in  $S^\tau$  in  $O(z + t)$  time and space.*

Moreover, in [16, Algorithm 1], the authors show how to translate each substring of  $S$  of length at most  $\tau$  to a substring in  $S^\tau$ . To decompress a substring of  $S$  we split it into chunks of length at most  $\tau$  and extract each chunk by repeatedly mapping it into the sources of the phrases until they can be retrieved as substrings of  $S^\tau$ .

The main idea of the algorithm, given a position  $a$  in  $S$ , is to move the position  $a$  to the source of the phrase covering it, until it falls into the last  $\tau$  elements of a phrase, at which point it is guaranteed that the position has at least  $\tau$  elements on the right belonging to the  $\tau$ -context. Moreover, while moving the position we guarantee that the substring of length at most  $\tau$  starting in the new position, say  $a'$ , is equivalent to the original substring of length at most  $\tau$  starting at the original position  $a$ . Namely,  $S[a..a + \tau - 1] = S[a'..a' + \tau - 1]$ . This leads to the following lemma.

<sup>1</sup> We inherit this restriction from Bille et al. [16].



**Lemma 2.2.** *Given positive integer  $\tau$  and string  $S[1..n]$ , let  $Z$  be the LZ factorization of  $S$  with  $z$  phrases. Given  $t = O(z)$  positions  $a_1, \dots, a_t$  in  $S$ , we can compute positions  $b_1, \dots, b_t$  in the  $\tau$ -context of  $S$  in  $O(z \log n)$  time and  $O(z)$  space, such that  $b_i \leq a_i$  and  $S[a_i.. \min(a_i + \tau - 1, n)] = S[b_i..b_i + (\min(a_i + \tau - 1, n) - a_i + 1)]$ , for all  $i = 1, \dots, t$ .*

To achieve this result, a mergeable dictionary data structure [73] is used. A *mergeable dictionary* maintains a dynamic collection  $\mathcal{G}$  of  $\ell$  sets  $\{G_1, G_2, \dots, G_\ell\}$  of  $t$  elements from an ordered universe  $\mathcal{U} = \{1, \dots, |\mathcal{U}|\}$ , under the operations:

- $(A, B) \leftarrow \text{split}(G, x)$ : Splits  $G \in \mathcal{G}$  into two sets  $A = \{y \in G \mid y \leq x\}$  and  $B = \{y \in G \mid y > x\}$ .  $G$  is removed from  $\mathcal{G}$  while  $A$  and  $B$  are inserted.
- $C \leftarrow \text{merge}(A, B)$ : Creates  $C = A \cup B$ .  $C$  is inserted into  $\mathcal{G}$ ,  $A$  and  $B$  are removed.
- $G' \leftarrow \text{shift}(G, x)$  for some  $x$  such that  $y + x \in \mathcal{U}$  for each  $y \in G$ : Creates the set  $G' = \{y + x \mid y \in G\}$ .  $G$  is removed from  $\mathcal{G}$  while  $G'$  is inserted.

in worst case  $O(t \log |\mathcal{U}|)$  time and using  $O(t)$  space.

In order to decompress the text  $S$  in  $O(z)$  working space, we must also compute the  $\tau$ -context string  $S^\tau$  from  $Z$  in  $O(z)$  working space. It is possible to compute an LZ factorization  $Z^\tau$  of  $S^\tau$  directly from  $Z$  by splitting every phrase of  $Z$  into two phrases of  $Z^\tau$  consisting of the first and last  $O(\tau)$  elements, respectively. Then, using Lemma 2.2 we find a substring in  $S^\tau$  that is identical to those phrases. Finally, using Lemma 2.1, we find the sources of those phrases as positions in the  $\tau$ -context string  $S^\tau$ . We can summarize it in the following lemma.

**Lemma 2.3.** *Let  $Z$  be the LZ factorization of  $S[1..n]$  with  $z$  phrases. We can construct an LZ factorization  $Z^\tau$  of  $S^\tau$  with  $O(z)$  phrases in  $O(z \log n)$  time and  $O(z)$  space.*

We then decompress the LZ factorization  $Z^\tau$  of the  $\tau$ -context string  $S^\tau$  naively in  $O(z\tau)$  time. A string of length  $z\tau$  can be stored in  $O(z\tau \frac{\log \sigma}{\log n})$  using word packing.

**Lemma 2.4.** *Let  $\tau$  be a positive integer and let  $Z$  be the LZ factorization of the text  $S$  of length  $n$  over an alphabet of size  $\sigma$  with  $z$  phrases. We can construct and store the  $\tau$ -context of  $S$  in  $O(z(\log n + \tau))$  time and  $O(z\tau \frac{\log \sigma}{\log n})$  space.*

The decompression algorithm is based on how to decompress a substring of the text  $S$  of length  $\ell$ . First we split the substring into consecutive substrings of length  $\tau$ , then we process a batch of  $z$  substrings at a time. Using Lemma 2.2 we can find a substring  $s'$  in the  $\tau$ -context of  $S$  for every substring  $s$  in the batch.

**Theorem 2.1.** *Let  $S[1..n]$  be a string compressed into an LZ factorization with  $z$  phrases and let  $\tau = k \log n$  for some positive integer  $k$ . It is possible to decompress a substring of  $S$  of length  $\ell$  in  $O(\ell + \frac{\ell}{k} + z \log n)$  time and  $O(zk \log \sigma)$  space.*

## 2.3 Practical Small Space LZ Decompression

We implemented Bille et al.'s approach as described in the previous section, using the data structure of Karczmarz [77], extended to support the `shift` operation, as the mergeable dictionary. The results were extremely disappointing: the algorithm took 266 seconds and consumed 248Mb of RAM to decompress a file of  $z = 35319$  phrases

of uncompressed size 92Mb. The naive algorithm, in contrast, took just 0.20 seconds, using 93Mb of RAM.

Subsequent profiling showed that the main bottleneck in our implementation was the mergeable dictionary operations, and so we set about designing a fast, practical replacement for our implementation of [77]. This alternative mergeable dictionary is the subject of the next subsection.

### 2.3.1 A Fast, Practical Mergeable Dictionary

In this section we show how we replaced the mergeable dictionary used in [16, Algorithm 1] with an ad-hoc data structure  $\mathcal{B}$  that holds  $t$  elements from an ordered universe  $\mathcal{U} = \{1, \dots, |\mathcal{U}|\}$  with an attached satellite information  $r(\cdot)$  (i.e. the rank with which the elements are inserted into  $\mathcal{B}$ ), under the operations:

- $\text{insert}(x, \tilde{r})$ : insert an element  $x \in \mathcal{U}$  in  $\mathcal{B}$  with  $r(x) = \tilde{r}$ .
- $\text{report}(a, c, O)$  for some  $1 \leq a < c \leq |\mathcal{U}|$ : for each  $y \in \mathcal{B}$  such that  $a \leq y \leq c$  stores  $y$  in position  $r(y)$  in the output array  $O$ .  $y$  is removed from  $\mathcal{B}$ .
- $\text{shift}(a, c, x)$  for some  $1 \leq a < c \leq |\mathcal{U}|$  and some negative integer  $x$  such that  $c + x < a$  and  $a + x, c + x \in \mathcal{U}$ : for each  $y \in \mathcal{B}$  such that  $a \leq y \leq c$  insert  $y + x$  in  $\mathcal{B}$  with  $r(y + x) = r(y)$ , while  $y$  is removed from  $\mathcal{B}$ .

Let  $m$  be the smallest element involved by the operation  $\text{report}(a, c, O)$  or by the operation  $\text{shift}(a, c, x)$ , i.e.  $m = a$ . We require that every future operation  $\text{insert}$ ,  $\text{report}$  or  $\text{shift}$  does not involve any element greater than or equals to  $m$ , namely it is possible to  $\text{insert}(x, \tilde{r})$  only if  $x < m$ , it is possible to  $\text{report}(a, c, O)$  only if  $c < m$  and it is possible to  $\text{shift}(a, c, x)$  only if  $c < m$ .

We represent the data structure  $\mathcal{B}$  as follows. Let  $b$  be a positive integer. We divide the universe  $\mathcal{U}$  into  $\lfloor \frac{|\mathcal{U}|}{b} \rfloor + 1$  buckets of length  $b$  such that for all  $i = 0, \dots, \lfloor \frac{|\mathcal{U}|}{b} \rfloor$  an element  $x \in \mathcal{U}$  is contained in the  $i$ -th bucket if  $b \cdot i \leq x \leq b \cdot (i + 1) - 1$ .

We represent each bucket as a dynamic array which keeps a position  $x$  into the universe  $\mathcal{U}$  storing the relative position of  $x$  within the bucket, i.e. storing the remainder  $x' = x \bmod b$ . In order to reconstruct the original value  $x$ , given the remainder  $x'$  stored in the  $i$ -th bucket, we can retrieve  $x$  as  $x = b \cdot i + x'$ . Note that elements stored within the same bucket are not in order. We now describe how each operation is implemented, when  $\mathcal{B}$  holds  $t$  elements.

**Insert.** Let  $x$  be an element from the universe  $\mathcal{U}$  and let  $\tilde{r}$  its satellite information. We insert  $x$  into  $\mathcal{B}$  inserting the element  $x \bmod b$  to the dynamic array representing the  $\lfloor \frac{x}{b} \rfloor$ -th bucket, followed by its satellite information  $\tilde{r}$ . This operation takes  $O(1)$  time to be performed.

**Report.** Let  $a$  and  $c$  be two integers such that  $1 \leq a < c \leq |\mathcal{U}|$  and let  $O$  be an array. To report all the elements  $y$  in  $\mathcal{B}$  such that  $a \leq y \leq c$  we start scanning from the bucket  $\lfloor \frac{c}{b} \rfloor$  down to the bucket  $\lfloor \frac{a}{b} \rfloor$ . For every remainder  $y'$  in the buckets, we retrieve the corresponding element  $y$  and we put  $O[r(y)] = y$ . To perform the  $\text{report}$  operation we spent  $O(t)$  time.

**Shift.** Let  $a, c, x$  be three integers such that  $a + x, c + x \in \mathcal{U}$  and  $c + x < a$ . To shift by  $x$  all the elements  $y$  in  $\mathcal{B}$  such that  $a \leq y \leq c$ , as for the operation  $\text{report}$ , we start scanning from the bucket  $\lfloor \frac{c}{b} \rfloor$  down to the bucket  $\lfloor \frac{a}{b} \rfloor$ . For every remainder  $y'$  in the buckets, we retrieve the corresponding element  $y$ . If  $\lfloor \frac{y+x}{b} \rfloor$  leads to the same bucket as  $y$ , the remainder  $y'$  is updated to the current value  $(y + x) \bmod b$ . Otherwise, we  $\text{insert}(y + x, r(y))$ . To perform  $\text{shift}$  takes  $O(t)$  time.

To remove elements from  $\mathcal{B}$ , as required, after each operation `report` and `shift`, we free all the memory used for buckets with an index greater than  $\lfloor \frac{m}{b} \rfloor$ , where  $m$  is the minimum element involved. The elements belonging to the  $\lfloor \frac{m}{b} \rfloor$ -th bucket are excluded from further computations, by checking whether  $a \leq y \leq c$ , while performing the operations `report`( $a, c, O$ ) and `shift`( $a, c, x$ ).

For each bucket, we store its length and a pointer to its first element. Since elements are removed releasing the memory, we can only insert elements in the dynamic arrays. Then, we can implicitly store the capacity of the dynamic array using its length, i.e., the capacity is the upper power of 2 of the length.

To analyze the space complexity of the data structure, we consider only the case when all the `insert` operations are performed at the beginning. Let us consider  $\mathcal{B}$  after  $t$  `insert` operations. At this point, the `shift` operation is the only operation that can insert new elements in the dynamic arrays. Let us consider the `shift`( $a, c, x$ ) operation on  $\mathcal{B}$ , for some  $1 \leq a < c \leq |\mathcal{U}|$ . Let  $y$  be an element in the interval  $a \leq y \leq c$  that belongs to the  $i$ -th bucket. We can observe that the `shift` operation can, either create an element  $y'$  in a bucket  $j < \lfloor \frac{a}{b} \rfloor$ , or it modify the element in the  $i$ -th bucket. In the latter case, we have that the space consumption remains unchanged. In the former case we have that one element is created and the element  $y$  either is deleted, or it will be deleted before the element  $y'$  will be involved in a `shift` operation. There are  $O(\frac{|\mathcal{U}|}{b})$  buckets of length  $b$  and  $O(t)$  elements in the buckets. Then, we can conclude that the space complexity of the data structure is  $O(t + \frac{|\mathcal{U}|}{b})$ . We summarize this in the following lemma.

**Lemma 2.5.** *The data structure  $\mathcal{B}$  holding  $t$  elements (i.e., after  $t$  `insert` performed at the beginning), supports any sequence of  $k$  `shift` and `report` operations in worst case  $O(t \cdot k)$  time and  $O(t + \frac{|\mathcal{U}|}{b})$  space.*

We are now ready to use the  $\mathcal{B}$  data structure and rewrite Lemma 2.2.

**Lemma 2.6.** *Given a positive integer  $\tau$  and a text  $S$  of length  $n$ , let  $\mathcal{Z}$  be a LZ factorization of the text  $S$  with  $z$  phrases. Let  $\mathcal{B}$  be a data structure with bucket size  $b$ . Given  $t$  positions in  $S$ ,  $a_1, \dots, a_t$ , we can compute  $b_1, \dots, b_t$  positions in the  $\tau$ -context of  $S$  in  $O(z \cdot t)$  time and  $O(t + \frac{n}{b})$  space, such that  $b_i \leq a_i$  and  $S[a_i.. \min(a_i + \tau - 1, n)] = S[b_i.. b_i + (\min(a_i + \tau - 1, n) - a_i + 1)]$ , for all  $i = 1, \dots, t$ .*

*Proof.* To begin with, for each point  $a_i$  we insert an element  $x_i$  in  $\mathcal{B}$  with  $i$  as satellite information. Then we start scanning the phrases of  $\mathcal{Z}$  in the reverse order. The  $i$ -th phrase is processed as follows: (i) if the length of the current phrase is smaller than or equal to  $\tau$ , then we just skip to the next phrase, (ii) otherwise, we first report all the points that are between the beginning of the last phrase that has length greater than  $\tau$  and the first position of the last  $\tau$  elements of the current phrase, since all of those positions are in the  $\tau$ -context of  $S$ . Then, we scan all the elements that are within the current phrase, without the last  $\tau$  elements, and we shift them moving those positions to their corresponding positions in the source of the phrase, i.e. shifting each element by  $p_i - u_i$ . Once we have scanned all the phrases, we output all the elements between the beginning of the last phrase greater than  $\tau$  and the beginning of the text. At each stage a position  $x$  in  $\mathcal{B}$  is either reported or shifted.

In the former case, we have that every position inside a bucket within the beginning of the last phrase that has length greater than  $\tau$  and the first position of the last  $\tau$

elements of the current phrase, is already in the  $\tau$  context. Thus, it is followed by at least  $\tau$  characters. In the latter case, we are shifting the position  $x$  by  $p_i - q_i$ , thus moving to that position to its source. By definition of LZ factorization we have that  $S[x..x + \tau - 1] = S[x + p_i - u_i..x + p_i - u_i + \tau - 1]$ . Thus, at each stage of the algorithm we guarantee that if we move one position, the new position has the same  $\tau$  characters on the right and if we output one position, this position is in the  $\tau$ -context.

Note that for each phrase  $i$  which length is greater than  $\tau$  we perform one `report` and one `shift` and we have that  $u_i < u_i + \ell_i - \tau - 1 < u_i + \ell_i - \tau < u_j - 1 < u_j$ , where  $j$  is the index of the previous phrase which length is smaller than  $\tau$ . Furthermore, we have that  $p_i + \ell_i - \tau - 1 < u_i$  since each phrase is non overlapping. All those observations show that we are not violating the conditions for using the data structure  $\mathcal{B}$ .

We have  $t$  elements in  $\mathcal{B}$  that are inserted at the beginning and we perform  $O(z)$  `report` and one `shift` operations, thus from Lemma 2.5 we have that the worst case running time is  $O(z \cdot t)$  and we use  $O(t + \frac{n}{b})$  space.

Since Lemma 2.6 is used to compute the LZ factorization of the  $\tau$ -context directly from  $\mathcal{Z}$ . We can rewrite Lemma 2.3 according with the results of Lemma 2.6.

**Lemma 2.7.** *Let  $Z$  be the LZ factorization of string  $S[1..n]$  with  $z$  phrases over an alphabet of size  $\sigma$  and let  $\mathcal{B}$  be a data structure with bucket size  $b$ . We can construct an LZ factorization  $\mathcal{Z}^\tau$  of  $S^\tau$  with  $O(z)$  phrases in  $O(z^2)$  time and  $O(z + \frac{n+\sigma}{b})$  space.*

### 2.3.2 Hybrid Decompression

We now describe a further performance optimization, which can be seen as a hybrid of the naive and  $\tau$ -context based approaches.

Our first observation was that if, in the course of decompressing a chunk using the  $\tau$ -context, we encounter a phrase that has its source inside the current chunk, then we can decompress that phrase using the naive algorithm, provided we decompress it after the characters of the chunk that make up its source have been decompressed. We say such a phrase has an *in-chunk source*. Our strategy then is to decompress all phrases not having in-chunk sources with the  $\tau$ -context algorithm, and then decompress those that do using the (relatively much faster) naive algorithm.

We can extend the applicability of this idea at a slight (user controllable) increase in memory usage. In particular, we enhance the decompression algorithm with a cache of previously decompressed pieces of text. Now, as well as decompressing phrases with in-chunk sources, whenever we encounter a phrase that has its source in cache, we can decompress it naively. The next difficulty is to try to ensure that the pieces of text we keep in the cache will be heavily used.

To this end, we logically divide the text into blocks of length  $s$ . In a preprocessing phase we count, for each block, the number of times a character contained in the block is contained in a source for some phrase. The total count, which we call the block's *utility*, gives the number of times the block would be accessed by the naive algorithm. We maintain block utility as phrases are decompressed (when a phrase of length  $\ell$  that falls in block  $x$  is decompressed we decrease  $x$ 's utility by  $\ell$ ) in order to heuristically keep blocks with high remaining utility in cache to increase the chance of decompressing a phrase from cache. We update the cache each time we decompress a chunk of length  $z\tau$ .

Decompression now operates as follows.  $S$  is logically divided into blocks of length  $s$ . The initial utility of each block is computed with a scan over the phrases. As before, we decompress  $S$  in chunks of  $z\tau$  characters, right to left. For each chunk, scan all the phrases that compose the chunk. If a phrase’s source is in the current chunk, skip to the next phrase; otherwise, check if the blocks composing the phrase are stored in cache and, if so, we copy characters from cache to array storing the current partially decompressed chunk. If the blocks are not found in cache, we decompress the phrase using the  $\tau$ -context. We later rescan any phrases that were skipped — these must have their sources inside the current chunk and so can be naively decoded. Each time we decompress a substring we update its containing blocks’ utility. We then output the decompressed chunk and update cache as follows.

If  $c$  is the total cache size then we can store at most  $\lfloor c/s \rfloor$  blocks in it. After decoding a chunk we want to update cache to contain the  $\lfloor c/s \rfloor$  blocks with highest utility among those already in cache and those within the chunk. We maintain a minheap of (utility, block ID) pairs, keyed on utility. After decoding a chunk we iterate over all its blocks, replacing the heap root whenever we see a block with utility greater than it. This takes  $O(\log \lfloor c/s \rfloor)$  time as we are only manipulating block IDs, not blocks, and  $O(\lfloor z\tau/s \rfloor \log \lfloor c/s \rfloor)$  over a whole chunk. We then replace old blocks in cache with new blocks in  $O(c)$  time, for total time  $O(c + \lfloor z\tau/s \rfloor \log \lfloor c/s \rfloor)$ .

## 2.4 Experimental Results

We implemented variants of the decompression algorithms described in the previous sections and measured their performance on several real-world datasets.

### 2.4.1 Setup.

We performed experiments on a 2.10 GHz Intel® Xeon® E7-4830 v3 CPU equipped with 30 MiB L3 cache and 1.5 TiB of DDR5 main memory. Only a single thread of execution was used. The OS was Linux (Ubuntu 14.04, 64bit) running kernel 3.19.0. Programs were compiled using g++ 5.4 with `-O3 -msse4.2` options.

### 2.4.2 Data.

For our experiments we use real-world texts taken from the Pizza&Chilli corpus (<http://pizzachili.dcc.uchile.cl/repcorpus.html>). See Table 2.1.

### 2.4.3 Algorithms.

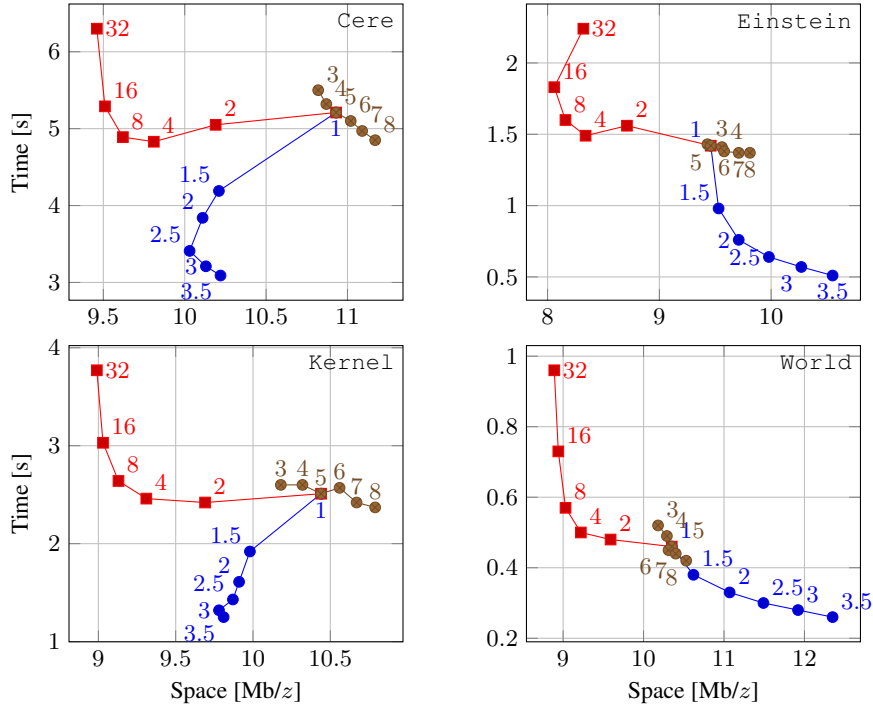
We compared the following different setups of our implementation:

- `plain`: the algorithm described in [16] using [77] for the mergeable dictionary.
- `bucket`: as `plain`, but using the mergeable dictionary described in Section 2.3.
- `cache-o`: as `bucket`, with cache.
- `cache-i`: as `bucket`, with in-chunk source decompression.
- `cache-a`: as `bucket`, with both cache and in chunk source decompression.

We can tune the following parameters:  $\tau$ , the length of the pieces of phrases that compose the  $\tau$ -context;  $b$ , the size of the buckets of the text  $S$ ;  $c$ , the total cahce size.

Name	Description	$\sigma$	$n/10^8$	$n/z$
cere	Baking yeast genomes	5	4.61	271.20
einstein	Wikipedia articles in German	139	0.92	2626.30
kernel	Linux Kernel sources	160	2.57	324.72
world	CIA World Leaders files	89	0.47	266.91

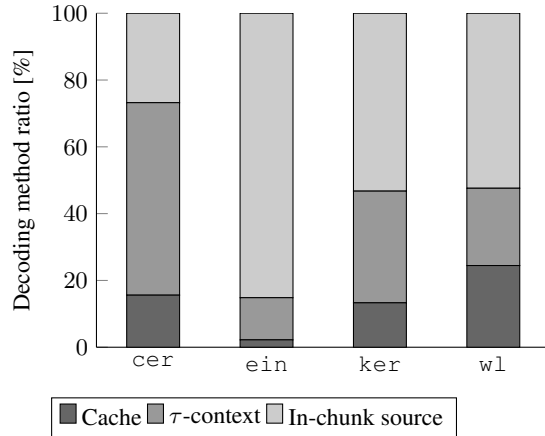
**Table 2.1:** Files used in the experiments. The value of  $n/z$  refers to the average length of a phrase in the LZ factorization and it is included as a measure of repetitiveness of the data.



**Fig. 2.1:** Each plot shows three different trends of the space-time consumption obtained varying three parameters:  $\tau$ -context length (—●—), block size (—■—), cache size (—●—). Note when one parameter is varying, the other two parameters are fixed to default values, i.e.  $\tau = \log n$ ,  $b = n/z$ , and  $c = 5\text{Mb}$ . The numbers near each point refer to the value of the parameter we are varying, e.g. the value 1.5 for the  $\tau$ -context length (—●—) means that the value of  $\tau = 1.5 \log n$ . The value 16 for the block size (—■—) means that the value of  $b = 16n/z$ . The value 5 for the cache size (—●—) means that the value of  $c = 5\text{Mb}$ .

	naive	plain	bucket	cache-o	cache-i	cache-a	cache-a*
cere	0.23 (35)	121 (21)	7.07 (10)	6.13 (11)	5.94 (10)	5.21 (11)	3.09 (10)
einstein	0.20 (329)	266 (898)	8.54 (11)	7.41 (12)	1.60 (8)	1.42 (9)	0.51 (11)
kernel	0.11 (42)	94 (4)	5.72 (11)	4.30 (11)	3.11 (9)	2.51 (10)	1.25 (10)
world	0.02 (34)	23 (22)	1.13 (11)	0.84 (11)	0.68 (10)	0.46 (10)	0.26 (12)

**Table 2.2:** Time in secs (in brackets memory usage in Mb/z) for the algorithms on each data set. Parameters used are  $\tau = \log n$ ,  $b = n/z$ ,  $c = 5\text{Mb}$ , except cache-a\* has  $\tau = 3.5 \log n$ .



**Fig. 2.2:** Each bar shows the percentage of each data set decoded that is by each part of algorithm `cache-a` (i.e., using cache, using the  $\tau$ -context or decoding naively from the current chunk). The parameters used are  $\tau = \log n$ ,  $b = n/z$ , and  $c = 5\text{Mb}$ .

#### 2.4.4 Results.

Runtime and memory usage is reported in Table 2.2. `cache-a*` represents a faster parameter configuration for the algorithm `cache-a`, on each dataset. `naive` decompression is always fastest, but always uses significantly more space than the other algorithms. Results for `einstein` show that the difference in memory usage can be enormous — `cache-a*` uses 30 times less memory than `naive`, and is only about two times slower.

#### 2.4.5 Effects of cache and in-chunk decompression.

Among the new algorithms: `cache-i` is always faster than `cache-o`, but the combination of these techniques in `cache-a` is always faster still, showing that both types of hybrid decompression matter. `cache-a*` is always at least two times faster than the algorithm `bucket`, using roughly the same amount of memory — demonstrating the effectiveness of hybrid decompression. Indeed, Figure 2.2 illustrates use of in-block sources and cache is generally correlated with faster decompression.

#### 2.4.6 Parameter Tuning.

In Figure 2.1 for each dataset, we report three different trends of the performance of the algorithm `cache-a` varying three parameters, i.e.  $\tau$ -context length, block size and cache size. For each one of these parameters, it is possible to identify a common trend among all data sets. The first relation is between  $\tau$  and runtime, the bigger the  $\tau$  is, the faster the algorithm is. On the other hand, the effect of  $\tau$  on the memory consumption seems to be data dependent for the smaller values of  $\tau$  while for bigger values of  $\tau$ , the memory consumption seems to grow linearly in  $\tau$ . Increasing bucket size  $b$  slows down the `cache-a` algorithm while decreasing memory consumption. The same trend can be seen in the variation of the cache size  $c$ , though the effect of this variation is smaller.





## Ulam-Rényi game

This chapter is devoted to the study of the Ulam-Rényi game with multi-interval questions. In the **Ulam-Rényi game with multi-interval questions**, two players, called Questioner and Responder—for reasons which will become immediately clear—fix three integer parameters:  $m \geq 0$ ,  $e \geq 0$  and  $k \geq 1$ . Then, Responder chooses a number  $x$  from the set  $\mathcal{U} = \{0, 1, \dots, 2^m - 1\}$ , and keeps it secret to Questioner. The task of Questioner is to identify the *secret* number  $x$  chosen by Responder asking  $k$ -interval queries. These are *yes-no* membership questions of the type “Does  $x$  belong to  $Q$ ?” where  $Q$  is any subset of the search space *which can be expressed as the union of  $\leq k$  intervals*. We identify a question with the subset  $Q$ . Therefore, the set of allowed questions is given by the family of sets:

$$\mathcal{T} = \left\{ \bigcup_{i=1}^k \{a_i, a_i + 1, \dots, b_i\} \mid 0 \leq a_1 \leq b_1 \leq a_2 \leq b_2 \leq \dots \leq a_k \leq b_k < 2^m \right\}.$$

With the aim of making Questioner’s search as long as possible, Responder can adversarially lie up to  $e$  times during the game i.e., by answering *yes* to a question whose correct answer is *no* or vice versa.

For any  $m$  and  $e$  let  $N_{\min}(2^m, e) = \min\{q \mid 2^{q-m} \geq \sum_{i=0}^e \binom{q}{i}\}$ . It is known (see, e.g., [15]) that in a game over a search space of cardinality  $n = 2^m$  and  $e$  lies allowed to Responder,  $N_{\min}(2^m, e)$  is a lower bound on the number of questions that Questioner has to ask in order to be sure to identify Responder’s secret number. This lower bound holds in the version of the game in which questions can refer to any subset, without the restriction to  $k$ -interval queries. A fortiori, the lower bound holds for the multi-interval game for any value of  $k$ . Strategies of size  $N_{\min}(2^m, e)$ , i.e., matching the information theoretic lower bound, are called *perfect*.

It is known that for any  $e \geq 0$  and up to finitely many exceptional values of  $m$ , Questioner can infallibly discover Responder’s secret number asking  $N_{\min}(2^m, e)$  questions. However, in general such *perfect* strategies rely on the availability of arbitrary subset questions [127]. When the cardinality of the search space is  $2^m$  the description of an arbitrary subset query requires  $2^m$  bits. Moreover, in order to implement the known strategies using  $N_{\min}(2^m, e)$  queries,  $\Theta(e2^m)$  bits are necessary to record the intermediate *states* of the game (see the next section for the details). In contrast, a  $k$ -interval-query can be expressed by simply providing the boundaries of the  $k$ -intervals defining the question, hence reducing the space requirements of the strategy to only  $2k \cdot m$  bits.

On the basis of these considerations, we will focus on the following problem:

### Main question

For given  $e \geq 0$ , denote by  $k_e$  the smallest integer  $k$  such that for all sufficiently large  $m$  Questioner has a perfect strategy in the multi-interval game over the set of  $m$ -bit numbers only using  $k$ -interval queries. What is the value of  $k_e$  for  $e = 1, 2, \dots$  ?

In [99] it was proved that for  $e = 2$  for any  $m \geq 0$  (up to finitely many exceptions) there exists a searching strategy for Questioner of size  $N_{\min}(m, e)$  (hence perfect) only using 2-interval questions, and, conversely, perfect strategies which only use 1-interval questions cannot generally exist. The case  $e = 1$  is analysed in [32] where it is shown that perfect strategies exist for  $e = 1$  even when only using 1-interval questions. However, simple comparison questions, namely yes-no questions of the type “Is  $x \leq q$ ?”, are not powerful enough to provide perfect strategies for the case  $e = 1$  [9, 126].

These results show that for  $e \leq 2$ , the answer to our main question is  $k_e = e$ .

In [32] it was proved that for any  $e \geq 1$  there exists  $k = O(e^2)$  such that for all sufficiently large  $m$  Questioner can identify an  $m$ -bit number by using exactly  $N(2^m, e)$   $k$ -interval questions when Responder can lie at most  $e$  times. In [32], it is also conjectured that  $k_e = O(e)$  interval might suffice for any  $e$  and all sufficiently large  $m$ . We will refer to this as the *linearity conjecture*.

### Our result

We focus on the case  $e = 3$ . We first show that for any sufficiently large  $m$ , there exists a strategy to identify an initially unknown  $m$ -bit number when up to 3 answers are lies, which matches the information theoretic lower bound and only uses 4-interval queries. We then show how to refine our main tool to turn the above asymptotic result into a complete characterization of the instances of the Ulam-Rényi game with 4-interval question and 3 lies that admit strategies using the theoretical minimum number of questions. For this, we build upon the result of [101] and show that if there exists a strategy for the *classical* Ulam-Rényi game with 3 lies over a search space that uses the information theoretic minimum number of questions, then the same strategy can be implemented using only 4-interval questions.

With reference to the *Main question* posed above, these results show that  $k_3 \leq 4$ , which significantly improves the best previously known bound of [32] yielding  $k_3 \leq 10$ . It remains an open problem whether  $k_3 = 4$ . More interesting, our novel analytic tool (Lemma 3.2) naturally lends itself to a generalization to any fixed  $e$ . The main open question is how to generalize Theorem 3.1 in order to prove the linearity conjecture  $k_e = O(e)$ .

### Related work

The Ulam-Rényi game [114, 131] has been extensively studied in various contexts including error correction codes [3, 15, 39, 53, 108], learning [14, 27, 38], many-valued logics [39, 40], wireless networks [88], psychophysics [78, 79], and, principally, sorting

and searching in the presence of errors (for the large literature on this topic, and the several variants studied, we refer the reader to the papers [21, 40, 53, 109] and the book [31]).

The contents of this chapter have been published in [41, 42].

### 3.1 Basic facts

From now on we concentrate on the case  $e = 3$  and 4-interval questions. Let  $Q$  be the subset defining a question asked by Questioner. Let  $\bar{Q}$  be the complement of  $Q$ , i.e.,  $\bar{Q} = \{0, 1, \dots, 2^m - 1\} \setminus Q$ . If Responder answers *yes* to question  $Q$ , then we say that any number  $y \in Q$  *satisfies* the answer and any  $y \in \bar{Q}$  *falsifies* the answer. If Responder answers *no* to question  $Q$ , then we say that any number  $y \in \bar{Q}$  *satisfies* the answer and any  $y \in Q$  *falsifies* the answer.

At any stage of the game, we can partition the search space into  $e + 2$  subsets,  $(A_0, A_1, A_2, A_3, A_{>3})$ , where  $A_j$  ( $j = 0, \dots, 3$ ) is the set of numbers falsifying exactly  $j$  of Responder's answers, and therefore contains Responder's secret number if he has lied exactly  $j$  times. Moreover,  $A_{>3}$  is the set of numbers falsifying more than  $e$  of the answers. Therefore,  $A_{>3}$  is the set of numbers that cannot be the secret, because, otherwise, Responder would have lied too many times. We refer to the vector  $(A_0, A_1, A_2, A_3, A_{>3})$  as the *state* of the game, since it is a complete record of Questioner's state of knowledge on the numbers which are candidates to be Responder's secret number, as resulting from the sequence of questions/answers exchanged so far.

We will find it convenient to have an alternative perspective on the state of the game. For a state  $\sigma = (A_0, A_1, A_2, A_3, A_{>3})$  and for each  $y \in \mathcal{U}$  we define  $\sigma(y)$  as the number of answers falsified by  $y$ , truncated at 4. Then for any  $i = 0, \dots, 3$ , we have  $\sigma(y) = i$  if and only if  $y \in A_i$ . For each  $i = 0, 1, \dots, 3$ , we will also write  $\sigma^{-1}(i)$  to denote the set  $A_i$ . And define  $\sigma^{-1}(4) = A_{>3} = \bigcup_{i=0}^3 A_i$  to denote the set of numbers that as a result of the answers received cannot be Responder's secret number. From now on, we refer to a state of the game as the state  $\sigma = (A_0, \dots, A_e)$ , since the set  $A_{>e}$  can be reconstructed from the sets  $A_0, \dots, A_e$  and the universe  $\mathcal{U}$ .

**Definition 3.1 (States and Supports).** *A state is a map  $\sigma : \mathcal{U} \rightarrow \{0, 1, 2, 3, 4\}$ . The type of  $\sigma$  is the quadruple  $\tau(\sigma) = (t_0, t_1, t_2, t_3)$  where  $t_i = |\sigma^{-1}(i)|$  for each  $i = 0, 1, 2, 3$ . The support  $\Sigma$  of  $\sigma$  is the set of all  $y \in \mathcal{U}$  such that  $\sigma(y) < 4$ . A state is final if and only if its support has cardinality at most one. The initial state  $\alpha$  is the function mapping each element of  $\mathcal{U}$  to 0.*

Let  $(A_0, \dots, A_e)$  be the current state and  $Q$  be the new question asked by Questioner. Questioner's new state of knowledge if Responder answers *yes* to  $Q$  is obtained by the following rules:

$$A_0 \leftarrow A_0 \cap Q \quad \text{and} \quad A_j \leftarrow (A_j \cap Q) \cup (A_{j-1} \cap \bar{Q}), \text{ for } j = 1, \dots, e$$

If Responder answers *no*, the above definitions and rules apply with  $Q$  replaced by its complement  $\bar{Q} = \{0, 1, \dots, 2^m - 1\} \setminus Q$ , i.e., answering *no* is the same as answering *yes* to the complementary question  $\bar{Q}$ .

With the above functional notation we formalise this as follows:

**Definition 3.2 (Answers and Resulting States).** Let  $\sigma$  be the current state with support  $\Sigma$  and  $Q$  be the new question asked by Questioner. Let  $b \in \{\text{yes}, \text{no}\}$  be the answer of Responder. Define the answer function  $b : \Sigma \rightarrow \{0, 1\}$  associated to question  $Q$  by stipulating that  $b(y) = 0$  if and only if  $y$  satisfies answer  $b$  to question  $Q$ . Then, the resulting state  $\sigma_b$  is given by  $\sigma_b(y) = \min\{\sigma(y) + b(y), 4\}$ .

More generally, starting from state  $\sigma$  after questions  $Q_1, \dots, Q_t$  with answers  $b_1, \dots, b_t$  the resulting state is  $\sigma_{b_1 b_2 \dots b_t} = \min\{\sigma(y) + \sum_{j=1}^t b_j(y), 4\}$ .

In particular, for  $\sigma$  being the initial state we have that the resulting state after  $t$  questions is the truncated sum of the corresponding answer functions associated to Responder's answers.

**Definition 3.3 (Strategy).** A strategy of size  $q$  is a full binary tree of depth  $q$  where each internal node  $\nu$  maps to a question  $Q_\nu$ . The left and right branch stemming out of  $\nu$  map to the function answers yes and no associated to question  $Q_\nu$ . Each leaf  $\ell$  is associated to the state  $\sigma^\ell$  resulting from the sequence of questions and answers associated to the nodes and branches on unique path from the root to  $\ell$ . In particular, if  $b_1, \dots, b_q$  are the answers/branches leading to  $\ell$  then we have  $\sigma^\ell = \alpha_{b_1 \dots b_q}$  as defined above.

The strategy is winning if and only if for all leaves  $\ell$  we have that  $\sigma^\ell$  is a final state.

We can also extend the above definition to an arbitrary starting state. Given a state  $\sigma$ , we say that a strategy  $\mathcal{S}$  of size  $q$  is winning for  $\sigma$  if for any root to leaf path in  $\mathcal{S}$  with associated answers  $b_1, \dots, b_q$  the state  $\sigma_{b_1 \dots b_q}$  is final.

We define the *character* of a state  $\sigma$  as  $ch(\sigma) = \min\{q \mid w_q(\sigma) \leq 2^q\}$ , where  $w_q(\sigma) = \sum_{j=0}^3 |\sigma^{-1}(j)| \sum_{\ell=0}^{3-j} \binom{q}{\ell}$  is referred to as the  $q$ th volume of  $\sigma$ . Intuitively, the  $q$ th volume counts the number of possible sequences of Responder's answers when there are  $q$  questions left.

We have that the lower bound  $N_{\min}(2^m, e)$ , mentioned in the introduction, coincides with the character of the initial state  $\sigma^0 = (\mathcal{U}, 0, \dots, 0)$  (see Proposition 3.1 below). Notice also that a state has character 0 if and only if it is a final state.

For a state  $\sigma$  and a question  $Q$  let  $\sigma_{yes}$  and  $\sigma_{no}$  be the resulting states according to whether Responder answers, respectively, yes or no, to question  $Q$  in state  $\sigma$ . Then, from the definition of the  $q$ th volume of a state, it follows that for each  $q \geq 1$ , we have  $w_q(\sigma) = w_{q-1}(\sigma_{yes}) + w_{q-1}(\sigma_{no})$ . A simple induction argument gives the following lower bound [15].

**Proposition 3.1.** Let  $\sigma$  be the state of the game. For any integers  $0 \leq q < ch(\sigma)$  and  $k \geq 1$ , starting from state  $\sigma$ , Questioner cannot determine Responder's secret number asking only  $q$  many  $k$ -interval-queries.

In order to finish the search within  $N_{\min}(2^m, e)$  queries, Questioner has to guarantee that each question asked induces a strict decrease of the character of the state of the game. The following lemma provides a sufficient condition for obtaining such a strict decrease of the character.

**Lemma 3.1.** Let  $\sigma$  be the current state, with  $q = ch(\sigma)$ . Let  $Q$  be Questioner's question and  $\sigma_{yes}$  and  $\sigma_{no}$  be the resulting states according to whether Responder answers, respectively, yes or no, to question  $Q$ .

If  $|w_{q-1}(\sigma_{yes}) - w_{q-1}(\sigma_{no})| \leq 1$  then it holds that  $ch(\sigma_{yes}) \leq q - 1$  and  $ch(\sigma_{no}) \leq q - 1$ .

*Proof.* Assume, w.l.o.g., that  $w_{q-1}(\sigma_{yes}) \geq w_{q-1}(\sigma_{no})$ . Then, from the hypothesis, it follows that  $w_{q-1}(\sigma_{no}) \geq w_{q-1}(\sigma_{yes}) - 1$ . By definition of character we have  $2^q \geq w_q(\sigma) = w_{q-1}(\sigma_{yes}) + w_{q-1}(\sigma_{no}) \geq 2w_{q-1}(\sigma_{yes}) - 1$ , hence,  $w_{q-1}(\sigma_{no}) \leq w_{q-1}(\sigma_{yes}) \leq 2^{q-1} + 1/2$ , which together with the integrality of the volume, implies that for both  $\sigma_{yes}$  and  $\sigma_{no}$  the  $(q - 1)$ th volume is not larger than  $2^{q-1}$ , hence their character is not larger than  $q - 1$ , as desired.  $\square$

A question which satisfies the hypothesis of Lemma 3.1 will be called *balanced*. A special case of balanced question is obtained when for a state  $\sigma = (A_0, \dots, A_e)$  the question  $Q$  is such that  $|Q \cap A_i| = |A_i|/2$  for each  $i = 0, \dots, e$ . In this case, we also call the question an *even splitting*.

**Definition 3.4 (Interval).** An interval in  $\mathcal{U}$  is either the empty set  $\emptyset$  or a set of consecutive elements  $[a, b] = \{x \in \mathcal{U} | a \leq x \leq b\}$ . The elements  $a, b$  are called the boundaries of the interval.

**Definition 3.5 (4-interval-question).** A 4-interval question (or simply a question) is any subset  $Q$  of  $\mathcal{U}$  such that  $Q = I_1 \cup I_2 \cup I_3 \cup I_4$  where for  $j = 1, 2, 3, 4$ ,  $I_j$  is an interval in  $\mathcal{U}$ .

The type of  $Q$  (w.r.t. state  $\sigma$ ), denoted by  $|Q|$ , is the quadruple  $|Q| = [a_0^Q, a_1^Q, a_2^Q, a_3^Q]$  where for each  $i = 0, 1, 2, 3$ ,  $a_i^Q = |Q \cap \sigma^{-1}(i)|$ .

Following [99] we visualize the search space as a necklace and restrict it to the set of numbers which are candidate to be the secret number, i.e., we identify  $\mathcal{U}$  with its support  $\Sigma = \mathcal{U} \cap \bigcup_{i=0}^3 A_i$ .

For any non-final state, i.e.,  $|\bigcup_{i=0}^3 A_i| > 1$ , for each  $x \in \bigcup_{i=0}^3 A_i$  we define the successor of  $x$  to be the number  $x + r \pmod{2^m}$  for the smallest  $0 < r < 2^m$  such that  $x + r \pmod{2^m} \in \bigcup_{i=0}^3 A_i$ . In particular, for the initial state, 0 is the successor of  $2^m - 1$ .

For  $a, b \in \bigcup_{i=0}^3 A_i$  we say that there is an arc from  $a$  to  $b$  and denote it by  $\langle a, b \rangle$  if the following two conditions hold: (i)  $\sigma(x) = \sigma(a)$  for each element  $x$  encountered when moving from  $a$  to  $b$  in  $\mathcal{U}$  passing from one element to its successor; (ii)  $\sigma(c) \neq \sigma(a)$  and  $\sigma(d) \neq \sigma(b)$  where  $a$  is the successor of  $c$  and  $d$  is the successor of  $b$ . We say that arc  $\langle a, b \rangle$  is on level  $\sigma(a)$  and call  $a$  and  $b$  the left and right boundary of the arc.

In words, an arc is a maximal sequence of consecutive elements lying on the same level of the state  $\sigma$ .

For the sake of definiteness, we allow an arc to be empty. Therefore, we can associate to a state  $\sigma$  a smallest sequence  $\mathcal{L}^\sigma$  of (possibly empty) arcs  $a_0, \dots, a_{r-1}$  such that for each  $i$  the levels of arcs  $a_i$  and  $a_{(i+1) \bmod r}$  differ exactly by 1. Note that, by requiring that the length  $r$  be minimum the sequence  $\mathcal{L}^\sigma$  is uniquely determined up to circular permutations.

For each  $i = 0, 1, \dots, r - 1$ , we say that arcs  $a_i$  and  $a_{(i+1) \bmod r}$  are *adjacent* (or *neighbours*). We say that  $a_i$  is a *saddle* if both adjacent arcs are on a lower level, i.e.,  $a_{(i-1) \bmod r}, a_{(i+1) \bmod r} \subseteq \sigma^{-1}(k - 1)$  and  $a_i \subseteq \sigma^{-1}(k)$  for some  $k$ .

We say that  $a_i$  is a *mode* if both adjacent arcs are on a higher level, i.e.,  $a_{(i-1) \bmod r}, a_{(i+1) \bmod r} \subseteq \sigma^{-1}(k + 1)$  and  $a_i \subseteq \sigma^{-1}(k)$  for some  $k$ .

We say that  $a_i$  is a *step* if for some  $k$ ,  $a_i \subseteq \sigma^{-1}(k)$  and either  $a_{(i-1) \bmod r} \subseteq \sigma^{-1}(k-1)$ ,  $a_{(i+1) \bmod r} \subseteq \sigma^{-1}(k+1)$  or  $a_{(i-1) \bmod r} \subseteq \sigma^{-1}(k+1)$ ,  $a_{(i+1) \bmod r} \subseteq \sigma^{-1}(k-1)$ .

Based on the above notions, we now define a well-shaped state for  $e$  lies.

**Definition 3.6.** Let  $\sigma$  be a state and  $\mathcal{L}^\sigma$  be its associated list of arcs. Then,  $\sigma$  is **well shaped** if and only if the following conditions hold:

- for  $i = 0, \dots, e - 1$ , in  $\mathcal{L}^\sigma$  there are exactly  $(2i + 1)$  arcs lying on level  $i$ .
- in  $\mathcal{L}^\sigma$  there are exactly  $e$  arcs lying on level  $e$ .

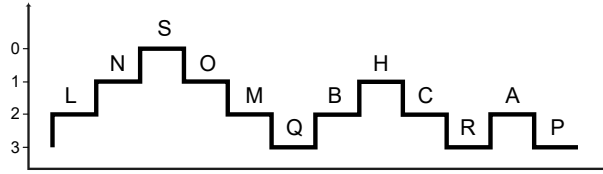
It is not hard to see that for the case  $e = 3$  under investigation, the only two feasible well-shaped states are described as follow:  $\sigma^{-1}(0)$  is an arc  $S$  in  $\Sigma$ ;  $\sigma^{-1}(1)$  is the disjoint union of three arcs  $H, N$  and  $O$  in  $\Sigma$  with  $N$  and  $O$  adjacent to  $S$ ;  $\sigma^{-1}(2)$  is the disjoint union of five arcs  $A, B, C, L$  and  $M$  in  $\Sigma$  with  $B$  and  $C$  adjacent to  $H$ ,  $L$  adjacent to  $N$ ;  $\sigma^{-1}(3)$  is the disjoint union of three arcs  $P, Q, R$  in  $\Sigma$  with  $R$  and  $P$  adjacent to  $A$ .

Starting with  $S$  and scanning  $\mathcal{U}$  with positive orientation, we can list the twelve arcs (restricted to  $\Sigma$ ) in one of the following two possibilities:

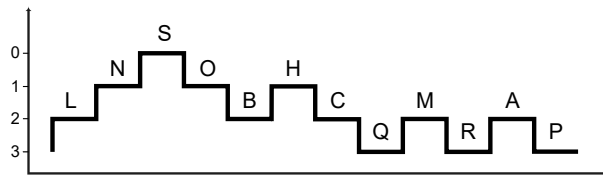
$$\sigma_1 = L^2 N^1 S^0 O^1 M^2 Q^3 B^2 H^1 C^2 R^3 A^2 P^3 \tag{3.1}$$

$$\sigma_2 = L^2 N^1 S^0 O^1 B^2 H^1 C^2 Q^3 M^2 R^3 A^2 P^3 \tag{3.2}$$

where for an arc  $X$  the notation  $X^i$  is meant to denote the fact that  $X \subseteq \sigma^{-1}(i)$ . Well-shaped states of type (3.1) and (3.2) are shown in Figures 3.1 and 3.2 respectively.



**Fig. 3.1:** A well-shaped state of type (3.1). Arcs  $S, H, A$  are *modes* at level 0, 1, 2, respectively. Arcs  $Q, R, P$  are *saddles* at level 3. The remaining arcs are *steps*. The arrow shows the positive orientation.



**Fig. 3.2:** A well-shaped state of type (3.2). Arcs  $S, H, M, A$  are *modes*. Arcs  $B, Q, R, P$  are *saddles*. The remaining arcs are *steps*. The arrow shows the positive orientation.

### 3.2 The key result - structure of perfect 4-interval strategies for 3 lies

The strategy we propose is based on the approach of Spencer [127]. We will show how to implement questions in this strategy as 4-interval questions. The main technical tool will be to show that we can define 4-interval balanced questions and also guarantee that each intermediate state is well-shaped.

In this section, we will characterise questions in terms of the ratio between the components of the question and the components of the state they are applied to. We will show conditions for the existence of questions that can be implemented using only 4 intervals and such that the resulting states are *well-shaped* whenever the state they are applied to is well shaped.

For each of them, we show how to select the exact amount of elements in the query among the arcs representing the state. Then, we prove that no other cases are allowed and finally we show that the well shapeness property of the state is preserved in both states resulting from the answer to the query.

Our main technical tool is the following theorem, whose proof is deferred to the next section.

**Theorem 3.1.** *Let  $\sigma$  be a well-shaped state of type  $\tau(\sigma) = (a_0, b_0, c_0, d_0)$ . For all integers  $0 \leq a \leq a_0$ ,  $0 \leq b \leq \lceil \frac{1}{2}b_0 \rceil$ ,  $0 \leq c \leq \lceil \frac{1}{2}c_0 \rceil$ ,  $0 \leq d \leq \lceil \frac{2}{3}d_0 \rceil$ , there exists a 4-interval question  $Q$  of type  $|Q| = [a, b, c, d]$  that we can ask in state  $\sigma$  and such that both the resulting “yes” and “no” states are well-shaped.*

Before going into the details of the proof, we now show how to employ Theorem 3.1 to show that asymptotically, every instance of the Ulam-Rényi game with 3 lies over a space of cardinality  $2^m$  admits a perfect strategy that only uses 4-interval questions. For this, we use the main result of [127] that rephrased in our setting  $e = 3$  guarantees that for all sufficiently large  $m$ , the strategy using the following two steps is perfect for the Ulam-Rényi game with 3 lies over the search space of size  $2^m$ :

1. As long as the state satisfies  $\sum_{i=0}^2 |\sigma^{-1}(i)| > 1$  ask a question  $Q$  of type  $[a_0^Q, a_1^Q, a_2^Q, a_3^Q]$  where, for  $i = 0, 1, 2$ ,  $a_i^Q \in \{\lfloor \frac{1}{2}|\sigma^{-1}(i)| \rfloor, \lceil \frac{1}{2}|\sigma^{-1}(i)| \rceil\}$  with the choice of whether to choose floor or ceiling alternating among those levels where  $|\sigma^{-1}(i)|$  is odd. The value of  $a_3^Q$  is appropriately computed, based on the choices of  $a_0^Q, a_1^Q, a_2^Q$ , in order to guarantee that the resulting question is balanced, i.e.,  $a_3^Q = \lfloor \frac{1}{2}(\sum_{j=0}^2 (|\sigma^{-1}(j)| - 2a_j^Q) \binom{q}{j} + \sigma^{-1}(3)) \rfloor$ , where  $q + 1$  is the character of  $\sigma$ ;
2. when the state satisfies  $\sum_{i=0}^2 |\sigma^{-1}(i)| \leq 1$ , ask a balanced question  $Q$  of type  $[0, 0, 0, a_3^Q]$ .

The main point in the argument of [127] is that, up to finitely many exceptions, for all  $m = \log |\mathcal{U}|$ , the value  $a_3^Q$  defined in 1. is feasible, in the sense that using the above rules yields  $0 \leq a_3^Q \leq |\sigma^{-1}(3)|$ .

We can now employ Theorem 3.1 to show that the above step can be implemented by a 4-interval-question. Let  $Q$  be the question defined in 1. Let  $\bar{Q}$  be the complementary question, and  $[a_0^{\bar{Q}}, a_1^{\bar{Q}}, a_2^{\bar{Q}}, a_3^{\bar{Q}}]$  denote its type. For  $i = 0, 1, 2$ , we have  $a_i^Q, a_i^{\bar{Q}} \leq \lfloor \frac{1}{2}|\sigma^{-1}(i)| \rfloor$ . Moreover, we have  $\min\{a_3^Q, a_3^{\bar{Q}}\} \leq \lfloor \frac{2}{3}|\sigma^{-1}(3)| \rfloor$ . Therefore, asking



$Q$  or  $\bar{Q}$  according to whether  $a_3^Q \leq a_3^{\bar{Q}}$  guarantees that the question satisfies the hypothesis of Theorem 3.1 and then it can be implemented as 4-interval question which also preserves the well-shape of the state.

The condition in 2. can also be easily guaranteed only relying on 4-interval questions. In fact, the following proposition shows that questions in point 2. are implementable by 4-interval questions, preserving the well-shape of the state.

**Proposition 3.2.** *Let  $\sigma$  be a well-shaped state with  $|\sigma^{-1}(3)| > 0$  and  $\sum_{i=0}^2 |\sigma^{-1}(i)| \leq 1$ . Let  $ch(\sigma) = q$ . Then, starting in state  $\sigma$  the Questioner can discover the Responder's secret number asking exactly  $q$  many 1-interval-queries.*

*Proof.* We prove the proposition by induction on  $q$ , the character of the state. If  $q = 1$ , the only possibility is  $\sum_{i=0}^2 |\sigma^{-1}(i)| = 0$  and  $|\sigma^{-1}(3)| = 2$ . Then, a question containing exactly one of the elements in  $\sigma^{-1}(3)$  is enough to conclude the search.

Now assume that  $q > 1$  and that the statement holds for any state with the same structure and character  $\leq q - 1$ .

If  $\sum_{i=0}^2 |\sigma^{-1}(i)| = 0$  then the solution is provided by the classical binary search in the set  $\sigma^{-1}(3)$ , which can be clearly implemented using 1-interval-queries. Notice also that the number of intervals needed to represent the new state, is never more than the number of intervals needed to represent  $\sigma$ .

Assume now that  $\sum_{i=0}^2 |\sigma^{-1}(i)| \neq 0$  and let  $j$  be the index such that  $|\sigma^{-1}(j)| = 1$ . Let  $c$  be the only element in the  $j$ th level.

Let  $\alpha = \sum_{i=0}^{3-j} \binom{q-1}{i} \leq 2^{q-1}$ . By the assumption of the character of  $\sigma$  follows that  $\sum_{i=0}^{3-j} \binom{q}{i} \leq w_q(\sigma) \leq 2^q$ , hence  $3 - j < q$  and in addition, because of the assumption  $ch(\sigma) = q$  we have  $\sum_{i=0}^{3-j} \binom{q-1}{i} + |\sigma^{-1}(3)| = w_{q-1}(\sigma) > 2^{q-1}$ .

Choose  $I$  to be the largest interval including  $c$  and  $2^{q-1} - \alpha$  other elements from the 3rd level. The above observation guarantees the existence of such interval. The possible states arising from such a question satisfy  $w_{q-1}(\sigma_{yes}) = 2^{q-1}$  and are such that  $w_{q-1}(\sigma_{no}) = w_q(\sigma) - w_{q-1}(\sigma_{yes}) \leq 2^q - 2^{q-1}$ . Hence, both states have character not larger than  $q - 1$ . It is also not hard to see that they both have a structure satisfying the hypothesis of the proposition. This proves the induction step.

Note that the number of intervals necessary to represent the new state is not larger than the initial one, then the state is also well-shaped.  $\square$

We have shown that any question in the perfect strategy of [127] can be implemented by 4-interval questions. We can summarise our discussion in the following theorem.

**Theorem 3.2.** *For all sufficiently large  $m$  in the game played over the search space  $\{0, \dots, 2^m - 1\}$  with 3 lies, there exists a perfect 4-interval strategy. In particular, the strategy uses at most  $N_{\min}(2^m, 3)$  questions and all the states of the game are well shaped, hence representable by exactly  $12 \log m$  bits (12 numbers from  $\mathcal{U}$ ).*

### 3.3 The proof of Theorem 3.1

Let  $\sigma$  be a state and  $\langle a, b \rangle$  be a non-empty arc of  $\sigma$ . We say that a question  $Q$  splits the arc  $\langle a, b \rangle$  if there exists an interval  $I$  in  $Q$  that intersects  $\langle a, b \rangle$  and contains exactly one of its boundaries  $a, b$ . In words, there is an interval in the question such that some



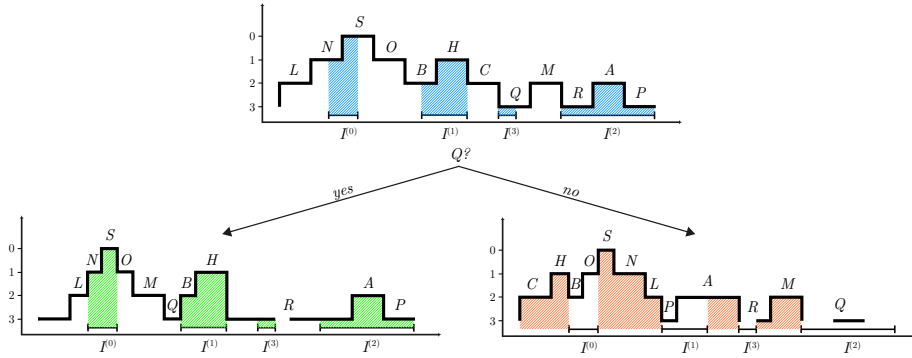
non-empty part of the arc satisfies a yes answer and some non-empty part of the arc satisfies a no answer.

If a question  $Q$  splits exactly one arc on level  $i$  of  $\sigma$  according to whether such an arc is a mode, a saddle, or a step, we say that *at level  $i$  the question  $Q$  (or, equivalently, an interval of  $Q$ ) is mode-splitting, saddle-splitting, step-splitting, respectively.*

Let  $Q$  be a *step-splitting* question at level  $i$ . Let  $\langle a, b \rangle$  be the arc at level  $i$  which is split by an interval  $I$  of  $Q$ . Then, by definition  $I$  contains exactly one of the boundaries of the arc. If  $I$  contains the boundary of the arc that flanks an arc at level  $i + 1$  we say that  $Q$  (or, equivalently, an interval of  $Q$ ) is *downward step-splitting*; if  $I$  contains the boundary of the arc that flanks an arc at level  $i - 1$  we say that  $Q$  (or, equivalently, an interval of  $Q$ ) is *upward step-splitting*.

We say that a question  $Q$  *covers entirely* the arc  $\langle a, b \rangle$  if  $[a, b]$  is contained in one of the intervals defining  $Q$ .

If a question  $Q$  covers entirely an arc on level  $i$  of  $\sigma$  according to whether such an arc is a mode, a saddle, a step, we say that *at level  $i$  the question  $Q$  (or, equivalently, an interval of  $Q$ ) is mode-covering, saddle-covering, step-covering, respectively.* Refer to Figure 3.3 for a pictorial representation of the interval types and questions effect on states.



**Fig. 3.3:** An example of state dynamics. The question  $Q$  is represented by the intervals  $I^{(0)}$ ,  $I^{(1)}$ ,  $I^{(2)}$  and  $I^{(3)}$ . The interval  $I^{(0)}$  is *mode-splitting* at level 0 and *upward step-splitting* at level 1;  $I^{(1)}$  is *mode-covering* at level 1 and *saddle-splitting* at level 2;  $I^{(2)}$  is *mode-covering* at level 2 and *saddle-covering* at level 3;  $I^{(3)}$  is *saddle-splitting* at level 3. In the resulting states the filled volumes indicate the arcs of the state remained unchanged. The  $\sigma^{yes}$ ,  $\sigma^{no}$  states are represented on the support of the original state  $\sigma$  to show how the elements belonging the lowest level disappear (the blank gaps on the shapes) from the support when they are in contradiction with more than 3 answers.

We will define conditions on the intersection between the intervals defining a question and the arcs of a (well-shaped) state  $\sigma$  such that both  $\sigma^{yes}$  and  $\sigma^{no}$  are well shaped. It turns out that these conditions can be defined locally, level by level and arc by arc.

Let  $\sigma$  be a well shaped state and let  $\mathcal{L}^\sigma$  be its associated list of arcs (including possibly empty ones) where consecutive arcs differ in level by exactly one. For each  $i = 0, 1, 2, 3$ , let  $\mathcal{L}^\sigma(i)$  denote the list of arcs on level  $i$  of  $\sigma$ ,  $|\mathcal{L}^\sigma(i)|$  denote the number of arcs on level  $i$  of  $\sigma$ .

Given a question  $Q$  asked in state  $\sigma$  we denote by  $\mathcal{L}_{yes}^\sigma$  and  $\mathcal{L}_{no}^\sigma$  the lists of arcs associated to the states  $\sigma^{yes}$  and  $\sigma^{no}$  resulting from answering *yes* and *no* to question  $Q$ .

For the rest of this section, we focus on questions  $Q$  satisfying the following property with respect to the current state  $\sigma$ :

**Property P:** For each  $i = 0, 1, 2$ , and each arc  $a \in \mathcal{L}^\sigma(i)$ , at most one interval of  $Q$  intersects  $a$ , and altogether,  $Q$  splits at most one arc on level  $i$ .

We now introduce some analytic tools to quantify the relationship between the state  $\sigma$  and the states  $\sigma^{yes}$  and  $\sigma^{no}$  resulting from the answer to  $Q$ . For each pair of consecutive arcs  $a$  and  $b$  in  $\mathcal{L}^\sigma$  we assume (for the sake of the analysis) that there is a *phantom* element  $\varepsilon_{ab}$  between  $a$  and  $b$  and sitting on level  $\max\{\ell_a, \ell_b\}$  where  $\ell_a, \ell_b$  are the levels of  $a$  and  $b$  respectively. For each arc  $x$  let us define  $x^+$  as the set containing all the elements in  $x$  and the phantom elements flanking  $x$ . Phantom elements are only used for analysing the effect of a question. So for every new question we consider only the phantom elements defined by the state where the question is asked. Moreover, as opposed to the *actual* elements and arcs of the state, which may change level as a result of the answer to the question, by definition, the level of a phantom element is fixed and (during the analysis of  $Q$ ) it remains the same also after the answer to the question  $Q$ .

Note that for each phantom element  $\varepsilon$ , and each list  $\mathcal{L} \in \{\mathcal{L}^\sigma, \mathcal{L}_{yes}^\sigma, \mathcal{L}_{no}^\sigma\}$  there is exactly one arc  $x \in \mathcal{L}$  which lies on the same level of  $\varepsilon$  and flanks it. We refer to this arc as the arc of  $\mathcal{L}$  containing  $\varepsilon$ . This might also be an empty arc that is in  $\mathcal{L}$  to guarantee that consecutive arcs differ by exactly one level.

Let  $\mathcal{E}^\sigma$  be the set of phantom elements and for each  $i = 0, 1, 2, 3$  let  $\mathcal{E}^\sigma(i)$  be the set of phantom elements on level  $i$ . It is not hard to see that we have  $|\mathcal{E}^\sigma| = |\mathcal{L}^\sigma|$ .

Fix an arc  $a \in \mathcal{L}^\sigma$ . Let  $\mathcal{A}_a^\sigma$  (resp.  $\mathcal{A}_a^{ans}$ , for  $ans \in \{yes, no\}$ ) be the set of arcs in  $\mathcal{L}^\sigma$  (resp.  $\mathcal{L}_{ans}^\sigma$ ) containing elements of  $a^+$ . For each  $i = 0, 1, 2, 3$ , let  $\mathcal{A}_a^\sigma(i)$  (resp.  $\mathcal{A}_a^{ans}(i)$ ) be the set of arcs in  $\mathcal{L}^\sigma(i)$  (resp.  $\mathcal{L}_{ans}^\sigma(i)$ ) containing elements of  $a^+$ . We have

$$\sum_{i=0}^3 \sum_{a \in \mathcal{L}^\sigma(i)} |\mathcal{A}_a^\sigma(i)| = |\mathcal{L}^\sigma| + |\mathcal{E}^\sigma| = 2|\mathcal{L}^\sigma| \quad \text{and} \quad \sum_{a \in \mathcal{L}^\sigma} |\mathcal{A}_a^\sigma(i)| = |\mathcal{L}^\sigma(i)| + |\mathcal{E}^\sigma(i)|. \quad (3.3)$$

We now focus on the cardinality of sets  $\mathcal{A}_a^{ans}(\ell)$ , for fixed  $i = 0, 1, 2, 3$ ,  $a \in \mathcal{L}^\sigma(i)$  and  $ans \in \{yes, no\}$ . We distinguish four cases according to the position of the phantom elements  $\varepsilon_1, \varepsilon_2 \in a^+$  (i.e., flanking  $a$ ).

*Case 1.*  $\varepsilon_1$  is on level  $i$  (like  $a$ ) and  $\varepsilon_2$  is on level  $i + 1$ . Therefore, their containing arcs in  $\mathcal{L}_{ans}^\sigma$  will be different, hence for  $\ell = i, i + 1$  we have  $|\mathcal{A}_a^\sigma(\ell)| = 1 \leq |\mathcal{A}_a^{ans}(\ell)|$ . The latter set might contain an additional arc on level  $\ell$  if and only if a proper subinterval of  $a$  including the element of  $a$  flanking  $\varepsilon_1$  falsifies  $ans$  and moves to level  $i + 1$ , creating a new arc also here. In formulas,  $|\mathcal{A}_a^{ans}(\ell)| = |\mathcal{A}_a^\sigma(\ell)| + b_{ans}^{(1)}(\ell)$  where  $b_{ans}^{(1)} = 1$  if  $\ell = i, i + 1$  and there exists a proper subarc of  $a$  flanking  $\varepsilon_1$  that falsifies  $ans$ ; and  $b_{ans}^{(1)}(\ell) = 0$  otherwise.

*Case 2.*  $\varepsilon_1, \varepsilon_2$  are both on level  $i + 1$  (while  $a \in \mathcal{L}^\sigma(i)$ ). Therefore, in  $\mathcal{A}_a^{ans}(i + 1)$  there will be two different arcs containing  $\varepsilon_1$  and  $\varepsilon_2$  and there will be one arc in  $\mathcal{A}_a^{ans}(i)$  containing the elements of  $a$  satisfying  $ans$  (the elements falsifying  $ans$  will be contained in one of the arcs containing a phantom element), unless *all* the elements in  $a$  falsify  $ans$ , in which case there will be no arc in  $\mathcal{A}_a^{ans}(i)$  and only one arc in

$\mathcal{A}_a^{ans}(i+1)$ . In formulas,  $|\mathcal{A}_a^{ans}(i)| = |\mathcal{A}_a^\sigma(i)| - b_{ans}^{(2)}(\ell)$  where  $b_{ans}^{(2)}(\ell) = 1$  if and only if  $\ell = i, i+1$  and all the elements of  $a$  falsify  $ans$ , and  $b_{ans}^{(2)}(\ell) = 0$  otherwise.

*Case 3.*  $i \leq 2$  and  $\varepsilon_1, \varepsilon_2$  are both on level  $i$ , like  $a$ , hence this is the arc of  $\mathcal{L}^\sigma$  containing them. This case is similar to *Case 1*. We have that  $\mathcal{A}_a^{ans}(\ell)$  contains an additional arc with respect to  $\mathcal{A}_a^\sigma(\ell)$  if and only if some elements of  $a$  falsify  $ans$  (the remaining elements are contained in the arcs containing the phantom elements). In formulas,  $|\mathcal{A}_a^{ans}(i)| = |\mathcal{A}_a^\sigma(i)| + b_{ans}^{(3)}(\ell)$  where  $b_{ans}^{(3)}(\ell) = 1$  if and only if  $\ell = i, i+1$  and some of the elements in  $a$  falsify  $ans$ , and  $b_{ans}^{(3)}(\ell) = 0$  otherwise.

*Case 4.*  $i = 3$  and  $\varepsilon_1, \varepsilon_2$  are both on level  $i$ , like  $a$ , hence this is the arc of  $\mathcal{L}^\sigma$  containing them. In this case, elements of  $a$  falsifying  $ans$  will simply disappear from the state  $\sigma^{ans}$  and  $\mathcal{A}_a^{ans}(\ell)$  will contain one arc for  $\ell = 3$  and 0 for  $\ell < 3$ . In case  $Q$  contains an interval that coincides with  $a$  then for  $ans = no$  we have a very special situation, i.e., there is no arc on level 3 and the two arcs on level 2 flanking  $a$  in  $\mathcal{L}^\sigma$  get merged into one single arc. We account for this exceptional case (when an interval of  $Q$  coincides with  $a$ ) by setting  $\mathcal{A}_a^{no}(2) = 1$  (even if according to the definition no arc on level 2 contains elements of  $a^+$ ). In formulas, we have  $|\mathcal{A}_a^{ans}(\ell)| = |\mathcal{A}_a^\sigma(\ell)| - b_{ans}^{(4)}(\ell)$  where  $b_{ans}^{(4)}(\ell) = 1$  if and only if there is an interval of  $Q$  which coincides with  $a$ ,  $\ell = 2, 3$ , and  $ans = no$ ; and  $b_{ans}^{(4)}(\ell) = 0$  otherwise.

We now observe that for  $i = 0, 1, 2$ , we have

$$\sum_{a \in \mathcal{L}^\sigma} |\mathcal{A}_a^{ans}(i)| = |\mathcal{L}_{ans}^\sigma(i)| + |\mathcal{E}^\sigma(i)|. \quad (3.4)$$

For each  $a \in \mathcal{L}^\sigma$ ,  $i = 0, 1, 2, 3$  and  $ans \in \{yes, no\}$ , let us now define  $\delta(a, Q, ans, \ell) = |\mathcal{A}_a^\sigma(i)| - |\mathcal{A}_a^{ans}(i)|$ , and  $\Delta_\ell^{ans}(\sigma, Q) = \sum_{a \in \mathcal{L}^\sigma} \delta(a, Q, ans, \ell)$ .

Then, for all  $i = 0, 1, 2$  we have that  $|\mathcal{L}_{ans}^\sigma(i)| = |\mathcal{L}^\sigma(i)| + \Delta_i^{ans}(\sigma, Q)$ .

Since  $\sigma$  is well shaped, by definition,  $\sigma^{yes}$  and  $\sigma^{no}$  are both well shaped if for each  $i = 0, 1, 2$  we have  $|\mathcal{L}^\sigma(i)| = |\mathcal{L}_{yes}^\sigma(i)| = |\mathcal{L}_{no}^\sigma(i)|$ . Thus, using (3.3) and (3.4) it follows that  $\sigma^{yes}$  and  $\sigma^{no}$  are well shaped if

$$\text{for all } i = 0, 1, 2, \quad \Delta_i^{yes}(\sigma, Q) = 0 \text{ and } \Delta_i^{no}(\sigma, Q) = 0, \quad (3.5)$$

as this will imply  $|\mathcal{L}_{yes}^\sigma(\ell)| = |\mathcal{L}_{no}^\sigma(\ell)| = |\mathcal{L}^\sigma(\ell)| = 2\ell - 1$  for  $\ell = 0, 1, 2$ , and consequently, also  $|\mathcal{L}_{yes}^\sigma(3)| = |\mathcal{L}_{no}^\sigma(3)| = |\mathcal{L}^\sigma(3)| = 3$ .

The following proposition, which is easily verified on the bases of the 4 cases analysed above, summarizes the values of  $\delta()$  which are significant for our analysis. For an example of all cases—except d) and g)—considered in the proposition, refer to Figure 3.3.

**Proposition 3.3.** *Let  $\sigma$  be a state and  $a$  be a non empty arc of  $\mathcal{L}^\sigma(i)$ . Let  $Q$  be a question that splits at most one arc per level and such that each arc is intersected by at most one interval of  $Q$ . Then we have*

- a) *if  $Q$  is saddle-splitting at level  $i$ , then for  $\ell = i, i+1$  we have  $\delta(a, Q, yes, \ell) = \delta(a, Q, no, \ell) = 1$ , i.e., the number of arcs is increased by 1 on level  $i$  and  $i+1$  both in the case of a yes and a no answer, since part of the saddle is transferred to the next level and therefore in the list of arcs there will be an additional (empty) arc between the part going to level  $i+1$  and the flanking arc at level  $i-1$ .*

- b) if  $Q$  is mode-splitting at level  $i$ , then for  $\ell = i, i + 1$  we have  $\delta(a, Q, \text{yes}, \ell) = \delta(a, Q, \text{no}, \ell) = 0$ , i.e., the number of arcs remains unchanged on level  $i$  and  $i + 1$  both in the case of a yes and a no answer, since the part of the mode that is transferred to level  $i + 1$  get merged with the flanking arc on level  $i + 1$ — recall that the value of  $\delta$  is defined with respect to the local changes within  $a^+$ .
- c) if  $Q$  is upward step-splitting at level  $i$ , then for  $\ell = i, i + 1$  we have  $\delta(a, Q, \text{yes}, \ell) = 0$  and  $\delta(a, Q, \text{no}, \ell) = 1$ , i.e., the number of arcs is increased by 1 on level  $i$  and  $i + 1$  only in the case of a no answer, since part of the step is transferred to the next level and therefore in the list of arcs there will be an additional (empty) arc between the part going to level  $i + 1$  and the flanking arc at level  $i - 1$ .
- d) if  $Q$  is downward step-splitting at level  $i$ , then for  $\ell = i, i + 1$  we have  $\delta(a, Q, \text{yes}, \ell) = 1$  and  $\delta(a, Q, \text{no}, \ell) = 0$ , i.e., the number of arcs is increased by 1 on level  $i$  and  $i + 1$  only in the case of a yes answer, since part of the step is transferred to the next level and therefore in the list of arcs there will be an additional (empty) arc between the part going to level  $i + 1$  and the flanking arc at level  $i - 1$ .
- e) if  $Q$  is saddle-covering at level  $i$  on a saddle  $a$ , then for  $\ell = i, i + 1$  we have  $\delta(a, Q, \text{yes}, \ell) = 0$  and  $\delta(a, Q, \text{no}, \ell) = 1$ , i.e., the number of arcs is increased by 1 on level  $i$  and  $i + 1$  only in the case of a no answer, since the saddle is transferred to the next level and therefore in the list of arcs there will be an additional (empty) arc between the saddle going to level  $i + 1$  and one of the flanking arc at level  $i - 1$ . — note that the other empty needed arc is the saddle arc becoming empty at level  $i$ .
- f) if  $Q$  is mode-covering at level  $i$  on a mode  $a$ , then for  $\ell = i, i + 1$  we have  $\delta(a, Q, \text{yes}, \ell) = 0$  and  $\delta(a, Q, \text{no}, \ell) = -1$ , i.e., the number of arcs is decreased by 1 on level  $i$  and  $i + 1$  only in the case of a no answer, since the mode is transferred to level  $i + 1$  and gets merged with both the flanking arcs at level  $i + 1$  into one single arc at level  $i + 1$ .
- g) if  $Q$  is step-covering at level  $i$  on the step  $a$ , then for  $\ell = i, i + 1$  we have  $\delta(a, Q, \text{yes}, \ell) = \delta(a, Q, \text{no}, \ell) = 0$ , i.e., the number of arcs remains unchanged on level  $i$  and  $i + 1$  both in the case of a yes and a no answer, since the arc (which was a step at level  $i$ ) is transferred to level  $i + 1$  and gets merged with the flanking arc on level  $i + 1$  and at level  $i$  we have a new empty arc (to satisfy the unit increase in the level of adjacent arcs).

For the complementary question  $\overline{Q}$ , the same rules apply with the role of yes and no swapped.

Using Proposition 3.3 and the condition in (3.5), we have the following sufficient conditions for building a question that preserves well-shapeness.

**Lemma 3.2.** *Given a well-shaped state  $\sigma$  and a question  $Q$ , if the following set of conditions is satisfied then both the resulting states  $\sigma_{\text{yes}}$  and  $\sigma_{\text{no}}$  are well-shaped. For each  $i = 0, 1, 2$  at most one arc is split on level  $i$ . Moreover, exactly one of the following holds*

- (i) at level  $i$  the question  $Q$  is mode-splitting;
- (ii) at level  $i$  the question  $Q$  is upward step-splitting and mode-covering.
- (iii) at level  $i$  the question  $Q$  is downward step-splitting and a mode is completely uncovered—equivalently the complementary question  $\overline{Q}$  is mode-covering at level  $i$ .

(iv) at level  $i$  the question  $Q$  is saddle-splitting and mode-covering and there is also a mode completely uncovered—equivalently the complementary question  $\overline{Q}$  is mode-covering at level  $i$ .

In addition, if besides the condition in (i)-(iv) the question  $Q$  is also saddle-covering (respectively a saddle is uncovered, i.e. covered in  $\overline{Q}$ ) then  $Q$  also covers (respectively leaves uncovered) a mode different from the one possibly used to satisfy (i)-(iv).

*Proof.* Fix  $i = 0, 1, 2$ . Let  $a \in \mathcal{L}^\sigma(i)$  be the split arc at level  $i$ . Then

- i) if  $Q$  is *mode-splitting* at level  $i$ , then for  $\ell = i, i+1$  we have that  $\delta(a, Q, \text{yes}, \ell) = \delta(a, Q, \text{no}, \ell) = 0$ .
- ii) if  $Q$  is *upward step-splitting* at level  $i$ , then for  $\ell = i, i+1$  we have  $\delta(a, Q, \text{yes}, \ell) = 0$  and  $\delta(a, Q, \text{no}, \ell) = 1$ . Moreover, let  $b \in \mathcal{L}^\sigma(i)$  be the *mode entirely covered* at level  $i$ , then for  $\ell = i, i+1$  we have  $\delta(b, Q, \text{no}, \ell) = -1$ .  
Summing up level by level all the  $\delta()$  elements, for  $\ell = i, i+1$  we have that  $\delta(a, Q, \text{no}, \ell) + \delta(b, Q, \text{no}, \ell) = 0$ .
- iii) if  $Q$  is *downward step-splitting* at level  $i$ , then for  $\ell = i, i+1$  we have that  $\delta(a, Q, \text{yes}, \ell) = 1$  and  $\delta(a, Q, \text{no}, \ell) = 0$ . Moreover, let  $b \in \mathcal{L}^\sigma(i)$  be the *completely uncovered mode* at level  $i$ , then for  $\ell = i, i+1$  we have that  $\delta(b, Q, \text{yes}, \ell) = -1$ .  
Summing up level by level all the  $\delta()$  elements, for  $\ell = i, i+1$  we have that  $\delta(a, Q, \text{yes}, \ell) + \delta(b, Q, \text{yes}, \ell) = 0$ .
- iv) if  $Q$  is *saddle step-splitting* at level  $i$ , then for  $\ell = i, i+1$  we have that  $\delta(a, Q, \text{yes}, \ell) = \delta(a, Q, \text{no}, \ell) = 1$ . Moreover, let  $b \in \mathcal{L}^\sigma(i)$  be the *entirely covered mode* at level  $i$ , then for  $\ell = i, i+1$  we have that  $\delta(b, Q, \text{no}, \ell) = -1$ . Also let  $c \in \mathcal{L}^\sigma(i)$  be the *completely uncovered mode* at level  $i$ , then for  $\ell = i, i+1$  we have that  $\delta(c, Q, \text{yes}, \ell) = -1$ .  
Summing up level by level all the  $\delta()$  elements, for  $\ell = i, i+1$  the result is  $\delta(a, Q, \text{yes}, \ell) + \delta(c, Q, \text{yes}, \ell) = 0$  and  $\delta(a, Q, \text{no}, \ell) + \delta(b, Q, \text{no}, \ell) = 0$ .

To verify the last part of the statement, let us now assume that besides the condition in (i)-(iv) the question  $Q$  is also saddle-covering (respectively a saddle is uncovered, i.e. covered in  $\overline{Q}$ ). Let  $b \in \mathcal{L}^\sigma(i)$  be a *saddle* at level  $i$  covered by some interval  $I_b$  of  $Q$  (respectively  $\overline{Q}$ ). Then for  $\ell = i, i+1$  we have that  $\delta(b, Q, \text{yes}, \ell) = 0$  and  $\delta(b, Q, \text{no}, \ell) = 1$  (respectively  $\delta(b, Q, \text{yes}, \ell) = 1$  and  $\delta(b, Q, \text{no}, \ell) = 0$ ). Let  $c \in \mathcal{L}^\sigma(i)$  be a *mode* at level  $i$ —recall that  $c$  is different from the one possibly used to satisfy the previous cases—covered by some interval  $I_c$  of  $Q$  (respectively  $\overline{Q}$ ). Then for  $\ell = i, i+1$  we have that  $\delta(c, Q, \text{no}, \ell) = -1$  (respectively  $\delta(c, Q, \text{yes}, \ell) = -1$ ). Summing up all the  $\delta()$  elements we have that  $\delta(b, Q, \text{no}, \ell) + \delta(c, Q, \text{no}, \ell) = 0$  (respectively  $\delta(b, Q, \text{yes}, \ell) + \delta(c, Q, \text{yes}, \ell) = 0$ ) for  $\ell = i, i+1$ .

It remains to consider the contribution given by the remaining arcs. These are only arcs covered completely by an interval either in  $Q$  or in  $\overline{Q}$ . Here we only deal with such arcs not already taken care of above. Note that these arcs are all step arcs. In fact, all mode arcs covered or uncovered by  $Q$  are already dealt with in the analysis of points (i)-(iv) above, or in the special case considering saddle arcs. Let  $a$  be such an arc and let  $i = 0, 1, 2$  such that  $a \in \mathcal{L}^\sigma(i)$ . Let  $I$  be the interval either in  $Q$  or in  $\overline{Q}$  covering  $a$ . By Proposition 3.3 we have that if  $a$  is a step arc  $\delta(a, Q, \text{yes}, \ell) = \delta(a, Q, \text{no}, \ell) = 0$  for every  $\ell = 0, 1, 2$ . Therefore, for each  $i = 0, 1, 2$ ,  $\Delta_i^{\text{yes}} = \Delta_i^{\text{no}} = 0$ , hence by (3.5) the resulting states  $\sigma_{\text{yes}}$  and  $\sigma_{\text{no}}$  are well shaped.  $\square$

Now we are ready to prove our main technical result.

**Theorem 3.1.** *Let  $\sigma$  be a well-shaped state of type  $\tau(\sigma) = (a_0, b_0, c_0, d_0)$ . For all integers  $0 \leq a \leq a_0$ ,  $0 \leq b \leq \lceil \frac{1}{2}b_0 \rceil$ ,  $0 \leq c \leq \lceil \frac{1}{2}c_0 \rceil$ ,  $0 \leq d \leq \lceil \frac{2}{3}d_0 \rceil$ , there exists a 4-interval question  $Q$  of type  $|Q| = [a, b, c, d]$  that we can ask in state  $\sigma$  and such that both the resulting “yes” and “no” states are well-shaped.*

*Proof.* We first show how to select the intervals of the question  $Q$  in order to satisfy the desired type. We proceed level by level. For each  $i = 0, 1, 2, 3$ , we show how to select up to 4 intervals that cover the required number of elements in the first  $i$  levels. For each level  $i = 0, 1, 2, 3$  we record in a set  $\mathcal{E}(i)$  the extremes of the intervals selected so far that have a neighbour on the next level. We refer to the elements in  $\mathcal{E}(i)$  as the boundaries at level  $i$ . When processing the next level, we try to select arcs neighbouring the elements in  $\mathcal{E}(i)$  since this means we can cover elements at the new level without using additional intervals. Arguing with respect to such boundaries, we show that the (sub)intervals selected at all level can be merged into at most 4 intervals. Hence the resulting question  $Q$  is a 4-interval question. Finally, we will show that asking  $Q$  in  $\sigma$  both the resulting states are well-shaped states.

Recall the arc notation used in (3.1)-(3.2). In our construction, a special role will be played by the arcs  $S, H, A$ , which are the greatest mode respectively of level 0, 1 and 2, and the larger between their two neighbouring arcs at the level immediately below. Therefore, let us denote by  $A^{(1)}$  the larger arc between  $N$  and  $O$ ; we denote by  $A^{(2)}$  be the larger arc between  $B$  and  $C$ ; and finally, we denote by  $A^{(3)}$  the larger arc between  $R$  and  $P$ .

Moreover, we denote by  $s^+$  the boundary between  $S$  and  $A^{(1)}$  and with  $s^-$  the other boundary of  $S$ .

Analogously, we denote by  $h^+$  the boundary between  $H$  and  $A^{(2)}$  and with  $h^-$  the other boundary of  $H$ .

We denote by  $a^+$  the boundary between  $A$  and  $A^{(3)}$  and with  $a^-$  the other boundary of  $A$ .

**Level 0.** For any  $0 < a \leq a_0$  there exists  $s^* \in S$  such that denoting by  $S^*$  the sub-arc of  $S$  between  $s^*$  and  $s^+$  we have that  $|S^*| = a$  and the boundary of  $S^*$  includes  $s^+$ . Then we have  $\mathcal{E}(0) \supseteq \{s^+\}$ .

Therefore, with one interval  $I^{(0)} = S^*$  we can accommodate the  $a$  elements on level 0 and guarantee that this interval has an extreme at  $s^+$ .

Moreover, the interval  $I^{(0)}$  on arc  $S$  is a mode-splitting interval.

**Level 1.** By the assumption  $b \leq \lceil \frac{1}{2}b_0 \rceil$ , and the definition of  $A^{(1)}$  it follows that  $b \leq |A^{(1)}| + |H|$ . We now argue by cases

1.  $b \leq |H|$ . Then there exists  $h^*$  in  $H$  such that the sub-arc  $H^* \subseteq H$  between  $h^*$  and  $h^+$  satisfies  $|H^*| = b$  and we can cover it with one interval  $I^{(1)}$  with a boundary at  $h^+$ .
2.  $|H| < b \leq |H| + |A^{(1)}|$ . Then, there exists  $x_1^* \in A^{(1)}$  such that letting  $X^{(1)}$  be the sub-arc of  $A^{(1)}$  between  $x_1^*$  and  $s^+$ , we have  $|X^{(1)}| + |H| = b$  and we can cover these  $b$  elements extending the previously defined  $I^{(0)}$  so that it covers  $X^{(1)}$  too and having  $I^{(1)} = H$ . In this case we have that the boundaries of  $I^{(1)}$  are both  $h^+$  and  $h^-$ .

Summarising, we can cover the  $a$  elements on level 0 and the  $b$  elements on level 1 with at most two intervals and guarantee that the boundaries of these intervals include  $h^+$ .

Moreover either the interval  $I^{(1)}$  on arc  $H$  is a mode-splitting interval or the interval  $I^{(1)}$  covers entirely the mode  $H$  and the interval  $I^{(0)}$  on arc  $A^{(1)}$  is a step-splitting interval.

Then, the choice of the intervals so far satisfies conditions in Lemma 3.2.

**Level 2.** Again we argue by cases

1.  $c \leq |A|$ . Then, there exists  $a^*$  in  $A$  such that letting  $A^*$  be the sub-arc of  $A$  between  $a^*$  and  $a^+$  we have  $|A^*| = c$  and we can cover it with one interval  $I^{(2)} = A^*$  with one boundary in  $a^+$ .
2.  $|A| < c \leq |A| + |A^{(2)}|$ . Then, there exists  $x^* \in A^{(2)}$  such that letting  $X^{(2)}$  be the sub-arc of  $A^{(2)}$  between  $x^*$  and  $h^+$ , we have  $|X^{(2)}| + |A| = c$  and we can cover these  $c$  elements extending the previously defined  $I^{(1)}$  so that it covers  $X^{(2)}$  too and having  $I^{(2)} = A$ . In this case we have that the boundaries of  $I^{(2)}$  are both  $a^+$  and  $a^-$ .

3.  $c > |A| + |A^{(2)}|$ . Let  $E$  denote the largest arc on level 2 not in  $\{A^{(2)}, A\}$ . Then, by the definition of  $A^{(2)}$  we have that  $|A| + |A^{(2)}| + |E| \geq |Y| + |Z|$  where  $Y$  and  $Z$  denote the arcs on level 2 not in  $\{A, A^{(2)}, E\}$ . This is true because, at least one of the arcs  $Y, Z$  is not larger than  $A^{(2)}$  and the other one is not larger than  $E$ .

Then, by the assumption  $c \leq \lceil \frac{c_0}{2} \rceil$  it follows that  $|A| + |A^{(2)}| + |E| \geq c$ . Let  $z$  be the boundary of  $A^{(2)}$  on the opposite side with respect to  $H$ . Let  $e^+$  be the boundary between  $E$  and a neighbouring arc at level 3 or at level 1 according to whether  $z$  is flanking an arc at level 1 or an arc at level 3—with reference to Figures 3.1, 3.2, an easy direct inspection shows that such a choice is always possible.

Therefore, there exists  $e^* \in E$  such that letting  $E^*$  be the sub-arc of  $E$  between  $e^*$  and  $e^+$  we have  $|A| + |A^{(2)}| + |E^*| = c$  and we can cover the corresponding set of elements by: (i) extending  $I^{(1)}$  from  $h^+$  and have it include the whole  $A^{(2)}$ ; (ii) defining  $I^{(2)} = A$ ; defining a fourth interval  $I^{(3)} = E^*$ . Therefore, we have that the boundaries of  $I^{(2)}$  are both  $a^+$  and  $a^-$  and, in the case of a  $\sigma$  of type in Fig. 3.2, the boundary of  $I^{(3)}$  includes  $e^+$ , and the boundary of  $I^{(1)}$  is the boundary  $z$  of the arc  $A^{(2)}$  where it joins its adjacent arc at level 3.

Summarising, we can cover the  $a$  elements on level 0 and the  $b$  elements on level 1 and the  $c$  elements of level 2 with at most four intervals. More precisely, if, proceeding as above we only use three intervals,  $I^{(0)}, I^{(1)}, I^{(2)}$ , (and set  $I^{(3)} = \emptyset$ ), we also guarantee that the boundaries of these intervals include  $a^+$ . On the other hand, if we use four intervals, (in particular,  $I^{(3)} \neq \emptyset$ ) we have that the boundaries of these intervals include  $a^+, a^-$  and exactly one between  $e^+, z$ . Therefore  $\{a^+, a^-\} \subset \mathcal{E}(2) \subset \{a^+, a^-, z, e^+\}$ . Notice that, since there are only three arcs at level 3; in the case where  $I^{(3)} \neq \emptyset$  either there is an arc on level 3 with both ends neighbouring the boundaries in  $\mathcal{E}(2)$ , or each arc on level 3 has one end neighbouring a boundary in  $\mathcal{E}(2)$ .

Moreover exactly one of the following cases holds (i) the interval  $I^{(2)}$  on arc  $A$  is a mode-splitting interval; (ii) the interval  $I^{(2)}$  covers entirely the mode  $H$  and the interval  $I^{(1)}$  on arc  $A^{(2)}$  is an upward step-splitting interval; (iii) the interval  $I^{(2)}$  covers the mode  $H$ , no interval intersects mode  $M$  and the interval  $I^{(1)}$  on arc  $A^{(2)}$  is saddle-splitting; (iv) the interval  $I^{(2)}$  covers the mode  $H$  and the interval  $I^{(1)}$  covers the arc  $A^{(2)}$  and the interval  $I^{(3)}$  on arc  $E$  is downward step-splitting; (v) the interval  $I^{(2)}$  covers the mode  $H$ , the interval  $I^{(1)}$  covers the arc  $A^{(2)}$ , hence it is saddle-covering, and the interval  $I^{(3)}$  on arc  $E$  is upward step-splitting. .



In all the above five cases, the (partial) question built so far, with the intervals defined for levels 0, 1, 2, satisfy the conditions of Lemma 3.2.

**Level 3.** Let us denote by  $W, U$  the two arcs at level 3 which are different from  $A^{(3)}$ , with  $|W| \geq |U|$ . Then, by definition we also have  $|A^{(3)}| \geq |U|$ , hence  $|A^{(3)}| + |W| \geq \frac{2}{3}d_0 \geq d$ . We now argue by cases:

1.  $d < |A^{(3)}|$ . Then, there exists  $x_3^*$  in  $A^{(3)}$  such that the sub-arc  $X^{(3)}$  between  $a^+$  and  $x_3^*$  has cardinality  $|X^{(3)}| = d$  and can be covered by extending  $I^{(2)}$  (which have a boundary at  $a^+$ ).
2.  $|A^{(3)}| < d \leq |A^{(3)}| + |W|$ .

We have two sub-cases:

- $I^{(3)} = \emptyset$ . I.e., for accommodating the question's type on Levels 0,1,2, we have only used three intervals. By assumption, there exists a sub-arc  $W^*$  of  $W$  such that  $|W^*| + |A^{(3)}| = d$ . Then, defining  $I^{(3)} = W^*$ , and extending  $I^{(2)}$  (as in the previous case) so that it includes the whole of  $A^{(3)}$  guaranteeing that the four intervals  $I^{(0)}, \dots, I^{(3)}$  define a question of the desired type.
- $I^{(3)} \neq \emptyset$ . Then, by the observations above, as a result of the construction on level 2, either there is an arc on level 3 with both ends at a boundary in  $\mathcal{E}(2)$  or each arc on level 3 has a boundary in  $\mathcal{E}(2)$ . In the latter case, we can clearly extend the intervals  $I^{(2)}$  and  $I^{(3)}$  in order to cover  $d$  elements on level 3. In the former case, let  $Z$  denote the arc with both ends at boundaries in  $\mathcal{E}(2)$ . If  $|Z| \geq d$ , we can simply extend  $I^{(2)}$  and  $I^{(3)}$  towards the internal part of  $Z$  until they include  $d$  elements of  $Z$ . If  $|Z| < d$  then we can extend  $I^{(2)}$  so that it includes  $Z$  and  $I^{(3)}$ .

Since the way intervals are extended on level 3 do not affect the arc covering and splitting on the previous level, we have that in all cases the resulting 4-interval question satisfies the conditions of Lemma 3.2, which guarantees that both resulting states are well-shaped. The proof is complete.  $\square$

Refer to Figures 3.4 and 3.5 for a pictorial representation of the 4-interval question construction in the proof of Theorem 3.1.

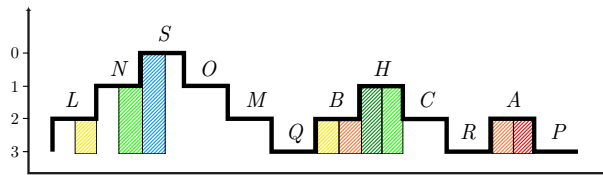


Fig. 3.4: A well-shaped state like in (3.1) and the cuts of a 4 interval question.

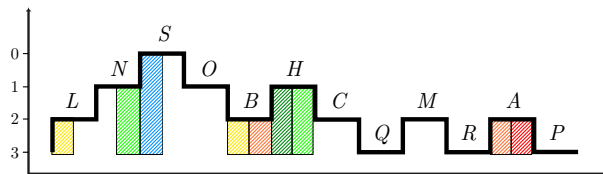


Fig. 3.5: A well-shaped state like in (3.2) and the cuts of a 4 interval question.



We assume the following relative order on the arcs' sizes. On level 1 we assume  $N \geq O$  and on level 2 we assume  $B \geq C$ ,  $L \geq M$  and  $A \geq M$ . The questions are depicted for both the feasible well-shaped states for a 3 lies game. Then on level 0 the arc  $S$  is split (the light blue question), on level 1 either  $H$  is split (the dark green question) or  $N$  is split (the dark green and the light green question) and  $H$  is covered entirely. On level 2 one of the following holds, either  $A$  is split (the red question) or  $B$  is split (the orange and red question) and the mode  $A$  is covered entirely, or  $L$  is split (the yellow, orange and red question) and the mode  $A$  is covered entirely. In order to guarantee the well-shapeness preservation, note that in the questions depicted in figure 3.5 on level 2 the mode  $M$  is entirely uncovered, moreover the interval splitting arc  $L$  is upward step-splitting in Figure 3.4 and downward step-splitting in Figure 3.5.

### 3.4 The non asymptotic strategy

In this section we show how to employ the machinery of Lemma 3.2 to obtain an exact (non-asymptotic) characterization of the instances of the Ulam-Rényi game with 3 lies on a search space of cardinality  $2^m$  that admit a perfect strategy only using 4-interval questions. For this we will exploit the perfect strategies of [101] and show that they can be implemented using only 4-interval questions. We are going to prove the following result.

**Theorem 3.3.** *For all  $m \in \mathbb{N} \setminus \{2, 3, 5\}$  in the game played over the search space  $\{0, \dots, 2^m - 1\}$  with 3 lies, there exists a perfect 4-interval strategy.*

The following three lemmas provides alternatives to Theorem 3.1 for guaranteeing the existence of 4-interval questions that preserves the well-shapeness of the resulting states.

**Lemma 3.3.** *Let  $\sigma$  be a well-shaped state of type  $\tau(\sigma) = (0, b_0, c_0, d_0)$ . For all integers  $0 \leq b \leq \lceil \frac{1}{2}b_0 \rceil$ ,  $0 \leq c \leq \lceil \frac{1}{2}c_0 \rceil$ ,  $0 \leq d \leq d_0$ , there exists a 4-interval question  $Q$  of type  $|Q| = [0, b, c, d]$  that can be asked in state  $\sigma$  and such that both the resulting "yes" and "no" states are well-shaped.*

*Proof.* The proof structure is analogous to the proof of Theorem 3.1

**Level 1.** By the assumption  $b \leq \lceil \frac{1}{2}b_0 \rceil$  and definition of  $H$  as the greatest mode of level 1 it follows that  $b \leq |H|$ . Then there exists  $h^*$  in  $H$  such that the sub-arc  $H^* \subseteq H$  between  $h^*$  and  $h^+$  satisfies  $|H^*| = b$  and we can cover it with one interval  $I^{(0)}$  with a boundary at  $h^+$ . Then we have  $\mathcal{E}(1) \supseteq \{h^+\}$ .

Summarising, we can cover the  $b$  elements on level 1 with at most one interval and guarantees that the boundaries of these intervals include  $h^+$ .

Moreover the interval  $I^{(0)}$  on arc  $H$  is a mode-splitting interval.

**Level 2.** As showed in Theorem 3.1, we can cover the  $b$  elements on level 1 and the  $c$  elements on level 2, with at most three intervals. Moreover, if, proceeding as in Theorem 3.1 we only use two intervals,  $I^{(0)}, I^{(1)}$ , (and set  $I^{(2)} = \emptyset$ ) we also guarantee that the boundaries of these intervals include  $a^+$ . Otherwise, if we use three intervals, we have that the boundaries of those intervals include  $a^+, a^-$  and exactly one between  $e^+, z$ .

**Level 3.** Let us denote by  $W, U$  the two arcs at level 3 which are different from  $A^{(3)}$ . We now argue by cases:

1.  $I^{(2)} = \emptyset$ . We show how to cover up to  $d_0$  elements on level 3. Extending  $I^{(1)}$  we can cover as much as we need from the arc  $A^{(3)}$ , after that we can use the two remaining intervals  $I^{(2)}$  and  $I^{(3)}$  to cover, respectively  $W$  and  $U$ . This guarantees that with at most four intervals  $I^{(0)}, \dots, I^{(3)}$  we have a question of the desired type.
2.  $I^{(2)} \neq \emptyset$ . Then, by the observations above, both the boundaries include  $a^+$  and  $a^-$ . As before, Extending  $I^{(1)}$  we can cover as much as we need from the arc  $A^{(3)}$  and the other neighbouring arc of  $A$ , say  $W$ . Finally, we can use the remaining interval  $I^{(3)}$  to cover the last uncovered arc  $U$ .

In all cases the resulting 4-interval question satisfies the Lemma 3.2 conditions, which guarantees that both resulting states are well-shaped.  $\square$

**Lemma 3.4.** *Let  $\sigma$  be a well-shaped state of type  $\tau(\sigma) = (1, b_1, c_1, d_1)$ . For all integers  $\lfloor \frac{1}{4}c_1 \rfloor \leq c \leq \lceil \frac{1}{2}c_1 \rceil$ ,  $0 \leq d \leq d_1$ , there exists a 4-interval question  $Q$  of type  $|Q| = [1, 1, c, d]$  that we can ask in state  $\sigma$  and such that both the resulting “yes” and “no” states are well-shaped.*

*Proof.* Let  $U$  be the smallest arc on level 2, different from  $A$  and  $B$ . Then, assuming that  $|A| \geq \lfloor \frac{1}{4}c_1 \rfloor$ , we have that  $|U| \leq \lfloor \frac{1}{4}c_1 \rfloor$ . This is true because, by the minimality of  $U$  we have  $|U| \leq \lfloor \frac{(1-\frac{1}{4})c_1}{3} \rfloor \leq \lfloor \frac{1}{4}c_1 \rfloor$ .

Let  $V$  be the neighbouring arc of  $U$  at level 1. We denote with  $v^-$  the boundary between  $U$  and  $V$  and with  $u^+$  the other boundary of  $U$ . The existence and uniqueness of  $u^+$  is guaranteed by the definition of  $U$  that excludes that it is a saddle.

This proof has a different scheme with respect to the proof structure of Theorem 3.1. In fact, the interval used to accommodate the element on level 1, is defined after the evaluation of the intervals at level 2.

**Level 0.** This is treated as in Theorem 3.1, then we have one element of level 0 that is accommodated by interval  $I^{(0)}$ . The interval on arc  $S$  is a mode-covering interval.

**Level 2.** We argue by cases

1.  $\lfloor \frac{1}{4}c_1 \rfloor \leq c \leq |A|$ . Then, there exists  $a^*$  in  $A$  such that letting  $A^*$  be the sub-arc of  $A$  between  $a^*$  and  $a^+$  we have  $|A^*| + |U| = c$ .  
We can cover it using the interval  $I^{(1)} = U$  with one boundary in  $u^+$ , and with one other interval  $I^{(2)} = A^*$  with one boundary in  $a^+$ .
2.  $|A| < c \leq \lceil \frac{1}{2}c_1 \rceil$ . Again we argue by cases on the size of the mode  $H$  at level 1. In particular, we have that either the mode  $H$  contains at least one element, or the mode  $H$  is empty. In the former case, we can proceed as in the proof of Theorem 3.1 guaranteeing that the boundary  $h^+$  is included in the interval  $I^{(1)}$ . Moreover, if we use tree intervals, we guarantee that the interval  $I^{(2)}$  has the boundaries  $a^+$  and  $a^-$ , and if we use four intervals, we guarantee that the intervals  $I^{(1)}$  and  $I^{(3)}$  have exactly one boundary between  $e^+$  and  $z$ . In the latter case, we can argue by cases  
*Subcase 2.1*  $\lfloor \frac{1}{4}c_1 \rfloor \leq c \leq |A| + |U|$ . Then, there exists  $x^* \in U$  such that letting  $X^{(2)}$  be the sub-arc of  $U$  between  $x^*$  and  $u^-$ , we have  $|X^{(2)}| + |A| = c$  and we can cover these  $c$  elements using the interval  $I^{(1)}$  to covers  $X^{(2)}$  and having  $I^{(2)} = A$ . In this case we have that the boundaries of  $I^{(2)}$  are both  $a^+$  and  $a^-$ .  
*Subcase 2.2*  $c > |A| + |U|$ . Let  $E$  denote the largest arc on level 2 not in  $\{U, A\}$ . Note that, since the mode at level 1 is empty we can glue together the arcs  $B$  and  $C$  flanking the mode. Then, we have that  $|A| + |U| + |E| \geq |Y|$  where  $Y$  is the

arc on level 2 not in  $\{A, U, E\}$ . This is true because,  $E \geq Y$ . Therefore, there exists  $e^* \in E$  such that letting  $E^*$  be the sub-arc of  $E$  between  $e^*$  and  $e^+$  we have  $|A| + |U| + |E^*| = c$  and we can cover the corresponding set of elements by: (i) using  $I^{(1)}$  to cover the whole  $U$ ; (ii) defining  $I^{(2)} = A$ ; defining a fourth interval  $I^{(3)} = E^*$ . Therefore, we have that the boundaries of  $I^{(2)}$  are both  $a^+$  and  $a^-$  and, the boundary of  $I^{(3)}$  includes  $e^+$ , and the boundary of  $I^{(1)}$  is the boundary  $u^+$  of the arc  $U$  where this joins its adjacent arc at level 3.

Summarizing, we can cover one element on level 0 and the  $c$  elements of level 2 with at most four intervals. More precisely, the main difference with the analysis on Theorem 3.1, is that if we only use three intervals,  $I^{(0)}, I^{(1)}, I^{(2)}$ , (and set  $I^{(3)} = \emptyset$ ), we also guarantee that the boundaries of these intervals include  $a^+$  and one between  $a^-$  and  $u^+$ . Conversely, if we use four intervals, we have that the boundaries of these intervals include  $a^+, a^-$  and exactly one between  $e^+, z$ . Thus, either  $\{a^+, u^+\} \subset \mathcal{E}(2)$  or  $\{a^+, a^-\} \subset \mathcal{E}(2) \subset \{a^+, a^-, z, e^+\}$ . Indeed, if the mode  $H$  is empty, and we are using four intervals, we have that the boundaries of these intervals include  $a^+, a^-, e^+$  and  $u^+$ . Thus we have that  $\{a^+, a^-, e^+, u^+\} \subset \mathcal{E}(2)$ .

Moreover, the analysis of the splitting intervals is the same as the cases on Theorem 3.1, in addition we have that in the first case, the interval  $I^{(1)}$  covers the arc  $U$  and the interval  $I^{(2)}$  on arc  $A^*$  is a mode-splitting interval.

**Level 1.** From the previous arguments, we have that one boundary between  $u^-$  and  $h^+$  is included in the interval  $I^{(1)}$ . Then, if  $I^{(1)}$  include the boundary  $u^-$  then we can extend it from  $u^-$  up to  $v^*$  in  $V$  such that letting  $X^*$  be the sub arc of  $V$  between  $v^*$  and  $u^-$  with  $|X^*| = 1$ . Otherwise, if the boundary included in  $I^{(1)}$  is  $h^+$  then we can extend  $I^{(1)}$  from  $u^-$  up to  $h^*$  in  $H$  such that letting  $X^*$  be the sub arc of  $H$  between  $h^*$  and  $u^-$  with  $|X^*| = 1$ . Then, we can cover one element on level 0, one element on level 1 and the  $c$  elements of level 2 with at most four intervals. And the boundaries at level 3 remain unchanged. Moreover, the interval  $I^{(1)}$  covers the arc  $X^*$  and is mode-splitting if the arc  $X$  is the mode  $H$ , otherwise is downward-step splitting.

**Level 3.** Let us denote by  $W, Z$  the two arcs at level 3 which are different from  $A^{(3)}$ , with  $|W| \geq |Z|$ .

1.  $d \leq |A^{(3)}| + |W|$ . This case is handled like in Theorem 3.1.
2.  $|A^{(3)}| + |W| < d$ . There exists a sub-arc  $Z^*$  of  $Z$  such that  $|Z^*| + |W| + |A^{(3)}| = d$ . As a result of the construction used on the previous level, we have that  $\mathcal{E}(2)$  has at least two boundaries. One boundary is one end of the arc  $A^{(3)}$ . The other boundary in  $\mathcal{E}(2)$  is one end of the arc  $X \in \{A^{(3)}, W, Z\}$ . We have two sub-cases:  
*Subcase 2.1*  $I^{(3)} = \emptyset$ . Then we have that either  $\{a^+, a^-\} \subseteq \mathcal{E}(2)$  or  $\{a^+, u^-\} \subseteq \mathcal{E}(2)$ . Moreover, either there is an arc on level 3 with both ends at a boundary in  $\mathcal{E}(2)$  or two arcs on level 3 with a boundary in  $\mathcal{E}(2)$ . In the former case, we can extend the interval  $I^{(2)}$  to cover the arcs  $X$ , that is the arc  $A^{(3)}$ , including the interval  $I^{(1)}$ . Then, redefining the interval  $I^{(1)}$  as  $I^{(1)} = W$  and defining  $I^{(3)} = Z^*$ , we guarantee that the four intervals  $I^{(0)}, \dots, I^{(3)}$  define a question of the desired type. In the latter case, we can extend the interval  $I^{(2)}$  to cover the arcs  $A^{(3)}$  and  $X$ , that is different from  $A^{(3)}$ . Then, defining  $I^{(3)} = Z^*$ , it guarantees that the four intervals  $I^{(0)}, \dots, I^{(3)}$  define a question of the desired type.

*Subcase 2.2*  $I^{(3)} \neq \emptyset$ . Then, by the observations above, as a result of the construction on level 2, either there is an arc on level 3 with both ends at a boundary in  $\mathcal{E}(2)$  or each arc on level 3 has a boundary in  $\mathcal{E}(2)$ . In the latter case, we can

clearly extend the intervals  $I^{(2)}$  and  $I^{(3)}$  in order to cover  $d$  elements on level 3. In the former case, we can extend the interval  $I^{(2)}$  to cover the arcs  $X$  and  $A^{(3)}$ , including the interval with the boundary at one end of  $X$ , say  $I^{(3)}$ . Then, redefining the interval  $I^{(3)}$  as  $I^{(3)} = Z^*$ , it guarantees that the four intervals  $I^{(0)}, \dots, I^{(3)}$  define a question of the desired type.

Since in all the cases, the conditions of Lemma 3.2 are satisfied, the resulting states are well shaped.  $\square$

**Lemma 3.5.** *Let  $\sigma$  be a well-shaped state with type  $\tau(\sigma) = (1, 1, c_2, d_2)$ , for every integer  $0 \leq d \leq d_2$ , there exists a 4-interval question  $Q$  of type  $|Q| = [1, 0, 2, d]$  such that the resulting “yes” and “no” states are well-shaped.*

*Proof.* Let  $E$  be a non empty arc on level 2 and let  $e^+$  its boundary with  $U \in \{P, Q, R\}$ . We use one interval  $I^{(0)}$  to cover one element from the mode  $S$ . If  $|E| \geq 2$ , we take an interval  $I^{(1)}$  on the arc  $E$ , in particular there exists  $e^*$  in  $E$  such that the sub arc  $E^* \subseteq E$  with  $|E^*| = 2$  is covered by  $I^{(1)}$ . In order to cover  $d$  elements on level 3, we extend the interval  $I^{(1)}$  over  $U$  to cover up to  $|U|$  elements. Then, we use the remaining two intervals  $I^{(2)}, I^{(3)}$  to cover the remaining  $d - |U|$  elements over the arcs  $\{P, Q, R\} \setminus \{U\}$ . In this case, we have that the interval  $I^{(0)}$  is a mode-splitting interval. The interval  $I^{(1)}$  either is a mode-splitting interval or it is a downward step-splitting interval while a mode at level 2 is completely uncovered. The intervals  $I^{(2)}$  and  $I^{(3)}$  involves only arcs on level 3, thus they are not involved in the arc covering and spitting on the lower levels. The conditions of Lemma 3.2 are satisfied, then we have that both the resulting states are well shaped. Otherwise,  $|E| = 1$ , let  $F$  be a non empty arc, different from  $E$ , on level 2 and let  $f^+$  its boundary with  $V \in \{P, Q, R\} \setminus \{U\}$ . Then, we use the interval  $I^{(1)}$  to cover entirely the arc  $E$ , and there exists  $f^* \subseteq F$  such that the sub arc  $F^* \subseteq F$  is the arc spanning from  $f^*$  to  $f^+$  which has  $|F^*| = 1$ , covered by an additional interval  $I^{(2)}$ . In order to cover  $d$  elements on level 3, as before, we extend the interval  $I^{(1)}$  over  $U$  to cover up to  $|U|$  elements and we extend the interval  $I^{(2)}$  over  $V$  to cover up to  $|V|$  elements. Finally, if necessary, the remaining interval  $I^{(3)}$  is used to include the remaining  $d - |U| - |V|$  elements over the last arc  $\{P, Q, R\} \setminus \{U, V\}$ . In this case, we have that the interval  $I^{(0)}$  is a mode-splitting interval. The interval  $I^{(1)}$  either is a mode-covering interval or it is a step-covering interval while the interval  $I^{(2)}$  is either a mode-splitting interval or it is a downward step-splitting interval. If the interval  $I^{(2)}$  is a downward step-splitting interval and the interval  $I^{(1)}$  is a mode-covering interval, then there exists a completely uncovered mode. The interval  $I^{(3)}$  involves only arcs on level 3, thus they are not involved in the arc covering and spitting on the lower levels. The conditions of Lemma 3.2 are satisfied, then both the resulting states are well shaped.  $\square$

**Lemma 3.6.** *Let  $\sigma$  be a well-shaped state of type  $\tau(\sigma) = (1, 0, 3, d_3)$ . For every integer  $0 \leq d \leq d_3$  there there exists a 4-interval question  $Q$  of type  $|Q| = [1, 0, 0, d]$  such that the resulting “yes” and “no” states are well-shaped.*

*Proof.* We use one interval  $I^{(0)}$  to cover one element from the mode  $A$ . Then we use the remaining three intervals  $I^{(1)}, \dots, I^{(3)}$  to cover  $d$  elements over the remaining three arcs, respectively  $P, Q$  and  $R$ . We have that the interval  $I^{(0)}$  is a mode-covering interval, while the intervals  $I^{(1)}, \dots, I^{(3)}$  involves only arcs at level 3 that does not affect the splitting covering of the lower levels. The conditions of Lemma 2 are satisfied and the resulting states are well shaped.  $\square$

### 3.4.1 The proof of the main theorem

**Definition 3.7 (nice and 4-interval nice state).** *Following the a standard terminology in the area, we say that a state  $\sigma$  is nice if there exists a strategy  $S$  of size  $ch(\sigma)$  that is winning for  $\sigma$ . In addition we say that a state  $\sigma$  is 4-interval nice if: (i)  $\sigma$  is well shaped; (ii) there exists a strategy  $S$  of size  $ch(\sigma)$  that is winning for  $\sigma$  and only uses 4-interval questions*

A direct consequence of the above definition is that a state  $\sigma$  is 4-interval nice if it is well-shaped and, either it is a final state, or there is a 4-interval question for  $\sigma$  such that both the resulting *yes* and *no* states are 4-interval nice.

With this definition we can state the following, which gives a sufficient condition for the claim of Theorem 3.3.

**Lemma 3.7.** *Fix an integer  $m \in \mathcal{N}$ . If every well-shaped state of type  $(1, m, \binom{m}{2}, \binom{m}{3})$  is 4-interval nice then there exists a 4-interval question perfect strategy in the game with 3 lies over the space of size  $2^m$ .*

*Proof.* In the light of Definition 3.7 we will show that if  $(1, m, \binom{m}{2}, \binom{m}{3})$  is 4-interval nice then  $(2^m, 0, 0, 0)$  is 4-interval nice.

We have the following claim directly following from Theorem 3.1.

*Claim* For each  $j = 0, 1, \dots, m-1$  let  $\sigma^{(j)} = (2^{m-j}, j2^{m-j}, \binom{j}{2}2^{m-j}, \binom{j}{3}2^{m-j})$ . By Theorem 3.1 if  $\sigma$  is a well-shaped state of type  $\sigma^{(j)}$  then there is a 4-interval question  $Q$  of type  $|Q| = [\frac{|\sigma^{-1}(0)|}{2}, \frac{|\sigma^{-1}(1)|}{2}, \frac{|\sigma^{-1}(2)|}{2}, \frac{|\sigma^{-1}(3)|}{2}]$  such that both the resulting *yes* and *no* state are well shaped of type  $\sigma^{(j+1)}$ .

Therefore, starting from the well-shaped state  $(2^m, 0, 0, 0)$  by repeated application of the Claim, we have that there is a sequence of 4-interval questions such that for every possible sequence of answers the resulting state is well-shaped and of type  $(1, m, \binom{m}{2}, \binom{m}{3})$ . Since all the above questions are balanced, we also have that  $ch(1, m, \binom{m}{2}, \binom{m}{3}) = ch(2^m, 0, 0, 0) - m$ .

Therefore if it is true that every well-shaped state of type  $(1, m, \binom{m}{2}, \binom{m}{3})$  is 4-interval nice, it follows that  $(2^m, 0, 0, 0)$  is also 4-interval nice, as desired.  $\square$

The following lemma is actually a rephrasing (in the present terminology) of Proposition 3.2.

**Lemma 3.8.** *For  $n \in \mathbb{N}$  let  $T(n) = \{(1, 0, 0, n), (0, 1, 0, n), (0, 0, 1, n), (0, 0, 0, n)\}$ . Fix  $n \in \mathbb{N}$  and let  $\sigma$  be a state of type  $\tau \in T(n)$ . Then  $\sigma$  is 4-interval nice.*

*Proof.* First we observe that every state in  $\sigma$  is well-shaped since it has one arc on level  $j \leq 2$  and one arc on level 3. The existence of a perfect strategy only using 4-interval questions is a direct consequence of Proposition 3.2.  $\square$

**Definition 3.8 (0-typical state [70][101]).**

*Let  $\sigma$  be a state of type  $(t_0, t_1, t_2, t_3)$  with  $ch(t_0, t_1, t_2, t_3) = q$ . We say that  $\sigma$  is 0-typical if the following holds*

$$t_0 = 0; \quad t_2 \geq t_1 - 1; \quad t_3 \geq q.$$

**Lemma 3.9.** *Let  $\sigma$  be a 0-typical well shaped state of character  $\geq 12$ , then there exists a 4-interval question  $Q$  such that both the resulting states  $\sigma_{yes}$  and  $\sigma_{no}$  are 0-typical well-shaped and  $ch(\sigma_{yes}) < ch(\sigma)$  and  $ch(\sigma_{no}) < ch(\sigma)$ .*

*Proof.* Let  $(0, t_1, t_2, t_3)$  be the type of  $\sigma$ . From [70, Theorems 3.3, 3.4] and [101, Lemma 2] we have that if the type of  $Q$  is defined according to the following cases, then both the resulting states  $\sigma_{yes}$  and  $\sigma_{no}$  are 0-typical and in addition  $ch(\sigma_{yes}) < ch(\sigma)$  and  $ch(\sigma_{no}) < ch(\sigma)$ .

Case 1.  $ch(\sigma) = k \geq 3$  and  $t_2 \geq 3k - 3$

Case 1.1.  $t_1$  and  $t_2$  are even numbers. Then  $Q$  is chosen of type  $[0, \frac{t_1}{2}, \frac{t_2}{2}, \lfloor \frac{t_3}{2} \rfloor]$ .

Case 1.2.  $t_1$  is even and  $t_2$  is odd. Then  $Q$  is chosen of type  $[0, \frac{t_1}{2}, \lfloor \frac{t_2}{2} \rfloor, \lfloor \frac{t_3+k-1}{2} \rfloor]$ .

Case 1.3.  $t_1$  is odd and  $t_2$  is even. Then  $Q$  is chosen of type<sup>1</sup>  $[0, t_1 - \lfloor \frac{t_1}{2} \rfloor, t_2 - (\frac{t_2}{2} + B_1), t_3 - \lfloor \frac{t_3-t-1}{2} \rfloor]$ , where  $B_1 = \lfloor \frac{1}{2} \lfloor \frac{k^2-k+2}{2(k-1)} \rfloor \rfloor$  and  $t = (2B_1 + 1)(k-1) - \frac{k^2-k+2}{2}$ .

Case 1.4.  $t_1$  and  $t_2$  are odd. Then  $Q$  is chosen of type<sup>2</sup>  $[0, t_1 - \lfloor \frac{t_1}{2} \rfloor, t_2 - (\lfloor \frac{t_2}{2} \rfloor + B_1), t_3 - \lfloor \frac{t_3-t-1}{2} \rfloor]$ , where  $B_1 = \lfloor \frac{1}{2} \lfloor \frac{k^2-k+2}{2(k-1)} \rfloor \rfloor$  and  $t = 2B_1(k-1) - \frac{k^2-k+2}{2}$ .

Case 2.  $ch(\sigma) = k \geq 4$  and  $t_3 \geq k^2$ .

Case 2.1.  $t_1$  and  $t_2$  are even numbers. Then  $Q$  is chosen of type  $[0, \frac{t_1}{2}, \frac{t_2}{2}, \lfloor \frac{t_3}{2} \rfloor]$ .

Case 2.2.  $t_1$  is even and  $t_2$  is odd. Then  $Q$  is chosen of type  $[0, \frac{t_1}{2}, \lfloor \frac{t_2}{2} \rfloor, \lfloor \frac{t_3+k-1}{2} \rfloor]$ .

Case 2.3.  $t_1$  is odd and  $t_2$  is even. Then  $Q$  is chosen of type  $[0, \lfloor \frac{t_1}{2} \rfloor, \frac{t_2}{2}, \lfloor \frac{t_3+t}{2} \rfloor]$ , where  $t = (\frac{k^2-3k+2}{2})$ .

Case 2.4.  $t_1$  and  $t_2$  are odd. Then  $Q$  is chosen of type<sup>3</sup>  $[0, t_1 - \lfloor \frac{t_1}{2} \rfloor, t_2 - \lfloor \frac{t_2}{2} \rfloor, t_3 - \lfloor \frac{t_3+t}{2} \rfloor]$ , where  $t = (\frac{k^2-5k+4}{2})$ .

By Lemma 3.3, for each one of the type considered in the above 8 cases, there is a 4-interval question of that type such that the resulting states are well-shaped. This completes the proof of the claim.  $\square$

**The 0-typical states of character  $\leq 12$ .** Let  $\tilde{W}$  be the set of quadruples  $(0, t_1, t_2, t_3)$  such that for each  $\tau \in \tilde{W}$  the following two conditions are satisfied:

- every state  $\sigma$  of type  $\tau$  is nice
- there exist  $0 \leq \alpha_1 \leq \lceil t_1/2 \rceil$  and  $0 \leq \alpha_2 \leq \lceil t_2/2 \rceil$  and  $0 \leq \alpha_3 \leq t_3$  such that asking a question of type  $[0, \alpha_1, \alpha_2, \alpha_3]$  in a state  $\sigma$  of type  $\tau$  both the resulting states  $\sigma_{yes}$  and  $\sigma_{no}$  have character smaller than  $\sigma$ .

Note that if  $\sigma$  is a well shaped state whose character is in  $\tilde{W}$  then it is also 4-interval nice, by Lemma 3.3.

We can exhaustively compute all the states in  $\tilde{W}$  of character  $\leq 12$ . More precisely, we will compute all the pairs  $t_1, t_2$ , such that there exists  $t_3$  and  $(0, t_1, t_2, t_3) \in \tilde{W}$ ;  $ch(0, t_1, t_2, t_3) \leq 12$ .

Let

$$\tilde{M}(t_1, t_2) = \begin{cases} 1 & t_1 = t_2 = 0; \\ 0 & t_1 + t_2 = 1; \\ \min_{0 \leq \alpha_1 \leq \lfloor \frac{t_1}{2} \rfloor, 0 \leq \alpha_2 \leq \lfloor \frac{t_2}{2} \rfloor} \text{Min}C(t_1, t_2, \alpha_1, \alpha_2) & \text{otherwise,} \end{cases}$$

<sup>1</sup> In [70, Theorem 3.3] the equivalent question of type  $[0, \lfloor \frac{t_1}{2} \rfloor, \frac{t_2}{2} + B_1, \lfloor \frac{t_3-t-1}{2} \rfloor]$  is used.

<sup>2</sup> In [70, Theorem 3.3] the equivalent question, type  $[0, \lfloor \frac{t_1}{2} \rfloor, \lfloor \frac{t_2}{2} \rfloor + B_1, \lfloor \frac{t_3-t-1}{2} \rfloor]$ , is used.

<sup>3</sup> In [70, Theorem 3.4] the equivalent question of type  $[0, \lfloor \frac{t_1}{2} \rfloor, \lfloor \frac{t_2}{2} \rfloor, \lfloor \frac{t_3+t}{2} \rfloor]$ , is used.

where setting  $\tilde{\sigma} = (0, t_1, t_2, 0)$  and  $\tilde{\sigma}_{yes} = (0, y_1, y_2, y_3)$  and  $\tilde{\sigma}_{no} = (0, n_1, n_2, n_3)$  being the resulting *yes* and *no* states when question  $[0, \alpha_1, \alpha_2]$  is asked in state  $\tilde{\sigma}$  and

- $k_1 = ch(\tilde{\sigma}_{yes})$
- $k_2 = ch(\tilde{\sigma}_{no})$
- $k_3 = ch(0, y_1, y_2, \tilde{M}(y_1, y_2))$
- $k_4 = ch(0, n_1, n_2, \tilde{M}(n_1, n_2))$

we have  $\tilde{Min}C(t_1, t_2, \alpha_1, \alpha_2) = \min\{t_3 \mid ch(0, t_1, t_2, t_3) > \max\{k_1, k_2, k_3, k_4\}\}$ .

It is not hard to see that the quantities  $\tilde{M}$  and  $\tilde{Min}C$  can be computed by a dynamic programming approach. The following proposition shows that they correctly characterize quadruples in  $\tilde{W}$ .

**Proposition 3.4.** *The quadruple  $\tau = (0, t_1, t_2, d)$  is in  $\tilde{W}$  if and only if  $d \geq \tilde{M}(t_1, t_2)$ .*

*Proof.* The statement immediately follows by [70, Definition 3.14, Corollary 3.15, Proposition 3.16] where it is shown that a state  $(0, t_1, t_2, t_3)$  is nice if and only if  $t_3 \geq M(t_1, t_2)$  where  $M(t_1, t_2)$  is defined like  $\tilde{M}(t_1, t_2)$  but for the fact that  $\alpha_i$  can be as large as  $t_i$  for  $i = 1, 2$ .  $\square$

By using the above functions, we can have an algorithm that exhaustively compute all the 0-typical states of character  $\leq 12$  which are not in  $\tilde{W}$ . It turns out all such states have character  $\leq 11$ . The states are reported in Table 3.1. It turns out that the output of the computation of  $\tilde{M}(t_1, t_2)$  coincides with the output of the computation of  $M(t_1, t_2)$ , the function considered in [70] which does not require  $\alpha_i \leq \lceil \frac{t_i}{2} \rceil$  ( $i = 1, 2$ ). In other words, every well shaped state of character  $\leq 12$  which is nice is also 4-interval nice.

The following proposition summarizes the above discussion.

**Proposition 3.5.** *Let  $\sigma$  be a 0-typical well shaped state of type  $(0, t_1, t_2, t_3)$ . If  $ch(0, t_1, t_2, t_3) \geq 12$ , then  $\sigma$  is 4-interval nice. If  $ch(0, t_1, t_2, t_3) \leq 12$  then  $\sigma$  is 4-interval nice unless it is one of the states in Table 3.1.*

**Lemma 3.10.** *Let  $\sigma$  be a well-shaped state of type  $(1, 0, 3, n)$  with  $n \geq 7$ , then  $\sigma$  is 4-interval nice. Namely, there exists a 4-interval question  $Q$  such that both the resulting states  $\sigma_{yes}$  and  $\sigma_{no}$  are 4-interval nice.*

*Proof.* From [101, Lemma 5] for  $7 \leq n \leq 9$ , we have that if the question  $Q$  is defined as in Table 3.2 then the resulting states  $\sigma_{yes}$  and  $\sigma_{no}$  have character strictly smaller than  $ch(\sigma)$  and they are nice. In addition, by Lemmas 3.3 and 3.6, we have that each question reported in the table can be implemented using at most 4-intervals. Moreover, the resulting states are also 4-interval nice as consequence of Lemma 3.8 and as consequence of the observation that the state  $(0, 0, 4, 0)$  corresponds to the state  $(4, 0)$  that is nice by [109] and, since the state has only four elements, it is straightforward to implement every question with at most 4-intervals. From those observations, it follows that a well shaped state of type  $(1, 0, 3, n)$ , for  $7 \leq n \leq 9$  is 4-interval nice.

Let us now consider the case  $n \geq 10$ . Let the question  $Q$  be a question of type  $[1, 0, 3, x]$ , then the resulting states  $\sigma_{yes}$  and  $\sigma_{no}$  have types  $(1, 0, 0, 3 + x)$  and  $(0, 1, 3, n - x)$  respectively. Again from [101, Lemma 5] for  $n \geq 10$ , we have that if  $x = \lfloor (n + 3q + \binom{q}{3})/2 \rfloor$ , where  $ch(\sigma) = q + 1$ , the resulting states  $\sigma_{yes}$  and  $\sigma_{no}$  are nice. Moreover, from Lemma 3.6 it follows that the question  $Q$  can be implemented using at most 4 intervals, then we have that the resulting states  $\sigma_{yes}$  and  $\sigma_{no}$  are also well shaped. Indeed, note that  $\sigma_{yes}$  is 4-interval nice by Lemma 3.8 while  $\sigma_{no}$  is 0-typical, then it is 4-interval nice by Proposition 3.5.



Character State	Values of $t_3$	Character State	Values of $t_3$		
$ch = 6$	$(0, 1, 5, t_3)$	$6 \leq t_3 \leq 7$	$ch = 9$	$(0, 7, 16, t_3)$	$9 \leq t_3 \leq 30$
	$(0, 2, 1, t_3)$	$6 \leq t_3 \leq 13$		$(0, 7, 17, t_3)$	$9 \leq t_3 \leq 20$
	$(0, 2, 2, 6)$			$(0, 7, 18, t_3)$	$9 \leq t_3 \leq 10$
$ch = 7$	$(0, 2, 7, t_3)$	$7 \leq t_3 \leq 14$	$(0, 8, 9, t_3)$	$9 \leq t_3 \leq 54$	
	$(0, 3, 2, t_3)$	$7 \leq t_3 \leq 25$	$(0, 8, 10, t_3)$	$9 \leq t_3 \leq 44$	
	$(0, 3, 3, t_3)$	$7 \leq t_3 \leq 17$	$(0, 8, 11, t_3)$	$9 \leq t_3 \leq 34$	
	$(0, 3, 4, t_3)$	$7 \leq t_3 \leq 9$	$(0, 8, 12, t_3)$	$9 \leq t_3 \leq 24$	
			$(0, 8, 13, t_3)$	$9 \leq t_3 \leq 14$	
			$(0, 9, 8, t_3)$	$9 \leq t_3 \leq 18$	
		$ch = 10$	$(0, 13, 25, t_3)$	$10 \leq t_3 \leq 21$	
$ch = 8$	$(0, 3, 15, t_3)$	$8 \leq t_3 \leq 10$	$(0, 13, 26, 10)$		
	$(0, 4, 9, t_3)$	$8 \leq t_3 \leq 27$	$(0, 14, 17, t_3)$	$10 \leq t_3 \leq 53$	
	$(0, 4, 10, t_3)$	$8 \leq t_3 \leq 18$	$(0, 14, 18, t_3)$	$10 \leq t_3 \leq 42$	
	$(0, 4, 11, t_3)$	$8 \leq t_3 \leq 9$	$(0, 14, 19, t_3)$	$10 \leq t_3 \leq 31$	
	$(0, 5, 4, t_3)$	$8 \leq t_3 \leq 35$	$(0, 14, 20, t_3)$	$10 \leq t_3 \leq 20$	
	$(0, 5, 5, t_3)$	$8 \leq t_3 \leq 26$	$(0, 15, 14, t_3)$	$10 \leq t_3 \leq 30$	
	$(0, 5, 6, t_3)$	$8 \leq t_3 \leq 17$	$(0, 15, 15, t_3)$	$10 \leq t_3 \leq 19$	
	$(0, 5, 7, 8)$		$ch = 11$	$(0, 25, 30, t_3)$	$11 \leq t_3 \leq 13$

**Table 3.1:** The table shows all the states of character  $\leq 12$  which are 0-typical but non nice [70][101]. This set coincides with the set of types of well shaped states that are 0-typical but non 4-interval nice.

**Lemma 3.11.** Fix  $m \geq 33$  and let  $\sigma$  be a well shaped state of type  $(1, m, \binom{m}{2}, \binom{m}{3})$ . Then, there exists a sequence of three 4-interval questions such that all the resulting states are 4 interval-nice and of characters  $\leq ch(\sigma) - 3$ .

*Proof.* By [101, Lemma 7] for a state  $\sigma$  satisfying the hypothesis of the lemma, there exists a sequence of three questions such that all the resulting states are nice and of character  $\leq ch(\sigma) - 3$ . In particular, denoting by  $(1, b_0, c_0, d_0)$  the state of type  $(1, m, \binom{m}{2}, \binom{m}{3})$  the questions are defined according to the following rules, where  $Q_i$  is the  $i$ th question ( $i = 1, 2, 3$ ) and  $YES_i$  (resp.  $NO_i$ ) denotes the state resulting from the answer *yes* (resp. *no*) to question  $Q_i$ .

$$\text{Let } q = ch(1, m, \binom{m}{2}, \binom{m}{3}) - 2.$$

**Table 3.2:** First part of the proof of Lemma 3.10 and first part of the proof of [101, Lemma 5]

Case State	Question	Implementation	Yes-State	No-State
$n = 7$	$(1, 0, 3, 7)$	$[1, 0, 0, 2]?$ Lemma 3.6	$(1, 0, 0, 5)^a$	$(0, 1, 3, 5)$
	$(0, 1, 3, 5)$	$[0, 1, 0, 5]?$ Lemma 3.3	$(0, 1, 0, 8)^a$	$(0, 0, 4, 0)^b$
$n = 8$	$(1, 0, 3, 8)$	$[1, 0, 0, 3]?$ Theorem 3.6	$(1, 0, 0, 6)^a$	$(0, 1, 3, 5)$
	$(0, 1, 3, 5)$	$[0, 1, 0, 5]?$ Lemma 3.3	$(0, 1, 0, 8)^a$	$(0, 0, 4, 0)^b$
$n = 9$	$(1, 0, 3, 9)$	$[1, 0, 0, 4]?$ Theorem 3.6	$(1, 0, 0, 7)^a$	$(0, 1, 3, 5)$
	$(0, 1, 3, 5)$	$[0, 1, 0, 5]?$ Lemma 3.3	$(0, 1, 0, 8)^a$	$(0, 0, 4, 0)^b$

<sup>a</sup> 4-interval nice by Lemma 3.8. <sup>b</sup> Nice by [109] and having only four elements all the questions can be implemented using at most 4 intervals



Let  $Q_1$  be a question of type  $|Q_1| = [1, \lfloor \frac{b_0}{2} \rfloor, \lfloor \frac{c_0}{2} \rfloor - \lfloor \frac{b_0}{2} \rfloor, x]?$  With  $x = \lfloor \frac{\alpha}{2} \rfloor$  where  $\alpha = d_0 + 2 \lfloor \frac{c_0}{2} \rfloor - c_0 - 2 \lfloor \frac{b_0}{2} \rfloor - \binom{q+1}{3} + \binom{q+1}{2}(b_0 + 1 - 2 \lfloor \frac{b_0}{2} \rfloor) + (q+2)(c_0 + 4 \lfloor \frac{b_0}{2} \rfloor - b_0 - 2 \lfloor \frac{c_0}{2} \rfloor)$ .

Thus,  $YES_1 = (1, \lfloor \frac{b_0}{2} \rfloor, b_0 - 2 \lfloor \frac{b_0}{2} \rfloor + \lfloor \frac{c_0}{2} \rfloor, c_0 - \lfloor \frac{c_0}{2} \rfloor + \lfloor \frac{b_0}{2} \rfloor + x)$  and  $NO_1 = (0, b_0 + 1 - \lfloor \frac{b_0}{2} \rfloor, 2 \lfloor \frac{b_0}{2} \rfloor + c_0 - \lfloor \frac{c_0}{2} \rfloor, \lfloor \frac{c_0}{2} \rfloor - \lfloor \frac{b_0}{2} \rfloor + d_0 - x)$  are the resulting states.

Let us now denote the type of  $YES_1$  as  $(1, b_1, c_1, d_1)$  then let  $Q_2$  be a question of type  $|Q_2| = [1, 1, \lfloor \frac{c_1}{2} \rfloor - \lfloor \frac{b_1}{2} \rfloor, y]?$  with  $y = \lfloor \frac{\beta}{2} \rfloor$  where  $\beta = (b_1 - 1) \binom{q}{2} - \binom{q}{3} + (q + 1)(c_1 + 2 - b_1 + 2 \lfloor \frac{b_1}{2} \rfloor - 2 \lfloor \frac{c_1}{2} \rfloor) + d_1 - c_1 + 2 \lfloor \frac{c_1}{2} \rfloor - 2 \lfloor \frac{b_1}{2} \rfloor$ .

Thus, the type of the resulting states  $YES_2$  and  $NO_2$  are given by  $|YES_2| = (1, 1, b_1 - 1 + \lfloor \frac{c_1}{2} \rfloor - \lfloor \frac{b_1}{2} \rfloor, c_1 - \lfloor \frac{c_1}{2} \rfloor + \lfloor \frac{b_1}{2} \rfloor + y)$  and  $|NO_2| = (0, b_1, 1 + c_1 - \lfloor \frac{c_1}{2} \rfloor + \lfloor \frac{b_1}{2} \rfloor, \lfloor \frac{c_1}{2} \rfloor - \lfloor \frac{b_1}{2} \rfloor + d_1 - y)$ .

Let us denote the type of  $YES_2$  by  $(1, b_1, c_1, d_1)$  then question  $Q_3$  is chosen to be of type  $|Q_3| = [1, 0, 2, z]?$  with  $z = \lfloor \frac{d_2 + 4 - c_2 - \binom{q-1}{3} + 2 \binom{q-1}{2} + q(c_2 - 5)}{2} \rfloor$ .

We have that the resulting states have type  $|YES_3| = (1, 0, 3, c_2 - 2 + z)$  and  $|NO_3| = (0, 2, c_2 - 2, 2 + d_2 - z)$ .

It remains to show that each one of the above questions can be implemented using only 4 intervals and guaranteeing that the resulting state is also well-shaped. This is true of  $Q_1$  by Theorem 3.1; in addition question  $Q_2$  applied to the state  $YES_1$  satisfies the constraints of Lemma 3.4. Finally, question  $Q_3$  asked in state  $YES_2$  satisfies the constraints of Lemma 3.5.

Finally, we observe that the states  $NO_1$ ,  $NO_2$  and  $NO_3$  are 4-interval nice by Proposition 3.5 and the states  $YES_3$  is 4-interval nice by Lemma 3.10.  $\square$

**Table 3.3:** Case  $m = 1$ , after the first question and case  $m = 4$ , after the first four questions

	State	Question	Implementation	Yes-State	No-State
$m = 1$	$(1, 1, 0, 0)$	$[1, 0, 0, 0]?$	Straightforward	$(1, 0, 1, 0)$	$(0, 2, 0, 0)$
	$(1, 0, 1, 0)$	$[1, 0, 0, 0]?$	Straightforward	$(1, 0, 0, 1)^a$	$(0, 1, 1, 0)$
	$(0, 2, 0, 0)$	$[0, 1, 0, 0]?$	Straightforward	$(0, 1, 1, 0)$	$(0, 1, 1, 0)$
	$(0, 1, 1, 0)$	$[0, 1, 0, 0]?$	Straightforward	$(0, 1, 0, 1)^a$	$(0, 0, 2, 0)^b$
$m = 4$	$(1, 4, 6, 4)$	$[1, 1, 3, 2]?$	Lemma 3.4	$(1, 1, 6, 5)$	$(0, 4, 4, 5)$
	$(1, 1, 6, 5)$	$[1, 0, 2, 3]?$	Lemma 3.5	$(1, 0, 3, 7)^c$	$(0, 2, 4, 4)$
	$(0, 4, 4, 5)$	$[0, 2, 2, 3]?$	Lemma 3.3	$(0, 2, 4, 5)$	$(0, 2, 4, 4)$
	$(0, 2, 4, 4)$	$[0, 1, 2, 2]?$	Lemma 3.3	$(0, 1, 3, 4)$	$(0, 1, 3, 4)$
	$(0, 2, 4, 5)$	$[0, 1, 2, 3]?$	Lemma 3.3	$(0, 1, 3, 5)$	$(0, 1, 3, 4)$
	$(0, 1, 3, 4)$	$[0, 1, 0, 4]?$	Lemma 3.3	$(0, 1, 0, 7)^a$	$(0, 0, 4, 0)^b$
	$(0, 1, 3, 5)$	$[0, 1, 0, 5]?$	Lemma 3.3	$(0, 1, 0, 8)^a$	$(0, 0, 4, 0)^b$

<sup>a</sup> 4-interval nice by Lemma 3.8. <sup>b</sup> Nice by [109] and having less than four elements all the questions can be implemented using at most 4 intervals

**Corollary 3.1.** Fix  $m \in \mathcal{N} = \mathbb{N} \setminus \{2, 3, 5\}$  and let  $\sigma$  be a well shaped state of type  $(1, m, \binom{m}{2}, \binom{m}{3})$ . Then  $\sigma$  is 4-interval nice.

*Proof.* The searching strategy for  $m = 1$  is summarized in Table 3.3. Since the number of elements is  $\leq 2$  trivially every question is a 4-interval question. The strategy for

$m = 4$  are summarized in Table 3.3. The reason for the implementability of each question involved by only using 4 intervals is indicated in the column “Implementation” referring to the appropriate lemma.

Before proceeding to the analysis of the remaining cases, we note that given two states  $\sigma = (t_0, t_1, t_2, t_3)$  and  $\sigma' = (t'_0, t'_1, t'_2, t'_3)$  such that  $ch(\sigma) = ch(\sigma')$  and for all  $i = 0, 1, 2, 3$  it holds that  $t'_i \leq t_i$ , thus a perfect strategy for  $\sigma$  immediately gives a perfect strategy for  $\sigma'$ . Therefore, in order to prove the claim for  $6 \leq m \leq 32$ , it suffice to consider the cases  $m \in \{8, 12, 17, 23, 32\}$ . In fact for each  $6 \leq m \leq 32$  such that  $m \notin \{8, 12, 17, 23, 32\}$  there exists  $m' \in \{8, 12, 17, 23, 32\}$  such that  $m' \geq m$  and  $ch(2^m, 0, 0, 0) = ch(2^{m'}, 0, 0, 0)$ . Hence, a perfect 4-interval strategy for the former implies a perfect 4-interval strategy for the latter.

For each  $m \in \{8, 12, 17, 23, 32\}$  a 4-interval perfect strategy is described in Table 3.4. The analysis of the first three questions together with Lemma 3.10 and Proposition 3.5 is sufficient to prove the 4-interval niceness. Finally, for every  $m \geq 33$ , the state  $(1, m, \binom{m}{2}, \binom{m}{3})$  is 4-interval nice by Lemma 3.11.  $\square$

**Table 3.4:** Proof of Corollary 3.1 for  $6 \leq m \leq 32$  and proof of [101, Lemma 6]

Case State	Question	Implementation	Yes-State	No-State	
$m = 8$ (1, 8, 28, 56)	[1, 4, 10, 22]?	Theorem 3.1	(1, 4, 14, 40)	(0, 5, 22, 44) <sup>a</sup>	
	(1, 4, 14, 40)	[1, 1, 5, 36]?	Lemma 3.4	(1, 1, 8, 45)	(0, 4, 10, 9) <sup>a</sup>
	(1, 1, 8, 45)	[1, 0, 2, 29]?	Lemma 3.5	(1, 0, 3, 35) <sup>b</sup>	(0, 2, 6, 18) <sup>a</sup>
$m = 12$ (1, 12, 66, 220)	[1, 6, 27, 110]?	Theorem 3.1	(1, 6, 33, 149)	(0, 7, 45, 137) <sup>a</sup>	
	(1, 6, 33, 149)	[1, 1, 13, 136]?	Lemma 3.4	(1, 1, 18, 156)	(0, 6, 21, 26) <sup>a</sup>
	(1, 1, 18, 156)	[1, 0, 2, 120]?	Lemma 3.5	(1, 0, 3, 136) <sup>b</sup>	(0, 2, 16, 38) <sup>a</sup>
$m = 17$ (1, 17, 136, 680)	[1, 8, 60, 373]?	Theorem 3.1	(1, 8, 69, 449)	(0, 10, 84, 367) <sup>a</sup>	
	(1, 8, 69, 449)	[1, 1, 30, 344]?	Lemma 3.4	(1, 1, 37, 383)	(0, 2, 35, 69) <sup>a</sup>
	(1, 1, 37, 383)	[1, 0, 2, 316]?	Lemma 3.5	(1, 0, 3, 351) <sup>b</sup>	(0, 2, 35, 69) <sup>a</sup>
$m = 23$ (1, 23, 253, 1771)	[1, 11, 115, 946]?	Theorem 3.1	(1, 11, 127, 1084)	(0, 13, 149, 940) <sup>a</sup>	
	(1, 11, 127, 1084)	[1, 1, 58, 767]?	Lemma 3.4	(1, 1, 68, 836)	(0, 11, 70, 375) <sup>a</sup>
	(1, 1, 68, 836)	[1, 0, 2, 700]?	Lemma 3.5	(1, 0, 3, 766) <sup>b</sup>	(0, 2, 66, 138) <sup>a</sup>
$m = 32$ (1, 32, 496, 4960)	[1, 16, 232, 2545]?	Theorem 3.1	(1, 16, 248, 2809)	(0, 17, 280, 2647) <sup>a</sup>	
	(1, 16, 248, 2809)	[1, 1, 116, 1852]?	Lemma 3.4	(1, 1, 131, 1984)	(0, 16, 133, 1073) <sup>a</sup>
	(1, 1, 131, 1984)	[1, 0, 2, 1635]?	Lemma 3.5	(1, 0, 3, 1764) <sup>b</sup>	(0, 2, 129, 351) <sup>a</sup>

<sup>a</sup> 4-interval nice by Proposition 3.5. <sup>b</sup> 4-interval nice by Lemma 3.10

---

## Prefix normal words

This chapter treats prefix normal words.

Prefix normal words are binary words with the property that no factor has more 1s than the prefix of the same length. For example, the word 1101011010 is prefix normal, while 1101011011 is not since the factor 11011 has more ones than the prefix 11010. Prefix normal words were introduced by Fici and Lipták in [57] motivated by the problem of Jumbled Pattern Matching [5, 6, 7, 23, 29, 34, 66, 85], a type of approximate pattern matching. In the binary version of Jumbled Pattern Matching [49, 65, 98], given a text and two non negative integers  $x$  and  $y$ , we ask if, within the text, there exists a substring that has exactly  $x$  0s and  $y$  1s. Prefix normal words can be used to build an index for the problem, via so-called *prefix normal forms*.

Prefix normal words have been recently extensively studied [11, 24, 25, 35, 36, 37, 62]. The language of prefix normal words has been connected to the Binary Reflected Gray Code [121], and prefix normal words have been applied to a certain class of graphs [20]. Furthermore, it has been shown that prefix normal words form a bubble language [24, 25, 119] a family of binary languages with efficient<sup>1</sup> generation algorithms [120] which can be listed as a Gray code. A Gray code is a listing of a class of words such that consecutive words differ by a constant number of operations. Bubble languages include well-known classes of words such as Lyndon words,  $k$ -ary Dyck words, and necklaces.

Generation algorithms can be used to count the number of prefix normal words of a given length. The sequence of numbers of prefix normal words of length  $n$  is present in the On-Line Encyclopedia of Integer Sequences (OEIS [124]) as sequence A194850. Furthermore, there are two other sequences related to prefix normal words in the OEIS: A238109 (a list of prefix normal words over the alphabet  $\{1, 2\}$ ), and A238110 (maximum size of the equivalence class of words with same prefix normal form).

In Section 4.2 we present a new recursive generation algorithm for prefix normal words of length  $n$ . In combinatorial generation, the aim is to find a way of efficiently listing (but not necessarily outputting) each one of a given class of combinatorial objects. Often it is necessary to examine each one of such objects, even though their number may be very large, typically exponential. The latest volume 4A of Donald Knuth's *The*

---

<sup>1</sup> In combinatorial generation, the term *efficient* is used in the sense that the cost to generate one word, without outputting it, should be small — in the best case, this cost is constant amortized time (CAT).

*Art of Computer Programming* devotes over 200 pages to combinatorial generation of basic combinatorial patterns [82], such as permutations and bitstrings.

At the time of the development of this algorithm, and of its publication [35, 36], the only known result on the running time of the previous generation algorithm from [24] was amortized  $\mathcal{O}(n)$  per word. It was conjectured that its actual running time is amortized  $\mathcal{O}(\log n)$  per word. Recent results [112] show that the *critical prefix* length — the critical prefix of a binary word is the first run of 1s followed by the first run of 0s. — of a prefix normal word is  $\mathcal{O}(\log^2 n)$  in expectation over all prefix normal words of length  $n$ . This result improves the running time of the previous generation algorithm from [24] to amortized  $\mathcal{O}(\log^2 n)$  per word.

Therefore our new algorithm was superior in that its running time is worst-case  $\mathcal{O}(n)$  time per word, and it allows new insights into properties of prefix normal words. Our new algorithm recursively generates all prefix normal words from a seed word, applying two operations, which are referred to as *bubble* and *flip*. It can be applied (a) to produce all prefix normal words of fixed length, or (b) to produce all prefix normal words of fixed length sharing the same critical prefix. This could prove useful in counting prefix normal words of fixed length: this number grows exponentially and asymptotically is  $2^{n-\theta(\log^2 n)}$  [11], for prefix normal words of length  $n$ . It has been also shown that this number is related to the number of prefix normal words of length  $n$  that are palindromes [62]. However, neither a closed form nor a generating function are known [25]. As a step towards counting prefix normal words, we characterize the number of prefix normal words of length  $n$  having critical prefix of length greater than  $\lceil n/2 \rceil$ . Furthermore, a slight change in the algorithm produces a Gray code on prefix normal words of length  $n$ .

In Section 4.3 we show some results about the extension of prefix normal words. Pursuing the investigation of underlying structures of prefix normal words, we introduce two operations on finite prefix normal words, which produce, in the limit, infinite prefix normal words, when repeatedly applied. There exists periodic, ultimately periodic, and aperiodic infinite prefix normal words (formal definition follow).

Among aperiodic infinite words, Sturmian words are a well known and widely studied class of binary words. In Section 4.4 we use one of the previous extension operations to establish connections between infinite prefix normal words and Sturmian words, thus providing a complete characterization of Sturmian words which are prefix normal. In Section 4.5 we establish connections between infinite prefix normal words and lexicographic order. As in the finite case, we show that infinite prefix normal words are infinite prenecklaces (formal definition follows) and that there exist infinite binary words of each of the following categories: both prefix normal and Lyndon, prefix normal but not Lyndon, Lyndon but not prefix normal, and neither of the two. Furthermore, we compare prefix normal words with the max- and min-words of [110]. In Section 4.6 we recall the notion of prefix normal *forms* from [57], as well as the fact that prefix normal forms of a word are prefix normal words. We exploit the connections between prefix normal forms and Abelian complexity [115]. We show that it is always possible to obtain the Abelian complexity of a word given its prefix normal forms, while the converse is not always possible. We provide sufficient conditions to obtain the prefix normal forms of a word, given its Abelian complexity. Finally, in Section 4.7 we characterize the periodicity and aperiodicity of prefix normal words with respect to their minimum density, a new parameter we introduce in this context.

Partial contents of this chapter have been published in [35, 36, 37].

## 4.1 Basics

A finite (resp. infinite) binary word  $w$  is a finite (resp. infinite) sequence of elements from  $\{0, 1\}$ . Thus, an infinite word is a mapping  $w : \mathbb{N} \rightarrow \{0, 1\}$ , where  $\mathbb{N}$  denotes the set of positive integers. We denote the  $i$ 'th character of  $w$  by  $w_i$ , and, if  $w$  is finite, its length by  $|w|$ . Note that we index words from 1. The empty word, denoted  $\varepsilon$ , is the unique word with length 0. The set of binary words of length  $n$  is denoted  $\{0, 1\}^n$ , the set of all finite words by  $\{0, 1\}^* = \cup_{n \geq 0} \{0, 1\}^n$ , and the set of all infinite binary words by  $\{0, 1\}^\omega$ . For a finite word  $u = u_1 \cdots u_n$  we denote by  $u^{\text{rev}} = u_n \cdots u_1$  the reverse of  $u$ , and by  $\bar{u} = \bar{u}_1 \cdots \bar{u}_n$  the complement of  $u$ , where  $\bar{a} = 1 - a$  for  $a \in \{0, 1\}$ . For two words  $u, v$ , where  $u$  is finite and  $v$  is finite or infinite, we write  $uv$  for their concatenation. If  $w = uxv$  then  $u$  is called a prefix,  $x$  a factor (or substring), and  $v$  a suffix of  $w$ . We denote by  $w_i \cdots w_j$ , for  $i \leq j$ , the factor of  $w$  spanning the positions  $i$  through  $j$ . We denote the set of factors of  $w$  by  $Fct(w)$  and its prefix of length  $i$  by  $\text{pref}_w(i)$ , where  $\text{pref}_w(0) = \varepsilon$ . For a finite word  $u$ , we write  $|u|_1$  for the number of 1s, and  $|u|_0$  for the number of 0s in  $u$ , and refer to  $|u|_1$  as the *weight* of  $u$ . The *Parikh vector* of  $u$  is  $pv(u) = (|u|_0, |u|_1)$ . A word  $w$  is called *balanced* if for all  $u, v \in Fct(w)$ ,  $|u| = |v|$  implies  $||u|_1 - |v|_1| \leq 1$ , and *c-balanced* if  $|u| = |v|$  implies  $||u|_1 - |v|_1| \leq c$ .

For an integer  $k \geq 1$  and  $u \in \{0, 1\}^n$ ,  $u^k$  denotes the  $kn$ -length word  $uuu \cdots u$  ( $k$ -fold concatenation of  $u$ ) and  $u^\omega$  the infinite word  $uuu \cdots$ . An infinite word  $w$  is called *periodic* if  $w = u^\omega$  for some non-empty word  $u$ , and *ultimately periodic* if  $w = vu^\omega$  for some  $v$  and non-empty  $u$ . If  $w = vu^\omega$ , with  $v$  possibly empty, then we refer to  $u$  as a period of  $w$  [47]. A word that is neither periodic nor ultimately periodic is called *aperiodic*. We set  $0 < 1$  and denote by  $\leq_{\text{lex}}$  the *lexicographic order* between words, i.e.  $u \leq_{\text{lex}} v$  if  $u$  is a prefix of  $v$  or there is an index  $i \geq 1$  s.t.  $u_i < v_i$  and  $\text{pref}_u(i-1) = \text{pref}_v(i-1)$ . For an operation  $\text{op} : \{0, 1\}^* \rightarrow \{0, 1\}^*$ , we denote by  $\text{op}^{(i)}$  the  $i$ th iteration of  $\text{op}$ . Further, let  $\text{op}^*(w) = \{\text{op}^{(i)}(w) \mid i \geq 1\}$  and  $\text{op}^\omega(w) = \lim_{i \rightarrow \infty} \text{op}^{(i)}(w)$ , if it exists.

**Definition 4.1 (Prefix weight, prefix density, maximum and minimum 1s and 0s functions).** Let  $w$  be a (finite or infinite) binary word. We define the following functions:

- $P_w(i) = |\text{pref}_w(i)|_1$ , the weight of the prefix of length  $i$ ,
- $D_w(i) = P_w(i)/i$ , the density of the prefix of length  $i$ ,
- $F_w^1(i) = \max\{|u|_1 : u \in Fct(w), |u| = i\}$  and  $f_w^1(i) = \min\{|u|_1 : u \in Fct(w), |u| = i\}$ , the maximum resp. minimum number of 1s in a factor of length  $i$ ,
- $F_w^0(i) = \max\{|u|_0 : u \in Fct(w), |u| = i\}$  and  $f_w^0(i) = \min\{|u|_0 : u \in Fct(w), |u| = i\}$ , the maximum resp. minimum number of 0s in a factor of length  $i$ .

Note that in the context of succinct indexing, the function  $P_w(i)$  is often called  $\text{rank}_1(w, i)$ . If clear from the context we write  $P(i)$  for  $P_w(i)$ . We are now ready to define prefix normal words.

**Definition 4.2 (Prefix normal words, prefix normal condition).** A (finite or infinite) binary word  $w$  is called 1-prefix normal, or simply prefix normal, if  $P_w(i) = F_w^1(i)$  for all  $i \geq 1$  (for all  $1 \leq i \leq |w|$  if  $w$  is finite). It is called 0-prefix normal if  $i - P_w(i) = F_w^0(i)$  for all  $i \geq 1$  (for all  $1 \leq i \leq |w|$  if  $w$  is finite). We denote the set of all finite 1-prefix normal words by  $\mathcal{L}_{\text{fin}}$ , the set of all infinite 1-prefix normal words by  $\mathcal{L}_{\text{inf}}$ , and  $\mathcal{L} = \mathcal{L}_{\text{fin}} \cup \mathcal{L}_{\text{inf}}$ . The set of prefix normal words of length  $n$  is denoted  $\mathcal{L}_n$ . Given a binary word  $w$ , we say that a factor  $u$  of  $w$  satisfies the prefix normal condition if  $|u|_1 \leq P_w(|u|)$ .

In other words, a word  $w$  is prefix normal (i.e. 1-prefix normal) if no factor  $u$  of  $w$  has more 1s than the prefix of the same length, i.e.,  $|u|_1 \leq P_w(|u|)$ , thus satisfy the prefix normal condition. Note that unless further specified, by prefix normal we mean 1-prefix normal.

For a prefix normal word  $u$ , every factor of the infinite word  $u0^\omega$  respects the prefix normal condition, thus  $u0^\omega$  is an infinite prefix normal word. Clearly, as for finite words, it holds that an infinite word is prefix normal if and only if all its prefixes are prefix normal. Therefore, the existence of infinite prefix normal words can also be derived from König's Lemma (see [92]), which states that the existence of an infinite prefix-closed set of finite words implies the existence of an infinite word which has all its prefixes in the set.

*Example 5.* The word 110100110110 is not prefix normal since the factor 11011 has four 1s, which is more than in the prefix 11010 of length 5. The word 110100110010, on the other hand, is prefix normal. The infinite word  $(11001)^\omega$  is not prefix normal, because it has 111 as a factor, which has more 1s than the prefix of length 3, but the word  $(11010)^\omega$  is. The periodic words  $0^\omega$ ,  $1^\omega$  and  $(10)^\omega$  are prefix normal; the ultimately periodic word  $1(10)^\omega$  is prefix normal; the aperiodic word  $10100100010000 \cdots = \lim_{n \rightarrow \infty} 1010^2 \cdots 10^n$  is prefix normal.

It is easy to see that the number of prefix normal words grows exponentially, by noting that  $1^n w$  is prefix normal for any  $w$  of length  $n$ . In Table 4.1, we list all prefix normal words for lengths  $n \leq 5$ . It has been shown that the number of prefix normal words is  $2^{n-\theta(\log^2 n)}$ , however finding a closed form remains an open problem, see [25] for partial results and [11] for asymptotics. The cardinalities of  $\mathcal{L}_n$  for  $n \leq 50$  can be found in the On-Line Encyclopedia of Integer Sequences (OEIS [124]) as sequence A194850.

**Table 4.1:** The set  $\mathcal{L}_n$  of prefix normal words of length  $n$  for  $n = 1, 2, 3, 4, 5$ .

$\mathcal{L}_1$	$\mathcal{L}_2$	$\mathcal{L}_3$	$\mathcal{L}_4$	$\mathcal{L}_5$
0	00	000 110	0000 1010 1110	00000 10010 11000 11011 11110
1	10 11	100 111 101	1000 1100 1111 1001 1101	10000 10100 11001 11100 11111 10001 10101 11010 11101

Next, we give some basic facts about prefix normal words which will be needed in the following.

**Fact 4.1 (Basic facts about prefix normal words [25])** Let  $w \in \{0, 1\}^n$ .

- (i) If  $w \in \mathcal{L}_{\text{fin}}$ , then either  $w = 0^n$  or  $w_1 = 1$ .
- (ii)  $w \in \mathcal{L}$  if and only if  $\text{pref}_i(w) \in \mathcal{L}$  for  $i = 1, \dots, n$ .
- (iii) If  $w \in \mathcal{L}$  then  $w0^i \in \mathcal{L}$  for all  $i = 1, 2, \dots$ .
- (iv) Let  $w \in \mathcal{L}_{\text{fin}}$ . Then  $w1 \in \mathcal{L}_{\text{fin}}$  if and only if for all  $1 \leq i < n$ , we have  $P_w(i+1) > |w_{n-i+1} \cdots w_n|_1$ .

Furthermore, we introduce the *critical prefix* of word. The length of the critical prefix plays an important role in the analysis of the previous generation algorithm for prefix normal words [24].

**Definition 4.3 (Critical prefix).** A non-empty binary word  $w$  can be uniquely written in the form  $w = 1^s 0^t \gamma$ , where  $s, t \geq 0$ ,  $s = 0$  implies  $t > 0$ , and  $\gamma \in 1\{0, 1\}^* \cup \{\varepsilon\}$ . We refer to  $1^s 0^t$  as the *critical prefix* of  $w$ .

*Example 4.1.* For example, the critical prefix of 1100001001 is 110000, that of 0011101001 is 00, while the critical prefix of 1111000000 is 1111000000.

In [112], the authors show that the *critical prefix* length of a prefix normal word is  $\mathcal{O}(\log^2 n)$  in expectation over all prefix normal words of length  $n$ . In Section 4.2.3, we will see how to adapt our algorithm to generate all prefix normal words with critical prefix  $1^s 0^t$  in a single execution.

We, now, briefly discuss *combinatorial Gray codes*. Recall that a *Gray code* is a listing of all bitstrings (or binary words) of length  $n$  such that two successive words differ by exactly one bit. In other words, a Gray code is a sequence  $w^{(1)}, w^{(2)}, \dots, w^{(2^n)} \in \{0, 1\}^n$  such that  $d_H(w^{(i)}, w^{(i+1)}) = 1$  for  $i = 1, \dots, 2^n - 1$ , where  $d_H(x, y) = |\{1 \leq j \leq n : x_j \neq y_j\}|$  is the *Hamming distance* between two equal-length words  $x$  and  $y$ .

This definition has been generalized in several ways, we give a definition following [117, ch. 5].

**Definition 4.4 (Combinatorial Gray Code).** Given a set of combinatorial objects  $\mathcal{S}$  and a relation  $C$  on  $\mathcal{S}$  (the *closeness relation*), a combinatorial Gray code for  $\mathcal{S}$  is a listing  $s_1, s_2, \dots, s_{|\mathcal{S}|}$  of the elements of  $\mathcal{S}$ , such that  $(s_i, s_{i+1}) \in C$  for  $i = 1, 2, \dots, |\mathcal{S}| - 1$ . If we also require that  $(s_{|\mathcal{S}|}, s_1) \in C$ , then the code is called *cyclic*.

In particular, a listing of the elements of a binary language  $\mathcal{S} \subseteq \{0, 1\}^n$ , such that each two subsequent words have Hamming distance bounded by a constant, is a combinatorial Gray code for  $\mathcal{S}$ . Note that the specification ‘combinatorial’ is often dropped, so the term *Gray code* is frequently used in this more general sense.

To close this section, we introduce the concepts of *minimum density* and *slope*, related to the *density* of the word.

**Definition 4.5 (Minimum density, minimum density prefix, slope).** Let  $w \in \{0, 1\}^* \cup \{0, 1\}^\omega$ . Define the minimum density of  $w$  as  $\delta(w) = \inf\{D_w(i) \mid 1 \leq i\}$ . If this infimum is attained somewhere, then we also define

$$\iota(w) = \min\{j \mid \forall i : D_w(j) \leq D_w(i)\}, \quad \text{and} \quad \kappa(w) = P_w(\iota(w)).$$

We refer to  $\text{pref}_w(\iota(w))$  as the *minimum-density prefix*, the *shortest prefix with density*  $\delta(w)$ .

For an infinite word  $w$ , we define the *slope* of  $w$  as  $\lim_{i \rightarrow \infty} D_w(i)$ , if this limit exists.

*Example 4.2.* For  $w = 110100101001$  and  $u = 110100101010$  we have  $\delta(w) = 5/11$ ,  $\iota(w) = 11$ ,  $\kappa(w) = 5$ , and  $\delta(u) = 1/2$ ,  $\iota(u) = 6$ ,  $\kappa(u) = 3$ . For the infinite words  $v = (10)^\omega$  and  $v' = 1(10)^\omega$ , we have  $\delta(v) = \delta(v') = 1/2$ , and  $\iota(v) = 2$ ,  $\kappa(v) = 1$ , while  $\iota(v')$  is undefined, since no prefix attains density  $1/2$ .

*Remark 1.* Note that  $\iota(w)$  is always defined for finite words, while for infinite words, a prefix which attains the infimum may or may not exist. We note further that density and slope are different properties of (infinite) binary words. In particular, while  $\delta(w)$  exists for every  $w$ , the limit  $\lim_{i \rightarrow \infty} D_w(i)$  may not exist, i.e.,  $w$  may or may not have a slope.



As an example, consider the word  $w = v_0v_1v_2 \cdots$ , where for each  $i$ ,  $v_i = 1^i 0^{2^i}$ . Then,  $\delta(w) = 1/2$  and  $\lim_{i \rightarrow \infty} D_w(i)$  does not exist, since  $D_w(i)$  has an infinite subsequence constant  $1/2$ , and another which tends to  $2/3$ .

Moreover, even for words  $w$  for which the slope is defined, this can be different from the minimum density. If  $w$  has slope  $\alpha$ , then  $\alpha = \delta(w)$  if and only if for all  $i$ ,  $D_w(i) \geq \alpha$ . For instance, the infinite word  $01^\omega$  has slope 1 but its minimum density is 0. On the other hand, the infinite word  $1(10)^\omega$  has both slope and minimum density  $1/2$ .

## 4.2 The Bubble-Flip algorithm

In this section we present our new generation algorithm for all prefix normal words of a given length. We show that the words are generated in lexicographic order. We also show how our procedure can be easily adapted to generate all prefix normal words of a given length with the same critical prefix.

Comparing our new algorithm to the algorithm of [24], both algorithms generate prefix normal words recursively, but they differ in fundamental ways. The algorithm of [24] is an application of a general schema for generating bubble languages, using a language-specific oracle. It generates separately the sets of prefix normal words with fixed weight  $d$ , i.e. all prefix normal words of length  $n$  containing  $d$  1s. The computation tree is not binary, since each word  $w$  can have up to  $t$  children, where  $t$  is the number of 0s in the first run of 0s of  $w$ . The algorithm uses an additional linear size data structure which it inherits from the parent node and modifies for the current node. A basic feature of the computation tree is that all words have the same fixed suffix, in other words, for the subtree rooted in the word  $w = 1^s 0^t \gamma$ , all nodes are of the form  $v\gamma$ , for some  $v$ .

In contrast, our new algorithm generates all prefix normal words of length  $n$  (except for  $0^n$  and  $10^{n-1}$ ) in one single recursive call, starting from  $110^{n-2}$ . The computation tree is binary, since each word can have at most two children, namely the one produced by the operation *bubble*, and the one by *flip*. Finally, for certain words  $w$ , the words in the subtree rooted in  $w$  have the same critical prefix as  $w$ . This last property allows us to explore the sets of prefix normal words with fixed critical prefix.

### 4.2.1 The algorithm

Let  $w \in \{0, 1\}^n$ . We let  $\text{RightmostOne}(w)$  be the largest index  $r$  such that  $w_r = 1$ , if it exists, and  $\infty$  otherwise. We will use the following operations on prefix normal words:

**Definition 4.6 (Operation flip).** Given  $w \in \{0, 1\}^n$ , and  $1 \leq j \leq n$ , we define  $\text{flip}(w, j)$  to be the binary word obtained by changing the  $j$ -th character in  $w$ , i.e.,  $\text{flip}(w, j) = w_1 w_2 \cdots w_{j-1} \bar{w}_j w_{j+1} \cdots w_n$ , where  $\bar{x}$  is  $1 - x$ .

**Definition 4.7 (Operation bubble).** Given  $w \in \{0, 1\}^n \setminus \{0^n\}$  and  $r = \text{RightmostOne}(w) < n$ , we define  $\text{bubble}(w) = w_1 w_2 \cdots w_{r-1} 0 10^{n-r-1}$ , i.e., the word obtained from  $w$  by shifting the rightmost 1 one position to the right.

We start by giving a simple characterization of those flip-operations which preserve prefix normality.



**Lemma 4.1.** *Let  $w \in \mathcal{L}_n$  such that  $r = \text{RightmostOne}(w) < n$  and let  $j$  be an index with  $r < j \leq n$ . Then  $w' = \text{flip}(w, j)$  is not prefix normal if and only if there exists a  $1 \leq k < r$  such that  $|w_{r-k+1} \cdots w_r|_1 = P_w(k)$  and  $|w_{k+1} \cdots w_{k+j-r}|_1 = 0$ .*

*Proof.* If there exists a  $1 \leq k < r$  such that  $|w_{r-k+1} \cdots w_r|_1 = P_w(k)$  and  $|w_{k+1} \cdots w_{k+j-r}|_1 = 0$ , then for the factor  $u = w'_{r-k+1} \cdots w'_j$  of  $w'$ , we have  $|u| = k + (j - r)$  and  $|u|_1 = P_w(k) + 1 > P_w(k + (j - r)) = P_w(|u|)$ , thus  $w'$  is not prefix normal.

Conversely, note that  $w' \in \mathcal{L}$  if and only if  $v = \text{pref}_j(w') \in \mathcal{L}$ , by Fact 4.1 (ii) and (iii). If  $v \notin \mathcal{L}$ , then, by Fact 4.1 (iv), there exists a suffix  $u$  of  $w_1 \cdots w_{j-1}$  such that  $|u|_1 \geq P_w(|u| + 1)$ . Clearly,  $u$  cannot be shorter than  $j - r - 1$ , since then  $|u|_1 = 0 < P_w(|u| + 1)$ , since  $w$  is prefix normal and contains at least one 1. So  $u$  spans the position  $r$  of the last one of  $w$ . Let us write  $u = u'0^{j-r-1}$ , with  $k := |u'|$ . So we have  $P_w(k) \geq |u'|_1 = |u|_1 \geq P_w(|u| + 1)$ , implying  $|u'|_1 = |w_{r-k+1} \cdots w_r|_1 = P_w(k)$  by monotonicity of  $P$ . Moreover, again by the monotonicity of  $P$ , we get  $P_w(k) = P_w(|u| + 1)$ , which implies that the factor  $w_{k+1} \cdots w_{k+j-r}$  consists of only 0s.  $\square$

---

Algorithm I: COMPUTE  $\varphi$

---

**input** : A prefix normal word  $w$ .

**output** : The leftmost index  $j$ , after the rightmost 1 of  $w$ , such that  $\text{flip}(w, j)$  is prefix normal.

```

1   $r \leftarrow \text{RightmostOne}(w)$ ,  $f \leftarrow 0$ ,  $g \leftarrow 0$ ,  $i \leftarrow 1$ ,  $max \leftarrow 0$ 
2  while  $i < r$  do
3     $f \leftarrow f + w_i$ ,  $g \leftarrow g + w_{r-i+1}$ 
4    if  $f = g$  then
5       $l \leftarrow 0$ ,  $i \leftarrow i + 1$ 
6      while  $i < r$  and  $w_i = 0$  do
7         $l \leftarrow l + 1$ ,  $i \leftarrow i + 1$ 
8      if  $l > max$  then
9         $max \leftarrow l$ 
10   else
11      $i \leftarrow i + 1$ 
12 return  $\min\{r + max + 1, n + 1\}$ 

```

---

**Definition 4.8 (Phi).** *Let  $w \in \mathcal{L}_n \setminus \{0^n\}$ . Let  $r = \text{RightmostOne}(w)$ . Define  $\varphi(w)$  as the minimum  $j$  such that  $r < j \leq n$  and  $\text{flip}(w, j)$  is prefix normal, and  $\varphi(w) = n + 1$  if no such  $j$  exists.*

*Example 4.3.* For the word  $w = 1101001001011000$ , we have  $\varphi(w) = 16$ , since the words  $\text{flip}(w, 14)$  and  $\text{flip}(w, 15)$  both violate the prefix normal condition, for the prefixes of length 3 and 6, respectively.

**Lemma 4.2.** *Let  $w \in \mathcal{L}_n \setminus \{0^n\}$  and let  $r = \text{RightmostOne}(w)$ . Let  $m$  be the maximum length of a run of zeros following a prefix of  $w_1 \cdots w_{r-1}$  which has the same number of 1s as the suffix of  $w_1 \cdots w_r$  of the same length. Formally,*

$$m = \max_{1 \leq \ell < r} \{ \ell \mid \exists k \text{ s.t. } |w_{r-k+1} \cdots w_r|_1 = P_w(k) \text{ and } |w_{k+1} \cdots w_{k+\ell}|_1 = 0 \},$$

where we set the maximum of the empty set to 0. Then,  $\varphi(w) = \min(r + m + 1, n + 1)$ .

*Proof.* We first show that  $\varphi(w) \leq r + m + 1$ . We can assume that  $m < n - r$ , for otherwise the desired inequality holds by definition. Let  $m' = m + 1$ . Then, there are no  $j, k \in \{1, \dots, r - 1\}$  such that  $j - k = m'$ ,  $|w_1 \cdots w_k|_1 = |w_{r-k+1} \cdots w_r|_1$  and  $|w_{k+1} \cdots w_j|_1 = 0$ . Thus, by Lemma 4.1, we have that  $\text{flip}(w, r + m') \in \mathcal{L}$ , hence  $\varphi(w) \leq r + m' = r + m + 1$ .

Let now  $j, k$  be indices attaining the maximum in the definition of  $m$ , i.e.,  $1 < k < j < r$ ,  $j - k = m$ ,  $|w_1 \cdots w_k|_1 = |w_{r-k+1} \cdots w_r|_1$  and  $|w_{k+1} \cdots w_j|_1 = 0$ . Let  $0 < m' \leq m$  then for  $j' = k + m'$  we have  $|w_1 \cdots w_{j'}|_1 = |w_{r-k+1} \cdots w_r|_1$  and  $|w_{k+1} \cdots w_{j'}|_1 = 0$ . Then, by Lemma 4.1,  $\text{flip}(w, r + m') \notin \mathcal{L}$ . Hence  $\varphi(w) > r + m'$ , for  $m' \leq m$ , and in particular  $\varphi(w) \geq r + m + 1$ , which completes the proof.  $\square$

Algorithm 1 implements the idea of Lemma 4.2 to compute  $\varphi$ . For a given prefix normal word  $w$ , it finds the position  $r$  of the rightmost 1 in  $w$ . Then, for each length  $i$  such that the number of 1s in  $\text{pref}_i(w)$  (counted by  $f$ ) is the same as the number of 1s in  $w_{r-i+1} \cdots w_r$  (counted by  $g$ ), the algorithm counts the number of 0s in  $w$  following  $\text{pref}_i(w)$  and sets  $m$  to the maximum of the length of such runs of 0's. By Lemma 4.2 and the definition of  $\varphi$  it follows that  $\min\{r + m + 1, n + 1\}$  is equal to  $\varphi$ , as correctly returned by Algorithm 1. It is not hard to see that the algorithm has linear running time since the two while-loops are only executed as long as  $i < r$ , and the variable  $i$  increases at each iteration of either loop. Therefore, the total number of iterations of the two loops together is upper bounded by  $r \leq n$ . Thus, we have proved the following lemma:

**Lemma 4.3.** *For  $w \in \mathcal{L}_n \setminus \{0^n\}$ , Algorithm 1 computes  $\varphi(w)$  in  $O(\text{RightmostOne}(w))$ , hence  $O(n)$  time.*

The next lemma gives the basis of our algorithm: applying either of the two operations  $\text{flip}(w, \varphi(w))$  or  $\text{bubble}(w)$  to a prefix normal word  $w$  results in another prefix normal word.

**Lemma 4.4.** *Let  $w \in \mathcal{L}_n \setminus \{0^n\}$ . Then the following holds:*

- a) for every  $\ell$ , such that  $\varphi(w) \leq \ell \leq n$ ,  $\text{flip}(w, \ell)$  is prefix normal, and
- b) if  $|w|_1 \geq 2$  then  $\text{bubble}(w)$  is prefix normal.

*Proof.* Let  $r = \text{RightmostOne}(w)$ . In order to show a) we can proceed as in the proof of the upper bound in Lemma 4.2. Fix  $\varphi(w) \leq \ell \leq n$ , and let  $m' = \ell - r$ . Then, by Lemma 4.2, there exist no  $1 < j < k < r$  such the  $k - j = m'$  and  $|w_1 \cdots w_k|_1 = |w_{r-k+1} \cdots w_r|_1$  and  $|w_{k+1} \cdots w_j|_1 = 0$ . This, by Lemma 4.1, implies that  $\text{flip}(w, \ell) \in \mathcal{L}$ .

For b), let  $r' = \max\{i < r \mid w_i = 1\}$ , i.e.,  $r'$  is the position of the penultimate 1 of  $w$ . Let  $w' = w_1 \cdots w_{r'} 0^{n-r'}$ . By Fact 4.1 we have that  $w' \in \mathcal{L}$ . Moreover,  $r \geq \varphi(w')$ , since  $\text{flip}(w', r) = w \in \mathcal{L}$ . Therefore, by a) we have that  $\text{bubble}(w) = \text{flip}(w', r + 1) \in \mathcal{L}$ .  $\square$

**Definition 4.9** ( $\mathcal{PN}$ ). Given  $w \in \mathcal{L}_n \setminus \{0^n\}$  with  $r = \text{RightmostOne}(w)$ , we define  $\mathcal{PN}(w)$  as the set of all prefix normal words  $v$  of length  $n$  such that  $v = w_1 \cdots w_{r-1} \gamma$  for some  $\gamma$  with  $|\gamma|_1 > 0$ . Formally,

$$\mathcal{PN}(w) = \{v \in \mathcal{L}_n \mid v = w_1 \cdots w_{r-1} \gamma, |\gamma|_1 > 0\}.$$

We will use the convention that  $\mathcal{PN}(\text{flip}(w, \varphi(w))) = \emptyset$  if  $\varphi(w) > n$ , furthermore  $\mathcal{PN}(\text{bubble}(w)) = \emptyset$  if  $\text{RightmostOne}(w) = n$ , since then  $\text{flip}(w, \varphi(w))$  resp.  $\text{bubble}(w)$  are undefined.

**Lemma 4.5.** Given  $w \in \mathcal{L}_n \setminus \{0^n, 10^{n-1}\}$ , we have

$$\mathcal{PN}(w) = \{w\} \cup \mathcal{PN}(\text{flip}(w, \varphi(w))) \cup \mathcal{PN}(\text{bubble}(w)).$$

Moreover, these three sets are pairwise disjoint.

*Proof.* It is easy to see that the sets  $\{w\}$ ,  $\mathcal{PN}(\text{bubble}(w))$ ,  $\mathcal{PN}(\text{flip}(w, \varphi(w)))$  are pairwise disjoint.

The inclusion  $\mathcal{PN}(w) \supseteq \{w\} \cup \mathcal{PN}(\text{flip}(w, \varphi(w))) \cup \mathcal{PN}(\text{bubble}(w))$  follows from the definition of  $\mathcal{PN}$  (Def. 4.9).

Now let  $x \in \mathcal{PN}(w) \setminus \{w\}$  and  $r = \text{RightmostOne}(w)$ . We argue by cases according to the character  $x_r$ .

*Case 1.*  $x_r = 0$ . Then,  $x = w_1 \cdots w_{r-1} 0 \gamma$  for some  $\gamma \in \{0, 1\}^{n-r}$  such that  $|\gamma|_1 > 0$ . Since  $\text{bubble}(w) = w_1 \cdots w_{r-1} 0 10^{n-r-1}$ , it follows that  $x \in \mathcal{PN}(\text{bubble}(w))$ .

*Case 2.*  $x_r = 1$ . Then, since  $x \neq w$ , we also have that  $|x_{r+1} \cdots x_n|_1 > 0$ . Therefore,  $x = w_1 \cdots w_{r-1} 1 \gamma$  for some  $\gamma \in \{0, 1\}^{n-r}$  such that  $|\gamma|_1 > 0$ .

Let  $r' = \min\{i > r \mid x_{r'} = 1\}$ . Since  $x \in \mathcal{L}$ , we have that  $\text{pref}_r(x) 0^{n-r}$ ,  $\text{pref}_{r'}(x) 0^{n-r'} \in \mathcal{L}$ . Moreover,  $\text{pref}_{r'}(x) 0^{n-r'} = \text{flip}(\text{pref}_r(x) 0^{n-r}, r')$ , hence,  $r' \geq \varphi(\text{pref}_r(x) 0^{n-r}) = \varphi(w)$ . Therefore,  $x = w_1 \cdots w_r 0^{\varphi(w)-r-1} \gamma$  for some  $|\gamma|_1 > 1$ . This, by definition, means that  $x \in \mathcal{PN}(\text{flip}(w, \varphi(w)))$ .  $\square$

We are now ready to give an algorithm computing all words in the set  $\mathcal{PN}(w)$  for a prefix normal word  $w$ . The pseudocode is given in Algorithm 2. The procedure generates recursively the set  $\mathcal{PN}(w)$  as the union of  $\mathcal{PN}(\text{flip}(w, \varphi(w)))$  and  $\mathcal{PN}(\text{bubble}(w))$ . The call to subroutine  $\text{Visit}()$  is a placeholder indicating that the algorithm has generated a new word in  $\mathcal{PN}(w)$ , which could be printed, or examined, or processed, as required. By Lemma 4.5 we know that  $\text{Visit}()$  is executed for each word in  $\mathcal{PN}(w)$  exactly once.

In order to ease the running time analysis, we next introduce a tree  $\mathcal{T}(w)$  on  $\mathcal{PN}(w)$ . This tree coincides with the computation tree of  $\text{GENERATE } \mathcal{PN}(w)$ , but it will be useful to argue about it independently of the algorithm.

**Definition 4.10** (Tree on  $\mathcal{PN}(w)$ ). Let  $w \in \mathcal{L}_n \setminus \{0^n, 10^{n-1}\}$ . Then we denote by  $\mathcal{T}(w)$  the rooted binary tree  $\mathcal{T}$  with  $V(\mathcal{T}) = \mathcal{PN}(w)$ , root  $w$ , and for a node  $v$ , (1) the left child of  $v$  is empty if  $v_n = 1$  and it is  $\text{bubble}(v)$  otherwise, and (2) the right child of  $v$  is empty if  $\varphi(v) = n + 1$ , and it is  $\text{flip}(v, \varphi(v))$  otherwise.

The tree  $\mathcal{PN}(w)$  has the following easy-to-see properties.

**Observation 4.1** (Properties of  $\mathcal{T}(w)$ ) There are three types of nodes: the root  $w$ , bubble-nodes (left children), and flip-nodes (right children).

---

**Algorithm 2:** GENERATE  $\mathcal{PN}(w)$ 


---

**input :** A prefix normal word  $w$  such that  $|w|_1 > 1$ .

**output :** The set  $\mathcal{PN}(w)$ .

```

1 if RightmostOne( $w$ )  $\neq n$  then
2   |  $w' = \text{bubble}(w)$ 
3   | GENERATE  $\mathcal{PN}(w')$ 
4  $\text{Visit}()$ 
5  $j = \varphi(w)$ 
6 if  $j \leq n$  then
7   |  $w'' = \text{flip}(w, j)$ 
8   | GENERATE  $\mathcal{PN}(w'')$ 

```

---

1. The leftmost descendant of  $w$  has maximal depth, namely  $n - r$ , where  $r = \text{RightmostOne}(w)$ .
2. If a node  $v$  has a right child, then it also has a left child.
3. If a node  $v$  has no right child, then no descendant of  $v$  has a right child. Thus in this case, the subtree rooted in  $v$  is a path of length  $n - r'$ , consisting only of bubble-nodes, where  $r' = \text{RightmostOne}(v)$ .

The next lemma gives correctness, the generation order, and running time of algorithm GENERATE  $\mathcal{PN}(w)$ .

**Lemma 4.6.** For  $w \in \mathcal{L}_n \setminus \{0^n, 10^{n-1}\}$ , Algorithm 2 generates all prefix normal words in  $\mathcal{PN}(w)$  in lexicographic order in  $O(n)$  time per word.

*Proof.* Algorithm 2 recursively generates first all words in  $\mathcal{PN}(\text{bubble}(w))$ , then the word  $w$ , and finally the words in  $\mathcal{PN}(\text{flip}(w, \varphi(w)))$ . As we saw above (Lemma 4.5), these sets form a partition of  $\mathcal{PN}(w)$ , hence every word  $v \in \mathcal{PN}(w)$  is generated exactly once. Moreover, by definition of  $\mathcal{PN}$ , for every  $u \in \mathcal{PN}(\text{bubble}(w))$  it holds that  $u = w_1 \cdots w_{r-1} 0 \gamma$  with  $|\gamma| = n - r$  and  $|\gamma|_1 > 0$ , thus it follows that  $u <_{\text{lex}} w$ . In addition, for every  $v \in \mathcal{PN}(\text{flip}(w, \varphi(w)))$  it holds that  $v = w_1 \cdots w_{r-1} 1 \beta \gamma$  where  $|\beta| = k = \varphi(w) - r - 1$ ,  $|\beta|_1 = 0$ ,  $|\gamma| = n - r - k$  and  $|\gamma|_1 > 0$ , thus  $w <_{\text{lex}} v$ . Since these relations hold at every level of the recursion, it follows that the words are generated by Algorithm 2 in lexicographic order.

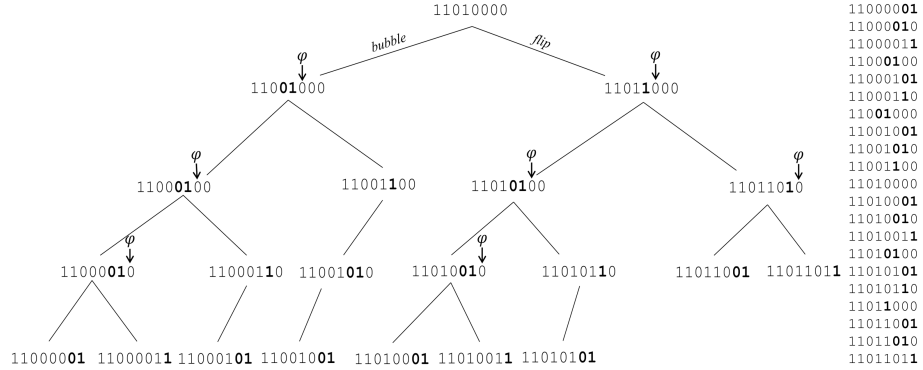
For the running time, note that in each node  $v$ , the algorithm spends  $O(n)$  time on the computation of  $\varphi(v)$  (Lemma 4.3), and if  $v_n \neq 1$ , another  $O(1)$  time on computing  $\text{bubble}(v)$ , and finally, if  $\varphi(v) \leq n$ , further  $O(1)$  time on computing  $\text{flip}(v, \varphi(v))$ . This gives a total running time of  $O(n \cdot |\mathcal{PN}(w)|)$ , so  $O(n)$  amortized time per word. We now show that it actually runs in  $O(n)$  time per word.

Notice that the algorithm performs an in-order traversal of the tree  $\mathcal{T}(w)$ . Given a node  $v$ , the next node visited by the algorithm is given by:

$$\text{next}(v) = \begin{cases} \text{leftmost descendant of right child,} & \text{if } \varphi(v) \leq n, \\ \text{parent}(v), & \text{if } \varphi(v) > n \text{ and } v \text{ is a left child,} \\ \text{parent of first left child on path from } v \text{ to root,} & \text{otherwise.} \end{cases}$$

In all three cases, the algorithm first computes  $\varphi(v)$ , taking  $O(n)$  time by Lemma 4.3. In the first case, it then descends down to the leftmost descendant of the right child,

which takes  $n - \varphi(v)$  bubble operations, in  $O(n)$  time. In the second case, the parent is reached by one operation (moving the last 1 one position to the left if  $v$  is a left child, and flipping the last 1 if  $v$  is a right child), taking  $O(1)$  time. Finally, in the third case, we have up to depth of  $v$  many steps of the latter kind, each taking constant time, so again in total  $O(n)$  time. In all three cases, we get a total of  $O(n)$  time before the next word is visited.  $\square$



**Fig. 4.1:** The words in  $\mathcal{PN}(11010000)$  represented as a tree. If a node of the tree is word  $w$ , then its left child is  $\text{bubble}(w)$  and its right child is  $\text{flip}(w, \varphi(w))$ . In the tree, the position of  $\varphi(w)$  is indicated, whenever  $\varphi(w) \leq n$ ; bubble operations (in the left child) resp. flip operations (in the right child) are highlighted in bold. Algorithm 2 generates these words by performing an in-order traversal of the tree. The corresponding list of words is given on the right.

Now we are ready to present the full algorithm generating all prefix normal words of length  $n$ , see Algorithm 3 (BUBBLE-FLIP). It first visits the two prefix normal words  $0^n$  and  $10^{n-1}$ , and then generates recursively all words in  $\mathcal{L}_n$  containing at least two 1s, from the starting word  $110^{n-2}$ .

---

**Algorithm 3: BUBBLE-FLIP**

---

**input** : An integer  $n$ .  
**output** : All prefix normal words of length  $n$ .

- 1  $w = 0^n$
- 2  $\text{Visit}()$
- 3  $w = 10^{n-1}$
- 4  $\text{Visit}()$
- 5  $w = 110^{n-2}$
- 6 GENERATE  $\mathcal{PN}(w)$

---

**Theorem 4.1.** *The BUBBLE-FLIP algorithm generates all prefix normal words of length  $n$ , in lexicographic order, and in  $O(n)$  time per word.*

*Proof.* Recall that by Fact 4.1(i) every prefix normal word of length  $n$ , other than  $0^n$ , has 1 as its first character. It is easy to see that there is only one prefix normal word of

length  $n$  with a single 1, namely  $10^{n-1}$ . Moreover, by Fact 4.1(i) and the definition of  $\mathcal{PN}$ , the set of all prefix normal words of length  $n$  with at least two 1s coincides with  $\mathcal{PN}(110^{n-2})$ . By Lemma 4.6, this set is generated by  $\text{GENERATE } \mathcal{PN}(110^{n-2})$  in lexicographic order and in  $O(n)$  time per word. Noting that prepending  $0^n$  and  $10^{n-1}$  preserves the lexicographic order concludes the proof.  $\square$

#### 4.2.2 Listing $\mathcal{L}_n$ as a combinatorial Gray code

The algorithm  $\text{GENERATE } \mathcal{PN}(w)$  (Algorithm 2) performs an in-order traversal of the nodes of the tree  $\mathcal{T}(w)$ . If instead we do a post-order traversal, we get a combinatorial Gray code of  $\mathcal{L}_n$ , as we will show next. First note that the change in the traversal order can be achieved by moving line 4 in Algorithm 2 to the end of the code, resulting in Algorithm 4.

---

Algorithm 4:  $\text{GENERATE2 } \mathcal{PN}(w)$

---

**input** : A prefix normal word  $w$  such that  $|w|_1 > 1$ .  
**output** : A combinatorial Gray code on  $\mathcal{PN}(w)$ .

- 1 **if**  $\text{RightmostOne}(w) \neq n$  **then**
- 2      $w' = \text{bubble}(w)$
- 3      $\text{GENERATE2 } \mathcal{PN}(w')$
- 4      $j = \varphi(w)$
- 5     **if**  $j \leq n$  **then**
- 6          $w'' = \text{flip}(w, j)$
- 7          $\text{GENERATE2 } \mathcal{PN}(w'')$
- 8  $\text{Visit}()$

---

**Lemma 4.7.** *In a post-order traversal of  $\mathcal{T}(w)$ , two consecutive words have Hamming distance at most 3.*

*Proof.* Let  $v$  be some node visited during the traversal of  $\mathcal{T}(w)$ . If  $v$  is a flip-node, then the next node in the listing will be its parent node  $v'$ . Since  $v = \text{flip}(v', \varphi(v'))$ ,  $v'$  is at Hamming distance 1 from  $v$ . Otherwise  $v$  is a bubble-node, i.e.  $v = u010^k$  and its parent is  $u10^{k+1}$  for some word  $u$  and integer  $k$ . If  $v$  has no right sibling, then the next node visited is its parent, at Hamming distance 2 from  $v$ . Else the next node  $v'$  is the leftmost descendant of  $v$ 's right sibling, i.e.  $v' = u10^k1$ , and the Hamming distance to  $v$  is at most 3.  $\square$

*Example 4.4.* The words in  $\mathcal{PN}(11010000)$  (Fig. 4.1) are listed by Algorithm 4 as follows: 11000001, 11000011, 11000010, 11000101, 11000110, 11000100, 11001001, 11001010, 11001100, 11001000, 11010001, 11010011, 11010010, 11010101, 11010110, 11010100, 11011001, 11011011, 11011010, 11011000, 11010000.

**Theorem 4.2.** *The BUBBLE-FLIP algorithm using a post-order traversal produces a cyclic combinatorial Gray code on  $\mathcal{L}_n$ , generating each word in time  $O(n)$ .*

*Proof.* By Lemma 4.7, GENERATE2  $\mathcal{PN}(110^{n-2})$  produces a combinatorial Gray code. By visiting the two words  $0^n$  and  $10^{n-1}$  first, followed by GENERATE2  $\mathcal{PN}(110^{n-2})$ , we get a combinatorial Gray code on all of  $\mathcal{L}_n$ . The last word in this code is the root  $110^{n-2}$  and  $d_H(110^{n-2}, 0^n) = 2 \leq 3$ , thus this code is also cyclic.

Since only the order of the tree traversal changed w.r.t. the previous algorithm, it follows immediately that the algorithm visits  $\mathcal{L}_n$  in amortized  $O(n)$  time per word, since the overall running time is, as before,  $O(n|\mathcal{L}|)$ .

To see that the time to visit the next word is  $O(n)$ , we distinguish two cases according to the type of node. If  $v$  is a flip-node, then the next node is its parent, taking  $O(1)$  time to reach. If  $v$  is a bubble-node, then we have to check whether it has a right sibling by computing  $\varphi(v')$ , where  $v'$  is the parent of  $v$ , in  $O(n)$  time. If  $\varphi(v') > n$ , then the next node is  $v'$ . If  $\varphi(v') \leq n$ , then we have to reach the leftmost descendant of  $\text{flip}(v', \varphi(v'))$ , passing along the way only bubble-nodes. This takes  $n - \varphi(v')$  time, so altogether  $O(n)$  time for the node  $v$ .  $\square$

### 4.2.3 Prefix normal words with given critical prefix

Recall Definition 4.3. It was conjectured in [24] that the average length of the critical prefix taken over all prefix normal words is  $O(\log n)$ . Using the BUBBLE-FLIP algorithm, we can generate all prefix normal words with a given critical prefix  $u$ , which could prove useful in proving or disproving this conjecture. Moreover, if we succeed in counting prefix normal words with critical prefix  $u = 1^s 0^t$ , then this could lead to an enumeration of  $|\mathcal{L}_n|$ , another open problem on prefix normal words [25].

In the following lemma, we present a characterization of prefix normal words of length  $n$  with the same critical prefix  $1^s 0^t$  in terms of our generation algorithm. For  $s \geq 1, t \geq 0$ , let us denote by  $\text{CritSet}(s, t, n)$  the set of all prefix normal words of length  $n$  and critical prefix  $1^s 0^t$ . Note that there is only one prefix normal word whose critical prefix has  $s = 0$ , namely  $0^n$ .

**Lemma 4.8.** *Fix  $s \geq 1$  and  $t \geq 0$ , and let  $u = 1^s 0^t$ . Then,*

$$\text{CritSet}(s, t, n) = \begin{cases} \{u\} & \text{if } s + t = n, \\ \{v\} \cup \mathcal{PN}(\text{flip}(v, \varphi(v))), & \text{if } s + t < n, \end{cases}$$

where  $v = u10^{n-(s+t+1)}$ .

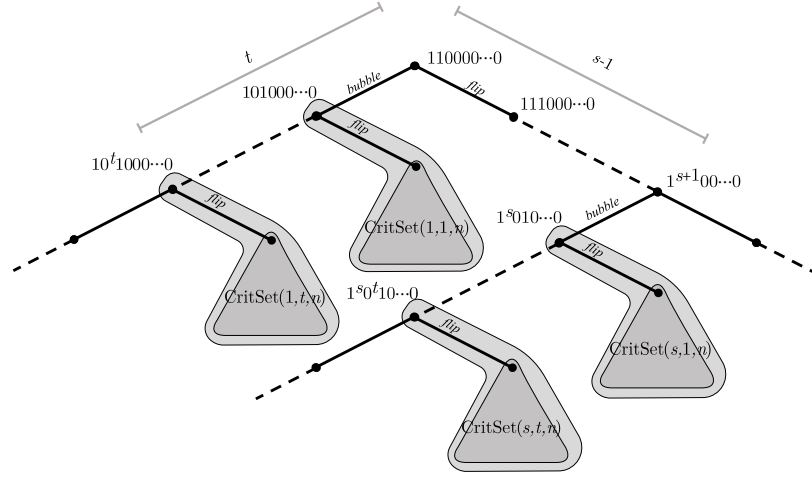
*Proof.* If  $s + t = n$ , then clearly  $\text{CritSet}(s, t, n) = \{u\}$ . Otherwise,

$$\begin{aligned} \text{CritSet}(s, t, n) &= \{u10^{n-(s+t+1)}\} \cup \{u1\gamma \in \mathcal{L}_n \mid |\gamma|_1 > 0\} \\ &= \{v\} \cup \{u1\gamma \in \mathcal{L}_n \mid \gamma_1, \dots, \gamma_{\varphi(v)-(s+t+2)} = 0, |\gamma|_1 > 0\} \\ &= \{v\} \cup \mathcal{PN}(\text{flip}(v, \varphi(v))), \end{aligned}$$

where the first equality holds by definition of critical prefix, the second by definition of  $\varphi(v)$ , and the third by definition of  $\mathcal{PN}$ .  $\square$

In Fig. 4.2, we give a sketch of the placement of some of the sets with same critical prefix within  $\mathcal{T}(110^{n-2})$ , which, as the reader will recall, contains all prefix normal words of length  $n$  except  $0^n$  and  $10^{n-1}$ . The nodes in the tree are labeled with the corresponding generated word, and we have highlighted the subtrees corresponding

to  $\text{CritSet}(1, 1, n)$ ,  $\text{CritSet}(1, t, n)$ ,  $\text{CritSet}(s, 1, n)$  and  $\text{CritSet}(s, t, n)$ . Let us take a closer look at  $\text{CritSet}(s, t, n)$  for  $s, t \geq 2$ . The word  $1^s 0^t 1 0^{n-(s+t+1)}$  is reached starting from the root  $110^{n-2}$ , traversing  $s - 1$  right branches (i.e. flip-branches), passing through the word  $1^s 0 1 0^{n-(s+1)}$ , and then traversing  $t$  left branches (i.e. bubble-branches). The set  $\text{CritSet}(s, t, n)$  is then equal to the word  $1^s 0^t 1 0^{n-(s+t+1)}$  together with its right subtree.



**Fig. 4.2:** A sketch of the computation tree of Algorithm 2 for the set  $w = 110^{n-2}$ , highlighting the subtrees corresponding to sets of prefix normal words with the same critical prefix.

Apart from revealing the recursive structure of sets of prefix normal words with the same critical prefix, the BUBBLE-FLIP algorithm allows us to collect experimental data on the size of  $\text{CritSet}(s, t, n)$  for different values of  $s, t$ , and  $n$ . We give some of these numbers, for  $n = 16, 32$  and small values of  $s$ , see Table 4.2. It was already known [24] that, for  $n \leq 50$ , the average critical prefix length, taken over all  $w \in \mathcal{L}_n$ , is approximately  $\log n$ ; with the new algorithm we are able to generate more precise data. In Fig. 4.3, we plot the relative number of prefix normal words with a given critical prefix length, for lengths  $n = 16$  and  $n = 32$ .

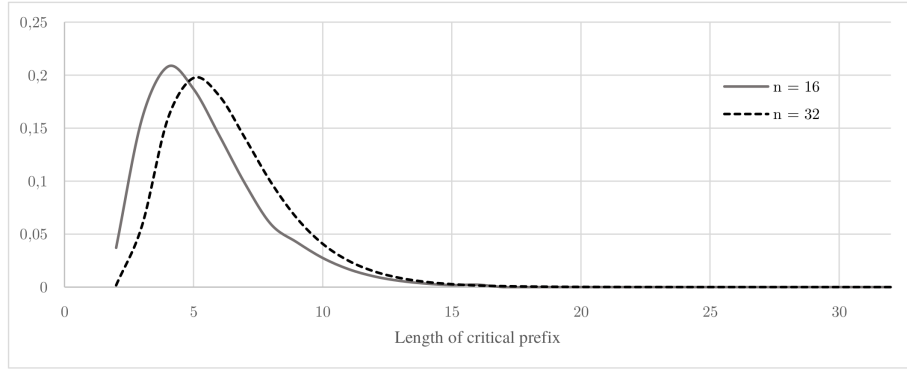
To better understand the average length of the critical prefix of prefix normal words, we provide a closed formula for  $\text{CritSet}(s, t, n)$  having  $s + t < \lceil n/2 \rceil$ . Recalling the definition of critical prefix, a word  $w \in \text{CritSet}(s, t, n)$  has the form  $w = 1^s 0^t 1 \{0, 1\}^\eta$ , where  $\eta = n - s - t - 1$ .

**Definition 4.11.** Given two integers  $i \leq n$ , let  $\mathcal{B}_n^i$  be the set of all binary words of length  $n$  with  $i$  1s. Formally

$$\mathcal{B}_n^i = \{w \mid w \in \{0, 1\}^n \text{ and } |w|_1 = i\} \quad \text{and} \quad |\mathcal{B}_n^i| = \binom{n}{i}$$

**Lemma 4.9.** Given three integers  $n, s$  and  $t$  such that  $s+t < n$ , and  $\eta = n - s - t - 1 < s + t$ . Then, it holds that





**Fig. 4.3:** The frequency of prefix normal words with given critical prefix length, in percentage of the total number of prefix normal words of length  $n$ , for  $n = 16$  (solid) and  $n = 32$  (dashed).

	$t$					
	1	2	3	4	5	6
1	284 663	14 295	2226	597	220	100
2	9 453 217	979 458	162 336	38 404	11 679	4317
3	25 025 726	4 907 605	1 103 214	293 913	91 632	32 459
$s$ 4	27 244 624	7 961 078	2 338 632	732 602	248 717	91 441
5	20 423 789	7 521 441	2 677 376	964 483	360 542	144 460
6	12 789 981	5 378 726	2 178 190	874 907	358 717	151 429
7	7 270 699	3 301 575	1 454 694	633 310	276 593	121 726

	$t$									
	7	8	9	10	11	12	13	14	15	
1	53	30	16	11	9	7	5	3	1	
2	1788	813	451	276	161	90	47	16	15	
3	12 606	5815	2962	1475	723	346	121	106	92	
$s$ 4	37 967	16 994	7693	3507	1594	576	470	378	299	
5	61 139	26 459	11 658	5169	1941	1471	1093	794	562	
6	65 165	28 543	12 605	4944	3473	2380	1586	1024	638	
7	54 118	24 188	9949	6476	4096	2510	1486	848	466	

	$t$																
	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
2	14	13	12	11	10	9	8	7	6	5	4	3	2	1	1	0	0
3	79	67	56	46	37	29	22	16	11	7	4	2	1	1	0	0	0
$s$ 4	232	176	130	93	64	42	26	15	8	4	2	1	1	0	0	0	0
5	386	256	163	99	57	31	16	8	4	2	1	1	0	0	0	0	0
6	382	219	120	63	32	16	8	4	2	1	1	0	0	0	0	0	0
7	247	127	64	32	16	8	4	2	1	1	0	0	0	0	0	0	0

**Table 4.2:** The size of  $\text{CritSet}(s, t, n)$  for  $n = 32$ ,  $s = 1, \dots, 7$  and  $t = 1, \dots, 32$

$$\text{CritSet}(s, t, n) = \left\{ w \mid w = 1^s 0^t 1\gamma \text{ where } \gamma \in \cup_{j=0}^{\min\{\eta, s-1\}} \mathcal{B}_\eta^j \right\}$$

$$\text{thus } |\text{CritSet}(s, t, n)| = \sum_{j=0}^{\min\{\eta, s-1\}} \binom{\eta}{j}.$$

*Proof.* Let  $w = 1^s 0^t 1\gamma$ , where  $\gamma \in \cup_{j=0}^{\min\{\eta, s-1\}} \mathcal{B}_\eta^j$ . The following case based analysis shows that  $w$  is prefix normal.

Let  $u = w_{j+1} \dots w_{j+k}$  be a substring of  $w$ , where  $k$  is the length of  $w$ . If  $j \geq s$  then

$$|u|_1 \leq \min\{|u|, |\gamma|_1 + 1\} \leq \min\{|u|, s\} \leq P_w(|u|).$$

If  $j < s$  and  $k = |u| > j$  then

$$P_w(|u|) = |w_{j+1} \dots w_{j+k-j}|_1 + j \geq |w_{j+1} \dots w_{j+k-j}|_1 + |w_{j+k-j+1} \dots w_{j+k}|_1 = |u|_1.$$

If  $j < s$  and  $k = |u| \leq j$  then, since the first  $j \leq s$  characters of  $w$  are 1's we have

$$P_w(|u|) = |u| \geq |u|_1.$$

We are now going to show that no other words of the form  $w = 1^s 0^t 1\gamma$  is prefix normal. Let  $\gamma \in \cup_{j=s}^{\eta} \mathcal{B}_\eta^j$ . We have that  $|1\gamma| = \eta + 1 \leq s + t$  thus  $|1\gamma|_1 > P_w(\eta + 1)$ . Hence  $w$  is not prefix normal, concluding the proof.

**Definition 4.12.** Let  $\mathcal{C}_n(k)$  be the set of all prefix normal words of length  $n$  with critical prefix length  $k = s + t$ . Namely we have that

$$\mathcal{C}_n(k) = \bigcup_{s=1}^{k-1} \text{CritSet}(s, k-s, n)$$

**Theorem 4.3.** Given  $n \geq 4$  for every  $k \geq \lceil \frac{n}{2} \rceil$  it holds that

$$|\mathcal{C}_n(k)| = 2^{n-k-2} (3k - n - 1)$$

*Proof.* By definition we have that  $|\mathcal{C}_n(k)| = \sum_{s=1}^{k-1} |\text{CritSet}(s, k-s, n)|$ . Let  $\eta = n - k - 1$ , then we can split the sum into two parts, one from 1 up to  $\eta$  and the second one from  $\eta + 1$  up to  $k - 1$  as follow

$$|\mathcal{C}_n(k)| = \sum_{s=1}^{\eta} |\text{CritSet}(s, k-s, n)| + \sum_{s=\eta+1}^{k-1} |\text{CritSet}(s, k-s, n)|$$

By Lemma 4.9 we can rewrite it as follow

$$|\mathcal{C}_n(k)| = \sum_{s=1}^{\eta} \sum_{j=0}^{s-1} \binom{\eta}{j} + \sum_{s=\eta+1}^{k-1} 2^\eta$$

The former sum can be rewritten as follow

$$\begin{aligned}
\sum_{s=1}^{\eta} \sum_{j=0}^{s-1} \binom{\eta}{j} &= \underbrace{\binom{\eta}{0}}_{s=1} + \underbrace{\binom{\eta}{0} + \binom{\eta}{1}}_{s=2} + \cdots + \underbrace{\binom{\eta}{0} + \binom{\eta}{1} + \cdots + \binom{\eta}{\eta-1}}_{s=\eta} \\
&= \eta \binom{\eta}{0} + (\eta-1) \binom{\eta}{1} + \cdots + (\eta-i) \binom{\eta}{i} + \cdots + \binom{\eta}{\eta-1} \\
&= \sum_{i=0}^{\eta-1} (\eta-i) \binom{\eta}{i} = \sum_{i=0}^{\eta-1} \frac{(\eta-i)\eta!}{(\eta-i)!i!} = \sum_{i=0}^{\eta-1} \frac{(\eta-i)\eta(\eta-1)!}{(\eta-i)(\eta-i-1)!i!} \\
&= \sum_{i=0}^{\eta-1} \frac{\eta(\eta-1)!}{(\eta-i-1)!i!} = \sum_{i=0}^{\eta-1} \eta \binom{\eta-1}{i} \\
&= 2^{\eta-1} \eta
\end{aligned}$$

The latter sum can be rewritten as follow

$$\sum_{s=\eta+1}^{k-1} 2^{\eta} = 2^{\eta}(k-1-\eta+1-1) = 2^{\eta}(2k-n)$$

Keeping all together we have that

$$|\mathcal{C}_n(k)| = 2^{\eta-1}\eta + 2^{\eta}(2k-n) = 2^{n-k-2}(3k-n-1)$$

□

#### 4.2.4 Practical improvements of the algorithm

The running time of the algorithm is dominated by the time spent at each node for computing the value of  $\varphi$ , which, in general, takes time linear in  $n$ , the length of the words. Therefore the overall generation of  $\mathcal{L}_n$  takes  $O(n|\mathcal{L}_n|)$  time. One way of improving the running time of the overall generation would be to achieve faster amortized computation of  $\varphi$  by exploiting the relationship between  $\varphi(w)$  and  $\varphi(w')$  for words  $w$  and  $w'$  generated at close nodes of the recursion tree. Next we present two attempts in this direction. We show two cases where the  $\varphi(w)$  can be computed in sublinear time. This implies a faster generation algorithm, absolutely, however, since the number of nodes falling in such cases is only  $o(|\mathcal{L}_n|)$  we do not achieve any significant asymptotic improvement on the overall generation.

The first practical improvement can be obtained from the following lemma. It shows that given a node  $w$  of the generation tree, for all nodes  $w'$  in the subtree rooted in  $w$ , which are reachable from  $w$  by traversing only flip-branches, the value  $\varphi(w')$  can be computed in time  $O(\text{RightmostOne}(w))$ . Note that on such a *rightward-path* words have a strictly increasing number of 1s. Therefore, the result of the lemma provides a strict improvement on the original estimate that for each word  $w'$  in such rightward-path the computation of  $\varphi(w')$  requires  $\Theta(\text{RightmostOne}(w'))$ . This gives an improvement for nodes along the right branches of the tree only; the improvement gets better as we move further down a right path.

**Lemma 4.10.** *Let  $w \in \mathcal{L}_n$  and let*

$$v^{(j)} = \begin{cases} w & j = 0 \\ \text{flip}(v^{(j-1)}, \varphi(v^{(j-1)})) & j > 0 \end{cases}$$

*i.e.,  $v^{(j)}$  is the word produced by applying  $j$  times the flip operation starting from  $w$ . For each  $j \geq 0$  and  $k \geq 1$ , we have that  $v = \text{flip}(v^{(j)}, \text{RightmostOne}(v^{(j)} + k))$  is in  $\mathcal{L}_n$  if and only if for all  $t = 1, \dots, \text{RightmostOne}(w)$  it holds that  $|v_{\text{RightmostOne}(v^{(j)}+k-t+1} \dots v_{\text{RightmostOne}(v^{(j)}+k)}|_1 \leq |w_1 \dots w_t|_1$ , i.e., the suffix of  $v_1 \dots v_{\text{RightmostOne}(v^{(j)}+k)}$  of length  $t$  satisfies the prefix normal condition.*

*Proof.* Assume otherwise and let  $j$  and  $k$  be the smallest integers such that  $v = \text{flip}(v^{(j)}, \text{RightmostOne}(v^{(j)} + k))$  is a counterexample—we first choose the smallest  $j$  such that there is a  $k$  and then among all such  $k$ 's we choose the smallest, given the choice of  $j$ .

Let  $n_0 = \text{RightmostOne}(v^{(j)} + k)$ . We write  $P(i)$  for  $P_v(i)$ , and denote by  $S(i)$  the number of 1s in the  $i$ -length suffix of  $v_1 \dots v_{n_0}$ . Let  $r = \text{RightmostOne}(w)$ . By assumption,  $S(t) \leq P(t)$  for all  $t \leq r$ , but there is an  $m > r$  such that  $S(m) > P(m)$ . Choose this  $m$  minimal. By definition, using the properties of the  $\varphi$  function, we have that  $v^{(j)} \in \mathcal{L}_n$ . Moreover, by the minimality of the choice of  $j$  and  $k$ , it holds that  $v_{n_0-m+1} \dots v_{n_0-1}$  satisfies the prefix normal condition, i.e.  $|v_{n_0-m+1} \dots v_{n_0-1}|_1 \leq P(m-1)$ . Therefore, it must hold that  $P(m-1) = P(m)$ , hence  $v_m = 0$ . Since  $m > r$  and  $v_m = 0$ , there must be  $0 \leq j' \leq j$  such that  $\varphi(v^{(j')}) < n_0 - m < \varphi(v^{(j'+1)})$ , i.e., the flip operation that produces  $v^{(j'+1)}$  has to be done on a position following  $n_0 - m$ . This means that for some  $t' < m$ ,  $|v_{m-t'+1} \dots v_m|_1 = P(t')$ , otherwise we would have  $v_m = 1$ . Let  $m' = m - t'$ . Thus we have  $P(m) = P(m') + P(t')$ . On the other hand,  $S(m) = S(m') + |v_{n_0-m+1} \dots v_{n_0-m+t'}|_1 \leq P(m') + P(t') = P(m)$ , where the inequality holds by the minimality of  $m$  and of  $n_0$ , respectively. But this is a contradiction to our assumption that  $S(m) > P(m)$ .  $\square$

Second, we show how to derive  $\varphi(v')$  for a bubble-node  $v'$  from  $\varphi(v)$ , where  $v$  is the parent of  $v'$ . This gives an improvement (from linear to constant) for all nodes of the form  $\text{bubble}^*(v)$  of some node  $v$ , spreading out the cost of computing  $\varphi(v)$  for  $v$  over all bubble-descendants of  $v$ . Note that this covers the case of Observation 4.1, part 3, which tells us that we can skip the computation of  $\varphi(v)$  if the parent of  $v$  does not have a flip-child.

**Lemma 4.11.** *Let  $w$  be a prefix normal word  $w$  of length  $n$  with  $|w|_1 \geq 2$  and  $r = \text{RightmostOne}(w) \neq n$ . Then*

$$\varphi(\text{bubble}(w)) = \begin{cases} \min\{n+1, \varphi(w) + 2\} & \text{if } |w|_1 = 2, \\ \varphi(w) & \text{if } |w_1 \dots w_{\varphi(w)-r}|_1 > 1, \\ \min\{n+1, \varphi(w) + 1\} & \text{otherwise.} \end{cases} \quad (4.1)$$

*In particular,  $\varphi(\text{bubble}(w))$  can be computed in constant time, given  $\varphi(w)$ .*

*Proof.* An immediate observation is that  $\varphi(w) \leq \varphi(\text{bubble}(w))$ . Therefore, if  $\varphi(w) = n + 1$  the claim holds trivially.

*Case 1.*  $|w|_1 = 2$ . Then we can write  $w$  as  $w = 10^{r-2}10^{n-r}$  and  $\text{bubble}(w) = 10^{r-1}10^{n-r-1}$ . It is then easy to see that we have  $\varphi(w) = \varphi(10^{r-2}10^{n-r}) = \min\{n +$

$1, r + t + 1\}$  and  $\varphi(\text{bubble}(w)) = \varphi(10^{r-1}10^{n-r-1}) = \min\{n + 1, (r + 1) + (t + 1) + 1\} = \min\{n + 1, \varphi(w) + 2\}$ , as desired. Since  $w \in \mathcal{L}_n$ , we have  $w_1 = 1$ .

*Case 2.*  $|w_1 \dots w_{\varphi(w)-r}|_1 > 1$ . First of all, let us observe that we have  $|w|_1 > 2$ . For otherwise, if  $|w|_1 = 2$ , the analysis of the previous case implies that  $\varphi(w) - r = r - 1$  hence  $|w_1 \dots w_{\varphi(w)-r}|_1 = 1$ , contradicting the standing hypothesis. From  $|w_1 \dots w_{\varphi(w)-r}|_1 > 1$ , it follows that  $\varphi(w) > r + 1$ . Moreover, we have  $\varphi(w) < 2r$ , since  $w_1 \dots w_{r-1}10^{r-2}1 \in \mathcal{L}_n$  (by Fact 4.1 (iv)).

Now let  $w' = \text{flip}(w, \varphi(w))$  and  $w'' = \text{flip}(\text{bubble}(w), \varphi(w))$ , i.e.,

$$\begin{aligned} w' &= w_1 \dots w_{r-1}10^{\varphi(w)-r-1}10^{n-\varphi(w)}, \\ w'' &= w_1 \dots w_{r-1}010^{\varphi(w)-r-2}10^{n-\varphi(w)}. \end{aligned}$$

By the definition of  $\varphi$  we have  $w' \in \mathcal{L}_n$ . Moreover, it holds that  $\text{bubble}(w) = w_1 \dots w_{r-1}010^{n-r-1} \in \mathcal{L}_n$ . For proving the claim, it is enough to show that  $w'' \in \mathcal{L}_n$ .

Let  $S_{w''}(i) = |w''_{\varphi(w)-i+1} \dots w''_{\varphi(w)}|_1$  and  $S_{w'}(i) = |w'_{\varphi(w)-i+1} \dots w'_{\varphi(w)}|_1$ . It is not hard to see that for each  $i \notin \{r, \varphi(w) - r\}$ , it holds that  $S_{w''}(i) = S_{w'}(i) \leq P_{w'}(i) = P_{w''}(i)$ , where the inequality follows from the prefix normality of  $w'$ . Moreover, for  $i = \varphi(w) - r$ , we have  $S_{w''}(\varphi(w) - r) = 2$  and since  $\varphi(w) - r < r$ , we also have  $P_{w''}(\varphi(w) - r) = P_{w'}(\varphi(w) - r) = P_w(\varphi(w) - r) > 1$  (by the standing hypothesis). Finally, for  $i = r$ , using again  $\varphi(w) - r < r$ , it follows that  $S_{w''}(r) = S_{w'}(r) \leq P_{w'}(r) = P_{w''}(r)$ . In conclusion, we have  $S_{w''}(i) \leq P_{w'}(i)$  for each  $i = 1, \dots, \varphi(w)$ , hence, by Fact 4.1 (iv), the word  $w_1 \dots w_{r-1}010^{\varphi(w)-r-2}1 \in \mathcal{L}$  and by Fact 4.1 (iii),  $w'' \in \mathcal{L}_n$ , which concludes the proof of this case.

*Case 3.*  $|w_1 \dots w_{\varphi(w)-r}|_1 = 1$  and  $|w|_1 > 2$ . Proceeding as in the previous case, we have that  $S_{w'}(\varphi(w) - r) = 2 > P_w(\varphi(w) - r) = P_{w'}(\varphi(w) - r)$ , which implies that  $w' \notin \mathcal{L}_n$ , hence  $\varphi(\text{bubble}(w)) \geq \varphi(w) + 1$ . Let

$$w''' = w_1 \dots w_{r-1}010^{\varphi(w)-r-1}10^{n-\varphi(w)-1} = \text{flip}(\text{bubble}(w), \varphi(w) + 1).$$

It is enough to show that  $w''' \in \mathcal{L}_n$ . Let us redefine  $S_{w'''}(i) = |w'''_{\varphi(w)-i+2} \dots w'''_{\varphi(w)+1}|_1$  and  $S_{w'}(i) = |w'_{\varphi(w)-i+1} \dots w'_{\varphi(w)}|_1$ . It is not hard to see that for each  $i \in \{1, \dots, \varphi(w)\}$ , it holds that  $S_{w'''}(i) \leq S_{w'}(i)$ . Moreover, for each  $i \in \{1, \dots, \varphi(w) - 1\} \setminus \{r\}$ , we have  $P_{w'}(i) = P_{w'''}(i)$ . Thus, for each  $i \in \{1, \dots, \varphi(w) - 1\} \setminus \{r\}$ , it holds that  $S_{w'''}(i) \leq S_{w'}(i) \leq P_{w'}(i) = P_{w'''}(i)$ , where the second inequality follows from the prefix normality of  $w'$ .

For  $i = \varphi(w)$ , using  $w'''_1 = 1 = w'''_{\varphi(w)+1}$  we have  $S_{w'''}(\varphi(w)) = |w|_1 = P_{w'''}(\varphi(w))$ .

For  $i = r$ , we have  $\varphi(w) + 2 - r \leq r + 1$ . If  $\varphi(w) + 2 - r = r + 1$ , i.e.,  $\varphi(w) = 2r - 1$  then  $S_{w'''}(r) = 2 = |w'_r \dots w'_{\varphi(w)}|_1 \leq P_{w'}(r)$ . Since  $P_{w'}(r) = |w|_1 \geq 3$ , we have  $P_{w'''}(r) = P_{w'}(r) - 1 \geq 2 = S_{w'''}(r)$ .

If  $\varphi(w) + 2 - r \leq r$ , then

$$\begin{aligned} P_{w'''}(r) - S_{w'''}(r) &= |w'''_1 \dots w'''_r|_1 - |w'''_{\varphi(w)+2-r} \dots w'''_{\varphi(w)+1}|_1 \\ &= |w'''_1 \dots w'''_{\varphi(w)+1-r}|_1 - |w'''_{r+1} \dots w'''_{\varphi(w)+1}|_1 \\ &= P_{w'''}(\varphi(w) + 1 - r) - S_{w'''}(\varphi(w) + 1 - r) \geq 0, \end{aligned}$$

where the middle equality follows by removing from the two words the common intersection, and the last inequality comes from the previous subcase, as  $\varphi(w) + 1 - r \in \{1, \dots, \varphi(w) - 1\} \setminus \{r\}$ .

In conclusion, we have  $S_{w'''}(i) \leq P_{w'''}(i)$  for each  $i = 1, \dots, \varphi(w) + 1$ , hence, by Fact 4.1 (iv) the word  $w_1 \dots w_{r-1} 010^{\varphi(w)-r-1} 1 \in \mathcal{L}$  and by Fact 4.1 (iii)  $w''' \in \mathcal{L}_n$ , which concludes the proof of this case. The proof of (4.1) is complete.

We now argue that  $\varphi(\text{bubble}(w))$  can be computed in constant time. Our result says that knowing  $\text{RightmostOne}(w)$  and the position of the second leftmost 1 in  $w$ , then  $\varphi(\text{bubble}(w))$  can be computed applying (4.1), i.e., in  $O(1)$  time. In fact, the condition  $|w_1 \dots w_{\varphi(w)-r}|_1 > 1$  is equivalent to checking that the second leftmost 1 of  $w$  is in a position not larger than  $\varphi(w) - \text{RightmostOne}(w)$ . It is not hard to see that  $\text{RightmostOne}(w)$  and the position of the second leftmost 1 of  $w$  can be computed and maintained for each node on the generation tree without increasing the computation by more than a constant amount of time per node.  $\square$

We provide the following examples to illustrate the two improvements.

*Example 6.* For the first improvement, consider the word  $w = 11001010^{n-7}$  with  $n$  being some large number. Let  $w^{(1)}, w^{(2)}, \dots, w^{(i)}$  be the words generated on the right path rooted at  $w$ , i.e.,  $w^{(1)}$  is the flip-child of  $w$ ,  $w^{(2)}$  is the flip-child of  $w^{(1)}$  and so on.

It is not hard to see that  $w^{(1)} = 1100101010^{n-9}$ ,  $w^{(2)} = 110010101010^{n-11}$ , and in general  $w^{(i)} = 1100101(01)^i 0^{n-7-2i}$  for any  $i = 1, 2, \dots, \frac{n-7}{2}$ .

What Lemma 9 guarantees is that, for  $i = 1, \dots, \frac{n-7}{2}$ ,  $w^{(i)} = \text{flip}(w^{(i-1)}, \varphi(w^{(i-1)}))$  can be computed in time  $\Theta(\text{RightmostOne}(w))$  rather than  $\Theta(\text{RightmostOne}(w^{(i-1)}))$ . Therefore, in total, to generate them all, we need  $\Theta(\text{RightmostOne}(w) \cdot n)$ . Without applying Lemma 9, i.e., computing  $w^{(i)} = \text{flip}(w^{(i-1)}, \varphi(w^{(i-1)}))$  using Algorithm 1, in time  $\text{RightmostOne}(w^{(i-1)}) = 7 + 2(i-1)$ , we would need in total  $\Theta(n^2)$  time.

*Example 7.* For the second improvement, consider the word  $w = 100100000000$ , for which it holds that  $|w|_1 = 2$ . We have that  $\varphi(w) = 7$ , and indeed, the word  $\text{bubble}(w) = 100010000000$  has  $\varphi(\text{bubble}(w)) = 9$ . Consider now the word  $w = 110001010000$ . We have  $\text{RightmostOne}(w) = 8$ ,  $\varphi(w) = 11$ , and  $|w_1 \dots w_{11-8}|_1 = 2 > 1$ . It is not hard to verify that for  $\text{bubble}(w) = 110001001000$ , we have  $\varphi(\text{bubble}(w)) = 11$ . Finally, consider the word  $w = 101001001000$ . We have  $\varphi(w) = 11$ , and since  $|10|_1 \leq 1$ , it holds that  $\text{bubble}(w) = 101001000100$  and  $\varphi(\text{bubble}(w)) = 12$ .

### 4.3 On infinite extensions of prefix normal words

In this section, we study how to construct infinite prefix normal words. We focus on infinite extensions of finite prefix normal words which satisfy the prefix normal condition at every finite point. We provide two operations that extend a starting finite prefix normal word, the *flipext* operation builds prefix normal words which are in a certain sense densest among all possible infinite extensions of the starting word. We show that words in this class are ultimately periodic, and we are able to determine both the size and the density of the period and to upper bound the starting point of the periodic behavior. The second operation is the *lazy- $\alpha$ -flipext* operation that, conversely, builds prefix normal words which are the lowest dense among all possible infinite extensions of the starting word, having minimum density  $\alpha$ .

We now define an operation on finite prefix normal words which is similar to the flip operation from Sec. 4.2: it takes a prefix normal word  $w$  ending in a 1 and *extends*

it by a run of 0s followed by a new 1, in such a way that this 1 is placed in the first possible position without violating prefix normality.

**Definition 4.13 (Operation flipext).** Let  $w \in \mathcal{L} \cap \{0, 1\}^*1$ . Define  $\text{flipext}(w)$  as the finite word  $w0^k1$ , where  $k = \min\{j \mid w0^j1 \in \mathcal{L}\}$ . We further define the infinite word  $v = \text{flipext}^\omega(w) = \lim_{i \rightarrow \infty} \text{flipext}^{(i)}(w)$ .

For a prefix normal word  $w$ , the word  $w0^{|w|}1$  is always prefix normal, so the operation  $\text{flipext}$  is well-defined. Let  $w \in \mathcal{L}$  and  $r = \text{RightmostOne}(w) < |w|$ . Then  $\text{flipext}(\text{pref}_r(w))$  is a prefix of  $\text{flip}(w, \varphi(w))$  if and only if  $\varphi(w) \leq |w|$ , in particular,  $\text{flip}(w, \varphi(w)) = \text{flipext}(\text{pref}_r(w)) \cdot 0^{|w| - \varphi(w)}$ .

**Definition 4.14 (Iota-factorization).** Let  $w$  be a finite binary word, or an infinite binary word such that  $\iota = \iota(w)$  exists. The *iota-factorization* of  $w$  is the factorization of  $w$  into  $\iota$ -length factors, i.e. the representation of  $w$  in the form

$$\begin{aligned} w &= u_1 u_2 \cdots u_r v, \\ &\text{where } r = \lfloor |w|/\iota \rfloor, |u_i| = \iota \text{ for } i = 1, \dots, r, \text{ and } |v| < \iota, \quad \text{for } w \text{ finite, and} \\ w &= u_1 u_2 \cdots, \\ &\text{where } |u_i| = \iota \text{ for all } i, \quad \text{for } w \text{ infinite.} \end{aligned}$$

### 4.3.1 Flip extensions and ultimate periodicity

**Lemma 4.12.** Let  $w$  be a finite or infinite prefix normal word, such that  $\iota = \iota(w)$  exists. Let  $w = u_1 u_2 \cdots$  be the *iota-factorization* of  $w$ . Then for all  $i$ ,  $|u_i|_1 = \kappa(w)$ .

*Proof.* Since  $w$  is prefix normal,  $|u_i| \leq \kappa = \kappa(w)$ . On the other hand, assume there is an  $i_0$  for which  $|u_{i_0}|_1 < \kappa$ . Then the prefix  $u_1 u_2 \cdots u_{i_0}$  has fewer than  $i_0 \kappa$  many 1s, and thus density less than  $i_0 \kappa / i_0 \iota = \kappa / \iota = D(\iota)$ , in contradiction to the definition of  $\iota$ .  $\square$

The next lemma states that the *iota-factorization* of a word  $w$  constitutes a non-increasing sequence w.r.t. lexicographic order, as long as  $w$  satisfies a weaker condition than prefix normality, namely that factors of length  $\iota(w)$  obey the prefix normal condition. That this does not imply prefix normality can be seen on the example  $(1110010)^\omega$ , which is not prefix normal.

**Lemma 4.13.** Let  $w$  be a finite or infinite binary word, such that  $\iota = \iota(w)$  exists. Let  $w = u_1 u_2 \cdots$  be the *iota-factorization* of  $w$ . If for every factor  $u$  of length  $\iota$  satisfies the prefix normal condition, then for all  $i$ ,  $u_i \geq_{\text{lex}} u_{i+1}$ .

*Proof.* Let us write  $u_i = u_{i,1} \cdots u_{i,\iota}$ . Let  $a(i, j) = |u_{i,1} \cdots u_{i,j}|_1$  denote the number of 1s in the  $j$ -length prefix of  $u_i$ , and  $b(i, j) = |u_{i,j+1} \cdots u_{i,\iota}|_1$  the number of 1s in the suffix of length  $\iota - j$ . By Lemma 4.12, we have that  $a(i, j) + b(i, j) = \kappa$ . On the other hand,  $b(i, j) + a(i+1, j) \leq \kappa$ , since all  $\iota$ -length factors satisfy the prefix normal condition. Thus, for all  $i$ :  $a(i, j) \geq a(i+1, j)$ .

If  $u_i \neq u_{i+1}$ , let  $h = \min\{j \mid j = 1, \dots, \iota : a(i, j) > a(i+1, j)\}$ . Thus, for every  $j < h$ , we have  $u_{i,j} = u_{i+1,j}$  and  $u_{i,h} = 1, u_{i+1,h} = 0$ , implying  $u_i \geq_{\text{lex}} u_{i+1}$ .  $\square$

**Corollary 4.1.** *Let  $w$  be a finite or infinite prefix normal word, such that  $\iota = \iota(w)$  exists. Then for all  $i$ ,  $u_i \geq_{\text{lex}} u_{i+1}$ , where  $u_i$  is the  $i$ 'th factor in the iota-factorization of  $w$ .*

We now prove that the flipext operation does not change the minimum density. This means that among all infinite prefix normal extensions of a word  $w \in \mathcal{L}$ , the word  $v = \text{flipext}^\omega(w)$  has the highest minimum density.

**Lemma 4.14.** *Let  $w \in \mathcal{L}$  such that  $w_n = 1$ , and let  $v \in \text{flipext}^*(w) \cup \{\text{flipext}^\omega(w)\}$ . Then  $\delta(v) = \delta(w)$ , and as a consequence,  $\iota(v) = \iota(w)$  and  $\kappa(v) = \kappa(w)$ .*

*Proof.* Assume otherwise. Then there exists a minimal index  $i$  such that  $D_v(i) < \delta(w) =: \delta$ . Clearly,  $i > |w|$ , by definition of  $\delta$ . Since  $i$  is minimal, it follows that  $D_v(i-1) \geq \delta$ , which implies  $v_i = 0$ . Since  $i > |w|$ , there was an iteration of flipext, say the  $j$ 'th iteration, which produced the extension containing position  $i$ , i.e.  $|\text{flipext}^{(j-1)}(w)| < i < |\text{flipext}^{(j)}(w)|$ . Since  $v_i = 0$ , this implies that there is an  $m$  such that the factor  $v_{i-m+1} \cdots v_{i-1}1$  would have violated the prefix normal condition, i.e.  $|v_{i-m+1} \cdots v_{i-1}1|_1 > P_v(m)$ . This implies  $|v_{i-m+1} \cdots v_{i-1}0|_1 = P_v(m)$  (because  $v$  is prefix normal). Now consider the prefix  $\text{pref}_i(v) = v_1 \cdots v_i$ , and let us write  $i = i' + m$ . Since  $i$  was chosen minimal, we have that  $D_v(i'), D_v(m) \geq \delta$ . Since  $D_v(i') = \frac{P_v(i')}{i'}$ ,  $D_v(m) = \frac{P_v(m)}{m}$ , this implies

$$D_v(i) = \frac{P_v(i)}{i} = \frac{P_v(i') + P_v(m)}{i' + m} \geq \delta,$$

in contradiction to the assumption.  $\square$

**Theorem 4.4.** *Let  $w \in \mathcal{L}$  and  $v = \text{flipext}^\omega(w)$ . Then  $v$  is ultimately periodic. In particular,  $v$  can be written as  $v = ux^\omega$ , where  $|x| = \iota(w)$  and  $|x|_1 = \kappa(w)$ .*

*Proof.* By Lemma 4.14,  $\iota(v) = \iota(w)$ , and by Lemma 4.12, in the iota-factorization of  $w$ , all factors  $u_i$  have  $\kappa(w)$  1s. Moreover, by Corollary 4.1, the factors  $u_i$  constitute a lexicographically non-increasing sequence. Since all  $u_i$  have the same length  $\iota(w)$ , and there are finitely many binary words of length  $\iota(w)$ , the claim follows.  $\square$

We can further show that the period  $x$  from the previous theorem is prefix normal, as long as it starts in a position which is congruent 1 modulo  $\iota$ , in other words, if it is one of the factors in the iota-factorization of  $v$ .

**Lemma 4.15.** *Let  $w \in \mathcal{L}$  and  $v = \text{flipext}^\omega(w) = ux^\omega$  such that  $x$  is the  $k$ 'th factor in the iota-factorization of  $v$ , for some  $k \geq 1$ . Then  $x$  is prefix normal.*

*Proof.* First note that if  $v = x^\omega$ , then  $x$  is prefix normal by the prefix normality of  $v$ . Else, assume for a contradiction that  $x$  is not prefix normal. Let  $\alpha$  be a factor of  $x$  of minimal length s.t.  $|\alpha|_1 > |\beta|_1$ , where  $\beta$  is the prefix of  $x$  of length  $|\alpha|$ . Then  $\beta$  and  $\alpha$  are disjoint due to the minimality assumption. In other words, there is a (possibly empty) word  $\gamma$  s.t.  $\beta\gamma\alpha$  is a prefix of  $x$ .

Since  $x$  is a  $\iota$ -factor of  $v$ , therefore the prefix of  $v$  before  $x$  has length  $t\iota$  for some  $t \geq 1$ . Let  $x = \beta\gamma\alpha\nu$ , and write  $x'$  for the rotation  $\nu\beta\gamma\alpha$  of  $x$ . Now consider the word  $s = \gamma\alpha(x')^t$ , which has length  $|\gamma| + |\alpha| + t\iota$ . By Theorem 4.4,  $|x|_1 = \kappa$ , and since  $x'$  is a rotation of  $x$ , also  $|x'|_1 = \kappa$ . Therefore, for the factor  $s$  of  $v$  it holds that  $|s|_1 = |\gamma|_1 + |\alpha|_1 + t\kappa > |\gamma|_1 + |\beta|_1 + t\kappa = P_v(|s|)$ , in contradiction to  $v \in \mathcal{L}$ .  $\square$



Next we show that for a word  $v \in \text{flipext}^*(w)$ , in order to check the prefix normality of an extension of  $v$ , it is enough to verify that the suffixes up to length  $|w|$  satisfy the prefix normal condition.

**Lemma 4.16.** *Let  $w$  be prefix normal and  $v' \in \text{flipext}^*(w)$ . Then for all  $k \geq 0$  and  $v = v'0^k1$ ,  $v \in \mathcal{L}$  if and only if for all  $1 \leq j \leq |w|$ , the suffixes of  $v$  of length  $j$  satisfy the prefix normal condition.*

*Proof.* Directly from Lemma 4.10.  $\square$

By Theorem 4.4, we know that  $v = \text{flipext}^\omega(w)$  has the form  $v = ux^\omega$  for some  $x$ , whose length and density we can infer from  $w$ . The next theorem gives an upper bound on the waiting time for  $x$ , both in terms of the length of the non-periodic prefix  $u$ , and in the number of times a factor can occur before we can be sure that we have essentially found the periodic factor  $x$  (up to rotation).

**Theorem 4.5.** *Let  $w \in \mathcal{L}$  and  $v = \text{flipext}^\omega(w)$ . Let us write  $v = ux^\omega$ , with  $|x| = \iota(w)$  and  $x$  not a suffix of  $u$ . Let  $\iota = \iota(w)$ ,  $\kappa = \kappa(w)$ , and  $m = \left\lceil \frac{|w|}{\iota} \right\rceil$ . Then*

1.  $|u| \leq \left(\binom{\iota}{\kappa} - 1\right)m\iota$ , and
2. *if for some  $y \in \{0, 1\}^\iota$ , it holds that  $y^{m+1}$  occurs with starting position  $j > |w|$ , then  $y$  is a rotation of  $x$ .*

*Proof.* 1.: Assuming 2., then every  $\iota$ -length factor  $y$  which is not the final period can occur at most  $m$  times consecutively. By Cor. 4.1, consecutive non-equal factors in the iota-factorization of  $v$  are lexicographically decreasing, so no factor  $y$  can reoccur again once it has been replaced by another factor. By Theorem 4.4, the density of each factor is  $\kappa$ . There are at most  $\binom{\iota}{\kappa}$  such  $y$  which are lexicographically smaller than  $\text{pref}_\iota(w)$ , and each of these has length  $\iota$ .

2.: By Lemma 4.16, in order to produce the next character of  $v$ , the operation  $\text{flipext}$  needs to access only the last  $|w|$  many characters of the current word. After  $m + 1$  repetitions of  $u$ , it holds that the  $|w|$ -length factor ending at position  $i$  is equal to the  $|w|$ -length factor at position  $i - \iota$ , which proves the claim.  $\square$

The following lemma motivates our interest in infinite words of the form  $\text{flipext}^\omega(w)$ . It says that  $\text{flipext}^\omega(w)$  is the prefix normal word with the maximum number of 1's in each prefix among all prefix normal words having  $w$  as prefix.

**Lemma 4.17.** *Let  $w \in \mathcal{L}$ ,  $v = \text{flipext}^\omega(w)$ , and let  $z \in \mathcal{L}$  such that  $\text{pref}_{|w|}(z) = w$ . Then for every  $i = 1, 2, \dots$ , we have  $P_v(i) \geq P_z(i)$ .*

*Proof.* By contradiction, let  $i > |w|$  be the smallest integer such that  $P_v(i) < P_z(i)$ . Then, by the minimality of  $i$ , we have  $P_v(i - 1) \geq P_z(i - 1)$ , hence  $v_i = 0$  and  $z_i = 1$ . The definition of the operation  $\text{flipext}$  together with  $v_i = 0$  implies the existence of some  $j > 0$  such that  $P_v(j + 1) = |v_{i-j} \dots v_{i-1}|_1$  by Fact 4.1 (iv), for otherwise we would have  $v_i = 1$ . By the minimality of  $i$  it must also hold that  $P_z(j + 1) \leq P_v(j + 1)$ . Let us write  $v' = v_{i-j} \dots v_{i-1}$  and  $z' = z_{i-j} \dots z_{i-1}$ . Now assume that  $|z'|_1 \geq |v'|_1$ . Since  $|v'|_1 = P_v(j + 1) \geq P_z(j + 1) \geq |z'|_1$ , this implies  $P_v(j + 1) = P_z(j + 1)$ . But then  $P_z(j + 1) = P_v(j + 1) = |v'|_1 < |z'|_1 + 1 = |z'z_i|_1$ , in contradiction to  $z$  being prefix normal. So we have  $|z'|_1 < |v'|_1$ . Once more by the minimality of  $i$ , it also holds that  $P_v(i - j - 1) \geq P_z(i - j - 1)$ , leading to

$$P_v(i-1) = P_v(i-j-1) + |v'|_1 > P_z(i-j-1) + |z'|_1 = P_z(i-1),$$

which implies  $P_v(i) \geq P_z(i)$ , contradicting the initial assumption, and completing the proof.  $\square$

### 4.3.2 Lazy flip extensions

We now define a different operation that given a prefix normal word  $w$ , extends it by adding 0s as long as the minimum density of the resulting word is not smaller than  $\delta(w)$ , and only then adding a 1. We show that this operation preserves the prefix normality of the resulting word.

**Definition 4.15 (Operation lazy- $\alpha$ -flipext).** *Let  $\alpha \in (0, 1]$  and let  $w \in \mathcal{L}_{\text{fin}}$  with  $\delta(w) \geq \alpha$ . We define lazy- $\alpha$ -flipext( $w$ ) as the finite word  $w0^k1$  where  $k = \max\{j \mid \delta(w0^j) \geq \alpha\}$ . We further define the infinite word  $v = \text{lazy-}\alpha\text{-flipext}^\omega(w) = \lim_{i \rightarrow \infty} \text{lazy-}\alpha\text{-flipext}^{(i)}(w)$ .*

*Example 8.* Let  $w = 111$  and let  $\alpha = \sqrt{2} - 1$ , then lazy- $\alpha$ -flipext( $w$ ) = 11100001, since  $\delta(1110000) = 3/7 \geq \alpha$  and  $\delta(11100000) = 3/8 < \alpha$ .

Further, lazy- $\alpha$ -flipext<sup>(2)</sup>( $w$ ) = 1110000101, since  $\delta(111000010) = 4/9 \geq \alpha$  and  $\delta(1110000100) = 2/5 < \alpha$ .

**Lemma 4.18.** *Let  $\alpha \in (0, 1]$ . For every  $w \in \mathcal{L}_{\text{fin}}$  with  $\delta(w) \geq \alpha$ , the word  $v = \text{lazy-}\alpha\text{-flipext}(w)$  is also prefix normal, with  $\delta(v) \geq \alpha$ .*

*Proof.* First note that  $\delta(v) \geq \alpha$  by definition. Now write  $v = w0^k1$ , and let  $u = \text{flipext}(w) = w0^\ell1$ . Recall that  $\ell = \min\{j \mid w0^j1 \in \mathcal{L}\}$ . If  $k < \ell$ , this implies  $\delta(u) < \alpha$ , in contradiction to Proposition 4.14, since  $\delta(u) = \delta(w) \geq \alpha$ . Thus  $k \geq \ell$ , from which follows that  $v \in \mathcal{L}$ .

**Corollary 4.2.** *Let  $\alpha \in (0, 1]$  and  $w \in \mathcal{L}_{\text{fin}}$  with  $\delta(w) \geq \alpha$ . Then  $v = \text{lazy-}\alpha\text{-flipext}^\omega(w)$  is an infinite prefix normal word and  $\delta(v) = \alpha$ .*

*Proof.* That  $v$  is prefix normal follows from Lemma 4.1 and from Lemma 4.18, which also implies that  $\delta(v) \geq \alpha$ . If  $\delta(v) > \alpha$  was true, then for a suitably long prefix  $i$ , we would get a contradiction to the definition of the lazy- $\alpha$ -flipext operation.  $\square$

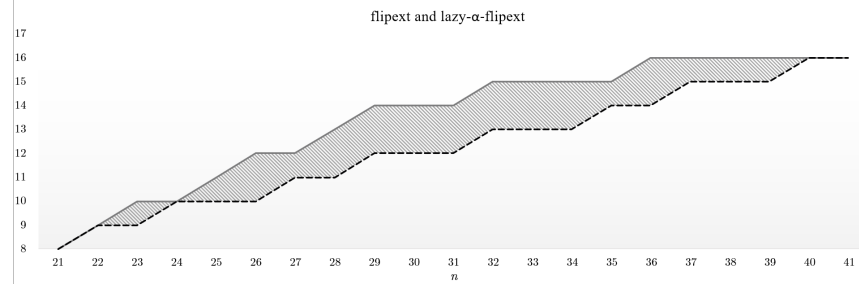
Fix  $w \in \mathcal{L}_{\text{fin}}$ , by definition of lazy- $\alpha$ -flipext we have that, given  $\alpha = \delta(w)$ , the lazy-flipext operation applied to  $w$  generates a prefix normal word that has the minimum number of 1s in the prefix among all prefix normal words having minimum density equals to  $\delta(w)$  and  $w$  as prefix. This is summarized in the following proposition.

**Proposition 4.1.** *Let  $w \in \mathcal{L}_{\text{fin}}$ ,  $\alpha = \delta(w)$ , and  $v = \text{lazy-}\alpha\text{-flipext}^\omega(w)$ . Then, for every  $z \in \mathcal{L}_{\text{inf}}$  such that  $\text{pref}_{|w|}(z) = w$  and  $\delta(w) \geq \delta(v)$ , for each  $i = 1, 2, \dots$  we have  $P_v(i) \leq P_z(i)$ .*

From Lemma 4.17 and Proposition 4.1, we have that, given  $w \in \mathcal{L}_{\text{fin}}$ , every prefix normal word with  $w$  as prefix and minimum density equals to  $\delta(w)$  is lexicographically between flipext <sup>$\omega$</sup> ( $w$ ) and lazy- $\alpha$ -flipext <sup>$\omega$</sup> ( $w$ ), where  $\alpha = \delta(w)$ . Formally, let  $u = \text{flipext}^\omega(w)$ ,  $\alpha = \delta(w)$ , and  $v = \text{lazy-}\alpha\text{-flipext}^\omega(w)$  for every  $z \in \mathcal{L}_{\text{inf}}$  such that  $\text{pref}_{|w|}(z) = w$  and  $\delta(w) \geq \delta(v)$ , we have that  $P_v(i) \leq P_z(i) \leq P_u(i)$  and  $v \leq_{\text{lex}} u$ .

*Example 9.* Let  $w = 1101101100100010000001$  and let  $\alpha = \delta(w) = 8/13$ , then  $u = \text{flipext}^8(w) = w101101100100010000001$  and  $v = \text{lazy-}\alpha\text{-flipext}^8(w) = w01001010010010100100$ .

Let  $p = w100111100100010000001$  and  $q = w101101010100001000001$ , we have that for all  $1 \leq i \leq 42$ ,  $P_v(i) \leq P_q(i) \leq P_p(i) \leq P_u(i)$  and  $v \leq_{\text{lex}} q \leq_{\text{lex}} p \leq_{\text{lex}} u$ . Note that  $p$  is not prefix normal, while  $q$  is prefix normal.



**Fig. 4.4:** Given  $w = 1101101100100010000001$  the plot represents the last characters of  $\text{flipext}^8(w)$  (solid) and the  $\text{lazy-}\alpha\text{-flipext}^8(w)$  (dashed). See Example 9. A 1 corresponds to a diagonal segment in direction NE, while a 0 to one in direction SE. On the  $x$ -axis we have the length of the prefix, and on the  $y$ -axis, the number of 1s minus the number of 0s in the prefix. The shaded area denotes the area in which there are all prefix normal words with  $w$  as prefix and minimum density equal to  $\delta(w)$ . Note that not all the binary words in that area are prefix normal.

### 4.4 Prefix normal words and Sturmian words

In the previous section, we presented operations that generate binary words by extension. In particular, the  $\text{lazy-}\alpha\text{-flipext}$  operation extends a finite binary word with as few 1s as possible, in order to preserve its minimum density. This is reminiscent of the characterization of Sturmian words in terms of mechanical words and the slope. Led by this analogy, in this section we provide a complete characterization of Sturmian words which are prefix normal. We refer the interested reader to [92, Chapter 2], for a comprehensive treatment of Sturmian words. Here we briefly recall some facts we will need later.

**Definition 4.16 (Sturmian words).** Let  $w \in \{0, 1\}^\omega$ . Then  $w$  is called Sturmian if it is balanced and aperiodic.

An equivalent definition of Sturmian words is that they are irrational mechanical, a definition we recall next.

**Definition 4.17 (Mechanical words).** Given two real numbers  $0 \leq \alpha \leq 1$  and  $0 \leq \tau < 1$ , the lower mechanical word  $s_{\alpha,\tau} = s_{\alpha,\tau}(1) s_{\alpha,\tau}(2) \cdots$  and the upper mechanical word  $s'_{\alpha,\tau} = s'_{\alpha,\tau}(1) s'_{\alpha,\tau}(2) \cdots$  are given by

$$\begin{aligned} s_{\alpha,\tau}(n) &= \lfloor \alpha n + \tau \rfloor - \lfloor \alpha(n-1) + \tau \rfloor \\ s'_{\alpha,\tau}(n) &= \lceil \alpha n + \tau \rceil - \lceil \alpha(n-1) + \tau \rceil \end{aligned} \quad (n \geq 1).$$

Then  $\alpha$  is called the slope and  $\tau$  the intercept of  $s_{\alpha,\tau}, s'_{\alpha,\tau}$ . A word  $w$  is called mechanical if  $w = s_{\alpha,\tau}$  or  $w = s'_{\alpha,\tau}$  for some  $\alpha, \tau$ . It is called rational mechanical (resp. irrational mechanical) if  $\alpha$  is rational (resp. irrational).

- Fact 4.2 (Some facts about Sturmian words[92])**
1. An infinite binary word is Sturmian if and only if it is irrational mechanical.
  2. For  $\tau = 0$ , and  $\alpha$  irrational, there exists a word  $c_\alpha$ , called the characteristic word with slope  $\alpha$ , s.t.  $s_{\alpha,0} = 0c_\alpha$  and  $s'_{\alpha,0} = 1c_\alpha$ . This word  $c_\alpha$  is a Sturmian word itself, with both slope and intercept  $\alpha$ .
  3. For two Sturmian words  $w$  and  $v$  with the same slope, we have  $\text{Fct}(w) = \text{Fct}(v)$ .

We now show that the word  $\text{lazy-}\alpha\text{-flipext}^\omega(1)$  coincides with the upper mechanical word  $s'_{\alpha,0}$  which also implies that  $s'_{\alpha,0}$  is prefix normal as noted in the following corollary.

**Lemma 4.19.** Fix  $\alpha \in (0, 1]$  and let  $v = \text{lazy-}\alpha\text{-flipext}^\omega(1)$ . Let  $s = s'_{\alpha,0}$  be the upper mechanical word of slope  $\alpha$  and intercept 0. Then  $v = s$ .

*Proof.* Let  $s_i$  and  $v_i$  denote the  $i$ th character of  $s$  and  $v$  respectively. We argue by induction on  $i$  that  $v_i = s_i$ . The claim is true for  $i = 1$  since, directly from the definitions we have  $v_1 = 1 = s_1$ . Let  $n > 1$  and assume that for each  $i < n$  we have  $v_i = s_i$ . For the induction step we argue according to the character  $s_n$ .

(i) If  $s_n = 1$ , by definition  $\lceil n\alpha \rceil - \lceil (n-1)\alpha \rceil = 1$ . Thus,  $\lceil (n-1)\alpha \rceil < n\alpha$ . Using this inequality and the induction hypothesis together with the definition of  $s'_{\alpha,0}$  we have that  $|v_1 \cdots v_{n-1}|_1 = |s_1 \cdots s_{n-1}|_1 = \lceil (n-1)\alpha \rceil < \alpha n$ . Therefore  $|v_1 \cdots v_{n-1}0|_1 = |v_1 \cdots v_{n-1}|_1 < \alpha n$  which means that  $\delta(v_1 \cdots v_{n-1}0) < \alpha$ , hence by definition  $\text{lazy-}\alpha\text{-flipext}(v_1 \cdots v_{n-1}) = v_1 \cdots v_{n-1}1$ , i.e.,  $v_n = 1 = s_n$ .

(ii) If  $s_n = 0$ , by definition  $\lceil n\alpha \rceil - \lceil (n-1)\alpha \rceil = 0$ . Thus,  $\lceil (n-1)\alpha \rceil \geq n\alpha$ . Using this inequality and the induction hypothesis together with the definition of  $s'_{\alpha,0}$  we have that  $|v_1 \cdots v_{n-1}|_1 = |s_1 \cdots s_{n-1}|_1 = \lceil (n-1)\alpha \rceil \geq \alpha n$ . Therefore  $|v_1 \cdots v_{n-1}0|_1 = |v_1 \cdots v_{n-1}|_1 \geq \alpha n$  which means that  $\delta(v_1 \cdots v_{n-1}0) \geq \alpha$ , hence by definition  $\text{lazy-}\alpha\text{-flipext}(v_1 \cdots v_{n-1}) = v_1 \cdots v_{n-1}0 \cdots 01$ , i.e.,  $v_n = 0 = s_n$ .  $\square$

**Corollary 4.3.** Given  $\alpha \in (0, 1]$  then  $s'_{\alpha,0}$  is an infinite prefix normal word and  $\delta(s'_{\alpha,0}) = \alpha$ .

The following theorem fully characterizes those Sturmian words which are prefix normal.

**Theorem 4.6.** A Sturmian word  $s$  of slope  $\alpha$  is prefix normal if and only if  $s = 1c_\alpha$ , where  $c_\alpha$  is the characteristic Sturmian word with slope  $\alpha$ .

*Proof.* By definition,  $\alpha$  is irrational. Let  $s = s'_{\alpha,0}$ . Then  $s$  is Sturmian and prefix normal by Corollary 4.3. Let  $t$  be a Sturmian word with the same slope  $\alpha$  which is also prefix normal. By Fact 4.2,  $s$  and  $t$  have the same factors.

Assume, by contradiction, that  $s \neq t$ , hence there exists  $i \geq 1$  such that  $|s_1 \cdots s_i|_1 \neq |t_1 \cdots t_i|_1$ . Assume, without loss of generality (since we can, if necessary, swap  $s$  and  $t$  in the following argument), that  $|s_1 \cdots s_i|_1 > |t_1 \cdots t_i|_1$ . Then, since  $s_1 \cdots s_i$  is also a factor of  $t$ , there is a  $j \geq 1$  such that  $t_{j+1} \cdots t_{j+i} = s_1 \cdots s_i$ , hence  $|t_{j+1} \cdots t_{j+i}|_1 > |t_1 \cdots t_i|_1$  contradicting the assumption that  $t$  is prefix normal.  $\square$

## 4.5 Prefix normal words and lexicographic order

In this section, we study the relationship between lexicographic order and prefix normality. A class of well-known binary words connected with prefix normality are Lyndon words. Notice that the prefix normal condition is different from the Lyndon condition<sup>2</sup>: For finite words, there are words which are both Lyndon and prefix normal (e.g. 110010), words which are Lyndon but not prefix normal (11100110110), words which are prefix normal but not Lyndon (110101), and words which are neither (101100). In the final part of the chapter, we will put infinite prefix normal words and their prefix normal forms in the context of lexicographic orderings, and compare them to infinite Lyndon words [123], necklaces, prenecklaces [92, 118], and the max- and min-words of [110].

A finite *Lyndon word* is one which is lexicographically strictly greater than all of its conjugates:  $w$  is Lyndon if and only if for all non-empty  $u, v$  s.t.  $w = uv$ , we have  $w >_{\text{lex}} vu$ . A *necklace* is a word which is greater than or equal to all its conjugates, and a *prenecklace* is one which can be extended to become a necklace, i.e. which is the prefix of some necklace [92, 118]. As we saw in the introduction, in the finite case, prefix normality and Lyndon property are orthogonal concepts. However, the set of finite prefix normal words is included in the set of prenecklaces [25].

An infinite word is *Lyndon* if an infinite number of its prefixes is Lyndon [123]. In the infinite case, we have a similar situation as in the finite case. There are words which are both Lyndon and prefix normal:  $10^\omega$ ,  $110(10)^\omega$ ; Lyndon but not prefix normal:  $11100(110)^\omega$ ; prefix normal but not Lyndon:  $(10)^\omega$ ; and neither of the two:  $(01)^\omega$ .

Next we show that a prefix normal word cannot be lexicographically smaller than any of its suffixes. Let  $\text{shift}_i(w) = w_i w_{i+1} w_{i+2} \cdots$  denote the infinite word  $v$  s.t.  $w = w_1 \cdots w_{i-1} v$ , i.e.  $v$  is the suffix of  $w$  starting at position  $i$ .

**Lemma 4.20.** *Let  $w \in \mathcal{L}_{\text{inf}}$ . Then  $w \geq_{\text{lex}} \text{shift}_i(w)$  for all  $i \geq 1$ .*

*Proof.* Assume that there exists a suffix  $v = \text{shift}_i(w)$  of  $w$  s.t.  $v >_{\text{lex}} w$ . Then there is an index  $j$  with  $v_1 \cdots v_{j-1} = w_1 \cdots w_{j-1}$  and  $v_j > w_j$ , implying  $v_j = 1$  and  $w_j = 0$ . But then  $|w_i \cdots w_{i+j-1}|_1 = |v_1 \cdots v_j|_1 > |w_1 \cdots w_j|_1$ , in contradiction to  $w \in \mathcal{L}_{\text{inf}}$ .  $\square$

In the finite case, it is easy to see that a word  $w$  is a prenecklace if and only if  $w \geq_{\text{lex}} v$  for every suffix  $v$  of  $w$ . This motivates our definition of infinite prenecklaces. The situation is the same as in the finite case: prefix normal words form a proper subset of prenecklaces.

**Definition 4.18.** *Let  $w \in \{0, 1\}^\omega$ . Then  $w$  is an infinite prenecklace if for all  $i \geq 1$ ,  $w \geq_{\text{lex}} \text{shift}_i(w)$ . We denote by  $\mathcal{P}_{\text{inf}}$  the set of infinite prenecklaces.*

**Proposition 4.2.** *We have  $\mathcal{L}_{\text{inf}} \subsetneq \mathcal{P}_{\text{inf}}$ .*

*Proof.* The inclusion follows from Lemma 4.20. An example of a word which is an infinite prenecklace but not prefix normal is  $11100(110)^\omega$ .  $\square$

<sup>2</sup> For ease of presentation, we are using Lyndon to mean lexicographically *greatest* among its conjugates; this is equivalent to the usual definition up to renaming of characters.

There is another interesting relationship between lexicographic order and the prefix normal forms of an infinite word. In [110], two words were associated to an infinite binary word  $w$ , called  $\max(w)$  (resp.  $\min(w)$ ), defined as the word whose prefix of length  $n$  is the lexicographically greatest (resp. smallest)  $n$ -length factor of  $w$ . It is easy to see that these words always exist. The following was shown in [110]:<sup>3</sup>

**Theorem 4.7 ([110]).** *Let  $w$  be an infinite binary word. Then*

1.  $w$  is (rational or irrational) mechanical with its intercept equal to its slope if and only if  $0w \leq_{\text{lex}} \min(w) \leq_{\text{lex}} \max(w) \leq_{\text{lex}} 1w$ , and
2.  $w$  is characteristic Sturmian if and only if  $\min(w) = 0w$  and  $\max(w) = 1w$ .

**Lemma 4.21.** *Let  $w \in \{0, 1\}^\omega$ . Then  $\text{PNF}_1(w) \geq_{\text{lex}} \max(w)$  and  $\text{PNF}_0(w) \leq_{\text{lex}} \min(w)$ .*

*Proof.* Assume otherwise, and let  $w' := \text{PNF}_1(w)$ ,  $v := \max(w)$ . If  $w' < v$ , then there is an index  $j$  s.t.  $w'_1 \cdots w'_{j-1} = v_1 \cdots v_{j-1}$  and  $w'_j = 0$  and  $v_j = 1$ . This implies that  $v_1 \cdots v_j$  has one more 1s than  $w'_1 \cdots w'_j$ . But  $|w'_1 \cdots w'_j|_1 = F_w^1(j)$ , a contradiction, since  $v_1 \cdots v_j$  is a factor of  $w$ . The second claim follows analogously.  $\square$

Finally, from Theorems 4.10 and 4.7 we get the following corollary:

**Corollary 4.4.** *Let  $w$  be an infinite binary word. Then  $w$  is characteristic Sturmian if and only if  $0w = \text{PNF}_0(w) = \min(w)$  and  $1w = \text{PNF}_1(w) = \max(w)$ .*

## 4.6 Prefix normal words, prefix normal forms, abelian complexity

In Section 4.4, we have shown that a Sturmian word  $w$  is prefix normal if and only if  $w = 1c_\alpha$  for some  $\alpha$ , where  $c_\alpha$  is the characteristic word of slope  $\alpha$  (Theorem 4.6). Indeed, the Fibonacci word

$$f = 0100101001001010010100100101001001 \cdots$$

is not prefix normal (it begins with a 0 but is not constant 0). But we can turn it into a prefix normal word by prepending a 1, i.e. the word  $1f$  is prefix normal. We show in fact that every Sturmian word  $w$  can be turned into a prefix normal word by prepending a fixed number of 1s, which only depends on the slope of  $w$ . This follows from a more general result regarding  $c$ -balanced words (Lemma 4.23).

As another example, the Thue-Morse word

<sup>3</sup> The terminology in [110] differs from ours (we are following [92]). In order to help the reader we here highlight the differences: (i) a periodic Sturmian in [110] is a rational mechanical word, (ii) a proper Sturmian word in [110] is an irrational mechanical word (i.e., a Sturmian word), and (iii) a standard Sturmian word in [110] is a mechanical word for with intercept  $\tau = \alpha$ , thus a proper standard Sturmian word is a characteristic Sturmian word  $c_\alpha$ . Note that all mechanical words in [110] are defined for  $n \geq 1$  since the definition of mechanical word is: the lower mechanical word is defined as  $s_{\alpha, \tau}(n) = \lfloor \alpha(n+1) + \tau \rfloor - \lfloor \alpha n + \tau \rfloor$  for  $n \geq 1$ , and analogously for the upper mechanical word. Therefore, an intercept  $\tau = 0$  in [110] is equivalent to an intercept of  $\tau = \alpha$  (the slope) in [92].

$$\mathbf{tm} = 01101001100101101001011001101001 \dots$$

is not prefix normal, but  $11\mathbf{tm}$  is. However, the binary Champernowne word, which is constructed by concatenating the binary expansions of the integers in ascending order, namely

$$\mathbf{c} = 0110111001011101111000100110101011 \dots$$

is not prefix normal and cannot be turned into a prefix normal word by prepending a finite number of 1s, because  $\mathbf{c}$  has arbitrarily long runs of 1s.

One might conclude that every word with bounded abelian complexity can be turned into a prefix normal word by prepending a fixed number of 1s, as is the case for the words above:  $\mathbf{f}$  has abelian complexity constant 2,  $\mathbf{tm}$  has abelian complexity bounded by 3, and  $\mathbf{c}$  has unbounded abelian complexity. However, this is not the case (see Section 4.6).

The notion of prefix normal *forms* from [57] can be extended to infinite words. They can be used, similarly to the finite case, to encode the abelian complexity of the original word. The study of abelian complexity of infinite words was initiated in [115], and continued e.g. in [18, 26, 76, 94, 130]. We establish a close relationship between the abelian complexity and the prefix normal forms of  $w$ . We demonstrate how this close connection can be used to derive results about the prefix normal forms of a word  $w$ . In some cases, such as for Sturmian words and words which are morphic images under the Thue-Morse morphism, we are able to explicitly give the prefix normal forms of the word. Conversely, knowing its prefix normal forms allows us to derive results about the abelian complexity of a word. We also provide an algorithm to compute the prefix normal forms of words that are *binary uniform morphisms*, based on an algorithm that computes their abelian complexity [19].

Given an infinite word  $w$ , the *abelian complexity* function of  $w$ , denoted  $\psi_w$ , is given by  $\psi_w(n) = |\{pv(u) \mid u \in \text{Fct}(w), |u| = n\}|$ , the number of Parikh vectors of  $n$ -length factors of  $w$ . A word  $w$  is said to have bounded abelian complexity if there exists a  $c$  s.t. for all  $n$ ,  $\psi_w(n) \leq c$ . Note that a binary word is  $c$ -balanced if and only if its abelian complexity is bounded by  $c + 1$ . We denote the set of Parikh vectors of factors of a word  $w$  by  $\Pi(w) = \{pv(u) \mid u \in \text{Fct}(w)\}$ . Thus,  $\psi_w(n) = |\Pi(w) \cap \{(x, y) \mid x + y = n\}|$ . In this section, we study the connection between prefix normal words and abelian complexity.

#### 4.6.1 Balanced and $c$ -balanced words.

Based on the examples above, one could conclude that any word with bounded abelian complexity can be turned into a prefix normal word by prepending a fixed number of 1s. However, consider the word  $w = 01^\omega$ , which is balanced, i.e. its abelian complexity function is bounded by 2. It is easy to see that  $1^k w \notin \mathcal{L}$  for every  $k \in \mathbb{N}$ .

Sturmian words are precisely the words which are aperiodic and whose abelian complexity is constant 2 [115]. For Sturmian words, it is always possible to prepend a finite number of 1s to get a prefix normal word, as we will see next. Recall that for a Sturmian word  $w$ , at least one of  $0w$  and  $1w$  is Sturmian, with both being Sturmian if and only if  $w$  is characteristic [92].

**Lemma 4.22.** *Let  $w$  be a Sturmian word. Then*

1.  $1w \in \mathcal{L}$  if and only if  $0w$  is Sturmian,
2. if  $0w$  is not Sturmian, then  $1^n w \in \mathcal{L}$  for  $n = \lceil 1/(1 - \alpha) \rceil$ .

*Proof.* 1. Let  $0w$  be Sturmian and let  $u$  be some factor of  $1w$ . If  $u$  is a prefix of  $1w$ , there is nothing to show, therefore let  $u \in \text{Fct}(w)$ , with  $|u| = n$  and  $|u|_1 = k$ . Since  $0w$  is Sturmian, we have that the prefix of  $0w$  of length  $n$  has at least  $k - 1$  1s, thus  $P_{1w}(n) \geq k = |u|_1$ , as desired. Conversely, if  $0w$  is not Sturmian, this means that it is not balanced, therefore there exists a factor  $u$  of  $w$  s.t.  $||u|_1 - |0w_1 \cdots w_{n-1}|_1| \geq 2$ , where  $|u| = n$ . Since  $w$  is Sturmian, we have that  $||w_1 \cdots w_{n-1}|_1 - |u_1 \cdots u_{n-1}|_1| \leq 1$  and  $||w_1 \cdots w_{n-1}|_1 - |u_2 \cdots u_n|_1| \leq 1$ . Let  $|w_1 \cdots w_{n-1}|_1 = k$ , then this implies, by a case-by-case consideration, that  $|u_1 \cdots u_{n-1}|_1 = |u_2 \cdots u_n|_1 = k + 1$ , and thus  $|1w_1 \cdots w_{n-1}|_1 = k + 1 < k + 2 = |u|_1$ , showing that  $1w$  is not prefix normal.

2. First note that a Sturmian word of slope  $\alpha$  cannot have a run of 1s of length  $1/(1 - \alpha)$ . To see this, it is enough to argue about the upper mechanical word of slope  $\alpha$  and intercept 0 (since all the other words with the same slope have the same set of factors). Let us write  $s = s_{\alpha,0} = s_1 s_2 \cdots$

Now  $s$  has a run of  $n$  1s iff there exists an  $i \geq 0$  such that  $s_{i+1} = s_{i+2} = \cdots = s_{i+n} = 1$ . By the definition of mechanical words, we have that the last condition is equivalent to

$$\lceil \alpha(i + n) \rceil - \lceil \alpha i \rceil = n.$$

On the other hand, if  $n \geq \frac{1}{1-\alpha}$ , i.e.,  $\alpha \leq \frac{n-1}{n}$  we have that the sum of the character  $\sum_{j=1}^n s_{i+j}$  satisfies

$$\begin{aligned} \sum_{j=1}^n s_{i+j} &= \lceil \alpha(i + n) \rceil - \lceil \alpha i \rceil \\ &\leq \lceil \alpha i \rceil + \lceil \alpha n \rceil - \lceil \alpha i \rceil \\ &= \lceil \alpha n \rceil \\ &< \alpha n + 1 \\ &\leq \frac{n-1}{n} \times n + 1 = n. \end{aligned}$$

i.e., strictly smaller than  $n$ , i.e., we have a contradiction  $s_{i+1} \cdots s_{i+n} \neq 1^n$ .

Now fix  $n = \lceil 1/(1 - \alpha) \rceil$  and let  $w' = 1^n w$ . Let  $u \in \text{Fct}(w)$ . Since, as shown above,  $1^n$  is not a factor, if  $|u| \leq n$ , there is nothing to show. So let  $|u| = n + m$ . Then  $|u_1 \cdots u_n|_1 \leq n - 1$ , and since  $w$  is balanced, we have that  $|w_1 \cdots w_m|_1 \geq |u_{n+1} \cdots u_{n+m}|_1 - 1$ , yielding that  $P_{w'}(n + m) \geq n + |u_{n+1} \cdots u_{n+m}|_1 - 1 \geq |u|_1$ .  $\square$

**Lemma 4.23.** *Let  $w$  be a  $c$ -balanced word. If there exists a positive integer  $n$  s.t.  $1^n \notin \text{Fct}(w)$ , then the word  $z = 1^{nc} w$  is prefix normal.*

*Proof.* We are going to show that every factor  $u$  of  $z$  satisfies the prefix normal condition  $|u|_1 \leq P_z(|u|)$ . It is not hard to see that we can limit ourselves to only considering factors  $u$  such that  $u$  does not overlap with the prefix of  $z$  of the same length.

If  $|u| \leq nc$  then  $|u|_1 \leq |u| = P_z(|u|)$ . Assume now that  $u = u' u''$  with  $|u'| = nc$  and  $|u''| > 0$ . Since  $u'$  is a factor of  $w$  of size  $nc$  the condition that  $w$  does not contain



a factor  $1^n$  implies that  $u'$  contains at least  $c$  0s, i.e.,  $|u'|_1 \leq |u'| - c$ . Moreover, since  $w$  is  $c$ -balanced, we have that  $|u''|_1 \leq P_w(|u''|) + c$ . Therefore, observing that  $\text{pref}_z(|u|) = \text{pref}_z(|u'| + |u''|) = 1^{nc} \text{pref}_w(|u''|)$  we have that  $P_z(|u|) = nc + P_w(|u''|) \geq |u'|_1 + |u''|_1 = |u|_1$ .  $\square$

In particular, Lemma 4.23 implies that any  $c$ -balanced word with infinitely many 0s can be turned into a prefix normal word by prepending a finite number of 1s, since such a word cannot have arbitrarily long runs of 1s. Note, however, that the number of 1s to prepend from Lemma 4.23 is not tight, as can be seen e.g. from the Thue-Morse word  $\mathbf{tm}$ : the longest run of 1s in  $\mathbf{tm}$  is 2 and  $\mathbf{tm}$  is 2-balanced, but  $11\mathbf{tm}$  is prefix normal, as will be shown in the next section (Lemma 4.26).

#### 4.6.2 Prefix normal forms and abelian complexity.

Recall that for a word  $w$ ,  $F_w^a(i)$  is the maximum number of  $a$ 's in a factor of  $w$  of length  $i$ , for  $a \in \{0, 1\}$ .

**Definition 4.19 (Prefix normal forms).** *Let  $w \in \{0, 1\}^\omega$ . Define the words  $w'$  and  $w''$  by setting, for  $n \geq 1$ ,  $w'_n = F_w^1(n) - F_w^1(n-1)$  and  $w''_n = F_w^0(n) - F_w^0(n-1)$ . We refer to  $w'$  as the prefix normal form of  $w$  w.r.t. 1 and to  $w''$  as the prefix normal form of  $w$  w.r.t. 0, denoted  $\text{PNF}_1(w)$  resp.  $\text{PNF}_0(w)$ .*

In other words,  $\text{PNF}_1(w)$  is the sequence of first differences of the maximum-1s function  $F_w^1$  of  $w$ . Similarly,  $\text{PNF}_0(w)$  can be obtained by complementing the sequence of first differences of the maximum-0s function  $F_w^0$  of  $w$ . Note that for all  $n$  and  $a \in \{0, 1\}$ , either  $F_w^a(n+1) = F_w^a(n)$  or  $F_w^a(n+1) = F_w^a(n) + 1$ , and therefore  $w'$  and  $w''$  are words over the alphabet  $\{0, 1\}$ . In particular, by construction, the two prefix normal words allow us to recover the maximum-1s and minimum-1s functions of  $w$ :

**Observation 4.2** *Let  $w$  be an infinite binary word and  $w' = \text{PNF}_1(w)$ ,  $w'' = \text{PNF}_0(w)$ . Then  $P_{w'}(n) = F_w^1(n)$  and  $P_{w''}(n) = n - F_w^0(n) = f_w^1(n)$ .*

**Lemma 4.24.** *Let  $w \in \{0, 1\}^\omega$ . Then  $\text{PNF}_1(w)$  is the unique 1-prefix normal word  $w'$  s.t.  $F_{w'}^1 = F_w^1$ . Similarly,  $\text{PNF}_0(w)$  is the unique 0-prefix normal word  $w''$  s.t.  $F_{w''}^0 = F_w^0$ .*

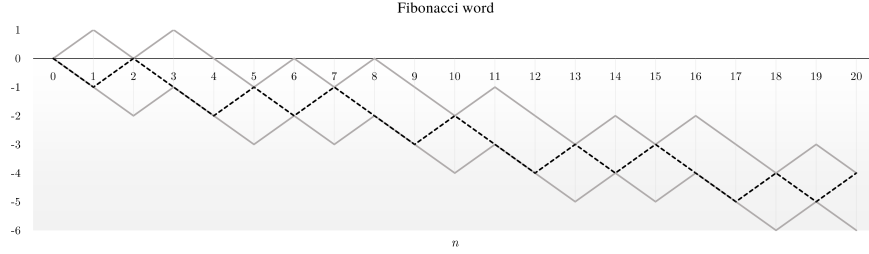
*Proof.* Let  $w' = \text{PNF}_1(w)$  and  $w'' = \text{PNF}_0(w)$ . First note that, by construction,  $F_{w'}^1 = F_w^1$  and  $F_{w''}^0 = F_w^0$ . It is easy to see that  $w'$  is 1-prefix normal and  $w''$  is 0-prefix normal. For uniqueness, note that for  $a \in \{0, 1\}$  and an  $a$ -prefix normal word  $v$ , we have  $\text{PNF}_a(v) = v$ .  $\square$

*Example 10.* The two prefix normal forms and the maximum-1s and maximum-0s functions of the Fibonacci word  $\mathbf{f} = 01001010010010100101 \dots$  are given in Table 4.3.

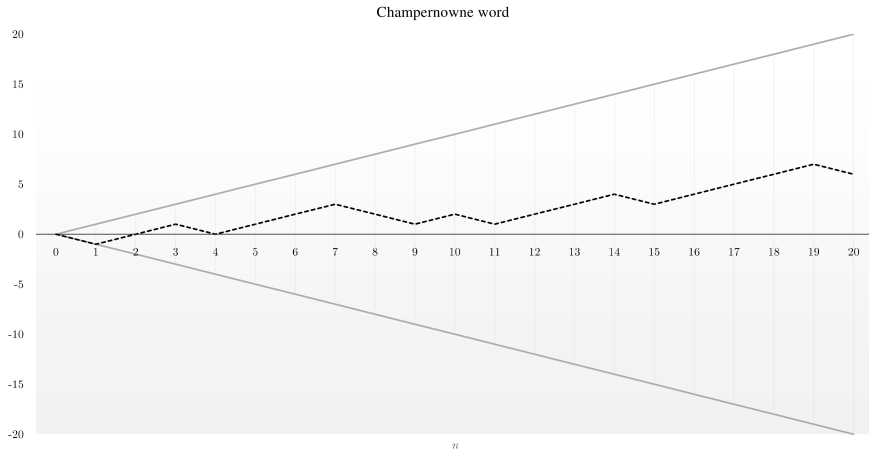
Now we can connect the prefix normal forms of  $w$  to the abelian complexity of  $w$  in the following way. Given  $w' = \text{PNF}_1(w)$  and  $w'' = \text{PNF}_0(w)$ , the number of Parikh vectors of  $k$ -length factors is precisely the difference in 1s in the prefix of length  $k$  of  $w'$  and of  $w''$  plus 1. For example, Fig. 4.5 shows the prefix normal forms of the Fibonacci word. The vertical line at 5 cuts through points  $(5, -1)$  and  $(5, -3)$ , meaning that there are two Parikh vectors of factors of length 5, namely  $(2, 3)$  and  $(1, 4)$ . The Fibonacci

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
$F_{\mathbf{f}}^0(n)$	1	2	2	3	4	4	5	5	6	7	7	8	9	9	10	10	11	12	12	13
$F_{\mathbf{f}}^1(n)$	1	1	2	2	2	3	3	4	4	4	5	5	5	6	6	7	7	7	8	8
$\text{PNF}_0(\mathbf{f})$	0	0	1	0	0	1	0	1	0	0	1	0	0	1	0	1	0	0	1	0
$\text{PNF}_1(\mathbf{f})$	1	0	1	0	0	1	0	1	0	0	1	0	0	1	0	1	0	0	1	0

**Table 4.3:** The maximum number of 0s and 1s ( $F_{\mathbf{f}}^0(n)$  and  $F_{\mathbf{f}}^1(n)$  resp.) for all  $n = 1, \dots, 20$  of the Fibonacci word  $\mathbf{f}$ , and the prefix normal forms of  $\mathbf{f}$ .



**Fig. 4.5:** The Fibonacci word (dashed) and its prefix normal forms (solid).



**Fig. 4.6:** The Champernowne word (dashed) and its prefix normal forms (solid).

word, being a Sturmian word, has constant abelian complexity 2. An example of a word with unbounded abelian complexity is the Champernowne word, whose prefix normal forms are  $1^\omega$  resp.  $0^\omega$ . (Fig. 4.6, Appx.).

**Theorem 4.8.** Let  $w, v \in \{0, 1\}^\omega$ .

1. We have  $\psi_w(n) = P_{w'}(n) - P_{w''}(n) + 1$ , where  $w' = \text{PNF}_1(w)$  and  $w'' = \text{PNF}_0(w)$ .
2. We have  $\Pi(w) = \Pi(v)$  if and only if  $\text{PNF}_0(w) = \text{PNF}_0(v)$  and  $\text{PNF}_1(w) = \text{PNF}_1(v)$ .

*Proof.* 1. Fix an integer  $n \geq 1$ . By definition, we have that for every factor  $u$  of  $w$  of length  $n$  we have  $n - F_w^0(n) \leq |u|_1 \leq F_w^1(n)$ . Therefore  $\psi_w(n) \leq F_w^1(n) - (n - F_w^0(n)) + 1$ .

Conversely, since  $w$  contains a factor  $u'$  of length  $n$  with  $F_w^1(n)$  many 1s and a factor  $u''$  of length  $n$  with  $n - F_w^0(n)$  many 1s, if we scan  $w$  between an occurrence of  $u'$  and an occurrence of  $u''$ , for each  $x \in \{|u''|_1, \dots, |u'|_1\}$  there must be a factor  $u'''$  of size  $n$  such that  $|u'''|_1 = x$ . Therefore  $\psi_w(n) \geq F_w^1(n) - (n - F_w^0(n)) + 1$ . We can conclude that  $\psi_w(n) = F_w^1(n) - (n - F_w^0(n)) + 1$ . The desired result then follows by observing that  $n - F_w^0(n) = n - |\text{pref}_{\text{PNF}_0(w)}(n)|_0 = P_{\text{PNF}_0(w)}(n)$  and  $F_w^1(n) = P_{\text{PNF}_1(w)}(n)$ .

2. Follows directly from Observation 4.2.  $\square$

Theorem 4.8 means that if we know the prefix normal forms of a word, then we can compute its abelian complexity. Conversely, the abelian complexity is the *width* of the area enclosed by the two words  $\text{PNF}_1(w)$  and  $\text{PNF}_0(w)$ . In general, this fact alone does not give us the PNFs; but if we know more about the word itself, then we may be able to compute the prefix normal forms, as we will see in the case of the paperfolding word.

We will now give two examples of the close connection between abelian complexity and prefix normal forms, using some recent results about the abelian complexity of infinite words.

1. *The paperfolding word.* The first few characters of the ordinary paperfolding word are given by

$$\mathbf{p} = 0010011000110110001001110011011 \dots$$

The paperfolding word was originally introduced in [52]. One definition is given by:  $\mathbf{p}_n = 0$  if  $n' \equiv 1 \pmod 4$  and  $\mathbf{p}_n = 1$  if  $n' \equiv 3 \pmod 4$ , where  $n'$  is the unique odd integer such that  $n = n'2^k$  for some  $k$  [94]. The abelian complexity function of the paperfolding word was fully determined in [94], giving the following initial values for  $\psi_{\mathbf{p}}(n)$ , for  $n \geq 1$ : 2, 3, 4, 3, 4, 5, 4, 3, 4, 5, 6, 5, 4, 5, 4, 3, 4, 5, 6, 5, and a recursive formula for the computation of all values. The authors note that for the paperfolding word, it holds that if  $u \in \text{Fct}(\mathbf{p})$ , then also  $\overline{u^{\text{rev}}} \in \text{Fct}(\mathbf{p})$ . This implies

$$F_{\mathbf{p}}^1(n) = F_{\mathbf{p}}^0(n) \text{ for all } n, \text{ and thus } \text{PNF}_0(\mathbf{p}) = \overline{\text{PNF}_1(\mathbf{p})}.$$

Moreover, from Thm. 4.8 we get that  $F_{\mathbf{p}}^1(n) = P_{\text{PNF}_1(\mathbf{p})}(n) = (\psi_{\mathbf{p}}(n) + n - 1)/2$ , and thus we can determine the prefix normal forms of  $\mathbf{p}$ , see Fig. 4.7.

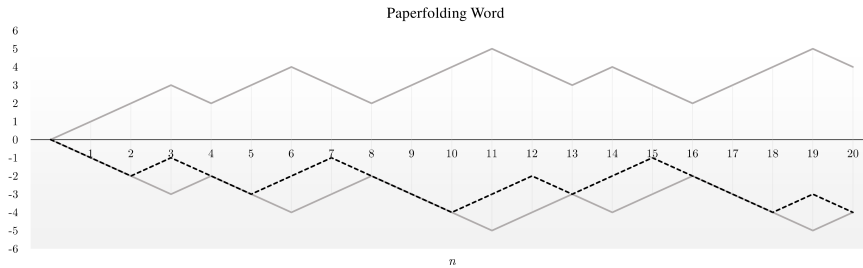


Fig. 4.7: The paperfolding word (dashed) and its prefix normal forms (solid).

This same argument holds in general as long as the word has the symmetric property similar to the paperfolding word:

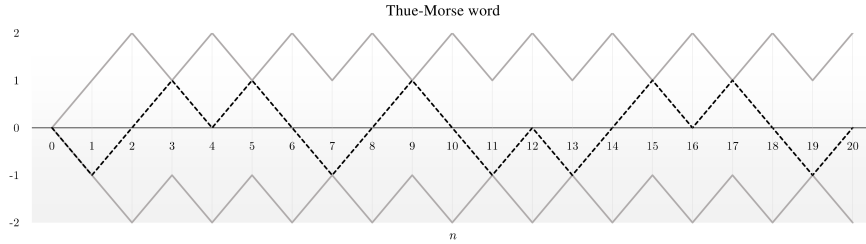
**Lemma 4.25.** *Let  $w \in \{0, 1\}^\omega$ . If for all  $u \in \text{Fct}(w)$ , it holds that  $\bar{u} \in \text{Fct}(w)$  or  $\overline{u^{\text{rev}}} \in \text{Fct}(w)$ , then  $F_w^1(n) = F_w^0(n)$  for all  $n$ ,  $\text{PNF}_0(w) = \text{PNF}_1(w)$ , and  $F_w^1(n) = (\psi_w(n) + n - 1)/2$ .*

*Proof.* Same as for the special case of the paperfolding word.  $\square$

2. *Morphic images under the Thue-Morse morphism.* The Thue-Morse word beginning with 0, which we denote by  $\mathbf{tm}$ , is one of the two fixpoints of the Thue-Morse morphism  $\mu_{\text{TM}}$ , where  $\mu_{\text{TM}}(0) = 01$  and  $\mu_{\text{TM}}(1) = 10$ :

$$\mathbf{tm} = \mu_{\text{TM}}^{(\omega)}(0) = 01101001100101101001011001101001 \dots$$

The word  $\mathbf{tm}$  has abelian complexity function  $\psi_{\mathbf{tm}}(n) = 2$  for  $n$  odd and  $\psi_{\mathbf{tm}}(n) = 3$  for  $n > 1$  even [115]. Since  $\mathbf{tm}$  fulfils the condition that  $u \in \text{Fct}(\mathbf{tm})$  implies  $\bar{u} \in \text{Fct}(\mathbf{tm})$ , we can apply Lemma 4.25, and compute the prefix normal forms of  $\mathbf{tm}$  as  $\text{PNF}_1(\mathbf{tm}) = 1(10)^\omega$  and  $\text{PNF}_0(\mathbf{tm}) = 0(01)^\omega$ , see Fig. 4.8.



**Fig. 4.8:** The Thue-Morse word (dashed) and its prefix normal forms (solid).

For the proof of the abelian complexity of  $\mathbf{tm}$  in [115], the Parikh vectors were computed for each length, so we do not really need Lemma 4.25 but could have got the prefix normal forms directly. Moreover, a much more general result was given in [115]:

**Theorem 4.9 ([115]).** *Let  $w$  be an aperiodic infinite binary word. Then  $\psi_w = \psi_{\mathbf{tm}}$  if and only if  $w = \mu_{\text{TM}}(w')$  or  $w = 0\mu_{\text{TM}}(w')$  or  $w = 1\mu_{\text{TM}}(w')$  for some word  $w'$ .*

The abelian complexity function does not in general determine the prefix normal forms, as can be seen on the example of Sturmian words, which all have the same abelian complexity function but different prefix normal forms. However,  $\psi_{\mathbf{tm}}$  does, due to its values  $\psi_{\mathbf{tm}}(n) = 2$  for  $n$  odd and  $= 3$  for  $n$  even, and to the fact that both  $F_{\mathbf{tm}}^1$  and  $F_{\mathbf{tm}}^0$  have difference function with values from  $\{0, 1\}$ : notice that the only pair of such functions with width 2 resp. 3 are the PNFs of  $\mathbf{tm}$ . Therefore, we can deduce the following from Theorem 4.9:

**Corollary 4.5.** *For an aperiodic infinite binary word  $w$ ,  $\text{PNF}_1(w) = 1(10)^\omega$  and  $\text{PNF}_0(w) = 0(01)^\omega$  if and only if  $w = \mu_{\text{TM}}(w')$  or  $w = 0\mu_{\text{TM}}(w')$  or  $w = 1\mu_{\text{TM}}(w')$  for some word  $w'$ .*

To conclude this section, we return to the question of how many 1s need to be prepended to make the Thue-Morse word prefix normal.

**Lemma 4.26.** *We have  $11\mathbf{tm} \in \mathcal{L}$ . Moreover, this is minimal since  $1\mathbf{tm}$  is not prefix normal.*

*Proof.* We will show that for every prefix, the number of 1s in the prefix of  $11\mathbf{tm}$  is greater than or equal to the the number of 1s in the prefix of  $\text{PNF}_1(\mathbf{tm})$  of the same length. Let  $v = \text{PNF}_1(\mathbf{tm})$  and  $u = 11\mathbf{tm}$ . It is easy to see that  $P_v(n) = \lfloor \frac{n}{2} \rfloor + 1$  and

$$P_u(n) = \begin{cases} \frac{n}{2} + 1 & \text{if } n \text{ is even} \\ \lfloor \frac{n}{2} \rfloor + 2 & \text{if } n \text{ is odd and } u_n = 1 \\ \lfloor \frac{n}{2} \rfloor + 1 & \text{if } n \text{ is odd and } u_n = 0 \end{cases}$$

Thus for all  $n \geq 1$  it holds that  $P_u(n) \geq P_v(n)$ , implying that  $11\mathbf{tm} \in \mathcal{L}$ .

For minimality, note that  $1\mathbf{tm}$  is not prefix normal, since  $11$  is a factor of  $\mathbf{tm}$ .  $\square$

### 4.6.3 Prefix normal forms of Sturmian words.

Let  $w$  be a Sturmian word. As we saw in Sec. 4.4, the only 1-prefix normal word in the class of Sturmian words with the same slope  $\alpha$  is the upper mechanical word  $s'_{\alpha,0} = 1c_\alpha$ .

**Theorem 4.10.** *Let  $w$  be an irrational mechanical word with slope  $\alpha$ , i.e. a Sturmian word. Then  $\text{PNF}_1(w) = 1c_\alpha$  and  $\text{PNF}_0(w) = 0c_\alpha$ , where  $c_\alpha$  is the characteristic word of slope  $\alpha$ .*

*Proof.* Since the characteristic word  $c_\alpha$  has the same slope as  $w$ , we have  $Fct(w) = Fct(c_\alpha)$  by Fact 4.2. The abelian complexity of  $w$  is constant 2 [115], thus a factor of length  $k$  can have either  $F_w^1(k)$  or  $F_w^1(k) - 1$  1s. Let us call a factor  $u$  of  $w$  *heavy* if  $|u|_1 = F_w^1(k)$ , and *light* otherwise. We have to show that every prefix of  $1c_\alpha$  is heavy. It is known [92] that the prefixes of the characteristic word are precisely the reverses of its right special factors, where a factor  $u$  is called right special if both  $u0$  and  $u1$  are factors. Thus, every prefix  $v$  of  $1c_\alpha$  has the form  $v = 1u^{\text{rev}}$ , where both  $u1$  and  $u0$  are factors of  $w$ , therefore  $v = 1u^{\text{rev}}$  is heavy. The fact that  $\text{PNF}_0(w) = 0c_\alpha$  follows analogously.  $\square$

### 4.6.4 Prefix normal forms of binary uniform morphisms

In [19] the authors provide an algorithm that computes the abelian complexity of a morphic word that is the fix point of a binary uniform morphism. — A binary *uniform* morphism  $\mu$  is a function  $\mu : \{0, 1\} \rightarrow \{0, 1\}^*$  such that  $|\mu(0)| = |\mu(1)|$ . — We now briefly summarize the main idea of the algorithm. Refer to [19] for all the details.

Given a binary uniform morphism  $\mu$  of length  $\ell = |\mu(0)| = |\mu(1)|$ , the algorithm stores two tables, with the following information: In the first table, for all  $a, b \in \{0, 1\}$ , we store the *border table*  $ab$ . This table contains, for all  $c, d \in \{0, 1\}$ , the minimum number of 0s in a factor  $s$  of length  $k = 2, \dots, 2\ell$  such that  $s = uv$ , where  $u$  is a non-empty prefix of  $\mu(c)$  starting with character  $a$  and  $v$  is a non-empty suffix of  $\mu(d)$  starting with character  $b$ , if such  $s$  exists or the entry is left blank. In the second table, for all  $a, b \in \{0, 1\}$ , we store the minimum number of 0s in a factor  $s \in Fct(\mu^\omega(0))$  of length  $k \geq 2$  such that  $s$  has  $a$  as first character and  $b$  as last character.

In order to compute the second table, we can compute the first  $\ell$  entries via a brute-force algorithm over the images of  $\mu(ab)$ , for all  $a, b \in \{0, 1\}$ , since if a factor intersects the images of three letters, then this factor has length greater than  $\ell$ . In order

to fill the rest of the second table we can consider the following. Let  $v$  be the factor of length  $n > \ell$  that has the minimum number of 0s and has  $a$  as first character and  $b$  as last character, for some  $a, b \in \{0, 1\}$ . We can write  $v$  as a factor of an image of a shorter factor  $s$  in  $\mu^\omega(0)$ , in particular  $s = cud$  where  $c, d \in \{0, 1\}$  and  $u$  is a possible empty factor. Thus, we can write  $v = x\mu(u)y$  where  $x$  is a non-empty suffix of  $\mu(c)$  starting with a character  $a$  while  $y$  is a non-empty prefix of  $\mu(d)$  with  $b$  as last character. We can assume without loss of generality that  $|\mu(0)|_0 \geq |\mu(1)|_0$ , then in order to get the minimum number of 0s in  $v$  we have to find all the possible lengths  $k \equiv n \pmod{\ell}$  such that  $|x| + |y| = k$ . Then we can look up into the border table  $ab$ , for every possible  $c, d \in \{0, 1\}$ , in correspondence of row  $k$  we can get the minimum number of 0s for  $x$  and  $y$ . In order to get the minimum number of 0s in  $\mu(u)$  we look up into the second table to the minimum number of 0s of a factor of length  $m = (n - k)/\ell + 2$  that starts and ends with  $c$  and  $d$  respectively. Keeping all together we get the minimum number of 0s for the factor  $v$  of length  $n$  that starts with an  $a$  and ends with a  $b$ .

This algorithm can be easily modified in order to compute the minimum number of 1s in a factor of a given length. Thus the algorithm explicitly computes the  $F_w^0(i)$  and  $F_w^1(i)$  for all  $i = 1, 2, \dots, n$ , from which we can compute the prefix normal forms of the morphic word in  $O(n)$  time, leading to the following lemma.

**Lemma 4.27.** *Given a binary uniform morphism  $\varphi$ , let  $w$  be the fixed point of  $\varphi$ . The prefix of length  $n$  of  $\text{PNF}_w(n)$  can be computed in  $O(n)$  time using [19, Algorithm 1]*

*Example 11.* Let us consider the following morphism  $\mu(0) = 0101$  and  $\mu(1) = 1100$  where first characters of  $\mu^\omega(0)$  are  $\mu^\omega(0) = 0100110001000100\dots$ . In Table ?? we show the border tables computed for  $ab \in \{00, 01, 10, 11\}$ .

$k$	border table 00				border table 01				border table 10				border table 11			
	00	01	10	11	00	01	10	11	00	01	10	11	00	01	10	11
2			2					1								0
3	2		3			1	2	1						1	0	
4			3	2	2	1	3	2	2	1	3			1		2
5	3	2	4	3			2	3			2	3		2	1	3
6		3		4	3	2	4		3	2	4	3				3
7	4	3								3	4	3		3		4
8		4			4							4				4

**Table 4.4:** The border tables of the morphism  $\mu(0) = 0101$  and  $\mu(1) = 1100$ .

As an example, in border table 11, row 2 the only non empty element corresponds to column 01, since the factor 11 is the factor composed by the last character of the image  $\mu(0)$  and the first character of the image  $\mu(1)$ . In Table ?? we report few entries of the second table.

In order to compute the value of the last row of column 10, we proceed as follows: we have that  $n = 11$  and since  $\ell = 4$ , then the possible values of  $k$  such that  $k \equiv n \pmod{\ell}$  are  $k \in \{3, 7\}$ . For  $k = 3$  we have that the border table 10 has 3rd row blank, thus there are no factors of length 11 that start with a 1 and end with a 0. Then the only case we have to consider is  $k = 7$ . In the border table 10, for,  $k = 7$ , we have that the only possible values of  $cd$  are  $\{01, 10, 11\}$ . For  $cd = 11$  we look up row 7 of the border table 10 at column 11 and we get that the minimum number of 0s is 3. From the second table, we know that the minimum number of 0s in a factor of length

$n$	list 00	list 01	list 10	list 11
2	2	1	1	0
3	2	1	1	0
4	2	1	1	1
5	2	2	2	1
6	3	2	2	2
7	3	3	3	2
8	4	3	3	3
9	4	4	4	3
10	5	4	4	4
11	5	5	5	4

**Table 4.5:** Each column list  $ab$  reports the minimum number of 0's in a factor of length  $n$  starting with  $a$  and ending with  $b$ , of the morphism  $\mu(0) = 0101$  and  $\mu(1) = 1100$ .

$m = (11 - 7)/4 + 2 = 3$  starting and ending with a 1 is 1. Since  $|\mu(0)|_0 = 2$  we have that the minimum number of 0s in a factor of length 11 starting with a 1 and ending with a 0 is 5.

Fix  $n = 11$ , the minimum value in row  $n$  is the minimum number of 0's in a factor of length 11 of the fix point of  $\mu$ , i.e. for  $n = 11$  the minimum vale is 4. Thus, we have that the maximum number of 1's in a factor of length 11 is  $11 - 4 = 7 = F_w^1(11)$ . The first values of the  $F_w^1$  function are 1, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, . . . , from which we can compute the first values of  $\text{PNF}_1(w) = 11101010101 \dots$ .

### 4.7 A characterization of periodic and aperiodic prefix normal words with respect to minimum density

In this section, we provide a characterization of periodicity and aperiodicity of prefix normal words with respect to their minimum density. The following result shows that every ultimately periodic infinite prefix normal word has rational minimum density.

**Lemma 4.28.** *Let  $v$  be an infinite ultimately periodic binary word with minimum density  $\delta(v) = \alpha$ . Then  $\alpha \in \mathbb{Q}$ .*

*Proof.* Let us write  $v = ux^\omega$  with  $x$  not a suffix of  $u$ .

For  $i = 0, 1, \dots, |x| - 1$ , let  $y_i$  be the prefix of length  $|u| + i$  of  $v$ , i.e.,  $y_i = ux_1x_2 \dots x_i$ . Trivially, if for some  $i$  we have that  $\delta(y_i) \leq \delta(v)$  the claim directly follows from  $y_i$  being a finite prefix of  $v$ .

Let us now assume that for each  $i = 0, 1, \dots, |x| - 1$  it holds that  $\delta(v) < \delta(y_i)$  and let  $i^* = \min\{i \mid \delta(y_i) \leq \delta(y_j) \text{ for each } j \neq i\}$ , hence  $\delta(v) < \delta(y_{i^*})$ .

For every  $n \geq |u| + |x|$  let  $i_n = |u| + ((n - |u|) \bmod |x|)$  and  $k_n = \lfloor (n - |u|) / |x| \rfloor$ , i.e.,  $|u| \leq i_n \leq |u| + |x| - 1$  and  $n = i_n + k_n|x|$ .

Then, we have that

$$D_v(n) = \frac{|y_{i_n}|_1 + k_n|x|_1}{|y_{i_n}| + k_n|x|} \geq \min\{\delta(y_{i_n}), \delta(x)\} \geq \min\{\delta(y_{i^*}), \delta(x)\}. \quad (4.2)$$

Moreover, we also have that

$$\lim_{k \rightarrow \infty} D_v(|u| + i^* + k|x|) = \lim_{k \rightarrow \infty} \frac{|y_{i^*}|_1 + k|x|_1}{|y_{i^*}| + k|x|} = \delta(x). \quad (4.3)$$

We cannot have  $\delta(x) \geq \delta(y_{i^*})$ , since by (4.2)  $\delta(y_{i^*})$  is a rational lower bound on  $D_v(n)$  (for each  $n \geq 1$ ) which is achieved by  $D_v(|u| + i^*)$ , contradicting the standing hypothesis  $\delta(v) < \delta(y_{i^*})$ .

Therefore, we must have  $\delta(x) < \delta(y_{i^*})$ , and from (4.2) we have  $D_v(n) \geq \delta(x)$  and from (4.3) we also have that for each  $\varepsilon > 0$  there exists  $k > 0$  such that  $D_v(|u| + i^* + k|x|) < \delta(x) + \varepsilon$ . Therefore,  $\delta(v) = \inf\{D_v(n) \mid n \geq 1\} = \delta(x)$ , which is a rational number, since  $x$  is a finite string.  $\square$

We now show that, while periodicity is characterized by rational density the converse is not true. It turns out that for every  $\alpha \in (0, 1)$ , both rational and irrational, there exists an aperiodic prefix normal word with minimum density  $\alpha$ .

**Lemma 4.29.** *Fix  $\alpha \in (0, 1)$ , and let  $(a_n)_{n \in \mathbb{N}}$  be a strictly decreasing infinite sequence of rational numbers from  $(0, 1)$  converging to  $\alpha$ . For each  $i = 1, 2, \dots$ , let the binary word  $v^{(i)}$  be defined by*

$$v^{(i)} = \begin{cases} 1^{\lceil 10a_1 \rceil} 0^{10 - \lceil 10a_1 \rceil} & i = 1 \\ \text{pref}_{\text{flipext}^\omega(v^{(i-1)})}(k_i |v^{(i-1)}|) 0^{\ell_i} & i > 1 \end{cases}$$

where  $\ell_i$  is defined by

$$\ell_i = \begin{cases} 10 - \lceil 10a_1 \rceil & i = 1 \\ \left\lfloor k_i \left( \frac{|v^{(i-1)}|_1 - a_i |v^{(i-1)}|}{a_i} \right) \right\rfloor & i > 1, \end{cases}$$

and  $k_i$  is the smallest integer greater than one such that  $\ell_i > \ell_{i-1}$ .

Then  $v = \lim_{i \rightarrow \infty} v^{(i)}$  is an aperiodic infinite prefix normal word such that  $\delta(v) = \alpha$ .

*Proof.* The statement is a direct consequence of the following claim.

*Claim* The following properties hold

1.  $\delta(v^{(i)}) \geq a_i$  for each  $i \geq 1$ ;
2.  $\iota(v^{(i)}) = |v^{(i)}|$  for each  $i \geq 1$ ;
3.  $\delta(v^{(i)}) < \delta(v^{(i-1)})$  for each  $i \geq 2$ ;
4.  $|v^{(i)}|_1 > |v^{(i-1)}|_1$  for each  $i \geq 2$ ;
5.  $\delta(v^{(i)}) \leq a_i \left( \frac{k_i |v^{(i-1)}|_1}{k_i |v^{(i-1)}|_1 - a_i} \right)$  for each  $i \geq 2$ .

*Proof of the claim.* By direct inspection we have that properties 1 and 2 hold for  $v^{(1)}$ . We now argue by induction. Fix  $i > 1$  and let us assume that properties 1 and 2 hold for  $v^{(i-1)}$ . Then, since  $a_i < a_{i-1}$  we have

$$\frac{|v^{(i-1)}|_1}{a_i} > \frac{|v^{(i-1)}|_1}{a_{i-1}} \geq |v^{(i-1)}|,$$

where the last inequality follows from property 1 and 2. Therefore,  $\left( \frac{|v^{(i-1)}|_1 - a_i |v^{(i-1)}|}{a_i} \right) > 0$ , hence there exists  $k_i > 1$  such that  $\left\lfloor k_i \left( \frac{|v^{(i-1)}|_1 - a_i |v^{(i-1)}|}{a_i} \right) \right\rfloor > \ell_{i-1}$ . In particular,  $\ell_i$  is well defined.

By property 2, we have  $\iota(v^{(i-1)}) = |v^{(i-1)}|$  hence by Proposition 4.14, we have  $D_{\text{flipext}^\omega(v^{(i-1)})}(k_i |v^{(i-1)}|) = \delta(v^{(i-1)})$  and also  $\delta(\text{pref}_{\text{flipext}^\omega(v^{(i-1)})}(k_i |v^{(i-1)}|)) = \delta(v^{(i-1)})$ .



Moreover, since  $\ell_i > 0$  it is not hard to see from the definition of  $v^{(i)}$  that

$$\delta(v^{(i)}) = D_{v^{(i)}}(|v^{(i)}|) = \frac{k_i |v^{(i-1)}|_1}{k_i |v^{(i-1)}|_1 + \ell_i} < \delta(v^{(i-1)}), \quad (4.4)$$

which shows that property 3 and property 2 hold for  $v^{(i)}$ . In addition, because of  $k_i > 1$  and (by Proposition 4.14)

$$|v^{(i)}|_1 = |\text{pref}_{\text{flipext}^\omega(v^{(i-1)})}(k_i |v^{(i-1)}|)|_1 = k_i |v^{(i-1)}|_1$$

it follows that property 4 also holds for  $v^{(i)}$ .

The definition of  $\ell_i$  together with the well known property  $x - 1 < \lfloor x \rfloor \leq x$  imply that

$$\frac{k_i}{a_i} \left( |v^{(i-1)}|_1 - a_i |v^{(i-1)}| \right) - 1 < \ell_i \leq k_i \left( \frac{|v^{(i-1)}|_1}{a_i} - |v^{(i-1)}| \right). \quad (4.5)$$

Using the right inequality of (4.5) in (4.4) we have  $\delta(v^{(i)}) \geq a_i$  showing that property 1 holds for  $v^{(i)}$ .

In addition, using the left inequality of (4.5) in (4.4) we have

$$\delta(v^{(i)}) \leq a_i \left( \frac{k_i |v^{(i-1)}|_1}{k_i |v^{(i-1)}|_1 - a_i} \right)$$

showing that property 5 holds for  $v^{(i)}$ . The proof of the claim is complete.

In order to see that  $v$  is aperiodic, it is enough to observe that  $v \neq 0^\omega$  and for each  $i \geq 1$  it contains a distinct run of  $\ell_i$  0s, with  $\ell_i$  being a strictly increasing sequence.

In order to show that  $\delta(v) = \alpha$ , we will prove that  $\lim_{i \rightarrow \infty} \delta(v^{(i)}) = \alpha$ .

Since,  $\lim_{i \rightarrow \infty} a_i = \alpha$  and for each  $i \geq 1$ ,  $k_i > 1$  and  $|v^{(i)}|_1 > |v^{(i-1)}|_1$ , we have that

$$\lim_{i \rightarrow \infty} a_i \frac{k_i |v^{(i-1)}|_1}{k_i |v^{(i-1)}|_1 - a_i} = \lim_{i \rightarrow \infty} a_i = \alpha.$$

Hence, from properties 4 and 5 of the Claim above, we have the desired result  $\lim_{i \rightarrow \infty} \delta(v^{(i)}) = \lim_{i \rightarrow \infty} a_i = \alpha$ .

Summarizing, we have shown the following result.

**Theorem 4.11.** *For every  $\alpha \in (0, 1)$  (rational or irrational) there is an infinite aperiodic prefix normal word of minimum density  $\alpha$ . On the other hand, for every ultimately periodic infinite prefix normal word  $w$  the minimum density  $\delta(w)$  is a rational number.*



---

## Pattern discovery in colored strings

This chapter is devoted to colored strings.

In recent years, embedded systems have become increasingly pervasive and are becoming fundamental components of everyday life. In line with this, embedded systems are required to perform more and more demanding tasks, and in many circumstances, peoples' lives are now dependent on the correct functioning of these devices. This, in turn, has led to an increasingly complex design process for embedded systems, where a major design task is to evaluate and check the correctness of the functionality from the early stages of the development process. This functionality checking is usually done using *assertions* — logic formulae expressed in temporal logic such as Linear Temporal Logic (LTL) or Computation Tree Logic (CTL) — that provide a way to express desirable properties of the device. Assertions are typically written by hand by the designers and it might take months to obtain a set of assertions that is small and effective (i.e. it covers all functionalities of the device) [63]. In order to help designers with the verification process, methodologies and tools have been developed which automatically generate assertions from simulation traces of an implementation of the device [50, 51, 91, 133]. The objective is to provide a small set of assertions that cover all behaviors of the device, in order to extend the basic manually-defined set of assertions.

A simulation trace can be viewed as a table, which records, for every simulation instant  $T$ , the value assumed by the input and output ports of the device. Figure 5.1a shows an example of a simulation trace of a device with three input ports  $\mathcal{I} = \{i_1, i_2, i_3\}$  and two output ports  $\mathcal{O} = \{o_1, o_2\}$ . An assertion is a logic formula expressed in temporal logic that must remain true in the whole trace. The simplest assertions involve only conditions occurring at the same simulation instant. In the simulation trace in Figure 5.1a, from the solid and dashed shaded boxes, we can assert that each time we have  $i_1 = 1$ ,  $i_2 = 0$ , and  $i_3 = 1$ , then  $o_1 = 1$  and  $o_2 = 1$ . On the other hand, we cannot assert that each time we have  $i_1 = 1$ ,  $i_2 = 1$ , and  $i_3 = 0$ , then  $o_1 = 1$  and  $o_2 = 1$ , because there is a counterexample in the simulation trace, namely at instant  $T = 9$ , where  $o_1 = 0$  and  $o_2 = 0$ . Note that the assertions do not need to contain all input and output variables, e.g. we can assert that  $i_1 = 0$  and  $i_3 = 0$  implies  $o_2 = 0$ .

Among all possible types of assertions that can be expressed in temporal logic, an interesting one is given by chains of *next*: sequences of consecutive input values that, when provided to the device, uniquely determine their output, after a certain number of simulation instants. For example, in the simulation trace in Figure 5.1a, we can

(a) Simulation trace.

$T$	$i_1$	$i_2$	$i_3$	$o_1$	$o_2$
1	0	1	0	0	0
2	1	1	0	1	0
3	0	1	0	0	0
4	1	1	0	1	1
5	0	1	0	0	0
6	1	1	0	1	0
7	1	0	1	1	1
8	0	1	0	1	0
9	1	1	0	0	0
10	0	1	0	0	0
11	1	0	1	1	1

(b) Mapping of the input and output alphabet.

Input alphabet.				Output alphabet.		
$i_1$	$i_2$	$i_3$	$\Sigma$	$o_1$	$o_2$	$\Gamma$
0	1	0	a	0	0	$x$
1	0	1	b	1	0	$y$
1	1	0	c	1	1	$z$

(c) The colored string associated with the simulation trace.

$x$	$y$	$x$	$z$	$x$	$y$	$z$	$y$	$x$	$x$	$z$
a	c	a	c	a	c	b	a	c	a	b
1	2	3	4	5	6	7	8	9	10	11

**Fig. 5.1:** Example of a simulation trace of a device having three input ports  $\mathcal{I} = \{i_1, i_2, i_3\}$ , and two output ports  $\mathcal{O} = \{o_1, o_2\}$ . The mapping of the input and output values of the trace into the input and output alphabet respectively. The colored string associated to the simulation trace after the mapping. The solid and dashed shaded and non-shaded boxed values in the simulation trace highlight that every time we see the sequence of input values, then we have the corresponding output value. The solid non-shaded boxed characters in the colored string are the mapping of the corresponding solid non-shaded boxed values in the simulation trace.

assert that each time we have, for  $(i_1, i_2, i_3)$ , the values  $(0, 1, 0)$ ,  $(1, 1, 0)$ ,  $(0, 1, 0)$  in consecutive simulation instants, then, three instants later, we will see  $o_1 = 1$  and  $o_2 = 0$ .

We model simulation traces with *colored strings*. A colored string is a string over an alphabet  $\Sigma$ , where each position is additionally assigned a color from an alphabet  $\Gamma$ . We will set  $\Sigma$  as the set of tuples of possible values for the input ports  $i_1, \dots, i_k$  and  $\Gamma$  as that of the output traces  $o_1, \dots, o_r$ . The objective then is to identify patterns in the string whose occurrence is always followed by the same color at some given distance.

## Related Work

Pattern mining was originally motivated by the need to discover frequent itemsets and association rules in basket data, i.e. items that were frequently bought together in a retail store. The seminal *Apriori algorithm* [1] can discover that type of pattern and has become very popular (with many extensions and variations) due to its wide applicability in other data-intensive domains. However, the original pattern mining systems did not consider time relations, e.g., between entries of the database in which the basket data are stored. This has motivated the study of so-called *sequential pattern mining* [2].

In sequential pattern mining, *episodes* are partially ordered sequences of events that appears close to each other in the sequence [96]. Events in episodes may be in a dependency relation, hence episodes are represented as directed acyclic graphs. Given episodes of the sequence, it is possible to build *episode rules* that establish antecedent-consequent relations among episodes. These relations can predict the future events of the sequence, thus useful to understand the behavior of the sequence itself. This finds application in several tasks, e.g., traffic jam prediction [30], fault prediction in manufacturing plants [90], and bank-customer trends prediction [55]. For a complete overview on sequential pattern mining we refer the reader to some of the many surveys in that area [64, 93, 107].

Unfortunately, the above setting is not applicable to our problem, since here time is given only in a relative sense, i.e., whether an event happens before (of after) another event, while we need to count exactly the instants that occurs between the two events.

In the *string mining problem* [54, 59, 60, 61, 132], one aims to discover strings that appear as a substring in more than  $\omega$  strings in a collection, where  $\omega$  is a user-defined parameter called *support* of the string. This can be also used to find strings that discriminate between two collections, i.e., strings that are frequent in one collection and not frequent in the other. These strings are called *emerging strings* and find important applications in data mining [113], knowledge discovery in databases [28] and in bioinformatics [17]. In the knowledge discovery on databases field, the problem has been extended to mining frequent subsequences [74] and distinguishing subsequence patterns with gap constraints [75, 104, 135, 138].

In [72] Hui proposed a solution for the *color set size problem*. Here, given a tree and a coloring of its leaves, the objective is to find for all internal nodes of the tree the number of distinct colors in the leaves of its subtree. In the paper, the color set size problem is applied to several string matching and string mining problems, e.g., given a collection of  $m$  strings, find the longest pattern which appears in at least  $1 \leq k \leq m$  strings. Note that if the tree of the color set size problem is the suffix tree of a string  $s$ , then  $s$  with the coloring of its suffixes can be seen as a colored string. In spite of this similarity, both, the problems that we solve, and the approaches we use, are different.

In assertion mining, the two existing tools, *GoldMine* [133] and *A-Team* [50], are based on data mining algorithms. In particular, *GoldMine* [133] extracts assertions that predicate only on one instant of the simulation trace—i.e. they do not involve any notion of time—, using decision tree based mining or association mining [1]. Furthermore, using static analysis techniques together with sequential pattern mining, it extracts temporal assertions. The tool *A-Team* [50], requires the user to provide the template of the temporal assertions that they want to extract. For example, in order to extract the properties of our example in Fig. 5.1a, one needs to provide a template stating that we want a property of the form: “a property  $p_1$ , at the next simulation instant a property  $p_2$ , at the next simulation instant a property  $p_3$ , then after three simulation instants a property  $p_4$ ”. Given a set of templates, the software, using an Apriori algorithm, extracts propositions (logic formulae containing the logical connectives  $\neg$ ,  $\vee$ , and  $\wedge$ ) from the trace. Once the propositions have been extracted, the tool generates the assertion by instantiating the extracted propositions in the templates, using a decision-tree-based algorithm to find formulas that fit in the template and are verified in the simulation trace, i.e. if the trace contains no counterexample.

### Our contribution

In this work we introduce colored strings, and propose and analyze two pattern discovery problems on colored strings which correspond to simplified pattern mining tasks w.r.t. the assertion mining problem. In both problems, we are given a colored string as input. In the first problem, given a particular color, we want to find all minimal substrings which occur followed always at a the same distance by the given color. In the second problem, the color is not fixed, i.e. we want to find all minimal substrings which occur followed always at a the same distance by the same color. We give formal definitions of these problems in Section 5.1.1.

Although these problems are simpler than the original assertion mining problem, the solution to our problem contains all the information, possibly filtered, to recover the desired set of minimal assertions in a second stage. For example, let us assume that the device that produced the simulation trace in Figure 5.1a has a behavior such that every time that  $i_1 = 0$ , and at the next instant  $i_1 = 1$ , and at the next instant  $i_1 = 0$ , then after three instants  $o_1 = 1$  and  $o_2 = 0$ . A solution to our problem will include all patterns of length 3 for which  $i_1 = 0, 1, 0$ , while  $i_2$  and  $i_3$  have arbitrary values, since all of these will result in  $o_1 = 1$  and  $o_2 = 0$  three instances later.

We first upper bound the number of minimal patterns by  $\mathcal{O}(n^2)$ . We then propose two algorithms which find all minimal patterns, when only one color is of interest (`base`), and when one is interested in all colors (`base-all`). Both of these algorithms use the suffix tree of the reverse string as underlying data structure. We note that since this is a pattern mining problem, every efficient algorithm for the problem will necessarily use a dedicated string data structure (or index), such as a suffix tree, since all occurrences of substrings have to be considered concurrently.

Then we show that in the case of one color, the first algorithm can be improved. The new algorithm, referred to as `skipping`, also uses the suffix tree as its underlying data structure, together with an appropriately defined priority queue, which allows to reduce the number of computations in practice, even though the theoretical running time of the new algorithm is worse, namely  $\mathcal{O}(n^2 \log n)$ . We provide an experimental evaluation of the proposed approaches. Finally, we consider the case where there are restrictions on the patterns that have to be reported. If these restrictions are considered as part of the problem, we can provide some optimizations that further speed up the computation of the `skipping` algorithm.

The rest of the chapter is structured as follows. In Section 5.1 we fix definitions and notations, and give the problem statements. In Section 5.2 we present baseline algorithms, `base` and `base-all`, that solve these problems. In Section 5.3 we present the modified algorithm `skipping`, which solves the pattern discovery problem for only one color. In Section 5.4 we introduce real-world data oriented restrictions on the output. In Section 5.5 we present an experimental evaluation of the proposed approaches.

## 5.1 Basics

Let  $\Sigma$  be a finite ordered set. We refer to  $\Sigma$  as *alphabet* and to its elements as *characters*. A *string over  $\Sigma$*  is a finite sequence of characters  $S = S[1, n]$ , where  $|S| = n$  is the *length* of string  $S$ . We denote by  $\varepsilon$  the *empty string*, the unique string of length 0. Note that we number strings starting from 1, and we use the array-notation for strings: we denote the  $i$ 'th character of  $S$  by  $S[i]$  and use  $S[i, j]$  to refer to the string  $S[i] \cdots S[j]$ , if  $i \leq j$ , while  $S[i, j] = \varepsilon$  if  $i > j$ . Given string  $S = S[1, n]$ , the *reverse string* is the string  $S^{\text{rev}} = S[n]S[n-1] \cdots S[1]$ . For string  $S$  and  $1 \leq i \leq n$ ,  $\text{Pref}_i(S) = S[1, i]$  is called the  $i$ 'th *prefix* of  $S$ , and  $\text{Suf}_i(S) = S[i, n]$  is called the  $i$ 'th *suffix* of  $S$ . A *substring* of a string  $S$  is a string  $T$  for which there exist  $i, j$  s.t.  $T = S[i, j]$ ; in this case the position  $i$  is referred to as an *occurrence* of  $T$  in  $S$ . A substring  $T$  of  $S$  is called *proper* if  $T \neq S$ . When  $S$  is clear from the context, then we may refer to  $T$  simply as a *substring*.

### 5.1.1 Colored strings

Given two finite sets  $\Sigma$  (the alphabet) and  $\Gamma$  (the colors), a *colored string* over  $(\Sigma, \Gamma)$  is a string  $S = S[1, n]$  over  $\Sigma$  together with a coloring function  $f_S : \{1, \dots, n\} \rightarrow \Gamma$ . We denote by  $\sigma = |\Sigma|$  and  $\gamma = |\Gamma|$  the number of characters resp. of colors. Given a colored string  $S$  of length  $n$ , its reverse is denoted  $S^{\text{rev}}$ , and its coloring function  $f_{S^{\text{rev}}}$  is defined by  $f_{S^{\text{rev}}}(i) = f_S(n - i + 1)$ , for  $i = 1, \dots, n$ . When  $S$  is clear from the context, we write  $f$  for  $f_S$  and  $f^{\text{rev}}$  for  $f_{S^{\text{rev}}}$ .

We are interested in those substrings which are always followed by a given color  $y$ , at a given distance  $d$ . Look at the following example.

*Example 12.* Let  $S = acacacbacab$ , with colors  $xyxzxyzyxxz$ :

$x$	$y$	$x$	$z$	$x$	$y$	$z$	$y$	$x$	$x$	$z$
$a$	$c$	$a$	$c$	$a$	$c$	$b$	$a$	$c$	$a$	$b$
$1$	$2$	$3$	$4$	$5$	$6$	$7$	$8$	$9$	$10$	$11$

The substring  $aca$  occurs 3 times in  $S$ , at positions 1, 3, and 8. In positions 1 and 3 it is followed by a  $y$  at distance 3, while at position 8, the corresponding position is beyond the end of the string.

This leads to the following definitions:

**Definition 5.1** (*y-good, y-unique, minimal*). Let  $S$  be a colored string over  $(\Sigma, \Gamma)$ ,  $y \in \Gamma$  a color,  $d \leq n$  a non-negative integer, and  $T = T[1, m]$  a substring of  $S$ .

1. An occurrence  $i$  of  $T$  is called *y-good* with delay  $d$  (or *(y, d)-good*) if  $f(i + m - 1 + d) = y$ .
2.  $T$  is called *y-unique* with delay  $d$  (or *(y, d)-unique*) if for every occurrence  $i$  of  $T$ ,  $i$  is *(y, d)-good* or  $i + m - 1 + d > n$ .
3.  $T$  is called *minimally (y, d)-unique* if there exists no proper substring  $U$  of  $T$  which is *y-unique* with delay  $d'$ , for some  $d'$  s.t.  $U = T[i, j]$  and  $d' = d + |T| - j$ .

Returning to Example 12, the occurrence of  $aca$  in position 1 is  $(y, 3)$ - and  $(y, 5)$ -good, that in position 3 is  $(y, 1)$ - and  $(y, 3)$ -good, while that in position 8 is not  $(y, d)$ -good for any  $d$ . Therefore, the substring  $T = aca$  is a  $(y, 3)$ -unique substring of  $S$ , since every occurrence  $i$  of  $aca$  is either  $(y, 3)$ -good (pos. 1 and 3) or  $i + m - 1 + d > n$  (pos. 8). However,  $aca$  is not minimal, since its substring  $ca$  is also  $(y, 3)$ -unique (and  $d' = d$ , since  $ca$  is a suffix of  $aca$ ).

The introduction of minimally  $(y, d)$ -unique substrings serves to restrict the output size. Let  $T = aXb$  be  $(y, d)$ -unique, with  $a, b \in \Sigma$  and  $X \in \Sigma^*$ . We call  $T$  *left-minimal* if  $Xb$  is not  $(y, d)$ -unique, and *right-minimal* if  $aX$  is not  $(y, d + 1)$ -unique. We make the following simple observations about  $(y, d)$ -unique substrings. (Note that 2 is a special case of 3.)

**Observation 5.1** Let  $S \in \Sigma^*$  and let  $T$  be a  $(y, d)$ -unique substring of  $S$ .

1.  $T$  is minimal if and only if it is left- and right-minimal.
2. If  $T$  is a suffix of  $T'$ , then  $T'$  is also  $(y, d)$ -unique.

3. If  $T' = UTV$  is a superstring of  $T$  such that  $|V| \leq d$ , then  $T'$  is  $(y, d - |V|)$ -unique.

We are now ready to formally state the problems treated in this chapter.

**Problem 1 (Pattern Discovery Problem).** Given a colored string  $S$  and a color  $y$ , report all pairs  $(T, d)$  such that  $T$  is a minimally  $(y, d)$ -unique substring of  $S$ .

**Problem 2 (Unrestricted-Output Pattern Discovery Problem).** Given a colored string  $S$ , report all triples  $(T, y, d)$  such that  $T$  is a minimally  $(y, d)$ -unique substring of  $S$ .

We next give an upper bound on the number of minimally  $(y, d)$ -unique substrings.

**Lemma 5.1.** *Given string  $S$  of length  $n$ , the number of minimally  $(y, d)$ -unique substrings of  $S$ , over all  $y \in \Gamma$  and  $d = 0, \dots, n$ , is  $\mathcal{O}(n^2)$ .*

*Proof.* Note that, given a position  $j$  and a delay  $d$ , every substring occurrence ending in  $j$  is  $(f_S(j + d), d)$ -good. Therefore, for a substring  $u$  with an occurrence ending in position  $j$ , and for fixed  $d$ , it holds that, if  $u$  is  $(y, d)$ -unique for some  $y$ , then  $y = f_S(j + d)$ . Moreover, it follows from Observation 5.1 that, given  $y, d$ , and  $j$ , at most one minimally  $(y, d)$ -unique substring can end at position  $j$ . Altogether we have that the number of minimally  $(y, d)$ -unique substrings is  $\mathcal{O}(n^2)$ , over all  $y$  and  $d$ .

### 5.1.2 Suffix trees and suffix arrays

Let  $S$  be a string over  $\Sigma$  and  $\$$  a new character not belonging to  $\Sigma$ . We denote by  $\mathcal{T}(S)$  the *suffix tree* of  $S\$$ , i.e. the compact trie of the suffixes of  $S\$$ . For a general introduction to suffix trees, see, e.g., [69, 95, 125]. Here we recall some basic facts.

The suffix tree  $\mathcal{T}(S)$  is a rooted tree in which all internal nodes are branching. Each edge is labeled with a non-empty substring of  $S$  so that the labels of any two outgoing edges from the same node start with a different character. Edge labels are stored in form of two pointers  $[i, j]$  into the string with the property that  $S[i, j]$  equals the label of the edge. If  $|S| = n$ , then  $\mathcal{T}(S)$  has exactly  $n + 1$  leaves, each labeled by a position from  $\{1, \dots, n + 1\}$ , denoted  $ln(v)$  (for leaf number). For a node  $v$  in  $\mathcal{T}(S)$ , we denote by  $L(v)$  the concatenation of the edge labels on the path from the root to node  $v$ . The string  $L(v)$  is sometimes referred to as the substring represented by node  $v$ . If  $v$  is a leaf with  $ln(v) = i$ , then  $L(v)$  is equal to the  $i$ 'th suffix of  $S\$$ ,  $\text{Suf}_i(S\$)$ . For a node  $v$ , we denote by  $td(v)$  its *treedepth*, the number of edges on the path from the root to  $v$ , and by  $sd(v) = |L(v)|$  its *stringdepth*, the length of the string represented by  $v$ . Given node  $v$  not equal to the root,  $parent(v)$  is the unique node which is next on the path from  $v$  to the root. Given a node  $v$  which is not a leaf and a character  $c \in \Sigma$ ,  $child(v, c)$  returns the unique node  $u$  with parent  $v$  such that the label of the edge  $(v, u)$  starts with character  $c$ , or the empty pointer if no such node exists.

Given a node  $u$  with parent  $v$ , a *locus* is a pair  $(u, t)$  s.t.  $sd(v) < t \leq sd(u)$ . Let  $[i, j]$  be the label of edge  $(v, u)$  and  $k = t - sd(v)$ . We define  $L(u, t)$  as the string  $L(v) \cdot S[i, i + k - 1]$ , the substring represented by locus  $(u, t)$ . Note that if  $t = sd(u)$ , then  $L(u, t) = L(u)$ . It is an important property of suffix trees that there is a one-to-one correspondence between loci of  $\mathcal{T}(S)$  and substrings of  $S\$$ . This allows us to define, for a substring  $T$  of  $S$  (which is also a substring of  $S\$$ ), the *locus of  $T$* ,  $loc(T) = loc(T, \mathcal{T}(S))$  as the unique locus  $(u, t)$  in  $\mathcal{T}(S)$  with the property that



$L(u, t) = T$ . Given a substring  $T$  of  $S$  with locus  $loc(T) = (u, t)$ , the set of occurrences of  $T$  is given by the set  $\{ln(v) \mid v \text{ is leaf in the subtree rooted in } u\}$ .

Let  $u$  be a node and  $L(u) = aT$ , where  $a \in \Sigma$  and  $T \in \Sigma^*$ . The *suffix link* of  $u$  is defined as  $slink(u) = loc(T)$ . It can be shown that for any node  $u$ ,  $slink(u)$  is a node of  $\mathcal{T}(S)$  (rather than just a locus). Suffix links can also be defined for loci: for locus  $(u, t)$  with  $L(u, t) = aT$ , define  $slink(u, t) = loc(T)$ ; these are also called *implicit suffix links*. Suffix links are often represented by directed edges, see Figure 5.2.

Given a suffix tree  $\mathcal{T}(S)$  with  $k$  nodes, and a node  $u$  of  $\mathcal{T}(S)$ , let  $r$  be the rank of the node  $u$  in the breadth-first search traversal of the tree. We define the reverse index BFS of  $u$  as  $iBFS(u) = k - r$ . Refer to Figure 5.3 for an example of the reverse index BFS values.

Given the string  $S$  of length  $n$ , we denote by  $SA_S[1, n + 1]$  the *suffix array* of  $S\$$ . We refer the reader to, e.g., [95], for a general introduction to suffix arrays.

The suffix array  $SA_S[1, n + 1]$  of a string  $S\$$  is a permutation of  $\{1, \dots, n + 1\}$  such that  $SA_S[i] = j$  if and only if  $S[j, n]\$$  is  $i$ -th suffix in the lexicographically ordered list of suffixes of  $S\$$ . The suffix array  $SA_S$  and the suffix tree  $\mathcal{T}(S)$  are deeply related. We can obtain the  $SA_S$  by listing the leaves of the suffix tree  $\mathcal{T}(S)$  from left to right — assuming that the children are ordered according to the first characters of their edge labels —. In particular, for an inner node  $u$ , the leaves in the subtree rooted in  $u$  yield an interval of the suffix array  $SA_S[i, j]$  such that  $\{ln(v) \mid v \text{ is leaf in the subtree rooted in } u\} = \{SA_S[k] \mid i \leq k \leq j\}$ .

### 5.1.3 Maximum-oriented indexed priority queue

A maximum-oriented indexed priority queue [122, Sec. 2.4] denoted by  $IPQ$ , is a data structure that collects a set of  $m$  items with keys  $k_1, \dots, k_m$  respectively, and provides the following operations:

- `insert( $i, k$ )` : insert the element at index  $i$  with key  $k_i = k$ .
- `promote( $i, k$ )` : increase the value of the key  $k_i$ , associated with  $i$ , to  $k \geq k_i$ .
- `demote( $i, k$ )` : decrease the value of the key  $k_i$ , associated with  $i$ , to  $k \leq k_i$ .
- `( $i, k$ ) ← max()` : return the index  $i$  and the value  $k$  of the item with maximum key  $k_i$ ; if two items have the same key value, we report the item with larger index.
- `$k$  ← keyOf( $i$ )` : return the value of the key  $k_i$  associated with index  $i$ .
- `$b$  ← isEmpty()` : return *true* if the  $IPQ$  is empty and *false* otherwise.
- `delete( $i$ )` : remove the element at index  $i$  from the  $IPQ$ .

The operations `insert`, `promote`, `demote` and `delete` run in  $\mathcal{O}(\log(m))$  time, while the operations `max`, `keyOf` and `isEmpty` are performed in  $\mathcal{O}(1)$  time.

For our purposes, we also require a function  `$b$  ← allNegative()` that returns *true* if all key values are negative, and *false* otherwise.

We use the  $IPQ$  to store keys associated to nodes  $u$  of a suffix tree  $\mathcal{T}(S)$  using  $iBFS(u)$  as index. For ease of presentation, in slight abuse of notation, we will use  $u$  and  $iBFS(u)$  interchangeably.

### 5.1.4 Rank, select, and range maximum query

A *bitvector*  $B[1, n]$  of length  $n$  is an array of  $n$  bits. For all  $1 \leq i \leq n$  and  $b \in \{0, 1\}$ , we define  $\text{rank}_b(B, i)$  as the number of occurrences of  $b$  in  $B[1, i]$ , and  $\text{select}_b(B, i)$

as the index of the  $i$ -th occurrence of the symbol  $b$  in  $B$ . If  $i > \text{rank}_b(B, n)$  then  $\text{select}_b(B, i) = n + 1$ . Furthermore, we set  $\text{select}_b(B, 0) = 0$ . For both rank and select operations, if  $b$  is omitted we assume  $b = 1$ . Given a bitvector  $B$ , rank and select operations can be supported in  $\mathcal{O}(1)$  time using  $o(n)$  bits of extra space [44].

For an array  $A[1, n]$  of  $n$  integers and  $1 \leq i \leq j \leq n$ , a *range maximum query*  $\text{rMq}_A(i, j)$  returns the position of the maximum element of  $A[i, j]$ . This answer can be provided in  $\mathcal{O}(1)$  time using  $2n + o(n)$  bits of space [58].

Given  $A[1, n]$  and the range maximum query data structure for  $A$ , we can compute the position of the second greatest element of  $A[i, j]$  in  $\mathcal{O}(1)$  time. In particular, let  $a = \text{rMq}_A[i, j]$ , we have three cases: (i) if  $a = i$ , then the position of the second greatest element of  $A[i, j]$  is  $c = \text{rMq}_A[a + 1, j]$ ; (ii) if  $a = j$ , then the position of the second greatest element of  $A[i, j]$  is  $b = \text{rMq}_A[i, a - 1]$ ; (iii) otherwise, let  $b = \text{rMq}_A[i, a - 1]$ , and  $c = \text{rMq}_A[a + 1, j]$ . The position of the second greatest element of  $A[i, j]$  is  $b$  if  $A[b] \geq A[c]$ , otherwise it is  $c$ , since  $A[c] > A[b]$ .

## 5.2 A pattern discovery algorithm for colored strings using the suffix tree

Our main tool will be the suffix tree of the reverse string,  $\mathcal{T} = \mathcal{T}(S^{\text{rev}})$ . Note that loci in  $\mathcal{T}$  correspond to ending positions of substrings of  $S$  in the following sense. Given a locus  $(u, t)$  of  $\mathcal{T}$ , let  $U = L(u, t)^{\text{rev}}$ . Then  $U$  is a substring of  $S$ , and its occurrences are exactly the positions  $i - |S| + 1$ , where  $i = n - \ln(v) + 1$  for some leaf  $v$  in the subtree rooted in  $u$ . In the next lemma we show how to identify  $(y, d)$ -unique substrings of  $S$  with  $\mathcal{T}$ , the suffix tree of  $S^{\text{rev}}$ .

**Lemma 5.2.** *Let  $U$  be a substring of  $S$ ,  $\mathcal{T} = \mathcal{T}(S^{\text{rev}})$ , and  $(u, t) = \text{loc}(U^{\text{rev}}, \mathcal{T})$ . Then  $U$  is  $y$ -unique with delay  $d$  in  $S$  if and only if for all leaves  $v$  in the subtree rooted in  $u$ ,  $S^{\text{rev}}[\ln(v) - d]$  is colored  $y$  under  $f^{\text{rev}}$ . In particular,  $U$  is  $y$ -unique with delay 0 in  $S$  if and only if all leaves in the subtree rooted in  $u$  are colored  $y$  under  $f^{\text{rev}}$ .*

*Proof.* It is easy to see that position  $i - |U| + 1$  is a  $y$ -good occurrence of  $U$  in  $S$  with delay 0 if and only if  $U^{\text{rev}}$  is a prefix of  $\text{Suf}_{n-i+1}(S^{\text{rev}})$  and  $f^{\text{rev}}(n - i + 1) = y$ . By the properties of the suffix tree, all occurrences of  $U^{\text{rev}}$  correspond to the leaves of the subtree rooted in  $u$ , where  $(u, t) = \text{loc}(U^{\text{rev}}, \mathcal{T})$ . Thus,  $U$  is  $(y, 0)$ -unique if and only if all of its occurrences are  $(y, 0)$ -good, which is the case if and only if all leaves of the subtree rooted in  $u$  are colored  $y$  under  $f^{\text{rev}}$ . More generally, position  $i - |U| + 1$  is a  $y$ -good occurrence of  $U$  in  $S$  with delay  $d$  if and only if  $\text{Suf}_{n-i+1}(S^{\text{rev}})$  is prefixed by  $U^{\text{rev}}$  and  $f^{\text{rev}}(n - i + 1 - d) = y$ . Thus  $U$  is  $(y, d)$ -unique if and only if for all leaves  $v$  in the subtree rooted in  $u$ ,  $S^{\text{rev}}[\ln(v) - d]$  is colored  $y$  under  $f^{\text{rev}}$ .

In the following, we will refer to a node  $u$  of  $\mathcal{T}$  as  $(y, d)$ -unique if  $L(u)^{\text{rev}}$  is a  $(y, d)$ -unique substring of  $S$ . We can now state the following corollary:

**Corollary 5.1.** *Let  $U$  be a substring of  $S$ ,  $\mathcal{T} = \mathcal{T}(S^{\text{rev}})$ , and  $(u, t) = \text{loc}(U^{\text{rev}}, \mathcal{T})$  such that  $u$  is an inner node of  $\mathcal{T}(S)$ . Then  $U$  is  $(y, d)$ -unique in  $S$  if and only if all children of  $u$  are  $(y, d)$ -unique.*

### 5.2.1 Finding all $(y, d)$ -unique substrings

Our first algorithm ALGO1 uses the suffix tree  $\mathcal{T}$  of the reverse string to identify all  $(y, d)$ -unique substrings of  $S$ , not only the minimal ones, for fixed  $y$  and  $d$ . It marks the  $(y, d)$ -unique nodes of  $\mathcal{T}$  in a postorder traversal of the tree. Note that if  $i > n - d$ , then position  $i$  cannot be  $(y, d)$ -good, simply because the position in which we would expect a  $y$  lies beyond the end of string  $S$ . In correspondence with the definition of  $(y, d)$ -unique substrings (Definition 5.1), we will treat such positions as if they were  $(y, d)$ -good.

The function  $g(u) : V(\mathcal{T}) \rightarrow \{0, 1\}$  is defined as follows:

- for a leaf  $u$  with leaf number  $ln(u) = i$ :

$$g(u) = \begin{cases} 1 & \text{if } i \leq d, \\ 1 & \text{if } i > d \text{ and } f(i - d) = y, \\ 0 & \text{otherwise,} \end{cases}$$

- for an inner node  $u$ :

$$g(u) = \begin{cases} y & \text{if } g(v) = 1 \text{ for all children } v \text{ of } u, \\ 0 & \text{otherwise.} \end{cases}$$

The algorithm computes  $g(u)$  for every node  $u$  in a bottom-up fashion, assigning  $g(u) = 1$  if and only if  $u$  is  $(y, d)$ -unique or if it is too close to the beginning of the string  $S^{\text{rev}}$ . If  $g(u) = 1$ , in addition it outputs all strings represented along the incoming edge of  $u$ , except for substrings which contain the  $\$$ -sign, i.e. suffixes of  $S^{\text{rev}}\$$ . For details, see Algorithm 5.

*Analysis:* For fixed  $d$ , computing  $g$  takes amortized  $\mathcal{O}(n)$  time over the whole tree, since computing  $g(u)$  is linear in the number of children of  $u$ , and therefore, charging the check whether for a child  $v$ ,  $g(v) = 1$ , to the child node, we get constant time per node. So, for fixed  $d$ , we have  $\mathcal{O}(n + K) = \mathcal{O}(n^2)$  time, where  $K$  is the number of  $(y, d)$ -unique substrings. Altogether, for  $d = 0, \dots, n$ , the algorithm takes  $\mathcal{O}(n^3)$  time.

*Example 13.* In the running example (Fig. 5.2), for color  $y$  and delay  $d = 3$ , the leaf nodes 9, 2, 7, 1, and 3 are marked with 1, and therefore the only inner node  $u$  which gets  $g(u) = 1$  is the parent of leaves number 9, 2, 7. The algorithm outputs the  $(y, 3)$ -unique substrings `ba`, `cbaca`, `acbaca`, `cacbaca`, `acacbaca`, `cacacbaca`, `acacacbaca`, `caca`, `acaca`, `ca`, `aca`, `ab`, `cab`, `acab`, `bacab`, `cbacab`, `acbacab`, `cacbacab`, `acacbacab`, `cacacbacab`, `bac`, `cbac`, `acbac`, `cacbac`, `acacbac`, `cacacbac`, `acacacbac`.

*Remark:* Note that some of these substrings do not occur even once in a position such that the last character is followed by a  $y$  with delay  $d = 3$ . For instance, the only occurrence of the substring `bac` in  $S$  is at position 7, so we would expect to see color  $y$  at position  $9 + 3 = 12$ , but the string  $S$  ends at position 11. We will treat this and similar questions in Section 5.4.

Algorithm 5: ALGO1

---

```

input : A colored string  $S$ , the suffix tree  $\mathcal{T}$  of  $S^{\text{rev}}$ , and  $y \in \Gamma$ .
output : All pairs  $(T, d)$  such that  $T$  is a  $(y, d)$ -unique substring of  $S$ .

1 for  $d \leftarrow 0$  to  $n$  do
2    $\lfloor$  UNIQUE( $root, y, d$ )

3 procedure UNIQUE( $u, y, d$ ):
4   if  $u$  is a leaf then //  $u$  is a leaf
5      $i \leftarrow \ln(u)$ 
6     if  $i \leq d$  or  $f^{\text{rev}}(i - d) = y$  then
7        $\lfloor$   $g(u) \leftarrow 1$ 
8     else
9        $\lfloor$   $g(u) \leftarrow 0$ 
10  else //  $u$  is an inner node
11     $\lfloor$   $g(u) \leftarrow \bigwedge_{v \text{ child of } u} \text{UNIQUE}(v, y, d)$ 
12  if  $g(u) = 1$  then
13    if  $u$  is a leaf then // do not output  $\$$ -substrings
14       $\lfloor$  output  $L(u, t)^{\text{rev}}$  for every  $t = sd(\text{parent}(u)) + 1, \dots, sd(u) - 1$ 
15    else
16       $\lfloor$  output  $(L(u, t)^{\text{rev}}, d)$  for every  $t = sd(\text{parent}(u)) + 1, \dots, sd(u)$ 
17  return  $g(u)$ 

```

---

### 5.2.2 Outputting only minimally $(y, d)$ -unique substrings

We next modify Algorithm ALGO1 to output only minimally  $(y, d)$ -unique substrings. As already noted, the work done by ALGO1 in each node is constant except for the output step, which is proportional to the length of the edge label leading to  $u$ .

In terms of the suffix tree  $\mathcal{T}$  of  $S^{\text{rev}}$ , minimality can be translated into conditions on the parent node and on the suffix link parent node (equivalently: the suffix link) in  $\mathcal{T}$ . We first need another definition:

**Definition 5.2 (Left-minimal nodes, left-minimal labels).** *Let  $u$  be a node of  $\mathcal{T} = \mathcal{T}(S^{\text{rev}})$ , different from the root, and let  $v = \text{parent}(u)$ . We call  $u$  left-minimal for  $(y, d)$  if  $u$  is  $(y, d)$ -unique but  $v$  is not and the label of the edge  $(v, u)$  is not equal to  $\$$ . If  $u$  is  $(y, d)$ -unique and left-minimal, then we can define  $\text{Left-min}(u) = x_1 \cdot L(v)^{\text{rev}}$ , the left-minimal  $(y, d)$ -unique substring of  $S$  associated to  $u$ , where  $x = x_1 \cdots x_k \in \Sigma^+$  is the label of edge  $(v, u)$ .*

*Example 14.* In our running example, let node  $u$  be the parent of leaf nodes 9, 2, 7, i.e.  $u = \text{loc}(\mathcal{T}, \text{aca})$ . Then  $u$  is left-minimal, since it is  $(y, 3)$ -unique but its parent is not. Its left-minimal label is  $\text{Left-min}(u) = \text{ca}$ . See Fig. 5.2.

It is easy to modify Algorithm 5 to output only left-minimal substrings: Whenever for an inner node  $u$  we get  $g(u) = 0$ , then for every child  $v$  of  $u$  with  $g(v) = 1$ , we output  $\text{Left-min}(v)$  (if defined). This can be done by replacing lines 12 to 16 in Algorithm 5 by:

```

12 if  $g(u) = 0$  then
13   for each child  $v$  of  $u$  with  $g(v) = 1$  do
14     if  $Left\text{-}min(v)$  is defined then
15        $\lfloor$  output ( $Left\text{-}min(v)$ ,  $d$ )

```

*Example 15.* The resulting algorithm now outputs, for color  $y$  and  $d = 3$ , the left-minimal substrings  $ca$ ,  $ab$ ,  $bac$ .

However, we are interested in substrings which are both left- and right-minimal. While left-minimality can be identified by checking the parent of a node  $u$ , for right-minimality, Observation 5.1 part (3) tells us that we need to check whether the string without the last character is  $(y, d + 1)$ -unique. In  $\mathcal{T}$ , this translates to checking the suffix link of the locus of the left-minimal substring  $Left\text{-}min(u)$ .

**Proposition 5.1.** *Let  $u$  be an inner node of  $\mathcal{T} = \mathcal{T}(S^{\text{rev}})$ , different from the root, such that  $L(u)^{\text{rev}}$  is  $(y, d)$ -unique in  $S$ . Let  $v = \text{parent}(u)$ , and  $x_1$  be the first character on the edge  $(v, u)$ . Further let  $t = sd(v) + 1$ , and  $(u', t) = \text{slink}(u, t)$ . Then the substring  $U = x_1 \cdot L(v)^{\text{rev}}$  is minimally  $(y, d)$ -unique in  $S$  if and only if  $v$  is not  $(y, d)$ -unique and  $u'$  is not  $(y, d + 1)$ -unique.*

*Proof.* For sufficiency, let  $U$  be minimally  $(y, d)$ -unique in  $S$ . Since  $L(v)^{\text{rev}} = x_1 U$ , and  $U$  is left-minimal, therefore  $v$  is not  $(y, d)$ -unique. Similarly, if  $U' = L(u', t)^{\text{rev}}$ , then we have that  $U = U'a$ , and  $u'$  is not  $(y, d + 1)$ -unique by right-minimality of  $U$ .

Conversely, since  $u$  is  $(y, d)$ -unique and  $v$  is not, by definition of left-minimality,  $U = Left\text{-}min(u)$  is left-minimal  $(y, d)$ -unique in  $S$ . Let  $U' = L(u', t)^{\text{rev}}$ , thus  $U = U'a$ , for some character  $a \in \Sigma^+$ . Since  $U'$  is not  $(y, d + 1)$ -unique, therefore  $U$  is right-minimal.

We can use Proposition 5.1 as follows. Once a left-minimal  $(y, d)$ -unique node  $u$  has been found, check whether  $u'$  is  $(y, d + 1)$ -unique, where  $u'$  is the node below the locus  $\text{slink}(u, sd(\text{parent}(u)) + 1)$ . It is easy to find node  $u'$  by noting that  $u' = \text{child}(\text{slink}(\text{parent}(u)), x_1)$ , where  $x_1$  is the first character of the edge label leading to  $u$ . But how do we know whether  $u'$  is  $(y, d + 1)$ -unique?

The answer is that we will process the distances  $d$  in descending order, from  $d = n$  down to  $d = 0$ . At the end of the iteration for  $d$ , we retain the information, keeping a flag on every node  $u$  which was identified as  $(y, d)$ -unique (i.e. which had  $g(u) = 1$ ). During the iteration for  $d - 1$ , we can then query node  $u'$  to find out whether it is  $(y, d)$ -unique. For details, see Algorithm 6.

*Example 16.* In the running example, we know from the previous round for  $d = 4$  that the only nodes that are  $(y, 4)$ -unique are the leaves number 4, 2, 1, 10, 3, and 8. We can now deduce that the substring  $ca$  is right-minimal, because  $u = \text{loc}(ca)$  is not  $(y, 4)$ -unique, and  $\text{slink}(\text{loc}(\mathcal{T}, ca^{\text{rev}})) = (u, 1)$ . Looking at the string  $S$  we see that  $ca$  is indeed right-minimal, since  $c$  is not  $(y, 3)$ -unique: it has an occurrence, in position 6, which is not followed by a  $y$  but by an  $x$  at position  $10 = 6 + 4$  (delay 4). Similarly, the other two left-minimal substrings  $ab$  and  $bac$  are also right-minimal, because their respective suffix links are not  $(y, 4)$ -unique.

Algorithm 6: ALGO2

---

```

input : a colored string  $S$ , the suffix tree  $\mathcal{T}$  of  $S^{\text{rev}}$  with suffix links, and  $y \in \Gamma$ .
output : all pairs  $(T, d)$  such that  $T$  is a minimally  $(y, d)$ -unique substring of  $S$ .

1 for  $d \leftarrow n$  downto 0 do
2    $\lfloor$  MINUNIQUE( $root, y, d$ )

3 procedure MINUNIQUE( $u, y, d$ ):
4   if  $u$  is a leaf then //  $u$  is a leaf
5      $i \leftarrow \text{ln}(u)$ 
6     if  $i \leq d$  or  $f^{\text{rev}}(i - d) = y$  then
7        $\lfloor$   $g(u) \leftarrow 1$ 
8     else
9        $\lfloor$   $g(u) \leftarrow 0$ 
10  else //  $u$  is an inner node
11     $\lfloor$   $g(u) \leftarrow \wedge_{v \text{ child of } u} \text{MINUNIQUE}(v, y, d)$ 
12  if  $g(u) = 0$  then // outputting minimal substrings for
    children
13    for each child  $v$  with  $g(v) = 1$  do
14      if  $\text{Left-min}(v)$  is defined then
15         $(v', t) \leftarrow \text{slink}(v, \text{sd}(u) + 1)$ 
16        if  $v'$  is not  $(y, d + 1)$ -unique then // flag from previous
          round
17           $\lfloor$  output  $(\text{Left-min}(v), d)$ 
18  return  $g(u)$ 

```

---

*Analysis:* For fixed  $d$ , the time spent on each leaf is constant (lines 5 to 10 in ALGO2); we charge the check of  $g(v)$  in line 12 to the child  $v$ , as well the work in lines 14 to 18 (computing  $\text{Left-min}(v)$  and checking the flag on  $v'$  from the previous round); these are all constant time operations, so we have amortized constant time per node, and thus  $\mathcal{O}(n)$  time for fixed  $d$ . Therefore, the total time taken by Algorithm 6 is  $\mathcal{O}(n^2)$ .

### 5.2.3 An algorithm for all colors

In some situations, one is interested in all minimally  $(y, d)$ -unique substrings, for any color  $y$ . Our third algorithm deals with this case (Problem 2). It is similar to ALGO2, except it uses a different coloring function  $g'$ . The new function  $g' : V \rightarrow \Gamma \cup \{*, 0\}$ , is defined as follows:

- for a leaf  $u$  with leaf number  $\text{ln}(u) = i$ :

$$g'(u) = \begin{cases} f^{\text{rev}}(i - d) & \text{if } i - d > 0, \\ * & \text{if } i - d \leq 0 \end{cases}$$

- for an inner node  $u$ :

$$g'(u) = \begin{cases} * & \text{if for all children } v \text{ of } u: g'(v) = *, \\ y \in \Gamma & \text{if } u \text{ has at least one child } v \text{ with } g'(v) = y \text{ and} \\ & \text{for all other children } v' \text{ of } u: g'(v') \in \{y, *\}, \\ 0 & \text{otherwise.} \end{cases}$$

Thus a node  $u$  is colored  $y$  if and only if all leaves of the subtree rooted in  $u$  are either colored  $y$  or  $*$ , and at least one leaf is colored  $y$ . We refer to such a node as *monochromatic*. A node is colored  $*$  if all leaves in the subtree are within  $d$  of the beginning of  $S^{\text{rev}}$ ; such a node can have monochromatic ancestors in the tree. Finally, a node is colored 0 if in its subtree there are at least two leaves which are colored by different colors from  $\Gamma$ . For a node colored 0, all of its ancestors are also colored 0.

*Example 17.* In our example, for  $d = 3$ , the leaves 11, 4, and 8 are colored  $z$ , the leaves 9 and 7 are colored  $y$ , the leaves 5, 10, and 6 are colored  $x$ , and the leaves 1, 2, and 3 are colored  $*$ . The only monochromatic inner nodes are  $\text{loc}(b)$  (colored  $x$ ), and  $\text{loc}(aca)$  (colored  $y$ ), while all others are colored 0. See Figure 5.2b.

Thus, Algorithm 7 finds all minimally  $(y, d)$ -unique substrings for all colors simultaneously, using the same ideas as ALGO2. The main difference is that now the coloring function  $g'$  is not binary, and accordingly, the information we have to store from the previous round (which will be needed to decide whether the substring is right-minimal) is no longer binary. See ALGO3 for details.

*Analysis:* The algorithm has  $n$  iterations, every iteration takes  $\mathcal{O}(n)$  time, so altogether we have again  $\mathcal{O}(n^2)$  time.

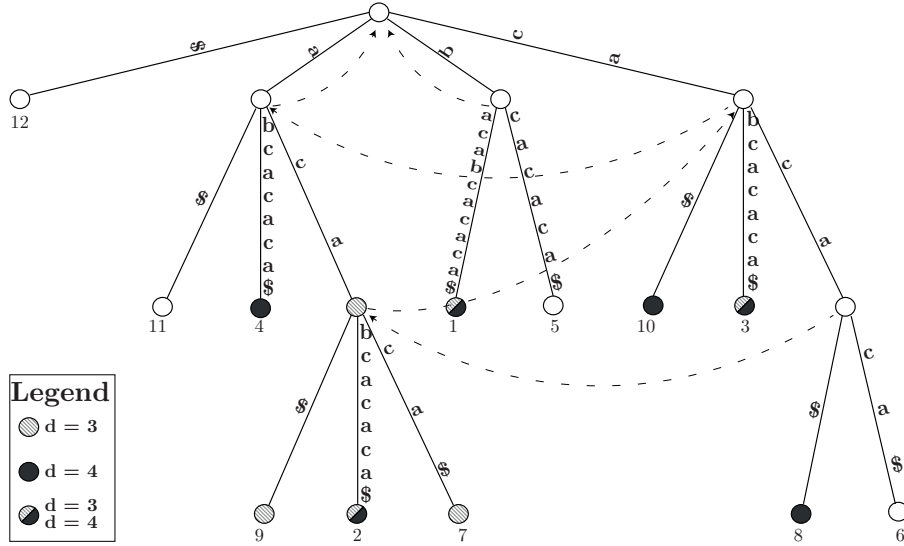
Therefore, if the color of interest is not part of the input, we can solve the problem in  $\mathcal{O}(n^2)$  time, which is also a worst-case lower bound on the output size, see Sec. 5.1. However, if the color  $y$  is part of the input, then this algorithm can be further improved. We will present this improvement in the next section.

### 5.3 Skipping Algorithm

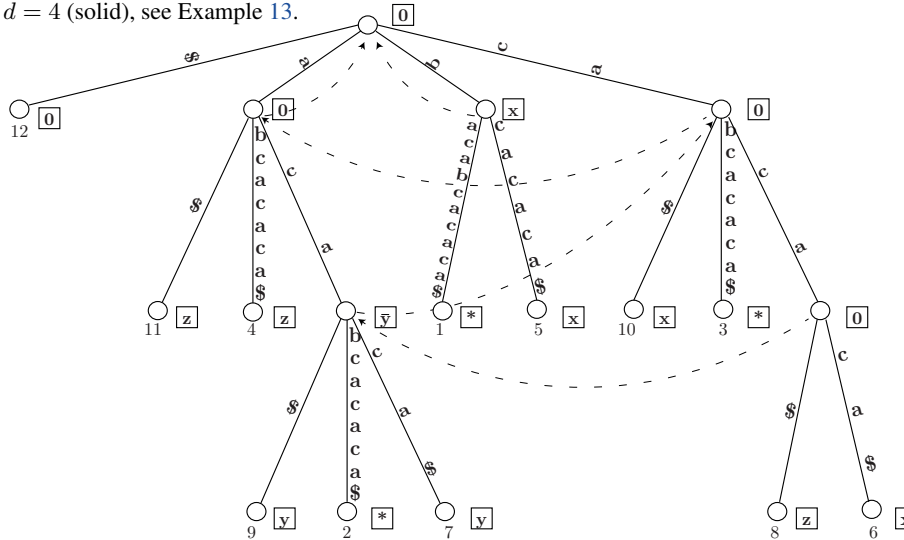
In this section, we discuss the discovering of  $(y, d)$ -unique substrings that are minimal. As in the baseline algorithm, we build the suffix tree  $\mathcal{T}(S^{\text{rev}})$  and, intuitively, we navigate it discovering all left-minimal  $(y, d)$ -unique substrings one by one, reporting only those that are minimal. Thus, according to Proposition 5.1, we have to discover all left-minimal  $(y, d+1)$ -unique substrings before discovering left-minimal  $(y, d)$ -unique substrings.

In order to discover them, fix  $\ell$ , for each node  $u$  of  $\mathcal{T}(S^{\text{rev}})$ , we consider which is the highest possible delay  $d$  smaller than  $\ell$  such that  $L(u)^{\text{rev}}$  can be  $(y, d)$ -unique, denoted by  $h(u, \ell)$ . We consider four different cases:

- If  $u$  is a leaf, then  $L(u)^{\text{rev}}$  is the  $j$ -prefix of  $S$ , where  $j = n - \ln(u) + 1 = |L(u)|$ 
  - If  $\ln(u) < \ell$ , then  $j + \ell - 1 > n$  thus  $L(u)^{\text{rev}}$  is  $(y, \ell - 1)$ -unique since the position of the color is beyond the end of the string, thus  $h(u, \ell) = \ell - 1$ .
  - If  $\ln(u) \geq \ell$  and there exists an  $i < \ell$  such that  $f(j + i) = y$ , then the highest possible value  $d < \ell$  such that  $L(u)^{\text{rev}}$  is  $(y, d)$ -unique is given by the position of the furthest occurrence of  $y$  within a distance of  $\ell - 1$  from  $j$ , thus  $h(u, \ell) = \max\{i < \ell \mid f(j + i) = y\}$ .



(a) The nodes are colored according to function  $g$  for the character  $y$ , for  $d = 3$  (dashed) and for  $d = 4$  (solid), see Example 13.



(b) The nodes are marked according to function  $g'$  for  $d = 3$ , see Example 17.

**Fig. 5.2:** The suffix tree  $\mathcal{T}$  of the reverse string  $S^{\text{rev}} = \text{bacabacacaca}$ , where  $S = \text{acacacbacab}$ , see Example 12. For clarity, the edges carry the label itself rather than a pair of pointers into the string. Suffix links are drawn as dotted directed edges.



Algorithm 7: ALGO3

---

```

input : a colored string  $S$ , and the suffix tree  $\mathcal{T}$  of  $S^{\text{rev}}$  with suffix links.
output : all triples  $(T, y, d)$  such that  $T$  is a minimally  $(y, d)$ -unique substring of  $S$ 

1 for  $d \leftarrow n$  downto 0 do
2    $\lfloor$  ALLCOLORSMINUNIQUE( $root, d$ )

3 procedure ALLCOLORSMINUNIQUE( $u, d$ ):
4   if  $u$  is a leaf then //  $u$  is a leaf
5      $i \leftarrow \text{ln}(u)$ 
6     if  $i \leq d$  then
7        $\lfloor$   $g'(u) \leftarrow *$ 
8     else
9        $\lfloor$   $g'(u) \leftarrow f^{\text{rev}}(i - d)$ 
10    else //  $u$  is an inner node
11       $X \leftarrow \{ \text{ALLCOLORSMINUNIQUE}(v, y, d) \mid v \text{ child of } u \}$ 
12      if  $X = \{*\}$  then
13         $\lfloor$   $g'(u) \leftarrow *$ 
14      else
15        if  $X = \{y\}$  or  $X = \{y, *\}$  with  $y \in \Gamma$  then
16           $\lfloor$   $g'(u) \leftarrow y$ 
17        else
18           $\lfloor$   $g'(u) \leftarrow 0$ 
19    if  $g'(u) = 0$  then // outputting minimal substrings for
        children
20      for each child  $v$  with  $g'(v) = y \in \Gamma$  do
21        if  $\text{Left-min}(v)$  is defined then
22           $(v', t) \leftarrow \text{slink}(v, \text{sd}(u) + 1)$ 
23          if  $v'$  is not  $(y, d + 1)$ -unique then // flag from previous
            round
24             $\lfloor$  output  $(\text{Left-min}(v), y, d)$ 
25    return  $g'(u)$ 

```

---

– Otherwise, if such  $i$  does not exist, we set  $h(u, \ell) = -1$ .

- If  $u$  is an internal node of  $\mathcal{T}(S^{\text{rev}})$ , then let  $k = \min\{h(v, \ell) \mid v \text{ child of } u\}$ , since it is not possible that  $L(u)^{\text{rev}}$  is  $(y, d')$ -unique, for any  $k < d' < \ell$ , thus  $h(u, \ell) = k$ .

Note that, in the latter case, we do not know if  $L(u)^{\text{rev}}$  is  $(y, d)$ -unique for  $d = h(u, \ell)$ , unless for all nodes  $v$  in the subtree rooted in  $u$ , there exists an  $\ell_v$  such that  $h(u, \ell) < \ell_v \leq \ell$  and  $h(v, \ell_v) = h(u, \ell)$ . In particular this is true if for all nodes  $v$ , it holds that  $h(v, d + 1) = h(u, \ell)$ .

The definition of  $h(u, \ell)$  is as follows:

$$h(u, \ell) = \begin{cases} \ell - 1 & \text{if } u \text{ is a leaf and } \ln(u) < \ell, \\ \max\{i < \ell \mid f(n - \ln(u) + 1 + i) = y\} & \text{if } u \text{ is a leaf and such } i \text{ exists,} \\ \min\{h(v, \ell) \mid v \text{ child of } u\} & \text{if } u \text{ is an inner node,} \\ -1 & \text{otherwise.} \end{cases}$$

In order to evaluate  $h(u, \ell) = \max\{i < \ell \mid f(j + i) = y\}$  in the case where  $u$  is a leaf and such  $i$  exists, we define a bitvector  $b_y[1, 2n]$  such that

$$b_y[i] = \begin{cases} 1 & \text{if } f(i) = y \text{ or } i > n, \\ 0 & \text{otherwise.} \end{cases}$$

We equip  $b_y$  with a data structure to support `rank` and `select` queries. Given a node  $u$  such that  $\ln(u) \geq \ell$ , let  $j = n - \ln(u) + 1$ . We have that  $h(u, \ell) = \max\{\text{select}(b_y, \text{rank}(b_y, j + \ell)) - j, -1\}$ .

*Example 18.* In our running example, whose suffix tree is depicted in Figure 5.2, let us consider the node  $u$  corresponding to the substring `aca` in the string  $S^{\text{rev}}$ . In order to compute  $h(u, 9)$ , we have to recursively compute the  $h$  function for all children of  $u$ . Let  $v$ ,  $s$ , and  $t$  be the leaves corresponding to the 9-th, 2-nd, and 7-th suffix of  $S^{\text{rev}}$ , respectively. If we remove the dollar character from the end of the string  $S^{\text{rev}}$ , then the 9-th, 2-nd, and 7-th suffix of  $S^{\text{rev}}$  corresponds to the 3-rd, 10-th, and 8-th prefix of  $S$ , respectively. We have that  $h(s, 9) = h(t, 9) = 8$ , since the furthest possible  $y$  at distance smaller than 9 from the 10-th and 8-th prefix of  $S$  are beyond the end of the string. While, the furthest possible  $y$  at distance smaller than 9 from the 3-rd prefix of  $S$  is at distance 5. Thus  $h(v, 9) = h(u, 9) = 5$ . The intuition is that the highest possible  $d$ , smaller than 9 such that the substring `aca` can be  $(y, d)$ -unique cannot be larger than 5, since there is an occurrence of `aca` that has no  $y$ 's at distance between 6 and 8.

Let us now compute  $h(u, 3)$ . We have that  $h(s, 3) = 2$ , since the furthest possible  $y$  at distance smaller than 9 corresponding to the 10-th prefix of  $S$  is beyond the end of the string. For the 8-th prefix of  $S$  we have that the furthest  $y$  at distance smaller than 3 is at distance 1, thus  $h(t, 3) = 1$ . While, for the leaf  $v$  there is no  $y$  at distance smaller than 3, thus  $h(v, 3) = -1$ . Hence, we have that  $h(u, 3) = -1$ .

We use the  $h(u, \ell)$  function in the following way, during the discovery process of all  $(y, d)$ -unique substrings of  $S$ , provided that we have already discovered all  $(y, d + 1)$ -unique substrings of  $S$ . Let  $\ell = d + 1$ , for all nodes  $u$  of  $\mathcal{T}(S^{\text{rev}})$  we store the values  $h(u, \ell)$ . We discover the minimally  $(y, d)$ -unique substrings of  $S$ , finding all nodes  $u$  such that  $h(u, \ell) = d$ . Among those, the nodes that are also left-minimal are those nodes  $u$  such that,  $h(\text{parent}(u), \ell) < d$ . We then check if  $u$  is also right-minimal by checking if its suffix-link parent is  $(y, d + 1)$ -unique, as in Algorithm 6.

The key idea of the skipping algorithm is to keep the values  $h(u, \ell)$  updated during the process. Let  $H(u)$  be the array that, at the beginning of the discovery of all  $(y, d)$ -unique substrings of  $S$ , stores the values  $h(u, \ell)$ . We want to keep the array  $H$  updated in a way such that, after we discovered all  $(y, d)$ -unique substrings of  $S$ , for all nodes  $u$ ,  $H(u) = h(u, \ell - 1)$ . Thus, once we discover that a node  $u$  is left-minimal  $(y, d)$ -unique, we update the value of  $H(u) = h(u, \ell - 1)$ . We then update the following values:

- for all nodes  $v$  in the subtree rooted in  $u$ , we update the values  $H(u) = h(u, \ell - 1)$ .
- for all nodes  $p$  ancestors of  $u$ , we update the values  $H(p) = \min(H(p), h(u, \ell - 1))$

In order to efficiently find all nodes  $u$  such that  $h(u, \ell) = d$  and  $h(\text{parent}(u), \ell) < d$ , we use a *maximum-oriented indexed priority queue*, storing the values of  $H(u)$  as keys and  $iBFS(u)$  as index. Under this condition, if two nodes have the same key value, then parents have higher priority than their children in  $IPQ$ . We keep the priority queue updated using a `demote` operation while we discover left-minimal nodes and we update the values of the array  $H$  stored as keys of  $IPQ$ . Algorithm 8 shows how to compute  $h(u, \ell)$  for a given node  $u$ , and how we update the values of the keys in the  $IPQ$  for all children  $v$  of  $u$ .

---

Algorithm 8: HIGHEST POSSIBLE VALUE OF  $d$ .

---

**input** : A node  $u$  in the suffix tree  $\mathcal{T}(S^{\text{rev}})$  and a threshold  $\ell$ .  
**output** : The highest possible delay  $d$  smaller than  $\ell$  such that  $L(u)^{\text{rev}}$  can be  $(y, d)$ -unique.

```

1 procedure  $h(u, \ell)$ :
2    $min_d \leftarrow \ell - 1$ 
3   if  $u$  is a leaf then
4      $j \leftarrow n - \ln(u) + 1$ 
5      $min_d \leftarrow \max\{\text{select}(b_y, \text{rank}(b_y, j + \ell)) - j, -1\}$ 
6   else
7     forall children  $v$  of  $u$  do
8        $d = h(v, \ell)$ 
9       if  $min_d < d$  then
10         $min_d \leftarrow d$ 
11    $IPQ.\text{demote}(u, min_d)$ 
12   return  $min_d$ 

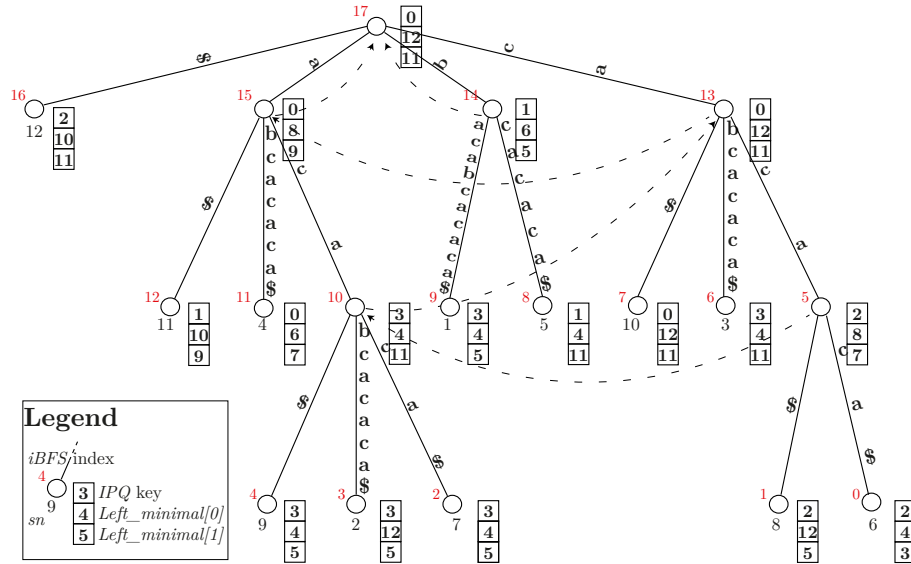
```

---

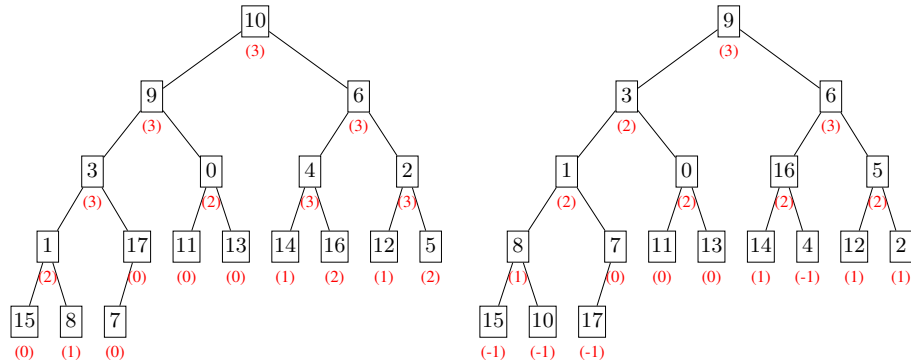
The skipping algorithm summarized in Algorithm 9, initially prepares the priority queue  $IPQ$  inserting all nodes of  $\mathcal{T}(S^{\text{rev}})$  with key  $n + 1$ . Then we repeat the following until there exists a node with non negative key: we extract the max element  $(u, \ell)$  of  $IPQ$ , we decide whether or not it has to be reported, i.e. if it is right-minimal; we apply Algorithm 8 to update the key values of all nodes in the subtree of  $u$  and then we update the values of the keys of all ancestors of  $u$ .

*Analysis:* For all nodes  $u$  in  $\mathcal{T}(S^{\text{rev}})$ , we observe that the key value associated to  $u$  in  $IPQ$  at the beginning is set to  $n + 1$ . Each time Algorithm 9 and Algorithm 8 visit a node in  $\mathcal{T}(S^{\text{rev}})$ , the key value of  $u$  in  $IPQ$  is decreased performing a `demote()` operation until its value becomes negative. Thus, for each node we perform at most  $n + 1$  `demote()` operations. Since the number of nodes in  $\mathcal{T}(S^{\text{rev}})$  is linear in  $n$ , we have that Algorithm 9 runs in  $\mathcal{O}(n^2 \log(n))$  time.

*Example 19.* In our running example, we want to report all minimally  $(y, d)$ -unique substrings of the colored string, for the character  $y$ . Using Figure 5.3, we now show how we discover that the substring `ca` is  $(y, 3)$ -unique. Let the tree in Figure 5.3b be the indexed priority queue after 48 iterations of Algorithm 8. We have that the maximum element in the indexed priority queue  $IPQ$  is the node of  $\mathcal{T}(S^{\text{rev}})$  corresponding to the 10-th node in the reverse index BFS of the tree, as reported in Figure 5.3a. The associated key value of the maximum element is 3, which means that the corresponding



(a) The suffix tree of  $\mathcal{T}$  of  $S^{rev}$ , reporting the values of  $IPQ$ ,  $left\_minimal[0]$ , and  $left\_minimal[1]$  for each node, after 48 iterations of Algorithm 8.



(b) Indexed priority queue  $IPQ$  after 48 iterations of Algorithm 8.

(c) Indexed priority queue  $IPQ$  after 49 iterations of Algorithm 8.

**Fig. 5.3:** Top (5.3a): The suffix tree of  $\mathcal{T}$  of the reverse string  $S^{rev} = bacabacacaca$ , reporting the values of  $IPQ$ ,  $left\_minimal[0]$ , and  $left\_minimal[1]$  for each node, after 48 iterations of Algorithm 8. In red, in the upper left of each node, we report the reverse index BFS of the node, below each leaf we report the associated suffix number, on the right of the node we report the values of  $IPQ$ ,  $left\_minimal[0]$ , and  $left\_minimal[1]$ . Bottom: The indexed priority queue  $IPQ$  after 48 (5.3b) and 49 (5.3c) iterations of Algorithm 8. In the nodes of the priority queue we have the index of the nodes of the suffix tree  $\mathcal{T}(S^{rev})$  numbered in the reverse index BFS. Below each node, in red and in brackets, the value of the key associated to each index.

Algorithm 9: SKIPPING

---

```

input : A colored string  $S$ , and a color  $y \in \Gamma$ 
output : All minimally  $(y, d)$ -unique substrings of  $S$ .

1 forall nodes  $v$  of  $\mathcal{T}(S^{\text{rev}})$  do
2    $\lfloor$   $IPQ.\text{insert}(v, n + 1)$ 
3 while  $IPQ.\text{allNegative}() = \text{false}$  do
4    $(u, d) \leftarrow IPQ.\text{max}()$ 
5    $(u', t) = \text{slink}(u, sd(\text{parent}(u)) + 1)$ 
6   if  $u'$  is not  $(y, d + 1)$ -unique then // flag from previous round
7      $\lfloor$   $\text{output}(d, \text{Left-min}(u))$ 
8      $\text{min}_d = h(u, d)$ 
9   forall ancestors  $v$  of  $u$  do
10    if  $IPQ.\text{keyOf}(v) > \text{min}_d$  then
11       $\lfloor$   $IPQ.\text{demote}(v, \text{min}_d)$ 

```

---

substring is left-minimal  $(y, 3)$ -unique. In order to decide if the corresponding substring is also right-minimal, we check if the suffix link parent of the node number 10, which is the node number 13, is left-minimal for  $d = 4$ . The value of the last even value such that the node number 13 has been left-minimal is set to 12. Thus the node 10 is minimally  $(y, 3)$ -unique and has to be reported. We now compute the  $h(u, \ell)$  function for the node number 10,  $u$ , and  $\ell = 3$ . As shown in Example 18, we have that the  $h$  function for the nodes number 2, 3, and 4 are 1, 2,  $-1$ . Thus, the value of the  $h$  function for the node number 10 is  $-1$ . We then update the values of all parents of the node number 10. This results in an update of the values of the indexed priority queue  $IPQ$  as reported in Figure 5.3c.

### 5.3.1 Right-minimality check

According to Proposition 5.1, in order to decide if a node is left-minimal  $(y, d)$ -unique, we have to check that the suffix link parent  $u' = \text{slink}(u)$  is not a left-minimal  $(y, d + 1)$ -unique node. Since we discover  $(y, d)$ -unique substring in a decreasing order of  $d$ , it is enough to store, for each node, the previous value of  $d$  such that the node is left-minimal.

Given a node  $u$ , we store this information in two arrays, indexed by the  $iBFS(u)$ . In one array we store the last even values of  $d$  such that the node was left-minimal. In the other array we store the last odd values of  $d$  such that the node was left-minimal. This prevents possible overwriting of information, e.g., let  $v = \text{slink}(u)$  such that  $v$  is left-minimal  $(y, d + 1)$ -unique and left-minimal  $(y, d)$ -unique node. Let us assume that  $v$  is processed before the node  $u$  that is left-minimal  $(y, d)$ -unique. If we had only one array holding the information of the last value of  $d$  such that a node was left-minimal, then this value for  $v$  would now be  $d$ , instead of  $d + 1$ . Thus, we would erroneously conclude that  $v$  is also right-minimal, hence that it is minimally  $(y, d)$ -unique. Using one array to store even values of  $d$  and one array to store odd values of  $d$ , we solve this problem, since  $v$  updates the array associated to the parity of  $d$ , while  $u$  queries the one associated to the parity of  $d + 1$ .

We can replace lines 6 to 7 of Algorithm 9 with the following lines of code, where we assume that at the beginning  $left\_minimal[b][u] = \infty$  for all  $b = \{0, 1\}$  and for all node  $u$ .

```

6 report ← (left_minimal[(d + 1) mod 2][u'] ≠ d + 1)
7 if report then
8   └─ output (d, Left-min(u))
9 left_minimal[d mod 2][u] ← d

```

See Figure 5.3a for an example of the values of the arrays  $left\_minimal[0]$  and  $left\_minimal[1]$ .

## 5.4 Output restrictions and algorithm improvement

In this section we discuss some practically-minded output restrictions. They can be implemented as a filter to the output, thus discarding some solutions, but if they are considered as part of the problem, they lead to an improvement for the skipping algorithm.

The output restrictions are the following. Note that our definition of  $(y, d)$ -unique allows that a substring occurs only once, or that none of its occurrences is followed by a  $y$  with delay  $d$ , because they are all close to the end of string. In this section, we restrict our attention to  $(y, d)$ -unique substrings which have at least two occurrences followed by  $y$  with delay  $d$ .

Given a colored string  $S$ , let  $T$  be minimally  $(y, d)$ -unique. We report  $(T, d)$  if and only if the following holds:

1. There are at least two occurrences of  $T$  in  $S$ .
2. Let  $i$  be the second smallest occurrence of  $T$  in  $S$ , then  $i + |T| - 1 + d \leq n$ .

A substring  $T$  that satisfies the above conditions is called *real type* minimally  $(y, d)$ -unique substring. In order to satisfy those conditions, it is enough to perform the output operations at line 7 of Algorithm 9 and at line 17 of Algorithm 6 if the node  $u$  is not a leaf and the value of the second greatest suffix of  $S^{\text{rev}}$  in the subtree rooted in  $u$  is greater than or equal to  $d$ . Since each node  $u$  in the suffix tree  $\mathcal{T}(S^{\text{rev}})$ , corresponds to an interval  $[i, j]$  of the suffix array of  $S^{\text{rev}}$ , we can find the second greatest suffix using a range maximum query *rMq* data structure built on the suffix array of  $S^{\text{rev}}$ . Then, the second greatest suffix can be found in  $\mathcal{O}(1)$  time, using  $2n + o(n)$  bits of extra space.

The  $h(u, \ell)$  function is used in Algorithm 9 in order to find left-minimal nodes in the suffix tree. If we consider the output restrictions as part of the problem, then we do not have to report minimally  $(y, d)$ -unique substrings that occur only once, i.e., leaves in  $\mathcal{T}(S^{\text{rev}})$ . Then, for all nodes  $u$  such that all children of  $u$  are leaves, we can directly compute the highest value of  $d < \ell$  such that  $L(u)^{\text{rev}}$  is  $(y, d)$ -unique. This leads to the definition of the *fast\_h*( $u, \ell$ ) function for a node  $u$  of  $\mathcal{T}(S^{\text{rev}})$ . The function *fast\_h*( $u, \ell$ ) is defined similarly to the function  $h(u, \ell)$  with the additional following case:

- If all children of  $u$  are leaves, we can directly compute the highest value of  $d < \ell$  such that  $L(u)^{\text{rev}}$  is  $(y, \ell)$ -unique as the largest value  $d < \ell$  such that, for each child  $v$  of  $u$ , it holds that  $h(v, d + 1) = d$ . In other words, we are looking for the largest  $d < \ell$  such that all children of  $v$  are  $(y, d)$ -unique.

The definition of  $fast\_h(u, \ell)$  is as follows:

$$fast\_h(u, \ell) = \begin{cases} \ell - 1 & \text{if } u \text{ is a leaf and } \ln(u) < \ell, \\ \max\{i < \ell \mid f^{\text{rev}}(\ln(u) - i) = y\} & \text{if } u \text{ is a leaf and such } i \text{ exists,} \\ \max\{i < \ell \mid fast\_h(v, i + 1) = i & \text{if all children of } u \text{ are leaves} \\ \quad \text{for all } v \text{ child of } u\} & \text{and such } i \text{ exists,} \\ \min\{fast\_h(v, \ell) \mid v \text{ child of } u\} & \text{if } u \text{ is an inner node,} \\ -1 & \text{otherwise.} \end{cases}$$

The additional case of  $fast\_h(u, \ell)$  can be computed as follows. Let  $u$  be a node such that all children of  $u$  are leaves. We set  $i = \ell$ , and compute the values  $h(v, i)$  where  $v$  is a child of  $u$ . We update the value of  $i = \min(i, h(v, i) + 1)$ , compute the value of  $h(v', i)$  where  $v'$  is the next child of  $u$ , and update the value of  $i = \min(i, h(v', i) + 1)$ . We continue iterating until all children  $v$  of  $u$  have the same value  $h(v, i)$ , possibly  $-1$ . Algorithm 10 summarizes these improvements to Algorithm 8. In order to use the  $fast\_h(u, \ell)$  function in Algorithm 9, it is enough to replace the  $h()$  function at line 8 by the  $fast\_h()$  function.

---

Algorithm 10: HIGHEST POSSIBLE VALUE OF  $d$ .

---

**input** : A node  $u$  in the suffix tree  $\mathcal{T}(S^{\text{rev}})$ , and a threshold  $\ell$ .  
**output** : The highest possible delay  $d$  smaller than  $\ell$  such that  $L(u)^{\text{rev}}$  can be  $(y, d)$ -unique.

```

1 procedure  $fast\_h(u, \ell)$ :
2    $min_d \leftarrow \ell - 1$ 
3   if  $u$  is a leaf then
4      $j \leftarrow n - \ln(u) + 1$ 
5      $min_d \leftarrow \max\{\text{select}(b_y, \text{rank}(b_y, j + \ell)) - j, -1\}$ 
6   else if all children of  $u$  are leaves then
7     repeat
8        $is\_changed \leftarrow false$ 
9       forall children  $v$  of  $u$  do
10         $d = fast\_h(v, min_d + 1)$ 
11        if  $min_d < d$  then
12           $is\_changed \leftarrow true$ 
13           $min_d \leftarrow d$ 
14      until  $is\_changed$  AND  $min_d \geq 0$ 
15   else
16     forall children  $v$  of  $u$  do
17        $d = fast\_h(v, \ell)$ 
18       if  $min_d < d$  then
19          $min_d \leftarrow d$ 
20    $IPQ.\text{demote}(u, min_d)$ 
21   return  $min_d$ 

```

---

## 5.5 Experimental results

We implemented the algorithms presented in the previous sections and measured their performance on randomly generated datasets and real-world datasets. The implementation is available online at <https://github.com/maxrossi91/colored-strings-miner>.

### 5.5.1 Setup

We performed experiments on a 3.4 GHz Intel Core i7-6700 CPU equipped with 8 MiB L3 cache and 16 GiB of DDR4 main memory. The machine had no other significant CPU tasks running, and only a single thread of execution was used.

The OS was Linux (Ubuntu 16.04, 64bit) running kernel 4.4.0. All programs were compiled using g++ version 5.4.0 with `-O3 -DNDEBUG -funroll-loops -msse4.2` options.

All given runtimes were recorded with the C++11 `high_resolution_clock` time measurement facility.

### 5.5.2 Data

We used two different datasets; the first one consists of randomly generated data, while the second one consists of real-world data.

The randomly generated data are colored strings generated using the C library function `rand()`. We varied the length  $n = 100, 1000, 10\,000, 100\,000$ , the alphabet size  $\sigma = 2, 4, 8, 16, 32$ , and the number of colors  $\gamma = 2, 4, 8, 16, 32$ . In all cases except for  $n = 100\,000$ , we used seeds 0, 9843, 27 837, 19 341, 29 044; for  $n = 100\,000$ , we used only seed 0. The string is generated one character (and its color) at a time, i.e. fixing  $\sigma$  and  $\gamma$ , the string of length  $n = 1000$  is a prefix of the string  $n = 10\,000$ . The strings are generated using a uniform distribution of characters and colors. We report only the results of experiments for the values of length  $n = 1000, 10\,000, 100\,000$ , alphabet size  $\sigma = 2, 8, 32$ , number of colors  $\gamma = 2, 8, 32$ , and seed 0, since these are representative of the trend we observed in all our experiments.

The real-world data is the result of a simulation on a set of established benchmarks in embedded systems verification [22, 45, 102], reported in Table 5.1. The benchmarks are descriptions of hardware design at the register-transfer level (RTL) of abstraction. Each design is composed of a set of primary input bits (PIs) and a set of primary output bits (POs). Primary inputs and primary outputs are grouped into ports. The simulation of designs is a sequence of temporal events which act to capture the effects of the values given as inputs for the design into the design itself, and consequently the effects of the input values on the values assumed by the outputs. We simulated the benchmarks providing as inputs randomly generated sequences using an automatic test pattern generator (ATPG). The result of the simulation is collected in a simulation trace, which stores, for each temporal event, the values of the primary inputs and of the primary outputs. For each simulation event, we consider the values of all primary inputs as characters of the alphabet  $\Sigma$ , and the values of a port of the primary outputs as colors. In other words, for simulation event  $i$ ,  $S[i]$  is the value of the primary inputs, and  $f_S(i)$  is the value of the primary outputs.



Design	Description	PIs	POs	$n$	$\sigma$	$\gamma$	$n_y$
b03	Resource arbiter [45]	6	4	100 000	17	5	3210
b06	Interrupt handler [45]	4	6	100 000	5	4	44 259
s386	Synthesized controller [22]	9	7	100 000	129	2	8290
camellia	Symmetric key block cypher [102]	262	131	103 615	70	224	2292
serial	Serial data transmitter	11	2	100 000	118	2	16 353
master	Wishbone bus master [102]	134	135	100 000	417	80	759

**Table 5.1:** Real-world datasets used in the experiments. In the column *Design* we report the name of the hardware design that we used to generate the simulation trace. In column *PIs* we give the number of primary inputs of the design, while in *POs* that of its primary outputs. In column  $n$  we report the length of the simulation trace, and in columns  $\sigma$  and  $\gamma$  the size of the alphabet and the number of colors, respectively. For each design we fixed a color  $y$ , and the value  $n_y$  refers to the number of  $y$  characters in the simulation trace.

### 5.5.3 Algorithms

We compared the following implementations:

- `base`: the baseline algorithm (Algorithm 6)
- `skip`: the skipping algorithm (Algorithm 9) using the  $h$  function (Algorithm 8)
- `real`: the skipping algorithm (Algorithm 9) using the  $fast\_h$  function (Algorithm 10)
- `base-all`: the baseline algorithm for all colors (Algorithm 7)

All algorithms report minimally  $(y, d)$ -unique substrings only if they are *real type*. We used the `sdsl-lite` library [67] for compressed suffix trees, range maximum query, and rank and select supports for bit vector implementations.

### 5.5.4 Results

We performed all experiments five times and report the average execution time over the five runs. Experimental results are reported in Table 5.2 and Table 5.3.

Table 5.2 shows the results of the executions of `base`, `skip` and `real` algorithms over the randomly generated strings data.

We can observe how the algorithms scale

1. with respect to an increase in the numbers of colors, which has the effect of reducing the number of  $y$ -colored characters;
2. with respect to an increase in the alphabet size; and
3. with respect to an increase in the length of the text.

We see that all three algorithms behave the same in all cases. Increasing the number of colors (case 1), the running time decreases. Conversely, when the size of the text alphabet increases (case 2), the running time increases also. Finally, we observe a quadratic dependence of the running time on text length (case 3); this is in accordance with our theoretic results (see Sec. 5.2 and 5.3).

Table 5.2 shows that the `skip` algorithm is almost always faster than the `base` algorithm, and that the average speedup is 1.49, with a maximum of 2.15. Moreover, we have that the `real` algorithm is almost always faster than the `skip` algorithm, and the average speedup is 1.17, with a maximum of 2.15. Finally, the average speedup

Alphabets		Execution Time (sec)			Speedup (ratio)		
$\sigma$	$\gamma$	base	skip	real	base/skip	skip/real	base/real
<i>N</i> = 1000							
2	2	0.91	0.88	0.86	1.04	1.03	1.07
	8	0.84	0.48	0.42	1.76	1.13	1.99
	32	0.83	0.38	0.33	<b>2.15</b>	1.18	2.53
8	2	1.29	1.26	1.29	1.02	0.98	1.00
	8	1.13	0.73	0.67	1.56	1.08	1.69
	32	1.10	0.60	0.53	1.83	1.13	2.07
32	2	1.88	1.82	1.91	1.03	0.95	0.99
	8	1.69	1.22	1.04	1.39	1.18	1.63
	32	1.67	0.95	0.85	1.75	1.11	1.95
<i>N</i> = 10000							
2	2	101.45	100.82	96.68	1.01	1.04	1.05
	8	93.98	53.65	47.04	1.75	1.14	2.00
	32	91.56	43.16	36.39	2.12	1.19	2.52
8	2	159.95	154.15	146.07	1.04	1.06	1.10
	8	140.40	92.95	76.19	1.51	1.22	1.84
	32	136.95	78.38	61.55	1.75	1.27	2.23
32	2	227.84	202.74	198.09	1.12	1.02	1.15
	8	213.22	128.94	111.89	1.65	1.15	1.91
	32	211.44	111.49	94.49	1.90	1.18	2.24
<i>N</i> = 100000							
2	2	10 732.92	11 584.92	10 689.51	0.93	1.08	1.00
	8	9919.61	6414.88	5319.75	1.55	1.21	1.86
	32	9744.38	5226.59	4187.82	1.86	1.25	2.33
8	2	18 732.49	18 339.14	17 026.05	1.02	1.08	1.10
	8	16 538.51	11 099.78	8681.53	1.49	1.28	1.91
	32	16 191.54	9544.45	7031.47	1.70	1.36	2.30
32	2	28 879.24	27 642.08	23 857.90	1.04	1.16	1.21
	8	25 670.11	17 259.61	11 676.85	1.49	1.48	2.20
	32	24 991.60	15 001.72	9238.99	1.67	<b>1.62</b>	<b>2.71</b>

**Table 5.2:** Results of the executions of *base*, *skip* and *real* algorithms over the randomly generated strings data. The first two columns report the size of the text alphabet  $\sigma$  and the number of colors  $\gamma$ . The next three columns report the execution time in seconds of algorithms *base*, *skip* and *real*, respectively. The last three columns report the speedup ratio between the execution times of algorithms *base* and *skip*, *skip* and *real*, and *base* and *real*, respectively.

Design	Execution Time (sec)			Speedup (ratio)		
	base	skip	real	base/skip	skip/real	base/real
b03	4088.90	4344.18	3570.79	0.94	1.22	1.15
b06	4935.17	6188.51	5189.33	0.80	1.19	0.95
s386	5183.01	6142.26	4434.23	0.84	1.39	1.17
camellia	5155.23	1273.20	1256.35	4.05	1.01	4.10
serial	5221.75	1124.62	1129.30	4.64	1.00	4.62
master	5466.83	326.31	324.66	16.75	1.01	16.84

**Table 5.3:** Results of the executions of *base*, *skip* and *real* algorithms over the real dataset. The first column reports the name of the design where the simulation trace is retrieved, then the other six columns report, as in Table 5.2, the execution times and the speedups of the algorithms *base*, *skip* and *real*.

between `real` and `base` is 1.76, with a maximum of 2.71 in the case of  $N = 100\,000$ ,  $\sigma = 32$  and  $\gamma = 32$ .

Table 5.3 shows the results of the executions of `base`, `skip` and `real` algorithms over the real dataset. On real-world data, we have the same trend as with randomly generated strings, but here the speedup of the `real` algorithm with respect to the `base` algorithm is much higher, namely 4.60 on average, with a maximum of 16.84 on the master device. However, on three of the six datasets, `base` is faster than `skip`, while it is even faster than `real` on one.

Next, we report the experimental results comparing the `base` and `base-all` algorithms. The setup is the same as in the previous case, i.e. we performed five runs of each experiment and give the average execution time. The results on the randomly generated and real-world datasets are reported in Tables 5.4 and 5.5, respectively.

From both Table 5.4 and Table 5.5, we can observe that the running time of the `base` algorithm is not heavily affected by the fact that we are looking for a specific color: there is a small increase in running time from `base` (reporting all real-type minimally  $(y, d)$ -unique substrings for any  $d$  and just *one* color  $y$ ), to `base-all`, reporting all real-type minimally  $(y, d)$ -unique substrings for any  $d$  and for *all* colors  $y$ .

On the real-world data, `base-all` outperforms `base` on three of the six datasets, while `base` is faster on the other three. Note that the number of patterns is considerably larger for `base-all`. The reason why `base-all` is faster than `base` lies in the number of times that the right-minimality check is performed, i.e. the number of times that the condition in line 11 and in line 19 is true. In `base`, the number of times that the condition is true is at least the number of times that the same condition is true in `base-all`, for the following reason. Consider a node  $u$  of the suffix tree. If  $u$  corresponds to a  $(y, d)$ -unique substring, then, fixed  $d$ ,  $g(u) \neq 0$  as well as  $g'(u) \neq 0$ , while if  $u$  corresponds to a  $(z, d)$ -unique substring, then  $g'(u) \neq 0$  while  $g(u) = 0$  for some  $z \neq y$ . In particular, the number of times that the condition is true in `base` is at least the difference of the number of patterns between `base` and `base-all`, which can be considerable. On the other hand, if this difference is too large, then the handling of the array storing the properties affects the running time more than the right-minimality check does.

Alphabets		Execution Time (sec)		Speedup (ratio)	Number of Properties	
$\sigma$	$\gamma$	base	base-all	$\frac{\text{base-all}}{\text{base}}$	base	base-all
$N=1000$						
2	2	0.91	1.00	1.09	26 894	50 922
	8	0.84	0.99	1.17	1745	15 563
	32	0.83	0.98	1.18	76	3996
8	2	1.29	1.45	1.13	30 219	56 241
	8	1.13	1.34	1.18	1516	12 919
	32	1.10	1.35	1.23	75	3306
32	2	1.88	2.06	1.09	25 120	46 758
	8	1.69	1.95	1.16	1245	10 578
	32	1.67	1.92	1.15	40	2585
$N=10\,000$						
2	2	101.45	110.11	1.09	2 374 231	4 699 647
	8	93.98	108.06	1.15	187 202	1 447 913
	32	91.56	106.67	1.17	11 767	370 303
8	2	159.95	179.30	1.12	2 844 680	5 607 007
	8	140.40	166.85	1.19	167 431	1 294 765
	32	136.95	163.98	1.20	9989	317 444
32	2	227.84	244.59	1.07	1 466 242	2 892 791
	8	213.22	234.80	1.10	83 320	642 806
	32	211.44	232.75	1.10	4948	156 947
$N=100\,000$						
2	2	10 732.92	11 612.09	1.08	239 039 415	473 572 454
	8	9919.61	11 488.66	1.16	17 680 770	145 246 888
	32	9744.38	11 386.70	1.17	1 129 991	37 254 203
8	2	18 732.49	20 811.91	1.11	279 720 849	552 304 418
	8	16 538.51	19 359.00	1.17	15 517 256	127 614 675
	32	16 191.54	19 038.86	1.18	947 858	31 264 100
32	2	28 879.24	32 666.95	1.13	243 283 926	479 770 368
	8	25 670.11	29 795.52	1.16	11 982 556	98 601 898
	32	24 991.60	29 257.62	1.17	713 137	23 592 691

**Table 5.4:** Results of the execution of algorithms `base` and `base-all` over the randomly generated strings data. The first two columns report the size of the text alphabet  $\sigma$  and the number of colors  $\gamma$ . The next two columns report the execution time in seconds of algorithms `base` and `base-all`, respectively. The fourth column reports the speedup ratio between the execution times of algorithms `base-all` and `base`. Finally, the last two columns report the number of properties extracted by the `base` and the `base-all` algorithms, respectively.

Design	Execution Time ( <i>sec</i> )		Speedup ( <i>ratio</i> ) $\frac{\text{base-all}}{\text{base}}$	Number of Properties	
	base	base-all		base	base-all
b03	4088.90	6767.77	1.66	999 191 361 224 140	
b06	4935.17	5723.07	1.16	223 070 824 409 476 680	
s386	5183.01	9639.66	1.86	5 012 263 558 001 254	
camellia	5142.75	4176.15	0.81	77 261 2 470 894	
serial	5221.75	3976.23	0.76	2 085 855 11 653 080	
master	5466.83	4080.29	0.75	252 231 34 812 555	

**Table 5.5:** Results of the execution of algorithms `base` and `base-all` over the real-world dataset. The first column reports the name of the design where the simulation trace is retrieved. The next two columns report the execution time in seconds of algorithms `base` and `base-all`, respectively. The fourth column reports the speedup ratio between the execution times of algorithms `base-all` and `base`. Finally, the last two columns report the number of properties extracted from the `base` and the `base-all` algorithms, respectively.



---

## Greedy minimum-entropy coupling

This chapter is devoted to greedy additive approximation algorithms for minimum-entropy coupling problems.

Given two random variables  $X$  and  $Y$  with probability distributions  $P_x = (p(x_1), p(x_2), \dots, p(x_n))$  and  $P_y = (p(y_1), p(y_2), \dots, p(y_m))$  respectively, the *minimum-entropy coupling* problem is to find the minimum-entropy joint distribution among all possible joint distributions of  $X$  and  $Y$  with marginal distributions equal to  $P_x$  and  $P_y$ , respectively. The minimum entropy coupling problem is known to be NP-hard [87].

By causality discovery, we refer to the problem of identifying the causality direction between correlated random variables. More clearly, given correlation data samples about two phenomena  $A$  and  $B$  (e.g., pairs of values of measured altitude ( $A$ ) and temperature ( $B$ )) the aim is to determine whether it is  $A$  that causes  $B$  or vice versa.

In Pearl's model of causality [105], given two random variables  $X$  and  $Y$ , if  $X$  causes  $Y$  then there exists an exogenous random variable  $E$  independent of  $X$  and a function  $f(\cdot)$  such that  $Y = f(X, E)$ . However, this is not enough for the identification of the causal direction since in general, it is possible to find pairs  $(f, E)$  and  $(g, E')$  such that both  $Y = f(X, E)$  and  $X = g(Y, E')$  hold. In order to identify the correct causal direction between  $X$  and  $Y$ , Kocaoglu *et al.* [83] postulated that in the true causal direction, the entropy of the exogenous random variable  $E$  is small.

The problem of trying to infer causation from correlation is a fundamental problem in many important contexts like network security, portfolio analysis, weather forecast and climate change studies, just to mention a few.

Finding the minimum-entropy joint distribution  $H(X, Y)$  of two random variables  $X$  and  $Y$ , given their marginal distributions, is equivalent to the problem of finding the joint distribution that maximizes the mutual information,  $I(X; Y)$ , due to the well known relationship between these to quantities  $H(XY) = H(X) + H(Y) - I(X; Y)$ .

Finding the minimum-entropy joint distribution also plays a central role in the problem of order-reduction of stochastic processes [134].

### Related work

Kocaoglu *et al.* [83] (independently [103]) proposed a greedy heuristic for the minimum entropy coupling problem and showed that the solution provided by their algorithm is guaranteed to be a local minimum. They conjectured, giving only experimental evidence, that their algorithm provides an additive approximation not larger than 1.

In [84], *Kocaoglu et al.* proposed a variant of the greedy heuristic presented in [83] for which they provide an additive approximation factor that depends on the difference of the sorted marginals, and it can scale with  $\log(n)$ , where  $n$  is the number of states of the marginals.

*Cicalese et al.*[33] gave a different greedy algorithm and proved that it guarantees additive approximation 1 (and, in general,  $\log(m)$ , for a minimum entropy coupling of  $m$  distributions).

### Our contribution

For the case of two random variables  $X, Y$ , we show that the algorithm proposed by *Kocaoglu et al.* [83] also guarantees an additive approximation of 1. We also give another greedy algorithm to obtain a coupling of small entropy when the probability distributions of  $X$  and  $Y$  given as marginals are ordered from the highest to the smallest value. This algorithm provides an additive 1-approximation of a minimum entropy coupling. The interesting aspect of this approach is about the property on which the approximation guarantee is based, and we believe that this property might be of independent interest also in other settings where couplings are used.

The results presented in this chapter appeared in [116].

## 6.1 Basic facts

### 6.1.1 Notation

We represent the set  $\{1, \dots, n\}$  by  $[n]$ . Given an array of  $n$  elements  $a = (a_1, a_2, \dots, a_n)$ , we refer to the  $i$ -th element of  $a$  as  $a_i$  and to the first  $i$  elements of  $a$  as  $a_{[i]}$ . Let  $M$  be an  $n \times n$  matrix, for all  $i, j = 1, \dots, n$  we denote by  $M_{ij}$  the element in the  $i$ -th row and  $j$ -th column of  $M$ . We denote by  $M^{[i]}$ , the  $i \times i$  matrix of the first  $i$  rows and columns of  $M$ .

### 6.1.2 Joint Entropy Minimization Algorithm

Without loss of generality, we can consider the problem of finding the minimum-entropy coupling when the given marginal distributions,  $p = (p_1, p_2, \dots, p_n)$  and  $q = (q_1, q_2, \dots, q_n)$ , have each  $n$  elements. For our purpose, we recall in Algorithm 11, the algorithm provided by *Kocaoglu et al.* [83] in the case of two marginals. The algorithm, at each iteration, finds the largest residual mass probability of the two marginals  $p$  and  $q$ ; places the minimum of them in the position corresponding to their coordinates in the joint probability matrix  $M$ ; updates the residual mass probability of the two marginals, removing from both the maximal elements, the value of the minimum of them already placed in  $M$ . In this case, the algorithm terminates in at most  $2n - 1$  steps.

In order to analyze Algorithm 11, we define  $p^{(i)} = (p_1^{(i)}, \dots, p_n^{(i)})$  as the *residual pieces* of  $p$  after  $i$  calls of the **UpdateRoutine**, e.g.,  $p^{(0)} = p$ . From now on, we will refer to the  $i$ -th call of the **UpdateRoutine** as the  $i$ -th temporal instant, or briefly instant.



---

Algorithm 11: Joint Entropy Minimization Algorithm [83]

---

**Require:** Marginal distributions  $\{p, q\}$  with  $n$  states  
**Ensure:** An  $n \times n$  matrix  $M$  that is coupling of  $p$  and  $q$ .

- 1: Initialize the matrix  $M_{ij} = 0, i, j = 1, \dots, n$ .
- 2: Initialize  $r = 1$ .
- 3: **while**  $r > 0$  **do**
- 4:    $(\{p, q\}, r) = \text{UpdateRoutine}(\{p, q\}, r)$
- 5: **end while**
- 6: **return**  $P$ .
- 7: **UpdateRoutine** $(\{p, q\}, r)$
- 8:  $i = \arg \max_k \{p_k\}$ .
- 9:  $j = \arg \max_k \{q_k\}$ .
- 10:  $M_{ij} = \min\{p_i, q_j\}$ .
- 11:  $p_i = p_i - M_{ij}$ .
- 12:  $q_j = q_j - M_{ij}$ .
- 13:  $r = r - M_{ij}$ .
- 14: **return**  $\{p, q\}, r$

---

### 6.1.3 Majorization

In order to prove that Algorithm 11 guarantees an additive approximation factor of 1, we use the same tool used in [33], i.e. majorization theory[97], of which we are going to recall few notions. Let  $\mathcal{P}_n = \{(p_1, p_2, \dots, p_n) \mid p_1 \geq p_2 \geq \dots \geq p_n \geq 0 \text{ and } \sum_{i=1}^n p_i = 1\}$ .

**Definition 6.1.** Given probability distributions  $p \in \mathcal{P}_n$  and  $q \in \mathcal{P}_n$ , we say that  $p$  is majorized by  $q$ , denoted by  $p \preceq q$  if and only if for all  $k = 1, 2, \dots, n$  it holds that  $\sum_{i=1}^k p_i \leq \sum_{i=1}^k q_i$ .

Given the set  $\mathcal{P}_n$ , the majorization relation  $\preceq$  is a partial ordering on  $\mathcal{P}_n$  and the partially ordered set  $(\mathcal{P}_n, \preceq)$  is a *lattice* [43] (independently [48]). It follows that for all pairs  $p, q \in \mathcal{P}_n$  there exists a unique *greatest lower bound*, denoted by  $p \wedge q$ , i.e.  $p \wedge q \preceq p$  and  $p \wedge q \preceq q$  and for all  $r \in \mathcal{P}_n$  such that  $r \preceq p$  and  $r \preceq q$  then  $r \preceq p \wedge q$ . Given  $p, q \in \mathcal{P}_n$ , it is possible to compute the value of  $p \wedge q$  as shown in [43], that we recall in the following fact.

**Fact 6.1** [43] Given probability distributions  $p = (p_1, p_2, \dots, p_n) \in \mathcal{P}_n$  and  $q = (q_1, q_2, \dots, q_n) \in \mathcal{P}_n$ , let  $z = (z_1, z_2, \dots, z_n) \in \mathcal{P}_n$  such that  $z = p \wedge q$ . Then  $z_1 = \min\{p_1, q_1\}$  and for each  $k = 2, \dots, n$  it holds that

$$z_k = \min\left(\sum_{i=1}^k p_i, \sum_{i=1}^k q_i\right) - z_{k-1}.$$

We are going to summarize in the following lemma an important result provided in [33], i.e., the entropy of any coupling between  $p$  and  $q$  is not smaller than the entropy of the greatest lower bound  $p \wedge q$ .

**Lemma 6.1.** [33] For any  $p$  and  $q$ , let  $M$  be one of the possible couplings between  $p$  and  $q$ . It holds that

$$H(M) \geq H(p \wedge q).$$

This implies that the minimum-entropy coupling matrix  $M^*$  between  $p$  and  $q$  has entropy not smaller than  $H(p \wedge q)$ . To provide a 1-bit approximation for Algorithm 11, it is enough to show that the produced coupling matrix  $M$  has entropy  $H(M) \leq H(p \wedge q) + 1$ .

In order to do that, we introduce two sequences of  $2n$  elements. Let  $s$  be the array of length  $2n$  storing the non-zero elements of  $M$  in non-increasing order. Since the non-zero elements of  $M$  are at most  $2n - 1$ , we fill the remaining elements of  $s$  with 0s. In particular  $s = (s_1, s_2, \dots, s_{2n})$  such that  $s_1 \geq s_2 \geq \dots \geq s_{2n}$  and they represent the elements given by Algorithm 11. Let  $p_*^{(i)}$  and  $q_*^{(i)}$  be the maximum residual piece in  $p^{(i)}$  and  $q^{(i)}$  at instant  $i$ , respectively. Considering those *residual pieces* that are maximal, for  $i = 1, \dots, 2n$  we have that  $s_i = \min(p_*^{(i)}, q_*^{(i)})$  if it exists, otherwise  $s_i = 0$ . Furthermore, let  $hz$  be the array of the two halves of each element of  $z$ . In particular  $hz = (hz_1, hz_2, \dots, hz_{2n})$  such that  $hz_1 \geq hz_2 \geq \dots \geq hz_{2n}$  where, for each  $i = 1, \dots, n$  we have that  $hz_{2i-1} = z_i/2$  and  $hz_{2i} = z_i/2$ . Note that  $H(hz) = H(z) + 1 = H(p \wedge q) + 1$ . Let  $S$  and  $HZ$  be the prefix sums of  $s$  and  $hz$  respectively. In particular, their  $k$ -th elements are given by  $S_k = \sum_{i=1}^k s_i$  and  $HZ_k = \sum_{i=1}^k hz_i$ .

Recalling the Shur-concavity of the entropy function [97], we have that given  $p, q \in \mathcal{P}_n$  such that  $p \preceq q$  then  $H(p) \geq H(q)$ . Thus, showing that for all  $k \in [2n]$ ,  $S_k \geq HZ_k$  we have that  $hz \preceq s$  thus  $H(s) \leq H(hz)$  then  $H(M) \leq H(p \wedge q) + 1$ .

## 6.2 Main Theorem

In this section, we will show that the algorithm summarized in Algorithm 11 guarantees an additive approximation factor of 1. We can assume without loss of generality that the marginal distributions  $p, q \in \mathcal{P}_n$ , thus, they are given in non-increasing order, i.e.,  $p_1 \geq p_2 \geq \dots \geq p_n$  and  $q_1 \geq q_2 \geq \dots \geq q_n$ .

**Theorem 6.1.** *Let  $p, q \in \mathcal{P}_n$  and  $z = p \wedge q$ . Let  $s$  be the distribution obtained by the application of Algorithm 11. Then  $H(s) \leq H(z) + 1$ .*

*Proof.* We are going to show that for all  $k \in [2n]$  it holds that either  $S_k \geq \sum_{i=1}^{\lceil \frac{k}{2} \rceil} p_i$  or  $S_k \geq \sum_{i=1}^{\lceil \frac{k}{2} \rceil} q_i$ . Note that after instant  $k$ , Algorithm 11 modified the residual pieces of at most the first  $k$  elements of  $p$  and  $q$ , i.e. for all  $n \geq i > k$ ,  $p_i^{(k)} = p_i$  and  $q_i^{(k)} = q_i$ . Let  $P^{(k)}$  be the set of indices of residual pieces of  $p$  whose value is 0. Analogously, we can define  $Q^{(k)}$  as the set of indices of residual pieces of  $q$  whose value is 0 at instant  $k$ . Formally,  $P^{(k)} = \{j \in [k] \mid p_j^{(k)} = 0\}$  and  $Q^{(k)} = \{j \in [k] \mid q_j^{(k)} = 0\}$ . We can assume, without loss of generality, that  $|P^{(k)}| \geq |Q^{(k)}|$ . In particular, we have that  $|P^{(k)}| \geq \lceil \frac{k}{2} \rceil$  and we will show that under this condition, it holds that  $S_k \geq \sum_{i=1}^{\lceil \frac{k}{2} \rceil} p_i$ .

We start partitioning the residual pieces in  $P^{(k)}$  according to their indices in  $p$ , i.e. if their index is greater than  $\lceil \frac{k}{2} \rceil$ . Let  $U = \{j \in [\lceil \frac{k}{2} \rceil] \mid p_j^{(k)} = 0\}$ , let  $R = \{j \in [k] \mid n \geq j > \lceil \frac{k}{2} \rceil, p_j^{(k)} = 0\}$ . Furthermore, we define  $W = \{j \in [\lceil \frac{k}{2} \rceil] \mid p_j^{(k)} > 0\}$  that is the set of elements which are among the first  $\lceil \frac{k}{2} \rceil$  elements of  $p$  but their residual piece at instant  $k$  is not 0. Finally, let  $V = \{j \in [k] \mid n \geq j > \lceil \frac{k}{2} \rceil, p_j^{(k)} > 0\}$  that is the set of elements in the last  $\lceil \frac{k}{2} \rceil$  among the first  $k$  elements of  $p$  which their residual piece at

instant  $k$  is not 0. Note that since  $|P^{(k)}| \geq |Q^{(k)}|$  and  $|P^{(k)}| + |Q^{(k)}| \geq k$  then at least  $\lceil \frac{k}{2} \rceil$  elements of  $p$  are 0, thus it holds that  $|W| \leq |R|$ .

Then we can write the prefix sums of  $s$  as follows

$$S_k = \sum_{i \in U} p_i + \sum_{i \in R} p_i + \sum_{i \in W} (p_i - p_i^{(k)}) + \sum_{i \in V} (p_i - p_i^{(k)})$$

We are now going to show that for all  $i \in R$  it holds that for all  $j \in W$ ,  $p_i \geq p_j^{(k)}$ . Assume otherwise, and let  $t < k$  be the instant when the residual piece of  $p_i$  was maximal and smaller than the maximal element of  $q$ , i.e.  $p_i^{(t-1)} > 0$  and  $p_i^{(t)} = 0$ . From this follows that  $p_i^{(t-1)} \geq p_j^{(t-1)} \geq p_j^{(k)}$  in contradiction with the assumptions. Hence, we have that

$$\sum_{i \in R} p_i + \sum_{i \in W} (p_i - p_i^{(k)}) \geq \sum_{i \in W} p_i$$

Keeping all this together,

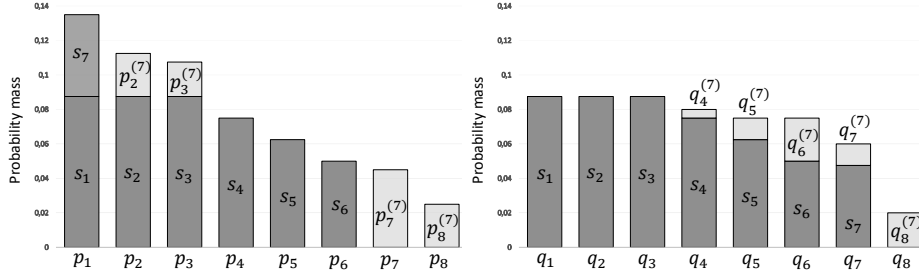
$$S_k \geq \sum_{i \in U} p_i + \sum_{i \in W} p_i = \sum_{j \in [\lceil \frac{k}{2} \rceil]} p_j \geq HZ_k$$

Then we can conclude that  $H(s) \leq H(hz) = H(z) + 1$ .

Hence, we have shown that Algorithm 11 guarantees an additive approximation of 1 in the case of two marginals.

Figure 6.1 shows a pictorial representation of an example of the residual pieces among the first 8 elements of two marginals  $p$  and  $q$  after 7 execution instants, i.e. 7 calls of the **UpdateRoutine**. At instant 1,  $p_1 > q_1$  hence  $p_1$  is split into  $s_1$  and  $p_1^{(1)}$ , while  $q_1$  is entirely covered by  $s_1$ . At instant 2 we have that the largest elements among the residual pieces of  $p$  and  $q$  are  $p_2$  and  $q_2$  respectively. Since  $p_2 > q_2$  we have that  $p_2$  is split into  $s_2$  and  $p_2^{(2)}$ , while  $q_2$  is entirely covered by  $s_2$ . Similarly, we can argue about  $p_3$  and  $q_3$  at instant 3. At instant 4 we have that the largest elements among the residual pieces of  $p$  and  $q$  are  $p_4$  and  $q_4$ , in this case we have that  $p_4 < q_4$  hence  $q_4$  is split into  $s_4$  and  $q_4^{(4)}$ , while  $p_4$  is entirely covered by  $s_4$ . We have the same situation at instants 5 and 6 for the variables  $p_5$  and  $q_5$ , and for the variables  $p_6$  and  $q_6$ . At the last instant, we have that the residual piece of  $p_1$ ,  $p_1^{(6)} < q_7$ , then  $q_7$  is split into  $s_7$  and  $q_7^{(7)}$ , while  $p_1^{(6)}$  is entirely covered by  $s_7$ . Note that the elements  $p_8$  and  $q_8$  are not involved in the computation before instant 8, since there are at least 7 elements greater than them.

In the example we have that  $P^{(7)} = \{1, 4, 5, 6\}$  and  $Q^{(7)} = \{1, 2, 3\}$ , thus, at instant 7, Theorem 6.1 holds for  $p$ . In particular, we have that the sets  $U, R, W, V$  are  $U = \{1, 4\}$ ,  $R = \{5, 6\}$ ,  $W = \{2, 3\}$  and  $V = \{7\}$ . We want to show that  $\sum_{i=1}^7 s_i \geq \sum_{i=1}^4 p_i$ . We have that  $s_4 = p_4$  and  $s_1 + s_7 = p_1$ , hence we have to show that  $s_2 + s_3 + s_5 + s_6 \geq p_2 + p_3$ . Since  $p_2 = s_2 - p_2^{(7)}$  and  $p_3 = s_3 - p_3^{(7)}$  we have to show that  $s_5 + s_6 \geq p_2^{(7)} + p_3^{(7)}$ . Note that  $p_5$  and  $p_6$  are the largest elements of  $p$  at instants 5 and 6 respectively, hence  $p_5 > p_2^{(4)}$  and  $p_6 > p_3^{(5)}$ . Since for all  $i$  and for all  $k$  we have that  $p_i^{(k)} \geq p_i^{(k+1)}$ , we have shown that  $s_5 + s_6 \geq p_2^{(7)} + p_3^{(7)}$ , as in Theorem 6.1.

(a) Residual pieces of  $p_1, \dots, p_8$  after 7 instants (b) Residual pieces of  $q_1, \dots, q_8$  after 7 instants

**Fig. 6.1:** Values of output and residual pieces after 7 instants of execution of Algorithm 11. Dark gray bars  $s_1, s_2, \dots, s_7$  represent the probability mass placed by Algorithm 11 after 7 instants. Light gray bars  $p_2^{(7)}, p_3^{(7)}, p_7^{(7)}, p_8^{(7)}, q_4^{(7)}, q_5^{(7)}, q_6^{(7)}, q_7^{(7)}, q_8^{(7)}$  represents the residual piece of  $p_2, p_3, p_7, p_8, q_4, q_5, q_6, q_7, q_8$  respectively, while the residual pieces  $p_2^{(7)} = p_3^{(7)} = p_7^{(7)} = p_8^{(7)} = q_1^{(7)} = q_2^{(7)} = q_3^{(7)} = 0$ . The sum of the stacked bars  $p_1, p_2, \dots, p_8$  and  $q_1, q_2, \dots, q_8$  represent the probability mass of the first 8 elements of  $p$  and  $q$  respectively.

### 6.3 A General approach for additive approximation on couplings

In this section, we present another algorithm that provides an additive 1-approximation of a minimum entropy coupling. The algorithm, summarized in Algorithm 12, at each step  $i$ , places as much as possible in position  $M_{ii}$ , i.e.  $\min\{p_i, q_i\}$ ; redistributes the residual pieces of the first  $i$  non-satisfied rows/columns as in Algorithm 11, i.e. from the heaviest to the lightest. The main idea behind Algorithm 11 is summarized in the following lemma. We believe that this is an independent property that can be used to derive additive approximations for coupling construction algorithms in other settings.

---

#### Algorithm 12: Diagonal algorithm

---

**Require:** Marginal distributions  $\{p, q\}$  with  $n$  states  
**Ensure:** An  $n \times n$  matrix  $M$  that is coupling of  $p$  and  $q$ .

- 1: **for**  $i = 1, \dots, n$  **do**
- 2:    $M_{ii} = \min\{p_i, q_i\}$
- 3:    $r = z_i - M_{ii}$
- 4:   **while**  $r > 0$  **do**
- 5:      $(\{p_{[i]}, q_{[i]}\}, r) = \text{UpdateRoutine}(\{p_{[i]}, q_{[i]}\}, r)$
- 6:   **end while**
- 7: **end for**

---

First, we prove the following properties of Algorithm 12.

**Lemma 6.2.** *Given two marginal distributions  $p, q \in \mathcal{P}_n$ , let  $M$  be the  $n \times n$  coupling matrix produced by Algorithm 12. The following properties holds.*

1. *For all  $i = 1, \dots, n$ , there are at most  $2i - 1$  non-zero elements in  $M^{[i]}$  and the total mass quantity in  $M^{[i]}$  equals to  $\sum_{j=1}^i z_j$ .*
2. *Furthermore, if an element  $z_i$  is split into  $t+1$  elements (i.e produces  $t$  new elements in  $M^{[i]}$ ) then among  $z_1, \dots, z_{i-1}$  at least  $t$  had not been split.*

*Proof.* In order to show 1), we can observe that each split (i.e. a new element with respect to the original diagonal) means that a new constraint has been satisfied and

there are at most  $i - 1$  new constraints in  $M^{[i]}$ . If  $\sum_{j=1}^i p_j \neq \sum_{j=1}^i q_j$ , then it can be satisfied with  $z_1 + \dots + z_i$  total mass. (Note that if  $\sum_{j=1}^i p_j = \sum_{j=1}^i q_j$  then, two constraints are automatically satisfied, there are still at most  $i - 1$  more to be fulfilled)

In order to prove 2), we can assume that  $z_i$  is split to produce  $t$  more elements and also (by contradiction) less than  $t$  among  $z_1, \dots, z_{i-1}$  were not split, i.e.  $\ell > i - t$  were split into at least two parts. Therefore there would be  $i + \ell + t > i + i - t + t = 2i > 2i - 1$  elements against 1).

We are now ready to show that Algorithm 12 produce a minimum-entropy coupling of two marginal distribution that is a 1-bit approximation.

**Theorem 6.2.** *Given  $p, q \in \mathcal{P}_n$ . Algorithm 12 produce a matrix  $M$  coupling of  $p$  and  $q$ , such that it is an additive approximation with factor 1 with respect to the minimum-entropy coupling matrix  $M^*$ , in particular  $H(M) \leq H(M^*) + 1$ .*

*Proof.* Given Lemma 6.2, let  $\tilde{m}^{(i)} = (m_1^{(i-1)}, \dots, m_k^{(i-1)}, z_i^{(1)}, \dots, z_i^{(t)}, 0)$  be the  $2i - 1$  non-zero elements of  $M^{[i]}$  with an extra zero. In particular,  $m_1^{(i-1)}, \dots, m_k^{(i-1)}$  are the non-zero elements of  $M^{[i-1]}$ , and  $z_i^{(1)}, \dots, z_i^{(t)}$  are the  $t$  elements of  $z_i$  (produced at step 5).

Again from Lemma 6.2, it holds that for all  $i = 1, \dots, n$

$$\sum_{j=1}^{2i-1} \tilde{m}_j^{(i)} = \sum_{j=1}^k \tilde{m}_j^{(i)} + \sum_{j=1}^t z_i^{(j)} = \sum_{j=1}^{i-1} z_j + z_i = \sum_{j=1}^i z_j$$

Thus we have that  $\sum_{j=1}^{2i-1} \tilde{m}_j^{(i)} \geq \sum_{j=1}^{2i-1} H Z_j$  hence  $\sum_{j=1}^{2i} \tilde{m}_j^{(i)} \geq \sum_{j=1}^{2i} H Z_j$ .

We have shown that  $hz \preceq \tilde{m}^{(n)}$  and it follows that  $H(M) \leq H(M^*) + 1$ .

Algorithm 2 by *Kocaoglu et al.*[84], in the case of two marginal distributions whose elements are in non-decreasing order, is similar to Algorithm 12. While Algorithm 12 fills the matrix  $M^{[i]}$  before considering the matrix  $M^{[i+1]}$ , Algorithm 2 by *Kocaoglu et al.*[84] first places the maximal masses in the diagonal of the matrix, then it fills the matrix computing the **UpdateRoutine** on the residual masses in the marginal distributions. Thus, Algorithm 12 and Algorithm 2 by *Kocaoglu et al.*[84] produce different outputs. Furthermore, it is not possible to directly apply the same technique in order to show that Algorithm 2 by *Kocaoglu et al.*[84] also guarantees an additive approximation of 1 of a minimum entropy coupling problem.



## Conclusion

In this thesis we focused on sequential data, studying algorithmics and data structures for problems of coding, indexing, and mining.

In the area of coding, the first topic we addressed is related to Lempel-Ziv 77 parsing. In Chapter 2 we studied the problem of decompressing a Lempel-Ziv 77 parsing in space proportional to the parsing itself. We implemented Bille et al.'s algorithm [16] using Karczmarz's mergeable dictionary [77], extended to support the `shift` operation. The performances of the algorithm were extremely disappointing in terms of time and space consumption, compared to the naive algorithm. We replaced Karczmarz's mergeable dictionary with an ad-hoc data structure tailored for Bille et al.'s algorithm. We further presented a hybrid approach that mixes the naive decompression algorithm and Bille et al.'s approach. The hybrid setting uses a cache to speed up the decompression, storing already decompressed parts of the text that will be highly referred. Thus, when a block has to be decompressed is in the cache, we naively decompress it from the cache, instead of using the slower Bille et al.'s approach. Experimental results, performed on real-world texts, showed that the ad-hoc data structure outperforms the previous implementation, making the time and space consumption of Bille et al.'s algorithm comparable with the one of the naive approach. The results confirm that Bille et al.'s algorithm uses much less space than the naive algorithm, while it uses more time to decompress the text. Furthermore, the hybrid approach results effective in all datasets we tested.

Error-correcting adaptive codes are the second topic in the area of coding that we addressed. We focused on the multi-interval Ulam-Rényi game. In Chapter 3 we investigated the case where up to 3 answers are lies. We first presented our novel analytic tool (Lemma 3.2). We used it to show that for any sufficiently large  $m$ , there exists an optimal strategy to identify an initially unknown  $m$ -bit number that only uses 4-interval queries. We turned this asymptotic result into a complete characterization of instances of the Ulam-Rényi game that admit strategies using the theoretical minimum number of questions, using 4-interval questions when up to 3 answers are lies. For this, we refined our main tool (Theorem 3.1) and we built upon the result of [101], mapping all optimal solutions for the *classical* Ulam-Rényi game with 3 lies, into solutions for the multi-interval variant, that uses 4-interval questions.

In Chapter 4 we focused on prefix normal words in the context of indexing. We proposed a new generation algorithm that lists all prefix normal words of a given length  $n$ . The algorithm runs in  $\mathcal{O}(n)$  time per word and lists the prefix normal words in

lexicographic order. With a slight change in the code, the obtained algorithm lists prefix normal words as a combinatorial Gray code. We also used the generation algorithm to list all prefix normal words sharing the same critical prefix. We presented a closed formula for counting some of the sets sharing specific critical prefixes. We introduced infinite prefix normal words and we showed that the operation *flipext*, one of the operations used in the generation algorithm, if repeatedly applied, generates infinite prefix normal words, starting from a seed word. We showed that the word generated in this way is an ultimately periodic infinite word, for which we can predict the length and the density of the period. We defined another operation called *lazy- $\alpha$ -flipext* that can be used to generate infinite prefix normal words that are Sturmian words. We then showed that the only Sturmian words that are prefix normal are the ones produced using the *lazy- $\alpha$ -flipext*. This leads to a complete characterization of which infinite prefix normal words are Sturmian words. We established connections between infinite prefix normal words and lexicographic order, showing that, as in the finite case, infinite prefix normal words are infinite prenecklaces and that there are infinite binary words that are: both prefix normal and Lyndon, prefix normal but not Lyndon, Lyndon but not prefix normal, and neither of the two. Furthermore, we compared prefix normal words with the max- and min-words of [110]. We also established connections between infinite prefix normal words and abelian complexity, showing that it is always possible to compute the abelian complexity function of a word, given their prefix normal forms. We also presented sufficient conditions to obtain the prefix normal forms of a word given its abelian complexity. Furthermore, we extended a recent result on computing the abelian complexity function of binary uniform morphisms to compute the prefix normal forms of binary uniform morphisms. Finally, we characterized periodicity and aperiodicity of infinite prefix normal words according to their minimum density.

In the area of mining of sequential data, in Chapter 5 we treated problems of pattern discovery in colored strings a new type of strings we proposed. Motivated by applications in embedded system verification. These are strings such that each position of the string is assigned a color from a finite set of colors. We studied two different pattern discovery problems on colored strings. The first problem is to find all minimally  $(y, d)$ -unique substrings of the colored string, for a given color  $y$  and any delay  $d$ . We proposed two different approaches, which we refer to as baseline approach and skipping approach. Both algorithms use a suffix tree on the reverse of the colored string as underlying data structure. They discover the patterns starting from the ones with highest delay to the ones with the lowest delay. The two algorithms differ in the way in which minimality information is propagated along the suffix tree. The baseline algorithm traverses the whole tree separately for each delay value, propagating a coloring function from the leaves to the root of the suffix tree. During each traversal, the algorithm goes through all distinct substrings of the text, and uses the coloring function to identify which substrings are minimally  $(y, d)$ -unique. On the other hand, the skipping algorithm stores, for each distinct substring, the next delay value such that the substring is  $(y, d)$ -unique, during the discovery process. It uses a maximum-oriented indexed priority queue to find these values and to identify minimally  $(y, d)$ -unique substrings. Even though the theoretical analysis we provided for the skipping algorithm results in a worst upper bound on the running time than the one for the baseline algorithm, we show in our experiments that it is faster in practice, especially on real-life instances. We also proposed a variant of the minimality condition oriented toward real-world application instances. Those conditions allow us to develop a faster core function of the skipping



algorithm, resulting in a more effective performance in practice. The second problem we proposed is to find all minimally  $(y, d)$ -unique substrings of the colored string, for all colors  $y$  concurrently. We modified our baseline algorithm, defining a new coloring function, noting that for fixed  $d$ , a substring can be  $(y, d)$ -unique at most for one color  $y$ . The introduction of the new coloring function and the fact that now all colors  $y$  are of interest, increases the running time with respect to the baseline algorithm only negligibly, an effect we observed in the experiments on randomly generated data, while only in half of the cases for real-world data.

In the last chapter of this thesis we considered the problem of minimum entropy coupling. We focused on algorithms for this problem with additive approximation guarantees. We showed that for the case of two random variables  $X, Y$ , the algorithm proposed by *Kocaoglu et al.* [83] also guarantees an additive approximation of 1. We presented another greedy algorithm to obtain a coupling of small entropy when the probability distributions in the marginals are ordered from the highest to the smallest value. This algorithm provides an additive 1-approximation of a minimum entropy coupling. The main interesting aspect is about the property that we used, that we believe it might be of independent interest also in other settings where couplings are used.

## 7.1 Future work

We conclude this chapter with suggestions of directions for further research that this thesis opened up.

On the Lempel-Ziv 77 decompression algorithm side, one further direction of research is to improve the performance of the bottleneck phase, caused by the mergeable dictionary data structure. In Chapter 2 we presented an ad-hoc data structure that speeds up the decompression process. This data structure cannot be used as a mergeable dictionary, and it does not satisfy the theoretical bounds required by the decompression algorithm to guarantee its linearity. An interesting question is to find a mergeable dictionary with the correct theoretical bounds to ensure the linearity of the decompression algorithm, which is also a practical solution for the mergeable dictionary problem.

The results on the multi-interval Ulam-Rényi game, presented in Chapter 3, open different research directions. We showed that if up to 3 errors are allowed then there exists an optimal strategy that uses 4-interval questions, thus  $k_3 \leq 4$ . One open question is whether the bound for  $k_3$  is tight, i.e.  $k_3 = 4$ . More interesting, by our novel analytic tool (Lemma 3.2) naturally lends itself to a generalization to any fixed  $e$ . The main open question is how to generalize Theorem 3.1 in order to prove the linearity conjecture  $k_e = O(e)$ .

Open problems on prefix normal words have been already stated in Chapter 4. Among those, the most important questions concern counting the number of prefix normal words of a given length. There are other questions left open in Chapter 4. Leveraging on the practical improvements of the generation algorithm in Section 4.2.4, the first question that arises from Chapter 4 is whether there exists a CAT<sup>1</sup> generation algorithm for prefix normal words. From the observations in Section 4.3 we have that all possible extensions of a prefix normal word  $w$ , preserving the minimum density of  $w$ , are lexicographically between  $\text{flipext}(w)$  and  $\text{lazy-}\alpha\text{-flipext}(w)$ , where  $\alpha = \delta(w)$ .

<sup>1</sup> Remark that in combinatorial generation, the cost to generate one word, without outputting it in the best case is constant amortized time (CAT).

One interesting investigation could be the characterization of all prefix normal words that lie in between.

Colored strings are strings that we used to model the problem of assertion mining, in Chapter 5. It would be interesting to study the number of pattern of interest on average. To further connect the pattern discovery problem with the assertion mining problem, one should investigate other output restrictions, such as to bound the minimal and maximal length of a substring, as well as to bound the minimal and maximal value of the delay. Furthermore, it would be interesting to extend the discovery problem to search for subsequences.

In Chapter 6 we investigated the problem of providing approximation guarantees for greedy algorithms that solve minimum entropy coupling problems. In particular, we focused on the case where the number of the considered random variables is 2, but the greedy algorithm proposed in [84] can be extended in the case where  $m$  random variables are considered. The question is whether the approximation guarantee can be extended in the case of  $m$  random variables. Furthermore, in Section 6.3 we propose a criterion that exploits the structural properties of the coupling matrix. This criterion is sufficient to provide that the coupling matrix is a 1-bit approximation of the optimal coupling. This criterion does not cover the case of the algorithm proposed in [84], but we wonder whether there is another structural property that is also necessary to provide that the coupling matrix is a 1-bit approximation of the optimal coupling. Finally, the criterion is stated only for the case of 2 random variables. The question is if it can also be extended for the case with more than two variables.

---

## References

- [1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. 20th International Conference on Very Large Data Bases (VLDB)*, volume 1215, pages 487–499. Morgan Kaufmann, 1994.
- [2] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proc. Eleventh International Conference on Data Engineering (ICDE)*, volume 95, pages 3–14. IEEE Computer Society, 1995.
- [3] R. Ahlswede, F. Cicalese, C. Deppe, and U. Vaccaro. Two batch search with lie cost. *IEEE Transactions on Information Theory*, 55(4):1433–1439, 2009.
- [4] A. Al-Hafeedh, M. Crochemore, L. Ilie, E. Kopylova, W. Smyth, G. Tischler, and M. Yusufu. A Comparison of Index-Based Lempel-Ziv LZ77 Factorization Algorithms. *ACM Computing Surveys*, 45(1):5:1–5:17, 2012.
- [5] A. Amir, A. Apostolico, T. Hirst, G. M. Landau, N. Lewenstein, and L. Rozenberg. Algorithms for Jumbled Indexing, Jumbled Border and Jumbled Square on run-length encoded strings. *Theoretical Computer Science*, 656:146–159, 2016.
- [6] A. Amir, A. Butman, and E. Porat. On the relationship between histogram indexing and block-mass indexing. *Philosophical Transactions of The Royal Society A: Mathematical Physical and Engineering Sciences*, 372(2016):1–14, 2014.
- [7] A. Amir, T. M. Chan, M. Lewenstein, and N. Lewenstein. On Hardness of Jumbled Indexing. In *Proc. 41st International Colloquium on Automata, Languages, and Programming (ICALP 2014)*, volume 8572 of LNCS, pages 114–125. Springer, 2014.
- [8] A. Apostolico, M. Crochemore, M. Farach-Colton, Z. Galil, and S. Muthukrishnan. Forty Years of Suffix Trees. *Commun. ACM*, 59(4):66–73, 2016.
- [9] V. Auletta, A. Negro, and G. Parlati. Some results on searching with lies. In *Proc. 4th Italian Conf. on Theoretical Computer Science*, pages 24–37. World Scientific, 1992.
- [10] G. Badkobeh, M. Crochemore, and C. Toopsuwan. Computing the Maximal-Exponent Repeats of an Overlap-Free String in Linear Time. In *Proc. 19th Symposium on String Processing and Information Retrieval (SPIRE 2012)*, volume 7608 of LNCS, pages 61–72. Springer, 2012.
- [11] P. Balister and S. Gerke. The asymptotic number of prefix normal words. *Theoretical Computer Science*, 784:75–80, 2019.

- [12] D. Belazzougui, T. Gagie, P. Gawrychowski, J. Kärkkäinen, A. O. Pereira, S. J. Puglisi, and Y. Tabei. Queries on LZ-bounded encodings. In *Proc. 2015 Data Compression Conference (DCC)*, pages 83–92. IEEE, 2015.
- [13] D. Belazzougui and S. J. Puglisi. Range Predecessor and Lempel-Ziv Parsing. In *Proc. 27th Annual Symposium on Discrete Algorithms (SODA)*, pages 2053–2071. SIAM, 2016.
- [14] M. Ben-Or and A. Hassidim. The bayesian learner is optimal for noisy binary search (and pretty good for quantum as well). In *Proc. 2008 49th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 221–230. IEEE, 2008.
- [15] E. R. Berlekamp. Block coding for the binary symmetric channel with noiseless, delayless feedback. *Error-correcting codes*, pages 61–68, 1968.
- [16] P. Bille, M. B. Ettienne, T. Gagie, I. L. Gørtz, and N. Prezza. Fast Lempel-Ziv Decompression in Linear Space. *arXiv preprint arXiv:1802.10347*, 2018.
- [17] F. Birzele and S. Kramer. A new representation for protein secondary structure prediction based on frequent patterns. *Bioinformatics*, 22(21):2628–2634, 2006.
- [18] F. Blanchet-Sadri, N. Fox, and N. Rampersad. On the asymptotic abelian complexity of morphic words. *Advances in Applied Mathematics*, 61:46–84, 2014.
- [19] F. Blanchet-Sadri, D. Seita, and D. Wise. Computing abelian complexity of binary uniform morphic words. *Theoretical Computer Science*, 640:41–51, 2016.
- [20] A. Blondin Massé, J. de Carufel, A. Goupil, M. Lapointe, É. Nadeau, and É. Vandomme. Leaf realization problem, caterpillar graphs and prefix normal words. *Theoretical Computer Science*, 732:1–13, 2018.
- [21] M. Braverman, J. Mao, and S. M. Weinberg. Parallel algorithms for select and partition with noisy comparisons. In *Proc. forty-eighth annual ACM symposium on Theory of Computing (STOC)*, pages 851–862. ACM, 2016.
- [22] F. Brglez, D. Bryan, and K. Kozminski. Combinational profiles of sequential benchmark circuits. In *Proc. 1989 IEEE international symposium on circuits and systems (ISACS)*, volume 3, pages 1929–1934. IEEE, 1989.
- [23] P. Burcsi, F. Cicalese, G. Fici, and Zs. Lipták. Algorithms for Jumbled Pattern Matching in Strings. *International Journal of Foundations of Computer Science*, 23:357–374, 2012.
- [24] P. Burcsi, G. Fici, Zs. Lipták, F. Ruskey, and J. Sawada. On Combinatorial Generation of Prefix Normal Words. In *Proc. 25th Annual Symposium on Combinatorial Pattern Matching (CPM 2014)*, volume 8486 of *LNCS*, pages 60–69. Springer, 2014.
- [25] P. Burcsi, G. Fici, Zs. Lipták, F. Ruskey, and J. Sawada. On prefix normal words and prefix normal forms. *Theoretical Computer Science*, 659:1–13, 2017.
- [26] J. Cassaigne and I. Kaboré. Abelian complexity and frequencies of letters in infinite words. *Int. Journal of Foundations of Computer Science*, 27(05):631–649, 2016.
- [27] N. Cesa-Bianchi, Y. Freund, D. Haussler, D. P. Helmbold, R. E. Schapire, and M. K. Warmuth. How to use expert advice. *Journal of the ACM (JACM)*, 44(3):427–485, 1997.
- [28] S. Chan, B. Kao, C. L. Yip, and M. Tang. Mining emerging substrings. In *Proc. Eighth International Conference on Database Systems for Advanced Applications (DASFAA 2003)*, pages 119–126. IEEE, 2003.

- [29] T. M. Chan and M. Lewenstein. Clustered Integer 3SUM via Additive Combinatorics. In *Proc. 47th Annual ACM on Symposium on Theory of Computing (STOC 2015)*, pages 31–40. ACM, 2015.
- [30] C.-W. Cho, Y. Zheng, Y.-H. Wu, and A. L. Chen. A tree-based approach for event prediction using episode rules over event streams. In *Proc. International Conference on Database and Expert Systems Applications (DEXA 2008)*, volume 5181 of *LNCS*, pages 225–240. Springer, 2008.
- [31] F. Cicalese. *Fault-Tolerant Search Algorithms*. Springer-Verlag, 2013.
- [32] F. Cicalese. Perfect strategies for the Ulam-Rényi game with multi-interval questions. *Theory of Computing Systems*, 54(4):578–594, 2014.
- [33] F. Cicalese, L. Gargano, and U. Vaccaro. How to find a joint probability distribution of minimum entropy (almost) given the marginals. In *Proc. 2017 IEEE International Symposium on Information Theory (ISIT 2017)*, pages 2173–2177. IEEE, 2017.
- [34] F. Cicalese, E. S. Laber, O. Weimann, and R. Yuster. Approximating the maximum consecutive subsums of a sequence. *Theoretical Computer Science*, 525:130–137, 2014.
- [35] F. Cicalese, Zs. Lipták, and M. Rossi. Bubble-Flip—A New Generation Algorithm for Prefix Normal Words. In *Proc. 12th International Conference Language and Automata Theory and Applications (LATA 2018)*, volume 10792 of *LNCS*. Springer, 2018.
- [36] F. Cicalese, Zs. Lipták, and M. Rossi. Bubble-Flip—A new generation algorithm for prefix normal words. *Theoretical Computer Science*, 743:38–52, 2018.
- [37] F. Cicalese, Zs. Lipták, and M. Rossi. On infinite prefix normal words. In *Proc. International Conference on Current Trends in Theory and Practice of Informatics (SOFSEM 2019)*, volume 11376 of *LNCS*, pages 122–135. Springer, 2019.
- [38] F. Cicalese and D. Mundici. Learning and the art of fault-tolerant guesswork. In *Adaptivity and Learning - An Interdisciplinary Debate*, pages 115–140. Springer, 2003.
- [39] F. Cicalese and D. Mundici. Recent developments of feedback coding and its relations with many-valued logic. In *Proof, Computation and Agency*, volume 352, pages 115–131. Springer-Verlag Synthese Library, 2011.
- [40] F. Cicalese, D. Mundici, and U. Vaccaro. Rota-Metropolis cubic logic and Ulam-Rényi games. In *Algebraic Combinatorics and Computer Science—A Tribute to Giancarlo Rota*, pages 197–244. Springer Italia, 2001.
- [41] F. Cicalese and M. Rossi. On the Multi-Interval Ulam-Rényi Game: for 3 lies 4 intervals suffice. In *Proc. of 18th Italian Conference on Theoretical Computer Science (ICTCS 2017)*, volume 1949, pages 39–50. CEUR-WS, 2017.
- [42] F. Cicalese and M. Rossi. On the Multi-Interval Ulam-Rényi Game: for 3 lies 4 intervals suffice. *Theoretical Computer Science*, 809:339–356, 2020.
- [43] F. Cicalese and U. Vaccaro. Supermodularity and subadditivity properties of the entropy on the majorization lattice. *IEEE Transactions on Information Theory*, 48(4):933–938, 2002.
- [44] D. Clark. *Compact PAT trees*. PhD thesis, University of Waterloo, 1997.
- [45] F. Corno, M. S. Reorda, and G. Squillero. Rt-level itc’99 benchmarks and first atpg results. *IEEE Design & Test of computers*, 17(3):44–53, 2000.

- [46] T. M. Cover and J. A. Thomas. *Elements of information theory*. John Wiley & Sons, 2012.
- [47] M. Crochemore and W. Rytter. *Jewels of Stringology*. World Scientific, 2003.
- [48] P. Cuff, T. Cover, G. Kumar, and L. Zhao. A lattice of gambles. In *Proc. 2011 IEEE International Symposium on Information Theory Proceedings (ISIT 2013)*, pages 1762–1766. IEEE, 2011.
- [49] L. F. I. Cunha, S. Dantas, T. Gagie, R. Wittler, L. A. B. Kowada, and J. Stoye. Faster Jumbled Indexing for Binary Run-Length Encoded Strings. In *Proc. 28th Annual Symposium on Combinatorial Pattern Matching (CPM 2017)*, volume 78 of *LIPICs*, pages 19:1–19:9, 2017.
- [50] A. Danese, N. Dalla Riva, and G. Pravadelli. A-team: Automatic template-based assertion miner. In *Proc. 2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2017.
- [51] A. Danese, T. Ghasempouri, and G. Pravadelli. Automatic extraction of assertions from execution traces of behavioural models. In *Proc. 2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 67–72. IEEE, 2015.
- [52] C. Davis and D. Knuth. Number representations and dragon curves, I, II. *J. Recr. Math.*, 3:133–149 and 161–181, 1970.
- [53] C. Deppe. Coding with feedback and searching with lies. In *Entropy, Search, Complexity*, volume 16, pages 27–70. Bolyai Society Math. Studies, 2007.
- [54] J. Dhaliwal, S. J. Puglisi, and A. Turpin. Practical efficient string mining. *IEEE Transactions on Knowledge and Data Engineering*, 24(4):735–744, 2010.
- [55] L. Fahed, A. Brun, and A. Boyer. DEER: Distant and Essential Episode Rules for early prediction. *Expert Systems with Applications*, 93:283–298, 2018.
- [56] M. Farach and M. Thorup. String matching in Lempel-Ziv compressed strings. *Algorithmica*, 20(4):388–404, 1998.
- [57] G. Fici and Zs. Lipták. On prefix normal words. In *Proc. 15th Intern. Conf. on Developments in Language Theory (DLT 2011)*, volume 6795 of *LNCS*, pages 228–238. Springer, 2011.
- [58] J. Fischer and V. Heun. Space-Efficient Preprocessing Schemes for Range Minimum Queries on Static Arrays. *SIAM J. Comput.*, 40(2):465–492, 2011.
- [59] J. Fischer, V. Heun, and S. Kramer. Fast frequent string mining using suffix arrays. In *Proc. Fifth IEEE International Conference on Data Mining (ICDM 2005)*, pages 609–612. IEEE, 2005.
- [60] J. Fischer, V. Heun, and S. Kramer. Optimal string mining under frequency constraints. In *Proc. European Conference on Principles of Data Mining and Knowledge Discovery (PKDD 2006)*, volume 4213 of *LNCS*, pages 139–150. Springer, 2006.
- [61] J. Fischer, V. Mäkinen, and N. Välimäki. Space efficient string mining under frequency constraints. In *Proc. Eighth IEEE International Conference on Data Mining (ICDM 2008)*, pages 193–202. IEEE, 2008.
- [62] P. Fleischmann, D. Nowotka, M. Kulczynski, and D. B. Poulsen. On collapsing prefix normal words. In *To appear in the Proc. 14th Int. Conf. on Combinatorics on Language and Automata Theory and Applications (LATA 2020)*, 2020.
- [63] H. D. Foster, A. C. Krolnik, and D. J. Lacey. *Assertion-based design*. Springer Science & Business Media, 2004.

- [64] P. Fournier-Viger, J. C.-W. Lin, R. U. Kiran, Y. S. Koh, and R. Thomas. A survey of sequential pattern mining. *Data Science and Pattern Recognition*, 1(1):54–77, 2017.
- [65] T. Gagie, D. Hermelin, G. M. Landau, and O. Weimann. Binary jumbled pattern matching on trees and tree-like structures. *Algorithmica*, 73(3):571–588, 2015.
- [66] E. Giaquinta and S. Grabowski. New algorithms for binary jumbled pattern matching. *Inf. Process. Lett.*, 113(14–16):538–542, 2013.
- [67] S. Gog, T. Beller, A. Moffat, and M. Petri. From Theory to Practice: Plug and Play with Succinct Data Structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*, volume 8504 of *LNCS*, pages 326–337. Springer, 2014.
- [68] K. Goto and H. Bannai. Simpler and Faster Lempel Ziv Factorization. In *Proc. 2013 Data Compression Conference (DCC)*, pages 133–142. IEEE Computer Society, 2013.
- [69] D. Gusfield. *Algorithms on Strings, Trees, and Sequences : Computer Science and Computational Biology*. Cambridge University Press, Cambridge, United Kingdom, 1997.
- [70] W. Guzicki. Ulam’s searching game with two lies. *Journal of Combinatorial Theory, Series A*, 54(1):1–19, 1990.
- [71] J. Han, J. Pei, and M. Kamber. *Data mining: concepts and techniques*. Elsevier, 2011.
- [72] L. C. K. Hui. Color Set Size Problem with Application to String Matching. In *Proc. Third Annual Symposium on Combinatorial Pattern Matching (CPM 1992)*, volume 644 of *LNCS*, pages 230–243. Springer, 1992.
- [73] J. Iacono and Ö. Özkan. Mergeable Dictionaries. In *37th Int. Colloquium on Automata, Languages and Programming (ICALP 2010)*, volume 6198 of *LNCS*, pages 164–175. Springer, 2010.
- [74] K. Iwanuma, R. Ishihara, Y. Takano, and H. Nabeshima. Extracting frequent subsequences from a single long data sequence a novel anti-monotonic measure and a simple on-line algorithm. In *Proc. Fifth IEEE International Conference on Data Mining (ICDM’05)*, pages 186–195. IEEE, 2005.
- [75] X. Ji, J. Bailey, and G. Dong. Mining minimal distinguishing subsequence patterns with gap constraints. *Knowledge and Information Systems*, 11(3):259–286, 2007.
- [76] I. Kaboré and B. Kientéga. Abelian Complexity of Thue-Morse Word over a Ternary Alphabet. In *Proc. 11th Int. Conf. on Combinatorics on Words WORDS 2017*, volume 10432 of *LNCS*, pages 132–143. Springer, 2017.
- [77] A. Karczmarz. A simple mergeable dictionary. In *Proc. 15th Scandinavian Symp. on Algorithm Theory (SWAT)*, volume 53 of *LIPICs*, pages 7:1–7:13, 2016.
- [78] E. Kelareva. Adaptive psychophysical procedures and Ulam’s game. Master’s thesis, Australian National University, 2006.
- [79] E. Kelareva, J. Mewing, A. Turpin, and A. Wirth. Adaptive psychophysical procedures, loss functions, and entropy. *Attention, Perception, & Psychophysics*, 72(7):2003–2012, 2010.
- [80] D. Kempa and S. J. Puglisi. Lempel-Ziv factorization: Simple, fast, practical. In *Proc. 2013 Proceedings of the Fifteenth Workshop on Algorithm Engineering and Experiments (ALENEX13)*, pages 103–112. SIAM, 2013.

- [81] D. E. Knuth. *The art of computer programming*, volume 3. Pearson Education, 1997.
- [82] D. E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 3: Generating All Combinations and Partitions*. Addison-Wesley Professional, 2005.
- [83] M. Kocaoglu, A. G. Dimakis, S. Vishwanath, and B. Hassibi. Entropic causal inference. In *Proc. Thirty-First AAAI Conference on Artificial Intelligence*, pages 1156–1162. AAAI Press, 2017.
- [84] M. Kocaoglu, A. G. Dimakis, S. Vishwanath, and B. Hassibi. Entropic causality and greedy minimum entropy coupling. In *Proc. 2017 IEEE International Symposium on Information Theory (ISIT 2017)*, pages 1465–1469. IEEE, 2017.
- [85] T. Kociumaka, J. Radoszewski, and W. Rytter. Efficient indexes for jumbled pattern matching with constant-sized alphabet. *Algorithmica*, 77(4):1194–1215, 2017.
- [86] R. Kolpakov and G. Kucherov. Finding Maximal Repetitions in a Word in Linear Time. In *Proc. 40th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 596–604. IEEE Computer Society, 1999.
- [87] M. Kovačević, I. Stanojević, and V. Šenk. On the entropy of couplings. *Information and Computation*, 242:369–382, 2015.
- [88] W. Kozma and L. Lazos. Dealing with liars: Misbehavior identification via Rényi-Ulam games. In *Proc. International Conference on Security and Privacy in Communication Systems*, volume 19, pages 207–227. Springer, 2009.
- [89] S. Kreft and G. Navarro. On compressing and indexing repetitive sequences. *Theoretical Computer Science*, 483:115–133, 2013.
- [90] S. Laxman, V. Tankasali, and R. W. White. Stream prediction using a generative model based on frequent episodes in event sequences. In *Proc. 14th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD 2008)*, pages 453–461. ACM, 2008.
- [91] L. Liu, D. Sheridan, V. Athavale, and S. Vasudevan. Automatic generation of assertions from system level design using data mining. In *Proc. Ninth ACM/IEEE International Conference on Formal Methods and Models for Codesign*, pages 191–200. IEEE Computer Society, 2011.
- [92] M. Lothaire. *Algebraic Combinatorics on Words*. Cambridge Univ. Press, 2002.
- [93] N. R. Mabroukeh and C. I. Ezeife. A taxonomy of sequential pattern mining algorithms. *ACM Computing Surveys (CSUR)*, 43(1):3, 2010.
- [94] B. Madill and N. Rampersad. The abelian complexity of the paperfolding word. *Discrete Mathematics*, 313(7):831–838, 2013.
- [95] V. Mäkinen, D. Belazzougui, F. Cunial, and A. I. Tomescu. *Genome-Scale Algorithm Design*. Cambridge University Press, 2015.
- [96] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovery of frequent episodes in event sequences. *Data mining and knowledge discovery*, 1(3):259–289, 1997.
- [97] A. W. Marshall, I. Olkin, and B. C. Arnold. *Inequalities: theory of majorization and its applications*, volume 143. Springer, 1979.
- [98] T. M. Moosa and M. S. Rahman. Sub-quadratic time and linear space data structures for permutation matching in binary strings. *J. Discr. Alg.*, 10:5–9, 2012.
- [99] D. Mundici and A. Trombetta. Optimal comparison strategies in Ulam’s searching game with two errors. *Theoretical Computer Science*, 182(1-2):217–232, 1997.



- [100] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys (CSUR)*, 39(1):2, 2007.
- [101] A. Negro and M. Sereno. Ulam’s searching game with three lies. *Advances in Applied Mathematics*, 13(4):404–428, 1992.
- [102] OpenCores. Available at <https://opencores.org/>. Accessed 05-03-2019.
- [103] A. Painsky, S. Rosset, and M. Feder. Memoryless representation of markov processes. In *Proc. 2013 IEEE International Symposium on Information Theory (ISIT 2013)*, pages 2294–298. IEEE, 2013.
- [104] T. Pang, L. Duan, J. Li-Ling, and G. Dong. Mining Similarity-Aware Distinguishing Sequential Patterns from Biomedical Sequences. In *Proc. Second International Conference on Data Science in Cyberspace (DSC 2017)*, pages 43–52. IEEE, 2017.
- [105] J. Pearl. *Causality*. Cambridge university press, 2009.
- [106] J. Pearl, M. Glymour, and N. P. Jewell. *Causal inference in statistics: A primer*. John Wiley & Sons, 2016.
- [107] J. Pei, J. Han, and W. Wang. Constraint-based sequential pattern mining: the pattern-growth methods. *Journal of Intelligent Information Systems*, 28(2):133–160, 2007.
- [108] A. Pelc. Coding with bounded error fraction. *Ars Combinatoria*, 24:17–22, 1987.
- [109] A. Pelc. Searching games with errors—fifty years of coping with liars. *Theoretical Computer Science*, 270(1-2):71–109, 2002.
- [110] G. Pirillo. Inequalities characterizing standard sturmian and episturmian words. *Theoretical Computer Science*, 341(1-3):276–292, 2005.
- [111] S. J. Puglisi and M. Rossi. On Lempel-Ziv decompression in small space. In *Proc. 2019 Data Compression Conference (DCC)*, pages 221–230. IEEE, 2019.
- [112] R. Raman and Zs. Lipták. Personal communication.
- [113] A. Raza and S. Kramer. Accelerating pattern-based time series classification: a linear time and space string mining approach. *Knowledge and Information Systems*, pages 1–29, 2019.
- [114] A. Rényi. On a problem of information theory. *MTA Mat. Kut. Int. Kozl. B*, 6:505–516, 1961.
- [115] G. Richomme, K. Saari, and L. Q. Zamboni. Abelian complexity of minimal subshifts. *J. London Math. Society*, 83(1):79–95, 2011.
- [116] M. Rossi. Greedy additive approximation algorithms for minimum-entropy coupling problem. In *2019 IEEE International Symposium on Information Theory (ISIT 2019)*, pages 1127–1131. IEEE, 2019.
- [117] F. Ruskey. *Combinatorial Generation*. 2003.
- [118] F. Ruskey, C. Savage, and T. Wang. Generating necklaces. *J. Algorithms*, 13(3):414–430, 1992.
- [119] F. Ruskey, J. Sawada, and A. Williams. Binary bubble languages and cool-lex order. *J. Comb. Theory, Ser. A*, 119(1):155–169, 2012.
- [120] J. Sawada and A. Williams. Efficient oracles for generating binary bubble languages. *Electr. J. Comb.*, 19(1):P42, 2012.
- [121] J. Sawada, A. Williams, and D. Wong. Inside the Binary Reflected Gray Code: Flip-Swap languages in 2-Gray code order. Unpublished manuscript, 2017.
- [122] R. Sedgewick and K. Wayne. *Algorithms*. Addison-Wesley Professional, 2011.

- [123] R. Siromoney, L. Mathew, V. Dare, and K. Subramanian. Infinite Lyndon words. *Inf. Proc. Letters*, 50:101–104, 1994.
- [124] N. J. A. Sloane. The On-Line Encyclopedia of Integer Sequences. Available electronically at <http://oeis.org>.
- [125] B. Smyth. *Computing Patterns in Strings*. Pearson Addison-Wesley, Essex, England, 2003.
- [126] J. Spencer. Guess a number with lying. *Mathematics Magazine*, 57(2):105–108, 1984.
- [127] J. Spencer. Ulam’s searching game with a fixed number of lies. *Theoretical Computer Science*, 95(2):307–321, 1992.
- [128] Z. D. Stephens, S. Y. Lee, F. Faghri, R. H. Campbell, C. Zhai, M. J. Efron, R. Iyer, M. C. Schatz, S. Sinha, and G. E. Robinson. Big data: astronomical or genetical? *PLoS biology*, 13(7):e1002195, 2015.
- [129] A. Tietäväinen. On the nonexistence of perfect codes over finite fields. *SIAM Journal on Applied Mathematics*, 24(1):88–96, 1973.
- [130] O. Turek. Abelian complexity of the Tribonacci word. *J. of Integer Sequences*, 18, 2015.
- [131] S. M. Ulam. *Adventures of a Mathematician*. Scribner’s, New York, 1976.
- [132] N. Välimäki and S. J. Puglisi. Distributed string mining for high-throughput sequencing data. In *Proc. 12th International Workshop on Algorithms in Bioinformatics (WABI 2012)*, volume 7534 of *LNCS*, pages 441–452. Springer, 2012.
- [133] S. Vasudevan, D. Sheridan, S. Patel, D. Tcheng, B. Tuohy, and D. Johnson. Goldmine: Automatic assertion generation using data mining and static analysis. In *Proc. 2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, pages 626–629. IEEE, 2010.
- [134] M. Vidyasagar. A metric between probability distributions on finite sets of different cardinalities and applications to order reduction. *IEEE Transactions on Automatic Control*, 57(10):2464–2477, 2012.
- [135] X. Wang, L. Duan, G. Dong, Z. Yu, and C. Tang. Efficient mining of density-aware distinguishing sequential patterns with gap constraints. In *International Conference on Database Systems for Advanced Applications (DASFAA 2014)*, volume 8421 of *LNCS*, pages 372–387. Springer, 2014.
- [136] P. Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory (SWAT 1973)*, pages 1–11. IEEE, 1973.
- [137] T. A. Welch. A technique for high-performance data compression. *Computer*, (6):8–19, 1984.
- [138] X. Wu, X. Zhu, Y. He, and A. N. Arslan. PMBC: Pattern mining from biological sequences with wildcard constraints. *Computers in Biology and Medicine*, 43(5):481–492, 2013.
- [139] J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.
- [140] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE transactions on Information Theory*, 24(5):530–536, 1978.