List Decoding of Universal Polar Codes

Boaz Shuval, Ido Tal
Department of Electrical Engineering,
Technion, Haifa 32000, Israel.
Email: {bshuval@, idotal@ee.}technion.ac.il

***Abstract*—A list decoding scheme for universal polar codes is presented. Our scheme applies to the universal polar codes first introduced by Şaşoğlu and Wang, and generalized to processes with memory by the authors. These codes are based on the concatenation of different polar transforms: a sequence of "slow" transforms and Arıkan's original "fast" transform. List decoding of polar codes has been previously presented in the context of the fast transform. However, the slow transform is markedly different and requires new techniques and data structures. We show that list decoding is possible with space complexity $O(\mathcal{L} \cdot N)$ and time complexity $O(\mathcal{L} \cdot N \log N)$, where N is the overall blocklength and $\mathcal{L}$ is the list size.**

I. Introduction

Polar codes, introduced by Arıkan [1], are a rich a family of codes. They have been extended to many settings, e.g., [2]–[14]. Specifically, in [10], a universal polar coding construction was presented. It applies to a setting where the channel is unknown to the encoder, but the decoder has full channel knowledge, obtained, e.g., by channel estimation (see also [9] for another approach). In [15], the construction of [10] was generalized to apply to a large class of channels and sources with memory. List decoding is a technique to improve upon the successive-cancellation (SC) decoding error of polar codes. Its implementation in [16] applies to Arıkan's seminal polar codes [1] (see also [17]), but does not trivially extend to the universal construction. In this paper, we present such an extension.

The universal polar codes of [10], [15] are based on concatenating recursive transforms of two types. One of which, called the *fast* transform, is Arıkan's transform [1]; the other, called the *slow* transform, is different. A key step in this concatenation is joining multiple copies of a slow transform such that their outputs are fed into multiple fast transforms. Both transforms above can be described using *layers*. Specifically, the layer keeps track of the recursion depth.

Let x_0^{N-1} denote the transform input and u_0^{N-1} its output. Consider first the fast transform [1]. Here, decoding is performed via successive-cancellation, in which symbols are decoded successively. Namely, we decode symbol u_i after having decoded the previous symbols u_0^{i-1} and under the assumption that our previous decoding decisions $\hat{u}_0^{i-1}$ are correct. The slow transform can also be decoded using successive-cancellation. A crucial property of SC decoding of the fast transform is that in decoding symbol u_i, typically only a small number of layers are involved. In stark contrast, as we will later see, in SC decoding of the slow transform, typically *all* layers are involved.

Successive-Cancellation List (SCL) decoding is a generalization of SC decoding, in which multiple "decoding paths" are considered in parallel. That is, instead of making a hard decision on the value of $\hat{u}_i$ at stage i, we allow for multiple hypotheses. We keep the list of hypotheses manageable by pruning it and keeping the $\mathcal{L}$ most-likely paths. In a fast transform of blocklength N this can be accomplished in space $O(\mathcal{L} \cdot N)$ and time $O(\mathcal{L} \cdot N \log N)$. The crucial property highlighted in the previous paragraph is what enables this for the fast transform. The crucial property does not hold for the slow transform. Thus, different techniques are needed to accomplish SCL decoding with the same space and time complexity for the universal concatenated transform. We indeed accomplish this, by exploiting a certain cyclic property of the slow transform. We leverage this property through a dedicated data-structure, termed the *cyclic exponential array*.

Due to length constraints, we focus on the slow transform and only outline the entire construction. Proofs and numerical results are omitted. A full paper with all the details is in preparation.

II. The Slow Transform

The slow transform introduced in [10] was streamlined and generalized to settings with memory in [15]. We now describe the slow transform of [15]. Unlike [15], We use zero-based indexing, as it is more amenable to implementation.

The slow transform is one-to-one and onto. It transforms a vector of N bits into another vector of N bits. The transform is recursively defined. The initial step is an identity mapping of length $N_0 = 2L_0 + M_0$. Parameters L_0 and M_0 are selected according to the memory properties of the setting and the desired rate, see [15]. Every step of the recursion doubles the transform length. After n steps it transforms vectors of length $N = 2^n N_0$.

Borrowing terminology from [16], we describe a slow transform of recursion depth n using *layers*, *phases*, and *branches* as follows. A slow transform of size $N = 2^n N_0$ — i.e., of recursion depth n — has $n+1$ layers, from 0 to n. Layer 0 is associated with the transform input x_0^{N-1}; layer n is associated with the transform output u_0^{N-1}. Each layer of the transform is divided into branches, each comprising a contiguous set of indices called phases. The number of branches in layer λ is $2^{n-\lambda}$; each branch comprises $2^\lambda N_0$ phases. The mapping between index i in layer λ and its phase φ and branch β is

$$i = \langle \varphi, \beta \rangle_\lambda \triangleq \varphi + \beta \cdot 2^\lambda N_0, \quad 0 \le \varphi < 2^\lambda N_0, \quad 0 \le \beta < 2^{n-\lambda}.$$

When the layer λ is obvious from the context, we omit it and simply write $i = \langle \varphi, \beta \rangle$.

We divide the phases within a branch β of any layer λ to several sets. As in [15], we assume that M_0 is even, and define

$$L_\lambda = 2^\lambda (L_0 + 1) - 1, \quad M_\lambda = 2^\lambda (M_0 - 2) + 2. \tag{1}$$

In other words, $L_\lambda = 2L_{\lambda-1} + 1$ and $M_\lambda = 2(M_{\lambda-1} - 1)$. Observe that $2L_\lambda + M_\lambda = 2^\lambda N_0 \triangleq N_\lambda$, the number of phases in a branch

of layer λ. The lateral set, $[\mathrm{lat}(\lambda)]$, consists of $2L_\lambda$ phases. The medial set, $[\mathrm{med}(\lambda)]$, consists of the remaining M_λ phases. These sets are further subdivided. The mapping of phases to sets in any branch of layer λ is given by:

$$[\mathrm{lat}_1(\lambda)] \triangleq \{\varphi \mid \quad 0 \le \varphi \le L_\lambda - 1\}, \tag{2a}$$
$$[\mathrm{med}_-(\lambda)] \triangleq \{\varphi \mid \varphi = L_\lambda + 2k, \quad 0 \le k < M_\lambda/2\}, \tag{2b}$$
$$[\mathrm{med}_+(\lambda)] \triangleq \{\varphi \mid \varphi = L_\lambda + 2k + 1, \ 0 \le k < M_\lambda/2\}, \tag{2c}$$
$$[\mathrm{lat}_2(\lambda)] \triangleq \{\varphi \mid L_\lambda + M_\lambda \le \varphi \le N_\lambda - 1\}, \tag{2d}$$
$$[\mathrm{lat}(\lambda)] \triangleq [\mathrm{lat}_1(\lambda)] \cup [\mathrm{lat}_2(\lambda)], \tag{2e}$$
$$[\mathrm{med}(\lambda)] \triangleq [\mathrm{med}_-(n)] \cup [\mathrm{med}_+(n)]. \tag{2f}$$

Observe that the first L_λ phases of a branch are lateral, the next M_λ phases are medial and alternate between $[\mathrm{med}_-(\lambda)]$ and $[\mathrm{med}_+(\lambda)]$, and the final L_λ indices are again lateral.

Let x_0^{N-1} be the input and u_0^{N-1} the output of a slow transform of recursion depth n. Denote index $i = \langle \varphi, \beta \rangle_\lambda$ of the vector corresponding to layer λ by $u_i^{(\lambda)} = u_{\langle \varphi, \beta \rangle}^{(\lambda)}$. In particular, $u_i^{(0)} = x_i$ and $u_i^{(n)} = u_i$. We also denote

$$\psi = \left\lfloor \frac{\varphi}{2} \right\rfloor, \quad \psi' = \left\lfloor \frac{\varphi - 1}{2} \right\rfloor.$$

Then, the slow transform recursion for $\lambda \ge 1$ is given by

$$\varphi \in [\mathrm{lat}(\lambda)] \Rightarrow u_{\langle \varphi, \beta \rangle}^{(\lambda)} = \begin{cases} u_{\langle \psi, 2\beta \rangle}^{(\lambda-1)}, & \varphi \text{ even}, \\ u_{\langle \psi, 2\beta+1 \rangle}^{(\lambda-1)}, & \varphi \text{ odd}, \end{cases} \tag{3}$$

$$\varphi \in [\mathrm{med}(\lambda)] \Rightarrow u_{\langle \varphi, \beta \rangle}^{(\lambda)} = \begin{cases} u_{\langle \psi'+1, 2\beta \rangle}^{(\lambda-1)} + u_{\langle \psi', 2\beta+1 \rangle}^{(\lambda-1)}, & \varphi \text{ odd}, \\ u_{\langle \psi', 2\beta+1 \rangle}^{(\lambda-1)}, & \varphi \text{ even}, \ \psi' \in [\mathrm{med}_-(\lambda - 1)], \\ u_{\langle \psi'+1, 2\beta \rangle}^{(\lambda-1)}, & \varphi \text{ even}, \ \psi' \in [\mathrm{med}_+(\lambda - 1)]. \end{cases} \tag{4}$$

Observe from (1), (2b), and (2c) that since $\lambda \ge 1$, $\varphi \in [\mathrm{med}(\lambda)]$ is even if and only if $\varphi \in [\mathrm{med}_+(\lambda)]$. When $\lambda \ge 2$, we have

$$\psi' \in [\mathrm{med}_-(\lambda - 1)] \iff \psi' \text{ is odd},$$
$$\psi' \in [\mathrm{med}_+(\lambda - 1)] \iff \psi' \text{ is even}.$$

By (3) and (4), branch β of layer λ is formed from branches 2β and $2\beta + 1$ of layer $\lambda - 1$. From (1) and (2), *all* lateral phases of branches 2β, $2\beta + 1$ of layer $\lambda - 1$ are transformed into lateral phases of branch β of layer λ. Additionally, medial phases $\langle L_{\lambda-1}, 2\beta \rangle_{\lambda-1}$ and $\langle L_{\lambda-1} + M_{\lambda-1} - 1, 2\beta + 1 \rangle_{\lambda-1}$ become lateral phases of layer λ.

The operation in (4) consists of minus and plus transforms: a minus transform for odd medial φ, and a plus transform for even medial φ. Equation (4) reveals a cardinal difference between the slow transform and Arıkan's fast transform. In the fast case, the minus transform operates on the *same* phase of two consecutive branches. In the slow case, the minus transform operates on *consecutive* phases of two consecutive branches.

III. Successive-Cancellation for the Slow Transform

The original decoding algorithm of polar codes is successive-cancellation. More generally, SC is also employed for encoding [8]. Better coding results can be obtained using SCL decoding. However, we first discuss SC decoding.

The universal scheme [10], [15] employs a joint transform consisting of slow and fast transforms. The joint transform is recursive as well, and hence conveniently described via hyperlayers, hyperbranches, and hyperphases — to be detailed in the full paper. SC can be used for encoding and decoding this joint transform. Due to length constraints, we focus our discussion on SC for the slow transform — a cardinal building block. We remark that the slow transform, used exclusively, is not sufficient for coding as it polarizes too slowly. Our full paper will provide details on the joint transform.

SC is used in a probabilistic setting. Denote random variables using capital letters. For channel coding, X_0^{N-1} is the channel input and Y_0^{N-1} is the corresponding output. Their joint probability is governed by a hidden Markov state chain (see [15] for full details of the model): $\mathbb{P}(X_i, Y_i, S_i | S_{i-1})$, where S_i is the state at time i. The states belong to a finite set $\mathcal{S}$. We also denote $\mathcal{X} = \{0, 1\}$.

Algorithm 1 is a general high-level description of SC for the above setting. For each of the N phases, we first compute

$$p(u, s, s'; \hat{u}_0^{\varphi-1}, y_0^{N-1}) \tag{5}$$
$$\triangleq \mathbb{P}\left(U_\varphi = u, S_{-1} = s, S_{N-1} = s'; \ U_0^{\varphi-1} = \hat{u}_0^{\varphi-1}, Y_0^{N-1} = y_0^{N-1}\right),$$

where U_0^{N-1} is the transform of X_0^{N-1}. Every phase φ is either used to carry information bits (such a phase is called a *data phase*) or not. In [1], non-data phases were called 'frozen.' More generally [8], these are shaping phases. Either way, for non-data phases u_φ is determined via a mapping[1] $\mathcal{F}_\varphi(u_0^{\varphi-1})$. In [8], this mapping utilizes common randomness between encoder and decoder. When φ is a data phase, we determine $\hat{u}_\varphi$ using a maximum aposteriori criterion. That is, we compute

$$p(u; \hat{u}_0^{\varphi-1}, y_0^{N-1}) \triangleq \sum_{s, s' \in \mathcal{S}} p(u, s, s'; \hat{u}_0^{\varphi-1}, y_0^{N-1}), \tag{6}$$

and select $\hat{u}_\varphi = \arg\max_{u \in \mathcal{X}} p(u; \hat{u}_0^{\varphi-1}, y_0^{N-1})$.

Algorithm 1: A high-level description of SC

Input: received y_0^{N-1} (or empty vector for encoding)
Output: transformed codeword $\hat{u}_0^{N-1}$

```
for φ = 0, 1, ..., N − 1 do
    compute p(u, s, s'; û_0^{φ−1}, y_0^{N−1}) for u ∈ X; s, s' ∈ S
    if φ is a frozen (shaping) phase then
        set û_φ ← F_φ(û_0^{φ−1})
    else
        set û_φ ← arg max_{u∈X} p(u; û_0^{φ−1}, y_0^{N−1})
return û_0^{N−1}
```

A. A first implementation of Algorithm 1

Our first implementation mirrors Algorithms 1 – 4 of [16], modified and generalized to the slow transform and to settings with memory. Thus, we first employ straightforward data structures: arrays. Later, when considering list decoding, we will show that the space complexity can be reduced using enhanced data structures.

[1] Note that the Cyclic Redundancy Check (CRC) variant [16] of polar codes places CRC bits in certain phases. Under our definition, these are also shaping phases. The same comment applies for the dynamically frozen bits of [18].

To implement Algorithm 1, we need a way to compute $p(u,s,s';\hat{u}_0^{\varphi-1},y_0^{N-1})$. This is accomplished using the recursive description (3) and (4). The intermediate calculations required for computing $p(u,s,s';\hat{u}_0^{\varphi-1},y_0^{N-1})$ are common between different phases φ. We store some of the intermediate calculations for time complexity reduction.

Our implementation utilizes two main data structures: one for keeping track of intermediate bit decisions and the other for storing intermediate probabilities. Specifically, for each layer $0 \le \lambda \le n$ we define a *bit-decision array* B_λ of size $N = 2^\lambda N_0 \cdot 2^{n-\lambda}$. The array starts out uninitialized, and when the algorithm concludes it holds bit decisions:

$$B_\lambda[\langle\varphi,\beta\rangle] = \hat{u}^{(\lambda)}_{\langle\varphi,\beta\rangle}.$$

We further define, for each layer λ, a *probabilities array* P_λ of size $N \times |\mathcal{X}| \times |\mathcal{S}| \times |\mathcal{S}|$. When the algorithm concludes,

$$P_\lambda[\langle\varphi,\beta\rangle][u,s,s'] = p^{(\lambda)}_{\langle\varphi,\beta\rangle}(u,s,s'),$$

where we define $p^{(\lambda)}_{\langle\varphi,\beta\rangle}(u,s,s')$ in (7). For brevity, we denote $\Lambda = 2^\lambda N_0$, $\tilde{S} = S_{\beta\Lambda-1}$, $\tilde{S}' = S_{(\beta+1)\Lambda-1}$, $v_\varphi = u^{(\lambda)}_{\langle\varphi,\beta\rangle}$, $\hat{v}_\varphi = \hat{u}^{(\lambda)}_{\langle\varphi,\beta\rangle}$, and $\tilde{y}_\varphi = y_{\langle\varphi,\beta\rangle_\lambda}$. Capital versions of v_φ and $\tilde{y}_\varphi$ denote random variables. Then,

$$p^{(\lambda)}_{\langle\varphi,\beta\rangle}(u,s,s') \tag{7}$$
$$= \mathbb{P}\left(V_\varphi = u, \tilde{S} = s, \tilde{S}' = s'; V_0^{\varphi-1} = \hat{v}_0^{\varphi-1}, \tilde{Y}_0^{\Lambda-1} = \tilde{y}_0^{\Lambda-1}\right).$$

Observe that when $\lambda = 0$ then $V_\varphi = X_\varphi$, the channel input, and $\Lambda = N_0$. Thus, $p^{(0)}_{\langle\varphi,\beta\rangle}$ involves a sub-vector of the output Y_0^{N-1} of size N_0. Due to the Markov property, we can compute $p^{(0)}_{\langle\varphi,\beta\rangle}$ directly from the joint distribution $\mathbb{P}(X_i,Y_i,S_i|S_{i-1})$. Observe that when $\lambda = n$, $p^{(n)}_{\langle\varphi,\beta\rangle}(u,s,s') = p(u,s,s';\hat{u}_0^{\varphi-1},y_0^{N-1})$.

Our implementation is given in Algorithm 2. Its main loop (lines 7 – 15) iterates over all phases of the single branch of the last layer n. For each last-layer phase it recursively calculates relevant probabilities of the probabilities array (line 8), decides on the value of the last-layer phase (lines 9 – 12), and finally propagates this value throughout the transform (lines 13 – 15).

An additional array in our implementation is the *tracker* $T_\lambda[i]$, $i \in \{0,1\}$. For each layer $0 \le \lambda < n$, each of its two elements is either empty or holds a phase-branch pair $(\bar{\varphi},\bar{\beta})$. A `resetTracker` function sets all of its elements over all layers to empty. Whenever $B_\lambda[\langle\varphi,\beta\rangle]$ is updated, the tracker is also updated. As will soon become apparent, no more than two phase-branch pairs $(\bar{\varphi},\bar{\beta})$ are updated per layer λ in an iteration of the main loop of Algorithm 2. Thus, $T_\lambda[i] = (\bar{\varphi},\bar{\beta})$ means that in the previous iteration of the main loop, phase $\bar{\varphi}$ of branch $\bar{\beta}$ in layer λ was updated.

We will soon see that recursive probability calculation needs to know which phases and branches were updated in every layer in the previous iteration of the main loop of Algorithm 2. This is accomplished via the tracker array. Specifically, before propagating bit decisions throughout the transform, we reset the tracker (line 14 in Algorithm 2).

Algorithm 3, invoked with `recursivelyCalcP`(λ,φ,β), computes $p^{(\lambda)}_{\langle\varphi,\beta\rangle}(u,s,s')$ for all $u \in \mathcal{X}$ and $s,s' \in \mathcal{S}$. It does this by utilizing the relationships in equations (3) and (4) and the Markov property. However, it must first ensure that the relevant indices in $P_{\lambda-1}$ had been computed. The branches and phases in layer $\lambda-1$ for which $P_{\lambda-1}$ is updated depend on λ, φ, and β. There are three cases.

1) When $\lambda = 0$ (line 6), we update the base probabilities if needed. Efficient implementation, especially when list decoding is involved, requires another algorithm, `updateBaseProbs`, see our full paper.
2) If $\lambda > 0$ and $\varphi = 0$ (line 9), no bit decisions had been propagated, and we recurse to the previous layer.
3) Otherwise, we use the tracker array $T_{\lambda-1}$ (line 16). Note that this calls upon a 'next phase' $\bar{\varphi}+1$ in a branch. This 'next phase' may exceed the size of the branch. This happens when all phases in a branch had their bit decisions propagated to. Thus (lines 1–4) we set to zero $p^{(\lambda-1)}_{\langle\bar{\varphi},\bar{\beta}\rangle}(u,\sigma,\sigma')$ for all $u \ne B_{\lambda-1}[\langle\bar{\varphi},\bar{\beta}\rangle]$, and $\sigma,\sigma' \in \mathcal{S}$.

Finally, in lines 17–20, `recursivelyCalcP`(λ,φ,β) computes $p^{(\lambda)}_{\langle\varphi,\beta\rangle}(\cdot,\cdot,\cdot)$. The computation depends on the type of phase φ: lateral or medial, and uses equations (3) and (4), respectively. It also relies on the Markov property and the adaptation of minus and plus transforms to this case, see [19], [20]. Details of `lateralProbHelper` and `medialProbHelper` will appear in our full paper.

Algorithm 4 resolves equations (3) and (4) recursively: it is invoked after $B_\lambda[\langle\varphi,\beta\rangle]$ had been set, and propagates this throughout the transform. The recursive computation is performed from layer λ to layer $\lambda-1$, the opposite direction to that of equations (3) and (4). When $\lambda = 0$, `recursivelyUpdateB`(λ,φ,β) cannot propagate to a previous layer, so it simply returns. Otherwise, its operation depends on the type of phase φ.

1) $\varphi \in [\text{lat}(\lambda)]$ (lines 3–8). By (3), a lateral phase of layer λ passes-through directly to a single phase of layer $\lambda-1$. Thus, only a single phase-branch pair of $B_{\lambda-1}$ is updated, and the algorithm recurses to layer $\lambda-1$. After every update of the bit-decision array we also update the tracker. In this case, $T_{\lambda-1}$ has only one non-empty entry.
2) $\varphi \in [\text{med\_}(\lambda)]$ (line 1). Medial phases come in minus and plus pairs, in this order. Both members of the pair

Algorithm 2: First implementation of SC decoder

Input: received y_0^{N-1} (empty vector for encoding)
Output: transformed codeword $\hat{u}_0^{N-1}$

1 **for** $\beta = 0,1,\ldots,2^n-1$ **do** // Initialization
2 **for** $u \in \mathcal{X}, s \in \mathcal{S}, s' \in \mathcal{S}$ **do**
3 set $\varphi \leftarrow 0$ // Other phases updated later
4 $P_0[\langle\varphi,\beta\rangle][u,s,s'] \leftarrow p^{(0)}_{\langle\varphi,\beta\rangle}(u,s,s')$
5 resetTracker()
6 set $\beta \leftarrow 0$ // The only branch of layer n
7 **for** $\varphi = 0,1,\ldots,N-1$ **do** // Main loop
8 recursivelyCalcP(n,φ,β)
9 **if** *φ is a frozen (shaping) phase* **then**
10 set $\hat{u}_\varphi \leftarrow \mathcal{F}_\varphi(\hat{u}_0^{\varphi-1})$
11 **else**
12 set $\hat{u}_\varphi \leftarrow \arg\max_{u\in\mathcal{X}} p(u;\hat{u}_0^{\varphi-1},y_0^{N-1})$
13 set $B_n[\langle\varphi,\beta\rangle] \leftarrow \hat{u}_\varphi$
14 resetTracker()
15 recursivelyUpdateB(n,φ,β)
16 **return** $\hat{u}_0^{N-1}$

Algorithm 3: `recursivelyCalcP`$(\lambda, \varphi, \beta)$

Input: λ = layer, β = branch in layer, φ = phase in branch

```
1  if φ = 2^λ N_0 then // after last phase in branch
2      for u ∈ 𝒳 do
           // Set probability zero to u different
           than last decision
3          if u ≠ B_λ[⟨φ − 1, β⟩] then
4              for s, s′ ∈ 𝒮 do P_λ[⟨φ − 1, β⟩][u, s, s′] ← 0
5      return
6  if λ = 0 then
7      if φ > 0 then updateBaseProbs(φ, β)
8      return // Stopping condition
9  if φ = 0 then
10     recursivelyCalcP(λ − 1, φ, 2β)
11     recursivelyCalcP(λ − 1, φ, 2β + 1)
12 else // φ > 0
13     for i ∈ {0, 1} do
14         if T_{λ−1}[i] is not empty then
15             (φ̄, β̄) ← T_{λ−1}[i]
               // Prepare next phase in branch
16             recursivelyCalcP(λ − 1, φ̄ + 1, β̄)
   // Compute P_λ[⟨φ, β⟩][u, s, s′] for all u, s, s′
17 if φ ∈ [lat(λ)] then
18     lateralProbHelper(λ, φ, β) // use (3)
19 else if φ ∈ [med(λ)] then
20     medialProbHelper(λ, φ, β) // use (4)
21 return
```

are required to resolve (4) for $\lambda - 1$. Since this is the first member of the pair, we must wait. Nothing is updated, so return without recursing.

3) $\varphi \in [\text{med}_+(\lambda)]$ (lines 9–21). We now have the left-hand side of (4) for two consecutive phases, a minus and a plus pair (for $\varphi - 1$ odd and φ even), and can resolve for the right-hand side, namely layer $\lambda - 1$. Two phase-branch pairs of $B_{\lambda-1}$ are updated, and entered into the tracker $T_{\lambda-1}$. The algorithm recurses for these two pairs.

Algorithm 4 highlights an important observation on the slow transform. Recall from (1), (2), and (4) that consecutive medial phases of layer λ are formed from two medial phases of layer $\lambda - 1$. One of these phases is in $[\text{med}_-(\lambda - 1)]$ and the other is in $[\text{med}_+(\lambda - 1)]$. Thus, when the algorithm is invoked for phase $\varphi \in [\text{med}_+(\lambda)]$ it recursively invokes the algorithm twice, once for a phase in $[\text{med}_-(\lambda - 1)]$ and once for a phase in $[\text{med}_+(\lambda - 1)]$. Hence, when Algorithm 2 invokes `recursivelyUpdateB`$(n, \varphi, 0)$ for $\varphi \in [\text{med}_+(n)]$, two entries of the bit-decision array will be updated for *every* layer $0 \le \lambda \le n-1$. This is in stark contrast to the fast transform (which typically updates the bit-decision array for only a few layers), and the reason that the list decoder implementation of [16] does not carry through.

The following lemma will be crucial for the list decoder's bookkeeping. To this end, we first define a *branc*.

Definition 1 (branc)**.** A *branc* contains two consecutive branches. The branc of branch β is numbered $\lfloor \beta/2 \rfloor$. In other words, branches β and $\beta + 1$ are in the same branc if their bit-expansions are equal up to the least significant bit.[2]

[2]A 'branc' is a 'branch' whose "least significant letter," 'h', is dropped.

Algorithm 4: `recursivelyUpdateB`$(\lambda, \varphi, \beta)$

Input: λ = layer, β = branch in layer, φ = phase in branch

```
1  if λ = 0 or φ ∈ [med_−(λ)] then
2      return // Only medial plus or lateral
        phases are propagated
3  if φ ∈ [lat(λ)] then
4      set ψ ← ⌊φ/2⌋
5      set β̄ ← 2β + (φ mod 2) // See (3)
6      B_{λ−1}[⟨ψ, β̄⟩] ← B_λ[⟨φ, β⟩]
7      T_{λ−1}[0] ← (ψ, β̄)
8      recursivelyUpdateB(λ − 1, ψ, β̄)
9  else // See (4)
10     set ψ′ ← ⌊(φ − 1)/2⌋
11     if φ ∈ [med_+(λ)] then
12         if ψ′ ∈ [med_−(λ − 1)] then
13             B_{λ−1}[⟨ψ′ + 1, 2β⟩] ← B_λ[⟨φ, β⟩] + B_λ[⟨φ − 1, β⟩]
14             B_{λ−1}[⟨ψ′, 2β + 1⟩] ← B_λ[⟨φ, β⟩]
15         else
16             B_{λ−1}[⟨ψ′, 2β + 1⟩] ← B_λ[⟨φ, β⟩] + B_λ[⟨φ − 1, β⟩]
17             B_{λ−1}[⟨ψ′ + 1, 2β⟩] ← B_λ[⟨φ, β⟩]
18         T_{λ−1}[0] ← (ψ′, 2β + 1)
19         T_{λ−1}[1] ← (ψ′ + 1, 2β)
20         recursivelyUpdateB(λ − 1, ψ′, 2β + 1)
21         recursivelyUpdateB(λ − 1, ψ′ + 1, 2β)
```

For example, there are eight brancs for layer $\lambda = 4$: branc $0 = \langle 000 \rangle_2$ contains branches $0 = \langle 0000 \rangle_2$ and $1 = \langle 0001 \rangle_2$, branc 1 contains branches 2 and 3, etc.

We order brancs in bit-reversed cyclic order. Thus, for $\lambda = 4$, brancs are ordered: $0 = \langle 000 \rangle_2$, $4 = \langle 100 \rangle_2$, $2 = \langle 010 \rangle_2$, $6 = \langle 110 \rangle_2$, $1 = \langle 001 \rangle_2$, $5 = \langle 101 \rangle_2$, $3 = \langle 011 \rangle_2$, $7 = \langle 111 \rangle_2$. Since the order is cyclic, the next branc after 7 is 0.

We say that a branc of B_λ is updated if B_λ is updated for at least one of the branches β in the branc. Namely, $B_\lambda[\varphi, \beta]$ is updated for some phase φ and branch β in the branc. A similar definition holds for P_λ.

Lemma 1. *For each layer λ, the brancs of B_λ are updated in bit-reversed cyclic order during the entire run of Algorithm 2.*

Corollary 2. *For each layer λ, the brancs of P_λ are updated in bit-reversed cyclic order during the entire run of the main loop of Algorithm 2, save for the first iteration, $\varphi = 0$.*

The following theorem reduces the space complexity.

Theorem 3. *Algorithms 2 to 4 can be implemented with per-layer bit-decision arrays and probabilities arrays indexed only by branch.*

I.e., the bit-decision array can be indexed as $B_\lambda[\beta]$ and the probabilities array as $P_\lambda[\beta](u, s, s')$. Their entries will refer to the last updated phase in the relevant layer and branch. Note that this entails changing the interface of `recursivelyUpdateB` to also pass the bit-decision of the previous phase. Namely, lines 13, 16, 20, 21 of Algorithm 4 must be changed.

IV. List Decoding and the Cyclic Exponential Array

List decoding uses a number, $\mathcal{L}$, of decoding paths. A path up to phase φ is split at the decision point (the bit decision for phase φ in the last layer n, when φ is a data phase). Thus,

the number of paths is doubled at every split. If this number exceeds $\mathcal{L}$, we prune the list and keep the $\mathcal{L}$ most likely paths.

The paths differ in their bit decisions in several places, and consequently the arrays B_λ and P_λ differ for different paths. The essence of an *efficient* implementation of list decoding is to share portions of these arrays among paths. Namely, if a portion of an array is the same for two paths, we store it in memory once. Conversely, array portions that are to be written to by a path at the current phase must not be shared.

The universal transform is formed by concatenating a sequence of slow transforms and a final fast transform. A key building block in list decoding for the universal transform is the *cyclic exponential array* (CEA). This is a data structure that enables sharing array portions efficiently. We focus on the slow transform, due to space limitations.

Recall from Theorem 3 that the arrays B_λ and P_λ may be indexed by branch only. Further note from Lemma 1 and Corollary 2 that these arrays are updated in a cyclic order.

The CEA data structure holds generic objects:

- For B_λ it holds pairs of bit decisions, one pair for each branc.
- For P_λ it holds pairs of 'probability datums,' one pair for each branc. A probability datum holds $|\mathcal{X}| \cdot |\mathcal{S}|^2$ probabilities, indexed by u, s, and s'.

A CEA contains 2^λ objects for some λ. The CEA supports two operations: `read`(i) and `write`(i). The operation `read`(i) returns the object stored at position i of the CEA. The operation `write`(i) stores an object at position i of the CEA. The first `write` must be called with index $i = 0$. For subsequent calls, if the previous call of `write` was with index i, the current call must be with either index i or $i + 1$, modulo the CEA size 2^λ. Bit-reversal is performed by the caller.

Internally, a CEA of size 2^λ holds the following variables:

- `lastIndexWrittenTo`: the last index written to by `write`.
- `lastWrittenValue`: the object last written by `write`.
- Arrays $\texttt{currentCycleArray}_\tau$, $0 \le \tau < \lambda$. Array $\texttt{currentCycleArray}_\tau$ holds 2^τ objects; its indexing is zero-based.
- `previousCycleArray`: holds 2^λ objects; its indexing is zero-based.

When $\texttt{lastIndexWrittenTo} = i$,

- `lastWrittenValue` holds the object written to by the latest write, `write`(i).
- For $j > i$, $\texttt{previousCycleArray}[j]$ holds the object written to by the latest `write`(j).
- For $j < i$, the object written to by the latest `write`(j) is in $\texttt{currentCycleArray}_\tau[k]$, where τ and k are computed as follows. Let $i = \langle i_{\lambda-1} i_{\lambda-2} \cdots i_0 \rangle_2$ and $j = \langle j_{\lambda-1} j_{\lambda-2} \cdots j_0 \rangle_2$ be the binary bit expansions of i and j respectively, with i_0 (j_0) the least significant bit of i (j). Then, τ is the largest integer such that $i_\tau = 1$ and $j_\tau = 0$, and $k = \langle j_\tau j_{\tau-1} \cdots j_0 \rangle_2$.

Example 1. Let $\lambda = 4$ and $\texttt{lastIndexWrittenTo} = 11 = \langle 1011 \rangle_2$. Then, for $0 \le j \le 15$, the location that `read`(j) will access is the cell numbered j in the following:

$$\begin{aligned}
\texttt{currentCycleArray}_3 &\equiv \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{bmatrix}, \\
\texttt{currentCycleArray}_2 &\equiv \begin{bmatrix} \text{N/A} & \text{N/A} & \text{N/A} & \text{N/A} \end{bmatrix}, \\
\texttt{currentCycleArray}_1 &\equiv \begin{bmatrix} 8 & 9 \end{bmatrix}, \\
\texttt{currentCycleArray}_0 &\equiv \begin{bmatrix} 10 \end{bmatrix}, \\
\texttt{lastWrittenValue} &\equiv 11, \\
\texttt{previousCycleArray} &\equiv \begin{bmatrix} \text{N/A} \cdots \text{N/A} & 12 & 13 & 14 & 15 \end{bmatrix}.
\end{aligned}$$

The only legal `write` operations are `write`(11) and `write`(12): `write`(11) changes `lastWrittenValue` only; `write`(12) first copies into $\texttt{currentCycleArray}_2$ objects from `lastWrittenValue` and $\texttt{currentCycleArray}_\tau$, $\tau = 0, 1$. Then, it changes `lastWrittenValue` and `lastIndexWrittenTo`. Crucially, the two largest arrays, `previousCycleArray` and $\texttt{currentCycleArray}_3$ are unchanged.

The following theorem details which internal variables of a CEA are changed during a `write` operation. This is crucial with respect to list decoding. Namely, it details which variables can be shared among paths after a split, and which cannot. To this end, the binary bit expansions of i and j respectively are $i = \langle i_{\lambda-1} i_{\lambda-2} \cdots i_0 \rangle_2$ and $j = \langle j_{\lambda-1} j_{\lambda-2} \cdots j_0 \rangle_2$.

Theorem 4. *Let* $\texttt{lastIndexWrittenTo} = i$ *and consider* `write`(j) *for* $j = (i + 1) \bmod 2^\lambda$. *Apart from* `lastWrittenValue` *and* `lastIndexWrittenTo`, *a single array is changed:*

- *If* $j = 0$, *only* `previousCycleArray` *is changed.*
- *Otherwise, let* τ *be the largest integer such that* $j_\tau = 1$ *and* $i_\tau = 0$. *Then, only* $\texttt{currentCycleArray}_\tau$ *is changed.*

Corollary 5. *Let* i, j, *and* τ *be as in Theorem 4, and let* 2^λ *be the CEA size. Then the time complexity of* `write`(j) *is* $O(2^\lambda)$ *if* $j = 0$ *and* $O(2^\tau)$ *otherwise. Thus, a sequence of* 2^λ *write operations spanning all indices* j *takes time* $O(\lambda \cdot 2^\lambda)$.

The number of layers $n+1$ in a slow transform of blocklength $N = 2^n N_0$ is $O(\log N)$. Thus,

Corollary 6. *SCL for a slow transform of length* N *can be accomplished with space complexity* $O(\mathcal{L} \cdot N)$ *and time complexity* $O(\mathcal{L} \cdot N \log^2 N)$.

One might infer from Corollary 6 that the overall time complexity of SCL decoding of a universal transform of blocklength N is $O(\mathcal{L} \cdot N \log^2 N)$. This would happen in a straightforward implementation. However, the universal transform has a parallel structure, in which multiple identical slow transforms are decoded in lockstep. This allows for significant savings in time complexity, by having the CEA objects be pointers to arrays whose length is the number of slow transform copies. Copying an object is simply copying a pointer, and a single copy operation suffices for all parallel slow transforms. Hence, the bookkeeping associated with all parallel slow transforms is not a function of the number of parallel transforms, only the blocklength of a single slow transform. Thus, it is possible to show the following.

Theorem 7. *SCL for a universal transform of blocklength* N *can be accomplished with space complexity* $O(\mathcal{L} \cdot N)$ *and time complexity* $O(\mathcal{L} \cdot N \log N)$.

REFERENCES

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

[2] E. Şaşoğlu, E. Telatar, and E. Arıkan, "Polarization for arbitrary discrete memoryless channels," in *2009 IEEE Information Theory Workshop*, October 2009, pp. 144–148.

[3] S. B. Korada and R. L. Urbanke, "Polar codes are optimal for lossy source coding," *IEEE Transactions on Information Theory*, vol. 56, no. 4, pp. 1751–1768, April 2010.

[4] E. Arıkan, "Source polarization," in *2010 IEEE Int. Sym. on Information Theory*, June 2010, pp. 899–903.

[5] E. Hof and S. Shamai, "Secrecy-achieving polar-coding," in *2010 IEEE Information Theory Workshop*, August 2010, pp. 1–5.

[6] S. B. Korada, E. Şaşoğlu, and R. Urbanke, "Polar codes: Characterization of exponent, bounds, and constructions," *IEEE Transactions on Information Theory*, vol. 56, no. 12, pp. 6253–6264, Dec 2010.

[7] H. Mahdavifar and A. Vardy, "Achieving the secrecy capacity of wiretap channels using polar codes," *IEEE Transactions on Information Theory*, vol. 57, no. 10, pp. 6428–6443, October 2011.

[8] J. Honda and H. Yamamoto, "Polar coding without alphabet extension for asymmetric models," *IEEE Transactions on Information Theory*, vol. 59, no. 12, pp. 7829–7838, December 2013.

[9] S. H. Hassani and R. Urbanke, "Universal polar codes," in *2014 IEEE International Symposium on Information Theory*, June 2014, pp. 1451–1455.

[10] E. Şaşoğlu and L. Wang, "Universal polarization," *IEEE Transactions on Information Theory*, vol. 62, no. 6, pp. 2937–2946, June 2016.

[11] E. Şaşoğlu and I. Tal, "Polar coding for processes with memory," *IEEE Transactions on Information Theory*, vol. 65, no. 4, pp. 1994–2003, April 2019.

[12] B. Shuval and I. Tal, "Fast polarization for processes with memory," *IEEE Transactions on Information Theory*, vol. 65, no. 4, pp. 2004–2020, April 2019.

[13] D. Goldin and D. Burshtein, "Performance bounds of concatenated polar coding schemes," *IEEE Transactions on Information Theory*, vol. 65, no. 11, pp. 7131–7148, Nov 2019.

[14] I. Tal, H. D. Pfister, A. Fazeli, and A. Vardy, "Polar codes for the deletion channel: Weak and strong polarization," 2019. [Online]. Available: http://arxiv.org/abs/1904.13385

[15] B. Shuval and I. Tal, "Universal polarization for processes with memory," 2018. [Online]. Available: http://arxiv.org/abs/1811.05727

[16] I. Tal and A. Vardy, "List decoding of polar codes," *IEEE Transactions on Information Theory*, vol. 61, no. 5, pp. 2213–2226, May 2015.

[17] I. Dumer and K. Shabunov, "Soft decision decoding of reed-muller codes: recursive lists," *IEEE Transactions on Information Theory*, vol. 52, no. 3, pp. 1260–1266, March 2006.

[18] P. Trifonov and V. Miloslavskaya, "Polar codes with dynamic frozen symbols and their decoding by directed search," in *2013 IEEE Information Theory Workshop (ITW)*, Sep. 2013, pp. 1–5.

[19] R. Wang, R. Liu, and Y. Hou, "Joint successive cancellation decoding of polar codes over intersymbol interference channels," *CoRR*, vol. abs/1404.3001, 2014. [Online]. Available: http://arxiv.org/abs/1404.3001

[20] R. Wang, J. Honda, H. Yamamoto, R. Liu, and Y. Hou, "Construction of polar codes for channels with memory," in *2015 IEEE Information Theory Workshop*, October 2015, pp. 187–191.