List Decoding of Polar Codes

Ido Tal Alexander Vardy
University of California San Diego,
La Jolla, CA 92093, USA
Email: idotal@ieee.org, 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, up to L decoding paths are considered concurrently at each decoding stage. Simulation results show that the resulting performance is very close to that of a maximum-likelihood decoder, even for moderate values of L. Thus it appears that the proposed list decoder bridges the gap between successive-cancellation and maximum-likelihood decoding of polar codes.**

The specific list-decoding algorithm that achieves this performance doubles the number of decoding paths at each decoding step, and then uses a pruning procedure to discard all but the L "best" paths. In order to implement this algorithm, we introduce a natural pruning criterion that can be easily evaluated. Nevertheless, straightforward implementation still requires $O(L \cdot n^2)$ time, which is in stark contrast with the $O(n \log n)$ complexity of the original successive-cancellation decoder. We utilize the structure of polar codes to overcome this problem. Specifically, we devise an efficient, numerically stable, implementation taking only $O(L \cdot n \log n)$ time and $O(L \cdot n)$ space.

I. Introduction

Polar codes, recently discovered by Arıkan [1], are a major breakthrough in coding theory. They are the first and currently only family of codes known to have an explicit construction and efficient encoding and decoding algorithms, while also being capacity achieving over binary input symmetric memoryless channels. Their probability of error is known to approach $O(2^{-\sqrt{n}})$ [2], with generalizations giving even better asymptotic results [3].

Of course, "capacity achieving" is an asymptotic property, and the main sticking point of polar codes to date is that their performance at short to moderate block lengths is disappointing. As we ponder why, we identify two possible culprits: either the codes themselves are inherently weak at these lengths, or the successive cancellation (SC) decoder employed to decode them is significantly degraded with respect to Maximum Likelihood (ML) decoding performance. More so, the two possible culprits are complementary, and so both may occur.

In this paper we show an improvement to the SC decoder, namely, a successive cancellation list (SCL) decoder. Our list decoder has a corresponding *list size* L, and setting $L = 1$ results in the classic SC decoder. As can be seen in Figure 1, our algorithm improves upon the classic SC decoder. Indeed, Figure 1 shows a wide range in which our algorithm has performance very close to that of the ML decoder[1], even for moderate values of L. Thus as we have suspected, the sub-optimality of the SC decoder plays a role in the disappointing performance of polar codes.

[1]The lower bound on ML was got by performing decoding with $L = 32$, and counting the number of times the decoded codeword was more likely than the transmitted one.

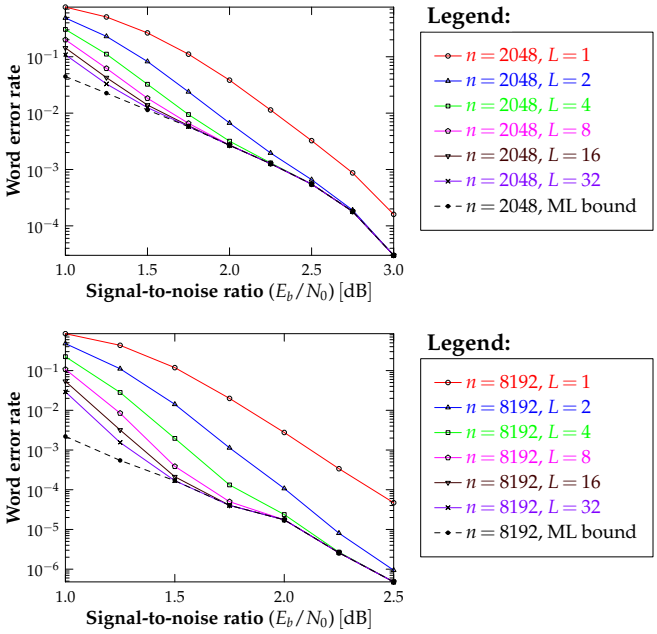

Fig. 1. Word error rate of a length $n = 2048$ (top) and $n = 8192$ (bottom) rate $1/2$ polar code optimized for SNR=2 dB under various list sizes. Code construction was carried out via the method proposed in [4].

The structure of this paper is as follows. In Section II, we present Arıkan's SC decoder in a notation that will be useful to us later on. In Section III, we show how the space complexity of the SC decoder can be brought down from $O(n \log n)$ to $O(n)$. This observation will later help us in Section IV, where we presents our successive cancellation list decoder with time complexity $O(L \cdot n \log n)$.

II. Formalization of the Successive Cancellation Decoder

The Successive Cancellation decoder is due to Arıkan [1]. In this section, we recast it using our notation, for future reference. However, let us start by stating some conventions and defining the underlying bit channels [1].

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 $\mathbf{u} = (u_i)_{i=0}^{n-1}$ the information bits vector (including the frozen bits), and by $\mathbf{c} = (c_i)_{i=0}^{n-1}$ 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 $\mathbf{y} = (y_i)_{i=0}^{n-1}$. A decoding algorithm is

978-1-4577-0594-6/11/$26.00 ©2011 IEEE

then applied to $\mathbf{y}$, resulting in a decoded codeword $\hat{\mathbf{c}}$ having corresponding information bits $\hat{\mathbf{u}}$.

For *layer* $0 \leq \lambda \leq m$, denote hereafter $\Lambda = 2^\lambda$. Recall that for $0 \leq \varphi < 2^\lambda$, *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 $W_\lambda^{(\varphi)}(\mathbf{z}_0^{\Lambda-1}, \mathbf{u}_0^{\varphi-1}|u_\varphi)$. In our context, $\mathbf{z}_0^{\Lambda-1}$ is always a contiguous subvector of $\mathbf{y}$.

For $1 \leq \lambda \leq m$, recall the recursive definition of a bit channel [1, Equations (22) and (23)] : let $0 \leq 2\psi < \Lambda$, then

$$W_\lambda^{(2\psi)}(\mathbf{z}_0^{\Lambda-1}, \mathbf{u}_0^{2\psi-1}|u_{2\psi}) = \sum_{u_{2\psi+1}} \frac{1}{2} \underbrace{W_{\lambda-1}^{(\psi)}(\mathbf{z}_0^{\Lambda/2-1}, \mathbf{u}_{0,\text{even}}^{2\psi-1} \oplus \mathbf{u}_{0,\text{odd}}^{2\psi-1}|u_{2\psi} \oplus u_{2\psi+1})}_{\text{even branch}} \cdot \underbrace{W_{\lambda-1}^{(\psi)}(\mathbf{z}_{\Lambda/2}^{\Lambda-1}, \mathbf{u}_{0,\text{odd}}^{2\psi-1}|u_{2\psi+1})}_{\text{odd branch}} \quad (1)$$

and

$$W_\lambda^{(2\psi+1)}(\mathbf{z}_0^{\Lambda-1}, \mathbf{u}_0^{2\psi}|u_{2\psi+1}) = \frac{1}{2} \underbrace{W_{\lambda-1}^{(\psi)}(\mathbf{z}_0^{\Lambda/2-1}, \mathbf{u}_{0,\text{even}}^{2\psi-1} \oplus \mathbf{u}_{0,\text{odd}}^{2\psi-1}|u_{2\psi} \oplus u_{2\psi+1})}_{\text{even branch}} \cdot \underbrace{W_{\lambda-1}^{(\psi)}(\mathbf{z}_{\Lambda/2}^{\Lambda-1}, \mathbf{u}_{0,\text{odd}}^{2\psi-1}|u_{2\psi+1})}_{\text{odd branch}} \quad (2)$$

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

For $0 \leq \lambda \leq m$ define the following quotient/remainder shorthand. Let $0 \leq \varphi < 2^\lambda$ and $0 \leq \beta < 2^{m-\lambda}$, then

$$\langle \varphi, \beta \rangle_\lambda = \varphi + 2^\lambda \cdot \beta \, .$$

Note that each integer $0 \leq i < 2^m$ has a unique representation as $i = \langle \varphi, \beta \rangle_\lambda$. For reasons which will become clear, we call φ and β the *phase* and *branch* parts of i, respectively. Thus, we will say that layer λ has 2^λ phases, each phase consisting of $2^{m-\lambda}$ branches.

Our first implementation of the SC decoder (Algorithms 1–3) will be straightforward, but somewhat wasteful in terms of space. It will make use of two sets of arrays. The first set of arrays is defined as follows. For each $0 \leq \lambda \leq m$, we will have a *probabilities array*, denoted by P_λ, of length 2^m, indexed by an integer $0 \leq i < 2^m$. Each element $P_\lambda[i]$ will contain a probability pair, indexed as $P_\lambda[i][0]$ and $P_\lambda[i][1]$. The second set consists of *bit arrays*, denoted by B_λ. They have the same length and indexing of probability arrays. However, each element $B_\lambda[i]$ of a bit array consists of a single bit. Initially, most array elements will be uninitialized, and will become initialized as the algorithm runs its course. Note that the total space needed for these arrays is $O(n \log n)$. In the interest of brevity, for a generic array A we abbreviate $A_\lambda[\langle \varphi, \beta \rangle_\lambda]$ as $A_\lambda[\langle \varphi, \beta \rangle]$. Also, we use the shorthand $A_\lambda[\langle \varphi, * \rangle]$ to symbolize all the elements $A_\lambda[\langle \varphi, \beta \rangle]$ for $0 \leq \beta < 2^{m-\lambda}$.

Due to space limitations, we will not give a full proof of the correctness of our implementation, but rather a short explanation. For $\lambda > 0$ and $0 \leq \varphi < 2^\lambda$, recall the recursive definition of $W_\lambda^{(\varphi)}(\mathbf{z}_0^{\Lambda-1}, \mathbf{u}_0^{\varphi-1}|u_\varphi)$ given in either (1) or (2), depending on the parity of φ. In both cases, the channel $W_{\lambda-1}^{(\psi)}$, $\psi = \lfloor \varphi/2 \rfloor$ is used with two different outputs. Thus, we need a simple way of defining which set of outputs 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 $0 \leq \beta < 2^{m-\lambda}$. Since the channel $W_m^{(\varphi)}$ has only one vector pair of outputs associated with it, $(\mathbf{y}_0^n, \hat{\mathbf{u}}_0^{\varphi-1})$, we give a branch number of $\beta = 0$ to each such pair. Next, we proceed recursively as follows. Consider a channel $W_\lambda^{(\varphi)}$ with outputs $(\mathbf{z}_0^{\Lambda-1}, \mathbf{u}_0^{\varphi-1})$ and corresponding branch number β. The output $(\mathbf{z}_0^{\Lambda/2-1}, \mathbf{u}_{0,\text{even}}^{2\psi-1} \oplus \mathbf{u}_{0,\text{odd}}^{2\psi-1})$ associated with $W_{\lambda-1}^{(\psi)}$ will have a branch number of 2β, while the output $(\mathbf{z}_{\Lambda/2}^{\Lambda-1}, \mathbf{u}_{0,\text{odd}}^{2\psi-1})$ will

Algorithm 1: First implementation of SC decoder

```
for β = 0, 1, ..., n − 1 do // Initialization
    P_0[⟨0, β⟩][0] ← W(y_β|0), P_0[⟨0, β⟩][1] ← W(y_β|1)
for φ = 0, 1, ..., n − 1 do // Main loop
    recursivelyCalcP(m, φ)
    if û_φ is frozen then
        set B_m[⟨φ, 0⟩] to the frozen value
    else
        if P_m[⟨φ, 0⟩][0] > P_m[⟨φ, 0⟩][1] then
            set B_m[⟨φ, 0⟩] ← 0
        else
            set B_m[⟨φ, 0⟩] ← 1
    if φ mod 2 = 1 then
        recursivelyUpdateB(m, φ)
return the decoded codeword: ĉ = (B_0[⟨0, β⟩])_{β=0}^{n−1}
```

Algorithm 2: recursivelyUpdateB(λ, φ)

```
Require: φ is odd
set ψ ← ⌊φ/2⌋
for β = 0, 1, ..., 2^{m−λ} − 1 do
    B_{λ−1}[⟨ψ, 2β⟩] ← B_λ[⟨φ − 1, β⟩] ⊕ B_λ[⟨φ, β⟩]
    B_{λ−1}[⟨ψ, 2β + 1⟩] ← B_λ[⟨φ, β⟩]
if ψ mod 2 = 1 then
    recursivelyUpdateB(λ − 1, ψ)
```

Algorithm 3: recursivelyCalcP(λ, φ)

```
if λ = 0 then return // Stopping condition
set ψ ← ⌊φ/2⌋
// Recurse first, if needed
if φ mod 2 = 0 then recursivelyCalcP(λ − 1, ψ)
for β = 0, 1, ..., 2^{m−λ} − 1 do // calculation
    if φ mod 2 = 0 then // apply Equation (1)
        for u′ ∈ {0, 1} do
            P_λ[⟨φ, β⟩][u′] ← Σ_{u″} (1/2) P_{λ−1}[⟨ψ, 2β⟩][u′ ⊕ u″] ·
            P_{λ−1}[⟨ψ, 2β + 1⟩][u″]
    else // apply Equation (2)
        set u′ ← B_λ[⟨φ − 1, β⟩]
        for u″ ∈ {0, 1} do
            P_λ[⟨φ, β⟩][u″] ← (1/2) P_{λ−1}[⟨ψ, 2β⟩][u′ ⊕ u″] ·
            P_{λ−1}[⟨ψ, 2β + 1⟩][u″]
```

have a branch number of $2\beta + 1$ (recall the even branch/odd branch naming). Similarly to tagging the output of a channel by the branch number β, we do the same for the input to it.

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 [5].

As a first step towards this end, consider the probability pair array P_m. By examining the main loop in Algorithm 1, 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 $P_m[\langle \varphi'', 0 \rangle]$ are 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. Specifically, for all $0 \leq \lambda \leq m$, let us now define the number of elements in P_λ to be $2^{m-\lambda}$. Accordingly, we must also replace all references of the generic form $P_\lambda[\langle \varphi, \beta \rangle]$ by $P_\lambda[\beta]$.

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 2. 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. So, let us make one more substitution: replace $C_\lambda[\psi + \beta \cdot 2^{\lambda-1}][\varphi \mod 2]$ by $C_\lambda[\beta][\varphi \mod 2]$, and resize each array C_λ to have $2^{m-\lambda}$ bit pairs. The alert reader will notice that a further reduction in space is possible, since for $\lambda = 0$ we will always have that $\varphi = 0$ and thus its parity is always even. However, this reduction does not affect the asymptotic space complexity which is now indeed down to $O(n)$.

The main loop of the new algorithm is given as Algorithm 4. The helper functions are given as Algorithms 5 and 6, with the added condition of ignoring the ℓ parameter (defined and used in the next section) and any lines making reference to it.

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.

Algorithm 4: Space efficient SC decoder, main loop

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

```
// Initialization
for β = 0, 1, ..., n − 1 do
    set P_0[β][0] ← W(y_β|0), P_0[β][1] ← W(y_β|1)
// Main loop
for φ = 0, 1, ..., n − 1 do
    recursivelyCalcP(m, φ)
    if û_φ is frozen then
        set C_m[0][φ mod 2] to the frozen value
    else
        if P_m[0][0] > P_m[0][1] then
            set C_m[0][φ mod 2] ← 0
        else
            set C_m[0][φ mod 2] ← 1
    if φ mod 2 = 1 then
        recursivelyUpdateC(m, φ)
return the decoded codeword: ĉ = (C_0[β][0])_{β=0}^{n−1}
```

Algorithm 5: recursivelyCalcP(λ, φ, ℓ)

```
// stopping condition
if λ = 0 then return
set ψ ← ⌊φ/2⌋
// Recurse first, if needed
if φ mod 2 = 0 then recursivelyCalcP(λ − 1, ψ)
// Perform the calculation
P_λ ← getArrayPointer_P(λ, ℓ)
P_{λ−1} ← getArrayPointer_P(λ − 1, ℓ)
for β = 0, 1, ..., 2^{m−λ} − 1 do
    if φ mod 2 = 0 then // apply Equation (1)
        for u' ∈ {0, 1} do
            P_λ[β][u'] ←
            Σ_{u''} (1/2) P_{λ−1}[2β][u' ⊕ u''] · P_{λ−1}[2β + 1][u'']
    else                // apply Equation (2)
        C_λ ← getArrayPointer_C(λ, ℓ)
        set u' ← C_λ[β][0]
        for u'' ∈ {0, 1} do
            P_λ[β][u''] ←
            (1/2) P_{λ−1}[2β][u' ⊕ u''] · P_{λ−1}[2β + 1][u'']
```

Algorithm 6: recursivelyUpdateC(λ, φ, ℓ)

Require: φ is odd

```
set C_λ ← getArrayPointer_C(λ, ℓ)
set C_{λ−1} ← getArrayPointer_C(λ − 1, ℓ)
set ψ ← ⌊φ/2⌋
for β = 0, 1, ..., 2^{m−λ} − 1 do
    C_{λ−1}[2β][ψ mod 2] ← C_λ[β][0] ⊕ C_λ[β][1]
    C_{λ−1}[2β + 1][ψ mod 2] ← C_λ[β][1]
if ψ mod 2 = 1 then
    recursivelyUpdateC(λ − 1, ψ)
```

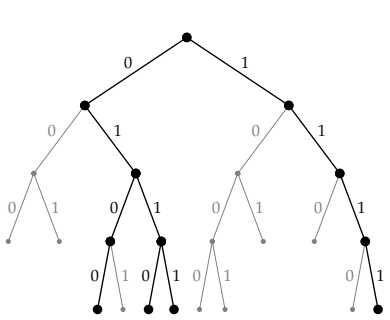

Fig. 2. Decoding paths of unfrozen bits for $L = 4$.

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. We note at this point that successive cancellation list decoding is not a new idea: it was applied in [6] to Reed-Muller codes[2].

Recall the main loop of an SC decoder, in which 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. Namely, at each phase, when decoding a non-frozen bit, we split each decoding path into two paths (see Figure 2). Of course, since the number of paths grows exponentially, 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.

Consider the following outline for a naive implementation of an SCL decoder. Each time a decoding path is split in two, the data structures used by the "parent" path are duplicated, with one copy given to the first split and the other to the second. Since the number of splits occurring is $O(L \cdot n)$, and since the size of the data structures used by each path is at least $O(n)$, the copying operation alone would take time at least $O(L \cdot n^2)$. 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 $O(L \cdot n^2)$. 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 $O(L \cdot n^2)$.

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. On the other hand, when looking at the main loop of Algorithm 4 and unwinding the recursion, we see that P_λ is accessed only every $2^{m-\lambda}$ incrementations of φ in Algorithm 4. 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 "lazy copy". Namely, 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.

[2]In a somewhat different version of successive cancellation than that of Arıkan's, at least in exposition.

The previous high level description translates into quite a bit of book-keeping, through the use of auxiliary arrays and queues. Unfortunately, space limitations prevent us from furnishing all of the relevant pseudo code in this paper. Algorithms 5–10 are all that we have managed to fit in, and we hope the names of the missing functions give a good hint as to what they do. Likewise, we do not elaborate on how we have chosen the pruning function. Lastly, if one were to try to implement our pseudo code, they would quickly discover numerical problems; specifically underflow. This can be overcome through careful normalization of the probabilities. We close by promising that a full version of this paper, with all the missing details and explanations, will be posted on arXiv.

Algorithm 7: assignInitialPath

```
inactivePathIndices ← new queue
inactiveArrayIndices ← new array of size m, the elements of which are queues
activePath ← new boolean array of size L
arrayReferenceCount ← new 2-D array of size m × L
pathIndexToArrayIndex ← new 2-D array of size m × L
arrayPointer_P ← new 2-D array of size m × L, the elements of which are array pointers
arrayPointer_C ← new 2-D array of size m × L, the elements of which are array pointers
// Initialization of data structures
for λ = 0, 1, ..., m do
    for s = 0, 1, ..., L − 1 do
        arrayPointer_P[λ][s] ← new array of float pairs of size 2^(m−λ)
        arrayPointer_C[λ][s] ← new array of bit pairs of size 2^(m−λ)
        arrayReferenceCount[λ][s] ← 0
        push(inactiveArrayIndices[λ], s)
for ℓ = 0, 1, ..., L − 1 do
    activePath[ℓ] ← false
    push(inactivePathIndices, ℓ)
// Get the new path index, and mark its arrays
ℓ ← pop(inactivePathIndices)
activePath[ℓ] ← true
for λ = 0, 1, ..., m do
    s ← pop(inactiveArrayIndices[λ])
    pathIndexToArrayIndex[λ][ℓ] ← s
    arrayReferenceCount[λ][s] ← 1
return ℓ
```

Algorithm 8: continuePaths_FrozenBit(φ)

```
for ℓ = 0, 1, ..., L − 1 do
    if pathIndexInactive(ℓ) then continue
    C_m ← getArrayPointer_C(m, ℓ)
    set C_m[0][φ mod 2] to the frozen value of index φ
```

Algorithm 9: continuePaths_UnfrozenBit(φ)

forksArray $\leftarrow$ **new** (float,bit,index)-triplets array of size $2L$
$i \leftarrow 0$
// populate forksArray
for $\ell = 0, 1, \ldots, L-1$ **do**
  if pathIndexInactive(ℓ) **then continue**
  $P_m \leftarrow$ getArrayPointer_P(m, ℓ)
  forksArray $[2i] \leftarrow (P_m[0][0], 0, \ell)$
  forksArray $[2i+1] \leftarrow (P_m[0][1], 1, \ell)$
  $i \leftarrow i + 1$
// pivot forksArray, possible in $O(L)$ time
$\rho \leftarrow \min(2i, L)$
rearrange the entries of forksArray so that for all $\alpha < \rho$ and $\beta \geq \rho$ we have that forksArray$[\alpha][0] \geq$ forksArray$[\beta][0]$
// Pick the best ρ forks
contForks $\leftarrow$ **new** (boolean,boolean)-pairs array of size L
initialize all elements of contForks to (**false**,**false**)
for $r = 0, 1, \ldots, \rho - 1$ **do**
  $\ell \leftarrow$ forksArray$[r][2]$
  $b \leftarrow$ forksArray$[r][1]$
  contForks$[\ell][b] \leftarrow$ **true**
// Kill-off non-continuing paths
for $\ell = 0, 1, \ldots, L-1$ **do**
  if pathIndexInactive(ℓ) **then**
    continue
  if contForks$[\ell][0] =$ **false and** contForks$[\ell][1] =$ **false then**
    killPath(ℓ)
// Continue relevant paths, and duplicate if necessary
for $\ell = 0, 1, \ldots, L-1$ **do**
  if contForks$[\ell][0] =$ **false and** contForks$[\ell][1] =$ **false then** // both forks are bad, or invalid
    continue
  $C_m \leftarrow$ getArrayPointer_C(m, ℓ)
  if contForks$[\ell][0] =$ **true and** contForks$[\ell][1] =$ **true then** // both forks are good
    set $C_m[0][\varphi \mod 2] \leftarrow 0$
    $\ell' \leftarrow$ clonePath(ℓ)
    $C_m \leftarrow$ getArrayPointer_C(m, ℓ')
    set $C_m[0][\varphi \mod 2] \leftarrow 1$
  else// exactly one fork is good
    if contForks$[\ell][0] =$ **true then**
      set $C_m[0][\varphi \mod 2] \leftarrow 0$
    else
      set $C_m[0][\varphi \mod 2] \leftarrow 1$

Algorithm 10: 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
$\ell \leftarrow$ assignInitialPath()
$P_0 \leftarrow$ getArrayPointer_P$(0, \ell)$
for $\beta = 0, 1, \ldots, n-1$ **do**
  set $P_0[\beta][0] \leftarrow W(y_\beta|0)$, $P_0[\beta][1] \leftarrow W(y_\beta|1)$
// Main loop
for $\varphi = 0, 1, \ldots, n-1$ **do**
  for $\ell = 0, 1, \ldots, L-1$ **do**
    if pathIndexInactive(ℓ) **then**
      continue
    recursivelyCalcP(m, φ, ℓ)
  if $\hat{u}_\varphi$ *is frozen* **then**
    continuePaths_FrozenBit(φ)
  else
    continuePaths_UnfrozenBit(φ)
  if $\varphi \mod 2 = 1$ **then**
    for $\ell = 0, 1, \ldots, L-1$ **do**
      if pathIndexInactive(ℓ) **then**
        continue
      recursivelyUpdateC (m, φ, ℓ)
// Find the best codeword in the list
$\ell' \leftarrow 0$, $p' \leftarrow 0$
for $\ell = 0, 1, \ldots, L-1$ **do**
  if pathIndexInactive(ℓ) **then**
    continue
  $C_m \leftarrow$ getArrayPointer_C(m, ℓ)
  $P_m \leftarrow$ getArrayPointer_P(m, ℓ)
  if $p' < P_m[0][C_m[0][1]]$ **then**
    $\ell' \leftarrow \ell$, $p' \leftarrow P_m[0][C_m[0][1]]$
set $C_0 \leftarrow$ getArrayPointer_C$(0, \ell')$
return $\hat{\mathbf{c}} = (C_0[\beta][0])_{\beta=0}^{n-1}$

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 and E. Telatar, "On the rate of channel polarization," in *Proc. IEEE Int'l Symp. Inform. Theory (ISIT'2009)*, Seoul, South Korea, 2009, pp. 1493–1495.

[3] 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.

[4] I. Tal and A. Vardy, "How to construct polar codes," arXiv:1105.6164v1.

[5] C. Leroux, I. Tal, A. Vardy, and W. J. Gross, "Hardware architectures for successive cancellation decoding of polar codes," arXiv:1011.2919v1.

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