List Decoding of Polar Codes

Ido Tal
Technion — Israel Institute of Technology, Haifa, 32000, Israel
idotal@ieee.org

Alexander Vardy
University of California San Diego, La Jolla, CA 92093, USA
avardy@ucsd.edu

***Abstract*—We describe a successive-cancellation *list decoder* for polar codes, which is a generalization of the classic successive-cancellation decoder of Arıkan. In the proposed list decoder, L decoding paths are considered concurrently at each decoding stage, where L is an integer parameter. At the end of the decoding process, the most likely among the L paths is selected as the single codeword at the decoder output. Simulations show that the resulting performance is very close to that of maximum-likelihood decoding, even for moderate values of L. Alternatively, if a genie is allowed to pick the transmitted codeword from the list, the results are comparable to the performance of current state-of-the-art LDPC codes. We show that such a genie can be easily implemented using simple CRC precoding.**

The specific list-decoding algorithm that achieves this performance doubles the number of decoding paths for each information bit, and then uses a pruning procedure to discard all but the L most likely paths. However, straightforward implementation of this algorithm requires $\Omega(Ln^2)$ time, which is in stark contrast with the $O(n \log n)$ complexity of the original successive-cancellation decoder. In this paper, we utilize the structure of polar codes along with certain algorithmic transformations in order to overcome this problem: we devise an efficient, numerically stable, implementation of the proposed list decoder that takes only $O(Ln \log n)$ time and $O(Ln)$ space.

I. Introduction

THE discovery of channel polarization and polar codes by Arıkan [1] is universally recognized as a major breakthrough in coding theory. Polar codes provably achieve the capacity of memoryless symmetric channels, with low encoding and decoding complexity. Moreover, polar codes have an explicit construction (there is no random ensemble to choose from) and a beautiful recursive structure that makes them inherently suitable for efficient implementation in hardware [7], [12].

These remarkable properties of polar codes have generated an enormous interest, see [2], [3], [6], [8], [14], [15] and references therein. Nevertheless, the impact of polar coding in practice has been, so far, negligible. Although polar codes achieve capacity asymptotically, empirical studies indicate that for short and moderate block lengths, successive-cancellation decoding of polar codes does not perform as well as turbo codes or LDPC codes. As we ponder why, we identify two possible causes: either the codes themselves are weak at these lengths, or there is a significant performance gap between successive-cancellation and maximum-likelihood decoding. In fact, the two causes are complementary and, as we shall see, both contribute to the problem.

In this paper, we propose an improvement to the successive-cancellation decoder of [1], namely, a *successive-cancellation list decoder*. Our decoder is governed by a single integer parameter L, which denotes the *list size*. As in [1], we decode the input bits successively one-by-one. However, in the proposed decoder, L decoding paths are considered concurrently at each decoding stage. Specifically, our decoder doubles the number of decoding paths for each information bit u_i to be decoded, thus pursuing both $u_i = 0$ and $u_i = 1$ options, and then uses a pruning procedure to discard all but the L most likely paths. At the end of the decoding process, the most likely among the L decoding paths is selected as the decoder output (thus, in contrast to most list-decoding algorithms in the literature, the output of our decoder is *not* a list but a single codeword).

The performance of the list-decoding algorithm outlined above is encouraging. For example, Figure 1 shows our simulation results for a polar code of rate half and length 2048 on a binary-input AWGN channel, under successive-cancellation decoding and under list decoding. We also include in Figure 1 a lower bound on the probability of word error under maximum-likelihood decoding (such a bound can be readily evaluated in list-decoding simulations). As can be seen from Figure 1, the performance of our list-decoding algorithm is very close to that of maximum-likelihood decoding, even for moderate values of L. The results in Figure 1 are representative: we found that for a wide range of polar codes of various lengths, list decoding effectively bridges the performance gap between successive-cancellation decoding and maximum-likelihood decoding.

Unfortunately, even under maximum-likelihood decoding, the performance of polar codes falls short in comparison to LDPC and turbo codes of comparable length. However, it turns out that we can do much better. We have observed in simulations that with high probability, the transmitted codeword is on the list we generate, but it is *not* the most likely codeword on the list. It is therefore not selected as the decoder output. This means that performance could be

The paper was presented in part at the 2011 IEEE International Symposium on Information Theory, Saint Petersburg, Russia, July 31 – August 5, 2011. Research supported in part by the National Science Foundation grants CCF–1116820 and CCF–1405119 and by the Binational Science Foundation grant 2012016.

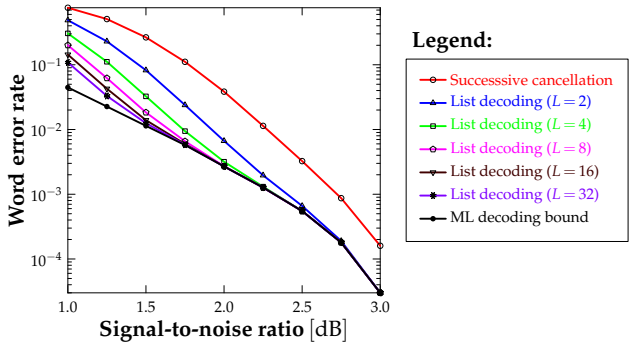


Fig. 1: List-decoding performance for a polar code of length $n = 2048$ and rate $R = 0.5$ on the BPSK-modulated Gaussian channel. The code was constructed using the methods of [15], with optimization for $E_b/N_0 = 2$ dB.

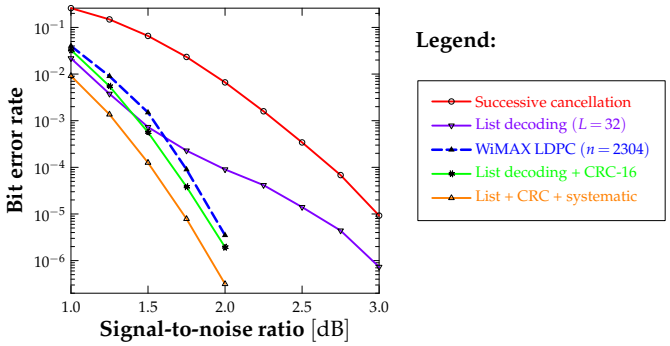


Fig. 2: List-decoding performance of a polar code, with and without CRC precoding, as compared to the LDPC code used in the WiMAX standard. The LDPC code is of length 2304 and rate 0.5, with simulation results taken from [16]. The polar code is the same as in Figure 1, simulated over the BPSK-AWGN channel; list size is $L = 32$ throughout, and CRC is 16 bits long.

further improved if we had a genie aided decoder capable of identifying the transmitted codeword whenever it is on the list. But such a genie can be easily implemented, for example using CRC precoding (see Section V for more details). As can be seen from Figure 2, the resulting BER is lower than the BER achieved with the LDPC code currently used in the WiMAX standard [16]. The situation in Figure 2 is, again, representative: it has been confirmed in numerous simulations that the performance of polar codes under list decoding with CRC is comparable to state-of-the-art turbo and LDPC codes.

Figure 3 summarizes some of these simulation results, using a uniform scale devised by Yury Polyanskiy [10], [11]. It can be seen from Figure 3 that at lengths 512 and 1024, polar coding with CRC performs better than any code currently known, when considering a target error-probability of 10^{-4}.

The discussion in the foregoing paragraph suggests that list decoding of polar codes may hold promise for a number of applications in data communications and storage. However, the proposed list-decoding algorithm presents a significant computational challenge. As shown in Section IV, straightforward implementation of this algorithm requires $\Theta(Ln^2)$ operations, which is in stark contrast with the

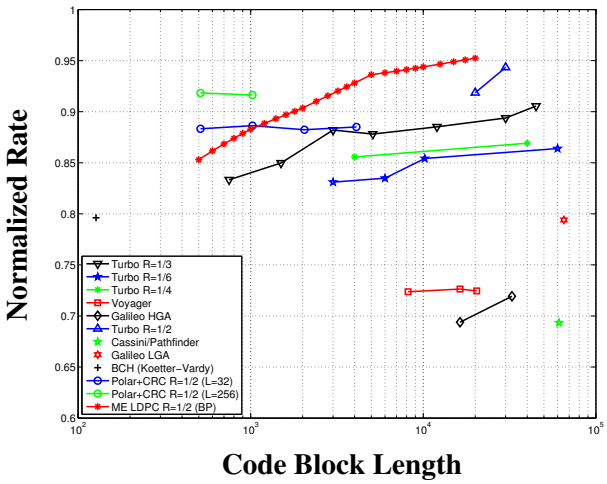


Fig. 3: Comparison of normalized rate for various families of codes. Higher normalized rates correspond to better codes [11]. The points on this plot are obtained as follows. Fix a binary-input AWGN channel and a desired BER of 10^{-4}. Simulate the performance of a code $\mathbb{C}$ of length n and rate R to find the SNR required to achieve this BER. Then use the bounds of Polyanskyi, Poor, and Verdú [11] to find the largest possible coding rate R^* at this SNR, length, and BER. The normalized rate is the ratio R/R^*.

$O(n \log n)$ complexity of Arıkan's successive-cancellation decoder [1]. Indeed, although a $\Theta(Ln^2)$ decoding algorithm is certainly polynomial-time, such decoding complexity would be prohibitive for most applications in practice. One of the main technical contributions of this paper is this: we utilize the recursive structure of polar codes along with certain "lazy-copy" algorithmic techniques [4] in order to overcome this problem. Specifically, we devise an efficient, numerically stable, implementation of the proposed list-decoding algorithm that runs in time $O(Ln \log n)$.

The rest of this paper is organized as follows. In Section II, we describe Arıkan's successive-cancellation decoder. The decoder is formalized in terms of two algorithms (Algorithm 2 and Algorithm 3), in a notation that will become essential in the subsequent sections. In Section III, we show how the space complexity of Arıkan's successive-cancellation decoder can be reduced from $O(n \log n)$ to $O(n)$; this observation will later help us reduce the complexity of the proposed list decoder. In Section IV, we present our main result herein: we describe a successive-cancellation list-decoding algorithm for polar codes, with time complexity $O(Ln \log n)$ and space complexity $O(Ln)$. The algorithm is first illustrated by means of a simple example (see Figure 4 and Figure 5). It is then presented in sufficient detail to allow an independent implementation. Finally, in Section V, we show how list decoding of polar codes can be concatenated with simple CRC precoding to obtain the results reported in Figures 2 and 3.

By necessity, this paper contains a fair amount of algorithmic detail. Thus, on a first reading, the reader is advised to skip directly to Section IV and peruse the first three paragraphs. Doing so will provide a high-level understanding of the list-decoding algorithm proposed in this paper and will also show

why a naive implementation of this algorithm is too costly.

II. Formalization of the Successive Cancellation Decoder

The successive cancellation (SC) decoder is due to Arıkan [1]. In this section, we recast it using our notation, for future reference. Let the polar code under consideration have length $n = 2^m$ and dimension k. Thus, the number of frozen bits is $n - k$. We denote by $\mathbf{u} = (\mathsf{u}_i)_{i=0}^{n-1} = \mathsf{u}_0^{n-1}$ the information bits vector (including the frozen bits), and by $\mathbf{c} = \mathsf{c}_0^{n-1}$ the corresponding codeword, which is sent over a binary-input channel $W : \mathcal{X} \to \mathcal{Y}$, where $\mathcal{X} = \{0, 1\}$. At the other end of the channel, we get the received word $\mathbf{y} = \mathsf{y}_0^{n-1}$. A decoding algorithm is then applied to $\mathbf{y}$, resulting in a decoded codeword $\hat{\mathbf{c}}$ having corresponding information bits $\hat{\mathbf{u}}$. Note that our use of the sans-serif font is reserved only for the above vectors of length n; vectors resulting from recursive definitions will have a different font.

A. An outline of Successive Cancellation

In essence, the SC algorithm is an efficient method of calculating $n = 2^m$ probability pairs, corresponding to n recursively defined channels. A high-level description of the SC decoding algorithm is given in Algorithm 1. In words, for $\varphi = 0, 1, \ldots, n-1$, we must calculate the pair of probabilities $W_m^{(\varphi)}(\mathbf{y}_0^{n-1}, \hat{\mathbf{u}}_0^{\varphi-1}|0)$ and $W_m^{(\varphi)}(\mathbf{y}_0^{n-1}, \hat{\mathbf{u}}_0^{\varphi-1}|1)$, defined shortly. Then, we must make a decision as to the value of $\hat{\mathsf{u}}_\varphi$ according to the more likely probability.

Algorithm 1: A high-level description of the SC decoder

Input: the received vector $\mathbf{y}$
Output: a decoded codeword $\hat{\mathbf{c}}$

for $\varphi = 0, 1, \ldots, n-1$ **do**
  calculate $W_m^{(\varphi)}(\mathbf{y}_0^{n-1}, \hat{\mathbf{u}}_0^{\varphi-1}|0)$ and $W_m^{(\varphi)}(\mathbf{y}_0^{n-1}, \hat{\mathbf{u}}_0^{\varphi-1}|1)$
  if u_φ *is frozen* **then**
    set $\hat{\mathsf{u}}_\varphi$ to the frozen value of u_φ
  else
    if $W_m^{(\varphi)}(\mathbf{y}_0^{n-1}, \hat{\mathbf{u}}_0^{\varphi-1}|0) > W_m^{(\varphi)}(\mathbf{y}_0^{n-1}, \hat{\mathbf{u}}_0^{\varphi-1}|1)$ **then**
      set $\hat{\mathsf{u}}_\varphi \leftarrow 0$
    else
      set $\hat{\mathsf{u}}_\varphi \leftarrow 1$
return the codeword $\hat{\mathbf{c}}$ corresponding to $\hat{\mathbf{u}}$

We now proceed to define *bit-channels* $W_\lambda^{(\varphi)}$, where $0 \le \lambda \le m$ and $0 \le \varphi < 2^\lambda$. In order to aid in the exposition, we name index φ the *phase* and index λ the *layer*. For the sake of brevity, for layer $0 \le \lambda \le m$ denote hereafter

$$\Lambda = 2^\lambda \,. \tag{1}$$

Thus,

$$0 \le \varphi < \Lambda \,. \tag{2}$$

Bit channel $W_\lambda^{(\varphi)}$ is a binary input channel with output alphabet $\mathcal{Y}^\Lambda \times \mathcal{X}^\varphi$, the conditional probability of which we generically denote as

$$W_\lambda^{(\varphi)}(\mathbf{y}_0^{\Lambda-1}, \mathbf{u}_0^{\varphi-1}|u_\varphi) \,. \tag{3}$$

In our context, $\mathbf{y}_0^{\Lambda-1}$ is always a contiguous subvector of the received vector $\mathbf{y}$. We next cite the recursive bit-channel equations [1, Equations (22) and (23)], and note that the "branch" parts will be shortly explained. Let $0 \le 2\psi < \Lambda$, then

$$\begin{aligned} &\overbrace{W_\lambda^{(2\psi)}(\mathbf{y}_0^{\Lambda-1}, \mathbf{u}_0^{2\psi-1}|u_{2\psi})}^{\text{branch } \beta} \\ &= \sum_{u_{2\psi+1}} \frac{1}{2} \underbrace{W_{\lambda-1}^{(\psi)}(\mathbf{y}_0^{\Lambda/2-1}, \mathbf{u}_{0,\text{even}}^{2\psi-1} \oplus \mathbf{u}_{0,\text{odd}}^{2\psi-1}|u_{2\psi} \oplus u_{2\psi+1})}_{\text{branch } 2\beta} \\ &\qquad \cdot \underbrace{W_{\lambda-1}^{(\psi)}(\mathbf{y}_{\Lambda/2}^{\Lambda-1}, \mathbf{u}_{0,\text{odd}}^{2\psi-1}|u_{2\psi+1})}_{\text{branch } 2\beta+1} \end{aligned} \tag{4}$$

and

$$\begin{aligned} &\overbrace{W_\lambda^{(2\psi+1)}(\mathbf{y}_0^{\Lambda-1}, \mathbf{u}_0^{2\psi}|u_{2\psi+1})}^{\text{branch } \beta} \\ &= \frac{1}{2} \underbrace{W_{\lambda-1}^{(\psi)}(\mathbf{y}_0^{\Lambda/2-1}, \mathbf{u}_{0,\text{even}}^{2\psi-1} \oplus \mathbf{u}_{0,\text{odd}}^{2\psi-1}|u_{2\psi} \oplus u_{2\psi+1})}_{\text{branch } 2\beta} \\ &\qquad \cdot \underbrace{W_{\lambda-1}^{(\psi)}(\mathbf{y}_{\Lambda/2}^{\Lambda-1}, \mathbf{u}_{0,\text{odd}}^{2\psi-1}|u_{2\psi+1})}_{\text{branch } 2\beta+1} \end{aligned} \tag{5}$$

with "stopping condition" $W_0^{(0)}(y|u) = W(y|u)$.

B. Detailed description

For Algorithm 1 to become well defined, we must now specify how the probability pair associated with $W_m^{(\varphi)}$ is calculated computationally. As was observed in [1], the calculations implied by the recursions (4) and (5) can be reused in a dynamic programming fashion [4, Chapter 15]. Following this line, we now show an implementation that is straightforward, yet somewhat wasteful in terms of space.

For $\lambda > 0$ and $0 \le \varphi < \Lambda$, recall the recursive definition of $W_\lambda^{(\varphi)}(\mathbf{y}_0^{\Lambda-1}, \mathbf{u}_0^{\varphi-1}|u_\varphi)$ given in either (4) or (5), depending on the parity of φ. For either $\varphi = 2\psi$ or $\varphi = 2\psi+1$, the channel $W_{\lambda-1}^{(\psi)}$ is evaluated with output $(\mathbf{y}_0^{\Lambda/2-1}, \mathbf{u}_{0,\text{even}}^{2\psi-1} \oplus \mathbf{u}_{0,\text{odd}}^{2\psi-1})$, as well as with output $(\mathbf{y}_{\Lambda/2}^{\Lambda-1}, \mathbf{u}_{0,\text{odd}}^{2\psi-1})$. Since our algorithm will make use of these recursions, we need a simple way of defining which output we are referring to. We do this by specifying, apart from the layer λ and the phase φ which define the channel, the *branch* number

$$0 \le \beta < 2^{m-\lambda} \,. \tag{6}$$

Definition 1 (Association of branch number with output): Since, during the run of the SC algorithm, the last-layer channel $W_m^{(\varphi)}$ is only evaluated with a single output[1], $(\mathbf{y}_0^{n-1}, \hat{\mathbf{u}}_0^{\varphi-1})$, we give a branch number of $\beta = 0$ to each such output. Next, we proceed recursively as follows. For $\lambda > 0$, consider a channel $W_\lambda^{(\varphi)}$ with output $(\mathbf{y}_0^{\Lambda-1}, \hat{\mathbf{u}}_0^{\varphi-1})$ and corresponding branch number β. Denote $\psi = \lfloor \varphi/2 \rfloor$. The output $(\mathbf{y}_0^{\Lambda/2-1}, \hat{\mathbf{u}}_{0,\text{even}}^{2\psi-1} \oplus \hat{\mathbf{u}}_{0,\text{odd}}^{2\psi-1})$ associated with $W_{\lambda-1}^{(\psi)}$ will have a branch number of 2β, while the output $(\mathbf{y}_{\Lambda/2}^{\Lambda-1}, \hat{\mathbf{u}}_{0,\text{odd}}^{2\psi-1})$

will have a branch number of $2\beta + 1$. Finally, we mention that for the sake of brevity, we will talk about the output corresponding to *branch β of a channel*, although this is slightly inaccurate.

We now introduce our first data structure. For each layer $0 \leq \lambda \leq m$, we will have a *probabilities array*, denoted by P_λ, indexed by an integer $0 \leq i < 2^m$ and a bit $b \in \{0, 1\}$. For a given layer λ, an index i will correspond to a phase $0 \leq \varphi < \Lambda$ and branch $0 \leq \beta < 2^{m-\lambda}$ using the following quotient/reminder representation.

$$i = \langle \varphi, \beta \rangle_\lambda = \varphi + 2^\lambda \cdot \beta \ . \tag{7}$$

In order to avoid repetition, we use the following shorthand

$$P_\lambda[\langle \varphi, \beta \rangle] = P_\lambda[\langle \varphi, \beta \rangle_\lambda] \ . \tag{8}$$

The probabilities array data structure P_λ will be used as follows. Let a layer $0 \leq \lambda \leq m$, phase $0 \leq \varphi < \Lambda$, and branch $0 \leq \beta < 2^{m-\lambda}$ be given. Denote the output corresponding to branch β of $W_\lambda^{(\varphi)}$ as $(\mathbf{y}_0^{\Lambda-1}, \hat{\mathbf{u}}_0^{\varphi-1})$. Then, ultimately, we will have for both values of b that

$$P_\lambda[\langle \varphi, \beta \rangle][b] = W_\lambda^{(\varphi)}(\mathbf{y}_0^{\Lambda-1}, \hat{\mathbf{u}}_0^{\varphi-1} | b) \ . \tag{9}$$

Definition 2 (Association of branch number with input): Analogously to defining the output corresponding to a branch β, we now define the input corresponding to a branch. As in the "output" case, we start at layer m and continue recursively according to (4) and (5). That is, consider the channel $W_m^{(\varphi)}$. Let $\hat{\mathsf{u}}_\varphi$ be the corresponding input which Algorithm 1 sets, either in line 7 or in line 9. With respect to $W_m^{(\varphi)}$, we let this input have a branch number of $\beta = 0$. Next, we proceed recursively as follows. For layer $\lambda > 0$ and $0 \leq \psi < 2^{\lambda-1}$, let $u_{2\psi}$ and $u_{2\psi+1}$ be the inputs corresponding to branch $0 \leq \beta < 2^{m-\lambda}$ of $W_\lambda^{(2\psi)}$ and $W_\lambda^{(2\psi+1)}$, respectively. Then, in light of (5), we define the inputs corresponding to branches 2β and $2\beta + 1$ of $W_{\lambda-1}^{(\psi)}$ as $u_{2\psi} \oplus u_{2\psi+1}$ and $u_{2\psi+1}$, respectively.

The following lemma points at the natural meaning that a branch number has at layer $\lambda = 0$. It is proved using a straightforward induction.

Lemma 1: Let $\mathbf{y}$ and $\hat{\mathbf{c}}$ be as in Algorithm 1, the received vector and the decoded codeword. Consider layer $\lambda = 0$, and thus set $\varphi = 0$. Next, fix a branch number $0 \leq \beta < 2^n$. Then, the input and output corresponding to branch β of $W_0^{(0)}$ are y_β and $\hat{\mathsf{c}}_\beta$, respectively.

We now introduce our second, and last, data structure for this section. For each layer $0 \leq \lambda \leq m$, we will have a *bit array*, denoted by B_λ, and indexed by an integer $0 \leq i < 2^m$, as in (7). The data structure will be used as follows. Let layer $0 \leq \lambda \leq m$, phase $0 \leq \varphi < \Lambda$, and branch $0 \leq \beta < 2^{m-\lambda}$ be given. Denote the input corresponding to branch β of $W_\lambda^{(\varphi)}$ as $\hat{u}(\lambda, \varphi, \beta)$. Then, we will ultimately have that

$$B_\lambda[\langle \varphi, \beta \rangle] = \hat{u}(\lambda, \varphi, \beta) \ , \tag{10}$$

where we have used the same shorthand as in (8). Notice that the total memory consumed by our algorithm is $O(n \log n)$.

Our first implementation of the SC decoder is given as Algorithms 2–4. The main loop is given in Algorithm 2, and follows the high-level description given in Algorithm 1. Note that the elements of the probabilities arrays P_λ and bit array B_λ start-out uninitialized, and become initialized as the algorithm runs its course. The code to initialize the array values is given in Algorithms 3 and 4.

Algorithm 2: First implementation of SC decoder

Input: the received vector $\mathbf{y}$
Output: a decoded codeword $\hat{\mathbf{c}}$

for $\beta = 0, 1, \ldots, n-1$ **do** // Initialization
  $P_0[\langle 0, \beta \rangle][0] \leftarrow W(\mathsf{y}_\beta | 0)$, $P_0[\langle 0, \beta \rangle][1] \leftarrow W(\mathsf{y}_\beta | 1)$
for $\varphi = 0, 1, \ldots, n-1$ **do** // Main loop
  recursivelyCalcP(m, φ)
  if u_φ *is frozen* **then**
    set $B_m[\langle \varphi, 0 \rangle]$ to the frozen value of u_φ
  else
    if $P_m[\langle \varphi, 0 \rangle][0] > P_m[\langle \varphi, 0 \rangle][1]$ **then**
      set $B_m[\langle \varphi, 0 \rangle] \leftarrow 0$
    else
      set $B_m[\langle \varphi, 0 \rangle] \leftarrow 1$
  if $\varphi \mod 2 = 1$ **then**
    recursivelyUpdateB(m, φ)
return the decoded codeword: $\hat{\mathbf{c}} = (B_0[\langle 0, \beta \rangle])_{\beta=0}^{n-1}$

Algorithm 3: recursivelyCalcP(λ, φ) *implementation I*

Input: layer λ and phase φ

if $\lambda = 0$ **then return** // Stopping condition
set $\psi \leftarrow \lfloor \varphi / 2 \rfloor$
// Recurse first, if needed
if $\varphi \mod 2 = 0$ **then** recursivelyCalcP$(\lambda - 1, \psi)$
for $\beta = 0, 1, \ldots, 2^{m-\lambda} - 1$ **do** // calculation
  if $\varphi \mod 2 = 0$ **then** // apply Equation (4)
    for $u' \in \{0, 1\}$ **do**
      $P_\lambda[\langle \varphi, \beta \rangle][u'] \leftarrow \sum_{u''} \frac{1}{2} P_{\lambda-1}[\langle \psi, 2\beta \rangle][u' \oplus u''] \cdot P_{\lambda-1}[\langle \psi, 2\beta + 1 \rangle][u'']$
  else // apply Equation (5)
    set $u' \leftarrow B_\lambda[\langle \varphi - 1, \beta \rangle]$
    for $u'' \in \{0, 1\}$ **do**
      $P_\lambda[\langle \varphi, \beta \rangle][u''] \leftarrow \frac{1}{2} P_{\lambda-1}[\langle \psi, 2\beta \rangle][u' \oplus u''] \cdot P_{\lambda-1}[\langle \psi, 2\beta + 1 \rangle][u'']$

Algorithm 4: recursivelyUpdateB(λ, φ) *implementation I*

Require : φ is odd

set $\psi \leftarrow \lfloor \varphi / 2 \rfloor$
for $\beta = 0, 1, \ldots, 2^{m-\lambda} - 1$ **do**
  $B_{\lambda-1}[\langle \psi, 2\beta \rangle] \leftarrow B_\lambda[\langle \varphi - 1, \beta \rangle] \oplus B_\lambda[\langle \varphi, \beta \rangle]$
  $B_{\lambda-1}[\langle \psi, 2\beta + 1 \rangle] \leftarrow B_\lambda[\langle \varphi, \beta \rangle]$
if $\psi \mod 2 = 1$ **then**
  recursivelyUpdateB$(\lambda - 1, \psi)$

[1]Recall that since the sans-serif font is used, $\mathbf{y} = \mathbf{y}_0^{n-1}$ is the received word corresponding to the codeword $\mathbf{c} = \mathbf{c}_0^{n-1}$ sent over the physical channel, while $\hat{\mathbf{u}}_0^{\varphi-1}$ corresponds to the first φ information plus frozen bits defining the decoded codeword: $\hat{\mathbf{c}} = G\hat{\mathbf{u}}$, where G is the $n \times n$ Arıkan generator matrix.

Lemma 2: Algorithms 2–4 are a valid implementation of the SC decoder.

Proof: We first note that in addition to proving the claim explicitly stated in the lemma, we must also prove an implicit claim. Namely, we must prove that the actions taken by the algorithm are well defined. Specifically, we must prove that when an array element is read from, it was already written to (it is initialized).

Both the implicit and explicit claims are easily derived from the following observation. For a given $0 \leq \varphi < n$, consider iteration φ of the main loop in Algorithm 2. Fix a layer $0 \leq \lambda \leq m$, and a branch $0 \leq \beta < 2^{m-\lambda}$. If we suspend the run of the algorithm just after the iteration ends, then (9) holds with φ' instead of φ, for all

$$0 \leq \varphi' \leq \left\lfloor \frac{\varphi}{2^{m-\lambda}} \right\rfloor .$$

Similarly, (10) holds with φ' instead of φ, for all

$$0 \leq \varphi' < \left\lfloor \frac{\varphi+1}{2^{m-\lambda}} \right\rfloor .$$

The above observation is proved by induction on φ. ∎

III. Space-Efficient Successive Cancellation Decoding

The running time of the SC decoder is $O(n \log n)$, and our implementation is no exception. As we have previously noted, the space complexity of our algorithm is $O(n \log n)$ as well. However, we will now show how to bring the space complexity down to $O(n)$. The observation that one can reduce the space complexity to $O(n)$ was noted, in the context of VLSI design, in [7].

As a first step towards this end, consider the probability pair array P_m. By examining the main loop in Algorithm 2, we quickly see that if we are currently at phase φ, then we will never again make use of $P_m[\langle \varphi', 0 \rangle]$ for all $\varphi' < \varphi$. On the other hand, we see that $P_m[\langle \varphi'', 0 \rangle]$ is uninitialized for all $\varphi'' > \varphi$. Thus, instead of reading and writing to $P_m[\langle \varphi, 0 \rangle]$, we can essentially disregard the phase information, and use only the first element $P_m[0]$ of the array, discarding all the rest. By the recursive nature of polar codes, this observation — disregarding the phase information — can be exploited for a general layer λ as well. The following lemma makes the above claims formal. The proof follows easily from Line 2 of Algorithm 3, and by noting that

$$\lfloor \lfloor \varphi / 2^i \rfloor / 2 \rfloor = \lfloor \varphi / 2^{i+1} \rfloor .$$

Lemma 3: During iteration φ of the main loop of Algorithm 2, the only elements of the arrays P_λ which are possibly read from or written to have the form $P_\lambda[\langle \varphi', \beta \rangle][0]$ and $P_\lambda[\langle \varphi', \beta \rangle][1]$, where $0 \leq \lambda \leq m$, $0 \leq \varphi < 2^\lambda$, $0 \leq \beta < 2^{m-\lambda}$, and $\varphi' = \lfloor \varphi / 2^{m-\lambda} \rfloor$.

With the above lemma at hand, and since $\lfloor \varphi / 2^i \rfloor$ is a non-decreasing function of φ, we are justified to carry out the following alternation of the algorithm. For all $0 \leq \lambda \leq m$, let us now define the number of elements in P_λ to be $2^{m-\lambda}$. Accordingly,

$$P_\lambda[\langle \varphi, \beta \rangle] \text{ is replaced by } P_\lambda[\beta] . \tag{11}$$

This change does not affect the final output of the algorithm.

Note that the total space needed to hold the P arrays has gone down from $O(n \log n)$ to $O(n)$. We would now like to do the same for the B arrays. However, as things are currently stated, we can not disregard the phase, as can be seen for example in Line 3 of Algorithm 4. The solution is a simple renaming. As a first step, let us define for each $0 \leq \lambda \leq m$ an array C_λ consisting of bit pairs and having length $n/2$. Next, let a generic reference of the form $B_\lambda[\langle \varphi, \beta \rangle]$ be replaced by $C_\lambda[\psi + \beta \cdot 2^{\lambda-1}][\varphi \mod 2]$, where $\psi = \lfloor \varphi/2 \rfloor$. Note that we have done nothing more than rename the elements of B_λ as elements of C_λ. However, we now see that as before we can disregard the value of ψ and take note only of the parity of φ (the proof is essentially the same as before, and left to the reader). So, let us make one more substitution: replace every instance of $C_\lambda[\psi + \beta \cdot 2^{\lambda-1}][\varphi \mod 2]$ by $C_\lambda[\beta][\varphi \mod 2]$, and resize each array C_λ to have $2^{m-\lambda}$ bit pairs. To sum up,

$$B_\lambda[\langle \varphi, \beta \rangle] \text{ is replaced by } C_\lambda[\beta][\varphi \mod 2] . \tag{12}$$

The alert reader will notice that a further reduction in space is possible: for $\lambda = 0$ we will always have that $\varphi = 0$, and thus the parity of φ is always even. However, this reduction does not affect the asymptotic space complexity which is now indeed down to $O(n)$.

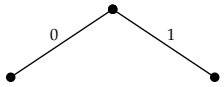
We end this subsection by mentioning that although we were concerned here with reducing the *space* complexity of our SC decoder, the observations made with this goal in mind will be of great use in analyzing the *time* complexity of our list decoder.

IV. Successive Cancellation List Decoder

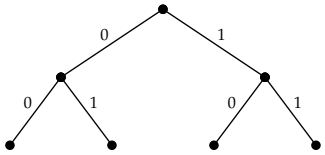
In this section we introduce and define our algorithm, the successive cancellation list (SCL) decoder. Our list decoder has a parameter L, called the *list size*. Generally speaking, larger values of L mean lower error rates but longer running times and larger memory usage. We note at this point that successive cancellation list decoding is not a new idea: it was applied in [5] to Reed-Muller codes

Recall the main loop of an SC decoder, where at each phase we must decide on the value of $\hat{u}_\varphi$. In an SCL decoder, instead of deciding to set the value of an unfrozen $\hat{u}_\varphi$ to either a 0 or a 1, we inspect both options. That is, let a "path" be a certain decision on the values of $\hat{\mathbf{u}}_0^\varphi$, for $0 \leq \varphi < n$. When decoding a non-frozen bit $\hat{u}_{\varphi+1}$, we split the decoding path $\hat{\mathbf{u}}_0^\varphi$ into two paths (see Figure 4). Both of the new paths will have $\hat{\mathbf{u}}_0^\varphi$ as a prefix. One new path will end with "0" while the other ends in "1". Since each split doubles the number of paths to be examined, we must prune them, and the maximum number of paths allowed is the specified list size, L. Naturally, we would like to keep the "best" paths at each stage, and thus require a pruning criterion. Our pruning criterion will be to keep the most likely paths.

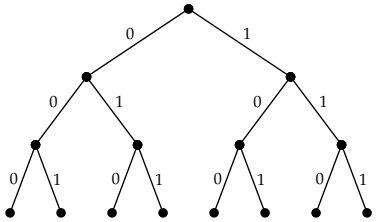
Figure 5 considers the same decoding run depicted in Figure 4 and tracks the evolution of how the C_λ arrays are allocated and used. Each sub-figure represents the state of the C_λ arrays at a different stage (but note that the jth subfigure in Figure 4 generally *does not* correspond to



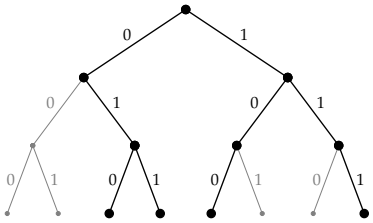
(i) Algorithm starts. First unfrozen bit can be either 0 or 1.



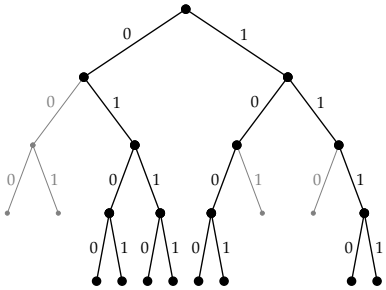
(ii) Algorithm continues. Second unfrozen bits can be either 0 or 1. The number of paths is not more than $L = 4$, so no need to prune yet.



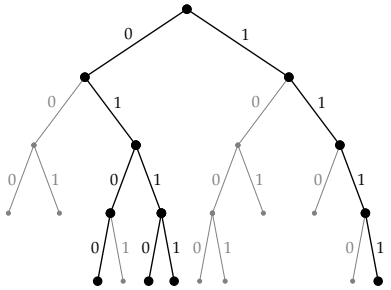
(iii) Considering all options for first, second, and third bits results in 8 decoding paths; too much, since $L = 4$.



(iv) Prune the 8 paths into $L = 4$ most promising paths.

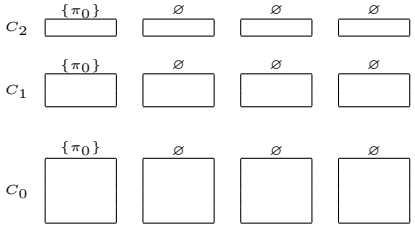


(v) Continue the 4 active paths by considering both options of the fourth unfrozen bit. The number of paths doubles to 8, which is too much ($L = 4$).

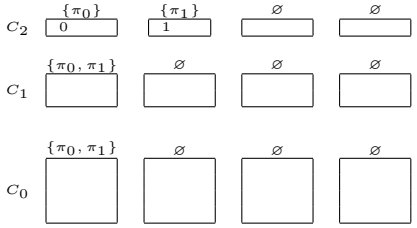


(vi) Again, prune to $L = 4$ best paths.

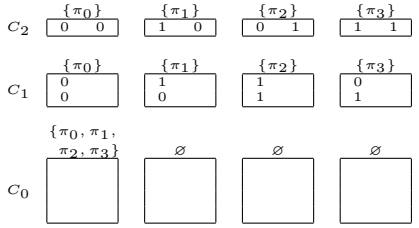
Fig. 4: Evolution of decoding paths. We assume for simplicity that $n = 4$ and all bits are unfrozen. The list size is $L = 4$: each level has at most 4 nodes with paths that continue downward. Discontinued paths are colored gray.



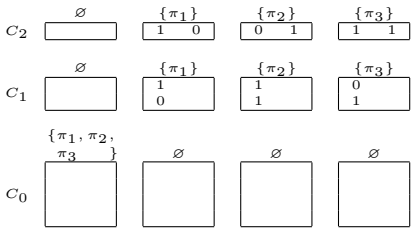
(a) $\varphi = 0$, state at the start of main loop. One active path, π_0, which is empty.



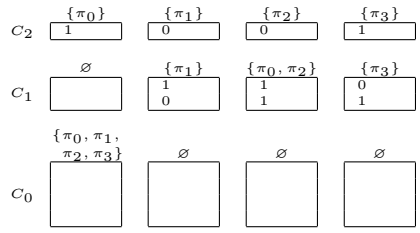
(b) $\varphi = 0$, state at the end of the main loop. Two active paths, $\pi_0 = 0$, $\pi_1 = 1$.



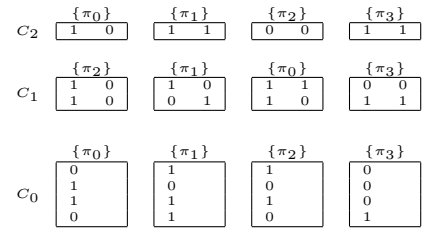
(c) $\varphi = 1$, state at the end of the main loop. Four active paths, $\pi_0 = 00$, $\pi_1 = 10$, $\pi_2 = 01$, $\pi_3 = 11$. Also state when $\varphi = 2$ and Line 7 of the main loop has executed.



(d) $\varphi = 2$, state after Line 14 of the main loop has been called, and we are before the first execution of Line 19 in Algorithm 13. At this stage, π_0 has been killed off, and the only active paths are $\pi_1 = 10$, $\pi_2 = 01$, $\pi_3 = 11$.



(e) $\varphi = 2$, state at the end of the main loop. Four active paths, $\pi_0 = 011$, $\pi_1 = 100$, $\pi_2 = 010$, $\pi_3 = 111$.



(f) $\varphi = 3$, state at the end of the main loop. Four active paths, $\pi_0 = 0110$, $\pi_1 = 0111$, $\pi_2 = 0100$, $\pi_3 = 1111$.

Fig. 5: Evolution of the usage of C_λ arrays. The example run is the same as in Figure 4.

the jth subfigure in Figure 5). Paths over an array represent the assignment information encoded (redundantly) in the activePath, pathIndexToArrayIndex, inactiveArrayIndices, and arrayReferenceCount data structures. A "$\varnothing$" designates an array not assigned to any path. We now expand on the various sub-figures of Figure 5. Note that the following list contains references to algorithms defined later on; on a first read, the message we want to get across is that paths with a common prefix can typically share non-empty arrays. This is clearly seen, for example, in Subfigure 5(e). In what follows, when we refer to the "main loop", we mean the for-loop in Algorithm 12.

(a) Here, $\varphi = 0$ and we are at the start of main loop. We have one active path, π_0, which is empty. The path was assigned C_2, C_1 and C_0 arrays by the `assignInitialPath` function on Line 2 of Algorithm 12, before the main loop started.

(b) Now, we still have $\varphi = 0$, but have reached the end of the main loop. There are two active paths, $\pi_0 = 0$, $\pi_1 = 1$ (as is also depicted in Subfigure 4(i)). The current path π_1 is the result of the `clonePath` operation applied to the path π_0 of the previous subfigure. This operation was carried out on Line 25 of Algorithm 13. Initially, both paths shared all C arrays, as specified in the `clonePath` operation. However, at the current stage, both paths have private C_2 arrays, because of the `getArrayPointer_C` call on Line 26 of Algorithm 13.

(c) We are at the end of the main loop, with $\varphi = 1$. We have four active paths, $\pi_0 = 00$, $\pi_1 = 10$, $\pi_2 = 01$, $\pi_3 = 11$ (Subfigure 4(ii)). The current π_2 path is the result of applying `clonePath` on the previous π_0, in Line 25 of Algorithm 13. The same holds true with respect to the current π_3 and the previous π_1. As in the previous sub-figure, each path has a distinct C_2 array, by virtue of the `getArrayPointer_C` call on Line 26 of Algorithm 13. Since $\varphi = 1$, `recursivelyUpdateC` is called on Line 16 of Algorithm 12, resulting in each path updating (Lines 7–8, Algorithm 11) a private copy (Line 5, Algorithm 11) of C_1.
We next note that sub-figure 5(c) also represents the state we are at when $\varphi = 2$ and Line 7 of the main loop has executed. We might expect the state to have changed, following the call to `recursivelyCalcP`, which contains a call to `getArrayPointer_C` (Line 9, Algorithm 10). Moreover, since $\varphi = 2$, `recursivelyCalcP` calls itself recursively once, which results in a second set of calls to `getArrayPointer_C`. However, by the previous paragraph, we know that at the end of the main loop each path already had a private copy of its C_2 and C_1 arrays. Thus, the calls to `getArrayPointer_C` do not change anything in this respect.

(d) We are now in the middle of executing Line 14 of the main loop, with $\varphi = 2$ (see Subfigure 4(iv) for a partial depiction). Namely, the "kill-loop" in Algorithm 13 has just finished, and we are before the first execution of Line 19 in Algorithm 13. At this stage, π_0 has been killed off, and the only active paths are $\pi_1 = 10$, $\pi_2 = 01$, $\pi_3 = 11$. This results in free C_2 and C_1 arrays. Array C_2 will be re-assigned shortly, followed by C_1.

(e) End of the main loop, with $\varphi = 2$ (Subfigure 4(iv)). We are back to four active paths, $\pi_0 = 011$, $\pi_1 = 100$, $\pi_2 = 010$, $\pi_3 = 111$. As mentioned, the previous π_0 was killed (Line 18, Algorithm 13). On the other hand, the current π_0 and π_2 are decedents of the previous π_2. Namely, the current π_0 is the result of applying `clonePath` on previous π_2. Thus, both π_0 and π_2 share the same C_2 array. However, the current π_0 is assigned a private C_2 array, de-assigned by the previous kill operation. The current path π_1 is a continuation of the previous π_1 (only one branch was followed, in Line 30 of Algorithm 13). The same is true with respect to the current and previous π_3, with Line 32 in place of 30.
Note that each path is currently encoded in the C_2 and C_1 arrays. For example, consider π_0. Since the C_2 array assigned to it has first entry equal to 1, we deduce that the corresponding u_2 bit equals 1. Since the C_1 array has entries $\alpha = 1, \beta = 1$, we deduce that $u_0 = \alpha + \beta = 0$ and $u_1 = \beta = 1$.

(f) End of main loop, with $\varphi = 3$ (Subfigure 4(vi)). Four active paths, $\pi_0 = 0110$, $\pi_1 = 0111$, $\pi_2 = 0100$, $\pi_3 = 1111$. As before, one path was killed (previous π_1), one path was cloned and split into two paths (previous π_0 split into current π_0 and current π_1), and two paths had one surviving branch each (π_2 and π_3). Note that the function `recursivelyUpdateC` is called in Line 16 of Algorithm 12. Since $\varphi = 3$ this results in two recursive calls. Thus, each path has a private copy of C_2, C_1, and C_0. The codewords corresponding to each path are now stored in the C_0 arrays.

Consider the following outline for a naive implementation of an SCL decoder. Each time a decoding path is split into two forks, the data structures used by the "parent" path are duplicated, with one copy given to the first fork and the other to the second. Since the number of splits is $\Omega(L \cdot n)$, and since the size of the data structures used by each path is $\Omega(n)$, the copying operation alone would take time $\Omega(L \cdot n^2)$. In fact, the running time is easily seen to be $\Theta(L \cdot n^2)$. This running time is clearly impractical for all but the shortest of codes. However, all known (to us) implementations of successive cancellation list decoding have complexity at least $\Omega(L \cdot n^2)$. Our main contribution in this section is the following: we show how to implement SCL decoding with time complexity $O(L \cdot n \log n)$ instead of $\Omega(L \cdot n^2)$.

The key observation is as follows. Consider the P arrays of the last section, and recall that the size of P_λ is proportional to $2^{m-\lambda}$. Thus, the cost of copying P_λ grows exponentially small with λ. Next, consider the main loop of Algorithm 2. Unwinding the recursion, we see that P_λ is accessed only every $2^{m-\lambda}$ increments of φ. Obviously, this is still the case after the size-reduction replacment given in (11) is carried out. Put another way, the bigger P_λ is, the less frequently it is accessed. The same observation applies to the C arrays.

This observation suggest the use of a so-called "lazy-copy" implementation. Briefly, at each given stage, the same array may be *flagged as belonging to more than one decoding path*. However, when a given decoding path needs access to an array it is sharing with another path, a copy is made. The copy is initially private, belonging only to the path that must access it. However, as the algorithm progresses (φ increases), the array may be shared by several paths that are descendants (continuations) of the path which needed initial private access. The following sub-sections are concerned with exactly specifying how this lazy-copy is implemented, as well as with proving the implementation is valid and analyzing its performance.

A. Low-level functions

We now discuss the low-level functions and data structures by which the "lazy-copy" methodology is realized. We note in advance that since our aim was to keep the exposition as simple as possible, we have avoided some obvious optimizations. The following data structures are defined and initialized in Algorithm 5.

Algorithm 5: `initializeDataStructures()`

inactivePathIndices $\leftarrow$ **new** stack with capacity L
activePath $\leftarrow$ **new** boolean array of size L
arrayPointer_P $\leftarrow$ **new** 2-D array of size $(m+1) \times L$, the elements of which are array pointers
arrayPointer_C $\leftarrow$ **new** 2-D array of size $(m+1) \times L$, the elements of which are array pointers
pathIndexToArrayIndex $\leftarrow$ **new** 2-D array of size $(m+1) \times L$
inactiveArrayIndices $\leftarrow$ **new** array of size $m+1$, the elements of which are stacks with capacity L
arrayReferenceCount $\leftarrow$ **new** 2-D array of size $(m+1) \times L$
`// Initialization of data structures`
for $\lambda = 0, 1, \ldots, m$ **do**
  for $s = 0, 1, \ldots, L-1$ **do**
    arrayPointer_P$[\lambda][s] \leftarrow$ **new** array of float pairs of size $2^{m-\lambda}$
    arrayPointer_C$[\lambda][s] \leftarrow$ **new** array of bit pairs of size $2^{m-\lambda}$
    arrayReferenceCount$[\lambda][s] \leftarrow 0$
    push(inactiveArrayIndices$[\lambda], s$)
for $\ell = 0, 1, \ldots, L-1$ **do**
  activePath$[\ell] \leftarrow$ **false**
  push(inactivePathIndices, ℓ)

Each path will have an index ℓ, where $0 \leq \ell < L$. At first, only one path will be active. As the algorithm runs its course, paths will change states between "active" and "inactive". The inactivePathIndices stack [4, Section 10.1] will hold the indices of the inactive paths. We assume the "array" implementation of a stack, in which both "push" and "pop" operations take $O(1)$ time and a stack of capacity L takes $O(L)$ space. The activePath array is a boolean array such that activePath$[\ell]$ is true iff path ℓ is active. Note that, essentially, both inactivePathIndices and activePath store the same information. The utility of this redundancy will be made clear shortly.

For every layer λ, we will have a "bank" of L probability-pair arrays for use by the active paths. At any given moment, some of these arrays might be used by several paths, while others might not be used by any path. Each such array is pointed to by an element of arrayPointer_P. Likewise, we will have a bank of bit-pair arrays, pointed to by elements of arrayPointer_C.

The pathIndexToArrayIndex array is used as follows. For a given layer λ and path index ℓ, the probability-pair array and bit-pair array corresponding to layer λ of path ℓ are pointed to by

$$\text{arrayPointer\_P}[\lambda][\text{pathIndexToArrayIndex}[\lambda][\ell]]$$

and

$$\text{arrayPointer\_C}[\lambda][\text{pathIndexToArrayIndex}[\lambda][\ell]] \ ,$$

respectively.

Recall that at any given moment, some probability-pair and bit-pair arrays from our bank might be used by multiple paths, while others may not be used by any. The value of arrayReferenceCount$[\lambda][s]$ denotes the number of paths currently using the array pointed to by arrayPointer_P$[\lambda][s]$. Note that this is also the number of paths making use of arrayPointer_C$[\lambda][s]$. The index s is contained in the stack inactiveArrayIndices$[\lambda]$ iff arrayReferenceCount$[\lambda][s]$ is zero.

Now that we have discussed how the data structures are initialized, we continue and discuss the low-level functions by which paths are made active and inactive. We start by mentioning Algorithm 6, by which the initial path of the algorithm is assigned and allocated. In words, we choose a path index ℓ that is not currently in use (none of them are), and mark it as used. Then, for each layer λ, we mark (through pathIndexToArrayIndex) an index s such that both arrayPointer_P$[\lambda][s]$ and arrayPointer_C$[\lambda][s]$ are allocated to the current path.

Algorithm 6: `assignInitialPath()`

Output: index ℓ of initial path

$\ell \leftarrow$ **pop**(inactivePathIndices)
activePath$[\ell] \leftarrow$ **true**
`// Associate arrays with path index`
for $\lambda = 0, 1, \ldots, m$ **do**
  $s \leftarrow$ **pop**(inactiveArrayIndices$[\lambda]$)
  pathIndexToArrayIndex$[\lambda][\ell] \leftarrow s$
  arrayReferenceCount$[\lambda][s] \leftarrow 1$
return ℓ

Algorithm 7 is used to clone a path — the final step before splitting that path in two. The logic is very similar to that of Algorithm 6, but now we make the two paths share bit-arrays and probability arrays.

Algorithm 8 is used to terminate a path, which is achieved by marking it as inactive. After this is done, the arrays marked as associated with the path must be dealt with as follows. Since the path is inactive, we think of it as not having any associated arrays, and thus all the arrays that were previously associated with the path must have their reference count decreased by one. The goal of all previously discussed low-level functions was essentially

Algorithm 7: `clonePath`(ℓ)

Input: index ℓ of path to clone
Output: index ℓ' of copy

```
1 ℓ' ← pop(inactivePathIndices)
2 activePath[ℓ'] ← true
  // Make ℓ' reference same arrays as ℓ
3 for λ = 0, 1, ..., m do
4     s ← pathIndexToArrayIndex[λ][ℓ]
5     pathIndexToArrayIndex[λ][ℓ'] ← s
6     arrayReferenceCount[λ][s]++
7 return ℓ'
```

Algorithm 8: `killPath`(ℓ)

Input: index ℓ of path to kill

```
  // Mark the path index ℓ as inactive
1 activePath[ℓ] ← false
2 push(inactivePathIndices, ℓ)
  // Disassociate arrays with path index
3 for λ = 0, 1, ..., m do
4     s ← pathIndexToArrayIndex[λ][ℓ]
5     arrayReferenceCount[λ][s]−−
6     if arrayReferenceCount[λ][s] = 0 then
7         push(inactiveArrayIndices[λ], s)
```

to enable the abstraction implemented by the functions `getArrayPointer_P` and `getArrayPointer_C`. The function `getArrayPointer_P` is called each time a higher-level function needs to access (either for reading or writing) the probability-pair array associated with a certain path ℓ and layer λ. The implementation of `getArrayPointer_P` is give in Algorithm 9. There are two cases to consider: either the array is associated with more than one path or it is not. If it is not, then nothing needs to be done, and we return a pointer to the array. On the other hand, if the array is shared, we make a private copy for path ℓ, and return a pointer to that copy. By doing so, we ensure that two paths will never write to the same array. The function `getArrayPointer_C` is used in the same manner for bit-pair arrays, and has exactly the same implementation, up to the obvious changes.

At this point, we remind the reader that we are deliberately sacrificing speed for simplicity. Namely, each such function is called either before reading or writing to an array, but the copy operation is really needed only before writing.

We have now finished defining almost all of our low-level functions. At this point, we should specify the constraints one should follow when using them and what one can expect if these constraints are met. We start with the former.

Definition 3 (Valid calling sequence): Consider a sequence $(f_t)_{t=0}^{T}$ of $T+1$ calls to the low-level functions implemented in Algorithms 5–9. We say that the sequence is *valid* if the following traits hold.

Initialized: The one and only index t for which f_t is equal to `initializeDataStructures` is $t = 0$. The one and only index t for which f_t is equal to `assignInitialPath` is $t = 1$.

Balanced: For $1 \leq t \leq T$, denote the number of times the function `clonePath` was called up to and including stage t as

$$\#^{(t)}_{\texttt{clonePath}} = |\left\{1 \leq i \leq t \ : \ f_i \text{ is } \texttt{clonePath}\right\}| \ .$$

Algorithm 9: `getArrayPointer_P`(λ, ℓ)

Input: layer λ and path index ℓ
Output: pointer to corresponding probability pair array

```
  // getArrayPointer_C(λ, ℓ) is defined
     identically, up to the obvious changes
     in lines 6 and 10
1 s ← pathIndexToArrayIndex[λ][ℓ]
2 if arrayReferenceCount[λ][s] = 1 then
3     s' ← s
4 else
5     s' ← pop(inactiveArrayIndices[λ])
6     copy the contents of the array pointed to by
      arrayPointer_P[λ][s] into that pointed to by
      arrayPointer_P[λ][s']
7     arrayReferenceCount[λ][s]−−
8     arrayReferenceCount[λ][s'] ← 1
9     pathIndexToArrayIndex[λ][ℓ] ← s'
10 return arrayPointer_P[λ][s']
```

Define $\#^{(t)}_{\texttt{killPath}}$ similarly. Then, for every $1 \leq t \leq L$, we require that

$$1 \leq \left(1 + \#^{(t)}_{\texttt{clonePath}} - \#^{(t)}_{\texttt{killPath}}\right) \leq L \ . \tag{13}$$

Active: We say that path ℓ is active at the end of stage $1 \leq t \leq T$ if the following two conditions hold. First, there exists an index $1 \leq i \leq t$ for which f_i is either `clonePath` with corresponding output ℓ or `assignInitialPath` with output ℓ. Second, there is no intermediate index $i < j \leq t$ for which f_j is `killPath` with input ℓ. For each $1 \leq t < T$ we require that if f_{t+1} has input ℓ, then ℓ is active at the end of stage t.

We start by stating that the most basic thing one would expect to hold does indeed hold.

Lemma 4: Let $(f_t)_{t=0}^{T}$ be a valid sequence of calls to the low-level functions implemented in Algorithms 5–9. Then, the run is well defined: i) A "pop" operation is never carried out on a empty stack, ii) a "push" operation never results in a stack with more than L elements, and iii) a "read" operation from any array defined in lines 2–7 of Algorithm 5 is always preceded by a "write" operation to the same location in the array.

Proof: The proof boils-down to proving the following four statements concurrently for the end of each step $1 \leq t \leq T$, by induction on t.

- I A path index ℓ is active by Definition 3 iff activePath$[\ell]$ is true iff inactivePathIndices does not contain the index ℓ.
- II The bracketed expression in (13) is the number of active paths at the end of stage t.
- III The value of arrayReferenceCount$[\lambda][s]$ is positive iff the stack inactiveArrayIndices$[\lambda]$ does not contain the index s, and is zero otherwise.
- IV The value of arrayReferenceCount$[\lambda][s]$ is equal to the number of active paths ℓ for which pathIndexToArrayIndex$[\lambda][\ell] = s$.

■

We are now close to formalizing the utility of our low-level functions. But first, we must formalize the concept of a descendant path. Let $(f_t)_{t=0}^{T}$ be a valid sequence of calls. Next,

let ℓ be an active path index at the end of stage $1 \leq t < T$. Henceforth, let us abbreviate the phrase "path index ℓ at the end of stage t" by "$[\ell, t]$". We say that $[\ell', t+1]$ is a child of $[\ell, t]$ if i) ℓ' is active at the end of stage $t+1$, and ii) either $\ell' = \ell$ or f_{t+1} was the `clonePath` operation with input ℓ and output ℓ'. Likewise, we say that $[\ell', t']$ is a descendant of $[\ell, t]$ if $1 \leq t \leq t'$ and there is a (possibly empty) hereditary chain.

We now broaden our definition of a valid function calling sequence by allowing reads and writes to arrays.

Fresh pointer: consider the case where $t > 1$ and f_t is either the `getArrayPointer_P` or `getArrayPointer_C` function with input (λ, ℓ) and output p. Then, for valid indices i, we allow read and write operations to $p[i]$ after stage t but only before any stage $t' > t$ for which $f_{t'}$ is either `clonePath` or `killPath`.

Informally, the following lemma states that each path effectively sees a private set of arrays.

Lemma 5: Let $(f_t)_{t=0}^{T}$ be a valid sequence of calls to the low-level functions implemented in Algorithms 5–9. Assume the read/write operations between stages satisfy the "fresh pointer" condition.

Let the function f_t be `getArrayPointer_P` with input (λ, ℓ) and output p. Similarly, for stage $t' \geq t$, let $f_{t'}$ be `getArrayPointer_P` with input (λ, ℓ') and output p'. Assume that $[\ell', t']$ is a descendant of $[\ell, t]$.

Consider a "fresh pointer" write operation to $p[i]$. Similarly, consider a "fresh pointer" read operation from $p'[i]$ carried out after the "write" operation. Then, assuming no intermediate "write" operations of the above nature, the value written is the value read.

A similar claim holds for `getArrayPointer_C`.

Proof: With the observations made in the proof of Lemma 4 at hand, a simple induction on t is all that is needed. ∎

B. Mid-level functions

In this section we introduce Algorithms 10 and 11, our revised implementation of Algorithms 3 and 4, respectively, for the list decoding setting.

One first notes that our new implementations loop over all path indices ℓ. Thus, our new implementations make use of the functions `getArrayPointer_P` and `getArrayPointer_C` in order to assure that the consistency of calculations is preserved, despite multiple paths sharing information. In addition, Algorithm 10 contains code to normalize probabilities. The normalization is needed for a technical reason (to avoid floating-point underflow), and will be expanded on shortly.

We start out by noting that the "fresh pointer" condition we have imposed on ourselves indeed holds. To see this, consider first Algorithm 10. The key point to note is that neither the `killPath` nor the `clonePath` function is called from inside the algorithm. The same observation holds for Algorithm 11. Thus, the "fresh pointer" condition is met, and Lemma 5 holds.

We now consider the normalization step carried out in lines 20–25 of Algorithm 10. Recall that a floating-point

Algorithm 10: `recursivelyCalcP`(λ, φ) *list version*

Input: layer λ and phase φ

1 **if** $\lambda = 0$ **then return** `// Stopping condition`
2 set $\psi \leftarrow \lfloor \varphi/2 \rfloor$
`// Recurse first, if needed`
3 **if** $\varphi \mod 2 = 0$ **then** `recursivelyCalcP`$(\lambda - 1, \psi)$
`// Perform the calculation`
4 $\sigma \leftarrow 0$
5 **for** $\ell = 0, 1, \ldots, L-1$ **do**
6 **if** activePath$[\ell] =$ **false then continue**
7 $P_\lambda \leftarrow$ `getArrayPointer_P`(λ, ℓ)
8 $P_{\lambda-1} \leftarrow$ `getArrayPointer_P`$(\lambda - 1, \ell)$
9 $C_\lambda \leftarrow$ `getArrayPointer_C`(λ, ℓ)
10 **for** $\beta = 0, 1, \ldots, 2^{m-\lambda} - 1$ **do**
11 **if** $\varphi \mod 2 = 0$ **then**
`// apply Equation (4)`
12 **for** $u' \in \{0, 1\}$ **do**
13 $P_\lambda[\beta][u'] \leftarrow$ $\sum_{u''} \frac{1}{2} P_{\lambda-1}[2\beta][u' \oplus u''] \cdot P_{\lambda-1}[2\beta+1][u'']$
14 $\sigma \leftarrow \max(\sigma, P_\lambda[\beta][u'])$
15 **else** `// apply Equation (5)`
16 set $u' \leftarrow C_\lambda[\beta][0]$
17 **for** $u'' \in \{0, 1\}$ **do**
18 $P_\lambda[\beta][u''] \leftarrow$ $\frac{1}{2} P_{\lambda-1}[2\beta][u' \oplus u''] \cdot P_{\lambda-1}[2\beta+1][u'']$
19 $\sigma \leftarrow \max(\sigma, P_\lambda[\beta][u''])$
`// normalize probabilities`
`// In no-normalization variant, set` σ `to 1 here`
20 **for** $\ell = 0, 1, \ldots, L-1$ **do**
21 **if** activePath$[\ell] =$ **false then continue**
22 $P_\lambda \leftarrow$ `getArrayPointer_P`(λ, ℓ)
23 **for** $\beta = 0, 1, \ldots, 2^{m-\lambda} - 1$ **do**
24 **for** $u \in \{0, 1\}$ **do**
25 $P_\lambda[\beta][u] \leftarrow P_\lambda[\beta][u] / \sigma$

Algorithm 11: `recursivelyUpdateC`(λ, φ) *list version*

Input: layer λ and phase φ
Require : φ is odd

1 set $\psi \leftarrow \lfloor \varphi/2 \rfloor$
2 **for** $\ell = 0, 1, \ldots, L-1$ **do**
3 **if** activePath$[\ell] =$ **false then continue**
4 set $C_\lambda \leftarrow$ `getArrayPointer_C`(λ, ℓ)
5 set $C_{\lambda-1} \leftarrow$ `getArrayPointer_C`$(\lambda - 1, \ell)$
6 **for** $\beta = 0, 1, \ldots, 2^{m-\lambda} - 1$ **do**
7 $C_{\lambda-1}[2\beta][\psi \mod 2] \leftarrow C_\lambda[\beta][0] \oplus C_\lambda[\beta][1]$
8 $C_{\lambda-1}[2\beta+1][\psi \mod 2] \leftarrow C_\lambda[\beta][1]$
9 **if** $\psi \mod 2 = 1$ **then**
10 `recursivelyUpdateC`$(\lambda - 1, \psi)$

variable can not be used to hold arbitrarily small positive reals, and in a typical implementation, the result of a calculation that is "too small" will be rounded to 0. This scenario is called an "underflow".

We now confess that all our previous implementations of SC decoders were prone to "underflow". To see this, consider line 2 in the outline implementation given in Algorithm 1. Denote by $\mathbf{Y}$ and $\mathbf{U}$ the random vectors corresponding to $\mathbf{y}$ and $\mathbf{u}$, respectively. For $b \in \{0,1\}$ we have that

$$W_m^{(\varphi)}(\mathbf{y}_0^{n-1}, \hat{\mathbf{u}}_0^{\varphi-1} | b) = 2 \cdot \mathbb{P}(\mathbf{Y}_0^{n-1} = \mathbf{y}_0^{n-1}, \mathbf{U}_0^{\varphi-1} = \hat{\mathbf{u}}_0^{\varphi-1}, \mathsf{U}_\varphi = b) \leq 2 \cdot \mathbb{P}(\mathbf{U}_0^{\varphi-1} = \hat{\mathbf{u}}_0^{\varphi-1}, \mathsf{U}_\varphi = b) = 2^{-\varphi} \ .$$

Recall that φ iterates from 0 to $n-1$. Thus, for codes having length greater than some small constant, the comparison in line 2 of Algorithm 1 ultimately becomes meaningless when implemented using standard floating point arithmetic. The same holds for all of our previous implementations.

Fortunately, there is a solution to this problem. After the probabilities are calculated in lines 5–19 of Algorithm 10, we normalizethe highest probability to be 1 in lines 20–25.

We claim that apart for avoiding underflows, normalization does not alter our algorithm in that it does not change the chosen codeword. To see this, consider a variant of Algorithm 10, termed Algorithm 10', in which normalization is not carried out. That is, in Algorithm 10', just before line 20, we set the variable σ to 1. The following lemma states that for all $0 \leq \lambda \leq m$, both algorithm variants produce array entries $\mathsf{arrayPointer\_P}[\lambda][s]$ which differ up to a positive normalization constants β_λ. As can be seen in lines 17–25 of Algorithm 12 ahead, the returned codeword is the result of comparing probabilities in $\mathsf{arrayPointer\_P}[m]$. Thus, normalization indeed does not alter the returned codeword.

Lemma 6: Let two program executions be defined as follows. Execution A: Algorithm 10 is called with input parameters λ_0, φ_0, and a given state of the data structures. Execution B: Algorithm 10' is called with the same input parameters as Execution A, and the same state of the data structures, apart from the following. There exist positive reals $\alpha_0, \alpha_1, \ldots, \alpha_m$ such that for all $0 \leq \lambda \leq m$ and all $0 \leq s < \ell$, the value of $\mathsf{arrayPointer\_P}[\lambda][s]$ at the start of Execution B is α_λ times the value of $\mathsf{arrayPointer\_P}[\lambda][s]$ at the start of execution A.

Then, there exist positive reals $\beta_0, \beta_1, \ldots, \beta_m$ such that for all $0 \leq \lambda \leq m$ and all $0 \leq s < \ell$, the value of $\mathsf{arrayPointer\_P}[\lambda][s]$ at the end of Execution B is β_λ times the value of $\mathsf{arrayPointer\_P}[\lambda][s]$ at the end of execution A.

Proof: Recall the following about Algorithm 10 (and Algorithm 10'). When execution begins, a recursive call is made on line 3, if needed. Then, P_λ is transformed based on $P_{\lambda-1}$ and C_λ. Consider first the run of both algorithms during the innermost recursive call (the corresponding input parameter λ is the same for both algorithms, by inspection). By lines 13, 18, and 25, the ratio between P_λ entries in both runs is simply $(\alpha_{\lambda-1})^2$, divided by the value of σ after the main loop has finished executing. It is easily seen that $\sigma > 0$. Thus, after the innermost recursive call finishes in both algorithms, the assumption on the proportionality of $\mathsf{arrayPointer\_P}$ entries continues to hold. We now continue inductively: the claim is proved in much the same way for the pre-ultimate recursion, etc. ■

C. High-level functions

We now turn our attention to the high-level functions of our algorithm. Consider the topmost function, given in Algorithm 12. We start by noting that by lines 1 and 2, we have that condition "initialized" in Definition 3 is satisfied. Also, for the inductive basis, we have that condition "balanced" holds for $t = 1$ at the end of line 2. Next, notice that lines 3–5 are in-line with our "fresh pointer" condition. Next, consider lines 6–16, the main loop. These are the analog of the main loop in Algorithm 2, with the size-reduction replacements as per (11) and (12). After the main loop has finished, we pick (in lines 17–25) the most likely codeword from our list and return it.

Algorithm 12: SCL decoder, main loop

Input: the received vector $\mathbf{y}$ and a list size L as a global
Output: a decoded codeword $\hat{\mathbf{c}}$

```
   // Initialization
1  initializeDataStructures()
2  ℓ ← assignInitialPath()
3  P_0 ← getArrayPointer_P(0, ℓ)
4  for β = 0, 1, ..., n − 1 do
5      set P_0[β][0] ← W(y_β|0), P_0[β][1] ← W(y_β|1)
   // Main loop
6  for φ = 0, 1, ..., n − 1 do
7      recursivelyCalcP(m, φ)
8      if u_φ is frozen then
9          for ℓ = 0, 1, ..., L − 1 do
10             if activePath[ℓ] = false then continue
11             C_m ← getArrayPointer_C(m, ℓ)
12             set C_m[0][φ mod 2] to the frozen value of u_φ
13     else
14         continuePaths_UnfrozenBit(φ)
15     if φ mod 2 = 1 then
16         recursivelyUpdateC (m, φ)
   // Return the best codeword in the list
17 ℓ' ← 0, p' ← 0
18 for ℓ = 0, 1, ..., L − 1 do
19     if activePath[ℓ] = false then continue
20     C_m ← getArrayPointer_C(m, ℓ)
21     P_m ← getArrayPointer_P(m, ℓ)
22     if p' < P_m[0][C_m[0][1]] then
23         ℓ' ← ℓ, p' ← P_m[0][C_m[0][1]]
24 set C_0 ← getArrayPointer_C(0, ℓ')
25 return ĉ = (C_0[β][0])_{β=0}^{n−1}
```

Algorithm 13, `continuePaths_UnfrozenBit`, is the analog of lines 8–11 in Algorithm 2. However, now, instead of choosing the most likely fork out of 2 possible forks, we must typically choose the L most likely forks out of $2L$ possible forks. The most interesting line is 14, in which the best ρ forks are marked. Surprisingly[2], this can be done in $O(L)$ time [4, Section 9.3]. After the forks are marked, we first kill the paths

[2]The $O(L)$ time result is rather theoretical. Since L is typically a small number, the fastest way to achieve our selection goal would be through simple sorting.

Algorithm 13: `continuePaths_UnfrozenBit`(φ)

Input: phase φ

1 probForks $\leftarrow$ **new** 2-D float array of size $L \times 2$
2 $i \leftarrow 0$
`// populate` probForks
3 **for** $\ell = 0, 1, \ldots, L-1$ **do**
4 **if** activePath$[\ell] =$ **true then**
5 $P_m \leftarrow$ `getArrayPointer_P`(m, ℓ)
6 probForks $[\ell][0] \leftarrow P_m[0][0]$
7 probForks $[\ell][1] \leftarrow P_m[0][1]$
8 $i \leftarrow i + 1$
9 **else**
10 probForks $[\ell][0] \leftarrow -1$
11 probForks $[\ell][1] \leftarrow -1$
12 $\rho \leftarrow \min(2i, L)$
13 contForks $\leftarrow$ **new** 2-D boolean array of size $L \times 2$
`// The following is possible in` $O(L)$ `time`
14 populate contForks such that contForks$[\ell][b]$ is **true** iff probForks $[\ell][b]$ is one of the ρ largest entries in probForks (and ties are broken arbitrarily)
`// First, kill-off non-continuing paths`
15 **for** $\ell = 0, 1, \ldots, L-1$ **do**
16 **if** activePath$[\ell] =$ **false then continue**
17 **if** contForks$[\ell][0] =$ **false and** contForks$[\ell][1] =$ **false then**
18 `killPath`(ℓ)
`// Then, continue relevant paths, and duplicate if necessary`
19 **for** $\ell = 0, 1, \ldots, L-1$ **do**
20 **if** contForks$[\ell][0] =$ **false and** contForks$[\ell][1] =$ **false then** `// both forks are bad, or invalid`
21 **continue**
22 $C_m \leftarrow$ `getArrayPointer_C`(m, ℓ)
23 **if** contForks$[\ell][0] =$ **true and** contForks$[\ell][1] =$ **true then** `// both forks are good`
24 set $C_m[0][\varphi \mod 2] \leftarrow 0$
25 $\ell' \leftarrow$ `clonePath`(ℓ)
26 $C_m \leftarrow$ `getArrayPointer_C`(m, ℓ')
27 set $C_m[0][\varphi \mod 2] \leftarrow 1$
28 **else**`// exactly one fork is good`
29 **if** contForks$[\ell][0] =$ **true then**
30 set $C_m[0][\varphi \mod 2] \leftarrow 0$
31 **else**
32 set $C_m[0][\varphi \mod 2] \leftarrow 1$

for which both forks are discontinued, and then continue paths for which one or both of the forks are marked. In case of the latter, the path is first split. Note that we must first kill paths and only then split paths in order for the "balanced" constraint (13) to hold. Namely, this way, we will not have more than L active paths at a time.

The point of Algorithm 13 is to prune our list and leave only the L "best" paths. This is indeed achieved, in the following sense. At stage φ we would like to rank each path according to the probability

$$W_m^{(\varphi)}(\mathbf{y}_0^{n-1}, \hat{\mathbf{u}}_0^{\varphi-1} | \hat{\mathrm{u}}_\varphi) \ .$$

By (9) and (11), this would indeed by the case if our floating point variables were "perfect", and the normalization step in lines 20–25 of Algorithm 10 were not carried out. By Lemma 6, we see that this is still the case if normalization is carried out.

With respect to the above, consider the last part of Algorithm 12: rows 17–25, in which we claim to choose the most likely codeword. The claim is justified, by (9)–(12) and Lemma 6. Namely, the value of $P_m[0][C_m[0][1]]$ is simply

$$W_m^{(n-1)}(\mathbf{y}_0^{n-1}, \hat{\mathbf{u}}_0^{n-2} | \hat{u}_{n-1}) = \frac{1}{2^{n-1}} \cdot P(\mathbf{y}_0^{n-1} | \hat{\mathbf{u}}_0^{n-1}) \ ,$$

up to a normalization constant.

We now prove our two main result.

Theorem 7: The space complexity of the SCL decoder is $O(L \cdot n)$.

Proof: All the data-structures of our list decoder are allocated in Algorithm 5, and it can be checked that the total space used by them is $O(L \cdot n)$. Apart from these, the space complexity needed in order to perform the selection operation in line 14 of Algorithm 13 is $O(L)$. Lastly, the various local variables needed by the algorithm take $O(1)$ space, and the stack needed in order to implement the recursion takes $O(\log n)$ space. ∎

Theorem 8: The running time of the SCL decoder is $O(L \cdot n \log n)$.

Proof: Recall that by our notation $m = \log n$. The following bottom-to-top table summarizes the running time of each function. The notation O_Σ will be explained shortly.

function	running time
`initializeDataStructures`$()$	$O(L \cdot m)$
`assignInitialPath`$()$	$O(m)$
`clonePath`(ℓ)	$O(m)$
`killPath`(ℓ)	$O(m)$
`getArrayPointer_P`(λ, ℓ)	$O(2^{m-\lambda})$
`getArrayPointer_C`(λ, ℓ)	$O(2^{m-\lambda})$
`recursivelyCalcP`$(m, \cdot)$	$O_\Sigma(L \cdot m \cdot n)$
`recursivelyUpdateC`$(m, \cdot)$	$O_\Sigma(L \cdot m \cdot n)$
`continuePaths_UnfrozenBit`(φ)	$O(L \cdot m)$
SCL decoder	$O(L \cdot m \cdot n)$

The first 7 functions in the table, the low-level functions, are easily checked to have the stated running time. Note that the running time of `getArrayPointer_P` and `getArrayPointer_C` is due to the copy operation in line 6 of Algorithm 6 applied to an array of size $O(2^{m-\lambda})$. Thus, as was previously mentioned, reducing the size of our arrays has helped us reduce the running time of our list decoding algorithm.

Next, let us consider the two mid-level functions, namely, `recursivelyCalcP` and `recursivelyUpdateC`. The notation

$$\texttt{recursivelyCalcP}(m, \cdot) \in O_\Sigma(L \cdot m \cdot n)$$

means that total running time of the n function calls

$$\texttt{recursivelyCalcP}(m, \varphi) \ , \quad 0 \le \varphi < 2^m$$

is $O(L \cdot m \cdot n)$. To see this, denote by $f(\lambda)$ the total running time of the above with m replaced by λ. By splitting the running time of Algorithm 10 into a non-recursive part and a recursive part, we have that for $\lambda > 0$

$$f(\lambda) = 2^\lambda \cdot O(L \cdot 2^{m-\lambda}) + f(\lambda - 1) \ .$$

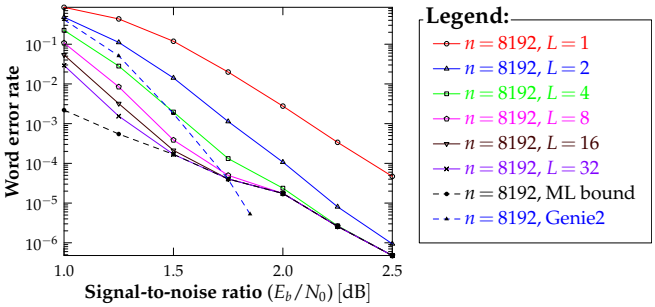


Fig. 6: Word error rate of a length $n = 8192$ rate $1/2$ polar code optimized for SNR=2 dB under various list sizes. Code construction was carried out via the method proposed in [15]. For reference, the Genie2 plot is the error rate if standard successive cancellation is used, and a genie corrects at most 2 wrong bit decisions.

Thus, it easily follows that

$$f(m) \in O(L \cdot m \cdot 2^m) = O(L \cdot m \cdot n) \, .$$

In essentially the same way, we can prove that the total running time of the `recursivelyUpdateC`(m, φ) over all 2^{n-1} valid (odd) values of φ is $O(m \cdot n)$. Note that the two mid-level functions are invoked in lines 7 and 16 of Algorithm 12, on all valid inputs.

The running time of the high-level functions is easily checked to agree with the table.

■

V. Modified polar codes

The plots in Figures 1 and 6 were obtained by simulation. The performance of our decoder for various list sizes is given by the solid lines in the figures. As expected, we see that as the list size L increases, the performance of our decoder improves. We also notice a diminishing-returns phenomenon in terms of increasing the list size. Namely, the difference in error correction performance is more dramatic when small list sizes are increased. The reason for this turns out to be simple, as we now show.

The dashed line, termed the "ML bound" was obtained as follows. During our simulations for $L = 32$, each time a decoding failure occurred, we checked whether the decoded codeword was more likely than the transmitted codeword. That is, whether $W(\mathbf{y}|\hat{\mathbf{c}}) > W(\mathbf{y}|\mathbf{c})$. If so, then the optimal ML decoder would surely misdecode $\mathbf{y}$ as well. The dashed line records the frequency of the above event, and is thus a lower-bound on the error probability of the ML decoder. Thus, for an SNR value greater than about 1.5 dB, Figure 1 suggests that we have an essentially optimal decoder when $L = 32$.

Can we do even better? At first, the answer seems to be an obvious "no", at least for the region in which our decoder is essentially optimal. However, it turns out that if we are willing to accept a small change in our definition of a polar code, we can dramatically improve performance.

During simulations we noticed that often, when a decoding error occurred, the path corresponding to the transmitted codeword was a member of the final list. However, since there was a more likely path in the list, the codeword corresponding to that path was returned, which resulted in a decoding error. Thus, if only we had a "genie" to tell us at the final stage which path to pick from our list, we could improve the performance of our decoder.

Fortunately, such a genie is easy to implement. Recall that we have k unfrozen bits that we are free to set. Instead of setting all of them to information bits we wish to transmit, we employ the following simple concatenation scheme. For some small constant r, we set the first $k - r$ unfrozen bits to information bits. The last r unfrozen bits will hold the r-bit CRC [9, Section 8.8] value[3] of the first $k - r$ unfrozen bits. Note this new encoding is a slight variation of our polar coding scheme. Also, note that we incur a penalty in rate, since the rate of our code is now $(k-r)/n$ instead of the previous k/n.

What we have gained is an approximation to a genie: at the final stage of decoding, instead of searching for the most likely codeword in lines 17–25 of Algorithm 12, we can do the following. A path for which the CRC is invalid can not correspond to the transmitted codeword. Thus, we refine our selection as follows. If at least one path has a correct CRC, then we remove from our list all paths having incorrect CRC and then choose the most likely path. Otherwise, we select the most likely path in the hope of reducing the number of bits in error, but with the knowledge that we have at least one bit in error.

Figure 2 contains a comparison of decoding performance between the original polar codes and the slightly tweaked version presented in this section. A further improvement in bit-error-rate (but not in block-error-rate) is attained when the decoding is performed systematically [2]. The application of systematic polar coding to a list decoding setting is attributed to [13]. We also note Figure 7, in which the performance of the CRC variant of our decoder is plotted, for various list sizes.

References

[1] E. Arıkan, "Channel polarization: A method for constructing capacity-achieving codes for symmetric binary-input memoryless channels," *IEEE Trans. Inform. Theory*, vol. 55, pp. 3051–3073, 2009.

[2] E. Arıkan, "Systematic polar coding," *IEEE Comm. Lett.*, vol. 15, pp. 860–862, 2011.

[3] E. Arıkan and E. Telatar, "On the rate of channel polarization," in *Proc. IEEE Int'l Symp. Inform. Theory*, pp. 1493–1495, Seoul, South Korea, July 2009.

[4] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. Cambridge, Massachusetts: The MIT Press, 2001.

[5] I. Dumer and K. Shabunov, "Soft-decision decoding of Reed-Muller codes: recursive lists," *IEEE Trans. Inform. Theory*, vol. 52, pp. 1260–1266, 2006.

[6] S.B. Korada, E. Şaşoğlu, and R. Urbanke, "Polar codes: Characterization of exponent, bounds, and constructions," *IEEE Trans. Inform. Theory*, vol. 56, pp. 6253–6264, 2010.

[7] C. Leroux, A.J. Raymond, G. Sarkis, I. Tal, A.Vardy, and W.J. Gross, "Hardware implementation of successive-cancellation decoders for polar codes," *Journal of Signal Processing Systems*, vol. 69, no. 3, pp. 305–315, December 2012.

[8] M. Mondelli, S. H. Hassani, and R. Urbanke, "Scaling exponent of list decoders with applications to polar codes," `arXiv:1304.5220v2`, 2013.

[9] W.W. Peterson and E.J. Weldon, *Error-Correcting Codes*, 2nd ed. Cambridge, Massachusetts: The MIT Press, 1972.

[3] A binary linear code having a corresponding $k \times r$ parity-check matrix constructed as follows will do just as well. Let the first $k - r$ columns be chosen at random and the last r columns be equal to the identity matrix. The complexity of computing the parity is simply $O((k - r) \cdot r)$.

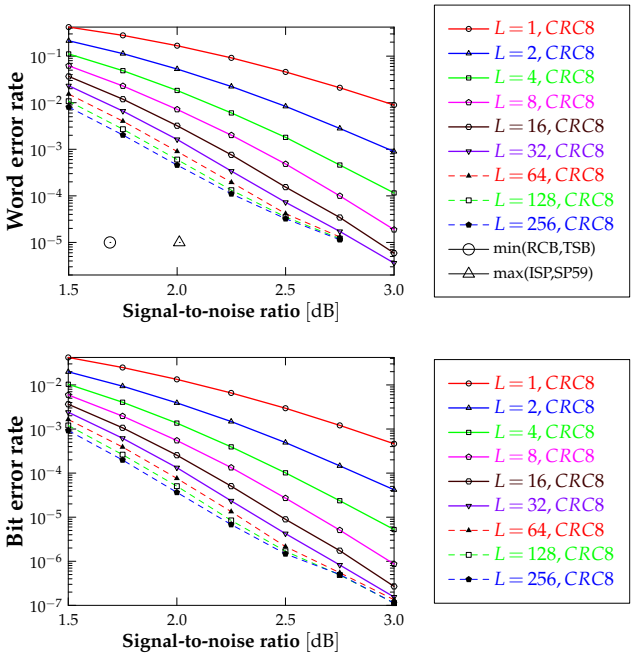


Fig. 7: Word error rate (top) and bit error rate (bottom) of a length $n = 512$ rate $1/2$ polar code for various list sizes. A CRC of size 8 was used as described in Section V. In the top plot, the two dots [17, Figure 9] represent upper and lower bounds on the SNR needed to reach a word error rate of 10^{-5}.

[10] Y. Polyanskiy, private communication, 2012.

[11] Y. Polyanskiy, H.V. Poor, and S. Verdú, "Channel coding rate in the finite blocklength regime," *IEEE Trans. Inform. Theory*, vol. 56, pp. 2307–2359, 2010.

[12] G. Sarkis, P. Giard, A.Vardy, C. Thibeault, and W.J. Gross, "Fast polar decoders: algorithm and implementation," *IEEE J. Select. Areas Comm.*, vol. 32, no. 5, pp. 946–957, May 2014.

[13] G. Sarkis and W.J. Gross, "Systematic encoding of polar codes for list decoding," 2011, private communication.

[14] E. Şaşoğlu, "Polarization and polar codes," *Foundations and Trends in Communications and Information Theory*, vol. 8, no. 4, pp. 259–381, October 2012.

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

[16] TurboBest, "IEEE 802.16e LDPC Encoder/Decoder Core." [Online]. Available: http://www.turbobest.com/tb_ldpc80216e.htm

[17] G. Wiechman and I. Sason, "An improved sphere-packing bound for finite-length codes over symmetric memoryless channels," *IEEE Trans. Inform. Theory*, vol. 54, pp. 1962–1990, 2008.

Ido Tal was born in Haifa, Israel, in 1975. He received the B.Sc., M. Sc., and Ph.D. degrees in computer science from Technion—Israel Institute of Technology, Haifa, Israel, in 1998, 2003 and 2009, respectively. During 2010– 2012 he was a postdoctoral scholar at the University of California at San Diego. In 2012 he joined the Electrical Engineering Department at Technion. His research interests include constrained coding and error-control coding.

Alexander Vardy (S88–M91–SM94–F98) was born in Moscow, U.S.S.R., in 1963. He earned his B.Sc. (summa cum laude) from the Technion, Israel, in 1985, and Ph.D. from the Tel-Aviv University, Israel, in 1991. During 1985–1990 he was with the Israeli Air Force, where he worked on electronic counter measures systems and algorithms. During the years 1992 and 1993 he was a Visiting Scientist at the IBM Almaden Research Center, in San Jose, CA. From 1993 to 1998, he was with the University of Illinois at Urbana-Champaign, first as an Assistant Professor then as an Associate Professor. Since 1998, he has been with the University of California San Diego (UCSD), where he is the Jack Keil Wolf Endowed Chair Professor in the Department of Electrical and Computer Engineering, with joint appointments in the Department of Computer Science and the Department of Mathematics. While on sabbatical from UCSD, he has held long-term visiting appointments with CNRS, France, the EPFL, Switzerland, and the Technion, Israel.

His research interests include error-correcting codes, algebraic and iterative decoding algorithms, lattices and sphere packings, coding for digital media, cryptography and computational complexity theory, and fun math problems.

He received an IBM Invention Achievement Award in 1993, and NSF Research Initiation and CAREER awards in 1994 and 1995. In 1996, he was appointed Fellow in the Center for Advanced Study at the University of Illinois, and received the Xerox Award for faculty research. In the same year, he became a Fellow of the Packard Foundation. He received the IEEE Information Theory Society Paper Award (jointly with Ralf Koetter) for the year 2004. In 2005, he received the Fulbright Senior Scholar Fellowship, and the Best Paper Award at the IEEE Symposium on Foundations of Computer Science (FOCS). During 1995–1998, he was an Associate Editor for Coding Theory and during 1998–2001, he was the Editor-in-Chief of the IEEE TRANSACTIONS ON INFORMATION THEORY. From 2003 to 2009, he was an Editor for the *SIAM Journal on Discrete Mathematics*. He is currently serving on the Executive Editorial Board for the IEEE TRANSACTIONS ON INFORMATION THEORY. He has been a member of the Board of Governors of the IEEE Information Theory Society during 1998–2006, and again starting in 2011.