

Art of Constructing Low-complexity Encoders/Decoders for Constrained Block Codes*

Dharmendra S. Modha and Brian H. Marcus
IBM Almaden Research Center
650 Harry Road, San Jose, CA 95120-6099
emails: {dmodha,marcus}@almaden.ibm.com

April 4, 1999

Abstract

A rate $p : q$ block encoder is a dataword-to-codeword assignment from 2^p p -bit datawords to 2^q q -bit codewords, and the corresponding block decoder is the inverse of the encoder. When designing block encoders/decoders for constrained systems, often, more than 2^p codewords are available. In this paper, as our main contribution, we propose efficient heuristic computer algorithms to (i) eliminate the excess codewords; and (ii) to construct low hardware complexity block encoders/decoders. For $(0, 4/4)$ and $(0, 3/6)$ PRML constraints, block encoders/decoders generated using the proposed algorithms are comparable in complexity to human-generated encoders/decoders, but are significantly simpler than lexicographical encoders/decoders.

Keywords: binary trees, dataword-to-codeword assignment, excess codewords, logic synthesis, mapping-by-gated-partitioning, modulation codes, PRML, RLL

*This manuscript is for peer review purposes only. Please do not distribute.

1 Introduction

Constrained coding is used in magnetic recording systems to encode unconstrained user sequences into channel output sequences that satisfy certain hard constraints such as various limits on the run lengths of zeroes, see, for example, Immink [1], Marcus, Siegel, and Wolf [2], and Immink, Siegel, and Wolf [3]. Block codes have been widely used for converting unconstrained user sequences into desired constraint sequences. The basic idea in a rate $p : q$ block code is to identify a codebook containing 2^p q -bit codewords that satisfy the desired constraint, and to design an encoder that assigns each 2^p p -bit dataword in a one-to-one and onto fashion to a q -bit codeword in the codebook. In other words, a block encoder is a dataword-to-codeword assignment. The corresponding block decoder is the inverse mapping or the codeword-to-dataword assignment.

We motivate the problem of interest using a concrete example of $(d, k) = (0, 2)$ run-length limited (RLL) constraint which demands that runs of consecutive symbols “0” must not be more than 2. We are interested in a rate 4 : 5 block code for this constraint. A set of valid 5-bit codewords for this constraint can be obtained by starting from all 5-bit words and eliminating all words that have more than two consecutive symbols “0” anywhere in the words and by eliminating all words that have more than one symbol “0” at the beginning or at the end of the word. This process leaves a set of 17 codewords which can be freely concatenated without violating the constraint. Using these 17 codewords, in Table 1 we display three different block encoders. The first encoder corresponds to the classical “GCR code” [2, Table III], the second encoder has been constructed using the methods introduced in this paper, and the third encoder has been constructed by in a lexicographical fashion, that is, by lexicographically ordering all 17 codewords, deleting the “largest” codeword, and then assigning a 4-bit dataword to each remaining codeword corresponding to the codeword’s rank in such an ordering. These three codes differ from each other in (a) the choice of codebook, that is, the GCR code and the lexicographical code ignore the codeword “11111” while the new code ignores the codeword “10101”; and in (b) the choice of encoder. While all three encoders have the same rate 4 : 5, they require different hardware complexities to implement as shown in Table 2. The “AREA” and “GATES” numbers reported in Table 2 (and throughout this paper) were generated using the Berkeley SIS logic synthesis program [4]; we used the SA-12E cell library for the CMOS 6SF technology [5]. It can be seen from Table 2 that the new code has the lowest complexity, the GCR code has complexity quite comparable to the new code, and the lexicographical code has relatively large complexity. We say that the GCR code and the new code exploit the degrees of freedom permitted in the choice of the 16 word codebook and in the choice of the dataword-to-codeword assignment to construct low-complexity block encoders/decoders.

We now cast the problem of interest in formal notation. For $r \geq 1$, define a r -bit *word* as an r -tuple of bits, that is, as an element of $\{0, 1\}^r$. Let $\mathcal{X} = \{0, 1\}^p$ denote the set of all

Datawords	GCR	New	Lexicographical
excess	11111	10101	11111
0000	11001	10010	01001
0001	11011	10110	01010
0010	10010	10011	01011
0011	10011	10111	01101
0100	11101	01001	01110
0101	10101	01101	01111
0110	10110	11001	10010
0111	10111	11101	10011
1000	11010	01010	10101
1001	01001	01110	10110
1010	01010	01011	10111
1011	01011	01111	11001
1100	11110	11010	11010
1101	01101	11110	11011
1110	01110	11011	11101
1111	01111	11111	11110

Table 1: Various block encoders for the $(0, 2)$ RLL constraint

		GCR	New	Lexicographical
ENCODER	AREA	39	31	91
	GATES	8	8	19
DECODER	AREA	79	53	135
	GATES	18	13	29

Table 2: Complexities of various encoders/decoders for the $(0, 2)$ RLL constraint

p-bit *datawords* or *userwords*. Clearly, $|\mathcal{X}| = 2^p$. Let $\mathcal{Y} \subset \{0, 1\}^q$ denote the set of all q-bit *codewords* satisfying some constraint. Clearly, $|\mathcal{Y}| \leq 2^q$. We assume that $|\mathcal{Y}| \geq |\mathcal{X}| = 2^p$, that is, we have at least as many codewords as there are datawords. A *codebook* \mathcal{C} is defined as a set of q-bit codewords in \mathcal{Y} such that $|\mathcal{C}| = |\mathcal{X}|$. In other words, the codebook \mathcal{C} is obtained from the set of valid codewords \mathcal{Y} by deleting “excess codewords.” A *block encoder* \mathcal{E} is defined a one-to-one mapping from the set of datawords \mathcal{X} onto the codebook \mathcal{C} . A *block decoder* $\mathcal{D} : \mathcal{C} \rightarrow \mathcal{X}$ is simply the inverse of the block encoder. In this paper, we are interested in:

- selecting a codebook \mathcal{C} ; and in
- constructing a “low complexity” encoder \mathcal{E} and a corresponding decoder \mathcal{D} .

Observe that there are

$$\binom{|\mathcal{Y}|}{2^p}$$

ways to select a codebook \mathcal{C} from \mathcal{Y} . Furthermore, given a codebook \mathcal{C} , there are $(2^p)!$ ways to select an encoder \mathcal{E} . Thus, all together, there are

$$\frac{(|\mathcal{Y}|)!}{(|\mathcal{Y}| - 2^p)!}$$

ways of selecting a codebook and an encoder. As an example, for the rate 4 : 5 block code discussed above there are $17! \approx 3.5568 \times 10^{14}$ ways to select a codebook and an encoder! Thus, a brute-force search is out of the question for even relatively low rate block codes.

Currently such a task is performed in a laborious, ad-hoc, and human-centric fashion, and becomes nearly impossible for very high-rate codes. For examples of various human-generated block encoders/decoders, see, Eggenberger and Patel [6], Marcus, Patel, and Siegel [7], and Galbraith [8]. Constructing low-complexity encoder/decoders for very high rate codes is of immense economical value—as these codes may be implemented in mass-market magnetic recording systems. Moreover, low hardware complexity leads to reduced power consumption.

As our main contribution, we propose efficient heuristic computer algorithms to select a codebook and to construct low-complexity encoder/decoder. We now summarize this paper and outline its organization.

- In Section 2, we introduce the notion of a *simple* set which contains 2^s words such that s coordinates of the set are *free* (or don’t-cares) and the remaining coordinates of the set are either *fixed* or are *dependent* on one of the free coordinates.

In this section, we introduce the *Constraint-Tree* algorithm, a binary-tree based recursive algorithm, to decompose the set of q-bit codewords \mathcal{Y} into disjoint, simple sets.

- In Section 3, we define the *codebook* \mathcal{C} as the union of the largest simple subsets of \mathcal{Y} such that \mathcal{C} contains exactly 2^p codewords.

In this section, using Huffman codes or arithmetic codes, we partition the set of p -bit datawords into disjoint, simple sets such that there is a one-to-one correspondence between a simple set of codewords and a simple set of datawords.

Finally, we construct the block decoder in a piece-meal fashion by defining it as a mapping from a simple sets of datawords to the corresponding simple set of codewords. The constructed decoder has an intuitively pleasing description as a certain fixed-rate two-part code.

- In Section 4, we apply the ideas in Sections 2 and 3 to rate $8 : 9$ block codes for $(0, 4/4)$ and $(0, 3/6)$ PRML constraints. For these constraints, our block encoders/decoders are comparable in complexity to those published in Eggenberger and Patel [6], but are significantly simpler than lexicographical encoders/decoders.
- In Section 5, we include a discussion of our algorithms and their possible extensions.

2 The Constraint-Tree Algorithm

2.1 Simple Sets of Words

We introduce the main new concept of this paper: a simple set of words. Simple sets will be the basic building blocks with which we design codebooks and low-complexity encoders/decoders.

Let $\mathcal{Z} \subset \{0, 1\}^r$ denote a set of r -bit words. Let $Z = (z_0, z_1, \dots, z_{r-1})$ denote a word in \mathcal{Z} . We say that coordinate i , $0 \leq i \leq r-1$, is *fixed* or *constant* in \mathcal{Z} , if for all $Z \in \mathcal{Z}$ either $z_i = 0$ or $z_i = 1$. We say that coordinate i , $1 \leq i \leq r-1$, is *dependent* in \mathcal{Z} , if it is not fixed and there exists a coordinate $j < i$ such that for all $Z \in \mathcal{Z}$ either $z_i = z_j$ or $z_i = \bar{z}_j$. In other words, we say that coordinate i , $1 \leq i \leq r-1$, is dependent in \mathcal{Z} , if there exists a coordinate $j < i$ such that coordinate i is a 1-bit Boolean function of the coordinate j . We say that coordinate i , $0 \leq i \leq r-1$, is *potentially free* in \mathcal{Z} , if it is neither fixed nor dependent in \mathcal{Z} . We say that a set of r -bit words \mathcal{Z} containing exactly s potentially free coordinates is *simple* if it contains exactly 2^s words. Potentially free coordinates of a simple set are called *free*—these coordinates may assume any binary value independent of other free coordinates. Intuitively, a set of 2^s r -bit words is simple if it contains s free coordinates, and $(r-s)$ fixed or dependent coordinates. The reader can check that any single codeword (with $r \geq 1$) or a set of two codewords (with $r \geq 2$) always constitutes a simple set.

As an example, the following set of 7-bit words constitutes a simple set.

0	1	2	3	4	5	6
0	0	1	1	0	0	0
0	1	1	1	1	0	0
1	0	1	0	0	0	1
1	1	1	0	1	0	1

To be precise, coordinates 2 and 5 are fixed, the coordinates 3 and 6 are dependent on the coordinate 0, the coordinate 4 is dependent on the coordinate 1, and the coordinates 0 and 1 are free. Finally, there are 2 free coordinates and 4 words. Note that if we were to delete any one of the four codewords, the resulting set would not be simple. However, if we were to delete any two or three of the four codewords, then the resulting set would be simple.

A simple set of r -bit words \mathcal{Z} can be *compactly* represented in *symbolic notation* as a r -tuple of symbols taken from the alphabet: $0, 1, a_0, \dots, a_{r-1}, \bar{a}_0, \dots, \bar{a}_{r-1}$. The entries in the r -tuple are determined using the following rules:

- R.1:** If the coordinate i is fixed in \mathcal{Z} , and takes value 0 or 1 then i -th value in the r -tuple is set to 0 or 1, respectively.
- R.2:** If the coordinate i is free in \mathcal{Z} , then i -th value in the r -tuple is set to a_i .
- R.3:** If the coordinate i is dependent in \mathcal{Z} , then find the *leftmost* $j < i$ such that coordinate i is a 1-bit Boolean function of coordinate j . Now, set i -th value in the r -tuple to a_j if coordinates i and j are always equal or to \bar{a}_j otherwise.

By applying these rules, we may compactly write the set of four 7-bit words exhibited above as

$$(a_0, a_1, 1, \bar{a}_0, a_1, 0, a_0).$$

2.2 Finding Simple Sets of Words

In Figure 1, we present a binary-tree based recursive partitioning algorithm to decompose an arbitrary set of words into disjoint simple sets of words.

The algorithm is initially invoked with the following *inputs*:

- \mathcal{Z} , an arbitrary set of r -bit words;
- An index set $I_F \subset \{0, 1, \dots, r-1\}$ that contains the coordinates that are *known* to be either fixed or dependent. Since, initially, none of the coordinates are known to be fixed or dependent, we set $I_F = \phi$ —the empty set.
- An index set $I_P \subset \{0, 1, \dots, r-1\}$ that contains the coordinates that are either fixed, dependent, or potentially free. Initially, we set $I_P = \{0, 1, \dots, r-1\}$.

Constraint_Tree(\mathcal{Z}, I_F, I_P, R)

Step 1: For every $i \in I_P$ such that coordinate i is fixed in \mathcal{Z}

- delete i from I_P and move i to I_F ;
- if i -th coordinate is 0, then set R_i to 0, else to 1;

Step 2: For every $i \in I_P$ such that coordinate i is dependent in \mathcal{Z}

- find leftmost $j < i$ such that coordinate i is a function of coordinate j ;
- delete i from I_P and move i to I_F ;
- if i - and j -th coordinates are equal, then set R_i to a_j , else to \bar{a}_j ;

Step 3: If $|\mathcal{Z}| = 2^{|I_P|}$, then:

- for every i in I_P , set $R_i = a_i$;
- return $\{(\mathcal{Z}, R)\}$;

Step 4: For every $i \in I_P$, compute $W_i(\mathcal{Z})$ (or $\bar{W}_i(\mathcal{Z})$) as the number of words in \mathcal{Z} such that coordinate i takes value 1 (or 0);

Find the *most constrained* coordinate

$$\hat{j} = \arg \min_{j \in I_P} \{ \min\{W_j(\mathcal{Z}), \bar{W}_j(\mathcal{Z})\} \}$$

The ties in “arg min” are broken by selecting the smallest coordinate.

Step 5: Partition \mathcal{Z} into two disjoint sets of words and recurse

- set $\mathcal{Z}_0 = \{Z = (z_0, z_1, \dots, z_{q-1}) \in \mathcal{Z} \mid z_{\hat{j}} = 0\}$;
- set $\mathcal{Z}_1 = \{Z = (z_0, z_1, \dots, z_{q-1}) \in \mathcal{Z} \mid z_{\hat{j}} = 1\}$;
- $L_0 = \mathbf{Constraint_Tree}(\mathcal{Z}_0, I_F, I_P, R)$;
- $L_1 = \mathbf{Constraint_Tree}(\mathcal{Z}_1, I_F, I_P, R)$;

Step 6: return $L_0 \cup L_1$;

Figure 1: The Constraint-Tree algorithm, a binary tree-based recursive partitioning algorithm, for decomposing an arbitrary set of r -bit words into disjoint simple sets.

- An r -tuple R which may contain symbols taken from the alphabet: $0, 1, a_0, \dots, a_{r-1}, \bar{a}_0, \dots, \bar{a}_{r-1}$. Since, initially, no r -tuple based compact representation is known for \mathcal{Z} , R is uninitialized. For $0 \leq i \leq (r-1)$, the i -th value in R is denoted as R_i .

As *outputs*, the algorithm produces a list of simple subsets of \mathcal{Z} such that each simple set comes packaged with its r -tuple based symbolic representation.

Although the algorithm is fairly self-explanatory, we briefly explain various steps for completeness.

- Step 1 removes fixed coordinates in \mathcal{Z} from I_P , while Step 2 removes dependent coordinates in \mathcal{Z} from I_P .
- In Step 3, we know from Steps 1 and 2 that the coordinates in I_P are neither fixed nor dependent in \mathcal{Z} . Hence, by definition, the coordinates in I_P are potentially free in \mathcal{Z} . Thus, to check whether \mathcal{Z} is a simple set, it suffices to check that $|\mathcal{Z}| = 2^{|I_P|}$. If, indeed, \mathcal{Z} is a simple set, then the algorithm returns the simple set \mathcal{Z} along with its r -tuple based symbolic representation. In this case, the coordinates in I_P are truly free, and R satisfies the rules **R.1**, **R.2**, and **R.3**. Step 4, 5, and 6 are executed, only if \mathcal{Z} is not a simple set.
- In Step 4, we know that \mathcal{Z} is not simple. Hence, it must be that $|\mathcal{Z}| < 2^{|I_P|}$. We now want to split \mathcal{Z} into two subsets such that each subset may be a simple set. The question is: How should one partition a not-simple set? We compute the most constrained coordinate \hat{j} in I_P as the coordinate that has the most lopsided distribution of zeroes and ones. In Step 5, we partition \mathcal{Z} into two subsets \mathcal{Z}_0 and \mathcal{Z}_1 such that \hat{j} is 0 in the former and 1 in the latter set. This step makes coordinate \hat{j} fixed in \mathcal{Z}_0 and \mathcal{Z}_1 . Finally, we recursively apply the algorithm to both \mathcal{Z}_0 and \mathcal{Z}_1 .

Splitting \mathcal{Z} on the most constrained bit is the main heuristic idea that makes the algorithm work. Intuitively, we recursively fix the most constrained coordinates, till we find a simple set. We made this choice since, after all, we are trying to design codebooks and low-complexity encoders/decoders for *constrained systems*. Thus, if we think of fixed or dependent coordinates as “constrained coordinates” and the potentially free coordinates as “unconstrained coordinates”, then we are recursively determining which coordinates to constrain. Also, observe that, as the recursion progresses, \mathcal{Z}_0 and \mathcal{Z}_1 may be further split themselves; however, they may be split on different coordinates.

We may think about each simple subset as a leaf node in the binary tree, where the path from the root to the leaf captures the “constrained coordinates” in the leaf and the leaf defines the “unconstrained coordinates.” Thus, our heuristic systematically moves constrained coordinates to the top levels of the binary tree, while implicitly moving the unconstrained coordinates towards the leaves.

Observe that in Step 5 we are guaranteed that there are no fixed coordinates in I_P , hence we have that $|\mathcal{Z}| > |\mathcal{Z}_0| \geq 1$ and $|\mathcal{Z}| > |\mathcal{Z}_1| \geq 1$. Consequently, the recursion is guaranteed to terminate. In the worst case, we will end up with decomposition of \mathcal{Z} into individual words.

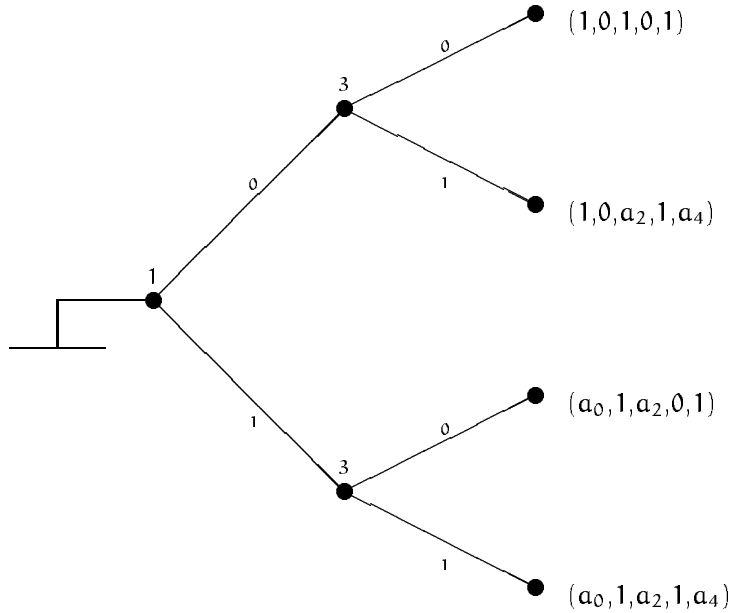


Figure 2: The binary tree produced by the Constraint-Tree algorithm for $(0, 2)$ RLL constraint. The number above each internal node shows the most constrained coordinate at that node; this coordinate is constrained (fixed) to 0 along the top branch and to 1 along the bottom branch. Each leaf node constitutes a simple set of codewords and its canonical symbolic representation in tuple-based notation is shown. By observing the “New” column of Table 1, the reader can verify that $(1, 0, 1, 0, 1)$ corresponds to the first codeword, $(1, 0, a_2, 1, a_4)$ corresponds to 2-5 codewords, $(a_0, 1, a_2, 0, 1)$ corresponds to 6-9 codewords, and, finally, $(a_0, 1, a_2, 1, a_4)$ corresponds to 10-17 codewords in the “New” column, respectively.

2.3 Example

We now briefly work through the rate $4 : 5$ code for the $(0, 2)$ RLL constraint. The algorithm starts with the 17 5-bit codewords in the “New” column of Table 1. The codewords are not assumed to be in any particular order. The algorithm discovers that this set is not simple, and splits this set along the most constrained coordinate 1 into two subsets, namely, the first 5 and the last 12 codewords in the “New” column of Table 1. Now, while recursing, the algorithm discovers that the coordinate 0 is fixed in the first 5 codewords, and that these codewords do not constitute a simple set either. It splits this set along the most constrained coordinate 3 into

the first codeword and 2-5 codewords in the “New” column of Table 1. The algorithm discovers that these sets are both simple, and hence this thread of recursion is stopped. Continuing along the thread of recursion associated with the last 12 codewords in the “New” column of Table 1, the algorithm discovers that this set is not simple. It splits this set along the most constrained coordinate 3 into 6-9 and 10-17 codewords in the “New” column of Table 1. Finally, the algorithm discovers that these sets are both simple. Thus, this thread of recursion is also stopped. The final binary tree produced by the algorithm is shown in Figure 2.

3 Codebook and Encoder/Decoder Design

Equipped with the notion of a simple set, and an algorithm for finding simple sets, we now return to the main problem of interest, that is, designing codebooks and low-complexity encoders/decoders for a block code.

3.1 Algorithm for Codebook Design

Given a list $\mathcal{Y} \subset \{0, 1\}^q$ of q -bit codewords satisfying some constraint, we apply the Constraint-Tree algorithm to \mathcal{Y} . This yields a list of simple sets of \mathcal{Y} . We write this list as follows:

$$\{(\mathcal{Y}_i, \mathbf{R}^{(\mathcal{Y}_i)})\}_{i=1}^{\ell}$$

where ℓ denotes the number of simple sets, \mathcal{Y}_i denotes the i -th simple set, and $\mathbf{R}^{(\mathcal{Y}_i)}$ denotes a compact q -tuple based symbolic representation of \mathcal{Y}_i using the alphabet $0, 1, a_0, \dots, a_{q-1}, \bar{a}_0, \dots, \bar{a}_{q-1}$. By construction, we are guaranteed that

$$\mathcal{Y} = \bigcup_{i=1}^{\ell} \mathcal{Y}_i \text{ and that } \mathcal{Y}_i \cap \mathcal{Y}_j = \emptyset \text{ whenever } i \neq j.$$

For $1 \leq i \leq \ell$, let $n_i = |\mathcal{Y}_i|$ denote the number of codewords in \mathcal{Y}_i , and let $s_i = \log_2(n_i)$ denote the number of free coordinates in \mathcal{Y}_i . We assume that the simple sets have been sorted in the descending order of their cardinality, that is, $n_1 \geq n_2 \geq \dots \geq n_{\ell}$. Since $\sum_{i=1}^{\ell} n_i \geq 2^p$, there is a unique $k \geq 1$ such that

$$\sum_{i=1}^k n_i \geq 2^p > \sum_{i=1}^{k-1} n_i.$$

If $k = 1$, then $n_1 \geq 2^p$ and we define the *codebook* \mathcal{C} to be any simple subset of \mathcal{Y}_1 defined by fixing exactly $s_1 - p$ of the free coordinates of \mathcal{Y}_1 (note that the codebook has cardinality 2^p). Otherwise, we define the *codebook* \mathcal{C} to be the union of the largest k simple sets

$$\mathcal{C} = \bigcup_{i=1}^k \mathcal{Y}_i. \tag{1}$$

We claim that the codebook again has cardinality 2^p , that is,

$$\sum_{i=1}^k n_i = 2^p. \quad (2)$$

To see this, we argue as follows. Let

$$h = 2^p - \sum_{i=1}^{k-1} n_i. \quad (3)$$

Clearly, $h \leq n_k$. Now, since $2^p \geq n_1 \geq \dots \geq n_{k-1}$ and all of the n_i are powers of 2, it follows that n_{k-1} divides the right hand side of (3). Thus, n_{k-1} divides h . In particular, $h \geq n_{k-1} \geq n_k$. Since also $h \leq n_k$, it follows that $h = n_k$, and so we have (2).

As an example, for the rate 4 : 5 code for the (0, 2) RLL constraint, the 16 word codebook can be constructed as a union of the three simple sets $\mathcal{Y}_1 \equiv (a_0, 1, a_2, 1, a_4)$, $\mathcal{Y}_2 \equiv (a_0, 1, a_2, 0, 1)$, and $\mathcal{Y}_3 \equiv (1, 0, a_2, 1, a_4)$, and by discarding the smallest simple set $\mathcal{Y}_4 \equiv (1, 0, 1, 0, 1)$.

3.2 Simple Sets of Datawords

We know from (1) that the codebook \mathcal{C} can be written as a disjoint union of simple subsets of \mathcal{Y} . We first partition the set of p -bit datawords into k disjoint, simple sets $\mathcal{X}_1, \mathcal{X}_2, \dots, \mathcal{X}_k$ such that

$$|\mathcal{X}_i| = |\mathcal{Y}_i|, 1 \leq i \leq k.$$

In other words, we would like to have exactly one simple set of datawords for every simple set of codewords. We now describe our partitioning scheme.

Define

$$l_i = p - s_i, 1 \leq i \leq k. \quad (4)$$

We now seek a prefix-free code (or an instantaneous code) of lengths l_1, l_2, \dots, l_k bits for each simple set $\mathcal{Y}_1, \mathcal{Y}_2, \dots, \mathcal{Y}_k$, respectively. To see that such a code exists, define a distribution of masses over the simple sets $\mathcal{Y}_1, \mathcal{Y}_2, \dots, \mathcal{Y}_k$ as

$$P(\mathcal{Y}_i) = \frac{n_i}{2^p} = 2^{s_i - p} = 2^{l_i}.$$

By construction, $P(\mathcal{Y}_i) > 0$ for every $1 \leq i \leq k$, and $\sum_{i=1}^k P(\mathcal{Y}_i) = 1$. Thus, P is a legitimate distribution. Furthermore, P is a dyadic distribution, that is, the mass assigned to each simple set is a power of 2. For such a distribution, a prefix-free code exists that satisfies (4), see, Cover and Thomas [9, Theorem 5.11.2]. Furthermore, such a code can be found using the Huffman algorithm [10, 9] or using arithmetic coding [11]. For $1 \leq i \leq k$, let

$$L(\mathcal{Y}_i) = (L_0(\mathcal{Y}_i), L_1(\mathcal{Y}_i), \dots, L_{l_i-1}(\mathcal{Y}_i)) \quad (5)$$

denote a l_i -tuple of bits that represents the prefix-free code for \mathcal{Y}_i .

Now, for $1 \leq i \leq k$, we define a simple set \mathcal{X}_i corresponding to \mathcal{Y}_i in a compact symbolic fashion as a p -tuple

$$\mathcal{X}_i \equiv R^{(X,i)} = (L_0(\mathcal{Y}_i), L_1(\mathcal{Y}_i), \dots, L_{l_i-1}(\mathcal{Y}_i), a_{l_i}, a_{l_i+1}, \dots, a_{p-1}). \quad (6)$$

In words, the simple set \mathcal{X}_i consists of l_i fixed coordinates and s_i free coordinates, and no dependent coordinates. Note that $R^{(X,i)}$ satisfies the rules **R.1**, **R.2**, and **R.3** of a symbolic representation. By construction, for $1 \leq i \leq k$, we have that

$$|\mathcal{X}_i| = 2^{s_i} = n_i = |\mathcal{Y}_i|$$

as desired. Furthermore, since the code $\mathcal{Y}_i \rightarrow L(\mathcal{Y}_i)$ is prefix-free, we have that

$$\mathcal{X}_i \cap \mathcal{X}_j = \emptyset \text{ whenever } i \neq j.$$

Now, it follows from (2) that

$$\mathcal{X} = \bigcup_{i=1}^k \mathcal{X}_i.$$

In other words, $\{\mathcal{X}_i\}_{i=1}^k$ represents a disjoint partitioning of the set of datawords \mathcal{X} .

As an example, for the rate 4 : 5 code for the (0, 2) RLL constraint, we can write the dyadic distribution P as

$$P(\mathcal{Y}_1) = \frac{1}{2}, P(\mathcal{Y}_2) = \frac{1}{4}, P(\mathcal{Y}_3) = \frac{1}{4},$$

and a prefix-free code for this distribution as

$$L(\mathcal{Y}_1) = (1), L(\mathcal{Y}_2) = (0, 1), L(\mathcal{Y}_3) = (0, 0). \quad (7)$$

Now, using (5), (6), and (7), the three simple sets of datawords can be written as

$$\mathcal{X}_1 \equiv (1, a_1, a_2, a_3), \mathcal{X}_2 \equiv (0, 1, a_2, a_3), \mathcal{X}_3 \equiv (0, 0, a_2, a_3).$$

3.3 Algorithm for Constructing the Decoder

In our framework, it is more natural to first construct the decoder $\mathcal{D} : \mathcal{C} \rightarrow \mathcal{X}$. We define the decoder in a piece-meal fashion by defining it over each simple set of codewords. Let

$$Y = (y_0, y_1, \dots, y_{q-1})$$

be a given q -bit codeword in \mathcal{Y} , we now determine a p -bit dataword

$$\mathcal{D}(Y) = X = (x_0, x_1, \dots, x_{p-1}) \in \mathcal{X}$$

as follows.

- Determine the index $1 \leq i \leq k$ such that $Y \in \mathcal{Y}_i$. We can now determine the first l_i coordinates of X as

$$\begin{aligned}
x_0 &= L_0(\mathcal{Y}_i) \\
x_1 &= L_1(\mathcal{Y}_i) \\
&\vdots \\
x_{l_i-1} &= L_{l_i-1}(\mathcal{Y}_i),
\end{aligned} \tag{8}$$

where $(L_0(\mathcal{Y}_i), L_1(\mathcal{Y}_i), \dots, L_{l_i-1}(\mathcal{Y}_i))$ are as in (5).

- Let $f(i, 1) \leq f(i, 2) \leq \dots \leq f(i, s_i)$ denote the free coordinates in \mathcal{Y}_i . Observe that we still need to determine the last $s_i = p - l_i$ coordinates of X . Let $g(i, l_i), g(i, l_i+1), \dots, g(i, p-1)$ denote a permutation of the s_i numbers: $l_i, l_i + 1, \dots, p - 1$. Note that one may simply use $g(i, j) = j$, for $l_i \leq j \leq p - 1$, to obtain a legal decoder. However, by judiciously selecting the permutation g , we can generally obtain a lower-complexity decoder. We will specify the specific low-complexity permutation used in this paper in Section 3.4. We now determine the last s_i coordinates of X by directly mapping the s_i free coordinates of Y to them as

$$\begin{aligned}
x_{g(i, l_i)} &= y_{f(i, 1)} \\
x_{g(i, l_i+1)} &= y_{f(i, 2)} \\
&\vdots \\
x_{g(i, p-1)} &= y_{f(i, s_i)}.
\end{aligned} \tag{9}$$

Intuitively, we are doing a *fixed-rate two-part coding*. The first l_i coordinates of X code the index i of the simple set \mathcal{Y}_i . The last s_i coordinates of X code the actual codeword Y in \mathcal{Y}_i .

It is easy to show that the decoder \mathcal{D} constructed above is, in fact, one-to-one and onto. Hence, it has a well defined inverse, which we use as the desired encoder $\mathcal{E} : \mathcal{X} \rightarrow \mathcal{C}$. The inverse is easy to compute: reverse the “codeword-to-dataword” assignments to “dataword-to-codeword” assignments.

It follows from our construction that if the codeword Y is in the simple set \mathcal{Y}_i , for some $1 \leq i \leq k$, then the corresponding dataword $\mathcal{D}(Y) = X$ must be in \mathcal{X}_i . Thus, one can think of the decoder as mapping

$$\mathcal{D} : \mathcal{Y}_i \rightarrow \mathcal{X}_i, 1 \leq i \leq k.$$

Let $R^{(Y, i)}$ and $R^{(X, i)}$ denote the q - and p -tuple based symbolic representations of \mathcal{Y}_i and \mathcal{X}_i , respectively. We define a p -tuple $R^{(Y \rightarrow X, i)}$ as follows:

$$R_j^{(Y \rightarrow X, i)} = \begin{cases} L_j(\mathcal{Y}_i) & \text{if } 0 \leq j \leq l_i - 1, \\ a_{f(i, g(i, j) - l_i + 1)} & \text{if } l_i \leq j \leq p - 1. \end{cases} \tag{10}$$

Observe that $R^{(Y \rightarrow X, i)}$ and $R^{(X, i)}$ agree in all their fixed coordinates. The p -tuple $R^{(Y \rightarrow X, i)}$ uses a larger alphabet, namely, $0, 1, a_0, \dots, a_{q-1}, \bar{a}_0, \dots, \bar{a}_{q-1}$, than $R^{(X, i)}$ which uses the alphabet $0, 1, a_0, \dots, a_{p-1}, \bar{a}_0, \dots, \bar{a}_{p-1}$. Hence, $R^{(Y \rightarrow X, i)}$ does not satisfy the rules **R.1**, **R.2**, and **R.3** for a valid symbolic tuple-based representation. It is intended to symbolically capture equations (8) and (9). We now symbolically write the decoder $\mathcal{D} : \mathcal{Y}_i \rightarrow \mathcal{X}_i$ in a compact fashion as

$$R^{(Y, i)} \rightarrow R^{(Y \rightarrow X, i)} \quad (11)$$

We will extensively use this notation in the following section and in Tables 3, 4, and 6.

As an example, for the rate 4 : 5 code for the (0, 2) RLL constraint, in the notation of (10) and (11), we can write the three simple sets of datawords as follows.

$$\begin{array}{l} \mathcal{X}_1 \equiv (1 \quad a_{f(1, g(1, 1))} \quad a_{f(1, g(1, 2))} \quad a_{f(1, g(1, 3))}) \\ \mathcal{X}_2 \equiv (0 \quad 1 \quad a_{f(2, g(2, 2)-1)} \quad a_{f(2, g(2, 3)-1)}) \\ \mathcal{X}_3 \equiv (0 \quad 0 \quad a_{f(3, g(3, 2)-1)} \quad a_{f(3, g(3, 3)-1)}) \end{array}$$

The permutation g (in general and for this example) will be determined in the next section.

3.4 Algorithm for Permuting the Free Coordinates

To complete the construction of the decoder and the corresponding encoder, we still need to specify the permutation g of the free coordinates of each simple set of datawords in (9). Notice that, for every $1 \leq i \leq k$, every permutation of the s_i numbers $l_i, l_i + 1, \dots, p - 1$ is acceptable; however, we would like to exploit this freedom in the choice of the permutation to *share logic across the simple sets*, and, hence, obtain lower-complexity decoder and encoder. We now present a simple heuristic algorithm that typically leads to a 20–30% reduction in the complexity of the resulting encoders/decoders.

Arrange the k simple sets of codewords $\{\mathcal{Y}_i\}_{i=1}^k$ as a $k \times q$ matrix \mathbf{C} whose columns denote the q coordinates $0, 1, \dots, q - 1$ and whose rows denote the symbolic representations $R^{(Y, i)}$ of the simple sets. Similarly, arrange the k simple sets of datawords $\{\mathcal{X}_i\}_{i=1}^k$ as a $k \times p$ matrix \mathbf{D} whose columns denote the p coordinates $0, 1, \dots, p - 1$ and whose rows denote the symbolic representations $R^{(Y \rightarrow X, i)}$ of the simple sets of datawords as a function of the symbolic representations $R^{(Y, i)}$ of the corresponding simple sets of codewords. The reader may peek ahead to Tables 3, 4, or 6 to intuitively understand such a matrix based symbolic representation. In such a representation, for every $1 \leq i \leq k$, the problem is to determine the last l_i coordinates of $R^{(Y \rightarrow X, i)}$ in (10).

Step 1 Initially, all free coordinates of all rows of \mathbf{D} are marked as *unfilled*, and all free coordinates of all rows of \mathbf{C} are marked as *unused*. All fixed coordinates of all rows of \mathbf{D} are marked *filled* and all fixed coordinates of all rows of \mathbf{C} are marked as *used*.

	Codewords = \mathbf{C}					Datawords = \mathbf{D}					
Step 1	($\boxed{a_0}$	1	$\boxed{a_2}$	1	$\boxed{a_4}$)	\longrightarrow	(1	\square	\square	\square)	
all free coordinates	($\boxed{a_0}$	1	$\boxed{a_2}$	0	1)	\longrightarrow	(0	1	\square	\square)	all free coordinates
unused	(1	0	$\boxed{a_2}$	1	$\boxed{a_4}$)	\longrightarrow	(0	0	\square	\square)	unfilled
Steps 3.a-3.e	($\boxed{a_0}$	1	a_2	1	$\boxed{a_4}$)	\longrightarrow	(1	\square	\square	a_2)	
use a_2	($\boxed{a_0}$	1	a_2	0	1)	\longrightarrow	(0	1	\square	a_2)	fill a_2
	(1	0	a_2	1	$\boxed{a_4}$)	\longrightarrow	(0	0	\square	a_2)	
Steps 3.a-3.e	($\boxed{a_0}$	1	a_2	1	a_4)	\longrightarrow	(1	\square	a_4	a_2)	
use a_4	($\boxed{a_0}$	1	a_2	0	1)	\longrightarrow	(0	1	\square	a_2)	fill a_4
	(1	0	a_2	1	a_4)	\longrightarrow	(0	0	a_4	a_2)	
Step 3.f	(a_0	1	a_2	1	a_4)	\longrightarrow	(1	a_0	a_4	a_2)	
use a_0	(a_0	1	a_2	0	1)	\longrightarrow	(0	1	a_0	a_2)	fill a_0
	(1	0	a_2	1	a_4)	\longrightarrow	(0	0	a_4	a_2)	

Table 3: For the $(0, 2)$ RLL constraint, we display the sequence of steps executed by the algorithm for permuting the free coordinates. Unused free coordinates of the matrix \mathbf{C} are shown inside a box, and unfilled free coordinates of the matrix \mathbf{D} are denoted by a \square .

Step 2 If, for some row of \mathbf{D} , say i , it happens that there is exactly one unfilled free coordinate, say, j , then it must happen that the corresponding row of \mathbf{C} has exactly one unused free coordinate, say, j' . Now, assign j' to j . Precisely, set $g(i, j) = j'$. In the i -th row of \mathbf{D} , mark j as filled. In the i -th row of \mathbf{C} , mark j' as used.

Step 3 Iterate Steps 3.a through 3.f, while there is at least one unfilled free coordinate in some row of \mathbf{D} .

Step 3.a Find the column, say, j^* , of \mathbf{D} that has the maximum number of unfilled free coordinates. In case of ties, select the largest coordinate.

Step 3.b For $0 \leq j \leq q-1$, compute d_j : the number of rows of \mathbf{D} such that the coordinate j^* has been filled with the free coordinate j , that is, $f(i, g(i, j^*)) - l_i + 1 = j$.

Step 3.c For $0 \leq j \leq q-1$, compute e_j : the number of rows of \mathbf{C} for which the coordinate j is unused and the corresponding row of \mathbf{D} is unfilled in coordinate j^* .

Step 3.d Compute

$$j^\dagger = \arg \max_{0 \leq j \leq q-1} \{d_j + e_j\},$$

where ties in “arg max” are broken by selecting the largest coordinate.

Step 3.e For every row of \mathbf{D} , say, i , such that j^* is unfilled and the coordinate j^\dagger is unused in the corresponding row of \mathbf{C} , then assign j^\dagger to j^* in that row. Precisely, set $g(i, j^*) = i' + l_i - 1$, where i' is such that $f(i, i') = j^\dagger$. In the i -th row of \mathbf{D} , mark j^* as filled. In the i -th row of \mathbf{C} , mark j^\dagger as used.

Step 3.f Repeat Step 2.

As an example, for the rate 4 : 5 code for the (0, 2) RLL constraint, we show the sequence of steps executed by the algorithm for permuting the free coordinates in Table 3. The reader can check that the last three rows of Table 3 corresponds to the mapping from the “New” column of Table 1 to the “Datawords” column in Table 1.

4 Examples

In this section, we illustrate the methods developed in Sections 2 and 3 on two constrained systems that are useful in magnetic recording systems employing a detection scheme known as partial response maximum likelihood (PRML). For more discussion on PRML channels, we refer the interested reader to [2, 3] and to references therein. We consider the (0, \mathbf{G}/\mathbf{I}) constraints which have been used in PRML channels, where the parameter 0 symbolizes that no minimum run lengths of zeroes is demanded in the channel output code bit sequence, the parameter \mathbf{G} represents the “global” constraint that maximum run lengths of zeroes in the channel output code bit sequence is no more than \mathbf{G} , and the parameter \mathbf{I} represents the “interleaved” constraint that maximum run lengths of zeroes is no more than \mathbf{I} in either the even index or the odd index subsequences of the channel output code bit sequence. In particular, in this paper, we consider (0, 4/4) and (0, 3/6) PRML constraints.

4.1 (0, \mathbf{G}/\mathbf{I}) = (0, 4/4)

Capacity of this constraint is 0.961366 which is greater than $8/9 = 0.8888$, and hence it is natural to seek a rate 8 : 9 code for this constraint. In fact, a block code with rate 8 : 9 is known, see, Eggenberger and Patel [6]—referred to as EP hereafter. These authors found a set of 279 9-bit codewords by starting from all 9-bit words and eliminating all words that have more than four consecutive symbols “0” anywhere in the word, more than two consecutive symbols “0” at the beginning or the end of the word, or more than two consecutive symbols “0” at the beginning or the end in even or odd sub-words. These 279 codewords can be freely concatenated without violating the constraint, and hence constitute a block code.

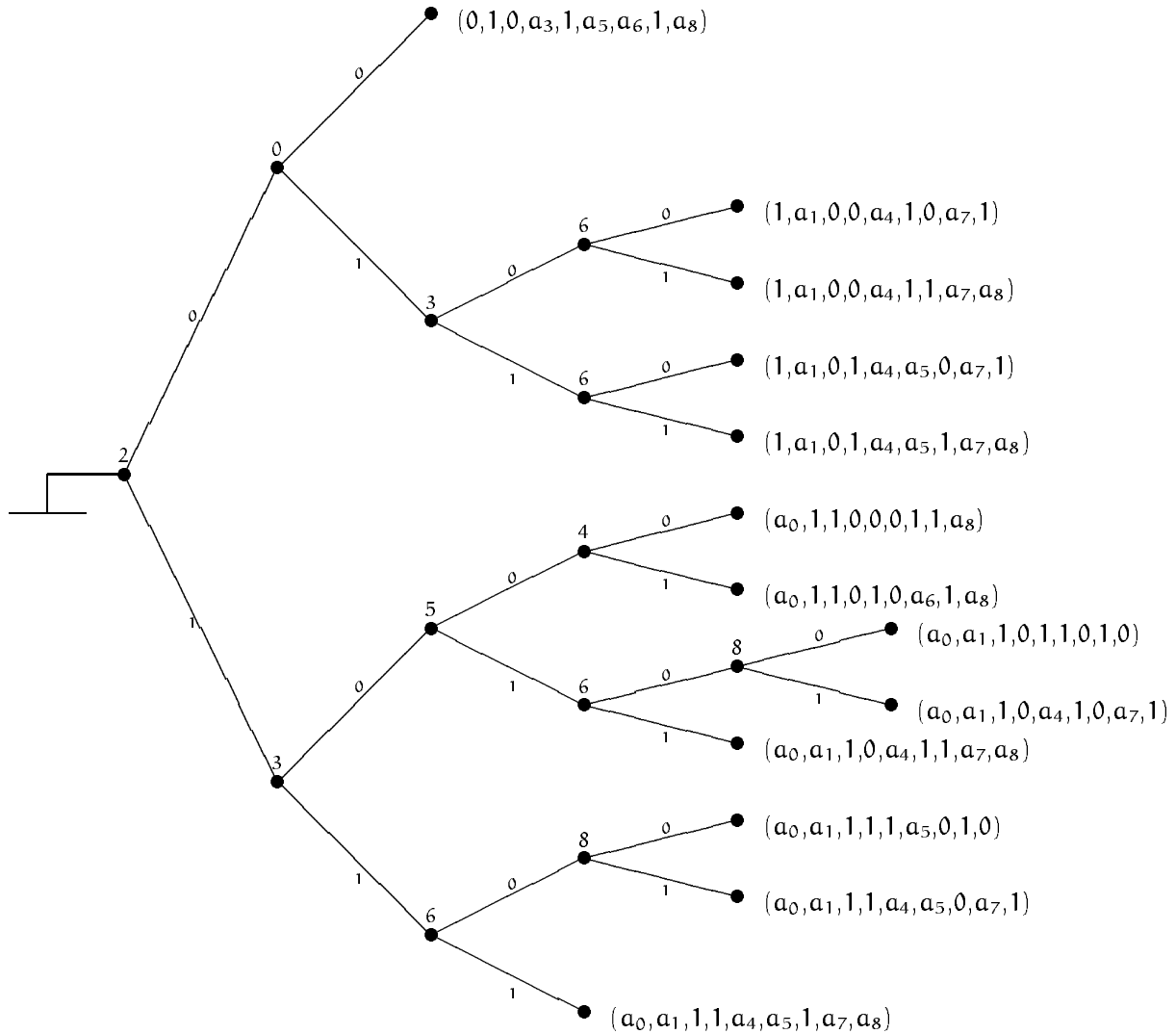


Figure 3: The binary tree produced by the Constraint-Tree algorithm for the $(0,4/4)$ PRML constraint. The number above each internal node shows the most constrained coordinate at that node; this coordinate is constrained (fixed) to 0 along the top branch and to 1 along the bottom branch. Each leaf node constitutes a simple set of codewords and its canonical symbolic representation in tuple-based notation is shown. To avoid clutter, the simple sets of codewords which have been deleted are not shown.

Codewords									Datawords							
(a ₀ a ₁ 1 1 a ₄ a ₅ 1 a ₇ a ₈)	→	(1 1 a ₀ a ₅ a ₈ a ₄ a ₇ a ₁)														
(a ₀ a ₁ 1 1 a ₄ a ₅ 0 a ₇ 1)	→	(1 0 1 a ₅ a ₀ a ₄ a ₇ a ₁)														
(a ₀ a ₁ 1 0 a ₄ 1 1 a ₇ a ₈)	→	(1 0 0 a ₀ a ₈ a ₄ a ₇ a ₁)														
(1 a ₁ 0 1 a ₄ a ₅ 1 a ₇ a ₈)	→	(0 1 1 a ₅ a ₈ a ₄ a ₇ a ₁)														
(a ₀ a ₁ 1 0 a ₄ 1 0 a ₇ 1)	→	(0 1 0 1 a ₀ a ₄ a ₇ a ₁)														
(1 a ₁ 0 1 a ₄ a ₅ 0 a ₇ 1)	→	(0 1 0 0 a ₅ a ₄ a ₇ a ₁)														
(1 a ₁ 0 0 a ₄ 1 1 a ₇ a ₈)	→	(0 0 1 1 a ₈ a ₄ a ₇ a ₁)														
(0 1 0 a ₃ 1 a ₅ a ₆ 1 a ₈)	→	(0 0 1 0 a ₈ a ₃ a ₅ a ₆)														
(a ₀ a ₁ 1 1 1 a ₅ 0 1 0)	→	(0 0 0 1 1 a ₅ a ₀ a ₁)														
(a ₀ 1 1 0 1 0 a ₆ 1 a ₈)	→	(0 0 0 1 0 a ₆ a ₀ a ₈)														
(1 a ₁ 0 0 a ₄ 1 0 a ₇ 1)	→	(0 0 0 0 1 a ₄ a ₇ a ₁)														
(a ₀ a ₁ 1 0 1 1 0 1 0)	→	(0 0 0 0 0 1 a ₀ a ₁)														
(a ₀ 1 1 0 0 0 1 1 a ₈)	→	(0 0 0 0 0 0 a ₀ a ₈)														
(1 a ₁ 0 1 1 a ₅ 0 1 0)		excess														
(0 1 0 1 1 a ₅ 1 0 a ₈)		excess														
(1 1 0 0 1 0 a ₆ 1 a ₈)		excess														
(0 1 0 1 1 a ₅ 0 0 1)		excess														
(1 1 0 0 0 0 1 1 a ₈)		excess														
(1 a ₁ 0 0 1 1 0 1 0)		excess														
(0 1 0 0 1 1 1 0 a ₈)		excess														
(a ₀ 1 1 0 0 0 0 1 1)		excess														
(0 1 0 0 1 1 0 0 1)		excess														

Table 4: Decoder for the (0, 4/4) PRML constraint

		Eggenberger-Patel	New	Lexicographical
ENCODER	AREA	201	246	1345
	GATES	39	47	263
DECODER	AREA	161	271	1518
	GATES	33	58	310

Table 5: Complexities of various encoders/decoders for the (0, 4/4) PRML constraint

We applied the Constraint-Tree algorithm to the above set of 279 9-bit codewords [6, Figure 4]. The resulting binary tree is shown in Figure 3. As a result of the algorithm, we found 22 simple sets that are displayed in the left column of Table 4 using the compact tuple-based symbolic notation. Using (1), we define the codebook \mathcal{C} as the union of the largest 13 of these simple sets. Finally, we applied the ideas in Section 3 to design a mapping from these 13 9-bit simple sets of codewords to their corresponding 8-bit simple sets of datawords. This mapping, namely, the decoder, is displayed in Table 4 using notation in (10) and (11). While reading Table 4, one should think of the datawords as a function of the corresponding codewords.

In Table 5, we compare the hardware complexities of the encoders/decoders in EP, in Table 4, and in a lexicographical assignment. The last assignment is obtained by lexicographically ordering the 9-bit codewords and deleting the last 23 9-bit codewords, and then assigning to each 9-bit codeword a 8-bit dataword corresponding to the codeword’s rank in the lexicographically ordered list of the remaining 256 codewords. The “AREA” and “GATES” for the encoder/decoder in EP were obtained by inputting their logic equations in the Berkeley SIS logic synthesis program (and by using SA-12E cell library for CMOS 6SF technology), while the numbers for our encoder/decoder and the lexicographical encoder/decoder were obtained by inputting their respective “codewords-to-datawords” (decoder) and “datawords-to-codewords” (encoder) assignments in SIS. It can be seen from Table 5 that our encoder/decoder are comparable to those in EP, and are far superior to their lexicographical counterparts which serves as a baseline.

4.2 $(0, \mathbf{G/I}) = (0, 3/6)$

Capacity of this constraint is 0.944539 which is greater than $8/9 = 0.8888$, and hence it is natural to seek a rate 8 : 9 code for this constraint. In fact, two block codes with rate 8 : 9 are known, see, Eggenberger and Patel [6]. These authors found a set of 272 9-bit codewords by starting from all 9-bit words and eliminating all words that have more than three consecutive symbols “0” anywhere in the word, more than two consecutive symbols “0” at the beginning of the word, more than one symbol “0” at the end of the word, and more than three consecutive symbols “0” anywhere in the even or odd sub-words. (Another block code with 272 9-bit words can be found similarly, the only difference being that the words have no more than one consecutive symbol “0” at the beginning and no more than two consecutive symbols “0” at the end.) In this paper, we only use the former block code, the complexity of the latter code is nearly the same. These 272 codewords can be freely concatenated without violating the constraint, and hence constitute a block code.

We applied the algorithm in Figure 1 to the above set of 272 9-bit codewords [6, Figure 4]. As a result, we found 24 simple sets that are displayed in the left column of Table 6 using the compact tuple-based symbolic notation. Using (1), we define the codebook \mathcal{C} as the union

Codewords									Datawords							
(a ₀ a ₁ 1 1 a ₄ a ₅ a ₆ 1 a ₈)	→	(1 1 a ₄ a ₁ a ₀ a ₅ a ₈ a ₆)														
(a ₀ a ₁ 1 0 1 a ₅ a ₆ 1 a ₈)	→	(1 0 1 a ₁ a ₀ a ₅ a ₈ a ₆)														
(1 a ₁ 0 a ₃ 1 a ₅ a ₆ 1 a ₈)	→	(1 0 0 a ₁ a ₃ a ₅ a ₈ a ₆)														
(a ₀ 1 a ₂ a ₃ 1 a ₅ a ₆ 0 1)	→	(0 1 1 a ₂ a ₀ a ₅ a ₃ a ₆)														
(a ₀ a ₁ 1 0 0 1 a ₆ 1 a ₈)	→	(0 1 0 1 a ₀ a ₁ a ₈ a ₆)														
(0 1 0 a ₃ 1 a ₅ a ₆ 1 a ₈)	→	(0 1 0 0 a ₃ a ₅ a ₈ a ₆)														
(a ₀ 0 1 a ₃ a ₄ 1 a ₆ 0 1)	→	(0 0 1 1 a ₀ a ₄ a ₃ a ₆)														
(a ₀ a ₁ 1 0 0 0 1 1 a ₈)	→	(0 0 1 0 1 a ₀ a ₈ a ₁)														
(1 a ₁ 0 1 0 a ₅ 1 1 a ₈)	→	(0 0 1 0 0 a ₅ a ₈ a ₁)														
(1 1 a ₂ a ₃ 0 1 a ₆ 0 1)	→	(0 0 0 1 1 a ₂ a ₃ a ₆)														
(1 a ₁ 0 1 0 a ₅ 0 1 1)	→	(0 0 0 1 0 1 a ₅ a ₁)														
(0 1 0 1 0 a ₅ 1 1 a ₈)	→	(0 0 0 1 0 0 a ₈ a ₅)														
(a ₀ 1 0 0 0 1 1 1 a ₈)	→	(0 0 0 0 1 1 a ₈ a ₀)														
(0 1 1 a ₃ 0 1 a ₆ 0 1)	→	(0 0 0 0 1 0 a ₃ a ₆)														
(a ₀ 1 1 a ₃ 0 0 1 0 1)	→	(0 0 0 0 0 1 a ₃ a ₀)														
(1 0 0 1 1 a ₅ a ₆ 0 1)	→	(0 0 0 0 0 0 a ₅ a ₆)														
(a ₀ 0 1 1 1 0 a ₆ 0 1)		excess														
(a ₀ 0 1 1 0 0 1 0 1)		excess														
(a ₀ 1 0 1 0 0 1 0 1)		excess														
(1 0 0 0 1 1 a ₆ 0 1)		excess														
(0 1 0 a ₃ 0 1 1 0 1)		excess														
(1 0 0 1 0 1 a ₆ 0 1)		excess														
(1 0 0 1 0 0 1 0 1)		excess														
(1 1 0 0 0 1 0 1 1)		excess														

Table 6: Decoder for the (0, 3/6) PRML constraint

		Eggenberger- Patel	New	Lexicographical
ENCODER	AREA	292	293	1408
	GATES	64	59	281
DECODER	AREA	258	376	1105
	GATES	52	76	219

Table 7: Complexities of various encoders/decoders for the (0, 3/6) PRML constraint

of the largest 16 of these simple sets. Finally, we applied the ideas in Section 3 to design a mapping from these 16 9-bit simple sets of codewords to their corresponding 8-bit simple sets of datawords. This mapping, namely, the decoder, is displayed in Table 6 using notation in (10) and (11).

In Table 7, we compare the hardware complexities of the encoders/decoders in EP, in Table 6, and in a lexicographical assignment. It can be seen from Table 7 that our encoder/decoder are comparable to those in EP, and are far superior to their lexicographical counterparts which serves as a baseline.

5 Discussion

Remark 5.1 (Logic synthesis) In this paper, we have focussed on algorithms for generating “dataword-to-codeword” and corresponding “codeword-to-dataword” assignments. Having generated these assignments, we did not actually generate the corresponding logic equations; rather, we employed the Berkeley SIS logic synthesis program [4]. Now, we outline a conceptual method for generating such logic equations for the decoder; such a method is useful for analysis and discussion. Let $Y = (y_0, y_1, \dots, y_{q-1})$ denote a q -bit word, and let $X = (x_0, x_1, \dots, x_{p-1})$ denote a p -bit dataword.

D.1 For $1 \leq i \leq k$, let $M_i(Y)$ denote the indicator function for the simple set of codewords \mathcal{Y}_i such that $M_i(Y) = 1$ if $Y \in \mathcal{Y}_i$ and is zero otherwise. Observe that the binary tree generated by the Constraint-Tree algorithm yields a natural decision tree for computing these indicator functions.

D.2 We can now compute $X = \mathcal{D}(Y)$ as follows. For $0 \leq j \leq p - 1$, write

$$x_j = \left(M_1 \wedge R_j^{(Y \rightarrow X, 1)} \right) \vee \left(M_2 \wedge R_j^{(Y \rightarrow X, 2)} \right) \vee \dots \vee \left(M_k \wedge R_j^{(Y \rightarrow X, k)} \right) \quad (12)$$

where \vee and \wedge represent binary-input OR and AND gates, respectively.

Similarly, we can generate logic equations for the encoder as follows.

E.1 For $1 \leq i \leq k$, let $N_i(X)$ denote the indicator function for the simple set of datawords \mathcal{X}_i . Observe that the binary-tree associated with the prefix-free code in (5) yields a natural decision tree for computing these indicator functions.

E.2 We can now compute $Y = \mathcal{E}(X)$ as follows. For $0 \leq j \leq q - 1$, write

$$y_j = \left(N_1 \wedge R_j^{(X \rightarrow Y, 1)} \right) \vee \left(N_2 \wedge R_j^{(X \rightarrow Y, 2)} \right) \vee \dots \vee \left(N_k \wedge R_j^{(X \rightarrow Y, k)} \right) \quad (13)$$

where \vee and \wedge represent binary-input OR and AND gates, respectively. $R^{(X \rightarrow Y, i)}$ symbolically captures the encoder $\mathcal{E} : \mathcal{X}_i \rightarrow \mathcal{Y}_i$ in the same fashion that $R^{(Y \rightarrow X, i)}$ symbolically

captures the decoder $\mathcal{D} : \mathcal{Y}_i \rightarrow \mathcal{X}_i$ in (10) and (11). We leave the precise definition of $R^{(X \rightarrow Y, i)}$ to the reader as an exercise.

For $0 \leq j \leq p-1$, let α_j denote the number of *different literals* from the alphabet a_0, a_1, \dots, a_{q-1} that appear in the set

$$\left\{ R_j^{(Y \rightarrow X, i)}, 1 \leq i \leq k \right\}.$$

For example, in Table 6, $\alpha_0 = 0$, $\alpha_1 = 0$, $\alpha_2 = 1$, $\alpha_3 = 2$, $\alpha_4 = 2$, $\alpha_5 = 5$, $\alpha_6 = 3$, and $\alpha_7 = 4$. For $0 \leq j \leq p-1$, let β_j denote the number of simple sets (numbered $i = 1, 2, \dots, k$) such that $R_j^{(Y \rightarrow X, i)}$ takes value 0. For example, in Table 6, $\beta_0 = 13$, $\beta_1 = 12$, $\beta_2 = 10$, $\beta_3 = 7$, $\beta_4 = 5$, $\beta_5 = 3$, $\beta_6 = 0$, and $\beta_7 = 0$.

Proposition *If $k > 1$, the decoder can be constructed using a circuit that uses no more binary-input AND gates than*

$$\sum_{j=0}^{p-1} \alpha_j + \sum_{i=1}^k (\log_2(2^p/n_i) - 1), \quad (14)$$

and uses no more binary-input OR gates than

$$\sum_{j=0}^{p-1} (k - \beta_j - 1). \quad (15)$$

Proof: The first term in (14) is the number of AND gates required in (12), and the second term in (14) is the number of AND gates required to compute the indicator functions M_i , $1 \leq i \leq k$. Equation (15) is the number of OR gates required in (12). \square

It is also possible to compute similar upper bounds on the circuit complexity of the encoder; we omit the precise results for brevity. The following three remarks point out practical implications of the above result.

Remark 5.2 (less simple sets are better) The fewer the number of simple sets k , the fewer the number of binary-input AND gates in (14) and the fewer the number of binary-input OR gates in (15). Hence, as a rule of thumb, the smaller the number of simple sets, the less complex the resulting encoders/decoders. For example, the $(0, 4/4)$ PRML constraint has 13 simple sets, while the $(0, 3/6)$ PRML constraint has 16 simple sets. It can be seen from Tables 5 and 7 that our encoders/decoders for the $(0, 4/4)$ PRML constraint are simpler than our encoders/decoders for the $(0, 3/6)$ PRML constraint.

Remark 5.3 (zeroes are better than ones) If for some i and j , the term $R_j^{(Y \rightarrow X, i)}$ is zero, then the corresponding term can be deleted from (12). This saves a binary-input OR gate. Thus, we prefer the term $R_j^{(Y \rightarrow X, i)}$ to be zero and not one. In other words, by increasing the

number of zeroes, we increase $\sum_{j=0}^{p-1} \beta_j$, and hence decrease (15). Observe that in Tables 4 and 6 there are more zeroes than ones in the prefix-part of the Datawords matrix. This is by design. We now exhibit complexities of the encoders and decoders for the two PRML constraints where we have exchanged a zero by a one and vice versa in the prefix-part of the Datawords matrix in Tables 4 and 6.

		(0,4/4)	(0,3/6)
ENCODER	AREA	246	291
	GATES	51	61
DECODER	AREA	339	464
	GATES	70	94

By comparing the (0,4/4) column above to the “New” column in Table 5 and by comparing the (0,3/6) column above to the “New” column in Table 7, it follows that the complexity of the encoders remains essentially the same, but the decoders become 20-30% more complex when there are more ones than zeroes.

Remark 5.4 (coordinate permutation algorithm helps) The algorithm for permuting the free coordinates essentially attempts to keep the term $R_j^{(Y \rightarrow X, i)}$ in (12) and the term $R_j^{(X \rightarrow Y, i)}$ in (13) constant as a function of i . The net effect is that some of the AND gates in (12) and (13) can be eliminated by combining terms. In other words, for the decoder, we can decrease the term $\sum_{j=0}^{p-1} \alpha_j$ in (14). We now exhibit complexities of encoders/decoders for the two PRML constraints by using the trivial permutation $g(i, j) = j$ in (9).

		(0,4/4)	(0,3/6)
ENCODER	AREA	348	355
	GATES	69	73
DECODER	AREA	341	441
	GATES	71	93

By comparing the (0,4/4) column above to the “New” column in Table 5, it follows that—without the algorithm for permuting the coordinates—the encoder becomes 40% more complex and the decoder becomes 30% more complex. Similarly, by comparing the (0,3/6) column above to the “New” column in Table 7, it follows that—without the algorithm for permuting the coordinates—both the encoder and the decoder become 20% more complex.

Remark 5.5 (lexicographical assignments as trees) We now contrast the structure discovered by the Constraint-Tree algorithm to that discovered by lexicographical codes. Observe that if in Step 5 of Figure 1 we partition the set \mathcal{Z} on the smallest available coordinate (instead of the most constrained coordinate), then we obtain a lexicographical tree. Such a tree has a much larger number of simple sets than the one found by the Constraint-Tree algorithm, and,

hence, we expect the lexicographical assignments to be generally more complex—a fact that is borne out by Tables 2, 5, and 7.

Remark 5.6 (Merging two simple sets) Let \mathcal{Z}_1 and \mathcal{Z}_2 be r -bit simple sets. We say that \mathcal{Z}_1 and \mathcal{Z}_2 are *mergable*, if $|\mathcal{Z}_1| = |\mathcal{Z}_2|$, and the r -tuple based symbolic representations of \mathcal{Z}_1 and \mathcal{Z}_2 agree on all free coordinates, but disagree on at least one fixed or dependent coordinate. However, if the simple sets disagree on a dependent coordinate, then we require that the corresponding literals in \mathcal{Z}_1 and \mathcal{Z}_2 should be complements of each other. It is easy to see that if \mathcal{Z}_1 and \mathcal{Z}_2 are mergable, then $\mathcal{Z}_1 \cup \mathcal{Z}_2$ has exactly one more free coordinate. Treating $\mathcal{Z}_1 \cup \mathcal{Z}_2$ as one simple set will lead to strictly smaller number of simple sets k , and, hence, perhaps to simpler encoders/decoders (see Remark 5.2). As an example, consider two simple sets:

$$\begin{aligned} & (a_0, 0, 1, 1, 0, 0, 1, 0, 1) \\ & (a_0, 1, 0, 1, 0, 0, 1, 0, 1) \end{aligned}$$

that arise in Table 4. These two sets are mergable, the merged simple set can be written compactly as:

$$(a_0, a_1, \bar{a}_1, 1, 0, 0, 1, 0, 1)$$

Finding efficient algorithms for merging simple sets is currently an open problem.

Remark 5.7 (computational complexity) We now crudely analyze the worst-case complexity of the Constraint-Tree algorithm in Figure 1. For brevity, write $|\mathcal{Y}| = n$. Also, initially, we know that $|I_P| = q$. In Step 1, we would like to eliminate fixed coordinates. To check whether a coordinate is fixed requires $O(n)$ computation in the worst case. Initially, there are q coordinates, hence, in the worst case, Step 1 may require $O(qn)$ computation. In Step 2, we would like to eliminate dependent coordinates. To check whether a coordinate i is dependent on some coordinate $j < i$ requires $O(n)$ computation in the worst case. Initially, there are q coordinates, hence, in the worst case, Step 2 may require $O(q^2n)$ computation. Step 3 is essentially free. In Step 4, we compute the most constrained coordinate, this requires $O(qn)$ computation. In Step 5, we partition the set of codewords into two subsets, this requires $O(n)$ computation. Finally, in Step 5, we recursively apply the algorithm to each of the two subsets. Due to recursion, in the worst case, there may $\lceil \log_2 n \rceil$ passes through the set of codewords. Putting it all together, the Constraint-Tree algorithm has a worst case computational complexity of $O(q^2n \log_2 n)$.

Remark 5.8 (“Not-so” simple sets) We now extend the notion of simple sets to d -simple sets for $d \geq 1$. In our new notion, 1-simple sets will correspond to the simple sets defined in Section 2. Let $\mathcal{Z} \subset \{0, 1\}^r$ denote a set of r -bit words. Let $Z = (z_0, z_1, \dots, z_{r-1})$ denote a word

in \mathcal{Z} . As before, we say that coordinate i , $0 \leq i \leq r-1$, is *fixed* or *constant* in \mathcal{Z} , if for all $Z \in \mathcal{Z}$ either $z_i = 0$ or $z_i = 1$. We say that coordinate i , $1 \leq i \leq r-1$ is *d-dependent* in \mathcal{Z} for some $d \geq 1$, if it is not fixed and there exists $1 \leq t \leq d$ coordinates $j_1 < j_2 < \dots < j_t < i$ such that coordinate i is a t -bit Boolean function of these t coordinates. We say that coordinate i , $0 \leq i \leq r-1$, is *d-free* in \mathcal{Z} for some $d \geq 1$, if it is neither fixed nor d -dependent in \mathcal{Z} . We say that a set of r -bit words \mathcal{Z} containing exactly s d -free coordinates is *d-simple* if it contains exactly 2^s words.

It is easy to modify the algorithm in Figure 1 to find d -simple sets by writing Step 2 as a sequence of d -substeps, where the first substep eliminates 1-dependent coordinates, the second substep eliminates 2-dependent coordinates, and so on. Finding d -simple sets might lead to even lower hardware complexity codes, since we might be able to decompose a set of codewords into fewer d -simple sets for some $d \geq 1$. However, in the worst case, Step 2 will require $O(2^{2^d} q^{d+1} n)$ computation, where 2^{2^d} represents the number of one-output d -input Boolean functions. Hence, it follows from the discussion in Remark 5.7 that the modified algorithm for finding d -simple sets will have a worst case computational complexity of $O(2^{2^d} q^{d+1} n \log_2 n)$.

Remark 5.9 (Finite-state codes) A *finite-state encoder* consists of a finite set of states V and a collection of transitions $I \xrightarrow{u/v} J$ from state I to state J with input block u and output block v . Such a transition is called an *outgoing transition* from state I . The idea is that at state I , a p -bit input user block u is encoded to a q -bit output constrained block v and the encoder passes to state J (we assume that for each state I , p and q are constant over all outgoing transitions, but those constants are allowed to change from one state to another). Each state must have exactly 2^p outgoing transitions, one to accommodate each p -bit user block. Note that a block encoder is simply a finite-state encoder with only one state.

Now, suppose that we are given a finite-state encoder as above except that (1) each transition has only an output label but no input label and (2) each state has at least (but not necessarily exactly) 2^p outgoing transitions. Let O_I denote the set of outgoing transitions, each represented as a pair (v, J) . A true finite-state encoder is then obtained from a data-to-codeword assignment which gives a definition for each state I of a one-to-one mapping f_I from the set of all 2^p binary p -bit blocks into O_I . The f_I then completely define the encoder: delete all transitions in O_I outside of the image of f_I and endow each remaining transition (v, J) with the input label $u = f_I^{-1}(v, J)$.

One can simply apply the methods of this paper to each state I to construct such an f_I . But it is often desirable for the encoder to be *sliding-block decodable*, i.e., an output block v decodes to an input block u as a function of a certain number, m , of preceding blocks (the memory) and a certain number, a , of following blocks (the anticipation), but otherwise independent of state information. Sliding-block decodability requires certain consistency conditions among the f_I to hold. For instance, consider the special case $m = 0$ and $a = 1$. Then, we must have that

whenever

- $(v, J) \in O_I$ and $(v', J') \in O_{I'}$,
- $v = v'$, and
- J and J' have outgoing transitions with a common output label,

then $f_I^{-1}(v, J) = f_{I'}^{-1}(v', J')$.

We are currently working on heuristic methods to achieve such a consistency condition with low complexity assignments for f_I .

Remark 5.10 (conjecture of computational hardness) Suppose we are given a finite set Ω of one-output Boolean functions (or gates). A Ω -circuit is a directed acyclic graph whose nodes are elements of Ω . We assume that Ω is *complete*, that is, any Boolean function can be computed using an Ω -circuit. The *circuit complexity* of a Boolean function, with respect to Ω , is the smallest number of gates in an Ω -circuit computing the function [12, 13]; and is written as $\Xi_\Omega(\cdot)$.

The codebook \mathcal{C} is a subset of $\mathcal{Y} \subset \{0, 1\}^q$ such that $|\mathcal{C}| = 2^p$. Given a codebook \mathcal{C} , the encoder \mathcal{E} is a one-to-one mapping from $\mathcal{X} = \{0, 1\}^p$ onto \mathcal{C} . We write the corresponding decoder as $\mathcal{D}(\mathcal{E})$. The problem that we are interested in is to compute an optimal codebook \mathcal{C}^* such that

$$\mathcal{C}^* = \arg \min_{\mathcal{C}} \left\{ \min_{\mathcal{E}} \{ \Xi_\Omega(\mathcal{E}) + \Xi_\Omega(\mathcal{D}(\mathcal{E})) \} \right\}$$

where $\arg \min$ is over all subsets \mathcal{C} of \mathcal{Y} such that $|\mathcal{C}| = 2^p$, and \min is over all one-to-one Boolean functions \mathcal{E} that map $\{0, 1\}^p$ onto \mathcal{C} . Also, we would like to compute the corresponding optimal encoder \mathcal{E}^* such that

$$\mathcal{E}^* = \arg \min_{\mathcal{E}} \{ \Xi_\Omega(\mathcal{E}) + \Xi_\Omega(\mathcal{D}(\mathcal{E})) \},$$

where $\arg \min$ is over all one-to-one Boolean functions \mathcal{E} that map $\{0, 1\}^p$ onto \mathcal{C}^* .

We believe that finding such optimal codebooks and encoders is computationally hard [14], and, hence, in this paper, we have focussed on the heuristic “art” of constructing low-complexity encoders and decoders.

Acknowledgments

We are grateful to Mario Blaum for introducing us to the heuristic of “mapping-by-gated-partitioning,” to Lars Thon for introducing us to the Berkeley SIS program, and Chuck Cox for sharing his implementation of the SA-12E cell library for use with SIS.

References

- [1] K. A. S. Immink, *Coding Techniques for Digital Recorders*. New York, USA: Prentice Hall, 1991.
- [2] B. H. Marcus, P. H. Siegel, and J. K. Wolf, "Finite-state modulation codes for data storage," *IEEE J. Selected Areas in Communications*, vol. 10, no. 1, pp. 5–37, January 1992.
- [3] K. A. S. Immink, P. H. Siegel, and J. K. Wolf, "Codes for digital recorders," *IEEE Trans. Inform. Theory*, vol. 44, no. 6, pp. 2260–2299, October 1998.
- [4] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli, "SIS: A system for sequential circuit synthesis," Tech. Rep. UCB/ERL M92/41, Electronics Research Laboratory, Department of Electrical Engineering and Computer Science, University of California, Berkeley, 4 May 1992.
- [5] IBM, *ASIC SA-12E Databook*, 1998. <http://www.chips.ibm.com/techlib/products/asics/databooks.html>.
- [6] J. S. Eggenberger and A. M. Patel, "Method and apparatus for implementing PRML codes." US Patent No. 4707681, <http://www.patents.ibm.com>, November 17 1987.
- [7] B. H. Marcus, A. M. Patel, and P. H. Siegel, "Method and apparatus for implementing a PRML code." US Patent No. 4786890, <http://www.patents.ibm.com>, November 22 1988.
- [8] R. L. Galbraith, "Method and apparatus for implementing PRML codes with maximum ones." US Patent No. 5196849, <http://www.patents.ibm.com>, March 13 1993.
- [9] T. M. Cover and J. A. Thomas, *Elements of Information Theory*. New York, USA: John Wiley & Sons, 1991.
- [10] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proc. IRE*, vol. 40, pp. 1098–1101, September 1952.
- [11] J. Rissanen, "Generalized Kraft inequality and arithmetic coding," *IBM J. Res. Dev.*, vol. 20, no. 3, pp. 198–203, 1976.
- [12] C. E. Shannon, "The synthesis of two-terminal switching circuits," *Bell System Technical J.*, vol. 28, pp. 59–98, 1949.
- [13] I. Wegener, *The Complexity of Boolean Functions*. Chichester, UK: John Wiley & Sons, 1987.
- [14] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, USA: W. H. Freeman & Company, 1979.