Towards Enhanced Decoding of Polar Codes and PAC Codes

Mohammad Rowshan
B.Eng. (Hons), M.Sc.

Supervisor

Prof. Emanuele Viterbo

A thesis submitted for the degree of

Doctor of Philosophy

to

Department of Electrical and Computer Systems Engineering
Monash University, Clayton, Victoria, Australia

July 2021

Copyright Notice

"The more I know, the more I realize I know nothing."

— Socrates

Abstract

Error control coding refers to the techniques that enable the reliable delivery of digital data over unreliable communication channels and storage media. Among various codes available for channel coding, polar codes are the first capacity-achieving codes with explicit construction. Due to various advantages, the 3rd Generation Partnership Project (3GPP) adopted polar codes for the control channel of the enhanced mobile broadband (eMBB) in the 5th generation (5G) of the mobile communications standard. Due to the demapping (or channel splitting) stage in the decoding process, which is performed by the successive cancellation (SC) algorithm, the time and computational complexity of decoding polar codes are higher than their competitors in which the received signals from the physical channel are directly used in the decoding process. Moreover, the error correction performance of long polar codes is not competitive against low-density parity-check (LDPC) codes.

This thesis focuses on improving list decoding of polar codes in terms of error correction performance and the computational/time complexity. Towards these goals, we follow various paths to achieve them: For improvement of error correction performance, 1) we propose an approach to characterize the sub-blocks of a code in terms of the possibility of elimination of the correct path. Then we modify the code to reduce this possibility within the sub-blocks during list decoding. 2) For the cases when the list decoding fails, we suggest shifting the pruning window at certain bit positions to recover the correct path or, in other words, to avoid the elimination of the correct path. On the other hand, for the complexity reduction, 1) we propose to use local list sizes for different segments of a code depending on the characteristics of the segments instead of employing a constant list size. 2) an efficient partial rewinding scheme is suggested to reduce the complexity of the re-decoding process based on the available intermediate information from the first run of the SC algorithm. The partial rewinding scheme can be used for the SC-flip algorithm, the shifted-pruning scheme, and the Fano algorithm.

Furthermore, we focus on different aspects of the recently introduced polarization-adjusted convolutional (PAC) codes: 1) we analyze the properties of PAC codes and the impact of convolutional precoding on the weight distribution of polar codes, as well as an approach to slightly improve the weight distribution of PAC codes by a modified precoding stage, 2) we adapt the list decoding and the list Viterbi decoding algorithms to PAC codes and analyze the error correction performance, the computational/time complexity as well as the path sorting complexity of these decoders, and 3) we propose an adaptive metric for Fano and stack decoding of PAC codes along with various tree search strategies to reduce the complexity of these decoders.

Declaration

I hereby declare that this thesis contains no material which has been accepted for the award of any other degree or diploma in any university or equivalent institution, and that, to the best of my knowledge and belief, this thesis contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

Mohammad Rowshan
July 15, 2021

Acknowledgements

This doctoral research project would not have been possible without the support of many people. First of all, I would like to express my sincere gratitude to my main supervisor, Prof. Emanuele Viterbo, for his support, guidance, insights, and the freedom he gave me. I came to Monash University without having any background in coding theory. I owe him whatever I have learned. Moreover, I learned from his outstanding character and patience. His skills in motivating people are incredible. My respect also goes to Dr. Shuiyin Liu, who taught me how to take the initial steps in conducting a research in my first year, and Dr. Yi Hong for her general guidance and support. I would like to thank Dr. Rino Micheloni and Ms. Alessia Marelli from Microsemi for the partial support of this project. The opportunity to visit École Polytechnique Fédérale de Lausanne (EPFL) became possible by the generous financial support of the Faculty of Engineering at Monash University, through the graduate research international travel award (GRITA), the help of Dr. Alexios Balatsoukas-Stimming, and Ms. Paniara Ioanna for doing the paper works of my travel and permit in Switzerland. Many thanks to Prof. Andreas P. Burg, my primary advisor at EPFL for hosting me in his lab and his insights on my works. Also, I would like to appreciate Dr. Son Hoang Dau, Dr. Anindya Gupta, Dr. Lakshmi Natarajan for their group lessons on the coding theory. The friendly staff of the department of Electrical and Computer Systems Engineering (ECSE) and the Faculty of Engineering at Monash University, in particular Dr. Lilian Khaw, who helped me in editing my first few papers, were quite helpful in this journey. I also made some good friends for which I am deeply grateful. Thank you, Fariba Abbasi, for the valuable discussions we had, Viduranga Wijekoon, Tharaj Taj, and Raviteja Patchava, for being such good friends in the lab. Finally, I would like to thank my parents and siblings for their continuous support and love. I am so happy to have you in my life. Although doing my Ph.D. took a lot of hard work and effort, I am glad to have gone through this experience.

Table of Contents

Publications During Enrolment

Journal Papers

J1 M. Rowshan, E. Viterbo, R. Micheloni, and A. Marelli, "Repetition-assisted Decoding of Polar Codes," *IET Electronics Letters,* vol. 55, no. 5, pp. 270-272, 2019.[1]

J2 M. Rowshan, A. Burg and E. Viterbo, "Polarization-adjusted (PAC) Codes: Fano Decoding vs List Decoding," *Transactions on Vehicular Technology*, vol. 70, no. 2, pp. 1434-1447, 2021.

J3 M. Rowshan and E. Viterbo, "List Viterbi Decoding of PAC Codes," in *Transactions on Vehicular Technology*, vol. 70, no. 3, pp. 2428-2435, March 2021.

J4 M. Rowshan and E. Viterbo, "Generalized Shifted-pruning for Path Recovery in List Decoding of Polar Codes," submitted to *IEEE Transactions*, 2021.

J5 M. Rowshan and E. Viterbo, "Efficient Partial Rewinding of Successive Cancellation-based Decoders," submitted to *IEEE Transactions*, 2021.

Conference Papers

C1 M. Rowshan and E. Viterbo, "Stepped List Decoding for Polar Codes," *IEEE Int. Symp. on Turbo Codes & Iter. Sig. Proc. (ISTC),* Hong Kong, Dec, 2018, pp1-5.

C2 M. Rowshan and E. Viterbo, "How to Modify Polar Codes for List Decoding," *2019 IEEE International Symposium on Information Theory (ISIT)*, Paris, France, 2019, pp. 1772-1776.

C3 M. Rowshan and E. Viterbo, "Improved List Decoding of Polar Codes by Shifted-pruning," *2019 IEEE Information Theory Workshop (ITW)*, Visby, Sweden, 2019, pp. 1-5.

C4 M. Rowshan, A. Burg and E. Viterbo, "Complexity-efficient Fano Decoding of Polarization-adjusted Convolutional (PAC) Codes," *2020 International Symposium on Information Theory and Its Applications (ISITA)*, Kapolei, HI, USA, 2020, pp. 200-204.

C5 M. Rowshan and E. Viterbo, R. Micheloni, and A. Marelli, "Logarithmic Non-uniform Quantization for List Decoding of Polar Codes," *2021 IEEE 11th Annual Computing and Communication Workshop and Conference (CCWC)*, NV, USA, 2021, pp. 1161-1166. [1]

C6 M. Rowshan and E. Viterbo, "Shifted-pruning for Path Recovery in List Decoding of Polar Codes," *2021 IEEE 11th Annual Computing and Communication Workshop and Conference (CCWC)*, NV, USA, 2021, pp.

C7 M. Rowshan and E. Viterbo, "On Convolutional Precoding in PAC Codes," *IEEE GLOBECOM*, Madrid, Spain, Dec 2021.

[1]The material of these papers (J1 and C5) is not included in this thesis.

List of Figures

List of Tables

List of Algorithms

Abbreviations

3GPP	3rd Generation Partnership Project
5G	5th Generation of Wireless Communications Standard
AWGN	Additive White Gaussian Noise
BEC	Binary Erasure Channel
BER	Bit Error Rate
BI-AWGN	Binary-Input Additive White Gaussian Noise
BI-DMS	Binary-Input Discrete Memoryless and Symmetric Channel
BLER	Block Error Rate
BMS	Binary Memoryless Symmetric
BP	Belief Propagation
BF	Bit-flipping
BPSK	Binary Phase Shift Keying
BSC	Binary Symmetric Channel
CA-SCLD	Cyclic Redundancy Check Aided Successive Cancellation List Decoding/Decoder
CC	Convolutional Codes
CRC	Cyclic Redundancy Check
CS	Critical/Crucial Set
Div	Diversion
DE	Density Evolution
DEGA	Density Evolution with Gaussian Approximation
DMC	Discrete Memoryless Channel
eMBB	Enhanced Mobile Broadband
FER	Frame Error Rate (a.k.a BLER, block error rate)
GA	Gaussian Approximation
I.I.D.	Independent and Identically Distributed
LD	List Decoding
LDPC	Low-Density Party-Check
LLR	Log-Likelihood Ratio
LP	Linear Program
LR	Likelihood Ratio
LVA	List Viterbi Algorithm
MAP	Maximum Aposteriori Probability
MLD	Maximum Likelihood Decoder
ML	Maximum Likelihood
PAC	Polarization-adjusted Convolutional
PM	Path Metric
RM	Reed-Muller
SC	Successive-Cancellation
SCD	Successive Cancellation Decoding/Decoder
SCL	Successive Cancellation List
SCLD	Successive Cancellation List Decoding/Decoder
SP	Shifted-pruning
SNR	Signal-to-Noise Ratio
VA	Viterbi Algorithm
WAVA	Wrap-around Viterbi Algorithm

Notations

$\mathcal{A}$	The set of indices of the non-frozen sub-channels
$\mathcal{A}^c$	The compliment of set $\mathcal{A}$
$\|\mathcal{A}\|$	The cardinality of set $\mathcal{A}$
α	The scaling factor for the adaptive metric in sequential decoding
$[a:b]$	A sequence of integer numbers from a to b (also $[a..b]$)
a_x^y	A sequence of $a_x, a_{x+1}, ..., a_{y-1}, a_y$
$a_x^y[l]$	A sequence of $a_x, a_{x+1}, ..., a_{y-1}, a_y$ for the l-th path
$\text{bin}(i)$	Binary representation of the decimal number i; $\text{bin}(i) = i_{n-1}...i_1 i_0$
d_{min}	The minimum Hamming distance of a code
Δ	The increment value for the metric threshold T in Fano decoding
δ_i	The metric for coordinate i used for prioritizing the shift
$\mathbf{c}$	The coefficient vector of the generator polynomial in convolutional codes (also $\mathbf{g}$)
$\mathbf{g}_j$	The j-th row of the polar transform's matrix G_N
$\mathbf{G}_N$	The matrix $N \times N$ used for polar transformation
κ	The number of shifts in terms of paths in the shifted-pruning scheme
L	List size in the list decoding and list Viterbi decoding.
λ_m^i	LLR at stage m and and row i of the polar transform's factor graph
m_i	The branch metric at coordinate i of a codeword in sequential decoding
$M_t(s)$	The metric of the path originating from state s at coordinate t in VA
$M_t(s,l)$	The metric of l-th path originating from state s at coordinate t in LVA
μ_i	The path metric at coordinate i of a codeword in sequential decoding
$\mu_t(s,s')$	The metric of the branch between state s and s' at coordinate t of a codeword
N	Code length, $N = 2^n$
$p_{e,i}$	Error probability of sub-channel i
$\mathbf{P}_n$	The matrix $2^n \times 2^n$ used for polar transformation
$PM_l^{(i)}$	The metric of l-th path at coordinate i of a codeword
Ψ_x	The total number of stages in the sorting network of decoder 'x'
R	Code rate; $R = K/N$
r	CRC length
$\mathcal{S}$	The set of all the states on the trellis
T	Maximum number of re-decoding attempts, a.k.a the metric threshold for backtracking
w_{min}	the minimum weight of a code
W	A binary memoryless symmetric (BMS) channel
W^N	N independent uses of channel W
W_N	A vector channel resulting from polar transformation
$W_N^{(i)}$	An i-th sub-channel of vector channel W_N

Chapter 1

Introduction

> "A journey of a thousand miles begins with a single step.
>
> — Lao Tzu

Error control coding (a.k.a channel coding) is a way of presenting data in a communication channel that adds redundancy to facilitate the detection and correction of errors. Such methods are widely used in wireless communications and storage systems (memories). Several high-performance channel codes were developed in recent decades, allowing information to be reliably transmitted at rates that closely approach the theoretical limit imposed by the channel capacity. More specifically, turbo codes were used in the 3rd Generation (3G) and 4th Generation (4G) of mobile communication standards, while the low-density parity-check (LDPC) codes were adopted to WiFi and satellite standards.

Polar codes [1] have explicit construction and low complexity encoding and decoding algorithms. They support variable code rates and code lengths (i.e., effortless methods are available for puncturing and shortening, respectively). In particular, polar codes offer a strong error correction performance for short block-lengths and low code rates compared to LDPC codes and turbo codes. The various advantages of polar codes convinced the 3rd Generation Partnership Project (3GPP) to adopt polar codes for physical layer control channel of enhanced mobile broadband (eMBB) for the 5th Generation (5G) of mobile communications standard [2], a.k.a new radio (NR).

However, polar codes are less mature than LDPC and turbo codes. During the past decade, the efforts were focused on improving the error correction performance of polar codes through developing practical decoding algorithms and improving the code construction of polar codes. The improvement obtained through code construction has not been significant. However, the various proposed algorithms for decoding have provided a competitive error correction performance for short block-lengths compared with counterparts.

Polar codes are founded on the polarization effect resulting from the channel synthesis in a particular fashion. The idea of building synthetic channels originated from the concatenated schemes [3] employed in the sequential decoding of convolutional codes by Massey [4], and Pinsker [5] in order to boost the cutoff rate. The cutoff rate is said to be "boosted" when

the sum of the cutoff rates of the synthesized channels is greater than the sum of the cutoff rates of the raw channels. The key idea in boosting the cutoff rate is to build a vector channel where the independent copies of the raw channels are transformed into multiple correlated channels. In Pinsker's scheme, the inner block code (with length N) is suggested to be chosen at random. This scheme requires maximum likelihood (ML) decoding with prohibitive complexity. Polar codes allow using a more practical decoder with the complexity of $O(N \log N)$. Unlike Pinsker's scheme, where the outer convolutional transforms are identical, in multi-level coding and multi-stage decoding (MLC/MSD), originally proposed in [6] as an efficient coded-modulation technique, N convolutional codes at different rates $\{R_i\}$ are used, which consequently require a chain of N outer convolutional decoders. On the other hand, polar coding was originally designed as a low-complexity recursive channel combining and splitting operation, where the polarization effect constrains the rates R_i to either 0 or 1. They turned out to be so effective that no outer code was employed to achieve the original aim of boosting the cutoff rate to channel capacity.

The result of synthesis is a polarized vector channel with N sub-channels. Throughout this thesis, we may use bit-channel and sub-channel interchangeably. We transmit information over a subset of the sub-channels called good sub-channels. Good sub-channels are selected explicitly based on the reliability of the channels. This process is called *code construction*. Since the rest of the sub-channels carry a known symbol as added redundancy, this process is sometimes called *rate profiling*. In [1], the Bhattacharyya parameter was used as a reliability metric for binary erasure channels (BEC). Density evolution (DE) was proposed in [7] for a more accurate reliability evaluation. However, it suffers from excessive complexity. To reduce the complexity of DE, a method based on the upper bound and lower bound on the error probability of the bit-channel was proposed in [8]. To further reduce the computational complexity of DE, Gaussian approximation (GA) to evolve the mean of log-likelihood ratios (LRRs) throughout the decoding was proposed in [9]. Recently, an SNR-independent low complexity method was proposed [10, 11], which gives the reliability ordering as a function of bit indices. Additionally, modified constructions of polar codes have also been considered. In [12], the evolution of LLRs of non-frozen bits (a.k.a free bits) during iterative Belief propagation (BP) decoding of polar codes was used to identify weak bit-channels and then to modify the conventional polar code construction by swapping these weak bit-channels with strong frozen bits-channels.

From the decoding point of view, the error correction performance of finite-length polar codes under successive cancellation (SC) decoding is not competitive due to partially polarized

sub-channels. To address this issue, the SC list (SCL) decoding (SCLD) was proposed in [13]. This yields an error correction performance comparable to maximum-likelihood (ML) decoding. Also, it was observed that further improvement could be obtained by concatenating cyclic redundancy check (CRC) bits to polar codes. The weak side of list decoding is that it suffers from high computational complexity and large memory requirements.

Although the CRC-aided SCL decoder provides a competitive performance, its main drawbacks are high computational complexity and large memory requirement. To reduce the size of storage and processing elements in the hardware implementation, the internal soft messages were changed from log-likelihood (LL) to LLR in [14]. However, the total memory area still accounts for 40%-45% of the total silicon area. In [15], it was shown that a logarithmic non-uniform quantization of channel and intermediate LLRs can reduce the number of bits required for intermediate LLRs while we can improve the block error rate due to higher precision in particular for long codes. In another attempt, the tree/list pruning method was proposed to reduce the complexity. In this method, the path list is pruned using a threshold obtained either online or offline [16, 17]. Although this method introduces a computational overhead in the list pruning procedure, it can substantially reduce overall computational complexity. In [18], the computational complexity was reduced by dropping the frequently split paths from the list. Additionally, a counter was used to recognize the correct path, and then it was switched to an SC decoder for decoding the rest of the bits. Nevertheless, this method cannot provide the performance of a CRC-aided decoder. Also, similar to the tree pruning method, it requires the conventional (memory-intensive) SCL decoding. Furthermore, a bound for list size and information-theoretic aspects of list decoding were studied in [19, 20].

A simplified SC (SSC) polar decoder was introduced in [21] to reduce the complexity and latency. This decoder was further developed and used for list decoder [22]. In the SSC polar decoder, several types of nodes associated with pattern-specific constituent codes is recognized in the binary tree of decoding. Relying on those nodes, the recursion in the decoding process is greatly simplified. Segmenting or partitioning based on multi-CRC schemes proposed in [23, 24] is another method in which the memory requirement reduces. In this method, every partition estimates a sub-block of the code by performing CRC-aided SCL decoding, and then the estimated bits are passed to the next partition. However, this method saves the memory significantly and contributes slightly to the complexity reduction, it requires employing several short CRCs at the cost of an increase in the effective code rate, consequently affecting the error correction performance. The effective code rate is defined from the polar coding point of view

as the number of non-frozen bits (including information bits and CRC bits) divided by the total number of bits. For instance, $4 \times 8 = 32$ bits are used for CRCs in a 4-partition scheme, compared with 16 bits in conventional CRC-aided SCL decoding. Needless to mention that the probability of undetected error (false detection of the correct path) by employing short CRCs increases.

When SC or SCL decoding fails, we may be able to correct the error(s) in additional decoding attempts. Bit-flipping [25] is a popular method to improve the error correction performance of the SC decoder by a single or multiple flipping of the low-reliability bit(s) in each re-decoding attempt. The numerical experiments in [25] showed a predominant portion of the decoding failures occurs by a single error in bit estimation due to channel noise. Thus, by finding the first erroneous bit and flipping the estimated value, error propagation can be prevented. This idea was further improved by using an improved metric for single and multiple flipping in [26] which was later computational simplified [27]. An offline-obtained error distribution in [28] and a static set of critical coordinates [18] were employed to find the potential coordinates of single or multiple errors. The performance of these methods can approach the performance of SCL decoding with a moderate list size $L = 2$ or, in the case of employing multiple flipping, it can reach the performance of $L = 4$ or 8. However, in terms of complexity, nested/multiple bit-flipping may require many attempts (tens or hundreds of attempts), making multiple flipping impractical. Note that the receiving messages from the physical channel cannot wait for such a considerable number of iterations, and we cannot afford to have a vast input buffer and output buffer to regulate the input and out streams bits.

The attempt to re-decode the failed SCL decoding was first started in [29] by gradually increasing the list size (L) by a factor of two until reaching the predefined maximum list size or succeeding in decoding. However, increasing the list size contributes to a larger computational and time complexity and requires enough hardware resources to support that. Also, as the results showed, the performance gain beyond list size $L > 32$ is less than 0.1 dB obtained at a very high cost of doubling the resources. The work in [30] showed that we can improve the performance by repeating the bits transmitted over low-reliability bit-channels in the code and exploiting them in re-decoding. In [31], a bit-flipping scheme based on a modified critical set was employed in CRC-aided SCL decoding. Later, we proposed the shifted-pruning scheme in [32][1] where the pruning window was shifted by $\kappa = L$ (i.e., selecting the worst L paths in

[1]The work in [32] was submitted to ITW2019 on Apr. 10, 2019, presented on Aug. 26 in Visby, Sweden, and published on IEEE Xplore on Feb. 10, 2020

the sorted list instead of the best ones) at the critical bit coordinates. Later, This scheme was improved in [33] by employing a computationally simple LLR-based metric and a variable κ (e.g., $\kappa = L/2$) where an approximate coordinate of elimination of the correct path was able to correct the error. Moreover, the original shifted-pruning scheme in [32] was improved in [34] by an ordered critical set in an adaptive list size scheme. Independently of us, similar work was proposed in [35]. In this work, inspired by [26], a probabilistic metric was employed that had a poor performance on medium and long codes. A different metric based on a probability ratio was suggested in [36]. The approximation of this metric in the log domain is similar to the path metric range (PMR) suggested in [33, 37, 38]. Nevertheless, this metric also suffers from poor performance for relatively long codes.

Besides SC and SCL decoders, other well-known decoders were also adapted to polar codes. However, they could not provide good performance at a low computational cost. A belief propagation (BP) decoder was used in [39] to compare the performance of polar codes with that of Reed-Muller (RM) codes over BEC. The results showed a distinct performance advantage for polar codes over RM codes as the code length increased at the cost of high computational complexity. Since then, many researchers have focused on reducing the complexity of parallel processing in BP decoding. In [40], a belief propagation list (BPL) decoder was proposed with comparable performance to the SCL decoder. The proposed decoder is composed of multiple parallel independent BP decoders based on differently permuted polar code factor graphs. A list of possible transmitted codewords is generated, and the one closest to the received vector, in terms of Euclidean distance, is picked. The *Viterbi* and *BCJR* decoding algorithms were considered in [3] to improve the bit error rate (BER) and block error rate (BLER) of polar codes. However, these decoders are only applicable to short polar codes because for decoding long codes, too many states are required. The ML decoding (optimization) problem was relaxed to linear programming (LP) decoding problem in [41]. The original LP decoder had high time complexity and performed very poorly over the AWGN channel. Then, an adaptive version was provided in [42] which significantly improved the FER performance. A sphere decoding (SD) was adapted in [43] which can achieve ML performance by traversing all the possible codewords. Later on, researchers focused on reducing the complexity of this method. In [44], a sphere list decoding was proposed to make the FER performance comparable with small list-size SCL decoding for short codes. The SC stack decoding was adapted in [45] to store a number of candidate partial paths in a stack and try to find the best path in it. SC stack decoding with enough stack depth has a dynamic time complexity while maintaining the same performance

as the SC list. However, only the top path in the stack is concerned in SC stack, and it takes high time complexity to traverse the whole stack when inserting a new path. Eventually, the ordered statistic decoding (OSD) was considered in [46] to decode short polar codes. Since the complexity of OSD is high for codes with lengths larger than 64 bits, [46] proposed a threshold-based OSD decoder to decrease the number of tested codewords and consequently to reduce the complexity.

Recently in [47], Arıkan proposed a concatenation of a convolutional transform with the polarization transform [1] in which a message is first encoded using a convolutional transform and then transmitted over polarized synthetic channels as shown in Fig. 6.1. These codes are called "polarization-adjusted convolutional (PAC) codes". It was shown that PAC codes could outperform short polar codes without CRC concatenation [48, 49]s, and medium-length CRC-PAC codes can outperform CRC-polar [48]. In general, a properly designed upper-triangular pre-transformation [50] such as convolutional transform could improve the distance properties of polar codes. Later, the reason behind the performance improvement of PAC codes was revealed in [51]. It turned out that the inclusion of frozen rows in the polar transform could increase the Hamming weight of the resulting codewords. In [48, 52], we also studied the implementation of tree search algorithms, including the conventional list decoding, stack decoding, and complexity-efficient Fano decoding for PAC codes. Furthermore, PAC codes were compared with convolutional codes in terms of performance and complexity in [53] and systematic PAC codes were discussed in [54, 55] and shown that they can improve the bit error rate (BER) of PAC codes, just like polar codes. Due to the convolutional pre-transformation, PAC codes can also be easily encoded and decoded based on the trellis by employing the Viterbi algorithm (VA) [56, 57] and the list-type VA [58]. The basic Viterbi algorithm was employed in [59] as a maximum likelihood (ML) decoder for short polar codes in comparison with Reed-Muller (RM) codes. Additionally, a performance comparison of convolutional codes under M-algorithm and LVA was provided in [60]. We adapted LVA to PAC codes in [61] and analyzed the significant reduction in the sorting complexity of LVA.

It has been shown that employing large kernels for constructing polar codes improves the error exponents and consequently provides a better error correction performance. However, the large kernel polar codes suffer from high complexity decoding processes [62]. Furthermore, a new concatenation scheme was introduced recently that concatenates outer polar codes with inner repetition codes known as polar coded repetition [63, 64].

1.1 Key Contributions

The major contributions of this thesis are briefly listed below.

Stepped List Decoding

In the successive cancellation list (SCL) decoding of polar codes, as the list size increases, the error correction performance improves. However, a large list size results in high computational complexity and large memory requirement. We investigated the list decoding process by introducing a new instrumental notion named path metric range (*PMR*) to elucidate the properties of the evolution of the path metrics (PMs) within the list throughout the decoding process. Then, we advocate that the list size can change step-wise depending on *PMR*. Alternatively, the local list size for the sub-blocks of the code can be determined based on the bit error probability of the sub-blocks. In the proposed *stepped list decoding* scheme, the error correction performance of the conventional list decoding is preserved while the computational complexity reduces significantly. The reduction in complexity is SNR-independent and achieved without introducing any computational overhead.

Modified Polar codes for List Decoding

Polar codes are constructed based on the reliability of the individual bit-channels by exact calculation of this metric or by approximation methods. This construction is consistent with the successive cancellation (SC) decoding, where one error in the successive estimation of the bit-values results in the decoding failure. However, in SCL decoding, the correct candidate may remain on the list by tolerating multiple penalties due to distance from the received signals. This characteristic of list decoding demands a different approach in code design. In this work, we propose a general approach to modify the polar codes constructed by conventional methods. In this approach, a bit-swapping technique is employed to re-distribute the low-reliability bits in the sub-blocks. This redistribution reduces the overall probability of the elimination of the *correct path*. The numerical results for polar codes of various lengths, rates, and list sizes constructed with the Bhattacharyya parameter and density evolution with Gaussian approximation show improvements which vary from 0.1 dB to 0.4 dB.

Shifted-pruning Scheme For Path Recovery in List Decoding

In SCL decoding, the tree pruning operation retains the L best paths with respect to a metric at every decoding step. However, the correct path might be among the L worst paths due to imposed penalties. In this case, the correct path is pruned, and the decoding process fails. Shifted-pruning (SP) scheme can recover the correct path by additional decoding attempts when decoding fails, in which the *pruning window* is shifted by $\kappa \leq L$ paths over certain bit positions. A special case of shifted-pruning scheme where $\kappa = L$ is known as SCL-flip decoding, which was independently proposed in 2019. For this scheme, a metric that performs well on any code length and rate was proposed, and a nested shift-pruning scheme was suggested for improving the FER performance.

Efficient Partial Rewinding of SC Algorithm

It is known that for calculation of every decision log-likelihood ratio (LLR) in SC algorithm, we need at most $N - 1$ values for intermediate LLRs (excluding channel LLRs) and partial sums, instead of $N \cdot \log_2 N$ values (N is the code length). In the SC-flip decoding and shifted-pruning scheme, we need to re-decode the codeword when the decoding fails in the first run. The common practice is re-decoding from the first bit. Starting the decoding from scratch contributes to the computational complexity significantly. An efficient re-decoding scheme based on the partial rewinding of successive cancellation algorithm is proposed to reduce the additional decoding attempts' complexity significantly. Then, this scheme is evaluated on the shifted-pruning scheme.

Convolutional Pre-coding and List Decoding of PAC Codes

We explicitly show why the convolutional precoding reduces the number of minimum-weight codewords. Furthermore, we show where the precoding stage is not effective based on the rate profile. Then, we adapt the list decoding algorithm to PAC codes. Additionally, we recognize the potential weakness of the convolutional precoding, unequal error protection (UEP) of the information bits. Finally, we assess the possibility of mitigating this weakness by irregular convolutional precoding.

Complexity-efficient Sequential Decoding of PAC Codes

The sequential decoding (including Fano decoding and stack decoding) is first adapted to decode PAC codes. Then, to reduce the complexity of sequential decoding of PAC/polar codes, we propose (i) an adaptive heuristic metric, (ii) tree search constraints for backtracking to avoid exploration of unlikely sub-paths, and (iii) tree search strategies consistent with the pattern of error occurrence in polar codes. These contribute to the reduction of the average decoding time complexity from 50% to 80%, trading with 0.05 to 0.3 dB degradation in error correction performance within FER=10^{-3} range, respectively, relative to not applying the corresponding search strategies. Additionally, as an essential ingredient in memory-efficient Fano decoding of PAC/polar codes, an efficient computation method for the intermediate LLRs and partial sums is provided. This algorithmic method, which is different from the aforementioned partial rewinding scheme, effectively backtracks and avoids storing all the intermediate information or restarting the decoding process. Eventually, the sequential and list decoding algorithms are compared in terms of performance, complexity, and resource requirements.

List Viterbi Decoding of PAC Codes

Motivated by the fact that the list Viterbi algorithm (LVA) sorts the candidate paths locally at each trellis node, we adapt the trellis, path metric, and the local sorter of LVA to PAC codes. Then, we show how the error correction performance moves from the poor performance of the Viterbi algorithm (VA) to the superior performance of list decoding by changing the constraint length, list size, and the sorting strategy (local sorting and global sorting) in the LVA. We also analyze the complexity of the local sorting of the paths in LVA relative to the global sorting in the list decoding, and we observe that LVA has a significantly lower sorting complexity than list decoding.

1.2 Thesis Organisation

The remainder of this thesis is organized as follows. Chapter 2 reviews the concept of channel polarization, the construction of polar codes, concatenated polar codes, the encoding and decoding algorithms. Chapter 3 covers the notion of adjusted list decoding by two approaches: 1) Allocating local list sizes to the segments of a code instead of fixed list size, and 2) balancing

the probability of error in different segments of a code while employing a constant list size throughout the decoding. In Chapter 4, the concept of shifted pruning is introduced. Then, a metric for prioritizing the bit position for shifting the pruning window is suggested. Additionally, a double shifting scheme is proposed. Chapter 5 studies the basic properties of updating the order of intermediate information in successive cancellation algorithms. Then, an efficient scheme for partial rewinding of the SC algorithm is suggested. Chapters 6-8 are focused on PAC codes. In Chapter 6, the properties of PAC codes are studied, and the list decoding algorithm is adapted to PAC codes. In Chapter 7, an adaptive metric for the Fano decoding algorithm is proposed. Additionally, several tree search strategies to reduce the complexity of Fano decoding are suggested. Chapter 8 adapts the list Viterbi algorithm to PAC codes and analyzes the sorting complexity of this algorithm in comparison with the tree-based list decoding along with the error correction performance.

Chapter 2

Polar Codes: A Review

"If I have seen further than others, it is by standing upon the shoulders of giants."
— Isaac Newton

In this chapter, we review polar codes, PAC codes, and their decoding algorithms. We mainly revise the schemes and techniques used to efficiently decode polar codes in terms of computational or time complexity, memory requirements, and other factors. Hence, this chapter introduces the notations and concepts we will use throughout this thesis.

2.1 Notations and Preliminaries

Let us consider the system given in Fig. 2.1, in which polar code as an error correction code (ECC) is used in communications and storage systems. We introduce all the parameters and the components of this system in this chapter. The block length of the polar codes is denoted by $N = 2^n$, where n is an integer ($n > 0$). The signals shown by boldface lowercase letters are vectors. The uncoded bit vector $\mathbf{u} \in \mathbb{F}_2^N$ or $u_1^N \in \{0,1\}^N$ is input to the polar encoder. The channel in the system is an arbitrary memoryless channel $\mathcal{W} : \mathcal{X} \longrightarrow \mathcal{Y}$ with input alphabet $\mathcal{X} = \{0,1\}$, output alphabet $\mathcal{Y}$, and transition probabilities $\{W(y|x) : x \in \mathcal{X}, y \in \mathcal{Y}\}$. The

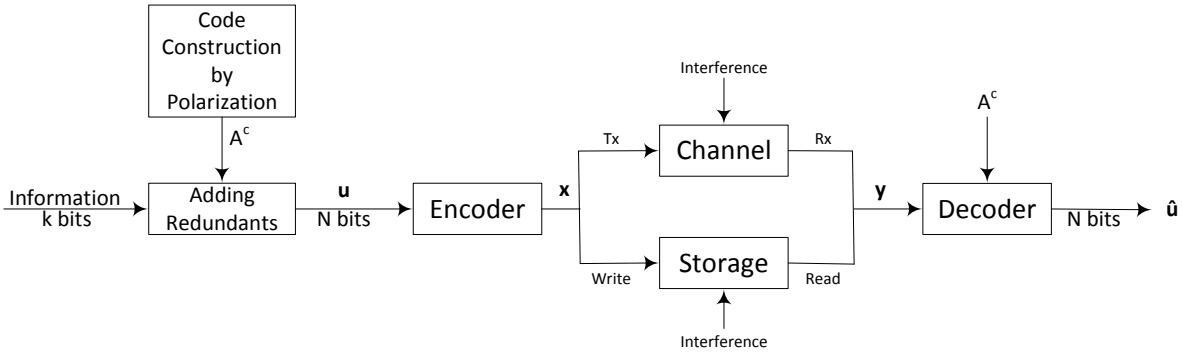

Fig. 2.1: ECC in Communication and Storage systems with polar codes

channel mutual information with equiprobable inputs, or symmetric capacity, is defined by

$$I(W) = \sum_{y\in\mathcal{Y}} \sum_{x\in\mathcal{X}} \frac{1}{2} W(y|x) \log \frac{W(y|x)}{\frac{1}{2}W(y|0) + \frac{1}{2}W(y|1)}, \tag{2.1}$$

and the corresponding Bhattacharyya parameter by

$$Z(W) = \sum_{y\in\mathcal{Y}} \sqrt{W(y|0)W(y|1)}. \tag{2.2}$$

We denote the channel input and output sequences by x_1^N and y_1^N, respectively, with corresponding vector channel $W_N(y_1^N|x_1^N)$.

In each use of the system, a codeword is transmitted, and a channel output vector $y \in \mathcal{Y}^N$ is received. The receiver first calculates the log-likelihood ratio (LLR) vector $\lambda = (\lambda^1, ..., \lambda^N)$ with

$$\lambda^i = \ln \frac{W(y_i|x_i = 0)}{W(y_i|x_i = 1)}, \tag{2.3}$$

for each element of the channel output vector and feeds it into a decoder for polar codes.

Throughout this thesis, all symbols and vector operations are over the binary field $\mathcal{F}_2$. The logarithms are performed in base-2 unless stated otherwise.

2.2 Channel Polarization

The key idea of polar codes lies in using a polarization transformation that converts N identical and independent copies of a physical channel W, $\{W_N^{(i)} : 1 \leq i \leq N\}$, into N virtual/synthetic channels which are either better or worse than the original channel W.

Consider the matrix $\mathbf{G}_2 \triangleq \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$, and let $\mathbf{G}_N = \mathbf{G}_2^{\otimes n}$ be the n-th Kronecker power of $\mathbf{G}_2$, where $n = log_2 N$. We define $W_N(y_1^N|u_1^N) = W_N(y_1^N|u_1^N\mathbf{G}_N)$ as the polarized vector channel from the input bits. From $W_N(y_1^N|u_1^N)$, the bit-channel (a.k.a sub-channel) $i \in [N]$ is implicitly defined as

$$W_N^{(i)}\left(y_1^N, u_1^{i-1}|u_i\right) = \sum_{u_{i+1}^N} \frac{1}{2^{N-i}} W_N\left(y_1^N|u_1^N\right). \tag{2.4}$$

The channel polarization theorem [1] states that $I(W_N^{(i)})$ converges to either 0 or 1 as N approaches infinity. It can also be shown that the fraction of the channels that become perfect converges to the mutual information of the original channel W, i.e., $I(W)$, meaning that polar codes are *capacity achieving* while the fraction of extremely bad channels approaches to $(I(W) - 1)$.

Polar codes with rate $R = K/N$ are constructed by selecting K indices with the highest $I(W_N^{(i)})$ or the lowest $Z(W_N^{(i)})$ for $i \in [N]$. These are dedicated to information bits and called the *non-frozen set*, $\mathcal{A}$. The input bits corresponding to *frozen set* $\mathcal{A}^c$ are usually set to zero. We further discuss the construction of polar codes in the next section.

2.3 Code Construction (Channel Selection)

The aim in constructing polar codes is to determine $\mathcal{A}$ set. Arıkan proposed a recursive calculation algorithm based on the Bhattacharyya parameters for channel-reliability evaluation in his seminal paper [1]. If the original binary discrete memoryless channel (B-DMC) is a binary erasure channel (BEC), the erasure probabilities of the polarized channels can be tracked with low complexity of $O(N \log N)$ by using this recursive algorithm. Equivalently, the corresponding capacities can also be recursively calculated for the case of BEC. The complexity can be further reduced to $O(N)$ if the intermediate Bhattacharyya parameters are used instead [7]. However, for channels other than BEC, the computational complexity grows exponentially with the code length and input alphabet size.

To construct a polar code over an arbitrary symmetric B-DMC, Mori, and Tanaka in [7] proposed the use of density evolution (DE), which are widely used in designing the LDPC codes, for tracing the probability density function (PDF) of LLRs at variable (f) and check (g) nodes in the factor graph shown in Fig. 2.2. The convolutions of the PDFs are performed at the variable and check nodes. Based on the PDF of LLRs at the first stage of the variable nodes, the error probabilities of all the polarized channels can be obtained. Similar to the Bhattacharyya parameters, the order of magnitude of convolutions is $O(N)$. In practice, to keep the complexity to an acceptable level, the PDF of LLRs should be quantified into q levels. Thus, the computational complexity of the quantized density evolution is $O(q^2 N)$. However, a typical value of q is 10^5, implying a substantial computational burden in practical application. The difficulty is further aggravated by the quantization errors, which are accumulated over

multiple polarization stages. Indeed, as noted in [7] it is challenging to find an optimal tradeoff between the implementation complexity and calculation precision.

Tal and Vardy [8] proposed an effective method to solve this problem by controlling the quantization errors and through appropriate approximation. They wisely introduced two approximation methods called upgrading and degrading quantization to get a lower and upper bound on the error probability of each channel. Both methods transform the relevant channel into a new one with a smaller output alphabet in terms of μ. Then the construction complexity with their algorithm can be evaluated as $O(N\mu^2 \log \mu)$. A typical value of μ is 256, so it is much less complex than DE.

Although the increase in the output alphabet size can improve the precision of density functions in this algorithm, it increases the algorithm complexity. For binary input additive white Gaussian noise (AWGN) channels, the most commonly used channel model by coding theorists, an alternative method [9] called Gaussian approximation (GA) can be applied in the construction of polar codes. The GA has lower complexity than Tal and Vardy's method when applied to binary input AWGN channels but yield almost the same precision. From a practical point of view, GA is a more attractive choice than other methods.

In the previously introduced construction methods, polar codes are constructed as a function of the channel parameters, such as designed signal-to-noise ratio (SNR). Also, their computational complexity scales linearly with the code block-length, and therefore unacceptable for practical systems with varying parameters such as block-length and code rate. They are even infeasible to be used for an on-the-fly implementation of a low-latency encoder/decoder. Recently, the polarization weight (PW) method was introduced in [65] which is a channel-independent approximation method for estimating the sub-channel reliability as a function of its index. This method relies on the $\beta-expansion$ notion borrowed from number theory. The polar codes can be recursively constructed by continuously solving several polynomial equations at each recursive step. From these polynomial equations, we can extract an interval for β, such that ranking the synthetic channels through a closed-form $\beta-expansion$ preserves the property of nested frozen sets is a desired feature for low-complex construction.

2.4 Encoding

In the original polar coding, the non-systematic form was used. In non-systematic form, the input to the polar encoder is the full word of dimension N consisting of the information bits placed at the positions in the reliable set $\mathcal{A}$ together with the foreknown bit placed at the positions in the frozen set $\mathcal{A}^c$, which leads to the following description of the encoding process

$$\mathbf{x} = \mathbf{u}\mathbf{G}_\mathbf{N} = \mathbf{B}_\mathbf{N}\mathbf{G}_\mathbf{2}^{\otimes \mathbf{n}} \tag{2.5}$$

where $\mathbf{u}$ and $\mathbf{x}$ are the input and output vectors, respectively, $\mathbf{G}_N$ is the generator matrix acting as the combining transform, kernel matrix $G_2 \triangleq \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$, B_N is an $N \times N$ bit-reversal permutation matrix, and $(\cdot)^{\otimes n}$ denotes the n-th Kronecker power. The systematic coding [47] can improve the bit error rate (BER) performance of polar codes whose frame error rate (FER) performance remains unchanged.

2.5 Decoding

Polar codes were introduced along with a successive cancellation (SC) decoding algorithm in [1]. Due to the poor performance of SC in the finite block lengths, several decoders based on SC decoding have been introduced. We reviewed most of them in Chapter 1, in the following sections, we focus on the most popular one.

2.5.1 SC Decoding

The channel combining in the encoding process introduces a correlation between the source bits. As a result, each coded bit with a given index relies on all of its preceding source bits with smaller indices. This kind of correlation can be conceptually treated as *interference* in the source-bit domain, which leads to a significantly better decoding performance when exploited. Therefore, the bits are decoded one at a time in a specified order. The bit decision $\hat{u}_i$ is made before the calculations to find the next bit $\hat{u}_{i+1}$ starts, and already decided bits influence the decision of the following bit decisions. Successive cancellation of the "interference" caused by the previous bits improves the reliability of retrieving the source bits.

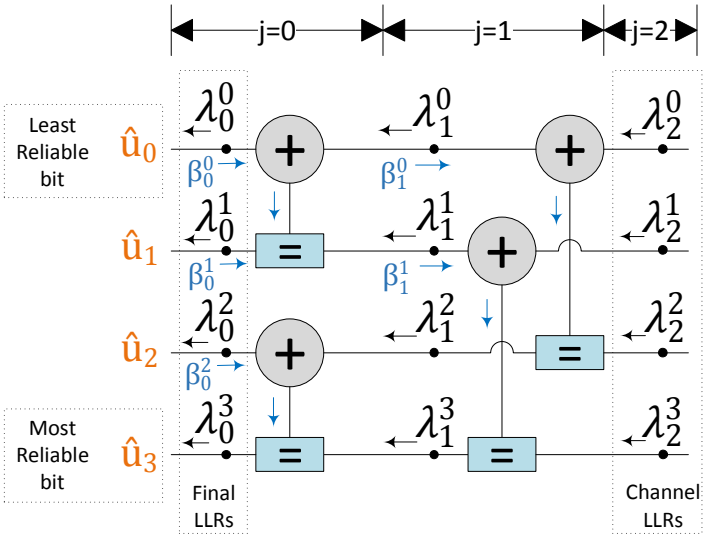

Fig. 2.2: Internal/intermediate information in SC Decoding of 4 bits

In SC decoding, the information bits are estimated by hard decision based on the final evolved LLRs λ_i^0 as shown in Fig. 2.2. When decoding the i-th bit, if $i \notin \mathcal{A}$, regardless of final LLR value λ_i^0, $\hat{u}_i$ is set as a frozen bit, i.e., $\hat{u}_i = 0$. Otherwise, u_i is decided by a maximum likelihood (ML) rule as equation (2.6) based on the previously estimated vector $(\hat{u}_1, ..., \hat{u}_{i-1})$.

$$\hat{u}_i = h(\lambda_0^i) = \begin{cases} 0 & \lambda_0^i = \ln \frac{P(Y, \hat{u}_0^{i-1} | \hat{u}_i = 0)}{P(Y, \hat{u}_0^{i-1} | \hat{u}_i = 1)} > 0, \\ 1 & \text{otherwise} \end{cases} \tag{2.6}$$

The decoder estimates the transmitted bits successively by computing LLRs of the indexed edges. The LLR of edge (i, j) is computed by

$$\lambda_j^i = \begin{cases} f(\lambda_{j+1}^i, \lambda_{j+1}^{i+2^j}) & \text{if } B(i,j) = 0 \\ g(\lambda_{j+1}^{i-2^j}, \lambda_{j+1}^i, \hat{\beta}_j^{i-2^j}) & \text{if } B(i,j) = 1 \end{cases} \tag{2.7}$$

where $0 \leq i < N$, $0 \leq j \leq n$, $B(i,j) = \lfloor \frac{i}{2^j} \rfloor \mod 2$ and $\hat{\beta}_j^i$ denotes the partial sum, which corresponds to the propagation of estimated bits $\hat{u}_i$ backward into the factor graph shown in Fig. 2.2. Note that i and j denote the bit index and stage index, respectively.

The f and g functions in (2.7), which are illustrated in Fig. 2.3, can be well approximated by:

$$f(\lambda_a, \lambda_b) \approx \text{sgn}(\lambda_a) \cdot \text{sgn}(\lambda_b) \cdot \min(|\lambda_a|, |\lambda_b|) \tag{2.8}$$

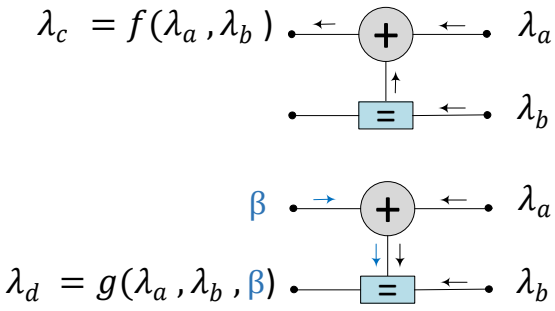

Fig. 2.3: Internal LLR (λ) calculations

$$g(\lambda_a, \lambda_b, \hat{\beta}) = (-1)^{\hat{\beta}}\lambda_a + \lambda_b \tag{2.9}$$

where λ_a and λ_b are the incoming LLRs to a node and $\hat{\beta}$ is the partial sum of previously decided bits.

2.5.2 SC List (SCL) Decoding

The polar codes under successive cancellation (SC) decoding suffer from poor error correction performance for short and medium block lengths. To address this issue, the SCL decoding and CRC-aided SCL decoding were adapted to polar codes in [13]. Unlike SC decoding, which selects/follows a particular path at each decision step, the SCL decoding considers both possible values $u_i = 0$ and $u_i = 1$ and finds a path through the decoding tree which has the highest probability to be the transmitted sequence u_0^{N-1}. For optimal decoding, the probability to be *maximized* is

$$P(\hat{u}_0^{N-1}|y_0^{N-1}) = \prod_{t=0}^{N-1} P(\hat{u}_t|\hat{u}_0^{i-1}, y_0^{N-1}) \tag{2.10}$$

However, we cannot traverse the entire decoding tree, i.e., considering all the paths by SC and SCL decoding. Therefore, the solution obtained from the decoding can be sub-optimal. In SCL decoding, the probability of partial path l representing the sequence $\hat{u}_0^{i-1} = (\hat{u}_0, \hat{u}_1, \ldots, \hat{u}_{i-1})$ is computed by

$$P(\hat{u}_0^{i}[l]|y_0^{N-1}) = \prod_{t=0}^{i} P(\hat{u}_i[l]|\hat{u}_0^{i-1}[l], y_0^{N-1}) \tag{2.11}$$

In practice, it is more convenient and practical to deal with the logarithm of (2.10). Hence,

we take the logarithm of (2.11). Since $\log(x) < 0$ for $x < 1$, we multiply the resulting logarithm by -1 to have a positive metric. Therefore, we get the following logarithmic *path metric*

$$\begin{aligned} PM_l^{(i-1)} &= -\log P(\hat{u}_0^{i-1}[l]|y_0^{N-1}) \\ &= -\sum_{j=0}^{i-1} \log P(\hat{u_j}[l]|\hat{u}_0^{j-1}[l], y_0^{N-1}) \end{aligned} \tag{2.12}$$

for the sequence $\hat{u}_0^{i-1}$ at position l.

Now let the sequence $\hat{u}_0^i$ be obtained by appending $\hat{u}_i$ to $\hat{u}_0^{i-1}$ on path l and suppose $\hat{u}_i$. The path metric for this longer sequence is

$$PM_l^{(i)} = -\sum_{j=0}^{i} \log P(\hat{u}_j[l]|\hat{u}_0^{j-1}[l], y_0^{N-1}) \tag{2.13}$$

$$= PM_l^{(i-1)} + \mu_l^{(i)} \tag{2.14}$$

where $\mu_l^{(i)} = -\log P(\hat{u}_i[l]|\hat{u}_0^{i-1}[l], y_0^{N-1})$ denotes the *branch metric*, and $PM_i^{(-1)} = 0$. Hence, the path metric along a path ending at bit i is obtained by adding the path metric ending at bit $i-1$ to the branch metric at bit i. Throughout this paper, we assume that the index l indicates the position of a sorted path list, i.e., $PM_1^{(i)} \leq PM_2^{(i)} \leq ... \leq PM_L^{(i)}$ and l_c denotes the index of the correct path.

To simplify the arithmetic operation, we can define μ_i as

$$\begin{aligned} \mu_l^i &= -\log P(\hat{u}_i[l]|\hat{u}_0^{i-1}[l], y_0^{N-1}) \\ &= -\log\left(\frac{e^{(1-\hat{u}_i[l])\lambda_0^i[l]}}{e^{\lambda_0^i[l]}+1}\right) = \log\left(1+e^{-(1-2\hat{u}_i[l])\lambda_0^i[l]}\right) \end{aligned} \tag{2.15}$$

where the last equality holds only for $\hat{u}_i[l] = 0$ and 1. For the value of $\hat{u}_i[l]$ that equals $h(\lambda_0^i[l])$, the term $e^{-(1-2\hat{u}_i)\lambda_0^i} = e^{-|\lambda_0^i|}$ is small and hence $\log(1 + e^{-|\lambda_0^i|}) \approx 0$. Otherwise, we can approximate $\log(1 + e^{|\lambda_0^i|}) \approx |\lambda_0^i|$. Thus

$$\mu_l^i = \mu_l^i(\lambda_0^i[l], \hat{u}_i[l]) \approx \begin{cases} 0 & \text{if } \hat{u}_t[l] = h(\lambda_0^i[l]) \\ |\lambda_0^i[l]| & \text{otherwise} \end{cases} \tag{2.16}$$

As (2.16) shows, the path of the less likely bit value is penalized by $|\lambda_0^i[l]|$. This value is

called *penalty* throughout this thesis. At each decoding step, the L paths with smallest metrics are chosen among $2L$ paths and stored in ascending order from $PM_1^{(i)}$ to $PM_L^{(i)}$. After decoding the last bit, the path with the smallest path metric, i.e., $PM_1^{(N-1)}$, or the path that passes the CRC is selected as the estimated sequence.

Additionally, when the SCL decoding fails, the correct path might still be in the list but not in the position of the most likely path. Adding an r-bit CRC as an outer code to the information bits can assist the decoder in error detection and finding the correct path among the L paths. However, this concatenation increases the polar code rate to $(K + r)/N$. In this paper, $P(N, K + r)$ denotes a polar code of length N with K information bits concatenated with r-bit CRC.

2.6 Polarization-adjusted Convolutional Codes

Polarization-adjusted convolutional (PAC) codes are denoted by $PAC(N, K, \mathcal{A}, \mathbf{c})$, where $N = 2^n$ is the length of the PAC code. A rate profiler first maps the K information bits to N bits. Then, the convolutional transform (with polynomial coefficients vector $\mathbf{c}$) scrambles the resulting N bits before feeding them to the classical polar transform as shown in Fig. 2.4. The information bits $\mathbf{d} = [d_0, d_1, ..., d_{K-1}]$ are interspersed with $N - K$ zeros and mapped to the vector $\mathbf{v} = [v_0, v_1, ..., v_{N-1}]$ using a rate-profile which defines the code construction. The rate-profile is defined by the index set $\mathcal{A} \subseteq \{0, \ldots, N - 1\}$, where the information bits appear in $\mathbf{v}$. This set can be defined as the indices of sub-channels in the polarized vector channel with high reliability. These sub-channels are called *good channels*. The bit values in the remaining positions, $\mathcal{A}^c$, in $\mathbf{v}$ are set to 0.

The input vector $\mathbf{v}$ is transformed to vector $\mathbf{u} = [u_0, \ldots, u_{N-1}]$ as $u_i = \sum_{j=0}^{m} c_j v_{i-j}$ using the binary generator polynomial of degree m, with coefficients $\mathbf{c} = [c_0, \ldots, c_m]$. This convolutional transformation combines m previous input bits stored in a shift register with the current input bit v_i as shown in Fig. 2.5 to calculate u_i. The parameter m is known as the *memory* of the shift register and by including the current input bit we have the *constraint length* $m + 1$ of the convolutional code.

Equivalently, the convolution operation can be represented in the form of Toeplitz matrix where the rows of a *generator matrix* G are formed by shifting the vector $\mathbf{c} = (c_0, c_1, \ldots c_m)$ one element at a row. Note that c_0 by convention is always $c_0 = 1$, hence it is an upper-triangular

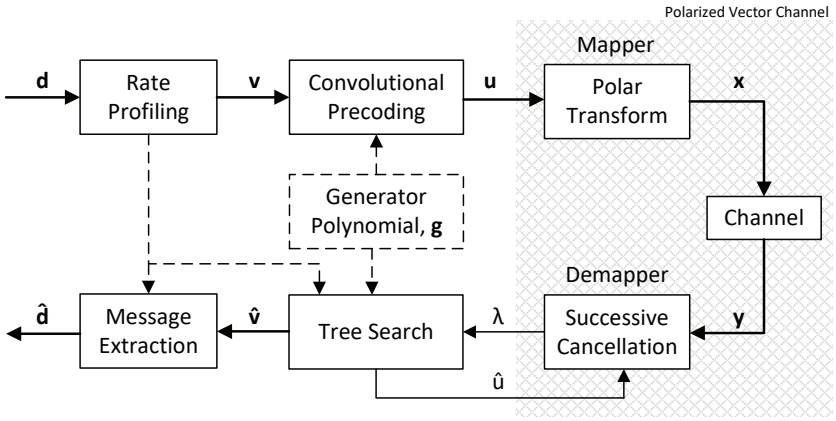

Fig. 2.4: PAC Encoding and Decoding Scheme

matrix. Then, we can obtain u by matrix multiplication as $\mathbf{u} = \mathbf{vG}$. As a result of this pre-transformation, u_i for $i \in \mathcal{A}^c$ are no longer frozen as in polar codes.

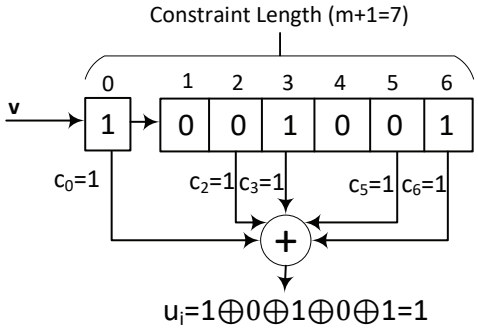

Fig. 2.5: An example of convolution using a shift-register

Since this convolutional transformation is one-to-one, it is not equivalent to a classical generator matrix of convolutional codes. The rate-profiling process performed before the convolutional transformation creates the redundancy by inserting $N - K$ zeros in the length-K input sequence $\mathbf{d}$.

Finally, as shown in Fig. 1 of [61], vector $\mathbf{u}$ is mapped to vector $\mathbf{x}$ ($\mathbf{x} = \mathbf{uP}_n$) by the polar transform $\mathbf{P}_n = \mathbf{P}^{\otimes n}$ defined as the n-th Kronecker power of $\mathbf{P} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$. Note that the notations $\mathbf{G}_2$ and $\mathbf{G}_N$ in polar codes have changed to $\mathbf{P}$ and $\mathbf{P}^n$ in PAC codes, respectively, due to the use of $\mathbf{G}$ for the convolutional transform. We also use diffident notations, depending on the context, in this thesis.

Chapter 3

Adjusted List Decoding

> *"It's so much easier to suggest solutions when you don't know too much about the problem."*
>
> — Malcolm Forbes

List decoding is the most popular decoding algorithm used for polar codes. From Chapter 2, we know that in the list decoding process, a number of candidates (usually a power of two) with the highest likelihood (or at the closest distance) to be the transmitted message are retained, and the rest are discarded after decoding each non-frozen bit. The maximum number of candidates is determined by a parameter named *list size* denoted by L. This parameter is fixed throughout the decoding process. However, if we divide a code block into equal-length segments, a.k.a partitions, with sufficient number of frozen bits to take advantage of position recovery explained in Section 3.3.3, usually 4 segments, we realize that the probability of the bit errors in the segments, $P_{seg}^{(i)} = \sum_{j=N_{seg}\cdot i}^{N_{seg}\cdot(i+1)-1} p_{e,j}$ where $i = 0, ..., N/N_{seg} - 1$, $j \in \mathcal{A}$ and N_{seg} is the segment size in bits, varies significantly, in particular in the first and last segments. Note that since we have a number of candidates in the list decoding, the block errors, i.e., the elimination of the correct path, may occur after more than one-bit error. A model will be presented in Chapter 4 for block errors in the segments. The imbalance of the bit error probability in the segments implies that we have overcapacity in terms of list size in some segments. As Fig. 3.1 shows the sketch of an example, while P_{seg} varies in the segments, the list size L remains constant. On the other hand, balancing $P_{seg}^{(i)}$ for $i = 0, 1, ..., N_{seg} - 1$ by employing a goal-oriented code design, may improve the block error rate.

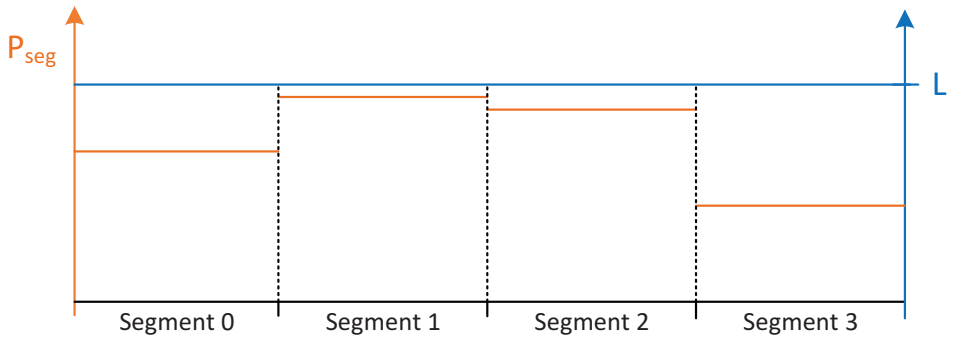

Fig. 3.1: The probability of bit error in the segments, P_{seg}

In this chapter, we follow two paths to balance the available resources with the probability of elimination of the correct candidate in the segments:

- First, we adjust the list size of each segment to the possibility of elimination of the correct candidate. Hence, the list size will not remain constant throughout the decoding process. This can reduce the computational complexity significantly with negligible degradation in the error correction performance. This path was studied in [37].
- Then, we keep the list size constant, but we balance the possibility of elimination of the correct candidate in the segments by modifying the polar code. This path was studied in [38].

In practice, the use of probability may not be convenient. Hence we introduce a parameter called *path metric range* or *PMR* to simplify the computations. Nevertheless, we employ the error probability of the segments later as an alternative and more accurate method. Let us begin by introducing the path metric range.

3.1 Path Metric Range, PMR

In this section, we investigate the behavior of the list decoder with respect to the evolution of the path metrics within the list throughout decoding a codeword. The relation between this evolution and the probability of an error occurrence is empirically analyzed [37]. Then, we will advocate that a fixed list size is not essential, and it can change throughout the decoding process, from one *partition* to another, depending on the properties of such partitions.

Definition 3.1 Partitions *[24] are defined as the sub-trees of the decoding tree, associated with code's sub-blocks of length* 2^m*,* $m < n$*, that divides the codewords into* 2^{n-m} *equal-length sub-blocks. The* j*-th partition and its associated sub-block are denoted by* P_j*, for* $0 \leq j \leq 2^{n-m} - 1$*.*

In order to characterize the partitions with respect to likelihood of an error occurrence in the list decoding, we introduce a new parameter that helps us to understand the evolution of path metrics throughout the decoding process:

Definition 3.2 Path metric range *for the* i*-th decoding step is defined as* $\mathrm{PMR}_i = PM_L^{(i)} - PM_1^{(i)}$*, where* $0 \leq i \leq N-1$*, assuming that the path metrics are sorted in ascending order, i.e.,* $PM_1^{(i)} < PM_2^{(i)} < \cdots < PM_L^{(i)}$*.*

Fig. 3.2 shows the changes of PMR throughout the decoding process for $PC(1024, 820)$ (orange curve), along with final LLRs (blue bars) and the frozen bits (red bars). As can be seen, the PMR curve elucidates the evolution of the path metrics within the list. Note that the path metric range scales with L; thus, PMR reduces if a smaller list size is used.

Now, to analyze the changes in PMR value with respect to LLRs, the following lemma is introduced.

Lemma 3.1 *If u_i is an unfrozen bit, i.e., $i \in A$, and $|\lambda_0^i[l]| < \mathrm{PMR}_{i-1}$ for all l then $\mathrm{PMR}_i < \mathrm{PMR}_{i-1}$.*

Proof: Assuming $PM_1^{(i-1)} < ... < PM_L^{(i-1)}$. After splitting the paths at step/bit i, the $2L$ path metrics are $PM_1^{(i-1)}, PM_1^{(i-1)} + |\lambda_0^i[1]|$, ... , $PM_L^{(i-1)}, PM_L^{(i-1)} + |\lambda_0^i[L]|$. The relation $PM_l^{(i-1)} < PM_l^{(i-1)} + |\lambda_0^i[l]|$ holds for $l = 1, 2, ..., L$. Thus, considering $|\lambda_0^i[l]| < PMR_{i-1}$, then $PM_1^{(i-1)} + |\lambda_0^i[1]| < PM_L^{(i-1)}$. Therefore all the paths greater or equal to $PM_L^{(i-1)}$ are pruned in order to make room for at least the new path $PM_1^{(i-1)} + |\lambda_0^i[1]|$. As a result, $PM_L^{(i)} < PM_L^{(i-1)}$, then $PMR_i < PMR_{i-1}$. ■

As a result of Lemma 3.1, a subsequence of k bits such that $|\lambda_0^m[l]| < PMR_{m-1}$, for all l and $m = i \dots i + k$ leads to a sharp drop of the PM range i.e.,$PMR_{i+k} \ll PMR_i$.

Here, let us distinguish the bit-channels causing the PMR drop by the following definition:

Definition 3.3 Crucial bits *are defined as the unfrozen bits with $|\lambda_0^i[l]| < \mathrm{PMR}_{i-1}$. S^j denotes the set of indices of crucial bits in the j-th partition.*

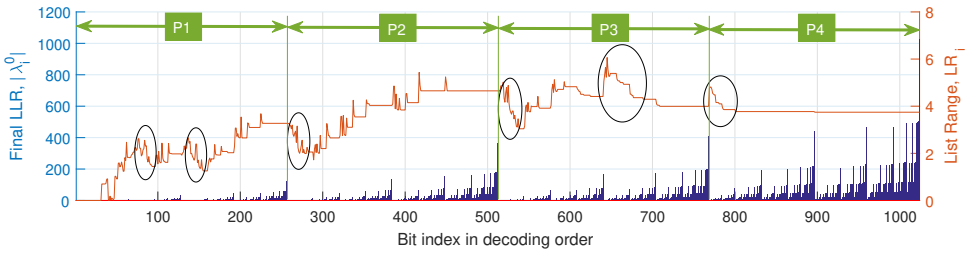

Fig. 3.2: Absolute values (average in 100 iterations) of bit-channel LLRs $|\lambda_0^i|$ and path metric range (LR_i), in natural decoding order for PC(1024,820)

The local minima on the PMR curve (orange) in Fig. 3.2 appear after a series of crucial bits (see circled sections in Fig. 3.2) where most of the errors are observed.

From Lemma 3.1 and its following discussion, one can infer that if u_i and u_j are crucial bits (possibly in different partitions) and $PMR_i \ll PMR_j$, there exists a list size L', $L' < L$, so that $|\lambda_0^j[l]| < PMR'_j$ still holds for all l. Note that since PMR scales with L, then $PMR'_j < PMR_j$. Here, L is the initial list size and PMR' is the new path metric range obtained after reducing the list size to L'. This leads us to the conclusion that reducing the list size in the partition with higher minimum PMR does not affect the error correction performance, if the new list size for that partition is chosen optimally.

As a practical method, the inverse of the average PMR over the crucial bits in j-th partition $PMR_{avg}^{(j)}$ can be employed to determine the local list size L_j, i.e.,

$$\log_2(L_j) \propto \frac{1}{PMR_{avg}^{(j)}} = \frac{|S^j|}{\sum_{S^j} PMR_i} \ .$$

For instance, the 4-tuple of average PMR (rounded) over crucial bits in the four partitions of the example shown in Fig. 3.2 is $(PMR_{avg}^{(j)})_{1\leq j\leq 4} = (2, 3, 4, 4)$. Thus, the list size for the four partitions are obtained by the mapping $(2, 3, 4, 4) \rightarrow (2^t, 2^{t-1}, 2^{t-2}, 2^{t-2})$, where the maximum list size ($L = 2^t$) is assigned to the partition with the smallest PMR_{avg}. Alternatively, if we use the PMR curve as a graphical tool, the maximum list size is assigned to the partition in which the global minimum of PMR curve is located. The list size for the rest of the partitions are assigned with respect to the local minima of PMR. Note that the PMR curve changes for different code lengths and code rates.

Now, we describe the impact of PMR on the possibility of error occurrence. Assuming l_c denotes the index of the correct path. In the list decoding process, the correct path may not always correspond to the smallest path metric $PM_1^{(i)}$. Due to the penalties, it may have a larger path metric $PM_{l_c}^{(i)} > PM_1^{(i)}$.

Fig. 3.3 shows different scenarios for the movement of the correct path (with index l_c) within the list throughout list decoding. The frequent penalty scenario mainly occurs in the partition(s) in which the local PMR_{avg} is smaller. The olive and orange curves in Fig. 3.3 show the changes in l_c in the indices below 300 which are located in P_1 and P_2. In the scenario shown by olive curve, the correct path is pruned after several penalties, while the orange curve shows the scenario where the correct path remains in the list.These scenarios show that the list size in P_1 should be large enough to retain the correct path within the list in case of bearing frequent penalties while in the subsequent partitions P_2 and P_3 where PMR_{avg}s corresponding

to crucial bits gradually increases, the list size can be reduced without any significant effects on the error correction performance.

In the last partition, since $PMR_{avg}^{(4)}$ is relatively large, if the correct path is penalized over some crucial bits, the path will remain in the list due to low number of crucial bits in that partition and consequently low probability of frequent penalties. Note that the index of penalized path does not increase significantly when L is large. The blue curve in Fig. 3.3 illustrates this scenario.

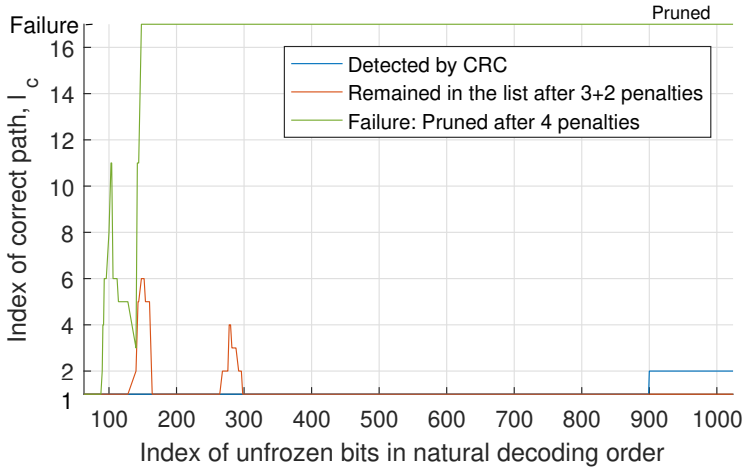

Fig. 3.3: Some sampled movements of correct path in the list, representing different scenarios, for PC(1024,820) and $L = 16$

3.2 Stepped List Decoding

Since the average path metric range (PMR_{avg}) varies from one partition to another, the fixed list size (L) in conventional list decoding is mainly effective on the partition(s) with the relatively small PMR_{avg}. In the partition(s) with significantly larger PMR_{avg}, the potential of the L is not fully used. Hence, we can allocate different list sizes to different partitions based on PMR_{avg}. In this scheme, the list size changes stepwise from one partition to another and that is the reason for calling it *stepped list decoding*. The effective list size for partitions are allocated based on PMR_{avg}. The computed PMR_{avg} for all the partitions are clustered with respect to a significant difference among them. For instance, the 4-tuple of rounded PMR_{avg} for four partitions of PC(1024,256) at 1dB is (-,38,23,24), where 23 and 24 will be in one cluster and therefore an identical list size will be assigned to both P_3 and P_4. Thus, the list size

mapping could be $(-, 38, 23, 24) \rightarrow (2, 2^{t-1}, 2^t, 2^t)$. As another example, since the 4-tuple of rounded PMR_{avg} for four partitions of PC(1024,512) at 1.4dB is (8,6,6,9), the list sizes could be allocated as $(2^{t-1}, 2^t, 2^t, 2^{t-2})$. It will be seen in section 3.2.2 that by proper allocation of list sizes to the partitions, the error correction performance will not degrade.

Note that in this chapter, although the examples are based on four partitions, the number of partitions could be larger.

Although the memory requirement differs from one code rate to another in stepped list decoding, the largest memory requirement can be used in multi-mode scheme, where different settings are employed by changing the code parameters.

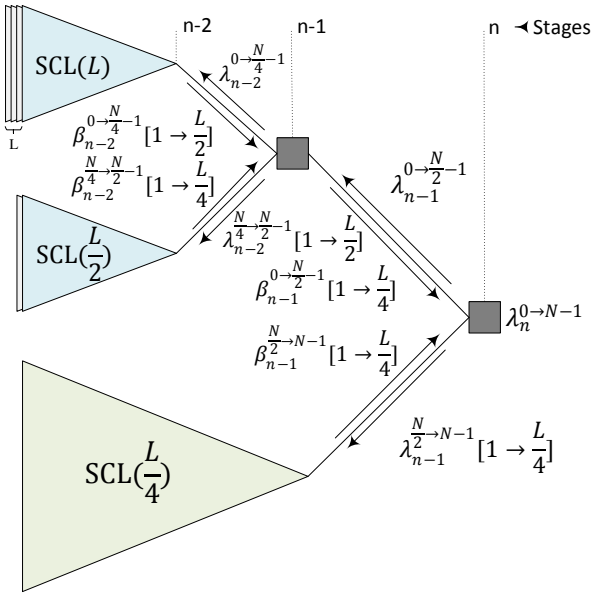

Fig. 3.4: Stepped list decoding tree for PC(1024,820)

3.2.1 Computational Complexity and Memory Requirement

The proposed stepped list decoding requires significantly less memory space and less computations than conventional (CRC-aided) SC List decoding process. In this section, we investigate the impact of the stepped list decoding on computational complexity and memory requirement for two examples discussed in the previous section, PC(1024,820) and PC(1024,512), where the 4-tuples of the list sizes for four partitions are $(L, L/2, L/4, L/4)$ and $(L/2, L, L, L/4)$, respectively, instead of fixed list size L for all the partitions.

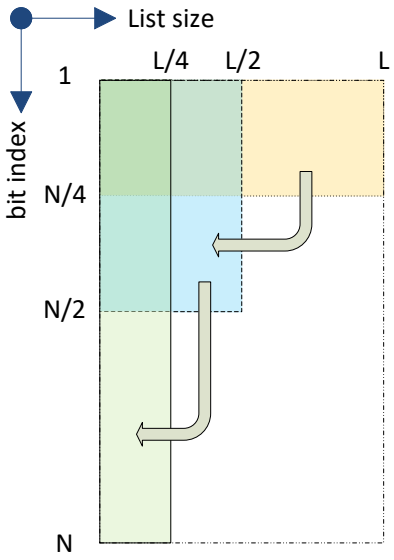

Fig. 3.5: Sketch of path memory in stepped list decoding of PC(1024,820)

3.2.1.1 Computational Complexity

Consider the computational complexity of the conventional list decoding, $L \cdot N \log_2 N$. Since the list size in the stepped list decoding changes, the computational complexity changes to (3.1), showing that the complexity reduces 50% in 3-step scheme (Fig. 3.4).

$$\underbrace{L\frac{N}{4}\log_2 N}_{\text{1st partition}} + \underbrace{\frac{L}{2}\frac{N}{4}\log_2 N}_{\text{2nd partition}} + \underbrace{\frac{L}{4}\frac{N}{2}\log_2 N}_{\text{3rd/4th partition}} = \frac{1}{2}LN\log_2 N \tag{3.1}$$

Similarly, the computational complexity for PC(1024,512) with the 4-tuple of list sizes $(L/2, L, L, L/4)$ can be derived as $\frac{21}{32}LN\log_2 N$, which is 34% less than conventional list decoding process.

Note that the reduction in complexity is SNR-independent unlike the tree/list pruning techniques.

3.2.1.2 Memory Requirement for Candidate Paths

Similar to computational complexity, the memory required for storing the estimated bits of the candidate paths is directly proportional to L. The partitioning helps to allocate the path memory efficiently. As the decoding proceeds from the last bit of one partition to the first bit of the next partition, since L is halved, half of the allocated memory is freed, as shown in Fig.

3.5. This freed space can be used for the next partition.

As a result, the memory reduction for the proposed 3-step scheme for PC(1024,820) is $L \cdot N - L/4 \cdot N$ bits or 75%.

Similarly, the path memory requirement for PC(1024,512) with the 4-tuple of list sizes $(L/2, L, L, L/4)$ can be derived as $\frac{3}{4}LN$, which is 33% less than conventional list decoding.

3.2.1.3 Memory Requirement for LLRs and Partial Sums

The memories required for internal LLRs and partial sums are proportional to the list size, similar to the path memory. In the conventional list decoding, the internal LLRs, $\lambda_{0\rightarrow n-1}^{0\rightarrow N-1}$, need $(N-1)\cdot L \cdot Q$ bits and the partial sums, $\beta_{0\rightarrow n-1}^{0\rightarrow N-1}$, require $(N-1)\cdot L$ bits [66]. We know that $N/2 \cdot L$ and $N/4 \cdot L$ out of $(N-1)\cdot L$ memory elements are allocated to stage $n-1$ and stage $n-2$, respectively. However, in the stepped list decoding for PC(1024,820), since the list size in every subsequent partition is halved, only half of the partial sums of preceding stage are sent backward. The aforementioned process is depicted in Fig. 3.4 by an index range in the bracket. Note that no bracketed index ranges have been added to λ_{n-1} of bit-channels 0 to $\frac{N}{2}-1$ and λ_{n-2} of bit-channels 0 to $\frac{N}{4}-1$ because they are not dependent on partial sums; therefore, they are the same for all paths in the list.

According to the above discussion for PC(1024,820), the memory requirement for LLRs and partial sums can reduce from (3.2) in the conventional list decoding, to (3.3) in the stepped list decoding.

$$M_{SCL} = \underbrace{L(N-1)Q_i}_{\text{Internal LLRs}} + \underbrace{L(N-1)}_{\text{Partial sums}} \tag{3.2}$$

$$\begin{aligned} M_{SCL-Stepped} = & \underbrace{\Big(L(\frac{N}{4}-1) + \frac{L}{2}(\frac{N}{4}) + \frac{L}{4}(\frac{N}{2})\Big)Q_i}_{\text{Internal LLRs}} \\ & + \underbrace{\Big(L(\frac{N}{4}-1) + \frac{L}{2}(\frac{N}{4}) + \frac{L}{4}(\frac{N}{2})\Big)}_{\text{Partial sums}} \\ = & \underbrace{L(\frac{1}{2}N-1)Q_i}_{\text{Internal LLRs}} + \underbrace{L(\frac{1}{2}N-1)}_{\text{Partial sums}} \end{aligned} \tag{3.3}$$

where Q_i is the number of bits used in quantization of internal LLRs. Note that in all cases, channel LLRs require an additional memory of NQ_{ch} bits, where Q_{ch} is the number of quantization bits for channel LLRs. This is omitted from the above equations for simplicity.

By comparing (3.3) with (3.2), it is concluded that the memory reduction using stepped list decoding is 50% in the proposed 3-step scheme for PC(1024,820).

Similarly, the memory required for internal LLRs and partial sums of PC(1024,512) with the 4-tuple of list sizes $(L/2, L, L, L/4)$ can be derived as $L(\frac{21}{32}N-1)Q_i + L(\frac{21}{32}N-1)$, which is 34% less than the conventional list decoding.

3.2.2 Numerical Results

The LLR-based stepped CA-SCL decoder is implemented for polar code of $N=2^{10}$ and the code rates $R = K/N = 0.8$ and 0.5 over AWGN channel. The polar codes are constructed using Bhattacharyya parameter (heuristic) method and optimized for high SNRs. The 16-bit CRC generator polynomial $g(x) = x^{16}+x^{12}+x^5+1$ is used for correct path detection. For stepped list decoding of PC(1024,820) and PC(1024,512), the list size 4-tuples $(32, 16, 8, 8)$ and $(16, 32, 32, 8)$, respectively, are used for the partitions. Fig. 3.6 compares the performance of conventional CRC-aided SCL decoder with the proposed one. The proposed stepped list decoding preserves the performance of conventional list decoding with fixed list size in various code rates.

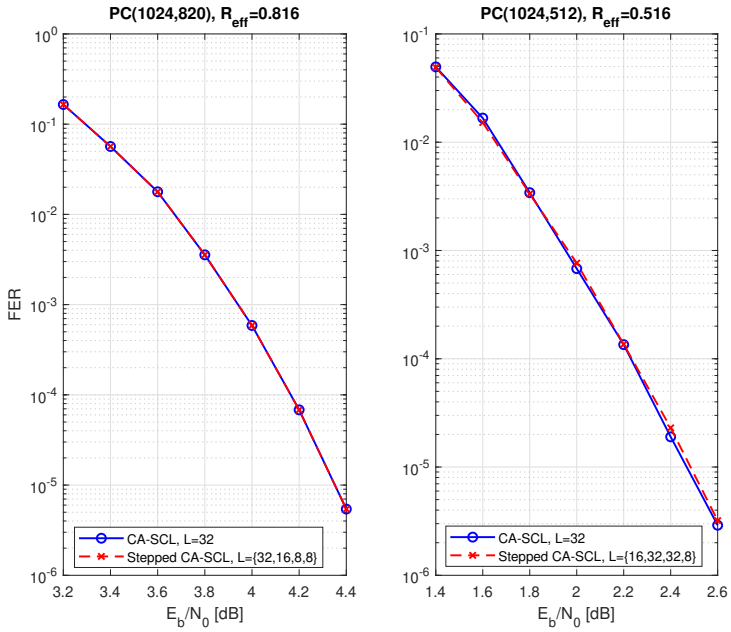

Fig. 3.6: Performance of the Stepped CA-SCL vs CA-SCL Decoding

3.2.3 An Alternative Algorithm

As finding average PMR_i is costly and requires running tens of iterations, an alternative method is proposed in this section. This method is based on the probability of bit error in each segment, $P_{seg}^{(i)}$

$$P_{seg}^{(i)} = \sum_{j=N_{seg}\cdot i}^{N_{seg}\cdot(i+1)-1} p_{e,j} \tag{3.4}$$

where $i = 0, ..., N/N_{seg} - 1$, $j \in \mathcal{A}$, $p_{e,j} \approx Q(\sqrt{|\lambda_0^i|/2})$ is the error probability of bit j and N_{seg} is the segment size in bits. Let us denote denote $n_{seg} = N/N_{seg}$ $\mathbf{P_{seg}} = [P_{seg}^{(0)}, P_{seg}^{(1)}, ..., P_{seg}^{(n_{seg}-1)}]$

By discarding the segments with $P_{seg}^{(i)} = 0$, i.e., the segments fully covered by frozen bits, we find the standard deviation σ_{seg} of $P_{seg}^{(i)}$'s. The standard deviation helps us in classification of the segments, to distinguish a meaningful difference between the segments in terms of distribution of low-reliability bits. Then, Algorithm 1 allocates local list sizes to the segments. Observe that we have three options for the list sizes of the segments; $L, L/2, L/4$. The segments with σ_{seg} and $2\sigma_{seg}$ less than P_{seg}^{max} will get list size $L/2$ and $L/4$, respectively. Note that in line 7, we have a constant for the number of frozen bits in the segments. This constraint guarantees to some extent the backward movement of the correct path as shown in Fig. 3.3. This downward movement is called position recovery and will be discussed further in section 3.3.3.

In this method, we trade off FER performance with complexity and memory requirement. Note that the FER performance in some cases based on the method in the previous section will degrade significantly, however, in the alternative method, the FER performance will be preserved almost for any codes. In fact, the reason for FER degradation with the main method for Stepped decoding is not taking advantage of frozen bits in the beginning of each segment for position recovery.

Fig. 3.7 shows numerical results for FER performance using the alternative algorithm, Algorithm 1 based on four segments for various codes. The generator polynomial used for CRC and for precoding in PAC codes are 0xA5 and 0o133, respectively. The polar codes for $N = 256$ and 512 are designed by DE/GA method [9] with design-SNRs 2 dB and 4 dB, respectively.

As can be observed, the performance of the codes is preserved with this method, however there is a negligible degradation when we use CRC-polar codes. The main reason is a smaller number of choices for detecting the correct path at the end of decoding (due to elimination of the correct path in the last segment) while in non-CRC-aided decoding, the path with highest

Algorithm 1: Stepped List Decoding: Allocation of local list sizes to the segments

input : $\mathcal{A}$, L, $\mathbf{P_{seg}}$, σ_{seg}, n_{seg}
output: $\mathbf{L} = [L_0, L_1, ..., L_{n_{seg}-1}]$

// Initialization;
$P_{seg}^{max} = \max(\mathbf{P_{seg}})$;
$\mathbf{N_{seg}^{fz}} \leftarrow$ Find the number of frozen bits in each segment
for $i \leftarrow 0$ **to** $n_{seg} - 1$ **do**
    if $P_{seg}^{(i)} = 0$ **then**
        $L_i = 1$;
    else if $P_{seg}^{(i)} < P_{seg}^{max} - \sigma_{seg}$ *and* $N_{seg}^{fz}[i] > 3$ **then**
        if $P_{seg}^{(i)} < P_{seg}^{max} - \sigma_{seg}$ **then**
            if $L_{i-1} \leq L/2$ *and* $i \neq 0$ **then**
                $L_i = L/4$;
            else
                $L_i = L/2$;
        else
            $L_i = L/2$;
    else if $P_{seg}^{(i)} < P_{seg}^{max} - \sigma_{seg}$ **then**
        $L_i = L_{i-1}$;
    else
        $L_i = L$;

likelihood is selected as a solution.

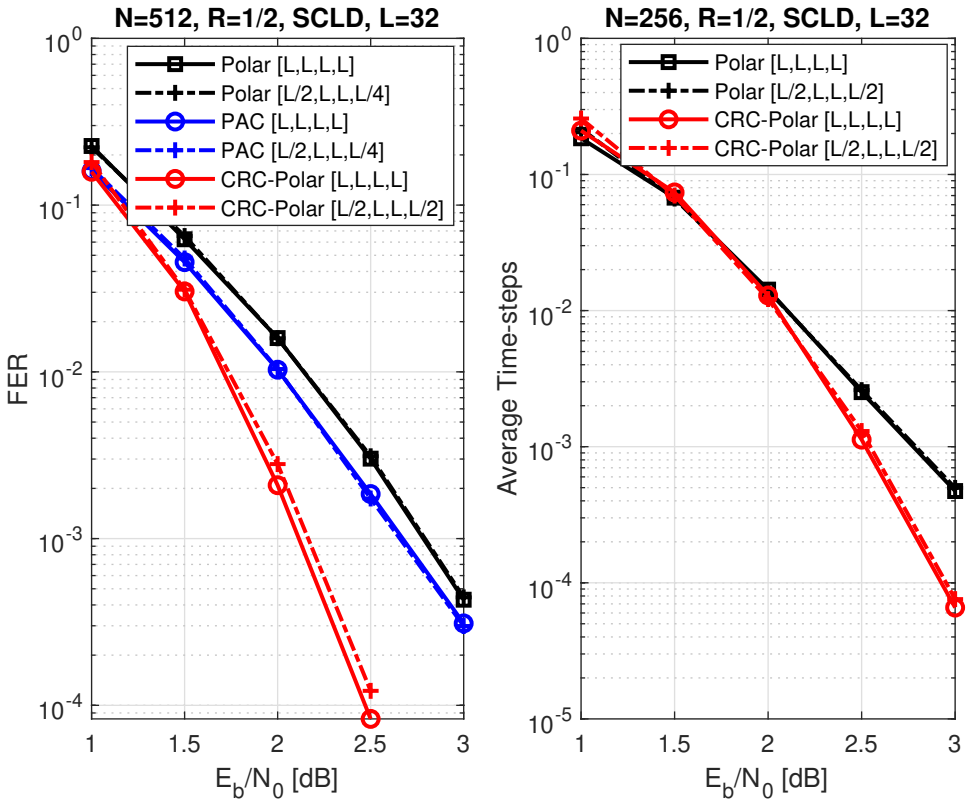

Fig. 3.7: Performance comparison of the alternative algorithm for allocation of the local list sizes

3.3 Error Occurrence in List Decoding

Toward balancing the error probability in the segments and improving the performance of the code, as discussed earlier in this chapter, the properties of list decoding is first investigated in this section. This analysis is instrumental in understanding the behavior of list decoding and will be used to devise an approach for code modification in Section 3.4.

3.3.1 Error Occurrence in SC and SCL Decoding

In the SC decoding, the error occurs by the first wrong estimation of a bit-value and then this error successively propagated. However, in SCL decoding, the correct candidate might remain in the list until end of the process despite one wrong error in estimation of a bit-value based on ML rule in (8.5) and accepting a penalty based on (8.6).

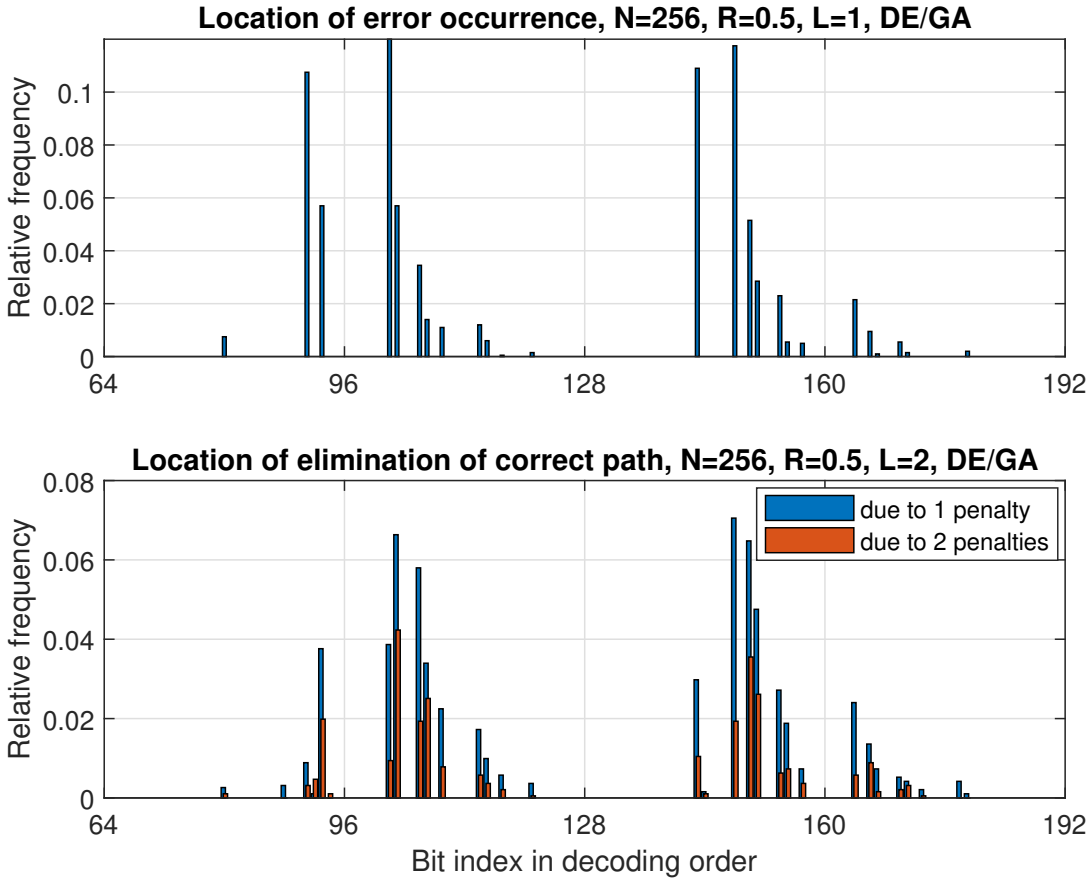

Fig. 3.8: Error occurrence in SC (or SCL when L=1) and SCL decoding

As Fig. 3.8 illustrates, on the bit indices where the peak of errors in SC decoding, the SCL decoding can retain the correct path in the list and thus the error (i.e., the elimination of the correct path due to one penalty) on those bits reduces. The orange bars shows the location of correct path elimination due to two penalties which are located on the indices larger than the indices of peak of errors on the SC graph. That means SCL decoding is tolerating a penalty (in case of L=2) and retaining the correct path until the second penalty occurs. This is the reason why the peaks of elimination due to 2 penalties in segments 64-128 and 128-192 in the bottom graph appear after the peaks in the corresponding segments in the top graph. Note that among the data collected in SCL decoding with L=2 in Fig. 3.8, only 17 eliminations out of 2000 eliminations occurred due to 3 or 4 penalties which can be explained by recovery phenomenon in section 3.3.3.

An interesting point here is that unlike SC decoding, most of the the eliminations are not occurring on the least reliable bits in segments 64-128 and 128-192 although most of the penalties

occurring there.

In order to devise a model for probability of elimination of the correct path, in the next section, we propose a parameter that expands our understanding of list decoding behavior.

3.3.2 Path Metric Range as a Tool

Evolution of the PMs of the paths remaining in the list throughout the decoding process indicates when the correct path might be eliminated from the list, i.e., block error occurrence. To characterize the sub-blocks of polar codes with high likelihood of error occurrence, we use the parameter PMR and the notion of crucial bits introduced Section 3.1.

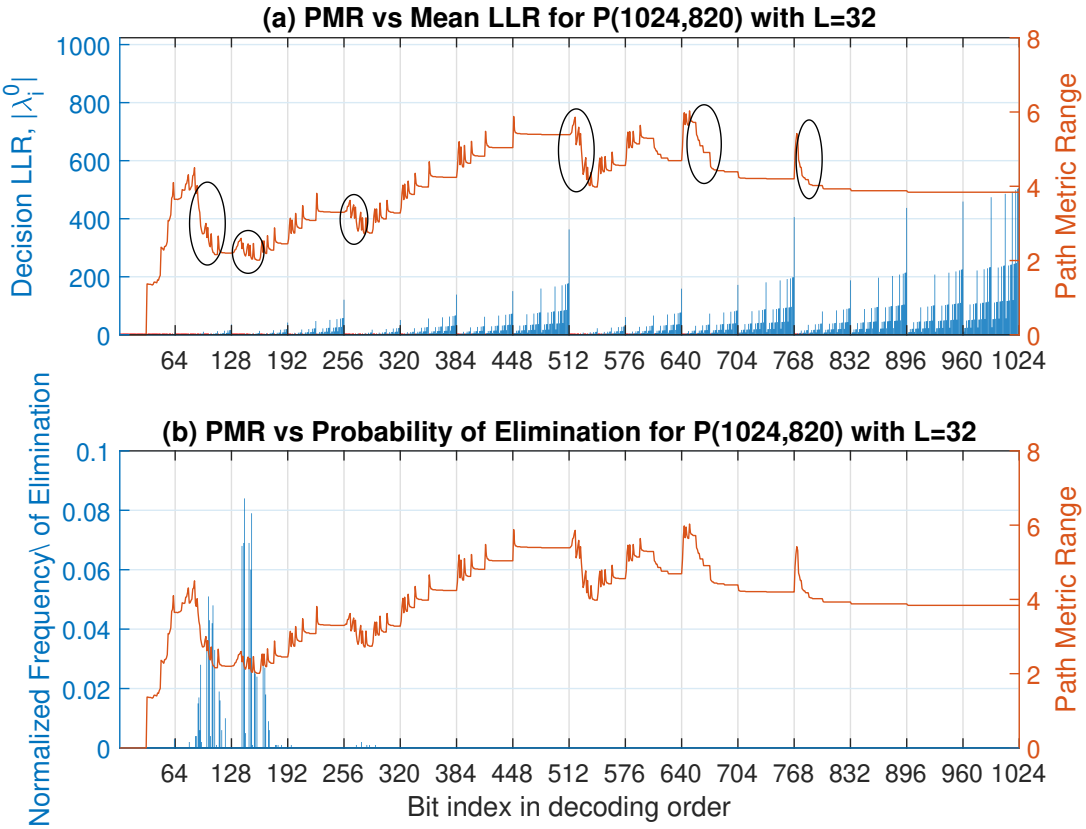

Fig. 3.9: Absolute values of bit-channel LLRs $|\lambda_0^i|$ and path metric range (PMR), averaged, in decoding order for $\mathcal{P}(1024, 820)$ and L=32

The local minima on the PMR curve (orange) in Fig. 3.9 (a) appear after a series of crucial bit. Most of the eliminations of the correct path are observed in the sub-block that includes the global minimum given that there are enough number of low-reliability bit-channels. The reason is that the bit-channels with low-reliability and relatively small LLR could be penalized

and due to small or narrow PMR, this results in $PM_{l_c}^{(i-1)} + |\lambda_i[l_c]| > PM_L^{(i-1)}$, which is led to elimination of the correct path in the pruning process. We can write this inequality in terms of PMR as follows: $(PM_{l_c}^{(i-1)} - PM_1^{(i-1)}) + |\lambda_i[l_c]| > PMR_{i-1}$. In [17], $(PM_{l_c} - PM_1)$ was called relative path metric (RPM). Thus, the narrower, the PMR is, the higher the possibility of elimination of correct path will be. If PMR is large or wide, a large penalty $|\lambda_i[l_c]|$ is required for elimination of correct path which is less likely to happen due to higher reliability of the bit-channels with large LLR magnitude.

The following lemma indicates the bit-channels with relatively higher reliability in the sub-block where PMR stays steady.

Lemma 3.2 *If $i \in A$, i.e., u_i is a free bit, and $|\lambda_0^i[l]| > \mathrm{PMR}_{i-1}$ for all l then $\mathrm{PMR}_i = \mathrm{PMR}_{i-1}$.*

Proof: Since $|\lambda_0^i[l]| > PMR_{i-1}$ for all l, then PM of penalized paths at decoding step i will be $PM_l^{(i-1)} + |\lambda_0^i[l]| > PM_L^{(i-1)}$, causing these paths to be pruned. Thus, since $PM_l^{(i)} = PM_l^{(i-1)}$, the path metric range remains unchanged, i.e., $PMR_i = PMR_{i-1}$. ■

Note that increase in PMR is caused by the frozen bits where the penalty (mainly due to error propagation, refer to the discussion in Section 3.3.3) increases PM value of most of the paths in the list except the correct path. This consequently widens PMR over frozen bit-channels as shown by jumps on PMR curve in Fig. 3.9 (a).

More reliable sub-blocks are the ones in which there is a smaller number of crucial bits and the local minimum of PMR curve is significantly larger than global minimum. Note that the number of sub-blocks has to be power of 2 to be matched with the distribution of row-weights in $\mathbf{G}_N$.

3.3.3 Position Recovery of Correct Path

When the correct path (with index l_c) is penalized during list decoding, i.e., $PM_{l_c}^{(i)} + |\lambda_0^i[l_c]|$ in (8.6), it moves from a position with a higher likelihood of being the correct path to a position with the lower likelihood (e.g. from position 2 to 4 in the ascending ordered list based on PM-value). However, under certain circumstances which will be explained in this section, the correct path moves in the opposite direction and gradually recovers the position 1 as shown in Fig. 3.10.

It is known that there are two sources of error in estimating the bit-values: 1) channel

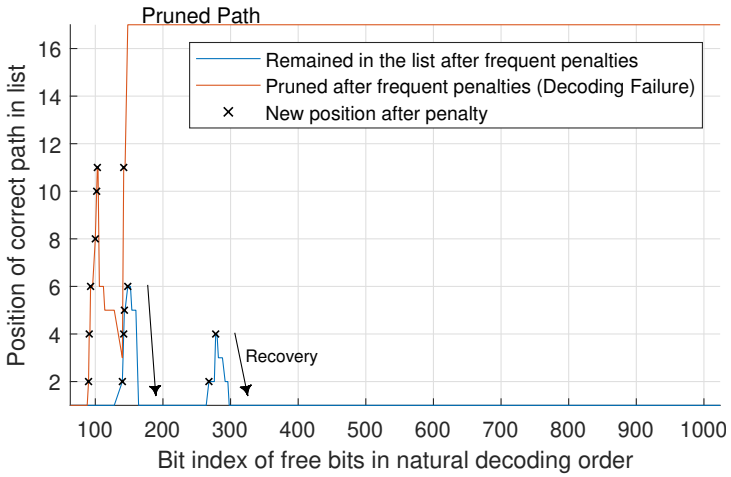

Fig. 3.10: Examples for movement of the correct path within the sorted list of paths for $\mathcal{P}(1024, 820)$ with L=16

noise that affects low reliability bit-channels (event N), 2) error propagation (event G) due to dependency to the previously estimated bits. Let us first consider the scenario that estimation error occurs due to event G only. In decoding of frozen bits, since the bit-values are known, the bits are estimated correctly; however, due to error propagation, inequality (3.5) holds for the probability of estimating u_i caused by event G, P_G, which may lead to imposing penalty on the incorrect paths.

$$P_G(\hat{u}_i = 0|\mathbf{U}_0^{i-1} = \hat{\mathbf{u}}_0^{i-1}[l_c]) > P_G(\hat{u}_i = 0|\mathbf{U}_0^{i-1} \neq \hat{\mathbf{u}}_0^{i-1}[l]) \tag{3.5}$$

where $l = \{1, 2, \cdots, L\} \backslash l_c$.

Inequality (3.5) says that when all the previous bits U_0^{i-1} are estimated correctly (i.e., $\mathbf{U}_0^{i-1} = \hat{\mathbf{u}}_0^{i-1}[l_c]$), the probability of estimating bit i correctly (without penalty) is higher than the cases that some of the previous bits are estimated incorrectly, which is true about incorrect paths. As a result, the incorrect paths over the frozen bits with relatively high reliability are penalized in most cases. This might end in the following movement of PM_{l_c}, for instance: $(PM_1^{(i-1)}, \ldots, PM_{l_c-1}^{(i-1)}, PM_{l_c}^{(i-1)}, \ldots, PM_L^{(i-1)}) \rightarrow (PM_1^{(i-1)} + |\lambda_0^i[1]|, \ldots, PM_{l_c}^{(i-1)}, PM_{l_c-1}^{(i-1)} + |\lambda_0^i[l_c - 1]|, \ldots, PM_L^{(i-1)} + |\lambda_0^i[L]|)$. By accumulation of these penalties after a sub-sequence of frozen bits (e.g. k bits), the path metrics of incorrect paths grow larger, and this eventually results in $PM_l^{(i+k)} > PM_{l_c}^{(i+k)}$ for $l = \{1, 2, \cdots, L\} \backslash l_c$, which means $l_c = 1$.

Now, let us involve the estimation error caused by event N in the process, particularly over low reliability bits where P_N is relatively high. In this scenario, the correct path may also be penalized. Therefore, the correct path does not move to a more likelihood position.

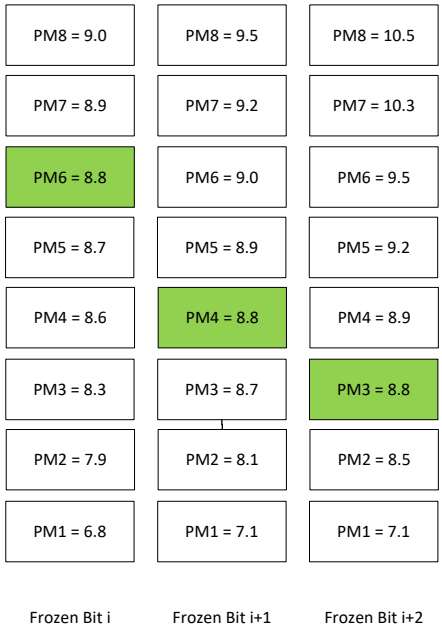

Fig. 3.11: A numerical example of growing of path metrics (PMs) of incorrect path resulted in pushing the correct path (green boxes) downward

Nevertheless, as our observation shows, at a sub-sequence of frozen bits which includes high reliability bits, the absence of event N in the decoding of a few bits is enough for a partial recovery to happen.

The heatmap in Fig. 3.12 shows the accumulated movements of correct path throughput the decoding process. As can be seen, there is a downward movement over a subsequence of bit-channels which are all or predominantly frozen bits. Note that the recovery might be partial or full. The full recovery mean the correct path can move down to position 1.

In [9], this recovery occurs by introducing dynamic frozen symbols of type II where u_i, $i \in \bar{\mathcal{A}}$, is set to some linear function of non-frozen bits u_0^{i-1}. As a result, the incorrect paths are most likely penalized and their path metrics grow resulting in the correct path moving downward. The experiments show that employing dynamic frozen symbols results in obtaining a code with higher minimum distance. note that in the scheme proposed in [9], there is no CRC to detect the correct path at the end of the decoding process. Thus dynamic frozen bits of type I and II help to move the correct path downward and keep it on the most likely position in the list. However, finding the effective linear combination of some random non-frozen bits to perform a full recovery is a difficult job.

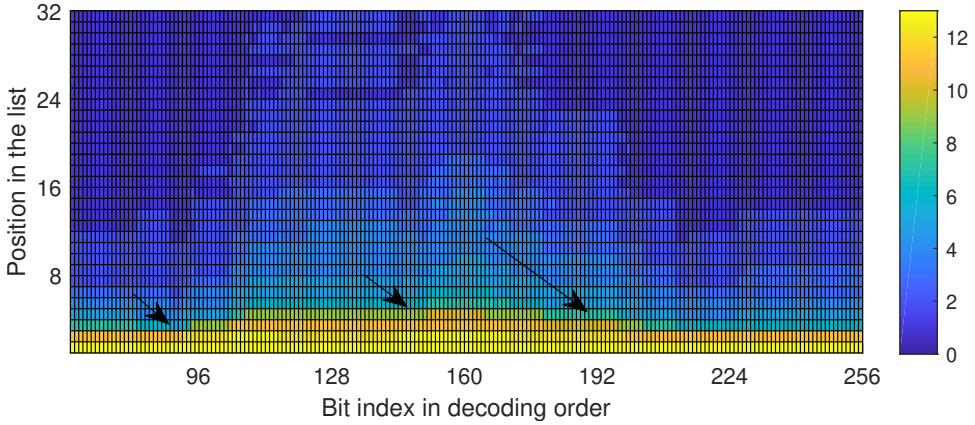

Fig. 3.12: Movement of correct path during decoding of 320,000 random codewords of P(256,128) with L=32. The arrows show the recovery

3.3.4 Elimination of Correct Path

The correct path is pruned from the list, predominately, due to multiple penalties. The larger the list size is, a larger number of penalties can be tolerated. Experimental results show that the events resulting in the elimination of the correct path can be classified into three categories for which the probability of elimination are estimated as follows:

3.3.4.1 Penalty in segment(s) with small PMR

When the correct path is penalized and the penalty value is large enough, the updated path metric becomes larger than the largest path metric in the list from the previous decoding step. i.e., $PM_{l_c}^{(i-1)} + |\lambda_i[l_c]| > PM_L^{(i-1)}$, then the correct path is pruned from the list. The correct path could be on the position 1, $l_c = 1$, in this case, PMR should be locally small such that the updated $PM_{l_c}^{(i)}$ could surpass $PM_L^{(i-1)}$. However, if the correct path has been penalized in the previous decoding steps and $l_c > 1$, the pruning might also occur over the bits with locally large PMR. Fig. 3.13 shows that most of the eliminations are happening where PMR is at low level, where the reliability of the bit-channels is relatively low. The elimination at bits with relatively high PMR occurs due to $l_c > 1$. This event is denoted event A and the probability of elimination in this case is estimated as below:

$$Pr_{A_i} \approx Pr(P_i | \cap_{v=1}^{i-1} \overline{E}_v) \cdot Pr(\mathrm{PM}_{l_c}^{(i-1)} + |\lambda_i[l_c]| > \mathrm{PM}_L^{(i-1)}) \tag{3.6}$$

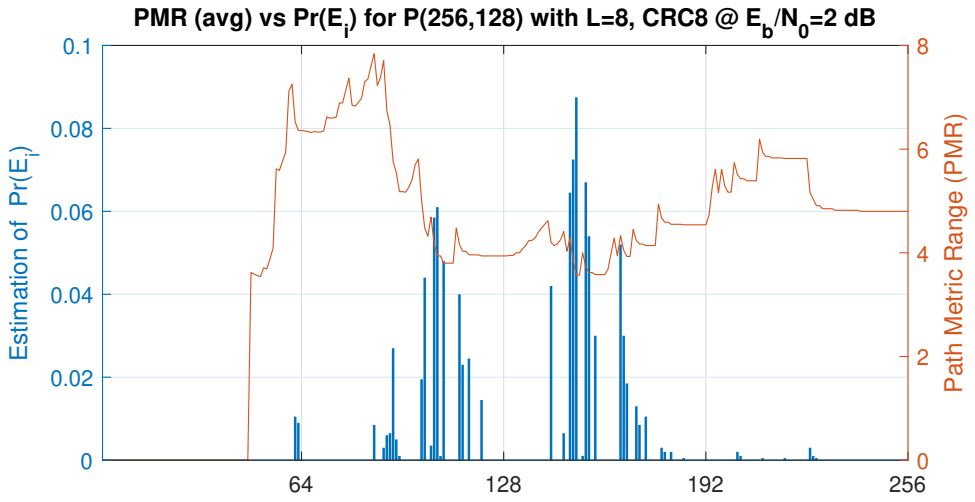

Fig. 3.13: Relation between PMR and the event of elimination of correct path

where P_i and E_i denote the events that the correct path being penalized and being eliminated, respectively, at bit i.

3.3.4.2 Incorrect paths versus correct path

The correct path is not penalized but the position of the correct path in the list is such that the path metric(s) of penalized branches of incorrect paths become smaller than path metric of the correct path. This event happens when there are incorrect paths with index $l < l_c$ which have path metric $PM_l^{(i-1)} < PM_{l_c}^{(i-1)}$ and quite small $|\lambda_i[l]|$ such that the penalized split paths will have path metric $PM_l^{(i)} + |\lambda_i[l]| < PM_{l_c}$. This results in pushing out the correct path from the list in the pruning process in order to make room for the incorrect path with smaller path metric. This event only happens on the bit-channels with least reliability scores given that the aforementioned condition satisfies. Thus, the share of this event in total number of elimination is not significant. The probability of this event which is denoted by B is estimated as follows:

$$Pr_{B_i} \approx \Big(1 - Pr(P_i | \cap_{v=1}^{i-1} \overline{E}_v)\Big) \cdot \prod_{L-l_c \leq |\{l | l<l_c\}|,\, l_c \geq L/2} Pr(PM_l^{(i-1)} + |\lambda_i[l]| < PM_{l_c}^{(i-1)}) \tag{3.7}$$

3.3.4.3 Combination of event A and B

When the correct path is penalized but the condition for event A is not satisfied, if the condition of even B satisfies for $l_c < L/2$, the correct path is eliminated. In the other words, if the amount of penalty is not enough to make the correct path pruned, the incorrect paths with $l < l_c$ satisfy the condition of event B , the correct path is pruned. The experimental results show that this event which we denote by C rarely happens.

In summary, the probability of elimination of the correct path at bit i is upper bounded by:

$$Pr(E_i) \leq Pr_{A_i} + Pr_{B_i} + Pr_{C_i} \tag{3.8}$$

However, since event B and C do not contribute significantly in total number of eliminations of the correct paths, for the sake of simplicity, we approximate the probability of elimination by $Pr(E_i) \approx Pr_{A_i}$

Now Let us analyze the effect of change of parameters involved in (3.6). As discussed in [37], throughout the decoding process, by increasing the bit index within a block-segment, PMR and consequently PM_L decrease over low-reliability bits while $|\lambda|$ increases. That means the term $Pr(PM_{l_c}^{(i-1)} + |\lambda_i[l_c]| > PM_L^{(i-1)})$ increases. In this situation, if the correct path is penalized over bit i (the event P_i) given that the correct path is still in the list, the probability of elimination, $Pr(E_i)$, also increases. Now, the position of the correct path in the list, l, depends on the history of penalties tolerated by the correct path over the previous bits. If the correct path has been penalized at least once over the previous bits, then $l_c > 1$ and PM_{l_c} is closer to PM_L; therefore, $Pr(|\lambda^i[l_c]| > PM_L^{(i-1)} - PM_{l_c}^{(i-1)})$ increases. On the other hand, as $|\lambda_0^i|$ increases, $Pr(P_i) \approx Q(\sqrt{|\lambda_0^i|/2})$ [9] reduces. As a result, $Pr(E_i)$ for *vulnerable bits* [30] increases and reaches a peak and then gradually reduces as the blue bars in Fig. 3.13 show for sub-blocks covering bits 64-128 and bits 128-192. The set of vulnerable bits $\mathcal{V}_j$ are the low-reliability non-frozen bits usually with minimum row weight (in $\mathbf{G}_N$) located in the j-th sub-block.

3.4 Goal-oriented Code Modification

In this section, we show how to modify the construction of polar codes based on what we learned from the analysis in Section 3.3. This modification adapts polar codes to the list

decoding process and results in an improvement in the error correction performance.

3.4.1 How to reduce probability of elimination?

Before proposing a method for code modification, we look at the factors involved in the elimination of the correct path. As (3.6) shows, low-reliability bits can contribute in the elimination in two ways: (1) If PMR is large, they may penalize and move the correct path to a location in the list with index $l_c > 1$. If the penalty $|\lambda_i[l_c]|$ is large, the correct path might be eliminated directly despite large PMR. (2) If PMR is small, they may cause pruning of the correct path regardless of the value of index l_c. Note that PM_L in (3.6) is proportional to PMR; $PMR_i = PM_L^{(i)} - PM_1^{(i)}$. Thus, if we can shift the PMR curve upward in the segments where the probability of elimination is high, we can improve the error correction performance. In section 3.3.3, it was explained how frozen bits cause growing the path metrics in the list and consequently increasing PMR. The gradual increase of PMR in Fig. 3.13 is due to the frozen bits. Now, if we freeze a low-reliability bit which may cause the elimination immediately (i.e., over that specific bit) or later (i.e., over next bits by increasing $l_c > 1$), not only it will contribute in reducing the eliminations due to that specific bit, but also affect the PM_L or PMR over the next bits and consequently reduces $P(E_i)$ according to (3.6). Consider a segment of bit-channels, e.g. segment 64-128 in Fig. 3.13, the question is which bit to freeze? Shall we freeze the bit that the largest number of eliminations is occurring there (the tallest blue bar) or the least reliable bit? Note that these two bits are not the same. Freezing the least reliable bit will have more impact because that bit is also indirectly contributing in the eliminations over the next bits. In order to retain the code rate unchanged, we need to unfreeze a *strong frozen bit*, i.e., frozen bit with relatively high reliability, in a segment that the least number of elimination is happening there. In Fig. 3.13, this segment is bit-segment 192-256. Although this unfrozen bit has a low reliability but since PMR and PM_L are relatively large, according to (3.6), it does not cause a significant number of elimination. The process of unfreezing and freezing bit-channels in pairs is called bit-swapping.

3.4.2 Code Modification

Based on the divide-and-conquer principle, we divide the block code into sub-blocks (indexed by $j = 0, 1, ...$). By choosing the right length for sub-blocks, $M = N/2^m = 2^{n-m}$ where $2^m = sb$ is the number of sub-blocks, based on the recovery phenomenon and the redistribution of some

of the vulnerable bits among the sub-blocks, we can reduce the probability of elimination in the sub-blocks upper bounded by $\sum_{i\in\{j.2^m,\ldots,(j+1).2^m-1\}} P(E_i)$. The limited redistribution of bits among sub-blocks by bit-swapping results in balancing the probability of elimination in different sub-blocks.

Since bit-swapping (i.e., unfreezing a strong frozen bit in response to freezing a vulnerable bit) affects the properties of sub-blocks, this process is performed one pair at a time, then the properties of sub-blocks are extracted again and the next pair for swapping is chosen accordingly. Here, for the sake of simplicity and also due to effectiveness of freezing least reliable non-frozen bits (see section 3.4.1), we can find a certain number of low-reliability non-frozen bits in the whole code block and then list them in each sub-block and count them to find $\{|\mathcal{V}_j|$. These numbers along with minimum local PMRs (averaged over enough number of iterations) could be used for characterizing the sub-blocks. For measuring the reliability of bits, one cane use the evolved mean of LLRs extracted from density evolution with Gaussian approximation (DE/GA) method [9]. Note that the sub-blocks with larger $\{|\mathcal{V}_j|$ usually have smaller min PMR because there are more crucial bits that cause dropping the PMR value gradually in the sub-block.

The objective is to minimize the maximum element of the set $\{|\mathcal{V}_1|, |\mathcal{V}_2|, \ldots, |\mathcal{V}_{sb}|\}$, subjected to three constraints: 1) the number of the bits frozen is equal to the number of the bits freed (to retain the code rate), 2) the number of the vulnerable bits frozen in each sub-block is proportional to the number of vulnerable bits in that sub-block, 3) the number of the frozen bits freed in each sub-block is inversely proportional to the total number of crucial bits in that sub-block.

3.4.3 Bit-Swapping Algorithm

To modify the construction, assuming b pairs of free/frozen bits are required to swap. Since we cannot determine the optimal value of b, if b is chosen larger than optimal in Algorithm 2, after an optimal number of pairs was swapped, the swapping will alternate between a specific pair of free and frozen bits back and forth; for instance, between indices $142 \rightleftarrows 91$, which does not affect the performance. Thus, choosing the right value for b is not crucial. These b bits are selected one at a time from a fixed number of candidates (assuming there are $c + c$ candidates among frozen and free bits). The number of combinations for bit-swapping appears to be huge. For instance, if $c = 15$ bits and $b = 10$ bits, the total number of combinations will be $\binom{15}{10} \cdot \binom{15}{10} = 3003 \times 3003$. Fortunately, the constraints for swapping the bits in sub-blocks

reduce the number of potential combinations significantly.

Algorithm 2 illustrates a greedy search that implements the bit-swapping process. In this algorithm, a pair of one free bit and one frozen bit are swapped at each cycle of the main loop in line 1. For the bit-swapping process, the properties of bit-channels and sub-blocks such as reliability of bit-channels, the number of low reliable bits in each sub-block, are required. Thus, at the beginning of every cycle of the main loop, the average of LLRs and PMRs are extracted from the decoding of some random codewords (line 3). $\overline{\text{LLR}}$ is used as a reliability measure in the algorithm, and it helps in finding the best pairs for swapping at each cycle. The $\overline{\text{LLR}}$ are sorted and then in lines 4 and 5, the indices of c most reliable frozen bits and c least reliable free bits are stored in the first row of the 2-dimensional vectors W and Z, respectively, along with their associated sub-blocks in the second row. The first inner loop in line 12 to 19 is used for finding the best candidate bit to freeze. This bit is searched among the least reliable bits in the sub-blocks with the largest number of low reliable bits because according to Section 3.3.4, the probability of elimination in these sub-blocks is higher. The number of free potential candidates (top c indices in W) in the sub-blocks is counted in line 6 and the sorted indices of sub-blocks are stored in vector B_w.

The second inner loop in line 22 to 29 is used for finding a strong frozen bit to make it free. This bit is searched among the most reliable frozen bits in the sub-blocks with the small number of crucial bits, $|S^j|$. The values of $D[j] = |S^j|$ are computed based on $\overline{\text{PMR}}$, then sorted and stored in vector D in line 8. This algorithm requires to recognize the end of tree expansion (i.e., the index of $\log_2 L$-th free bit named $logL\_bit$) in order to avoid freezing any bit before that (line 9).

Note that for obtaining a better results, the process of bit-swapping could be performed by considering more than one choice for freezing, in particular, at each swapping step and studying the its effect on the next swapping step. For good codes constructed using DE/GA or Tal-Vardy method, a fewer number of swapping is required and the bits for freezing should be chosen carefully. This approach may not improve significantly the short codes because the number of choices for swapping is limited.

As we suggested for the stepped list decoding in Section 3.2.3, since finding $\overline{\text{LLR}}$ and $\overline{\text{PMR}}$ is costly, alternatively, one can use $p_{e,j}$ and $P_{seg}^{(i)}$ in (3.4), as measures for reliability of bit j and segment i, respectively. Note that similar to Algorithm 2 where we update $\overline{\text{PMR}}$ after swapping each pair of bits (freezing and unfreezing), we need to update $P_{seg}^{(i)}$ as well.

Algorithm 2: Goal-oriented Code Modification: Bit-Swapping Process

input : $\mathcal{A}$, L, b, c, sb
output: modified $\mathcal{A}$

for $i \leftarrow 1$ **to** b **do**
  // Initialization;
  Extract $\overline{\text{LLR}}$ and $\overline{\text{PMR}}$ vectors from decoder($\mathcal{A}$, L);
  $[W[1], W[2]] \leftarrow$ 1: indices of top c low-reliability free bit-channels in ascending order of their $\overline{\text{LLR}}$s, and 2: indices of their associated sub-blocks;
  $[Z[1], Z[2]] \leftarrow$ 1: indices of top c high-reliability frozen bit-channels in descending order of $\overline{\text{LLR}}$s, and 2: indices of their associated sub-blocks;
  $[B_w, B_z] \leftarrow$ count number of bits in W and Z vectors existing in sub-block 1 to sub-block sb;
  $B_w^s \leftarrow$ indices of sorted B_w in descending order;
  $D \leftarrow$ count crucial bits existing in each sub-block, then sort the non-zero ones in ascending order;
  $logL\_bit \leftarrow$ index of $(\log_2 L)$-th free bit;
  // Freezing a vulnerable bit:
  $j \leftarrow 0$;
  do
    $j \leftarrow j + 1$;
    for $k \leftarrow 1$ **to** sb **do**
      if $W[2][j] = B_w^s[k]$
      and $B_w[W[2][j]] > B_z[W[2][j]]$ **then**
        exclude bit $W[1][j]$ from set $\mathcal{A}$;
        break;
  while $k = sb+1$;
  // Unfreezing a strong frozen bit:
  $j \leftarrow 0$;
  do
    $j \leftarrow j + 1$;
    for $k \leftarrow 1$ **to** sb **do**
      if $Z[2][j] = D[k]$
      and $Z[1][j] > logL\_bit$ **then**
        include bit $Z[1][j]$ into set $\mathcal{A}$;
        break;
  while $k = sb+1$;

3.5 Numerical Results

The LLR-based CRC-aided (CA) SCL decoder is used for evaluation of bit-swapping on polar codes of length $N=2^{10}$ and the code rates $R = K/N = 0.8$ and 0.5 over AWGN channel. The 16-bit CRC generator polynomial $g(x) = x^{16}+x^{12}+x^5+1$ is used in CA-SCL decoding. Polar codes are first constructed using Bhattacharyya parameter method and optimized for high SNRs. The design-SNRs for $R = 0.5$ and 0.8 are 5 and 7 dB, respectively. The bit-swapping process is performed at $E_b/N_0 = 2$ and 4 dB for $R = 0.5$ and 0.8, respectively. Fig. 3.14 compares the performance of polar codes constructed using the Bhattacharyya parameter method and modified code via bit-swapping under CA-SCL decoding. The proposed code modification for list decoding improves the performance of polar codes constructed by Bhattacharyya parameter method in almost the entire E_b/N_0 range by 0.2 dB and 0.4 dB for $R = 0.5$ and 0.8, respectively. Since there are significantly more low reliable bit-channels in high-rate codes for redistribution in the modification process, the amount of improvement is more than that in medium-rate codes. Fig. 3.15 illustrates the changes reflected on the PMR curves after bit-swapping, corresponding to the results in Fig. 3.14. As another example, Fig. 3.16 shows the improvement of P(512, 256) constructed using DE/GA method. This code was not modified using the Alg. 2 but manually 3 bits from sub-blocks covering bits 64-128 and 128-192 were chosen based on the reliability to be swapped with the frozen bits in the two last sub-blocks. Similarly, Fig. 3.17 shows the performance gain of 0.1-0.2 dB for P(256,128) designed by DE/GA method under CA-SCL decoding.

3.5.1 Summary

In this chapter, we considered an imbalance in the fixed resources (computations and memory, which are reflected in the list size) used for list decoding in each segment of the block and the need for these resources based on the probability of elimination of the correct path in various segments. To adjust the resources to the need, we followed two tracks: 1) allocation of the list size locally based on the characteristics of the segments, 2) changing the characteristics of the segments to be adjusted with the available resources. To this end, we first analyzed the list decoding process by introducing a new parameter named path metric range and tracking the correct path within the list concerning the value of this parameter. Then, as the first track, we proposed a complexity-reduced memory-efficient list decoder in which the list size changes

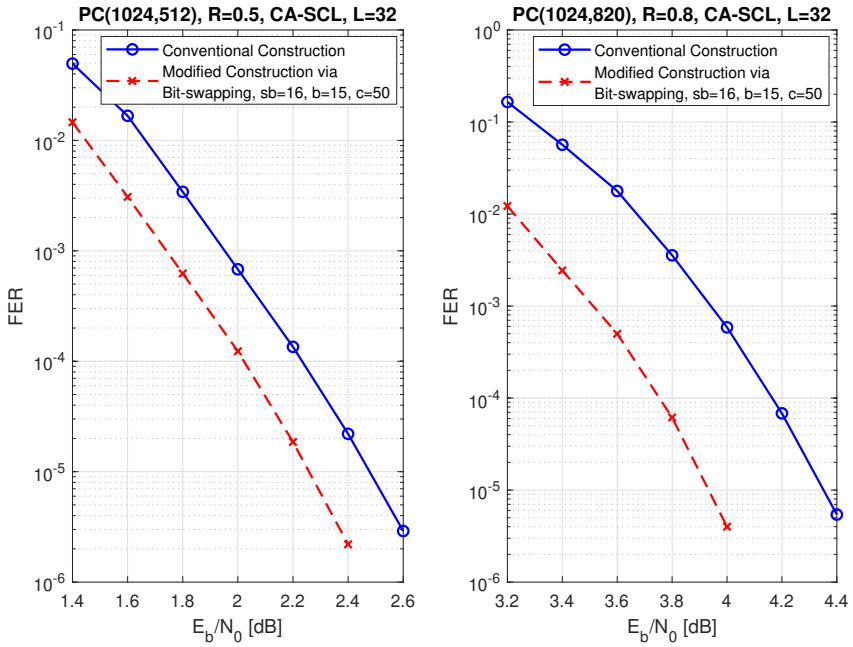

Fig. 3.14: Performance of the Modified Construction via Bit-swapping and Conventional Construction

in the subsequent partitions of the decoding tree. The results of simulations for polar codes of length 1Kb and $R = 0.5$ and 0.8 showed that the performance of the conventional list decoder in the stepped list decoding scheme is preserved. However, the stepped SCL decoding maximally can reduce the path memory by 75% and LLRs memory and partial sums memory and computational complexity by 50%. The stepped list decoding can reduce the complexity in the partitioned SCL decoding and lower the memory requirement in tree-pruning schemes. Alternatively, it was suggested to use the segments' bit error probability for allocating the local list sizes for the segments. The latter method was more effective in preserving the FER performance on almost any code, while the former can degrade the performance though it requires less memory space.

For the second track, we analyzed the recovery of the correct path and the probability of elimination of the correct path. Then, an approach to modify the code construction to adapt it to list decoding was proposed. Based on this approach, a greedy algorithm for bit-swapping was introduced to modify the construction. The numerical results showed a significant improvement in error correction performance. The improvement in high-rate codes and long codes was more significant than the other codes. This method for bit-swapping also can improve the performance of polar codes which are not optimized.

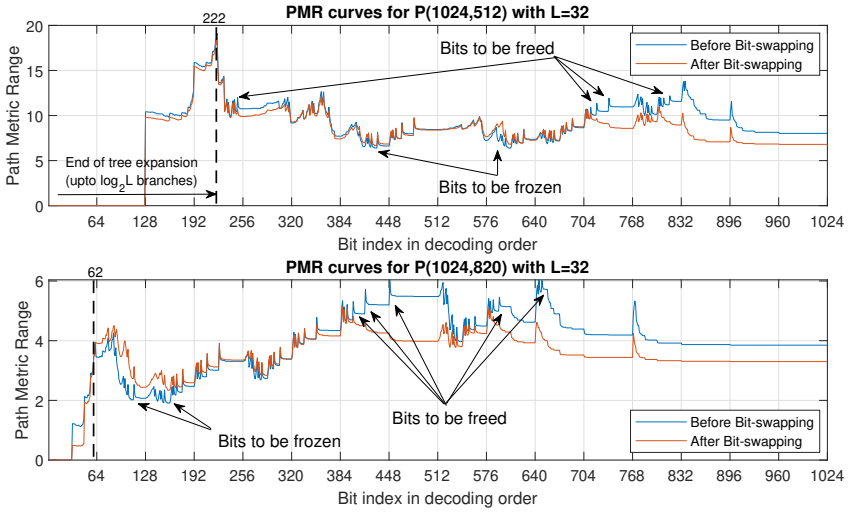

Fig. 3.15: Effect of Bit-swapping on PMR curve

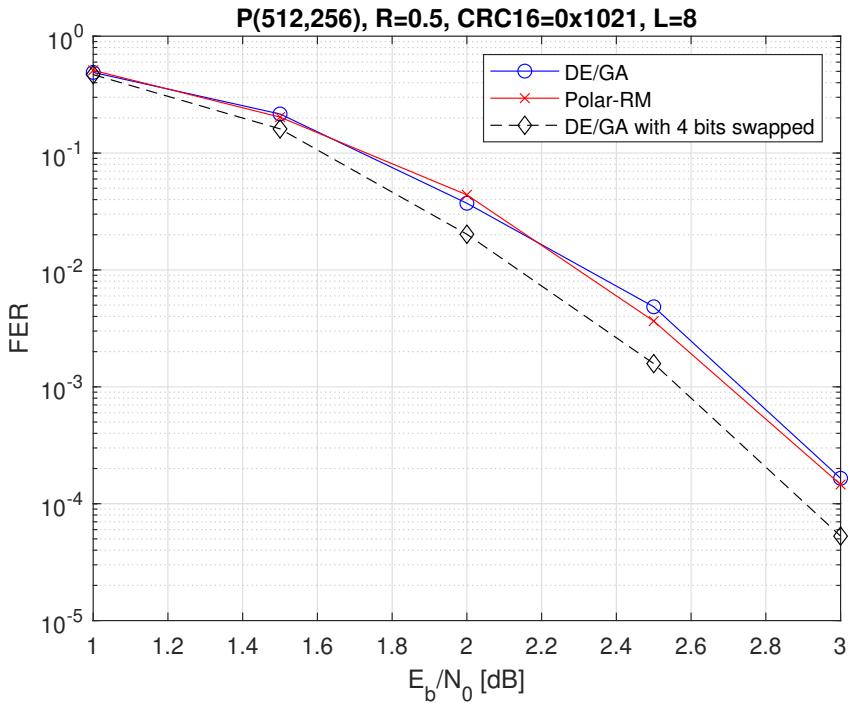

Fig. 3.16: Improved performance of the code constructed by DE/GA method

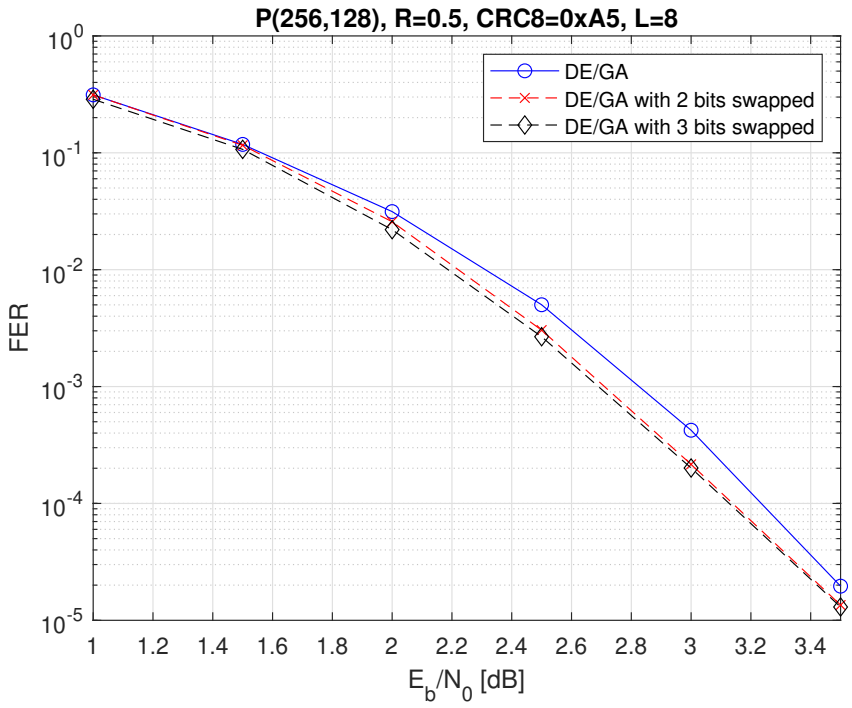

Fig. 3.17: Improved performance of the code constructed by DE/GA method

Chapter 4

Shifted-pruning Scheme for Path Recovery

"Each problem that I solved became a rule which served afterwards to solve other problems."

— René Descartes

It is known that the correct path in the SCL decoding may remain in the list after the error occurrence in the estimation of the bit values, unlike SC decoding. However, it may no longer be the most likely path at position $l = 1$ in the sorted list. In this chapter, we analyze the elimination of the correct path based on some numerical findings. Then, we look at the movement of the correct path in the sorted list and propose a metric that indicates the possibility of the elimination. This elimination can be avoided by changing the list-pruning rule. Where the probability of the elimination of the correct path is high, we shift the pruning window to keep the correct path in the list. This scheme is called *shifted-pruning scheme* and can avoid multiple errors in bit estimation (represented by accumulated penalties). Note that the shifted-pruning scheme is also known as bit-flip for SCLD [35][1] or SCL flip decoding.

4.1 Elimination of the Correct Path

4.1.1 Numerical Analysis

As we observed in (2.14), the path metric is calculated recursively by accumulating the penalties when $\hat{u}_t[l] \neq h(\lambda_0^i[l])$. Fig. 4.1 illustrates the number of penalties ($\rho = 1, 2, ..., 6$) causing the elimination of the correct path for $P(256, 128+8)$ under CRC-aided SCL decoding with 8-bit CRC generator polynomial $g(x) = x^8+x^7+x^6+x^4+x^2+1$ at different list sizes. These data are collected over 2000 block errors at FER=10^{-3}. This figure shows that as the list size increases, a larger number of penalties is required to eliminate the correct path. In simple words, as the list becomes wider, the correct path has more space to move within the sorted list and therefore it can tolerate more penalties. We will show the probability of progressive accumulation of penalties in the next section.

[1]The work in [32], independently and before [35], was submitted to ITW2019 on Apr. 10, 2019, presented on Aug. 26 in Visby, Sweden, and published on IEEE Xplore on Feb. 10, 2020

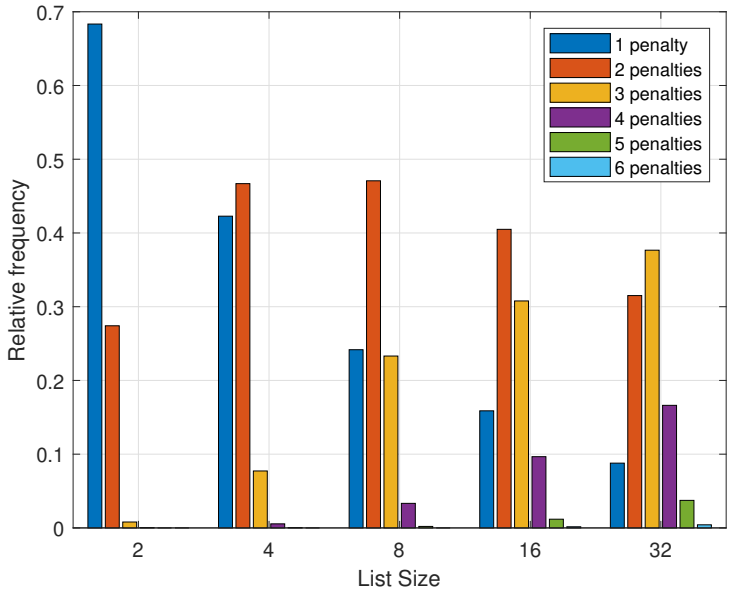

Fig. 4.1: Relative frequency of number of penalties leading to elimination of the correct path at different list sizes for $P(256, 128 + 8)$ under SCL decoding

Another interesting observation in Fig. 4.1 is that a single penalty is the cause of less than 20% of the block errors at a relatively large list size (e.g. $L = 16, 32$). Our observation shows that these single penalties occur over the bit-channels with relatively large $|\lambda_0^i|$ where PM_i of the correct path because the penalty becomes larger than the *path metric range* [37], $PMR_i = PM_L^{(i)} - PM_1^{(i)}$, and consequently the correct path is pruned. It was shown in [37] that PMR value is near or at local minimum over these bit-channels.

Fig. 4.2 shows relative frequency of the elimination of the correct path at different bit positions and Fig. 4.3 compares it with PMR for every bit position. As can be seen, the relative frequencies of eliminations in two middle 64-bit segments of $N = 256$ are shown by different colors. Segments are equilength ordered set of bits which are obtained by dividing a block code x_0^{N-1} into M sub-blocks of size 2^m bits where $M = N/2^m = 2^{n-m}$. One can observe a trend of increasing the frequency of elimination from the beginning of a sub-block, reaching a peak and then decreasing the frequency. Later in this section, we will explain the reason behind this trend based on the error probability of bit-channels, and then we will define a set of bits that contribute the most in the penalties.

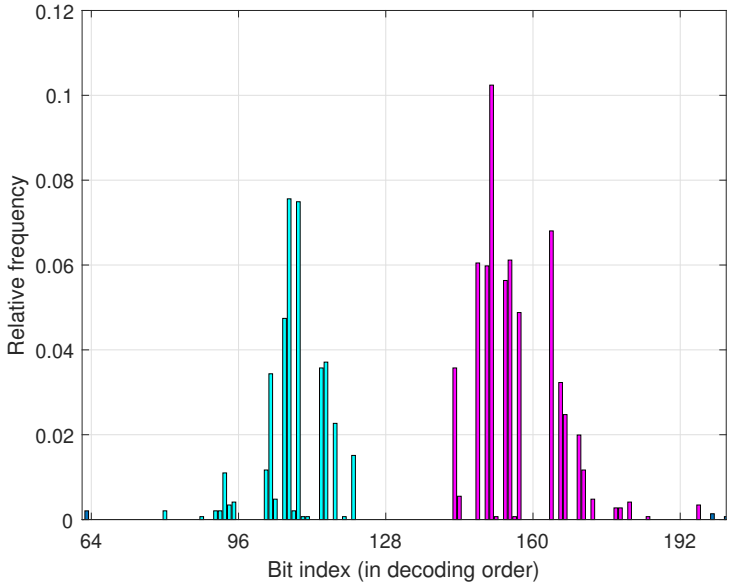

Fig. 4.2: Relative frequency of elimination caused by more than one penalty over bit-channels for $N = 256$, $R = 0.5$ and $L = 8$

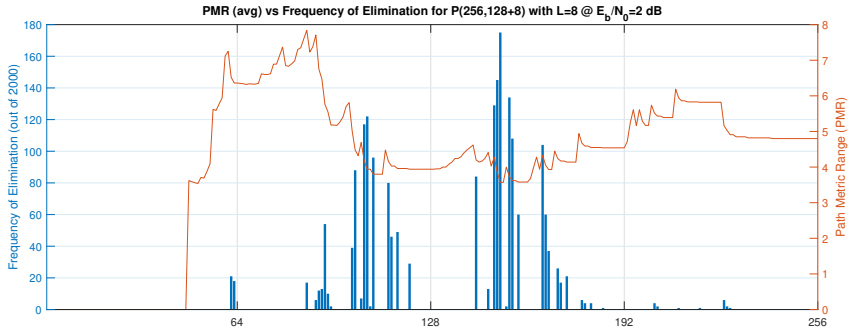

Fig. 4.3: Comparison of PMR and the relative frequency of elimination caused by more than one penalty over bit-channels for $N = 256$, $R = 0.5$ and $L = 8$

4.1.2 An Effective Metric

We are interested in a metric for the list decoding that resembles the reliability of bit-channels in a single path SC decoding. That is, a metric that indicates where the possibility of the elimination of the correct path is relatively higher than other bit coordinates. When the difference between $P(\hat{u}_i[l] = 0|\hat{u}_0^{i-1}[l], y_0^{N-1})$ and $P(\hat{u}_i[l] = 1|\hat{u}_0^{i-1}[l], y_0^{N-1})$ is small for every $l \in [1, L]$ due to small $|\lambda_0^i[l]|$ (c.f. (2.15)), the possibility of bit error at i for every $l \in [1, L]$ is relatively high. Given that we are not aware of the location of the correct path among the L paths before path splitting at each decoding step i, this could be the coordinate that the correct path can be eliminated. To indicate these coordinates, we use the probability ratio

$$\Delta_i = \frac{P(\hat{u}_0^i[2L])}{P(\hat{u}_0^i[1])} \tag{4.1}$$

Clearly, $P(\hat{u}_0^i[2L]) < P(\hat{u}_0^i[1])$. The large value of this ratio indicates that $P(\hat{u}_0^i[2L])$ and $P(\hat{u}_0^i[1])$ are close. This implies that the correct path could be among L worst path in terms of path metric due to high possibility of bit error for all the paths.

In practice, we prefer addition over division. Hence, we take the logarithm of (4.1) and as $P(\hat{u}_0^i[2L]|y_0^{N-1}) < P(\hat{u}_0^i[1]|y_0^{N-1})$, we multiply it by -1 to have a positive metric.

$$\begin{aligned}
\delta_i = -\log(\Delta_i) &= -\log\big(\frac{P(\hat{u}_0^i[2L]|y_0^{N-1})}{P(\hat{u}_0^i[1]|y_0^{N-1})}\big) \\
&= \log P(\hat{u}_0^i[1]|y_0^{N-1}) - \log P(\hat{u}_0^i[2L]|y_0^{N-1}) \\
&= -\big(-\log P(\hat{u}_0^i[1]|y_0^{N-1})\big) + \big(-\log P(\hat{u}_0^i[2L]|y_0^{N-1})\big) \\
&\overset{(2.12)}{=} PM_{2L}^{(i)} - PM_1^{(i)}
\end{aligned} \tag{4.2}$$

The metric δ_i is denoted also by PMR2 as δ_i is an expanded path metric range (PMR) covering path 1 to path $2L$.

Algorithm 3 illustrates a modified list decoder to implement the shifted-pruning scheme. In this algorithm, the length of the shift is equal to $\kappa = L/2$ as assigned in line 1. In the modified SC list decoding, if the decoding fails (line 10), the shifted-pruning scheme is performed on one of the bits in the 1st column of σ at every decoding attempt. The 2nd column of σ includes σ_i of those bits in the first run. The maximum number of attempts equals T, as shown in line 12. The shifted-pruning operation is performed in line 27 over only one bit at each decoding

attempt. Note that in this algorithm, $\mathcal{L}$ can be considered as a path list and a path is an object with properties such as the data vector, the intermediate LLR vector, partial sums vector, etc.

Now, we present a simple model for error occurrence in the list decoding. To explain the trend observed in Fig. 4.2, first we consider the elimination due to two penalties only which follows the same trend. Assuming the first non-frozen bit in a segment is indexed i, the possibility of occurring the second penalty increases by factor of $\binom{b}{1}$ as we proceed with the decoding from bit $i+b$ onward where $b = 1, 2, ...$ In this analysis, we just consider the low-reliability bits of the segment. Note that here we are assuming the second penalty occurs on the current bit and the first penalty on one of the previous bits. For example, if the second penalty occurs on bit $i+1$ where $j = 1$, the first penalty could have only occurred on bit i, while in case of the occurrence of the second penalty on bit $i+4$ where $j = 4$, the first penalty could have occurred on bit i, $i+1$, $i+2$, or $i+3$. Now, let j and ν denote the index of current bit and the number of penalties occurred up to and including bit j in a segment, respectively. In general, the probability that the ν-th penalty occurs at bit $j = 0, 1, ...$ is

$$P_j(\nu) \approx p_{e,j} \sum_{\substack{\mathcal{B}\subset\{0,\ldots,j-1\} \\ |\mathcal{B}|=\nu-1}} \prod_{r\in\mathcal{B}} p_{e,r} \prod_{s\in\mathcal{B}^c} (1-p_{e,s}) \tag{4.3}$$

where $p_{e,j}$, $p_{e,r}$, and $p_{e,s}$ are the bit error probability due to channel noise for bit j. The sum in (4.3) represent the probability of occurring $\nu-1$ penalties out of j bits which are located before j-th bit. This probability follows the Poisson binomial distribution in which we consider all the combinations of $\nu-1$ independent penalties with unequal probabilities. Although the error probability of the bit-channels are not generally independent, this model gives a good approximation. Sizing the segments properly results in the preservation of the shape of binomial distribution shown in Fig. 4.2.

Note that the probability of elimination of the correct path differs from the probability of ν accumulated penalties though they look proportional. The penalties imposed on the correct path move it within the sorted list towards $l = L$ and beyond (i.e., elimination event). The more penalties are imposed, the larger the movement within the list will be.

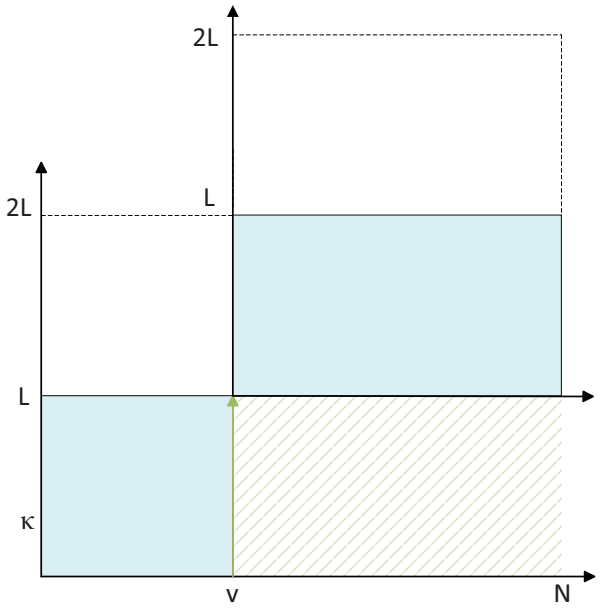

Fig. 4.4: Shifting the pruning window by L paths during list pruning operation at bit $v \in \mathcal{V}$

4.2 Shifted-pruning Scheme for Path Recovery

The correct path is pruned from the list when it has a relatively large PM due to the accumulated penalties and falls among the paths with indices $L+1$ to $2L$ in the sorted list. Thus, one can think of changing the rule for tree pruning to avoid the elimination of the correct path.

When the decoding fails in the first attempt in the list decoding of polar codes, an effective way to recover the correct path is to retain the L paths with the largest PMs (instead of L paths with the smallest PM values) over a bit (or bits in multiple decoding attempts) where there is a high probability of elimination of the correct path. This operation shown in Fig. 4.4 is named *shifted-pruning* because in the process of selection of the paths to remain in the list, the reference for the first path in the *pruning window* is shifted by $\kappa = L$, i.e. we select the path $\kappa + 1$ to path $L + \kappa$ (instead of path 1 to path L) in the PM-based sorted list of paths.

Although $\kappa = L$ seems to be a natural value to recover the correct path where it is pruned, it turns out that it suffers from a drawback. The bit index that the correct path is pruned should be identified accurately, otherwise the correct path is missed. This exact identification may require many full decoding attempts which contributes in the time complexity. However, when $\kappa < L$, e.g. $\kappa = L/2$, depending on the position of the correct path in the sorted list, it

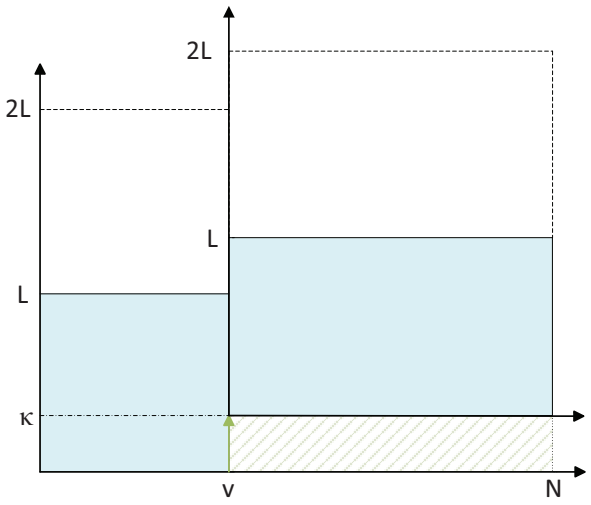

Fig. 4.5: Shifting the pruning window by κ paths during list pruning operation at bit $v \in \mathcal{V}$

can still avoid the elimination of the correct path. Fig. 4.6 illustrates an example of elimination due to three penalties at bit $i+1$, $i+2$, and $i+4$ recovered by $\kappa = L$ (right) and $\kappa = L/2$ (left). As Fig. 4.6 shows, the positions $L+1$ to $2L-L/2$ cover nearly 90% of the pruned paths. These statistics were collected from the same experiment as the one discussed in Section 4.1. On the other hand, if instead of the exact position of the elimination, the pruning window is shifted over a bit near that, by choosing $\kappa = L/2$ the correct paths located at positions $L/2+1$ to L may still remain in the list. Hence, the problem of exact identification of the elimination position can be relaxed by $\kappa = L/2$.

As a general scheme for shifting the pruning window, κ can vary in interval $0 \leq \kappa_i \leq L$ on different bit indices, $0 \leq i \leq N-1$. Fig. 4.5 shows shifting the pruning window by κ paths in general over bit v. Obviously, $\kappa_i = 0$ for $i \in \mathcal{A}^c$ and high-reliability bit indices. Finding a practical method to determine κ_i is an open problem. Studying the movement of the correct path using Monte Carlo method may provide a set of patterns for $\{k_i\}$. A good pattern is the one that can reduce the complexity substantially at low degradation cost. In section 4.4, we will show the results of a simple pattern for the variable shifting scheme.

4.3 Toward Nested Shifted-pruning Scheme

Elimination of the correct path may not be prevented by just one-time shifting throughout decoding a codeword. Let us first look at some statistics, and then we refine our scope of the

Algorithm 3: List Decoder with Shifted-pruning

input : the received vector y_1^N, non-frozen set $\mathcal{A}$, L
output: the recovered message bits $\hat{u}_1^N$

$[T, t, \kappa, crcPass] \leftarrow [20, 0, L/2, false]$
$\delta[1 : K + r, 1 : 2] \leftarrow \{0\}$
do
  if *t=0* **then**
    $\hat{u}_1^N[1..L], \delta \leftarrow$ SCLD(y_1^N,$\mathcal{A}$, L, t, κ)
    Sort δ with respect to 2nd column
  else
    $\hat{u}_1^N[1..L] \leftarrow$ SCLD(y_1^N,$\mathcal{A}$, L, t, κ)
  for $l \leftarrow 1$ **to** L **do**
    if CRC($\hat{u}_1^N[l]$) = *success* **then**
      $\hat{u}_1^N \leftarrow \hat{u}_1^N[l]$
      crcPass = *true*
  if crcPass = *false* **then**
    $t \leftarrow t + 1$
while $t \leq T$ AND crcPass = *false*
return $\hat{u}_1^N$
subroutine SCLD(y_1^N,$\mathcal{A}$, L, t, κ)**:**
  $j \leftarrow 1$; $\mathcal{L} \leftarrow \{1\}$ // a single path in the list
  for $i \leftarrow 1$ **to** N **do**
    Perform step i of SCL Decoding:
    - Prune paths when $i \in \mathcal{A}$ AND $|\mathcal{L}| > L$:
    if $t \not> 0$ **then**
      $\mathcal{L} \leftarrow \{1, ..., L\}$
      $\delta[j, 1 : 2] \leftarrow [i, PM_{2L}^{(i)} - PM_1^{(i)}]$
      $j \leftarrow j + 1$
    else
      if $\delta[t, 1] \neq i$ **then**
        $\mathcal{L} \leftarrow \{1, ..., L\}$
      else
        $\mathcal{L} \leftarrow \{\kappa + 1, ..., L + \kappa\}$
  return $\hat{u}_1^N[1..L], \delta$

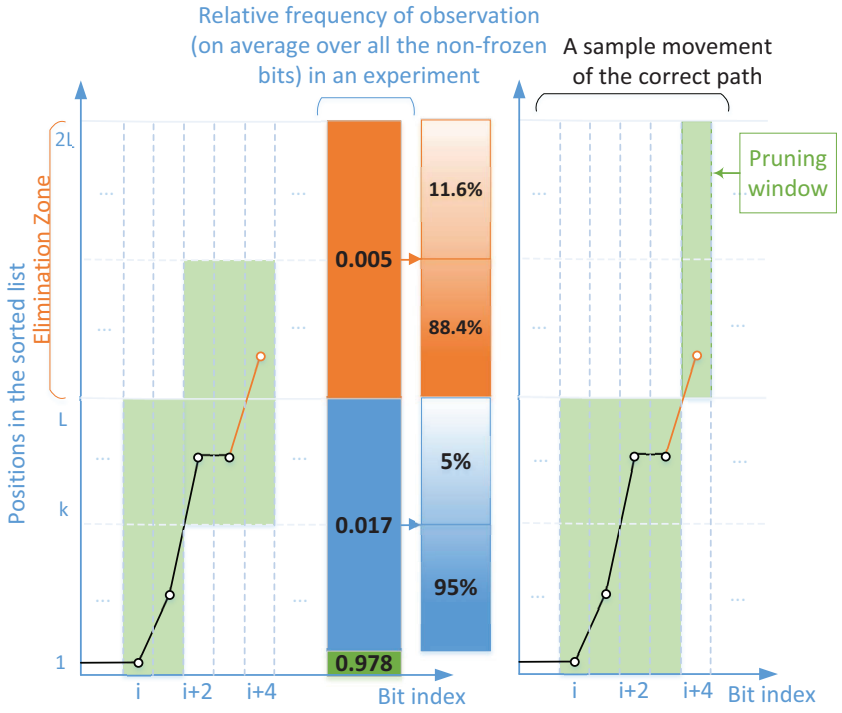

Fig. 4.6: $L/2$-shift ($\kappa = L/2$) vs L-shift during pruning operation.

investigation and propose a scheme. Suppose there is an oracle in the list decoder that can avoid the elimination at any bit indices by shifting the pruning window. Fig. 4.7 shows a relative frequency of need to one shift comparing with multiple shifts of the pruning window to recover the correct path throughout the decoding process of one codeword. As this figure shows, at high SNR regimes, the number of multiple shifting is relatively small. Now, let us observe the performance improvement due to multiple shifts. Fig. 4.8 shows the error correction gain obtained from multiple shifts is significantly higher than one-time shifting. The notation "SP" in the figure is used for oracle-assisted shifted-pruning and x in "SPx" indicates the number of shifts applied throughout one decoding iteration to avoid the elimination of the correct path. Observe that SP3 achieves the performance of maximum likelihood (ML) decoding. As can be seen in Fig. 4.8, if we use the full critical set in additional decoding attempts for the case SP1, the FER performance is expected to be as good as the oracle-assisted performance. Note that we use 16-bit CRC to reduce the probability of the false detection of the correct path in this example significantly. As the other dashed curves show, employment of nested shifts could improve the performance further. However, finding the right combination of the bit indices for shifting requires a massive search.

Given that the error prevention by more than two shifts occurs rarely at the high SNR

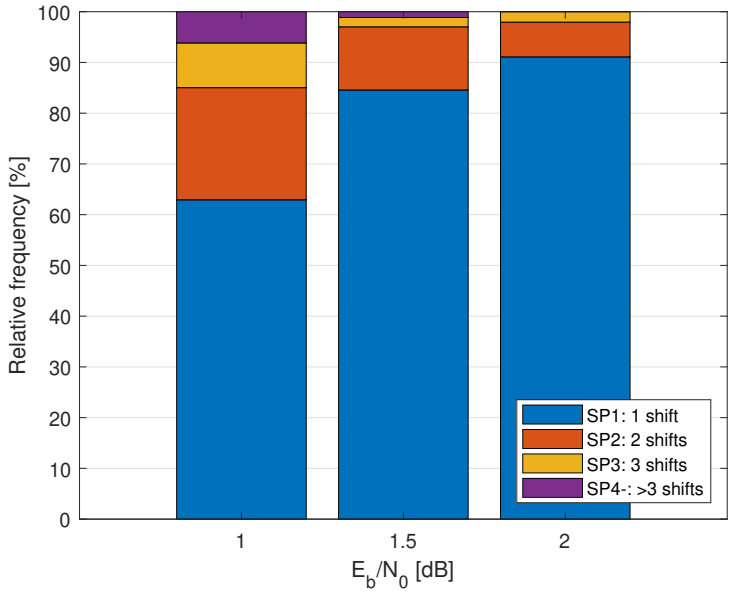

Fig. 4.7: Relative frequency of the number of oracle-assisted recoveries of correct path throughout decoding by one or multiple shifting of the pruning window (SPx,x=1,2,...) in 30000 codewords of P(512,256+12) under CRC12-aided list decoding with list size $L = 8$.

regimes, due to difficulty and need to a computationally-expensive search for finding the right combination of positions for more than two shifts, we only focus on double shifting in this section. Fig. 4.9 illustrates a sketch of double shifting.

Two shifts can occur in the vicinity, i.e., $v_2 - v_1 \leq d$ (d is the vicinity parameter) or far from each other in two different segments. Fig. 4.10 shows the proportion of occurrence of these two cases at different SNR regimes.

We cover multiple shifting occurring in different segments in Section 4.3.1 and the pair-shifting in the vicinity in Section 4.3.2.

4.3.1 Segmented Shifted-pruning

In segmented/partitioned list decoding [23,24], we have to use multiple short CRCs to detect the correct path in each segment. Unfortunately, this may cost us a code rate when the sum of these short CRCs is larger than a single CRC for the whole codeword and consequently this brings a slight performance degradation. Note that the probability of undetected error (false detection of the correct path) by a short CRC is high [30]. Considering that we need to run additional decoding in the shifted-pruning scheme, the probability that an incorrect path is

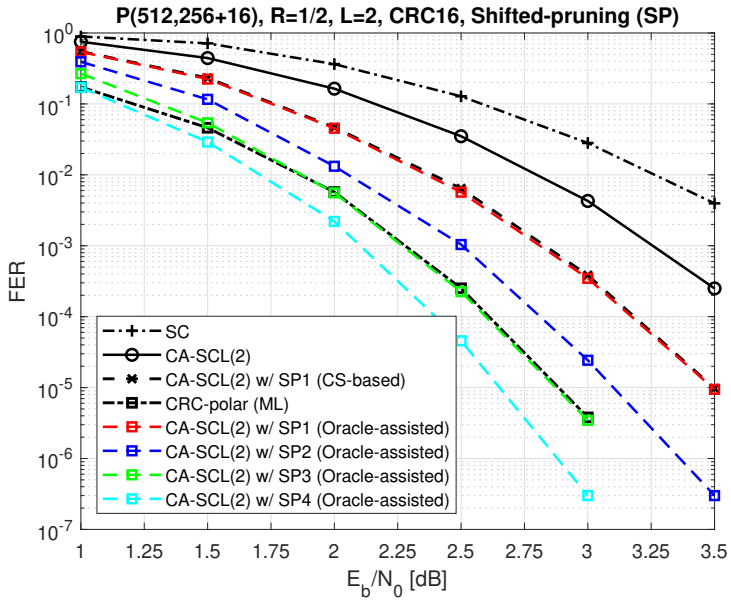

Fig. 4.8: Performance oracle-assisted list decoding with multiple shifts

detected by CRC as the correct path increases significantly due to increase in the number of iterations. Thus, the bad news is that we cannot expect to obtain the performance of nested shifts as shown in Fig. 4.8. However, there is a good news which is a significant reduction in the computational complexity by using segmented list decoding. The reduction in complexity comes from the fact that we do not need to apply shifting on the whole code-block. For example, in decoding P(512,256+2×8) with two segments for which 8-bit CRCs are used, if the elimination occurs in the first segment only, we just apply the additional decoding iterations on that segment and once the correct path was recovered, the second segment may not require additional decoding attempts. Therefore, the additional computational complexity introduced by additional attempts halves for this codeword. Note that the failures through detecting incorrect paths due to employing short CRCs are traded off by successes due to multiple shifts, one shift in each segment, and overall, the performance improvement remains almost the same at small list size.

For computational complexity comparisons, since the block-length N is fixed, we drop $N \cdot \log N$ from the computational complexity $O(LN \log N)$, hence, we use the average list size L as a measure of complexity of the shifted-pruning scheme. Now, let denote the total iterations and total decoded messages during decoding, and number of segments by t, c, and s, respectively, the computational complexity of non-segmented and segmented decoding schemes are computed by $O((\frac{t-c}{c}+1)\cdot L) = O(\frac{t}{c}\cdot L)$ and $O((\frac{t-c}{s\cdot c}+1)\cdot L)$. Note that in the segmented decoding, t

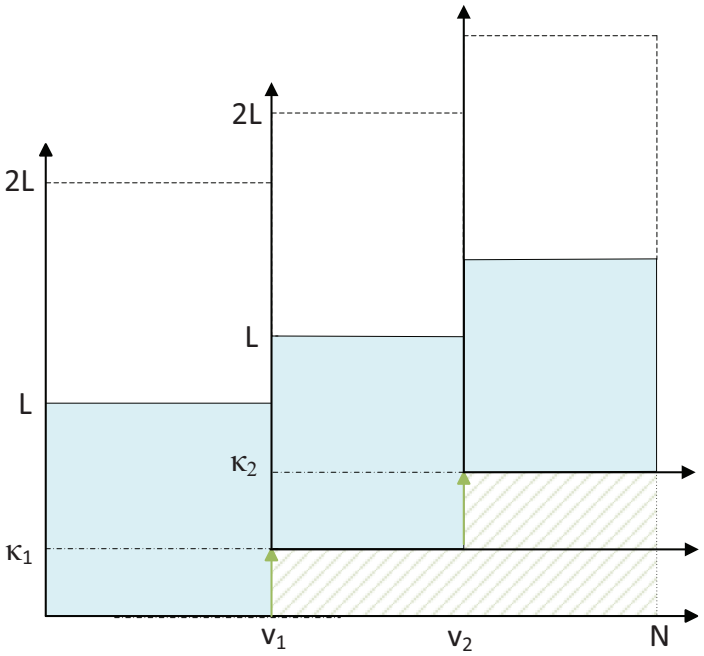

Fig. 4.9: Nested shifting by k paths at bit $v_1, v_2 \in \mathcal{V}$ during pruning operation

refers to the iteration in each segment; hence the total iterations are sum of iterations at all segments. Furthermore, $t - c$ indicates the additional iterations performed for shifted-pruning to correct the error.

4.3.2 Double-shifting: Ordered-pairs

As the numerical analysis showed, in the presence of severe noise, when the correct path is recovered by shifting the pruning window, it might be eliminated at the next consecutive low-reliability bit or a bit in the vicinity due to bit-channels correlation. Here, we propose a scheme to avoid the elimination by shifting at two close positions within a segment. Obviously, first we try to recover the correct path by a single shift. As Fig. 4.11 shows, if the initial T attempts fail, multiple shifting may recover the correct path. We do not always try the pair-shifting but we perform it when the chance of recovery by pair-shifting is high. We use the following criteria: If among top 5 out of T sorted positions with respect to the metric, the majority of them belong to a specific segment. This could be a sign that we are facing a low-reliability segment. Otherwise, it is not suggested to try multiple shifting as it is quite costly in terms of time and computational complexity.

After detecting a low-reliability segment, we sort the bit indices belong to this segment

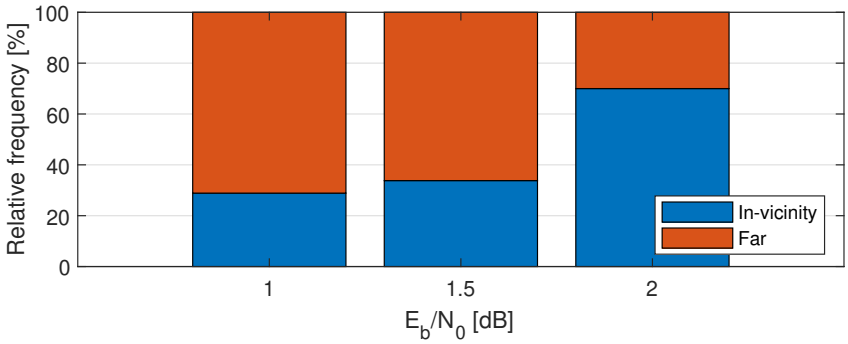

Fig. 4.10: Relative frequency of the number of 2 shifts at positions in vicinity ($v_2 - v_1 \leq 10$) and at positions in farther distance ($v_2 - v_1 > 10$).

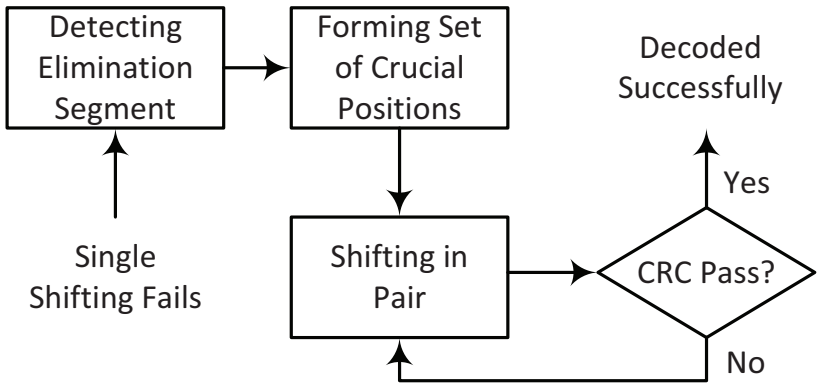

Fig. 4.11: The procedure of double-shifting when the single shift at T positions fails.

appeared among the T positions in descending order. Let us call these bit indices as critical bits of the segment. The pairs are selected as shown in Fig. 4.12. That is, the first sorted position is paired with the second position. If the decoding fails, the first position is paired with the third position. After trying all the combinations of the first position with with the three positions in the vicinity, we try all the combinations of the second position paired with each of the three positions before that. Let us denote the number of positions in the vicinity for pairing as π. This parameter should not be large as the possibility of elimination for the second time is higher in the close vicinity due to a stronger correlation between the bit-channels in the vicinity. Needless to mention that a large π may not increase the chance of recovery but it adds a significant number of failed iterations. One can choose $\pi = 2$ instead of 3 in the vicinity to reduce the complexity because the second time elimination is predominantly occurs at a close position. Note that the vicinity parameter d is different with π because in between v_1 and v_2, there might be frozen bit-channels or non-frozen reliable bit-channels.

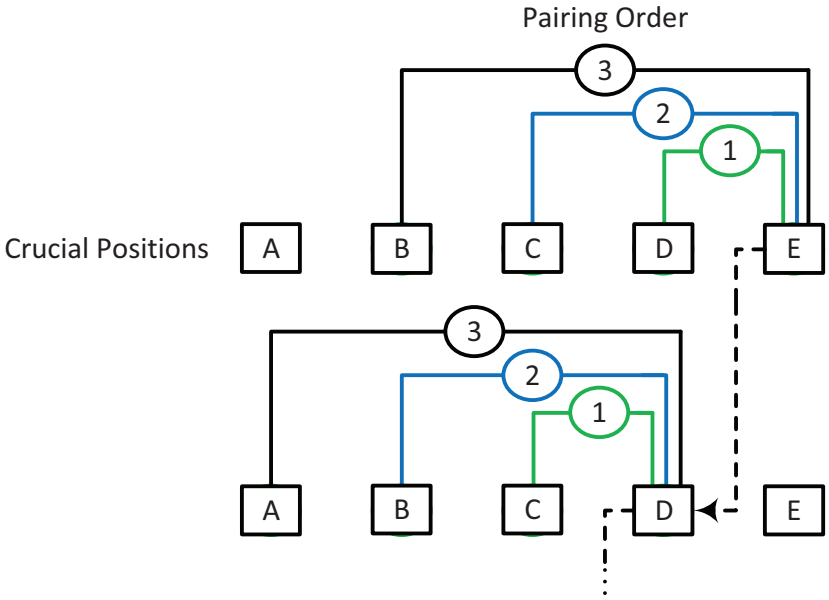

Fig. 4.12: Position pairing scheme for double shifting. Positions A,B,C,D,E belong to one segment where A<B<C<D<E. The pairing starts from E: (E,D), (E,C), (E,B), then D: (D,C), (D,B), and son on.

4.4 Numerical results

To evaluate the performance of the proposed metric δ_i, a.k.a PMR2, polar codes of length $N = 2^9$ and 2^{10} with the code rate of $R = K/N = 0.5$ are constructed using density evolution under Gaussian approximation [9] while optimized for high SNRs with design-SNRs of 2 dB and 3 dB, respectively. The LLR-based CRC-aided (CA) SCL decoder is used with 12-bit and 16-bit CRC generator polynomial of 0xC06 and 0x1012. Note that we use the same polynomials for the rest of the simulations as well. Fig. 4.13 compares the FER performance of PMR2 with the performance of the conventional CRC-aided (CA) SCL decoding. The gain is about 0.2 dB at FER range of 10^{-3} for both. As it was discussed in section 4.1.2, the metric in [35] does not provide a good performance for medium code lengths.

Fig. 4.14 and 4.15 show the FER performance and the relative complexity of the segmented shifted pruning using two segments protected by two 8-bit CRCs with the generator polynomial 0xA5. The code P(512,256+16) was constructed by DE/GA method and design-SNR=4. According to this figure, a significant reduction in the complexity is obtained at the cost of o.1 dB degradation in the performance. Furthermore, the performance for list size $L = 2$ overlaps with the non-segmented decoding except at high SNRs, while there is a gap of about 0.1 dB

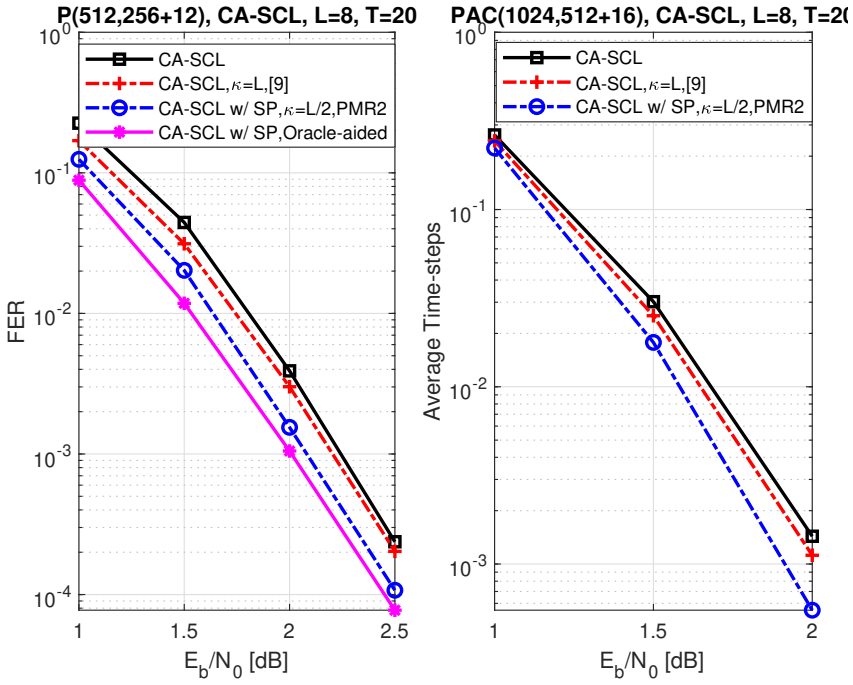

Fig. 4.13: FER performance for P(512,256+12) and P(1024,512+16)

for $L = 8$. This gap appears due to the high probability of undetected error when using short CRCs, particularly when the number of candidates to check at large list size is significantly more than $L = 2$. On the other hand, this slight degradation can be traded with a significant reduction in the computational complexity as shown in Fig. 4.15.

Fig. 4.16 illustrates the FER performance gain and the complexity growth of double-shifting under the ordered-pair scheme in comparison with single shifting. It seems that double shifting is approaching the performance of the oracle-assisted with a single shift. We observe in [32] that the oracle-assisted gain can be obtained by T equal to the size of the critical set [18] which is large and imposes a huge complexity (see [32]) while double shifting can approach that performance with less complexity.

4.5 Summary

In this chapter, we analyzed the elimination of the correct path in the list decoding process. Then, the shifted-pruning scheme was proposed to significantly reduce the probability of elimination of the correct path in additional decoding attempts when the decoding fails. Then, we suggested a new metric for shifted pruning scheme that significantly improves the performance

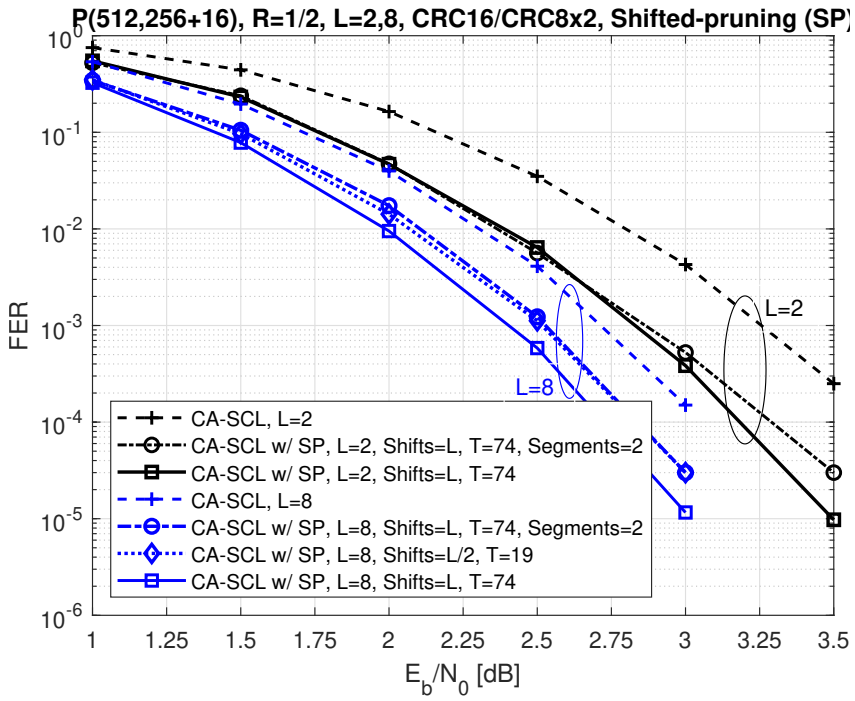

Fig. 4.14: Error correction performance under segmented CA-SCL decoding with shifted pruning (SP) and CA-SCL decoding with constrained shifted pruning (SP)

of the shifted pruning scheme compared with the available metrics. In contrast, this metric is computationally simple and works with LLR-based SCL decoding. Then, a double-shifting scheme is suggested to prevent a portion of errors that require two shifts by an oracle. This scheme improves the FER performance at a reasonable complexity cost.

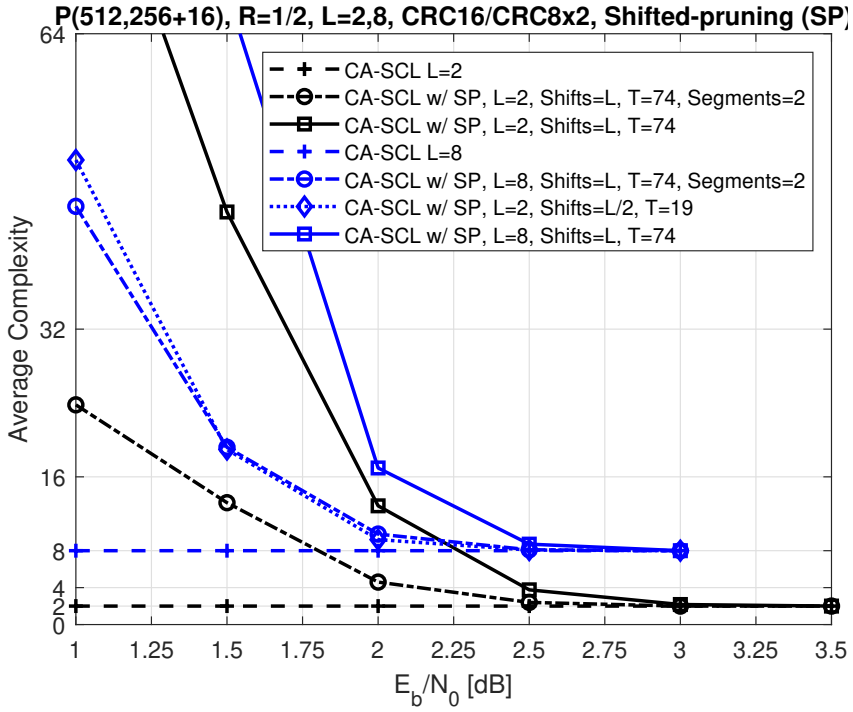

Fig. 4.15: Average complexity under segmented CA-SCL decoding with shifted pruning (SP) and CA-SCL decoding with constrained shifted pruning (SP)

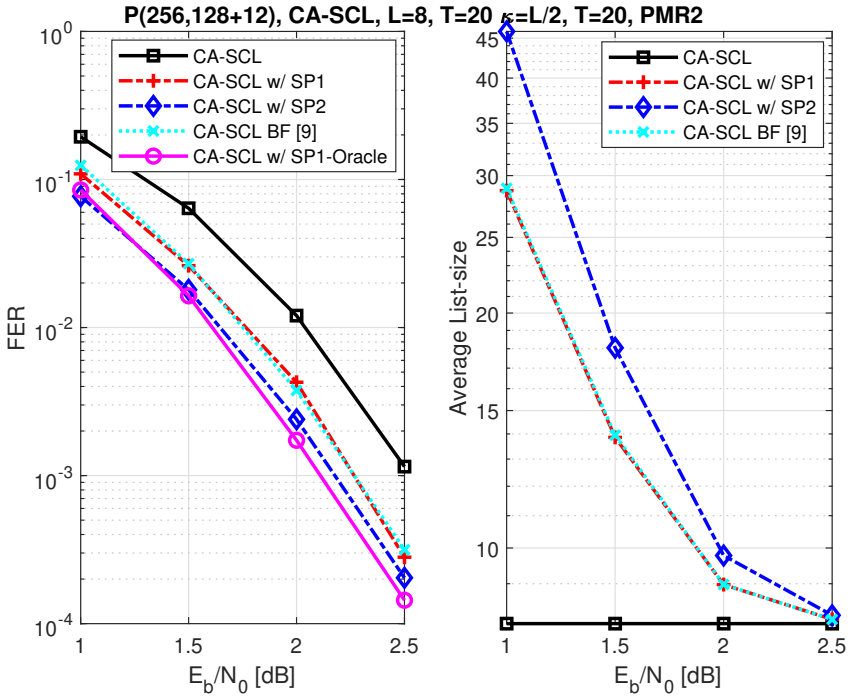

Fig. 4.16: Comparison of FER performance and the computational complexity (equivalent to average list size considering additional attempts) of double-shifting and single shifting

Chapter 5

Efficient Partial Rewind of SC Algorithm

> *"If I had an hour to solve a problem I'd spend 55 minutes thinking about the problem and 5 minutes thinking about solutions."*
>
> — Albert Einstein

Decoding of polar codes and their variants requires passing the channel log-likelihood ratios (LLRs) through the factor graph shown in 2.2. The evolved information at the output of the factor graph is used to make a hard decision or to calculate a metric in the SC-based decoders. The evolved LLR, a.k.a decision LLR, is obtained for each bit-channel successively. To calculate each decision LLR, we need to access the intermediate information on the factor graph. There are two ways to access them:

- **Naive approach**: We can store all the $N \cdot \log_2 N$ intermediate values, including LLRs and partial sums on the factor graph. This approach is acceptable for short codes under SC decoding or Fano decoding. However, as the code gets longer, in particular under list decoding or stack decoding, this approach will be expensive in terms of memory requirement.
- **Efficient approach**: It was observed in [66] that for calculating every decision LLR, we need at most $N - 1$ intermediate LLRs (excluding channel LLRs) and partial sums. We will discuss this approach in detail in Section 5.1.

Some decoding schemes require additional decoding attempts when the decoding fails in the first attempt. These schemes are as follows:

- **SC-flip decoding**: In this scheme, when SC decoding fails, the decoding is repeated from scratch, while in the additional attempts, a single or multiple bits are flipped throughout the SC decoding process [25].
- **Shifted-pruning**: This scheme was discussed in Chapter 4. Note that a special case of this scheme is also called SCL flip scheme or path-flipping for SCL decoding.
- **Fano decoding**: In this scheme, the decoder may have a back-tracking or backward movement to explore the other paths on the decoding tree. Unlike the first two schemes

where the decoding of a codeword is completed, and then the additional decoding is repeated from the first bit, in the Fano algorithm, the backward movement occurs somewhere between the first bit and the last bit. This scheme is discussed in Chapter 7.

In this chapter, we propose an approach to efficiently rewind the SC algorithm from the last bit to the position that we need to flip the value of a bit in SC-flip decoding or to shift the pruning window in the shifted-pruning scheme [67]. We will discuss this method for Fano decoding in Chapter 7.

5.1 Efficient Updating of Intermediate Information

5.1.1 Intermediate LLRs

The factor graph shown in Fig. 5.1 has $N \log_2 N$ nodes however, as it was shown in [66], it is sufficient to update/access at most $N-1$ intermediate LLRs (including decision LLR) out of $N \log_2 N$ LLRs for decoding any bit $u_i, 0 \leq i \leq N-1$. Fig. 5.1 illustrates the LLRs associated with decoding bit u_3 in a tree form on the factor graph. As can be seen, there are 2^s LLRs in stage s for $s = 0, ..., n$. Hence, according to the geometric series, we need a total memory space of

$$\sum_{s=0}^{n} 2^s = 2^{n+1} - 1 = 2N - 1 \tag{5.1}$$

Suppose u_i is the bit that was just decoded and $\text{bin}(i) = i_{n-1}...i_0$ is the binary representation of index i where the least significant bit is indexed 0 and most significant bit is indexed $n-1$. . The stages are updated from right to left (where $s = 0$). The first stage to be updated is obtained by *finding the first one, ffo*, or the position of the least significant bit set to one as

$$\eta(i) = \text{ffo}(i_{n-1}...i_0) = \begin{cases} \min\limits_{i_t=1}(t) & i > 0 \\ n-1 & i = 0 \end{cases}. \tag{5.2}$$

Note that in the semi-parallel hardware architecture, since the LLRs are stored in blocks, memory usage is inefficient such that there will be some unused memory space. In fact, the reduction in the number of processing elements is traded with slightly higher clock cycles and larger memory space (for more information, see [66]).

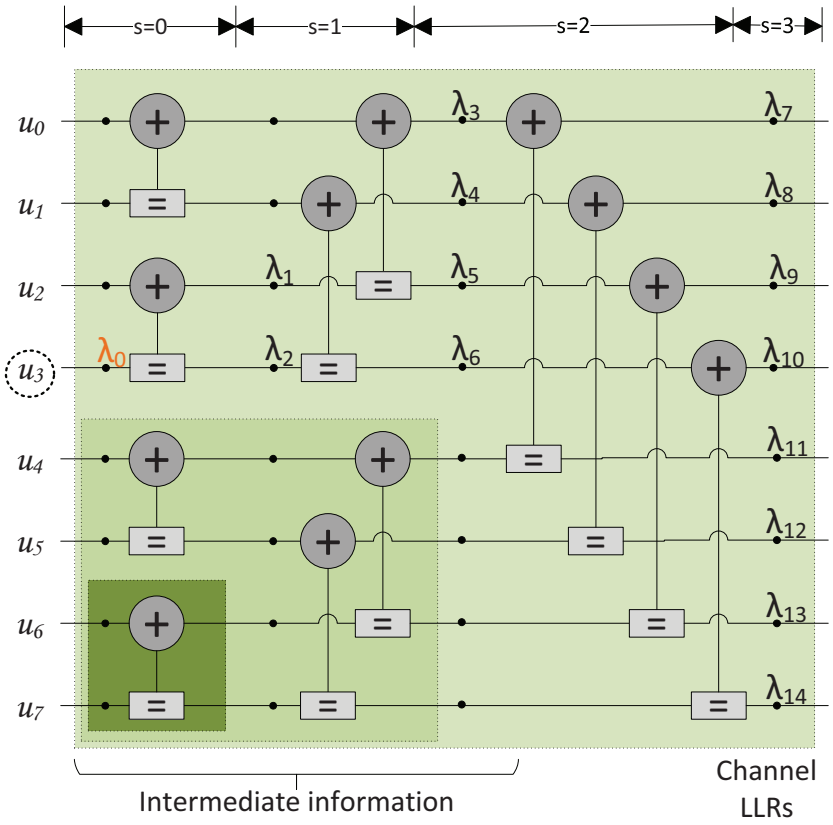

Fig. 5.1: An illustrative example for updating LLRs for decoding bit u_3. $\lambda_0^3 = \lambda_0$ is computed based on λ_1, λ_2 and $\beta_0^2 = \beta_0 = \hat{u}_2$ (see Fig. 5.2).

5.1.2 Partial Sums

The Partial sums are the other set of intermediate information needed for the SC process. It turns out that we need the same memory space for the partial sums as well, i.e., at most $N-1$ memory elements. It was observed in [66] that we need to store 2^s bits corresponding to 2^s nodes of type **g** at stage s, which are waiting to be summed with the next decoded bit. Here, let us define an operator that indicates the last stage to be updated. The last stage that its partial sums to be updated is obtained by *finding the first zero, ffz*, or the position of the least significant bit set to zero as

$$\psi(i) = \text{ffz}(i_{n-1}...i_0) = \min_{i_t=0}(t). \tag{5.3}$$

It turns out that this is the only stage that consists of **g** nodes in the process of updating LLRs from stage $s = \text{ffo}(\text{bin}(i))$ up to $s = 0$. Clearly, after decoding the last bit where there is no

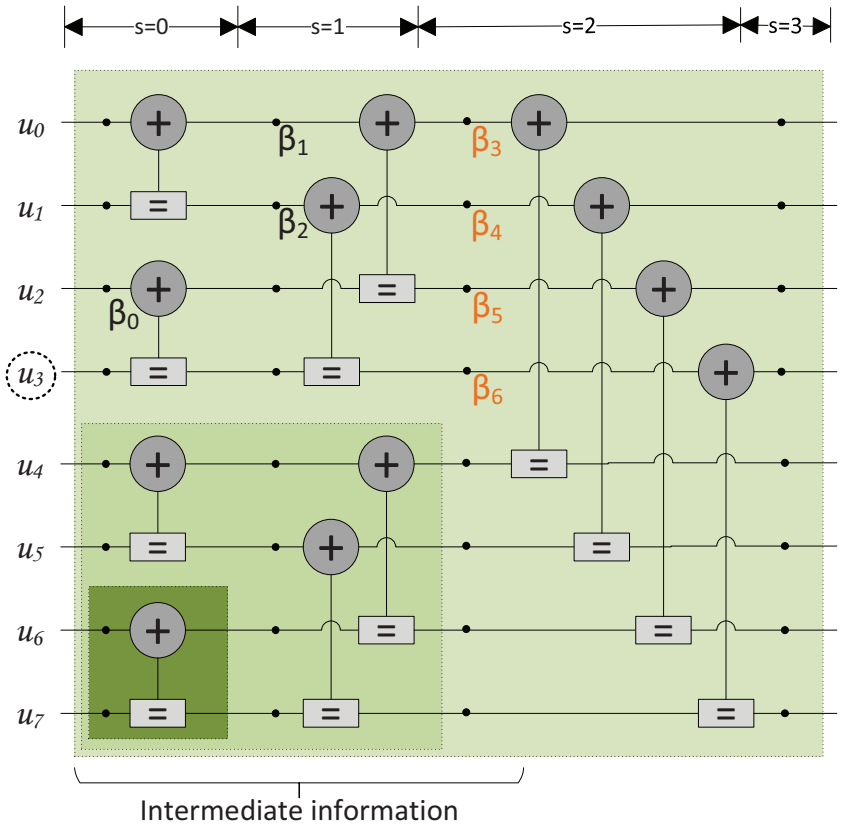

Fig. 5.2: An illustrative example for updating partial sums of stage $s = 2$ after decoding u_3.

zero in the binary representation of the index, $\text{bin}(N-1) = 11...1$, there is no need to update partial sums.

Fig. 5.2 shows $N - 1 = 8 - 1 = 7$ partial sums (β_0 to β_6) associated to u_3. The β values in orange are updated after decoding bit u_3 as $\beta_6 = u_3$, $\beta_5 = u_3 \oplus \beta_0$, $\beta_4 = u_3 \oplus \beta_2$, and $\beta_3 = u_3 \oplus \beta_1 \oplus \beta_3$.

There are methods proposed in [68, 69] for hardware implementation that require slightly less memory space for updating partial sums.

You may notice that for $i \in [0, N-2]$, we have

$$\psi(i) = \eta(i+1) \tag{5.4}$$

That is the reason why at any bit $i \in [1, N-1]$, the stage $\eta(i)$ where its LLRs needs to be updated consists of only **g** nodes. Therefore, after decoding bit $i-1$, the partial sums of this stage are updated to be used for the **g** nodes at stage $\eta(i)$.

5.2 Properties of SC Process

We discover some properties of the SC algorithm that can help us to rewind the process efficiently. The goal is not storing all the $N \log_2 N$ values for LLRs and partial sums or restarting the SC process from bit 0 in the SC-based decoding when a re-decoding attempt is required. First, let us define an operator that helps us in the upcoming analysis.

Definition 5.1 *The operator $\phi(j)$ finds the last zero, flz, or the position of the most significant bit set to zero in the binary representation of $j = (j_{n-1}...j_0)_2$ indexed in reverse order as*

$$\phi(j) = flz(j_{n-1}...j_0) = \begin{cases} n-1-\max\limits_{j_t=0}(t) & j < 2^n - 1, \\ n-1 & j = 2^n - 1 \end{cases} \tag{5.5}$$

for every $t \in [0, n-1]$. We denote the output of the operator $\phi(j)$ by parameter p.

Note that since the indexing is in the opposite direction when the most significant bit is set to zero, i.e., $j_{n-1} = 0$, then we get $p = 0$, and when the only zero bit is $j_0 = 0$ or there is no 0-value bit, then $p = n - 1$.

Definition 5.2 (Set $\mathcal{Z}_p$) *We group the bit indices $j \in [0, 1, ..., 2^n - 1]$ based on the identical $p = \phi(j)$ into n sets denoted by set $\mathcal{Z}_p$ with order $p = 0, 1, ..., n-1$, or*

$$\mathcal{Z}_p = \{j \in [0, 2^n - 1] : \phi(j) = p\} \tag{5.6}$$

Example 5.1 *For $n = 3$, we can group the indices 0 to 7 into the following sets: $\mathcal{Z}_0 = \{0, 1, 2, 3\}$, $\mathcal{Z}_1 = \{4, 5\}$, and $\mathcal{Z}_2 = \{6, 7\}$.*

Remark 5.1 *The distribution of non-frozen indices in set $\mathcal{A}$ among sets $\mathcal{Z}_p, p \in [0, n-1]$ depends on the code rate. Observe that as the code rate reduces, a fewer non-frozen indices will exist in low order $\mathcal{Z}_p$, i.e., $\mathcal{Z}_p$ with small p.*

Lemma 5.1 (Properties of $\mathcal{Z}_p$) *For any $n > 0$ and $p \in [0, n-1]$, set $\mathcal{Z}_p$ has the following properties:*

a. The boundaries of set $\mathcal{Z}_p$ are

$$\mathcal{Z}_p = \begin{cases} [2^n - 2^{n-p}, 2^n - 2^{n-(p+1)} - 1] & 0 \leq p < n-1, \\ [2^n - 2^{n-p}, 2^n - 1] & p = n-1 \end{cases} \tag{5.7}$$

b. The size of set $\mathcal{Z}_p$ is

$$|\mathcal{Z}_p| = \begin{cases} 2^{n-p-1} & 0 \leq p < n-1, \\ 2 & p = n-1 \end{cases} \tag{5.8}$$

c. The smallest element in set $\mathcal{Z}_p$ is

$$z_p = \min(\mathcal{Z}_p) = \sum_{x=n-p}^{n-1} 2^x = 2^n - 2^{n-p} \tag{5.9}$$

Proof:

a. Let us introduce a special notation for the binary representation of a positive integer with length n first. Given $\{0,1\}^x$ indicates a mixed string of 0 and 1, and $\{b\}^x, b \in \{0,1\}$ denotes a uniform string of either 0 or 1, both with length x. In set $\mathcal{Z}_p, p < n-1$, observe that the elements are in the form of $\{1\}^p + \{0\} + \{0,1\}^{n-(p+1)}$ where the operator '+' is used for concatenation and $\{1\}^p$ is/are the most significant bits. Then, the smallest element of set $\mathcal{Z}_p$ in binary is

$$\{1\}^p + \{0\} + \{0\}^{n-(p+1)} = \{1\}^p + \{0\}^{n-p}$$

which is equivalent to

$$\sum_{x=n-p}^{n-1} 2^x = 2^n - 2^{n-p}$$

in decimal. Similarly, one can see that the largest element in set $\mathcal{Z}_p, p < n-1$ is

$$\{1\}^p + \{0\} + \{1\}^{n-(p+2)} = \left(\{1\}^n\right)_2 - \left(\{1\} + \{0\}^{n-(p+2)}\right)_2$$

which is equivalent to $(2^n - 1) - 2^{n-(p+1)}$ in decimal.

Note that the largest element in set $\mathcal{Z}_p, p = n-1$ is $\left(\{1\}^n\right)_2 = 2^{n-1}$ while the smallest element follows the relationship discussed above.

b. Given the interval $[\min(\mathcal{Z}_p), \max(\mathcal{Z}_p)]$ in part a of this lemma, we can find the size of set $\mathcal{Z}_p$ by $\max(\mathcal{Z}_p) - \min(\mathcal{Z}_p) + 1$.

c. It follows from part a of this lemma that the lower bound of the values in set $\mathcal{Z}_p$ in binary is $\{1\}^p + \{0\}^{n-p}$ which is equivalent to $\sum_{x=n-p}^{n-1} 2^x = 2^n - 2^{n-p}$ in decimal.

■

Example 5.2 *For $n = 4$, we have z_p for $p = 0, 1, ..., n-1$ as*

$$z_0 = (0000)_2 = 0, z_1 = (1000)_2 = 8, z_2 = (1100)_2 = 12, \text{ and } z_3 = (1110)_2 = 14$$

or based on the lower bound of $\mathcal{Z}_p$ in Lemma 5.1 as

$$z_0 = 2^n - 2^{n-0} = 16 - 2^4 = 0, z_1 = 16 - 2^3 = 8, z_2 = 16 - 2^2 = 12, \text{ and } z_3 = 16 - 2 = 14$$

Let us find the deepest updated stage while decoding any bit i within set $\mathcal{Z}_p$ in the following lemma.

Lemma 5.2 *For any $i \in \mathcal{Z}_p, p \in [0, n-1]$, and $z_p = \min(\mathcal{Z}_p)$ we have*

$$\max_{i \in \mathcal{Z}_p}(\eta(i)) = \eta(z_p) \tag{5.10}$$

Proof: Let us recall the notation $\{1\}^p + \{0\} + \{0,1\}^{n-(p+1)}$ for $i \in \mathcal{Z}_p, p < n-1$ where the operator '+' is used for concatenation and $\{1\}^p$ is/are the most significant bits. According to (5.2), the maximum value for $\eta(i)$, i.e., the largest index for the least significant bit set to one for $i \in \mathcal{Z}_p$, is obtained when we have

$$\text{bin}(i) = \{1\}^p + \{0\} + \{0\}^{n-(p+1)} = \{1\}^p + \{0\}^{n-p}$$

which is the smallest element in set $i \in \mathcal{Z}_p, p < n-1$, i.e., $z_p = \min(\mathcal{Z}_p)$.

For $p = n-1$, although the notation is in the form of $\{1\}^{n-1} + \{0,1\}^1$, the largest index for the least significant bit set to one is similarly obtained from $\{1\}^{n-1} + \{0\}$ which is the smallest in set $\mathcal{Z}_{n-1}$ ■

Clearly, when $p = 0$, we have $\max(\eta(i)) = \eta(0) = n-1$ for any $i \in \mathcal{Z}_0$.

Remark 5.2 *From Lemma 5.2 we conclude that the deepest stage that the intermediate LLRs are overwritten/updated is when decoding the smallest bit index in set $\mathcal{Z}_p$. Recall that the partial sums used at stage $\eta(z_p)$ are provided after decoding bit with index $z_p - 1$ according to (5.4).*

Now we consider updating intermediate information for i in different sets of $\mathcal{Z}_p$.

Lemma 5.3 *For any $p, p' \in [0, n-1]$, $p < p'$, we have*

$$\eta(z_p) > \eta(z_{p'}) \tag{5.11}$$

Proof: It follows directly from (5.2) and part c of Lemma 5.1. Note that z_p is in the form of $\{1\}^p + \{0\}^{n-p}$. It can be observed that for the smaller p, the position of the least significant bit set to one has a larger index. Therefore, $\eta(z_p)$ is larger. ■

Corollary 5.1 *For any $p, p' \in [0, n-1]$, $p < p'$, we have*

$$\psi(z_p - 1) > \psi(z_{p'} - 1) \tag{5.12}$$

Proof: As $\eta(z_p) = \psi(z_p - 1)$ according to (5.4), then based on Lemma 5.3, it follows that $\psi(z_p - 1) > \psi(z_{p'} - 1)$. ■

Remark 5.3 *From Lemma 5.3 and Corollary 5.1, we conclude that intermediate LLRs and partial sums of stage $\eta(z_p)$ are not overwritten/updated when we are decoding any bit with index $i \in \mathcal{Z}_{p'}, p < p'$. Hence,*

Remark 5.4 *As per Remark 5.3 and the fact that updating the intermediate information is performed from stage $\eta(z_p)$ to stage 0, rewinding the SC algorithm from bit $i \in \mathcal{Z}_{p'}$ to bit z_p, $p < p'$ does not require any additional update of the intermediate LLRs or partial sums.*

We will use Remarks 5.3 and 5.4 in the proposed scheme.

5.3 Efficient Partial Rewinding

We learned in Section 5.1 that we could save memory significantly by knowing the required intermediate LLRs and partial sums needed for decoding each bit. However, there is a drawback

to this efficiency. Since we use limited space for intermediate information instead of $N \log_2 N$ memory elements, we have to overwrite the current values we no longer need to proceed with decoding. In the normal decoding process, the overwriting operation does not cause any data corruption. However, if we need to move backward like in SC-flip, shifted-pruning, or Fano decoding, we may no longer access the intermediate information as it may have been lost due to overwriting.

In this section, based on the update properties we studied in Section 5.2, a scheme is proposed such that rewinding the SC algorithm is performed efficiently by significantly fewer computations comparing with restarting the algorithm.

Suppose the SC algorithm is decoding bit i and needs to rewind the SC process to bit $j, j < i$, and $i, j \in \mathcal{A}$. In SC-flip scheme and shifted-pruning-scheme, we have $i = 2^n - 1$ however, in Fano decoding, $i \leq 2^n - 1, i \in \mathcal{A}$. Since the required intermediate information for decoding bit j may partially be overwritten, we may need to rewind further to a position denoted by j_p. From j_p, the SC algorithm proceeds with the normal decoding up to position j. We shift the pruning window at this position, or we flip the bit u_j and then continue the normal SC-based decoding.

Now, the question is what the position j_p is? Let us assume $i \in \mathcal{Z}_{p'}$ and $j \in \mathcal{Z}_p$. Then,

$$j_p = \begin{cases} z_p & \text{if } z_p < z_{p'} \\ z_{p'} & \text{if } z_p = z_{p'} \end{cases} \tag{5.13}$$

Example 5.3 *Suppose $N = 2^5$ and we need to rewind the SC algorithm from position $i = 31 = (11111)_2$ to $j = 19 = (10011)_2$. We know that $i \in \mathcal{Z}_4$ and $j \in \mathcal{Z}_1$. Therefore, according to (5.13), $j_p = z_p = 16 = (10000)_2$.*

Recursion for Case $\mathbf{z_p = z_{p'}}$: For the case $z_p = z_{p'}$ in (5.13), we may choose a position k, $j_p < k \leq j$ for rewinding, which is more efficient. To this end, let us $k \leftarrow j$ and $m \leftarrow n$, then while $\phi(k) \neq 0$:

- first, truncate the binary representation of $k = (k_{n-1}...k_1k_0)_2$ by removing the bits from position $m - 1 - \phi(k)$ to the most significant bit (inclusive), i.e. position $m - 1$. Note that after truncation, we have a binary number with length $m = m - (\phi(k) + 1)$.
- secondly, find the new set $\mathcal{Z}_{p''}$ such that $k \in \mathcal{Z}_{p''}$ for $k = \sum_{t=m-(\phi(k)+1)}^{n-1} j_t \cdot 2^t - k$.

- then, $j_p = \sum_{t=m-(\phi(k)+1)}^{n-1} j_t \cdot 2^t + z_{p''}$.

We can continue the above procedure recursively to minimize $z_{p''} - j$. Note that in this recursion, k and m are being replaced with new values at each iteration.

Example 5.4 *Suppose $N = 2^5$ and we need to rewind the SC algorithm from position $i = 22 = (10110)_2$ to $j = 19 = (10011)_2$. We know that $i, j \in \mathcal{Z}_1$ and therefore $j_p = z_p = 16 = (10000)_2$. We truncate $j = (10011)_2$ as mentioned above. We get $k = (011)_2$, $k \in \mathcal{Z}_0$, and $z_{p''} = 0$. Hence, the new j_p is $j_p = z_p + z_{p''} = 16$ which is the same as before.*

Example 5.5 *Suppose $N = 2^5$ and we need to rewind the SC algorithm from position $i = 22 = (10110)_2$ to $j = 20 = (10100)_2$. We know that $i, j \in \mathcal{Z}_1$ and therefore $j_p = z_p = 16 = (10000)_2$. However, if we truncate $j = (10100)_2$ as mentioned above, we get $t = (100)_2$, $t \in \mathcal{Z}_1$, and $z_{p''} = 4$. Hence, the new j_p is $j_p = z_p + z_{p''} = 16 + 4 = 20$.*

One can observe that the recursion is not used in the schemes that the rewind is performed from the last bit index. The reason is that bit index $2^n - 1 \in \mathcal{Z}_n$ and this set has only one other element which is $2^n - 2 = z_{n-1}$. On the other hand, if we need to rewind the SC process to a bit index smaller than z_{n-1}, the target bit index will fall into another set $\mathcal{Z}_p$ with different z_p. Hence, this may be used for Fano decoding where the case $z_p = z_{p'}$ is possible. Note that we do not numerically evaluate this approach for Fano decoding as we do not have any other approach to compare with. We can either use this approach or simply we can store all $N \log_2 N$ intermediate LLRs and partial sums and trade a significant complexity reduction with the memory efficiency.

Now, let us revisit the shifted-pruning scheme discussed in Chapter 4. In this scheme (and in the SC-flip scheme), we may need to repeat the SC rewinding process up to T times. Therefore, we need to take this into our consideration. Assuming $t \in [1, T]$ indicates the current iteration, and $j(t)$ and $j_p(t)$ denotes the j and j_p of iteration t, then j_p of the current iteration is obtained by considering $j_p(t-1)$ as follows:

$$j_p = \begin{cases} j_p(t-1) & \text{if } j_p(t) > j_p(t-1) \\ j_p(t) & \text{otherwise} \end{cases} \tag{5.14}$$

As (5.14) shows, if the destination position of the current iteration $j(t)$ is larger than the destination position of the previous iteration, the intermediate information is not valid. The

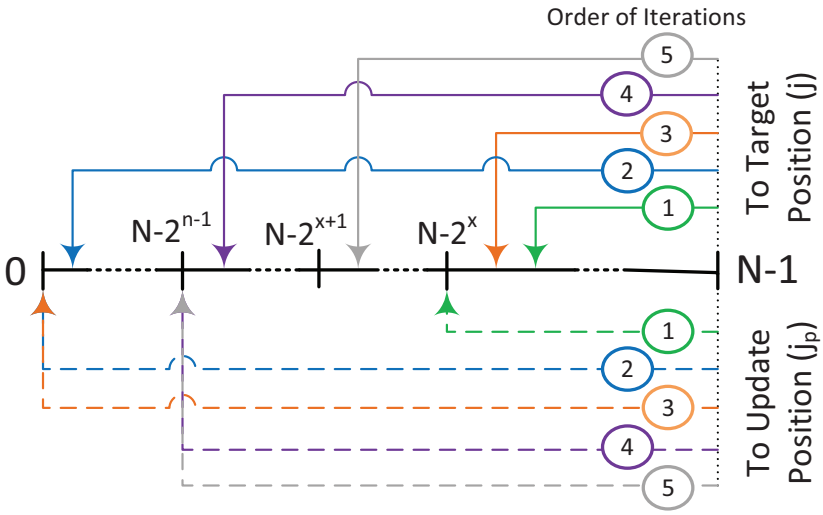

Fig. 5.3: An illustrative example comparing the target position j and update position j_p

reason is that some modification (bit-flipping or shifted-pruning) occurred at position $j(t-1)$ that affects not only the intermediate information but also the decoded data. In other words, we need to go to position $j(t-1)$ and undo the modification and proceed with the decoding up to the position $j(t)$ and then perform the modification of the current iteration. Note that if both $j(t)$ and $j(t-1)$ are in the same $\mathcal{Z}_p$, then $j_p(t) = j_p(t-1)$, hence there will be no difference.

Fig. 5.3 compares j and j_p for an example where 5 iterations are occurring.

Furthermore, when rewinding the list decoder from the last bit position, $N-1$, to position j_p, some of the paths that existed at position j_p in the previous iteration might be eliminated in between and be replaced with other paths. This potential replacement should be addressed when we have a list of paths/candidates, such as in the shifted-pruning scheme, not in the SC-flip scheme. To simplify the problem, we can limit the positions j_p to $j_p = 2^{n-1}$. Because all the computations of the intermediate LLRs from this position, 2^{n-1}, up to the last position, $2^n - 1$, are performed solely based on the channel LLRs and partial sums of stage $\psi(2^{n-1}-1)$. Hence, we need to store the decoded data, $\mathbf{u}[0 : 2^{n-1} - 1]$, and the path metric of all the paths at position $2^{n-1} - 1$. Partials sums can be stored as well or can be computed simply by $\mathbf{u}[0 : 2^{n-1} - 1]\mathbf{G}_{N/2}$.

5.4 Numerical Results

We show that in the additional decoding attempts in SC list decoding and SC decoding, the average complexity (in terms of required time-steps and node visits) can be significantly reduced by partial rewinding instead of full rewinding of SC-based decoder. Note that taking average over all the decoding attempts including the successful attempts in the first run does not give a good insight and a fair comparison in particular at medium and high SNR regimes. The reason is that only a small portion of the total attempts fail requiring additional attempts, e.g., less than 10 failures in 10^4 decoding attempts in the FER range of 10^-4. Hence, the impact of this small portion becomes negligible on the average number of total attempts per codeword at high SNR regimes.

Figures 5.4 and 5.5 compare the average computational complexity of shifted-pruning scheme with and without partial rewinding for two different codes. In Fig. 5.4, the FER and time complexity of polar code of (512,256+12) constructed with DEGA (2dB) and concatenated with CRC12 with polynomial 0xC06 under SC list decoding with list size $L = 8$ with shifted-pruning (SP) are shown. The FER before and after using the efficient partial rewind (PR) scheme clearly shows that the proposed efficient partial rewind scheme does not degrade the decoder's performance as we expected. However, it reduces the average time-steps over additional iterations (when the decoding fails) by over 30% (from $2N - 2 = 1022$ time-steps (or clock cycles) [66] down to about 700 time-steps). The average time steps over all the iterations also reduce, but at high SNRs, it approaches 1022. The reason is that at high SNR regimes, the number of errors, FER, is low. Compared with the total number of codewords decoded successfully, just a small number of codewords are failed to be decoded in the first attempt and need additional attempts/iterations.

As Fig. 5.5 shows, the reduction in the average time complexity for efficient partial rewind scheme improves for polar code P(512,128+12) constructed with DEGA (1 dB). The average time-steps over additional iterations by about 45% (from $2N - 2 = 1022$ time-steps down to about 570 time-steps). The reason is that at a low code rate of $R = 1/4$, the positions j for shifting the pruning window are mostly located in the interval $[N/2, N - 1]$ where the partial rewinding can be effective in reducing the complexity. Recall that if $j \in [0, N/2 - 1] = \mathcal{Z}_0$, then $j_p = z_p = 0$. That means a full rewind is needed. One can guess that the reduction in the complexity would be less at high code rates where the position j for shifting are dominantly

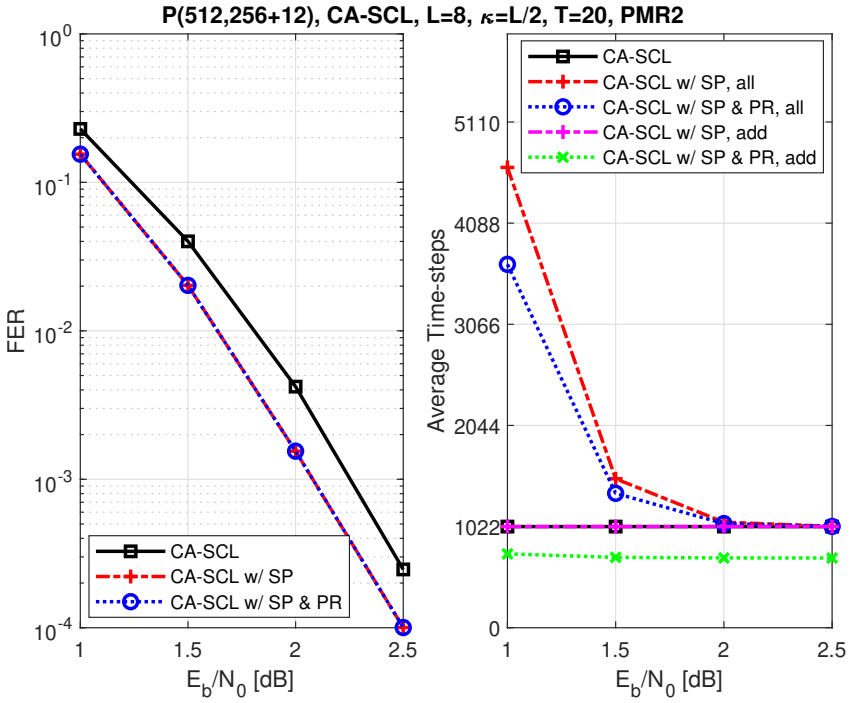

Fig. 5.4: Comparison of FER and average time complexity of P(512,256+12) under CA-SCL decoding without and with (w/) shifted-pruning scheme (SP), and with partial rewinding (PR). 'all' and 'add' indicate average over all the decoding iterations and average only over additional iterations for shifted-pruning, respectively.

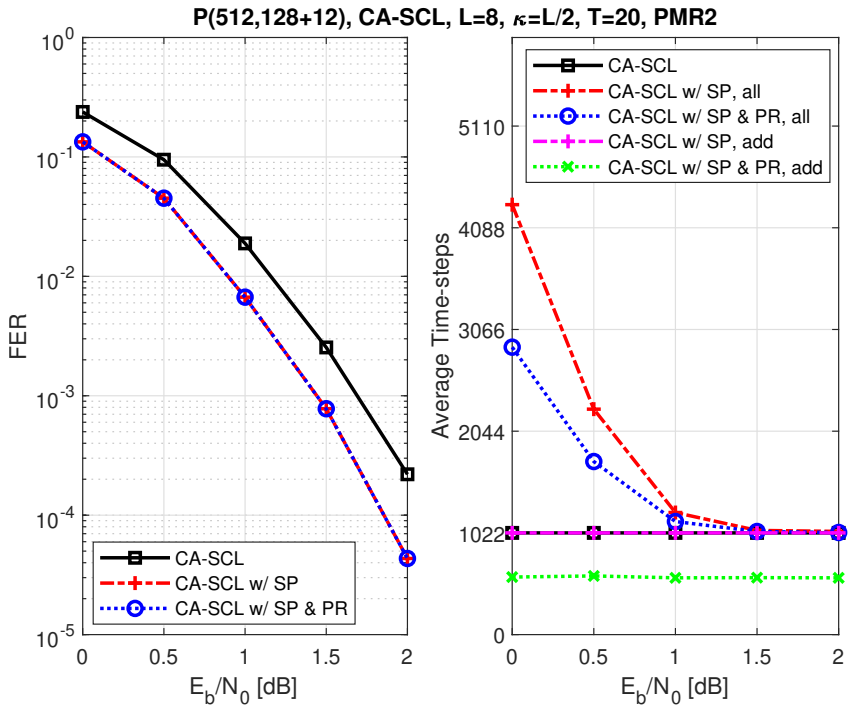

Fig. 5.5: Comparison of FER and average time complexity of P(512,128+12) under CA-SCL decoding without and with (w/) shifted-pruning scheme (SP), and with partial rewinding (PR). 'all' and 'add' indicate average over all the decoding iterations and average only over additional iterations for shifted-pruning, respectively.

located in $[0, N/2-1] = \mathcal{Z}_0$ as the reliability of these bit-positions are less relative to the ones in $[N/2, N-1]$.

Similarly, we can show a significant reduction in the complexity of the additional attempts in the SC-flip decoding algorithm. Fig. 5.6, 5.7, and 5.8 illustrate the reduction in the node visits on average for CRC-polar codes of length $N = 512$ at rates $R = 1/4, 1/2, 3/4$. The metric used in the SC-flip implementation is similar to the one in [25] as our purpose in this work is not the performance of SC-flip but to show the reduction in the complexity. Hence, a similar result can be obtained by applying the partial rewind on any variant of the SC-flip decoder. As can be seen, the FER remains unchanged by partial rewind, while the additional decoding attempts are performed with significantly lower node visits on average. This reduction increases at high SNR regimes as the targeted positions for bit-flipping become more accurate and their number decreases. The main contribution to this decrease is related to 5.14 where $j_p = j_p(t)$ in the fewer additional attempts, mostly one attempt.

Fig. 5.9 compares the time complexity at at rates $R = 1/4, 1/2, 3/4$. By recalling Remark

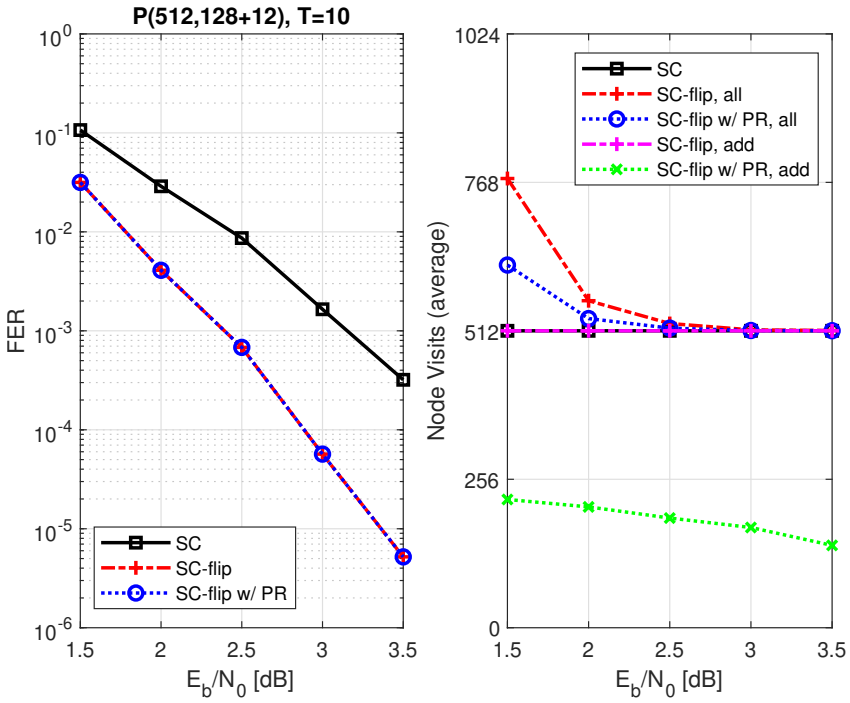

Fig. 5.6: Comparison of FER and average node visits of P(512,128+12) under SC decoding without and with (w/) bit-flipping, and with partial rewinding (PR). 'all' and 'add' indicate average over all the decoding iterations and average only over additional iterations for bit-flipping, respectively.

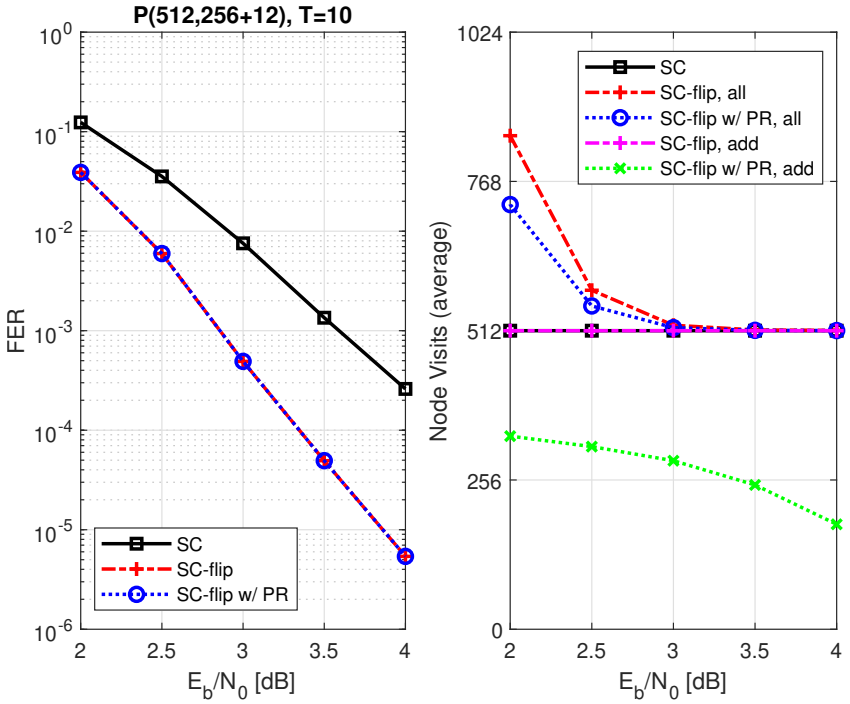

Fig. 5.7: Comparison of FER and average node visits of P(512,256+12) under SC decoding without and with (w/) bit-flipping, and with partial rewinding (PR). 'all' and 'add' indicate average over all the decoding iterations and average only over additional iterations for bit-flipping, respectively.

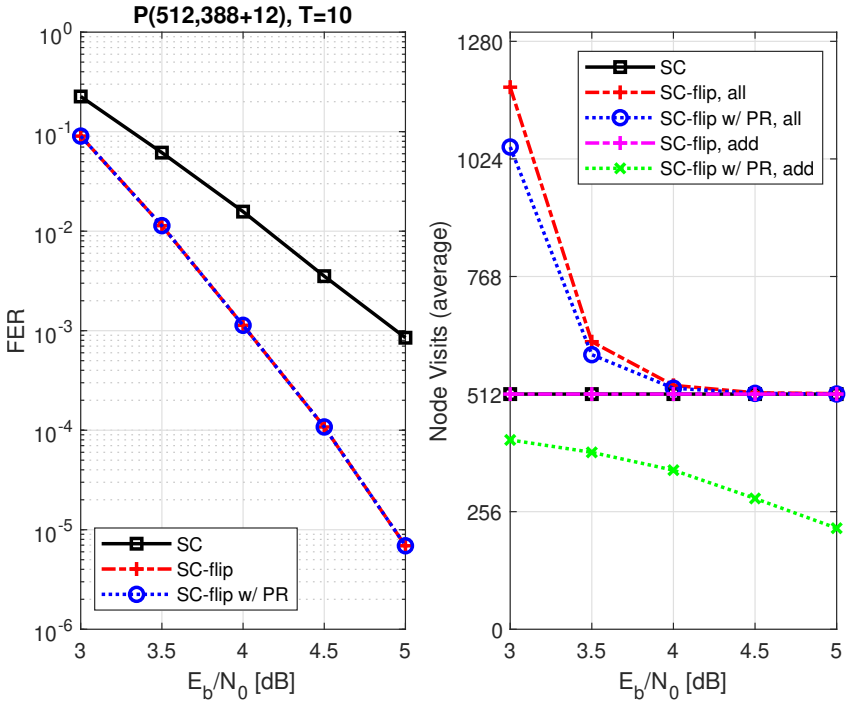

Fig. 5.8: Comparison of FER and average node visits of P(512,388+12) under SC decoding without and with (w/) bit-flipping, and with partial rewinding (PR). 'all' and 'add' indicate average over all the decoding iterations and average only over additional iterations for bit-flipping, respectively.

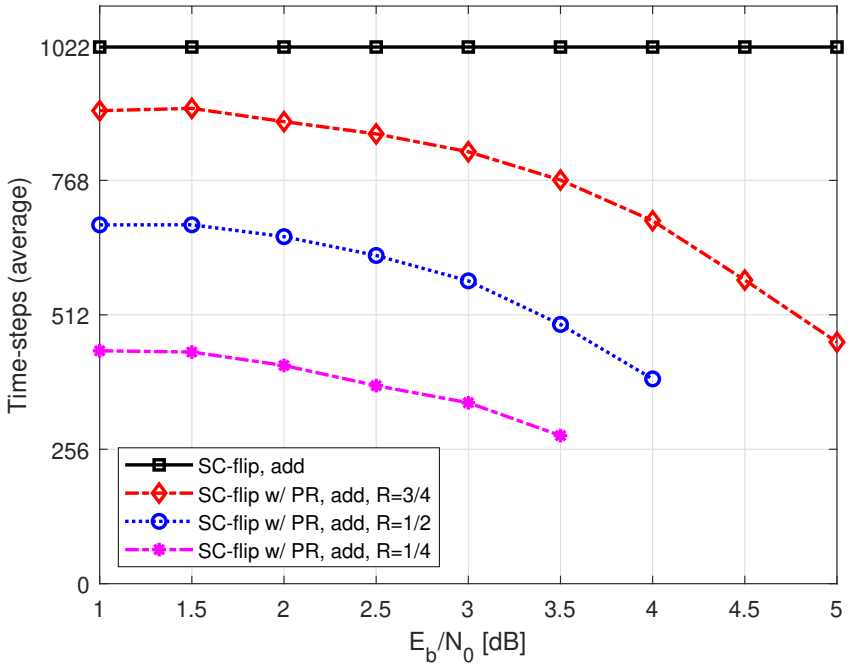

Fig. 5.9: Comparison of the average time-steps of codes with length $N = 512$ with different code rates under SC-flip decoding with (w/) partial rewinding (PR). 'add' indicate average only over additional iterations for bit-flipping.

5.1, one can observe that at low code rates, $(N-1) - j_p$ on average decreases significantly comparing with high rates, therefore, we expect to visit a fewer nodes in the additional decoding attempts and consequently the time complexity reduces more than high code rates. Similar to node visits, this reduction increases at high SNR regimes as the targeted positions for bit-flipping become more accurate and their number decreases. Note that the average time complexity over additional iterations does not depend on the code rate if we don't use partial rewinding as we start re-decoding from bit 0 for any code rate.

5.5 Summary

When decoding fails in the first decoding attempt, a partial rewind of the SC process for additional attempts is needed in the memory-efficient SC-based decoders. In this chapter, an efficient partial rewinding approach based on the properties of the SC algorithm was proposed. This approach relies on the properties of the SC process and its updating schedule. Then, this scheme was adapted to multiple rewinds, and to SC list decoding, where there exists more than one path comparing with SC decoding. The numerical results showed a significant reduction in

the average time and computational complexity of additional decoding attempts in the SC-flip decoding and SC list decoding under the shifted-pruning scheme while the performance remains the same.

Chapter 6

Convolutional Pre-coding and List decoding of PAC Codes

"I just wondered how things were put together."

— Claude Shannon

This chapter covers the recently introduced Polarization-adjusted convolutional (PAC) codes. PAC codes are concatenated codes in which a convolutional transform is employed before the polar transform. In this scheme, the polar transform (as a mapper) and the successive cancellation process (as a demapper) present a synthetic vector channel to the convolutional transformation. It was shown numerically that the reason for the superiority of error correction performance of PAC codes relative to polar codes is the improvement of polar codes' weight distribution due to this concatenation. In this chapter, we explicitly show why the convolutional precoding reduces the number of minimum-weight codewords. Furthermore, we show wherein the precoding stage is not effective with respect to the underlying rate-profile. In other words, we answer why PAC codes have a significantly smaller number of minimum-weight codewords compared to polar codes. Then, we recognize the potential weakness of the convolutional precoding, which is unequal error protection (UEP) of the information bits. Finally, we assess the possibility of mitigating this weakness by irregular convolutional precoding.

6.1 Polarization-adjusted Convolutional Codes

Polarization-adjusted convolutional codes, denoted by $PAC(N, K, \mathcal{B}, \mathbf{c})$, are based on the outer convolutional transform and inner polar transform. One may consider PAC coding as a polar coding scheme in which the inputs to the frozen bit-channels are linear combinations of previous bits obtained by convolutional transforms. Thus, given that the previous bits have been estimated correctly, the decoder can still determine the value transmitted by the corresponding "bad channels". In the following Sections, the encoding and decoding of PAC codes are described in detail.

The information bits $\mathbf{d} = (d_0, d_1, ..., d_{K-1})$ are first mapped to a vector $\mathbf{v} = (v_0, v_1, ..., v_{N-1})$

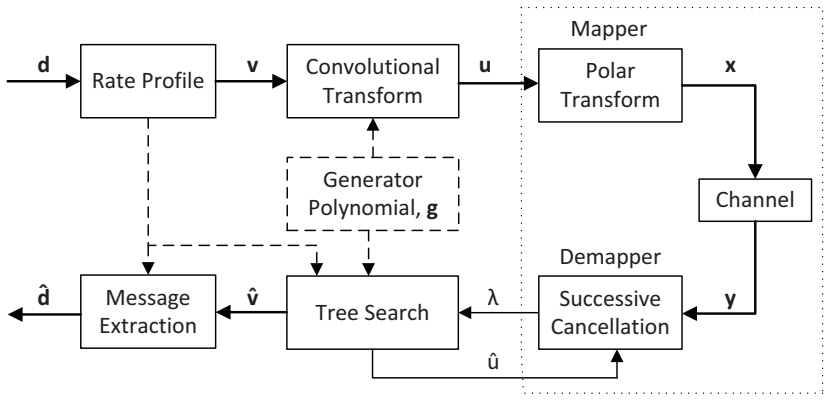

Fig. 6.1: PAC coding scheme

using a rate-profile. The rate-profile (a.k.a. code construction) is formed based on the index set $\mathcal{B}$ such that $v_{\mathcal{B}} = d$, and $v_{\mathcal{B}^c} = 0$. Note that the constraint $v_{\mathcal{B}^c} = 0$ simply leads to an irregular tree code.

After rate-profiling, the vector $\mathbf{v}$ is transformed using a convolutional generator polynomial $\mathbf{c} = [c_0, ..., c_m]$ to $u_i = \sum_{j=0}^{m} c_j v_{i-j}$, where $\mathbf{g}_i \in \{0, 1\}$ as discussed in Section 7.1 (see sub-routine *convTransform* in Algorithm 4). Equivalently, the convolutional transform (CT) can be represented in matrix form where the rows of an upper-triangular *generator matrix* $\mathbf{G}$ are formed by shifting the vector $\mathbf{c} = [\mathbf{g}_0, \dots \mathbf{g}_m]$. The number of rows equals the block-length. Given the generator matrix $\mathbf{G}$, we can encode the message block $\mathbf{v}$ as $\mathbf{u} = \mathbf{vG}$. As a result of this pre-transformation, u_i for $i \in \mathcal{B}^c$ are no longer fixed or known a priori (as 0's in $\mathbf{u}$) - unlike in conventional. In fact, these formerly frozen bits are acting as parity check (PC) bits [18] or dynamic frozen bits [9].

Then, as Fig. 6.1 shows, vector $\mathbf{u}$ is mapped to $\mathbf{x}$ by employing the conventional polar transform $\mathbf{P}_n$ defined in Section 2.6. Hence, the the N-bit rate-profiled data (or block-length) should be a power of 2, i.e., $N = 2^n$. In summary, the polar transformation is performed by $\mathbf{x} = \mathbf{uP}_n$. Algorithm 4 summarizes the encoding process. In this algorithm, cState and currState represent the *current state* of the m-bit memory.

6.2 Minimum-weight Codewords in PAC Codes

The minimum Hamming weight or in short min-weight defines the error correction capability of a code. The linear block codes can correct up to $\lfloor (d_{min} - 1)/2 \rfloor$ errors. Note that the min-

Algorithm 4: Encoding of PAC Codes

input : profiled information bits $\mathbf{v}$, $\mathbf{c}$
output: the codeword $\mathbf{x}$

$\mathbf{u} \leftarrow$ convTrans($\mathbf{v}$, $\mathbf{c}$)
$\mathbf{x} \leftarrow$ polarTrans($\mathbf{u}$) `// Like polar encoder`
return $\mathbf{x}$;
subroutine `convTrans(`$\mathbf{v}$, $\mathbf{c}$`):`
    cState[1,...,$|\mathbf{c}|-1$] $\leftarrow$ [0,...,0] `// currState`
    for $i \leftarrow 0$ **to** $|\mathbf{v}|-1$ **do**
        $(u_i, cState) \leftarrow$ conv1bTrans(v_i, cState, $\mathbf{c}$)
    return $\mathbf{u}$;
subroutine `conv1bTrans(`v, *currState*, $\mathbf{c}$`):`
    $u \leftarrow v \cdot \mathbf{g}_0$
    for $j \leftarrow 1$ **to** $|\mathbf{c}|$ **do**
        if $\mathbf{g}_j = 1$ **then**
            $u \leftarrow u \oplus$ currState[$j-1$]
    nextState $\leftarrow [v_i]$ + currState[1,...,$|\mathbf{c}|-2$]
    return (u, nextState);

weight (w_{min}) is also minimum Hamming distance (d_{min}) (see 3.17 in [70]). Besides min-weight, the number of min-weight codewords is also important. It was shown in [70] that at high E_b/N_0, the upper bound for block error probability under soft-decision maximum likelihood decoding (MLD) can be approximated by

$$P_e^{ML} \approx A_{d_{min}} Q(\sqrt{2 d_{min} \cdot R \cdot E_b/N_0}) \tag{6.1}$$

where $A_{d_{min}}$ is the number of min-weight codewords, a.k.a error coefficient, $Q(\cdot)$ is the tail probability of the normal distribution $\mathcal{N}(0,1)$, and R is the code rate. As $A_{d_{min}}$ is directly proportional with the upper bound for the error correction performance of a code, it can be used as a measure to anticipate the direction of change in the block error rate when $A_{d_{min}}$ changes.

Enumeration of minimum Hamming weight codewords of PAC codes in [48] showed that they have a significantly smaller number of min-weight codewords in comparison with polar codes. Table 6.1 compares min-weight codewords of polar codes and PAC codes. In this work, the method discussed in [48] was employed with $L = 2^{19}$ to obtain $A_{d_{min}}$.

As can be seen, $A_{d_{min}} = A_{16} = 94488$ for the polar code (128,64,16) constructed with RM rate profile, whereas $A_{16} \approx 3120$ is much smaller for the PAC code (128,64,16) with the same

Table 6.1: The number of min-weight codewords, $A_{d_{min}}$, with RM-polar rate profile

	(128,32,16)	(128,64,16)	(128,96,8)
Polar Codes	56	94488	74288
PAC Codes	56	3120	13904
	(64,16,16)	(64,32,8)	(64,48,4)
Polar Codes	364	664	432
PAC Codes	236	472	320

rate profile. In this section, we discuss the reason behind this significant reduction.

It was shown in [71] by example that a properly designed upper-triangular matrix $\mathbf{G}$ in general may remove some of the bit-patterns with minimum Hamming weight from the codebook as a result of $\mathbf{GP}_n$ matrix multiplication in $\mathbf{v}\big(\mathbf{GP}_n\big)$. However, in this work, we show how convolutional precoding, i.e., $\mathbf{vG}$ matrix multiplication in $\big(\mathbf{vG}\big)\mathbf{P}_n$, can avoid generating some of the minimum weight codewords available in the codebook of polar codes generated by $\mathbf{vP}_n$.

First, let us look at the process that min-weight codewords are generated. The codewords of any linear block code are formed by combining or adding the rows of its generator matrix. This summation of the rows is realized by matrix multiplication of the message vector, and the generator matrix and the coordinates of 1's in the message vector determine which row(s) should be combined. The simplest codeword is formed by the individual rows of the generator matrix, and it occurs when the Hamming weight of the message vector is one. Hence, the rows of $\mathbf{P}_n$ in $\mathcal{A}$ with min-weight are individually considered as min-weight codewords. The other min-weight codewords are generated by the combination of two or more rows in $\mathcal{A}$. Here, we just show it for the case of individual row codewords, as the other cases follow the same concept.

We define the cosets resulting from combining a min-weight row at coordinate i with possibly other rows with indices larger than i as

$$C(0_0^{i-1}, 1) = \mathbf{g}_i \oplus \bigoplus_{k \in I} \mathbf{g}_k \tag{6.2}$$

where $I \subset \{j | j > i\}$. The following lemma defines a lower bound for the weight of codewords in the coset $C(0_0^{i-1}, 1)$. The notation $w(.)$ is used for the Hamming weight of vectors.

Lemma 6.1 *The weight of any codeword in the coset* $C(0_0^{i-1}, 1)$ *is* $w\big(C(0_0^{i-1}, 1)\big) \geq w(\mathbf{g}_i)$.

Proof: This can be shown by mathematical induction (see [71, Corollary 1]). ■

Now, given a polar code with length N, the index set $\mathcal{A}$ and $A_{d_{min}}$, we show by construction

that if we apply the precoding on the same length and rate profile, in a mapping of min-weight polar codewords to the corresponding PAC codewords, some of the min-weight codewords may find a larger weight as a result of precoding. This mapping is shown in Fig. 6.2 where only a portion of min-weight codewords on the left hand side (i.e. polar codebook) are mapped to the collection of min-weight codewords in PAC codes on the right hand side, shown by arrow (i).

Let us consider all the min-weight rows in $\mathbf{P}_n$ as a subset of all the min-weight codewords of polar codes. If we attempt to produce such min-weight codewords in PAC coding, we shall see that as a result of convolutional precoding $\mathbf{vG}$, 1) some of these codewords are kept unchanged, 2) some are replaced with a different min-weight codewords, and 3) some are replaced with codewords with larger weights.

Note that in polar coding, $\mathbf{v} = \mathbf{u}$ as there is no precoding operation. Now, consider a row $\mathbf{g}_i$ of $\mathbf{P}_n$ with $w(\mathbf{g}_i) = w_{min}$ as a minimum weight codeword of the polar code. In order to generate such a codeword, we need a vector $\mathbf{u}$ such that $u_i = 1$ and $u_j = 0$ for $j \neq i$, then $\mathbf{u}\mathbf{P}_n = \mathbf{g}_i$. However, such a vector $\mathbf{u}$ may not be obtained by precoding $\mathbf{vG} = \mathbf{u}$.

Recall $u_j = \sum_{k=0}^{m} c_k v_{j-k}$ from Section 2.6. In order to get $u_j = 0$ for any $j > i$ and $j \in \mathcal{A}$, it is possible to choose either $v_j = 0$ (for the case $\sum_{k=1}^{m} c_k v_{j-k} = 0$) or $v_j = 1$ (when $\sum_{k=1}^{m} c_j v_{j-k} = 1$). However, for any $j \in \mathcal{A}^c$, by convention $v_j = 0$ in the rate profile. Hence, $u_j = 1$ when $\sum_{k=1}^{m} c_k v_{j-k} = 1$. This inevitably combines $\mathbf{g}_i$ with $\mathbf{g}_j$ for any $j \in \mathcal{A}^c$, $j > i$ where $u_j = 1$. As Lemma 6.1 showed, the resulting weight will be

$$w(\mathbf{g}_i \oplus \bigoplus_{j \in \mathcal{J} \subseteq \mathcal{A}^c} \mathbf{g}_j) \geq w_{min} \tag{6.3}$$

where $\mathcal{J} = \{j | j \in \mathcal{A}^c, j > i, \text{and } u_j = 1\}$.

Now, we look at the three aforementioned resulting cases:

1. $\mathbf{g}_i \oplus \bigoplus_{j \in \mathcal{J} \subseteq \mathcal{A}^c} \mathbf{g}_j = \mathbf{g}_i$: This case occurs where there is no $j \in \mathcal{A}^c$ for $j > i$ (i.e., $\mathcal{J} = \emptyset$) or depending on the choice of polynomial $\mathbf{c}$, we may get $u_j = 0$ for any $j \in \mathcal{A}^c$, $j > i$. See arrow (i) in Fig. 6.2.

2. $\mathbf{g}_i \oplus \bigoplus_{j \in \mathcal{J} \subseteq \mathcal{A}^c} \mathbf{g}_j = \mathbf{x}$ where $\mathbf{x} \neq \mathbf{g}_i$ but $w(\mathbf{x}) = w_{min}$: This case occurs where $w(\bigoplus_{j \in \mathcal{J} \subseteq \mathcal{A}^c} \mathbf{g}_j) =$

$2w(\mathbf{g}_i \wedge \bigoplus_{j\in\mathcal{J}\subseteq\mathcal{A}^c} \mathbf{g}_j)$ as according to the principle of inclusion exclusion, we have

$$w(\mathbf{g}_i \oplus \bigoplus_{j\in\mathcal{J}\subseteq\mathcal{A}^c} \mathbf{g}_j) = w(\mathbf{g}_i) + w(\bigoplus_{j\in\mathcal{J}\subseteq\mathcal{A}^c} \mathbf{g}_j) - 2w(\mathbf{g}_i \wedge \bigoplus_{j\in\mathcal{J}\subseteq\mathcal{A}^c} \mathbf{g}_j) \tag{6.4}$$

The operator wedge product $\wedge$ is equivalent to bit-wise ANDing. See arrow (i) in Fig. 6.2.

3. $\mathbf{g}_i \oplus \bigoplus_{j\in\mathcal{J}\subseteq\mathcal{A}^c} \mathbf{g}_j = \mathbf{x}$ where $\mathbf{x} \neq \mathbf{g}_i$ and $w(\mathbf{x}) > w_{min}$: This case occurs where $w(\bigoplus_{j\in\mathcal{J}\subseteq\mathcal{A}^c} \mathbf{g}_j) > 2w(\mathbf{g}_i \wedge \bigoplus_{j\in\mathcal{J}\subseteq\mathcal{A}^c} \mathbf{g}_j)$. See arrow (ii) in Fig. 6.2.

The second case is where the min-weight codewords in PAC codes differ from the ones in polar codes, however they are still min-weight codewords. The third case is where PAC codes lose some of the min-weight codewords that exist in polar codes. Note that the resulting larger weight codewords change the weight distribution of PAC codes.

In similar way, we can show that this event occurs for the minimum weight codewords resulting from the combination of more than one row of $\mathbf{P}_n$ with indices in set $\mathcal{A}$.

Example 6.1 *For the polar code and PAC code of (64,48,4) with RM-polar rate profile, we have* $A_4 = 432$ *and 320 for polar codes and PAC codes, respectively. The set* $\mathcal{M} = \{i | i \in \mathcal{A}, \text{and } w(\mathbf{g}_i) = w_{min}\} = \{20, 24, 34, 36, 40, 48\}$ *and the set* $\mathcal{N} = \{j | j \in \mathcal{A}^c, \text{and for any } i \in \mathcal{M}, j > i\} = \{32, 33\}$. *Assuming* $\mathbf{c} = [1, 0, 1, 1, 0, 1, 1]$, *then instead of* $\mathbf{g}_{20}$, *we will have* $\mathbf{g}_{20} \oplus \mathbf{g}_{32}$ *yet with weight* $w_{min} = 4$ *(case 2) in the codebook of PAC codes. Note that the elements of vector* $\mathbf{u}$ *are zeros except at coordinates 20 and 32, however, the vector* $\mathbf{v}$ *will have many non-zero elements in order to get the aforementioned vector* $\mathbf{u}$ *after precoding. Also, instead of* $\mathbf{g}_{24}$, *we will have* $\mathbf{g}_{24} \oplus \mathbf{g}_{33}$ *with weight 6 which is greater than* w_{min} *(case 3 shown by arrow (ii) in Fig. 6.2). The other rows with min-weight including* $\mathbf{g}_{34}$, $\mathbf{g}_{36}$, $\mathbf{g}_{40}$, *and* $\mathbf{g}_{48}$ *will exist unchanged in the codebook of PAC codes as there is no row* $j \in \mathcal{A}^c$ *for* $j > 34$ *(case 1).*

Note that by applying the precoding, there is no way to generate min-weight codewords other than based on the coset $C(0_0^{i-1}, 1)$ where $w(\mathbf{g}_i) = w_{min}$ as the following corollary concludes.

Corollary 6.1 *If* $i \in \mathcal{A}$ *and* $w(\mathbf{g}_i) > w_{min}$, *inclusion of row(s)* $k \in \mathcal{A}^c$ *for* $k > i$ *in the coset* $C(0_0^{i-1}, 1)$ *does not produce a coset with weight* w_{min} *or less.*

Proof: It follows directly from Lemma 6.1 the weight of the coset $C(0_0^{i-1}, 1)$ cannot be smaller than the weight of $\mathbf{g}_i$. ■

Now, consider the codewords with weight larger than w_{min} resulting from the coset $C(0_0^{i-1}, 1)$ where $w(\mathbf{g}_i) = w_{min}$. The inclusion of $\mathbf{g}_j$ for $j > i$ and $j \in \mathcal{A}^c$ in the coset as a result of precoding may reduce the weight of some of the corresponding codewords in the polar codes. This case is shown by arrow (iii) in Fig. 6.2.

Example 6.2 *For the polar code and PAC code of (32,16,8) with RM rate profile, the set* $\mathcal{M} = \{i | i \in \mathcal{A}, \text{and } w(\mathbf{g}_i) = w_{min}\} = \{13, 14, 21, 22, 25, 26, 28\}$ *and the set* $\mathcal{N} = \{j | j \in \mathcal{A}^c, \text{and for any } i \in \mathcal{M}, j > i\} = \{16, 17, 18, 20, 24\}$. *Considering the codeword resulting from the combination* $\mathbf{g}_{13} \oplus \mathbf{g}_{22}$ *which gives the weight* $w(\mathbf{g}_{13} \oplus \mathbf{g}_{22}) = 12$, *by inclusion of* $\mathbf{g}_{18}$ *(*$j = 18 \in \mathcal{A}^c$*), the weight will be* $w(\mathbf{g}_{13} \oplus \mathbf{g}_{22} \oplus \mathbf{g}_{18}) = 8$.

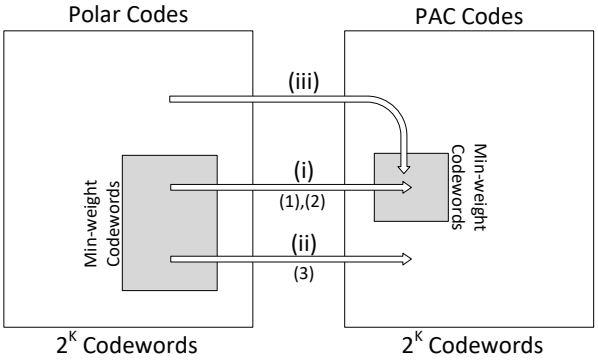

Fig. 6.2: Mapping of min-weight codewords in the codebook of polar codes to PAC codes'. The cases (1), (2), and (3) discussed in Section III are shown in the figure.

One can observe that statistically the case of getting the weight w_{min} or any specific weight as a result of inclusion of the rows in $\mathcal{A}^c$ is less frequent relative to the case of getting a weight larger than w_{min}. The numerical results of enumeration of min-weight codewords support this observation. As Table 6.1 shows the reduction in the min-weight codewords of polar codes except in an special case which is the subject of the following corollary.

Corollary 6.2 *Suppose* $\mathcal{M} = \{i | i \in \mathcal{A}, \text{and } w(\mathbf{g}_i) = w_{min}\}$. *If for any* $i \in \mathcal{M}$, *there is no* $j \in \mathcal{A}^c$ *such that* $j > i$, *then* $A_{d_{min}}(\mathbf{vGP}_n) = A_{d_{min}}(\mathbf{vP}_n)$.

Proof: In this case, there is no inclusion of rows with index $j \in \mathcal{A}^c$ in the coset $C(0_0^{i-1}, 1)$ where $w(\mathbf{g}_i)$. Hence, as it was discussed earlier, it is possible to find a vector $\mathbf{v}$ to generate all the possible combinations of rows identical to polar codes.

Suppose for every codeword $\mathbf{x}$ where $w(\mathbf{x}) = w_{min}$, there exists a vector $\mathbf{u}$ such that $\mathbf{uP}_n = \mathbf{x}$. Now, since there is no $j \in \mathcal{A}^c$ for $j > i$, then $v_j \in \{0, 1\}$. Hence, due to this flexibility of v_j value, there always exists a vector $\mathbf{v}$ such that $\mathbf{vG} = \mathbf{u}$. Therefore, since the vector $\mathbf{u}$ resulted from precoding equals $\mathbf{u}$ of polar coding, the min-weight codewords of polar codes and PAC codes will be the same. Recall that if there exists a $j \in \mathcal{A}^c$ for $j > i$, then $v_j = 0$ and due to the inflexibility, $\mathbf{u}$ in $\mathbf{vG} = \mathbf{u}$ will be limited to a subset of $\mathbf{u}$ in polar coding. ■

Example 6.3 *For the polar code and PAC code of (128,32,16) with RM-polar rate profile, we have* $A_{16} = 56$. *Knowing* $\mathcal{M} = \{114, 116, 120\}$, *for any* $i \in \mathcal{M}$, *there is no* $j \in \mathcal{A}^c$ *such that* $j > i$ *as the largest* j *in* $\mathcal{A}^c$ *is 113 which is smaller than 114.*

In summary, the inevitable inclusion of row(s) $\mathbf{g}_j$ for any $j \in \mathcal{A}^c$ and $j > i$ to the row combinations which are supposed to give min-weight codewords in polar codes may result in codewords with larger weights (arrow (ii) in Fig. 6.2). Note that this inclusion depends on $\mathbf{c}$ and $\mathbf{d}$ and here we just discussed the possibility of the inclusion in general, regardless of the choice of $\mathbf{c}$ which is discussed in the next section. Note that the selection of information bits in PAC codes, set $\mathcal{A}$, comparing with distance-based constructions such as [72] in which the focus is on maximizing d_{min}, or mixed distance and reliability-based constructions in [29, 73] where both reliability of the bit-channels and d_{min} are considered, is similar to polar codes. In this work, we used RM-polar rate-profile for short codes where the minimum distance was maximized and DE/GA for medium-length codes where the bit channels reliability is the main criteria in the code construction. As the numerical results for the performance showed in [48], this selection is valid for polar codes as well, i.e., short polar codes perform better with RM-polar rate-profile than reliability-based constructions. Nevertheless, the precoding does not change the minimum distance of the code as the coset defined in (6.2) does not apply on the bits in $\mathcal{A}^c$ simply because u_i in the coset cannot be zero.

6.2.1 PAC List Decoding

PAC codes as (irregular) tree codes can be decoded using the tree search algorithms discussed in Section 2.6. In this section, we consider the list decoding for PAC codes which trades a fixed time complexity for a large memory requirement (to store a list of paths) and is easier to implement. Then, in the next section and the rest of this chapter, we focus on Fano decoding which has a variable time complexity, but is much more memory-efficient. Note that the list

decoding in the context of convolutional codes is called *M-algorithm*. In the context of PAC codes, some results using list decoding were first presented by Huawei in ITW 2019 [74]. Later, we implemented list decoding for PAC codes in [75] independently of [49].

Algorithm 5 (see page 101) illustrates the list decoding approach. In the beginning, there is a single path in the list. When the index of the current bit is in the set $\mathcal{B}^c$, the decoder knows its value, usually $v_i = 0$ and therefore it is encoded into u_i based on the current memory state *currState* and the generator polynomial $\mathbf{c}$ in line 7. Note that the subroutine *conv1bTrans* is identical with the one in Algorithm 4 (see page 87). Then, using the decision LLR λ_0^i obtained in line 5, the corresponding path metric is calculated using subroutine *calcPM*. Eventually, the decoded value u_i is fed back into SC process in line 9 to calculate partial sums. On the other hand, if the index of the current bit is in the set $\mathcal{B}$ (see lines 19-26), there are two options for the value of v_i, 0 and 1, to be considered in line 24. For each option of 0 and 1, the aforementioned process for $i \in \mathcal{B}^c$ including convolutional encoding, and calculating path metric is performed and then the two encoded values $u_i = 0$ and 1 are fed back into SC process. The subroutines *updateLLRs*, *updatePartialSums*, and *prunePaths* in Algorithm 7 (see page 122) are identical to the ones used in Sc decoding and SCL decoding of polar codes. Note that the vectors λ and β are the LLRs and partial sums, respectively.

One can notice that the process of list decoding for PAC codes is similar to that for polar codes except for the additional convolutional re-encoding at each decoding step for which the next memory state is stored for each path. For medium and long block-lengths, we can also append CRC-bits or parity check (PC) bits to the information bits to help in detecting the correct path. To reduce the computational complexity and the performance of list decoding, the methods proposed in the literature such as in [37, 38] can be applied to PAC list decoding as well.

List decoding, with its non-backtracking tree search approach, requires very large list sizes (typically $L = 256$ or more) to reach the dispersion bound [76], as it will be shown in Section 6.3.3. More memory-efficient backtracking search algorithms such as the Fano algorithm can approach the dispersion bound at the cost of a higher average time complexity at low SNR regimes.

6.3 Numerical Results

In this Section, the error correction performance and the complexity of different tree search algorithms with different setups, using the previously discussed tree search complexity-reduction ideas and adaptive metric, are analyzed.

6.3.1 Rate Profiling

To obtain the numerical results in this Section, we use different rate-profiles such as Reed-Muller (RM), density evolution with Gaussian approximation [9], and the polarization weight (PW) [65] with minimum row-weights eliminated. Fig. 6.3 illustrates the aforementioned rate-profiles. Here, we briefly revise the RM-profile and the modified PW-profile.

The bit-channels for information bits are selected according to the row-weights ($w_i = w(\mathbf{g}_i)$ where $\mathbf{g}_i$ is the i-th row) of $\mathbf{G}_N$. When the candidate bit-channels with the smallest row-weight is more than need, the more reliable ones are selected. In this case, the rate-profile is called *RM-polar* [29]. In this work, the reliability measure is the mean LLR obtained from density evolution with Gaussian approximation (DEGA).

In this method, the bit-channels for information bits are selected among the ones with the largest polarization weight (W_i), $W_i = \sum_{j=0}^{n-1} b_j \cdot 2^{j \cdot \frac{1}{4}}$, where $i = b_{n-1}...b_0$ is the binary representation of i [65]. In order to improve the distance property, we propose to freeze the selected bit-channels with minimum row-weight and replace them with the bit-channels with lower W_i, but larger w_i.

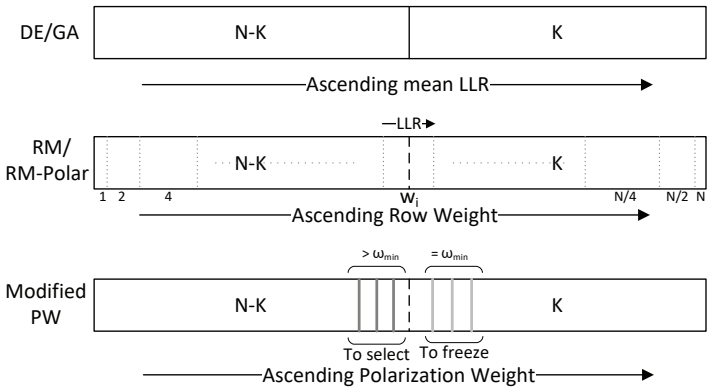

Fig. 6.3: Rate-profile Schemes

In the simulations, we employ different generator polynomials (0o36, 0o133, 0o177, and

0o1563 in octal format) with constraint lengths 5,7,7, and 10, respectively. The numerical results show that the difference among them in terms of FER is negligible in the low SNR regime and small in high SNRs.

Finally, for the purpose of comparison in the figures, we use the dispersion bound [76] a.k.a. Polyanskiy-Poor-Verdu (PPV) bound or finite-length bound which is a Gaussian approximation on the block error probability of finite-length block codes. Additionally, we employ lower bound on ML performance as well. This bound is obtained under list decoding with $L = 256$ by assuming that ML decoder would fail when $\hat{\mathbf{v}} \neq \mathbf{v}$ but $\sum_{i=0}^{N-1} ||\hat{x}_i - y_i|| < \sum_{i=0}^{N-1} ||x_i - y_i||$ where $\hat{\mathbf{x}} = \hat{\mathbf{v}}\mathbf{G}\mathbf{P}_\mathbf{n}$.

6.3.2 Distance Spectrum

As discussed in Section 6.1, by convolutional pre-coding, we are no longer transmitting fixed known values, e.g., 0 frozen bits, over low-reliability (bad) synthetic channels, but random values generated by a linear combination of information bits. To analyze the impact of this difference on polar codes, we use the multilevel SCLD-based search method in [77] to enumerate the codewords with the minimum Hamming distance, d_{min}. We use the size of $L = 2^{17}$ and in each iteration we introduce a one-bit error in the positions corresponding to the minimum row weight in P_n, when the all-zero codeword is transmitted and no noise is added. Re-encoding the candidate messages, remaining in the list at the end of decoding, shows that the number of codewords with the minimum Hamming weight $d_{min} = 16$ is $A_{16} = 94488$ for the polar code $P(128, 64)$ constructed with RM-profile, whereas $A_{16} = 3120$ for the PAC code $PAC(128, 64)$ with the same rate profile. Furthermore, the second minimum distance for the polar code is 24 with $A_{24} = 4465024$ while for the PAC code we observe $A_{18} = 2696$, $A_{20} = 95828$, $A_{22} = 352311$ and $A_{24} = 3065194$. Note that the minimum Hamming distance for $PAC(128, 64)$ with PW [65] rate profile is $d_{min} = 8$ with $A_8 = 256$ and $A_{12} = 960$, hence the FER performance of PW-profile is inferior to RM-profile. Hence, PW-profile for PAC(128,64) is not considered.

From the truncated union bound of the block error probability under ML decoding, $P_e^{ML} \approx A_{d_{min}} Q(\sqrt{2 d_{min} R E_b / N_0})$ [78], we can conclude that given the same d_{min} and decoder, the code with smaller $A_{d_{min}}$ should perform better. In [71], the authors show that a properly designed upper-triangular pre-transformation matrix for polar codes can reduce $A_{d_{min}}$ of the concatenated code. Note that the convolutional pre-transform in PAC codes has an upper-triangular Toeplitz matrix.

6.3.3 Performance of List Decoding

The list decoding of PAC codes over binary-input additive white Gaussian noise (BIAWGN) channels with BPSK modulation is simulated. The constraint length and the coefficients of the generator polynomial for the convolutional code are 7 ($m = 6$) and 0o133, respectively. For PAC(128,64), the rate-profile is formed by the Reed-Muller (RM) construction [29] with dSNR=3.5. In the list decoding, different list sizes are employed and the performance is compared with the performance of the P(128,64) polar code and finite-length bound [76] as shown in Fig. 6.4. The performance of the RM-profile and the modified PW-profile are identical as the resulted rate-profiles are identical. A serial concatenation of CRC with relatively short codes such as PAC(128,64) does not improve the error correction performance due to a significant rate loss and negative impact on the distance properties (e.g. in the case of PAC(128,64), the minimum Hamming distance drops to $d_{min} = 8$). However, an 8-bit CRC with a generator polynomial with coefficients 0xA6 improves the performance of PAC(512,256) in the high SNR regime significantly as shown in Fig. 6.4. The notation CxA-SCL used in Fig. 6.4 is defined as CRC-aided SCL decoding with x-bit CRC and L in SCL(L) is the list size. The rate-profile for this code is formed by density evolution with Gaussian approximation (DEGA) [79] with dSNR=2. One can observe that as the block-length increases, the performance of PAC codes under list decoding cannot compete with that of polar codes under CRC-aided list decoding and we need to add CRC bits as the outer code to detect the correct path in the list decoding.

6.4 Limits of Convolutional Precoding

We observed in Section 6.2 that precoding in PAC codes can reduce $A_{d_{min}}$. It is difficult to systematically design a generator polynomial $\mathbf{c}$ that provides the minimum $A_{d_{min}}$. Nonetheless, we can design the precoding stage to mitigate the potential weakness or shortcoming of convolutional precoding. To do so, we look at the precoding as a protection means for information bits similar to the convolutional codes.

Let us first study the distribution of the elements of set $\mathcal{A}$ in the rate-profile. Although the rate-profiles can be constructed with different methods [80], here we consider the RM-Polar rate-profile [29] which performs better on short codes. In this rate-profile, the weight of the rows in $\mathbf{P}_n$, denoted by $w(\mathbf{g}_j)$ for row j, plays an important role. As the code rate increases, $d_{min} = 2^{w(j)}$, where $w(j)$ is the weight of binary expansion of j, increases. That leaves gaps

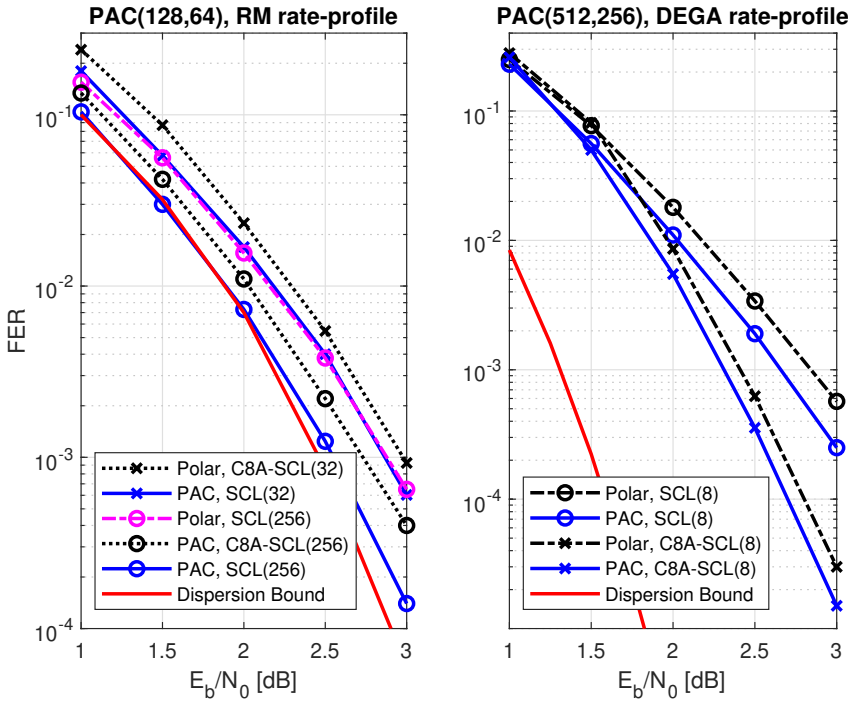

Fig. 6.4: Performance of PAC codes under list decoding

between the bits in the set $\mathcal{A}$ and excludes the rows with weights lower than d_{min}. Fig. 6.5 illustrates the gaps with white cells. When it comes to the convolution operation, these gaps makes the error protection of a subset of $\mathcal{A}$ weaker than the rest of the bits. Let us observe this weakness by an example. Consider bit $i = 38$ in PAC(64,32). Since $v_i = 0$ for any $i \in [32, 37]$, if the constraint length m is $m \leq 6$, then $u_{38} = \sum_{j=0}^{m} \mathbf{g}_j v_{i-j} = v_{38}$. As you may notice, no convolution is happening here. In fact, the effective generator polynomial for $i = 38$ and 39 is $\mathbf{c} = [0, ..., 0]$. As a result, the bit $i = 38$ which turns out to be transmitted over a relatively low-reliability sub-channel is left unprotected. Fig. 6.6 illustrates the case where $u_i = v_i$ as the shift-register is empty. Note that we do not face this issue in the convolutional codes as there is no prefixed zero values in the input sequence to the encoder.

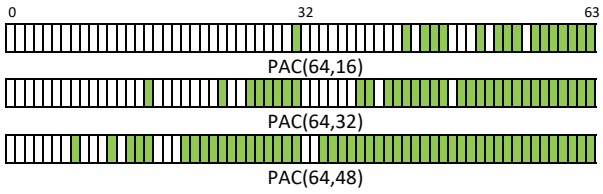

Fig. 6.5: RM-polar rate-profiles for block-length $N = 64$ and code rates $R = 1/4, 1/2, 3/4$. Green cells are in set $\mathcal{A}$.

This weakness may be mitigated by a longer constraint length and a proper generator polynomial $\mathbf{c}$ or by a different convolution scheme. The longer constraint length requires a longer memory size m for each path in the list decoding. A recommended long-memory polynomial is

$$\mathbf{c} = [\mathbf{c}_{(i)}|0, \ldots, 0|\mathbf{c}_{(ii)}] \tag{6.5}$$

where $\mathbf{c}_{(i)}$ and $\mathbf{c}_{(ii)}$ are the coefficients of two generator polynomials.

A smarter scheme that provides a longer memory without a large memory requirement, in particular under list decoding with large list size, is the scheme shown in Fig. 6.7. In this scheme, we add another shift register in parallel with the main shift-register, where we store a subset of input $\mathbf{v}$ stream, preferably the bits transmitted through low-reliability sub-channels. Note that the number of low-reliability bits in each segment is limited. Since the secondary shift-register has lower number of inputs, a subset of $\mathbf{v}$, the bits remains in the shift-register for a longer time-steps. This equivalent to having a longer memory.

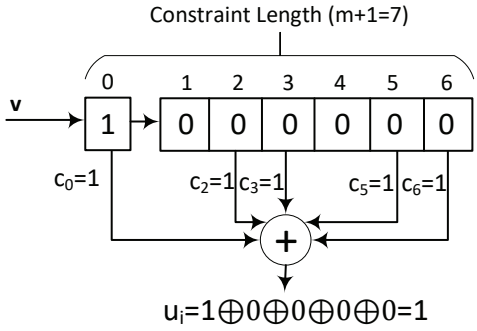

Fig. 6.6: An example of convolution in the presence of a zero sub-sequence in $\mathbf{v}$.

The proposed schemes result in a fewer number of min-weight codewords comparing with conventional PAC codes. Table 6.2 lists $A_{d_{min}}$ of some examples.

Lets us discuss the advantage of these example polynomials. Since $v_i = 0$ for any $i \in [16, 22]$ and this is the longest sub-sequence of zeros in the rate-profile, the constraint length $m + 1$ should be $m \geq 8$. The polynomial $\mathbf{c} = [1, 0, 1, 1, 0, 1, 1, 0, 1, 1]$ is an example that mitigates the unequal error protection resulting in a smaller $A_{d_{min}}$. A short polynomial such as $\mathbf{c} = [1, 0, 1, 1]$ results in a larger $A_{d_{min}}$ for the same reason. Intuitively, one can observe that this increase is due to less inclusion of rows of $\mathbf{P}_n$ corresponding to $v_i = 0$ in the row combinations as discussed earlier. We can also use a longer polynomial for $\mathbf{c}_{(i)}$ to improve it from UEP point of view. Lastly, the two polynomials $\mathbf{c}_{(a)} = [1, 0, 1, 1, 0, 1, 1]$ and $\mathbf{c}_{(b)} = [0, 0, 1, 0, 1]$ where $\mathbf{c}_{(b)}$ is used for

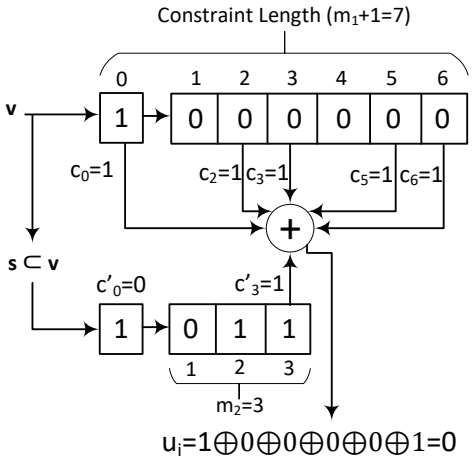

Fig. 6.7: A different scheme to mitigate the effect of unequal error protection with two generator polynomial $\mathbf{c}_{(a)} = [1,0,1,1,0,1,1]$ and $\mathbf{c}_{(b)} = [0,0,0,1]$.

a min-weight subset of indices in $\mathcal{A}$.

Fig. 6.8 illustrates the FER performance of some of the polynomials listed in Table 6.2 for PAC(128,64) with RM-polar rate-profile [29] and PAC(256,128) with density evolution/Gaussian approximation (DEGA 4dB) [80]. The number of codewords with weights d_{min} and $d_{min}+2$, i.e., A_{16} and A_{18}, for polar code P(256,128) are 60720 and 0, respectively, while these numbers for PAC(256,128) with $\mathbf{c} = [1,0,1,1,0,1,1]$ are 13424 and 1824, and for PAC(256,128) with $\mathbf{c}_{(a)} = [1,0,1,1,0,1,1] = \mathbf{c}_{(b)}$ are 12328 and 80, respectively. Note that $A_{d_{min}+2}$ contributes to the second term of P_e^{ML} (see [70]) which was approximated by (6.1), therefore a significant reduction in $A_{d_{min}+2}$ can improve the performance as well. As can be seen, the improvement using the new scheme(s) is nearly 0.1 dB at high SNRs which seems improving as the SNR increases. By concatenation of both polar and PAC codes of (256,128) with CRC, it is expected that the FER curves are shifted toward left, as it was shown in [48], however it is not the subject of this work.

6.5 Summary

This chapter investigated and analyzed the reason behind the reduction of min-weight codewords in PAC codes. We noticed the importance of rows of $\mathbf{G}$ with indices in set $\mathcal{A}^c$ in the weight distribution of the code. We also showed the limits of PAC codes and where the precoding stage is not effective depending on the code and set $\mathcal{A}$. Additionally, we investigated the

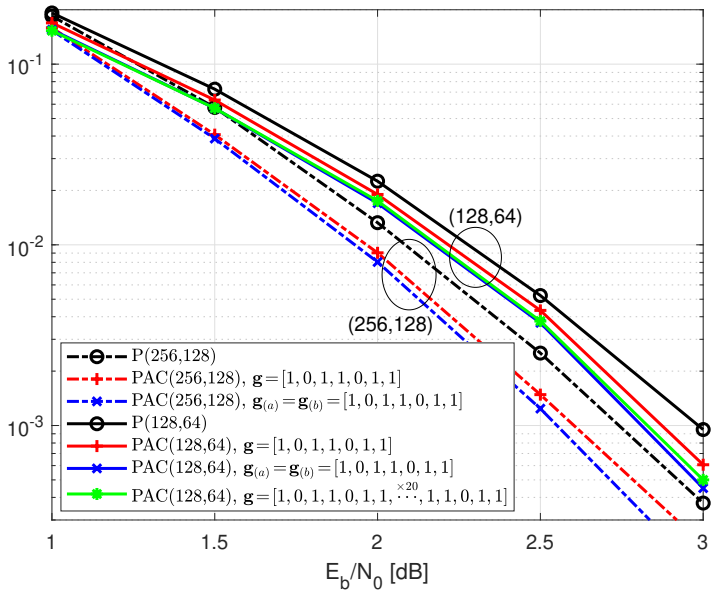

Fig. 6.8: Performance of polar codes and PAC codes with different precoding polynomials under list decoding with L=32. In the legends, $\mathbf{g}$ is equivalent to $\mathbf{c}$.

implementation of list decoding for PAC codes. Under list decoding, there is a significant performance gap between CRC-polar codes and PAC codes. However, this gap between polar and PAC codes reduces when employing another concatenation layer, such as CRC bits or parity check (PC) bits. Finally, we recognized the weakness of convolutional precoding and proposed approaches to mitigate it.

Algorithm 5: List Decoding of PAC codes

input : channel LLRs $\lambda_n^{0,N-1}$, $\mathcal{B}$, L, $\mathbf{c}$
output: recovered message bits $\hat{\mathbf{d}}$

$\mathcal{L} \leftarrow \{1\}$ `// a single path in the list`
$[\lambda, \beta] \leftarrow [\lambda_n^{0,N-1}+\{0\}, \{0\}]$
for $i \leftarrow 0$ **to** $N-1$ **do**
  if $i \notin \mathcal{B}$ **then**
    for $l \leftarrow 1$ **to** $|\mathcal{L}|$ **do**
      $\lambda_0^i[l] \leftarrow$ updateLLRs(l, i, $\lambda[l]$, $\beta[l]$)
      $\hat{v}_i[l] \leftarrow 0$
      $[\hat{u}_i[l]$, cState$[l]] \leftarrow$ conv1bTrans(v_i, cState$[l]$, $\mathbf{c}$)
      $PM_l^{(i)} \leftarrow$ calcPM($PM_l^{(i-1)}$, $\lambda_0^i[l]$, $\hat{u}_i[l]$)
      $\beta[l] \leftarrow$ updatePartialSums($\hat{u}_i[l]$, $\beta[l]$)
  else
    for $l \leftarrow 1$ **to** $|\mathcal{L}|$ **do**
      $\mathcal{L} \leftarrow$ duplicatePath($\mathcal{L}$, l, i, $\mathbf{c}$)
    if $|\mathcal{L}| > L$ **then**
      $\mathcal{L} \leftarrow$ prunePaths($\mathcal{L}$) `// like SCLD`
$\hat{\mathbf{d}} \leftarrow$ extractData($\hat{v}_1^N[0]$)
return $\hat{\mathbf{d}}$;
subroutine `duplicatePath(`$\mathcal{L}$, l, i, $\mathbf{c}$`):`
  $\mathcal{L} \leftarrow \mathcal{L} \cup \{l'\}$ `//` path l' `is a copy of path` l
  $\lambda_0^i[l] \leftarrow$ updateLLRs(l, i, $\lambda[l]$, $\beta[l]$)
  $(\hat{v}_i[l], \hat{v}_i[l']) \leftarrow (0, 1)$
  $[\hat{u}_i[l]$, cState$[l]] \leftarrow$ conv1bTrans($\hat{v}_i[l]$, cState$[l]$, $\mathbf{c}$)
  $[\hat{u}_i[l']$, cState$[l']] \leftarrow$ conv1bTrans($\hat{v}_i[l']$, cState$[l]$, $\mathbf{c}$)
  $PM_l^{(i)} \leftarrow$ calcPM($PM_l^{(i-1)}$, $\lambda_0^i[l]$, $\hat{u}_i[l]$)
  $PM_{l'}^{(i)} \leftarrow$ calcPM($PM_l^{(i-1)}$, $\lambda_0^i[l]$, $\hat{u}_i[l']$)
  $\beta[l] \leftarrow$ updatePartialSums($\hat{u}_i[l]$, $\beta[l]$)
  $\beta[l'] \leftarrow$ updatePartialSums($\hat{u}_i[l']$, $\beta[l]$)
  return $\mathcal{L}$;
subroutine `calcPM(`PM, λ_0, $\hat{u}$`):`
  if $\hat{u} = \frac{1}{2}(1 - \text{sgn}(\lambda_0))$ **then**
    $PM = PM$
  else
    $PM = PM + |\lambda_0|$
  return PM;

Table 6.2: The number of min-weight codewords, $A_{d_{min}}$, with RM rate profile for PAC code (128,64,16) under various precoding schemes. The polynomial $\mathbf{c} = [1]$ is equivalent to no precoding, hence the output of encoder is a polar code.

Polynomial	$\approx A_{16}$
$\mathbf{c} = [1]$	94488
$\mathbf{c} = [1, 0, 1, 1]$	7520
$\mathbf{c} = [1, 0, 1, 1, 0, 1, 1]$	3120
$\mathbf{c} = [1, 0, 1, 1, 0, 1, 1, 0, 1, 1]$	2812
$\mathbf{c} = [1, 0, 1, 1, 0, 1, 1, 0 \overset{\times 20}{\cdots\cdots}, 1, 1, 0, 1, 1]$	2556
$\mathbf{c}_{(a)} = [1, 0, 1, 1, 0, 1, 1]$ & $\mathbf{c}_{(b)} = [0, 0, 1, 1, 0, 1, 1]$	2574

Chapter 7

Sequential Decoding of PAC Codes

> *"It is better to solve one problem five different ways, than to solve five problems one way."*
>
> — George Pólya

Fano decoding is an efficient algorithm in terms of memory requirements and computation resources, and it has shown a good error correction performance. Nevertheless, Fano decoding has high average time complexity. The motivation of this work is to reduce the time and the computational complexity at the cost of a minor degradation in the practical range of frame error rate (FER), i.e., 10^{-2} to 10^{-4}. In this chapter, sequential decoding (including Fano decoding and stack decoding) is first adapted to decode PAC codes. Then, to reduce the complexity of sequential decoding of PAC/polar codes, we propose (i) an adaptive heuristic metric, (ii) tree search constraints for backtracking to avoid exploration of unlikely sub-paths, and (iii) tree search strategies consistent with the pattern of error occurrence in polar codes. These contribute to the reduction of the average decoding time complexity from 50% to 80%, trading with 0.05 to 0.3 dB degradation in error correction performance within FER=10^{-3} range, respectively, relative to not applying the corresponding search strategies. Additionally, as an essential ingredient in memory-efficient Fano decoding of PAC/polar codes, an efficient computation method for the intermediate LLRs and partial sums is provided. This method is effective in backtracking and avoids storing the intermediate information or restarting the decoding process. Eventually, the sequential decoding algorithms are compared with list decoding (discussed in Chapter 6) in terms of performance, complexity, and resource requirements.

7.1 Convolutional Codes and Fano Decoding

Convolutional codes (CCs) are a class of linear codes described by a tuple (n_0, k, m), where k is the number of information bits shifted into the encoder at each time slot (usually $k = 1$), n_0 is the number of corresponding outputted coded bits, and m is the number of previous input bits stored in a shift-register (a.k.a. *memory size*) [78]. Unlike the 1-to-1 convolutional transform in PAC codes where $n_0 = k = 1$ as illustrated in Section 6.1, the code rate of convolutional codes is given by k/n_0. The value $m + 1$, named *constraint length*, determines the number of

previous input bits plus the current bit that influence each coded bit. A larger constraint length generally provides greater resilience to bit errors.

The relation between the input bits $d_{i-m,i}$ and one of the n_0 output bits x_i, at time-step i, is obtained as a binary convolution $x_i = \sum_{j=0}^{m} g_j d_{i-j}$, where $g_i \in \{0, 1\}$. By representing bit sequences as polynomials in the delay variable D representing a time-step in the encoder, an output sequence $x(D)$ is obtained as $g(D)d(D)$, where $g(D) = \sum_{j=0}^{m} g_j D^j$ is the *generator polynomial*. Different generator polynomials are used for n_0 outputs, only one polynomial is employed in the pre-transformation of PAC codes.

Convolutional codes are decoded using the trellis-based Viterbi algorithm and the tree search sequential decoding algorithms. The Viterbi algorithm is a maximum likelihood decoding method that examines the entire state space of the encoder at each step. We have studied Viterbi decoding of PAC codes in [61].

On the other hand, the complexity of the sequential decoding is essentially independent of the memory of the encoder, since only one encoder state is examined at each step. The fundamental idea behind sequential decoding is to explore only the most promising path(s). If a path to a node looks "bad" we can discard all the paths through this node without a significant loss in the error correction performance compared to that of a maximum likelihood decoder [78].

In this work, we focus on Fano decoding which is a memory-efficient type of Sequential decoding algorithm. The *Fano algorithm* is a depth-first tree search, in which the decoder moves from a node either back to its parent node or to one of its children. The Fano decoder can visit a node only if its Fano path metric μ_F is larger than or equal to a certain value called threshold T. Threshold takes only discrete values $0, \pm\Delta, \pm 2\Delta, \ldots$.

Comparing the above described Fano decoding to the SC and SC list decoding described in Chapter 2, it is instructive to note two important differences: The SC decoding makes decisions to choose the node to visit at each step based on the branch metric. Thus, only one path is explored and the rest are discarded. However, SC list decoding explores multiple paths, but in contrast to Fano decoding, all of them have the same length. Thus, no backtracking is performed neither in SC nor in SC list decoding. Hence, only the path metric used to measure the likelihood of the paths in the sequential decoding must consider the difference in the lengths of partial paths by adding a bias, while in the SC and SC list decoding, the bias term is not required. A simpler sequential decoding algorithm is the *stack decoding* [78] where

the algorithm keeps a stack of size/depth D of partial paths sorted with respect to the path metric. The algorithm extends the path with the best metric at the top of the stack. The stack decoding is a memory intensive algorithm with a variable time complexity that instead of backtracking as in the Fano decoding, it selects to extend the best partial path in the stack at each time step.

The metric used in the sequential decoding of convolutional codes is a probabilistic path metric. We consider the set $\mathcal{X} = \{\mathbf{a}^{(1)}, \mathbf{a}^{(2)}, ..., \mathbf{a}^{(M)}\}$ of M partial sequences, representing partially explored paths with different lengths, to be compared. Let $n_{max} = \max\{n_1, n_2, ..., n_M\}$ denote the length of the longest sequence, and $\tilde{\mathbf{r}}$ the partial received sequence of length n_{max} symbols where each encoded symbol takes n bits corresponding to k information/uncoded bits, $R = k/n$. Hence, the sequences $\tilde{\mathbf{r}}$ and $\mathbf{a}^{(\ell)}$ are

$$\tilde{\mathbf{r}} = (\mathbf{r}_0 \mathbf{r}_1 \dots \mathbf{r}_{n_{max}-1}) = (r_0 \dots r_{n-1} \dots r_{nn_{max}-1})$$

$$\mathbf{a}^{(\ell)} = (\mathbf{a}_0^{(\ell)} \mathbf{a}_1^{(\ell)} \dots \mathbf{a}_{n_\ell-1}^{(\ell)}) = (a_0^{(\ell)} \dots a_{n-1}^{(\ell)} \dots a_{nn_\ell-1}^{(\ell)})$$

Among the sequences in $\mathcal{X}$, we choose the partial sequence $\mathbf{a}^{(\ell)}$ that maximizes the a-posterior probability $P(\mathbf{a}^{(\ell)}|\tilde{\mathbf{r}})$. According to Bayes' rule

$$P(\mathbf{a}^{(\ell)}|\tilde{\mathbf{r}}) = \frac{P(\mathbf{a}^{(\ell)})P(\tilde{\mathbf{r}}|\mathbf{a}^{(\ell)})}{P(\tilde{\mathbf{r}})} \tag{7.1}$$

Assuming the channels are memoryless, since the length of the sequence $\mathbf{a}^{(\ell)}$ is $n_\ell \leq n_{max}$, and there is no associated symbols in this sequence for $r_{n_\ell}, ..., r_{n_{max}-1}$, then we have

$$P(\tilde{\mathbf{r}}|\mathbf{a}^{(\ell)}) = \prod_{j=0}^{n_\ell-1} P(\mathbf{r}_j|\mathbf{a}_j^{(\ell)}) \prod_{j=n_\ell}^{n_{max}-1} P(\mathbf{r}_j) \tag{7.2}$$

Also, we can rewrite the denominator of (7.1) as

$$P(\tilde{\mathbf{r}}) = \prod_{j=0}^{n_\ell-1} P(\mathbf{r}_j) \prod_{j=n_\ell}^{n_{max}-1} P(\mathbf{r}_j) \tag{7.3}$$

Now, by substituting (7.2) and (7.3) in (7.1) and cancelling the common term $\prod_{j=n_\ell}^{n_{max}-1} P(\mathbf{r}_j)$, we have

$$P(\mathbf{a}^{\ell}|\tilde{\mathbf{r}}) = P(\mathbf{a}^{\ell}) \prod_{i=0}^{nn_\ell-1} \frac{P(r_i|a_i^{(\ell)})}{P(r_i)} \tag{7.4}$$

Suppose each encoded bit occurs with equal probability, then each sequence $\mathbf{a}^{(\ell)}$ occurs with probability $P(\mathbf{a}^{(\ell)}) = (2^{-k})^{n_\ell} = (2^{-nR})^{n_\ell} = (2^{-R})^{nn_\ell}$. Thus, by taking the base-2 logarithm of (7.5), we have

$$\log P(\mathbf{a}^{(\ell)}|\tilde{\mathbf{r}}) = \sum_{i=0}^{nn_\ell-1} \Big(\underbrace{\log P(r_i|a_i^{(\ell)})}_{\text{ML-metric}} \underbrace{- \log P(r_i) - R}_{\text{path-length bias}} \Big). \tag{7.5}$$

In order to adapt (7.5) for sequential (stack or Fano) decoding of polar/PAC codes, the path metric of list decoding can be used as the ML-metric term. Note that n in (7.5) is 1 for PAC codes as the convolutional transform is 1-to-1 resulting in $R = 1$. The simplest path-length bias in (7.5) could be a fixed bias parameter as suggested in [81]. A different bias function based on the cumulative density function (CDF) of the evolving LLRs was proposed in [79]. Further, [82] suggested to replace the path-length bias term with $\log(1 - p_{e,i})$, where $p_{e,i}$ is the error probability of i-th bit-channel.

Alternatively, in the computer science literature, the path metric of algorithm A, a graph traversal and path search algorithm, is written in the general form of [83]

$$f(\mathbf{a}^{(\ell)}) = g(\mathbf{a}^{(\ell)}) + h(\mathbf{a}^{(\ell)}) \tag{7.6}$$

where the first term measures the actual cost of the i-th partial path as follows,

$$g(\mathbf{a}^{(\ell)}) = \sum_{j=0}^{n_\ell-1} \log P(\mathbf{r}_j|\mathbf{a}_j^{(\ell)}) \tag{7.7}$$

and the second term is a heuristic estimate for the remaining cost of completing the path to its leaf with the best metric by following the corresponding (yet unknown) extension of $\mathbf{a}^{(\ell)}$. The choice of the heuristic function $h(\mathbf{a}^{(\ell)})$, determines the tradeoff between the complexity and the risk of accidentally abandoning a path that leads to the desired optimal solution.

We propose a heuristic to estimate $h(\mathbf{a}^{(\ell)})$ for Fano decoding of polar codes and PAC codes in Section 7.2.2.

7.2 Fano Decoding of PAC Codes

In this section, we first briefly explain the fundamentals of the Fano algorithm detailed in Algorithm 7 followed by the details of the proposed algorithm for updating the intermediate

information required in the SC decoding process in the backtracking (Section 7.2.1) and our novel path metric (Section 7.2.2). Then, we present several improvements to the Fano algorithm in order to reduce the time complexity in the following Section 7.3.

In the Fano algorithm, the decoder starts with the origin node ($i = 0$) and examines a sequence of adjacent nodes. At any step corresponding to the non-frozen bits in vector $\mathbf{v}$, it either moves forward to one of the successor nodes or moves backward to the non-frozen predecessor of the current node. The branch metric m_i is correspondingly added to the current path metric μ_{i-1} during forward movement (in lines 9 & 16-17 of Algorithm 7) or restored from memory during the backward movement. The algorithm stops when it reaches a terminal node ($i = N$). The search through the code tree is guided by a threshold T on the path metric (with initial value of $T = 0$). If the metric becomes less than the threshold as the algorithm follows the current path (line 42 of Algorithm 7), the search is backed up and another path is followed (Algorithm 9 is called in line 58 of Algorithm 7). If no paths can be found with a metric above the threshold, the threshold value is lowered (in line 26 of Algorithm 7) and the process is continued. A node in the tree may be visited more than once in the forward direction but a lower threshold each time. The algorithm eventually reaches a terminal node and stops. For more details on the Fano algorithm, see [78].

Note that (*i*) the Fano algorithm proposed here stores the path metric of good branch (the one with larger metric) and bad branch (the one with smaller metric) as well as memory states along the current path, (*ii*) the subroutines *updateLLRs* and *updatePartialSums* in Algorithm 7 and the rest of the chapter are identical to the ones used in SC decoding of polar codes, (*iii*) Algorithm 9 is called in line 44 to find a bit index that satisfies the threshold in order to move back, and (*iv*) *toDiverge* indicates that the branch with smaller metric should be chosen at information bit j and this choice is flagged in the j-element of vector δ. The rest of the Algorithm 7 and other algorithms are explained and referred to in the rest of the chapter.

7.2.1 Partial Rewind of SC Algorithm

The Fano algorithm performs forward and backward traversals in the decoding tree: while in the forward traversal, the calculation of the required intermediate LLRs and partial sums is straightforward and linear, a more sophisticated approach is required for the backward traversal or partial rewind of SC algorithm. Suppose that we want to move back from the i_{curr}-th bit to the i_{start}-th bit: First, we need to re-calculate $\lambda_0^{i_{start}}$ and a number of intermediate

LLRs. Since these LLRs are updated in-place when computing the metrics in natural order, they may no longer be available. As explained in [66], efficient decoders store at most $N-1$ intermediate/decision LLRs for decoding bits 0 to $N-1$, of which $N/2^{n-s}$ are associated to stage s ($0 \leq s \leq n-1$) of the LLR calculation algorithm. The number of intermediate LLRs to be updated varies between one, when moving from a bit with odd index i_{curr} to $i_{curr}-1$, and in extreme cases $N-1$, when moving from $i_{curr} \geq N/2$ to $i_{start} < N/2$.

In general, up to $\log_2 N$ stages should be activated to calculate the decision LLR at bit i_{start}, $\lambda_0^{i_{start}}$. The first stage to be activated (from right to left in Fig. 2.2) is determined by *find first set* (ffs) operation, here, *set* means 1, on the binary representation of bit index x, i.e., $\text{bin}(x) = x_{n-1}...x_1x_0$. The modified version of ffs [66] is defined below. Note that we assume the decoding is performed in natural order.

$$\text{ffs}^*(x_{n-1}...x_1x_0) = \begin{cases} \min(j) : x_j = 1 & x > 0, \\ n-1 & x = 0 \end{cases} \tag{7.8}$$

When i_{curr}, the index of the current bit, is odd, $\text{ffs}^*(\text{bin}(i_{curr})) = 0$, and $i_{start} = i_{curr} - 1$, we can calculate the decision LLR, $\lambda_0^{i_{start}}$, directly according to the f-node operation without any need to update the intermediate LLRs. As a consequence, when moving back to bit index $i_{start} < i_{curr} - 1$, we need to consider the ffs* of $i_{curr} - 1$ and/or $i_{start} - 1$ if i_{curr} and i_{start} both or either one is odd. This is controlled in lines 1-4 of Algorithm 6. Note that the stages to be updated are not necessarily $s = \text{ffs}^*(\text{bin}(i_{start})), ..., 1, 0$, but the deepest stage to be updated, s_{max}, is

$$s_{max} = \{\max(s) : s = \text{ffs}^*(\text{bin}(i_m)), i_{start} \leq i_m \leq i_{curr}\} \tag{7.9}$$

The relation (7.9) finds the deepest stage in the factor graph (from left to right in Fig. 2.2) at which the LLRs have been updated/overwritten while decoding bit i_{start} to i_{curr}. If $s_{max} \geq \text{ffs}^*(\text{bin}(i_{start}))$, we need to move back further to the bit i_{-1} at which $s_{max} = \text{ffs}^*(\text{bin}(i_{-1}))$. The subroutine $findsMaxPos$ in Algorithm 6 performs the operation of finding i_{-1}.

Example: Suppose the block-length is $N = 4$ and we are decoding bit $i_{curr} = 3$. The intermediate LLRs vector is $[\lambda_1^3, \lambda_1^2]$, excluding the decision LLR, λ_0^3 (see Fig. 2.2). Now, if we need to go one step back to bit $i_{start} = 2$, since $i_{curr} = 3$ is odd, we do not need to update the intermediate LLR vector, i.e., λ_0^2 can be directly calculated. However, for moving back to $i_{start} = 1$, since i_{start} is odd, we need to find $s_{max} = 1$ and calculate $[\lambda_1^1, \lambda_1^0]$. Only after this

update of the intermediate LLRs it is possible to calculate the decision LLR λ_0^1 and rewind the SC algorithm.

Note that the partial sums vector, β, is also updated in lines 12 and 19 during the aforementioned process.

Algorithm 6 shows an efficient approach for updating the intermediate LLRs.

Algorithm 6: Partial Rewinding: updateLLRsPSs - Updating intermediate LLRs & partial sums

```
   input  : i_start, i_curr, û, λ, β
   output: updated λ, updated β
 1 if i_curr%2 ≠ 0 then
 2 |   i_curr ← i_curr − 1
 3 if i_start%2 ≠ 0 then
 4 |   i_start ← i_start − 1
 5 s_start = ffs*(i_start)                                  // c.f (7.8)
 6 s_max ← sMax(i_start, i_curr)                            // c.f (7.9)
 7 if s_start ≤ s_max then
 8 |   i_−1 =find_sMaxPos(s_start,s_max, i_start)
 9 |   β ← updatePSBack(i_−1, s_max, û)
10 |   for i ← i_−1 to i_start do
11 |   |   λ ← updateLLRs(i, λ, β)
12 |   |   β ← updatePartialSums(i, û_i, , β)               // Identical w/ SCD
13 else
14 |   λ ← updateLLRs(i_start, λ, β)
15 return [λ, β];
16 subroutine updatePSBack(i_−1, s_max, û):
17 |   k ← 2^s_max
18 |   for i ← i_−1 + 1 − k to i_−1 do
19 |   |   β ← updatePartialSums(i, û_i, β)
20 |   return β;
21 subroutine find_sMaxPos(s_start, s_max, i_−1):
22 |   s' ← s_start
23 |   while s' < s_max do
24 |   |   i_−1 ← i_−1 − 2
25 |   |   if i_−1 > 0 then
26 |   |   |   s' ← ffs*(i_−1)                              // c.f (7.8)
27 |   |   else
28 |   |   |   s' ← n
29 |   return i_−1;
```

7.2.2 Heuristic Path Metric

The Fano path metric for each examined node plays an important role in the backtracking since it provides an indication for how likely it is that the partial path to the current node is correct. Efficient backtracking relies on this metric to a) select a point to branch off the currently best (possibly erroneous) path to explore promising alternative solutions and to b) abandon unlikely paths based on comparing their path metrics with the threshold T.

To provide such a metric, we follow the generic approach outlined in (6): the first term corresponds to the metric in list decoding while the second term is used to account for the different candidate path lengths in the Fano decoding. For every partial sequence $a^{(\ell)}$, we define the following metric:

$$\begin{aligned}\mu_\ell = M(a^{(\ell)}, \mathbf{y}) = &\sum_{j=0}^{n_\ell - 1} \log P(\hat{u}_j^{(\ell)} | \hat{\mathbf{u}}_{0,j-1}^{(\ell)}, \mathbf{y}) \\ &+ \sum_{j=n_\ell}^{N-1} \log E_{\mathbf{y}}\big[P(u_j | \mathbf{u}_{0,j-1}, \mathbf{y})\big]\end{aligned} \tag{7.10}$$

The second term is an expected metric for the continuation of the partial path with length $N - n_i$. Based on our observation of the actual metric obtained during decoding with or without backtracking, a good estimation of the second term, in case there is no error in the received signals, is $E_{\mathbf{y}}\big[P(u_j|\mathbf{u}_{0,j-1}, \mathbf{y})\big] \approx 1 - p_{e,j}$, where p_e is the error probability of the bit-channels which can be obtained from the methods used for the construction/rate-profile of polar codes.

Let us define the *expected metric* $B = E_{\mathbf{y}}[\mu_{N-1}]$ for the full-length path and the expected metric of the remaining partial path as

$$B = \sum_{j=0}^{N-1} \log(1 - p_{e,j}) \tag{7.11}$$

$$B_i^c = \sum_{j=i+1}^{N-1} \log(1 - p_{e,j}) = B - \sum_{j=0}^{i} \log(1 - p_{e,j}) \tag{7.12}$$

where $\log(1 - p_{e,j})$ is the *estimated branch metric*. Now, we can rewrite (7.10) as a recursion as follows:

$$\mu_j = \mu_{j-1} + m_j - \log(1 - p_{e,j}) \tag{7.13}$$

where $m_j = \log(P(\hat{u}_j|\hat{\mathbf{u}}_{0,j-1}, \mathbf{y}))$ is the actual branch metric and $\mu_{-1} = B$. Note that since the initial metric is $\mu_{-1} = B$, at each decoding step, the actual branch metric m_j is added and instead the estimated metric of the corresponding branch is deducted to maintain the relation in (7.10). Hence, although (7.13) looks similar to the metric in [82], the initial value and the foundation of the metric are quite different (in [82], $\mu_{-1} = 0$). Furthermore, one can optimize the FER performance by tuning the bias term, $\log(1 - p_{e,i})$. In particular, if the SNR dependent method in [79] is used to obtain $p_{e,i}$, one can improve FER performance, by changing the design-SNR.

7.3 Low-complexity Fano Decoding

In this section, we introduce an adaptive path metric depending on the noise level and different search strategies to limit the search space.

7.3.1 Adaptive Path Metric

A bit channel i with low reliability contributes to the metric update depending on the noise level, i.e., μ_i can be significantly smaller than μ_{i-1} (due to change in the magnitude and/or sign of the decision LLR) in the presence of large channel noise. This impact on the path metric can accumulate over time leading to a significant deviation from the expected metric in (7.11). Recall that due to channel dependency, a change in the channel LLR of one channel can affect the other low-reliability bit channels as well. Consequently, the metric of most of the examined branches denoted by μ' in Fig. 7.1 are most likely greater than the threshold, i.e., $\mu'_i > T$ for $i < i_{curr}$, where i_{curr} is defined in Section 7.2. This causes a large delay due to the exploration of many sub-paths during backtracking. Hence, the metric estimate for the path continuation represented by the second term in (7.10), is not in a fair way comparable with the actual metric of the current path as discussed in the previous section.

To compensate for such deviation, we suggest adapting the estimate (7.12) for the continuation of partial paths relative to the impact of the channel noise on the actual metric. This adaptation can be realized by a scaling factor α for the logarithm of the probability in

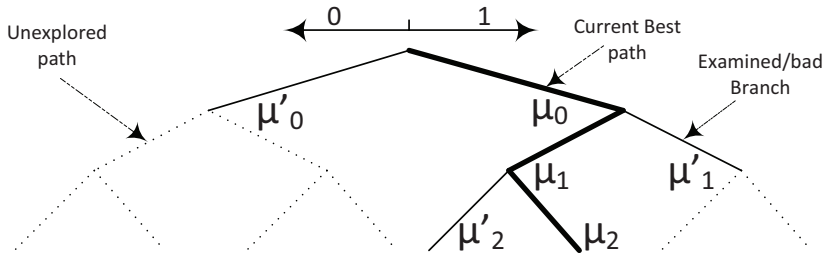

Fig. 7.1: Decoding tree: μ_js are the path metrics of the current best path (solid thick line) from the root to a node at level j and the μ'_js are the path metrics of the branches (solid thin line) diverging from the current best path.

(7.12) which in effect adapts the expected probability to the noise level. The effect of this scaling is as follows: $\alpha \log E_{\mathbf{y}}\big[P(u_j|\mathbf{u}_{0,j-1},\mathbf{y})\big] = \log\big(E_{\mathbf{y}}\big[P(u_j|\mathbf{u}_{0,j-1},\mathbf{y})\big]\big)^{\alpha}$. Since $\alpha \geq 1$ and $P(u_j|\mathbf{u}_{0,j-1},\mathbf{y}) < 1$, then $\big(E_{\mathbf{y}}\big[P(u_j|\mathbf{u}_{0,j-1},\mathbf{y})\big]\big)^{\alpha}$ becomes smaller, accounting for a larger noise variance.

The value of α is determined after visiting the nodes of the current path to some level of decoding tree. This level should cover a sufficient number of low-reliability bit-channels to reflect the noise effect on the metric in a fair way. Until this level/bit index denoted by i_{bu} in lines 43 and 46 of Algorithm 8, we do not perform backtracking although the metric drops below the threshold, T (as seen in lines 43-44 where the threshold is updated). Then, the scaling factor is obtained by

$$\alpha = \frac{\sum_{j=0}^{n_k} \log P(\hat{u}_j^{(\ell)}|\hat{\mathbf{u}}_{0,j-1}^{(\ell)},\mathbf{y})}{\sum_{j=0}^{n_k} \log E_{\mathbf{y}}\big[P(u_j|\mathbf{u}_{0,j-1},\mathbf{y})\big]} \tag{7.14}$$

This adaptation can be performed when $\alpha > 1$, i.e., when the actual metric is larger than the expected metric. In practice, a quantized version of this factor is more convenient to use in fixed-point arithmetic. Hence, $\alpha_q = \lceil \frac{\alpha}{\Delta_q} \rceil \Delta_q$, where Δ_q is an integer. For instance, in decoding $PAC(128, 64)$, we first follow the current best path to bit $i_{bu} = 38$. By taking $\Delta_q = 2$ and the effect of the ceiling operator, an effective value is obtained which further reduces the complexity with almost no degradation in performance. In low and medium code rates, one can choose to calculate α after the initial sequence of low-reliability bits, where the associated values in vector v are 0 (equivalent to the frozen bit-channels in polar codes).

After obtaining α, we need to update not only the metric of the current path, but also the metric of the examined branches, μ'_j in Fig. 7.1, along the current path.

To update the computed metrics we simply add the difference between the updated bias αB_j^c and the initial bias B_j^c to μ_j and μ'_j.

$$\mu'_j = \mu'_j + (\alpha - 1)B_j^c \tag{7.15}$$

Thus, the metrics are computed by considering α in the next decoding steps as

$$\mu_j = \mu_{j-1} + m_j - \alpha \cdot \log(1 - p_{e,j}) \tag{7.16}$$

Lines 10 and 16-17 of Algorithm 7 include α which is initialized at the beginning of the decoding, line 3 ($\alpha = 1$). The calculation of α and the metric updating process are shown in Algorithm 8, lines 46-53.

For hardware implementation, we are interested in simple arithmetic operations. Here, we suggest using an LLR-based metric instead of the metric based on the probability. To this end, we need to define m_j based on λ_0^j.

$$\begin{aligned} m_j(\lambda_0^j, \hat{u}_j) &= \log(P(\hat{u}_j|\hat{\mathbf{u}}_{0,j-1}, \mathbf{y})) = \log\left(\frac{e^{(1-\hat{u}_j)\lambda_0^j}}{e^{\lambda_0^j}+1}\right) \\ &= \log\left(1 + e^{-(1-2\hat{u}_j)\lambda_0^j}\right)^{-1} \end{aligned} \tag{7.17}$$

where the last equality holds only for $\hat{u}_j = 0$ and 1. Now, if $\hat{u} = \frac{1}{2}(1 - \text{sgn}(\lambda_0^j))$, the term $e^{-(1-2\hat{u})\lambda_0^j} = e^{-|\lambda_0^j|}$ is small and hence $\log(1 + e^{-|\lambda_0^j|}) \approx 0$. Otherwise, we can approximate $\log(1 + e^{|\lambda_0^j|}) \approx |\lambda_0^j|$. The term $\log(1 - p_{e,j})$ and $B = \sum_{j=0}^{N-1} \log(1 - p_{e,j})$ can be pre-computed offline and can be used in the metric computation.

Note that all the terms in (7.16) are negative and so are the metric values. To save one bit per metric in the storage, we can discard the bit representing the always negative sign from the values. In this case we need to modify the comparisons in the algorithms accordingly.

7.3.2 Constrained Tree Search

The tree search algorithm may explore the paths on the tree that are unlikely to be correct. Unfortunately, the threshold T can only be used to prune a subset of these paths since a too tight threshold would also be likely to prune the correct path. Prior knowledge about error occurrence can be exploited in order to constrain the tree traversal. In the following, we propose

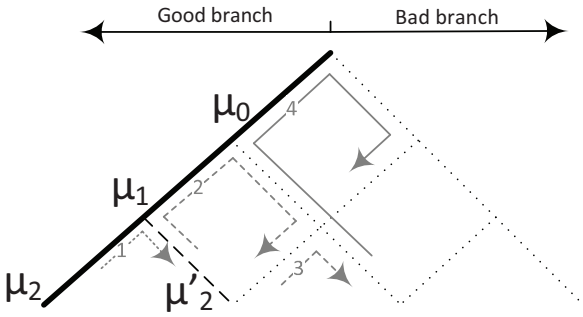

Fig. 7.2: Bottom-up backtracking

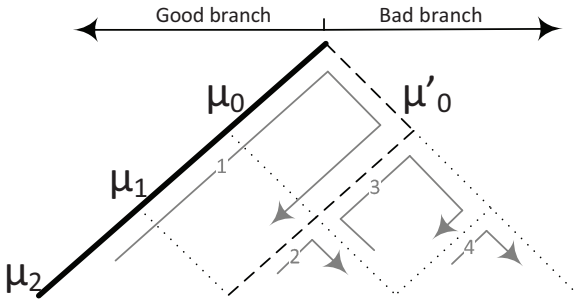

Fig. 7.3: Top-down backtracking

several effective constraints resulting in a significant reduction in time complexity at a small performance degradation:

Constraint on Number of Diversions from Best Path

By using a genie that corrects the error occurrence due to channel noise, we can observe that less than 1% of the frame errors are due to more than $b = 5$ bit-errors caused by the channel noise. Fig. 7.4 shows the relative frequency of error occurrence for different numbers of bit-errors. With this knowledge, we can avoid exploring the paths that diverge from the SC path at more than 5 bit-positions. If we can afford a degradation of error correction performance, we can reduce the maximum number of diversions while exploring the alternative paths. This would limit the number of visited nodes. For the example shown in Fig. 7.4, this number can be set to $b = 3$ or 4 bit-positions in order to reduce the number of visited node and consequently the time complexity. We will show a result after applying this constraint in Section 3.2.2. In algorithm 9, lines 21-22 implement the constraint for the maximum diversions.

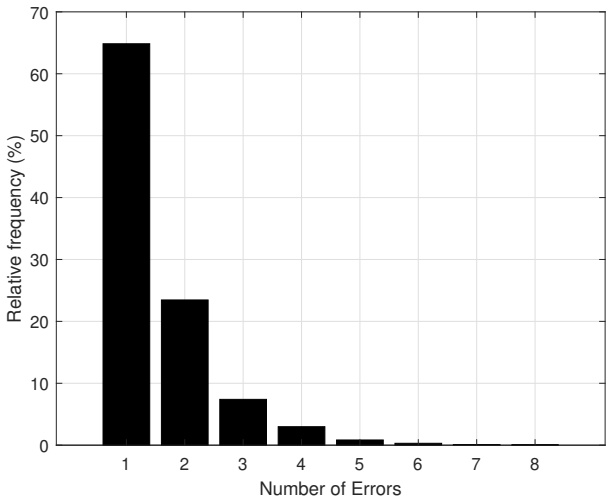

Fig. 7.4: Distribution (in %) of the number of error occurrence, extracted from 4000 decoding failures of PAC(128,64) with RM-profile at $E_b/N_0 = 2.5$ dB

Exploring a Subset of bad branches

The reliability of the bit-channels is known from methods such as density evolution [79]. Hence, during backtracking, we do not need to extend the partial path to the bad branches connecting to the nodes representing high-reliability bit-channels even if they satisfy the threshold condition. Thus, we only explore the sub-paths that originate from bad branches of the low-reliability bit-channels. This might introduce a small error rate degradation (due to not exploring all the bad branches), but it reduces the time complexity significantly. To this end, we collect the indices of the low-reliability bit-channels in the critical set $\mathcal{CS}$ [32, 84] and in the backtracking procedure, we only compare the threshold with the metrics of bad branches that are listed in the critical set. Lines 6 and 20 in Algorithm 9 enforce this constraint in top-down and bottom-up schemes (discussed in the next section), respectively.

Additionally, the constraint can be set to stop decoding and declaring decoding failure when the number of steps or clock iterations exceeds some limit or the path metric drops below a certain value. This could avoid cases with excessive run-time due to visiting a huge number of nodes. Also, we can stop decoding when the path metric drops below a certain value, since in this case, the decoder either fails correcting the error(s) or it may lead to a long decoding delay due to visiting a huge number of nodes in order to find the correct path.

7.3.3 Direction of Backtracking Traversal

Considering the properties of PAC codes which are mainly inherited from polar codes, we can devise different strategies that help to reduce the total number of nodes to visit during backtracking. When a decision error occurs during forward tree traversal, this error is propagated to the subsequent bits due to the sequential nature of decoding. In the conventional Fano decoding, backtracking starts from the latest decoded bit in a depth-first bottom-up direction, step by step as shown in Fig. 7.2. For example, in a code with 3 bits, in the first backtracking iteration shown by 1 in Fig. 7.2, the 3rd bit diverges from the SC path, i.e., $u_0 - u_1 - \bar{u}_2$. In the 2nd backtracking iteration, the 2nd bit diverges only, i.e., $u_0 - \bar{u}_1 - u_2$. Then the 2nd and 3rd bits diverge together, i.e., $u_0 - \bar{u}_1 - \bar{u}_2$. This process continues towards the top of the tree until (in the worst case) all the combinations of 1-bit, 2-bit, and 3-bit diversions are explored, assuming the threshold condition is satisfied by all the branches. However, as our observations show, the probability that the first error due to channel noise has occurred at one of the first decoded bits is higher. Further, there is no point in correcting the error that occurred due to error propagation. Thus, backtracking in a top-down fashion as shown in Fig. 7.3 is more consistent with the location of the first error and the subsequent propagated errors.

The top-down backtracking can only be performed on the bad branches that originate from the SC path as a reference path. The rest of the backtracking iterations follows the bottom-up fashion. Note that a good branch is determined as a local branch with a higher likelihood among two branches emerging from a parent node. Thus, a good branch could form a non-SC path any where on the decoding tree. However, the SC path is distinguished by following the good branches at all the decoding steps from the root to the leaf of the tree. This SC path is shown by the bold line in Fig. 7.2 and Fig. 7.3.

Choosing a bad branch in the backtracking is called a *diversion* and its corresponding metric is denoted by a prime symbol, i.e., μ', in Fig. 7.2 and Fig. 7.3. This diversion is equivalent to flipping a bit/bits [25] from the SC path in the SC decoding. In Algorithm 9, lines 5-12 and 14-25 implement the top-down and the bottom-up traversals, respectively.

7.3.4 Threshold Update Strategy

When the channel noise has a high impact on the decision LLRs of low-reliability bits, as discussed in Section 7.2.2, the best path metric μ drops significantly over a burst of low-

reliability bit-channels such that $\mu \ll T$. On the other hand, at every iteration of backtracking (i.e., exploring all the potential sub-paths branching off from the current path), the threshold is reduced by Δ. Thus, several backtracking iterations are required to satisfy $\mu > T - m\Delta$ for $m > 1$ (m is the number of backtracking iterations). If we skip the $m-1$ iterations and just we perform one iteration and then update the threshold at once using $T = \lfloor \frac{\mu}{\Delta} \rfloor \Delta$ to satisfy the condition $\mu > T$, we can proceed with the decoding of the current best path and avoid extra delay. There is a possibility that the correct path is not the most likely path and the decoder could find another path in one of the backtracking iterations that we are going to skip. However, our observation shows that the degradation due to skipping $m-1$ backtracking iterations is about 0.05 dB at the high SNR regime. The lines 24-26 in Algorithm 7 shows the implementation of this strategy.

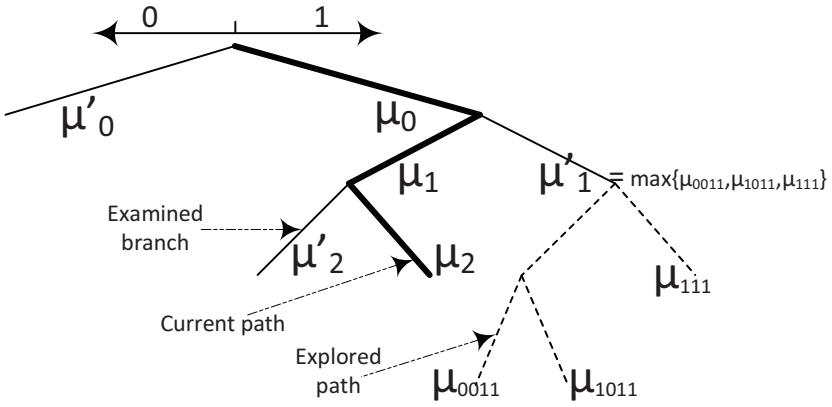

Fig. 7.5: Updating the Metric of Explored Branches

7.3.5 Updating Expected Metrics of Explored Paths

During backtracking, the sub-paths originated from the current best path through bad branches are partially explored. The exploration of the same paths (possibly with longer length) might be repeated later as we proceed with the decoding. Our aim is to benefit from the time spent to explore the sub-paths. By updating the path metric, μ'_j, at the bad branch originated from the current best path, as shown in Fig. 7.5, with the actual result of the exploration rather than the expected path metric, we may avoid re-exploring these paths in the next cycle(s) of backtracking. Since many sub-paths might originate from the same branch, we update μ'_j with the largest metric obtained among sub-paths. This process is performed in lines 55-57 of Algorithm 8. Here, we use μ'' instead for temporarily storing the actual path metric of first

sub-path explored and then comparing it with the actual metric of any new sub-path explored later. Then μ' is updated in line 33 of Algorithm 7. Note that by employing the adaptive heuristic metric, the effect of this updating becomes insignificant.

7.4 Numerical Results

The Fano decoding algorithm provides a performance near the dispersion bound, but as a variable-complexity decoding scheme, its average time complexity is extremely high. The Fano decoding of PAC(128,64) with RM-profile over BIAWGN channel is simulated. Similar to list decoding, the constraint length and the coefficients of the generator polynomial for convolutional codes are 7 ($m = 6$) and 0o133, respectively. The non-optimized design-SNR for obtaining the pre-computed bias term is 4 dB. By applying the ideas introduced in Section 7.2, such as adaptive metric (AD), top-down (TD) search strategy and imposing constraint on the number of diversions (Div) from SC path, it is observed in Fig. 7.6 (left) and Fig. 7.7 (left) that while the average time complexity drops significantly by 50% to 80%, depending on the techniques employed, the degradation in error correction performance is not high. Since the curves in Fig. 7.7 are almost straight in semi-logarithm scale, the complexity gains are preserved at high SNR regimes as well. Fig. 7.7 (right) shows the computational complexity of Fano, stack, and list decoding under different parameters and techniques. The computational complexity is measured by the total number of operations per codeword (comparisons and additions) performed through the factor graph in Fig. 2.2. As can be seen, the computational complexity of list decoding is significantly higher than Fano and stack decoding to achieve the same performance. Fig. 7.7 (left) shows the time complexity in terms of time steps, where each time step is defined as the time required for processing the node(s) in one stage of the factor graph shown in Fig. 2.2. Although the time complexity of the list decoding is significantly lower than Fano and stack decoding (left), we note that one time step in list decoding is longer than a time step in Fano decoding, due to the required sorting process.

Note that stack decoding has a lower time and computational complexity than Fano decoding because it does not need to trace back on the tree and explore other paths to find a promising one if there is any. The partial paths (sorted with respect to the metric) and their associated intermediate information are already available in the stack. Hence, stack decoding can save a significant amount of computations and time at the cost of a huge memory requirement. To compute the number of time steps (or clock cycles), we consider an architecture that is similar

to that in [66]. In this type of design, $2N-2$ time steps are required to decode a codeword [66]. However, in Fano decoding, due to possible backtracking, the number of required time steps is typically significantly larger than $2N-2$. Here, we take the average time steps over a large number of decoding iteration into account. For comparison, we also implemented Fano decoding for polar codes with RM-profile. Although, the average computational complexities of polar codes and PAC codes under Fano decoding are close, due to poor weight distribution of polar codes, PAC codes outperform polar codes.

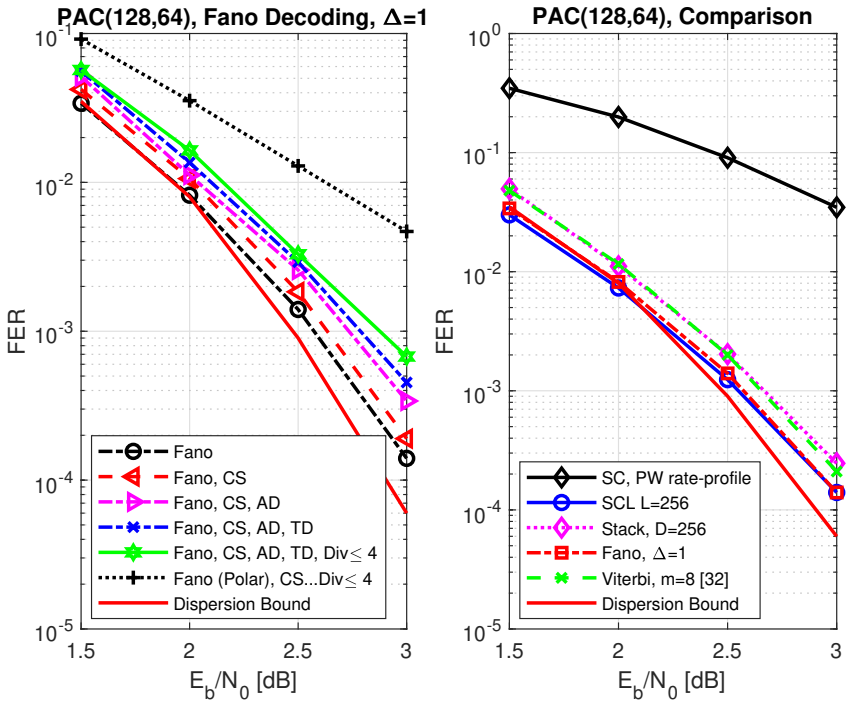

Fig. 7.6: Performance of PAC codes with RM rate-profile under Fano decoding with constrained search (CS), adaptive metric (AD), top-down tree traversal (TD), and a limited number of diversions (Div.) in comparison with other decoding schemes SC, SCL, stack, and Viterbi. Also showing performance of polar codes under Fano decoding "Fano (Polar)".

Another important observation in Fig. 7.6 (left) is that the performance gain of PAC codes over polar codes under Fano decoding is quite significant while the time and computational complexity of these two families of codes are close. However, this performance gain under list decoding as shown in Fig. 6.4 is smaller. Additionally, one can observe from the comparison of the performance of PAC(128,64) under list, stack, and Fano decoding in Fig. 7.6 (right) that Fano decoding provides a similar performance as list decoding but outperforms the stack decoding, while it requires significantly less hardware resources than list decoding and stack decoding. As shown in Table 7.4, the memory required for paths, intermediate LLRs and partial

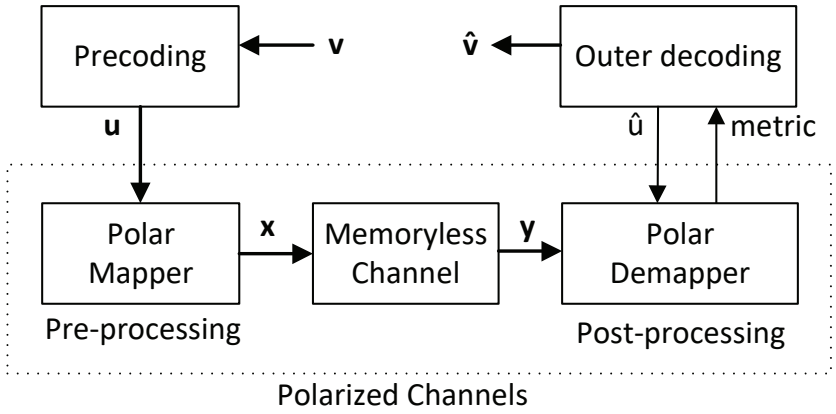

Fig. 7.7: Time and computational Complexity.

sums,which account for the majority of memory space, for list and stack decoding is L and D times that of Fano decoding. Note that in order to obtain a FER performance similar to Fano decoding, we need a very large list size L or stack depth D in the order of 128 or 256. This highlights the huge gap between Fano decoder and the other decoders in terms of hardware resources

The parameters P, and Q_i for $i = 1, 2, 3$ denote the number of processing elements (PE) [66] and the number of quantization bits, respectively.

Finally, Viterbi algorithm (VA) [61] with similar hardware resources as list decoding (except the $2L$-value sorter, replaced by a 2-value comparator) provides a close performance to Fano and list decoders.

7.5 Summary

In this chapter, we investigated the implementation of sequential decoding for PAC codes. Fano decoding has a high average time complexity but a low computational complexity relative to list decoding. For mitigating the time complexity, we propose several techniques and strategies, including an adaptive path metric and a heuristic to estimate a metric for the continuation of the partial paths, search constraints, and a combination of top-down and bottom-up search strategies. These strategies reduce the computational complexity as well. Also, to overcome the difficulty of obtaining the intermediate LLRs and partial sums during backtracking, we propose an algorithm to compute these intermediate information (LLRs and partial sums) efficiently without using extra memory to store them or any need to restart the decoding process. The

numerical results show that by using these techniques, the average time complexity drops by 50% to 80% at the cost of relatively small performance degradation. The adaptive heuristic metric and the search strategies proposed in this chapter can be used in polar coding. Although the time complexity of the Fano Decoding is variable and high, the software Fano decoder is significantly faster than the software list decoder with a large list size without using parallelism.

Due to the need for backtracking in the Fano decoding, the frequency of backtracking through the decoding tree increases prohibitively as the code length increases. Hence, we conclude that the Fano decoding can be used for short codes with medium to low code rates.

Overall, it appears that any proper pre-transformation such as convolutional transform [47], moving parity check bits [18], dynamic frozen bits [85], use of CRC bits for error detection [13], and a combination of them can improve the distance spectrum and results in an error correction performance gain. However, each pre-transformation may provide a different gain depending on the rate profile, block length, and code rate.

Algorithm 7: Fano Decoding of PAC Codes

input : Channel LLRs $\lambda_{\mathbf{n}}^{\mathbf{0,N-1}}$, , N, K, $\mathcal{A}$, $\mathbf{p_e}$, $\mathbf{c}$, Δ, i_{bu}
output: Information bits $\hat{\mathbf{d}}$

cState[1,...,$|\mathbf{c}|-1$] $\leftarrow$ {0} // Current state
currState[0:K-1][1:$|\mathbf{c}|-1$] $\leftarrow$ {0}
[i, j, T, λ, δ, β, b_{-1}, α_q, $\mathcal{CS}$] $\leftarrow$ [0, 0, 0, {0}, {0}, {0}, B, 1,generateCS($\mathcal{A}$)] // $\mathcal{CS}$ in [32]
[onMainPath, isBackTracking, toDiverge, biasUpdated] $\leftarrow$ [*True*, *False*, *False*, *False*]
while $i < N$ **do**
  $\lambda_0^i \leftarrow$ updateLLRs(i, λ, β) // like SCD
  if $i \notin \mathcal{A}$ **then**
    [$\hat{u}_i$, cState] $\leftarrow$ conv1bTrans(0, cState, $\mathbf{c}$) // Alg. 4
    $\mu_i \leftarrow \mu_{i-1} + m(\lambda_0^i, \hat{u}_i) - \alpha_q \cdot \log(1 - p_{e,j})$
    $\beta \leftarrow$ updatePartialSums($i, \hat{u}_i, \beta$) // like SCD
    $i \leftarrow i + 1$
  else
    [$\hat{u}^{(0)}$, cState$^{(0)}$]$\leftarrow$conv1bTrans(0, cState, $\mathbf{c}$)
    [$\hat{u}^{(1)}$, cState$^{(1)}$]$\leftarrow$conv1bTrans(1, cState, $\mathbf{c}$)
    $\mu^{(0)} \leftarrow \mu_{i-1} + m(\lambda_0^i, \hat{u}^{(0)}) - \alpha_q \cdot \log(1 - p_{e,j})$
    $\mu^{(1)} \leftarrow \mu_{i-1} + m(\lambda_0^i, \hat{u}^{(1)}) - \alpha_q \cdot \log(1 - p_{e,j})$
    [μ_{max},$\hat{v}_{max}$]$\leftarrow$ [$\mu^{(0)}$,0] if $\mu^{(0)} > \mu^{(1)}$, else [$\mu^{(1)}$,1]
    [μ_{min},$\hat{v}_{min}$]$\leftarrow$ [$\mu^{(0)}$,0] if $\mu^{(0)} < \mu^{(1)}$, else [$\mu^{(1)}$,1]
    if onMAINpath=*True* **and** isBackTracking = *True* **then**
      if $\mu_{min} > T$ **and** $\mathcal{CS}[j] = 1$ **and** $j < j_{end}$ **then**
        [onMAINpath, δ_j^s, j_{stem}] $\leftarrow$ [*False*, 1, j]
        [$\lambda^{\mathbf{s}}, \beta^{\mathbf{s}}$] $\leftarrow$ [λ, β]
      else if $j = j_{end}$ **then**
        isBackTracking = *False*
        $T = \lfloor \frac{\mu_{end}}{\Delta} \rfloor \Delta$ // Updating threshold
    if $\mu_{max} > T$ **then**
      if toDiverge = *False* **then**
        [$\hat{v}_i$, $\hat{u}_i$] $\leftarrow$ [$\hat{v}_{max}$, $\hat{u}^{(\hat{v}_{max})}$]
        if onMAINpath= *True* **and** δ_j^s = *1* **then**
          [μ_i, μ_i'] $\leftarrow$ [μ_{max}, μ_{min}]
        else
          [μ_i, μ_i'] $\leftarrow$ [μ_{max}, μ_i'']
        $\delta_j \leftarrow 0$
      else
        [$\hat{v}_i$, $\hat{u}_i$] $\leftarrow$ [$\hat{v}_{min}$, $\hat{u}^{(\hat{v}_{min})}$]
        [μ_i, μ_i'] $\leftarrow$ [μ_{min}, μ_{max}]
        [δ_j, toDiverge] $\leftarrow$ [1, *False*]
      [currState[j],cState]$\leftarrow$[cState,cState$^{(\hat{v}_i)}$]
      $\beta \leftarrow$ updatePartialSums($i, \hat{u}_i, \beta$)
      $[i, j] \leftarrow [i + 1, j + 1]$
    else
      if biasUpdated = *False* **and** $i < i_{bu}$ **then**
        $T = \lfloor \frac{\mu_{max}}{\Delta} \rfloor \Delta$ // Updating threshold
      else
        <Go to **Algorithm 8**>
return ($\hat{\mathbf{d}} \leftarrow$ extract($\hat{\mathbf{v}}$, $\mathcal{A}$)) // Dropping 0s

Algorithm 8: Fano Decoding (2): Lines 46-67 in Algorithm 7

```
46 if biasUpdated = False and i = i_bu then
47     if μ_max < B then
48         α_q = ⌈μ_max / (B·Δ_q)⌉ Δ_q
49         biasUpdated = True
50         for k ← 0 to j do
51             μ'_{A[k]} = μ'_{A[k]} + (α_q − 1) · B^c_{A[k]}
52         μ_{A[0]−1} = μ_{A[0]−1} + (α_q − 1) · B^c_{A[0]−1}
53         μ_{A[j]−1} = μ_{A[j]−1} + (α_q − 1) · B^c_{A[j]−1}
54 currState[j] ← cState
55 if onMAINpath = False then
56     if μ''_{A[j_stem]} < μ_max then
57         μ''_{A[j_stem]} ← μ_max
58 else
59     [j_end, μ_end] ← [j, μ_max]
60     [frmMAINpath, isBackTracking] ← [True, True]
61 [T, j', toDiverge] ← moveBack(μ'_{0,i}, j, T, δ_{0,j}, û, CS, frmMAINpath)
     // μ'_{0,i} = μ'_0, μ'_1, ..., μ'_i
62 if toDiverge = False and (j' = j_stem or j' = j) then
63     onMAINpath = True
64 else
65     onMAINpath = False
66 [i, j, frmMAINpath] ← [A[j'], j', False]
67 cState ← currState[j]
```

Algorithm 9: Fano Decoding (3): moveBack - Checking the examined nodes to move backward

input : the channel output μ', j, T, $\delta_{0,j}$, $\hat{\mathbf{u}}$, $\mathcal{CS}$, frmMAINpath
output: T, j', toDiverge,

```
isMovingBack ← False
while True do
    j' ← j
    if frmMAINpath = True then
                                                          // Top-down move
        for k ← 0 to j' − 1 do
            if μ'_{A[k]} > T and CS[k] = 1 then
                [j', j_stem, isMovingBack] ← [k, k, True]
                [λ^s, β^s] ← [λ, β]
                break
        if j' = j then
            toDiverge ← False
            return [T, j, toDiverge]
    else
                                                          // Bottom-up move
        for k ← j' − 1 to 0 do
            if j_stem = k then
                j' ← k
                [λ, β] ← [λ^s, β^s]
                toDiverge ← False
                return [T, j', toDiverge]
            if μ'_{A[k]} > T and CS[k] = 1 then
                if sum(δ_{0,k}) ≥ maxDiversions then
                    continue
                if δ_k = 1 then
                    [j', isMovingBack] ← [k, True]
                    break
    if isMovingBack = True then
        [i_cur, i_start] ← [A[j], A[j']]
        [λ, β] ← updateLLRsPSs(i_start, i_cur, û, λ, β)          // Alg. 6
        if δ_{j'} = 0 then
            toDiverge ← True
            return [T, j', toDiverge]
        else if j' = 0 then
            toDiverge ← False
            return [T, j', toDiverge]
```

Table 7.1: Comparison of hardware resources for Fano, stack, and list decoders

	Fano	Stack	List
Memory Requirement [bits]			
Path memory, $\mathbf{u}$	N	DN	LN
Intermediate LLRs, λ	$(N-1)Q_1$	$D(N-1)Q_1$	$L(N-1)Q_1$
Partial Sums, β	$N-1$	$D(N-1)$	$L(N-1)$
Path Metric, M	$2(N-K)Q_2$	DQ_2	LQ_2
Current State	$K \cdot m$	$D \cdot m$	$L \cdot m$
Critical Set flag, CS	N	0	0
Diversion flag, δ	N	0	0
Error probability, p_e	NQ_3	NQ_3	0
Computing Resources			
Processing Elements	P	P	LP
Comparison	A comparator	D-sorter	$2L$-sorter

Chapter 8

List Viterbi Decoding of PAC Codes

> *"It is better to do the right problem the wrong way than the wrong problem the right way."*
>
> — Richard Hamming

The list Viterbi algorithm (LVA) sorts the candidate paths locally at each trellis node. Motivated by this fact, we adapt the trellis, path metric, and the local sorter of LVA to PAC codes. Then we show how the error correction performance moves from the poor performance of the Viterbi algorithm (VA) to the superior performance of list decoding by changing the constraint length, list size, and the sorting strategy (local sorting and global sorting) in the LVA. Also, we analyze the complexity of the local sorting of the paths in LVA relative to the global sorting in the list decoding, and we show that LVA has a significantly lower sorting complexity than list decoding.

8.1 Trellis and Path Metric in (List) Viterbi Algorithm

The Viterbi algorithm [56] is the most popular decoding procedure for convolutional codes (CCs), which is based on their *trellis diagram* graphical representation [57]. A trellis is a graph where the nodes represent the encoder state. The branch sequences on the trellis are generated by a finite state machine with inputs $\mathbf{v}$ and states $\mathcal{S} = \{s_1, ..., s_{2^m}\}$ and the code is called the trellis code. The Viterbi algorithm traverses the trellis from left to right, finding the maximum likelihood transmitted sequence estimate, when reaching the last stage $t = N - 1$.

PAC codes can be encoded and decoded on the trellis by employing the Viterbi algorithm (VA) [56, 57] and the list-type VA [58]. The trellis used for PAC codes is an irregular trellis which is shown in Fig. 8.1 and Fig. 8.2. As shown, when there is a sub-sequence of at least m zeros in the input $\mathbf{v}$, the current states of all the paths on the trellis transit toward all-zero state.

In convolutional coding, there are three methods to obtain the finite code sequences: (1) code truncation where the encoder stops after a finite block-length, N, and the code sequence is truncated. This method leads to a substantial degradation of error protection, because the

last encoded information bits influence a small number of code bits. (2) code termination where we add some tail bits to the code sequence in order to ensure a predefined end state (usually, the all-zero state) of the encoder, which leads to low error probabilities for the last bits, (3) tail-biting where we choose a starting state that ensures the starting and ending states are the same (this state value does not necessarily have to be the all-zero state). This scheme avoids the rate loss incurred by zero-tail termination at the expense of a more complex decoder. For encoding PAC codes, we use the code truncation, thus we do not add any tail bits. This will not degrade the error protection of last bits because the last encoded bits are transmitted over the high-reliability sub-channels in the polar transform.

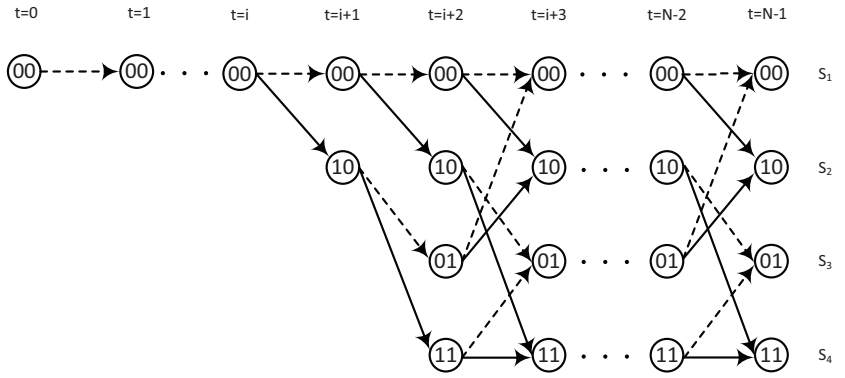

Fig. 8.1: The truncated trellis for PAC codes. Since $v_t = 0$ for $t \in \mathcal{A}^c$, the path does not split. The dashed-line arrows represent the input 0 and the solid-line arrows represent the input 1 to the convolutional transform.

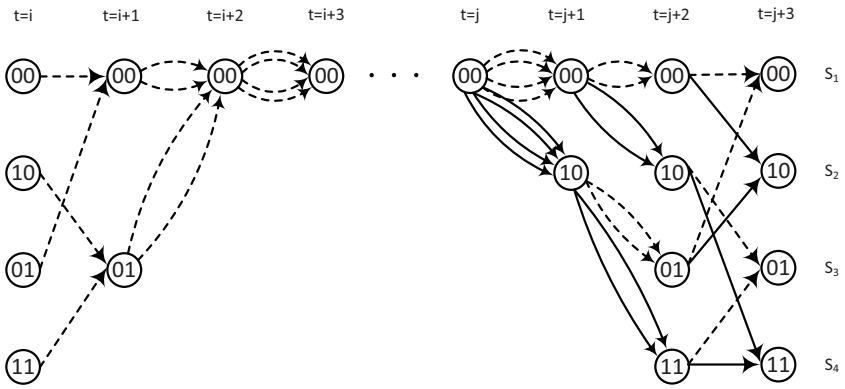

Fig. 8.2: The irregularity of the trellis where $v_t = 0$ for $t \in \mathcal{A}^c$ or $t = [i+1, ..., j]$ for $j > i$. The paths from $t = i+1$ to $t = j$ are not pruned.

The fundamental idea behind the Viterbi decoding is as follows. A coded sequence $\mathbf{u}$, the output of the convolutional transform in Fig. 6.1, corresponds to a path through the trellis.

Due to the noise in the channel, the received vector $\mathbf{y}$ after demapping may not correspond exactly to a path on the trellis. The decoder finds a path through the trellis which has the highest probability to be the transmitted sequence $\mathbf{u}$ over the polarized vector channel. The probability to be *maximized* is

$$P(\hat{\mathbf{u}}|\mathbf{y}) = \prod_{t=0}^{N-1} P(\hat{u}_t|\hat{u}_0^{t-1}, y_0^{N-1}) \tag{8.1}$$

In practice, it is convenient to deal with the logarithm of (8.1) to use an additive metric. Consider now a partial sequence $\hat{u}_0^{t-1} = [\hat{u}_0, \hat{u}_1, \ldots, \hat{u}_{t-1}]$ at the output of the convolutional transform. This sequence determines a path, or a sequence of states, through the trellis for the code.

Let $M_{t-1}(s') = -\sum_{i=0}^{t-1} \log P(\hat{u}_i|\hat{u}_0^{i-1}, y_0^{N-1})$ denote the *path metric* for the sequence $\hat{u}_0^{t-1}$ terminating in state s'. We seek to minimize the path metric for the entire codewords ($t = N-1$) to maximize the probability in (8.1).

Now let the sequence $\hat{u}_0^t$ be obtained by appending $\hat{u}_t$ to $\hat{u}_0^{t-1}$ and suppose $\hat{u}_t$ is such that the state at time $t+1$ is s. The path metric for this longer sequence is

$$\begin{aligned} M_t(s) &= -\sum_{i=0}^{t} \log P(\hat{u}_i|\hat{u}_0^{i-1}, y_0^{N-1}) && (8.2)\\ &= M_{t-1}(s') + \mu_t(s', s) && (8.3) \end{aligned}$$

where $\mu_t(s', s) = -\log P(\hat{u}_t|\hat{u}_0^{t-1}, y_0^{N-1})$ denotes the *branch metric* for the trellis transition from state s' at time t to state s at time $t+1$.

The path metric of a path to state s at time t is obtained by adding the path metric to the state s' at time $t-1$ to the branch metric for an input that moves the encoder from state s' to state s. If there is no such input, i.e., s' and s are not connected on the trellis, then the branch metric is considered ∞.

To simplify the arithmetic operation, we can define μ_t based on $\lambda_0^t(s', s)$ or simply λ_0^t.

$$\begin{aligned} \mu_t(s', s) &= -\log P(\hat{u}_t|\hat{u}_0^{t-1}, y_0^{N-1}) \\ &= -\log\left(\frac{e^{(1-\hat{u}_t)\lambda_0^t}}{e^{\lambda_0^t}+1}\right) = \log\left(1 + e^{-(1-2\hat{u}_t)\lambda_0^t}\right) \end{aligned} \tag{8.4}$$

where the last equality holds only for $\hat{u}_t = \hat{u}_t(s', s) = 0$ and 1. Now, for the value of $\hat{u}_t$ that equals $h(\lambda_0^t)$,

$$h(\lambda_0^t) = \begin{cases} 0 & \lambda_0^t > 0, \\ 1 & \text{otherwise} \end{cases} \tag{8.5}$$

the term $e^{-(1-2\hat{u}_t)\lambda_0^t} = e^{-|\lambda_0^t|}$ is small and hence $\log(1 + e^{-|\lambda_0^t|}) \approx 0$. Otherwise, we can approximate $\log(1 + e^{|\lambda_0^t|}) \approx |\lambda_0^t|$. Thus

$$\mu_t(s', s) = \mu_t(\lambda_0^t, \hat{u}_t) \approx \begin{cases} 0 & \text{if } \hat{u}_t = h(\lambda_0^t) \\ |\lambda_0^t| & \text{otherwise} \end{cases} \tag{8.6}$$

It turns out that this branch metric is equivalent to the one suggested for the list decoding of polar codes in [14, 86] and PAC codes in [48].

When paths merge at state s, we need to select one of them in order to extend it at the next time step. Suppose $M_{t-1}(s'_0)$ and $M_{t-1}(s'_1)$ are the path metrics of the paths ending at states $s'_0, s'_1 \in \{0, 1, ..., 2^m - 1\}$ at time t. Suppose further that both of these states are connected to state s at time $t + 1$, as illustrated in Fig. 8.3.

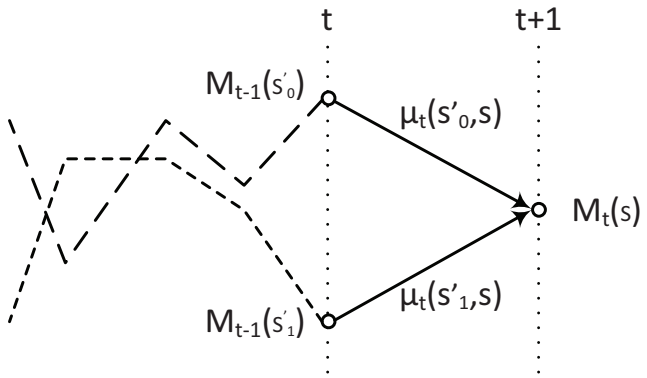

Fig. 8.3: Merging two paths at state s

According to the Bellman's principle of optimality [87], to obtain the ML path through the trellis, the path to any state at each time step must be locally an ML path. This is the governing principle of the Viterbi algorithm. Thus, when the two or more paths merge, the path with the smallest path metric is retained (the *survivor path* or in short the *survivor*) and the other path is eliminated from further consideration. This defines the *add-compare-select* step of the

Viterbi algorithm

$$M_t(s) = \min\{M_{t-1}(s'_0) + \mu_t(s'_0, s), \\ M_{t-1}(s'_1) + \mu_t(s'_1, s)\} \tag{8.7}$$

Note that the initial path metrics are $M_0(0) = 0$ and $M_0(s') = \infty$ for $s' = 1, 2, ..., 2^m - 1$.

In [58], the conventional Viterbi algorithm was generalized to list-type VA where instead of one path, the L paths with smallest metric are selected and extended at time t. Hence, (8.7) is generalized as

$$M_t(s,k) = \underset{\substack{1\le l\le L \\ s'}}{\overset{(k)}{\min}} \{M_{t-1}(s', l) + \mu_t(s', s)\} \tag{8.8}$$

where $\overset{(k)}{\min}$ denotes the k-th smallest value ($1 \le k \le L$).

From (8.8), one can observe some similarity between list decoding of PAC codes and list Viterbi algorithm. The main difference is that in the LVA, the paths are sorted locally at each state, while in list decoding all the paths are sorted globally and then half of them are discarded.

Algorithm 10 illustrates the list Viterbi algorithm. In the beginning, there is a single path in the list. When the index of the current bit is in the set $\mathcal{A}^c$, the decoder knows its value, usually $v_t = 0$, and therefore it is encoded into u_t based on the current memory state S and the generator polynomial $\mathbf{c}$ in line 8. Then, using the decision LLR λ_0^t obtained in line 6, the corresponding path metric is calculated using subroutine *calcM*. Note that in the algorithm 10, instead of $M_t(s,k)$ in 8.8, we use $M_t(k)$. Although the metric is calculated in lines 9 and 26-27 regardless of the current state of each corresponding path, when we sort the paths in line 16, we consider their current states. Eventually, the decoded value u_t is fed back into SC process in line 10 to calculate the partial sums. On the other hand, if the index of the current bit is in the set $\mathcal{A}$ (see lines 12-17), there are two options for the value of v_t, i.e., 0 and 1, to be considered in line 23. For each option of 0 and 1, the aforementioned process for $t \in \mathcal{A}^c$ including convolutional encoding, and calculating the path metric is performed and then the two encoded values $u_t = 0$ and 1 are fed back into SC process to update the partial sums β_π.

The vector $\lambda[\pi]$ as the input argument of the subroutine *updateLLRs* constitutes the $N-1$ intermediate LLR values of path π. The subroutine *updateLLRs* updates all the intermediate LLRs and gives $\lambda_0^t[\pi]$. Similarly, the vector β_π constitutes the $N-1$ intermediate partial sums of path π which is needed to compute the intermediate LLRs. The partial sums are updated after decoding each bit by the subroutine *updatePartialSums*. The subroutines *updateLLRs*,

updatePartialSums, and *prunePaths* in Algorithm 10 are identical to the ones used in SCL decoding of polar codes.

Note that the local path sorting is only performed for the decoding the bits with indices in $\mathcal{A}$ when the number of paths at each node exceeds L or the total number of paths exceeds $2^m \cdot L$ (as shown in lines 14-17 of Algorithm 10). Reaching to $2L$ paths takes $\log_2 L$ steps (i.e., after decoding the first $\log_2 L$ bits with indices in $\mathcal{A}$). Despite the irregularity of the trellis as shown in Fig. 8.2, since the the $2^m \cdot L$ paths are just extended over the bits with indices in $\mathcal{A}^c$ (that is, no path splitting or expansion is performed), the total number of sorting remains similar to LD, which is $(K - \log_2 L)$.

8.2 Generalization of List Viterbi Algorithm

Successive Cancellation List Viterbi algorithm (SC-LVA or in short LVA) for decoding of PAC codes can be considered a generalized decoder for PAC codes in a sense that it can be converted to SC decoding, SC list decoding and Viterbi decoding by changing the parameters of the algorithm.

In terms of sorting strategy for the path metrics at each time step, there are two strategies to consider:

- *global sorting* of all the paths regardless of their current states. In this case, LVA will not have a fixed number of survivors for each state (or at each node on the trellis) and the decoding reduces to LD of PAC codes. In this case, the performance improves by increasing the list size, L_G. A special case of list decoding is SC decoding when $L_G = 1$. Here, L_G denotes the global list size in LD. Hence, L refers to the local list size throughout this chapter.

- *local sorting* of the paths with the same current state (the paths connected to the same node on the trellis). This is the conventional LVA for PAC codes described in section 8.1. In this case, by increasing either the list size L or the number of states $|\mathcal{S}|$, while keeping the other parameter constant, the performance improves. However, if we keep the product of $L \cdot |\mathcal{S}|$ constant, an increase in L improves the performance as far as $|\mathcal{S}|$ is not too small. Note that in this case, if $|\mathcal{S}|$ becomes too small such as $|\mathcal{S}| = 2$, the convolution has a limited span and results in a degradation in FER performance as we will see in Section 8.4. Needless to mention that if we increase $L \cdot |\mathcal{S}|$, the performance

Algorithm 10: List Viterbi Decoding of PAC codes

input : $\mathcal{A}$, L, $\mathbf{c}$, $\lambda_n^{0,N-1}$
output: the recovered message bits $\hat{\mathbf{d}}$

$\Pi \leftarrow \{1\}$ // a single path in the list
$m \leftarrow |\mathbf{c}| - 1$ // memory size
for $t \leftarrow 0$ **to** $N-1$ **do**
  if $t \notin \mathcal{A}$ **then**
    for $\pi \leftarrow 1$ **to** $|\Pi|$ **do**
      $\lambda_0^t[\pi] \leftarrow$ updateLLRs(π, t, $\lambda[\pi]$, β_π) // updateLLRs: Identical with SCD's
      $\hat{v}_t[\pi] \leftarrow 0$
      $[\hat{u}_t[\pi]$, S$[\pi]] \leftarrow$ conv1bEnc(0, S$[\pi]$, $\mathbf{c}$)
      $M_t(\pi) \leftarrow M_{t-1}(\pi) + \mu_t(\lambda_0^t[\pi], \hat{u}_t[\pi])$ // cf. 8.6
      $\beta_\pi \leftarrow$ updatePartialSums($\hat{u}_t[\pi]$, β_π) // Identical w/ SCD's
  else
    for $\pi \leftarrow 1$ **to** $|\Pi|$ **do**
      duplicatePath(π, t, $\mathbf{c}$)
    if $|\Pi| > 2^m.L$ **then**
      for $s \leftarrow 1$ **to** 2^m **do**
        Sort $\{M_t(\pi)\}, \pi \in \Pi$: connected to s
        Retain L paths (π's) with smallest M_t
$\hat{v}_0^{N-1}[1:|\Pi|] \leftarrow$ sort($\hat{v}_0^{N-1}[1:|\Pi|]$) // in ascending order
$\hat{\mathbf{d}} \leftarrow$ extractData($\hat{v}_0^{N-1}[0]$, $\mathcal{A}$) // inverse of rate-profiling
return $\hat{\mathbf{d}}$;
subroutine duplicatePath(π, t, $\mathbf{c}$):
  $\Pi \leftarrow \Pi \cup \{\pi'\}$ // path π' is a copy of path π
  $\lambda_0^t[\pi] \leftarrow$ updateLLRs(π, t, $\lambda[\pi]$, β_π) // like SCD
  $(\hat{v}_t[\pi], \hat{v}_t[\pi']) \leftarrow (0, 1)$
  $[\hat{u}_t[\pi]$, S$[\pi]] \leftarrow$ conv1bEnc($\hat{v}_t[\pi]$, S$[\pi]$, $\mathbf{c}$)
  $[\hat{u}_t[\pi']$, S$[\pi']] \leftarrow$ conv1bEnc($\hat{v}_t[\pi']$, S$[\pi]$, $\mathbf{c}$)
  $M_t(\pi) \leftarrow M_{t-1}(\pi) + \mu_t(\lambda_0^t[\pi], \hat{u}_t[\pi])$ // cf. 8.6
  $M_t(\pi') \leftarrow M_{t-1}(\pi) + \mu_t(\lambda_0^t[\pi], \hat{u}_t[\pi'])$ // cf. 8.6
  $\beta_\pi \leftarrow$ updatePartialSums($\hat{u}_t[\pi]$, β_π) // like SCD
  $\beta_{\pi'} \leftarrow$ updatePartialSums($\hat{u}_t[\pi']$, β_π)
subroutine conv1bEnc(v, currState, $\mathbf{c}$):
  $u \leftarrow v \cdot g_0$
  for $j \leftarrow 1$ **to** $|\mathbf{c}|$ **do**
    if $g_j = 1$ **then**
      $u \leftarrow u \oplus$ currState$[j-1]$
  nextState $\leftarrow [v_i]$ + currState[1,...,$|\mathbf{c}| - 2$]
  return (u, nextState);

improves. Note that the PAC code changes by changing $|\mathcal{S}|$, since we are using a different $\mathbf{c}$. Since it was observed that the FER performance of PAC codes is not significantly affected by the change of $\mathbf{c}$, we can vary this parameter and the local list size and observe the tradeoffs of the different decoders.

Additionally, when we choose only one path at each state ($L = 1$), LVA is converted to a standard Viterbi algorithm (VA) for PAC codes, which was described in Section 8.1. In this case, as the number of states, $|\mathcal{S}|$, on the trellis increases, the performance improves. Also note that PAC coding with $\mathbf{c} = [1]$ or $m = 0$ is equivalent to polar coding simply because there is no pre-transformation or pre-coding in this case.

8.3 Sorting Complexity

As discussed in the previous section, the error correction performance of the decoding changes with the sorting strategy as well as the list size and the number of states. Now, let us analyse the sorting complexity in list decoding and list Viterbi decoding. Suppose the total number of survivor paths is the same in LD and LVA, i.e., $L_G = L \cdot 2^m$. As we will observe in the next section, in the condition of the same number of survivors, LD slightly outperforms LVA due to the global sorting strategy. However, in case of parallelism which is popular in the hardware design, the local sorting in LVA can improve the latency significantly.

Let us consider a bitonic sorter [88] with $1+\log L$ super-stages that can sort $2L$ path metrics shown in Fig. 8.4. At each super-stage with index $\psi \in \{1, ..., 1+\log L\}$, there are ψ stages (i.e., the number of stages at each super-stage equals the index (ψ) of the corresponding super-stage, see the top and the bottom of Fig. 8.4), each including L pairs of a component (shown by vertical connections in Fig. 8.4) consists of a comparator and 2-to-2 multiplexer, which work in parallel. This sorter was used for list decoding of polar codes in [89] and later improved in [90]. The length of the critical path of the sorter is determined by the total number of stages which is computed based on the sum of the arithmetic progression as follows:

$$\Psi_{LD} = \sum_{\psi=1}^{1+\log_2 L_G} \psi = \frac{1}{2}(1+\log_2 L_G)(2+\log_2 L_G) \tag{8.9}$$

From (8.9), one can see the impact of the list size, L_G, on the latency of the sorter and consequently the whole decoder. The pruned bitonic sorter suggested in [90] removes one stage

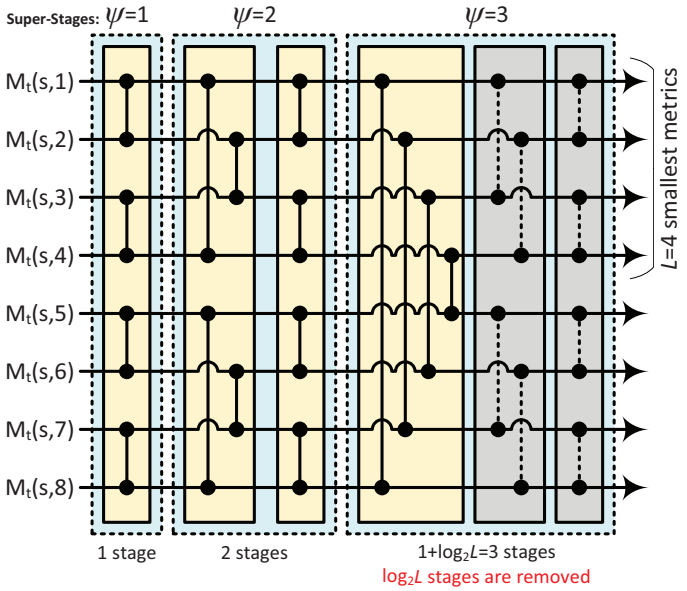

Fig. 8.4: The reduced bitonic sorting network for LVA with $L = 4$. The order of L smallest path metrics is not needed.

out of Ψ_{LD} stages, which is not significant in the case of large L_G. An efficient solution for a significant reduction in the number of stages is to employ LVA where the sorting is performed locally at each state. Therefore, the parameter L_G in (8.9) is divided by the number of states. It turns out the order of the sorted metrics in LVA is not needed unlike in the pruned bitonic sorter where the pruning is performed based on our prior knowledge about the order and the relations between adjacent metric before and after the tree extension. Hence, we can remove the last $\log_2 L_G$ stages in the last super-stage. As a result, the total number of stages is:

$$\Psi_{LVA} = \frac{1}{2}\left(1 + \log_2 \frac{L_G}{2^m}\right)\left(2 + \log_2 \frac{L_G}{2^m}\right) - \log_2 L_G \tag{8.10}$$

Thus, employing LVA results in a significant reduction in the total number of sorting stages for decoding a codeword, i.e., $K\Psi_{LD}$. For instance, for list decoding of PAC(256,128) with $m = 6$ and $L_G = 128$ which has 128 survivors at each decoding stage, the number of sorting stages is $\Psi_{LD} = 36$. In the improved sorters proposed in [91,92], Ψ_{LD} reduced to 28 and 27, respectively. However, in decoding of the same code under list VA with $m = 4$ and $L = 128/2^4 = 8$ which has 32 survivors at each decoding stage, $\Psi_{LVA} = (10 - 3) = 7$, which is 74% smaller than [92]. Note that this reduction comes at the cost of a slight degradation in the FER performance. In software implementation, the sequential sorting algorithms such as Heapsort and Mergesort

cannot perform better than $O(L'\log(L'))$ in terms of time complexity (where $L' = 2L_G$). By employing LVA, the time complexity reduces to $O(2^m(L'/2^m \log(L'/2^m))) = O(L'\log(L'/2^m))$.

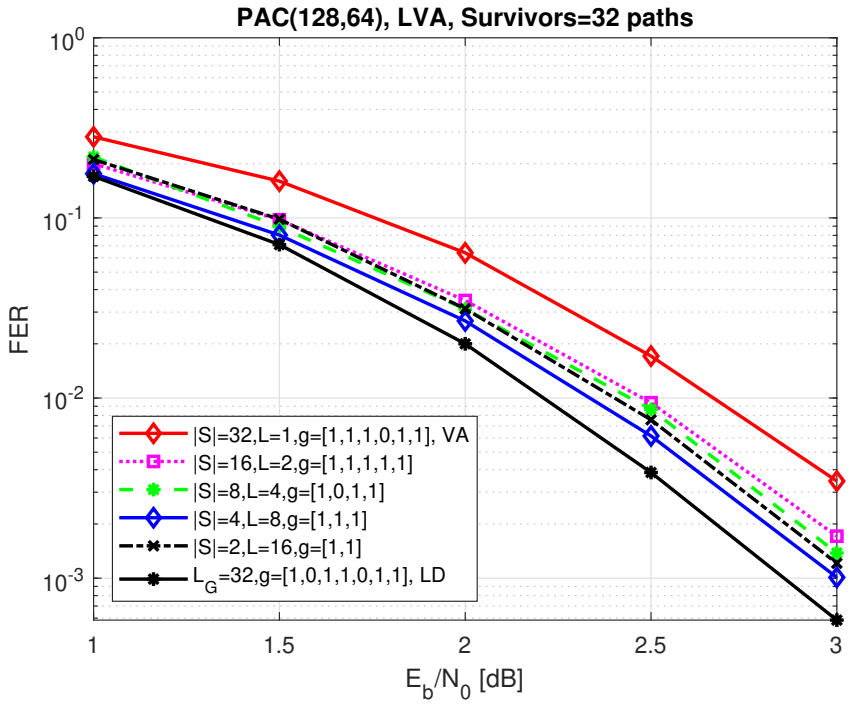

Fig. 8.5: FER Comparison under LVA with various parameters while the total number of paths is 32.

8.4 Numerical Results

In this Section, the error correction performance of list Viterbi algorithm for PAC(128,64) on the trellis with different setups is illustrated and analyzed. The RM rate-profile [48] is used in the results shown in this section. The codewords are modulated based on BPSK and transmitted over the AWGN channel. Fig. 8.5 compares the FER performance under LVA with various list sizes L, while the total number of survivor paths at each time step t remains constant (32 survivors). As can be seen, the performance improves as L increases. In order to understand the reason behind this improvement, suppose we have $2L$ paths at a state s, which are sorted in the ascending order (with indices $1 \leq k \leq 2L$). Since the path pruning operation is performed locally at each state, when the metric of the correct path $M_t(s, k_c)$ has an index $k_c > L$ in the sorted list (due to noise on a low reliability bit-channel), this event results in the elimination of the correct path. As L increases, the probability of elimination

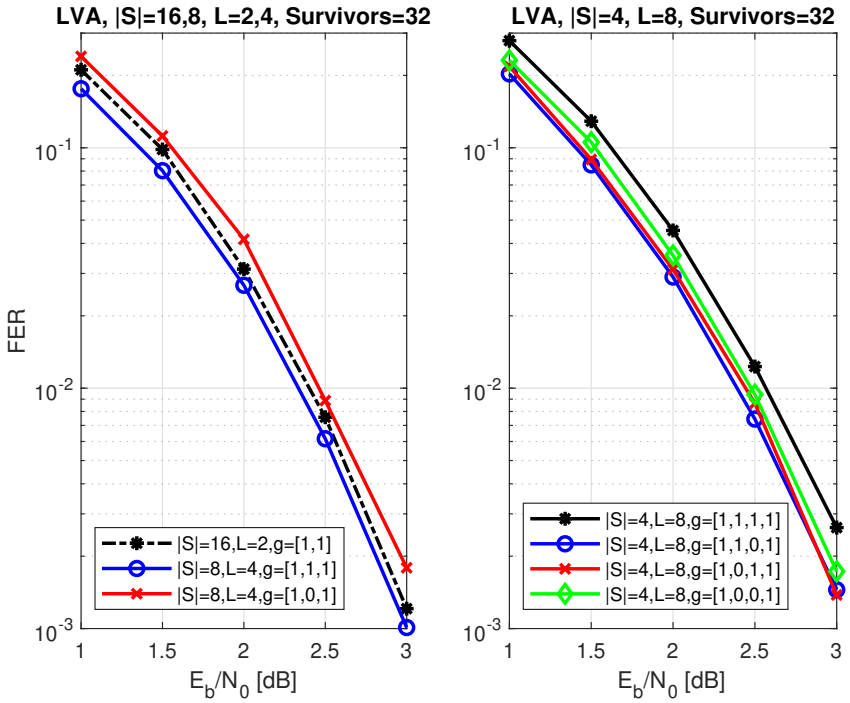

Fig. 8.6: FER Comparison of PAC(128,64) under LVA when convolutional generator polynomial ($\mathbf{c}$) changes.

of the correct path at a state s, $Pr(M_t(s,k_c) > M_t(s,L))$, decreases. Hence, decoding with a larger L performs better.

Note that increase in the number of states $|\mathcal{S}|$ or the memory size m improves the convolution by potentially combining a larger number of previous bits with the current bit. When m is large, the sensitivity of the FER performance to the choice of generator polynomial decreases. Although increase in either L or $|\mathcal{S}|$ or both improves the FER performance of LVA, the available resources are limited. Hence, we prioritize increase in $|\mathcal{S}|$ over m because the impact of change in L is more significant.

Furthermore, as the memory size m (or the number of states $|S|$) become smaller when L increases, the performance becomes more sensitive to the generator polynomial of the convolutional transform. Fig. 8.6 compares the performance under various generator polynomials. The generator polynomial in the form of [1,0,...,0,1] usually does not provide a good convolution by combining one of the previous bits in the shift register with the current bit. Unfortunately, there is no systematic method to find a good generator polynomial $\mathbf{c}$ and a computer search is required, as in the search of optimal convolutional codes. Fig. 8.7 illustrates the BER performance equivalent to some of the codes shown in Fig. 8.5. As can be seen, the gap between BER

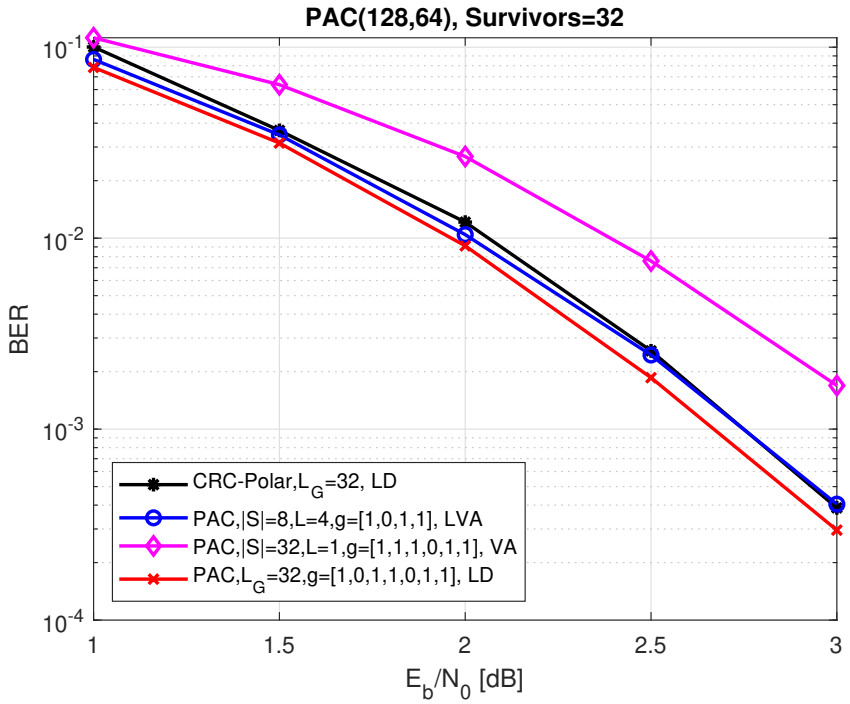

Fig. 8.7: BER Comparison of PAC codes and CRC-polar codes under LVA, VA and LD, the total number of paths is 32.

performance of PAC codes under LVA and LD is less in comparison with FER performance.

To compare the FER performance of PAC codes under LD with LVA while both use the same convolutional generator polynomial, we fix $\mathbf{c} = [1, 1, 1]$. Fig. 8.8 shows the results. As can be seen, the performance of LVA with 32 surviving paths ($|\mathcal{S}| \times L = 32$) is quite close to LD with the same number of surviving paths (L_G). The gap is slightly less than the one in Fig. 8.5 and 8.9 where we use a larger $|\mathcal{S}|$ for LD.

The FER performance of a tail-biting convolutional code CC(128,64) under the wrap-around Viterbi algorithm (WAVA) [93] is provided in Fig. 8.9 for comparison. Fig. 8.9 shows that as the total number of survivors increases, the gap between the performance of LD, VA and LVA decreases. This makes LVA a better candidate when employing a very large list size, given complexity advantage shown in Section 8.3. Conversely, when list size is small, the performance of LVA with total paths of $|\mathcal{S}| \times L$ is in between the performance of LD with list size L_G and $L_G/2$ and overlaps with the performance of polar codes concatenated with 8-bit CRC (0xA6) as it is shown in Fig. 8.9 for LVA with $|\mathcal{S}| \times L = 32$ and LD with $L_G = 16$. Nevertheless, in this case by using the bitonic sorting network in LVA, the number of sorting stages is $\Psi_{LVA} = (1 + \log_2(32/4))(2 + \log_2(32/4))/2 - \log_2 32 = 5$ which is still significantly

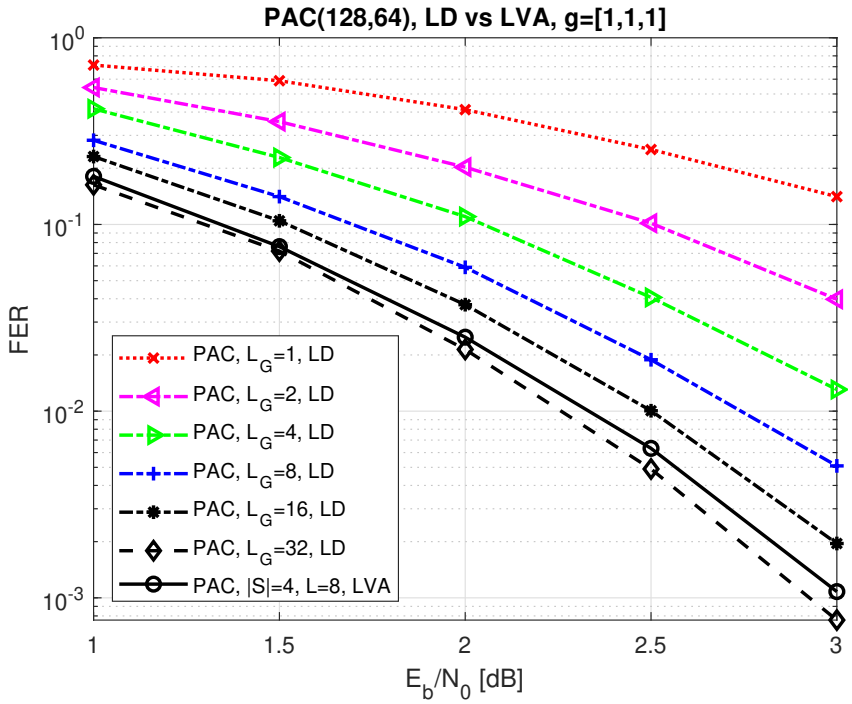

Fig. 8.8: FER Comparison of PAC codes under LD for $L_G = 1, ..., 16, 32$ with PAC codes under LVA with 32 surviving paths ($|\mathcal{S}| \times L = 32$) while $\mathbf{c} = [1, 1, 1]$ is fixed for all.

smaller than LD with the half of survivors, i.e., $\Psi_{LD} = 14, 10$ and 8 using the sorters proposed in [90,92] and [91]. This comparison clearly illustrates the advantage of local sorting over global sorting.

8.5 Summary

In this chapter, we investigated the implementation of the list Viterbi decoding for PAC codes. We showed that LVA could be considered a general decoding scheme, which can transition from list decoding to Viterbi algorithm decoding by changing the number of states and the local list size. The results showed that as the local list size increases, the performance improves. This implies that in the local sorting of the paths, the probability of discarding the correct path is higher than the global sorting in list decoding. On the other hand, local sorting has the advantage of significantly lower complexity than global sorting. Therefore, depending on the application, we can trade sorting complexity for performance, especially when the list size is large.

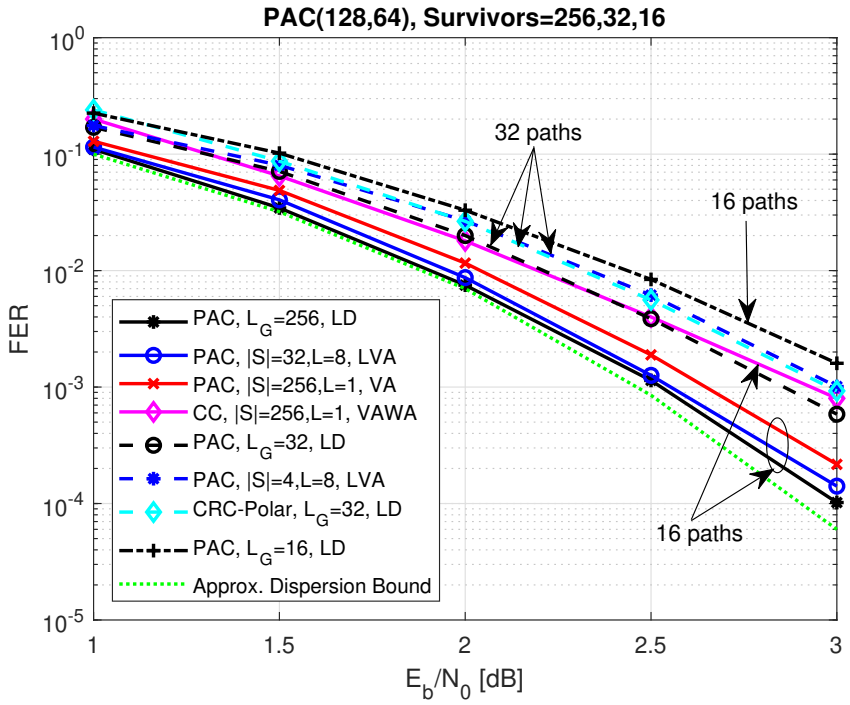

Fig. 8.9: FER Comparison under LVA with various parameters while the total number of paths are 256, 32, and 16. The coefficients of the generator polynomial used with $|S| = 256$ is $\mathbf{c} = [1, 1, 0, 1, 1, 0, 0, 1, 1]$ and for PAC codes unde LD is $\mathbf{c} = [1, 0, 1, 1, 0, 1, 1]$. For the rest, $\mathbf{c}$ is the same as the ones in Fig. 8.5.

Chapter 9

Conclusion and Future Work

> *"There are not more than five musical notes, yet the combinations of these five give rise to more melodies than can ever be heard."*
>
> — Sun Tzu, The Art of War

Polar codes and their variants as high-performance codes with explicit construction are gradually making their way to the applications with high-reliability requirements, such as the control channel of the 5th generation (5G) of mobile communications. Nevertheless, the popular decoder of these codes suffers from high computational complexity and medium latency comparing with the competitors. In this thesis, we proposed various schemes to improve the performance of CRC-polar codes and PAC codes or reduce the time/computational complexity of the underlying decoders. Although the primary focus was on the SC list decoding for CRC-polar codes, we investigated sequential decoding (including Fano and stack decoding) and list Viterbi decoding for PAC codes. We studied how to balance the resources and the performance in the code segments under list decoding. We adjusted the list size locally for different segments to reduce decoding complexity while preserving the performance. We modified the code to adjust the possibility of eliminating the correct path while using a fixed list size to improve the overall FER performance. In another work, we introduced a general scheme for recovery of the correct path in the list decoding through additional decoding attempts when the decoding fails. This scheme was called the shifted-pruning scheme as it shifts the pruning window to avoid eliminating the correct path. As a side result, we propose an efficient partial rewinding of the SC algorithm for re-decoding the same codeword. This scheme allows us to resume decoding from the closest position to the shifting position. Regarding PAC codes, we studied how convolutional precoding removes a significant number of min weight codewords and improves the weight distribution of polar codes. Also, the weak side of PAC codes where the precoding cannot improve the weight distribution was highlighted. The Fano decoding of PAC codes performed as well as list decoding of polar codes with quite a large list size while not requiring a massive amount of resources, although it suffers from variable latency. We suggested several techniques for tree search in Fano decoding aiming to reduce the complexity of memory-efficient Fano decoder. Adaptation of list Viterbi decoder to PAC codes revealed that this decoder can provide a FER performance close to list decoder with significantly lower

complexity for path sorting.

9.1 Suggestions for Future Work

A few suggestions related to the works discussed in this thesis are provided as follows:

Enhanced Shifted-pruning

In Chapter 4 and [32,33], we mentioned that the range of shift, κ, of pruning window in this scheme can be flexible. The parameter κ can vary throughout the decoding. The mixture of variable κ and variable L, as we used in stepped list decoding, can provide a fast and reliable decoding process based on some metric.

Goal-oriented Code Modification

We discussed the shortcoming of the code construction methods, which rely mainly on the reliability of bit-channels. Every decoding process has its requirements. The code can be modified for each decoder as we suggested an approach in Chapter 3. We believe that we can design codes to follow a specific goal. Examples are as follows:

- Nested shifted-pruning: Performing multiple shifting is computationally quite expensive because finding the right combination among so many combinations requires a massive search even by using effective metrics and considering the impact of shifting on the metrics of the proceeding bit positions like what was suggested in dynamic SC-flip decoding in [26]. However, if we limit the combinations to one specific segment, searching for the right combination becomes easy. We suggest designing a code in which one segment is relatively more prone to error than others. Then, the multiple shifting can be performed only within that specific segment.

- Stepped list decoding: We showed in Chapter 3 that by step-wise reducing the list size L in the segments in the descending order such as $[L, L, L/2, L/4]$, we can save a significant amount of memory space and computational complexity. However, as we showed there, due to the imbalance of the bit error rate in the segments, this scheme may result in FER degradation for some codes. We can design a code such that the probability of error in the segments is adjusted $[L, L, L/2, L/4]$ for the stepped list decoding.

Segment-oriented Code-design and Decoding

In [37] and [38], we showed that we could devise a more efficient decoding process or design a better code for a specific decoder by taking advantage of the correct path recovery phenomenon, given enough number of frozen bits in each segment. Furthermore, we detected the most probable segment where the correct path is pruned in Chapter 4. This approach could be further enhanced by employing analytical metrics based on the sub-channel reliability and the weight of the corresponding rows in $\mathbf{G}_N$.

List Viterbi Decoding

In Chapter 8 and [61], we showed that the time complexity of local sorting in the list Viterbi decoding is significantly lower than list decoding. As mentioned, this can improve the throughput of the decoder. We believe that due to this advantage of list Viterbi decoding, it is worth working further on this decoder, such as

- improving the pruning scheme at the nodes of the trellis locally to improve the performance,
- designing more efficient local sorter to further reduce the complexity, and
- implementing fast and efficient parallel hardware for list Viterbi decoding.

Bibliography

[1] E. Arıkan, "Channel polarization: A method for constructing capacity-achieving codes for symmetric binary-input memoryless channels," *IEEE Trans. Inf. Theory*, vol. 55, no. 7, pp. 3051–3073, Jul. 2009.

[2] 3GPP, "Final report of 3gpp tsg ran wg1 #87 v1.0.0," http://www.3gpp.org/ftp/tsg ran/WG1 RL1/TSGR1 87/Report/Final Minutes report RAN1%2387 v100.zip, Nov. 2016.

[3] E. Arikan *et al.*, "Performance of short polar codes under ML decoding," in *Proc. ICT-Mobile Summit*, Santander, Spain, June 2009, pp. 1–6.

[4] J. Massey, "Capacity, cutoff rate, and coding for a direct-detection optical channel," *IEEE Transactions on Communications*, vol. 29, no. 11, pp. 1615–1621, 1981.

[5] M. S. Pinsker, "Channel coding rate in the finite blocklength regime," *Problemy Peredachi Informatsii*, vol. 1, no. 1, pp. 84–86, 1965.

[6] H. Imai and S. Hirakawa, "A new multilevel coding method using error-correcting codes," *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 371–377, 1977.

[7] R. Mori and T. Tanaka, "Performance of polar codes with the construction using density evolution," *IEEE Commun. Lett.*, vol. 13, no. 7, pp. 519–521, Jul. 2009.

[8] I. Tal and A. Vardy, "How to construct polar codes," *IEEE Trans. Info. Theory*, vol. 59, no. 10, pp. 6562–6582, Oct. 2013.

[9] P. Trifonov, "Efficient design and decoding of polar codes," *IEEE Trans. Commun.*, vol. 60, no. 11, pp. 3221–3227, Nov. 2012.

[10] C. Schürch, "A partial order for the synthesized channels of a polar code," in *IEEE Int. Symp. Inform. Theory*, Barcelona, Spain, Jul. 2016, p. 220–224.

[11] M. Mondelli, S. H. Hassani, and R. Urbanke, "Construction of polar codes with sublinear complexity," in *IEEE Int. Symp. Inform. Theory*, Aachen, Germany, Jun. 2017, p. 1853–1857.

[12] M. Qin, A. B. J. Guo, A. G. i Fabregas, and P. Siegel, "Polar code constructions based on llr evolution," *IEEE Commun. Lett.*, vol. 21, no. 6, pp. 1221 – 1224, Jan. 2017.

[13] I. Tal and A. Vardy, "List decoding of polar codes," in *IEEE Int. Symp. on Information Theory*, St. Petersburg, Russia, Jul. 2011, p. 1–5.

[14] A. Balatsoukas-Stimming, A. J. Raymond, W. J. Gross, and A. Burg, "Hardware architecture for list successive cancellation decoding of polar codes," *IEEE Trans. Circuits Syst. II*, vol. 61, no. 8, pp. 609–613, Aug. 2014.

[15] M. Rowshan, E. Viterbo, R. Micheloni, and A. Marelli, "Logarithmic non-uniform quantization for list decoding of polar codes," in *2021 IEEE 11th Annual Computing and Communication Workshop and Conference (CCWC)*, 2021, pp. 1161–1166.

[16] K. Chen, B. Li, H. Shen, J. Jin, , and D. Tse, "Reduce the complexity of list decoding of polar codes by tree-pruning," *IEEE Commun. Lett.*, vol. 20, no. 2, pp. 204–207, Feb. 2016.

[17] J. Chen, Y. Fan, C. Xia, C. Tsui, J. Jin, K. Chen, and B. Li, "Low-complexity list successive-cancellation decoding of polar codes using list pruning," in *IEE Global Communications Conference*, Washington DC, USA, Dec. 2016, pp. 1–6.

[18] Z. Zhang, K. Qin, L. Zhang, H. Zhang, and G. T. Chen, "Progressive bit-flipping decoding of polar codes over layered critical sets," in *IEEE Global Communications Conf.*, Singapore, Dec. 2017, pp. 1–6.

[19] M. C. Coşkun and H. D. Pfister, "Bounds on the list size of successive cancellation list decoding," in *2020 International Conference on Signal Processing and Communications (SPCOM)*, 2020, pp. 1–5.

[20] M. C. Coşkun and H. D. Pfister, "An information-theoretic perspective on successive cancellation list decoding and polar code design," *arXiv preprint arXiv:2103.16680*, 2021.

[21] A. Alamdar-Yazdi and F. R. Kschischang, "A simplified successive cancellation decoder for polar codes," *IEEE Commun. Letter*, vol. 15, no. 12, pp. 1378–1380, Dec. 2011.

[22] G. Sarkis, P. Giard, A. Vardy, C. Thibeault, and W. J. Gross, "Fast list decoders for polar codes," *IEEE J Sel Areas Commun*, vol. 32, no. 5, pp. 946–957, May 2014.

[23] J. Guo, Z. Shi, Z. Liu, Z. Zhang, and Q. Liu, "Multi-CRC polar codes and their applications," *Commun. Lett.*, vol. 20, no. 2, pp. 212–215, Feb. 2016.

[24] S. A. Hashemi, C. Condo, F. Ercan, and W. J. Gross, "Memory-efficient polar decoders," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 7, no. 4, pp. 604–615, Dec. 2017.

[25] O. Afisiadis, A. Balatsoukas-Stimming, and A. Burg, "A low-complexity improved successive cancellation decoder for polar codes," in *Asilomar Conf. on Signals, Systems and Computers*, Pacific Grove, CA, USA, Nov. 2014, pp. 2116–2120.

[26] L. Chandesris, V. Savin, and D. Declercq, "Dynamic-SCFlip decoding of polar codes," *IEEE Trans. on Communications*, vol. 66, no. 6, pp. 2333–2345, June 2018.

[27] F. Ercan, T. Tonnellier, N. Doan, and W. J. Gross, "Practical dynamic SC-flip polar decoders: Algorithm and implementation," *IEEE Transactions on Signal Processing*, vol. 68, pp. 5441–5456, 2020.

[28] C. Condo, F. Ercan, and W. J. Gross, "Improved successive cancellation flip decoding of polar codes based on error distribution," in *2018 IEEE Wireless Communications and Networking Conference Workshops (WCNCW)*, 2018, pp. 19–24.

[29] B. Li, H. Shen, and D. Tse, "An adaptive successive cancellation list decoder for polar codes with cyclic redundancy check," *IEEE Communications Letters*, vol. 16, no. 12, pp. 2044–2047, Dec. 2012.

[30] M. Rowshan, E. Viterbo, R. Micheloni, and A. Marelli, "Repetition-assisted decoding of polar codes," *IET Electronics Letters*, vol. 55, no. 5, pp. 270–272, 2019.

[31] Y. Yongrun, P. Zhiwen, L. Nan, and Y. Xiaohu, "Successive cancellation list bit-flip decoder for polar codes," in *10th International Conference on Wireless Communications and Signal Processing (WCSP)*, 2018, pp. 1–6.

[32] M. Rowshan and E. Viterbo, "Improved list decoding of polar codes by shifted-pruning," in *2019 IEEE Information Theory Workshop (ITW)*, 2019, pp. 1–5.

[33] M. Rowshan and E. Viterbo, "Shifted pruning for path recovery in list decoding of polar codes," in *2021 IEEE 11th Annual Computing and Communication Workshop and Conference (CCWC)*, 2021, pp. 1179–1184.

[34] Y. Lv, H. Yin, and Y. Wang, "An adaptive ordered shifted-pruning list decoder for polar codes," *IEEE Access*, vol. 8, pp. 225 181–225 190, 2020.

[35] F. Cheng, A. Liu, Y. Zhang, and J. Ren, "Bit-flip algorithm for successive cancellation list decoder of polar codes," *IEEE Access*, vol. 7, pp. 58 346–58 352, 2019.

[36] Y.-H. Pan, C.-H. Wang, and Y.-L. Ueng, "Generalized SCL-flip decoding of polar codes," in *IEEE Global Commun. Conf. (GLOBECOM)*, 2020, pp. 1–6.

[37] M. Rowshan and E. Viterbo, "Stepped list decoding for polar codes," in *2018 IEEE 10th International Symposium on Turbo Codes Iterative Information Processing (ISTC)*, 2018, pp. 1–5.

[38] M. Rowshan and E. Viterbo, "How to modify polar codes for list decoding," in *2019 IEEE International Symposium on Information Theory (ISIT)*, 2019, pp. 1772–1776.

[39] E. Arikan, "A performance comparison of polar codes and reed-muller codes," *IEEE Commun. Lett.*, vol. 12, no. 6, pp. 447–449, Jun. 2008.

[40] A. Elkelesh, M. Ebada, S. Cammerer, and S. ten Brink, "Belief propagation list decoding of polar codes," *IEEE Communications Letters*, vol. E22, no. 8, pp. 1536–1539, Aug 2018.

[41] N. Goela, S. Korada, and M. Gastpar, "On LP decoding of polar codes," in *Proc. Inf. Theory Workshop*, Cairo, Eqypt, Aug. 2010, pp. 1–5.

[42] V. Taranalli and P. Siegel, "Adaptive linear programming decoding of polar codes," in *Proc. IEEE Int. Symp. Inf. Theory*, Honolulu, HI, USA, Jun. 2014, pp. 2982–2986.

[43] S. Kahraman and M. E. Celebi, "Code based efficient maximum likelihood decoding of short polar codes," in *Proc. IEEE Int. Symp. Inf. Theory*, Cambridge, MA, USA, Jul. 2012, pp. 1967–1971.

[44] S. A. Hashemi, C. Condo, and W. J. Gross, "List sphere decoding of polar codes," in *Proc. Asilomar Conf. on Signals, Systems and Computers*, Pacific Grove, CA, USA, Jul. 2015, pp. 1346–1350.

[45] K. Niu and K. Chen, "Stack decoding of polar codes," *IET Electronics Letters*, vol. 48, no. 12, pp. 695–697, Jun. 2012.

[46] D. Wu, Y. Li, X. Guo, and Y. Sun, "Ordered statistic decoding for short polar codes," *IEEE Communications Letters*, vol. 20, no. 6, pp. 1064–1067, Jun. 2016.

[47] E. Arikan, "Systematic polar coding," *IEEE Commun. Lett.*, vol. 15, no. 8, pp. 860–62, Aug. 2011.

[48] M. Rowshan, A. Burg, and E. Viterbo, "Polarization-adjusted convolutional (PAC) codes: Sequential decoding vs list decoding," *IEEE Trans. on Vehicular Technology*, vol. 70, no. 2, pp. 1434–1447, Feb. 2021.

[49] H. Yao, A. Fazeli, and A. Vardy, "List decoding of arıkan's PAC codes," in *2020 IEEE Intl Symp. on Inf. Theory (ISIT)*, Los Angeles, USA, Jun. 2020, pp. 443–448.

[50] B. Li, H. Shen, and D. Tse, "A RM-polar codes," *arXiv preprint arXiv:1407.5483*, 2014.

[51] M. Rowshan and E. Viterbo, "On convolutional precoding in PAC codes," *arXiv preprint arXiv:2103.12483*, 2021.

[52] M. Rowshan, A. Burg, and E. Viterbo, "Complexity-efficient fano decoding of polarization-adjusted convolutional (PAC) codes," in *2020 Intl Symp. on Inf. Theory and Its Applications (ISITA)*, Hawai'i, USA, Oct. 2020, pp. 200–204.

[53] M. Moradi, A. Mozammel, K. Qin, and E. Arikan, "Performance and complexity of sequential decoding of PAC codes," *arXiv preprint arXiv:2012.04990*, 2020.

[54] E. Arıkan, "Systematic encoding and shortening of PAC codes," *Entropy*, vol. 22, no. 11, p. 1301, 2020.

[55] T. Tonnellier and W. J. Gross, "On systematic polarization-adjusted convolutional (PAC) codes," *IEEE Communications Letters*, 2021.

[56] A. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," *IEEE Transactions on Information Theory*, vol. 13, no. 2, pp. 260–269, April 1967.

[57] G. Forney, "The Viterbi algorithm," *Proc. of the IEEE*, vol. 61, no. 3, pp. 268–278, March 1973.

[58] T. Hashimoto, "A list-type reduced-constraint generalization of the Viterbi algorithm," *IEEE Transactions on Information Theory*, vol. 33, no. 6, pp. 866–876, 1987.

[59] E. Arıkan, H. Kim, G. Markarian, U. Ozgur, and E. Poyraz, "Performance of short polar codes under ML decoding," in *Proc. ICT-Mobile Summit Conf.*, Santander, Spain, 2009, pp. 1–6.

[60] M. Mohammad, J.-H. Jong, C. Ravishankar, and C. Barnett, "A comparison between the M-algorithm and the list Viterbi algorithm," in *IEEE Military Communications Conf.*, San Diego, CA, 2008, pp. 1–5.

[61] M. Rowshan and E. Viterbo, "List Viterbi decoding of PAC codes," *IEEE Transactions on Vehicular Technology*, vol. 70, no. 3, pp. 2428–2435, 2021.

[62] F. Abbasi and E. Viterbo, "Large kernel polar codes with efficient window decoding," *IEEE Transactions on Vehicular Technology*, vol. 69, no. 11, pp. 14 031–14 036, 2020.

[63] F. Abbasi, H. Mahdavifar, and E. Viterbo, "Hybrid non-binary repeated polar codes for low-snr regime," in *2021 IEEE International Symposium on Information Theory (ISIT)*, 2021, pp. 1742–1747.

[64] F. Abbasi, H. Mahdavifar, and E. Viterbo, "Polar coded repetition for low-capacity channels," in *2020 IEEE Information Theory Workshop (ITW)*, 2021, pp. 1–5.

[65] G. He *et al.*, "β-expansion: A theoretical framework for fast and recursive construction of polar codes," in *IEEE Global Communications Conf.*, Singapore, Dec 2017, pp. 1–6.

[66] C. Leroux, A. J. Raymond, G. Sarkis, and W. J. Gross, "A semi-parallel successive-cancellation decoder for polar codes," *IEEE Trans. Signal Process.*, vol. 61, no. 2, pp. 289–299, Jan. 2013.

[67] M. Rowshan and E. Viterbo, "Efficient partial rewind of polar codes' successive cancellation-based decoders for re-decoding attempts," *arXiv preprint arXiv:2109.10466*, 2021.

[68] G. Berhault, C. Leroux, C. Jego, and D. Dallet, "Partial sums generation architecture for successive cancellation decoding of polar codes," in *SiPS 2013 Proceedings*, 2013, pp. 407–412.

[69] Y. Fan and C.-y. Tsui, "An efficient partial-sum network architecture for semi-parallel polar codes decoder implementation," *IEEE Transactions on Signal Processing*, vol. 62, no. 12, pp. 3165–3179, 2014.

[70] S. Lin and D. J. Costello, *Error Control Coding.* Upper Saddle River: Pearson Prentice Hall, 2004, pp. 395–400.

[71] B. Li, H. Zhang, and J. Gu, "On pre-transformed polar codes," *arXiv preprint arXiv:1912.06359*, 2019.

[72] V. Bioglio, F. Gabry, I. Land, and J. Belfiore, "Minimum-distance based construction of multi-kernel polar codes," in *GLOBECOM - IEEE Global Communications Conference*, 2017, pp. 1–6.

[73] V. Bioglio, F. Gabry, I. Land, and J. Belfiore, "Flexible design of multi-kernel polar codes by reliability and distance properties," in *IEEE Intl Symp. on Turbo Codes & Iterative Inf. Processing (ISTC)*, 2018, pp. 1–5.

[74] W. Tong, "Polar code design aspects and future challenges," in *invited talk in special session Polar Codes at 2019 IEEE Inf. Theory Workshop*, Visby, Sweden, 2019.

[75] M. Rowshan, A. Burg, and E. Viterbo, "Polarization-adjusted convolutional (PAC) codes: Sequential decoding vs list decoding," *arXiv preprint arXiv:2002.06805*, 2020.

[76] Y. Polyanskiy, H. V. Poor, and S. Verdu, "Channel coding rate in the finite blocklength regime," *IEEE Transactions on Information Theory*, vol. 56, no. 5, pp. 2307–2359, 2010.

[77] Q. Zhang, A. Liu, X. Pan, and K. Pan, "CRC code design for list decoding of polar codes," *IEEE Communications Letters*, vol. 21, no. 6, pp. 1229–1232, Jun. 2017.

[78] T. K. Moon, *Error Correction Coding: Mathematical Methods and Algorithms.* New Jersey, USA: John Wiley & Sons, 2005, pp. 451–534.

[79] P. Trifonov, "A score function for sequential decoding of polar codes," in *2018 IEEE International Symposium on Information Theory (ISIT)*, Vail, CO, USA, Jun. 2018, pp. 1470–1474.

[80] H. Vangala, E. Viterbo, and Y. Hong, "A comparative study of polar code constructions for the AWGN channel," *arXiv preprint arXiv:1501.02473*, 2015.

[81] R. G. Gallager, *Information Theory and Reliable Communication.* New Jersey, USA: John Wiley & Sons, 1968, pp. 263–286.

[82] M. Jeong and S. Hong, "SC-fano decoding of polar codes," *IEEE Access*, vol. 7, pp. 81 682–81 690, 2019.

[83] M. Sikora and D. J. Costello, "Supercode heuristics for tree search decoding," in *2008 IEEE Inf. Theory Workshop*, Porto, Portugal, 2008, pp. 411–415.

[84] J. Cui, Z. Zhang, X. Zhang, H. Li, and Q. Zeng, "Low-complexity improved progressive bit-flipping decoding for polar codes," in *IEEE 4th International Conference on Computer and Communications (ICCC)*, Chengdu, China, 2018, pp. 39–44.

[85] P. Trifonov and P. Semenov, "Generalized concatenated codes based on polar codes," in *Int. Symp. Wireless Communication Systems*, Aachen, Germany, Nov. 2011, pp. 442–446.

[86] B. Yuan and K. K. Parhi, "Low-latency successive-cancellation polar decoder architectures using 2-bit decoding," *IEEE Trans. on Circuits and Systems—I: Regular Papers*, vol. 61, no. 4, pp. 1241–1254, Apr. 2014.

[87] R. E. Bellman and S. E. Dreyfus, *Applied Dynamic Programming.* Princeton, NJ: Princeton University Press, 1962, pp. 263–286.

[88] K. E. Batcher, "Sorting networks and their applications," in *Proc. AFIPS Spring Joint Comput. Conf.*, vol. 32, 1968, pp. 307–314.

[89] J. Lin and Z. Yan, "Efficient list decoder architecture for polar codes," in *Proc. IEEE Int. Symp. on Circuits and Systems (ISCAS)*, June 2014, p. 1022–1025.

[90] A. Balatsoukas-Stimming, M. B. Parizi, and A. Burg, "On metric sorting for successive cancellation list decoding of polar codes," in *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*, Lisbon, 2015, pp. 1993–1996.

[91] H. Li, "Enhanced metric sorting for successive cancellation list decoding of polar codes," *IEEE Communications Letters*, vol. 22, no. 4, pp. 664–667, April 2018.

[92] X. Wang, T. Wang, J. Li, L. Shan, H. Cao, and Z. Li, "Improved metric sorting for successive cancellation list decoding of polar codes," *IEEE Communications Letters*, vol. 23, no. 7, pp. 1123–1126, July 2019.

[93] E. Liang, H. Yang, D. Divsalar, and R. D. Wesel, "List-decoded tail-biting convolutional codes with distance-spectrum optimal CRCs for 5G," in *2019 IEEE Global Communications Conference (GLOBECOM)*, 2019, pp. 1–6.